廖雪峰 Java教程



这是专门针对小白的零基础Java教程。为什么要学Java？因为Java是全球排名第一的编程语言，Java工程师也是市场需求最大的软件工程师，选择Java，就是选择了高薪。

下载手机APP
畅享精彩阅读

目　录

- 致谢
- Java教程
- Java快速入门
  - Java简介
    - 安装JDK
    - 第一个Java程序
    - Java代码助手
    - 使用IDE
    - 使用IDE练习插件
  - Java程序基础
    - Java程序基本结构
    - 变量和数据类型
    - 整数运算
    - 浮点数运算
    - 布尔运算
    - 字符和字符串
    - 数组类型
  - 流程控制
    - 输入和输出
    - if判断
    - switch多重选择
    - while循环
    - do while循环
    - for循环
    - break和continue
  - 数组操作
    - 遍历数组
    - 数组排序
    - 多维数组
    - 命令行参数
- 面向对象编程
  - 面向对象基础
    - 方法
    - 构造方法
    - 方法重载
    - 继承

多态
抽象类
接口
静态字段和静态方法
包
作用域
classpath和jar
模块
Java核心类
字符串和编码
StringBuilder
StringJoiner
包装类型
JavaBean
枚举类
BigInteger
BigDecimal
常用工具类
异常处理
Java的异常
捕获异常
抛出异常
自定义异常
使用断言
使用JDK Logging
使用Commons Logging
使用Log4j
使用SLF4J和Logback
反射
Class类
访问字段
调用方法
调用构造方法
获取继承关系
动态代理
注解
使用注解
定义注解

处理注解
泛型
什么是泛型
使用泛型
编写泛型
擦拭法
extends通配符
super通配符
泛型和反射
集合
Java集合简介
使用List
编写equals方法
使用Map
编写equals和hashCode
使用EnumMap
使用TreeMap
使用Properties
使用Set
使用Queue
使用PriorityQueue
使用Deque
使用Stack
使用Iterator
使用Collections
IO
File对象
InputStream
OutputStream
Filter模式
操作Zip
读取classpath资源
序列化
Reader
Writer
PrintStream和PrintWriter
日期与时间
基本概念

Date和Calendar
LocalDateTime
ZonedDateTime
DateTimeFormatter
Instant
最佳实践
单元测试
编写JUnit测试
使用Fixture
异常测试
条件测试
参数化测试
正则表达式
正则表达式简介
匹配规则
复杂匹配规则
分组匹配
非贪婪匹配
搜索和替换
加密与安全
编码算法
哈希算法
BouncyCastle
Hmac算法
对称加密算法
口令加密算法
密钥交换算法
非对称加密算法
签名算法
数字证书
多线程
多线程基础
创建新线程
线程的状态
中断线程
守护线程
线程同步
同步方法

死锁
使用wait和notify
使用ReentrantLock
使用Condition
使用ReadWriteLock
使用StampedLock
使用Concurrent集合
使用Atomic
使用线程池
使用Future
使用CompletableFuture
使用ForkJoin
使用ThreadLocal
Maven基础
Maven介绍
依赖管理
构建流程
使用插件
模块管理
使用mvnw
网络编程
网络编程基础
TCP编程
UDP编程
发送Email
接收Email
HTTP编程
XML与JSON
XML简介
使用DOM
使用SAX
使用Jackson
使用JSON
JDBC编程
JDBC简介
JDBC查询
JDBC更新
JDBC事务

        JDBC Batch
        JDBC连接池
函数式编程
        Lambda基础
        方法引用
        使用Stream
            创建Stream
            使用map
            使用filter
            使用reduce
            输出集合
            其他操作
设计模式
        创建型模式
            工厂方法
            抽象工厂
            生成器
            原型
            单例
        结构型模式
            适配器
            桥接
            组合
            装饰器
            外观
            享元
            代理
        行为型模式
            责任链
            命令

致谢

当前文档 《廖雪峰 Java教程》 由 进击的皇虫 使用 书栈网(BookStack.CN) 进行构建，生成于 2020-02-23。

书栈网仅提供文档编写、整理、归类等功能，以及对文档内容的生成和导出工具。

文档内容由网友们编写和整理，书栈网难以确认文档内容知识点是否错漏。如果您在阅读文档获取知识的时候，发现文档内容有不恰当的地方，请向我们反馈，让我们共同携手，将知识准确、高效且有效地传递给每一个人。

同时，如果您在日常工作、生活和学习中遇到有价值有营养的知识文档，欢迎分享到书栈网，为知识的传承献上您的一份力量！

如果当前文档生成时间太久，请到书栈网获取最新的文档，以跟上知识更新换代的步伐。

内容来源：廖雪峰 https://www.liaoxuefeng.com/wiki/1252599548343744

文档地址：http://www.bookstack.cn/books/liaoxuefeng-java

书栈官网：https://www.bookstack.cn

书栈开源：https://github.com/TruthHun

分享，让知识传承更久远！ 感谢知识的创造者，感谢知识的分享者，也感谢每一位阅读到此处的读者，因为我们都将成为知识的传承者。

Java教程

这是专门针对小白的零基础Java教程。

为什么要学Java？

因为Java是全球排名第一的编程语言，Java工程师也是市场需求最大的软件工程师，选择Java，就是选择了高薪。



为什么Java应用最广泛？

从互联网到企业平台，Java是应用最广泛的编程语言，原因在于：

- Java是基于JVM虚拟机的跨平台语言，一次编写，到处运行；
- Java程序易于编写，而且有内置垃圾收集，不必考虑内存管理；
- Java虚拟机拥有工业级的稳定性和高度优化的性能，且经过了长时期的考验；
- Java拥有最广泛的开源社区支持，各种高质量组件随时可用。

Java语言常年霸占着三大市场：

- 互联网和企业应用，这是Java EE的长期优势和市场地位；
- 大数据平台，主要有Hadoop、Spark、Flink等，他们都是Java或Scala（一种运行于JVM的编程语言）开发的；
- Android移动平台。

这意味着Java拥有最广泛的就业市场。

教程特色

虽然是零基础Java教程，但是覆盖了从基础到高级的Java核心编程，从小白成长到架构师，实现硬实力高薪就业！

还可以边学边练，而且可以在线练习！

并且，时刻更新至最新版Java！目前教程版本是：

Java 13!

最重要的是：

免费！

不要犹豫了！现在开始学习Java，从入门到架构师！



使用窄屏手机的童鞋，请点击左上角“目录”查看教程：



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java快速入门

本章的主要内容是快速掌握Java程序的基础知识，了解并使用变量和各种数据类型，介绍基本的程序流程控制语句。



通过本章的学习，可以编写基本的Java程序。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java简介

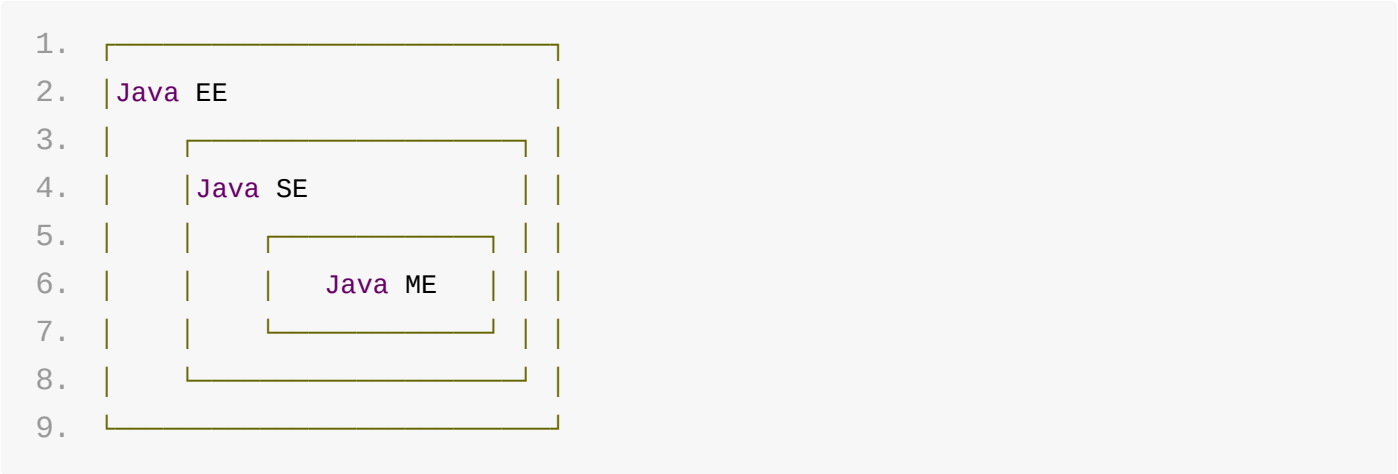
Java最早是由SUN公司（已被Oracle收购）的詹姆斯·高斯林（高司令，人称Java之父）在上个世纪90年代初开发的一种编程语言，最初被命名为Oak，目标是针对小型家电设备的嵌入式应用，结果市场没啥反响。谁料到互联网的崛起，让Oak重新焕发了生机，于是SUN公司改造了Oak，在1995年以Java的名称正式发布，原因是Oak已经被人注册了，因此SUN注册了Java这个商标。随着互联网的高速发展，Java逐渐成为最重要的网络编程语言。

Java介于编译型语言和解释型语言之间。编译型语言如C、C++，代码是直接编译成机器码执行，但是不同的平台（x86、ARM等）CPU的指令集不同，因此，需要编译出每一种平台的对应机器码。解释型语言如Python、Ruby没有这个问题，可以由解释器直接加载源码然后运行，代价是运行效率太低。而Java是将代码编译成一种“字节码”，它类似于抽象的CPU指令，然后，针对不同平台编写虚拟机，不同平台的虚拟机负责加载字节码并执行，这样就实现了“一次编写，到处运行”的效果。当然，这是针对Java开发者而言。对于虚拟机，需要为每个平台分别开发。为了保证不同平台、不同公司开发的虚拟机都能正确执行Java字节码，SUN公司制定了一系列的Java虚拟机规范。从实践的角度看，JVM的兼容性做得非常好，低版本的Java字节码完全可以正常运行在高版本的JVM上。

随着Java的发展，SUN给Java又分出了三个不同版本：

- Java SE: Standard Edition
- Java EE: Enterprise Edition
- Java ME: Micro Edition

这三者之间有啥关系呢？



简单来说，Java SE就是标准版，包含标准的JVM和标准库，而Java EE是企业版，它只是在Java SE的基础上加上了大量的API和库，以便方便开发Web应用、数据库、消息服务等，Java EE的应用使用的虚拟机和Java SE完全相同。

Java ME就和Java SE不同，它是一个针对嵌入式设备的“瘦身版”，Java SE的标准库无法在Java

ME上使用，Java ME的虚拟机也是“瘦身版”。

毫无疑问，Java SE是整个Java平台的核心，而Java EE是进一步学习Web应用所必须的。我们熟悉的Spring等框架都是Java EE开源生态系统的一部分。不幸的是，Java ME从来没有真正流行起来，反而是Android开发成为了移动平台的标准之一，因此，没有特殊需求，不建议学习Java ME。

因此我们推荐的Java学习路线图如下：

- 首先要学习Java SE，掌握Java语言本身、Java核心开发技术以及Java标准库的使用；
- 如果继续学习Java EE，那么Spring框架、数据库开发、分布式架构就是需要学习的；
- 如果要学习大数据开发，那么Hadoop、Spark、Flink这些大数据平台就是需要学习的，他们都基于Java或Scala开发；
- 如果想要学习移动开发，那么就深入Android平台，掌握Android App开发。

无论怎么选择，Java SE的核心技术是基础，这个教程的目的就是让你完全精通Java SE！

Java版本

从1995年发布1.0版本开始，到目前为止，最新的Java版本是Java 13：

时间	版本
1995	1.0
1998	1.2
2000	1.3
2002	1.4
2004	1.5 / 5.0
2005	1.6 / 6.0
2011	1.7 / 7.0
2014	1.8 / 8.0
2017/9	1.9 / 9.0
2018/3	10
2018/9	11
2019/3	12
2019/9	13

本教程使用的Java版本是最新版的**Java 13**。

名词解释

初学者学Java，经常听到JDK、JRE这些名词，它们到底是啥？

- JDK: Java Development Kit
- JRE: Java Runtime Environment

简单地说，JRE就是运行Java字节码的虚拟机。但是，如果只有Java源码，要编译成Java字节码，就需要JDK，因为JDK除了包含JRE，还提供了编译器、调试器等开发工具。

二者关系如下：



要学习Java开发，当然需要安装JDK了。

那JSR、JCP……又是啥？

- JSR规范: Java Specification Request
- JCP组织: Java Community Process

为了保证Java语言的规范性，SUN公司搞了一个JSR规范，凡是想给Java平台加一个功能，比如说访问数据库的功能，大家要先创建一个JSR规范，定义好接口，这样，各个数据库厂商都按照规范写出Java驱动程序，开发者就不用担心自己写的数据库代码在MySQL上能跑，却不能跑在PostgreSQL上。

所以JSR是一系列的规范，从JVM的内存模型到Web程序接口，全部都标准化了。而负责审核JSR的组织就是JCP。

一个JSR规范发布时，为了让大家有个参考，还要同时发布一个“参考实现”，以及一个“兼容性测试套件”：

- RI: Reference Implementation
- TCK: Technology Compatibility Kit

比如有人提议要搞一个基于Java开发的消息服务器，这个提议很好啊，但是光有提议还不行，得贴出真正能跑的代码，这就是RI。如果有其他人也想开发这样一个消息服务器，如何保证这些消息服务器对开

发者来说接口、功能都是相同的？所以还得提供TCK。

通常来说，RI只是一个“能跑”的正确的代码，它不追求速度，所以，如果真正要选择一个Java的消息服务器，一般是没人用RI的，大家都会选择一个有竞争力的商用或开源产品。

参考：Java消息服务JMS的JSR：https://jcp.org/en/jsr/detail?id=914



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





安装JDK

因为Java程序必须运行在JVM之上，所以，我们第一件事情就是安装JDK。

搜索JDK 13，确保从Oracle的官网下载最新的稳定版JDK：



找到Java SE 13.x的下载链接，下载安装即可。

设置环境变量

安装完JDK后，需要设置一个 `JAVA_HOME` 的环境变量，它指向JDK的安装目录。在Windows下，它是安装目录，类似：

```
C:\Program Files\Java\jdk-13
```

在Mac下，它在 `~/.bash_profile` 里，它是：

```
export JAVA_HOME=`/usr/libexec/java_home -v 13`
```

然后，把 `JAVA_HOME` 的 `bin` 目录附加到系统环境变量 `PATH` 上。在Windows下，它长这样：

```
Path=%JAVA_HOME%\bin;<现有的其他路径>
```

在Mac下，它在 `~/.bash_profile` 里，长这样：

```
export PATH=$JAVA_HOME/bin:$PATH
```

把 `JAVA_HOME` 的 `bin` 目录添加到 `PATH` 中是为了在任意文件夹下都可以运行 `java` 。打开命令提示符窗口，输入命令 `java -version` ，如果一切正常，你会看到如下输出：

```
┌──────────────────────────────────────────────────────┐
│Command Prompt                                - □ x   │
├──────────────────────────────────────────────────────┤
│Microsoft Windows [Version 10.0.0]                    │
│(c) 2015 Microsoft Corporation. All rights reserved.  │
│                                                      │
│C:\> java -version                                    │
│java version "13" ...                                 │
│Java(TM) SE Runtime Environment                       │
│Java HotSpot(TM) 64-Bit Server VM                     │
│                                                      │
│C:\>                                                  │
│                                                      │
│                                                      │
└──────────────────────────────────────────────────────┘
```

如果你看到的版本号不是 `13` ，而是 `12` 、 `1.8` 之类，说明系统存在多个JDK，且默认JDK不是JDK 13，需要把JDK 13提到 `PATH` 前面。

如果你得到一个错误输出：

```
┌──────────────────────────────────────────────────────┐
│Command Prompt                                - □ x   │
├──────────────────────────────────────────────────────┤
│Microsoft Windows [Version 10.0.0]                    │
│(c) 2015 Microsoft Corporation. All rights reserved.  │
│                                                      │
│C:\> java -version                                    │
│'java' is not recognized as an internal or external comm│
│and, operable program or batch file.                  │
│                                                      │
│C:\>                                                  │
│                                                      │
│                                                      │
└──────────────────────────────────────────────────────┘
```

这是因为系统无法找到Java虚拟机的程序 `java.exe` ，需要检查JAVA_HOME和PATH的配置。

可以参考如何设置或更改PATH系统变量。

JDK

细心的童鞋还可以在 `JAVA_HOME` 的 `bin` 目录下找到很多可执行文件：

- java：这个可执行程序其实就是JVM，运行Java程序，就是启动JVM，然后让JVM执行指定的编译后的代码；
- javac：这是Java的编译器，它用于把Java源码文件（以 `.java` 后缀结尾）编译为Java字节码文件（以 `.class` 后缀结尾）；
- jar：用于把一组 `.class` 文件打包成一个 `.jar` 文件，便于发布；
- javadoc：用于从Java源码中自动提取注释并生成文档；
- jdb：Java调试器，用于开发阶段的运行调试。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





第一个Java程序

我们来编写第一个Java程序。

打开文本编辑器，输入以下代码：

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

在一个Java程序中，你总能找到一个类似：

```
public class Hello {
    ...
}
```

的定义，这个定义被称为class（类），这里的类名是 `Hello` ，大小写敏感， `class` 用来定义一个类， `public` 表示这个类是公开的， `public` 、 `class` 都是Java的关键字，必须小写， `Hello` 是类的名字，按照习惯，首字母 `H` 要大写。而花括号 `{}` 中间则是类的定义。

注意到类的定义中，我们定义了一个名为 `main` 的方法：

```
    public static void main(String[] args) {
        ...
    }
```

方法是可执行的代码块，一个方法除了方法名 `main` ，还有用 `()` 括起来的方法参数，这里的 `main` 方法有一个参数，参数类型是 `String[]` ，参数名是 `args` ， `public` 、 `static` 用来修饰方法，这里表示它是一个公开的静态方法， `void` 是方法的返回类型，而花括号 `{}` 中间的就是方法的代码。

方法的代码每一行用 `;` 结束，这里只有一行代码，就是：

```
        System.out.println("Hello, world!");
```

它用来打印一个字符串到屏幕上。

Java规定，某个类定义的 `public static void main(String[] args)` 是Java程序的固定入口方法，因此，Java程序总是从 `main` 方法开始执行。

注意到Java源码的缩进不是必须的，但是用缩进后，格式好看，很容易看出代码块的开始和结束，缩进一般是4个空格或者一个tab。

最后，当我们把代码保存为文件时，文件名必须是 `Hello.java` ，而且文件名也要注意大小写，因为要和我们定义的类名 `Hello` 完全保持一致。

如何运行Java程序

Java源码本质上是一个文本文件，我们需要先用 `javac` 把 `Hello.java` 编译成字节码文件 `Hello.class` ，然后，用 `java` 命令执行这个字节码文件：

```
┌──────────────────┐
│    Hello.java    │<─── source code
└──────────────────┘
          | compile
          ▼
┌──────────────────┐
│   Hello.class    │<─── byte code
└──────────────────┘
          | execute
          ▼
┌──────────────────┐
│    Run on JVM    │
└──────────────────┘
```

因此，可执行文件 `javac` 是编译器，而可执行文件 `java` 就是虚拟机。

第一步，在保存 `Hello.java` 的目录下执行命令 `javac Hello.java` ：

```
$ javac Hello.java
```

如果源代码无误，上述命令不会有任何输出，而当前目录下会产生一个 `Hello.class` 文件：

```
$ ls
Hello.class    Hello.java
```

第二步，执行 `Hello.class` ，使用命令 `java Hello` ：

```
$ java Hello
Hello, world!
```

注意：给虚拟机传递的参数 `Hello` 是我们定义的类名，虚拟机自动查找对应的class文件并执行。

有一些童鞋可能知道，直接运行 `java Hello.java` 也是可以的：

```
$ java Hello.java
Hello, world!
```

这是Java 11新增的一个功能，它可以直接运行一个单文件源码！

需要注意的是，在实际项目中，单个不依赖第三方库的Java源码是非常罕见的，所以，绝大多数情况下，我们无法直接运行一个Java源码文件，原因是它需要依赖其他的库。

小结

一个Java源码只能定义一个 `public` 类型的class，并且class名称和文件名要完全一致；

使用 `javac` 可以将 `.java` 源码编译成 `.class` 字节码；

使用 `java` 可以运行一个已编译的Java程序，参数是类名。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java代码助手

Java代码运行助手可以让你在线输入Java代码，然后通过本机运行的一个Java程序来执行代码。原理如下：

- 在网页输入代码；
- 点击Run按钮，代码被发送到本机正在运行的Java代码运行助手；
- Java代码运行助手将代码保存为临时文件，然后调用Java虚拟机执行代码；
- 网页显示代码执行结果：



下载

点击右键，目标另存为：LearnJava.java

运行

在存放 `LearnJava.java` 的目录下运行命令：

```
C:\Users\michael\Downloads> java LearnJava.java
```

如果看到 `Ready for Java code on port 39193…` 表示运行成功。

不要关闭命令行窗口，最小化放到后台运行即可：

```
┌──────────────────────────────────────────────────────┐
│Command Prompt                                - □ x   │
├──────────────────────────────────────────────────────┤
│Microsoft Windows [Version 10.0.0]                    │
│(c) 2015 Microsoft Corporation. All rights reserved.  │
│                                                      │
│C:\Users\michael\Downloads> java LearnJava.java       │
│Ready for Java code on port 39193...                  │
```

```
 9. |Press Ctrl + C to exit...                                    |
10. |                                                             |
11. |                                                             |
12. |                                                             |
13. |                                                             |
14. |                                                             |
15. └─────────────────────────────────────────────────────────────┘
```

试试效果

需要支持HTML5的浏览器:

- IE >= 9
- Firefox
- Chrome
- Sarafi

```
// 测试代码
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, world");
    }
}
```

▶ Run

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用IDE

IDE是集成开发环境：Integrated Development Environment的缩写。

使用IDE的好处在于按，可以把编写代码、组织项目、编译、运行、调试等放到一个环境中运行，能极大地提高开发效率。

IDE提升开发效率主要靠以下几点：

- 编辑器的自动提示，可以大大提高敲代码的速度；
- 代码修改后可以自动重新编译，并直接运行；
- 可以方便地进行断点调试。

目前，流行的用于Java开发的IDE有：

Eclipse

Eclipse是由IBM开发并捐赠给开源社区的一个IDE，也是目前应用最广泛的IDE。Eclipse的特点是它本身是Java开发的，并且基于插件结构，即使是对Java开发的支持也是通过插件JDT实现的。

除了用于Java开发，Eclipse配合插件也可以作为C/C++开发环境、PHP开发环境、Rust开发环境等。

IntelliJ Idea

IntelliJ Idea是由JetBrains公司开发的一个功能强大的IDE，分为免费版和商用付费版。JetBrains公司的IDE平台也是基于IDE平台+语言插件的模式，支持Python开发环境、Ruby开发环境、PHP开发环境等，这些开发环境也分为免费版和付费版。

NetBeans

NetBeans是最早由SUN开发的开源IDE，由于使用人数较少，目前已不再流行。

使用Eclipse

你可以使用任何IDE进行Java学习和开发。我们不讨论任何关于IDE的优劣，本教程使用Eclipse作为开发演示环境，原因在于：

- 完全免费使用；
- 所有功能完全满足Java开发需求。

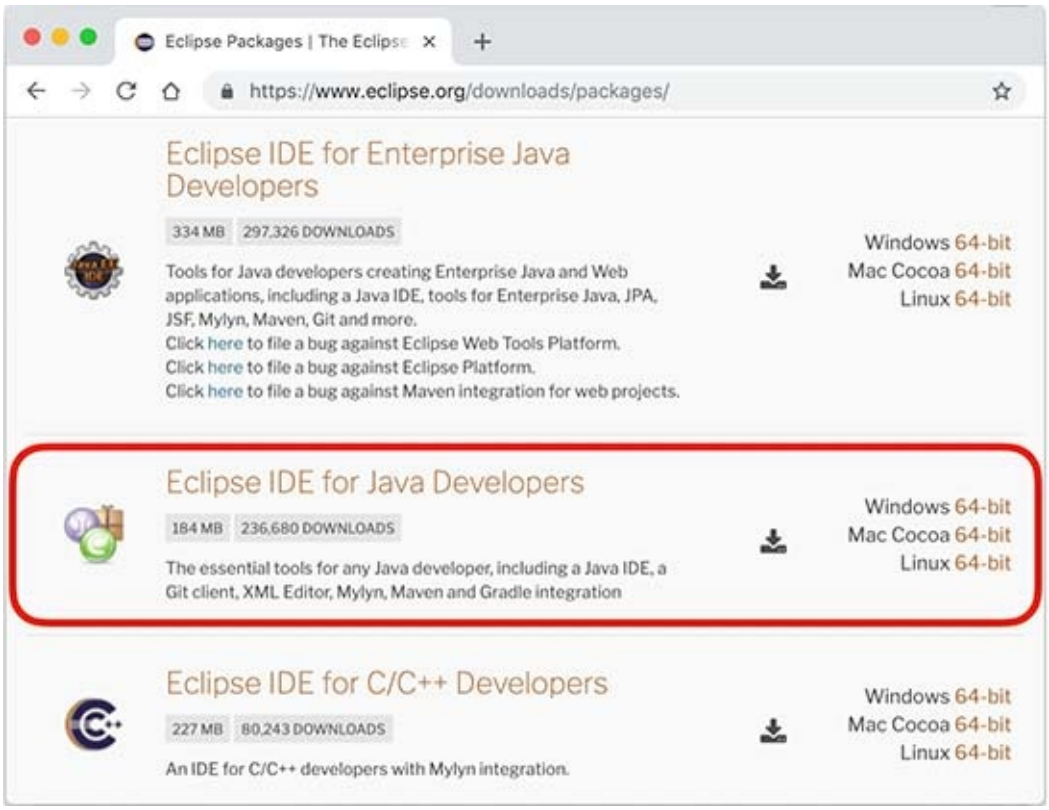
如果你使用Eclipse作为开发环境来学习本教程，还可以获得一个额外的好处：教程提供了一个基于

Eclipse的IDE练习插件，可以直接在线导入Java工程！

安装Eclipse

Eclipse的发行版提供了预打包的开发环境，包括Java、JavaEE、C++、PHP、Rust等。从这里下载：

我们需要下载的版本是Eclipse IDE for Java Developers：



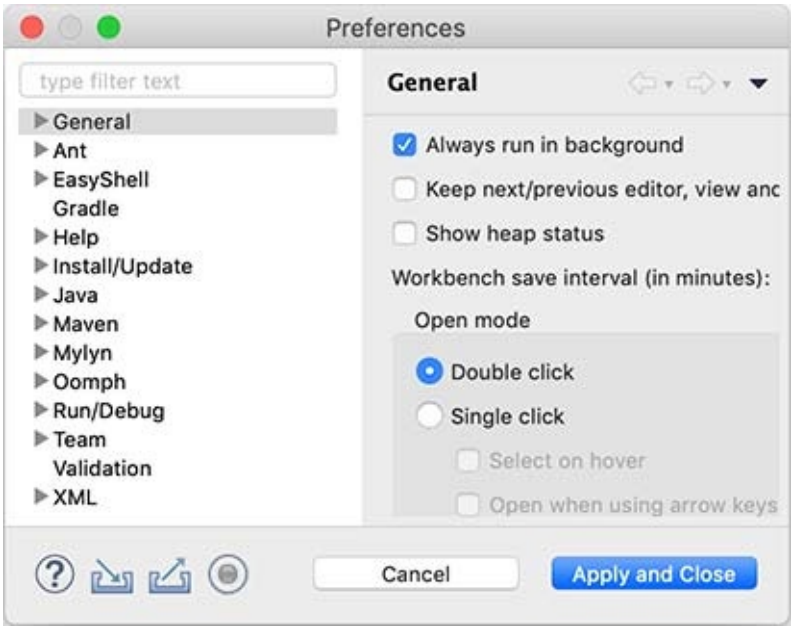
根据操作系统是Windows、Mac还是Linux，从右边选择对应的下载链接。

注意：教程从头到尾并不需要用到Enterprise Java的功能，所以不需要下载Eclipse IDE for Enterprise Java Developers

设置Eclipse

下载并安装完成后，我们启动Eclipse，对IDE环境做一个基本设置：

选择菜单“Eclipse/Window”-“Preferences”，打开配置对话框：



我们需要调整以下设置项：

General > Editors > Text Editors

钩上“Show line numbers”，这样编辑器会显示行号；

General > Workspace

钩上“Refresh using native hooks or polling”，这样Eclipse会自动刷新文件夹的改动；

对于“Text file encoding”，如果Default不是 `UTF-8` ，一定要改为“Other: UTF-8”，所有文本文件均使用 `UTF-8` 编码；

对于“New text file line delimiter”，建议使用Unix，即换行符使用 `\n` 而不是Windows的 `\r\n` 。

Java > Compiler

将“Compiler compliance level”设置为 `13` ，本教程的所有代码均使用Java 13的语法，并且编译到Java 13的版本。

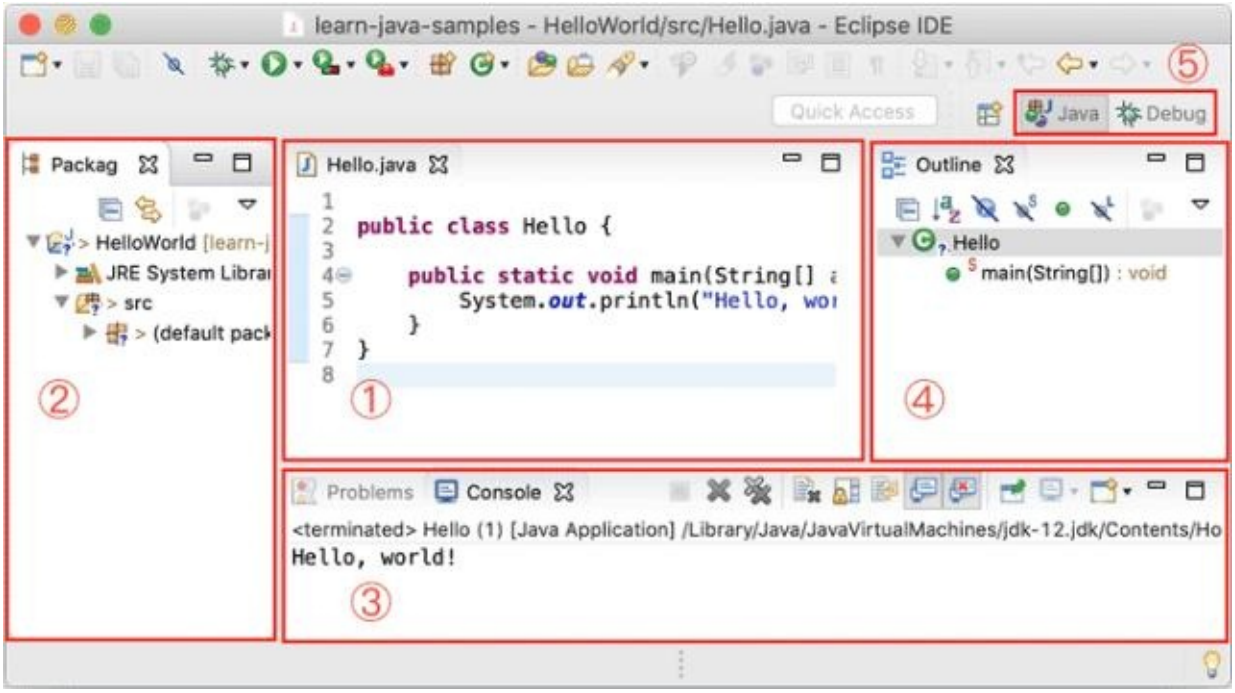
去掉“Use default compliance settings”并钩上“Enable preview features for Java 13”，这样我们就可以使用Java 13的预览功能。

Java > Installed JREs

在Installed JREs中应该看到Java SE 13，如果还有其他的JRE，可以删除，以确保Java SE 13是默认的JRE。

Eclipse IDE结构

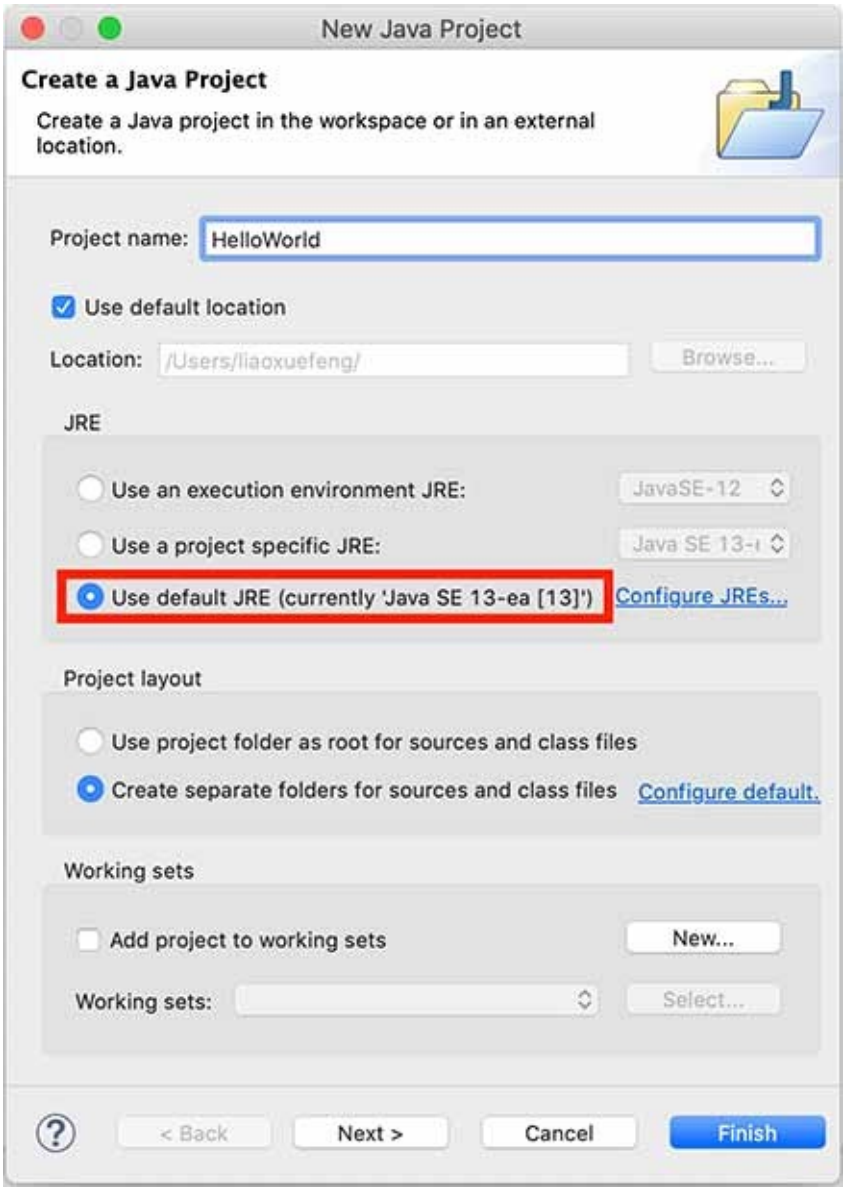
打开Eclipse后，整个IDE由若干个区域组成：



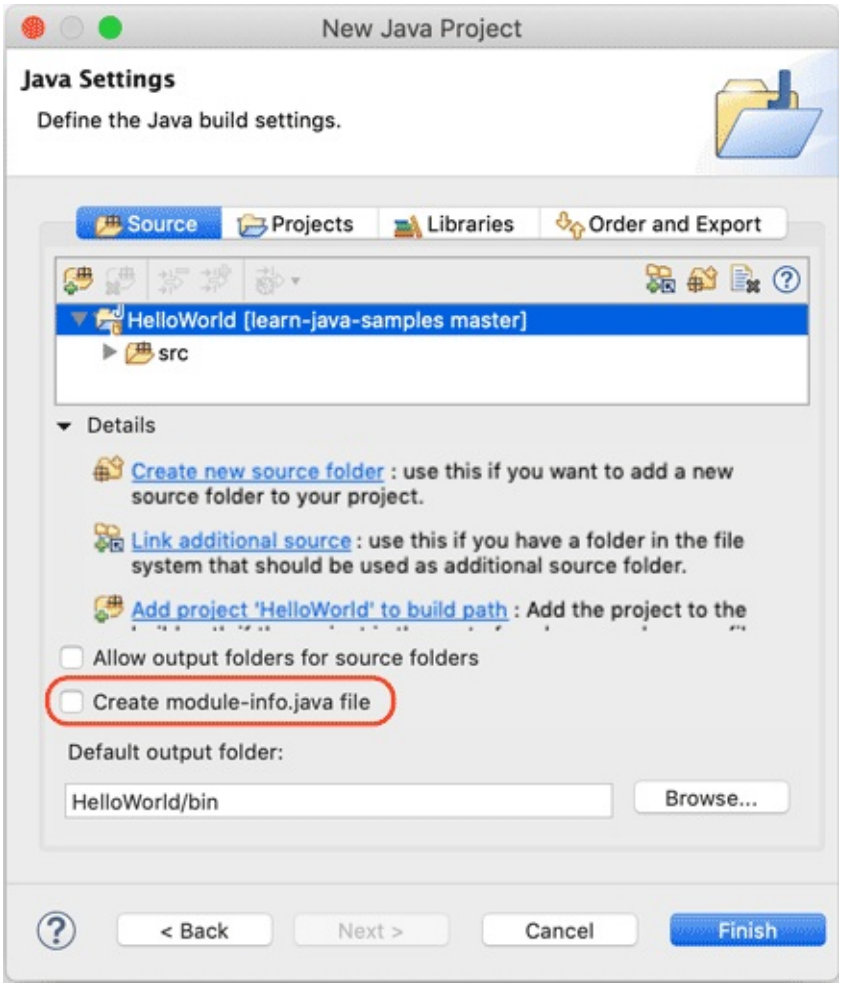
- 中间可编辑的文本区（见1）是编辑器，用于编辑源码；
- 分布在左右和下方的是视图：
  - Package Exploroer（见2）是Java项目的视图
  - Console（见3）是命令行输出视图
  - Outline（见4）是当前正在编辑的Java源码的结构视图
- 视图可以任意组合，然后把一组视图定义成一个Perspective（见5），Eclipse预定义了Java、Debug等几个Perspective，用于快速切换。

新建Java项目

在Eclipse菜单选择“File”-“New”-“Java Project”，填入 `HelloWorld` ，JRE选择 `Java SE 13` ：



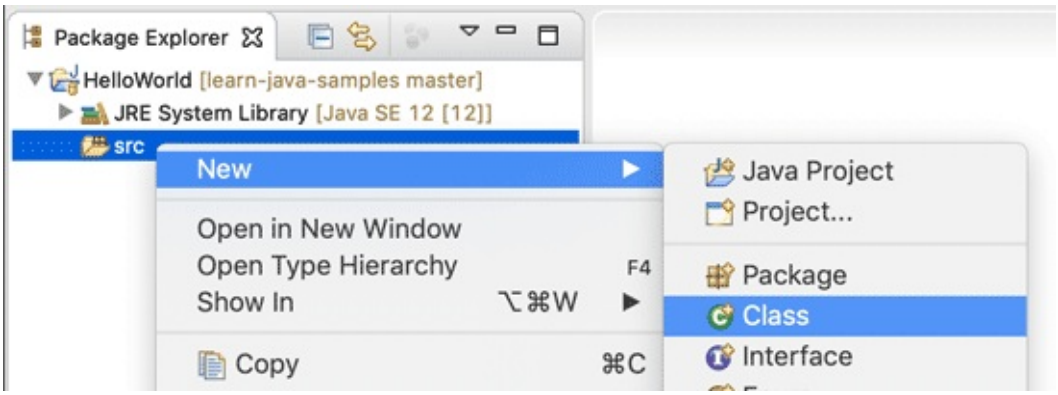
暂时不要勾选“Create module-info.java file”，因为模块化机制我们后面才会讲到：



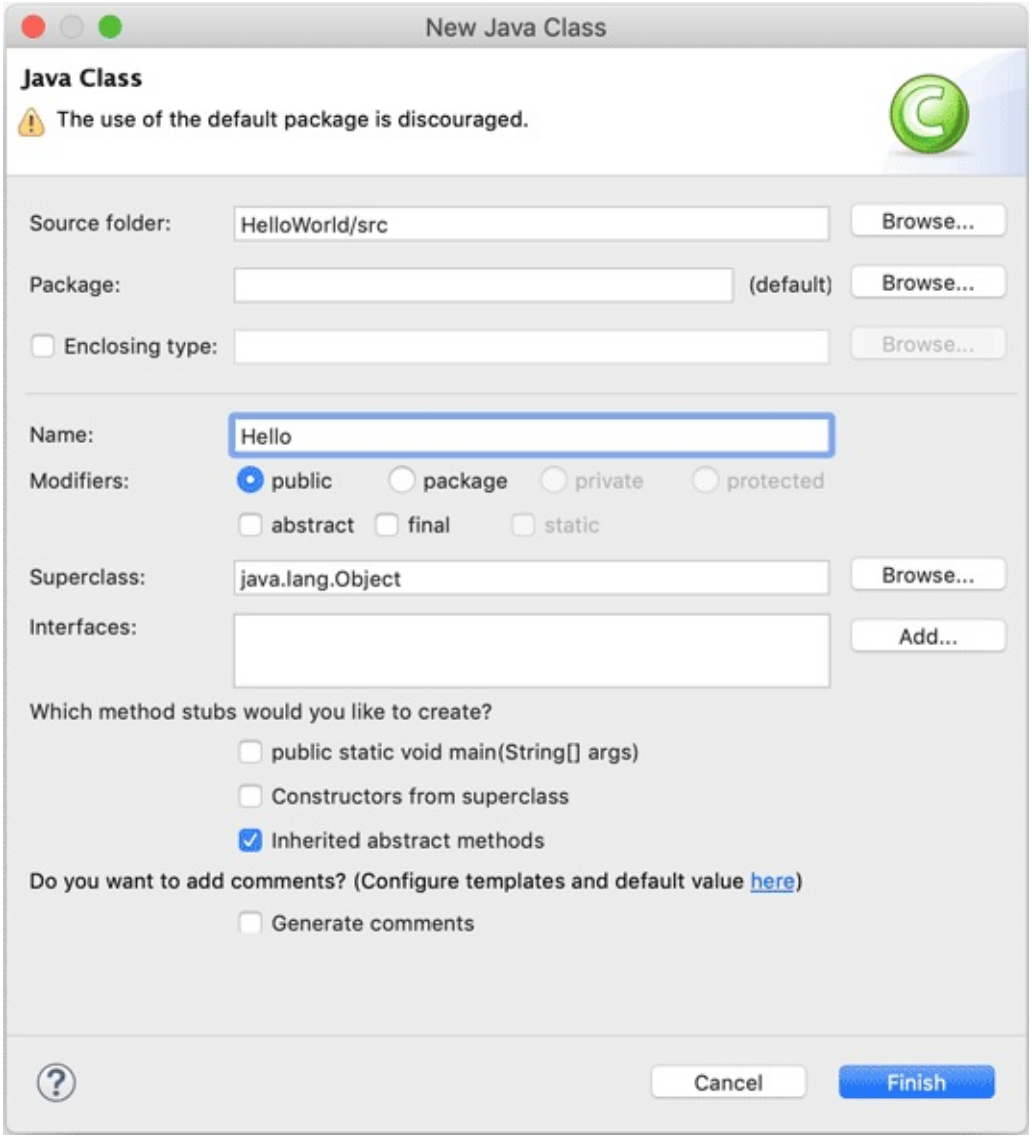
点击“Finish”就成功创建了一个名为 `HelloWorld` 的Java工程。

新建Java文件并运行

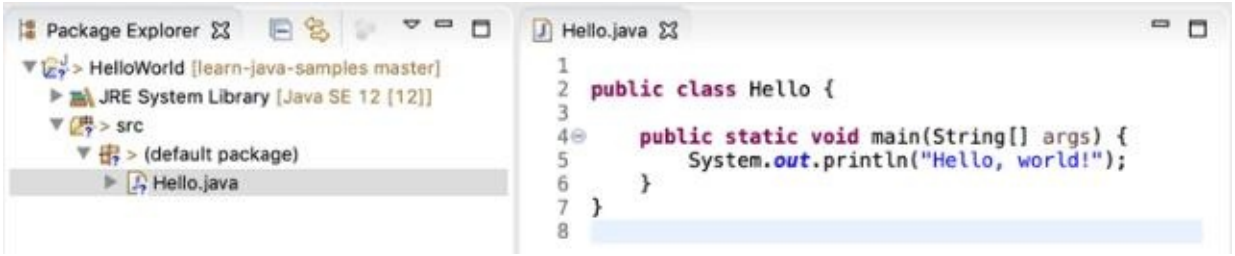
展开 `HelloWorld` 工程，选中源码目录 `src` ，点击右键，在弹出菜单中选择“New”-“Class”：



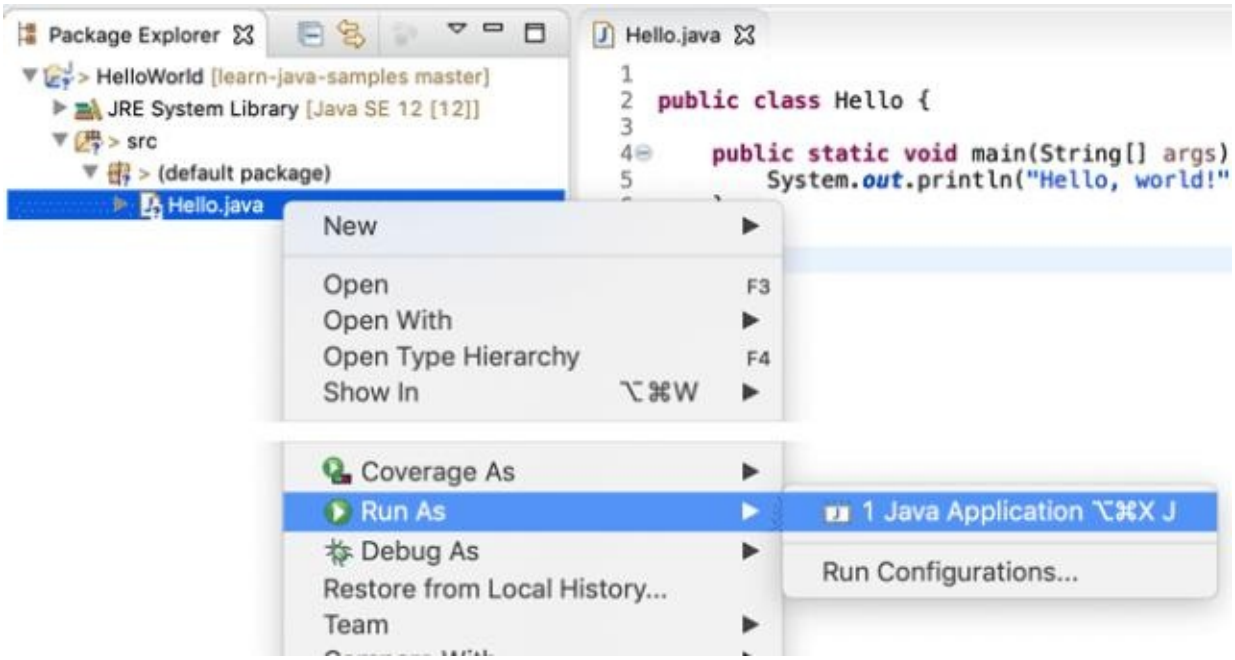
在弹出的对话框中， `Name` 一栏填入 `Hello` ：



点击”Finish“，就自动在 `src` 目录下创建了一个名为 `Hello.java` 的源文件。我们双击打开这个源文件，填上代码：



保存，然后选中文件 `Hello.java` ，点击右键，在弹出的菜单中选中“Run As…”-“Java Application”：



在 `Console` 窗口中就可以看到运行结果：



如果没有在主界面中看到 `Console` 窗口，请选中菜单“Window”-“Show View”-“Console”，即可显示。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论



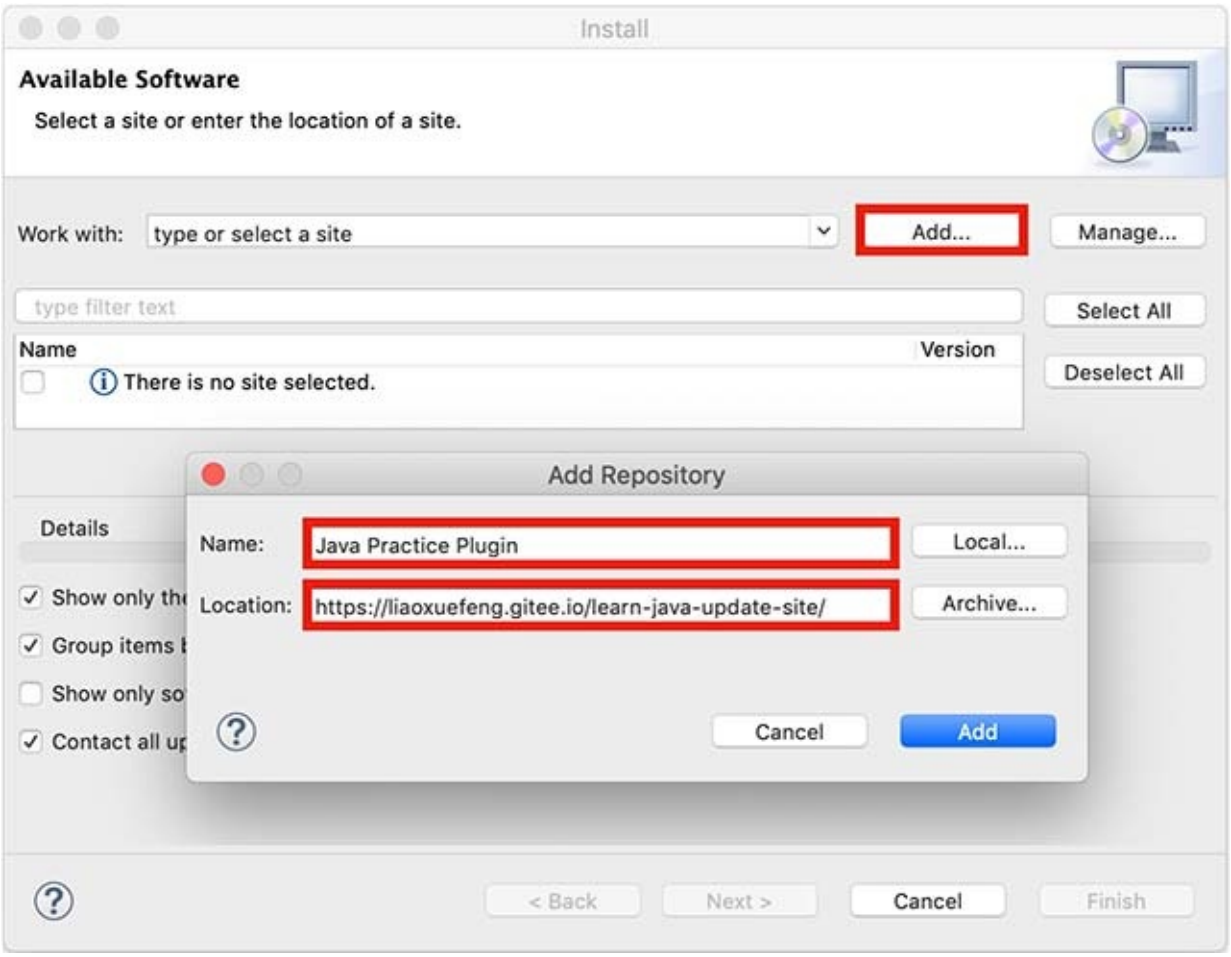


使用IDE练习插件

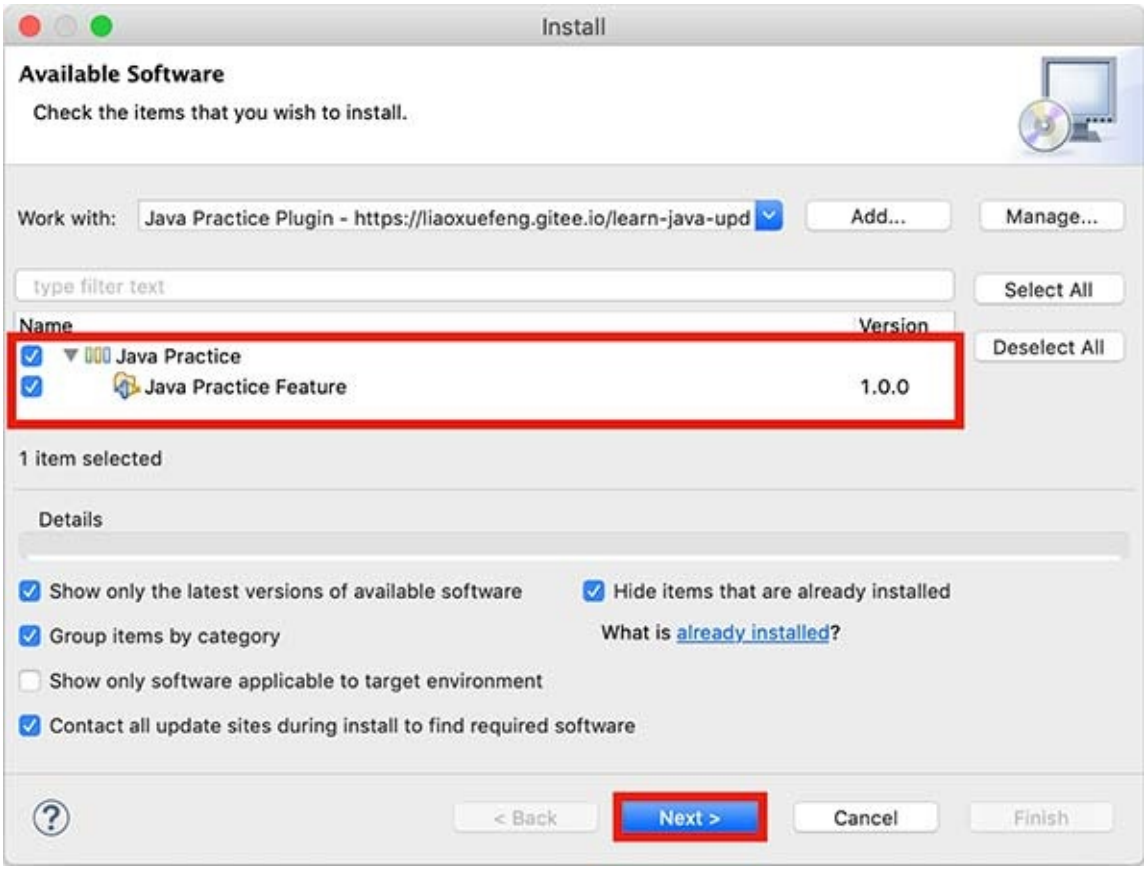
本教程提供一个Eclipse IDE的练习插件，可以非常方便地下载练习代码。

安装IDE练习插件

启动Eclipse，选择菜单“Help”-“Install New Software…”，在打开的对话框中：



点击“Add”，对Name填写一个任意的名称，例如“Java Practice Plugin”，对于Location，填入 `https://liaoxuefeng.gitee.io/learn-java-update-site/` ，然后点击“Add”添加：



在列表中选中“Java Practice Feature”，然后点击“Next”安装。

在安装过程中，由于插件代码没有数字签名，所以会弹出一个警告：

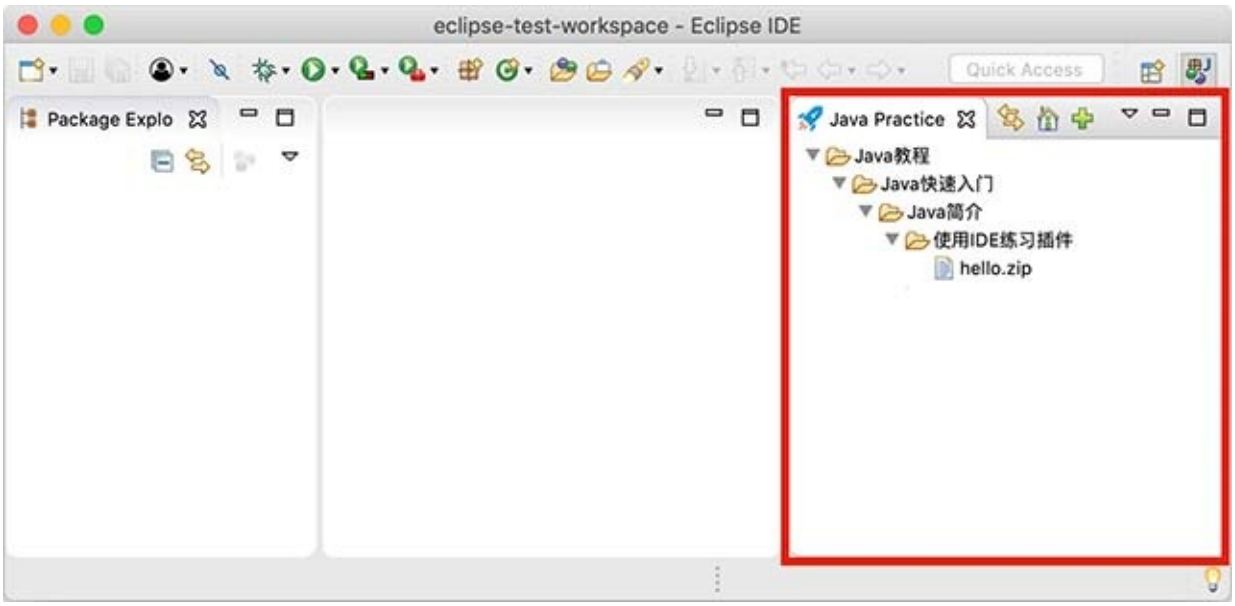


选择“Install anyway”继续安装，安装成功后，根据提示重启Eclipse即可。

重启Eclipse后，选择菜单“Window”-“Show View”-“Other…”：

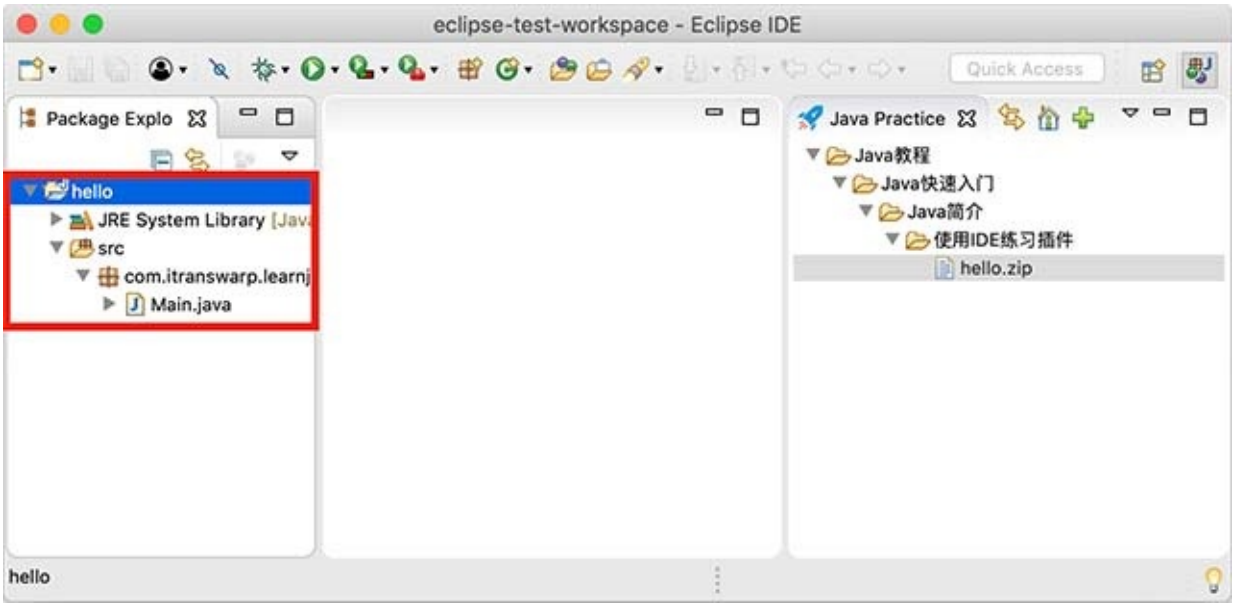


在弹出的对话框中选择“Java”-“Java Practice”，然后点击“Open”，即可在Eclipse中看到Java Practice插件：



导入练习

在“Java Practice”面板中，双击 `hello.zip` ，按照提示导入工程，即可直接下载并导入到Eclipse中：



是不是非常方便？

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java程序基础

本节我们将介绍Java程序的基础知识，包括：

- Java程序基本结构
- 变量和数据类型
- 整数运算
- 浮点数运算
- 布尔运算
- 字符和字符串
- 数组类型



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java程序基本结构

我们先剖析一个完整的Java程序，它的基本结构是什么：

```
/**
 * 可以用来自动创建文档的注释
 */
public class Hello {
    public static void main(String[] args) {
        // 向屏幕输出文本:
        System.out.println("Hello, world!");
        /* 多行注释开始
        注释内容
        注释结束 */
    }
} // class定义结束
```

因为Java是面向对象的语言，一个程序的基本单位就是 `class` ， `class` 是关键字，这里定义的 `class` 名字就是 `Hello` ：

```
public class Hello { // 类名是Hello
    // ...
} // class定义结束
```

类名要求：

- 类名必须以英文字母开头，后接字母，数字和下划线的组合
- 习惯以大写字母开头

要注意遵守命名习惯，好的类命名：

- Hello
- NoteBook
- VRPlayer

不好的类命名：

- hello
- Good123
- Note_Book
- _World

注意到 `public` 是访问修饰符，表示该 `class` 是公开的。

不写 `public` ，也能正确编译，但是这个类将无法从命令行执行。

在 `class` 内部，可以定义若干方法（method）：

```
public class Hello {
    public static void main(String[] args) { // 方法名是main
        // 方法代码...
    } // 方法定义结束
}
```

方法定义了一组执行语句，方法内部的代码将会被依次顺序执行。

这里的方法名是 `main` ，返回值是 `void` ，表示没有任何返回值。

我们注意到 `public` 除了可以修饰 `class` 外，也可以修饰方法。而关键字 `static` 是另一个修饰符，它表示静态方法，后面我们会讲解方法的类型，目前，我们只需要知道，Java入口程序规定的方法必须是静态方法，方法名必须为 `main` ，括号内的参数必须是String数组。

方法名也有命名规则，命名和 `class` 一样，但是首字母小写：

好的方法命名：

- main
- goodMorning
- playVR

不好的方法命名：

- Main
- good123
- good_morning
- _playVR

在方法内部，语句才是真正的执行代码。Java的每一行语句必须以分号结束：

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!"); // 语句
    }
}
```

在Java程序中，注释是一种给人阅读的文本，不是程序的一部分，所以编译器会自动忽略注释。

Java有3种注释，第一种是单行注释，以双斜线开头，直到这一行的结尾结束：

```
// 这是注释...
```

而多行注释以 `/*` 星号开头，以 `*/` 结束，可以有多行：

```
/*
这是注释
blablabla...
这也是注释
*/
```

还有一种特殊的多行注释，以 `/**` 开头，以 `*/` 结束，如果有多行，每行通常以星号开头：

```
/**
 * 可以用来自动创建文档的注释
 *
 * @auther liaoxuefeng
 */
public class Hello {
    public static void main(String[] args) {
        System.out.println("Hello, world!");
    }
}
```

这种特殊的多行注释需要写在类和方法的定义处，可以用于自动创建文档。

Java程序对格式没有明确的要求，多几个空格或者回车不影响程序的正确性，但是我们要养成良好的编程习惯，注意遵守Java社区约定的编码格式。

那约定的编码格式有哪些要求呢？其实我们在前面介绍的Eclipse IDE提供了快捷键 `Ctrl+Shift+F` （macOS是 `⌘+⇧+F` ）帮助我们快速格式化代码的功能，Eclipse就是按照约定的编码格式对代码进行格式化的，所以只需要看看格式化后的代码长啥样就行了。具体的代码格式要求可以在Eclipse的设置中 `Java` - `Code Style` 查看。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





变量和数据类型

变量

什么是变量？

变量就是初中数学的代数的概念，例如一个简单的方程，x，y都是变量：

$y = x^2 + 1$y=x2+1

在Java中，变量分为两种：基本类型的变量和引用类型的变量。

我们先讨论基本类型的变量。

在Java中，变量必须先定义后使用，在定义变量的时候，可以给它一个初始值。例如：

```
1.  int x = 1;
```

上述语句定义了一个整型 `int` 类型的变量，名称为 `x` ，初始值为 `1` 。

不写初始值，就相当于给它指定了默认值。默认值总是 `0` 。

来看一个完整的定义变量，然后打印变量值的例子：

```
// 定义并打印变量
public class Main {
    public static void main(String[] args) {
        int x = 100; // 定义int类型变量x，并赋予初始值100
        System.out.println(x); // 打印该变量的值
    }
}
```

▶ Run

变量的一个重要特点是可以重新赋值。例如，对变量 `x` ，先赋值 `100` ，再赋值 `200` ，观察两次打印的结果：

```
// 重新赋值变量
public class Main {
    public static void main(String[] args) {
        int x = 100; // 定义int类型变量x，并赋予初始值100
        System.out.println(x); // 打印该变量的值，观察是否为100
        x = 200; // 重新赋值为200
        System.out.println(x); // 打印该变量的值，观察是否为200
    }
}
```

▶ Run

注意到第一次定义变量 `x` 的时候，需要指定变量类型 `int` ，因此使用语句 `int x = 100;` 。而第二次重新赋值的时候，变量 `x` 已经存在了，不能再重复定义，因此不能指定变量类型 `int` ，必须使用语句 `x = 200;` 。

变量不但可以重新赋值，还可以赋值给其他变量。让我们来看一个例子：

```
// 变量之间的赋值
public class Main {
    public static void main(String[] args) {
        int n = 100; // 定义变量n，同时赋值为100
        System.out.println("n = " + n); // 打印n的值

        n = 200; // 变量n赋值为200
        System.out.println("n = " + n); // 打印n的值

        int x = n; // 变量x赋值为n（n的值为200，因此赋值后x的值也是200）
        System.out.println("x = " + x); // 打印x的值

        x = x + 100; // 变量x赋值为x+100（x的值为200，因此赋值后x的值是200+100=300）
        System.out.println("x = " + x); // 打印x的值
        System.out.println("n = " + n); // 再次打印n的值，n应该是200还是300？
    }
}
```

▶ Run

我们一行一行地分析代码执行流程：

执行 `int n = 100;` ，该语句定义了变量 `n` ，同时赋值为 `100` ，因此，JVM在内存中为变量 `n` 分配一个“存储单元”，填入值 `100` ：

```
1.        n
2.        |
3.        ▼
4.   ┌────┬────┬────┬────┬────┬────┬────┬────┐
```

```
|   |100|   |   |   |   |   |
└───┴───┴───┴───┴───┴───┴───┘
```

执行 `n = 200;` 时，JVM把 `200` 写入变量 `n` 的存储单元，因此，原有的值被覆盖，现在 `n` 的值为 `200` ：

```
      n
      |
      ▼
┌───┬───┬───┬───┬───┬───┬───┐
|   |200|   |   |   |   |   |
└───┴───┴───┴───┴───┴───┴───┘
```

执行 `int x = n;` 时，定义了一个新的变量 `x` ，同时对 `x` 赋值，因此，JVM需要新分配一个存储单元给变量 `x` ，并写入和变量 `n` 一样的值，结果是变量 `x` 的值也变为 `200` ：

```
      n           x
      |           |
      ▼           ▼
┌───┬───┬───┬───┬───┬───┬───┐
|   |200|   |   |200|   |   |
└───┴───┴───┴───┴───┴───┴───┘
```

执行 `x = x + 100;` 时，JVM首先计算等式右边的值 `x + 100` ，结果为 `300` （因为此刻 `x` 的值为 `200` ），然后，将结果 `300` 写入 `x` 的存储单元，因此，变量 `x` 最终的值变为 `300` ：

```

      n           x
      |           |
      ▼           ▼
┌───┬───┬───┬───┬───┬───┬───┐
|   |200|   |   |300|   |   |
└───┴───┴───┴───┴───┴───┴───┘
```

可见，变量可以反复赋值。注意，等号 `=` 是赋值语句，不是数学意义上的相等，否则无法解释 `x = x + 100` 。

基本数据类型

基本数据类型是CPU可以直接进行运算的类型。Java定义了以下几种基本数据类型：

- 整数类型：byte，short，int，long
- 浮点数类型：float，double
- 字符类型：char
- 布尔类型：boolean

Java定义的这些基本数据类型有什么区别呢？要了解这些区别，我们就必须简单了解一下计算机内存的基本结构。

计算机内存的最小存储单元是字节（byte），一个字节就是一个8位二进制数，即8个bit。它的二进制表示范围从 `00000000` ~ `11111111` ，换算成十进制是0~255，换算成十六进制是 `00` ~ `ff` 。

内存单元从0开始编号，称为内存地址。每个内存单元可以看作一间房间，内存地址就是门牌号。

```
  0   1   2   3   4   5   6  ...
┌───┬───┬───┬───┬───┬───┬───┬───
│   │   │   │   │   │   │   │...
└───┴───┴───┴───┴───┴───┴───┴───
```

一个字节是1byte，1024字节是1K，1024K是1M，1024M是1G，1024G是1T。一个拥有4T内存的计算机的字节数量就是：

```
4T = 4 x 1024G
   = 4 x 1024 x 1024M
   = 4 x 1024 x 1024 x 1024K
   = 4 x 1024 x 1024 x 1024 x 1024
   = 4398046511104
```

不同的数据类型占用的字节数不一样。我们看一下Java基本数据类型占用的字节数：

```
        ┌─┐
   byte │ │
        └─┘

        ┌─┬─┐
  short │ │ │
        └─┴─┘

        ┌─┬─┬─┬─┐
    int │ │ │ │ │
        └─┴─┴─┴─┘

        ┌─┬─┬─┬─┬─┬─┬─┬─┐
   long │ │ │ │ │ │ │ │ │
```

```
        └──┴──┴──┴──┴──┴──┴──┴──┘

        ┌──┬──┬──┬──┐
  float │  │  │  │  │
        └──┴──┴──┴──┘

        ┌──┬──┬──┬──┬──┬──┬──┬──┐
 double │  │  │  │  │  │  │  │  │
        └──┴──┴──┴──┴──┴──┴──┴──┘

        ┌──┬──┐
   char │  │  │
        └──┴──┘
```

`byte` 恰好就是一个字节，而 `long` 和 `double` 需要8个字节。

整型

对于整型类型，Java只定义了带符号的整型，因此，最高位的bit表示符号位（0表示正数，1表示负数）。各种整型能表示的最大范围如下：

- byte: -128 ~ 127
- short: -32768 ~ 32767
- int: -2147483648 ~ 2147483647
- long: -9223372036854775808 ~ 9223372036854775807

我们来看定义整型的例子：

```
// 定义整型
public class Main {
    public static void main(String[] args) {
        int i = 2147483647;
        int i2 = -2147483648;
        int i3 = 2_000_000_000; // 加下划线更容易识别
        int i4 = 0xff0000; // 十六进制表示的16711680
        int i5 = 0b1000000000; // 二进制表示的512
        long l = 9000000000000000000L; // long型的结尾需要加L
    }
}
```

▶ Run

特别注意：同一个数的不同进制的表示是完全相同的，例如 `15` = `0xf` = `0b1111` 。

浮点型

浮点类型的数就是小数，因为小数用科学计数法表示的时候，小数点是可以“浮动”的，如1234.5可以

表示成$12.345x10^2$，也可以表示成$1.2345x10^3$，所以称为浮点数。

下面是定义浮点数的例子：

```
float f1 = 3.14f;
float f2 = 3.14e38f; // 科学计数法表示的3.14x10^38
double d = 1.79e308;
double d2 = -1.79e308;
double d3 = 4.9e-324; // 科学计数法表示的4.9x10^-324
```

对于 `float` 类型，需要加上 `f` 后缀。

浮点数可表示的范围非常大， `float` 类型可最大表示$3.4x10^{38}$，而 `double` 类型可最大表示$1.79x10^{308}$。

布尔类型

布尔类型 `boolean` 只有 `true` 和 `false` 两个值，布尔类型总是关系运算的计算结果：

```
boolean b1 = true;
boolean b2 = false;
boolean isGreater = 5 > 3; // 计算结果为true
int age = 12;
boolean isAdult = age >= 18; // 计算结果为false
```

Java语言对布尔类型的存储并没有做规定，因为理论上存储布尔类型只需要1 bit，但是通常JVM内部会把 `boolean` 表示为4字节整数。

字符类型

字符类型 `char` 表示一个字符。Java的 `char` 类型除了可表示标准的ASCII外，还可以表示一个Unicode字符：

```
// 字符类型
public class Main {
    public static void main(String[] args) {
        char a = 'A';
        char zh = '中';
        System.out.println(a);
        System.out.println(zh);
    }
}
```

▶ Run

注意 `char` 类型使用单引号 `'` ，且仅有一个字符，要和双引号 `"` 的字符串类型区分开。

常量

定义变量的时候，如果加上 `final` 修饰符，这个变量就变成了常量：

```
final double PI = 3.14; // PI是一个常量
double r = 5.0;
double area = PI * r * r;
PI = 300; // compile error!
```

常量在定义时进行初始化后就不可再次赋值，再次赋值会导致编译错误。

常量的作用是用有意义的变量名来避免魔术数字（Magic number），例如，不要在代码中到处写 `3.14` ，而是定义一个常量。如果将来需要提高计算精度，我们只需要在常量的定义处修改，例如，改成 `3.1416` ，而不必在所有地方替换 `3.14` 。

根据习惯，常量名通常全部大写。

var关键字

有些时候，类型的名字太长，写起来比较麻烦。例如：

```
StringBuilder sb = new StringBuilder();
```

这个时候，如果想省略变量类型，可以使用 `var` 关键字：

```
var sb = new StringBuilder();
```

编译器会根据赋值语句自动推断出变量 `sb` 的类型是 `StringBuilder` 。对编译器来说，语句：

```
var sb = new StringBuilder();
```

实际上会自动变成：

```
StringBuilder sb = new StringBuilder();
```

因此，使用 `var` 定义变量，仅仅是少写了变量类型而已。

变量的作用范围

在Java中，多行语句用{ }括起来。很多控制语句，例如条件判断和循环，都以{ }作为它们自身的范围，例如：

```
if (...) { // if开始
    ...
    while (...) { while 开始
        ...
        if (...) { // if开始
            ...
        } // if结束
        ...
    } // while结束
    ...
} // if结束
```

只要正确地嵌套这些{ }，编译器就能识别出语句块的开始和结束。而在语句块中定义的变量，它有一个作用域，就是从定义处开始，到语句块结束。超出了作用域引用这些变量，编译器会报错。举个例子：

```
{
    ...
    int i = 0; // 变量i从这里开始定义
    ...
    {
        ...
        int x = 1; // 变量x从这里开始定义
        ...
        {
            ...
            String s = "hello"; // 变量s从这里开始定义
            ...
```

```
        } // 变量s作用域到此结束
        ...
        // 注意，这是一个新的变量s，它和上面的变量同名，
        // 但是因为作用域不同，它们是两个不同的变量：
        String s = "hi";
        ...
    } // 变量x和s作用域到此结束
    ...
} // 变量i作用域到此结束
```

定义变量时，要遵循作用域最小化原则，尽量将变量定义在尽可能小的作用域，并且，不要重复使用变量名。

小结

Java提供了两种变量类型：基本类型和引用类型

基本类型包括整型，浮点型，布尔型，字符型。

变量可重新赋值，等号是赋值语句，不是数学意义的等号。

常量在初始化后不可重新赋值，使用常量便于理解程序意图。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





整数运算

Java的整数运算遵循四则运算规则，可以使用任意嵌套的小括号。四则运算规则和初等数学一致。例如：

```
// 四则运算
public class Main {
    public static void main(String[] args) {
        int i = (100 + 200) * (99 - 88); // 3300
        int n = 7 * (5 + (i - 9)); // 23072
        System.out.println(i);
        System.out.println(n);
    }
}
```

▶ Run

整数的数值表示不但是精确的，而且整数运算永远是精确的，即使是除法也是精确的，因为两个整数相除只能得到结果的整数部分：

```
1.  int x = 12345 / 67; // 184
```

求余运算使用 `%` ：

```
1.  int y = 12345 % 67; // 12345÷67的余数是17
```

特别注意：整数的除法对于除数为0时运行时将报错，但编译不会报错。

溢出

要特别注意，整数由于存在范围限制，如果计算结果超出了范围，就会产生溢出，而溢出不会出错，却会得到一个奇怪的结果：

```
// 运算溢出
public class Main {
    public static void main(String[] args) {
        int x = 2147483640;
        int y = 15;
        int sum = x + y;
        System.out.println(sum); // -2147483641
    }
}
```

▶ Run

要解释上述结果，我们把整数 `2147483640` 和 `15` 换成二进制做加法：

```
  0111 1111 1111 1111 1111 1111 1111 1000
+ 0000 0000 0000 0000 0000 0000 0000 1111
---------------------------------------
  1000 0000 0000 0000 0000 0000 0000 0111
```

由于最高位计算结果为 `1` ，因此，加法结果变成了一个负数。

要解决上面的问题，可以把 `int` 换成 `long` 类型，由于 `long` 可表示的整型范围更大，所以结果就不会溢出：

```
long x = 2147483640;
long y = 15;
long sum = x + y;
System.out.println(sum); // 2147483655
```

还有一种简写的运算符，即 `+=` ， `-=` ， `*=` ， `/=` ，它们的使用方法如下：

```
n += 100; // 3409, 相当于 n = n + 100;
n -= 100; // 3309, 相当于 n = n - 100;
```

自增/自减

Java还提供了 `++` 运算和 `--` 运算，它们可以对一个整数进行加1和减1的操作：

```
// 自增/自减运算
public class Main {
    public static void main(String[] args) {
        int n = 3300;
        n++; // 3301, 相当于 n = n + 1;
        n--; // 3300, 相当于 n = n - 1;
        int y = 100 + (++n); // 不要这么写
        System.out.println(y);
    }
}
```

▶ Run

注意 `++` 写在前面和后面计算结果是不同的， `++n` 表示先加1再引用n， `n++` 表示先引用n再加1。不建议把 `++` 运算混入到常规运算中，容易自己把自己搞懵了。

移位运算

在计算机中，整数总是以二进制的形式表示。例如， `int` 类型的整数 `7` 使用4字节表示的二进制如下：

```
00000000 0000000 0000000 00000111
```

可以对整数进行移位运算。对整数 `7` 左移1位将得到整数 `14` ，左移两位将得到整数 `28` ：

```
int n = 7;       // 00000000 00000000 00000000 00000111 = 7
int a = n << 1;  // 00000000 00000000 00000000 00001110 = 14
int b = n << 2;  // 00000000 00000000 00000000 00011100 = 28
int c = n << 28; // 01110000 00000000 00000000 00000000 = 1879048192
int d = n << 29; // 11100000 00000000 00000000 00000000 = -536870912
```

左移29位时，由于最高位变成 `1` ，因此结果变成了负数。

类似的，对整数28进行右移，结果如下：

```
int n = 7;       // 00000000 00000000 00000000 00000111 = 7
int a = n >> 1;  // 00000000 00000000 00000000 00000011 = 3
int b = n >> 2;  // 00000000 00000000 00000000 00000001 = 1
int c = n >> 3;  // 00000000 00000000 00000000 00000000 = 0
```

如果对一个负数进行右移，最高位的 `1` 不动，结果仍然是一个负数：

```
int n = -536870912;
```

```
int a = n >> 1;  // 11110000 00000000 00000000 00000000 = -268435456
int b = n >> 2;  // 10111000 00000000 00000000 00000000 = -134217728
int c = n >> 28; // 11111111 11111111 11111111 11111110 = -2
int d = n >> 29; // 11111111 11111111 11111111 11111111 = -1
```

还有一种不带符号的右移运算，使用 `>>>` ，它的特点是符号位跟着动，因此，对一个负数进行 `>>>` 右移，它会变成正数，原因是最高位的 `1` 变成了 `0` ：

```
int n = -536870912;
int a = n >>> 1;  // 01110000 00000000 00000000 00000000 = 1879048192
int b = n >>> 2;  // 00111000 00000000 00000000 00000000 = 939524096
int c = n >>> 29; // 00000000 00000000 00000000 00000111 = 7
int d = n >>> 31; // 00000000 00000000 00000000 00000001 = 1
```

对 `byte` 和 `short` 类型进行移位时，会首先转换为 `int` 再进行位移。

仔细观察可发现，左移实际上就是不断地×2，右移实际上就是不断地÷2。

位运算

位运算是按位进行与、或、非和异或的运算。

与运算的规则是，必须两个数同时为 `1` ，结果才为 `1` ：

```
n = 0 & 0; // 0
n = 0 & 1; // 0
n = 1 & 0; // 0
n = 1 & 1; // 1
```

或运算的规则是，只要任意一个为 `1` ，结果就为 `1` ：

```
n = 0 | 0; // 0
n = 0 | 1; // 1
n = 1 | 0; // 1
n = 1 | 1; // 1
```

非运算的规则是， `0` 和 `1` 互换：

```
n = ~0; // 1
n = ~1; // 0
```

异或运算的规则是，如果两个数不同，结果为 `1` ，否则为 `0` ：

```
1. n = 0 ^ 0; // 0
2. n = 0 ^ 1; // 1
3. n = 1 ^ 0; // 1
4. n = 1 ^ 1; // 0
```

对两个整数进行位运算，实际上就是按位对齐，然后依次对每一位进行运算。例如：

```
// 位运算
public class Main {
    public static void main(String[] args) {
        int i = 167776589; // 00001010 00000000 00010001 01001101
        int n = 167776512; // 00001010 00000000 00010001 00000000
        System.out.println(i & n); // 167776512
    }
}
```

▶ Run

上述按位与运算实际上可以看作两个整数表示的IP地址 `10.0.17.77` 和 `10.0.17.0` ，通过与运算，可以快速判断一个IP是否在给定的网段内。

运算优先级

在Java的计算表达式中，运算优先级从高到低依次是：

- `()`
- `!` `~` `++` `--`
- `*` `/` `%`
- `+` `-`
- `<<` `>>` `>>>`
- `&`
- `|`
- `+=` `-=` `*=` `/=`

记不住也没关系，只需要加括号就可以保证运算的优先级正确。

类型自动提升与强制转型

在运算过程中，如果参与运算的两个数类型不一致，那么计算结果为较大类型的整型。例如， `short` 和 `int` 计算，结果总是 `int` ，原因是 `short` 首先自动被转型为 `int` ：

```
// 类型自动提升与强制转型
public class Main {
    public static void main(String[] args) {
        short s = 1234;
        int i = 123456;
        int x = s + i; // s自动转型为int
        short y = s + i; // 编译错误!
    }
}
```

▶ Run

也可以将结果强制转型，即将大范围的整数转型为小范围的整数。强制转型使用 `(类型)` ，例如，将 `int` 强制转型为 `short` ：

```
1. int i = 12345;
2. short s = (short) i; // 12345
```

要注意，超出范围的强制转型会得到错误的结果，原因是转型时， `int` 的两个高位字节直接被扔掉，仅保留了低位的两个字节：

```
// 强制转型
public class Main {
    public static void main(String[] args) {
        int i1 = 1234567;
        short s1 = (short) i1; // -10617
        System.out.println(s1);
        int i2 = 12345678;
        short s2 = (short) i2; // 24910
        System.out.println(s2);
    }
}
```

▶ Run

因此，强制转型的结果很可能是错的。

练习

计算前N个自然数的和可以根据公示：

$\frac{(1+N)\times N}{2}$ 2(1+N)×N

请根据公式计算前N个自然数的和：

```
// 计算前N个自然数的和
public class Main {
    public static void main(String[] args) {
        int n = 100;
        // TODO: sum = 1 + 2 + ... + n
        int sum = ???;
        System.out.println(sum);
        System.out.println(sum == 5050 ? "测试通过" : "测试失败");
    }
}
```

▶ Run

下载练习：计算前N个自然数的和 （推荐使用IDE练习插件快速下载）

小结

整数运算的结果永远是精确的；

运算结果会自动提升；

可以强制转型，但超出范围的强制转型会得到错误的结果；

应该选择合适范围的整型（`int`或`long`），没有必要为了节省内存而使用`byte`和`short`进行整数运算。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





浮点数运算

浮点数运算和整数运算相比，只能进行加减乘除这些数值计算，不能做位运算和移位运算。

在计算机中，浮点数虽然表示的范围大，但是，浮点数有个非常重要的特点，就是浮点数常常无法精确表示。

举个栗子：

浮点数 `0.1` 在计算机中就无法精确表示，因为十进制的 `0.1` 换算成二进制是一个无限循环小数，很显然，无论使用 `float` 还是 `double` ，都只能存储一个 `0.1` 的近似值。但是， `0.5` 这个浮点数又可以精确地表示。

因为浮点数常常无法精确表示，因此，浮点数运算会产生误差：

```
// 浮点数运算误差
public class Main {
    public static void main(String[] args) {
        double x = 1.0 / 10;
        double y = 1 - 9.0 / 10;
        // 观察x和y是否相等:
        System.out.println(x);
        System.out.println(y);
    }
}
```

▶ Run

由于浮点数存在运算误差，所以比较两个浮点数是否相等常常会出现错误的结果。正确的比较方法是判断两个浮点数之差的绝对值是否小于一个很小的数：

```
// 比较x和y是否相等，先计算其差的绝对值:
double r = Math.abs(x - y);
// 再判断绝对值是否足够小:
if (r < 0.00001) {
    // 可以认为相等
} else {
    // 不相等
}
```

浮点数在内存的表示方法和整数比更加复杂。Java的浮点数完全遵循IEEE-754标准，这也是绝大多数计算机平台都支持的浮点数标准表示方法。

类型提升

如果参与运算的两个数其中一个是整型，那么整型可以自动提升到浮点型：

```
// 类型提升
public class Main {
    public static void main(String[] args) {
        int n = 5;
        double d = 1.2 + 24.0 / n; // 6.0
        System.out.println(d);
    }
}
```

▶ Run

需要特别注意，在一个复杂的四则运算中，两个整数的运算不会出现自动提升的情况。例如：

```
1.  double d = 1.2 + 24 / 5; // 5.2
```

计算结果为 `5.2` ，原因是编译器计算 `24 / 5` 这个子表达式时，按两个整数进行运算，结果仍为整数 `4` 。

溢出

整数运算在除数为 `0` 时会报错，而浮点数运算在除数为 `0` 时，不会报错，但会返回几个特殊值：

- `NaN` 表示Not a Number
- `Infinity` 表示无穷大
- `-Infinity` 表示负无穷大

例如：

```
1.  double d1 = 0.0 / 0; // NaN
2.  double d2 = 1.0 / 0; // Infinity
3.  double d3 = -1.0 / 0; // -Infinity
```

这三种特殊值在实际运算中很少碰到，我们只需要了解即可。

强制转型

可以将浮点数强制转型为整数。在转型时，浮点数的小数部分会被丢掉。如果转型后超过了整型能表示的最大范围，将返回整型的最大值。例如：

```
int n1 = (int) 12.3; // 12
int n2 = (int) 12.7; // 12
int n2 = (int) -12.7; // -12
int n3 = (int) (12.7 + 0.5); // 13
int n4 = (int) 1.2e20; // 2147483647
```

如果要进行四舍五入，可以对浮点数加上0.5再强制转型：

```
// 四舍五入
public class Main {
    public static void main(String[] args) {
        double d = 2.6;
        int n = (int) (d + 0.5);
        System.out.println(n);
    }
}
```

▶ Run

练习

根据一元二次方程$ax^2 + bx + c = 0$ax2+bx+c=0的求根公式：

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$2a−b±b2−4ac

计算出一元二次方程的两个解：

```
// 一元二次方程
public class Main {
    public static void main(String[] args) {
        double a = 1.0;
        double b = 3.0;
        double c = -4.0;
        // 求平方根可用 Math.sqrt():
        // System.out.println(Math.sqrt(2)); ==> 1.414
        // TODO:
        double r1 = 0;
        double r2 = 0;
        System.out.println(r1);
        System.out.println(r2);
        System.out.println(r1 == 1 && r2 == -4 ? "测试通过" : "测试失败");
    }
}
```

▶ Run

下载练习：计算一元二次方程的两个解 （推荐使用IDE练习插件快速下载）

小结

浮点数常常无法精确表示，并且浮点数的运算结果可能有误差；

比较两个浮点数通常比较它们的绝对值之差是否小于一个特定值；

整型和浮点型运算时，整型会自动提升为浮点型；

可以将浮点型强制转为整型，但超出范围后将始终返回整型的最大值。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





布尔运算

对于布尔类型 `boolean`，永远只有 `true` 和 `false` 两个值。

布尔运算是一种关系运算，包括以下几类：

- 比较运算符：`>`，`>=`，`<`，`<=`，`==`，`!=`
- 与运算 `&&`
- 或运算 `||`
- 非运算 `!`

下面是一些示例：

```
boolean isGreater = 5 > 3; // true
int age = 12;
boolean isZero = age == 0; // false
boolean isNonZero = !isZero; // true
boolean isAdult = age >= 18; // false
boolean isTeenager = age >6 && age <18; // true
```

关系运算符的优先级从高到低依次是：

- `!`
- `>`，`>=`，`<`，`<=`
- `==`，`!=`
- `&&`
- `||`

短路运算

布尔运算的一个重要特点是短路运算。如果一个布尔运算的表达式能提前确定结果，则后续的计算不再执行，直接返回结果。

因为 `false && x` 的结果总是 `false`，无论 `x` 是 `true` 还是 `false`，因此，与运算在确定第一个值为 `false` 后，不再继续计算，而是直接返回 `false`。

我们考察以下代码：

```
// 短路运算
public class Main {
    public static void main(String[] args) {
        boolean b = 5 < 3;
        boolean result = b && (5 / 0 > 0);
        System.out.println(result);
    }
}
```

▶ Run

如果没有短路运算， `&&` 后面的表达式会由于除数为 `0` 而报错，但实际上该语句并未报错，原因在于与运算是短路运算符，提前计算出了结果 `false` 。

如果变量 `b` 的值为 `true` ，则表达式变为 `true && (5 / 0 > 0)` 。因为无法进行短路运算，该表达式必定会由于除数为 `0` 而报错，可以自行测试。

类似的，对于 `||` 运算，只要能确定第一个值为 `true` ，后续计算也不再进行，而是直接返回 `true` ：

```
1.  boolean result = true || (5 / 0 > 0); // true
```

三元运算符

Java还提供一个三元运算符 `b ? x : y` ，它根据第一个布尔表达式的结果，分别返回后续两个表达式之一的计算结果。示例：

```
// 三元运算
public class Main {
    public static void main(String[] args) {
        int n = -100;
        int x = n >= 0 ? n : -n;
        System.out.println(x);
    }
}
```

▶ Run

上述语句的意思是，判断 `n >= 0` 是否成立，如果为 `true` ，则返回 `n` ，否则返回 `-n` 。这实际上是一个求绝对值的表达式。

注意到三元运算 `b ? x : y` 会首先计算 `b` ，如果 `b` 为 `true` ，则只计算 `x` ，否则，只计

算 `y` 。此外， `x` 和 `y` 的类型必须相同，因为返回值不是 `boolean` ，而是 `x` 和 `y` 之一。

练习

判断指定年龄是否是小学生（6 ~ 12岁）：

```
// 布尔运算
public class Main {
    public static void main(String[] args) {
        int age = 7;
        // primary student的定义: 6~12岁
        boolean isPrimaryStudent = ???;
        System.out.println(isPrimaryStudent ? "Yes" : "No");
    }
}
```

▶ Run

下载练习：判断指定年龄是否是小学生 （推荐使用IDE练习插件快速下载）

小结

与运算和或运算是短路运算；

三元运算 `b ? x : y` 后面的类型必须相同，三元运算也是“短路运算”，只计算 `x` 或 `y` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





字符和字符串

在Java中，字符和字符串是两个不同的类型。

字符类型

字符类型 `char` 是基本数据类型，它是 `character` 的缩写。一个 `char` 保存一个Unicode字符：

```
1. char c1 = 'A';
2. char c2 = '中';
```

因为Java在内存中总是使用Unicode表示字符，所以，一个英文字符和一个中文字符都用一个 `char` 类型表示，它们都占用两个字节。要显示一个字符的Unicode编码，只需将 `char` 类型直接赋值给 `int` 类型即可：

```
1. int n1 = 'A'; // 字母“A”的Unicodde编码是65
2. int n2 = '中'; // 汉字“中”的Unicode编码是20013
```

还可以直接用转义字符 `\u` +Unicode编码来表示一个字符：

```
1. // 注意是十六进制:
2. char c3 = '\u0041'; // 'A', 因为十六进制0041 = 十进制65
3. char c4 = '\u4e2d'; // '中', 因为十六进制4e2d = 十进制20013
```

字符串类型

和 `char` 类型不同，字符串类型 `String` 是引用类型，我们用双引号 `"…"` 表示字符串。一个字符串可以存储0个到任意个字符：

```
1. String s = ""; // 空字符串，包含0个字符
2. String s1 = "A"; // 包含一个字符
3. String s2 = "ABC"; // 包含3个字符
4. String s3 = "中文 ABC"; // 包含6个字符，其中有一个空格
```

因为字符串使用双引号 `"…"` 表示开始和结束，那如果字符串本身恰好包含一个 `"` 字符怎么表示？例如， `"abc"xyz"` ，编译器就无法判断中间的引号究竟是字符串的一部分还是表示字符串结束。这个时候，我们需要借助转义字符 `\` ：

```
1. String s = "abc\"xyz"; // 包含7个字符: a, b, c, ", x, y, z
```

因为 `\` 是转义字符，所以，两个 `\` 表示一个 `\` 字符：

```
String s = "abc\\xyz"; // 包含7个字符: a, b, c, \, x, y, z
```

常见的转义字符包括：

- `\"` 表示字符 `"`
- `\'` 表示字符 `'`
- `\` 表示字符 `\`
- `\n` 表示换行符
- `\r` 表示回车符
- `\t` 表示Tab
- `\u####` 表示一个Unicode编码的字符

例如：

```
String s = "ABC\n\u4e2d\u6587"; // 包含6个字符: A, B, C, 换行符, 中, 文
```

字符串连接

Java的编译器对字符串做了特殊照顾，可以使用 `+` 连接任意字符串和其他数据类型，这样极大地方便了字符串的处理。例如：

```
// 字符串连接
public class Main {
    public static void main(String[] args) {
        String s1 = "Hello";
        String s2 = "world";
        String s = s1 + " " + s2 + "!";
        System.out.println(s);
    }
}
```

▶ Run

如果用 `+` 连接字符串和其他数据类型，会将其他数据类型先自动转型为字符串，再连接：

```
// 字符串连接
public class Main {
    public static void main(String[] args) {
        int age = 25;
        String s = "age is " + age;
        System.out.println(s);
    }
}
```

▶ Run

多行字符串

如果我们要表示多行字符串，使用+号连接会非常不方便：

```
String s = "first line \n"
         + "second line \n"
         + "end";
```

从Java 13开始，字符串可以用 `"""…"""` 表示多行字符串（Text Blocks）了。举个例子：

```
// 多行字符串
public class Main {
    public static void main(String[] args) {
        String s = """
                   SELECT * FROM
                     users
                   WHERE id > 100
                   ORDER BY name DESC
                   """;
        System.out.println(s);
    }
}
```

▶ Run

上述多行字符串实际上是5行，在最后一个 `DESC` 后面还有一个 `\n` 。如果我们不想在字符串末尾加一个 `\n` ，就需要这么写：

```
String s = """
           SELECT * FROM
             users
           WHERE id > 100
           ORDER BY name DESC""";
```

还需要注意到，多行字符串前面共同的空格会被去掉，即：

```
String s = """
..........SELECT * FROM
..........  users
..........WHERE id > 100
..........ORDER BY name DESC
..........""";
```

用`.`标注的空格都会被去掉。

如果多行字符串的排版不规则，那么，去掉的空格就会变成这样：

```
String s = """
.........  SELECT * FROM
.........    users
.........WHERE id > 100
.........  ORDER BY name DESC
.........  """;
```

即总是以最短的行首空格为基准。

最后，由于多行字符串是作为Java 13的预览特性（Preview Language Features）实现的，编译的时候，我们还需要给编译器加上参数：

```
javac --source 13 --enable-preview Main.java
```

不可变特性

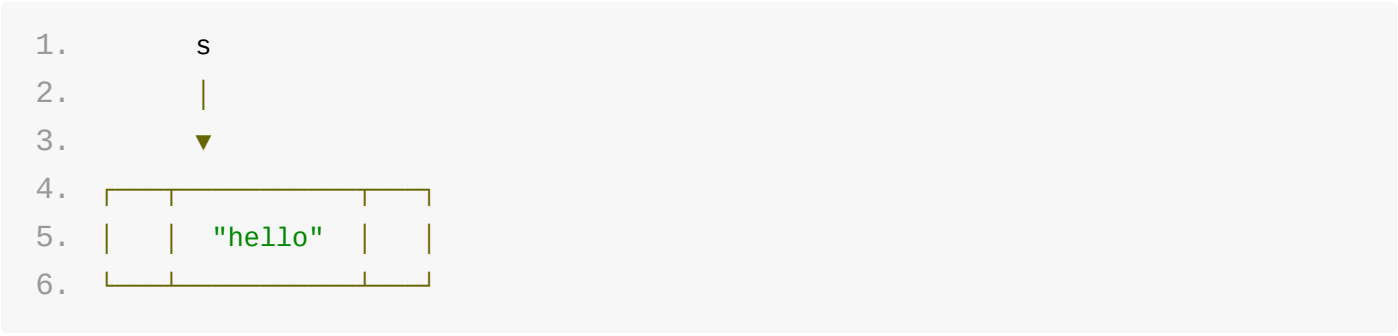
Java的字符串除了是一个引用类型外，还有个重要特点，就是字符串不可变。考察以下代码：

```
// 字符串不可变
public class Main {
    public static void main(String[] args) {
        String s = "hello";
        System.out.println(s); // 显示 hello
        s = "world";
        System.out.println(s); // 显示 world
    }
}
```

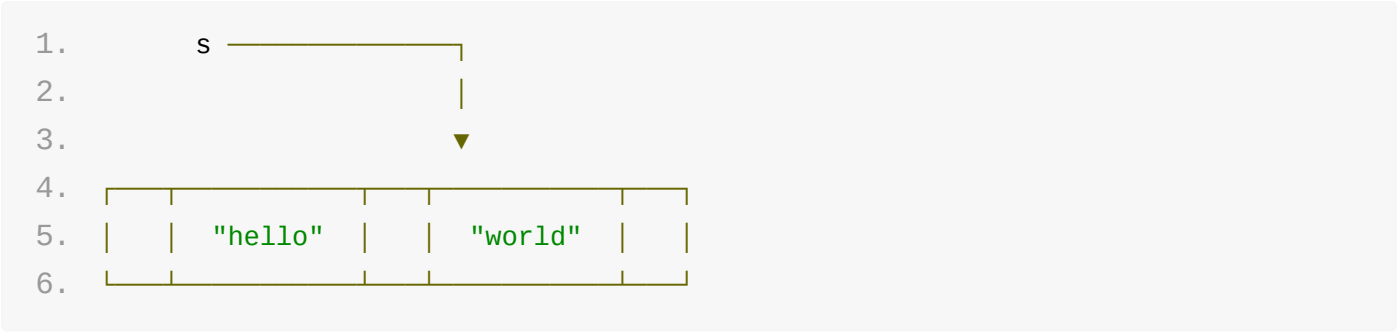
▶ Run

观察执行结果，难道字符串 `s` 变了吗？其实变的不是字符串，而是变量 `s` 的“指向”。

执行 `String s = "hello";` 时，JVM虚拟机先创建字符串 `"hello"` ，然后，把字符串变量 `s` 指向它：



紧接着，执行 `s = "world";` 时，JVM虚拟机先创建字符串 `"world"` ，然后，把字符串变量 `s` 指向它：



原来的字符串 `"hello"` 还在，只是我们无法通过变量 `s` 访问它而已。因此，字符串的不可变是指字符串内容不可变。

理解了引用类型的“指向”后，试解释下面的代码输出：

```
// 字符串不可变
public class Main {
    public static void main(String[] args) {
        String s = "hello";
        String t = s;
        s = "world";
        System.out.println(t); // t是"hello"还是"world"?
    }
}
```

▶ Run

空值null

引用类型的变量可以指向一个空值 `null` ，它表示不存在，即该变量不指向任何对象。例如：

```
String s1 = null; // s1是null
String s2; // 没有赋初值值，s2也是null
String s3 = s1; // s3也是null
String s4 = ""; // s4指向空字符串，不是null
```

注意要区分空值 `null` 和空字符串 `""` ，空字符串是一个有效的字符串对象，它不等于 `null` 。

练习

请将一组int值视为字符的Unicode编码，然后将它们拼成一个字符串：

```
public class Main {
    public static void main(String[] args) {
        // 请将下面一组int值视为字符的Unicode码，把它们拼成一个字符串：
        int a = 72;
        int b = 105;
        int c = 65281;
        // FIXME:
        String s = a + b + c;
        System.out.println(s);
    }
}
```

▶ Run

下载练习：Unicode值拼接字符串 （推荐使用IDE练习插件快速下载）

小结

Java的字符类型 `char` 是基本类型，字符串类型 `String` 是引用类型；

基本类型的变量是“持有”某个数值，引用类型的变量是“指向”某个对象；

引用类型的变量可以是空值 `null` ；

要区分空值 `null` 和空字符串 `""` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





数组类型

如果我们有一组类型相同的变量，例如，5位同学的成绩，可以这么写：

```
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int n1 = 68;
        int n2 = 79;
        int n3 = 91;
        int n4 = 85;
        int n5 = 62;
    }
}
```

但其实没有必要定义5个 `int` 变量。可以使用数组来表示“一组” `int` 类型。代码如下：

```
// 数组
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns = new int[5];
        ns[0] = 68;
        ns[1] = 79;
        ns[2] = 91;
        ns[3] = 85;
        ns[4] = 62;
    }
}
```

▶ Run

定义一个数组类型的变量，使用数组类型“类型[]”，例如， `int[]` 。和单个基本类型变量不同，数组变量初始化必须使用 `new int[5]` 表示创建一个可容纳5个 `int` 元素的数组。

Java的数组有几个特点：

- 数组所有元素初始化为默认值，整型都是 `0` ，浮点型是 `0.0` ，布尔型是 `false` ；
- 数组一旦创建后，大小就不可改变。

要访问数组中的某一个元素，需要使用索引。数组索引从 `0` 开始，例如，5个元素的数组，索引范围是 `0` ~ `4` 。

可以修改数组中的某一个元素，使用赋值语句，例如， `ns[1] = 79;` 。

可以用 `数组变量.length` 获取数组大小：

```
// 数组
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns = new int[5];
        System.out.println(ns.length); // 5
    }
}
```

▶ Run

数组是引用类型，在使用索引访问数组元素时，如果索引超出范围，运行时将报错：

```
// 数组
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns = new int[5];
        int n = 5;
        System.out.println(ns[n]); // 索引n不能超出范围
    }
}
```

▶ Run

也可以在定义数组时直接指定初始化的元素，这样就不必写出数组大小，而是由编译器自动推算数组大小。例如：

```
// 数组
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns = new int[] { 68, 79, 91, 85, 62 };
        System.out.println(ns.length); // 编译器自动推算数组大小为5
    }
}
```

▶ Run

还可以进一步简写为：

```
int[] ns = { 68, 79, 91, 85, 62 };
```

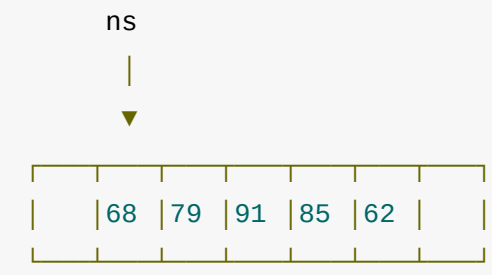
注意数组是引用类型，并且数组大小不可变。我们观察下面的代码：

```
// 数组
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns;
        ns = new int[] { 68, 79, 91, 85, 62 };
        System.out.println(ns.length); // 5
        ns = new int[] { 1, 2, 3 };
        System.out.println(ns.length); // 3
    }
}
```

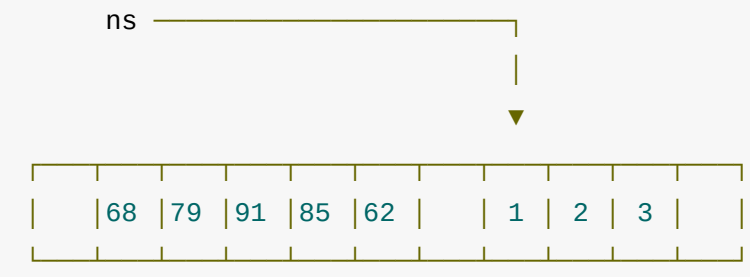
▶ Run

数组大小变了吗？看上去好像是变了，但其实根本没变。

对于数组 `ns` 来说，执行 `ns = new int[] { 68, 79, 91, 85, 62 };` 时，它指向一个5个元素的数组：



执行 `ns = new int[] { 1, 2, 3 };` 时，它指向一个_新的_3个元素的数组：



但是，原有的5个元素的数组并没有改变，只是无法通过变量 `ns` 引用到它们而已。

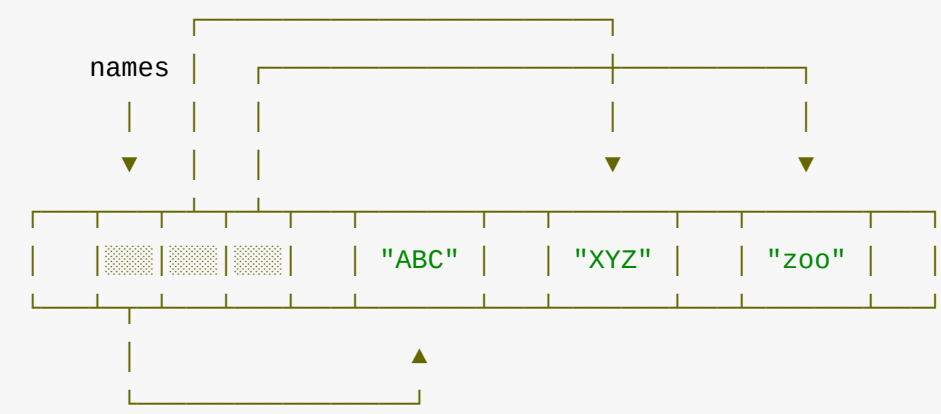
字符串数组

如果数组元素不是基本类型，而是一个引用类型，那么，修改数组元素会有哪些不同？

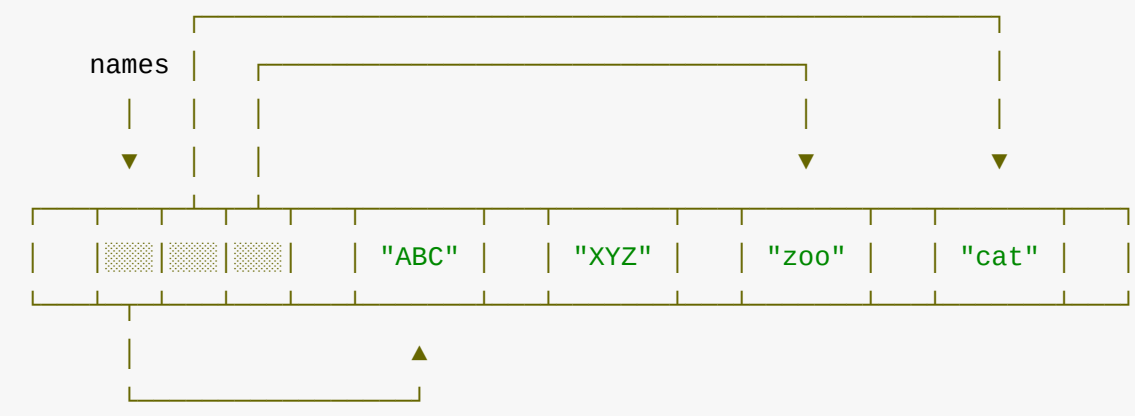
字符串是引用类型，因此我们先定义一个字符串数组：

```
String[] names = {
    "ABC", "XYZ", "zoo"
};
```

对于 `String[]` 类型的数组变量 `names` ，它实际上包含3个元素，但每个元素都指向某个字符串对象：



对 `names[1]` 进行赋值，例如 `names[1] = "cat";` ，效果如下：



这里注意到原来 `names[1]` 指向的字符串 `"XYZ"` 并没有改变，仅仅是将 `names[1]` 的引用从指向 `"XYZ"` 改成了指向 `"cat"` ，其结果是字符串 `"XYZ"` 再也无法通过 `names[1]` 访问到了。

对“指向”有了更深入的理解后，试解释如下代码：

```
// 数组
public class Main {
    public static void main(String[] args) {
        String[] names = {"ABC", "XYZ", "zoo"};
        String s = names[1];
        names[1] = "cat";
        System.out.println(s); // s是"XYZ"还是"cat"?
    }
}
```

▶ Run

小结

数组是同一数据类型的集合，数组一旦创建后，大小就不可变；

可以通过索引访问数组元素，但索引超出范围将报错；

数组元素可以是值类型（如`int`）或引用类型（如`String`），但数组本身是引用类型；

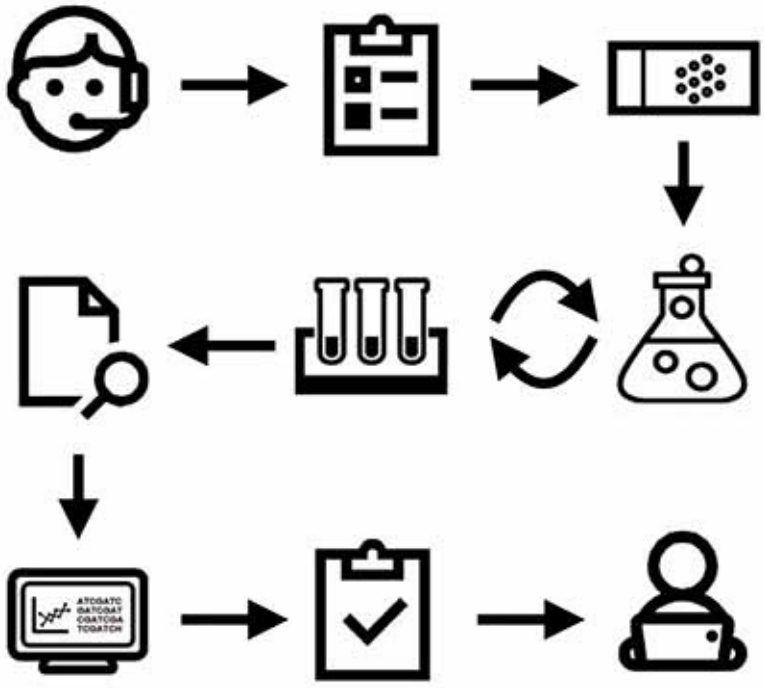
读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





流程控制

在Java程序中，JVM默认总是顺序执行以分号 `;` 结束的语句。但是，在实际的代码中，程序经常需要做条件判断、循环，因此，需要有多种流程控制语句，来实现程序的跳转和循环等功能。



本节我们将介绍 `if` 条件判断、 `switch` 多重选择和各种循环语句。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





输入和输出

输出

在前面的代码中，我们总是使用 `System.out.println()` 来向屏幕输出一些内容。

`println` 是print line的缩写，表示输出并换行。因此，如果输出后不想换行，可以用 `print()` ：

```
// 输出
public class Main {
    public static void main(String[] args) {
        System.out.print("A,");
        System.out.print("B,");
        System.out.print("C.");
        System.out.println();
        System.out.println("END");
    }
}
```

▶ Run

注意观察上述代码的执行效果。

格式化输出

Java还提供了格式化输出的功能。为什么要格式化输出？因为计算机表示的数据不一定适合人来阅读：

```
// 格式化输出
public class Main {
    public static void main(String[] args) {
        double d = 12900000;
        System.out.println(d); // 1.29E7
    }
}
```

▶ Run

如果要把数据显示成我们期望的格式，就需要使用格式化输出的功能。格式化输出使用 `System.out.printf()` ，通过使用占位符 `%?` ， `printf()` 可以把后面的参数格式化成指定格式：

```
// 格式化输出
public class Main {
    public static void main(String[] args) {
        double d = 3.1415926;
        System.out.printf("%.2f\n", d); // 显示两位小数3.14
        System.out.printf("%.4f\n", d); // 显示4位小数3.1416
    }
}
```

▶ Run

Java的格式化功能提供了多种占位符，可以把各种数据类型“格式化”成指定的字符串：

占位符	说明
%d	格式化输出整数
%x	格式化输出十六进制整数
%f	格式化输出浮点数
%e	格式化输出科学计数法表示的浮点数
%s	格式化字符串

注意，由于%表示占位符，因此，连续两个%%表示一个%字符本身。

占位符本身还可以有更详细的格式化参数。下面的例子把一个整数格式化成十六进制，并用0补足8位：

```
// 格式化输出
public class Main {
    public static void main(String[] args) {
        int n = 12345000;
        System.out.printf("n=%d, hex=%08x", n, n); // 注意，两个%占位符必须传入两个数
    }
}
```

▶ Run

详细的格式化参数请参考JDK文档java.util.Formatter

输入

和输出相比，Java的输入就要复杂得多。

我们先看一个从控制台读取一个字符串和一个整数的例子：

```java
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // 创建Scanner对象
        System.out.print("Input your name: "); // 打印提示
        String name = scanner.nextLine(); // 读取一行输入并获取字符串
        System.out.print("Input your age: "); // 打印提示
        int age = scanner.nextInt(); // 读取一行输入并获取整数
        System.out.printf("Hi, %s, you are %d\n", name, age); // 格式化输出
    }
}
```

首先，我们通过 `import` 语句导入 `java.util.Scanner` ， `import` 是导入某个类的语句，必须放到Java源代码的开头，后面我们在Java的 `package` 中会详细讲解如何使用 `import` 。

然后，创建 `Scanner` 对象并传入 `System.in` 。 `System.out` 代表标准输出流，而 `System.in` 代表标准输入流。直接使用 `System.in` 读取用户输入虽然是可以的，但需要更复杂的代码，而通过 `Scanner` 就可以简化后续的代码。

有了 `Scanner` 对象后，要读取用户输入的字符串，使用 `scanner.nextLine()` ，要读取用户输入的整数，使用 `scanner.nextInt()` 。 `Scanner` 会自动转换数据类型，因此不必手动转换。

要测试输入，我们不能在线运行它，因为输入必须从命令行读取，因此，需要走编译、执行的流程：

```
$ javac Main.java
```

这个程序编译时如果有警告，可以暂时忽略它，在后面学习IO的时候再详细解释。编译成功后，执行：

```
$ java Main
Input your name: Bob
Input your age: 12
Hi, Bob, you are 12
```

根据提示分别输入一个字符串和整数后，我们得到了格式化的输出。

练习

请帮小明同学设计一个程序，输入上次考试成绩（int）和本次考试成绩（int），然后输出成绩提高的百分比，保留两位小数位（例如，21.75%）。

下载练习：输入和输出练习 （推荐使用IDE练习插件快速下载）

小结

Java提供的输出包括：`System.out.println()` / `print()` / `printf()`，其中`printf()`可以格式化输出；

Java提供Scanner对象来方便输入，读取对应的类型可以使用：`scanner.nextLine()` / `nextInt()` / `nextDouble()` / …

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





if判断

在Java程序中，如果要根据条件来决定是否执行某一段代码，就需要 `if` 语句。

`if` 语句的基本语法是：

```
if (条件) {
    // 条件满足时执行
}
```

根据 `if` 的计算结果（ `true` 还是 `false` ），JVM决定是否执行 `if` 语句块（即花括号{}包含的所有语句）。

让我们来看一个例子：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 70;
        if (n >= 60) {
            System.out.println("及格了");
        }
        System.out.println("END");
    }
}
```

▶ Run

当条件 `n >= 60` 计算结果为 `true` 时， `if` 语句块被执行，将打印 `"及格了"` ，否则， `if` 语句块将被跳过。修改 `n` 的值可以看到执行效果。

注意到 `if` 语句包含的块可以包含多条语句：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 70;
        if (n >= 60) {
            System.out.println("及格了");
            System.out.println("恭喜你");
        }
        System.out.println("END");
    }
}
```

▶ Run

当 `if` 语句块只有一行语句时，可以省略花括号{}：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 70;
        if (n >= 60)
            System.out.println("及格了");
        System.out.println("END");
    }
}
```

▶ Run

但是，省略花括号并不总是一个好主意。假设某个时候，突然想给 `if` 语句块增加一条语句时：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 50;
        if (n >= 60)
            System.out.println("及格了");
            System.out.println("恭喜你"); // 注意这条语句不是if语句块的一部分
        System.out.println("END");
    }
}
```

▶ Run

由于使用缩进格式，很容易把两行语句都看成 `if` 语句的执行块，但实际上只有第一行语句是 `if` 的执行块。在使用git这些版本控制系统自动合并时更容易出问题，所以不推荐忽略花括号的写法。

else

`if` 语句还可以编写一个 `else { … }` ，当条件判断为 `false` 时，将执行 `else` 的语句块：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 70;
        if (n >= 60) {
            System.out.println("及格了");
        } else {
            System.out.println("挂科了");
        }
        System.out.println("END");
    }
}
```

▶ Run

修改上述代码 `n` 的值，观察 `if` 条件为 `true` 或 `false` 时，程序执行的语句块。

注意， `else` 不是必须的。

还可以用多个 `if … else if …` 串联。例如：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 70;
        if (n >= 90) {
            System.out.println("优秀");
        } else if (n >= 60) {
            System.out.println("及格了");
        } else {
            System.out.println("挂科了");
        }
        System.out.println("END");
    }
}
```

▶ Run

串联的效果其实相当于：

```
1. if (n >= 90) {
2.     // n >= 90为true:
3.     System.out.println("优秀");
4. } else {
```

```
    // n >= 90为false:
    if (n >= 60) {
        // n >= 60为true:
        System.out.println("及格了");
    } else {
        // n >= 60为false:
        System.out.println("挂科了");
    }
}
```

在串联使用多个 `if` 时，要特别注意判断顺序。观察下面的代码：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 100;
        if (n >= 60) {
            System.out.println("及格了");
        } else if (n >= 90) {
            System.out.println("优秀");
        } else {
            System.out.println("挂科了");
        }
    }
}
```

▶ Run

执行发现， `n = 100` 时，满足条件 `n >= 90` ，但输出的不是 `"优秀"` ，而是 `"及格了"` ，原因是 `if` 语句从上到下执行时，先判断 `n >= 60` 成功后，后续 `else` 不再执行，因此， `if (n >= 90)` 没有机会执行了。

正确的方式是按照判断范围从大到小依次判断：

```
// 从大到小依次判断：
if (n >= 90) {
    // ...
} else if (n >= 60) {
    // ...
} else {
    // ...
}
```

或者改写成从小到大依次判断：

```
// 从小到大依次判断：
if (n < 60) {
    // ...
} else if (n < 90) {
    // ...
} else {
    // ...
}
```

使用 `if` 时，还要特别注意边界条件。例如：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        int n = 90;
        if (n > 90) {
            System.out.println("优秀");
        } else if (n >= 60) {
            System.out.println("及格了");
        } else {
            System.out.println("挂科了");
        }
    }
}
```

▶ Run

假设我们期望90分或更高为“优秀”，上述代码输出的却是“及格”，原因是 `>` 和 `>=` 效果是不同的。

前面讲过了浮点数在计算机中常常无法精确表示，并且计算可能出现误差，因此，判断浮点数相等用 `==` 判断不靠谱：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        double x = 1 - 9.0 / 10;
        if (x == 0.1) {
            System.out.println("x is 0.1");
        } else {
            System.out.println("x is NOT 0.1");
        }
    }
}
```

▶ Run

正确的方法是利用差值小于某个临界值来判断：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        double x = 1 - 9.0 / 10;
        if (Math.abs(x - 0.1) < 0.00001) {
            System.out.println("x is 0.1");
        } else {
            System.out.println("x is NOT 0.1");
        }
    }
}
```

▶ Run

判断引用类型相等

在Java中，判断值类型的变量是否相等，可以使用 `==` 运算符。但是，判断引用类型的变量是否相等， `==` 表示“引用是否相等”，或者说，是否指向同一个对象。例如，下面的两个String类型，它们的内容是相同的，但是，分别指向不同的对象，用 `==` 判断，结果为 `false` ：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "HELLO".toLowerCase();
        System.out.println(s1);
        System.out.println(s2);
        if (s1 == s2) {
            System.out.println("s1 == s2");
        } else {
            System.out.println("s1 != s2");
        }
    }
}
```

▶ Run

要判断引用类型的变量内容是否相等，必须使用 `equals()` 方法：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "HELLO".toLowerCase();
        System.out.println(s1);
        System.out.println(s2);
        if (s1.equals(s2)) {
            System.out.println("s1 equals s2");
        } else {
            System.out.println("s1 not equals s2");
        }
    }
}
```

▶ Run

注意：执行语句 `s1.equals(s2)` 时，如果变量 `s1` 为 `null` ，会报 `NullPointerException` ：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        String s1 = null;
        if (s1.equals("hello")) {
            System.out.println("hello");
        }
    }
}
```

▶ Run

要避免 `NullPointerException` 错误，可以利用短路运算符 `&&` ：

```
// 条件判断
public class Main {
    public static void main(String[] args) {
        String s1 = null;
        if (s1 != null && s1.equals("hello")) {
            System.out.println("hello");
        }
    }
}
```

▶ Run

还可以把一定不是 `null` 的对象 `"hello"` 放到前面：例如： `if ("hello".equals(s)) { …`

`}` 。

练习

请用 `if … else` 编写一个程序，用于计算体质指数BMI，并打印结果。

BMI = 体重(kg)除以身高(m)的平方

BMI结果：

- 过轻：低于18.5
- 正常：18.5-25
- 过重：25-28
- 肥胖：28-32
- 非常肥胖：高于32

下载练习：BMI练习 （推荐使用IDE练习插件快速下载）

小结

`if … else` 可以做条件判断， `else` 是可选的；

不推荐省略花括号 `{}` ；

多个 `if … else` 串联要特别注意判断顺序；

要注意 `if` 的边界条件；

要注意浮点数判断相等不能直接用 `==` 运算符；

引用类型判断内容相等要使用 `equals()` ，注意避免 `NullPointerException` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





switch多重选择

除了if语句外，还有一种条件判断，是根据某个表达式的结果，分别去执行不同的分支。

例如，在游戏中，让用户选择选项：

- 单人模式
- 多人模式
- 退出游戏这时，`switch`语句就派上用场了。

`switch`语句根据`switch (表达式)`计算的结果，跳转到匹配的`case`结果，然后继续执行后续语句，直到遇到`break`结束执行。

我们看一个例子：

```
// switch
public class Main {
    public static void main(String[] args) {
        int option = 1;
        switch (option) {
        case 1:
            System.out.println("Selected 1");
            break;
        case 2:
            System.out.println("Selected 2");
            break;
        case 3:
            System.out.println("Selected 3");
            break;
        }
    }
}
```

▶ Run

修改`option`的值分别为`1`、`2`、`3`，观察执行结果。

如果`option`的值没有匹配到任何`case`，例如`option = 99`，那么，`switch`语句不会执行任何语句。这时，可以给`switch`语句加一个`default`，当没有匹配到任何`case`时，执行`default`：

```
// switch
public class Main {
    public static void main(String[] args) {
        int option = 99;
        switch (option) {
        case 1:
            System.out.println("Selected 1");
            break;
        case 2:
            System.out.println("Selected 2");
            break;
        case 3:
            System.out.println("Selected 3");
            break;
        default:
            System.out.println("Not selected");
            break;
        }
    }
}
```

▶ Run

如果把 `switch` 语句翻译成 `if` 语句，那么上述的代码相当于：

```
if (option == 1) {
    System.out.println("Selected 1");
} else if (option == 2) {
    System.out.println("Selected 2");
} else if (option == 3) {
    System.out.println("Selected 3");
} else {
    System.out.println("Not selected");
}
```

对于多个 `==` 判断的情况，使用 `switch` 结构更加清晰。

同时注意，上述“翻译”只有在 `switch` 语句中对每个 `case` 正确编写了 `break` 语句才能对应得上。

使用 `switch` 时，注意 `case` 语句并没有花括号 `{}` ，而且， `case` 语句具有“穿透性”，漏写 `break` 将导致意想不到的结果：

```
// switch
public class Main {
    public static void main(String[] args) {
        int option = 2;
        switch (option) {
        case 1:
            System.out.println("Selected 1");
        case 2:
            System.out.println("Selected 2");
        case 3:
            System.out.println("Selected 3");
        default:
            System.out.println("Not selected");
        }
    }
}
```

▶ Run

当 `option = 2` 时，将依次输出 `"Selected 2"` 、 `"Selected 3"` 、 `"Not selected"` ，原因是从匹配到 `case 2` 开始，后续语句将全部执行，直到遇到 `break` 语句。因此，任何时候都不要忘记写 `break` 。

如果有几个 `case` 语句执行的是同一组语句块，可以这么写：

```
// switch
public class Main {
    public static void main(String[] args) {
        int option = 2;
        switch (option) {
        case 1:
            System.out.println("Selected 1");
            break;
        case 2:
        case 3:
            System.out.println("Selected 2, 3");
            break;
        default:
            System.out.println("Not selected");
            break;
        }
    }
}
```

▶ Run

使用 `switch` 语句时，只要保证有 `break` ， `case` 的顺序不影响程序逻辑：

```
1.  switch (option) {
2.  case 3:
```

```
    ...
    break;
case 2:
    ...
    break;
case 1:
    ...
    break;
}
```

但是仍然建议按照自然顺序排列，便于阅读。

`switch` 语句还可以匹配字符串。字符串匹配时，是比较“内容相等”。例如：

```
// switch
public class Main {
    public static void main(String[] args) {
        String fruit = "apple";
        switch (fruit) {
        case "apple":
            System.out.println("Selected apple");
            break;
        case "pear":
            System.out.println("Selected pear");
            break;
        case "mango":
            System.out.println("Selected mango");
            break;
        default:
            System.out.println("No fruit selected");
            break;
        }
    }
}
```

▶ Run

`switch` 语句还可以使用枚举类型，枚举类型我们在后面讲解。

编译检查

使用IDE时，可以自动检查是否漏写了 `break` 语句和 `default` 语句，方法是打开IDE的编译检查。

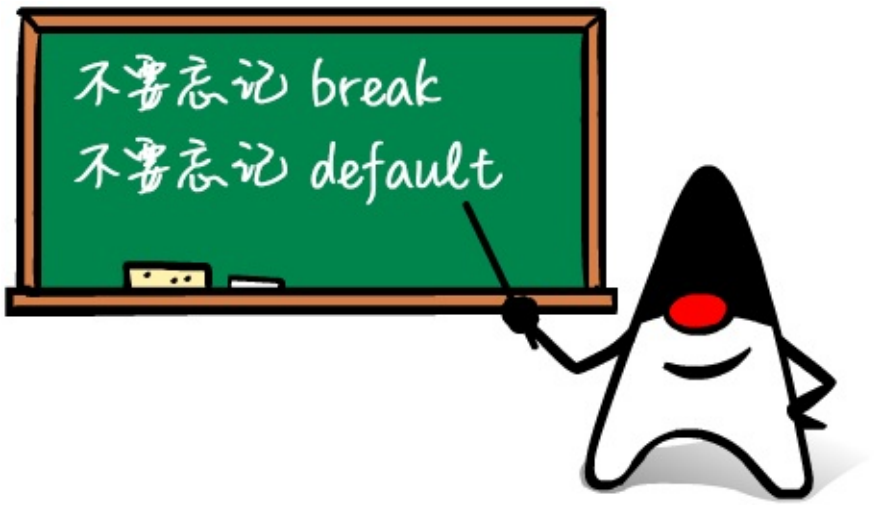
在Eclipse中，选择 `Preferences` - `Java` - `Compiler` - `Errors/Warnings` - `Potential programming problems` ，将以下检查标记为Warning：

- 'switch' is missing 'default' case
- 'switch' case fall-through

在Idea中，选择 `Preferences` - `Editor` - `Inspections` - `Java` - `Control flow issues` ，将以下检查标记为Warning:

- Fallthrough in 'switch' statement
- 'switch' statement without 'default' branch

当 `switch` 语句存在问题时，即可在IDE中获得警告提示。



switch表达式

使用 `switch` 时，如果遗漏了 `break` ，就会造成严重的逻辑错误，而且不易在源代码中发现错误。从Java 12开始， `switch` 语句升级为更简洁的表达式语法，使用类似模式匹配（Pattern Matching）的方法，保证只有一种路径会被执行，并且不需要 `break` 语句:

```
// switch
public class Main {
    public static void main(String[] args) {
        String fruit = "apple";
        switch (fruit) {
        case "apple" -> System.out.println("Selected apple");
        case "pear" -> System.out.println("Selected pear");
        case "mango" -> {
            System.out.println("Selected mango");
            System.out.println("Good choice!");
        }
        default -> System.out.println("No fruit selected");
        }
    }
}
```

▶ Run

注意新语法使用 `->` ，如果有多条语句，需要用 `{}` 括起来。不要写 `break` 语句，因为新语法只会执行匹配的语句，没有穿透效应。

很多时候，我们还可能用 `switch` 语句给某个变量赋值。例如：

```
int opt;
switch (fruit) {
case "apple":
    opt = 1;
    break;
case "pear":
case "mango":
    opt = 2;
    break;
default:
    opt = 0;
    break;
}
```

使用新的 `switch` 语法，不但不需要 `break` ，还可以直接返回值。把上面的代码改写如下：

```
// switch
public class Main {
    public static void main(String[] args) {
        String fruit = "apple";
        int opt = switch (fruit) {
            case "apple" -> 1;
            case "pear", "mango" -> 2;
            default -> 0;
        }; // 注意赋值语句要以;结束
        System.out.println("opt = " + opt);
    }
}
```

▶ Run

这样可以获得更简洁的代码。

yield

大多数时候，在 `switch` 表达式内部，我们会返回简单的值。

但是，如果需要复杂的语句，我们也可以写很多语句，放到 `{…}` 里，然后，用 `yield` 返回一个值作为 `switch` 语句的返回值：

```
// yield
public class Main {
    public static void main(String[] args) {
        String fruit = "orange";
        int opt = switch (fruit) {
            case "apple" -> 1;
            case "pear", "mango" -> 2;
            default -> {
                int code = fruit.hashCode();
                yield code; // switch语句返回值
            }
        };
        System.out.println("opt = " + opt);
    }
}
```

▶ Run

由于 `switch` 表达式是作为Java 13的预览特性（Preview Language Features）实现的，编译的时候，我们还需要给编译器加上参数：

```
1. javac --source 13 --enable-preview Main.java
```

这样才能正常编译。

练习

使用 `switch` 实现一个简单的石头、剪子、布游戏。

下载练习：switch练习 （推荐使用IDE练习插件快速下载）

小结

`switch` 语句可以做多重选择，然后执行匹配的 `case` 语句后续代码；

`switch` 的计算结果必须是整型、字符串或枚举类型；

注意千万不要漏写 `break` ，建议打开 `fall-through` 警告；

总是写上 `default` ，建议打开 `missing default` 警告；

从Java 13开始， `switch` 语句升级为表达式，不再需要 `break` ，并且允许使用 `yield` 返回值。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





while循环

循环语句就是让计算机根据条件做循环计算，在条件满足时继续循环，条件不满足时退出循环。

例如，计算从1到100的和：

```
1. 1 + 2 + 3 + 4 + … + 100 = ?
```

除了用数列公式外，完全可以让计算机做100次循环累加。因为计算机的特点是计算速度非常快，我们让计算机循环一亿次也用不到1秒，所以很多计算的任务，人去算是算不了的，但是计算机算，使用循环这种简单粗暴的方法就可以快速得到结果。

我们先看Java提供的 `while` 条件循环。它的基本用法是：

```
1. while (条件表达式) {
2.     循环语句
3. }
4. // 继续执行后续代码
```

`while` 循环在每次循环开始前，首先判断条件是否成立。如果计算结果为 `true` ，就把循环体内的语句执行一遍，如果计算结果为 `false` ，那就直接跳到 `while` 循环的末尾，继续往下执行。

我们用while循环来累加1到100，可以这么写：

```
// while
public class Main {
    public static void main(String[] args) {
        int sum = 0; // 累加的和，初始化为0
        int n = 1;
        while (n <= 100) { // 循环条件是n <= 100
            sum = sum + n; // 把n累加到sum中
            n ++; // n自身加1
        }
        System.out.println(sum); // 5050
    }
}
```

▶ Run

注意到 `while` 循环是先判断循环条件，再循环，因此，有可能一次循环都不做。

对于循环条件判断，以及自增变量的处理，要特别注意边界条件。思考一下下面的代码为何没有获得正确结果：

```
// while
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        int n = 0;
        while (n <= 100) {
            n ++;
            sum = sum + n;
        }
        System.out.println(sum);
    }
}
```

▶ Run

如果循环条件永远满足，那这个循环就变成了死循环。死循环将导致100%的CPU占用，用户会感觉电脑运行缓慢，所以要避免编写死循环代码。

如果循环条件的逻辑写得有问题，也会造成意料之外的结果：

```
// while
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        int n = 1;
        while (n > 0) {
            sum = sum + n;
            n ++;
        }
        System.out.println(n); // -2147483648
        System.out.println(sum);
    }
}
```

▶ Run

表面上看，上面的 `while` 循环是一个死循环，但是，Java的 `int` 类型有最大值，达到最大值后，再加1会变成负数，结果，意外退出了 `while` 循环。

练习

使用 `while` 计算从 `m` 到 `n` 的和：

```
// while
public class Main {
        public static void main(String[] args) {
                int sum = 0;
                int m = 20;
                int n = 100;
                // 使用while计算M+...+N:
                while (false) {
                }
                System.out.println(sum);
        }
}
```

▶ Run

下载练习：while练习 （推荐使用IDE练习插件快速下载）

小结

`while` 循环先判断循环条件是否满足，再执行循环语句；

`while` 循环可能一次都不执行；

编写循环时要注意循环条件，并避免死循环。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





do while循环

在Java中，`while`循环是先判断循环条件，再执行循环。而另一种`do while`循环则是先执行循环，再判断条件，条件满足时继续循环，条件不满足时退出。它的用法是：

```
do {
    执行循环语句
} while (条件表达式);
```

可见，`do while`循环会至少循环一次。

我们把对1到100的求和用`do while`循环改写一下：

```
// do-while
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        int n = 1;
        do {
            sum = sum + n;
            n ++;
        } while (n <= 100);
        System.out.println(sum);
    }
}
```

▶ Run

使用`do while`循环时，同样要注意循环条件的判断。

练习

使用`do while`循环计算从`m`到`n`的和。

```
// do while
public class Main {
        public static void main(String[] args) {
                int sum = 0;
        int m = 20;
                int n = 100;
                // 使用do while计算M+...+N:
                do {
                } while (false);
                System.out.println(sum);
        }
}
```

▶ Run

下载练习：do while练习 （推荐使用IDE练习插件快速下载）

小结

`do while` 循环先执行循环，再判断条件；

`do while` 循环会至少执行一次。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





for循环

除了 `while` 和 `do while` 循环，Java使用最广泛的是 `for` 循环。

`for` 循环的功能非常强大，它使用计数器实现循环。 `for` 循环会先初始化计数器，然后，在每次循环前检测循环条件，在每次循环后更新计数器。计数器变量通常命名为 `i` 。

我们把1到100求和用 `for` 循环改写一下：

```
// for
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        for (int i=1; i<=100; i++) {
            sum = sum + i;
        }
        System.out.println(sum);
    }
}
```

▶ Run

在 `for` 循环执行前，会先执行初始化语句 `int i=1` ，它定义了计数器变量 `i` 并赋初始值为 `1` ，然后，循环前先检查循环条件 `i<=100` ，循环后自动执行 `i++` ，因此，和 `while` 循环相比， `for` 循环把更新计数器的代码统一放到了一起。在 `for` 循环的循环体内部，不需要去更新变量 `i` 。

因此， `for` 循环的用法是：

```
for (初始条件; 循环检测条件; 循环后更新计数器) {
    // 执行语句
}
```

如果我们要对一个整型数组的所有元素求和，可以用 `for` 循环实现：

```
// for
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        int sum = 0;
        for (int i=0; i<ns.length; i++) {
            System.out.println("i = " + i + ", ns[i] = " + ns[i]);
            sum = sum + ns[i];
        }
        System.out.println("sum = " + sum);
    }
}
```

▶ Run

上面代码的循环条件是 `i<ns.length` 。因为 `ns` 数组的长度是 `5` ，因此，当循环 `5` 次后， `i` 的值被更新为 `5` ，就不满足循环条件，因此 `for` 循环结束。

思考：如果把循环条件改为i<=ns.length，会出现什么问题？

注意 `for` 循环的初始化计数器总是会被执行，并且 `for` 循环也可能循环0次。

使用 `for` 循环时，千万不要在循环体内修改计数器！在循环体中修改计数器常常导致莫名其妙的逻辑错误。对于下面的代码：

```
// for
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        for (int i=0; i<ns.length; i++) {
            System.out.println(ns[i]);
            i = i + 1;
        }
    }
}
```

▶ Run

虽然不会报错，但是，数组元素只打印了一半，原因是循环内部的 `i = i + 1` 导致了计数器变量每次循环实际上加了 `2` （因为 `for` 循环还会自动执行 `i++` ）。因此，在 `for` 循环中，不要修改计数器的值。计数器的初始化、判断条件、每次循环后的更新条件统一放到 `for()` 语句中可以一目了然。

如果希望只访问索引为奇数的数组元素，应该把 `for` 循环改写为：

```
1.  int[] ns = { 1, 4, 9, 16, 25 };
```

```
for (int i=0; i<ns.length; i=i+2) {
    System.out.println(ns[i]);
}
```

通过更新计数器的语句 `i=i+2` 就达到了这个效果，从而避免了在循环体内去修改变量 `i` 。

使用 `for` 循环时，计数器变量 `i` 要尽量定义在 `for` 循环中：

```
int[] ns = { 1, 4, 9, 16, 25 };
for (int i=0; i<ns.length; i++) {
    System.out.println(ns[i]);
}
// 无法访问i
int n = i; // compile error!
```

如果变量 `i` 定义在 `for` 循环外：

```
int[] ns = { 1, 4, 9, 16, 25 };
int i;
for (i=0; i<ns.length; i++) {
    System.out.println(ns[i]);
}
// 仍然可以使用i
int n = i;
```

那么，退出 `for` 循环后，变量 `i` 仍然可以被访问，这就破坏了变量应该把访问范围缩到最小的原则。

灵活使用for循环

`for` 循环还可以缺少初始化语句、循环条件和每次循环更新语句，例如：

```
// 不设置结束条件:
for (int i=0; ; i++) {
    ...
}
```

```
// 不设置结束条件和更新语句:
for (int i=0; ;) {
    ...
}
```

```
// 什么都不设置:
for (;;) {
    ...
}
```

通常不推荐这样写，但是，某些情况下，是可以省略 `for` 循环的某些语句的。

for each循环

`for` 循环经常用来遍历数组，因为通过计数器可以根据索引来访问数组的每个元素：

```
int[] ns = { 1, 4, 9, 16, 25 };
for (int i=0; i<ns.length; i++) {
    System.out.println(ns[i]);
}
```

但是，很多时候，我们实际上真正想要访问的是数组每个元素的值。Java还提供了另一种 `for each` 循环，它可以更简单地遍历数组：

```
// for each
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        for (int n : ns) {
            System.out.println(n);
        }
    }
}
```

▶ Run

和 `for` 循环相比， `for each` 循环的变量n不再是计数器，而是直接对应到数组的每个元素。 `for each` 循环的写法也更简洁。但是， `for each` 循环无法指定遍历顺序，也无法获取数组的索引。

除了数组外， `for each` 循环能够遍历所有“可迭代”的数据类型，包括后面会介绍的 `List` 、 `Map` 等。

练习1

给定一个数组，请用 `for` 循环倒序输出每一个元素：

```
// for
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        for (int i=?; ?; ?) {
            System.out.println(ns[i]);
        }
    }
}
```

▶ Run

练习2

利用 `for each` 循环对数组每个元素求和：

```
// for each
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        int sum = 0;
        for (???) {
            // TODO
        }
        System.out.println(sum); // 55
    }
}
```

▶ Run

练习3

圆周率π可以使用公式计算：

$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - ...$4π=1−31+51−71+91−…

请利用 `for` 循环计算π：

```
// for
public class Main {
    public static void main(String[] args) {
        double pi = 0;
        for (???) {
            // TODO
        }
        System.out.println(pi);
    }
}
```

▶ Run

下载练习：for循环计算π练习 （推荐使用IDE练习插件快速下载）

小结

`for` 循环通过计数器可以实现复杂循环；

`for each` 循环可以直接遍历数组的每个元素；

最佳实践：计数器变量定义在 `for` 循环内部，循环体内部不修改计数器；

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





break和continue

无论是 `while` 循环还是 `for` 循环，有两个特别的语句可以使用，就是 `break` 语句和 `continue` 语句。

break

在循环过程中，可以使用 `break` 语句跳出当前循环。我们来看一个例子：

```
// break
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        for (int i=1; ; i++) {
            sum = sum + i;
            if (i == 100) {
                break;
            }
        }
        System.out.println(sum);
    }
}
```

▶ Run

使用 `for` 循环计算从1到100时，我们并没有在 `for()` 中设置循环退出的检测条件。但是，在循环内部，我们用 `if` 判断，如果 `i==100` ，就通过 `break` 退出循环。

因此， `break` 语句通常都是配合 `if` 语句使用。要特别注意， `break` 语句总是跳出自己所在的那一层循环。例如：

```
// break
public class Main {
    public static void main(String[] args) {
        for (int i=1; i<=10; i++) {
            System.out.println("i = " + i);
            for (int j=1; j<=10; j++) {
                System.out.println("j = " + j);
                if (j >= i) {
                    break;
                }
            }
            // break跳到这里
            System.out.println("breaked");
        }
    }
}
```

▶ Run

上面的代码是两个 `for` 循环嵌套。因为 `break` 语句位于内层的 `for` 循环，因此，它会跳出内层 `for` 循环，但不会跳出外层 `for` 循环。

continue

`break` 会跳出当前循环，也就是整个循环都不会执行了。而 `continue` 则是提前结束本次循环，直接继续执行下次循环。我们看一个例子：

```
// continue
public class Main {
    public static void main(String[] args) {
        int sum = 0;
        for (int i=1; i<=10; i++) {
            System.out.println("begin i = " + i);
            if (i % 2 == 0) {
                continue; // continue语句会结束本次循环
            }
            sum = sum + i;
            System.out.println("end i = " + i);
        }
        System.out.println(sum); // 25
    }
}
```

▶ Run

注意观察 `continue` 语句的效果。当 `i` 为奇数时，完整地执行了整个循环，因此，会打印 `begin i=1` 和 `end i=1` 。在i为偶数时， `continue` 语句会提前结束本次循环，因此，会打印 `begin i=2` 但不会打印 `end i = 2` 。

在多层嵌套的循环中，`continue`语句同样是结束本次自己所在的循环。

小结

`break`语句可以跳出当前循环；

`break`语句通常配合`if`，在满足条件时提前结束整个循环；

`break`语句总是跳出最近的一层循环；

`continue`语句可以提前结束本次循环；

`continue`语句通常配合`if`，在满足条件时提前结束本次循环。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





数组操作

本节我们将讲解对数组的操作，包括：

- 遍历；
- 排序。

以及多维数组的概念。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





遍历数组

我们在Java程序基础里介绍了数组这种数据类型。有了数组，我们还需要来操作它。而数组最常见的一个操作就是遍历。

通过 `for` 循环就可以遍历数组。因为数组的每个元素都可以通过索引来访问，因此，使用标准的 `for` 循环可以完成一个数组的遍历：

```
// 遍历数组
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        for (int i=0; i<ns.length; i++) {
            int n = ns[i];
            System.out.println(n);
        }
    }
}
```

▶ Run

为了实现 `for` 循环遍历，初始条件为 `i=0` ，因为索引总是从 `0` 开始，继续循环的条件为 `i<ns.length` ，因为当 `i=ns.length` 时， `i` 已经超出了索引范围（索引范围是 `0` ~ `ns.length-1` ），每次循环后， `i++` 。

第二种方式是使用 `for each` 循环，直接迭代数组的每个元素：

```
// 遍历数组
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        for (int n : ns) {
            System.out.println(n);
        }
    }
}
```

▶ Run

注意：在 `for (int n : ns)` 循环中，变量 `n` 直接拿到 `ns` 数组的元素，而不是索引。

显然 `for each` 循环更加简洁。但是， `for each` 循环无法拿到数组的索引，因此，到底用哪一种 `for` 循环，取决于我们的需要。

打印数组内容

直接打印数组变量，得到的是数组在JVM中的引用地址：

```
int[] ns = { 1, 1, 2, 3, 5, 8 };
System.out.println(ns); // 类似 [I@7852e922
```

这并没有什么意义，因为我们希望打印的数组的元素内容。因此，使用 `for each` 循环来打印它：

```
int[] ns = { 1, 1, 2, 3, 5, 8 };
for (int n : ns) {
    System.out.print(n + ", ");
}
```

使用 `for each` 循环打印也很麻烦。幸好Java标准库提供了 `Arrays.toString()` ，可以快速打印数组内容：

```
// 遍历数组
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 1, 2, 3, 5, 8 };
        System.out.println(Arrays.toString(ns));
    }
}
```

▶ Run

练习

请按倒序遍历数组并打印每个元素：

```
public class Main {
    public static void main(String[] args) {
        int[] ns = { 1, 4, 9, 16, 25 };
        // 倒序打印数组元素:
        for (???) {
            System.out.println(???);
        }
    }
}
```

▶ Run

下载练习：倒序遍历数组练习 （推荐使用IDE练习插件快速下载）

小结

遍历数组可以使用 `for` 循环， `for` 循环可以访问数组索引， `for each` 循环直接迭代每个数组元素，但无法获取索引；

使用 `Arrays.toString()` 可以快速获取数组内容。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





数组排序

对数组进行排序是程序中非常基本的需求。常用的排序算法有冒泡排序、插入排序和快速排序等。

我们来看一下如何使用冒泡排序算法对一个整型数组从小到大进行排序：

```
// 冒泡排序
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] ns = { 28, 12, 89, 73, 65, 18, 96, 50, 8, 36 };
        // 排序前:
        System.out.println(Arrays.toString(ns));
        for (int i = 0; i < ns.length - 1; i++) {
            for (int j = 0; j < ns.length - i - 1; j++) {
                if (ns[j] > ns[j+1]) {
                    // 交换ns[j]和ns[j+1]:
                    int tmp = ns[j];
                    ns[j] = ns[j+1];
                    ns[j+1] = tmp;
                }
            }
        }
        // 排序后:
        System.out.println(Arrays.toString(ns));
    }
}
```

▶ Run

冒泡排序的特点是，每一轮循环后，最大的一个数被交换到末尾，因此，下一轮循环就可以“刨除”最后的数，每一轮循环都比上一轮循环的结束位置靠前一位。

另外，注意到交换两个变量的值必须借助一个临时变量。像这么写是错误的：

```
int x = 1;
int y = 2;

x = y; // x现在是2
y = x; // y现在还是2
```

正确的写法是：

```
int x = 1;
int y = 2;

```

```
int t = x; // 把x的值保存在临时变量t中, t现在是1
x = y; // x现在是2
y = t; // y现在是t的值1
```

实际上，Java的标准库已经内置了排序功能，我们只需要调用JDK提供的 `Arrays.sort()` 就可以排序：

```
// 排序
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] ns = { 28, 12, 89, 73, 65, 18, 96, 50, 8, 36 };
        Arrays.sort(ns);
        System.out.println(Arrays.toString(ns));
    }
}
```

▶ Run

必须注意，对数组排序实际上修改了数组本身。例如，排序前的数组是：

```
int[] ns = { 9, 3, 6, 5 };
```

在内存中，这个整型数组表示如下：



当我们调用 `Arrays.sort(ns);` 后，这个整型数组在内存中变为：

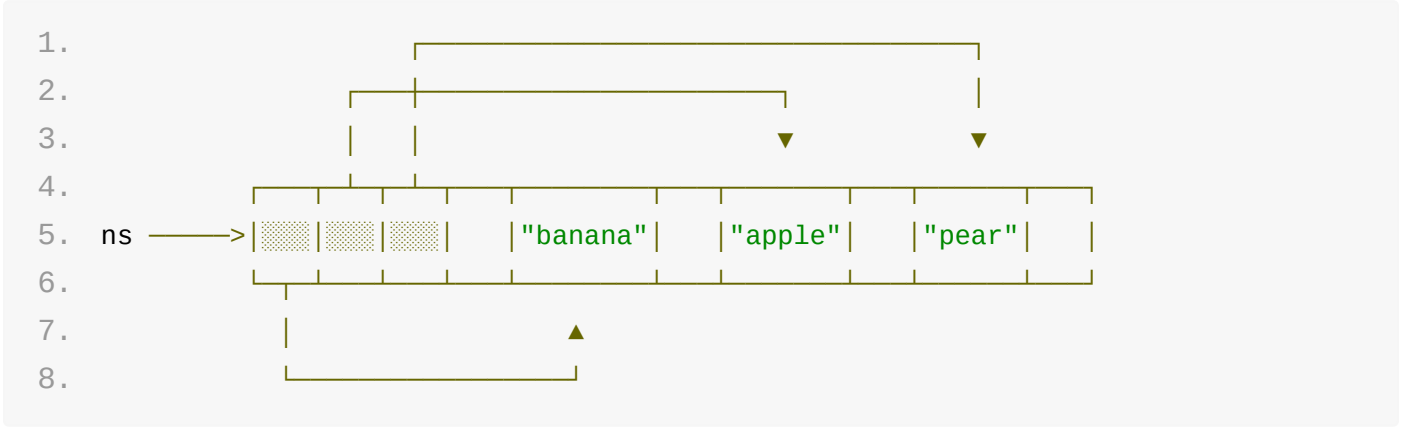


即变量 `ns` 指向的数组内容已经被改变了。

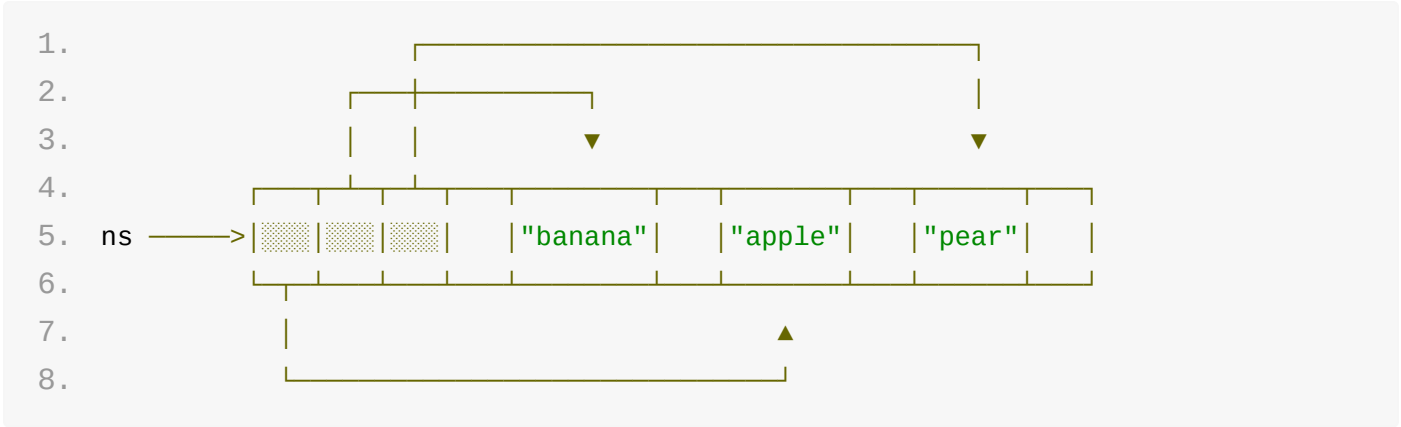
如果对一个字符串数组进行排序，例如：

```
String[] ns = { "banana", "apple", "pear" };
```

排序前，这个数组在内存中表示如下：



调用 `Arrays.sort(ns);` 排序后，这个数组在内存中表示如下：



原来的3个字符串在内存中均没有任何变化，但是 `ns` 数组的每个元素指向变化了。

练习

请思考如何实现对数组进行降序排序：

```
// 降序排序
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] ns = { 28, 12, 89, 73, 65, 18, 96, 50, 8, 36 };
        // 排序前:
        System.out.println(Arrays.toString(ns));
        // TODO:
        // 排序后:
        System.out.println(Arrays.toString(ns));
        if (Arrays.toString(ns).equals("[96, 89, 73, 65, 50, 36, 28, 18, 12, 8]")) {
            System.out.println("测试成功");
        } else {
            System.out.println("测试失败");
        }
    }
}
```

▶ Run

下载练习：降序排序练习 （推荐使用IDE练习插件快速下载）

小结

常用的排序算法有冒泡排序、插入排序和快速排序等；

冒泡排序使用两层 `for` 循环实现排序；

交换两个变量的值需要借助一个临时变量。

可以直接使用Java标准库提供的 `Arrays.sort()` 进行排序；

对数组排序会直接修改数组本身。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





多维数组

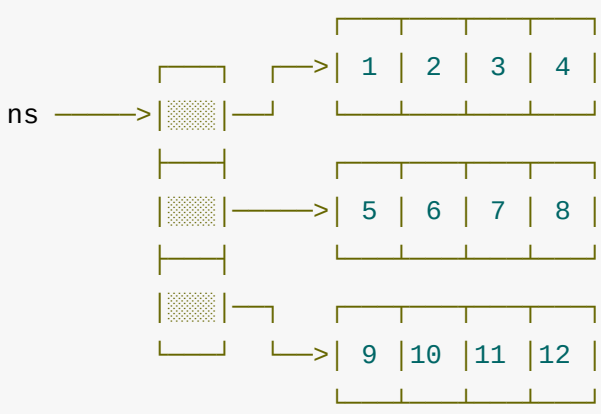
二维数组

二维数组就是数组的数组。定义一个二维数组如下：

```
// 二维数组
public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        System.out.println(ns.length); // 3
    }
}
```

▶ Run

因为 `ns` 包含3个数组，因此， `ns.length` 为 `3` 。实际上 `ns` 在内存中的结构如下：

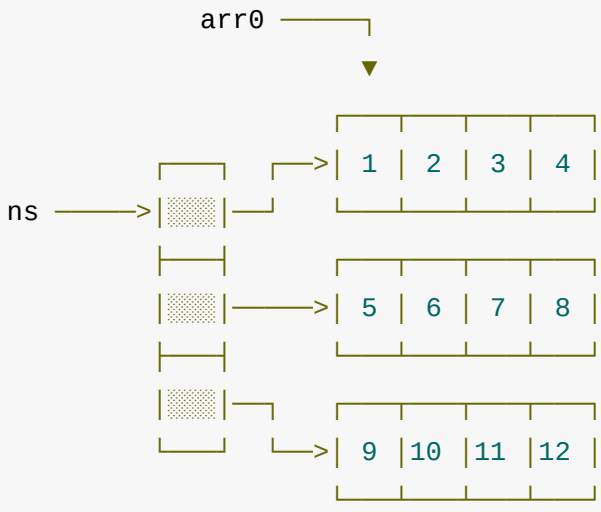


如果我们定义一个普通数组 `arr0` ，然后把 `ns[0]` 赋值给它：

```
// 二维数组
public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        int[] arr0 = ns[0];
        System.out.println(arr0.length); // 4
    }
}
```

▶ Run

实际上 `arr0` 就获取了 `ns` 数组的第0个元素。因为 `ns` 数组的每个元素也是一个数组，因此， `arr0` 指向的数组就是 `{ 1, 2, 3, 4 }` 。在内存中，结构如下：



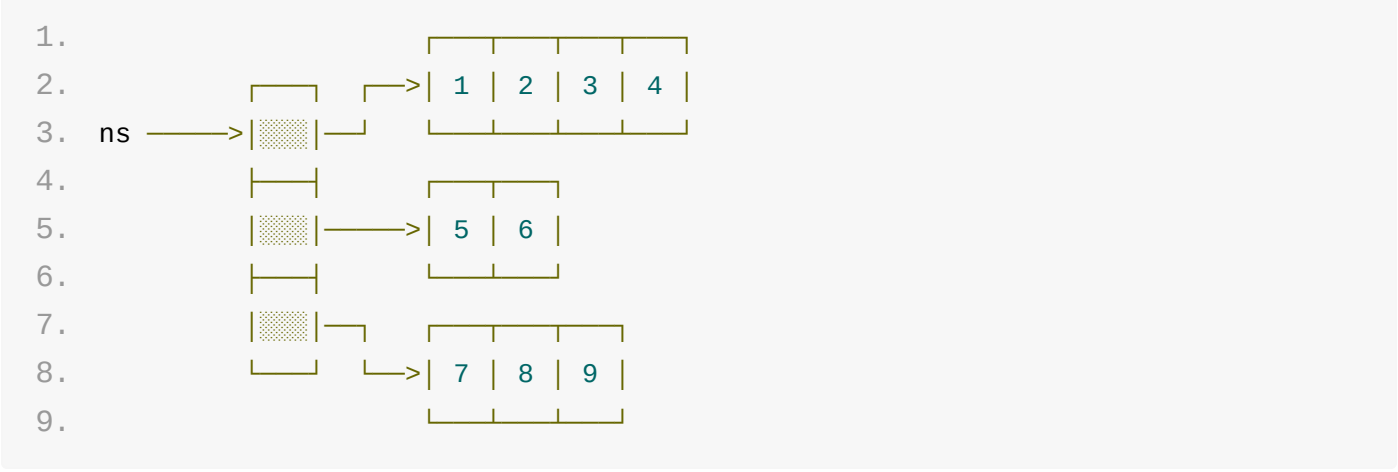
访问二维数组的某个元素需要使用 `array[row][col]` ，例如：

```
System.out.println(ns[1][2]); // 7
```

二维数组的每个数组元素的长度并不要求相同，例如，可以这么定义 `ns` 数组：

```
int[][] ns = {
    { 1, 2, 3, 4 },
    { 5, 6 },
    { 7, 8, 9 }
};
```

这个二维数组在内存中的结构如下：



要打印一个二维数组，可以使用两层嵌套的for循环：

```
for (int[] arr : ns) {
    for (int n : arr) {
        System.out.print(n);
        System.out.print(', ');
    }
    System.out.println();
}
```

或者使用Java标准库的 `Arrays.deepToString()` ：

```
// 二维数组
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[][] ns = {
            { 1, 2, 3, 4 },
            { 5, 6, 7, 8 },
            { 9, 10, 11, 12 }
        };
        System.out.println(Arrays.deepToString(ns));
    }
}
```

▶ Run

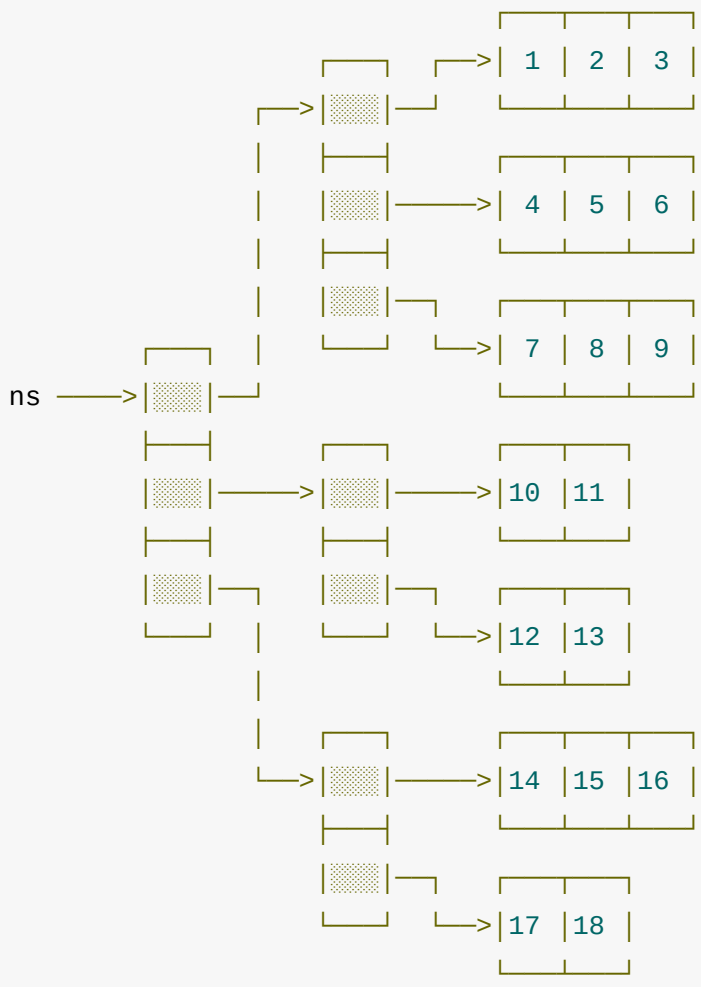
三维数组

三维数组就是二维数组的数组。可以这么定义一个三维数组：

```
int[][][] ns = {
    {
```

```
        {1, 2, 3},
        {4, 5, 6},
        {7, 8, 9}
    },
    {
        {10, 11},
        {12, 13}
    },
    {
        {14, 15, 16},
        {17, 18}
    }
};
```

它在内存中的结构如下：



如果我们要访问三维数组的某个元素，例如， `ns[2][0][1]` ，只需要顺着定位找到对应的最终元素 `15` 即可。

理论上，我们可以定义任意的N维数组。但在实际应用中，除了二维数组在某些时候还能用得上，更高

维度的数组很少使用。

练习

使用二维数组可以表示一组学生的各科成绩，请计算所有学生的平均分：

```
public class Main {
    public static void main(String[] args) {
        // 用二维数组表示的学生成绩：
        int[][] scores = {
                { 82, 90, 91 },
                { 68, 72, 64 },
                { 95, 91, 89 },
                { 67, 52, 60 },
                { 79, 81, 85 },
        };
        // TODO:
        double average = 0;
        System.out.println(average);



        if (Math.abs(average - 77.733333) < 0.000001) {
            System.out.println("测试成功");
        } else {
            System.out.println("测试失败");
        }
    }
}
```

▶ Run

下载练习：计算平均分 （推荐使用IDE练习插件快速下载）

小结

二维数组就是数组的数组，三维数组就是二维数组的数组；

多维数组的每个数组元素长度都不要求相同；

打印多维数组可以使用 `Arrays.deepToString()` ；

最常见的多维数组是二维数组，访问二维数组的一个元素使用 `array[row][col]` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





命令行参数

Java程序的入口是 `main` 方法，而 `main` 方法可以接受一个命令行参数，它是一个 `String[]` 数组。

这个命令行参数由JVM接收用户输入并传给 `main` 方法：

```
public class Main {
    public static void main(String[] args) {
        for (String arg : args) {
            System.out.println(arg);
        }
    }
}
```

我们可以利用接收到的命令行参数，根据不同的参数执行不同的代码。例如，实现一个 `-version` 参数，打印程序版本号：

```
public class Main {
    public static void main(String[] args) {
        for (String arg : args) {
            if ("-version".equals(arg)) {
                System.out.println("v 1.0");
                break;
            }
        }
    }
}
```

上面这个程序必须在命令行执行，我们先编译它：

```
$ javac Main.java
```

然后，执行的时候，给它传递一个 `-version` 参数：

```
$ java Main -version
v 1.0
```

这样，程序就可以根据传入的命令行参数，作出不同的响应。

小结

命令行参数类型是 `String[]` 数组;

命令行参数由JVM接收用户输入并传给 `main` 方法;

如何解析命令行参数需要由程序自己实现。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





面向对象编程

Java是一种面向对象的编程语言。面向对象编程，英文是Object-Oriented Programming，简称OOP。

那什么是面向对象编程？

和面向对象编程不同的，是面向过程编程。面向过程编程，是把模型分解成一步一步的过程。比如，老板告诉你，要编写一个TODO任务，必须按照以下步骤一步一步来：

- 读取文件；
- 编写TODO；
- 保存文件。



而面向对象编程，顾名思义，你得首先有个对象：



有了对象后，就可以和对象进行互动：

```
1. GirlFriend gf = new GirlFriend();
2. gf.name = "Alice";
3. gf.send("flowers");
```

因此，面向对象编程，是一种通过对象的方式，把现实世界映射到计算机模型的一种编程方法。

在本章中，我们将讨论：

面向对象的基本概念，包括：

- 类
- 实例
- 方法

面向对象的实现方式，包括：

- 继承
- 多态

Java语言本身提供的机制，包括：

- package
- classpath

- jar

以及Java标准库提供的核心类，包括：

- 字符串
- 包装类型
- JavaBean
- 枚举
- 常用工具类

通过本章的学习，完全可以理解并掌握面向对象编程的基本思想。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





面向对象基础

面向对象编程，是一种通过对象的方式，把现实世界映射到计算机模型的一种编程方法。

现实世界中，我们定义了“人”这种抽象概念，而具体的人则是“小明”、“小红”、“小军”等一个个具体的人。所以，“人”可以定义为一个类（class），而具体的人则是实例（instance）：

现实世界	计算机模型	Java代码
人	类 / class	class Person { }
小明	实例 / ming	Person ming = new Person()
小红	实例 / hong	Person hong = new Person()
小军	实例 / jun	Person jun = new Person()

同样的，“书”也是一种抽象的概念，所以它是类，而《Java核心技术》、《Java编程思想》、《Java学习笔记》则是实例：

现实世界	计算机模型	Java代码
书	类 / class	class Book { }
Java核心技术	实例 / book1	Book book1 = new Book()
Java编程思想	实例 / book2	Book book2 = new Book()
Java学习笔记	实例 / book3	Book book3 = new Book()

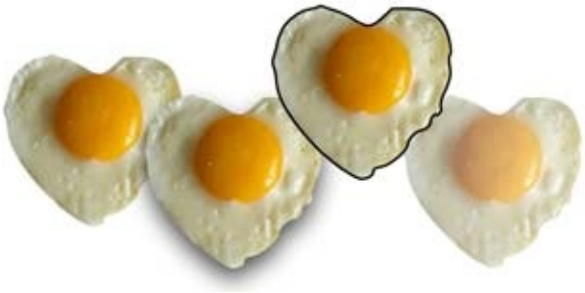
class和instance

所以，只要理解了class和instance的概念，基本上就明白了什么是面向对象编程。

class是一种对象模版，它定义了如何创建实例，因此，class本身就是一种数据类型：



而instance是对象实例，instance是根据class创建的实例，可以创建多个instance，每个instance类型相同，但各自属性可能不相同：



定义class

在Java中，创建一个类，例如，给这个类命名为 `Person` ，就是定义一个 `class` ：

```
class Person {
    public String name;
    public int age;
}
```

一个 `class` 可以包含多个字段（ `field` ），字段用来描述一个类的特征。上面的 `Person` 类，我们定义了两个字段，一个是 `String` 类型的字段，命名为 `name` ，一个是 `int` 类型的字段，命名为 `age` 。因此，通过 `class` ，把一组数据汇集到一个对象上，实现了数据封装。

`public` 是用来修饰字段的，它表示这个字段可以被外部访问。

我们再看另一个 `Book` 类的定义：

```
class Book {
    public String name;
    public String author;
    public String isbn;
    public double price;
}
```

请指出 `Book` 类的各个字段。

创建实例

定义了class，只是定义了对象模版，而要根据对象模版创建出真正的对象实例，必须用new操作符。

new操作符可以创建一个实例，然后，我们需要定义一个引用类型的变量来指向这个实例：

```
Person ming = new Person();
```

上述代码创建了一个Person类型的实例，并通过变量 `ming` 指向它。

注意区分 `Person ming` 是定义 `Person` 类型的变量 `ming` ，而 `new Person()` 是创建 `Person` 实例。

有了指向这个实例的变量，我们就可以通过这个变量来操作实例。访问实例变量可以用 `变量.字段` ，例如：

```
ming.name = "Xiao Ming"; // 对字段name赋值
ming.age = 12; // 对字段age赋值
System.out.println(ming.name); // 访问字段name

Person hong = new Person();
hong.name = "Xiao Hong";
hong.age = 15;
```

上述两个变量分别指向两个不同的实例，它们在内存中的结构如下：

```
                ┌──────────────────┐
ming ──────────>│Person instance   │
                ├──────────────────┤
                │name = "Xiao Ming"│
                │age = 12          │
                └──────────────────┘

                ┌──────────────────┐
hong ──────────>│Person instance   │
                ├──────────────────┤
                │name = "Xiao Hong"│
                │age = 15          │
                └──────────────────┘
```

两个 `instance` 拥有 `class` 定义的 `name` 和 `age` 字段，且各自都有一份独立的数据，互不干扰。

练习

请定义一个City类，该class具有如下字段：

- name: 名称，String类型
- latitude: 纬度，double类型
- longitude: 经度，double类型

实例化几个City并赋值，然后打印。

```
// City.java
public class Main {
    public static void main(String[] args) {
        City bj = new City();
        bj.name = "Beijing";
        bj.latitude = 39.903;
        bj.longitude = 116.401;
        System.out.println(bj.name);
        System.out.println("location: " + bj.latitude + ", " + bj.longitude);
    }
}

class City {
    ???
}
```

▶ Run

小结

在OOP中，`class` 和 `instance` 是“模版”和“实例”的关系；

定义 `class` 就是定义了一种数据类型，对应的 `instance` 是这种数据类型的实例；

`class` 定义的 `field` ，在每个 `instance` 都会拥有各自的 `field` ，且互不干扰；

通过 `new` 操作符创建新的 `instance` ，然后用变量指向它，即可通过变量来引用这个 `instance` ；

访问实例字段的方法是 `变量名.字段名` ；

指向 `instance` 的变量都是引用变量。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





方法

一个 `class` 可以包含多个 `field` ，例如，我们给 `Person` 类就定义了两个 `field` ：

```
class Person {
    public String name;
    public int age;
}
```

但是，直接把 `field` 用 `public` 暴露给外部可能会破坏封装性。比如，代码可以这样写：

```
Person ming = new Person();
ming.name = "Xiao Ming";
ming.age = -99; // age设置为负数
```

显然，直接操作 `field` ，容易造成逻辑混乱。为了避免外部代码直接去访问 `field` ，我们可以用 `private` 修饰 `field` ，拒绝外部访问：

```
class Person {
    private String name;
    private int age;
}
```

试试 `private` 修饰的 `field` 有什么效果：

```
// private field
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        ming.name = "Xiao Ming"; // 对字段name赋值
        ming.age = 12; // 对字段age赋值
    }
}

class Person {
    private String name;
    private int age;
}
```

▶ Run

是不是编译报错？把访问 `field` 的赋值语句去了就可以正常编译了。



把 `field` 从 `public` 改成 `private`，外部代码不能访问这些 `field`，那我们定义这些 `field` 有什么用？怎么才能给它赋值？怎么才能读取它的值？

所以我们需要使用方法（`method`）来让外部代码可以间接修改 `field`：

```
// private field
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        ming.setName("Xiao Ming"); // 设置name
        ming.setAge(12); // 设置age
        System.out.println(ming.getName() + ", " + ming.getAge());
    }
}

class Person {
    private String name;
    private int age;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        if (age < 0 || age > 100) {
            throw new IllegalArgumentException("invalid age value");
        }
        this.age = age;
    }
}
```

▶ Run

虽然外部代码不能直接修改 `private` 字段，但是，外部代码可以调用方法 `setName()` 和 `setAge()` 来间接修改 `private` 字段。在方法内部，我们就有机会检查参数对不对。比如， `setAge()` 就会检查传入的参数，参数超出了范围，直接报错。这样，外部代码就没有任何机会把 `age` 设置成不合理的值。

对 `setName()` 方法同样可以做检查，例如，不允许传入 `null` 和空字符串：

```
public void setName(String name) {
    if (name == null || name.isBlank()) {
        throw new IllegalArgumentException("invalid name");
    }
    this.name = name.strip(); // 去掉首尾空格
}
```

同样，外部代码不能直接读取 `private` 字段，但可以通过 `getName()` 和 `getAge()` 间接获取 `private` 字段的值。

所以，一个类通过定义方法，就可以给外部代码暴露一些操作的接口，同时，内部自己保证逻辑一致性。

调用方法的语法是 `实例变量.方法名(参数);` 。一个方法调用就是一个语句，所以不要忘了在末尾加 `;` 。例如： `ming.setName("Xiao Ming");` 。

定义方法

从上面的代码可以看出，定义方法的语法是：

```
修饰符 方法返回类型 方法名(方法参数列表) {
    若干方法语句;
    return 方法返回值;
}
```

方法返回值通过 `return` 语句实现，如果没有返回值，返回类型设置为 `void` ，可以省略 `return` 。

private方法

有 `public` 方法，自然就有 `private` 方法。和 `private` 字段一样， `private` 方法不允许外部调用，那我们定义 `private` 方法有什么用？

定义 `private` 方法的理由是内部方法是可以调用 `private` 方法的。例如：

```
// private method
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        ming.setBirth(2008);
        System.out.println(ming.getAge());
    }
}

class Person {
    private String name;
    private int birth;

    public void setBirth(int birth) {
        this.birth = birth;
    }

    public int getAge() {
        return calcAge(2019); // 调用private方法
    }

    // private方法:
    private int calcAge(int currentYear) {
        return currentYear - this.birth;
    }
}
```

▶ Run

观察上述代码，`calcAge()` 是一个 `private` 方法，外部代码无法调用，但是，内部方法 `getAge()` 可以调用它。

此外，我们还注意到，这个 `Person` 类只定义了 `birth` 字段，没有定义 `age` 字段，获取 `age` 时，通过方法 `getAge()` 返回的是一个实时计算的值，并非存储在某个字段的值。这说明方法可以封装一个类的对外接口，调用方不需要知道也不关心 `Person` 实例在内部到底有没有 `age` 字段。

this变量

在方法内部，可以使用一个隐含的变量 `this`，它始终指向当前实例。因此，通过 `this.field` 就可以访问当前实例的字段。

如果没有命名冲突，可以省略 `this`。例如：

```
class Person {
    private String name;

    public String getName() {
        return name; // 相当于this.name
```

```
    }
}
```

但是，如果有局部变量和字段重名，那么局部变量优先级更高，就必须加上 `this`：

```
class Person {
    private String name;

    public void setName(String name) {
        this.name = name; // 前面的this不可少，少了就变成局部变量name了
    }
}
```

方法参数

方法可以包含0个或任意个参数。方法参数用于接收传递给方法的变量值。调用方法时，必须严格按照参数的定义一一传递。例如：

```
class Person {
    ...
    public void setNameAndAge(String name, int age) {
        ...
    }
}
```

调用这个 `setNameAndAge()` 方法时，必须有两个参数，且第一个参数必须为 `String`，第二个参数必须为 `int`：

```
Person ming = new Person();
ming.setNameAndAge("Xiao Ming"); // 编译错误：参数个数不对
ming.setNameAndAge(12, "Xiao Ming"); // 编译错误：参数类型不对
```

可变参数

可变参数用 `类型...` 定义，可变参数相当于数组类型：

```
class Group {
    private String[] names;

    public void setNames(String... names) {
        this.names = names;
```

```
    }
}
```

上面的 `setNames()` 就定义了一个可变参数。调用时，可以这么写：

```
Group g = new Group();
g.setNames("Xiao Ming", "Xiao Hong", "Xiao Jun"); // 传入3个String
g.setNames("Xiao Ming", "Xiao Hong"); // 传入2个String
g.setNames("Xiao Ming"); // 传入1个String
g.setNames(); // 传入0个String
```

完全可以把可变参数改写为 `String[]` 类型：

```
class Group {
    private String[] names;

    public void setNames(String[] names) {
        this.names = names;
    }
}
```

但是，调用方需要自己先构造 `String[]` ，比较麻烦。例如：

```
Group g = new Group();
g.setNames(new String[] {"Xiao Ming", "Xiao Hong", "Xiao Jun"}); // 传入1个
String[]
```

另一个问题是，调用方可以传入 `null` ：

```
Group g = new Group();
g.setNames(null);
```

而可变参数可以保证无法传入 `null` ，因为传入0个参数时，接收到的实际值是一个空数组而不是 `null` 。

参数绑定

调用方把参数传递给实例方法时，调用时传递的值会按参数位置一一绑定。

那什么是参数绑定？

我们先观察一个基本类型参数的传递：

```
// 基本类型参数绑定
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        int n = 15; // n的值为15
        p.setAge(n); // 传入n的值
        System.out.println(p.getAge()); // 15
        n = 20; // n的值改为20
        System.out.println(p.getAge()); // 15还是20?
    }
}

class Person {
    private int age;

    public int getAge() {
        return this.age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

▶ Run

运行代码，从结果可知，修改外部的局部变量 `n` ，不影响实例 `p` 的 `age` 字段，原因是 `setAge()` 方法获得的参数，复制了 `n` 的值，因此， `p.age` 和局部变量 `n` 互不影响。

结论：基本类型参数的传递，是调用方值的复制。双方各自的后续修改，互不影响。

我们再看一个传递引用参数的例子：

```
// 引用类型参数绑定
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        String[] fullname = new String[] { "Homer", "Simpson" };
        p.setName(fullname); // 传入fullname数组
        System.out.println(p.getName()); // "Homer Simpson"
        fullname[0] = "Bart"; // fullname数组的第一个元素修改为"Bart"
        System.out.println(p.getName()); // "Homer Simpson"还是"Bart Simpson"?
    }
}

class Person {
    private String[] name;

    public String getName() {
        return this.name[0] + " " + this.name[1];
    }

    public void setName(String[] name) {
        this.name = name;
    }
}
```

▶ Run

注意到 `setName()` 的参数现在是一个数组。一开始，把 `fullname` 数组传进去，然后，修改 `fullname` 数组的内容，结果发现，实例 `p` 的字段 `p.name` 也被修改了！

结论：引用类型参数的传递，调用方的变量，和接收方的参数变量，指向的是同一个对象。双方任意一方对这个对象的修改，都会影响对方（因为指向同一个对象嘛）。

有了上面的结论，我们再看一个例子：

```
// 引用类型参数绑定
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        String bob = "Bob";
        p.setName(bob); // 传入bob变量
        System.out.println(p.getName()); // "Bob"
        bob = "Alice"; // bob改名为Alice
        System.out.println(p.getName()); // "Bob"还是"Alice"?
    }
}

class Person {
    private String name;

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

▶ Run

不要怀疑引用参数绑定的机制，试解释为什么上面的代码两次输出都是 `"Bob"` 。

练习

```
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        ming.setName("小明");
        ming.setAge(12);
        System.out.println(ming.getAge());
    }
}
```

```
class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

▶ Run

下载练习：给Person类增加getAge/setAge方法 （推荐使用IDE练习插件快速下载）

小结

- 方法可以让外部代码安全地访问实例字段；
- 方法是一组执行语句，并且可以执行任意逻辑；
- 方法内部遇到return时返回，void表示不返回任何值（注意和返回null不同）；
- 外部代码通过public方法操作实例，内部代码可以调用private方法；
- 理解方法的参数绑定。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





构造方法

创建实例的时候，我们经常需要同时初始化这个实例的字段，例如：

```
1. Person ming = new Person();
2. ming.setName("小明");
3. ming.setAge(12);
```

初始化对象实例需要3行代码，而且，如果忘了调用 `setName()` 或者 `setAge()` ，这个实例内部的状态就是不正确的。

能否在创建对象实例时就把内部字段全部初始化为合适的值？

完全可以。

这时，我们就需要构造方法。

创建实例的时候，实际上是通过构造方法来初始化实例的。我们先来定义一个构造方法，能在创建 `Person` 实例的时候，一次性传入 `name` 和 `age` ，完成初始化：

```
// 构造方法
public class Main {
    public static void main(String[] args) {
        Person p = new Person("Xiao Ming", 15);
        System.out.println(p.getName());
        System.out.println(p.getAge());
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

▶ Run

由于构造方法是如此特殊，所以构造方法的名称就是类名。构造方法的参数没有限制，在方法内部，也可以编写任意语句。但是，和普通方法相比，构造方法没有返回值（也没有 `void`），调用构造方法，必须用 `new` 操作符。

默认构造方法

是不是任何 `class` 都有构造方法？是的。

那前面我们并没有为 `Person` 类编写构造方法，为什么可以调用 `new Person()` ？

原因是如果一个类没有定义构造方法，编译器会自动为我们生成一个默认构造方法，它没有参数，也没有执行语句，类似这样：

```
1. class Person {
2.     public Person() {
3.     }
4. }
```

要特别注意的是，如果我们自定义了一个构造方法，那么，编译器就不再自动创建默认构造方法：

```
// 构造方法
public class Main {
    public static void main(String[] args) {
        Person p = new Person(); // 编译错误:找不到这个构造方法
    }
}

class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

▶ Run

如果既要能使用带参数的构造方法，又想保留不带参数的构造方法，那么只能把两个构造方法都定义出

来：

```
// 构造方法
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Xiao Ming", 15); // 既可以调用带参数的构造方法
        Person p2 = new Person(); // 也可以调用无参数构造方法
    }
}

class Person {
    private String name;
    private int age;

    public Person() {
    }

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() {
        return this.name;
    }

    public int getAge() {
        return this.age;
    }
}
```

▶ Run

没有在构造方法中初始化字段时，引用类型的字段默认是 `null` ，数值类型的字段用默认值， `int` 类型默认值是 `0` ，布尔类型默认值是 `false` ：

```
class Person {
    private String name; // 默认初始化为null
    private int age; // 默认初始化为0

    public Person() {
    }
}
```

也可以对字段直接进行初始化：

```
class Person {
    private String name = "Unamed";
    private int age = 10;
```

```
}
```

那么问题来了：既对字段进行初始化，又在构造方法中对字段进行初始化：

```
class Person {
    private String name = "Unamed";
    private int age = 10;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

当我们创建对象的时候，`new Person("Xiao Ming", 12)`得到的对象实例，字段的初始值是啥？

在Java中，创建对象实例的时候，按照如下顺序进行初始化：

- 先初始化字段，例如，`int age = 10;`表示字段初始化为`10`，`double salary;`表示字段默认初始化为`0`，`String name;`表示引用类型字段默认初始化为`null`；
- 执行构造方法的代码进行初始化。

因此，构造方法的代码由于后运行，所以，`new Person("Xiao Ming", 12)`的字段值最终由构造方法的代码确定。

多构造方法

可以定义多个构造方法，在通过`new`操作符调用的时候，编译器通过构造方法的参数数量、位置和类型自动区分：

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this.name = name;
        this.age = 12;
```

```
    }

    public Person() {
    }
}
```

如果调用 `new Person("Xiao Ming", 20);` ，会自动匹配到构造方法 `public Person(String, int)` 。

如果调用 `new Person("Xiao Ming");` ，会自动匹配到构造方法 `public Person(String)` 。

如果调用 `new Person();` ，会自动匹配到构造方法 `public Person()` 。

一个构造方法可以调用其他构造方法，这样做的目的是便于代码复用。调用其他构造方法的语法是 `this(…)` ：

```
class Person {
    private String name;
    private int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public Person(String name) {
        this(name, 18); // 调用另一个构造方法Person(String, int)
    }

    public Person() {
        this("Unnamed"); // 调用另一个构造方法Person(String)
    }
}
```

练习

请给 `Person` 类增加 `(String, int)` 的构造方法：

```
public class Main {
    public static void main(String[] args) {
        // TODO: 给Person增加构造方法:
        Person ming = new Person("小明", 12);
        System.out.println(ming.getName());
        System.out.println(ming.getAge());
    }
}
```

```
class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}
```

▶ Run

下载练习：给Person类增加(String, int)的构造方法 （推荐使用IDE练习插件快速下载）

小结

实例在创建时通过 `new` 操作符会调用其对应的构造方法，构造方法用于初始化实例；

没有定义构造方法时，编译器会自动创建一个默认的无参数构造方法；

可以定义多个构造方法，编译器根据参数自动判断；

可以在一个构造方法内部调用另一个构造方法，便于代码复用。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





方法重载

在一个类中，我们可以定义多个方法。如果有一系列方法，它们的功能都是类似的，只有参数有所不同，那么，可以把这一组方法名做成同名方法。例如，在 `Hello` 类中，定义多个 `hello()` 方法：

```
class Hello {
    public void hello() {
        System.out.println("Hello, world!");
    }

    public void hello(String name) {
        System.out.println("Hello, " + name + "!");
    }

    public void hello(String name, int age) {
        if (age < 18) {
            System.out.println("Hi, " + name + "!");
        } else {
            System.out.println("Hello, " + name + "!");
        }
    }
}
```

这种方法名相同，但各自的参数不同，称为方法重载（ `Overload` ）。

注意：方法重载的返回值类型通常都是相同的。

方法重载的目的是，功能类似的方法使用同一名字，更容易记住，因此，调用起来更简单。

举个例子， `String` 类提供了多个重载方法 `indexOf()` ，可以查找子串：

- `int indexOf(int ch)` ：根据字符的Unicode码查找；
- `int indexOf(String str)` ：根据字符串查找；
- `int indexOf(int ch, int fromIndex)` ：根据字符查找，但指定起始位置；
- `int indexOf(String str, int fromIndex)` 根据字符串查找，但指定起始位置。

试一试：

```
// String.indexOf()
```

```
public class Main {
    public static void main(String[] args) {
        String s = "Test string";
        int n1 = s.indexOf('t');
        int n2 = s.indexOf("st");
        int n3 = s.indexOf("st", 4);
        System.out.println(n1);
        System.out.println(n2);
        System.out.println(n3);
    }
}
```

▶ Run

练习

```
public class Main {
    public static void main(String[] args) {
        Person ming = new Person();
        Person hong = new Person();
        ming.setName("Xiao Ming");
        // TODO: 给Person增加重载方法setName(String, String):
        hong.setName("Xiao", "Hong");
        System.out.println(ming.getName());
        System.out.println(hong.getName());
    }
}
```

```
class Person {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

▶ Run

下载练习：给Person增加重载方法 （推荐使用IDE练习插件快速下载）

小结

方法重载是指多个方法的方法名相同，但各自的参数不同；

重载方法应该完成类似的功能，参考 `String` 的 `indexOf()` ；

重载方法返回值类型应该相同。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





继承

在前面的章节中，我们已经定义了 `Person` 类：

```
class Person {
    private String name;
    private int age;

    public String getName() {...}
    public void setName(String name) {...}
    public int getAge() {...}
    public void setAge(int age) {...}
}
```

现在，假设需要定义一个 `Student` 类，字段如下：

```
class Student {
    private String name;
    private int age;
    private int score;

    public String getName() {...}
    public void setName(String name) {...}
    public int getAge() {...}
    public void setAge(int age) {...}
    public int getScore() { … }
    public void setScore(int score) { … }
}
```

仔细观察，发现 `Student` 类包含了 `Person` 类已有的字段和方法，只是多出了一个 `score` 字段和相应的 `getScore()` 、 `setScore()` 方法。

能不能在 `Student` 中不要写重复的代码？

这个时候，继承就派上用场了。

继承是面向对象编程中非常强大的一种机制，它首先可以复用代码。当我们让 `Student` 从 `Person` 继承时， `Student` 就获得了 `Person` 的所有功能，我们只需要为 `Student` 编写新增的功能。

Java使用 `extends` 关键字来实现继承：

```
class Person {
    private String name;
    private int age;

    public String getName() {...}
    public void setName(String name) {...}
    public int getAge() {...}
    public void setAge(int age) {...}
}

class Student extends Person {
    // 不要重复name和age字段/方法,
    // 只需要定义新增score字段/方法:
    private int score;

    public int getScore() { … }
    public void setScore(int score) { … }
}
```

可见，通过继承，`Student`只需要编写额外的功能，不再需要重复代码。

在OOP的术语中，我们把`Person`称为超类（super class），父类（parent class），基类（base class），把`Student`称为子类（subclass），扩展类（extended class）。

继承树

注意到我们在定义`Person`的时候，没有写`extends`。在Java中，没有明确写`extends`的类，编译器会自动加上`extends Object`。所以，任何类，除了`Object`，都会继承自某个类。下图是`Person`、`Student`的继承树：

```
┌───────────┐
│  Object   │
└───────────┘
      ▲
      │
┌───────────┐
│  Person   │
└───────────┘
      ▲
      │
┌───────────┐
│  Student  │
```

```
└─────────────┘
```

Java只允许一个class继承自一个类，因此，一个类有且仅有一个父类。只有 `Object` 特殊，它没有父类。

类似的，如果我们定义一个继承自 `Person` 的 `Teacher` ，它们的继承树关系如下：

```
        ┌─────────────┐
        │  Object     │
        └─────────────┘
              ▲
              │
        ┌─────────────┐
        │  Person     │
        └─────────────┘
           ▲     ▲
           │     │
           │     │
┌──────────┐  ┌──────────┐
│  Student │  │  Teacher │
└──────────┘  └──────────┘
```

protected

继承有个特点，就是子类无法访问父类的 `private` 字段或者 `private` 方法。例如， `Student` 类就无法访问 `Person` 类的 `name` 和 `age` 字段：

```
class Person {
    private String name;
    private int age;
}

class Student extends Person {
    public String hello() {
        return "Hello, " + name; // 编译错误：无法访问name字段
    }
}
```

这使得继承的作用被削弱了。为了让子类可以访问父类的字段，我们需要把 `private` 改为 `protected` 。用 `protected` 修饰的字段可以被子类访问：

```
class Person {
    protected String name;
    protected int age;
}

class Student extends Person {
    public String hello() {
        return "Hello, " + name; // OK!
    }
}
```

因此，`protected` 关键字可以把字段和方法的访问权限控制在继承树内部，一个 `protected` 字段和方法可以被其子类，以及子类的子类所访问，后面我们还会详细讲解。

super

`super` 关键字表示父类（超类）。子类引用父类的字段时，可以用 `super.fieldName` 。例如：

```
class Student extends Person {
    public String hello() {
        return "Hello, " + super.name;
    }
}
```

实际上，这里使用 `super.name` ，或者 `this.name` ，或者 `name` ，效果都是一样的。编译器会自动定位到父类的 `name` 字段。

但是，在某些时候，就必须使用 `super` 。我们来看一个例子：

```
// super
public class Main {
    public static void main(String[] args) {
        Student s = new Student("Xiao Ming", 12, 89);
    }
}

class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}

class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        this.score = score;
    }
}
```

▶ Run

运行上面的代码，会得到一个编译错误，大意是在 `Student` 的构造方法中，无法调用 `Person` 的构造方法。

这是因为在Java中，任何 `class` 的构造方法，第一行语句必须是调用父类的构造方法。如果没有明确地调用父类的构造方法，编译器会帮我们自动加一句 `super();` ，所以， `Student` 类的构造方法实际上是这样：

```
class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(); // 自动调用父类的构造方法
        this.score = score;
    }
}
```

但是， `Person` 类并没有无参数的构造方法，因此，编译失败。

解决方法是调用 `Person` 类存在的某个构造方法。例如：

```
class Student extends Person {
```

```
    protected int score;

    public Student(String name, int age, int score) {
        super(name, age); // 调用父类的构造方法Person(String, int)
        this.score = score;
    }
}
```

这样就可以正常编译了！

因此我们得出结论：如果父类没有默认的构造方法，子类就必须显式调用 `super()` 并给出参数以便让编译器定位到父类的一个合适的构造方法。

这里还顺带引出了另一个问题：即子类不会继承任何父类的构造方法。子类默认的构造方法是编译器自动生成的，不是继承的。

向上转型

如果一个引用变量的类型是 `Student` ，那么它可以指向一个 `Student` 类型的实例：

```
Student s = new Student();
```

如果一个引用类型的变量是 `Person` ，那么它可以指向一个 `Person` 类型的实例：

```
Person p = new Person();
```

现在问题来了：如果 `Student` 是从 `Person` 继承下来的，那么，一个引用类型为 `Person` 的变量，能否指向 `Student` 类型的实例？

```
Person p = new Student(); // ???
```

测试一下就可以发现，这种指向是允许的！

这是因为 `Student` 继承自 `Person` ，因此，它拥有 `Person` 的全部功能。 `Person` 类型的变量，如果指向 `Student` 类型的实例，对它进行操作，是没有问题的！

这种把一个子类类型安全地变为父类类型的赋值，被称为向上转型（upcasting）。

向上转型实际上是把一个子类型安全地变为更加抽象的父类型：

```
Student s = new Student();
Person p = s; // upcasting, ok
Object o1 = p; // upcasting, ok
```

```
4.  Object o2 = s; // upcasting, ok
```

注意到继承树是 `Student > Person > Object` ，所以，可以把 `Student` 类型转型为 `Person` ，或者更高层次的 `Object` 。

向下转型

和向上转型相反，如果把一个父类类型强制转型为子类类型，就是向下转型（downcasting）。例如：

```
1.  Person p1 = new Student(); // upcasting, ok
2.  Person p2 = new Person();
3.  Student s1 = (Student) p1; // ok
4.  Student s2 = (Student) p2; // runtime error! ClassCastException!
```

如果测试上面的代码，可以发现：

`Person` 类型 `p1` 实际指向 `Student` 实例， `Person` 类型变量 `p2` 实际指向 `Person` 实例。在向下转型的时候，把 `p1` 转型为 `Student` 会成功，因为 `p1` 确实指向 `Student` 实例，把 `p2` 转型为 `Student` 会失败，因为 `p2` 的实际类型是 `Person` ，不能把父类变为子类，因为子类功能比父类多，多的功能无法凭空变出来。

因此，向下转型很可能会失败。失败的时候，Java虚拟机会报 `ClassCastException` 。

为了避免向下转型出错，Java提供了 `instanceof` 操作符，可以先判断一个实例究竟是不是某种类型：

```
1.  Person p = new Person();
2.  System.out.println(p instanceof Person); // true
3.  System.out.println(p instanceof Student); // false
4.
5.  Student s = new Student();
6.  System.out.println(s instanceof Person); // true
7.  System.out.println(s instanceof Student); // true
8.
9.  Student n = null;
10. System.out.println(n instanceof Student); // false
```

`instanceof` 实际上判断一个变量所指向的实例是否是指定类型，或者这个类型的子类。如果一个引用变量为 `null` ，那么对任何 `instanceof` 的判断都为 `false` 。

利用 `instanceof` ，在向下转型前可以先判断：

```
Person p = new Student();
if (p instanceof Student) {
    // 只有判断成功才会向下转型:
    Student s = (Student) p; // 一定会成功
}
```

区分继承和组合

在使用继承时，我们要注意逻辑一致性。

考察下面的 `Book` 类：

```
class Book {
    protected String name;
    public String getName() {...}
    public void setName(String name) {...}
}
```

这个 `Book` 类也有 `name` 字段，那么，我们能不能让 `Student` 继承自 `Book` 呢？

```
class Student extends Book {
    protected int score;
}
```

显然，从逻辑上讲，这是不合理的， `Student` 不应该从 `Book` 继承，而应该从 `Person` 继承。

究其原因，是因为 `Student` 是 `Person` 的一种，它们是is关系，而 `Student` 并不是 `Book` 。实际上 `Student` 和 `Book` 的关系是has关系。

具有has关系不应该使用继承，而是使用组合，即 `Student` 可以持有一个 `Book` 实例：

```
class Student extends Person {
    protected Book book;
    protected int score;
}
```

因此，继承是is关系，组合是has关系。

练习

定义 `PrimaryStudent` ，从 `Student` 继承，并新增一个 `grade` 字段：

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person("小明", 12);
        Student s = new Student("小红", 20, 99);
        // TODO: 定义PrimaryStudent，从Student继承，新增grade字段:
        Student ps = new PrimaryStudent("小军", 9, 100, 5);
        System.out.println(ps.getScore());
    }
}

class Person {
    protected String name;
    protected int age;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return age; }
    public void setAge(int age) { this.age = age; }
}

class Student extends Person {
    protected int score;

    public Student(String name, int age, int score) {
        super(name, age);
        this.score = score;
    }

    public int getScore() { return score; }
}
```

```
class PrimaryStudent {
    // TODO
}
```

▶ Run

下载练习：继承练习 （推荐使用IDE练习插件快速下载）

小结

- 继承是面向对象编程的一种强大的代码复用方式；
- Java只允许单继承，所有类最终的根类是 `Object` ；
- `protected` 允许子类访问父类的字段和方法；
- 子类的构造方法可以通过 `super()` 调用父类的构造方法；

- 可以安全地向上转型为更抽象的类型；
- 可以强制向下转型，最好借助 `instanceof` 判断；
- 子类和父类的关系是is，has关系不能用继承。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





多态

在继承关系中，子类如果定义了一个与父类方法签名完全相同的方法，被称为覆写（Override）。

例如，在 `Person` 类中，我们定义了 `run()` 方法：

```
class Person {
    public void run() {
        System.out.println("Person.run");
    }
}
```

在子类 `Student` 中，覆写这个 `run()` 方法：

```
class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}
```

Override和Overload不同的是，如果方法签名如果不同，就是Overload，Overload方法是一个新方法；如果方法签名相同，并且返回值也相同，就是 `Override` 。

注意：方法名相同，方法参数相同，但方法返回值不同，也是不同的方法。在Java程序中，出现这种情况，编译器会报错。

```
class Person {
    public void run() { … }
}

class Student extends Person {
    // 不是Override，因为参数不同:
    public void run(String s) { … }
    // 不是Override，因为返回值不同:
    public int run() { … }
}
```

加上 `@Override` 可以让编译器帮助检查是否进行了正确的覆写。希望进行覆写，但是不小心写错了方法签名，编译器会报错。

```
// override
public class Main {
    public static void main(String[] args) {
    }
}

class Person {
    public void run() {}
}

public class Student extends Person {
    @Override // Compile error!
    public void run(String s) {}
}
```

▶ Run

但是 `@Override` 不是必需的。

在上一节中，我们已经知道，引用变量的声明类型可能与其实际类型不符，例如：

```
1.  Person p = new Student();
```

现在，我们考虑一种情况，如果子类覆写了父类的方法：

```
// override
public class Main {
    public static void main(String[] args) {
        Person p = new Student();
        p.run(); // 应该打印Person.run还是Student.run?
    }
}

class Person {
    public void run() {
        System.out.println("Person.run");
    }
}

class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}
```

▶ Run

那么，一个实际类型为 `Student` ，引用类型为 `Person` 的变量，调用其 `run()` 方法，调用的是 `Person` 还是 `Student` 的 `run()` 方法？

运行一下上面的代码就可以知道，实际上调用的方法是 `Student` 的 `run()` 方法。因此可得出结论：

Java的实例方法调用是基于运行时的实际类型的动态调用，而非变量的声明类型。

这个非常重要的特性在面向对象编程中称之为多态。它的英文拼写非常复杂：Polymorphic。

多态

多态是指，针对某个类型的方法调用，其真正执行的方法取决于运行时期实际类型的方法。例如：

```
1. Person p = new Student();
2. p.run(); // 无法确定运行时究竟调用哪个run()方法
```

有童鞋会问，从上面的代码一看就明白，肯定调用的是 `Student` 的 `run()` 方法啊。

但是，假设我们编写这样一个方法：

```
1. public void runTwice(Person p) {
2.     p.run();
3.     p.run();
4. }
```

它传入的参数类型是 `Person` ，我们是无法知道传入的参数实际类型究竟是 `Person` ，还是 `Student` ，还是 `Person` 的其他子类，因此，也无法确定调用的是不是 `Person` 类定义的 `run()` 方法。

所以，多态的特性就是，运行期才能动态决定调用的子类方法。对某个类型调用某个方法，执行的实际方法可能是某个子类的覆写方法。这种不确定性的方法调用，究竟有什么作用？

我们还是来举栗子。

假设我们定义一种收入，需要给它报税，那么先定义一个 `Income` 类：

```
1. class Income {
2.     protected double income;
3.     public double getTax() {
4.         return income * 0.1; // 税率10%
5.     }
6. }
```

对于工资收入，可以减去一个基数，那么我们可以从 `Income` 派生出 `SalaryIncome` ，并覆

写 `getTax()`：

```
class Salary extends Income {
    @Override
    public double getTax() {
        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}
```

如果你享受国务院特殊津贴，那么按照规定，可以全部免税：

```
class StateCouncilSpecialAllowance extends Income {
    @Override
    public double getTax() {
        return 0;
    }
}
```

现在，我们要编写一个报税的财务软件，对于一个人的所有收入进行报税，可以这么写：

```
public double totalTax(Income... incomes) {
    double total = 0;
    for (Income income: incomes) {
        total = total + income.getTax();
    }
    return total;
}
```

来试一下：

```
// Polymorphic
public class Main {
    public static void main(String[] args) {
        // 给一个有普通收入、工资收入和享受国务院特殊津贴的小伙伴算税:
        Income[] incomes = new Income[] {
            new Income(3000),
            new Salary(7500),
            new StateCouncilSpecialAllowance(15000)
        };
        System.out.println(totalTax(incomes));
    }

    public static double totalTax(Income... incomes) {
        double total = 0;
        for (Income income: incomes) {
            total = total + income.getTax();
        }
        return total;
    }
}

class Income {
    protected double income;

    public Income(double income) {
        this.income = income;
    }

    public double getTax() {
        return income * 0.1; // 税率10%
    }
}

class Salary extends Income {
    public Salary(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        if (income <= 5000) {
            return 0;
        }
        return (income - 5000) * 0.2;
    }
}

class StateCouncilSpecialAllowance extends Income {
    public StateCouncilSpecialAllowance(double income) {
        super(income);
    }

    @Override
    public double getTax() {
        return 0;
    }
}
```

▶ Run

观察 `totalTax()` 方法：利用多态，`totalTax()` 方法只需要和 `Income` 打交道，它完全不需要知道 `Salary` 和 `StateCouncilSpecialAllowance` 的存在，就可以正确计算出总的税。如果我们要新增一种稿费收入，只需要从 `Income` 派生，然后正确覆写 `getTax()` 方法就可以。把新的类型传入 `totalTax()` ，不需要修改任何代码。

可见，多态具有一个非常强大的功能，就是允许添加更多类型的子类实现功能扩展，却不需要修改基于父类的代码。

覆写Object方法

因为所有的 `class` 最终都继承自 `Object` ，而 `Object` 定义了几个重要的方法：

- `toString()` ：把instance输出为 `String` ；
- `equals()` ：判断两个instance是否逻辑相等；
- `hashCode()` ：计算一个instance的哈希值。

在必要的情况下，我们可以覆写 `Object` 的这几个方法。例如：

```
class Person {
    ...
    // 显示更有意义的字符串:
    @Override
    public String toString() {
        return "Person:name=" + name;
    }

    // 比较是否相等:
    @Override
    public boolean equals(Object o) {
        // 当且仅当o为Person类型:
        if (o instanceof Person) {
            Person p = (Person) o;
            // 并且name字段相同时，返回true:
            return this.name.equals(p.name);
        }
        return false;
    }

    // 计算hash:
    @Override
    public int hashCode() {
        return this.name.hashCode();
    }
```

```
}
```

调用super

在子类的覆写方法中，如果要调用父类的被覆写的方法，可以通过 `super` 来调用。例如：

```
class Person {
    protected String name;
    public String hello() {
        return "Hello, " + name;
    }
}

Student extends Person {
    @Override
    public String hello() {
        // 调用父类的hello()方法:
        return super.hello() + "!";
    }
}
```

final

继承可以允许子类覆写父类的方法。如果一个父类不允许子类对它的某个方法进行覆写，可以把该方法标记为 `final` 。用 `final` 修饰的方法不能被 `Override` ：

```
class Person {
    protected String name;
    public final String hello() {
        return "Hello, " + name;
    }
}

Student extends Person {
    // compile error: 不允许覆写
    @Override
    public String hello() {
    }
}
```

如果一个类不希望任何其他类继承自它，那么可以把这个类本身标记为 `final` 。用 `final` 修饰的

类不能被继承：

```
final class Person {
    protected String name;
}

// compile error: 不允许继承自Person
Student extends Person {
}
```

对于一个类的实例字段，同样可以用 `final` 修饰。用 `final` 修饰的字段在初始化后不能被修改。例如：

```
class Person {
    public final String name = "Unamed";
}
```

对 `final` 字段重新赋值会报错：

```
Person p = new Person();
p.name = "New Name"; // compile error!
```

可以在构造方法中初始化final字段：

```
class Person {
    public final String name;
    public Person(String name) {
        this.name = name;
    }
}
```

这种方法更为常用，因为可以保证实例一旦创建，其 `final` 字段就不可修改。

练习

给一个有工资收入和稿费收入的小伙伴算税。

下载练习：计算所得税 （推荐使用IDE练习插件快速下载）

小结

- 子类可以覆写父类的方法（Override），覆写在子类中改变了父类方法的行为；

- Java的方法调用总是作用于运行期对象的实际类型，这种行为称为多态；
- `final` 修饰符有多种作用：
  - `final` 修饰的方法可以阻止被覆写；
  - `final` 修饰的class可以阻止被继承；
  - `final` 修饰的field必须在创建对象时初始化，随后不可修改。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





抽象类

由于多态的存在，每个子类都可以覆写父类的方法，例如：

```
class Person {
    public void run() { … }
}

class Student extends Person {
    @Override
    public void run() { … }
}

class Teacher extends Person {
    @Override
    public void run() { … }
}
```

从 `Person` 类派生的 `Student` 和 `Teacher` 都可以覆写 `run()` 方法。

如果父类 `Person` 的 `run()` 方法没有实际意义，能否去掉方法的执行语句？

```
class Person {
    public void run(); // Compile Error!
}
```

答案是不行，会导致编译错误，因为定义方法的时候，必须实现方法的语句。

能不能去掉父类的 `run()` 方法？

答案还是不行，因为去掉父类的 `run()` 方法，就失去了多态的特性。例如， `runTwice()` 就无法编译：

```
public void runTwice(Person p) {
    p.run(); // Person没有run()方法，会导致编译错误
    p.run();
}
```

如果父类的方法本身不需要实现任何功能，仅仅是为了定义方法签名，目的是让子类去覆写它，那么，可以把父类的方法声明为抽象方法：

```
class Person {
```

```
    public abstract void run();
}
```

把一个方法声明为 `abstract` ，表示它是一个抽象方法，本身没有实现任何方法语句。因为这个抽象方法本身是无法执行的，所以， `Person` 类也无法被实例化。编译器会告诉我们，无法编译 `Person` 类，因为它包含抽象方法。

必须把 `Person` 类本身也声明为 `abstract` ，才能正确编译它：

```
abstract class Person {
    public abstract void run();
}
```

抽象类

如果一个 `class` 定义了方法，但没有具体执行代码，这个方法就是抽象方法，抽象方法用 `abstract` 修饰。

因为无法执行抽象方法，因此这个类也必须申明为抽象类（`abstract class`）。

使用 `abstract` 修饰的类就是抽象类。我们无法实例化一个抽象类：

```
Person p = new Person(); // 编译错误
```

无法实例化的抽象类有什么用？

因为抽象类本身被设计成只能用于被继承，因此，抽象类可以强迫子类实现其定义的抽象方法，否则编译会报错。因此，抽象方法实际上相当于定义了“规范”。

例如， `Person` 类定义了抽象方法 `run()` ，那么，在实现子类 `Student` 的时候，就必须覆写 `run()` 方法：

```
// abstract class
public class Main {
    public static void main(String[] args) {
        Person p = new Student();
        p.run();
    }
}

abstract class Person {
    public abstract void run();
}

class Student extends Person {
    @Override
    public void run() {
        System.out.println("Student.run");
    }
}
```

▶ Run

面向抽象编程

当我们定义了抽象类 `Person`，以及具体的 `Student`、`Teacher` 子类的时候，我们可以通过抽象类 `Person` 类型去引用具体的子类的实例：

```
Person s = new Student();
Person t = new Teacher();
```

这种引用抽象类的好处在于，我们对其进行方法调用，并不关心 `Person` 类型变量的具体子类型：

```
// 不关心Person变量的具体子类型:
s.run();
t.run();
```

同样的代码，如果引用的是一个新的子类，我们仍然不关心具体类型：

```
// 同样不关心新的子类是如何实现run()方法的：
Person e = new Employee();
e.run();
```

这种尽量引用高层类型，避免引用实际子类型的方式，称之为面向抽象编程。

面向抽象编程的本质就是：

- 上层代码只定义规范（例如：`abstract class Person`）；

- 不需要子类就可以实现业务逻辑（正常编译）；
- 具体的业务逻辑由不同的子类实现，调用者并不关心。

练习

用抽象类给一个有工资收入和稿费收入的小伙伴算税。

下载练习：用抽象类算税 （推荐使用IDE练习插件快速下载）

小结

- 通过 `abstract` 定义的方法是抽象方法，它只有定义，没有实现。抽象方法定义了子类必须实现的接口规范；
- 定义了抽象方法的class必须被定义为抽象类，从抽象类继承的子类必须实现抽象方法；
- 如果不实现抽象方法，则该子类仍是一个抽象类；
- 面向抽象编程使得调用者只关心抽象方法的定义，不关心子类的具体实现。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





接口

在抽象类中，抽象方法本质上是定义接口规范：即规定高层类的接口，从而保证所有子类都有相同的接口实现，这样，多态就能发挥出威力。

如果一个抽象类没有字段，所有方法全部都是抽象方法：

```
abstract class Person {
    public abstract void run();
    public abstract String getName();
}
```

就可以把该抽象类改写为接口： `interface` 。

在Java中，使用 `interface` 可以声明一个接口：

```
interface Person {
    void run();
    String getName();
}
```

所谓 `interface` ，就是比抽象类还要抽象的纯抽象接口，因为它连字段都不能有。因为接口定义的所有方法默认都是 `public abstract` 的，所以这两个修饰符不需要写出来（写不写效果都一样）。

当一个具体的 `class` 去实现一个 `interface` 时，需要使用 `implements` 关键字。举个例子：

```
class Student implements Person {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println(this.name + " run");
    }

    @Override
    public String getName() {
        return this.name;
    }
```

```
}
```

我们知道，在Java中，一个类只能继承自另一个类，不能从多个类继承。但是，一个类可以实现多个 `interface` ，例如：

```
class Student implements Person, Hello { // 实现了两个interface
    ...
}
```

术语

注意区分术语：

Java的接口特指 `interface` 的定义，表示一个接口类型和一组方法签名，而编程接口泛指接口规范，如方法签名，数据格式，网络协议等。

抽象类和接口的对比如下：

	abstract class	interface
继承	只能extends一个class	可以implements多个interface
字段	可以定义实例字段	不能定义实例字段
抽象方法	可以定义抽象方法	可以定义抽象方法
非抽象方法	可以定义非抽象方法	可以定义default方法

接口继承

一个 `interface` 可以继承自另一个 `interface` 。 `interface` 继承自 `interface` 使用 `extends` ，它相当于扩展了接口的方法。例如：

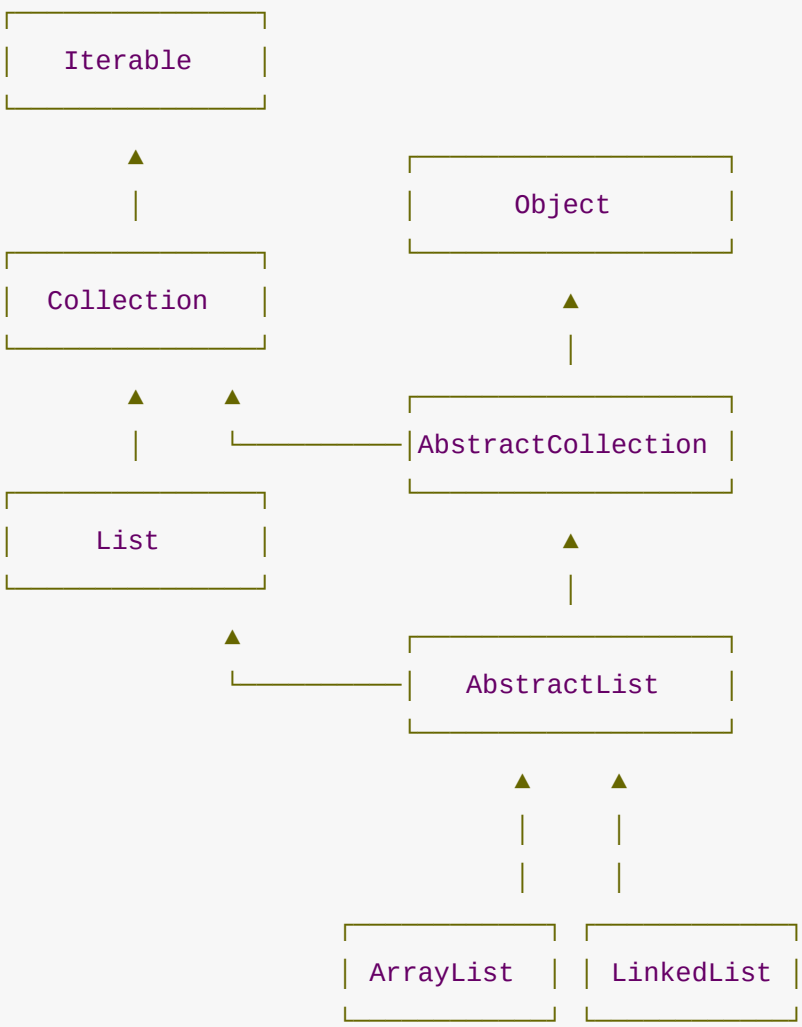
```
interface Hello {
    void hello();
}

interface Person extends Hello {
    void run();
    String getName();
}
```

此时， `Person` 接口继承自 `Hello` 接口，因此， `Person` 接口现在实际上有3个抽象方法签名，其中一个来自继承的 `Hello` 接口。

继承关系

合理设计 `interface` 和 `abstract class` 的继承关系，可以充分复用代码。一般来说，公共逻辑适合放在 `abstract class` 中，具体逻辑放到各个子类，而接口层次代表抽象程度。可以参考Java的集合类定义的一组接口、抽象类以及具体子类的继承关系：



在使用的时候，实例化的对象永远只能是某个具体的子类，但总是通过接口去引用它，因为接口比抽象类更抽象：

```
List list = new ArrayList(); // 用List接口引用具体子类的实例
Collection coll = list; // 向上转型为Collection接口
Iterable it = coll; // 向上转型为Iterable接口
```

default方法

在接口中，可以定义 `default` 方法。例如，把 `Person` 接口的 `run()` 方法改为 `default` 方法：

```
// interface
public class Main {
    public static void main(String[] args) {
        Person p = new Student("Xiao Ming");
        p.run();
    }
}

interface Person {
    String getName();
    default void run() {
        System.out.println(getName() + " run");
    }
}

class Student implements Person {
    private String name;

    public Student(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

▶ Run

实现类可以不必覆写 `default` 方法。 `default` 方法的目的是，当我们需要给接口新增一个方法时，会涉及到修改全部子类。如果新增的是 `default` 方法，那么子类就不必全部修改，只需要在需要覆写的地方去覆写新增方法。

`default` 方法和抽象类的普通方法是有所不同的。因为 `interface` 没有字段， `default` 方法无法访问字段，而抽象类的普通方法可以访问实例字段。

练习

用接口给一个有工资收入和稿费收入的小伙伴算税。

下载练习：用接口算税 （推荐使用IDE练习插件快速下载）

小结

Java的接口（interface）定义了纯抽象规范，一个类可以实现多个接口；

接口也是数据类型，适用于向上转型和向下转型；

接口的所有方法都是抽象方法，接口不能定义实例字段；

接口可以定义 `default` 方法（JDK>=1.8）。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





静态字段和静态方法

在一个 `class` 中定义的字段，我们称之为实例字段。实例字段的特点是，每个实例都有独立的字段，各个实例的同名字段互不影响。

还有一种字段，是用 `static` 修饰的字段，称为静态字段： `static field` 。

实例字段在每个实例中都有自己的一个独立“空间”，但是静态字段只有一个共享“空间”，所有实例都会共享该字段。举个例子：

```
1.  class Person {
2.      public String name;
3.      public int age;
4.      // 定义静态字段number:
5.      public static int number;
6.  }
```

我们来看看下面的代码：

```
// static field
public class Main {
    public static void main(String[] args) {
        Person ming = new Person("Xiao Ming", 12);
        Person hong = new Person("Xiao Hong", 15);
        ming.number = 88;
        System.out.println(hong.number);
        hong.number = 99;
        System.out.println(ming.number);
    }
}

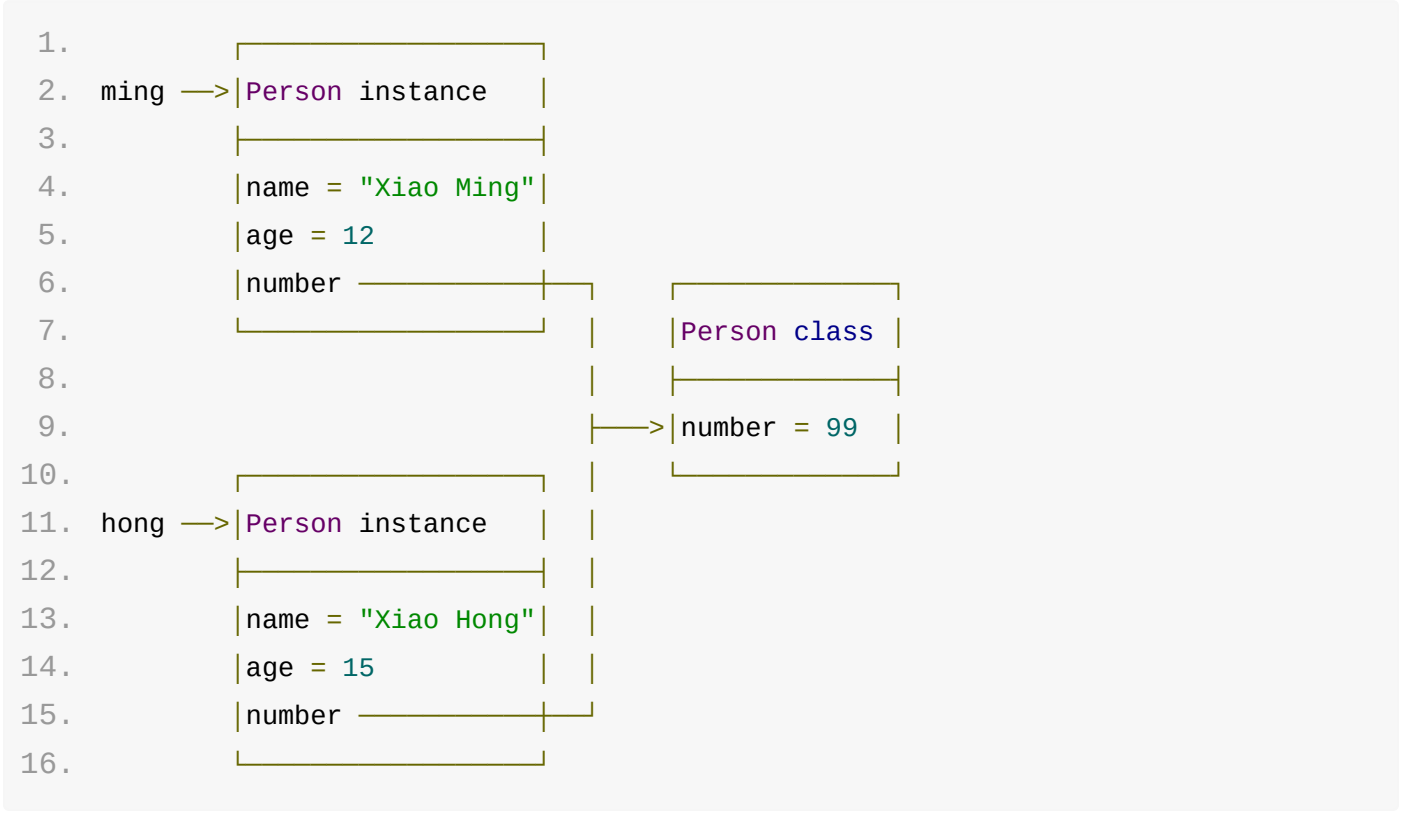
class Person {
    public String name;
    public int age;

    public static int number;

    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
}
```

▶ Run

对于静态字段，无论修改哪个实例的静态字段，效果都是一样的：所有实例的静态字段都被修改了，原因是静态字段并不属于实例：



虽然实例可以访问静态字段，但是它们指向的其实都是 `Person class` 的静态字段。所以，所有实例共享一个静态字段。

因此，不推荐用 `实例变量.静态字段` 去访问静态字段，因为在Java程序中，实例对象并没有静态字段。在代码中，实例对象能访问静态字段只是因为编译器可以根据实例类型自动转换为 `类名.静态字段` 来访问静态对象。

推荐用类名来访问静态字段。可以把静态字段理解为描述 `class` 本身的字段（非实例字段）。对于上面的代码，更好的写法是：

```
Person.number = 99;
System.out.println(Person.number);
```

静态方法

有静态字段，就有静态方法。用 `static` 修饰的方法称为静态方法。

调用实例方法必须通过一个实例变量，而调用静态方法则不需要实例变量，通过类名就可以调用。静态方法类似其它编程语言的函数。例如：

```
// static method
public class Main {
    public static void main(String[] args) {
        Person.setNumber(99);
        System.out.println(Person.number);
    }
}

class Person {
    public static int number;

    public static void setNumber(int value) {
        number = value;
    }
}
```

▶ Run

因为静态方法属于 `class` 而不属于实例，因此，静态方法内部，无法访问 `this` 变量，也无法访问实例字段，它只能访问静态字段。

通过实例变量也可以调用静态方法，但这只是编译器自动帮我们把实例改写成类名而已。

通常情况下，通过实例变量访问静态字段和静态方法，会得到一个编译警告。

静态方法经常用于工具类。例如：

- `Arrays.sort()`
- `Math.random()`

静态方法也经常用于辅助方法。注意到Java程序的入口 `main()` 也是静态方法。

接口的静态字段

因为 `interface` 是一个纯抽象类，所以它不能定义实例字段。但是， `interface` 是可以有静态字段的，并且静态字段必须为 `final` 类型：

```
1. public interface Person {
2.     public static final int MALE = 1;
3.     public static final int FEMALE = 2;
4. }
```

实际上，因为 `interface` 的字段只能是 `public static final` 类型，所以我们可以把这些修饰符都去掉，上述代码可以简写为：

```
1.  public interface Person {
2.      // 编译器会自动加上public statc final:
3.      int MALE = 1;
4.      int FEMALE = 2;
5.  }
```

编译器会自动把该字段变为 `public static final` 类型。

练习

给Person类增加一个静态字段count和静态方法getCount，统计实例创建的个数。

下载练习：静态字段和静态方法 （推荐使用IDE练习插件快速下载）

小结

- 静态字段属于所有实例“共享”的字段，实际上是属于 `class` 的字段；
- 调用静态方法不需要实例，无法访问 `this` ，但可以访问静态字段和其他静态方法；
- 静态方法常用于工具类和辅助方法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





包

在前面的代码中，我们把类和接口命名为 `Person` 、 `Student` 、 `Hello` 等简单名字。

在现实中，如果小明写了一个 `Person` 类，小红也写了一个 `Person` 类，现在，小白既想用小明的 `Person` ，也想用小红的 `Person` ，怎么办？

如果小军写了一个 `Arrays` 类，恰好JDK也自带了一个 `Arrays` 类，如何解决类名冲突？

在Java中，我们使用 `package` 来解决名字冲突。

Java定义了一种名字空间，称之为包： `package` 。一个类总是属于某个包，类名（比如 `Person` ）只是一个简写，真正的完整类名是 `包名.类名` 。

例如：

小明的 `Person` 类存放在包 `ming` 下面，因此，完整类名是 `ming.Person` ；

小红的 `Person` 类存放在包 `hong` 下面，因此，完整类名是 `hong.Person` ；

小军的 `Arrays` 类存放在包 `mr.jun` 下面，因此，完整类名是 `mr.jun.Arrays` ；

JDK的 `Arrays` 类存放在包 `java.util` 下面，因此，完整类名是 `java.util.Arrays` 。

在定义 `class` 的时候，我们需要在第一行声明这个 `class` 属于哪个包。

小明的 `Person.java` 文件：

```
1. package ming; // 申明包名ming
2.
3. public class Person {
4. }
```

小军的 `Arrays.java` 文件：

```
1. package mr.jun; // 申明包名mr.jun
2.
3. public class Arrays {
4. }
```

在Java虚拟机执行的时候，JVM只看完整类名，因此，只要包名不同，类就不同。

包可以是多层结构，用 `.` 隔开。例如： `java.util` 。

要特别注意：包没有父子关系。java.util和java.util.zip是不同的包，两者没有任何继承关系。

没有定义包名的 `class` ，它使用的是默认包，非常容易引起名字冲突，因此，不推荐不写包名的做法。

我们还需要按照包结构把上面的Java文件组织起来。假设以 `package_sample` 作为根目录， `src` 作为源码目录，那么所有文件结构就是：

```
package_sample
└─ src
    ├─ hong
    │  └─ Person.java
    │  ming
    │  └─ Person.java
    └─ mr
       └─ jun
          └─ Arrays.java
```

即所有Java文件对应的目录层次要和包的层次一致。

编译后的 `.class` 文件也需要按照包结构存放。如果使用IDE，把编译后的 `.class` 文件放到 `bin` 目录下，那么，编译的文件结构就是：

```
package_sample
└─ bin
   ├─ hong
   │  └─ Person.class
   │  ming
   │  └─ Person.class
   └─ mr
      └─ jun
         └─ Arrays.class
```

编译的命令相对比较复杂，我们需要在 `src` 目录下执行 `javac` 命令：

```
javac -d ../bin ming/Person.java hong/Person.java mr/jun/Arrays.java
```

在IDE中，会自动根据包结构编译所有Java源码，所以不必担心使用命令行编译的复杂命令。

包作用域

位于同一个包的类，可以访问包作用域的字段和方法。不用 `public` 、 `protected` 、 `private` 修饰的字段和方法就是包作用域。例如， `Person` 类定义在 `hello` 包下面：

```
package hello;

public class Person {
    // 包作用域:
    void hello() {
        System.out.println("Hello!");
    }
}
```

`Main` 类也定义在 `hello` 包下面:

```
package hello;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.hello(); // 可以调用, 因为Main和Person在同一个包
    }
}
```

import

在一个 `class` 中，我们总会引用其他的 `class` 。例如，小明的 `ming.Person` 类，如果要引用小军的 `mr.jun.Arrays` 类，他有三种写法:

第一种，直接写出完整类名，例如:

```
// Person.java
package ming;

public class Person {
    public void run() {
        mr.jun.Arrays arrays = new mr.jun.Arrays();
    }
}
```

很显然，每次写完整类名比较痛苦。

因此，第二种写法是用 `import` 语句，导入小军的 `Arrays` ，然后写简单类名:

```
// Person.java
```

```
package ming;

// 导入完整类名:
import mr.jun.Arrays;

public class Person {
    public void run() {
        Arrays arrays = new Arrays();
    }
}
```

在写 `import` 的时候，可以使用 `*` ，表示把这个包下面的所有 `class` 都导入进来（但不包括子包的 `class` ）：

```
// Person.java
package ming;

// 导入mr.jun包的所有class:
import mr.jun.*;

public class Person {
    public void run() {
        Arrays arrays = new Arrays();
    }
}
```

我们一般不推荐这种写法，因为在导入了多个包后，很难看出 `Arrays` 类属于哪个包。

还有一种 `import static` 的语法，它可以导入可以导入一个类的静态字段和静态方法：

```
package main;

// 导入System类的所有静态字段和静态方法:
import static java.lang.System.*;

public class Main {
    public static void main(String[] args) {
        // 相当于调用System.out.println(…)
        out.println("Hello, world!");
    }
}
```

`import static` 很少使用。

Java编译器最终编译出的 `.class` 文件只使用完整类名，因此，在代码中，当编译器遇到一个 `class` 名称时：

- 如果是完整类名，就直接根据完整类名查找这个 `class` ；
- 如果是简单类名，按下面的顺序依次查找：
  - 查找当前 `package` 是否存在这个 `class` ；
  - 查找 `import` 的包是否包含这个 `class` ；
  - 查找 `java.lang` 包是否包含这个 `class` 。

如果按照上面的规则还无法确定类名，则编译报错。

我们来看一个例子：

```
1.  // Main.java
2.  package test;
3.
4.  import java.text.Format;
5.
6.  public class Main {
7.      public static void main(String[] args) {
8.          java.util.List list; // ok, 使用完整类名 -> java.util.List
9.          Format format = null; // ok, 使用import的类 -> java.text.Format
10.         String s = "hi"; // ok, 使用java.lang包的String -> java.lang.String
            System.out.println(s); // ok, 使用java.lang包的System ->
11. java.lang.System
            MessageFormat mf = null; // 编译错误：无法找到MessageFormat: MessageFormat
12. cannot be resolved to a type
13.     }
14. }
```

因此，编写class的时候，编译器会自动帮我们做两个import动作：

- 默认自动 `import` 当前 `package` 的其他 `class` ；
- 默认自动 `import java.lang.*` 。

注意：自动导入的是java.lang包，但类似java.lang.reflect这些包仍需要手动导入。

如果有两个 `class` 名称相同，例如， `mr.jun.Arrays` 和 `java.util.Arrays` ，那么只

能 `import` 其中一个，另一个必须写完整类名。

最佳实践

为了避免名字冲突，我们需要确定唯一的包名。推荐的做法是使用倒置的域名来确保唯一性。例如：

- org.apache
- org.apache.commons.log
- com.liaoxuefeng.sample

子包就可以根据功能自行命名。

要注意不要和 `java.lang` 包的类重名，即自己的类不要使用这些名字：

- String
- System
- Runtime
- …

要注意也不要和JDK常用类重名：

- java.util.List
- java.text.Format
- java.math.BigInteger
- …

练习

请按如下包结构创建工程项目：

```
oop-package
└── src
    └── com
        └── itranswarp
            ├── sample
            │   └── Main.java
            └── world
                └── Person.java
```

下载练习：Package结构 （推荐使用IDE练习插件快速下载）

小结

Java内建的 `package` 机制是为了避免 `class` 命名冲突；

JDK的核心类使用 `java.lang` 包，编译器会自动导入；

JDK的其它常用类定义在 `java.util.` ， *`java.math.`* ， `java.text.*` ，……；

包名推荐使用倒置的域名，例如 `org.apache` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





作用域

在Java中，我们经常看到 `public` 、 `protected` 、 `private` 这些修饰符。在Java中，这些修饰符可以用来限定访问作用域。

public

定义为 `public` 的 `class` 、 `interface` 可以被其他任何类访问：

```
1. package abc;
2. 
3. public class Hello {
4.     public void hi() {
5.     }
6. }
```

上面的 `Hello` 是 `public` ，因此，可以被其他包的类访问：

```
1. package xyz;
2. 
3. class Main {
4.     void foo() {
5.         // Main可以访问Hello
6.         Hello h = new Hello();
7.     }
8. }
```

定义为 `public` 的 `field` 、 `method` 可以被其他类访问，前提是首先有访问 `class` 的权限：

```
1. package abc;
2. 
3. public class Hello {
4.     public void hi() {
5.     }
6. }
```

上面的 `hi()` 方法是 `public` ，可以被其他类调用，前提是首先要能访问 `Hello` 类：

```
1. package xyz;
2. 
3. class Main {
```

```
    void foo() {
        Hello h = new Hello();
        h.hi();
    }
}
```

private

定义为 `private` 的 `field` 、 `method` 无法被其他类访问：

```
package abc;

public class Hello {
    // 不能被其他类调用:
    private void hi() {
    }

    public void hello() {
        this.hi();
    }
}
```

实际上，确切地说， `private` 访问权限被限定在 `class` 的内部，而且与方法声明顺序无关。推荐把 `private` 方法放到后面，因为 `public` 方法定义了类对外提供的功能，阅读代码的时候，应该先关注 `public` 方法：

```
package abc;

public class Hello {
    public void hello() {
        this.hi();
    }

    private void hi() {
    }
}
```

由于Java支持嵌套类，如果一个类内部还定义了嵌套类，那么，嵌套类拥有访问 `private` 的权限：

```
// private
public class Main {
    public static void main(String[] args) {
        Inner i = new Inner();
        i.hi();
    }

    // private方法:
    private static void hello() {
        System.out.println("private hello!");
    }

    // 静态内部类:
    static class Inner {
        public void hi() {
            Main.hello();
        }
    }
}
```

▶ Run

定义在一个 `class` 内部的 `class` 称为嵌套类（ `nested class` ），Java支持好几种嵌套类。

protected

`protected` 作用于继承关系。定义为 `protected` 的字段和方法可以被子类访问，以及子类的子类：

```
1. package abc;
2.
3. public class Hello {
4.     // protected方法:
5.     protected void hi() {
6.     }
7. }
```

上面的 `protected` 方法可以被继承的类访问：

```
1. package xyz;
2.
3. class Main extends Hello {
4.     void foo() {
5.         Hello h = new Hello();
6.         // 可以访问protected方法:
7.         h.hi();
```

```
    }
}
```

package

最后，包作用域是指一个类允许访问同一个 `package` 的没有 `public` 、 `private` 修饰的 `class` ，以及没有 `public` 、 `protected` 、 `private` 修饰的字段和方法。

```
package abc;
// package权限的类:
class Hello {
    // package权限的方法:
    void hi() {
    }
}
```

只要在同一个包，就可以访问 `package` 权限的 `class` 、 `field` 和 `method` ：

```
package abc;

class Main {
    void foo() {
        // 可以访问package权限的类:
        Hello h = new Hello();
        // 可以调用package权限的方法:
        h.hi();
    }
}
```

注意，包名必须完全一致，包没有父子关系， `com.apache` 和 `com.apache.abc` 是不同的包。

局部变量

在方法内部定义的变量称为局部变量，局部变量作用域从变量声明处开始到对应的块结束。方法参数也是局部变量。

```
package abc;

public class Hello {
    void hi(String name) { // ①
        String s = name.toLowerCase(); // ②
```

```
        int len = s.length(); // ③
        if (len < 10) { // ④
            int p = 10 - len; // ⑤
            for (int i=0; i<10; i++) { // ⑥
                System.out.println(); // ⑦
            } // ⑧
        } // ⑨
    } // ⑩
}
```

我们观察上面的 `hi()` 方法代码：

- 方法参数name是局部变量，它的作用域是整个方法，即① ~ ⑩；
- 变量s的作用域是定义处到方法结束，即② ~ ⑩；
- 变量len的作用域是定义处到方法结束，即③ ~ ⑩；
- 变量p的作用域是定义处到if块结束，即⑤ ~ ⑨；
- 变量i的作用域是for循环，即⑥ ~ ⑧。

使用局部变量时，应该尽可能把局部变量的作用域缩小，尽可能延后声明局部变量。

final

Java还提供了一个 `final` 修饰符。 `final` 与访问权限不冲突，它有很多作用。

用 `final` 修饰 `class` 可以阻止被继承：

```
package abc;

// 无法被继承:
public final class Hello {
    private int n = 0;
    protected void hi(int t) {
        long i = t;
    }
}
```

用 `final` 修饰 `method` 可以阻止被子类覆写：

```
package abc;

```

```
public class Hello {
    // 无法被覆写:
    protected final void hi() {
    }
}
```

用 `final` 修饰 `field` 可以阻止被重新赋值：

```
package abc;

public class Hello {
    private final int n = 0;
    protected void hi() {
        this.n = 1; // error!
    }
}
```

用 `final` 修饰局部变量可以阻止被重新赋值：

```
package abc;

public class Hello {
    protected void hi(final int t) {
        t = 1; // error!
    }
}
```

最佳实践

如果不确定是否需要 `public` ，就不声明为 `public` ，即尽可能少地暴露对外的字段和方法。

把方法定义为 `package` 权限有助于测试，因为测试类和被测试类只要位于同一个 `package` ，测试代码就可以访问被测试类的 `package` 权限方法。

一个 `.java` 文件只能包含一个 `public` 类，但可以包含多个非 `public` 类。如果有 `public` 类，文件名必须和 `public` 类的名字相同。

小结

Java内建的访问权限包括 `public` 、 `protected` 、 `private` 和 `package` 权限；

Java在方法内部定义的变量是局部变量，局部变量的作用域从变量声明开始，到一个块结束；

`final` 修饰符不是访问权限，它可以修饰 `class` 、 `field` 和 `method` ；

一个 `.java` 文件只能包含一个 `public` 类，但可以包含多个非 `public` 类。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





classpath和jar

在Java中，我们经常听到 `classpath` 这个东西。网上有很多关于“如何设置classpath”的文章，但大部分设置都不靠谱。

到底什么是 `classpath` ？

`classpath` 是JVM用到的一个环境变量，它用来指示JVM如何搜索 `class` 。

因为Java是编译型语言，源码文件是 `.java` ，而编译后的 `.class` 文件才是真正可以被JVM执行的字节码。因此，JVM需要知道，如果要加载一个 `abc.xyz.Hello` 的类，应该去哪搜索对应的 `Hello.class` 文件。

所以， `classpath` 就是一组目录的集合，它设置的搜索路径与操作系统相关。例如，在Windows系统上，用 `;` 分隔，带空格的目录用 `""` 括起来，可能长这样：

```
C:\work\project1\bin;C:\shared;"D:\My Documents\project1\bin"
```

在Linux系统上，用 `:` 分隔，可能长这样：

```
/usr/shared:/usr/local/bin:/home/liaoxuefeng/bin
```

现在我们假设 `classpath` 是 `.;C:\work\project1\bin;C:\shared` ，当JVM在加载 `abc.xyz.Hello` 这个类时，会依次查找：

- <当前目录>\abc\xyz\Hello.class
- C:\work\project1\bin\abc\xyz\Hello.class
- C:\shared\abc\xyz\Hello.class

注意到 `.` 代表当前目录。如果JVM在某个路径下找到了对应的 `class` 文件，就不再往后继续搜索。如果所有路径下都没有找到，就报错。

`classpath` 的设定方法有两种：

在系统环境变量中设置 `classpath` 环境变量，不推荐；

在启动JVM时设置 `classpath` 变量，推荐。

我们强烈不推荐在系统环境变量中设置 `classpath` ，那样会污染整个系统环境。在启动JVM时设置 `classpath` 才是推荐的做法。实际上就是给 `java` 命令传入 `-classpath` 或 `-cp` 参数：

```
java -classpath .;C:\work\project1\bin;C:\shared abc.xyz.Hello
```

或者使用 `-cp` 的简写：

```
1.  java -cp .;C:\work\project1\bin;C:\shared abc.xyz.Hello
```

没有设置系统环境变量，也没有传入 `-cp` 参数，那么JVM默认的 `classpath` 为 `.` ，即当前目录：

```
1.  java abc.xyz.Hello
```

上述命令告诉JVM只在当前目录搜索 `Hello.class` 。

在IDE中运行Java程序，IDE自动传入的 `-cp` 参数是当前工程的 `bin` 目录和引入的jar包。

通常，我们在自己编写的 `class` 中，会引用Java核心库的 `class` ，例如， `String` 、 `ArrayList` 等。这些 `class` 应该上哪去找？

有很多“如何设置classpath”的文章会告诉你把JVM自带的 `rt.jar` 放入 `classpath` ，但事实上，根本不需要告诉JVM如何去Java核心库查找 `class` ，JVM怎么可能笨到连自己的核心库在哪都不知道？

不要把任何Java核心库添加到classpath中！JVM根本不依赖classpath加载核心库！

更好的做法是，不要设置 `classpath` ！默认的当前目录 `.` 对于绝大多数情况都够用了。

jar包

如果有很多 `.class` 文件，散落在各层目录中，肯定不便于管理。如果能把目录打一个包，变成一个文件，就方便多了。

jar包就是用来干这个事的，它可以把 `package` 组织的目录层级，以及各个目录下的所有文件（包括 `.class` 文件和其他文件）都打成一个jar文件，这样一来，无论是备份，还是发给客户，就简单多了。

jar包实际上就是一个zip格式的压缩文件，而jar包相当于目录。如果我们要执行一个jar包的 `class` ，就可以把jar包放到 `classpath` 中：

```
1.  java -cp ./hello.jar abc.xyz.Hello
```

这样JVM会自动在 `hello.jar` 文件里去搜索某个类。

那么问题来了：如何创建jar包？

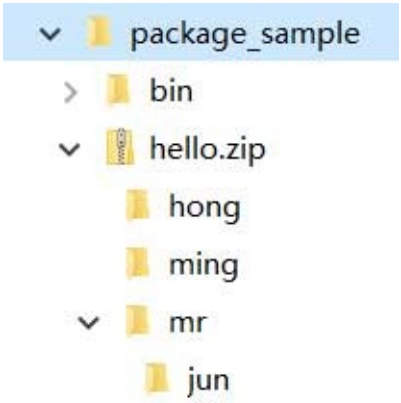
因为jar包就是zip包，所以，直接在资源管理器中，找到正确的目录，点击右键，在弹出的快捷菜单

中选择“发送到”，“压缩(zipped)文件夹”，就制作了一个zip文件。然后，把后缀从 `.zip` 改为 `.jar` ，一个jar包就创建成功。

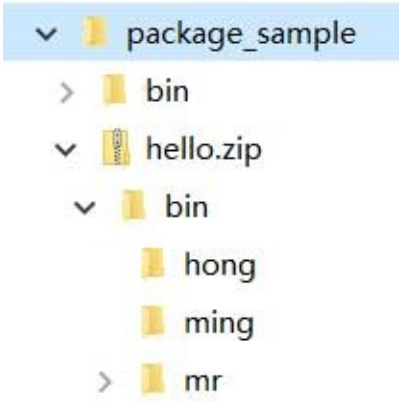
假设编译输出的目录结构是这样：

```
package_sample
└─ bin
   ├─ hong
   │  └─ Person.class
   │  ming
   │  └─ Person.class
   └─ mr
      └─ jun
         └─ Arrays.class
```

这里需要特别注意的是，jar包里的第一层目录，不能是 `bin` ，而应该是 `hong` 、 `ming` 、 `mr` 。如果在Windows的资源管理器中看，应该长这样：



如果长这样：



说明打包打得有问题，JVM仍然无法从jar包中查找正确的 `class` ，原因是 `hong.Person` 必须按 `hong/Person.class` 存放，而不是 `bin/hong/Person.class` 。

jar包还可以包含一个特殊的 `/META-INF/MANIFEST.MF` 文件， `MANIFEST.MF` 是纯文本，可以指定 `Main-Class` 和其它信息。JVM会自动读取这个 `MANIFEST.MF` 文件，如果存在 `Main-Class` ，我们就不必在命令行指定启动的类名，而是用更方便的命令：

```
1.  java -jar hello.jar
```

jar包还可以包含其它jar包，这个时候，就需要在 `MANIFEST.MF` 文件里配置 `classpath` 了。

在大型项目中，不可能手动编写 `MANIFEST.MF` 文件，再手动创建zip包。Java社区提供了大量的开源构建工具，例如Maven，可以非常方便地创建jar包。

小结

JVM通过环境变量 `classpath` 决定搜索 `class` 的路径和顺序；

不推荐设置系统环境变量 `classpath` ，始终建议通过 `-cp` 命令传入；

jar包相当于目录，可以包含很多 `.class` 文件，方便下载和使用；

`MANIFEST.MF` 文件可以提供jar包的信息，如 `Main-Class` ，这样可以直接运行jar包。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





模块

从Java 9开始，JDK又引入了模块（Module）。

什么是模块？这要从Java 9之前的版本说起。

我们知道，`.class`文件是JVM看到的最小可执行文件，而一个大型程序需要编写很多Class，并生成一堆`.class`文件，很不便于管理，所以，`jar`文件就是`class`文件的容器。

在Java 9之前，一个大型Java程序会生成自己的jar文件，同时引用依赖的第三方jar文件，而JVM自带的Java标准库，实际上也是以jar文件形式存放的，这个文件叫`rt.jar`，一共有60多M。

如果是自己开发的程序，除了一个自己的`app.jar`以外，还需要一堆第三方的jar包，运行一个Java程序，一般来说，命令行写这样：

```
1. java -cp app.jar:a.jar:b.jar:c.jar com.liaoxuefeng.sample.Main
```

注意：JVM自带的标准库rt.jar不要写到classpath中，写了反而会干扰JVM的正常运行。

如果漏写了某个运行时需要用到的jar，那么在运行期极有可能抛出`ClassNotFoundException`。

所以，jar只是用于存放class的容器，它并不关心class之间的依赖。

从Java 9开始引入的模块，主要是为了解决“依赖”这个问题。如果`a.jar`必须依赖另一个`b.jar`才能运行，那我们应该给`a.jar`加点说明啥的，让程序在编译和运行的时候能自动定位到`b.jar`，这种自带“依赖关系”的class容器就是模块。

为了表明Java模块化的决心，从Java 9开始，原有的Java标准库已经由一个单一巨大的`rt.jar`分拆成了几十个模块，这些模块以`.jmod`扩展名标识，可以在`$JAVA_HOME/jmods`目录下找到它们：

- java.base.jmod
- java.compiler.jmod
- java.datatransfer.jmod
- java.desktop.jmod
- …

这些`.jmod`文件每一个都是一个模块，模块名就是文件名。例如：模块`java.base`对应的文件就是`java.base.jmod`。模块之间的依赖关系已经被写入到模块内的`module-info.class`文件了。所有的模块都直接或间接地依赖`java.base`模块，只有`java.base`模块不依赖任何模块，它可以被看作是“根模块”，好比所有的类都是从`Object`直接或间接继承而来。

把一堆class封装为jar仅仅是一个打包的过程，而把一堆class封装为模块则不但需要打包，还需要

写入依赖关系，并且还可以包含二进制代码（通常是JNI扩展）。此外，模块支持多版本，即在同一个模块中可以为不同的JVM提供不同的版本。

编写模块

那么，我们应该如何编写模块呢？还是以具体的例子来说。首先，创建模块和原有的创建Java项目是完全一样的，以 `oop-module` 工程为例，它的目录结构如下：

```
oop-module
├── bin
├── build.sh
└── src
    ├── com
    │   └── itranswarp
    │       └── sample
    │           ├── Greeting.java
    │           └── Main.java
    └── module-info.java
```

其中， `bin` 目录存放编译后的class文件， `src` 目录存放源码，按包名的目录结构存放，仅仅在 `src` 目录下多了一个 `module-info.java` 这个文件，这就是模块的描述文件。在这个模块中，它长这样：

```
module hello.world {
    requires java.base; // 可不写，任何模块都会自动引入java.base
    requires java.xml;
}
```

其中， `module` 是关键字，后面的 `hello.world` 是模块的名称，它的命名规范与包一致。花括号的 `requires xxx;` 表示这个模块需要引用的其他模块名。除了 `java.base` 可以被自动引入外，这里我们引入了一个 `java.xml` 的模块。

当我们使用模块声明了依赖关系后，才能使用引入的模块。例如， `Main.java` 代码如下：

```
package com.itranswarp.sample;

// 必须引入java.xml模块后才能使用其中的类:
import javax.xml.XMLConstants;

public class Main {
    public static void main(String[] args) {
        Greeting g = new Greeting();
```

```
        System.out.println(g.hello(XMLConstants.XML_NS_PREFIX));
    }
}
```

如果把 `requires java.xml;` 从 `module-info.java` 中去掉，编译将报错。可见，模块的重要作用就是声明依赖关系。

下面，我们用JDK提供的命令行工具来编译并创建模块。

首先，我们把工作目录切换到 `oop-module` ，在当前目录下编译所有的 `.java` 文件，并存放到 `bin` 目录下，命令如下：

```
$ javac -d bin src/module-info.java src/com/itranswarp/sample/*.java
```

如果编译成功，现在项目结构如下：

```
oop-module
├── bin
│   ├── com
│   │   └── itranswarp
│   │       └── sample
│   │           ├── Greeting.class
│   │           └── Main.class
│   └── module-info.class
└── src
    ├── com
    │   └── itranswarp
    │       └── sample
    │           ├── Greeting.java
    │           └── Main.java
    └── module-info.java
```

注意到 `src` 目录下的 `module-info.java` 被编译到 `bin` 目录下的 `module-info.class` 。

下一步，我们需要把bin目录下的所有class文件先打包成jar，在打包的时候，注意传入 `–main-class` 参数，让这个jar包能自己定位 `main` 方法所在的类：

```
$ jar --create --file hello.jar --main-class com.itranswarp.sample.Main -C bin
.
```

现在我们就在当前目录下得到了 `hello.jar` 这个jar包，它和普通jar包并无区别，可以直接使用命令 `java -jar hello.jar` 来运行它。但是我们的目标是创建模块，所以，继续使用JDK自带

的 `jmod` 命令把一个jar包转换成模块：

```
$ jmod create --class-path hello.jar hello.jmod
```

于是，在当前目录下我们又得到了 `hello.jmod` 这个模块文件，这就是最后打包出来的传说中的模块！

运行模块

要运行一个jar，我们使用 `java -jar xxx.jar` 命令。要运行一个模块，我们只需要指定模块名。试试：

```
$ java --module-path hello.jmod --module hello.world
```

结果是一个错误：

```
Error occurred during initialization of boot layer
java.lang.module.FindException: JMOD format not supported at execution time:
hello.jmod
```

原因是 `.jmod` 不能被放入 `—module-path` 中。换成 `.jar` 就没问题了：

```
$ java --module-path hello.jar --module hello.world
Hello, xml!
```

那我们辛辛苦苦创建的 `hello.jmod` 有什么用？答案是我们可以用它来打包JRE。

打包JRE

前面讲了，为了支持模块化，Java 9首先带头把自己的一个巨大无比的 `rt.jar` 拆成了几十个 `.jmod` 模块，原因就是，运行Java程序的时候，实际上我们用到的JDK模块，并没有那么多。不需要的模块，完全可以删除。

过去发布一个Java应用程序，要运行它，必须下载一个完整的JRE，再运行jar包。而完整的JRE块头很大，有100多M。怎么给JRE瘦身呢？

现在，JRE自身的标准库已经分拆成了模块，只需要带上程序用到的模块，其他的模块就可以被裁剪掉。怎么裁剪JRE呢？并不是说把系统安装的JRE给删掉部分模块，而是“复制”一份JRE，但只带上用到的模块。为此，JDK提供了 `jlink` 命令来干这件事。命令如下：

```
$ jlink --module-path hello.jmod --add-modules java.base,java.xml,hello.world -
-output jre/
```

我们在 `–module-path` 参数指定了我们自己的模块 `hello.jmod` ，然后，在 `–add-modules` 参数中指定了我们用到的3个模块 `java.base` 、 `java.xml` 和 `hello.world` ，用 `,` 分隔。最后，在 `–output` 参数指定输出目录。

现在，在当前目录下，我们可以找到 `jre` 目录，这是一个完整的并且带有我们自己 `hello.jmod` 模块的JRE。试试直接运行这个JRE：

```
$ jre/bin/java --module hello.world
Hello, xml!
```

要分发我们自己的Java应用程序，只需要把这个 `jre` 目录打个包给对方发过去，对方直接运行上述命令即可，既不用下载安装JDK，也不用知道如何配置我们自己的模块，极大地方便了分发和部署。

访问权限

前面我们讲过，Java的class访问权限分为public、protected、private和默认的包访问权限。引入模块后，这些访问权限的规则就要稍微做些调整。

确切地说，class的这些访问权限只在一个模块内有效，模块和模块之间，例如，a模块要访问b模块的某个class，必要条件是b模块明确地导出了可以访问的包。

举个例子：我们编写的模块 `hello.world` 用到了模块 `java.xml` 的一个类 `javax.xml.XMLConstants` ，我们之所以能直接使用这个类，是因为模块 `java.xml` 的 `module-info.java` 中声明了若干导出：

```
module java.xml {
    exports java.xml;
    exports javax.xml.catalog;
    exports javax.xml.datatype;
    ...
}
```

只有它声明的导出的包，外部代码才被允许访问。换句话说，如果外部代码想要访问我们的 `hello.world` 模块中的 `com.itranswarp.sample.Greeting` 类，我们必须将其导出：

```
module hello.world {
    exports com.itranswarp.sample;

    requires java.base;
    requires java.xml;
}
```

因此，模块进一步隔离了代码的访问权限。

练习

请下载并练习如何打包模块和JRE。

下载练习：打包模块和JRE （推荐使用IDE练习插件快速下载）

小结

Java 9引入的模块目的是为了管理依赖；

使用模块可以按需打包JRE；

使用模块对类的访问权限有了进一步限制。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java核心类

本节我们将介绍Java的核心类，包括：

- 字符串
- StringBuilder
- StringJoiner
- 包装类型
- JavaBean
- 枚举
- 常用工具类



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论

字符串和编码

String

在Java中，`String`是一个引用类型，它本身也是一个`class`。但是，Java编译器对`String`有特殊处理，即可以直接用`"…"`来表示一个字符串：

```
1.  String s1 = "Hello!";
```

实际上字符串在`String`内部是通过一个`char[]`数组表示的，因此，按下面的写法也是可以的：

```
1.  String s2 = new String(new char[] {'H', 'e', 'l', 'l', 'o', '!'});
```

因为`String`太常用了，所以Java提供了`"…"`这种字符串字面量表示方法。

Java字符串的一个重要特点就是字符串不可变。这种不可变性是通过内部的`private final char[]`字段，以及没有任何修改`char[]`的方法实现的。

我们来看一个例子：

```
// String
public class Main {
    public static void main(String[] args) {
        String s = "Hello";
        System.out.println(s);
        s = s.toUpperCase();
        System.out.println(s);
    }
}
```

▶ Run

根据上面代码的输出，试解释字符串内容是否改变。

字符串比较

当我们想要比较两个字符串是否相同时，要特别注意，我们实际上是想比较字符串的内容是否相同。必须使用`equals()`方法而不能用`==`。

我们看下面的例子：

```
// String
public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "hello";
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
    }
}
```

▶ Run

从表面上看，两个字符串用 `==` 和 `equals()` 比较都为 `true` ，但实际上那只是Java编译器在编译期，会自动把所有相同的字符串当作一个对象放入常量池，自然 `s1` 和 `s2` 的引用就是相同的。

所以，这种 `==` 比较返回 `true` 纯属巧合。换一种写法， `==` 比较就会失败：

```
// String
public class Main {
    public static void main(String[] args) {
        String s1 = "hello";
        String s2 = "HELLO".toLowerCase();
        System.out.println(s1 == s2);
        System.out.println(s1.equals(s2));
    }
}
```

▶ Run

结论：两个字符串比较，必须总是使用 `equals()` 方法。

要忽略大小写比较，使用 `equalsIgnoreCase()` 方法。

`String` 类还提供了多种方法来搜索子串、提取子串。常用的方法有：

```
// 是否包含子串:
"Hello".contains("ll"); // true
```

注意到 `contains()` 方法的参数是 `CharSequence` 而不是 `String` ，因为 `CharSequence` 是 `String` 的父类。

搜索子串的更多的例子：

```
"Hello".indexOf("l"); // 2
"Hello".lastIndexOf("l"); // 3
```

```
3. "Hello".startsWith("He"); // true
4. "Hello".endsWith("lo"); // true
```

提取子串的例子：

```
1. "Hello".substring(2); // "llo"
2. "Hello".substring(2, 4); "ll"
```

注意索引号是从 `0` 开始的。

去除首尾空白字符

使用 `trim()` 方法可以移除字符串首尾空白字符。空白字符包括空格， `\t` ， `\r` ， `\n` ：

```
1. "  \tHello\r\n ".trim(); // "Hello"
```

注意： `trim()` 并没有改变字符串的内容，而是返回了一个新字符串。

另一个 `strip()` 方法也可以移除字符串首尾空白字符。它和 `trim()` 不同的是，类似中文的空格字符 `\u3000` 也会被移除：

```
1. "\u3000Hello\u3000".strip(); // "Hello"
2. " Hello ".stripLeading(); // "Hello "
3. " Hello ".stripTrailing(); // " Hello"
```

`String` 还提供了 `isEmpty()` 和 `isBlank()` 来判断字符串是否为空和空白字符串：

```
1. "".isEmpty(); // true，因为字符串长度为0
2. "  ".isEmpty(); // false，因为字符串长度不为0
3. "  \n".isBlank(); // true，因为只包含空白字符
4. " Hello ".isBlank(); // false，因为包含非空白字符
```

替换子串

要在字符串中替换子串，有两种方法。一种是根据字符或字符串替换：

```
1. String s = "hello";
2. s.replace('l', 'w'); // "hewwo"，所有字符'l'被替换为'w'
3. s.replace("ll", "~~"); // "he~~o"，所有子串"ll"被替换为"~~"
```

另一种是通过正则表达式替换：

```
String s = "A,,B;C ,D";
s.replaceAll("[\\,\\;\\s]+", ","); // "A,B,C,D"
```

上面的代码通过正则表达式，把匹配的子串统一替换为 `","` 。关于正则表达式的用法我们会在后面详细讲解。

分割字符串

要分割字符串，使用 `split()` 方法，并且传入的也是正则表达式：

```
String s = "A,B,C,D";
String[] ss = s.split("\\,"); // {"A", "B", "C", "D"}
```

拼接字符串

拼接字符串使用静态方法 `join()` ，它用指定的字符串连接字符串数组：

```
String[] arr = {"A", "B", "C"};
String s = String.join("***", arr); // "A***B***C"
```

类型转换

要把任意基本类型或引用类型转换为字符串，可以使用静态方法 `valueOf()` 。这是一个重载方法，编译器会根据参数自动选择合适的方法：

```
String.valueOf(123); // "123"
String.valueOf(45.67); // "45.67"
String.valueOf(true); // "true"
String.valueOf(new Object()); // 类似java.lang.Object@636be97c
```

要把字符串转换为其他类型，就需要根据情况。例如，把字符串转换为 `int` 类型：

```
int n1 = Integer.parseInt("123"); // 123
int n2 = Integer.parseInt("ff", 16); // 按十六进制转换，255
```

把字符串转换为 `boolean` 类型：

```
boolean b1 = Boolean.parseBoolean("true"); // true
boolean b2 = Boolean.parseBoolean("FALSE"); // false
```

要特别注意，`Integer` 有个 `getInteger(String)` 方法，它不是将字符串转换为 `int` ，而是把该字符串对应的系统变量转换为 `Integer` ：

```
1. Integer.getInteger("java.version"); // 版本号, 11
```

转换为char[]

`String` 和 `char[]` 类型可以互相转换，方法是：

```
1. char[] cs = "Hello".toCharArray(); // String -> char[]
2. String s = new String(cs); // char[] -> String
```

如果修改了 `char[]` 数组， `String` 并不会改变：

```
// String <-> char[]
public class Main {
    public static void main(String[] args) {
        char[] cs = "Hello".toCharArray();
        String s = new String(cs);
        System.out.println(s);
        cs[0] = 'X';
        System.out.println(s);
    }
}
```

▶ Run

这是因为通过 `new String(char[])` 创建新的 `String` 实例时，它并不会直接引用传入的 `char[]` 数组，而是会复制一份，所以，修改外部的 `char[]` 数组不会影响 `String` 实例内部的 `char[]` 数组，因为这是两个不同的数组。

从 `String` 的不变性设计可以看出，如果传入的对象有可能改变，我们需要复制而不是直接引用。

例如，下面的代码设计了一个 `Score` 类保存一组学生的成绩：

```
// int[]
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] scores = new int[] { 88, 77, 51, 66 };
        Score s = new Score(scores);
        s.printScores();
        scores[2] = 99;
        s.printScores();
    }
}

class Score {
    private int[] scores;
    public Score(int[] scores) {
        this.scores = scores;
    }

    public void printScores() {
        System.out.println(Arrays.toString(scores));
    }
}
```

▶ Run

观察两次输出，由于 `Score` 内部直接引用了外部传入的 `int[]` 数组，这会造成外部代码对 `int[]` 数组的修改，影响到 `Score` 类的字段。如果外部代码不可信，这就会造成安全隐患。

请修复 `Score` 的构造方法，使得外部代码对数组的修改不影响 `Score` 实例的 `int[]` 字段。

字符编码

在早期的计算机系统中，为了给字符编码，美国国家标准学会（American National Standard Institute: ANSI）制定了一套英文字母、数字和常用符号的编码，它占用一个字节，编码范围从 `0` 到 `127`，最高位始终为 `0`，称为 `ASCII` 编码。例如，字符 `'A'` 的编码是 `0x41`，字符 `'1'` 的编码是 `0x31`。

如果要把汉字也纳入计算机编码，很显然一个字节是不够的。`GB2312` 标准使用两个字节表示一个汉字，其中第一个字节的最高位始终为 `1`，以便和 `ASCII` 编码区分开。例如，汉字 `'中'` 的 `GB2312` 编码是 `0xd6d0`。

类似的，日文有 `Shift_JIS` 编码，韩文有 `EUC-KR` 编码，这些编码因为标准不统一，同时使用，就会产生冲突。

为了统一全球所有语言的编码，全球统一码联盟发布了 `Unicode` 编码，它把世界上主要语言都纳入同一个编码，这样，中文、日文、韩文和其他语言就不会冲突。

`Unicode` 编码需要两个或者更多字节表示，我们可以比较中英文字符

在 `ASCII` 、 `GB2312` 和 `Unicode` 的编码：

英文字符 `'A'` 的 `ASCII` 编码和 `Unicode` 编码：

```
          ┌────┐
ASCII:    | 41 |
          └────┘

          ┌────┬────┐
Unicode:  | 00 | 41 |
          └────┴────┘
```

英文字符的 `Unicode` 编码就是简单地在前面添加一个 `00` 字节。

中文字符 `'中'` 的 `GB2312` 编码和 `Unicode` 编码：

```
          ┌────┬────┐
GB2312:   | d6 | d0 |
          └────┴────┘

          ┌────┬────┐
Unicode:  | 4e | 2d |
          └────┴────┘
```

那我们经常使用的 `UTF-8` 又是什么编码呢？因为英文字符的 `Unicode` 编码高字节总是 `00` ，包含大量英文的文本会浪费空间，所以，出现了 `UTF-8` 编码，它是一种变长编码，用来把固定长度的 `Unicode` 编码变成1～4字节的变长编码。通过 `UTF-8` 编码，英文字符 `'A'` 的 `UTF-8` 编码变为 `0x41` ，正好和 `ASCII` 码一致，而中文 `'中'` 的 `UTF-8` 编码为3字节 `0xe4b8ad` 。

`UTF-8` 编码的另一个好处是容错能力强。如果传输过程中某些字符出错，不会影响后续字符，因为 `UTF-8` 编码依靠高字节位来确定一个字符究竟是几个字节，它经常用来作为传输编码。

在Java中， `char` 类型实际上就是两个字节的 `Unicode` 编码。如果我们要手动把字符串转换成其他编码，可以这样做：

```
byte[] b1 = "Hello".getBytes(); // 按ISO8859-1编码转换，不推荐
byte[] b2 = "Hello".getBytes("UTF-8"); // 按UTF-8编码转换
byte[] b2 = "Hello".getBytes("GBK"); // 按GBK编码转换
byte[] b3 = "Hello".getBytes(StandardCharsets.UTF_8); // 按UTF-8编码转换
```

注意：转换编码后，就不再是 `char` 类型，而是 `byte` 类型表示的数组。

如果要把已知编码的 `byte[]` 转换为 `String` ，可以这样做：

```
byte[] b = ...
String s1 = new String(b, "GBK"); // 按GBK转换
String s2 = new String(b, StandardCharsets.UTF_8); // 按UTF-8转换
```

始终牢记：Java的 `String` 和 `char` 在内存中总是以Unicode编码表示。

延伸阅读

对于不同版本的JDK， `String` 类在内存中有不同的优化方式。具体来说，早期JDK版本的 `String` 总是以 `char[]` 存储，它的定义如下：

```
public final class String {
    private final char[] value;
    private final int offset;
    private final int count;
}
```

而较新的JDK版本的 `String` 则以 `byte[]` 存储：如果 `String` 仅包含ASCII字符，则每个 `byte` 存储一个字符，否则，每两个 `byte` 存储一个字符，这样做的目的是为了节省内存，因为大量的长度较短的 `String` 通常仅包含ASCII字符：

```
public final class String {
    private final byte[] value;
    private final byte coder; // 0 = LATIN1, 1 = UTF16
```

对于使用者来说， `String` 内部的优化不影响任何已有代码，因为它的 `public` 方法签名是不变的。

小结

- Java字符串 `String` 是不可变对象；
- 字符串操作不改变原字符串内容，而是返回新字符串；
- 常用的字符串操作：提取子串、查找、替换、大小写转换等；
- Java使用Unicode编码表示 `String` 和 `char` ；
- 转换编码就是将 `String` 和 `byte[]` 转换，需要指定编码；
- 转换为 `byte[]` 时，始终优先考虑 `UTF-8` 编码。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





StringBuilder

Java编译器对 `String` 做了特殊处理，使得我们可以直接用 `+` 拼接字符串。

考察下面的循环代码：

```
String s = "";
for (int i = 0; i < 1000; i++) {
    s = s + "," + i;
}
```

虽然可以直接拼接字符串，但是，在循环中，每次循环都会创建新的字符串对象，然后扔掉旧的字符串。这样，绝大部分字符串都是临时对象，不但浪费内存，还会影响GC效率。

为了能高效拼接字符串，Java标准库提供了 `StringBuilder` ，它是一个可变对象，可以预分配缓冲区，这样，往 `StringBuilder` 中新增字符时，不会创建新的临时对象：

```
StringBuilder sb = new StringBuilder(1024);
for (int i = 0; i < 1000; i++) {
    sb.append(',');
    sb.append(i);
}
String s = sb.toString();
```

`StringBuilder` 还可以进行链式操作：

```
// 链式操作
public class Main {
    public static void main(String[] args) {
        var sb = new StringBuilder(1024);
        sb.append("Mr ")
          .append("Bob")
          .append("!")
          .insert(0, "Hello, ");
        System.out.println(sb.toString());
    }
}
```

▶ Run

如果我们查看 `StringBuilder` 的源码，可以发现，进行链式操作的关键是，定义的 `append()` 方法会返回 `this` ，这样，就可以不断调用自身的其他方法。

仿照 `StringBuilder` ，我们也可以设计支持链式操作的类。例如，一个可以不断增加的计数器：

```
// 链式操作
public class Main {
    public static void main(String[] args) {
        Adder adder = new Adder();
        adder.add(3)
             .add(5)
             .inc()
             .add(10);
        System.out.println(adder.value());
    }
}

class Adder {
    private int sum = 0;

    public Adder add(int n) {
        sum += n;
        return this;
    }

    public Adder inc() {
        sum ++;
        return this;
    }

    public int value() {
        return sum;
    }
}
```

▶ Run

注意：对于普通的字符串 `+` 操作，并不需要我们将其改写为 `StringBuilder` ，因为Java编译器在编译时就自动把多个连续的 `+` 操作编码为 `StringConcatFactory` 的操作。在运行期， `StringConcatFactory` 会自动把字符串连接操作优化为数组复制或者 `StringBuilder` 操作。

你可能还听说过 `StringBuffer` ，这是Java早期的一个 `StringBuilder` 的线程安全版本，它通过同步来保证多个线程操作 `StringBuffer` 也是安全的，但是同步会带来执行速度的下降。

`StringBuilder` 和 `StringBuffer` 接口完全相同，现在完全没有必要使用 `StringBuffer` 。

练习

请使用 `StringBuilder` 构造一个 `INSERT` 语句：

```
public class Main {
    public static void main(String[] args) {
        String[] fields = { "name", "position", "salary" };
        String table = "employee";
        String insert = buildInsertSql(table, fields);
        System.out.println(insert);
        String s = "INSERT INTO employee (name, position, salary) VALUES (?, ?, ?)";
        System.out.println(s.equals(insert) ? "测试成功" : "测试失败");
    }
    static String buildInsertSql(String table, String[] fields) {
        // TODO:
        return "";
    }
}
```

▶ Run

下载练习：StringBuilder练习 （推荐使用IDE练习插件快速下载）

小结

`StringBuilder`是可变对象，用来高效拼接字符串；

`StringBuilder`可以支持链式操作，实现链式操作的关键是返回实例本身；

`StringBuffer`是`StringBuilder`的线程安全版本，现在很少使用。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





StringJoiner

要高效拼接字符串，应该使用 `StringBuilder` 。

很多时候，我们拼接的字符串像这样：

```
// Hello Bob, Alice, Grace!
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sb = new StringBuilder();
        sb.append("Hello ");
        for (String name : names) {
            sb.append(name).append(", ");
        }
        // 注意去掉最后的", ":
        sb.delete(sb.length() - 2, sb.length());
        sb.append("!");
        System.out.println(sb.toString());
    }
}
```

▶ Run

类似用分隔符拼接数组的需求很常见，所以Java标准库还提供了一个 `StringJoiner` 来干这个事：

```
import java.util.StringJoiner;
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sj = new StringJoiner(", ");
        for (String name : names) {
            sj.add(name);
        }
        System.out.println(sj.toString());
    }
}
```

▶ Run

慢着！用 `StringJoiner` 的结果少了前面的 `"Hello "` 和结尾的 `"!"` ！遇到这种情况，需要给 `StringJoiner` 指定“开头”和“结尾”：

```
import java.util.StringJoiner;
public class Main {
    public static void main(String[] args) {
        String[] names = {"Bob", "Alice", "Grace"};
        var sj = new StringJoiner(", ", "Hello ", "!");
        for (String name : names) {
            sj.add(name);
        }
        System.out.println(sj.toString());
    }
}
```

▶ Run

那么 `StringJoiner` 内部是如何拼接字符串的呢？如果查看源码，可以发现， `StringJoiner` 内部实际上就是使用了 `StringBuilder` ，所以拼接效率和 `StringBuilder` 几乎是一模一样的。

String.join()

`String` 还提供了一个静态方法 `join()` ，这个方法在内部使用了 `StringJoiner` 来拼接字符串，在不需要指定“开头”和“结尾”的时候，用 `String.join()` 更方便：

```
String[] names = {"Bob", "Alice", "Grace"};
var s = String.join(", ", names);
```

练习

请使用 `StringJoiner` 构造一个 `SELECT` 语句：

```
import java.util.StringJoiner;

public class Main {
    public static void main(String[] args) {
        String[] fields = { "name", "position", "salary" };
        String table = "employee";
        String select = buildSelectSql(table, fields);
        System.out.println(select);
        System.out.println("SELECT name, position, salary FROM employee".equals(select) ? "测试成功" : "测试失败");
    }
```

```
    static String buildSelectSql(String table, String[] fields) {
        // TODO:
        return "";
    }
```

```
}
```

▶ Run

下载练习：StringJoiner练习 （推荐使用IDE练习插件快速下载）

小结

用指定分隔符拼接字符串数组时，使用 `StringJoiner` 或者 `String.join()` 更方便；

用 `StringJoiner` 拼接字符串时，还可以额外附加一个“开头”和“结尾”。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





包装类型

我们已经知道，Java的数据类型分两种：

- 基本类型：`byte`，`short`，`int`，`long`，`boolean`，`float`，`double`，`char`
- 引用类型：所有`class`和`interface`类型

引用类型可以赋值为`null`，表示空，但基本类型不能赋值为`null`：

```
String s = null;
int n = null; // compile error!
```

那么，如何把一个基本类型视为对象（引用类型）？

比如，想要把`int`基本类型变成一个引用类型，我们可以定义一个`Integer`类，它只包含一个实例字段`int`，这样，`Integer`类就可以视为`int`的包装类（Wrapper Class）：

```
public class Integer {
    private int value;

    public Integer(int value) {
        this.value = value;
    }

    public int intValue() {
        return this.value;
    }
}
```

定义好了`Integer`类，我们就可以把`int`和`Integer`互相转换：

```
Integer n = null;
Integer n2 = new Integer(99);
int n3 = n2.intValue();
```

实际上，因为包装类型非常有用，Java核心库为每种基本类型都提供了对应的包装类型：

基本类型	对应的引用类型
boolean	java.lang.Boolean
byte	java.lang.Byte

short	java.lang.Short
int	java.lang.Integer
long	java.lang.Long
float	java.lang.Float
double	java.lang.Double
char	java.lang.Character

我们可以直接使用，并不需要自己去定义：

```
// Integer:
public class Main {
    public static void main(String[] args) {
        int i = 100;
        // 通过new操作符创建Integer实例(不推荐使用,会有编译警告):
        Integer n1 = new Integer(i);
        // 通过静态方法valueOf(int)创建Integer实例:
        Integer n2 = Integer.valueOf(i);
        // 通过静态方法valueOf(String)创建Integer实例:
        Integer n3 = Integer.valueOf("100");
        System.out.println(n3.intValue());
    }
}
```

▶ Run

Auto Boxing

因为 `int` 和 `Integer` 可以互相转换：

```
int i = 100;
Integer n = Integer.valueOf(i);
int x = n.intValue();
```

所以，Java编译器可以帮助我们自动在 `int` 和 `Integer` 之间转型：

```
Integer n = 100; // 编译器自动使用Integer.valueOf(int)
int x = n; // 编译器自动使用Integer.intValue()
```

这种直接把 `int` 变为 `Integer` 的赋值写法，称为自动装箱（Auto Boxing），反过来，把 `Integer` 变为 `int` 的赋值写法，称为自动拆箱（Auto Unboxing）。

注意：自动装箱和自动拆箱只发生在编译阶段，目的是为了少写代码。

装箱和拆箱会影响代码的执行效率，因为编译后的 `class` 代码是严格区分基本类型和引用类型的。并

且，自动拆箱执行时可能会报 `NullPointerException` ：

```
// NullPointerException
public class Main {
    public static void main(String[] args) {
        Integer n = null;
        int i = n;
    }
}
```

▶ Run

不变类

所有的包装类型都是不变类。我们查看 `Integer` 的源码可知，它的核心代码如下：

```
1.  public final class Integer {
2.      private final int value;
3.  }
```

因此，一旦创建了 `Integer` 对象，该对象就是不变的。

对两个 `Integer` 实例进行比较要特别注意：绝对不能用 `==` 比较，因为 `Integer` 是引用类型，必须使用 `equals()` 比较：

```
// == or equals?
public class Main {
    public static void main(String[] args) {
        Integer x = 127;
        Integer y = 127;
        Integer m = 99999;
        Integer n = 99999;
        System.out.println("x == y: " + (x==y)); // true
        System.out.println("m == n: " + (m==n)); // false
        System.out.println("x.equals(y): " + x.equals(y)); // true
        System.out.println("m.equals(n): " + m.equals(n)); // true
    }
}
```

▶ Run

仔细观察结果的童鞋可以发现， `==` 比较，较小的两个相同的 `Integer` 返回 `true` ，较大的两个相同的 `Integer` 返回 `false` ，这是因为 `Integer` 是不变类，编译器把 `Integer x = 127;` 自

动变为 `Integer x = Integer.valueOf(127);` ，为了节省内存， `Integer.valueOf()` 对于较小的数，始终返回相同的实例，因此， `==` 比较“恰好”为 `true` ，但我们绝不能因为Java标准库的 `Integer` 内部有缓存优化就用 `==` 比较，必须用 `equals()` 方法比较两个 `Integer` 。

按照语义编程，而不是针对特定的底层实现去“优化”。

因为 `Integer.valueOf()` 可能始终返回同一个 `Integer` 实例，因此，在我们自己创建 `Integer` 的时候，以下两种方法：

- 方法1： `Integer n = new Integer(100);`
- 方法2： `Integer n = Integer.valueOf(100);`

方法2更好，因为方法1总是创建新的 `Integer` 实例，方法2把内部优化留给 `Integer` 的实现者去做，即使在当前版本没有优化，也有可能在下一个版本进行优化。

我们把能创建“新”对象的静态方法称为静态工厂方法。 `Integer.valueOf()` 就是静态工厂方法，它尽可能地返回缓存的实例以节省内存。

创建新对象时，优先选用静态工厂方法而不是new操作符。

如果我们考察 `Byte.valueOf()` 方法的源码，可以看到，标准库返回的 `Byte` 实例全部是缓存实例，但调用者并不关心静态工厂方法以何种方式创建新实例还是直接返回缓存的实例。

进制转换

`Integer` 类本身还提供了大量方法，例如，最常用的静态方法 `parseInt()` 可以把字符串解析成一个整数：

```
int x1 = Integer.parseInt("100"); // 100
int x2 = Integer.parseInt("100", 16); // 256,因为按16进制解析
```

`Integer` 还可以把整数格式化为指定进制的字符串：

```
// Integer:
public class Main {
    public static void main(String[] args) {
        System.out.println(Integer.toString(100)); // "100",表示为10进制
        System.out.println(Integer.toString(100, 36)); // "2s",表示为36进制
        System.out.println(Integer.toHexString(100)); // "64",表示为16进制
        System.out.println(Integer.toOctalString(100)); // "144",表示为8进制
        System.out.println(Integer.toBinaryString(100)); // "1100100",表示为2进制
    }
}
```

▶ Run

注意：上述方法的输出都是 `String` ，在计算机内存中，只用二进制表示，不存在十进制或十六进制的表示方法。 `int n = 100` 在内存中总是以4字节的二进制表示：

```
┌────────┬────────┬────────┬────────┐
|00000000|00000000|00000000|01100100|
└────────┴────────┴────────┴────────┘
```

我们经常使用的 `System.out.println(n);` 是依靠核心库自动把整数格式化为10进制输出并显示在屏幕上，使用 `Integer.toHexString(n)` 则通过核心库自动把整数格式化为16进制。

这里我们注意到程序设计的一个重要原则：数据的存储和显示要分离。

Java的包装类型还定义了一些有用的静态变量

```
// boolean只有两个值true/false，其包装类型只需要引用Boolean提供的静态字段：
Boolean t = Boolean.TRUE;
Boolean f = Boolean.FALSE;
// int可表示的最大/最小值:
int max = Integer.MAX_VALUE; // 2147483647
int min = Integer.MIN_VALUE; // -2147483648
// long类型占用的bit和byte数量:
int sizeOfLong = Long.SIZE; // 64 (bits)
int bytesOfLong = Long.BYTES; // 8 (bytes)
```

最后，所有的整数和浮点数的包装类型都继承自 `Number` ，因此，可以非常方便地直接通过包装类型获取各种基本类型：

```
// 向上转型为Number:
Number num = new Integer(999);
// 获取byte, int, long, float, double:
byte b = num.byteValue();
int n = num.intValue();
long ln = num.longValue();
float f = num.floatValue();
double d = num.doubleValue();
```

处理无符号整型

在Java中，并没有无符号整型（Unsigned）的基本数据类型。 `byte` 、 `short` 、 `int` 和 `long` 都是带符号整型，最高位是符号位。而C语言则提供了CPU支持的全部数据类型，包括无符号整型。无符号整型和有符号整型的转换在Java中就需要借助包装

类型的静态方法完成。

例如，byte是有符号整型，范围是 `-128` ~ `+127` ，但如果把 `byte` 看作无符号整型，它的范围就是 `0` ~ `255` 。我们把一个负的 `byte` 按无符号整型转换为 `int` ：

```
// Byte
public class Main {
    public static void main(String[] args) {
        byte x = -1;
        byte y = 127;
        System.out.println(Byte.toUnsignedInt(x)); // 255
        System.out.println(Byte.toUnsignedInt(y)); // 127
    }
}
```

▶ Run

因为 `byte` 的 `-1` 的二进制表示是 `11111111` ，以无符号整型转换后的 `int` 就是 `255` 。

类似的，可以把一个 `short` 按unsigned转换为 `int` ，把一个 `int` 按unsigned转换为 `long` 。

小结

Java核心库提供的包装类型可以把基本类型包装为 `class` ；

自动装箱和自动拆箱都是在编译期完成的（JDK>=1.5）；

装箱和拆箱会影响执行效率，且拆箱时可能发生 `NullPointerException` ；

包装类型的比较必须使用 `equals()` ；

整数和浮点数的包装类型都继承自 `Number` ；

包装类型提供了大量实用方法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





JavaBean

在Java中，有很多 `class` 的定义都符合这样的规范：

- 若干 `private` 实例字段；
- 通过 `public` 方法来读写实例字段。

例如：

```
public class Person {
    private String name;
    private int age;

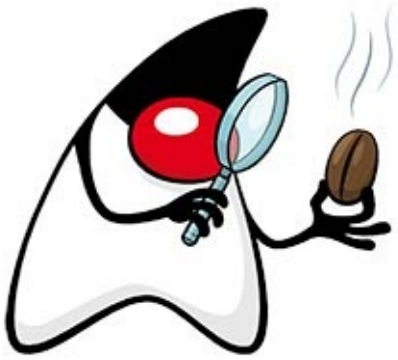
    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return this.age; }
    public void setAge(int age) { this.age = age; }
}
```

如果读写方法符合以下这种命名规范：

```
// 读方法:
public Type getXyz()
// 写方法:
public void setXyz(Type value)
```

那么这种 `class` 被称为 `JavaBean` ：



上面的字段是 `xyz` ，那么读写方法名分别以 `get` 和 `set` 开头，并且后接大写字母开头的字段名 `Xyz` ，因此两个读写方法名分别是 `getXyz()` 和 `setXyz()` 。

`boolean` 字段比较特殊，它的读方法一般命名为 `isXyz()` ：

```
// 读方法:
public boolean isChild()
// 写方法:
public void setChild(boolean value)
```

我们通常把一组对应的读方法（ `getter` ）和写方法（ `setter` ）称为属性（ `property` ）。例如， `name` 属性：

- 对应的读方法是 `String getName()`
- 对应的写方法是 `setName(String)`

只有 `getter` 的属性称为只读属性（read-only），例如，定义一个age只读属性：

- 对应的读方法是 `int getAge()`
- 无对应的写方法 `setAge(int)`

类似的，只有 `setter` 的属性称为只写属性（write-only）。

很明显，只读属性很常见，只写属性不常见。

属性只需要定义 `getter` 和 `setter` 方法，不一定需要对应的字段。例如， `child` 只读属性定义如下：

```
public class Person {
    private String name;
    private int age;

    public String getName() { return this.name; }
    public void setName(String name) { this.name = name; }

    public int getAge() { return this.age; }
    public void setAge(int age) { this.age = age; }

    public boolean isChild() {
        return age <= 6;
    }
}
```

可以看出， `getter` 和 `setter` 也是一种数据封装的方法。

JavaBean的作用

JavaBean主要用来传递数据，即把一组数据组合成一个JavaBean便于传输。此外，JavaBean可以方

便地被IDE工具分析，生成读写属性的代码，主要用在图形界面的可视化设计中。

通过IDE，可以快速生成 `getter` 和 `setter` 。例如，在Eclipse中，先输入以下代码：

```
1. public class Person {
2.     private String name;
3.     private int age;
4. }
```

然后，点击右键，在弹出的菜单中选择“Source”，“Generate Getters and Setters”，在弹出的对话框中选中需要生成 `getter` 和 `setter` 方法的字段，点击确定即可由IDE自动完成所有方法代码。

枚举JavaBean属性

要枚举一个JavaBean的所有属性，可以直接使用Java核心库提供的 `Introspector` ：

```
import java.beans.*;
public class Main {
    public static void main(String[] args) throws Exception {
        BeanInfo info = Introspector.getBeanInfo(Person.class);
        for (PropertyDescriptor pd : info.getPropertyDescriptors()) {
            System.out.println(pd.getName());
            System.out.println("  " + pd.getReadMethod());
            System.out.println("  " + pd.getWriteMethod());
        }
    }
}

class Person {
    private String name;
    private int age;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }
}
```

▶ Run

运行上述代码，可以列出所有的属性，以及对应的读写方法。注意 `class` 属性是从 `Object` 继承的 `getClass()` 方法带来的。

小结

JavaBean是一种符合命名规范的 `class` ，它通过 `getter` 和 `setter` 来定义属性；

属性是一种通用的叫法，并非Java语法规定；

可以利用IDE快速生成 `getter` 和 `setter` ；

使用 `Introspector.getBeanInfo()` 可以获取属性列表。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





枚举类

在Java中，我们可以通过 `static final` 来定义常量。例如，我们希望定义周一到周日这7个常量，可以用7个不同的 `int` 表示：

```
public class Weekday {
    public static final int SUN = 0;
    public static final int MON = 1;
    public static final int TUE = 2;
    public static final int WED = 3;
    public static final int THU = 4;
    public static final int FRI = 5;
    public static final int SAT = 6;
}
```

使用常量的时候，可以这么引用：

```
if (day == Weekday.SAT || day == Weekday.SUN) {
    // TODO: work at home
}
```

也可以把常量定义为字符串类型，例如，定义3种颜色的常量：

```
public class Color {
    public static final String RED = "r";
    public static final String GREEN = "g";
    public static final String BLUE = "b";
}
```

使用常量的时候，可以这么引用：

```
String color = ...
if (Color.RED.equals(color)) {
    // TODO:
}
```

无论是 `int` 常量还是 `String` 常量，使用这些常量来表示一组枚举值的时候，有一个严重的问题就是，编译器无法检查每个值的合理性。例如：

```
if (weekday == 6 || weekday == 7) {
    if (tasks == Weekday.MON) {
```

```
        // TODO:
    }
}
```

上述代码编译和运行均不会报错，但存在两个问题：

- 注意到 `Weekday` 定义的常量范围是 `0` ~ `6` ，并不包含 `7` ，编译器无法检查不在枚举中的 `int` 值；
- 定义的常量仍可与其他变量比较，但其用途并非是枚举星期值。

enum

为了让编译器能自动检查某个值在枚举的集合内，并且，不同用途的枚举需要不同的类型来标记，不能混用，我们可以使用 `enum` 来定义枚举类：

```
// enum
public class Main {
    public static void main(String[] args) {
        Weekday day = Weekday.SUN;
        if (day == Weekday.SAT || day == Weekday.SUN) {
            System.out.println("Work at home!");
        } else {
            System.out.println("Work at office!");
        }
    }
}

enum Weekday {
    SUN, MON, TUE, WED, THU, FRI, SAT;
}
```

▶ Run

注意到定义枚举类是通过关键字 `enum` 实现的，我们只需依次列出枚举的常量名。

和 `int` 定义的常量相比，使用 `enum` 定义枚举有如下好处：

首先， `enum` 常量本身带有类型信息，即 `Weekday.SUN` 类型是 `Weekday` ，编译器会自动检查出类型错误。例如，下面的语句不可能编译通过：

```
int day = 1;
if (day == Weekday.SUN) { // Compile error: bad operand types for binary operator '=='
}
```

其次，不可能引用到非枚举的值，因为无法通过编译。

最后，不同类型的枚举不能互相比较或者赋值，因为类型不符。例如，不能给一个 `Weekday` 枚举类型的变量赋值为 `Color` 枚举类型的值：

```
Weekday x = Weekday.SUN; // ok!
Weekday y = Color.RED; // Compile error: incompatible types
```

这就使得编译器可以在编译期自动检查出所有可能的潜在错误。

enum的比较

使用 `enum` 定义的枚举类是一种引用类型。前面我们讲到，引用类型比较，要使用 `equals()` 方法，如果使用 `==` 比较，它比较的是两个引用类型的变量是否是同一个对象。因此，引用类型比较，要始终使用 `equals()` 方法，但 `enum` 类型可以例外。

这是因为 `enum` 类型的每个常量在JVM中只有一个唯一实例，所以可以直接用 `==` 比较：

```
if (day == Weekday.FRI) { // ok!
}
if (day.equals(Weekday.SUN)) { // ok, but more code!
}
```

enum类型

通过 `enum` 定义的枚举类，和其他的 `class` 有什么区别？

答案是没有任何区别。 `enum` 定义的类型就是 `class` ，只不过它有以下几个特点：

- 定义的 `enum` 类型总是继承自 `java.lang.Enum` ，且无法被继承；
- 只能定义出 `enum` 的实例，而无法通过 `new` 操作符创建 `enum` 的实例；
- 定义的每个实例都是引用类型的唯一实例；
- 可以将 `enum` 类型用于 `switch` 语句。

例如，我们定义的 `Color` 枚举类：

```
public enum Color {
    RED, GREEN, BLUE;
}
```

编译器编译出的 `class` 大概就像这样：

```
public final class Color extends Enum { // 继承自Enum, 标记为final class
    // 每个实例均为全局唯一:
```

```
    public static final Color RED = new Color();
    public static final Color GREEN = new Color();
    public static final Color BLUE = new Color();
    // private构造方法，确保外部无法调用new操作符:
    private Color() {}
}
```

所以，编译后的 `enum` 类和普通 `class` 并没有任何区别。但是我们自己无法按定义普通 `class` 那样来定义 `enum` ，必须使用 `enum` 关键字，这是Java语法规定的。

因为 `enum` 是一个 `class` ，每个枚举的值都是 `class` 实例，因此，这些实例有一些方法：

name()

返回常量名，例如：

```
String s = Weekday.SUN.name(); // "SUN"
```

ordinal()

返回定义的常量的顺序，从0开始计数，例如：

```
int n = Weekday.MON.ordinal(); // 1
```

改变枚举常量定义的顺序就会导致 `ordinal()` 返回值发生变化。例如：

```
public enum Weekday {
    SUN, MON, TUE, WED, THU, FRI, SAT;
}
```

和

```
public enum Weekday {
    MON, TUE, WED, THU, FRI, SAT, SUN;
}
```

的 `ordinal` 就是不同的。如果在代码中编写了类似 `if(x.ordinal()==1)` 这样的语句，就要保证 `enum` 的枚举顺序不能变。新增的常量必须放在最后。

有些童鞋会想， `Weekday` 的枚举常量如果要和 `int` 转换，使用 `ordinal()` 不是非常方便？比如这样写：

```
String task = Weekday.MON.ordinal() + "/ppt";
saveToFile(task);
```

但是，如果不小心修改了枚举的顺序，编译器是无法检查出这种逻辑错误的。要编写健壮的代码，就不要依靠 `ordinal()` 的返回值。因为 `enum` 本身是 `class` ，所以我们可以定义 `private` 的构造方法，并且，给每个枚举常量添加字段：

```
// enum
public class Main {
    public static void main(String[] args) {
        Weekday day = Weekday.SUN;
        if (day.dayValue == 6 || day.dayValue == 0) {
            System.out.println("Work at home!");
        } else {
            System.out.println("Work at office!");
        }
    }
}

enum Weekday {
    MON(1), TUE(2), WED(3), THU(4), FRI(5), SAT(6), SUN(0);

    public final int dayValue;

    private Weekday(int dayValue) {
        this.dayValue = dayValue;
    }
}
```

▶ Run

这样就无需担心顺序的变化，新增枚举常量时，也需要指定一个 `int` 值。

注意：枚举类的字段也可以是非final类型，即可以在运行期修改，但是不推荐这样做！

默认情况下，对枚举常量调用 `toString()` 会返回和 `name()` 一样的字符串。但是， `toString()` 可以被覆写，而 `name()` 则不行。我们可以给 `Weekday` 添加 `toString()` 方法：

```
// enum
public class Main {
    public static void main(String[] args) {
        Weekday day = Weekday.SUN;
        if (day.dayValue == 6 || day.dayValue == 0) {
            System.out.println("Today is " + day + ". Work at home!");
        } else {
            System.out.println("Today is " + day + ". Work at office!");
        }
    }
}

enum Weekday {
    MON(1, "星期一"), TUE(2, "星期二"), WED(3, "星期三"), THU(4, "星期四"), FRI(5, "星期五"), SAT(6, "星期六"), SUN(0, "星期日");

    public final int dayValue;
    private final String chinese;

    private Weekday(int dayValue, String chinese) {
        this.dayValue = dayValue;
        this.chinese = chinese;
    }

    @Override
    public String toString() {
        return this.chinese;
    }
}
```

▶ Run

覆写 `toString()` 的目的是在输出时更有可读性。

注意：判断枚举常量的名字，要始终使用name()方法，绝不能调用toString()！

switch

最后，枚举类可以应用在 `switch` 语句中。因为枚举类天生具有类型信息和有限个枚举常量，所以比 `int` 、 `String` 类型更适合用在 `switch` 语句中：

```
// switch
public class Main {
    public static void main(String[] args) {
        Weekday day = Weekday.SUN;
        switch(day) {
        case MON:
        case TUE:
        case WED:
        case THU:
        case FRI:
            System.out.println("Today is " + day + ". Work at office!");
            break;
        case SAT:
        case SUN:
            System.out.println("Today is " + day + ". Work at home!");
            break;
        default:
            throw new RuntimeException("cannot process " + day);
        }
    }
}

enum Weekday {
    MON, TUE, WED, THU, FRI, SAT, SUN;
}
```

▶ Run

加上 `default` 语句，可以在漏写某个枚举常量时自动报错，从而及时发现错误。

小结

Java使用 `enum` 定义枚举类型，它被编译器编译为 `final class Xxx extends Enum { … }` ；

通过 `name()` 获取常量定义的字符串，注意不要使用 `toString()` ；

通过 `ordinal()` 返回常量定义的顺序（无实质意义）；

可以为 `enum` 编写构造方法、字段和方法

`enum` 的构造方法要声明为 `private` ，字段强烈建议声明为 `final` ；

`enum` 适合用在 `switch` 语句中。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





BigInteger

BigInteger

在Java中，由CPU原生提供的整型最大范围是64位 `long` 型整数。使用 `long` 型整数可以直接通过CPU指令进行计算，速度非常快。

如果我们使用的整数范围超过了 `long` 型怎么办？这个时候，就只能用软件来模拟一个大整数。 `java.math.BigInteger` 就是用来表示任意大小的整数。 `BigInteger` 内部用一个 `int[]` 数组来模拟一个非常大的整数：

```
BigInteger bi = new BigInteger("1234567890");
System.out.println(bi.pow(5)); //
286797186029971810723376143809367204829490000
```

对 `BigInteger` 做运算的时候，只能使用实例方法，例如，加法运算：

```
BigInteger i1 = new BigInteger("1234567890");
BigInteger i2 = new BigInteger("12345678901234567890");
BigInteger sum = i1.add(i2); // 12345678902469135780
```

和 `long` 型整数运算比， `BigInteger` 不会有范围限制，但缺点是速度比较慢。

也可以把 `BigInteger` 转换成 `long` 型：

```
BigInteger i = new BigInteger("123456789000");
System.out.println(i.longValue()); // 123456789000
System.out.println(i.multiply(i).longValueExact()); //
java.lang.ArithmeticException: BigInteger out of long range
```

使用 `longValueExact()` 方法时，如果超出了 `long` 型的范围，会抛出 `ArithmeticException` 。

`BigInteger` 和 `Integer` 、 `Long` 一样，也是不可变类，并且也继承自 `Number` 类。因为 `Number` 定义了转换为基本类型的几个方法：

- 转换为 `byte` ： `byteValue()`
- 转换为 `short` ： `shortValue()`
- 转换为 `int` ： `intValue()`
- 转换为 `long` ： `longValue()`
- 转换为 `float` ： `floatValue()`

- 转换为 `double` ： `doubleValue()`

因此，通过上述方法，可以把 `BigInteger` 转换成基本类型。如果 `BigInteger` 表示的范围超过了基本类型的范围，转换时将丢失高位信息，即结果不一定是准确的。如果需要准确地转换成基本类型，可以使用 `intValueExact()` 、 `longValueExact()` 等方法，在转换时如果超出范围，将直接抛出 `ArithmeticException` 异常。

如果 `BigInteger` 的值甚至超过了 `float` 的最大范围（3.4x10^{38}），那么返回的float是什么呢？

```
// BigInteger to float
import java.math.BigInteger;

public class Main {
    public static void main(String[] args) {
        BigInteger n = new BigInteger("999999").pow(99);
        float f = n.floatValue();
        System.out.println(f);
    }
}
```

▶ Run

小结

`BigInteger` 用于表示任意大小的整数；

`BigInteger` 是不变类，并且继承自 `Number` ；

将 `BigInteger` 转换成基本类型时可使用 `longValueExact()` 等方法保证结果准确。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





BigDecimal

和 `BigInteger` 类似， `BigDecimal` 可以表示一个任意大小且精度完全准确的浮点数。

```
BigDecimal bd = new BigDecimal("123.4567");
System.out.println(bd.multiply(bd)); // 15241.55677489
```

`BigDecimal` 用 `scale()` 表示小数位数，例如：

```
BigDecimal d1 = new BigDecimal("123.45");
BigDecimal d2 = new BigDecimal("123.4500");
BigDecimal d3 = new BigDecimal("1234500");
System.out.println(d1.scale()); // 2,两位小数
System.out.println(d2.scale()); // 4
System.out.println(d3.scale()); // 0
```

通过 `BigDecimal` 的 `stripTrailingZeros()` 方法，可以将一个 `BigDecimal` 格式化为一个相等的，但去掉了末尾0的 `BigDecimal` ：

```
BigDecimal d1 = new BigDecimal("123.4500");
BigDecimal d2 = d1.stripTrailingZeros();
System.out.println(d1.scale()); // 4
System.out.println(d2.scale()); // 2,因为去掉了00

BigDecimal d3 = new BigDecimal("1234500");
BigDecimal d4 = d3.stripTrailingZeros();
System.out.println(d3.scale()); // 0
System.out.println(d4.scale()); // -2
```

如果一个 `BigDecimal` 的 `scale()` 返回负数，例如， `-2` ，表示这个数是个整数，并且末尾有2个0。

可以对一个 `BigDecimal` 设置它的 `scale` ，如果精度比原始值低，那么按照指定的方法进行四舍五入或者直接截断：

```
import java.math.BigDecimal;
import java.math.RoundingMode;

public class Main {
    public static void main(String[] args) {
        BigDecimal d1 = new BigDecimal("123.456789");
        BigDecimal d2 = d1.setScale(4, RoundingMode.HALF_UP); // 四舍五入，123.4568
        BigDecimal d3 = d1.setScale(4, RoundingMode.DOWN); // 直接截断，123.4567
        System.out.println(d2);
        System.out.println(d3);
    }
}
```

▶ Run

对 `BigDecimal` 做加、减、乘时，精度不会丢失，但是做除法时，存在无法除尽的情况，这时，就必须指定精度以及如何进行截断：

```
1. BigDecimal d1 = new BigDecimal("123.456");
2. BigDecimal d2 = new BigDecimal("23.456789");
3. BigDecimal d3 = d1.divide(d2, 10, RoundingMode.HALF_UP); // 保留10位小数并四舍五入
4. BigDecimal d4 = d1.divide(d2); // 报错：ArithmeticException，因为除不尽
```

还可以对 `BigDecimal` 做除法的同时求余数：

```
import java.math.BigDecimal;

public class Main {
    public static void main(String[] args) {
        BigDecimal n = new BigDecimal("12.345");
        BigDecimal m = new BigDecimal("0.12");
        BigDecimal[] dr = n.divideAndRemainder(m);
        System.out.println(dr[0]); // 102
        System.out.println(dr[1]); // 0.105
    }
}
```

▶ Run

调用 `divideAndRemainder()` 方法时，返回的数组包含两个 `BigDecimal` ，分别是商和余数，其中商总是整数，余数不会大于除数。我们可以利用这个方法判断两个 `BigDecimal` 是否是整数倍数：

```
1. BigDecimal n = new BigDecimal("12.75");
2. BigDecimal m = new BigDecimal("0.15");
3. BigDecimal[] dr = n.divideAndRemainder(m);
4. if (dr[1].signum() == 0) {
5.     // n是m的整数倍
```

```
}
```

比较BigDecimal

在比较两个 `BigDecimal` 的值是否相等时，要特别注意，使用 `equals()` 方法不但要求两个 `BigDecimal` 的值相等，还要求它们的 `scale()` 相等：

```
BigDecimal d1 = new BigDecimal("123.456");
BigDecimal d2 = new BigDecimal("123.45600");
System.out.println(d1.equals(d2)); // false,因为scale不同
System.out.println(d1.equals(d2.stripTrailingZeros())); // true,因为d2去除尾部0后scale变为2
System.out.println(d1.compareTo(d2)); // 0
```

必须使用 `compareTo()` 方法来比较，它根据两个值的大小分别返回负数、正数和 `0` ，分别表示小于、大于和等于。

总是使用compareTo()比较两个BigDecimal的值，不要使用equals()！

如果查看 `BigDecimal` 的源码，可以发现，实际上一个 `BigDecimal` 是通过一个 `BigInteger` 和一个 `scale` 来表示的，即 `BigInteger` 表示一个完整的整数，而 `scale` 表示小数位数：

```
public class BigDecimal extends Number implements Comparable<BigDecimal> {
    private final BigInteger intVal;
    private final int scale;
}
```

`BigDecimal` 也是从 `Number` 继承的，也是不可变对象。

小结

`BigDecimal` 用于表示精确的小数，常用于财务计算；

比较 `BigDecimal` 的值是否相等，必须使用 `compareTo()` 而不能使用 `equals()` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





常用工具类

Java的核心库提供了大量的现成的类供我们使用。本节我们介绍几个常用的工具类。

Math

顾名思义， `Math` 类就是用来进行数学计算的，它提供了大量的静态方法来便于我们实现数学计算：

求绝对值：

```
1. Math.abs(-100); // 100
2. Math.abs(-7.8); // 7.8
```

取最大或最小值：

```
1. Math.max(100, 99); // 100
2. Math.min(1.2, 2.3); // 1.2
```

计算x^y次方：

```
1. Math.pow(2, 10); // 2的10次方=1024
```

计算$\sqrt{x}$：

```
1. Math.sqrt(2); // 1.414...
```

计算e^x次方：

```
1. Math.exp(2); // 7.389...
```

计算以e为底的对数：

```
1. Math.log(4); // 1.386...
```

计算以10为底的对数：

```
1. Math.log10(100); // 2
```

三角函数：

```
Math.sin(3.14); // 0.00159...
Math.cos(3.14); // -0.9999...
Math.tan(3.14); // -0.0015...
Math.asin(1.0); // 1.57079...
Math.acos(1.0); // 0.0
```

Math还提供了几个数学常量：

```
double pi = Math.PI; // 3.14159...
double e = Math.E; // 2.7182818...
Math.sin(Math.PI / 6); // sin(π/6) = 0.5
```

生成一个随机数x，x的范围是 `0 <= x < 1` ：

```
Math.random(); // 0.53907... 每次都不一样
```

如果我们要生成一个区间在 `[MIN, MAX)` 的随机数，可以借助 `Math.random()` 实现，计算如下：

```
// 区间在[MIN, MAX)的随机数
public class Main {
    public static void main(String[] args) {
        double x = Math.random(); // x的范围是[0,1)
        double min = 10;
        double max = 50;
        double y = x * (max - min) + min; // y的范围是[10,50)
        long n = (long) y; // n的范围是[10,50)的整数
        System.out.println(y);
        System.out.println(n);
    }
}
```

▶ Run

有些童鞋可能注意到Java标准库还提供了一个 `StrictMath` ，它提供了和 `Math` 几乎一模一样的方法。这两个类的区别在于，由于浮点数计算存在误差，不同的平台（例如x86和ARM）计算的结果可能不一致（指误差不同），因此， `StrictMath` 保证所有平台计算结果都是完全相同的，而 `Math` 会尽量针对平台优化计算速度，所以，绝大多数情况下，使用 `Math` 就足够了。

Random

`Random` 用来创建伪随机数。所谓伪随机数，是指只要给定一个初始的种子，产生的随机数序列是完全一样的。

要生成一个随机数，可以使用 `nextInt()` 、 `nextLong()` 、 `nextFloat()` 、 `nextDouble()` ：

```
Random r = new Random();
r.nextInt(); // 2071575453,每次都不一样
r.nextInt(10); // 5,生成一个[0,10)之间的int
r.nextLong(); // 8811649292570369305,每次都不一样
r.nextFloat(); // 0.54335...生成一个[0,1)之间的float
r.nextDouble(); // 0.3716...生成一个[0,1)之间的double
```

有童鞋问，每次运行程序，生成的随机数都是不同的，没看出伪随机数的特性来。

这是因为我们创建 `Random` 实例时，如果不给定种子，就使用系统当前时间戳作为种子，因此每次运行时，种子不同，得到的伪随机数序列就不同。

如果我们在创建 `Random` 实例时指定一个种子，就会得到完全确定的随机数序列：

```
import java.util.Random;
public class Main {
    public static void main(String[] args) {
        Random r = new Random(12345);
        for (int i = 0; i < 10; i++) {
            System.out.println(r.nextInt(100));
        }
        // 51, 80, 41, 28, 55...
    }
}
```

▶ Run

前面我们使用的 `Math.random()` 实际上内部调用了 `Random` 类，所以它也是伪随机数，只是我们无法指定种子。

SecureRandom

有伪随机数，就有真随机数。实际上真正的真随机数只能通过量子力学原理来获取，而我们想要的是一个不可预测的安全的随机数， `SecureRandom` 就是用来创建安全的随机数的：

```
SecureRandom sr = new SecureRandom();
System.out.println(sr.nextInt(100));
```

`SecureRandom` 无法指定种子，它使用RNG（random number generator）算法。JDK的 `SecureRandom` 实际上有多种不同的底层实现，有的使用安全随机种子加上伪随机数算法来产生安

全的随机数，有的使用真正的随机数生成器。实际使用的时候，可以优先获取高强度的安全随机数生成器，如果没有提供，再使用普通等级的安全随机数生成器：

```
import java.util.Arrays;
import java.security.SecureRandom;
import java.security.NoSuchAlgorithmException;

public class Main {
    public static void main(String[] args) {
        SecureRandom sr = null;
        try {
            sr = SecureRandom.getInstanceStrong(); // 获取高强度安全随机数生成器
        } catch (NoSuchAlgorithmException e) {
            sr = new SecureRandom(); // 获取普通的安全随机数生成器
        }
        byte[] buffer = new byte[16];
        sr.nextBytes(buffer); // 用安全随机数填充buffer
        System.out.println(Arrays.toString(buffer));
    }
}
```

▶ Run

`SecureRandom` 的安全性是通过操作系统提供的安全的随机种子来生成随机数。这个种子是通过CPU的热噪声、读写磁盘的字节、网络流量等各种随机事件产生的“熵”。

在密码学中，安全的随机数非常重要。如果使用不安全的伪随机数，所有加密体系都将被攻破。因此，时刻牢记必须使用 `SecureRandom` 来产生安全的随机数。

需要使用安全随机数的时候，必须使用SecureRandom，绝不能使用Random！

小结

Java提供的常用工具类有：

- Math：数学计算
- Random：生成伪随机数
- SecureRandom：生成安全的随机数

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论

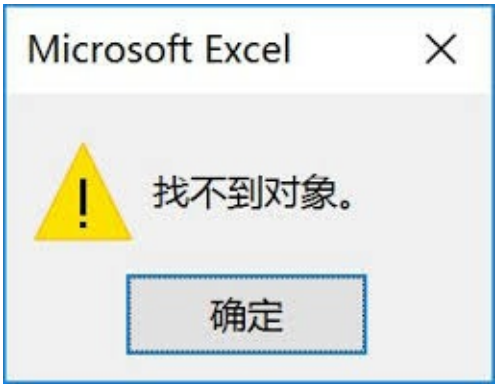




异常处理

程序运行的时候，经常会发生各种错误。

比如，使用Excel的时候，它有时候会报错：



本章我们讨论如何在Java程序中处理各种异常情况。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java的异常

在计算机程序运行的过程中，总是会出现各种各样的错误。

有一些错误是用户造成的，比如，希望用户输入一个 `int` 类型的年龄，但是用户的输入是 `abc` ：

```
// 假设用户输入了abc：
String s = "abc";
int n = Integer.parseInt(s); // NumberFormatException!
```

程序想要读写某个文件的内容，但是用户已经把它删除了：

```
// 用户删除了该文件：
String t = readFile("C:\\abc.txt"); // FileNotFoundException!
```

还有一些错误是随机出现，并且永远不可能避免的。比如：

- 网络突然断了，连接不到远程服务器；
- 内存耗尽，程序崩溃了；
- 用户点“打印”，但根本没有打印机；
- ……

所以，一个健壮的程序必须处理各种各样的错误。

所谓错误，就是程序调用某个函数的时候，如果失败了，就表示出错。

调用方如何获知调用失败的信息？有两种方法：

方法一：约定返回错误码。

例如，处理一个文件，如果返回 `0` ，表示成功，返回其他整数，表示约定的错误码：

```
int code = processFile("C:\\test.txt");
if (code == 0) {
    // ok:
} else {
    // error:
    switch (code) {
    case 1:
        // file not found:
    case 2:
        // no read permission:
    default:
```

```
        // unknown error:
    }
}
```

因为使用 `int` 类型的错误码，想要处理就非常麻烦。这种方式常见于底层C函数。

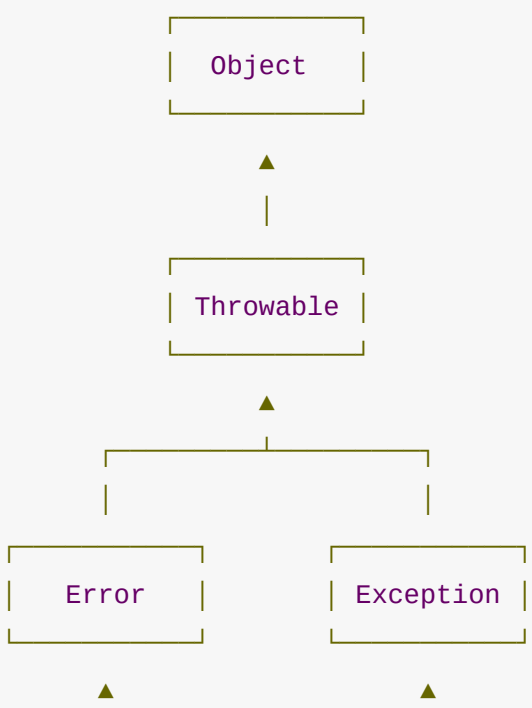
方法二：在语言层面上提供一个异常处理机制。

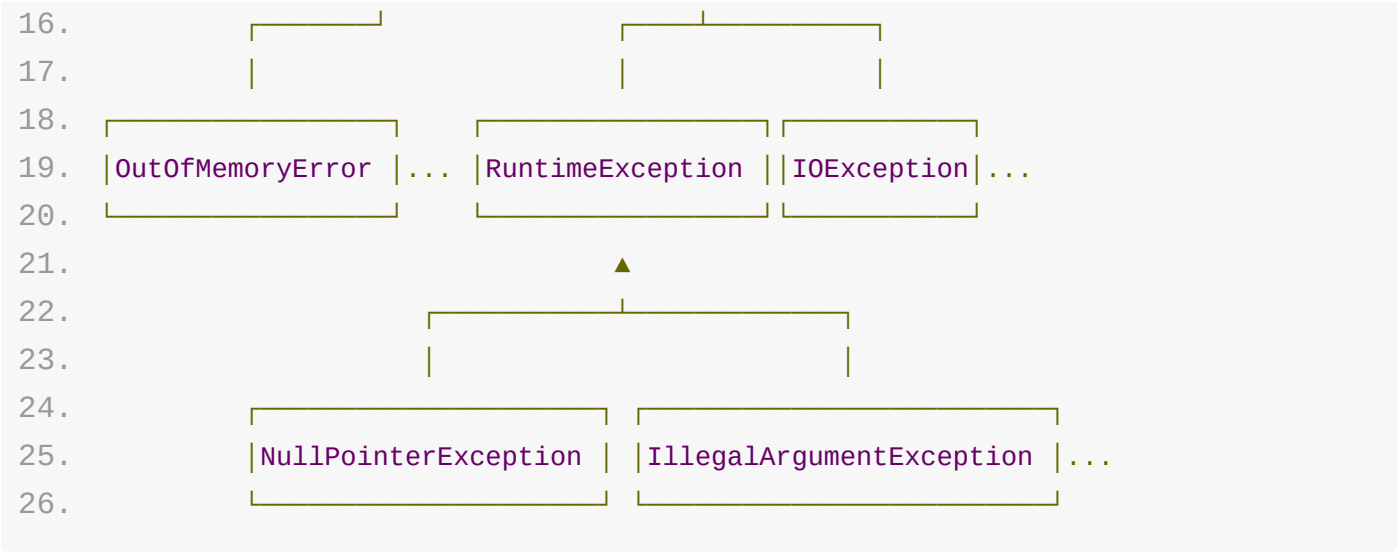
Java内置了一套异常处理机制，总是使用异常来表示错误。

异常是一种 `class` ，因此它本身带有类型信息。异常可以在任何地方抛出，但只需要在上层捕获，这样就和方法调用分离了：

```
try {
    String s = processFile("C:\\test.txt");
    // ok:
} catch (FileNotFoundException e) {
    // file not found:
} catch (SecurityException e) {
    // no read permission:
} catch (IOException e) {
    // io error:
} catch (Exception e) {
    // other error:
}
```

因为Java的异常是 `class` ，它的继承关系如下：





从继承关系可知： `Throwable` 是异常体系的根，它继承自 `Object` 。 `Throwable` 有两个体系： `Error` 和 `Exception` ， `Error` 表示严重的错误，程序对此一般无能为力，例如：

- `OutOfMemoryError` ：内存耗尽
- `NoClassDefFoundError` ：无法加载某个Class
- `StackOverflowError` ：栈溢出

而 `Exception` 则是运行时的错误，它可以被捕获并处理。

某些异常是应用程序逻辑处理的一部分，应该捕获并处理。例如：

- `NumberFormatException` ：数值类型的格式错误
- `FileNotFoundException` ：未找到文件
- `SocketException` ：读取网络失败

还有一些异常是程序逻辑编写不对造成的，应该修复程序本身。例如：

- `NullPointerException` ：对某个 `null` 的对象调用方法或字段
- `IndexOutOfBoundsException` ：数组索引越界

`Exception` 又分为两大类：

- `RuntimeException` 以及它的子类；
- 非 `RuntimeException` （包括 `IOException` 、 `ReflectiveOperationException` 等等）Java规定：

- 必须捕获的异常，包括 `Exception` 及其子类，但不包括 `RuntimeException` 及其子类，这种类型的异常称为Checked Exception。

- 不需要捕获的异常，包括 `Error` 及其子类， `RuntimeException` 及其子类。

捕获异常

捕获异常使用 `try…catch` 语句，把可能发生异常的代码放到 `try {…}` 中，然后使用 `catch` 捕获对应的 `Exception` 及其子类：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) {
        try {
            // 用指定编码转换String为byte[]:
            return s.getBytes("GBK");
        } catch (UnsupportedEncodingException e) {
            // 如果系统不支持GBK编码，会捕获到UnsupportedEncodingException:
            System.out.println(e); // 打印异常信息
            return s.getBytes(); // 尝试使用用默认编码
        }
    }
}
```

▶ Run

如果我们不捕获 `UnsupportedEncodingException` ，会出现编译失败的问题：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) {
        return s.getBytes("GBK");
    }
}
```

▶ Run

编译器会报错，错误信息类似：unreported exception UnsupportedEncodingException; must be caught or declared to be thrown，并且准确地指出需要捕获的语句是 `return s.getBytes("GBK");` 。意思是说，像 `UnsupportedEncodingException` 这样的Checked Exception，必须被捕获。

这是因为 `String.getBytes(String)` 方法定义是：

```
    public byte[] getBytes(String charsetName) throws UnsupportedEncodingException
1.  {
2.      ...
3.  }
```

在方法定义的时候，使用 `throws Xxx` 表示该方法可能抛出的异常类型。调用方在调用的时候，必须强制捕获这些异常，否则编译器会报错。

在 `toGBK()` 方法中，因为调用了 `String.getBytes(String)` 方法，就必须捕获 `UnsupportedEncodingException` 。我们也可以不捕获它，而是在方法定义处用throws表示 `toGBK()` 方法可能会抛出 `UnsupportedEncodingException` ，就可以让 `toGBK()` 方法通过编译器检查：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws UnsupportedEncodingException {
        return s.getBytes("GBK");
    }
}
```

▶ Run

上述代码仍然会得到编译错误，但这一次，编译器提示的不是调用 `return s.getBytes("GBK");` 的问题，而是 `byte[] bs = toGBK("中文");` 。因为在 `main()` 方法中，调用 `toGBK()` ，没有捕获它声明的可能抛出的 `UnsupportedEncodingException` 。

修复方法是在 `main()` 方法中捕获异常并处理：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        try {
            byte[] bs = toGBK("中文");
            System.out.println(Arrays.toString(bs));
        } catch (UnsupportedEncodingException e) {
            System.out.println(e);
        }
    }

    static byte[] toGBK(String s) throws UnsupportedEncodingException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}
```

▶ Run

可见，只要是方法声明的Checked Exception，不在调用层捕获，也必须在更高的调用层捕获。所有未捕获的异常，最终也必须在 `main()` 方法中捕获，不会出现漏写 `try` 的情况。这是由编译器保证的。 `main()` 方法也是最后捕获 `Exception` 的机会。

如果是测试代码，上面的写法就略显麻烦。如果不想写任何 `try` 代码，可以直接把 `main()` 方法定义为 `throws Exception` ：

```
// try...catch
import java.io.UnsupportedEncodingException;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) throws Exception {
        byte[] bs = toGBK("中文");
        System.out.println(Arrays.toString(bs));
    }

    static byte[] toGBK(String s) throws UnsupportedEncodingException {
        // 用指定编码转换String为byte[]:
        return s.getBytes("GBK");
    }
}
```

▶ Run

因为 `main()` 方法声明了可能抛出 `Exception` ，也就声明了可能抛出所有的 `Exception` ，因此在内部就无需捕获了。代价就是一旦发生异常，程序会立刻退出。

还有一些童鞋喜欢在 `toGBK()` 内部“消化”异常：

```
static byte[] toGBK(String s) {
    try {
        return s.getBytes("GBK");
    } catch (UnsupportedEncodingException e) {
        // 什么也不干
    }
    return null;
```

这种捕获后不处理的方式是非常不好的，即使真的什么也做不了，也要先把异常记录下来：

```
static byte[] toGBK(String s) {
    try {
        return s.getBytes("GBK");
    } catch (UnsupportedEncodingException e) {
        // 先记下来再说:
        e.printStackTrace();
    }
    return null;
```

所有异常都可以调用 `printStackTrace()` 方法打印异常栈，这是一个简单有用的快速打印异常的方法。

小结

Java使用异常来表示错误，并通过 `try … catch` 捕获异常；

Java的异常是 `class` ，并且从 `Throwable` 继承；

`Error` 是无需捕获的严重错误， `Exception` 是应该捕获的可处理的错误；

`RuntimeException` 无需强制捕获，非 `RuntimeException` （Checked Exception）需强制捕获，或者用 `throws` 声明；

不推荐捕获了异常但不进行任何处理。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





捕获异常

在Java中，凡是可能抛出异常的语句，都可以用 `try … catch` 捕获。把可能发生异常的语句放在 `try { … }` 中，然后使用 `catch` 捕获对应的 `Exception` 及其子类。

多catch语句

可以使用多个 `catch` 语句，每个 `catch` 分别捕获对应的 `Exception` 及其子类。JVM在捕获到异常后，会从上到下匹配 `catch` 语句，匹配到某个 `catch` 后，执行 `catch` 代码块，然后不再继续匹配。

简单地说就是：多个 `catch` 语句只有一个能被执行。例如：

```
1.  public static void main(String[] args) {
2.      try {
3.          process1();
4.          process2();
5.          process3();
6.      } catch (IOException e) {
7.          System.out.println(e);
8.      } catch (NumberFormatException e) {
9.          System.out.println(e);
10.     }
11. }
```

存在多个 `catch` 的时候， `catch` 的顺序非常重要：子类必须写在前面。例如：

```
1.  public static void main(String[] args) {
2.      try {
3.          process1();
4.          process2();
5.          process3();
6.      } catch (IOException e) {
7.          System.out.println("IO error");
8.      } catch (UnsupportedEncodingException e) { // 永远捕获不到
9.          System.out.println("Bad encoding");
10.     }
11. }
```

对于上面的代码， `UnsupportedEncodingException` 异常是永远捕获不到的，因为它是 `IOException` 的子类。当抛出 `UnsupportedEncodingException` 异常时，会被 `catch`

`(IOException e) { … }` 捕获并执行。

因此，正确的写法是把子类放到前面：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
    } catch (IOException e) {
        System.out.println("IO error");
    }
}
```

finally语句

无论是否有异常发生，如果我们都希望执行一些语句，例如清理工作，怎么写？

可以把执行语句写若干遍：正常执行的放到 `try` 中，每个 `catch` 再写一遍。例如：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
        System.out.println("END");
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
        System.out.println("END");
    } catch (IOException e) {
        System.out.println("IO error");
        System.out.println("END");
    }
}
```

上述代码无论是否发生异常，都会执行 `System.out.println("END");` 这条语句。

那么如何消除这些重复的代码？Java的 `try … catch` 机制还提供了 `finally` 语句， `finally` 语句块保证有无错误都会执行。上述代码可以改写如下：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (UnsupportedEncodingException e) {
        System.out.println("Bad encoding");
    } catch (IOException e) {
        System.out.println("IO error");
    } finally {
        System.out.println("END");
    }
}
```

注意 `finally` 有几个特点：

- `finally` 语句不是必须的，可写可不写；
- `finally` 总是最后执行。如果没有发生异常，就正常执行 `try { … }` 语句块，然后执行 `finally` 。如果发生了异常，就中断执行 `try { … }` 语句块，然后跳转执行匹配的 `catch` 语句块，最后执行 `finally` 。

可见， `finally` 是用来保证一些代码必须执行的。

某些情况下，可以没有 `catch` ，只使用 `try … finally` 结构。例如：

```
void process(String file) throws IOException {
    try {
        ...
    } finally {
        System.out.println("END");
    }
}
```

因为方法声明了可能抛出的异常，所以可以不写 `catch` 。

捕获多种异常

如果某些异常的处理逻辑相同，但是异常本身不存在继承关系，那么就得编写多条 `catch` 子句：

```
public static void main(String[] args) {
    try {
        process1();
```

```
            process2();
            process3();
        } catch (IOException e) {
            System.out.println("Bad input");
        } catch (NumberFormatException e) {
            System.out.println("Bad input");
        } catch (Exception e) {
            System.out.println("Unknown error");
        }
    }
```

因为处理 `IOException` 和 `NumberFormatException` 的代码是相同的，所以我们可以把它两用 `|` 合并到一起：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException | NumberFormatException e) { // IOException或NumberFormatException
        System.out.println("Bad input");
    } catch (Exception e) {
        System.out.println("Unknown error");
    }
}
```

练习

用 `try … catch` 捕获异常并处理。

下载练习：捕获异常练习 （推荐使用IDE练习插件快速下载）

小结

使用 `try … catch … finally` 时：

- 多个 `catch` 语句的匹配顺序非常重要，子类必须放在前面；
- `finally` 语句保证了有无异常都会执行，它是可选的；
- 一个 `catch` 语句也可以匹配多个非继承关系的异常。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





抛出异常

异常的传播

当某个方法抛出了异常时，如果当前方法没有捕获异常，异常就会被抛到上层调用方法，直到遇到某个 `try … catch` 被捕获为止：

```
// exception
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        process2();
    }

    static void process2() {
        Integer.parseInt(null); // 会抛出NumberFormatException
    }
}
```

▶ Run

通过 `printStackTrace()` 可以打印出方法的调用栈，类似：

```
1. java.lang.NumberFormatException: null
2.     at java.base/java.lang.Integer.parseInt(Integer.java:614)
3.     at java.base/java.lang.Integer.parseInt(Integer.java:770)
4.     at Main.process2(Main.java:16)
5.     at Main.process1(Main.java:12)
6.     at Main.main(Main.java:5)
```

`printStackTrace()` 对于调试错误非常有用，上述信息表示： `NumberFormatException` 是在 `java.lang.Integer.parseInt` 方法中被抛出的，调用层次从上到下依次是：

- `main()` 调用 `process1()` ；
- `process1()` 调用 `process2()` ；
- `process2()` 调用 `Integer.parseInt(String)` ；
- `Integer.parseInt(String)` 调用 `Integer.parseInt(String, int)` 。查看 `Integer.java` 源码可知，抛出异常的方法代码如下：

```
public static int parseInt(String s, int radix) throws NumberFormatException {
    if (s == null) {
        throw new NumberFormatException("null");
    }
    ...
}
```

并且，每层调用均给出了源代码的行号，可直接定位。

抛出异常

当发生错误时，例如，用户输入了非法的字符，我们就可以抛出异常。

如何抛出异常？参考 `Integer.parseInt()` 方法，抛出异常分两步：

- 创建某个 `Exception` 的实例；
- 用 `throw` 语句抛出。下面是一个例子：

```
void process2(String s) {
    if (s==null) {
        NullPointerException e = new NullPointerException();
        throw e;
    }
}
```

实际上，绝大部分抛出异常的代码都会合并写成一行：

```
void process2(String s) {
    if (s==null) {
        throw new NullPointerException();
    }
}
```

如果一个方法捕获了某个异常后，又在 `catch` 子句中抛出新的异常，就相当于把抛出的异常类型“转换”了：

```
void process1(String s) {
    try {
        process2();
    } catch (NullPointerException e) {
        throw new IllegalArgumentException();
    }
```

```
}

void process2(String s) {
    if (s==null) {
        throw new NullPointerException();
    }
}
```

当 `process2()` 抛出 `NullPointerException` 后，被 `process1()` 捕获，然后抛出 `IllegalArgumentException()` 。

如果在 `main()` 中捕获 `IllegalArgumentException` ，我们看看打印的异常栈：

```
// exception
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        try {
            process2();
        } catch (NullPointerException e) {
            throw new IllegalArgumentException();
        }
    }

    static void process2() {
        throw new NullPointerException();
    }
}
```

▶ Run

打印出的异常栈类似：

```
java.lang.IllegalArgumentException
    at Main.process1(Main.java:15)
    at Main.main(Main.java:5)
```

这说明新的异常丢失了原始异常信息，我们已经看不到原始异常 `NullPointerException` 的信息了。

为了能追踪到完整的异常栈，在构造异常的时候，把原始的 `Exception` 实例传进去，新的 `Exception` 就可以持有原始 `Exception` 信息。对上述代码改进如下：

```
// exception
public class Main {
    public static void main(String[] args) {
        try {
            process1();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    static void process1() {
        try {
            process2();
        } catch (NullPointerException e) {
            throw new IllegalArgumentException(e);
        }
    }

    static void process2() {
        throw new NullPointerException();
    }
}
```

▶ Run

运行上述代码，打印出的异常栈类似：

```
java.lang.IllegalArgumentException: java.lang.NullPointerException
    at Main.process1(Main.java:15)
    at Main.main(Main.java:5)
Caused by: java.lang.NullPointerException
    at Main.process2(Main.java:20)
    at Main.process1(Main.java:13)
```

注意到 `Caused by: Xxx` ，说明捕获的 `IllegalArgumentException` 并不是造成问题的根源，根源在于 `NullPointerException` ，是在 `Main.process2()` 方法抛出的。

在代码中获取原始异常可以使用 `Throwable.getCause()` 方法。如果返回 `null` ，说明已经是“根异常”了。

有了完整的异常栈的信息，我们才能快速定位并修复代码的问题。

如果我们在 `try` 或者 `catch` 语句块中抛出异常， `finally` 语句是否会执行？例如：

```
// exception
public class Main {
    public static void main(String[] args) {
        try {
            Integer.parseInt("abc");
        } catch (Exception e) {
            System.out.println("catched");
            throw new RuntimeException(e);
        } finally {
            System.out.println("finally");
        }
    }
}
```

▶ Run

上述代码执行结果如下：

```
1. catched
2. finally
   Exception in thread "main" java.lang.RuntimeException:
3. java.lang.NumberFormatException: For input string: "abc"
4.     at Main.main(Main.java:8)
5. Caused by: java.lang.NumberFormatException: For input string: "abc"
6.     at ...
```

第一行打印了 `catched` ，说明进入了 `catch` 语句块。第二行打印了 `finally` ，说明执行了 `finally` 语句块。

因此，在 `catch` 中抛出异常，不会影响 `finally` 的执行。JVM会先执行 `finally` ，然后抛出异常。

异常屏蔽

如果在执行 `finally` 语句时抛出异常，那么， `catch` 语句的异常还能否继续抛出？例如：

```
// exception
public class Main {
    public static void main(String[] args) {
        try {
            Integer.parseInt("abc");
        } catch (Exception e) {
            System.out.println("catched");
            throw new RuntimeException(e);
        } finally {
            System.out.println("finally");
            throw new IllegalArgumentException();
        }
    }
}
```

▶ Run

执行上述代码，发现异常信息如下：

```
1. catched
2. finally
3. Exception in thread "main" java.lang.IllegalArgumentException
4.     at Main.main(Main.java:11)
```

这说明 `finally` 抛出异常后，原来在 `catch` 中准备抛出的异常就“消失”了，因为只能抛出一个异常。没有被抛出的异常称为“被屏蔽”的异常（Suppressed Exception）。

在极少数的情况下，我们需要获知所有的异常。如何保存所有的异常信息？方法是先用 `origin` 变量保存原始异常，然后调用 `Throwable.addSuppressed()` ，把原始异常添加进来，最后在 `finally` 抛出：

```
// exception
public class Main {
    public static void main(String[] args) throws Exception {
        Exception origin = null;
        try {
            System.out.println(Integer.parseInt("abc"));
        } catch (Exception e) {
            origin = e;
            throw e;
        } finally {
            Exception e = new IllegalArgumentException();
            if (origin != null) {
                e.addSuppressed(origin);
            }
            throw e;
        }
    }
}
```

▶ Run

当 `catch` 和 `finally` 都抛出了异常时，虽然 `catch` 的异常被屏蔽了，但是， `finally` 抛出的异常仍然包含了它：

```
1. Exception in thread "main" java.lang.IllegalArgumentException
2.     at Main.main(Main.java:11)
3. Suppressed: java.lang.NumberFormatException: For input string: "abc"
       at
4. java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.ja
5.     at java.base/java.lang.Integer.parseInt(Integer.java:652)
6.     at java.base/java.lang.Integer.parseInt(Integer.java:770)
7.     at Main.main(Main.java:6)
```

通过 `Throwable.getSuppressed()` 可以获取所有的 `Suppressed Exception` 。

绝大多数情况下，在 `finally` 中不要抛出异常。因此，我们通常不需要关心 `Suppressed Exception` 。

练习

如果传入的参数为负，则抛出IllegalArgumentException。

下载练习：抛出异常练习 （推荐使用IDE练习插件快速下载）

小结

调用 `printStackTrace()` 可以打印异常的传播栈，对于调试非常有用；

捕获异常并再次抛出新的异常时，应该持有原始异常信息；

通常不要在 `finally` 中抛出异常。如果在 `finally` 中抛出异常，应该原始异常加入到原有异常中。调用方可通过 `Throwable.getSuppressed()` 获取所有添加的 `Suppressed Exception` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





自定义异常

Java标准库定义的常用异常包括：

```
Exception
│
├─ RuntimeException
│  │
│  ├─ NullPointerException
│  │
│  ├─ IndexOutOfBoundsException
│  │
│  ├─ SecurityException
│  │
│  └─ IllegalArgumentException
│     │
│     └─ NumberFormatException
│
├─ IOException
│  │
│  ├─ UnsupportedCharsetException
│  │
│  ├─ FileNotFoundException
│  │
│  └─ SocketException
│
├─ ParseException
│
├─ GeneralSecurityException
│
├─ SQLException
│
└─ TimeoutException
```

当我们在代码中需要抛出异常时，尽量使用JDK已定义的异常类型。例如，参数检查不合法，应该抛出 `IllegalArgumentException` ：

```
static void process1(int age) {
    if (age <= 0) {
        throw new IllegalArgumentException();
    }
```

```
}
```

在一个大型项目中，可以自定义新的异常类型，但是，保持一个合理的异常继承体系是非常重要的。

一个常见的做法是自定义一个 `BaseException` 作为“根异常”，然后，派生出各种业务类型的异常。

`BaseException` 需要从一个适合的 `Exception` 派生，通常建议从 `RuntimeException` 派生：

```
public class BaseException extends RuntimeException {
}
```

其他业务类型的异常就可以从 `BaseException` 派生：

```
public class UserNotFoundException extends BaseException {
}

public class LoginFailedException extends BaseException {
}

...
```

自定义的 `BaseException` 应该提供多个构造方法：

```
public class BaseException extends RuntimeException {
    public BaseException() {
        super();
    }

    public BaseException(String message, Throwable cause) {
        super(message, cause);
    }

    public BaseException(String message) {
        super(message);
    }

    public BaseException(Throwable cause) {
        super(cause);
    }
}
```

上述构造方法实际上都是原样照抄 `RuntimeException` 。这样，抛出异常的时候，就可以选择合适的

构造方法。通过IDE可以根据父类快速生成子类的构造方法。

练习

下载练习：从BaseException派生自定义异常 （推荐使用IDE练习插件快速下载）

小结

抛出异常时，尽量复用JDK已定义的异常类型；

自定义异常体系时，推荐从 `RuntimeException` 派生“根异常”，再派生出业务异常；

自定义异常时，应该提供多种构造方法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用断言

断言（Assertion）是一种调试程序的方式。在Java中，使用 `assert` 关键字来实现断言。

我们先看一个例子：

```
1. public static void main(String[] args) {
2.     double x = Math.abs(-123.45);
3.     assert x >= 0;
4.     System.out.println(x);
5. }
```

语句 `assert x >= 0;` 即为断言，断言条件 `x >= 0` 预期为 `true` 。如果计算结果为 `false` ，则断言失败，抛出 `AssertionError` 。

使用 `assert` 语句时，还可以添加一个可选的断言消息：

```
1. assert x >= 0 : "x must >= 0";
```

这样，断言失败的时候， `AssertionError` 会带上消息 `x must >= 0` ，更加便于调试。

Java断言的特点是：断言失败时会抛出 `AssertionError` ，导致程序结束退出。因此，断言不能用于可恢复的程序错误，只应该用于开发和测试阶段。

对于可恢复的程序错误，不应该使用断言。例如：

```
1. void sort(int[] arr) {
2.     assert arr != null;
3. }
```

应该抛出异常并在上层捕获：

```
1. void sort(int[] arr) {
2.     if (x == null) {
3.         throw new IllegalArgumentException("array cannot be null");
4.     }
5. }
```

当我们在程序中使用 `assert` 时，例如，一个简单的断言：

```
// assert
public class Main {
    public static void main(String[] args) {
        int x = -1;
        assert x > 0;
        System.out.println(x);
    }
}
```

▶ Run

断言 `x` 必须大于 `0` ，实际上 `x` 为 `-1` ，断言肯定失败。执行上述代码，发现程序并未抛出 `AssertionError` ，而是正常打印了 `x` 的值。

这是怎么肥四？为什么 `assert` 语句不起作用？

这是因为JVM默认关闭断言指令，即遇到 `assert` 语句就自动忽略了，不执行。

要执行 `assert` 语句，必须给Java虚拟机传递 `-enableassertions` （可简写为 `-ea` ）参数启用断言。所以，上述程序必须在命令行下运行才有效果：

```
$ java -ea Main.java
Exception in thread "main" java.lang.AssertionError
    at Main.main(Main.java:5)
```

还可以有选择地对特定地类启用断言，命令行参数是： `-ea:com.itranswarp.sample.Main` ，表示只对 `com.itranswarp.sample.Main` 这个类启用断言。

或者对特定地包启用断言，命令行参数是： `-ea:com.itranswarp.sample…` （注意结尾有3个 `.` ），表示对 `com.itranswarp.sample` 这个包启动断言。

实际开发中，很少使用断言。更好的方法是编写单元测试，后续我们会讲解 `JUnit` 的使用。

小结

断言是一种调试方式，断言失败会抛出 `AssertionError` ，只能在开发和测试阶段启用断言；

对可恢复的错误不能使用断言，而应该抛出异常；

断言很少被使用，更好的方法是编写单元测试。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用JDK Logging

在编写程序的过程中，发现程序运行结果与预期不符，怎么办？当然是用 `System.out.println()` 打印出执行过程中的某些变量，观察每一步的结果与代码逻辑是否符合，然后有针对性地修改代码。

代码改好了怎么办？当然是删除没有用的 `System.out.println()` 语句了。

如果改代码又改出问题怎么办？再加上 `System.out.println()` 。

反复这么搞几次，很快大家就发现使用 `System.out.println()` 非常麻烦。

怎么办？

解决方法是使用日志。

那什么是日志？日志就是Logging，它的目的是为了取代 `System.out.println()` 。

输出日志，而不是用 `System.out.println()` ，有以下几个好处：

- 可以设置输出样式，避免自己每次都写 `"ERROR: " + var` ；
- 可以设置输出级别，禁止某些级别输出。例如，只输出错误日志；
- 可以被重定向到文件，这样可以在程序运行结束后查看日志；
- 可以按包名控制日志级别，只输出某些包打的日志；
- 可以……总之就是好处很多啦。

那如何使用日志？

因为Java标准库内置了日志包 `java.util.logging` ，我们可以直接用。先看一个简单的例子：

```
// logging
import java.util.logging.Level;
import java.util.logging.Logger;
```

```
public class Hello {
    public static void main(String[] args) {
        Logger logger = Logger.getGlobal();
        logger.info("start process...");
        logger.warning("memory is running out...");
        logger.fine("ignored.");
        logger.severe("process will be terminated...");
    }
}
```

▶ Run

运行上述代码，得到类似如下的输出：

```
Mar 02, 2019 6:32:13 PM Hello main
INFO: start process...
Mar 02, 2019 6:32:13 PM Hello main
WARNING: memory is running out...
Mar 02, 2019 6:32:13 PM Hello main
SEVERE: process will be terminated...
```

对比可见，使用日志最大的好处是，它自动打印了时间、调用类、调用方法等很多有用的信息。

再仔细观察发现，4条日志，只打印了3条，`logger.fine()`没有打印。这是因为，日志的输出可以设定级别。JDK的Logging定义了7个日志级别，从严重到普通：

- SEVERE
- WARNING
- INFO
- CONFIG
- FINE
- FINER
- FINEST

因为默认级别是INFO，因此，INFO级别以下的日志，不会被打印出来。使用日志级别的好处在于，调整级别，就可以屏蔽掉很多调试相关的日志输出。

使用Java标准库内置的Logging有以下局限：

Logging系统在JVM启动时读取配置文件并完成初始化，一旦开始运行`main()`方法，就无法修改配置；

配置不太方便，需要在JVM启动时传递参数`-Djava.util.logging.config.file=<config-file-name>`。

因此，Java标准库内置的Logging使用并不是非常广泛。更方便的日志系统我们稍后介绍。

练习

使用logger.severe()打印异常：

```
import java.io.UnsupportedEncodingException;
import java.util.logging.Logger;

public class Main {
    public static void main(String[] args) {
        Logger logger = Logger.getLogger(Main.class.getName());
        logger.info("Start process...");
        try {
            "".getBytes("invalidCharsetName");
        } catch (UnsupportedEncodingException e) {
            // TODO: 使用logger.severe()打印异常
        }
        logger.info("Process end.");



    }
}
```

▶ Run

下载练习：打印异常 （推荐使用IDE练习插件快速下载）

小结

日志是为了替代 `System.out.println()` ，可以定义格式，重定向到文件等；

日志可以存档，便于追踪问题；

日志记录可以按级别分类，便于打开或关闭某些级别；

可以根据配置文件调整日志，无需修改代码；

Java标准库提供了 `java.util.logging` 来实现日志功能。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Commons Logging

和Java标准库提供的日志不同，Commons Logging是一个第三方日志库，它是由Apache创建的日志模块。

Commons Logging的特色是，它可以挂接不同的日志系统，并通过配置文件指定挂接的日志系统。默认情况下，Commons Loggin自动搜索并使用Log4j（Log4j是另一个流行的日志系统），如果没有找到Log4j，再使用JDK Logging。

使用Commons Logging只需要和两个类打交道，并且只有两步：

第一步，通过 `LogFactory` 获取 `Log` 类的实例；第二步，使用 `Log` 实例的方法打日志。

示例代码如下：

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class Main {
    public static void main(String[] args) {
        Log log = LogFactory.getLog(Main.class);
        log.info("start...");
        log.warn("end.");
    }
}
```

▶ Run

运行上述代码，肯定会得到编译错误，类似 `error: package org.apache.commons.logging does not exist`（找不到 `org.apache.commons.logging` 这个包）。因为Commons Logging是一个第三方提供的库，所以，必须先把它下载下来。下载后，解压，找到 `commons-logging-1.2.jar` 这个文件，再把Java源码 `Main.java` 放到一个目录下，例如 `work` 目录：

```
work
|
├─ commons-logging-1.2.jar
|
└─ Main.java
```

然后用 `javac` 编译 `Main.java`，编译的时候要指定 `classpath`，不然编译器找不到我们引用的 `org.apache.commons.logging` 包。编译命令如下：

```
javac -cp commons-logging-1.2.jar Main.java
```

如果编译成功，那么当前目录下就会多出一个 `Main.class` 文件：

```
work
|
├─ commons-logging-1.2.jar
|
├─ Main.java
|
└─ Main.class
```

现在可以执行这个 `Main.class` ，使用 `java` 命令，也必须指定 `classpath` ，命令如下：

```
java -cp .;commons-logging-1.2.jar Main
```

注意到传入的 `classpath` 有两部分：一个是 `.` ，一个是 `commons-logging-1.2.jar` ，用 `;` 分割。 `.` 表示当前目录，如果没有这个 `.` ，JVM不会在当前目录搜索 `Main.class` ，就会报错。

如果在Linux或macOS下运行，注意 `classpath` 的分隔符不是 `;` ，而是 `:` ：

```
java -cp .:commons-logging-1.2.jar Main
```

运行结果如下：

```
Mar 02, 2019 7:15:31 PM Main main
INFO: start...
Mar 02, 2019 7:15:31 PM Main main
WARNING: end.
```

Commons Logging定义了6个日志级别：

- FATAL
- ERROR
- WARNING
- INFO
- DEBUG
- TRACE

默认级别是 `INFO` 。

使用Commons Logging时，如果在静态方法中引用 `Log` ，通常直接定义一个静态类型变量：

```
// 在静态方法中引用Log:
public class Main {
    static final Log log = LogFactory.getLog(Main.class);

    static void foo() {
        log.info("foo");
    }
}
```

在实例方法中引用 `Log` ，通常定义一个实例变量：

```
// 在实例方法中引用Log:
public class Person {
    protected final Log log = LogFactory.getLog(getClass());

    void foo() {
        log.info("foo");
    }
}
```

注意到实例变量log的获取方式是 `LogFactory.getLog(getClass())` ，虽然也可以用 `LogFactory.getLog(Person.class)` ，但是前一种方式有个非常大的好处，就是子类可以直接使用该 `log` 实例。例如：

```
// 在子类中使用父类实例化的log:
public class Student extends Person {
    void bar() {
        log.info("bar");
    }
}
```

由于Java类的动态特性，子类获取的 `log` 字段实际上相当于 `LogFactory.getLog(Student.class)` ，但却是从父类继承而来，并且无需改动代码。

此外，Commons Logging的日志方法，例如 `info()` ，除了标准的 `info(String)` 外，还提供了一个非常有用的重载方法： `info(String, Throwable)` ，这使得记录异常更加简单：

```
try {
    ...
} catch (Exception e) {
    log.error("got exception!", e);
```

```
5. }
```

练习

使用 `log.error(String, Throwable)` 打印异常。

下载练习：Commons Logging练习 （推荐使用IDE练习插件快速下载）

小结

Commons Logging是使用最广泛的日志模块；

Commons Logging的API非常简单；

Commons Logging可以自动检测并使用其他日志模块。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Log4j

前面介绍了Commons Logging，可以作为“日志接口”来使用。而真正的“日志实现”可以使用Log4j。

Log4j是一种非常流行的日志框架，最新版本是2.x。

Log4j是一个组件化设计的日志系统，它的架构大致如下：

```
log.info("User signed in.");
  |
  |   ┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
  ├──>| Appender |──->|  Filter  |──->|  Layout  |──->| Console  |
  |   └──────────┘    └──────────┘    └──────────┘    └──────────┘
  |
  |   ┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
  ├──>| Appender |──->|  Filter  |──->|  Layout  |──->|   File   |
  |   └──────────┘    └──────────┘    └──────────┘    └──────────┘
  |
  |   ┌──────────┐    ┌──────────┐    ┌──────────┐    ┌──────────┐
  └──>| Appender |──->|  Filter  |──->|  Layout  |──->|  Socket  |
      └──────────┘    └──────────┘    └──────────┘    └──────────┘
```

当我们使用Log4j输出一条日志时，Log4j自动通过不同的Appender把同一条日志输出到不同的目的地。例如：

- console：输出到屏幕；
- file：输出到文件；
- socket：通过网络输出到远程计算机；
- jdbc：输出到数据库

在输出日志的过程中，通过Filter来过滤哪些log需要被输出，哪些log不需要被输出。例如，仅输出 `ERROR` 级别的日志。

最后，通过Layout来格式化日志信息，例如，自动添加日期、时间、方法名称等信息。

上述结构虽然复杂，但我们在实际使用的时候，并不需要关心Log4j的API，而是通过配置文件来配置它。

以XML配置为例，使用Log4j的时候，我们把一个 `log4j2.xml` 的文件放到 `classpath` 下就可以让Log4j读取配置文件并按照我们的配置来输出日志。下面是一个配置文件的例子：

```
<?xml version="1.0" encoding="UTF-8"?>
<Configuration>
```

```
    <Properties>
        <!-- 定义日志格式 -->
        <Property name="log.pattern">%d{MM-dd HH:mm:ss.SSS} [%t] %-5level
%logger{36}%n%msg%n%n</Property>
        <!-- 定义文件名变量 -->
        <Property name="file.err.filename">log/err.log</Property>
        <Property name="file.err.pattern">log/err.%i.log.gz</Property>
    </Properties>
    <!-- 定义Appender，即目的地 -->
    <Appenders>
        <!-- 定义输出到屏幕 -->
        <Console name="console" target="SYSTEM_OUT">
            <!-- 日志格式引用上面定义的log.pattern -->
            <PatternLayout pattern="${log.pattern}" />
        </Console>
        <!-- 定义输出到文件,文件名引用上面定义的file.err.filename -->
        <RollingFile name="err" bufferedIO="true"
fileName="${file.err.filename}" filePattern="${file.err.pattern}">
            <PatternLayout pattern="${log.pattern}" />
            <Policies>
                <!-- 根据文件大小自动切割日志 -->
                <SizeBasedTriggeringPolicy size="1 MB" />
            </Policies>
            <!-- 保留最近10份 -->
            <DefaultRolloverStrategy max="10" />
        </RollingFile>
    </Appenders>
    <Loggers>
        <Root level="info">
            <!-- 对info级别的日志，输出到console -->
            <AppenderRef ref="console" level="info" />
            <!-- 对error级别的日志，输出到err，即上面定义的RollingFile -->
            <AppenderRef ref="err" level="error" />
        </Root>
    </Loggers>
</Configuration>
```

虽然配置Log4j比较繁琐，但一旦配置完成，使用起来就非常方便。对上面的配置文件，凡是 `INFO` 级别的日志，会自动输出到屏幕，而 `ERROR` 级别的日志，不但会输出到屏幕，还会同时输出到文件。并且，一旦日志文件达到指定大小（1MB），Log4j就会自动切割新的日志文件，并最多保留10份。

有了配置文件还不够，因为Log4j也是一个第三方库，我们需要从这里下载Log4j，解压后，把以下3个jar包放到 `classpath` 中：

- log4j-api-2.x.jar
- log4j-core-2.x.jar
- log4j-jcl-2.x.jar

因为Commons Logging会自动发现并使用Log4j，所以，把上一节下载的 `commons-logging-1.2.jar` 也放到 `classpath` 中。

要打印日志，只需要按Commons Logging的写法写，不需要改动任何代码，就可以得到Log4j的日志输出，类似：

```
1. 03-03 12:09:45.880 [main] INFO  com.itranswarp.learnjava.Main
2. Start process...
```

最佳实践

在开发阶段，始终使用Commons Logging接口来写入日志，并且开发阶段无需引入Log4j。如果需要把日志写入文件，只需要把正确的配置文件和Log4j相关的jar包放入 `classpath` ，就可以自动把日志切换成使用Log4j写入，无需修改任何代码。

练习

根据配置文件，观察Log4j写入的日志文件。

下载练习：commons logging + log4j （推荐使用IDE练习插件快速下载）

小结

通过Commons Logging实现日志，不需要修改代码即可使用Log4j；

使用Log4j只需要把log4j2.xml和相关jar放入classpath；

如果要更换Log4j，只需要移除log4j2.xml和相关jar；

只有扩展Log4j时，才需要引用Log4j的接口（例如，将日志加密写入数据库的功能，需要自己开发）。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用SLF4J和Logback

前面介绍了Commons Logging和Log4j这一对好基友，它们一个负责充当日志API，一个负责实现日志底层，搭配使用非常便于开发。

有的童鞋可能还听说过SLF4J和Logback。这两个东东看上去也像日志，它们又是啥？

其实SLF4J类似于Commons Logging，也是一个日志接口，而Logback类似于Log4j，是一个日志的实现。

为什么有了Commons Logging和Log4j，又会蹦出来SLF4J和Logback？这是因为Java有着非常悠久的开源历史，不但OpenJDK本身是开源的，而且我们用到的第三方库，几乎全部都是开源的。开源生态丰富的一个特定就是，同一个功能，可以找到若干种互相竞争的开源库。

因为对Commons Logging的接口不满意，有人就搞了SLF4J。因为对Log4j的性能不满意，有人就搞了Logback。

我们先来看看SLF4J对Commons Logging的接口有何改进。在Commons Logging中，我们要打印日志，有时候得这么写：

```
1. int score = 99;
2. p.setScore(score);
3. log.info("Set score " + score + " for Person " + p.getName() + " ok.");
```

拼字符串是一个非常麻烦的事情，所以SLF4J的日志接口改进成这样了：

```
1. int score = 99;
2. p.setScore(score);
3. logger.info("Set score {} for Person {} ok.", score, p.getName());
```

我们靠猜也能猜出来，SLF4J的日志接口传入的是一个带占位符的字符串，用后面的变量自动替换占位符，所以看起来更加自然。

如何使用SLF4J？它的接口实际上和Commons Logging几乎一模一样：

```
1. import org.slf4j.Logger;
2. import org.slf4j.LoggerFactory;
3.
4. class Main {
5.     final Logger logger = LoggerFactory.getLogger(getClass());
6. }
```

对比一下Commons Logging和SLF4J的接口：

Commons Logging	SLF4J
org.apache.commons.logging.Log	org.slf4j.Logger
org.apache.commons.logging.LogFactory	org.slf4j.LoggerFactory

不同之处就是Log变成了Logger，LogFactory变成了LoggerFactory。

使用SLF4J和Logback和前面讲到的使用Commons Logging加Log4j是类似的，先分别下载SLF4J和Logback，然后把以下jar包放到classpath下：

- slf4j-api-1.7.x.jar
- logback-classic-1.2.x.jar
- logback-core-1.2.x.jar

然后使用SLF4J的Logger和LoggerFactory即可。和Log4j类似，我们仍然需要一个Logback的配置文件，把 `logback.xml` 放到classpath下，配置如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
        </encoder>
    </appender>

    <appender name="FILE"
class="ch.qos.logback.core.rolling.RollingFileAppender">
        <encoder>
            <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} -
%msg%n</pattern>
            <charset>utf-8</charset>
        </encoder>
        <file>log/output.log</file>
        <rollingPolicy
class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <fileNamePattern>log/output.log.%i</fileNamePattern>
        </rollingPolicy>
        <triggeringPolicy
class="ch.qos.logback.core.rolling.SizeBasedTriggeringPolicy">
            <MaxFileSize>1MB</MaxFileSize>
```

```
        </triggeringPolicy>
    </appender>

    <root level="INFO">
        <appender-ref ref="CONSOLE" />
        <appender-ref ref="FILE" />
    </root>
</configuration>
```

运行即可获得类似如下的输出：

```
13:15:25.328 [main] INFO  com.itranswarp.learnjava.Main - Start process...
```

从目前的趋势来看，越来越多的开源项目从Commons Logging加Log4j转向了SLF4J加Logback。

练习

根据配置文件，观察Logback写入的日志文件。

下载练习：slf4j+logback （推荐使用IDE练习插件快速下载）

小结

SLF4J和Logback可以取代Commons Logging和Log4j；

始终使用SLF4J的接口写入日志，使用Logback只需要配置，不需要修改代码。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





反射

什么是反射？

反射就是Reflection，Java的反射是指程序在运行期可以拿到一个对象的所有信息。

正常情况下，如果我们要调用一个对象的方法，或者访问一个对象的字段，通常会传入对象实例：

```
// Main.java
import com.itranswarp.learnjava.Person;

public class Main {
    String getFullName(Person p) {
        return p.getFirstName() + " " + p.getLastName();
    }
}
```

但是，如果不能获得 `Person` 类，只有一个 `Object` 实例，比如这样：

```
String getFullName(Object obj) {
    return ???
}
```

怎么办？有童鞋会说：强制转型啊！

```
String getFullName(Object obj) {
    Person p = (Person) obj;
    return p.getFirstName() + " " + p.getLastName();
}
```

强制转型的时候，你会发现一个问题：编译上面的代码，仍然需要引用 `Person` 类。不然，去掉 `import` 语句，你看能不能编译通过？

所以，反射是为了解决在运行期，对某个实例一无所知的情况下，如何调用其方法。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Class类

除了 `int` 等基本类型外，Java的其他类型全部都是 `class` （包括 `interface` ）。例如：

- `String`
- `Object`
- `Runnable`
- `Exception`
- …

仔细思考，我们可以得出结论： `class` （包括 `interface` ）的本质是数据类型（ `Type` ）。无继承关系的数据类型无法赋值：

```
Number n = new Double(123.456); // OK
String s = new Double(123.456); // compile error!
```

而 `class` 是由JVM在执行过程中动态加载的。JVM在第一次读取到一种 `class` 类型时，将其加载进内存。

每加载一种 `class` ，JVM就为其创建一个 `Class` 类型的实例，并关联起来。注意：这里的 `Class` 类型是一个名叫 `Class` 的 `class` 。它长这样：

```
public final class Class {
    private Class() {}
}
```

以 `String` 类为例，当JVM加载 `String` 类时，它首先读取 `String.class` 文件到内存，然后，为 `String` 类创建一个 `Class` 实例并关联起来：

```
Class cls = new Class(String);
```

这个 `Class` 实例是JVM内部创建的，如果我们查看JDK源码，可以发现 `Class` 类的构造方法是 `private` ，只有JVM能创建 `Class` 实例，我们自己的Java程序是无法创建 `Class` 实例的。

所以，JVM持有的每个 `Class` 实例都指向一个数据类型（ `class` 或 `interface` ）：

```
┌──────────────────────────────┐
│      Class Instance          │────────> String
├──────────────────────────────┤
│name = "java.lang.String"     │
└──────────────────────────────┘
```

```
┌──────────────────────────────┐
│      Class Instance          │────────> Random
├──────────────────────────────┤
│name = "java.util.Random"     │
└──────────────────────────────┘
┌──────────────────────────────┐
│      Class Instance          │────────> Runnable
├──────────────────────────────┤
│name = "java.lang.Runnable"   │
└──────────────────────────────┘
```

一个 `Class` 实例包含了该 `class` 的所有完整信息：

```
┌──────────────────────────────┐
│      Class Instance          │────────> String
├──────────────────────────────┤
│name = "java.lang.String"     │
├──────────────────────────────┤
│package = "java.lang"         │
├──────────────────────────────┤
│super = "java.lang.Object"    │
├──────────────────────────────┤
│interface = CharSequence...   │
├──────────────────────────────┤
│field = value[],hash,...      │
├──────────────────────────────┤
│method = indexOf()...         │
└──────────────────────────────┘
```

由于JVM为每个加载的 `class` 创建了对应的 `Class` 实例，并在实例中保存了该 `class` 的所有信息，包括类名、包名、父类、实现的接口、所有方法、字段等，因此，如果获取了某个 `Class` 实例，我们就可以通过这个 `Class` 实例获取到该实例对应的 `class` 的所有信息。

这种通过 `Class` 实例获取 `class` 信息的方法称为反射（`Reflection`）。

如何获取一个 `class` 的 `Class` 实例？有三个方法：

方法一：直接通过一个 `class` 的静态变量 `class` 获取：

```
Class cls = String.class;
```

方法二：如果我们有一个实例变量，可以通过该实例变量提供的 `getClass()` 方法获取：

```
String s = "Hello";
Class cls = s.getClass();
```

方法三：如果知道一个 `class` 的完整类名，可以通过静态方法 `Class.forName()` 获取：

```
Class cls = Class.forName("java.lang.String");
```

因为 `Class` 实例在JVM中是唯一的，所以，上述方法获取的 `Class` 实例是同一个实例。可以用 `==` 比较两个 `Class` 实例：

```
Class cls1 = String.class;

String s = "Hello";
Class cls2 = s.getClass();

boolean sameClass = cls1 == cls2; // true
```

注意一下 `Class` 实例比较和 `instanceof` 的差别：

```
Integer n = new Integer(123);

boolean b1 = n instanceof Integer; // true, 因为n是Integer类型
boolean b2 = n instanceof Number; // true, 因为n是Number类型的子类

boolean b3 = n.getClass() == Integer.class; // true, 因为n.getClass()返回
Integer.class
boolean b4 = n.getClass() == Number.class; // false, 因为
Integer.class!=Number.class
```

用 `instanceof` 不但匹配指定类型，还匹配指定类型的子类。而用 `==` 判断 `class` 实例可以精确地判断数据类型，但不能作子类型比较。

通常情况下，我们应该用 `instanceof` 判断数据类型，因为面向抽象编程的时候，我们不关心具体的子类型。只有在需要精确判断一个类型是不是某个 `class` 的时候，我们才使用 `==` 判断 `class` 实例。

因为反射的目的是为了获得某个实例的信息。因此，当我们拿到某个 `Object` 实例时，我们可以通过反射获取该 `Object` 的 `class` 信息：

```
void printObjectInfo(Object obj) {
    Class cls = obj.getClass();
}
```

要从 `Class` 实例获取获取的基本信息，参考下面的代码：

```
// reflection
public class Main {
    public static void main(String[] args) {
        printClassInfo("".getClass());
        printClassInfo(Runnable.class);
        printClassInfo(java.time.Month.class);
        printClassInfo(String[].class);
        printClassInfo(int.class);
    }

    static void printClassInfo(Class cls) {
        System.out.println("Class name: " + cls.getName());
        System.out.println("Simple name: " + cls.getSimpleName());
        if (cls.getPackage() != null) {
            System.out.println("Package name: " + cls.getPackage().getName());
        }
        System.out.println("is interface: " + cls.isInterface());
        System.out.println("is enum: " + cls.isEnum());
        System.out.println("is array: " + cls.isArray());
        System.out.println("is primitive: " + cls.isPrimitive());
    }
}
```

Run

注意到数组（例如 `String[]` ）也是一种 `Class` ，而且不同于 `String.class` ，它的类名是 `[Ljava.lang.String` 。此外，JVM为每一种基本类型如int也创建了 `Class` ，通过 `int.class` 访问。

如果获取到了一个 `Class` 实例，我们就可以通过该 `Class` 实例来创建对应类型的实例：

```
// 获取String的Class实例:
Class cls = String.class;
// 创建一个String实例:
String s = (String) cls.newInstance();
```

上述代码相当于 `new String()` 。通过 `Class.newInstance()` 可以创建类实例，它的局限是：只能调用 `public` 的无参数构造方法。带参数的构造方法，或者非 `public` 的构造方法都无法通过 `Class.newInstance()` 被调用。

动态加载

JVM在执行Java程序的时候，并不是一次性把所有用到的class全部加载到内存，而是第一次需要用到class时才加载。例如：

```
// Main.java
public class Main {
    public static void main(String[] args) {
        if (args.length > 0) {
            create(args[0]);
        }
    }

    static void create(String name) {
        Person p = new Person(name);
    }
}
```

当执行 `Main.java` 时，由于用到了 `Main` ，因此，JVM首先会把 `Main.class` 加载到内存。然而，并不会加载 `Person.class` ，除非程序执行到 `create()` 方法，JVM发现需要加载 `Person` 类时，才会首次加载 `Person.class` 。如果没有执行 `create()` 方法，那么 `Person.class` 根本就不会被加载。

这就是JVM动态加载 `class` 的特性。

动态加载 `class` 的特性对于Java程序非常重要。利用JVM动态加载 `class` 的特性，我们才能在运行期根据条件加载不同的实现类。例如，Commons Logging总是优先使用Log4j，只有当Log4j不存在时，才使用JDK的logging。利用JVM动态加载特性，大致的实现代码如下：

```
// Commons Logging优先使用Log4j:
LogFactory factory = null;
if (isClassPresent("org.apache.logging.log4j.Logger")) {
    factory = createLog4j();
} else {
    factory = createJdkLog();
}

boolean isClassPresent(String name) {
    try {
        Class.forName(name);
        return true;
    } catch (Exception e) {
        return false;
    }
}
```

这就是为什么我们只需要把Log4j的jar包放到classpath中，Commons Logging就会自动使用

Log4j的原因。

小结

JVM为每个加载的 `class` 及 `interface` 创建了对应的 `Class` 实例来保存 `class` 及 `interface` 的所有信息；

获取一个 `class` 对应的 `Class` 实例后，就可以获取该 `class` 的所有信息；

通过Class实例获取 `class` 信息的方法称为反射（Reflection）；

JVM总是动态加载 `class` ，可以在运行期根据条件来控制加载class。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





访问字段

对任意的一个 `Object` 实例，只要我们获取了它的 `Class` ，就可以获取它的一切信息。

我们先看看如何通过 `Class` 实例获取字段信息。 `Class` 类提供了以下几个方法来获取字段：

- `Field getField(name)`：根据字段名获取某个public的field（包括父类）
- `Field getDeclaredField(name)`：根据字段名获取当前类的某个field（不包括父类）
- `Field[] getFields()`：获取所有public的field（包括父类）
- `Field[] getDeclaredFields()`：获取当前类的所有field（不包括父类）

我们来看一下示例代码：

```
// reflection
public class Main {
    public static void main(String[] args) throws Exception {
        Class stdClass = Student.class;
        // 获取public字段"score":
        System.out.println(stdClass.getField("score"));
        // 获取继承的public字段"name":
        System.out.println(stdClass.getField("name"));
        // 获取private字段"grade":
        System.out.println(stdClass.getDeclaredField("grade"));
    }
}

class Student extends Person {
    public int score;
    private int grade;
}

class Person {
    public String name;
}
```

▶ Run

上述代码首先获取 `Student` 的 `Class` 实例，然后，分别获取 `public` 字段、继承的 `public` 字段以及 `private` 字段，打印出的 `Field` 类似：

```
1. public int Student.score
2. public java.lang.String Person.name
3. private int Student.grade
```

一个 `Field` 对象包含了一个字段的所有信息：

- `getName()` ：返回字段名称，例如， `"name"` ；

- `getType()`：返回字段类型，也是一个`Class`实例，例如，`String.class`；
- `getModifiers()`：返回字段的修饰符，它是一个`int`，不同的bit表示不同的含义。

以`String`类的`value`字段为例，它的定义是：

```
public final class String {
    private final byte[] value;
}
```

我们用反射获取该字段的信息，代码如下：

```
Field f = String.class.getDeclaredField("value");
f.getName(); // "value"
f.getType(); // class [B 表示byte[]类型
int m = f.getModifiers();
Modifier.isFinal(m); // true
Modifier.isPublic(m); // false
Modifier.isProtected(m); // false
Modifier.isPrivate(m); // true
Modifier.isStatic(m); // false
```

获取字段值

利用反射拿到字段的一个`Field`实例只是第一步，我们还可以拿到一个实例对应的该字段的值。

例如，对于一个`Person`实例，我们可以先拿到`name`字段对应的`Field`，再获取这个实例的`name`字段的值：

```
// reflection
import java.lang.reflect.Field;
public class Main {

    public static void main(String[] args) throws Exception {
        Object p = new Person("Xiao Ming");
        Class c = p.getClass();
        Field f = c.getDeclaredField("name");
        Object value = f.get(p);
        System.out.println(value); // "Xiao Ming"
    }
}

class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }
}
```

▶ Run

上述代码先获取 `Class` 实例，再获取 `Field` 实例，然后，用 `Field.get(Object)` 获取指定实例的指定字段的值。

运行代码，如果不出意外，会得到一个 `IllegalAccessException` ，这是因为 `name` 被定义为一个 `private` 字段，正常情况下， `Main` 类无法访问 `Person` 类的 `private` 字段。要修复错误，可以将 `private` 改为 `public` ，或者，在调用 `Object value = f.get(p);` 前，先写一句：

```
1.  f.setAccessible(true);
```

调用 `Field.setAccessible(true)` 的意思是，别管这个字段是不是 `public` ，一律允许访问。

可以试着加上上述语句，再运行代码，就可以打印出 `private` 字段的值。

有童鞋会问：如果使用反射可以获取 `private` 字段的值，那么类的封装还有什么意义？

答案是正常情况下，我们总是通过 `p.name` 来访问 `Person` 的 `name` 字段，编译器会根据 `public` 、 `protected` 和 `private` 决定是否允许访问字段，这样就达到了数据封装的目的。

而反射是一种非常规的用法，使用反射，首先代码非常繁琐，其次，它更多地是给工具或者底层框架来使用，目的是在不知道目标实例任何信息的情况下，获取特定字段的值。

此外， `setAccessible(true)` 可能会失败。如果JVM运行期存在 `SecurityManager` ，那么它会根据规则进行检查，有可能阻止 `setAccessible(true)` 。例如，某个 `SecurityManager` 可能不允许对 `java` 和 `javax` 开头的 `package` 的类调用 `setAccessible(true)` ，这样可以保证JVM核心

库的安全。

设置字段值

通过Field实例既然可以获取到指定实例的字段值，自然也可以设置字段的值。

设置字段值是通过 `Field.set(Object, Object)` 实现的，其中第一个 `Object` 参数是指定的实例，第二个 `Object` 参数是待修改的值。示例代码如下：

```
// reflection
import java.lang.reflect.Field;
public class Main {

    public static void main(String[] args) throws Exception {
        Person p = new Person("Xiao Ming");
        System.out.println(p.getName()); // "Xiao Ming"
        Class c = p.getClass();
        Field f = c.getDeclaredField("name");
        f.setAccessible(true);
        f.set(p, "Xiao Hong");
        System.out.println(p.getName()); // "Xiao Hong"
    }
}

class Person {
    private String name;

    public Person(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }
}
```

▶ Run

运行上述代码，打印的 `name` 字段从 `Xiao Ming` 变成了 `Xiao Hong` ，说明通过反射可以直接修改字段的值。

同样的，修改非 `public` 字段，需要首先调用 `setAccessible(true)` 。

练习

利用反射给字段赋值：下载练习：`reflect-field` （推荐使用IDE练习插件快速下载）

小结

Java的反射API提供的 `Field` 类封装了字段的所有信息：

通过 `Class` 实例的方法可以获取 `Field` 实例： `getField()` ， `getFields()` ， `getDeclaredField()` ， `getDeclaredFields()` ；

通过Field实例可以获取字段信息： `getName()` ， `getType()` ， `getModifiers()` ；

通过Field实例可以读取或设置某个对象的字段，如果存在访问限制，要首先调用 `setAccessible(true)` 来访问非 `public` 字段。

通过反射读写字段是一种非常规方法，它会破坏对象的封装。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





调用方法

我们已经能通过 `Class` 实例获取所有 `Field` 对象，同样的，可以通过 `Class` 实例获取所有 `Method` 信息。 `Class` 类提供了以下几个方法来获取 `Method` ：

- `Method getMethod(name, Class…)` ：获取某个 `public` 的 `Method` （包括父类）
- `Method getDeclaredMethod(name, Class…)` ：获取当前类的某个 `Method` （不包括父类）
- `Method[] getMethods()` ：获取所有 `public` 的 `Method` （包括父类）
- `Method[] getDeclaredMethods()` ：获取当前类的所有 `Method` （不包括父类）

我们来看一下示例代码：

```
// reflection
public class Main {
    public static void main(String[] args) throws Exception {
        Class stdClass = Student.class;
        // 获取public方法getScore，参数为String:
        System.out.println(stdClass.getMethod("getScore", String.class));
        // 获取继承的public方法getName，无参数:
        System.out.println(stdClass.getMethod("getName"));
        // 获取private方法getGrade，参数为int:
        System.out.println(stdClass.getDeclaredMethod("getGrade", int.class));
    }
}

class Student extends Person {
    public int getScore(String type) {
        return 99;
    }
    private int getGrade(int year) {
        return 1;
    }
}

class Person {
    public String getName() {
        return "Person";
    }
}
```

▶ Run

上述代码首先获取 `Student` 的 `Class` 实例，然后，分别获取 `public` 方法、继承的 `public` 方法以及 `private` 方法，打印出的 `Method` 类似：

```
1. public int Student.getScore(java.lang.String)
2. public java.lang.String Person.getName()
3. private int Student.getGrade(int)
```

一个 `Method` 对象包含一个方法的所有信息：

- `getName()`：返回方法名称，例如：`"getScore"`；
- `getReturnType()`：返回方法返回值类型，也是一个Class实例，例如：`String.class`；
- `getParameterTypes()`：返回方法的参数类型，是一个Class数组，例如：`{String.class, int.class}`；
- `getModifiers()`：返回方法的修饰符，它是一个 `int`，不同的bit表示不同的含义。

调用方法

当我们获取到一个 `Method` 对象时，就可以对它进行调用。我们以下面的代码为例：

```
1. String s = "Hello world";
2. String r = s.substring(6); // "world"
```

如果用反射来调用 `substring` 方法，需要以下代码：

```
// reflection
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        // String对象:
        String s = "Hello world";
        // 获取String substring(int)方法，参数为int:
        Method m = String.class.getMethod("substring", int.class);
        // 在s对象上调用该方法并获取结果:
        String r = (String) m.invoke(s, 6);
        // 打印调用结果:
        System.out.println(r);
    }
}
```

▶ Run

注意到 `substring()` 有两个重载方法，我们获取的是 `String substring(int)` 这个方法。思考一下如何获取 `String substring(int, int)` 方法。

对 `Method` 实例调用 `invoke` 就相当于调用该方法， `invoke` 的第一个参数是对象实例，即在哪个实例上调用该方法，后面的可变参数要与方法参数一致，否则将报错。

调用静态方法

如果获取到的Method表示一个静态方法，调用静态方法时，由于无需指定实例对象，所以 `invoke` 方法传入的第一个参数永远为 `null`。我们以 `Integer.parseInt(String)` 为例：

```
// reflection
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        // 获取Integer.parseInt(String)方法，参数为String:
        Method m = Integer.class.getMethod("parseInt", String.class);
        // 调用该静态方法并获取结果:
        Integer n = (Integer) m.invoke(null, "12345");
        // 打印调用结果:
        System.out.println(n);
    }
}
```

▶ Run

调用非public方法

和Field类似，对于非public方法，我们虽然可以通过 `Class.getDeclaredMethod()` 获取该方法实例，但直接对其调用将得到一个 `IllegalAccessException` 。为了调用非public方法，我们通过 `Method.setAccessible(true)` 允许其调用：

```
// reflection
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        Person p = new Person();
        Method m = p.getClass().getDeclaredMethod("setName", String.class);
        m.setAccessible(true);
        m.invoke(p, "Bob");
        System.out.println(p.name);
    }
}

class Person {
    String name;
    private void setName(String name) {
        this.name = name;
    }
}
```

▶ Run

此外， `setAccessible(true)` 可能会失败。如果JVM运行期存在 `SecurityManager` ，那么它会根据规则进行检查，有可能阻止 `setAccessible(true)` 。例如，某个 `SecurityManager` 可能不允许对 `java` 和 `javax` 开头的 `package` 的类调用 `setAccessible(true)` ，这样可以保证JVM核心库的安全。

多态

我们来考察这样一种情况：一个 `Person` 类定义了 `hello()` 方法，并且它的子类 `Student` 也覆写了 `hello()` 方法，那么，从 `Person.class` 获取的 `Method` ，作用于 `Student` 实例时，调用的方法到底是哪个？

```
// reflection
import java.lang.reflect.Method;

public class Main {
    public static void main(String[] args) throws Exception {
        // 获取Person的hello方法:
        Method h = Person.class.getMethod("hello");
        // 对Student实例调用hello方法:
        h.invoke(new Student());
    }
}

class Person {
    public void hello() {
        System.out.println("Person:hello");
    }
}

class Student extends Person {
    public void hello() {
        System.out.println("Student:hello");
    }
}
```

▶ Run

运行上述代码，发现打印出的是 `Student:hello` ，因此，使用反射调用方法时，仍然遵循多态原则：即总是调用实际类型的覆写方法（如果存在）。上述的反射代码：

```
1. Method m = Person.class.getMethod("hello");
2. m.invoke(new Student());
```

实际上相当于：

```
1. Person p = new Student();
2. p.hello();
```

练习

利用反射调用方法：下载练习：`reflect-method` （推荐使用IDE练习插件快速下载）

小结

Java的反射API提供的Method对象封装了方法的所有信息：

通过 `Class` 实例的方法可以获取 `Method` 实例： `getMethod()` ， `getMethods()` ， `getDeclaredMethod()` ， `getDeclaredMethods()` ；

通过 `Method` 实例可以获取方法信息： `getName()` ， `getReturnType()` ， `getParameterTypes()` ， `getModifiers()` ；

通过 `Method` 实例可以调用某个对象的方法： `Object invoke(Object instance, Object… parameters)` ；

通过设置 `setAccessible(true)` 来访问非 `public` 方法；

通过反射调用方法时，仍然遵循多态原则。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





调用构造方法

我们通常使用 `new` 操作符创建新的实例：

```
1.  Person p = new Person();
```

如果通过反射来创建新的实例，可以调用Class提供的newInstance()方法：

```
1.  Person p = Person.class.newInstance();
```

调用Class.newInstance()的局限是，它只能调用该类的public无参数构造方法。如果构造方法带有参数，或者不是public，就无法直接通过Class.newInstance()来调用。

为了调用任意的构造方法，Java的反射API提供了Constructor对象，它包含一个构造方法的所有信息，可以创建一个实例。Constructor对象和Method非常类似，不同之处仅在于它是一个构造方法，并且，调用结果总是返回实例：

```
import java.lang.reflect.Constructor;
public class Main {
    public static void main(String[] args) throws Exception {
        // 获取构造方法Integer(int):
        Constructor cons1 = Integer.class.getConstructor(int.class);
        // 调用构造方法:
        Integer n1 = (Integer) cons1.newInstance(123);
        System.out.println(n1);

        // 获取构造方法Integer(String)
        Constructor cons2 = Integer.class.getConstructor(String.class);
        Integer n2 = (Integer) cons2.newInstance("456");
        System.out.println(n2);
    }
}
```

▶ Run

通过Class实例获取Constructor的方法如下：

- `getConstructor(Class…)`：获取某个 `public` 的 `Constructor`；
- `getDeclaredConstructor(Class…)`：获取某个 `Constructor`；
- `getConstructors()`：获取所有 `public` 的 `Constructor`；
- `getDeclaredConstructors()`：获取所有 `Constructor`。

注意 `Constructor` 总是当前类定义的构造方法，和父类无关，因此不存在多态的问题。

调用非 `public` 的 `Constructor` 时，必须首先通过 `setAccessible(true)` 设置允许访

问。 `setAccessible(true)` 可能会失败。

小结

`Constructor` 对象封装了构造方法的所有信息；

通过 `Class` 实例的方法可以获取 `Constructor` 实例： `getConstructor()` ， `getConstructors()` ， `getDeclaredConstructor()` ， `getDeclaredConstructors()` ；

通过 `Constructor` 实例可以创建一个实例对象： `newInstance(Object… parameters)` ；通过设置 `setAccessible(true)` 来访问非 `public` 构造方法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





获取继承关系

当我们获取到某个 `Class` 对象时，实际上就获取到了一个类的类型：

```
1.  Class cls = String.class; // 获取到String的Class
```

还可以用实例的 `getClass()` 方法获取：

```
1.  String s = "";
2.  Class cls = s.getClass(); // s是String，因此获取到String的Class
```

最后一种获取 `Class` 的方法是通过 `Class.forName("")` ，传入 `Class` 的完整类名获取：

```
1.  Class s = Class.forName("java.lang.String");
```

这三种方式获取的 `Class` 实例都是同一个实例，因为JVM对每个加载的 `Class` 只创建一个 `Class` 实例来表示它的类型。

获取父类的Class

有了 `Class` 实例，我们还可以获取它的父类的 `Class` ：

```
// reflection
public class Main {
    public static void main(String[] args) throws Exception {
        Class i = Integer.class;
        Class n = i.getSuperclass();
        System.out.println(n);
        Class o = n.getSuperclass();
        System.out.println(o);
        System.out.println(o.getSuperclass());
    }
}
```

▶ Run

运行上述代码，可以看到， `Integer` 的父类类型是 `Number` ， `Number` 的父类是 `Object` ， `Object` 的父类是 `null` 。除 `Object` 外，其他任何非 `interface` 的 `Class` 都必定存在一个父类类型。

获取interface

由于一个类可能实现一个或多个接口，通过 `Class` 我们就可以查询到实现的接口类型。例如，查

询 `Integer` 实现的接口：

```
// reflection
import java.lang.reflect.Method;
```

```
public class Main {
    public static void main(String[] args) throws Exception {
        Class s = Integer.class;
        Class[] is = s.getInterfaces();
        for (Class i : is) {
            System.out.println(i);
        }
    }
}
```

▶ Run

运行上述代码可知， `Integer` 实现的接口有：

- java.lang.Comparable
- java.lang.constant.Constable
- java.lang.constant.ConstantDesc

要特别注意： `getInterfaces()` 只返回当前类直接实现的接口类型，并不包括其父类实现的接口类型：

```
// reflection
import java.lang.reflect.Method;
```

```
public class Main {
    public static void main(String[] args) throws Exception {
        Class s = Integer.class.getSuperclass();
        Class[] is = s.getInterfaces();
        for (Class i : is) {
            System.out.println(i);
        }
    }
}
```

▶ Run

`Integer` 的父类是 `Number` ， `Number` 实现的接口是 `java.io.Serializable` 。

此外，对所有 `interface` 的 `Class` 调用 `getSuperclass()` 返回的是 `null` ，获取接口的父接口要用 `getInterfaces()` ：

```
    System.out.println(java.io.DataInputStream.class.getSuperclass()); //
1.  java.io.FilterInputStream, 因为DataInputStream继承自FilterInputStream
```

```
    System.out.println(java.io.Closeable.class.getSuperclass()); // null, 对接口调用
2.  getSuperclass()总是返回null, 获取接口的父接口要用getInterfaces()
```

如果一个类没有实现任何 `interface` ，那么 `getInterfaces()` 返回空数组。

继承关系

当我们判断一个实例是否是某个类型时，正常情况下，使用 `instanceof` 操作符：

```
1.  Object n = Integer.valueOf(123);
2.  boolean isDouble = n instanceof Double; // false
3.  boolean isInteger = n instanceof Integer; // true
4.  boolean isNumber = n instanceof Number; // true
5.  boolean isSerializable = n instanceof java.io.Serializable; // true
```

如果是两个 `Class` 实例，要判断一个向上转型是否成立，可以调用 `isAssignableFrom()` ：

```
1.  // Integer i = ?
    Integer.class.isAssignableFrom(Integer.class); // true, 因为Integer可以赋值给
2.  Integer
3.  // Number n = ?
    Number.class.isAssignableFrom(Integer.class); // true, 因为Integer可以赋值给
4.  Number
5.  // Object o = ?
    Object.class.isAssignableFrom(Integer.class); // true, 因为Integer可以赋值给
6.  Object
7.  // Integer i = ?
    Integer.class.isAssignableFrom(Number.class); // false, 因为Number不能赋值给
8.  Integer
```

小结

通过 `Class` 对象可以获取继承关系：

- `Class getSuperclass()` ：获取父类类型；
- `Class[] getInterfaces()` ：获取当前类实现的所有接口。

通过 `Class` 对象的 `isAssignableFrom()` 方法可以判断一个向上转型是否可以实现。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





动态代理

我们来比较Java的 `class` 和 `interface` 的区别：

- 可以实例化 `class` （非 `abstract` ）；
- 不能实例化 `interface` 。

所有 `interface` 类型的变量总是通过向上转型并指向某个实例的：

```
1. CharSequence cs = new StringBuilder();
```

有没有可能不编写实现类，直接在运行期创建某个 `interface` 的实例呢？

这是可能的，因为Java标准库提供了一种动态代理（Dynamic Proxy）的机制：可以在运行期动态创建某个 `interface` 的实例。

什么叫运行期动态创建？听起来好像很复杂。所谓动态代理，是和静态相对应的。我们来看静态代码怎么写：

定义接口：

```
1. public interface Hello {
2.     void morning(String name);
3. }
```

编写实现类：

```
1. public class HelloWorld implements Hello {
2.     public void morning(String name) {
3.         System.out.println("Good morning, " + name);
4.     }
5. }
```

创建实例，转型为接口并调用：

```
1. Hello hello = new HelloWorld();
2. hello.morning("Bob");
```

这种方式就是我们通常编写代码的方式。

还有一种方式是动态代码，我们仍然先定义了接口 `Hello` ，但是我们并不去编写实现类，而是直接通过JDK提供的一个 `Proxy.newProxyInstance()` 创建了一个 `Hello` 接口对象。这种没有实现类但

是在运行期动态创建了一个接口对象的方式，我们称为动态代码。JDK提供的动态创建接口对象的方式，就叫动态代理。

一个最简单的动态代理实现如下：

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;
public class Main {
    public static void main(String[] args) {
        InvocationHandler handler = new InvocationHandler() {
            @Override
            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
                System.out.println(method);
                if (method.getName().equals("morning")) {
                    System.out.println("Good morning, " + args[0]);
                }
                return null;
            }
        };
        Hello hello = (Hello) Proxy.newProxyInstance(
            Hello.class.getClassLoader(), // 传入ClassLoader
            new Class[] { Hello.class }, // 传入要实现的接口
            handler); // 传入处理调用方法的InvocationHandler
        hello.morning("Bob");
    }
}

interface Hello {
    void morning(String name);
}
```

▶ Run

在运行期动态创建一个 `interface` 实例的方法如下：

- 定义一个 `InvocationHandler` 实例，它负责实现接口的方法调用；
- 通过 `Proxy.newProxyInstance()` 创建 `interface` 实例，它需要3个参数：
  - 使用的 `ClassLoader` ，通常就是接口类的 `ClassLoader` ；
  - 需要实现的接口数组，至少需要传入一个接口进去；
  - 用来处理接口方法调用的 `InvocationHandler` 实例。
- 将返回的 `Object` 强制转型为接口。动态代理实际上是JDK在运行期动态创建class字节码并加载的过程，它并没有什么黑魔法，把上面的动态代理改写为静态实现类大概长这样：

```
1.  public class HelloDynamicProxy implements Hello {
2.      InvocationHandler handler;
3.      public HelloDynamicProxy(InvocationHandler handler) {
4.          this.handler = handler;
5.      }
```

```
6.      public void morning(String name) {
7.          handler.invoke(
8.              this,
9.              Hello.class.getMethod("morning"),
10.             new Object[] { name });
11.     }
12. }
```

其实就是JDK帮我们自动编写了一个上述类（不需要源码，可以直接生成字节码），并不存在可以直接实例化接口的黑魔法。

小结

Java标准库提供了动态代理功能，允许在运行期动态创建一个接口的实例；

动态代理是通过 `Proxy` 创建代理对象，然后将接口方法“代理”给 `InvocationHandler` 完成的。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





注解

本节我们将介绍Java程序的一种特殊“注释”——注解（Annotation）。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用注解

什么是注解（Annotation）？注解是放在Java源码的类、方法、字段、参数前的一种特殊“注释”：

```
// this is a component:
@Resource("hello")
public class Hello {
    @Inject
    int n;

    @PostConstruct
    public void hello(@Param String name) {
        System.out.println(name);
    }

    @Override
    public String toString() {
        return "Hello";
    }
}
```

注释会被编译器直接忽略，注解则可以被编译器打包进入class文件，因此，注解是一种用作标注的“元数据”。

注解的作用

从JVM的角度看，注解本身对代码逻辑没有任何影响，如何使用注解完全由工具决定。

Java的注解可以分为三类：

第一类是由编译器使用的注解，例如：

- `@Override`：让编译器检查该方法是否正确地实现了覆写；
- `@SuppressWarnings`：告诉编译器忽略此处代码产生的警告。

这类注解不会被编译进入`.class`文件，它们在编译后就被编译器扔掉了。

第二类是由工具处理`.class`文件使用的注解，比如有些工具会在加载class的时候，对class做动态修改，实现一些特殊的功能。这类注解会被编译进入`.class`文件，但加载结束后并不会存在于内存中。这类注解只被一些底层库使用，一般我们不必自己处理。

第三类是在程序运行期能够读取的注解，它们在加载后一直存在于JVM中，这也是最常用的注解。例如，一个配置了`@PostConstruct`的方法会在调用构造方法后自动被调用（这是Java代码读取该注

解实现的功能，JVM并不会识别该注解）。

定义一个注解时，还可以定义配置参数。配置参数可以包括：

- 所有基本类型；
- String；
- 枚举类型；
- 基本类型、String以及枚举的数组。

因为配置参数必须是常量，所以，上述限制保证了注解在定义时就已经确定了每个参数的值。

注解的配置参数可以有默认值，缺少某个配置参数时将使用默认值。

此外，大部分注解会有一个名为 `value` 的配置参数，对此参数赋值，可以只写常量，相当于省略了value参数。

如果只写注解，相当于全部使用默认值。

举个栗子，对以下代码：

```
1.  public class Hello {
2.      @Check(min=0, max=100, value=55)
3.      public int n;
4.
5.      @Check(value=99)
6.      public int p;
7.
8.      @Check(99) // @Check(value=99)
9.      public int x;
10.
11.     @Check
12.     public int y;
13. }
```

`@Check` 就是一个注解。第一个 `@Check(min=0, max=100, value=55)` 明确定义了三个参数，第二个 `@Check(value=99)` 只定义了一个 `value` 参数，它实际上和 `@Check(99)` 是完全一样的。最后一个 `@Check` 表示所有参数都使用默认值。

小结

注解（Annotation）是Java语言用于工具处理的标注：

注解可以配置参数，没有指定配置的参数使用默认值；

如果参数名称是 `value` ，且只有一个参数，那么可以省略参数名称。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





定义注解

Java语言使用 `@interface` 语法来定义注解（ `Annotation` ），它的格式如下：

```
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

注解的参数类似无参数方法，可以用 `default` 设定一个默认值（强烈推荐）。最常用的参数应当命名为 `value` 。

元注解

有一些注解可以修饰其他注解，这些注解就称为元注解（meta annotation）。Java标准库已经定义了一些元注解，我们只需要使用元注解，通常不需要自己去编写元注解。

@Target

最常用的元注解是 `@Target` 。使用 `@Target` 可以定义 `Annotation` 能够被应用于源码的哪些位置：

- 类或接口： `ElementType.TYPE` ；
- 字段： `ElementType.FIELD` ；
- 方法： `ElementType.METHOD` ；
- 构造方法： `ElementType.CONSTRUCTOR` ；
- 方法参数： `ElementType.PARAMETER` 。

例如，定义注解 `@Report` 可用在方法上，我们必须添加一个 `@Target(ElementType.METHOD)` ：

```
@Target(ElementType.METHOD)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

定义注解 `@Report` 可用在方法或字段上，可以把 `@Target` 注解参数变为数组 `{ ElementType.METHOD, ElementType.FIELD }` ：

```
@Target({
```

```
    ElementType.METHOD,
    ElementType.FIELD
})
public @interface Report {
    ...
}
```

实际上 `@Target` 定义的 `value` 是 `ElementType[]` 数组，只有一个元素时，可以省略数组的写法。

@Retention

另一个重要的元注解 `@Retention` 定义了 `Annotation` 的生命周期：

- 仅编译期：`RetentionPolicy.SOURCE`；
- 仅class文件：`RetentionPolicy.CLASS`；
- 运行期：`RetentionPolicy.RUNTIME`。

如果 `@Retention` 不存在，则该 `Annotation` 默认为 `CLASS`。因为通常我们自定义的 `Annotation` 都是 `RUNTIME`，所以，务必要加上 `@Retention(RetentionPolicy.RUNTIME)` 这个元注解：

```
@Retention(RetentionPolicy.RUNTIME)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

@Repeatable

使用 `@Repeatable` 这个元注解可以定义 `Annotation` 是否可重复。这个注解应用不是特别广泛。

```
@Repeatable
@Target(ElementType.TYPE)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

经过 `@Repeatable` 修饰后，在某个类型声明处，就可以添加多个 `@Report` 注解：

```
@Report(type=1, level="debug")
@Report(type=2, level="warning")
public class Hello {
}
```

@Inherited

使用 `@Inherited` 定义子类是否可继承父类定义的 `Annotation`。`@Inherited` 仅针对 `@Target(ElementType.TYPE)` 类型的 `annotation` 有效，并且仅针对 `class` 的继承，对 `interface` 的继承无效：

```
@Inherited
@Target(ElementType.TYPE)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

在使用的时候，如果一个类用到了 `@Report`：

```
@Report(type=1)
public class Person {
}
```

则它的子类默认也定义了该注解：

```
public class Student extends Person {
}
```

如何定义Annotation

我们总结一下定义 `Annotation` 的步骤：

第一步，用 `@interface` 定义注解：

```
public @interface Report {
}
```

第二步，添加参数、默认值：

```
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

把最常用的参数定义为 `value()` ，推荐所有参数都尽量设置默认值。

第三步，用元注解配置注解：

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}
```

其中，必须设置 `@Target` 和 `@Retention` ， `@Retention` 一般设置为 `RUNTIME` ，因为我们自定义的注解通常要求在运行期读取。一般情况下，不必写 `@Inherited` 和 `@Repeatable` 。

小结

Java使用 `@interface` 定义注解：

可定义多个参数和默认值，核心参数使用 `value` 名称；

必须设置 `@Target` 来指定 `Annotation` 可以应用的范围；

应当设置 `@Retention(RetentionPolicy.RUNTIME)` 便于运行期读取该 `Annotation` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





处理注解

Java的注解本身对代码逻辑没有任何影响。根据 `@Retention` 的配置：

- `SOURCE` 类型的注解在编译期就被丢掉了；
- `CLASS` 类型的注解仅保存在class文件中，它们不会被加载进JVM；
- `RUNTIME` 类型的注解会被加载进JVM，并且在运行期可以被程序读取。

如何使用注解完全由工具决定。 `SOURCE` 类型的注解主要由编译器使用，因此我们一般只使用，不编写。 `CLASS` 类型的注解主要由底层工具库使用，涉及到class的加载，一般我们很少用到。只有 `RUNTIME` 类型的注解不但要使用，还经常需要编写。

因此，我们只讨论如何读取 `RUNTIME` 类型的注解。

因为注解定义后也是一种 `class` ，所有的注解都继承自 `java.lang.annotation.Annotation` ，因此，读取注解，需要使用反射API。

Java提供的使用反射API读取 `Annotation` 的方法包括：

判断某个注解是否存在于 `Class` 、 `Field` 、 `Method` 或 `Constructor` ：

- `Class.isAnnotationPresent(Class)`
- `Field.isAnnotationPresent(Class)`
- `Method.isAnnotationPresent(Class)`
- `Constructor.isAnnotationPresent(Class)`

例如：

```
// 判断@Report是否存在于Person类:
Person.class.isAnnotationPresent(Report.class);
```

使用反射API读取Annotation：

- `Class.getAnnotation(Class)`
- `Field.getAnnotation(Class)`
- `Method.getAnnotation(Class)`
- `Constructor.getAnnotation(Class)`

例如：

```
// 获取Person定义的@Report注解:
Report report = Person.class.getAnnotation(Report.class);
int type = report.type();
```

```
String level = report.level();
```

使用反射API读取 `Annotation` 有两种方法。方法一是先判断 `Annotation` 是否存在，如果存在，就直接读取：

```
Class cls = Person.class;
if (cls.isAnnotationPresent(Report.class)) {
    Report report = cls.getAnnotation(Report.class);
    ...
}
```

第二种方法是直接读取 `Annotation` ，如果 `Annotation` 不存在，将返回 `null` ：

```
Class cls = Person.class;
Report report = cls.getAnnotation(Report.class);
if (report != null) {
   ...
}
```

读取方法、字段和构造方法的 `Annotation` 和Class类似。但要读取方法参数的 `Annotation` 就比较麻烦一点，因为方法参数本身可以看成一个数组，而每个参数又可以定义多个注解，所以，一次获取方法参数的所有注解就必须用一个二维数组来表示。例如，对于以下方法定义的注解：

```
public void hello(@NotNull @Range(max=5) String name, @NotNull String prefix) {
}
```

要读取方法参数的注解，我们先用反射获取 `Method` 实例，然后读取方法参数的所有注解：

```
// 获取Method实例:
Method m = ...
// 获取所有参数的Annotation:
Annotation[][] annos = m.getParameterAnnotations();
// 第一个参数（索引为0）的所有Annotation:
Annotation[] annosOfName = annos[0];
for (Annotation anno : annosOfName) {
    if (anno instanceof Range) { // @Range注解
        Range r = (Range) anno;
    }
    if (anno instanceof NotNull) { // @NotNull注解
        NotNull n = (NotNull) anno;
    }
```

```
14. }
```

使用注解

注解如何使用，完全由程序自己决定。例如，JUnit是一个测试框架，它会自动运行所有标记为 `@Test` 的方法。

我们来看一个 `@Range` 注解，我们希望用它来定义一个 `String` 字段的规则：字段长度满足 `@Range` 的参数定义：

```
1. @Retention(RetentionPolicy.RUNTIME)
2. @Target(ElementType.FIELD)
3. public @interface Range {
4.     int min() default 0;
5.     int max() default 255;
6. }
```

在某个JavaBean中，我们可以使用该注解：

```
1. public class Person {
2.     @Range(min=1, max=20)
3.     public String name;
4.
5.     @Range(max=10)
6.     public String city;
7. }
```

但是，定义了注解，本身对程序逻辑没有任何影响。我们必须自己编写代码来使用注解。这里，我们编写一个 `Person` 实例的检查方法，它可以检查 `Person` 实例的 `String` 字段长度是否满足 `@Range` 的定义：

```
    void check(Person person) throws IllegalArgumentException,
1. ReflectiveOperationException {
2.     // 遍历所有Field:
3.     for (Field field : person.getClass().getFields()) {
4.         // 获取Field定义的@Range:
5.         Range range = field.getAnnotation(Range.class);
6.         // 如果@Range存在:
7.         if (range != null) {
8.             // 获取Field的值:
9.             Object value = field.get(person);
```

```
            // 如果值是String:
            if (value instanceof String) {
                String s = (String) value;
                // 判断值是否满足@Range的min/max:
                if (s.length() < range.min() || s.length() > range.max()) {
                    throw new IllegalArgumentException("Invalid field: " +
field.getName());
                }
            }
        }
    }
}
```

这样一来，我们通过 `@Range` 注解，配合 `check()` 方法，就可以完成 `Person` 实例的检查。注意检查逻辑完全是我们自己编写的，JVM不会自动给注解添加任何额外的逻辑。

练习

使用 `@Range` 注解来检查Java Bean的字段。如果字段类型是 `String` ，就检查 `String` 的长度，如果字段是 `int` ，就检查 `int` 的范围。

下载练习：annotation-range-check （推荐使用IDE练习插件快速下载）

小结

可以在运行期通过反射读取 `RUNTIME` 类型的注解，注意千万不要漏写 `@Retention(RetentionPolicy.RUNTIME)` ，否则运行期无法读取到该注解。

可以通过程序处理注解来实现相应的功能：

- 对JavaBean的属性值按规则进行检查；
- JUnit会自动运行 `@Test` 标记的测试方法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





泛型

泛型是一种“代码模板”，可以用一套代码套用各种类型。



本节我们详细讨论Java的泛型编程。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





什么是泛型

在讲解什么是泛型之前，我们先观察Java标准库提供的 `ArrayList` ，它可以看作“可变长度”的数组，因为用起来比数组更方便。

实际上 `ArrayList` 内部就是一个 `Object[]` 数组，配合存储一个当前分配的长度，就可以充当“可变数组”：

```
public class ArrayList {
    private Object[] array;
    private int size;
    public void add(Object e) {...}
    public void remove(int index) {...}
    public Object get(int index) {...}
}
```

如果用上述 `ArrayList` 存储 `String` 类型，会有这么几个缺点：

- 需要强制转型；
- 不方便，易出错。

例如，代码必须这么写：

```
ArrayList list = new ArrayList();
list.add("Hello");
// 获取到Object，必须强制转型为String:
String first = (String) list.get(0);
```

很容易出现ClassCastException，因为容易“误转型”：

```
list.add(new Integer(123));
// ERROR: ClassCastException:
String second = (String) list.get(1);
```

要解决上述问题，我们可以为 `String` 单独编写一种 `ArrayList` ：

```
public class StringArrayList {
    private String[] array;
    private int size;
    public void add(String e) {...}
    public void remove(int index) {...}
```

```
    public String get(int index) {...}
}
```

这样一来，存入的必须是 `String` ，取出的也一定是 `String` ，不需要强制转型，因为编译器会强制检查放入的类型：

```
StringArrayList list = new StringArrayList();
list.add("Hello");
String first = list.get(0);
// 编译错误：不允许放入非String类型：
list.add(new Integer(123));
```

问题暂时解决。

然而，新的问题是，如果要存储 `Integer` ，还需要为 `Integer` 单独编写一种 `ArrayList` ：

```
public class IntegerArrayList {
    private Integer[] array;
    private int size;
    public void add(Integer e) {...}
    public void remove(int index) {...}
    public Integer get(int index) {...}
}
```

实际上，还需要为其他所有class单独编写一种 `ArrayList` ：

- LongArrayList
- DoubleArrayList
- PersonArrayList
- …

这是不可能的，JDK的class就有上千个，而且它还不知道其他人编写的class。

为了解决新的问题，我们必须把 `ArrayList` 变成一种模板： `ArrayList<T>` ，代码如下：

```
public class ArrayList<T> {
    private T[] array;
    private int size;
    public void add(T e) {...}
    public void remove(int index) {...}
    public T get(int index) {...}
}
```

`T` 可以是任何class。这样一来，我们就实现了：编写一次模版，可以创建任意类型的 `ArrayList` ：

```
// 创建可以存储String的ArrayList:
ArrayList<String> strList = new ArrayList<String>();
// 创建可以存储Float的ArrayList:
ArrayList<Float> floatList = new ArrayList<Float>();
// 创建可以存储Person的ArrayList:
ArrayList<Person> personList = new ArrayList<Person>();
```

因此，泛型就是定义一种模板，例如 `ArrayList<T>` ，然后在代码中为用到的类创建对应的 `ArrayList<类型>` ：

```
ArrayList<String> strList = new ArrayList<String>();
```

由编译器针对类型作检查：

```
strList.add("hello"); // OK
String s = strList.get(0); // OK
strList.add(new Integer(123)); // compile error!
Integer n = strList.get(0); // compile error!
```

这样一来，既实现了编写一次，万能匹配，又通过编译器保证了类型安全：这就是泛型。

向上转型

在Java标准库中的 `ArrayList<T>` 实现了 `List<T>` 接口，它可以向上转型为 `List<T>` ：

```
public class ArrayList<T> implements List<T> {
    ...
}

List<String> list = new ArrayList<String>();
```

即类型 `ArrayList<T>` 可以向上转型为 `List<T>` 。

要特别注意：不能把 `ArrayList<Integer>` 向上转型为 `ArrayList<Number>` 或 `List<Number>` 。

这是为什么呢？假设 `ArrayList<Integer>` 可以向上转型为 `ArrayList<Number>` ，观察一下代码：

```
// 创建ArrayList<Integer>类型：
ArrayList<Integer> integerList = new ArrayList<Integer>();
// 添加一个Integer：
integerList.add(new Integer(123));
// “向上转型”为ArrayList<Number>：
ArrayList<Number> numberList = integerList;
// 添加一个Float，因为Float也是Number：
numberList.add(new Float(12.34));
// 从ArrayList<Integer>获取索引为1的元素（即添加的Float）：
Integer n = integerList.get(1); // ClassCastException!
```

我们把一个 `ArrayList<Integer>` 转型为 `ArrayList<Number>` 类型后，这个 `ArrayList<Number>` 就可以接受 `Float` 类型，因为 `Float` 是 `Number` 的子类。但是， `ArrayList<Number>` 实际上和 `ArrayList<Integer>` 是同一个对象，也就是 `ArrayList<Integer>` 类型，它不可能接受 `Float` 类型，所以在获取 `Integer` 的时候将产生 `ClassCastException` 。

实际上，编译器为了避免这种错误，根本就不允许把 `ArrayList<Integer>` 转型为 `ArrayList<Number>` 。

ArrayList<Integer>和ArrayList<Number>两者完全没有继承关系。

小结

泛型就是编写模板代码来适应任意类型；

泛型的好处是使用时不必对类型进行强制转换，它通过编译器对类型进行检查；

注意泛型的继承关系：可以把 `ArrayList<Integer>` 向上转型为 `List<Integer>` （ `T` 不能变！），但不能把 `ArrayList<Integer>` 向上转型为 `ArrayList<Number>` （ `T` 不能变成父类）。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用泛型

使用 `ArrayList` 时，如果不定义泛型类型时，泛型类型实际上就是 `Object` ：

```
// 编译器警告:
List list = new ArrayList();
list.add("Hello");
list.add("World");
String first = (String) list.get(0);
String second = (String) list.get(1);
```

此时，只能把 `<T>` 当作 `Object` 使用，没有发挥泛型的优势。

当我们定义泛型类型 `<String>` 后， `List<T>` 的泛型接口变为强类型 `List<String>` ：

```
// 无编译器警告:
List<String> list = new ArrayList<String>();
list.add("Hello");
list.add("World");
// 无强制转型:
String first = list.get(0);
String second = list.get(1);
```

当我们定义泛型类型 `<Number>` 后， `List<T>` 的泛型接口变为强类型 `List<Number>` ：

```
List<Number> list = new ArrayList<Number>();
list.add(new Integer(123));
list.add(new Double(12.34));
Number first = list.get(0);
Number second = list.get(1);
```

编译器如果能自动推断出泛型类型，就可以省略后面的泛型类型。例如，对于下面的代码：

```
List<Number> list = new ArrayList<Number>();
```

编译器看到泛型类型 `List<Number>` 就可以自动推断出后面的 `ArrayList<T>` 的泛型类型必须是 `ArrayList<Number>` ，因此，可以把代码简写为：

```
// 可以省略后面的Number，编译器可以自动推断泛型类型：
List<Number> list = new ArrayList<>();
```

泛型接口

除了 `ArrayList<T>` 使用了泛型，还可以在接口中使用泛型。例如， `Arrays.sort(Object[])` 可以对任意数组进行排序，但待排序的元素必须实现 `Comparable<T>` 这个泛型接口：

```
public interface Comparable<T> {
    /**
     * 返回-1: 当前实例比参数o小
     * 返回0: 当前实例与参数o相等
     * 返回1: 当前实例比参数o大
     */
    int compareTo(T o);
}
```

可以直接对 `String` 数组进行排序：

```
// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String[] ss = new String[] { "Orange", "Apple", "Pear" };
        Arrays.sort(ss);
        System.out.println(Arrays.toString(ss));
    }
}
```

▶ Run

这是因为 `String` 本身已经实现了 `Comparable<String>` 接口。如果换成我们自定义的 `Person` 类型试试：

```
// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Person[] ps = new Person[] {
            new Person("Bob", 61),
            new Person("Alice", 88),
            new Person("Lily", 75),
        };
        Arrays.sort(ps);
        System.out.println(Arrays.toString(ps));
    }
}

class Person {
    String name;
    int score;
    Person(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String toString() {
        return this.name + "," + this.score;
    }
}
```

▶ Run

运行程序，我们会得到 `ClassCastException` ，即无法将 `Person` 转型为 `Comparable` 。我们修改代码，让 `Person` 实现 `Comparable<T>` 接口：

```
// sort
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        Person[] ps = new Person[] {
            new Person("Bob", 61),
            new Person("Alice", 88),
            new Person("Lily", 75),
        };
        Arrays.sort(ps);
        System.out.println(Arrays.toString(ps));
    }
}
```

```
class Person implements Comparable<Person> {
    String name;
    int score;
    Person(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
    public String toString() {
        return this.name + "," + this.score;
    }
}
```

▶ Run

运行上述代码，可以正确实现按 `name` 进行排序。

也可以修改比较逻辑，例如，按 `score` 从高到低排序。请自行修改测试。

小结

使用泛型时，把泛型参数 `<T>` 替换为需要的class类型，例如： `ArrayList<String>` ， `ArrayList<Number>` 等；

可以省略编译器能自动推断出的类型，例如： `List<String> list = new ArrayList<>();` ；

不指定泛型参数类型时，编译器会给出警告，且只能将 `<T>` 视为 `Object` 类型；

可以在接口中定义泛型类型，实现此接口的类必须实现正确的泛型类型。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





编写泛型

编写泛型类比普通类要复杂。通常来说，泛型类一般用在集合类中，例如 `ArrayList<T>` ，我们很少需要编写泛型类。

如果我们确实需要编写一个泛型类，那么，应该如何编写它？

可以按照以下步骤来编写一个泛型类。

首先，按照某种类型，例如： `String` ，来编写类：

```
public class Pair {
    private String first;
    private String last;
    public Pair(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public String getFirst() {
        return first;
    }
    public String getLast() {
        return last;
    }
}
```

然后，标记所有的特定类型，这里是 `String` ：

```
public class Pair {
    private String first;
    private String last;
    public Pair(String first, String last) {
        this.first = first;
        this.last = last;
    }
    public String getFirst() {
        return first;
    }
    public String getLast() {
        return last;
    }
}
```

最后，把特定类型 `String` 替换为 `T` ，并申明 `<T>` ：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

熟练后即可直接从 `T` 开始编写。

静态方法

编写泛型类时，要特别注意，泛型类型 `<T>` 不能用于静态方法。例如：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 对静态方法使用<T>:
    public static Pair<T> create(T first, T last) {
        return new Pair<T>(first, last);
    }
}
```

上述代码会导致编译错误，我们无法在静态方法 `create()` 的方法参数和返回类型上使用泛型类型 `T` 。

有些同学在网上搜索发现，可以在 `static` 修饰符后面加一个 `<T>` ，编译就能通过：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 可以编译通过:
    public static <T> Pair<T> create(T first, T last) {
        return new Pair<T>(first, last);
    }
}
```

但实际上，这个 `<T>` 和 `Pair<T>` 类型的 `<T>` 已经没有任何关系了。

对于静态方法，我们可以单独改写为“泛型”方法，只需要使用另一个类型即可。对于上面的 `create()` 静态方法，我们应该把它改为另一种泛型类型，例如， `<K>` ：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public T getLast() { ... }

    // 静态泛型方法应该使用其他类型区分:
    public static <K> Pair<K> create(K first, K last) {
        return new Pair<K>(first, last);
    }
}
```

这样才能清楚地将静态方法的泛型类型和实例类型的泛型类型区分开。

多个泛型类型

泛型还可以定义多种类型。例如，我们希望 `Pair` 不总是存储两个类型一样的对象，就可以使用类型 `<T, K>` ：

```
public class Pair<T, K> {
    private T first;
    private K last;
    public Pair(T first, K last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() { ... }
    public K getLast() { ... }
}
```

使用的时候，需要指出两种类型：

```
Pair<String, Integer> p = new Pair<>("test", 123);
```

Java标准库的 `Map<K, V>` 就是使用两种泛型类型的例子。它对Key使用一种类型，对Value使用另一种类型。

小结

编写泛型时，需要定义泛型类型 `<T>` ；

静态方法不能引用泛型类型 `<T>` ，必须定义其他类型（例如 `<K>` ）来实现静态泛型方法；

泛型可以同时定义多种类型，例如 `Map<K, V>` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





擦拭法

泛型是一种类似”模板代码“的技术，不同语言的泛型实现方式不一定相同。

Java语言的泛型实现方式是擦拭法（Type Erasure）。

所谓擦拭法是指，虚拟机对泛型其实一无所知，所有的工作都是编译器做的。

例如，我们编写了一个泛型类 `Pair<T>` ，这是编译器看到的代码：

```
1.  public class Pair<T> {
2.      private T first;
3.      private T last;
4.      public Pair(T first, T last) {
5.          this.first = first;
6.          this.last = last;
7.      }
8.      public T getFirst() {
9.          return first;
10.     }
11.     public T getLast() {
12.         return last;
13.     }
14. }
```

而虚拟机根本不知道泛型。这是虚拟机执行的代码：

```
1.  public class Pair {
2.      private Object first;
3.      private Object last;
4.      public Pair(Object first, Object last) {
5.          this.first = first;
6.          this.last = last;
7.      }
8.      public Object getFirst() {
9.          return first;
10.     }
11.     public Object getLast() {
12.         return last;
13.     }
14. }
```

因此，Java使用擦拭法实现泛型，导致了：

- 编译器把类型 `<T>` 视为 `Object` ；
- 编译器根据 `<T>` 实现安全的强制转型。

使用泛型的时候，我们编写的代码也是编译器看到的代码：

```
Pair<String> p = new Pair<>("Hello", "world");
String first = p.getFirst();
String last = p.getLast();
```

而虚拟机执行的代码并没有泛型：

```
Pair p = new Pair("Hello", "world");
String first = (String) p.getFirst();
String last = (String) p.getLast();
```

所以，Java的泛型是由编译器在编译时实行的，编译器内部永远把所有类型 `T` 视为 `Object` 处理，但是，在需要转型的时候，编译器会根据 `T` 的类型自动为我们实行安全地强制转型。

了解了Java泛型的实现方式——擦拭法，我们就知道了Java泛型的局限：

局限一： `<T>` 不能是基本类型，例如 `int` ，因为实际类型是 `Object` ， `Object` 类型无法持有基本类型：

```
Pair<int> p = new Pair<>(1, 2); // compile error!
```

局限二：无法取得带泛型的 `Class` 。观察以下代码：

```
public class Main {
    public static void main(String[] args) {
        Pair<String> p1 = new Pair<>("Hello", "world");
        Pair<Integer> p2 = new Pair<>(123, 456);
        Class c1 = p1.getClass();
        Class c2 = p2.getClass();
        System.out.println(c1==c2); // true
        System.out.println(c1==Pair.class); // true
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

▶ Run

因为 `T` 是 `Object` ，我们对 `Pair<String>` 和 `Pair<Integer>` 类型获取 `Class` 时，获取到的是同一个 `Class` ，也就是 `Pair` 类的 `Class` 。

换句话说，所有泛型实例，无论 `T` 的类型是什么， `getClass()` 返回同一个 `Class` 实例，因为编译后它们全部都是 `Pair<Object>` 。

局限三：无法判断带泛型的 `Class` ：

```
Pair<Integer> p = new Pair<>(123, 456);
// Compile error:
if (p instanceof Pair<String>.class) {
}
```

原因和前面一样，并不存在 `Pair<String>.class` ，而是只有唯一的 `Pair.class` 。

局限四：不能实例化 `T` 类型：

```
public class Pair<T> {
    private T first;
    private T last;
```

```
    public Pair() {
        // Compile error:
        first = new T();
        last = new T();
    }
}
```

上述代码无法通过编译，因为构造方法的两行语句：

```
first = new T();
last = new T();
```

擦拭后实际上变成了：

```
first = new Object();
last = new Object();
```

这样一来，创建 `new Pair<String>()` 和创建 `new Pair<Integer>()` 就全部成了 `Object` ，显然编译器要阻止这种类型不对的代码。

要实例化 `T` 类型，我们必须借助额外的 `Class<T>` 参数：

```
public class Pair<T> {
    private T first;
    private T last;
    public Pair(Class<T> clazz) {
        first = clazz.newInstance();
        last = clazz.newInstance();
    }
}
```

上述代码借助 `Class<T>` 参数并通过反射来实例化 `T` 类型，使用的时候，也必须传入 `Class<T>` 。例如：

```
Pair<String> pair = new Pair<>(String.class);
```

因为传入了 `Class<String>` 的实例，所以我们借助 `String.class` 就可以实例化 `String` 类型。

不恰当的覆写方法

有些时候，一个看似正确定义的方法会无法通过编译。例如：

```
1.  public class Pair<T> {
2.      public boolean equals(T t) {
3.          return this == t;
4.      }
5.  }
```

这是因为，定义的 `equals(T t)` 方法实际上会被擦拭成 `equals(Object t)` ，而这个方法是继承自 `Object` 的，编译器会阻止一个实际上会变成覆写的泛型方法定义。

换个方法名，避开与 `Object.equals(Object)` 的冲突就可以成功编译：

```
1.  public class Pair<T> {
2.      public boolean same(T t) {
3.          return this == t;
4.      }
5.  }
```

泛型继承

一个类可以继承自一个泛型类。例如：父类的类型是 `Pair<Integer>` ，子类的类型是 `IntPair` ，可以这么继承：

```
1.  public class IntPair extends Pair<Integer> {
2.  }
```

使用的时候，因为子类 `IntPair` 并没有泛型类型，所以，正常使用即可：

```
1.  IntPair ip = new IntPair(1, 2);
```

前面讲了，我们无法获取 `Pair<T>` 的 `T` 类型，即给定一个变量 `Pair<Integer> p` ，无法从 `p` 中获取到 `Integer` 类型。

但是，在父类是泛型类型的情况下，编译器就必须把类型 `T` （对 `IntPair` 来说，也就是 `Integer` 类型）保存到子类的class文件中，不然编译器就不知道 `IntPair` 只能存取 `Integer` 这种类型。

在继承了泛型类型的情况下，子类可以获取父类的泛型类型。例如： `IntPair` 可以获取到父类的泛型类型 `Integer` 。获取父类的泛型类型代码比较复杂：

```
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;

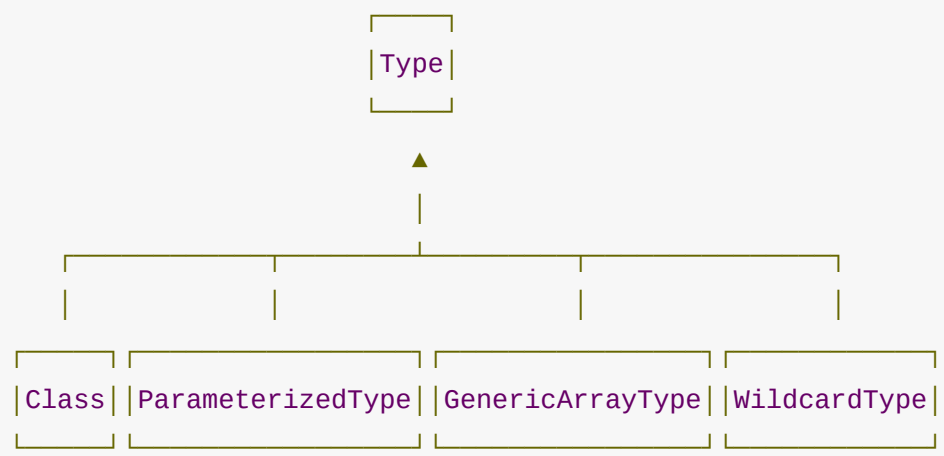
public class Main {
    public static void main(String[] args) {
        Class<IntPair> clazz = IntPair.class;
        Type t = clazz.getGenericSuperclass();
        if (t instanceof ParameterizedType) {
            ParameterizedType pt = (ParameterizedType) t;
            Type[] types = pt.getActualTypeArguments(); // 可能有多个泛型类型
            Type firstType = types[0]; // 取第一个泛型类型
            Class<?> typeClass = (Class<?>) firstType;
            System.out.println(typeClass); // Integer
        }
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}

class IntPair extends Pair<Integer> {
    public IntPair(Integer first, Integer last) {
        super(first, last);
    }
}
```

▶ Run

因为Java引入了泛型，所以，只用 `Class` 来标识类型已经不够了。实际上，Java的类型系统结构如下：



小结

Java的泛型是采用擦拭法实现的；

擦拭法决定了泛型 `<T>` ：

- 不能是基本类型，例如： `int` ；
- 不能获取带泛型类型的 `Class` ，例如： `Pair<String>.class` ；
- 不能判断带泛型类型的类型，例如： `x instanceof Pair<String>` ；
- 不能实例化 `T` 类型，例如： `new T()` 。

泛型方法要防止重复定义方法，例如： `public boolean equals(T obj)` ；

子类可以获取父类的泛型类型 `<T>` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





extends通配符

我们前面已经讲到了泛型的继承关系：`Pair<Integer>` 不是 `Pair<Number>` 的子类。

假设我们定义了 `Pair<T>`：

```
1. public class Pair<T> { ... }
```

然后，我们又针对 `Pair<Number>` 类型写了一个静态方法，它接收的参数类型是 `Pair<Number>`：

```
1. public class PairHelper {
2.     static int add(Pair<Number> p) {
3.         Number first = p.getFirst();
4.         Number last = p.getLast();
5.         return first.intValue() + last.intValue();
6.     }
7. }
```

上述代码是可以正常编译的。使用的时候，我们传入：

```
1. int sum = PairHelper.add(new Pair<Number>(1, 2));
```

注意：传入的类型是 `Pair<Number>`，实际参数类型是 `(Integer, Integer)`。

既然实际参数是 `Integer` 类型，试试传入 `Pair<Integer>`：

```
public class Main {
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

▶ Run

直接运行，会得到一个编译错误：

```
1.  incompatible types: Pair<Integer> cannot be converted to Pair<Number>
```

原因很明显，因为 `Pair<Integer>` 不是 `Pair<Number>` 的子类，因此， `add(Pair<Number>)` 不接受参数类型 `Pair<Integer>` 。

但是从 `add()` 方法的代码可知，传入 `Pair<Integer>` 是完全符合内部代码的类型规范，因为语句：

```
1.  Number first = p.getFirst();
2.  Number last = p.getLast();
```

实际类型是 `Integer` ，引用类型是 `Number` ，没有问题。问题在于方法参数类型定死了只能传入 `Pair<Number>` 。

有没有办法使得方法参数接受 `Pair<Integer>` ？办法是有的，这就是使用 `Pair<? extends Number>` 使得方法接收所有泛型类型为 `Number` 或 `Number` 子类的 `Pair` 类型。我们把代码改写

如下：

```
public class Main {
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<? extends Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
}

class Pair<T> {
    private T first;
    private T last;
    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }
    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
}
```

▶ Run

这样一来，给方法传入 `Pair<Integer>` 类型时，它符合参数 `Pair<? extends Number>` 类型。这种使用 `<? extends Number>` 的泛型定义称之为上界通配符（Upper Bounds Wildcards），即把泛型类型 `T` 的上界限定在 `Number` 了。

除了可以传入 `Pair<Integer>` 类型，我们还可以传入 `Pair<Double>` 类型， `Pair<BigDecimal>` 类型等等，因为 `Double` 和 `BigDecimal` 都是 `Number` 的子类。

如果我们考察对 `Pair<? extends Number>` 类型调用 `getFirst()` 方法，实际的方法签名变成了：

```
1.  <? extends Number> getFirst();
```

即返回值是 `Number` 或 `Number` 的子类，因此，可以安全赋值给 `Number` 类型的变量：

```
1.  Number x = p.getFirst();
```

然后，我们不可预测实际类型就是 `Integer` ，例如，下面的代码是无法通过编译的：

```
1.  Integer x = p.getFirst();
```

这是因为实际的返回类型可能是 `Integer` ，也可能是 `Double` 或者其他类型，编译器只能确定类型一定是 `Number` 的子类（包括 `Number` 类型本身），但具体类型无法确定。

我们再来考察一下 `Pair<T>` 的 `set` 方法：

```
public class Main {
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        int n = add(p);
        System.out.println(n);
    }

    static int add(Pair<? extends Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        p.setFirst(new Integer(first.intValue() + 100));
        p.setLast(new Integer(last.intValue() + 100));
        return p.getFirst().intValue() + p.getFirst().intValue();
    }
}

class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
```

▶ Run

不出意外，我们会得到一个编译错误：

```
1.  incompatible types: Integer cannot be converted to CAP#1
2.  where CAP#1 is a fresh type-variable:
3.      CAP#1 extends Number from capture of ? extends Number
```

编译错误发生在 `p.setFirst()` 传入的参数是 `Integer` 类型。有些童鞋会问了，既然 `p` 的定义是 `Pair<? extends Number>` ，那么 `setFirst(? extends Number)` 为什么不能传入 `Integer` ？

原因还在于擦拭法。如果我们传入的 `p` 是 `Pair<Double>` ，显然它满足参数定义 `Pair<? extends Number>` ，然而， `Pair<Double>` 的 `setFirst()` 显然无法接受 `Integer` 类型。

这就是 `<? extends Number>` 通配符的一个重要限制：方法参数签名 `setFirst(? extends Number)` 无法传递任何 `Number` 类型给 `setFirst(? extends Number)` 。

这里唯一的例外是可以给方法参数传入 `null` ：

```
p.setFirst(null); // ok, 但是后面会抛出NullPointerException
p.getFirst().intValue(); // NullPointerException
```

extends通配符的作用

如果我们考察Java标准库的 `java.util.List<T>` 接口，它实现的是一个类似“可变数组”的列表，主要功能包括：

```
public interface List<T> {
    int size(); // 获取个数
    T get(int index); // 根据索引获取指定元素
    void add(T t); // 添加一个新元素
    void remove(T t); // 删除一个已有元素
}
```

现在，让我们定义一个方法来处理列表的每个元素：

```
int sumOfList(List<? extends Integer> list) {
    int sum = 0;
    for (int i=0; i<list.size(); i++) {
        Integer n = list.get(i);
        sum = sum + n;
    }
    return sum;
}
```

为什么我们定义的方法参数类型是 `List<? extends Integer>` 而不是 `List<Integer>` ？从方法内部代码看，传入 `List<? extends Integer>` 或者 `List<Integer>` 是完全一样的，但是，注意到 `List<? extends Integer>` 的限制：

- 允许调用 `get()` 方法获取 `Integer` 的引用；
- 不允许调用 `set(? extends Integer)` 方法并传入任何 `Integer` 的引用（ `null` 除外）。

因此，方法参数类型 `List<? extends Integer>` 表明了该方法内部只会读取 `List` 的元素，不会修改 `List` 的元素（因为无法调用 `add(? extends Integer)` 、 `remove(? extends Integer)` 这些方法。换句话说，这是一个对参数 `List<? extends Integer>` 进行只读的方法（恶意调用 `set(null)` 除外）。

使用extends限定T类型

在定义泛型类型 `Pair<T>` 的时候，也可以使用 `extends` 通配符来限定 `T` 的类型：

```
1. public class Pair<T extends Number> { ... }
```

现在，我们只能定义：

```
1. Pair<Number> p1 = null;
2. Pair<Integer> p2 = new Pair<>(1, 2);
3. Pair<Double> p3 = null;
```

因为 `Number` 、 `Integer` 和 `Double` 都符合 `<T extends Number>` 。

非 `Number` 类型将无法通过编译：

```
1. Pair<String> p1 = null; // compile error!
2. Pair<Object> p2 = null; // compile error!
```

因为 `String` 、 `Object` 都不符合 `<T extends Number>` ，因为它们不是 `Number` 类型或 `Number` 的子类。

小结

使用类似 `<? extends Number>` 通配符作为方法参数时表示：

- 方法内部可以调用获取 `Number` 引用的方法，例如： `Number n = obj.getFirst();` ；
- 方法内部无法调用传入 `Number` 引用的方法（ `null` 除外），例如： `obj.setFirst(Number n);` 。

即一句话总结：使用 `extends` 通配符表示可以读，不能写。

使用类似 `<T extends Number>` 定义泛型类时表示：

- 泛型类型限定为 `Number` 以及 `Number` 的子类。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





super通配符

我们前面已经讲到了泛型的继承关系：`Pair<Integer>` 不是 `Pair<Number>` 的子类。

考察下面的 `set` 方法：

```
1. void set(Pair<Integer> p, Integer first, Integer last) {
2.     p.setFirst(first);
3.     p.setLast(last);
4. }
```

传入 `Pair<Integer>` 是允许的，但是传入 `Pair<Number>` 是不允许的。

和 `extends` 通配符相反，这次，我们希望接受 `Pair<Integer>` 类型，以及 `Pair<Number>` 、 `Pair<Object>` ，因为 `Number` 和 `Object` 是 `Integer` 的父类， `setFirst(Number)` 和 `setFirst(Object)` 实际上允许接受 `Integer` 类型。

我们使用 `super` 通配符来改写这个方法：

```
1. void set(Pair<? super Integer> p, Integer first, Integer last) {
2.     p.setFirst(first);
3.     p.setLast(last);
4. }
```

注意到 `Pair<? super Integer>` 表示，方法参数接受所有泛型类型为 `Integer` 或 `Integer` 父类的 `Pair` 类型。

下面的代码可以被正常编译：

```
public class Main {
    public static void main(String[] args) {
        Pair<Number> p1 = new Pair<>(12.3, 4.56);
        Pair<Integer> p2 = new Pair<>(123, 456);
        setSame(p1, 100);
        setSame(p2, 200);
        System.out.println(p1.getFirst() + ", " + p1.getLast());
        System.out.println(p2.getFirst() + ", " + p2.getLast());
    }

    static void setSame(Pair<? super Integer> p, Integer n) {
        p.setFirst(n);
        p.setLast(n);
    }
}

class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
```

▶ Run

考察 `Pair<? super Integer>` 的 `setFirst()` 方法，它的方法签名实际上是：

```
1.  void setFirst(? super Integer);
```

因此，可以安全地传入 `Integer` 类型。

再考察 `Pair<? super Integer>` 的 `getFirst()` 方法，它的方法签名实际上是：

```
1.  ? super Integer getFirst();
```

这里注意到我们无法使用 `Integer` 类型来接收 `getFirst()` 的返回值，即下面的语句将无法通过编译：

```
Integer x = p.getFirst();
```

因为如果传入的实际类型是 `Pair<Number>` ，编译器无法将 `Number` 类型转型为 `Integer` 。

注意：虽然 `Number` 是一个抽象类，我们无法直接实例化它。但是，即便 `Number` 不是抽象类，这里仍然无法通过编译。此外，传入 `Pair<Object>` 类型时，编译器也无法将 `Object` 类型转型为 `Integer` 。

唯一可以接收 `getFirst()` 方法返回值的是 `Object` 类型：

```
Object obj = p.getFirst();
```

因此，使用 `<? super Integer>` 通配符表示：

- 允许调用 `set(? super Integer)` 方法传入 `Integer` 的引用；
- 不允许调用 `get()` 方法获得 `Integer` 的引用。

唯一例外是可以获取 `Object` 的引用： `Object o = p.getFirst()` 。

换句话说，使用 `<? super Integer>` 通配符作为方法参数，表示方法内部代码对于参数只能写，不能读。

对比extends和super通配符

我们再回顾一下 `extends` 通配符。作为方法参数， `<? extends T>` 类型和 `<? super T>` 类型的区别在于：

- `<? extends T>` 允许调用读方法 `T get()` 获取 `T` 的引用，但不允许调用写方法 `set(T)` 传入 `T` 的引用（传入 `null` 除外）；
- `<? super T>` 允许调用写方法 `set(T)` 传入 `T` 的引用，但不允许调用读方法 `T get()` 获取 `T` 的引用（获取 `Object` 除外）。

一个是允许读不允许写，另一个是允许写不允许读。

先记住上面的结论，我们来看Java标准库的 `Collections` 类定义的 `copy()` 方法：

```
public class Collections {
    // 把src的每个元素复制到dest中:
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        for (int i=0; i<src.size(); i++) {
            T t = src.get(i);
            dest.add(t);
```

```
        }
    }
}
```

它的作用是把一个 `List` 的每个元素依次添加到另一个 `List` 中。它的第一个参数是 `List<? super T>` ，表示目标 `List` ，第二个参数 `List<? extends T>` ，表示要复制的 `List` 。我们可以简单地用 `for` 循环实现复制。在 `for` 循环中，我们可以看到，对于类型 `<? extends T>` 的变量 `src` ，我们可以安全地获取类型 `T` 的引用，而对于类型 `<? super T>` 的变量 `dest` ，我们可以安全地传入 `T` 的引用。

这个 `copy()` 方法的定义就完美地展示了 `extends` 和 `super` 的意图：

- `copy()` 方法内部不会读取 `dest` ，因为不能调用 `dest.get()` 来获取 `T` 的引用；
- `copy()` 方法内部也不会修改 `src` ，因为不能调用 `src.add(T)` 。

这是由编译器检查来实现的。如果在方法代码中意外修改了 `src` ，或者意外读取了 `dest` ，就会导致一个编译错误：

```
public class Collections {
    // 把src的每个元素复制到dest中:
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        ...
        T t = dest.get(0); // compile error!
        src.add(t); // compile error!
    }
}
```

这个 `copy()` 方法的另一个好处是可以安全地把一个 `List<Integer>` 添加到 `List<Number>` ，但是无法反过来添加：

```
// copy List<Integer> to List<Number> ok:
List<Number> numList = ...;
List<Integer> intList = ...;
Collections.copy(numList, intList);

// ERROR: cannot copy List<Number> to List<Integer>:
Collections.copy(intList, numList);
```

而这些都是通过 `super` 和 `extends` 通配符，并由编译器强制检查来实现的。

PECS原则

何时使用 `extends` ，何时使用 `super` ？为了便于记忆，我们可以用PECS原则：Producer Extends Consumer Super。

即：如果需要返回 `T` ，它是生产者（Producer），要使用 `extends` 通配符；如果需要写入 `T` ，它是消费者（Consumer），要使用 `super` 通配符。

还是以 `Collections` 的 `copy()` 方法为例：

```
public class Collections {
    public static <T> void copy(List<? super T> dest, List<? extends T> src) {
        for (int i=0; i<src.size(); i++) {
            T t = src.get(i); // src是producer
            dest.add(t); // dest是consumer
        }
    }
}
```

需要返回 `T` 的 `src` 是生产者，因此声明为 `List<? extends T>` ，需要写入 `T` 的 `dest` 是消费者，因此声明为 `List<? super T>` 。

无限定通配符

我们已经讨论了 `<? extends T>` 和 `<? super T>` 作为方法参数的作用。实际上，Java的泛型还允许使用无限定通配符（Unbounded Wildcard Type），即只定义一个 `?` ：

```
void sample(Pair<?> p) {
}
```

因为 `<?>` 通配符既没有 `extends` ，也没有 `super` ，因此：

- 不允许调用 `set(T)` 方法并传入引用（ `null` 除外）；
- 不允许调用 `T get()` 方法并获取 `T` 引用（只能获取 `Object` 引用）。

换句话说，既不能读，也不能写，那只能做一些 `null` 判断：

```
static boolean isNull(Pair<?> p) {
    return p.getFirst() == null || p.getLast() == null;
}
```

大多数情况下，可以引入泛型参数 `<T>` 消除 `<?>` 通配符：

```
static <T> boolean isNull(Pair<T> p) {
```

```
2.      return p.getFirst() == null || p.getLast() == null;
3.  }
```

`<?>` 通配符有一个独特的特点，就是： `Pair<?>` 是所有 `Pair<T>` 的超类：

```
public class Main {
    public static void main(String[] args) {
        Pair<Integer> p = new Pair<>(123, 456);
        Pair<?> p2 = p; // 安全地向上转型
        System.out.println(p2.getFirst() + ", " + p2.getLast());
    }
}

class Pair<T> {
    private T first;
    private T last;

    public Pair(T first, T last) {
        this.first = first;
        this.last = last;
    }

    public T getFirst() {
        return first;
    }
    public T getLast() {
        return last;
    }
    public void setFirst(T first) {
        this.first = first;
    }
    public void setLast(T last) {
        this.last = last;
    }
}
```

▶ Run

上述代码是可以正常编译运行的，因为 `Pair<Integer>` 是 `Pair<?>` 的子类，可以安全地向上转型。

小结

使用类似 `<? super Integer>` 通配符作为方法参数时表示：

- 方法内部可以调用传入 `Integer` 引用的方法，例如： `obj.setFirst(Integer n);` ；
- 方法内部无法调用获取 `Integer` 引用的方法（ `Object` 除外），例如： `Integer n = obj.getFirst();` 。

即使用 `super` 通配符表示只能写不能读。

使用 `extends` 和 `super` 通配符要遵循PECS原则。

无限定通配符 `<?>` 很少使用，可以用 `<T>` 替换，同时它是所有 `<T>` 类型的超类。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





泛型和反射

Java的部分反射API也是泛型。例如：`Class<T>` 就是泛型：

```
// compile warning:
Class clazz = String.class;
String str = (String) clazz.newInstance();

// no warning:
Class<String> clazz = String.class;
String str = clazz.newInstance();
```

调用 `Class` 的 `getSuperclass()` 方法返回的 `Class` 类型是 `Class<? super T>`：

```
Class<? super String> sup = String.class.getSuperclass();
```

构造方法 `Constructor<T>` 也是泛型：

```
Class<Integer> clazz = Integer.class;
Constructor<Integer> cons = clazz.getConstructor(int.class);
Integer i = cons.newInstance(123);
```

我们可以声明带泛型的数组，但不能用 `new` 操作符创建带泛型的数组：

```
Pair<String>[] ps = null; // ok
Pair<String>[] ps = new Pair<String>[2]; // compile error!
```

必须通过强制转型实现带泛型的数组：

```
@SuppressWarnings("unchecked")
Pair<String>[] ps = (Pair<String>[]) new Pair[2];
```

使用泛型数组要特别小心，因为数组实际上在运行期没有泛型，编译器可以强制检查变量 `ps`，因为它的类型是泛型数组。但是，编译器不会检查变量 `arr`，因为它不是泛型数组。因为这两个变量实际上指向同一个数组，所以，操作 `arr` 可能导致从 `ps` 获取元素时报错，例如，以下代码演示了不安全地使用带泛型的数组：

```
Pair[] arr = new Pair[2];
Pair<String>[] ps = (Pair<String>[]) arr;

```

```
ps[0] = new Pair<String>("a", "b");
arr[1] = new Pair<Integer>(1, 2);

// ClassCastException:
Pair<String> p = ps[1];
String s = p.getFirst();
```

要安全地使用泛型数组，必须扔掉 `arr` 的引用：

```
@SuppressWarnings("unchecked")
Pair<String>[] ps = (Pair<String>[]) new Pair[2];
```

上面的代码中，由于拿不到原始数组的引用，就只能对泛型数组 `ps` 进行操作，这种操作就是安全的。

带泛型的数组实际上是编译器的类型擦除：

```
Pair[] arr = new Pair[2];
Pair<String>[] ps = (Pair<String>[]) arr;

System.out.println(ps.getClass() == Pair[].class); // true

String s1 = (String) arr[0].getFirst();
String s2 = ps[0].getFirst();
```

所以我们不能直接创建泛型数组 `T[]` ，因为擦拭后代码变为 `Object[]` ：

```
// compile error:
public class Abc<T> {
    T[] createArray() {
        return new T[5];
    }
}
```

必须借助 `Class<T>` 来创建泛型数组：

```
T[] createArray(Class<T> cls) {
    return (T[]) Array.newInstance(cls, 5);
}
```

我们还可以利用可变参数创建泛型数组 `T[]` ：

```
1.  public class ArrayHelper {
2.      @SafeVarargs
3.      static <T> T[] asArray(T... objs) {
4.          return objs;
5.      }
6.  }
7.
8.  String[] ss = ArrayHelper.asArray("a", "b", "c");
9.  Integer[] ns = ArrayHelper.asArray(1, 2, 3);
```

谨慎使用泛型可变参数

在上面的例子中，我们看到，通过：

```
1.  static <T> T[] asArray(T... objs) {
2.      return objs;
3.  }
```

似乎可以安全地创建一个泛型数组。但实际上，这种方法非常危险。以下代码来自《Effective Java》的示例：

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String[] arr = asArray("one", "two", "three");
        System.out.println(Arrays.toString(arr));
        // ClassCastException:
        String[] firstTwo = pickTwo("one", "two", "three");
        System.out.println(Arrays.toString(firstTwo));
    }

    static <K> K[] pickTwo(K k1, K k2, K k3) {
        return asArray(k1, k2);
    }

    static <T> T[] asArray(T... objs) {
        return objs;
    }


}
```

▶ Run

直接调用 `asArray(T…)` 似乎没有问题，但是在另一个方法中，我们返回一个泛型数组就会产生 `ClassCastException` ，原因还是因为擦拭法，在 `pickTwo()` 方法内部，编译器无法检

测 `K[]` 的正确类型，因此返回了 `Object[]` 。

如果仔细观察，可以发现编译器对所有可变泛型参数都会发出警告，除非确认完全没有问题，才可以用 `@SafeVarargs` 消除警告。

如果在方法内部创建了泛型数组，最好不要将它返回给外部使用。

更详细的解释请参考《Effective Java》“Item 32: Combine generics and varargs judiciously”。

小结

部分反射API是泛型，例如： `Class<T>` ， `Constructor<T>` ；

可以声明带泛型的数组，但不能直接创建带泛型的数组，必须强制转型；

可以通过 `Array.newInstance(Class<T>, int)` 创建 `T[]` 数组，需要强制转型；

同时使用泛型和可变参数时需要特别小心。

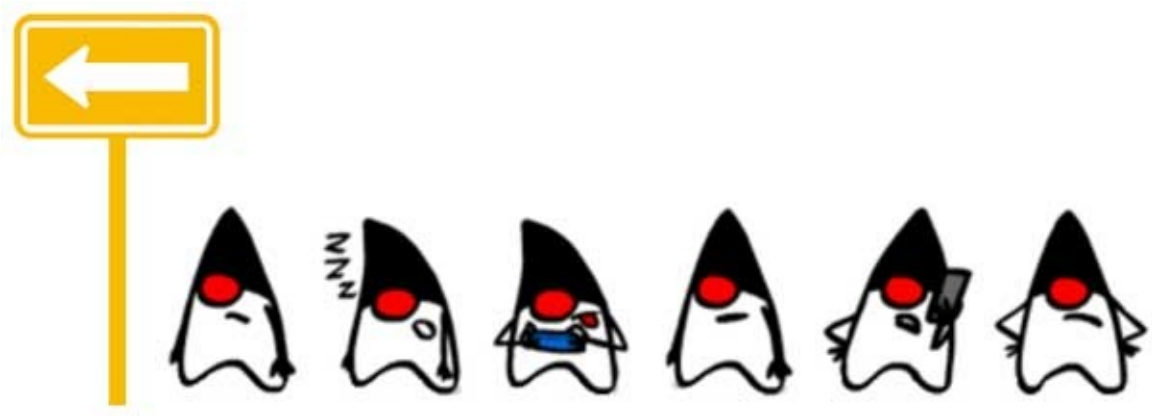
读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





集合

本节我们将介绍Java的集合类型。集合类型也是Java标准库中被使用最多的类型。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Java集合简介

什么是集合（Collection）？集合就是“由若干个确定的元素所构成的整体”。例如，5只小兔构成的集合：

```
┌ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┐

|   (\_(\     (\_/)     (\_/)     (\_/)      (\(\   |
     ( -.-)    (•.•)     (>.<)     (^.^)     (='.')
|  C(")_(")  (")_(")   (")_(")   (")_(")   O(_")")  |

└ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ─ ┘
```

在数学中，我们经常遇到集合的概念。例如：

- 有限集合：
  - 一个班所有的同学构成的集合；
  - 一个网站所有的商品构成的集合；
  - …
- 无限集合：
  - 全体自然数集合：1，2，3，……
  - 有理数集合；
  - 实数集合；
  - …

为什么要在计算机中引入集合呢？这是为了便于处理一组类似的数据，例如：

- 计算所有同学的总成绩和平均成绩；
- 列举所有的商品名称和价格；
- ……

在Java中，如果一个Java对象可以在内部持有若干其他Java对象，并对外提供访问接口，我们把这种Java对象称为集合。很显然，Java的数组可以看作是一种集合：

```
String[] ss = new String[10]; // 可以持有10个String对象
ss[0] = "Hello"; // 可以放入String对象
String first = ss[0]; // 可以获取String对象
```

既然Java提供了数组这种数据类型，可以充当集合，那么，我们为什么还需要其他集合类？这是因为数

组有如下限制：

- 数组初始化后大小不可变；
- 数组只能按索引顺序存取。

因此，我们需要各种不同类型的集合类来处理不同的数据，例如：

- 可变大小的顺序链表；
- 保证无重复元素的集合；
- …

Collection

Java标准库自带的 `java.util` 包提供了集合类： `Collection` ，它是除 `Map` 外所有其他集合类的根接口。Java的 `java.util` 包主要提供了以下三种类型的集合：

- `List` ：一种有序列表的集合，例如，按索引排列的 `Student` 的 `List` ；
- `Set` ：一种保证没有重复元素的集合，例如，所有无重复名称的 `Student` 的 `Set` ；
- `Map` ：一种通过键值（key-value）查找的映射表集合，例如，根据 `Student` 的 `name` 查找对应 `Student` 的 `Map` 。

Java集合的设计有几个特点：一是实现了接口和实现类相分离，例如，有序表的接口是 `List` ，具体的实现类有 `ArrayList` ， `LinkedList` 等，二是支持泛型，我们可以限制在一个集合中只能放入同一种数据类型的元素，例如：

```
1.  List<String> list = new ArrayList<>(); // 只能放入String类型
```

最后，Java访问集合总是通过统一的方式——迭代器（Iterator）来实现，它最明显的好处在于无需知道集合内部元素是按什么方式存储的。

由于Java的集合设计非常久远，中间经历过大规模改进，我们要注意到有一小部分集合类是遗留类，不应该继续使用：

- `Hashtable` ：一种线程安全的 `Map` 实现；
- `Vector` ：一种线程安全的 `List` 实现；
- `Stack` ：基于 `Vector` 实现的 `LIFO` 的栈。

还有一小部分接口是遗留接口，也不应该继续使用：

- `Enumeration<E>` ：已被 `Iterator<E>` 取代。

小结

Java的集合类定义在 `java.util` 包中，支持泛型，主要提供了3种集合类，包

括 `List` ， `Set` 和 `Map` 。Java集合使用统一的 `Iterator` 遍历，尽量不要使用遗留接口。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论



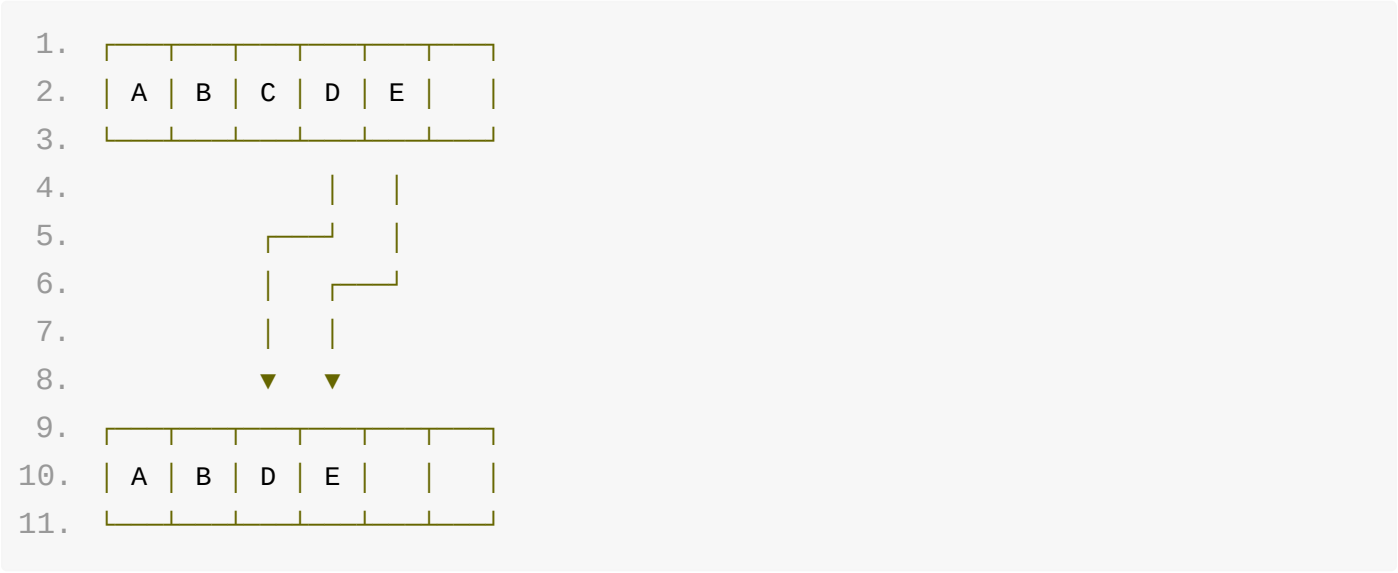


使用List

在集合类中，`List`是最基础的一种集合：它是一种有序链表。

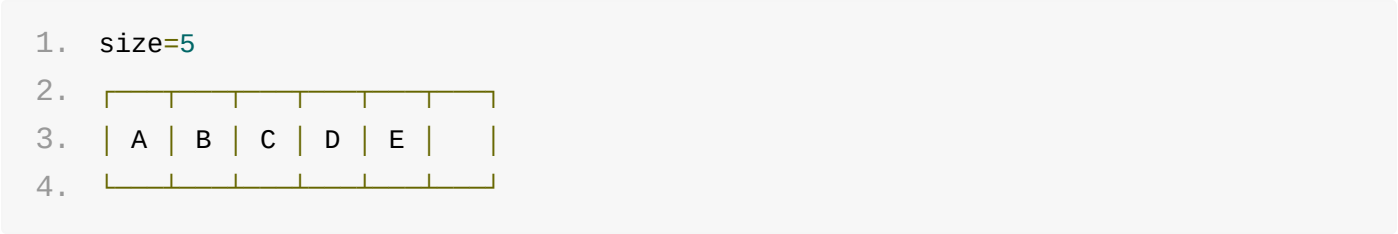
`List`的行为和数组几乎完全相同：`List`内部按照放入元素的先后顺序存放，每个元素都可以通过索引确定自己的位置，`List`的索引和数组一样，从`0`开始。

数组和`List`类似，也是有序结构，如果我们使用数组，在添加和删除元素的时候，会非常不方便。例如，从一个已有的数组`{'A', 'B', 'C', 'D', 'E'}`中删除索引为`2`的元素：

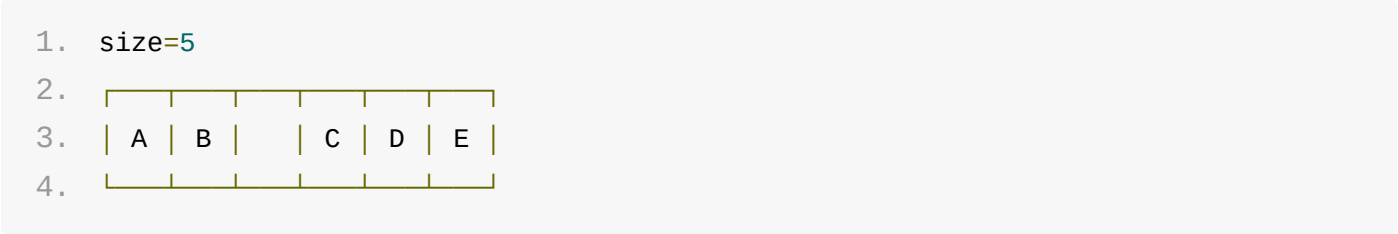


这个“删除”操作实际上是把`'C'`后面的元素依次往前挪一个位置，而“添加”操作实际上是把指定位置以后的元素都依次向后挪一个位置，腾出来的位置给新加的元素。这两种操作，用数组实现非常麻烦。

因此，在实际应用中，需要增删元素的有序列表，我们使用最多的是`ArrayList`。实际上，`ArrayList`在内部使用了数组来存储所有元素。例如，一个ArrayList拥有5个元素，实际数组大小为`6`（即有一个空位）：



当添加一个元素并指定索引到`ArrayList`时，`ArrayList`自动移动需要移动的元素：



然后，往内部指定索引的数组位置添加一个元素，然后把 `size` 加 `1`：

```
size=6
┌───┬───┬───┬───┬───┬───┐
│ A │ B │ F │ C │ D │ E │
└───┴───┴───┴───┴───┴───┘
```

继续添加元素，但是数组已满，没有空闲位置的时候，`ArrayList` 先创建一个更大的新数组，然后把旧数组的所有元素复制到新数组，紧接着用新数组取代旧数组：

```
size=6
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
│ A │ B │ F │ C │ D │ E │   │   │   │   │   │   │
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
```

现在，新数组就有了空位，可以继续添加一个元素到数组末尾，同时 `size` 加 `1`：

```
size=7
┌───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┬───┐
│ A │ B │ F │ C │ D │ E │ G │   │   │   │   │   │
└───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┴───┘
```

可见，`ArrayList` 把添加和删除的操作封装起来，让我们操作 `List` 类似于操作数组，却不用关心内部元素如何移动。

我们考察 `List<E>` 接口，可以看到几个主要的接口方法：

- 在末尾添加一个元素：`void add(E e)`
- 在指定索引添加一个元素：`void add(int index, E e)`
- 删除指定索引的元素：`int remove(int index)`
- 删除某个元素：`int remove(Object e)`
- 获取指定索引的元素：`E get(int index)`
- 获取链表大小（包含元素的个数）：`int size()`

但是，实现 `List` 接口并非只能通过数组（即 `ArrayList` 的实现方式）来实现，另一种 `LinkedList` 通过“链表”也实现了List接口。在 `LinkedList` 中，它的内部每个元素都指向下一个元素：

```
        ┌───┬───┐   ┌───┬───┐   ┌───┬───┐   ┌───┬───┐
HEAD ──>│ A │ ●─┼──>│ B │ ●─┼──>│ C │ ●─┼──>│ D │   │
        └───┴───┘   └───┴───┘   └───┴───┘   └───┴───┘
```

我们来比较一下 `ArrayList` 和 `LinkedList`：

	ArrayList	LinkedList
获取指定元素	速度很快	需要从头开始查找元素
添加元素到末尾	速度很快	速度很快
在指定位置添加/删除	需要移动元素	不需要移动元素
内存占用	少	较大

通常情况下，我们总是优先使用 `ArrayList`。

List的特点

使用 `List` 时，我们要关注 `List` 接口的规范。`List` 接口允许我们添加重复的元素，即 `List` 内部的元素可以重复：

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple"); // size=1
        list.add("pear"); // size=2
        list.add("apple"); // 允许重复添加元素，size=3
        System.out.println(list.size());
    }
}
```

▶ Run

`List` 还允许添加 `null`：

```
import java.util.ArrayList;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple"); // size=1
        list.add(null); // size=2
        list.add("pear"); // size=3
        String second = list.get(1); // null
        System.out.println(second);
    }
}
```

▶ Run

创建List

除了使用 `ArrayList` 和 `LinkedList` ，我们还可以通过 `List` 接口提供的 `of()` 方法，根据给定元素快速创建 `List` ：

```
1.  List<Integer> list = List.of(1, 2, 5);
```

但是 `List.of()` 方法不接受 `null` 值，如果传入 `null` ，会抛出 `NullPointerException` 异常。

遍历List

和数组类型，我们要遍历一个List，完全可以用for循环根据索引配合 `get(int)` 方法遍历：

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        for (int i=0; i<list.size(); i++) {
            String s = list.get(i);
            System.out.println(s);
        }
    }
}
```

▶ Run

但这种方式并不推荐，一是代码复杂，二是因为 `get(int)` 方法只有 `ArrayList` 的实现是高效的，换成 `LinkedList` 后，索引越大，访问速度越慢。

所以我们要始终坚持使用迭代器 `Iterator` 来访问 `List` 。 `Iterator` 本身也是一个对象，但它是由 `List` 的实例调用 `iterator()` 方法的时候创建的。 `Iterator` 对象知道如何遍历一个 `List` ，并且不同的 `List` 类型，返回的 `Iterator` 对象实现也是不同的，但总是具有最高的访问效率。

`Iterator` 对象有两个方法： `boolean hasNext()` 判断是否有下一个元素， `E next()` 返回下一个元素。因此，使用 `Iterator` 遍历 `List` 代码如下：

```
import java.util.Iterator;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
            String s = it.next();
            System.out.println(s);
        }
    }
}
```

▶ Run

有童鞋可能觉得使用 `Iterator` 访问 `List` 的代码比使用索引更复杂。但是，要记住，通过 `Iterator` 遍历 `List` 永远是最高效的方式。并且，由于 `Iterator` 遍历是如此常用，所以，Java的 `for each` 循环本身就可以帮我们使用 `Iterator` 遍历。把上面的代码再改写如下：

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        for (String s : list) {
            System.out.println(s);
        }
    }
}
```

▶ Run

上述代码就是我们编写遍历 `List` 的常见代码。

实际上，只要实现了 `Iterable` 接口的集合类都可以直接用 `for each` 循环来遍历，Java编译器本身并不知道如何遍历集合对象，但它会自动把 `for each` 循环变成 `Iterator` 的调用，原因就在于 `Iterable` 接口定义了一个 `Iterator<E> iterator()` 方法，强迫集合类必须返回一个 `Iterator` 实例。

List和Array转换

把 `List` 变为 `Array` 有三种方法，第一种是调用 `toArray()` 方法直接返回一个 `Object[]` 数组：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("apple", "pear", "banana");
        Object[] array = list.toArray();
        for (Object s : array) {
            System.out.println(s);
        }
    }
}
```

▶ Run

这种方法会丢失类型信息，所以实际应用很少。

第二种方式是给 `toArray(T[])` 传入一个类型相同的 `Array` ， `List` 内部自动把元素复制到传入的 `Array` 中：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        Integer[] array = list.toArray(new Integer[3]);
        for (Integer n : array) {
            System.out.println(n);
        }
    }
}
```

▶ Run

注意到这个 `toArray(T[])` 方法的泛型参数 `<T>` 并不是 `List` 接口定义的泛型参数 `<E>` ，所以，我们实际上可以传入其他类型的数组，例如我们传入 `Number` 类型的数组，返回的仍然是 `Number` 类型：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        Number[] array = list.toArray(new Number[3]);
        for (Number n : array) {
            System.out.println(n);
        }
    }
}
```

▶ Run

但是，如果我们传入类型不匹配的数组，例如， `String[]` 类型的数组，由于 `List` 的元素是 `Integer` ，所以无法放入 `String` 数组，这个方法会抛出 `ArrayStoreException` 。

如果我们传入的数组大小和 `List` 实际的元素个数不一致怎么办？根据List接口的文档，我们可以知道：

如果传入的数组不够大，那么 `List` 内部会创建一个新的刚好够大的数组，填充后返回；如果传入的数组比 `List` 元素还要多，那么填充完元素后，剩下的数组元素一律填充 `null` 。

实际上，最常用的是传入一个“恰好”大小的数组：

```
Integer[] array = list.toArray(new Integer[list.size()]);
```

最后一种更简洁的写法是通过 `List` 接口定义的 `T[] toArray(IntFunction<T[]> generator)` 方法：

```
Integer[] array = list.toArray(Integer[]::new);
```

这种函数式写法我们会在后续讲到。

反过来，把 `Array` 变为 `List` 就简单多了，通过 `List.of(T…)` 方法最简单：

```
Integer[] array = { 1, 2, 3 };
List<Integer> list = List.of(array);
```

对于JDK 11之前的版本，可以使用 `Arrays.asList(T…)` 方法把数组转换成 `List` 。

要注意的是，返回的 `List` 不一定就是 `ArrayList` 或者 `LinkedList` ，因为 `List` 只是一个接口，如果我们调用 `List.of()` ，它返回的是一个只读 `List` ：

```
import java.util.List;
```

```
public class Main {
    public static void main(String[] args) {
        List<Integer> list = List.of(12, 34, 56);
        list.add(999); // UnsupportedOperationException
    }
}
```

▶ Run

对只读 `List` 调用 `add()` 、 `remove()` 方法会抛出 `UnsupportedOperationException` 。

练习

给定一组连续的整数，例如：10，11，12，……，20，但其中缺失一个数字，试找出缺失的数字：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 构造从start到end的序列：
        final int start = 10;
        final int end = 20;
        List<Integer> list = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            list.add(i);
        }
        // 随机删除List中的一个元素:
        int removed = list.remove((int) (Math.random() * list.size()));
        int found = findMissingNumber(start, end, list);
        System.out.println(list.toString());
        System.out.println("missing number: " + found);
        System.out.println(removed == found ? "测试成功" : "测试失败");
    }
```

```
    static int findMissingNumber(int start, int end, List<Integer> list) {
        return 0;
    }
```

```
}
```

▶ Run

增强版：和上述题目一样，但整数不再有序，试找出缺失的数字：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 构造从start到end的序列：
        final int start = 10;
        final int end = 20;
        List<Integer> list = new ArrayList<>();
        for (int i = start; i <= end; i++) {
            list.add(i);
        }
        // 洗牌算法suffle可以随机交换List中的元素位置：
        Collections.shuffle(list);
        // 随机删除List中的一个元素：
        int removed = list.remove((int) (Math.random() * list.size()));
        int found = findMissingNumber(start, end, list);
        System.out.println(list.toString());
        System.out.println("missing number: " + found);
        System.out.println(removed == found ? "测试成功" : "测试失败");
    }
    static int findMissingNumber(int start, int end, List<Integer> list) {
        return 0;
    }
}
```

▶ Run

下载练习：找出缺失的数字 （推荐使用IDE练习插件快速下载）

小结

`List` 是按索引顺序访问的长度可变的有序表，优先使用 `ArrayList` 而不是 `LinkedList`；

可以直接使用 `for each` 遍历 `List`；

`List` 可以和 `Array` 相互转换。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





编写equals方法

我们知道 `List` 是一种有序链表： `List` 内部按照放入元素的先后顺序存放，并且每个元素都可以通过索引确定自己的位置。

`List` 还提供了 `boolean contains(Object o)` 方法来判断 `List` 是否包含某个指定元素。此外， `int indexOf(Object o)` 方法可以返回某个元素的索引，如果元素不存在，就返回 `-1` 。

我们来看一个例子：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("A", "B", "C");
        System.out.println(list.contains("C")); // true
        System.out.println(list.contains("X")); // false
        System.out.println(list.indexOf("C")); // 2
        System.out.println(list.indexOf("X")); // -1
    }
}
```

▶ Run

这里我们注意一个问题，我们往 `List` 中添加的 `"C"` 和调用 `contains("C")` 传入的 `"C"` 是不是同一个实例？

如果这两个 `"C"` 不是同一个实例，这段代码是否还能得到正确的结果？我们可以改写一下代码测试一下：

```
import java.util.List;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("A", "B", "C");
        System.out.println(list.contains(new String("C"))); // true or false?
        System.out.println(list.indexOf(new String("C"))); // 2 or -1?
    }
}
```

▶ Run

因为我们传入的是 `new String("C")` ，所以一定是不同的实例。结果仍然符合预期，这是为什么呢？

因为 `List` 内部并不是通过 `==` 判断两个元素是否相等，而是使用 `equals()` 方法判断两个元素

是否相等，例如 `contains()` 方法可以实现如下：

```
1.  public class ArrayList {
2.      Object[] elementData;
3.      public boolean contains(Object o) {
4.          for (int i = 0; i < size; i++) {
5.              if (o.equals(elementData[i])) {
6.                  return true;
7.              }
8.          }
9.          return false;
10.     }
11. }
```

因此，要正确使用 `List` 的 `contains()` 、 `indexOf()` 这些方法，放入的实例必须正确覆写 `equals()` 方法，否则，放进去的实例，查找不到。我们之所以能正常放入 `String` 、 `Integer` 这些对象，是因为Java标准库定义的这些类已经正确实现了 `equals()` 方法。

我们以 `Person` 对象为例，测试一下：

```
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Person> list = List.of(
            new Person("Xiao Ming"),
            new Person("Xiao Hong"),
            new Person("Bob")
        );
        System.out.println(list.contains(new Person("Bob"))); // false
    }
}

class Person {
    String name;
    public Person(String name) {
        this.name = name;
    }
}
```

▶ Run

不出意外，虽然放入了 `new Person("Bob")` ，但是用另一个 `new Person("Bob")` 查询不到，原因就是 `Person` 类没有覆写 `equals()` 方法。

编写equals

如何正确编写 `equals()` 方法？ `equals()` 方法要求我们必须满足以下条件：

- 自反性（Reflexive）：对于非 `null` 的 `x` 来说， `x.equals(x)` 必须返回 `true` ；
- 对称性（Symmetric）：对于非 `null` 的 `x` 和 `y` 来说，如果 `x.equals(y)` 为 `true` ，则 `y.equals(x)` 也必须为 `true` ；
- 传递性（Transitive）：对于非 `null` 的 `x` 、 `y` 和 `z` 来说，如果 `x.equals(y)` 为 `true` ， `y.equals(z)` 也为 `true` ，那么 `x.equals(z)` 也必须为 `true` ；
- 一致性（Consistent）：对于非 `null` 的 `x` 和 `y` 来说，只要 `x` 和 `y` 状态不变，则 `x.equals(y)` 总是一致地返回 `true` 或者 `false` ；
- 对 `null` 的比较：即 `x.equals(null)` 永远返回 `false` 。

上述规则看上去似乎非常复杂，但其实代码实现 `equals()` 方法是很简单的，我们以 `Person` 类为例：

```
1. public class Person {
2.     public String name;
3.     public int age;
4. }
```

首先，我们要定义“相等”的逻辑含义。对于 `Person` 类，如果 `name` 相等，并且 `age` 相等，我们就认为两个 `Person` 实例相等。

因此，编写 `equals()` 方法如下：

```
1. public boolean equals(Object o) {
2.     if (o instanceof Person) {
3.         Person p = (Person) o;
4.         return this.name.equals(p.name) && this.age == p.age;
5.     }
6.     return false;
7. }
```

对于引用字段比较，我们使用 `equals()` ，对于基本类型字段的比较，我们使用 `==` 。

如果 `this.name` 为 `null` ，那么 `equals()` 方法会报错，因此，需要继续改写如下：

```
1. public boolean equals(Object o) {
2.     if (o instanceof Person) {
3.         Person p = (Person) o;
4.         boolean nameEquals = false;
5.         if (this.name == null && p.name == null) {
```

```
            nameEquals = true;
        }
        if (this.name != null) {
            nameEquals = this.name.equals(p.name);
        }
        return nameEquals && this.age == p.age;
    }
    return false;
}
```

如果 `Person` 有好几个引用类型的字段，上面的写法就太复杂了。要简化引用类型的比较，我们使用 `Objects.equals()` 静态方法：

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        return Objects.equals(this.name, p.name) && this.age == p.age;
    }
    return false;
}
```

因此，我们总结一下 `equals()` 方法的正确编写方法：

- 先确定实例“相等”的逻辑，即哪些字段相等，就认为实例相等；
- 用 `instanceof` 判断传入的待比较的 `Object` 是不是当前类型，如果是，继续比较，否则，返回 `false`；
- 对引用类型用 `Objects.equals()` 比较，对基本类型直接用 `==` 比较。使用 `Objects.equals()` 比较两个引用类型是否相等的目的是省去了判断 `null` 的麻烦。两个引用类型都是 `null` 时它们也是相等的。

如果不调用 `List` 的 `contains()`、`indexOf()` 这些方法，那么放入的元素就不需要实现 `equals()` 方法。

练习

给Person类增加equals方法，使得调用indexOf()方法返回正常：

```
import java.util.List;
```

```
public class Main {
    public static void main(String[] args) {
        List<Person> list = List.of(
            new Person("Xiao", "Ming", 18),
            new Person("Xiao", "Hong", 25),
            new Person("Bob", "Smith", 20)
        );
        boolean exist = list.contains(new Person("Bob", "Smith", 20));
        System.out.println(exist ? "测试成功!" : "测试失败!");
    }
}

class Person {
    String firstName;
    String lastName;
    int age;
    public Person(String firstName, String lastName, int age) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
}
```

▶ Run

下载练习：覆写equals方法 （推荐使用IDE练习插件快速下载）

小结

在 `List` 中查找元素时， `List` 的实现类通过元素的 `equals()` 方法比较两个元素是否相等，因此，放入的元素必须正确覆写 `equals()` 方法，Java标准库提供的 `String` 、 `Integer` 等已经覆写了 `equals()` 方法；

编写 `equals()` 方法可借助 `Objects.equals()` 判断。

如果不在 `List` 中查找元素，就不必覆写 `equals()` 方法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Map

我们知道，`List`是一种顺序列表，如果有一个存储学生`Student`实例的`List`，要在`List`中根据`name`查找某个指定的`Student`的分数，应该怎么办？

最简单的方法是遍历`List`并判断`name`是否相等，然后返回指定元素：

```
List<Student> list = ...
Student target = null;
for (Student s : list) {
    if ("Xiao Ming".equals(s.name)) {
        target = s;
        break;
    }
}
System.out.println(target.score);
```

这种需求其实非常常见，即通过一个键去查询对应的值。使用`List`来实现存在效率非常低的问题，因为平均需要扫描一半的元素才能确定，而`Map`这种键值（key-value）映射表的数据结构，作用就是能高效通过`key`快速查找`value`（元素）。

用`Map`来实现根据`name`查询某个`Student`的代码如下：

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Student s = new Student("Xiao Ming", 99);
        Map<String, Student> map = new HashMap<>();
        map.put("Xiao Ming", s); // 将"Xiao Ming"和Student实例映射并关联
        Student target = map.get("Xiao Ming"); // 通过key查找并返回映射的Student实例
        System.out.println(target == s); // true，同一个实例
        System.out.println(target.score); // 99
        Student another = map.get("Bob"); // 通过另一个key查找
        System.out.println(another); // 未找到返回null
    }
}

class Student {
    public String name;
    public int score;
    public Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

▶ Run

通过上述代码可知：`Map<K, V>` 是一种键-值映射表，当我们调用 `put(K key, V value)` 方法时，就把 `key` 和 `value` 做了映射并放入 `Map`。当我们调用 `V get(K key)` 时，就可以通过 `key` 获取到对应的 `value`。如果 `key` 不存在，则返回 `null`。和 `List` 类似，`Map` 也是一个接口，最常用的实现类是 `HashMap`。

如果只是想查询某个 `key` 是否存在，可以调用 `boolean containsKey(K key)` 方法。

如果我们在存储 `Map` 映射关系的时候，对同一个key调用两次 `put()` 方法，分别放入不同的 `value`，会有什么问题呢？例如：

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        System.out.println(map.get("apple")); // 123
        map.put("apple", 789); // 再次放入apple作为key，但value变为789
        System.out.println(map.get("apple")); // 789
    }
}
```

▶ Run

重复放入 `key-value` 并不会有任何问题，但是一个 `key` 只能关联一个 `value` 。在上面的代码中，一开始我们把 `key` 对象 `"apple"` 映射到 `Integer` 对象 `123` ，然后再次调用 `put()` 方法把 `"apple"` 映射到 `789` ，这时，原来关联的 `value` 对象 `123` 就被“冲掉”了。实际上， `put()` 方法的签名是 `V put(K key, V value)` ，如果放入的 `key` 已经存在， `put()` 方法会返回被删除的旧的 `value` ，否则，返回 `null` 。

始终牢记：Map中不存在重复的key，因为放入相同的key，只会把原有的key-value对应的value给替换掉。

此外，在一个 `Map` 中，虽然 `key` 不能重复，但 `value` 是可以重复的：

```
1. Map<String, Integer> map = new HashMap<>();
2. map.put("apple", 123);
3. map.put("pear", 123); // ok
```

遍历Map

对 `Map` 来说，要遍历 `key` 可以使用 `for each` 循环遍历 `Map` 实例的 `keySet()` 方法返回的 `Set` 集合，它包含不重复的 `key` 的集合：

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (String key : map.keySet()) {
            Integer value = map.get(key);
            System.out.println(key + " = " + value);
        }
    }
}
```

▶ Run

同时遍历 `key` 和 `value` 可以使用 `for each` 循环遍历 `Map` 对象的 `entrySet()` 集合，它包含每一个 `key-value` 映射：

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new HashMap<>();
        map.put("apple", 123);
        map.put("pear", 456);
        map.put("banana", 789);
        for (Map.Entry<String, Integer> entry : map.entrySet()) {
            String key = entry.getKey();
            Integer value = entry.getValue();
            System.out.println(key + " = " + value);
        }
    }
}
```

▶ Run

`Map` 和 `List` 不同的是， `Map` 存储的是 `key-value` 的映射关系，并且，它不保证顺序。在遍历的时候，遍历的顺序既不一定是 `put()` 时放入的 `key` 的顺序，也不一定是 `key` 的排序顺序。使用 `Map` 时，任何依赖顺序的逻辑都是不可靠的。以 `HashMap` 为例，假设我们放入 `"A"` ， `"B"` ， `"C"` 这3个 `key` ，遍历的时候，每个 `key` 会保证被遍历一次且仅遍历一次，但顺序完全没有保证，甚至对于不同的JDK版本，相同的代码遍历的输出顺序都是不同的！

遍历Map时，不可假设输出的key是有序的！

练习

请编写一个根据 `name` 查找 `score` 的程序，并利用 `Map` 充当缓存，以提高查找效率：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Student> list = List.of(
            new Student("Bob", 78),
            new Student("Alice", 85),
            new Student("Brush", 66),
            new Student("Newton", 99));
        var holder = new Students(list);
        System.out.println(holder.getScore("Bob") == 78 ? "测试成功!" : "测试失败!");
        System.out.println(holder.getScore("Alice") == 85 ? "测试成功!" : "测试失败!");
        System.out.println(holder.getScore("Tom") == -1 ? "测试成功!" : "测试失败!");
    }
}

class Students {
    List<Student> list;
    Map<String, Integer> cache;

    Students(List<Student> list) {
        this.list = list;
        cache = new HashMap<>();
    }

    /**
     * 根据name查找score，找到返回score，未找到返回-1
     */
    int getScore(String name) {
        // 先在Map中查找:
        Integer score = this.cache.get(name);
        if (score == null) {
            // TODO:
        }
        return score == null ? -1 : score.intValue();
    }

    Integer findInList(String name) {
        for (var ss : this.list) {
            if (ss.name.equals(name)) {
                return ss.score;
            }
        }
        return null;
    }
}

class Student {
    String name;
    int score;

    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
}
```

▶ Run

下载练习：`find-student-score` （推荐使用IDE练习插件快速下载）

小结

`Map`是一种映射表，可以通过`key`快速查找`value`。

可以通过`for each`遍历`keySet()`，也可以通过`for each`遍历`entrySet()`，直接获取`key-value`。

最常用的一种`Map`实现是`HashMap`。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





编写equals和hashCode

我们知道Map是一种键-值（key-value）映射表，可以通过key快速查找对应的value。

以HashMap为例，观察下面的代码：

```
Map<String, Person> map = new HashMap<>();
map.put("a", new Person("Xiao Ming"));
map.put("b", new Person("Xiao Hong"));
map.put("c", new Person("Xiao Jun"));

map.get("a"); // Person("Xiao Ming")
map.get("x"); // null
```

`HashMap` 之所以能根据 `key` 直接拿到 `value`，原因是它内部通过空间换时间的方法，用一个大数组存储所有 `value`，并根据key直接计算出 `value` 应该存储在哪个索引：



如果 `key` 的值为 `"a"`，计算得到的索引总是 `1`，因此返回 `value` 为 `Person("Xiao Ming")`，如果 `key` 的值为 `"b"`，计算得到的索引总是 `5`，因此返回 `value` 为 `Person("Xiao Hong")`，这样，就不必遍历整个数组，即可直接读取 `key` 对应的 `value`。

当我们使用 `key` 存取 `value` 的时候，就会引出一个问题：

我们放入 `Map` 的 `key` 是字符串 `"a"` ，但是，当我们获取 `Map` 的 `value` 时，传入的变量不一定就是放入的那个 `key` 对象。

换句话讲，两个 `key` 应该是内容相同，但不一定是同一个对象。测试代码如下：

```
import java.util.HashMap;
import java.util.Map;

public class Main {
    public static void main(String[] args) {
        String key1 = "a";
        Map<String, Integer> map = new HashMap<>();
        map.put(key1, 123);

        String key2 = new String("a");
        map.get(key2); // 123

        System.out.println(key1 == key2); // false
        System.out.println(key1.equals(key2)); // true
    }
}
```

▶ Run

因为在 `Map` 的内部，对 `key` 做比较是通过 `equals()` 实现的，这一点和 `List` 查找元素需要正确覆写 `equals()` 是一样的，即正确使用 `Map` 必须保证：作为 `key` 的对象必须正确覆写 `equals()` 方法。

我们经常使用 `String` 作为 `key` ，因为 `String` 已经正确覆写了 `equals()` 方法。但如果我们放入的 `key` 是一个自己写的类，就必须保证正确覆写了 `equals()` 方法。

我们再思考一下 `HashMap` 为什么能通过 `key` 直接计算出 `value` 存储的索引。相同的 `key` 对象（使用 `equals()` 判断时返回 `true` ）必须要计算出相同的索引，否则，相同的 `key` 每次取出的 `value` 就不一定对。

通过 `key` 计算索引的方式就是调用 `key` 对象的 `hashCode()` 方法，它返回一个 `int` 整数。 `HashMap` 正是通过这个方法直接定位 `key` 对应的 `value` 的索引，继而直接返回 `value` 。

因此，正确使用 `Map` 必须保证：

- 作为 `key` 的对象必须正确覆写 `equals()` 方法，相等的两个 `key` 实例调用 `equals()` 必须返回 `true` ；
- 作为 `key` 的对象还必须正确覆写 `hashCode()` 方法，且 `hashCode()` 方法要严格遵循以下规范：
- 如果两个对象相等，则两个对象的 `hashCode()` 必须相等；

- 如果两个对象不相等，则两个对象的`hashCode()`尽量不要相等。

即对应两个实例`a`和`b`：

- 如果`a`和`b`相等，那么`a.equals(b)`一定为`true`，则`a.hashCode()`必须等于`b.hashCode()`；
- 如果`a`和`b`不相等，那么`a.equals(b)`一定为`false`，则`a.hashCode()`和`b.hashCode()`尽量不要相等。

上述第一条规范是正确性，必须保证实现，否则`HashMap`不能正常工作。

而第二条如果尽量满足，则可以保证查询效率，因为不同的对象，如果返回相同的`hashCode()`，会造成`Map`内部存储冲突，使存取的效率下降。

正确编写`equals()`的方法我们已经在编写equals方法一节中讲过了，以`Person`类为例：

```
public class Person {
    String firstName;
    String lastName;
    int age;
}
```

把需要比较的字段找出来：

- firstName
- lastName
- age

然后，引用类型使用`Objects.equals()`比较，基本类型使用`==`比较。

在正确实现`equals()`的基础上，我们还需要正确实现`hashCode()`，即上述3个字段分别相同的实例，`hashCode()`返回的`int`必须相同：

```
public class Person {
    String firstName;
    String lastName;
    int age;

    @Override
    int hashCode() {
        int h = 0;
        h = 31 * h + firstName.hashCode();
        h = 31 * h + lastName.hashCode();
        h = 31 * h + age;
```

```
        return h;
    }
}
```

注意到 `String` 类已经正确实现了 `hashCode()` 方法，我们在计算 `Person` 的 `hashCode()` 时，反复使用 `31*h` ，这样做的目的是为了尽量把不同的 `Person` 实例的 `hashCode()` 均匀分布到整个 `int` 范围。

和实现 `equals()` 方法遇到的问题类似，如果 `firstName` 或 `lastName` 为 `null` ，上述代码工作起来就会抛 `NullPointerException` 。为了解决这个问题，我们在计算 `hashCode()` 的时候，经常借助 `Objects.hash()` 来计算：

```
int hashCode() {
    return Objects.hash(firstName, lastName, age);
}
```

所以，编写 `equals()` 和 `hashCode()` 遵循的原则是：

`equals()` 用到的用于比较的每一个字段，都必须在 `hashCode()` 中用于计算； `equals()` 中没有使用到的字段，绝不可放在 `hashCode()` 中计算。

另外注意，对于放入 `HashMap` 的 `value` 对象，没有任何要求。

延伸阅读

既然 `HashMap` 内部使用了数组，通过计算 `key` 的 `hashCode()` 直接定位 `value` 所在的索引，那么第一个问题来了：hashCode()返回的 `int` 范围高达±21亿，先不考虑负数， `HashMap` 内部使用的数组得有多大？

实际上 `HashMap` 初始化时默认的数组大小只有16，任何 `key` ，无论它的 `hashCode()` 有多大，都可以简单地通过：

```
int index = key.hashCode() & 0xf; // 0xf = 15
```

把索引确定在0～15，即永远不会超出数组范围，上述算法只是一种最简单的实现。

第二个问题：如果添加超过16个 `key-value` 到 `HashMap` ，数组不够用了怎么办？

添加超过一定数量的 `key-value` 时， `HashMap` 会在内部自动扩容，每次扩容一倍，即长度为16的数组扩展为长度32，相应地，需要重新确定 `hashCode()` 计算的索引位置。例如，对长度为32的数组计算 `hashCode()` 对应的索引，计算方式要改为：

```
int index = key.hashCode() & 0x1f; // 0x1f = 31
```

由于扩容会导致重新分布已有的 `key-value` ，所以，频繁扩容对 `HashMap` 的性能影响很大。如果我们确定要使用一个容量为 `10000` 个 `key-value` 的 `HashMap` ，更好的方式是创建 `HashMap` 时就指定容量：

```
Map<String, Integer> map = new HashMap<>(10000);
```

虽然指定容量是 `10000` ，但 `HashMap` 内部的数组长度总是2^n，因此，实际数组长度被初始化为比 `10000` 大的 `16384` （2^{14}）。

最后一个问题：如果不同的两个 `key` ，例如 `"a"` 和 `"b"` ，它们的 `hashCode()` 恰好是相同的（这种情况是完全可能的，因为不相等的两个实例，只要求 `hashCode()` 尽量不相等），那么，当我们放入：

```
map.put("a", new Person("Xiao Ming"));
map.put("b", new Person("Xiao Hong"));
```

时，由于计算出的数组索引相同，后面放入的 `"Xiao Hong"` 会不会把 `"Xiao Ming"` 覆盖了？

当然不会！使用 `Map` 的时候，只要 `key` 不相同，它们映射的 `value` 就互不干扰。但是，在 `HashMap` 内部，确实可能存在不同的 `key` ，映射到相同的 `hashCode()` ，即相同的数组索引上，肿么办？

我们就假设 `"a"` 和 `"b"` 这两个 `key` 最终计算出的索引都是5，那么，在 `HashMap` 的数组中，实际存储的不是一个 `Person` 实例，而是一个 `List` ，它包含两个 `Entry` ，一个是 `"a"` 的映射，一个是 `"b"` 的映射：

```
   ┌──┐
0  │  │
   ├──┤
1  │  │
   ├──┤
2  │  │
   ├──┤
3  │  │
   ├──┤
4  │  │
   ├──┤
5  │ ●┼──> List<Entry<String, Person>>
   ├──┤
6  │  │
   ├──┤
```

```
16.  7 |    |
17.    └────┘
```

在查找的时候，例如：

```
1.  Person p = map.get("a");
```

HashMap内部通过 `"a"` 找到的实际上是 `List<Entry<String, Person>>` ，它还需要遍历这个 `List` ，并找到一个 `Entry` ，它的 `key` 字段是 `"a"` ，才能返回对应的 `Person` 实例。

我们把不同的 `key` 具有相同的 `hashCode()` 的情况称之为哈希冲突。在冲突的时候，一种最简单的解决办法是用 `List` 存储 `hashCode()` 相同的 `key-value` 。显然，如果冲突的概率越大，这个 `List` 就越长， `Map` 的 `get()` 方法效率就越低，这就是为什么要尽量满足条件二：

如果两个对象不相等，则两个对象的hashCode()尽量不要相等。

`hashCode()` 方法编写得越好， `HashMap` 工作的效率就越高。

小结

要正确使用 `HashMap` ，作为 `key` 的类必须正确覆写 `equals()` 和 `hashCode()` 方法；

一个类如果覆写了 `equals()` ，就必须覆写 `hashCode()` ，并且覆写规则是：

- 如果 `equals()` 返回 `true` ，则 `hashCode()` 返回值必须相等；
- 如果 `equals()` 返回 `false` ，则 `hashCode()` 返回值尽量不要相等。

实现 `hashCode()` 方法可以通过 `Objects.hashCode()` 辅助方法实现。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用EnumMap

因为 `HashMap` 是一种通过对key计算 `hashCode()` ，通过空间换时间的方式，直接定位到value所在的内部数组的索引，因此，查找效率非常高。

如果作为key的对象是 `enum` 类型，那么，还可以使用Java集合库提供的一种 `EnumMap` ，它在内部以一个非常紧凑的数组存储value，并且根据 `enum` 类型的key直接定位到内部数组的索引，并不需要计算 `hashCode()` ，不但效率最高，而且没有额外的空间浪费。

我们以 `DayOfWeek` 这个枚举类型为例，为它做一个“翻译”功能：

```
import java.time.DayOfWeek;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Map<DayOfWeek, String> map = new EnumMap<>(DayOfWeek.class);
        map.put(DayOfWeek.MONDAY, "星期一");
        map.put(DayOfWeek.TUESDAY, "星期二");
        map.put(DayOfWeek.WEDNESDAY, "星期三");
        map.put(DayOfWeek.THURSDAY, "星期四");
        map.put(DayOfWeek.FRIDAY, "星期五");
        map.put(DayOfWeek.SATURDAY, "星期六");
        map.put(DayOfWeek.SUNDAY, "星期日");
        System.out.println(map);
        System.out.println(map.get(DayOfWeek.MONDAY));
    }
}
```

▶ Run

使用 `EnumMap` 的时候，我们总是用 `Map` 接口来引用它，因此，实际上把 `HashMap` 和 `EnumMap` 互换，在客户端看来没有任何区别。

小结

如果 `Map` 的key是 `enum` 类型，推荐使用 `EnumMap` ，既保证速度，也不浪费空间。

使用 `EnumMap` 的时候，根据面向抽象编程的原则，应持有 `Map` 接口。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论

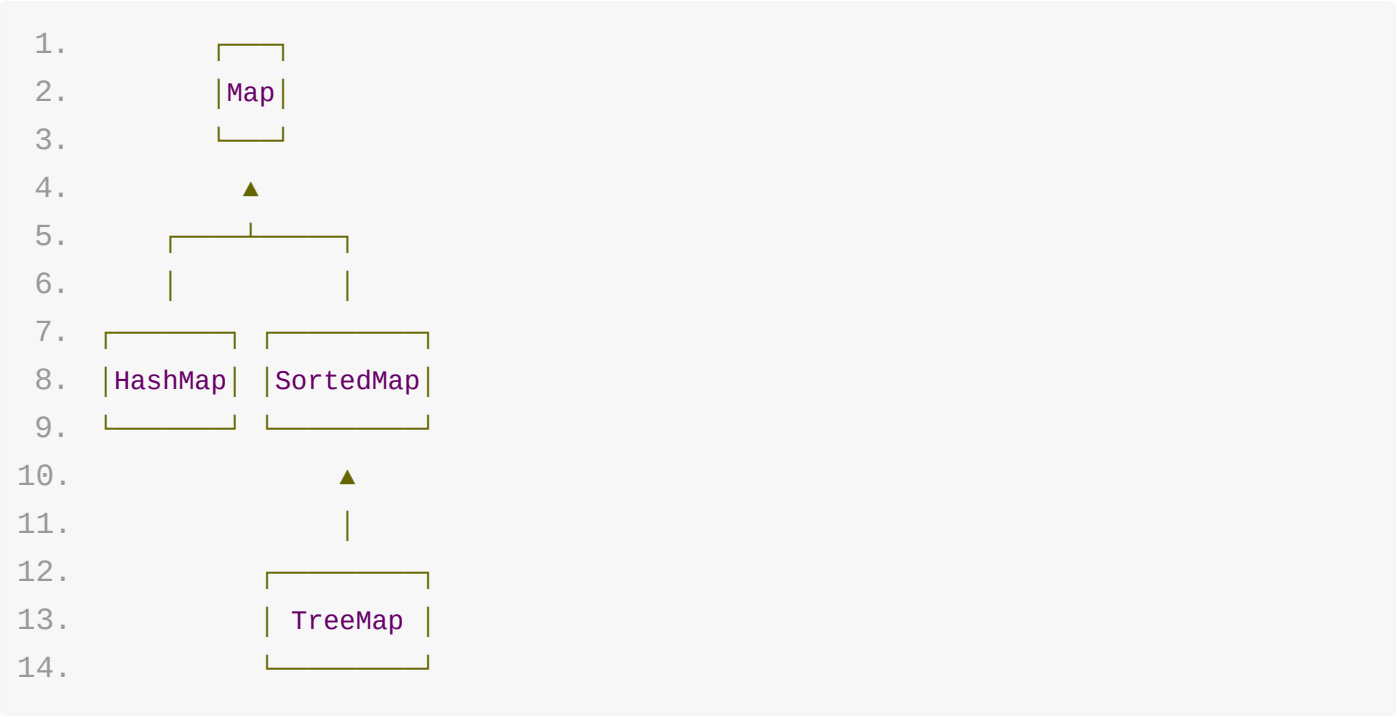




使用TreeMap

我们已经知道，`HashMap`是一种以空间换时间的映射表，它的实现原理决定了内部的Key是无序的，即遍历`HashMap`的Key时，其顺序是不可预测的（但每个Key都会遍历一次且仅遍历一次）。

还有一种`Map`，它在内部会对Key进行排序，这种`Map`就是`SortedMap`。注意到`SortedMap`是接口，它的实现类是`TreeMap`。



`SortedMap`保证遍历时以Key的顺序来进行排序。例如，放入的Key是`"apple"`、`"pear"`、`"orange"`，遍历的顺序一定是`"apple"`、`"orange"`、`"pear"`，因为`String`默认按字母排序：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<String, Integer> map = new TreeMap<>();
        map.put("orange", 1);
        map.put("apple", 2);
        map.put("pear", 3);
        for (String key : map.keySet()) {
            System.out.println(key);
        }
        // apple, orange, pear
    }
}
```

▶ Run

使用`TreeMap`时，放入的Key必须实现`Comparable`接口。`String`、`Integer`这些类已经

实现了 `Comparable` 接口，因此可以直接作为Key使用。作为Value的对象则没有任何要求。

如果作为Key的class没有实现 `Comparable` 接口，那么，必须在创建 `TreeMap` 时同时指定一个自定义排序算法：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Map<Person, Integer> map = new TreeMap<>(new Comparator<Person>() {
            public int compare(Person p1, Person p2) {
                return p1.name.compareTo(p2.name);
            }
        });
        map.put(new Person("Tom"), 1);
        map.put(new Person("Bob"), 2);
        map.put(new Person("Lily"), 3);
        for (Person key : map.keySet()) {
            System.out.println(key);
        }
        // {Person: Bob}, {Person: Lily}, {Person: Tom}
        System.out.println(map.get(new Person("Bob"))); // 2
    }
}

class Person {
    public String name;
    Person(String name) {
        this.name = name;
    }
    public String toString() {
        return "{Person: " + name + "}";
    }
}
```

▶ Run

注意到 `Comparator` 接口要求实现一个比较方法，它负责比较传入的两个元素 `a` 和 `b` ，如果 `a<b` ，则返回负数，通常是 `-1` ，如果 `a==b` ，则返回 `0` ，如果 `a>b` ，则返回正数，通常是 `1` 。 `TreeMap` 内部根据比较结果对Key进行排序。

从上述代码执行结果可知，打印的Key确实是按照 `Comparator` 定义的顺序排序的。如果要根据Key查找Value，我们可以传入一个 `new Person("Bob")` 作为Key，它会返回对应的 `Integer` 值 `2` 。

另外，注意到 `Person` 类并未覆写 `equals()` 和 `hashCode()` ，因为 `TreeMap` 不使用 `equals()` 和 `hashCode()` 。

我们来看一个稍微复杂的例子：这次我们定义了 `Student` 类，并用分数 `score` 进行排序，高分在前：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Map<Student, Integer> map = new TreeMap<>(new Comparator<Student>() {
            public int compare(Student p1, Student p2) {
                return p1.score > p2.score ? -1 : 1;
            }
        });
        map.put(new Student("Tom", 77), 1);
        map.put(new Student("Bob", 66), 2);
        map.put(new Student("Lily", 99), 3);
        for (Student key : map.keySet()) {
            System.out.println(key);
        }
        System.out.println(map.get(new Student("Bob", 66))); // null?
    }
}

class Student {
    public String name;
    public int score;
    Student(String name, int score) {
        this.name = name;
        this.score = score;
    }
    public String toString() {
        return String.format("{%s: score=%d}", name, score);
    }
}
```

▶ Run

在 `for` 循环中，我们确实得到了正确的顺序。但是，且慢！根据相同的Key: `new Student("Bob", 66)` 进行查找时，结果为 `null` ！

这是怎么肥四？难道 `TreeMap` 有问题？遇到 `TreeMap` 工作不正常时，我们首先回顾Java编程基本规则：出现问题，不要怀疑Java标准库，要从自身代码找原因。

在这个例子中， `TreeMap` 出现问题，原因其实出在这个 `Comparator` 上：

```
1.  public int compare(Student p1, Student p2) {
2.      return p1.score > p2.score ? -1 : 1;
3.  }
```

在 `p1.score` 和 `p2.score` 不相等的时候，它的返回值是正确的，但是，
在 `p1.score` 和 `p2.score` 相等的时候，它并没有返回 `0` ！这就是为什么 `TreeMap` 工作不正常的原因： `TreeMap` 在比较两个Key是否相等时，依赖Key的 `compareTo()` 方法或者 `Comparator.compare()` 方法。在两个Key相等时，必须返回 `0` 。因此，修改代码如下：

```
1.  public int compare(Student p1, Student p2) {
```

```
    if (p1.score == p2.score) {
        return 0;
    }
    return p1.score > p2.score ? -1 : 1;
}
```

或者直接借助 `Integer.compare(int, int)` 也可以返回正确的比较结果。

小结

`SortedMap` 在遍历时严格按照Key的顺序遍历，最常用的实现类是 `TreeMap` ；

作为 `SortedMap` 的Key必须实现 `Comparable` 接口，或者传入 `Comparator` ；

要严格按照 `compare()` 规范实现比较逻辑，否则， `TreeMap` 将不能正常工作。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Properties

在编写应用程序的时候，经常需要读写配置文件。例如，用户的设置：

```
# 上次最后打开的文件:
last_open_file=/data/hello.txt
# 自动保存文件的时间间隔:
auto_save_interval=60
```

配置文件的特点是，它的Key-Value一般都是 `String` - `String` 类型的，因此我们完全可以用 `Map<String, String>` 来表示它。

因为配置文件非常常用，所以Java集合库提供了一个 `Properties` 来表示一组“配置”。由于历史遗留原因， `Properties` 内部本质上是一个 `Hashtable` ，但我们只需要用到 `Properties` 自身关于读写配置的接口。

读取配置文件

用 `Properties` 读取配置文件非常简单。Java默认配置文件以 `.properties` 为扩展名，每行以 `key=value` 表示，以 `#` 课开头的是注释。以下是一个典型的配置文件：

```
# setting.properties

last_open_file=/data/hello.txt
auto_save_interval=60
```

可以从文件系统读取这个 `.properties` 文件：

```
String f = "setting.properties";
Properties props = new Properties();
props.load(new java.io.FileInputStream(f));

String filepath = props.getProperty("last_open_file");
String interval = props.getProperty("auto_save_interval", "120");
```

可见，用 `Properties` 读取配置文件，一共有三步：

- 创建 `Properties` 实例；
- 调用 `load()` 读取文件；
- 调用 `getProperty()` 获取配置。调用 `getProperty()` 获取配置时，如果key不存在，将返回 `null` 。我们还可以提供一个默认值，这样，当key不存在的时候，就返回默认值。

也可以从classpath读取 `.properties` 文件，因为 `load(InputStream)` 方法接收一个 `InputStream` 实例，表示一个字节流，它不一定是文件流，也可以是从jar包中读取的资源流：

```
1. Properties props = new Properties();
2. props.load(getClass().getResourceAsStream("/common/setting.properties"));
```

试试从内存读取一个字节流：

```
// properties
import java.io.*;
import java.util.Properties;

public class Main {
    public static void main(String[] args) throws IOException {
        String settings = "# test" + "\n" + "course=Java" + "\n" + "last_open_date=2019-08-
07T12:35:01";
        ByteArrayInputStream input = new ByteArrayInputStream(settings.getBytes("UTF-8"));
        Properties props = new Properties();
        props.load(input);

        System.out.println("course: " + props.getProperty("course"));
        System.out.println("last_open_date: " + props.getProperty("last_open_date"));
        System.out.println("last_open_file: " + props.getProperty("last_open_file"));
        System.out.println("auto_save: " + props.getProperty("auto_save", "60"));
    }
}
```

▶ Run

如果有多个 `.properties` 文件，可以反复调用 `load()` 读取，后读取的key-value会覆盖已读取的key-value：

```
1. Properties props = new Properties();
2. props.load(getClass().getResourceAsString("/common/setting.properties"));
3. props.load(new FileInputStream("C:\\conf\\setting.properties"));
```

上面的代码演示了 `Properties` 的一个常用用法：可以把默认配置文件放到classpath中，然后，根据机器的环境编写另一个配置文件，覆盖某些默认的配置。

`Properties` 设计的目的是存储 `String` 类型的key－value，但 `Properties` 实际上是从 `Hashtable` 派生的，它的设计实际上是有问题的，但是为了保持兼容性，现在已经没法修改了。除了 `getProperty()` 和 `setProperty()` 方法外，还有从 `Hashtable` 继承下来的 `get()` 和 `put()` 方法，这些方法的参数签名是 `Object` ，我们在使用 `Properties` 的时候，不要去调用这些从 `Hashtable` 继承下来的方法。

写入配置文件

如果通过 `setProperty()` 修改了 `Properties` 实例，可以把配置写入文件，以便下次启动时获得最新配置。写入配置文件使用 `store()` 方法：

```
Properties props = new Properties();
props.setProperty("url", "http://www.liaoxuefeng.com");
props.setProperty("language", "Java");
props.store(new FileOutputStream("C:\\conf\\setting.properties"), "这是写入的properties注释");
```

编码

早期版本的Java规定 `.properties` 文件编码是ASCII编码（ISO8859-1），如果涉及到中文就必须用 `name=\u4e2d\u6587` 来表示，非常别扭。从JDK9开始，Java的 `.properties` 文件可以使用UTF-8编码了。

不过，需要注意的是，由于 `load(InputStream)` 默认总是以ASCII编码读取字节流，所以会导致读到乱码。我们需要用另一个重载方法 `load(Reader)` 读取：

```
Properties props = new Properties();
props.load(new FileReader("settings.properties", StandardCharsets.UTF_8));
```

就可以正常读取中文。 `InputStream` 和 `Reader` 的区别是一个是字节流，一个是字符流。字符流在内存中已经以 `char` 类型表示了，不涉及编码问题。

小结

Java集合库提供的 `Properties` 用于读写配置文件 `.properties` 。 `.properties` 文件可以使用UTF-8编码。

可以从文件系统、classpath或其他任何地方读取 `.properties` 文件。

读写 `Properties` 时，注意仅使用 `getProperty()` 和 `setProperty()` 方法，不要调用继承而来的 `get()` 和 `put()` 等方法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Set

我们知道，`Map`用于存储key-value的映射，对于充当key的对象，是不能重复的，并且，不但需要正确覆写`equals()`方法，还要正确覆写`hashCode()`方法。

如果我们只需要存储不重复的key，并不需要存储映射的value，那么就可以使用`Set`。

`Set`用于存储不重复的元素集合，它主要提供以下几个方法：

- 将元素添加进`Set<E>`：`boolean add(E e)`
- 将元素从`Set<E>`删除：`boolean remove(Object e)`
- 判断是否包含元素：`boolean contains(Object e)`

我们来看几个简单的例子：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        System.out.println(set.add("abc")); // true
        System.out.println(set.add("xyz")); // true
        System.out.println(set.add("xyz")); // false，添加失败，因为元素已存在
        System.out.println(set.contains("xyz")); // true，元素存在
        System.out.println(set.contains("XYZ")); // false，元素不存在
        System.out.println(set.remove("hello")); // false，删除失败，因为元素不存在
        System.out.println(set.size()); // 2，一共两个元素
    }
}
```

▶ Run

`Set`实际上相当于只存储key、不存储value的`Map`。我们经常用`Set`用于去除重复元素。

因为放入`Set`的元素和`Map`的key类似，都要正确实现`equals()`和`hashCode()`方法，否则该元素无法正确地放入`Set`。

最常用的`Set`实现类是`HashSet`，实际上，`HashSet`仅仅是对`HashMap`的一个简单封装，它的核心代码如下：

```
public class HashSet<E> implements Set<E> {
    // 持有一个HashMap:
    private HashMap<E, Object> map = new HashMap<>();

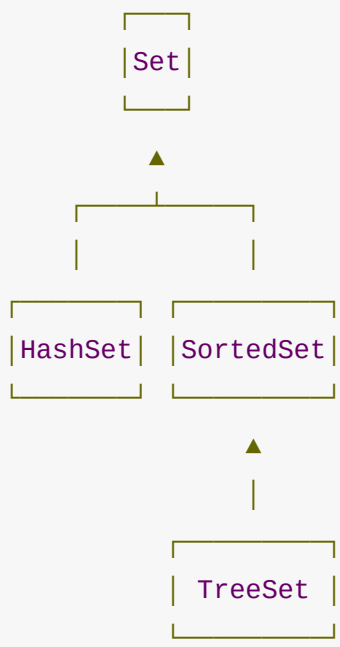
    // 放入HashMap的value:
    private static final Object PRESENT = new Object();
```

```
7.
8.      public boolean add(E e) {
9.          return map.put(e, PRESENT) == null;
10.     }
11.
12.     public boolean contains(Object o) {
13.         return map.containsKey(o);
14.     }
15.
16.     public boolean remove(Object o) {
17.         return map.remove(o) == PRESENT;
18.     }
19. }
```

`Set` 接口并不保证有序，而 `SortedSet` 接口则保证元素是有序的：

- `HashSet` 是无序的，因为它实现了 `Set` 接口，并没有实现 `SortedSet` 接口；
- `TreeSet` 是有序的，因为它实现了 `SortedSet` 接口。

用一张图表示：



我们来看 `HashSet` 的输出：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

▶ Run

注意输出的顺序既不是添加的顺序，也不是 `String` 排序的顺序，在不同版本的JDK中，这个顺序也可能是不同的。

把 `HashSet` 换成 `TreeSet` ，在遍历 `TreeSet` 时，输出就是有序的，这个顺序是元素的排序顺序：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        Set<String> set = new TreeSet<>();
        set.add("apple");
        set.add("banana");
        set.add("pear");
        set.add("orange");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

▶ Run

使用 `TreeSet` 和使用 `TreeMap` 的要求一样，添加的元素必须正确实现 `Comparable` 接口，如果没有实现 `Comparable` 接口，那么创建 `TreeSet` 时必须传入一个 `Comparator` 对象。

练习

在聊天软件中，发送方发送消息时，遇到网络超时后就会自动重发，因此，接收方可能会收到重复的消息，在显示给用户看的时候，需要首先去重。请练习使用 `Set` 去除重复的消息：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Message> received = List.of(
            new Message(1, "Hello!"),
            new Message(2, "发工资了吗？"),
            new Message(2, "发工资了吗？"),
            new Message(3, "去哪吃饭？"),
            new Message(3, "去哪吃饭？"),
            new Message(4, "Bye")
        );
        List<Message> displayMessages = process(received);
        for (Message message : displayMessages) {
            System.out.println(message.text);
        }
    }

    static List<Message> process(List<Message> received) {
        // TODO: 按sequence去除重复消息
        return received;
    }
}

class Message {
    public final int sequence;
    public final String text;
    public Message(int sequence, String text) {
        this.sequence = sequence;
        this.text = text;
    }
}
```

▶ Run

小结

`Set` 用于存储不重复的元素集合：

- 放入 `HashSet` 的元素与作为 `HashMap` 的key要求相同；
- 放入 `TreeSet` 的元素与作为 `TreeMap` 的Key要求相同；

利用 `Set` 可以去除重复元素；

遍历 `SortedSet` 按照元素的排序顺序遍历，也可以自定义排序算法。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论



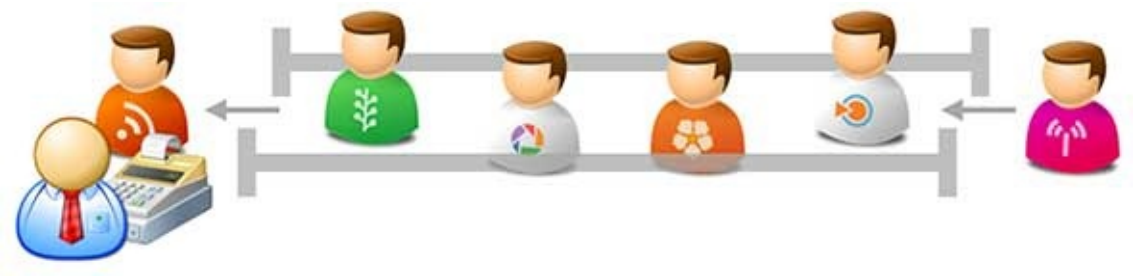


使用Queue

队列（`Queue`）是一种经常使用的集合。`Queue`实际上是实现了一个先进先出（FIFO: First In First Out）的有序表。它和`List`的区别在于，`List`可以在任意位置添加和删除元素，而`Queue`只有两个操作：

- 把元素添加到队列末尾；
- 从队列头部取出元素。

超市的收银台就是一个队列：



在Java的标准库中，队列接口`Queue`定义了以下几个方法：

- `int size()`：获取队列长度；
- `boolean add(E)` / `boolean offer(E)`：添加元素到队尾；
- `E remove()` / `E poll()`：获取队首元素并从队列中删除；
- `E element()` / `E peek()`：获取队首元素但并不从队列中删除。

对于具体的实现类，有的Queue有最大队列长度限制，有的Queue没有。注意到添加、删除和获取队列元素总是有两个方法，这是因为在添加或获取元素失败时，这两个方法的行为是不同的。我们用一个表格总结如下：

	throw Exception	返回false或null
添加元素到队尾	add(E e)	boolean offer(E e)
取队首元素并删除	E remove()	E poll()
取队首元素但不删除	E element()	E peek()

举个栗子，假设我们有一个队列，对它做一个添加操作，如果调用`add()`方法，当添加失败时（可能超过了队列的容量），它会抛出异常：

```
Queue<String> q = ...
try {
    q.add("Apple");
    System.out.println("添加成功");
} catch(IllegalStateException e) {
```

```
    System.out.println("添加失败");
}
```

如果我们调用 `offer()` 方法来添加元素，当添加失败时，它不会抛异常，而是返回 `false`：

```
Queue<String> q = ...
if (q.offer("Apple")) {
    System.out.println("添加成功");
} else {
    System.out.println("添加失败");
}
```

当我们需要从 `Queue` 中取出队首元素时，如果当前 `Queue` 是一个空队列，调用 `remove()` 方法，它会抛出异常：

```
Queue<String> q = ...
try {
    String s = q.remove();
    System.out.println("获取成功");
} catch(IllegalStateException e) {
    System.out.println("获取失败");
}
```

如果我们调用 `poll()` 方法来取出队首元素，当获取失败时，它不会抛异常，而是返回 `null`：

```
Queue<String> q = ...
String s = q.poll();
if (s != null) {
    System.out.println("获取成功");
} else {
    System.out.println("获取失败");
}
```

因此，两套方法可以根据需要来选择使用。

注意：不要把 `null` 添加到队列中，否则 `poll()` 方法返回 `null` 时，很难确定是取到了 `null` 元素还是队列为空。

接下来我们以 `poll()` 和 `peek()` 为例来说说“获取并删除”与“获取但不删除”的区别。对于 `Queue` 来说，每次调用 `poll()`，都会获取队首元素，并且获取到的元素已经从队列中被删除了：

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        // 从队列取出元素:
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // null,因为队列是空的
    }
}
```

▶ Run

如果用 `peek()` ，因为获取队首元素时，并不会从队列中删除这个元素，所以可以反复获取：

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<String> q = new LinkedList<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        // 队首永远都是apple，因为peek()不会删除它:
        System.out.println(q.peek()); // apple
        System.out.println(q.peek()); // apple
        System.out.println(q.peek()); // apple
    }
}
```

▶ Run

从上面的代码中，我们还可以发现， `LinkedList` 即实现了 `List` 接口，又实现了 `Queue` 接口，但是，在使用的时候，如果我们把它当作List，就获取List的引用，如果我们把它当作Queue，就获取Queue的引用：

```
// 这是一个List:
List<String> list = new LinkedList<>();
// 这是一个Queue:
Queue<String> queue = new LinkedList<>();
```

始终按照面向抽象编程的原则编写代码，可以大大提高代码的质量。

小结

队列 `Queue` 实现了一个先进先出（FIFO）的数据结构：

- 通过 `add()` / `offer()` 方法将元素添加到队尾；
- 通过 `remove()` / `poll()` 从队首获取元素并删除；
- 通过 `element()` / `peek()` 从队首获取元素但不删除。

要避免把 `null` 添加到队列。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用PriorityQueue

我们知道，`Queue`是一个先进先出（FIFO）的队列。

在银行柜台办业务时，我们假设只有一个柜台在办理业务，但是办理业务的人很多，怎么办？

可以每个人先取一个号，例如：`A1`、`A2`、`A3`……然后，按照号码顺序依次办理，实际上这就是一个`Queue`。

如果这时来了一个VIP客户，他的号码是`V1`，虽然当前排队的是`A10`、`A11`、`A12`……但是柜台下一个呼叫的客户号码却是`V1`。

这个时候，我们发现，要实现“VIP插队”的业务，用`Queue`就不行了，因为`Queue`会严格按FIFO的原则取出队首元素。我们需要的是优先队列：`PriorityQueue`。

`PriorityQueue`和`Queue`的区别在于，它的出队顺序与元素的优先级有关，对`PriorityQueue`调用`remove()`或`poll()`方法，返回的总是优先级最高的元素。

要使用`PriorityQueue`，我们就必须给每个元素定义“优先级”。我们以实际代码为例，先看看`PriorityQueue`的行为：

```
import java.util.PriorityQueue;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<String> q = new PriorityQueue<>();
        // 添加3个元素到队列:
        q.offer("apple");
        q.offer("pear");
        q.offer("banana");
        System.out.println(q.poll()); // apple
        System.out.println(q.poll()); // banana
        System.out.println(q.poll()); // pear
        System.out.println(q.poll()); // null,因为队列为空
    }
}
```

▶ Run

我们放入的顺序是`"apple"`、`"pear"`、`"banana"`，但是取出的顺序却是`"apple"`、`"banana"`、`"pear"`，这是因为从字符串的排序看，`"apple"`排在最前面，`"pear"`排在最后面。

因此，放入`PriorityQueue`的元素，必须实现`Comparable`接口，`PriorityQueue`会根据元素的排序顺序决定出队的优先级。

如果我们要放入的元素并没有实现 `Comparable` 接口怎么办？ `PriorityQueue` 允许我们提供一个 `Comparator` 对象来判断两个元素的顺序。我们以银行排队业务为例，实现一个 `PriorityQueue` ：

```
import java.util.Comparator;
import java.util.PriorityQueue;
import java.util.Queue;
```

```
public class Main {
    public static void main(String[] args) {
        Queue<User> q = new PriorityQueue<>(new UserComparator());
        // 添加3个元素到队列:
        q.offer(new User("Bob", "A1"));
        q.offer(new User("Alice", "A2"));
        q.offer(new User("Boss", "V1"));
        System.out.println(q.poll()); // Boss/V1
        System.out.println(q.poll()); // Bob/A1
        System.out.println(q.poll()); // Alice/A2
        System.out.println(q.poll()); // null,因为队列为空
    }
}

class UserComparator implements Comparator<User> {
    public int compare(User u1, User u2) {
        if (u1.number.charAt(0) == u2.number.charAt(0)) {
            // 如果两人的号都是A开头或者都是V开头,比较号的大小:
            return u1.number.compareTo(u2.number);
        }
        if (u1.number.charAt(0) == 'V') {
            // u1的号码是V开头,优先级高:
            return -1;
        } else {
            return 1;
        }
    }
}

class User {
    public final String name;
    public final String number;

    public User(String name, String number) {
        this.name = name;
        this.number = number;
    }

    public String toString() {
        return name + "/" + number;
    }
}
```

▶ Run

实现 `PriorityQueue` 的关键在于提供的 `UserComparator` 对象，它负责比较两个元素的大小（较小的在前）。 `UserComparator` 总是把 `V` 开头的号码优先返回，只有在开头相同的时候，才比较号码大小。

上面的 `UserComparator` 的比较逻辑其实还是有问题的，它会把 `A10` 排在 `A2` 的前面，请尝试修复该错误。

小结

`PriorityQueue` 实现了一个优先队列：从队首获取元素时，总是获取优先级最高的元素。

`PriorityQueue` 默认按元素比较的顺序排序（必须实现 `Comparable` 接口），也可以通过 `Comparator` 自定义排序算法（元素就不必实现 `Comparable` 接口）。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Deque

我们知道，`Queue`是队列，只能一头进，另一头出。

如果把条件放松一下，允许两头都进，两头都出，这种队列叫双端队列（Double Ended Queue），学名`Deque`。

Java集合提供了接口`Deque`来实现一个双端队列，它的功能是：

- 既可以添加到队尾，也可以添加到队首；
- 既可以从队首获取，又可以从队尾获取。

我们来比较一下`Queue`和`Deque`出队和入队的方法：

	Queue	Deque
添加元素到队尾	add(E e) / offer(E e)	addLast(E e) / offerLast(E e)
取队首元素并删除	E remove() / E poll()	E removeFirst() / E pollFirst()
取队首元素但不删除	E element() / E peek()	E getFirst() / E peekFirst()
添加元素到队首	无	addFirst(E e) / offerFirst(E e)
取队尾元素并删除	无	E removeLast() / E pollLast()
取队尾元素但不删除	无	E getLast() / E peekLast()

对于添加元素到队尾的操作，`Queue`提供了`add()`/`offer()`方法，而`Deque`提供了`addLast()`/`offerLast()`方法。添加元素到对首、取队尾元素的操作在`Queue`中不存在，在`Deque`中由`addFirst()`/`removeLast()`等方法提供。

注意到`Deque`接口实际上扩展自`Queue`：

```
public interface Deque<E> extends Queue<E> {
    ...
}
```

因此，`Queue`提供的`add()`/`offer()`方法在`Deque`中也可以使用，但是，使用`Deque`，最好不要调用`offer()`，而是调用`offerLast()`：

```
import java.util.Deque;
import java.util.LinkedList;
```

```
public class Main {
    public static void main(String[] args) {
        Deque<String> deque = new LinkedList<>();
        deque.offerLast("A"); // A
        deque.offerLast("B"); // B -> A
        deque.offerFirst("C"); // B -> A -> C
        System.out.println(deque.pollFirst()); // C, 剩下B -> A
        System.out.println(deque.pollLast()); // B
        System.out.println(deque.pollFirst()); // A
        System.out.println(deque.pollFirst()); // null
    }
}
```

▶ Run

如果直接写 `deque.offer()` ，我们就需要思考， `offer()` 实际上是 `offerLast()` ，我们明确地写上 `offerLast()` ，不需要思考就能一眼看出这是添加到队尾。

因此，使用 `Deque` ，推荐总是明确调用 `offerLast()` / `offerFirst()` 或者 `pollFirst()` / `pollLast()` 方法。

`Deque` 是一个接口，它的实现类有 `ArrayDeque` 和 `LinkedList` 。

我们发现 `LinkedList` 真是一个全能选手，它即是 `List` ，又是 `Queue` ，还是 `Deque` 。但是我们在使用的时候，总是用特定的接口来引用它，这是因为持有接口说明代码的抽象层次更高，而且接口本身定义的方法代表了特定的用途。

```
1. // 不推荐的写法:
2. LinkedList<String> d1 = new LinkedList<>();
3. d1.offerLast("z");
4. // 推荐的写法：
5. Deque<String> d2 = new LinkedList<>();
6. d2.offerLast("z");
```

可见面向抽象编程的一个原则就是：尽量持有接口，而不是具体的实现类。

练习

小结

`Deque` 实现了一个双端队列（Double Ended Queue），它可以：

- 将元素添加到队尾或队首： `addLast()` / `offerLast()` / `addFirst()` / `offerFirst()` ；

- 从队首/队尾获取元素并删除：`removeFirst()` / `pollFirst()` / `removeLast()` / `pollLast()`；
- 从队首/队尾获取元素但不删除：`getFirst()` / `peekFirst()` / `getLast()` / `peekLast()`；
- 总是调用`xxxFirst()` / `xxxLast()`以便与`Queue`的方法区分开；
- 避免把`null`添加到队列。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Stack

栈（Stack）是一种后进先出（LIFO: Last In First Out）的数据结构。

什么是LIFO呢？我们先回顾一下 `Queue` 的特点FIFO:

```
1.           ————————————————————————
2.    (\(\      (\(\    (\(\    (\(\      (\(\
3.   (='.') -> (='.')  (='.')  (='.') -> (='.')
4.  O(_")")   O(_")") O(_")") O(_")")   O(_")")
5.           ————————————————————————
```

所谓FIFO，是最先进队列的元素一定最早出队列，而LIFO是最后进 `Stack` 的元素一定最早出 `Stack` 。如何做到这一点呢？只需要把队列的一端封死:

```
1.           ————————————————————————————————┐
2.    (\(\       (\(\    (\(\    (\(\    (\(\ |
3.   (='.') <-> (='.')  (='.')  (='.')  (='.')|
4.  O(_")")    O(_")") O(_")") O(_")") O(_")")|
5.           ————————————————————————————————┘
```

因此， `Stack` 是这样一种数据结构：只能不断地往 `Stack` 中压入（push）元素，最后进去的必须最早弹出（pop）来:



`Stack` 只有入栈和出栈的操作:

- 把元素压栈: `push(E)` ;
- 把栈顶的元素“弹出”: `pop(E)` ;
- 取栈顶元素但不弹出: `peek(E)` 。

在Java中，我们用 `Deque` 可以实现 `Stack` 的功能:

- 把元素压栈：`push(E)` / `addFirst(E)`；
- 把栈顶的元素“弹出”：`pop(E)` / `removeFirst()`；
- 取栈顶元素但不弹出：`peek(E)` / `peekFirst()`。

为什么Java的集合类没有单独的 `Stack` 接口呢？因为有个遗留类名字就叫 `Stack`，出于兼容性考虑，所以没办法创建 `Stack` 接口，只能用 `Deque` 接口来“模拟”一个 `Stack` 了。

当我们把 `Deque` 作为 `Stack` 使用时，注意只调用 `push()` / `pop()` / `peek()` 方法，不要调用 `addFirst()` / `removeFirst()` / `peekFirst()` 方法，这样代码更加清晰。

Stack的作用

Stack在计算机中使用非常广泛，JVM在处理Java方法调用的时候就会通过栈这种数据结构维护方法调用的层次。例如：

```
1.  static void main(String[] args) {
2.      foo(123);
3.  }
4.
5.  static String foo(x) {
6.      return "F-" + bar(x + 1);
7.  }
8.
9.  static int bar(int x) {
10.     return x << 2;
11. }
```

JVM会创建方法调用栈，每调用一个方法时，先将参数压栈，然后执行对应的方法；当方法返回时，返回值压栈，调用方法通过出栈操作获得方法返回值。

因为方法调用栈有容量限制，嵌套调用过多会造成栈溢出，即引发 `StackOverflowError`：

```
// 测试无限递归调用
public class Main {
    public static void main(String[] args) {
        increase(1);
    }

    static int increase(int x) {
        return increase(x) + 1;
    }
}
```

▶ Run

我们再来看一个 `Stack` 的用途：对整数进行进制的转换就可以利用栈。

例如，我们要把一个 `int` 整数 `12500` 转换为十六进制表示的字符串，如何实现这个功能？

首先我们准备一个空栈：

```
1. |   |
2. |   |
3. |   |
4. |   |
5. |   |
6. |   |
7. |   |
8. |   |
9. └───┘
```

然后计算12500÷16=781…4，余数是 `4` ，把余数 `4` 压栈：

```
1. |   |
2. |   |
3. |   |
4. |   |
5. |   |
6. |   |
7. |   |
8. | 4 |
9. └───┘
```

然后计算781÷16=48…13，余数是 `13` ， `13` 的十六进制用字母 `D` 表示，把余数 `D` 压栈：

```
1. |   |
2. |   |
3. |   |
4. |   |
5. |   |
6. | D |
7. |   |
8. | 4 |
9. └───┘
```

然后计算48÷16=3…0，余数是 `0` ，把余数 `0` 压栈：

```
|   |
|   |
|   |
| 0 |
|   |
| D |
|   |
| 4 |
└───┘
```

最后计算3÷16=0…3，余数是 `3` ，把余数 `3` 压栈：

```
|   |
| 3 |
|   |
| 0 |
|   |
| D |
|   |
| 4 |
└───┘
```

当商是 `0` 的时候，计算结束，我们把栈的所有元素依次弹出，组成字符串 `30D4` ，这就是十进制整数 `12500` 的十六进制表示的字符串。

计算中缀表达式

在编写程序的时候，我们使用的带括号的数学表达式实际上是中缀表达式，即运算符在中间，例如： `1 + 2 * (9 - 5)` 。

但是计算机执行表达式的时候，它并不能直接计算中缀表达式，而是通过编译器把中缀表达式转换为后缀表达式，例如： `1 2 9 5 - * +` 。

这个编译过程就会用到栈。我们先跳过编译这一步（涉及运算优先级，代码比较复杂），看看如何通过栈计算后缀表达式。

计算后缀表达式不考虑优先级，直接从左到右依次计算，因此计算起来简单。首先准备一个空的栈：

```
|   |
|   |
|   |
|   |
```

```
|   |
|   |
|   |
|   |
└───┘
```

然后我们依次扫描后缀表达式 `1 2 9 5 - * +` ，遇到数字 `1` ，就直接扔到栈里：

```
|   |
|   |
|   |
|   |
|   |
|   |
|   |
| 1 |
└───┘
```

紧接着，遇到数字 `2` ， `9` ， `5` ，也扔到栈里：

```
|   |
| 5 |
|   |
| 9 |
|   |
| 2 |
|   |
| 1 |
└───┘
```

接下来遇到减号时，弹出栈顶的两个元素，并计算 `9-5=4` ，把结果 `4` 压栈：

```
|   |
|   |
|   |
| 4 |
|   |
| 2 |
|   |
| 1 |
└───┘
```

接下来遇到 `` 号时，弹出栈顶的两个元素，并计算 `2 4=8` ，把结果 `8` 压栈：

```
1.  |    |
2.  |    |
3.  |    |
4.  |    |
5.  |    |
6.  | 8  |
7.  |    |
8.  | 1  |
9.  └────┘
```

接下来遇到 `+` 号时，弹出栈顶的两个元素，并计算 `1+8=9` ，把结果 `9` 压栈：

```
1.  |    |
2.  |    |
3.  |    |
4.  |    |
5.  |    |
6.  |    |
7.  |    |
8.  | 9  |
9.  └────┘
```

扫描结束后，没有更多的计算了，弹出栈的唯一一个元素，得到计算结果 `9` 。

练习

请利用Stack把一个给定的整数转换为十六进制：

```
// 转十六进制
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String hex = toHex(12500);
        if (hex.equalsIgnoreCase("30D4")) {
            System.out.println("测试通过");
        } else {
            System.out.println("测试失败");
        }
    }

    static String toHex(int n) {
        return "";
    }
}
```

▶ Run

进阶练习：

请利用Stack把字符串中缀表达式编译为后缀表达式，然后再利用栈执行后缀表达式获得计算结果：

```
// 高难度练习，慎重选择！
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String exp = "1 + 2 * (9 - 5)";
        SuffixExpression se = compile(exp);
        int result = se.execute();
        System.out.println(exp + " = " + result + " " + (result == 1 + 2 * (9 - 5) ? "✓" : "✗"));
    }

    static SuffixExpression compile(String exp) {
        // TODO:
        return new SuffixExpression();
    }
}

class SuffixExpression {
    int execute() {
        // TODO:
        return 0;
    }
}
```

▶ Run

进阶练习2：

请把带变量的中缀表达式编译为后缀表达式，执行后缀表达式时，传入变量的值并获得计算结果：

```
// 超高难度练习，慎重选择！
import java.util.*;

public class Main {
    public static void main(String[] args) {
        String exp = "x + 2 * (y - 5)";
        SuffixExpression se = compile(exp);
        Map<String, Integer> env = Map.of("x", 1, "y", 9);
        int result = se.execute(env);
        System.out.println(exp + " = " + result + " " + (result == 1 + 2 * (9 - 5) ? "✓" : "✗"));
    }

    static SuffixExpression compile(String exp) {
        // TODO:
        return new SuffixExpression();
    }
}

class SuffixExpression {
    int execute(Map<String, Integer> env) {
        // TODO:
        return 0;
    }
}
```

▶ Run

下载练习：Stack练习 （推荐使用IDE练习插件快速下载）

小结

栈（Stack）是一种后进先出（LIFO）的数据结构，操作栈的元素的方法有：

- 把元素压栈：`push(E)`；
- 把栈顶的元素“弹出”：`pop(E)`；
- 取栈顶元素但不弹出：`peek(E)`。

在Java中，我们用`Deque`可以实现`Stack`的功能，注意只调用`push()`/`pop()`/`peek()`方法，避免调用`Deque`的其他方法。

最后，不要使用遗留类`Stack`。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Iterator

Java的集合类都可以使用 `for each` 循环， `List` 、 `Set` 和 `Queue` 会迭代每个元素， `Map` 会迭代每个key。以 `List` 为例：

```
List<String> list = List.of("Apple", "Orange", "Pear");
for (String s : list) {
    System.out.println(s);
}
```

实际上，Java编译器并不知道如何遍历 `List` 。上述代码能够编译通过，只是因为编译器把 `for each` 循环通过 `Iterator` 改写为了普通的 `for` 循环：

```
for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
     String s = it.next();
     System.out.println(s);
}
```

我们把这种通过 `Iterator` 对象遍历集合的模式称为迭代器。

使用迭代器的好处在于，调用方总是以统一的方式遍历各种集合类型，而不必关系它们内部的存储结构。

例如，我们虽然知道 `ArrayList` 在内部是以数组形式存储元素，并且，它还提供了 `get(int)` 方法。虽然我们可以用 `for` 循环遍历：

```
for (int i=0; i<list.size(); i++) {
    Object value = list.get(i);
}
```

但是这样一来，调用方就必须知道集合的内部存储结构。并且，如果把 `ArrayList` 换成 `LinkedList` ， `get(int)` 方法耗时会随着index的增加而增加。如果把 `ArrayList` 换成 `Set` ，上述代码就无法编译，因为 `Set` 内部没有索引。

用 `Iterator` 遍历就没有上述问题，因为 `Iterator` 对象是集合对象自己在内部创建的，它自己知道如何高效遍历内部的数据集合，调用方则获得了统一的代码，编译器才能把标准的 `for each` 循环自动转换为 `Iterator` 遍历。

如果我们自己编写了一个集合类，想要使用 `for each` 循环，只需满足以下条件：

- 集合类实现 `Iterable` 接口，该接口要求返回一个 `Iterator` 对象；
- 用 `Iterator` 对象迭代集合内部数据。

这里的关键在于，集合类通过调用 `iterator()` 方法，返回一个 `Iterator` 对象，这个对象必须自己知道如何遍历该集合。

一个简单的 `Iterator` 示例如下，它总是以倒序遍历集合：

```
// Iterator
import java.util.*;

public class Main {
    public static void main(String[] args) {
        ReverseList<String> rlist = new ReverseList<>();
        rlist.add("Apple");
        rlist.add("Orange");
        rlist.add("Pear");
        for (String s : rlist) {
            System.out.println(s);
        }
    }
}

class ReverseList<T> implements Iterable<T> {

    private List<T> list = new ArrayList<>();

    public void add(T t) {
        list.add(t);
    }

    @Override
    public Iterator<T> iterator() {
        return new ReverseIterator(list.size());
    }

    class ReverseIterator implements Iterator<T> {
        int index;

        ReverseIterator(int index) {
            this.index = index;
        }

        @Override
        public boolean hasNext() {
            return index > 0;
        }

        @Override
        public T next() {
            index--;
            return ReverseList.this.list.get(index);
        }
    }
}
```

▶ Run

虽然 `ReverseList` 和 `ReverseIterator` 的实现类稍微比较复杂，但是，注意到这是底层集合库，

只需编写一次。而调用方则完全按 `for each` 循环编写代码，根本不需要知道集合内部的存储逻辑和遍历逻辑。

在编写 `Iterator` 的时候，我们通常可以用一个内部类来实现 `Iterator` 接口，这个内部类可以直接访问对应的外部类的所有字段和方法。例如，上述代码中，内部类 `ReverseIterator` 可以用 `ReverseList.this` 获得当前外部类的 `this` 引用，然后，通过这个 `this` 引用就可以访问 `ReverseList` 的所有字段和方法。

小结

`Iterator` 是一种抽象的数据访问模型。使用 `Iterator` 模式进行迭代的好处有：

- 对任何集合都采用同一种访问模型；
- 调用者对集合内部结构一无所知；
- 集合类返回的 `Iterator` 对象知道如何迭代。

Java提供了标准的迭代器模型，即集合类实现 `java.util.Iterable` 接口，返回 `java.util.Iterator` 实例。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Collections

`Collections`是JDK提供的工具类，同样位于`java.util`包中。它提供了一系列静态方法，能更方便地操作各种集合。

注意Collections结尾多了一个s，不是Collection！

我们一般看方法名和参数就可以确认`Collections`提供的该方法的功能。例如，对于以下静态方法：

```
1. public static boolean addAll(Collection<? super T> c, T... elements) { ... }
```

`addAll()`方法可以给一个`Collection`类型的集合添加若干元素。因为方法签名是`Collection`，所以我们可以传入`List`，`Set`等各种集合类型。

创建空集合

`Collections`提供了一系列方法来创建空集合：

- 创建空List：`List<T> emptyList()`
- 创建空Map：`Map<K, V> emptyMap()`
- 创建空Set：`Set<T> emptySet()`

要注意到返回的空集合是不可变集合，无法向其中添加或删除元素。

此外，也可以用各个集合接口提供的`of(T…)`方法创建空集合。例如，以下创建空`List`的两个方法是等价的：

```
1. List<String> list1 = List.of();
2. List<String> list2 = Collections.emptyList();
```

创建单元素集合

`Collections`提供了一系列方法来创建一个单元素集合：

- 创建一个元素的List：`List<T> singletonList(T o)`
- 创建一个元素的Map：`Map<K, V> singletonMap(K key, V value)`
- 创建一个元素的Set：`Set<T> singleton(T o)`

要注意到返回的单元素集合也是不可变集合，无法向其中添加或删除元素。

此外，也可以用各个集合接口提供的`of(T…)`方法创建单元素集合。例如，以下创建单元素`List`的两个方法是等价的：

```
List<String> list1 = List.of("apple");
List<String> list2 = Collections.singleton("apple");
```

实际上，使用 `List.of(T…)` 更方便，因为它既可以创建空集合，也可以创建单元素集合，还可以创建任意个元素的集合：

```
List<String> list1 = List.of(); // empty list
List<String> list2 = List.of("apple"); // 1 element
List<String> list3 = List.of("apple", "pear"); // 2 elements
List<String> list4 = List.of("apple", "pear", "orange"); // 3 elements
```

排序

`Collections` 可以对 `List` 进行排序。因为排序会直接修改 `List` 元素的位置，因此必须传入可变 `List` ：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("apple");
        list.add("pear");
        list.add("orange");
        // 排序前:
        System.out.println(list);
        Collections.sort(list);
        // 排序后:
        System.out.println(list);
    }
}
```

▶ Run

洗牌

`Collections` 提供了洗牌算法，即传入一个有序的 `List` ，可以随机打乱 `List` 内部元素的顺序，效果相当于让计算机洗牌：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<Integer> list = new ArrayList<>();
        for (int i=0; i<10; i++) {
            list.add(i);
        }
        // 洗牌前:
        System.out.println(list);
        Collections.shuffle(list);
        // 洗牌后:
        System.out.println(list);
    }
}
```

▶ Run

不可变集合

`Collections` 还提供了一组方法把可变集合封装成不可变集合：

- 封装成不可变List: `List<T> unmodifiableList(List<? extends T> list)`
- 封装成不可变Set: `Set<T> unmodifiableSet(Set<? extends T> set)`
- 封装成不可变Map: `Map<K, V> unmodifiableMap(Map<? extends K, ? extends V> m)`

这种封装实际上是通过创建一个代理对象，拦截掉所有修改方法实现的。我们来看看效果：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable = Collections.unmodifiableList(mutable);
        immutable.add("orange"); // UnsupportedOperationException!
    }
}
```

▶ Run

然而，继续对原始的可变 `List` 进行增删是可以的，并且，会直接影响到封装后的“不可变” `List` ：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable = Collections.unmodifiableList(mutable);
        mutable.add("orange");
        System.out.println(immutable);
    }
}
```

▶ Run

因此，如果我们希望把一个可变 `List` 封装成不可变 `List` ，那么，返回不可变 `List` 后，最好立刻扔掉可变 `List` 的引用，这样可以保证后续操作不会意外改变原始对象，从而造成“不可变” `List` 变化了：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> mutable = new ArrayList<>();
        mutable.add("apple");
        mutable.add("pear");
        // 变为不可变集合:
        List<String> immutable = Collections.unmodifiableList(mutable);
        // 立刻扔掉mutable的引用:
        mutable = null;
        System.out.println(immutable);
    }
}
```

▶ Run

线程安全集合

`Collections` 还提供了一组方法，可以把线程不安全的集合变为线程安全的集合：

- 变为线程安全的List: `List<T> synchronizedList(List<T> list)`
- 变为线程安全的Set: `Set<T> synchronizedSet(Set<T> s)`
- 变为线程安全的Map: `Map<K,V> synchronizedMap(Map<K,V> m)`

多线程的概念我们会在后面讲。因为从Java 5开始，引入了更高效的并发集合类，所以上述这几个同步方法已经没有什么用了。

小结

`Collections` 类提供了一组工具方法来方便使用集合类：

- 创建空集合；
- 创建单元素集合；
- 创建不可变集合；
- 排序 / 洗牌等操作。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





IO

IO是指Input/Output，即输入和输出。以内存为中心：

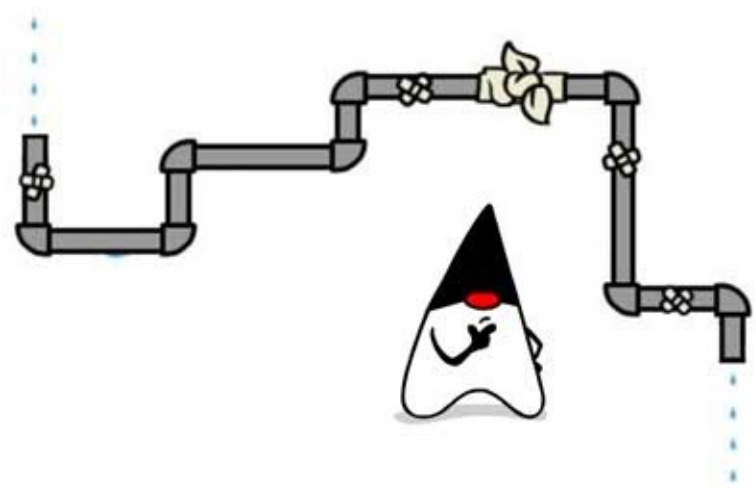
- Input指从外部读入数据到内存，例如，把文件从磁盘读取到内存，从网络读取数据到内存等等。
- Output指把数据从内存输出到外部，例如，把数据从内存写入到文件，把数据从内存输出到网络等等。

为什么要把数据读到内存才能处理这些数据？因为代码是在内存中运行的，数据也必须读到内存，最终的表示方式无非是byte数组，字符串等，都必须存放在内存里。

从Java代码来看，输入实际上就是从外部，例如，硬盘上的某个文件，把内容读到内存，并且以Java提供的某种数据类型表示，例如，`byte[]`，`String`，这样，后续代码才能处理这些数据。

因为内存有“易失性”的特点，所以必须把处理后的数据以某种方式输出，例如，写入到文件。Output实际上就是把Java表示的数据格式，例如，`byte[]`，`String`等输出到某个地方。

IO流是一种顺序读写数据的模式，它的特点是单向流动。数据类似自来水一样在水管中流动，所以我们把它称为IO流。



InputStream / OutputStream

IO流以`byte`（字节）为最小单位，因此也称为字节流。例如，我们要从磁盘读入一个文件，包含6个字符，就相当于读入了6个字节的数据：

```
┌──────────┐
│  Memory  │
└──────────┘
      ▲
```

```
        |0x48
        |0x65
        |0x6c
        |0x6c
        |0x6f
        |0x21
  ┌─────────────┐
  │ Hard Disk   │
  └─────────────┘
```

这6个字节是按顺序读入的，所以是输入字节流。

反过来，我们把6个字节从内存写入磁盘文件，就是输出字节流：

```
  ┌─────────────┐
  │   Memory    │
  └─────────────┘
        |0x21
        |0x6f
        |0x6c
        |0x6c
        |0x65
        |0x48
        ▼
  ┌─────────────┐
  │ Hard Disk   │
  └─────────────┘
```

在Java中，`InputStream`代表输入字节流，`OuputStream`代表输出字节流，这是最基本的两种IO流。

Reader / Writer

如果我们需要读写的是字符，并且字符不全是单字节表示的ASCII字符，那么，按照`char`来读写显然更方便，这种流称为字符流。

Java提供了`Reader`和`Writer`表示字符流，字符流传输的最小数据单位是`char`。

例如，我们把`char[]`数组`Hi你好`这4个字符用`Writer`字符流写入文件，并且使用UTF-8编码，得到的最终文件内容是8个字节，英文字符`H`和`i`各占一个字节，中文字符`你好`各占3个字节：

```
0x48
0x69
0xe4bda0
0xe5a5bd
```

反过来，我们用 `Reader` 读取以UTF-8编码的这8个字节，会从 `Reader` 中得到 `Hi你好` 这4个字符。

因此， `Reader` 和 `Writer` 本质上是一个能自动编解码的 `InputStream` 和 `OutputStream` 。

使用 `Reader` ，数据源虽然是字节，但我们读入的数据都是 `char` 类型的字符，原因是 `Reader` 内部把读入的 `byte` 做了编码，转换成了 `char` 。使用 `InputStream` ，我们读入的数据和原始二进制数据一模一样，是 `byte[]` 数组，但是我们可以自己把二进制 `byte[]` 数组按照某种编码转换为字符串。究竟使用 `Reader` 还是 `InputStream` ，要取决于具体的使用场景。如果数据源不是文本，就只能使用 `InputStream` ，如果数据源是文本，使用Reader更方便一些。 `Writer` 和 `OutputStream` 是类似的。

同步和异步

同步IO是指，读写IO时代码必须等待数据返回后才继续执行后续代码，它的优点是代码编写简单，缺点是CPU执行效率低。

而异步IO是指，读写IO时仅发出请求，然后立刻执行后续代码，它的优点是CPU执行效率高，缺点是代码编写复杂。

Java标准库的包 `java.io` 提供了同步IO，而 `java.nio` 则是异步IO。上面我们讨论的 `InputStream` 、 `OutputStream` 、 `Reader` 和 `Writer` 都是同步IO的抽象类，对应的具体实现类，以文件为例，有 `FileInputStream` 、 `FileOutputStream` 、 `FileReader` 和 `FileWriter` 。

本节我们只讨论Java的同步IO，即输入/输出流的IO模型。

小结

IO流是一种流式的数据输入/输出模型：

- 二进制数据以 `byte` 为最小单位在 `InputStream` / `OutputStream` 中单向流动；
- 字符数据以 `char` 为最小单位在 `Reader` / `Writer` 中单向流动。

Java标准库的 `java.io` 包提供了同步IO功能：

- 字节流接口： `InputStream` / `OutputStream` ；

- 字符流接口： `Reader` / `Writer` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





File对象

在计算机系统中，文件是非常重要的存储方式。Java的标准库 `java.io` 提供了 `File` 对象来操作文件和目录。

要构造一个 `File` 对象，需要传入文件路径：

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
        File f = new File("C:\\Windows\\notepad.exe");
        System.out.println(f);
    }
}
```

▶ Run

构造File对象时，既可以传入绝对路径，也可以传入相对路径。绝对路径是以根目录开头的完整路径，例如：

```
1. File f = new File("C:\\Windows\\notepad.exe");
```

注意Windows平台使用 `\` 作为路径分隔符，在Java字符串中需要用 `\` 表示一个 `\` 。Linux平台使用 `/` 作为路径分隔符：

```
1. File f = new File("/usr/bin/javac");
```

传入相对路径时，相对路径前面加上当前目录就是绝对路径：

```
1. // 假设当前目录是C:\Docs
2. File f1 = new File("sub\\javac"); // 绝对路径是C:\Docs\sub\javac
3. File f3 = new File(".\\sub\\javac"); // 绝对路径是C:\Docs\sub\javac
4. File f3 = new File("..\\sub\\javac"); // 绝对路径是C:\sub\javac
```

可以用 `.` 表示当前目录， `..` 表示上级目录。

File对象有3种形式表示的路径，一种是 `getPath()` ，返回构造方法传入的路径，一种是 `getAbsolutePath()` ，返回绝对路径，一种是 `getCanonicalPath` ，它和绝对路径类似，但是返回的是规范路径。

什么是规范路径？我们看以下代码：

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        File f = new File("..");
        System.out.println(f.getPath());
        System.out.println(f.getAbsolutePath());
        System.out.println(f.getCanonicalPath());
    }
}
```

▶ Run

绝对路径可以表示成 `C:\Windows\System32..\notepad.exe` ，而规范路径就是把 `.` 和 `..` 转换成标准的绝对路径后的路径： `C:\Windows\notepad.exe` 。

因为Windows和Linux的路径分隔符不同，File对象有一个静态变量用于表示当前平台的系统分隔符：

```
1.  System.out.println(File.separator); // 根据当前平台打印"\"或"/"
```

文件和目录

`File` 对象既可以表示文件，也可以表示目录。特别要注意的是，构造一个 `File` 对象，即使传入的文件或目录不存在，代码也不会出错，因为构造一个 `File` 对象，并不会导致任何磁盘操作。只有当我们调用 `File` 对象的某些方法的时候，才真正进行磁盘操作。

例如，调用 `isFile()` ，判断该 `File` 对象是否是一个已存在的文件，调用 `isDirectory()` ，判断该 `File` 对象是否是一个已存在的目录：

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        File f1 = new File("C:\\Windows");
        File f2 = new File("C:\\Windows\\notepad.exe");
        File f3 = new File("C:\\Windows\\nothing");
        System.out.println(f1.isFile());
        System.out.println(f1.isDirectory());
        System.out.println(f2.isFile());
        System.out.println(f2.isDirectory());
        System.out.println(f3.isFile());
        System.out.println(f3.isDirectory());
    }
}
```

▶ Run

用 `File` 对象获取到一个文件时，还可以进一步判断文件的权限和大小：

- `boolean canRead()`：是否可读；
- `boolean canWrite()`：是否可写；
- `boolean canExecute()`：是否可执行；
- `long length()`：文件字节大小。

对目录而言，是否可执行表示能否列出它包含的文件和子目录。

创建和删除文件

当File对象表示一个文件时，可以通过 `createNewFile()` 创建一个新文件，用 `delete()` 删除该文件：

```
File file = new File("/path/to/file");
if (file.createNewFile()) {
    // 文件创建成功:
    // TODO:
    if (file.delete()) {
        // 删除文件成功:
    }
}
```

有些时候，程序需要读写一些临时文件，File对象提供了 `createTempFile()` 来创建一个临时文件，以及 `deleteOnExit()` 在JVM退出时自动删除该文件。

```
import java.io.*;

public class Main {
    public static void main(String[] args) throws IOException {
        File f = File.createTempFile("tmp-", ".txt"); // 提供临时文件的前缀和后缀
        f.deleteOnExit(); // JVM退出时自动删除
        System.out.println(f.isFile());
        System.out.println(f.getAbsolutePath());
    }
}
```

▶ Run

遍历文件和目录

当File对象表示一个目录时，可以使用 `list()` 和 `listFiles()` 列出目录下的文件和子目录名。 `listFiles()` 提供了一系列重载方法，可以过滤不想要的文件和目录：

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        File f = new File("C:\\Windows");
        File[] fs1 = f.listFiles(); // 列出所有文件和子目录
        printFiles(fs1);
        File[] fs2 = f.listFiles(new FilenameFilter() { // 仅列出.exe文件
            public boolean accept(File dir, String name) {
                return name.endsWith(".exe"); // 返回true表示接受该文件
            }
        });
        printFiles(fs2);
    }

    static void printFiles(File[] files) {
        System.out.println("==========");
        if (files != null) {
            for (File f : files) {
                System.out.println(f);
            }
        }
        System.out.println("==========");
    }
}
```

▶ Run

和文件操作类似，File对象如果表示一个目录，可以通过以下方法创建和删除目录：

- `boolean mkdir()`：创建当前File对象表示的目录；
- `boolean mkdirs()`：创建当前File对象表示的目录，并在必要时将不存在的父目录也创建出来；
- `boolean delete()`：删除当前File对象表示的目录，当前目录必须为空才能删除成功。

Path

Java标准库还提供了一个`Path`对象，它位于`java.nio.file`包。`Path`对象和`File`对象类似，但操作更加简单：

```
import java.io.*;
import java.nio.file.*;
```

```
public class Main {
    public static void main(String[] args) throws IOException {
        Path p1 = Paths.get(".", "project", "study"); // 构造一个Path对象
        System.out.println(p1);
        Path p2 = p1.toAbsolutePath(); // 转换为绝对路径
        System.out.println(p2);
        Path p3 = p2.normalize(); // 转换为规范路径
        System.out.println(p3);
        File f = p3.toFile(); // 转换为File对象
        System.out.println(f);
        for (Path p : Paths.get("..").toAbsolutePath()) { // 可以直接遍历Path
            System.out.println("  " + p);
        }
    }
}
```

▶ Run

如果需要对目录进行复杂的拼接、遍历等操作，使用 `Path` 对象更方便。

练习

请利用 `File` 对象列出指定目录下的所有子目录和文件，并按层次打印。

例如，输出：

```
Documents/
  word/
    1.docx
    2.docx
    work/
      abc.doc
  ppt/
  other/
```

如果不指定参数，则使用当前目录，如果指定参数，则使用指定目录。

下载练习：`io-file` （推荐使用IDE练习插件快速下载）

小结

Java标准库的 `java.io.File` 对象表示一个文件或者目录：

- 创建 `File` 对象本身不涉及IO操作；

- 可以获取路径 / 绝对路径 / 规范路径：`getPath()` / `getAbsolutePath()` / `getCanonicalPath()`；
- 可以获取目录的文件和子目录：`list()` / `listFiles()`；
- 可以创建或删除文件和目录。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





InputStream

`InputStream` 就是Java标准库提供的最基本的输入流。它位于 `java.io` 这个包里。 `java.io` 包提供了所有同步IO的功能。

要特别注意的一点是， `InputStream` 并不是一个接口，而是一个抽象类，它是所有输入流的超类。这个抽象类定义的一个最重要的方法就是 `int read()` ，签名如下：

```
public abstract int read() throws IOException;
```

这个方法会读取输入流的下一个字节，并返回字节表示的 `int` 值（0~255）。如果已读到末尾，返回 `-1` 表示不能继续读取了。

`FileInputStream` 是 `InputStream` 的一个子类。顾名思义， `FileInputStream` 就是从文件流中读取数据。下面的代码演示了如何完整地读取一个 `FileInputStream` 的所有字节：

```
public void readFile() throws IOException {
    // 创建一个FileInputStream对象:
    InputStream input = new FileInputStream("src/readme.txt");
    for (;;) {
        int n = input.read(); // 反复调用read()方法，直到返回-1
        if (n == -1) {
            break;
        }
        System.out.println(n); // 打印byte的值
    }
    input.close(); // 关闭流
}
```

在计算机中，类似文件、网络端口这些资源，都是由操作系统统一管理的。应用程序在运行的过程中，如果打开了一个文件进行读写，完成后要及时地关闭，以便让操作系统把资源释放掉，否则，应用程序占用的资源会越来越多，不但白白占用内存，还会影响其他应用程序的运行。

`InputStream` 和 `OutputStream` 都是通过 `close()` 方法来关闭流。关闭流就会释放对应的底层资源。

我们还要注意到在读取或写入IO流的过程中，可能会发生错误，例如，文件不存在导致无法读取，没有写权限导致写入失败，等等，这些底层错误由Java虚拟机自动封装成 `IOException` 异常并抛出。因此，所有与IO操作相关的代码都必须正确处理 `IOException` 。

仔细观察上面的代码，会发现一个潜在的问题：如果读取过程中发生了IO错误， `InputStream` 就没法正确地关闭，资源也就没法及时释放。

因此，我们需要用 `try … finally` 来保证 `InputStream` 在无论是否发生IO错误的时候都能够正确地关闭：

```
public void readFile() throws IOException {
    InputStream input = null;
    try {
        input = new FileInputStream("src/readme.txt");
        int n;
        while ((n = input.read()) != -1) { // 利用while同时读取并判断
            System.out.println(n);
        }
    } finally {
        if (input != null) { input.close(); }
    }
}
```

用 `try … finally` 来编写上述代码会感觉比较复杂，更好的写法是利用Java 7引入的新的 `try(resource)` 的语法，只需要编写 `try` 语句，让编译器自动为我们关闭资源。推荐的写法如下：

```
public void readFile() throws IOException {
    try (InputStream input = new FileInputStream("src/readme.txt")) {
        int n;
        while ((n = input.read()) != -1) {
            System.out.println(n);
        }
    } // 编译器在此自动为我们写入finally并调用close()
}
```

实际上，编译器并不会特别地为 `InputStream` 加上自动关闭。编译器只看 `try(resource = …)` 中的对象是否实现了 `java.lang.AutoCloseable` 接口，如果实现了，就自动加上 `finally` 语句并调用 `close()` 方法。 `InputStream` 和 `OutputStream` 都实现了这个接口，因此，都可以用在 `try(resource)` 中。

缓冲

在读取流的时候，一次读取一个字节并不是最高效的方法。很多流支持一次性读取多个字节到缓冲区，对于文件和网络流来说，利用缓冲区一次性读取多个字节效率往往要高很多。 `InputStream` 提供了两个重载方法来支持读取多个字节：

- `int read(byte[] b)`：读取若干字节并填充到 `byte[]` 数组，返回读取的字节数

- `int read(byte[] b, int off, int len)`：指定`byte[]`数组的偏移量和最大填充数

利用上述方法一次读取多个字节时，需要先定义一个`byte[]`数组作为缓冲区，`read()`方法会尽可能多地读取字节到缓冲区，但不会超过缓冲区的大小。`read()`方法的返回值不再是字节的`int`值，而是返回实际读取了多少个字节。如果返回`-1`，表示没有更多的数据了。

利用缓冲区一次读取多个字节的代码如下：

```
public void readFile() throws IOException {
    try (InputStream input = new FileInputStream("src/readme.txt")) {
        // 定义1000个字节大小的缓冲区:
        byte[] buffer = new byte[1000];
        int n;
        while ((n = input.read(buffer)) != -1) { // 读取到缓冲区
            System.out.println("read " + n + " bytes.");
        }
    }
}
```

阻塞

在调用`InputStream`的`read()`方法读取数据时，我们说`read()`方法是阻塞（Blocking）的。它的意思是，对于下面的代码：

```
int n;
n = input.read(); // 必须等待read()方法返回才能执行下一行代码
int m = n;
```

执行到第二行代码时，必须等`read()`方法返回后才能继续。因为读取IO流相比执行普通代码，速度会慢很多，因此，无法确定`read()`方法调用到底要花费多长时间。

InputStream实现类

用`FileInputStream`可以从文件获取输入流，这是`InputStream`常用的一个实现类。此外，`ByteArrayInputStream`可以在内存中模拟一个`InputStream`：

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data = { 72, 101, 108, 108, 111, 33 };
        try (InputStream input = new ByteArrayInputStream(data)) {
            int n;
            while ((n = input.read()) != -1) {
                System.out.println((char)n);
            }
        }
    }
}
```

▶ Run

`ByteArrayInputStream` 实际上是把一个 `byte[]` 数组在内存中变成一个 `InputStream`，虽然实际应用不多，但测试的时候，可以用它来构造一个 `InputStream`。

举个栗子：我们想从文件中读取所有字节，并转换成 `char` 然后拼成一个字符串，可以这么写：

```
public class Main {
    public static void main(String[] args) throws IOException {
        String s;
        try (InputStream input = new FileInputStream("C:\\test\\README.txt")) {
            int n;
            StringBuilder sb = new StringBuilder();
            while ((n = input.read()) != -1) {
                sb.append((char) n);
            }
            s = sb.toString();
        }
        System.out.println(s);
    }
}
```

要测试上面的程序，就真的需要在本地硬盘上放一个真实的文本文件。如果我们把代码稍微改造一下，提取一个 `readAsString()` 的方法：

```
public class Main {
    public static void main(String[] args) throws IOException {
        String s;
        try (InputStream input = new FileInputStream("C:\\test\\README.txt")) {
            s = readAsString(input);
        }
```

```
            System.out.println(s);
        }

        public static String readAsString(InputStream input) throws IOException {
            int n;
            StringBuilder sb = new StringBuilder();
            while ((n = input.read()) != -1) {
                sb.append((char) n);
            }
            return sb.toString();
        }
    }
```

对这个 `String readAsString(InputStream input)` 方法进行测试就相当简单，因为不一定要传入一个真的 `FileInputStream`：

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data = { 72, 101, 108, 108, 111, 33 };
        try (InputStream input = new ByteArrayInputStream(data)) {
            String s = readAsString(input);
            System.out.println(s);
        }
    }

    public static String readAsString(InputStream input) throws IOException {
        int n;
        StringBuilder sb = new StringBuilder();
        while ((n = input.read()) != -1) {
            sb.append((char) n);
        }
        return sb.toString();
    }
}
```

▶ Run

这就是面向抽象编程原则的应用：接受 `InputStream` 抽象类型，而不是具体的 `FileInputStream` 类型，从而使得代码可以处理 `InputStream` 的任意实现类。

小结

Java标准库的 `java.io.InputStream` 定义了所有输入流的超类：

- `FileInputStream` 实现了文件流输入；

- `ByteArrayInputStream` 在内存中模拟一个字节流输入。

总是使用 `try(resource)` 来保证 `InputStream` 正确关闭。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





OutputStream

和 `InputStream` 相反， `OutputStream` 是Java标准库提供的最基本的输出流。

和 `InputStream` 类似， `OutputStream` 也是抽象类，它是所有输出流的超类。这个抽象类定义的一个最重要的方法就是 `void write(int b)` ，签名如下：

```
1.  public abstract void write(int b) throws IOException;
```

这个方法会写入一个字节到输出流。要注意的是，虽然传入的是 `int` 参数，但只会写入一个字节，即只写入 `int` 最低8位表示字节的部分（相当于 `b & 0xff` ）。

和 `InputStream` 类似， `OutputStream` 也提供了 `close()` 方法关闭输出流，以便释放系统资源。要特别注意： `OutputStream` 还提供了一个 `flush()` 方法，它的目的是将缓冲区的内容真正输出到目的地。

为什么要有 `flush()` ？因为向磁盘、网络写入数据的时候，出于效率的考虑，操作系统并不是输出一个字节就立刻写入到文件或者发送到网络，而是把输出的字节先放到内存的一个缓冲区里（本质上就是一个 `byte[]` 数组），等到缓冲区写满了，再一次性写入文件或者网络。对于很多IO设备来说，一次写一个字节和一次写1000个字节，花费的时间几乎是完全一样的，所以 `OutputStream` 有个 `flush()` 方法，能强制把缓冲区内容输出。

通常情况下，我们不需要调用这个 `flush()` 方法，因为缓冲区写满了 `OutputStream` 会自动调用它，并且，在调用 `close()` 方法关闭 `OutputStream` 之前，也会自动调用 `flush()` 方法。

但是，在某些情况下，我们必须手动调用 `flush()` 方法。举个栗子：

小明正在开发一款在线聊天软件，当用户输入一句话后，就通过 `OutputStream` 的 `write()` 方法写入网络流。小明测试的时候发现，发送方输入后，接收方根本收不到任何信息，怎么肥四？

原因就在于写入网络流是先写入内存缓冲区，等缓冲区满了才会一次性发送到网络。如果缓冲区大小是4K，则发送方要敲几千个字符后，操作系统才会把缓冲区的内容发送出去，这个时候，接收方会一次性收到大量消息。

解决办法就是每输入一句话后，立刻调用 `flush()` ，不管当前缓冲区是否已满，强迫操作系统把缓冲区的内容立刻发送出去。

实际上， `InputStream` 也有缓冲区。例如，从 `FileInputStream` 读取一个字节时，操作系统往往会一次性读取若干字节到缓冲区，并维护一个指针指向未读的缓冲区。然后，每次我们调用 `int read()` 读取下一个字节时，可以直接返回缓冲区的下一个字节，避免每次读一个字节都导致IO操作。当缓冲区全部读完后继续调用 `read()` ，则会触发操作系统的下一次读取并再次填满缓冲区。

FileOutputStream

我们以 `FileOutputStream` 为例，演示如何将若干个字节写入文件流：

```
public void writeFile() throws IOException {
    OutputStream output = new FileOutputStream("out/readme.txt");
    output.write(72); // H
    output.write(101); // e
    output.write(108); // l
    output.write(108); // l
    output.write(111); // o
    output.close();
}
```

每次写入一个字节非常麻烦，更常见的方法是一次性写入若干字节。这时，可以用 `OutputStream` 提供的重载方法 `void write(byte[])` 来实现：

```
public void writeFile() throws IOException {
    OutputStream output = new FileOutputStream("out/readme.txt");
    output.write("Hello".getBytes("UTF-8")); // Hello
    output.close();
}
```

和 `InputStream` 一样，上述代码没有考虑到在发生异常的情况下如何正确地关闭资源。写入过程也会经常发生IO错误，例如，磁盘已满，无权限写入等等。我们需要用 `try(resource)` 来保证 `OutputStream` 在无论是否发生IO错误的时候都能够正确地关闭：

```
public void writeFile() throws IOException {
    try (OutputStream output = new FileOutputStream("out/readme.txt")) {
        output.write("Hello".getBytes("UTF-8")); // Hello
    } // 编译器在此自动为我们写入finally并调用close()
}
```

阻塞

和 `InputStream` 一样， `OutputStream` 的 `write()` 方法也是阻塞的。

OutputStream实现类

用 `FileOutputStream` 可以从文件获取输出流，这是 `OutputStream` 常用的一个实现类。此外， `ByteArrayOutputStream` 可以在内存中模拟一个 `OutputStream` ：

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data;
        try (ByteArrayOutputStream output = new ByteArrayOutputStream()) {
            output.write("Hello ".getBytes("UTF-8"));
            output.write("world!".getBytes("UTF-8"));
            data = output.toByteArray();
        }
        System.out.println(new String(data, "UTF-8"));
    }
}
```

▶ Run

`ByteArrayOutputStream` 实际上是把一个 `byte[]` 数组在内存中变成一个 `OutputStream` ，虽然实际应用不多，但测试的时候，可以用它来构造一个 `OutputStream` 。

练习

请利用 `InputStream` 和 `OutputStream` ，编写一个复制文件的程序，它可以带参数运行：

```
1. java CopyFile.java source.txt copy.txt
```

下载练习：CopyFile练习 （推荐使用IDE练习插件快速下载）

小结

Java标准库的 `java.io.OutputStream` 定义了所有输出流的超类：

- `FileOutputStream` 实现了文件流输出；
- `ByteArrayOutputStream` 在内存中模拟一个字节流输出。

某些情况下需要手动调用 `OutputStream` 的 `flush()` 方法来强制输出缓冲区。

总是使用 `try(resource)` 来保证 `OutputStream` 正确关闭。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Filter模式

Java的IO标准库提供的 `InputStream` 根据来源可以包括：

- `FileInputStream`：从文件读取数据，是最终数据源；
- `ServletInputStream`：从HTTP请求读取数据，是最终数据源；
- `Socket.getInputStream()`：从TCP连接读取数据，是最终数据源；
- …

如果我们要给 `FileInputStream` 添加缓冲功能，则可以从 `FileInputStream` 派生一个类：

```
BufferedFileInputStream extends FileInputStream
```

如果要给 `FileInputStream` 添加计算签名的功能，类似的，也可以从 `FileInputStream` 派生一个类：

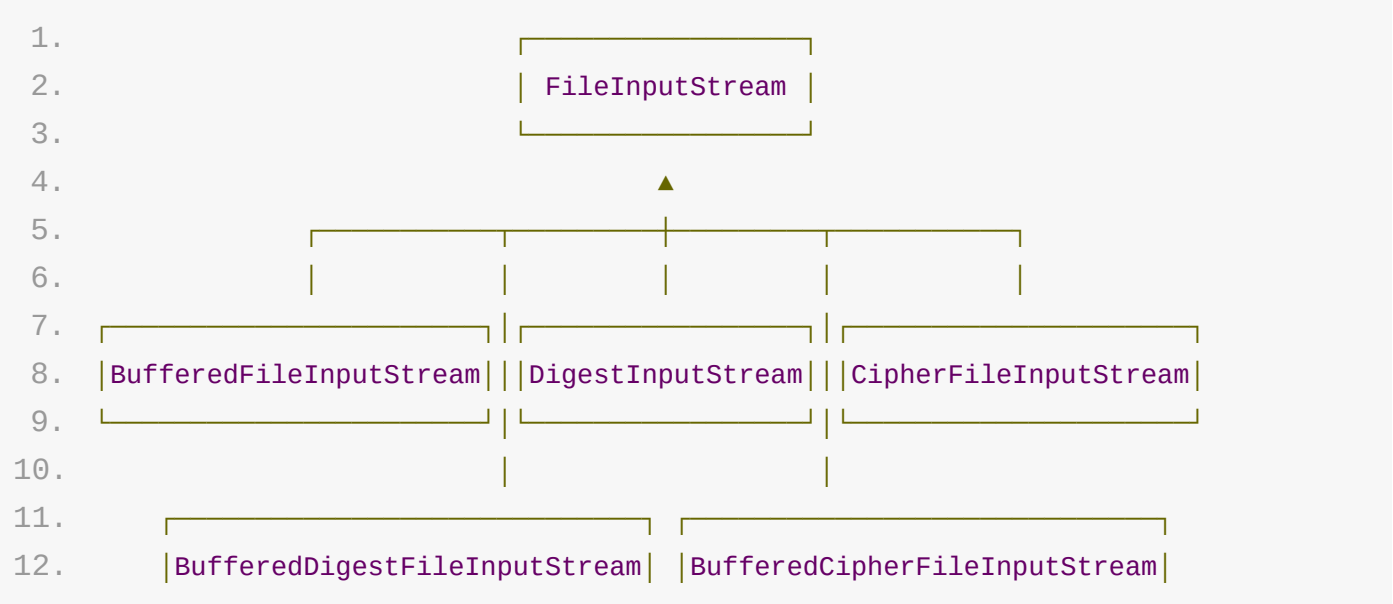
```
DigestFileInputStream extends FileInputStream
```

如果要给 `FileInputStream` 添加加密/解密功能，还是可以从 `FileInputStream` 派生一个类：

```
CipherFileInputStream extends FileInputStream
```

如果要给 `FileInputStream` 添加缓冲和签名的功能，那么我们还需要派生 `BufferedDigestFileInputStream`。如果要给 `FileInputStream` 添加缓冲和加解密的功能，则需要派生 `BufferedCipherFileInputStream`。

我们发现，给 `FileInputStream` 添加3种功能，至少需要3个子类。这3种功能的组合，又需要更多的子类：



```
13.                                                                    
```

这还只是针对 `FileInputStream` 设计，如果针对另一种 `InputStream` 设计，很快会出现子类爆炸的情况。

因此，直接使用继承，为各种 `InputStream` 附加更多的功能，根本无法控制代码的复杂度，很快就会失控。

为了解决依赖继承会导致子类数量失控的问题，JDK首先将 `InputStream` 分为两大类：

一类是直接提供数据的基础 `InputStream` ，例如：

- FileInputStream
- ByteArrayInputStream
- ServletInputStream
- …

一类是提供额外附加功能的 `InputStream` ，例如：

- BufferedInputStream
- DigestInputStream
- CipherInputStream
- …

当我们需要给一个“基础” `InputStream` 附加各种功能时，我们先确定这个能提供数据源的 `InputStream` ，因为我们需要的数据总得来自某个地方，例如， `FileInputStream` ，数据来源自文件：

```
1.  InputStream file = new FileInputStream("test.gz");
```

紧接着，我们希望 `FileInputStream` 能提供缓冲的功能来提高读取的效率，因此我们用 `BufferedInputStream` 包装这个 `InputStream` ，得到的包装类型是 `BufferedInputStream` ，但它仍然被视为一个 `InputStream` ：

```
1.  InputStream buffered = new BufferedInputStream(file);
```

最后，假设该文件已经用gzip压缩了，我们希望直接读取解压缩的内容，就可以再包装一个 `GZIPInputStream` ：

```
1.  InputStream gzip = new GZIPInputStream(buffered);
```

无论我们包装多少次，得到的对象始终是 `InputStream` ，我们直接用 `InputStream` 来引用它，就

可以正常读取：

```
┌──────────────────────────┐
│GZIPInputStream           │
│┌────────────────────────┐│
│||BufferedFileInputStream||
│||┌──────────────────────┐||
│|||   FileInputStream   |||
│||└──────────────────────┘||
│└────────────────────────┘│
└──────────────────────────┘
```

上述这种通过一个“基础”组件再叠加各种“附加”功能组件的模式，称之为Filter模式（或者装饰器模式：Decorator）。它可以让我们通过少量的类来实现各种功能的组合：

```
                ┌─────────────┐
                │ InputStream │
                └─────────────┘
                      ▲ ▲
┌─────────────────────┐ | | ┌─────────────────────┐
│  FileInputStream    |─┤ └─|FilterInputStream|
└─────────────────────┘ |   └─────────────────────┘
┌─────────────────────┐ |     ▲ ┌─────────────────────┐
|ByteArrayInputStream|─┤     ├─|BufferedInputStream|
└─────────────────────┘ |     | └─────────────────────┘
┌─────────────────────┐ |     | ┌─────────────────────┐
| ServletInputStream |─┘     ├─|  DataInputStream   |
└─────────────────────┘       | └─────────────────────┘
                              | ┌─────────────────────┐
                              └─|CheckedInputStream  |
                                └─────────────────────┘
```

类似的，`OutputStream` 也是以这种模式来提供各种功能：

```
                ┌─────────────┐
                |OutputStream |
                └─────────────┘
                      ▲ ▲
┌─────────────────────┐ | | ┌─────────────────────┐
│  FileOutputStream   |─┤ └─|FilterOutputStream|
└─────────────────────┘ |   └─────────────────────┘
┌─────────────────────┐ |     ▲ ┌─────────────────────┐
```

```
 9. |ByteArrayOutputStream|─┤     ├─|BufferedOutputStream|
10. └──────────────────────┘ |     | └──────────────────────┘
11. ┌──────────────────────┐ |     | ┌──────────────────────┐
12. | ServletOutputStream |─┘     ├─|  DataOutputStream  |
13. └──────────────────────┘       | └──────────────────────┘
14.                               | ┌──────────────────────┐
15.                               └─|CheckedOutputStream |
16.                                 └──────────────────────┘
```

编写FilterInputStream

我们也可以自己编写 `FilterInputStream` ，以便可以把自己的 `FilterInputStream` “叠加”到任何一个 `InputStream` 中。

下面的例子演示了如何编写一个 `CountInputStream` ，它的作用是对输入的字节进行计数：

```
import java.io.*;
public class Main {
    public static void main(String[] args) throws IOException {
        byte[] data = "hello, world!".getBytes("UTF-8");
        try (CountInputStream input = new CountInputStream(new ByteArrayInputStream(data))) {
            int n;
            while ((n = input.read()) != -1) {
                System.out.println((char)n);
            }
            System.out.println("Total read " + input.getBytesRead() + " bytes");
        }
    }
}

class CountInputStream extends FilterInputStream {
    private int count = 0;

    CountInputStream(InputStream in) {
        super(in);
    }

    public int getBytesRead() {
        return this.count;
    }

    public int read() throws IOException {
        int n = in.read();
        if (n != -1) {
            this.count ++;
        }
        return n;
    }

    public int read(byte[] b, int off, int len) throws IOException {
        int n = in.read(b, off, len);
        this.count += n;
        return n;
    }
}
```

▶ Run

注意到在叠加多个 `FilterInputStream` ，我们只需要持有最外层的 `InputStream` ，并且，当最外层的 `InputStream` 关闭时（在 `try(resource)` 块的结束处自动关闭），内层的 `InputStream` 的 `close()` 方法也会被自动调用，并最终调用到最核心的“基础” `InputStream` ，因此不存在资源泄露。

小结

Java的IO标准库使用Filter模式为 `InputStream` 和 `OutputStream` 增加功能：

- 可以把一个 `InputStream` 和任意个 `FilterInputStream` 组合；
- 可以把一个 `OutputStream` 和任意个 `FilterOutputStream` 组合。

Filter模式可以在运行期动态增加功能（又称Decorator模式）。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





操作Zip

`ZipInputStream` 是一种 `FilterInputStream` ，它可以直接读取zip包的内容：

```
┌──────────────────┐
│    InputStream   │
└──────────────────┘
         ▲
         │
┌──────────────────┐
│ FilterInputStream │
└──────────────────┘
         ▲
         │
┌──────────────────┐
│InflaterInputStream│
└──────────────────┘
         ▲
         │
┌──────────────────┐
│  ZipInputStream  │
└──────────────────┘
         ▲
         │
┌──────────────────┐
│  JarInputStream  │
└──────────────────┘
```

另一个 `JarInputStream` 是从 `ZipInputStream` 派生，它增加的主要功能是直接读取jar文件里面的 `MANIFEST.MF` 文件。因为本质上jar包就是zip包，只是额外附加了一些固定的描述文件。

读取zip包

我们来看看 `ZipInputStream` 的基本用法。

我们要创建一个 `ZipInputStream` ，通常是传入一个 `FileInputStream` 作为数据源，然后，循环调用 `getNextEntry()` ，直到返回 `null` ，表示zip流结束。

一个 `ZipEntry` 表示一个压缩文件或目录，如果是压缩文件，我们就用 `read()` 方法不断读取，直到返回 `-1` ：

```
try (ZipInputStream zip = new ZipInputStream(new FileInputStream(...))) {
```

```
    ZipEntry entry = null;
    while ((entry = zip.getNextEntry()) != null) {
        String name = entry.getName();
        if (!entry.isDirectory()) {
            int n;
            while ((n = zip.read()) != -1) {
                ...
            }
        }
    }
}
```

写入zip包

`ZipOutputStream` 是一种 `FilterOutputStream` ，它可以直接写入内容到zip包。我们要先创建一个 `ZipOutputStream` ，通常是包装一个 `FileOutputStream` ，然后，每写入一个文件前，先调用 `putNextEntry()` ，然后用 `write()` 写入 `byte[]` 数据，写入完毕后调用 `closeEntry()` 结束这个文件的打包。

```
try (ZipOutputStream zip = new ZipOutputStream(new FileOutputStream(...))) {
    File[] files = ...
    for (File file : files) {
        zip.putNextEntry(new ZipEntry(file.getName()));
        zip.write(getFileDataAsBytes(file));
        zip.closeEntry();
    }
}
```

上面的代码没有考虑文件的目录结构。如果要实现目录层次结构， `new ZipEntry(name)` 传入的 `name` 要用相对路径。

小结

`ZipInputStream` 可以读取zip格式的流， `ZipOutputStream` 可以把多份数据写入zip包；

配合 `FileInputStream` 和 `FileOutputStream` 就可以读写zip文件。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





读取classpath资源

很多Java程序启动的时候，都需要读取配置文件。例如，从一个 `.properties` 文件中读取配置：

```
String conf = "C:\\conf\\default.properties";
try (InputStream input = new FileInputStream(conf)) {
    // TODO:
}
```

这段代码要正常执行，必须在C盘创建 `conf` 目录，然后在目录里创建 `default.properties` 文件。但是，在Linux系统上，路径和Windows的又不一样。

因此，从磁盘的固定目录读取配置文件，不是一个好的办法。

有没有路径无关的读取文件的方式呢？

我们知道，Java存放 `.class` 的目录或jar包也可以包含任意其他类型的文件，例如：

- 配置文件，例如 `.properties` ；
- 图片文件，例如 `.jpg` ；
- 文本文件，例如 `.txt` ， `.csv` ；
- ……

从classpath读取文件就可以避免不同环境下文件路径不一致的问题：如果我们把 `default.properties` 文件放到classpath中，就不用关心它的实际存放路径。

在classpath中的资源文件，路径总是以 `/` 开头，我们先获取当前的 `Class` 对象，然后调用 `getResourceAsStream()` 就可以直接从classpath读取任意的资源文件：

```
try (InputStream input = getClass().getResourceAsStream("/default.properties"))
{
    // TODO:
}
```

调用 `getResourceAsStream()` 需要特别注意的一点是，如果资源文件不存在，它将返回 `null` 。因此，我们需要检查返回的 `InputStream` 是否为 `null` ，如果为 `null` ，表示资源文件在classpath中没有找到：

```
try (InputStream input = getClass().getResourceAsStream("/default.properties"))
{
    if (input != null) {
        // TODO:
```

```
    }
}
```

如果我们把默认的配置放到jar包中，再从外部文件系统读取一个可选的配置文件，就可以做到既有默认的配置文件，又可以让用户自己修改配置：

```
Properties props = new Properties();
props.load(inputStreamFromClassPath("/default.properties"));
props.load(inputStreamFromFile("./conf.properties"));
```

这样读取配置文件，应用程序启动就更加灵活。

小结

把资源存储在classpath中可以避免文件路径依赖；

`Class` 对象的 `getResourceAsStream()` 可以从classpath中读取指定资源；

根据classpath读取资源时，需要检查返回的 `InputStream` 是否为 `null` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





序列化

序列化是指把一个Java对象变成二进制内容，本质上就是一个 `byte[]` 数组。

为什么要把Java对象序列化呢？因为序列化后可以把 `byte[]` 保存到文件中，或者把 `byte[]` 通过网络传输到远程，这样，就相当于把Java对象存储到文件或者通过网络传输出去了。

有序列化，就有反序列化，即把一个二进制内容（也就是 `byte[]` 数组）变回Java对象。有了反序列化，保存到文件中的 `byte[]` 数组又可以“变回”Java对象，或者从网络上读取 `byte[]` 并把它“变回”Java对象。

我们来看看如何把一个Java对象序列化。

一个Java对象要能序列化，必须实现一个特殊的 `java.io.Serializable` 接口，它的定义如下：

```
1. public interface Serializable {
2. }
```

`Serializable` 接口没有定义任何方法，它是一个空接口。我们把这样的空接口称为“标记接口”（Marker Interface），实现了标记接口的类仅仅是给自身贴了个“标记”，并没有增加任何方法。

序列化

把一个Java对象变为 `byte[]` 数组，需要使用 `ObjectOutputStream` 。它负责把一个Java对象写入一个字节流：

```
import java.io.*;
import java.util.Arrays;

public class Main {
    public static void main(String[] args) throws IOException {
        ByteArrayOutputStream buffer = new ByteArrayOutputStream();
        try (ObjectOutputStream output = new ObjectOutputStream(buffer)) {
            // 写入int:
            output.writeInt(12345);
            // 写入String:
            output.writeUTF("Hello");
            // 写入Object:
            output.writeObject(Double.valueOf(123.456));
        }
        System.out.println(Arrays.toString(buffer.toByteArray()));
    }
}
```

▶ Run

`ObjectOutputStream` 既可以写入基本类型，如 `int` ， `boolean` ，也可以写入 `String` （以UTF-8编码），还可以写入实现了 `Serializable` 接口的 `Object` 。

因为写入 `Object` 时需要大量的类型信息，所以写入的内容很大。

反序列化

和 `ObjectOutputStream` 相反， `ObjectInputStream` 负责从一个字节流读取Java对象：

```
try (ObjectInputStream input = new ObjectInputStream(...)) {
    int n = input.readInt();
    String s = input.readUTF();
    Double d = (Double) input.readObject();
}
```

除了能读取基本类型和 `String` 类型外，调用 `readObject()` 可以直接返回一个 `Object` 对象。要把它变成一个特定类型，必须强制转型。

`readObject()` 可能抛出的异常有：

- `ClassNotFoundException` ：没有找到对应的Class；
- `InvalidClassException` ：Class不匹配。

对于 `ClassNotFoundException` ，这种情况常见于一台电脑上的Java程序把一个Java对象，例如， `Person` 对象序列化以后，通过网络传给另一台电脑上的另一个Java程序，但是这台电脑的Java程序并没有定义 `Person` 类，所以无法反序列化。

对于 `InvalidClassException` ，这种情况常见于序列化的 `Person` 对象定义了一个 `int` 类型的 `age` 字段，但是反序列化时， `Person` 类定义的 `age` 字段被改成了 `long` 类型，所以导致class不兼容。

为了避免这种class定义变动导致的不兼容，Java的序列化允许class定义一个特殊的 `serialVersionUID` 静态变量，用于标识Java类的序列化“版本”，通常可以由IDE自动生成。如果增加或修改了字段，可以改变 `serialVersionUID` 的值，这样就能自动阻止不匹配的class版本：

```
public class Person implements Serializable {
    private static final long serialVersionUID = 2709425275741743919L;
}
```

要特别注意反序列化的几个重要特点：

反序列化时，由JVM直接构造出Java对象，不调用构造方法，构造方法内部的代码，在反序列化时根本不可能执行。

安全性

因为Java的序列化机制可以导致一个实例能直接从 `byte[]` 数组创建，而不经过构造方法，因此，它存在一定的安全隐患。一个精心构造的 `byte[]` 数组被反序列化后可以执行特定的Java代码，从而导致严重的安全漏洞。

实际上，Java本身提供的基于对象的序列化和反序列化机制既存在安全性问题，也存在兼容性问题。更好的序列化方法是通过JSON这样的通用数据结构来实现，只输出基本类型（包括String）的内容，而不存储任何与代码相关的信息。

小结

可序列化的Java对象必须实现 `java.io.Serializable` 接口，类似 `Serializable` 这样的空接口被称为“标记接口”（Marker Interface）；

反序列化时不调用构造方法，可设置 `serialVersionUID` 作为版本号（非必需）；

Java的序列化机制仅适用于Java，如果需要与其它语言交换数据，必须使用通用的序列化方法，例如JSON。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Reader

`Reader` 是Java的IO库提供的另一个输入流接口。和 `InputStream` 的区别是， `InputStream` 是一个字节流，即以 `byte` 为单位读取，而 `Reader` 是一个字符流，即以 `char` 为单位读取：

InputStream	Reader
字节流，以 `byte` 为单位	字符流，以 `char` 为单位
读取字节（-1，0~255）： `int read()`	读取字符（-1，0~65535）： `int read()`
读到字节数组： `int read(byte[] b)`	读到字符数组： `int read(char[] c)`

`java.io.Reader` 是所有字符输入流的超类，它最主要的方法是：

```
public int read() throws IOException;
```

这个方法读取字符流的下一个字符，并返回字符表示的 `int` ，范围是 `0` ~ `65535` 。如果已读到末尾，返回 `-1` 。

FileReader

`FileReader` 是 `Reader` 的一个子类，它可以打开文件并获取 `Reader` 。下面的代码演示了如何完整地读取一个 `FileReader` 的所有字符：

```
public void readFile() throws IOException {
    // 创建一个FileReader对象:
    Reader reader = new FileReader("src/readme.txt"); // 字符编码是???
    for (;;) {
        int n = reader.read(); // 反复调用read()方法，直到返回-1
        if (n == -1) {
            break;
        }
        System.out.println((char)n); // 打印char
    }
    reader.close(); // 关闭流
}
```

如果我们读取一个纯ASCII编码的文本文件，上述代码工作是没有问题的。但如果文件中包含中文，就会出现乱码，因为 `FileReader` 默认的编码与系统相关，例如，Windows系统的默认编码可能是 `GBK` ，打开一个 `UTF-8` 编码的文本文件就会出现乱码。

要避免乱码问题，我们需要在创建 `FileReader` 时指定编码：

```
Reader reader = new FileReader("src/readme.txt", StandardCharsets.UTF_8);
```

和 `InputStream` 类似， `Reader` 也是一种资源，需要保证出错的时候也能正确关闭，所以我们需要用 `try (resource)` 来保证 `Reader` 在无论有没有IO错误的时候都能够正确地关闭：

```
try (Reader reader = new FileReader("src/readme.txt", StandardCharsets.UTF_8) {
    // TODO
}
```

`Reader` 还提供了一次性读取若干字符并填充到 `char[]` 数组的方法：

```
public int read(char[] c) throws IOException
```

它返回实际读入的字符个数，最大不超过 `char[]` 数组的长度。返回 `-1` 表示流结束。

利用这个方法，我们可以先设置一个缓冲区，然后，每次尽可能地填充缓冲区：

```
public void readFile() throws IOException {
    try (Reader reader = new FileReader("src/readme.txt",
StandardCharsets.UTF_8)) {
        char[] buffer = new char[1000];
        int n;
        while ((n = reader.read(buffer)) != -1) {
            System.out.println("read " + n + " chars.");
        }
    }
}
```

CharArrayReader

`CharArrayReader` 可以在内存中模拟一个 `Reader` ，它的作用实际上是把一个 `char[]` 数组变成一个 `Reader` ，这和 `ByteArrayInputStream` 非常类似：

```
try (Reader reader = new CharArrayReader("Hello".toCharArray())) {
}
```

StringReader

`StringReader` 可以直接把 `String` 作为数据源，它和 `CharArrayReader` 几乎一样：

```
try (Reader reader = new StringReader("Hello")) {
```

```
2. }
```

InputStreamReader

`Reader` 和 `InputStream` 有什么关系？

除了特殊的 `CharArrayReader` 和 `StringReader` ，普通的 `Reader` 实际上是基于 `InputStream` 构造的，因为 `Reader` 需要从 `InputStream` 中读入字节流（ `byte` ），然后，根据编码设置，再转换为 `char` 就可以实现字符流。如果我们查看 `FileReader` 的源码，它在内部实际上持有一个 `FileInputStream` 。

既然 `Reader` 本质上是一个基于 `InputStream` 的 `byte` 到 `char` 的转换器，那么，如果我们已经有一个 `InputStream` ，想把它转换为 `Reader` ，是完全可行的。 `InputStreamReader` 就是这样一个转换器，它可以把任何 `InputStream` 转换为 `Reader` 。示例代码如下：

```
1. // 持有InputStream:
2. InputStream input = new FileInputStream("src/readme.txt");
3. // 变换为Reader:
4. Reader reader = new InputStreamReader(input, "UTF-8");
```

构造 `InputStreamReader` 时，我们需要传入 `InputStream` ，还需要指定编码，就可以得到一个 `Reader` 对象。上述代码可以通过 `try (resource)` 更简洁地改写如下：

```
    try (Reader reader = new InputStreamReader(new
1. FileInputStream("src/readme.txt"), "UTF-8")) {
2.     // TODO:
3. }
```

上述代码实际上就是 `FileReader` 的一种实现方式。

使用 `try (resource)` 结构时，当我们关闭 `Reader` 时，它会在内部自动调用 `InputStream` 的 `close()` 方法，所以，只需要关闭最外层的 `Reader` 对象即可。

使用InputStreamReader，可以把一个InputStream转换成一个Reader。

小结

`Reader` 定义了所有字符输入流的超类：

- `FileReader` 实现了文件字符流输入，使用时需要指定编码；
- `CharArrayReader` 和 `StringReader` 可以在内存中模拟一个字符流输入。

`Reader` 是基于 `InputStream` 构造的：可以通过 `InputStreamReader` 在指定编码的同时将任

何 `InputStream` 转换为 `Reader` 。

总是使用 `try (resource)` 保证 `Reader` 正确关闭。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Writer

`Reader`是带编码转换器的`InputStream`，它把`byte`转换为`char`，而`Writer`就是带编码转换器的`OutputStream`，它把`char`转换为`byte`并输出。

`Writer`和`OutputStream`的区别如下：

OutputStream	Writer
字节流，以`byte`为单位	字符流，以`char`为单位
写入字节（0~255）：`void write(int b)`	写入字符（0~65535）：`void write(int c)`
写入字节数组：`void write(byte[] b)`	写入字符数组：`void write(char[] c)`
无对应方法	写入String：`void write(String s)`

`Writer`是所有字符输出流的超类，它提供的方法主要有：

- 写入一个字符（0~65535）：`void write(int c)`；
- 写入字符数组的所有字符：`void write(char[] c)`；
- 写入String表示的所有字符：`void write(String s)`。

FileWriter

`FileWriter`就是向文件中写入字符流的`Writer`。它的使用方法和`FileReader`类似：

```
try (Writer writer = new FileWriter("readme.txt", StandardCharsets.UTF_8)) {
    writer.write('H'); // 写入单个字符
    writer.write("Hello".toCharArray()); // 写入char[]
    writer.write("Hello"); // 写入String
}
```

CharArrayWriter

`CharArrayWriter`可以在内存中创建一个`Writer`，它的作用实际上是构造一个缓冲区，可以写入`char`，最后得到写入的`char[]`数组，这和`ByteArrayOutputStream`非常类似：

```
try (CharArrayWriter writer = new CharArrayWriter()) {
    writer.write(65);
    writer.write(66);
    writer.write(67);
    char[] data = writer.toCharArray(); // { 'A', 'B', 'C' }
}
```

StringWriter

`StringWriter` 也是一个基于内存的 `Writer` ，它和 `CharArrayWriter` 类似。实际上， `StringWriter` 在内部维护了一个 `StringBuffer` ，并对外提供了 `Writer` 接口。

OutputStreamWriter

除了 `CharArrayWriter` 和 `StringWriter` 外，普通的Writer实际上是基于 `OutputStream` 构造的，它接收 `char` ，然后在内部自动转换成一个或多个 `byte` ，并写入 `OutputStream` 。因此， `OutputStreamWriter` 就是一个将任意的 `OutputStream` 转换为 `Writer` 的转换器：

```
    try (Writer writer = new OutputStreamWriter(new FileOutputStream("readme.txt"),
1.  "UTF-8")) {
2.      // TODO:
3.  }
```

上述代码实际上就是 `FileWriter` 的一种实现方式。这和上一节的 `InputStreamReader` 是一样的。

小结

`Writer` 定义了所有字符输出流的超类：

- `FileWriter` 实现了文件字符流输出；
- `CharArrayWriter` 和 `StringWriter` 在内存中模拟一个字符流输出。

使用 `try (resource)` 保证 `Writer` 正确关闭。

`Writer` 是基于 `OutputStream` 构造的，可以通过 `OutputStreamWriter` 将 `OutputStream` 转换为 `Writer` ，转换时需要指定编码。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





PrintStream和PrintWriter

`PrintStream` 是一种 `FilterOutputStream` ，它在 `OutputStream` 的接口上，额外提供了一些写入各种数据类型的方法：

- 写入 `int` ： `print(int)`
- 写入 `boolean` ： `print(boolean)`
- 写入 `String` ： `print(String)`
- 写入 `Object` ： `print(Object)` ，实际上相当于 `print(object.toString())`
- …

以及对应的一组 `println()` 方法，它会自动加上换行符。

我们经常使用的 `System.out.println()` 实际上就是使用 `PrintStream` 打印各种数据。其中， `System.out` 是系统默认提供的 `PrintStream` ，表示标准输出：

```
1. System.out.print(12345); // 输出12345
2. System.out.print(new Object()); // 输出类似java.lang.Object@3c7a835a
3. System.out.println("Hello"); // 输出Hello并换行
```

`System.err` 是系统默认提供的标准错误输出。

`PrintStream` 和 `OutputStream` 相比，除了添加了一组 `print()` / `println()` 方法，可以打印各种数据类型，比较方便外，它还有一个额外的优点，就是不会抛出 `IOException` ，这样我们在编写代码的时候，就不必捕获 `IOException` 。

PrintWriter

`PrintStream` 最终输出的总是byte数据，而 `PrintWriter` 则是扩展了 `Writer` 接口，它的 `print()` / `println()` 方法最终输出的是 `char` 数据。两者的使用方法几乎是一模一样的：

```
import java.io.*;
public class Main {
    public static void main(String[] args)     {
        StringWriter buffer = new StringWriter();
        try (PrintWriter pw = new PrintWriter(buffer)) {
            pw.println("Hello");
            pw.println(12345);
            pw.println(true);
        }
        System.out.println(buffer.toString());
    }
}
```

▶ Run

小结

`PrintStream` 是一种能接收各种数据类型的输出，打印数据时比较方便：

- `System.out` 是标准输出；
- `System.err` 是标准错误输出。

`PrintWriter` 是基于 `Writer` 的输出。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论

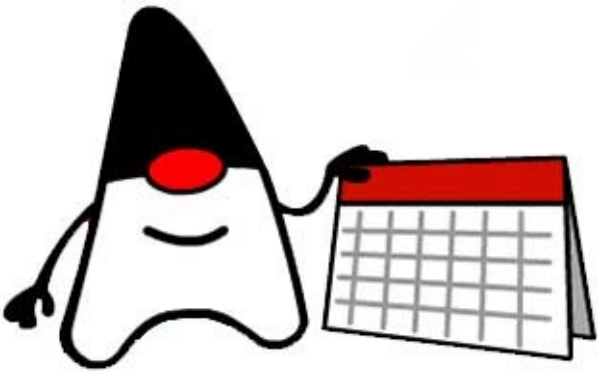




日期与时间

日期与时间是计算机处理的重要数据。绝大部分程序的运行都要和时间打交道。

本节我们将详细讲解Java程序如何正确处理日期与时间。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





基本概念

在计算机中，我们经常需要处理日期和时间。

这是日期:

- 2019-11-20
- 2020-1-1

这是时间:

- 12:30:59
- 2020-1-1 20:21:59

日期是指某一天，它不是连续变化的，而是应该被看成离散的。

而时间有两种概念，一种是不带日期的时间，例如，12:30:59。另一种是带日期的时间，例如，2020-1-1 20:21:59，只有这种带日期的时间能唯一确定某个时刻，不带日期的时间是无法确定一个唯一时刻的。

本地时间

当我们说当前时刻是2019年11月20日早上8:15的时候，我们说的实际上是本地时间。在国内就是北京时间。在这个时刻，如果地球上不同地方的人们同时看一眼手表，他们各自的本地时间是不同的:



所以，不同的时区，在同一时刻，本地时间是不同的。全球一共分为24个时区，伦敦所在的时区称为标准时区，其他时区按东 / 西偏移的小时区分，北京所在的时区是东八区。

时区

因为光靠本地时间还无法唯一确定一个准确的时刻，所以我们还需要给本地时间加上一个时区。时区有

好几种表示方式。

一种是以 `GMT` 或者 `UTC` 加时区偏移表示，例如： `GMT+08:00` 或者 `UTC+08:00` 表示东八区。

`GMT` 和 `UTC` 可以认为基本是等价的，只是 `UTC` 使用更精确的原子钟计时，每隔几年会有一个闰秒，我们在开发程序的时候可以忽略两者的误差，因为计算机的时钟在联网的时候会自动与时间服务器同步时间。

另一种是缩写，例如， `CST` 表示 `China Standard Time` ，也就是中国标准时间。但是 `CST` 也可以表示美国中部时间 `Central Standard Time USA` ，因此，缩写容易产生混淆，我们尽量不要使用缩写。

最后一种是以洲 / 城市表示，例如， `Asia/Shanghai` ，表示上海所在地的时区。特别注意城市名称不是任意的城市，而是由国际标准组织规定的城市。

因为时区的存在，东八区的2019年11月20日早上8:15，和西五区的2019年11月19日晚上19:15，他们的时刻是相同的：



时刻相同的意思就是，分别在两个时区的两个人，如果在这一刻通电话，他们各自报出自己手表上的时间，虽然本地时间是不同的，但是这两个时间表示的时刻是相同的。

夏令时

时区还不是最复杂的，更复杂的是夏令时。所谓夏令时，就是夏天开始的时候，把时间往后拨1小时，夏天结束的时候，再把时间往前拨1小时。我们国家实行过一段时间夏令时，1992年就废除了，但是矫情的美国人到现在还在使用，所以时间换算更加复杂。



因为涉及到夏令时，相同的时区，如果表示的方式不同，转换出的时间是不同的。我们举个栗子：

对于2019-11-20和2019-6-20两个日期来说，假设北京人在纽约：

- 如果以 `GMT` 或者 `UTC` 作为时区，无论日期是多少，时间都是 `19:00` ；
- 如果以国家 / 城市表示，例如 `America/NewYork` ，虽然纽约也在西五区，但是，因为夏令时的存在，在不同的日期， `GMT` 时间和纽约时间可能是不一样的：

时区	2019-11-20	2019-6-20
GMT-05:00	19:00	19:00
UTC-05:00	19:00	19:00
America/New_York	19:00	20:00

实行夏令时的不同地区，进入和退出夏令时的时间很可能是不同的。同一个地区，根据历史上是否实行过夏令时，标准时间在不同年份换算成当地时间也是不同的。因此，计算夏令时，没有统一的公式，必须按照一组给定的规则来算，并且，该规则要定期更新。

计算夏令时请使用标准库提供的相关类，不要试图自己计算夏令时。

本地化

在计算机中，通常使用 `Locale` 表示一个国家或地区的日期、时间、数字、货币等格式。 `Locale` 由 `语言_国家` 的字母缩写构成，例如， `zh_CN` 表示中文+中国， `en_US` 表示英文+美国。语言使用小写，国家使用大写。

对于日期来说，不同的Locale，例如，中国和美国的表示方式如下：

- zh_CN: 2016-11-30
- en_US: 11/30/2016

计算机用 `Locale` 在日期、时间、货币和字符串之间进行转换。一个电商网站会根据用户所在

的 `Locale` 对用户显示如下：

	中国用户	美国用户
购买价格	12000.00	12,000.00
购买日期	2016-11-30	11/30/2016

小结

在编写日期和时间的程序前，我们要准确理解日期、时间和时刻的概念。

由于存在本地时间，我们需要理解时区的概念，并且必须牢记由于夏令时的存在，同一地区用 `GMT/UTC` 和城市表示的时区可能导致时间不同。

计算机通过 `Locale` 来针对当地用户习惯格式化日期、时间、数字、货币等。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Date和Calendar

在计算机中，应该如何表示日期和时间呢？

我们经常看到的日期和时间表示方式如下：

- 2019-11-20 0:15:00 GMT+00:00
- 2019年11月20日8:15:00
- 11/19/2019 19:15:00 America/New_York

如果直接以字符串的形式存储，那么不同的格式，不同的语言会让表示方式非常繁琐。

在理解日期和时间的表示方式之前，我们先要理解数据的存储和展示。

当我们定义一个整型变量并赋值时：

```
int n = 123400;
```

编译器会把上述字符串（程序源码就是一个字符串）编译成字节码。在程序的运行期，变量 `n` 指向的内存实际上是一个4字节区域：

```
┌──┬──┬──┬──┐
│00│01│e2│08│
└──┴──┴──┴──┘
```

注意到计算机内存除了二进制的 `0` / `1` 外没有其他任何格式。上述十六机制是为了简化表示。

当我们用 `System.out.println(n)` 打印这个整数的时候，实际上 `println()` 这个方法在内部把 `int` 类型转换成 `String` 类型，然后打印出字符串 `123400` 。

类似的，我们也可以以十六进制的形式打印这个整数，或者，如果 `n` 表示一个价格，我们就以 `$123,400.00` 的形式来打印它：

```
import java.text.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        int n = 123400;
        // 123400
        System.out.println(n);
        // 1e208
        System.out.println(Integer.toHexString(n));
        // $123,400.00
        System.out.println(NumberFormat.getCurrencyInstance(Locale.US).format(n));
    }
}
```

▶ Run

可见，整数 `123400` 是数据的存储格式，它的存储格式非常简单。而我们打印的各种各样的字符串，则是数据的展示格式。展示格式有多种形式，但本质上它就是一个转换方法：

```
String toDisplay(int n) { ... }
```

理解了数据的存储和展示，我们回头看看以下几种日期和时间：

- 2019-11-20 0:15:01 GMT+00:00
- 2019年11月20日8:15:01
- 11/19/2019 19:15:01 America/New_York

它们实际上是数据的展示格式，分别按英国时区、中国时区、纽约时区对同一个时刻进行展示。而这个“同一个时刻”在计算机中存储的本质上只是一个整数，我们称它为 `Epoch Time` 。

`Epoch Time` 是计算从1970年1月1日零点（格林威治时区／GMT+00:00）到现在所经历的秒数，例如：

`1574208900` 表示从从1970年1月1日零点GMT时区到该时刻一共经历了1574208900秒，换算成伦敦、北京和纽约时间分别是：

```
1574208900 = 北京时间2019-11-20 8:15:00
           = 伦敦时间2019-11-20 0:15:00
           = 纽约时间2019-11-19 19:15:00
```



因此，在计算机中，只需要存储一个整数 `1574208900` 表示某一时刻。当需要显示为某一地区的当地时间时，我们就把它格式化为一个字符串：

```
1.  String displayDateTime(int n, String timezone) { ... }
```

`Epoch Time` 又称为时间戳，在不同的编程语言中，会有几种存储方式：

- 以秒为单位的整数：1574208900，缺点是精度只能到秒；
- 以毫秒为单位的整数：1574208900123，最后3位表示毫秒数；
- 以秒为单位的浮点数：1574208900.123，小数点后面表示零点几秒。

它们之间转换非常简单。而在Java程序中，时间戳通常是用 `long` 表示的毫秒数，即：

```
1.  long t = 1574208900123L;
```

转换成北京时间就是 `2019-11-20T8:15:00.123` 。要获取当前时间戳，可以使用 `System.currentTimeMillis()` ，这是Java程序获取时间戳最常用的方法。

标准库API

我们再来看一下Java标准库提供的API。Java标准库有两套处理日期和时间的API：

- 一套定义在 `java.util` 这个包里面，主要包括 `Date` 、 `Calendar` 和 `TimeZone` 这几个类；
- 一套新的API是在Java 8引入的，定义在 `java.time` 这个包里面，主要包括 `LocalDateTime` 、 `ZonedDateTime` 、 `ZoneId` 等。

为什么会有新旧两套API呢？因为历史遗留原因，旧的API存在很多问题，所以引入了新的API。

那么我们能不能跳过旧的API直接用新的API呢？如果涉及到遗留代码就不行，因为很多遗留代码仍然使用旧的API，所以目前仍然需要对旧的API有一定了解，很多时候还需要在新旧两种对象之间进行转

换。

本节我们快速讲解旧API的常用类型和方法。

Date

`java.util.Date` 是用于表示一个日期和时间的对象，注意与 `java.sql.Date` 区分，后者用在数据库中。如果观察Date的源码，可以发现它实际上存储了一个long类型的以毫秒表示的时间戳：

```
1. public class Date implements Serializable, Cloneable, Comparable<Date> {
2.
3.     private transient long fastTime;
4.
5.     ...
6. }
```

我们来看Date的基本用法：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        // 获取当前时间:
        Date date = new Date();
        System.out.println(date.getYear() + 1900); // 必须加上1900
        System.out.println(date.getMonth() + 1); // 0~11，必须加上1
        System.out.println(date.getDate()); // 1~31，不能加1
        // 转换为String:
        System.out.println(date.toString());
        // 转换为GMT时区:
        System.out.println(date.toGMTString());
        // 转换为本地时区:
        System.out.println(date.toLocaleString());
    }
}
```

▶ Run

注意 `getYear()` 返回的年份必须加上 `1900` ， `getMonth()` 返回的月份是 `0` ~ `11` 分别表示1~12月，所以要加1，而 `getDate()` 返回的日期范围是 `1` ~ `31` ，又不能加1。

打印本地时区表示的日期和时间时，不同的计算机可能会有不同的结果。如果我们想要针对用户的偏好精确地控制日期和时间的格式，就可以使用 `SimpleDateFormat` 对一个 `Date` 进行转换。它用预定义的字符串表示格式化：

- yyyy：年
- MM：月

- dd: 日
- HH: 小时
- mm: 分钟
- ss: 秒

我们来看如何以自定义的格式输出:

```
import java.text.*;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        // 获取当前时间:
        Date date = new Date();
        var sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        System.out.println(sdf.format(date));
    }
}
```

▶ Run

Java的格式化预定义了许多不同的格式，我们以 `MMM` 和 `E` 为例:

```
import java.text.*;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        // 获取当前时间:
        Date date = new Date();
        var sdf = new SimpleDateFormat("E MMM dd, yyyy");
        System.out.println(sdf.format(date));
    }
}
```

▶ Run

上述代码在不同的语言环境会打印出类似 `Sun Sep 15, 2019` 这样的日期。可以从JDK文档查看详细的格式说明。一般来说，字母越长，输出越长。以 `M` 为例，假设当前月份是9月:

- `M`: 输出 `9`
- `MM`: 输出 `09`
- `MMM`: 输出 `Sep`
- `MMMM`: 输出 `September`

`Date` 对象有几个严重的问题: 它不能转换时区，除了 `toGMTString()` 可以按 `GMT+0:00` 输出外，Date总是以当前计算机系统的默认时区为基础进行输出。此外，我们也很难对日期和时间进行加

减，计算两个日期相差多少天，计算某个月第一个星期一的日期等。

Calendar

`Calendar` 可以用于获取并设置年、月、日、时、分、秒，它和 `Date` 比，主要多了一个可以做简单的日期和时间运算的功能。

我们来看 `Calendar` 的基本用法：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        // 获取当前时间:
        Calendar c = Calendar.getInstance();
        int y = c.get(Calendar.YEAR);
        int m = 1 + c.get(Calendar.MONTH);
        int d = c.get(Calendar.DAY_OF_MONTH);
        int w = c.get(Calendar.DAY_OF_WEEK);
        int hh = c.get(Calendar.HOUR_OF_DAY);
        int mm = c.get(Calendar.MINUTE);
        int ss = c.get(Calendar.SECOND);
        int ms = c.get(Calendar.MILLISECOND);
        System.out.println(y + "-" + m + "-" + d + " " + w + " " + hh + ":" + mm + ":" + ss + "." +
ms);
    }
}
```

▶ Run

注意到 `Calendar` 获取年月日这些信息变成了 `get(int field)` ，返回的年份不必转换，返回的月份仍然要加1，返回的星期要特别注意， `1` ~ `7` 分别表示周日，周一，……，周六。

`Calendar` 只有一种方式获取，即 `Calendar.getInstance()` ，而且一获取到就是当前时间。如果我们想给它设置成特定的一个日期和时间，就必须先清除所有字段：

```
import java.text.*;
import java.util.*;
public class Main {
    public static void main(String[] args) {
        // 当前时间:
        Calendar c = Calendar.getInstance();
        // 清除所有:
        c.clear();
        // 设置2019年:
        c.set(Calendar.YEAR, 2019);
        // 设置9月:注意8表示9月:
        c.set(Calendar.MONTH, 8);
        // 设置2日:
        c.set(Calendar.DATE, 2);
        // 设置时间:
        c.set(Calendar.HOUR_OF_DAY, 21);
        c.set(Calendar.MINUTE, 22);
        c.set(Calendar.SECOND, 23);
        System.out.println(new SimpleDateFormat("yyyy-MM-dd HH:mm:ss").format(c.getTime()));
        // 2019-09-02 21:22:23
    }
}
```

▶ Run

利用 `Calendar.getTime()` 可以将一个 `Calendar` 对象转换成 `Date` 对象，然后就可以用 `SimpleDateFormat` 进行格式化了。

TimeZone

`Calendar` 和 `Date` 相比，它提供了时区转换的功能。时区用 `TimeZone` 对象表示：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        TimeZone tzDefault = TimeZone.getDefault(); // 当前时区
        TimeZone tzGMT9 = TimeZone.getTimeZone("GMT+09:00"); // GMT+9:00时区
        TimeZone tzNY = TimeZone.getTimeZone("America/New_York"); // 纽约时区
        System.out.println(tzDefault.getID()); // Asia/Shanghai
        System.out.println(tzGMT9.getID()); // GMT+09:00
        System.out.println(tzNY.getID()); // America/New_York
    }
}
```

▶ Run

时区的唯一标识是以字符串表示的ID，我们获取指定 `TimeZone` 对象也是以这个ID为参数获取， `GMT+09:00` 、 `Asia/Shanghai` 都是有效的时区ID。要列出系统支持的所有ID，请使用 `TimeZone.getAvailableIDs()` 。

有了时区，我们就可以对指定时间进行转换。例如，下面的例子演示了如何将北京时间 `2019-11-20 8:15:00` 转换为纽约时间：

```
import java.text.*;
import java.util.*;

public class Main {
    public static void main(String[] args) {
        // 当前时间:
        Calendar c = Calendar.getInstance();
        // 清除所有:
        c.clear();
        // 设置为北京时区:
        c.setTimeZone(TimeZone.getTimeZone("Asia/Shanghai"));
        // 设置年月日时分秒:
        c.set(2019, 10 /* 11月 */, 20, 8, 15, 0);
        // 显示时间:
        var sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        sdf.setTimeZone(TimeZone.getTimeZone("America/New_York"));
        System.out.println(sdf.format(c.getTime()));
        // 2019-11-19 19:15:00
    }
}
```

▶ Run

可见，利用 `Calendar` 进行时区转换的步骤是：

- 清除所有字段；
- 设定指定时区；
- 设定日期和时间；
- 创建 `SimpleDateFormat` 并设定目标时区；
- 格式化获取的 `Date` 对象（注意 `Date` 对象无时区信息，时区信息存储在 `SimpleDateFormat` 中）。因此，本质上时区转换只能通过 `SimpleDateFormat` 在显示的时候完成。

`Calendar` 也可以对日期和时间进行简单的加减：

```
import java.text.*;
import java.util.*;
```

```
public class Main {
    public static void main(String[] args) {
        // 当前时间:
        Calendar c = Calendar.getInstance();
        // 清除所有:
        c.clear();
        // 设置年月日时分秒:
        c.set(2019, 10 /* 11月 */, 20, 8, 15, 0);
        // 加5天并减去2小时:
        c.add(Calendar.DAY_OF_MONTH, 5);
        c.add(Calendar.HOUR_OF_DAY, -2);
        // 显示时间:
        var sdf = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        Date d = c.getTime();
        System.out.println(sdf.format(d));
        // 2019-11-25 6:15:00
    }
}
```

▶ Run

小结

计算机表示的时间是以整数表示的时间戳存储的，即Epoch Time，Java使用 `long` 型来表示以毫秒为单位的时间戳，通过 `System.currentTimeMillis()` 获取当前时间戳。

Java有两套日期和时间的API：

- 旧的Date、Calendar和TimeZone；
- 新的LocalDateTime、ZonedDateTime、ZoneId等。

分别位于 `java.util` 和 `java.time` 包中。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





LocalDateTime

从Java 8开始，`java.time`包提供了新的日期和时间API，主要涉及的类型有：

- 本地日期和时间：`LocalDateTime`，`LocalDate`，`LocalTime`；
- 带时区的日期和时间：`ZonedDateTime`；
- 时刻：`Instant`；
- 时区：`ZoneId`，`ZoneOffset`；
- 时间间隔：`Duration`。

以及一套新的用于取代`SimpleDateFormat`的格式化类型`DateTimeFormatter`。

和旧的API相比，新API严格区分了时刻、本地日期、本地时间和带时区的日期时间，并且，对日期和时间进行运算更加方便。

此外，新API修正了旧API不合理的常量设计：

- Month的范围用1~12表示1月到12月；
- Week的范围用1~7表示周一到周日。

最后，新API的类型几乎全部是不变类型（和String类似），可以放心使用不必担心被修改。

LocalDateTime

我们首先来看最常用的`LocalDateTime`，它表示一个本地日期和时间：

```
import java.time.*;

public class Main {
    public static void main(String[] args) {
        LocalDate d = LocalDate.now(); // 当前日期
        LocalTime t = LocalTime.now(); // 当前时间
        LocalDateTime dt = LocalDateTime.now(); // 当前日期和时间
        System.out.println(d); // 严格按照ISO 8601格式打印
        System.out.println(t); // 严格按照ISO 8601格式打印
        System.out.println(dt); // 严格按照ISO 8601格式打印
    }
}
```

▶ Run

本地日期和时间通过now()获取到的总是以当前默认时区返回的，和旧API不同，`LocalDateTime`、`LocalDate`和`LocalTime`默认严格按照ISO 8601规定的日期和时间格式进行打印。

上述代码其实有一个小问题，在获取3个类型的时候，由于执行一行代码总会消耗一点时间，因此，3个

类型的日期和时间很可能对不上（时间的毫秒数基本上不同）。为了保证获取到同一时刻的日期和时间，可以改写如下：

```
LocalDateTime dt = LocalDateTime.now(); // 当前日期和时间
LocalDate d = dt.toLocalDate(); // 转换到当前日期
LocalTime t = dt.toLocalTime(); // 转换到当前时间
```

反过来，通过指定的日期和时间创建 `LocalDateTime` 可以通过 `of()` 方法：

```
// 指定日期和时间:
LocalDate d2 = LocalDate.of(2019, 11, 30); // 2019-11-30, 注意11=11月
LocalTime t2 = LocalTime.of(15, 16, 17); // 15:16:17
LocalDateTime dt2 = LocalDateTime.of(2019, 11, 30, 15, 16, 17);
LocalDateTime dt3 = LocalDateTime.of(d2, t2);
```

因为严格按照ISO 8601的格式，因此，将字符串转换为 `LocalDateTime` 就可以传入标准格式：

```
LocalDateTime dt = LocalDateTime.parse("2019-11-19T15:16:17");
LocalDate d = LocalDate.parse("2019-11-19");
LocalTime t = LocalTime.parse("15:16:17");
```

注意ISO 8601规定的日期和时间分隔符是 `T` 。标准格式如下：

- 日期：yyyy-MM-dd
- 时间：HH:mm:ss
- 带毫秒的时间：HH:mm:ss.SSS
- 日期和时间：yyyy-MM-dd'T'HH:mm:ss
- 带毫秒的日期和时间：yyyy-MM-dd'T'HH:mm:ss.SSS

DateTimeFormatter

如果要自定义输出的格式，或者要把一个非ISO 8601格式的字符串解析成 `LocalDateTime` ，可以使用新的 `DateTimeFormatter` ：

```
import java.time.*;
import java.time.format.*;
```

```
public class Main {
    public static void main(String[] args) {
        // 自定义格式化:
        DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");
        System.out.println(dtf.format(LocalDateTime.now()));

        // 用自定义格式解析:
        LocalDateTime dt2 = LocalDateTime.parse("2019/11/30 15:16:17", dtf);
        System.out.println(dt2);
    }
}
```

▶ Run

`LocalDateTime` 提供了对日期和时间进行加减的非常简单的链式调用：

```
import java.time.*;
```

```
public class Main {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2019, 10, 26, 20, 30, 59);
        System.out.println(dt);
        // 加5天减3小时:
        LocalDateTime dt2 = dt.plusDays(5).minusHours(3);
        System.out.println(dt2); // 2019-10-31T17:30:59
        // 减1月:
        LocalDateTime dt3 = dt2.minusMonths(1);
        System.out.println(dt3); // 2019-09-30T17:30:59
    }
}
```

▶ Run

注意到月份加减会自动调整日期，例如从 `2019-10-31` 减去1个月得到的结果是 `2019-09-30` ，因为9月没有31日。

对日期和时间进行调整则使用 `withXxx()` 方法，例如： `withHour(15)` 会把 `10:11:12` 变为 `15:11:12` ：

- 调整年：withYear()
- 调整月：withMonth()
- 调整日：withDayOfMonth()
- 调整时：withHour()
- 调整分：withMinute()
- 调整秒：withSecond()

示例代码如下：

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        LocalDateTime dt = LocalDateTime.of(2019, 10, 26, 20, 30, 59);
        System.out.println(dt);
        // 日期变为31日:
        LocalDateTime dt2 = dt.withDayOfMonth(31);
        System.out.println(dt2); // 2019-10-31T20:30:59
        // 月份变为9:
        LocalDateTime dt3 = dt2.withMonth(9);
        System.out.println(dt3); // 2019-09-30T20:30:59
    }
}
```

▶ Run

同样注意到调整月份时，会相应地调整日期，即把 `2019-10-31` 的月份调整为 `9` 时，日期也自动变为 `30` 。

实际上， `LocalDateTime` 还有一个通用的 `with()` 方法允许我们做更复杂的运算。例如：

```
import java.time.*;
import java.time.temporal.*;
public class Main {
    public static void main(String[] args) {
        // 本月第一天0:00时刻:
        LocalDateTime firstDay = LocalDate.now().withDayOfMonth(1).atStartOfDay();
        System.out.println(firstDay);

        // 本月最后1天:
        LocalDate lastDay = LocalDate.now().with(TemporalAdjusters.lastDayOfMonth());
        System.out.println(lastDay);

        // 下月第1天:
        LocalDate nextMonthFirstDay = LocalDate.now().with(TemporalAdjusters.firstDayOfNextMonth());
        System.out.println(nextMonthFirstDay);

        // 本月第1个周一:
        LocalDate firstWeekday =
LocalDate.now().with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY));
        System.out.println(firstWeekday);
    }
}
```

▶ Run

对于计算某个月第1个周日这样的问题，新的API可以轻松完成。

要判断两个 `LocalDateTime` 的先后，可以使用 `isBefore()` 、 `isAfter()` 方法，对

于 `LocalDate` 和 `LocalTime` 类似：

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        LocalDateTime now = LocalDateTime.now();
        LocalDateTime target = LocalDateTime.of(2019, 11, 19, 8, 15, 0);
        System.out.println(now.isBefore(target));
        System.out.println(LocalDate.now().isBefore(LocalDate.of(2019, 11, 19)));
        System.out.println(LocalTime.now().isAfter(LocalTime.parse("08:15:00")));
    }
}
```

▶ Run

注意到 `LocalDateTime` 无法与时间戳进行转换，因为 `LocalDateTime` 没有时区，无法确定某一时刻。后面我们要介绍的 `ZonedDateTime` 相当于 `LocalDateTime` 加时区的组合，它具有时区，可以与 `long` 表示的时间戳进行转换。

Duration和Period

`Duration` 表示两个时刻之间的时间间隔。另一个类似的 `Period` 表示两个日期之间的天数：

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        LocalDateTime start = LocalDateTime.of(2019, 11, 19, 8, 15, 0);
        LocalDateTime end = LocalDateTime.of(2020, 1, 9, 19, 25, 30);
        Duration d = Duration.between(start, end);
        System.out.println(d); // PT1235H10M30S

        Period p = LocalDate.of(2019, 11, 19).until(LocalDate.of(2020, 1, 9));
        System.out.println(p); // P1M21D
    }
}
```

▶ Run

注意到两个 `LocalDateTime` 之间的差值使用 `Duration` 表示，类似 `PT1235H10M30S` ，表示1235小时10分钟30秒。而两个 `LocalDate` 之间的差值用 `Period` 表示，类似 `P1M21D` ，表示1个月21天。

`Duration` 和 `Period` 的表示方法也符合ISO 8601的格式，它以 `P...T...` 的形式表示， `P...T` 之间表示日期间隔， `T` 后面表示时间间隔。如果是 `PT...` 的格式表示仅有时间间隔。利用 `ofXxx()` 或者 `parse()` 方法也可以直接创建 `Duration` ：

```
Duration d1 = Duration.ofHours(10); // 10 hours
Duration d2 = Duration.parse("P1DT2H3M"); // 1 day, 2 hours, 3 minutes
```

有的童鞋可能发现Java 8引入的 `java.time` API。怎么和一个开源的Joda Time很像？难道JDK也开始抄袭开源了？其实正是因为开源的Joda Time设计很好，应用广泛，所以JDK团队邀请Joda Time的作者Stephen Colebourne共同设计了 `java.time` API。

小结

Java 8引入了新的日期和时间API，它们是不变类，默认按ISO 8601标准格式化和解析；

使用 `LocalDateTime` 可以非常方便地对日期和时间进行加减，或者调整日期和时间，它总是返回新对象；

使用 `isBefore()` 和 `isAfter()` 可以判断日期和时间的先后；

使用 `Duration` 和 `Period` 可以表示两个日期和时间的“区间间隔”。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





ZonedDateTime

`LocalDateTime` 总是表示本地日期和时间，要表示一个带时区的日期和时间，我们就需要 `ZonedDateTime` 。

可以简单地把 `ZonedDateTime` 理解成 `LocalDateTime` 加 `ZoneId` 。 `ZoneId` 是 `java.time` 引入的新的时区类，注意和旧的 `java.util.TimeZone` 区别。

要创建一个 `ZonedDateTime` 对象，有以下几种方法，一种是通过 `now()` 方法返回当前时间：

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        ZonedDateTime zbj = ZonedDateTime.now(); // 默认时区
        ZonedDateTime zny = ZonedDateTime.now(ZoneId.of("America/New_York")); // 用指定时区获取当前时间
        System.out.println(zbj);
        System.out.println(zny);
    }
}
```

▶ Run

观察打印的两个 `ZonedDateTime` ，发现它们时区不同，但表示的时间都是同一时刻（毫秒数不同是执行语句时的时间差）：

```
2019-09-15T20:58:18.786182+08:00[Asia/Shanghai]
2019-09-15T08:58:18.788860-04:00[America/New_York]
```

另一种方式是通过给一个 `LocalDateTime` 附加一个 `ZoneId` ，就可以变成 `ZonedDateTime` ：

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        LocalDateTime ldt = LocalDateTime.of(2019, 9, 15, 15, 16, 17);
        ZonedDateTime zbj = ldt.atZone(ZoneId.systemDefault());
        ZonedDateTime zny = ldt.atZone(ZoneId.of("America/New_York"));
        System.out.println(zbj);
        System.out.println(zny);
    }
}
```

▶ Run

以这种方式创建的 `ZonedDateTime` ，它的日期和时间与 `LocalDateTime` 相同，但附加的时区不同，因此是两个不同的时刻：

```
2019-09-15T15:16:17+08:00[Asia/Shanghai]
2019-09-15T15:16:17-04:00[America/New_York]
```

时区转换

要转换时区，首先我们需要有一个 `ZonedDateTime` 对象，然后，通过 `withZoneSameInstant()` 将关联时区转换到另一个时区，转换后日期和时间都会相应调整。

下面的代码演示了如何将北京时间转换为纽约时间：

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        // 以中国时区获取当前时间:
        ZonedDateTime zbj = ZonedDateTime.now(ZoneId.of("Asia/Shanghai"));
        // 转换为纽约时间:
        ZonedDateTime zny = zbj.withZoneSameInstant(ZoneId.of("America/New_York"));
        System.out.println(zbj);
        System.out.println(zny);
    }
}
```

▶ Run

要特别注意，时区转换的时候，由于夏令时的存在，不同的日期转换的结果很可能是不同的。这是北京时间9月15日的转换结果：

```
2019-09-15T21:05:50.187697+08:00[Asia/Shanghai]
2019-09-15T09:05:50.187697-04:00[America/New_York]
```

这是北京时间11月15日的转换结果：

```
2019-11-15T21:05:50.187697+08:00[Asia/Shanghai]
2019-11-15T08:05:50.187697-05:00[America/New_York]
```

两次转换后的纽约时间有1小时的夏令时时差。

涉及到时区时，千万不要自己计算时差，否则难以正确处理夏令时。

有了 `ZonedDateTime` ，将其转换为本地时间就非常简单：

```
1. ZonedDateTime zdt = ...
2. LocalDateTime ldt = zdt.toLocalDateTime();
```

转换为 `LocalDateTime` 时，直接丢弃了时区信息。

练习

某航线从北京飞到纽约需要13小时20分钟，请根据北京起飞日期和时间计算到达纽约的当地日期和时间。

```
import java.time.*;
public class Main {
    public static void main(String[] args) {
        LocalDateTime departureAtBeijing = LocalDateTime.of(2019, 9, 15, 13, 0, 0);
        int hours = 13;
        int minutes = 20;
        LocalDateTime arrivalAtNewYork = calculateArrivalAtNY(departureAtBeijing, hours, minutes);
        System.out.println(departureAtBeijing + " -> " + arrivalAtNewYork);
        // test:
        if (!LocalDateTime.of(2019, 10, 15, 14, 20, 0)
                .equals(calculateArrivalAtNY(LocalDateTime.of(2019, 10, 15, 13, 0, 0), 13, 20))) {
            System.err.println("测试失败!");
        } else if (!LocalDateTime.of(2019, 11, 15, 13, 20, 0)
                .equals(calculateArrivalAtNY(LocalDateTime.of(2019, 11, 15, 13, 0, 0), 13, 20))) {
            System.err.println("测试失败!");
        }
    }

    static LocalDateTime calculateArrivalAtNY(LocalDateTime bj, int h, int m) {
        return bj;
    }
}
```

▶ Run

提示： `ZonedDateTime` 仍然提供了 `plusDays()` 等加减操作。

下载练习：flight-time练习 （推荐使用IDE练习插件快速下载）

小结

`ZonedDateTime` 是带时区的日期和时间，可用于时区转换；

`ZonedDateTime` 和 `LocalDateTime` 可以相互转换。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





DateTimeFormatter

使用旧的 `Date` 对象时，我们用 `SimpleDateFormat` 进行格式化显示。使用新的 `LocalDateTime` 或 `ZonedLocalDateTime` 时，我们要进行格式化显示，就要使用 `DateTimeFormatter` 。

和 `SimpleDateFormat` 不同的是， `DateTimeFormatter` 不但是不变对象，它还是线程安全的。线程的概念我们会在后面涉及到。现在我们只需要记住：因为 `SimpleDateFormat` 不是线程安全的，使用的时候，只能在方法内部创建新的局部变量。而 `DateTimeFormatter` 可以只创建一个实例，到处引用。

创建 `DateTimeFormatter` 时，我们仍然通过传入格式化字符串实现：

```
1.  DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm");
```

格式化字符串的使用方式与 `SimpleDateFormat` 完全一致。

另一种创建 `DateTimeFormatter` 的方法是，传入格式化字符串时，同时指定 `Locale` ：

```
    DateTimeFormatter formatter = DateTimeFormatter.ofPattern("E, yyyy-MMMM-dd
1.  HH:mm", Locale.US);
```

这种方式可以按照 `Locale` 默认习惯格式化。我们来看实际效果：

```
import java.time.*;
import java.time.format.*;
import java.util.Locale;

public class Main {
    public static void main(String[] args) {
        ZonedDateTime zdt = ZonedDateTime.now();
        var formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm ZZZZ");
        System.out.println(formatter.format(zdt));

        var zhFormatter = DateTimeFormatter.ofPattern("yyyy MMM dd EE HH:mm", Locale.CHINA);
        System.out.println(zhFormatter.format(zdt));

        var usFormatter = DateTimeFormatter.ofPattern("E, MMMM/dd/yyyy HH:mm", Locale.US);
        System.out.println(usFormatter.format(zdt));
    }
}
```

▶ Run

在格式化字符串中，如果需要输出固定字符，可以用 `'xxx'` 表示。

运行上述代码，分别以默认方式、中国地区和美国地区对当前时间进行显示，结果如下：

```
2019-09-15T23:16 GMT+08:00
2019 9月 15 周日 23:16
Sun, September/15/2019 23:16
```

当我们直接调用 `System.out.println()` 对一个 `ZonedDateTime` 或者 `LocalDateTime` 实例进行打印的时候，实际上，调用的是它们的 `toString()` 方法，默认的 `toString()` 方法显示的字符串就是按照 `ISO 8601` 格式显示的，我们可以通过 `DateTimeFormatter` 预定义的几个静态变量来引用：

```
var ldt = LocalDateTime.now();
System.out.println(DateTimeFormatter.ISO_DATE.format(ldt));
System.out.println(DateTimeFormatter.ISO_DATE_TIME.format(ldt));
```

得到的输出和 `toString()` 类似：

```
2019-09-15
2019-09-15T23:16:51.56217
```

小结

对 `ZonedDateTime` 或 `LocalDateTime` 进行格式化，需要使用 `DateTimeFormatter` 类；

`DateTimeFormatter` 可以通过格式化字符串和 `Locale` 对日期和时间进行定制输出。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Instant

我们已经讲过，计算机存储的当前时间，本质上只是一个不断递增的整数。Java提供的 `System.currentTimeMillis()` 返回的就是以毫秒表示的当前时间戳。

这个当前时间戳在 `java.time` 中以 `Instant` 类型表示，我们用 `Instant.now()` 获取当前时间戳，效果和 `System.currentTimeMillis()` 类似：

```
import java.time.*;

public class Main {
    public static void main(String[] args) {
        Instant now = Instant.now();
        System.out.println(now.getEpochSecond()); // 秒
        System.out.println(now.toEpochMilli()); // 毫秒
    }
}
```

▶ Run

打印的结果类似：

```
1568568760
1568568760316
```

实际上， `Instant` 内部只有两个核心字段：

```
public final class Instant implements ... {
    private final long seconds;
    private final int nanos;
}
```

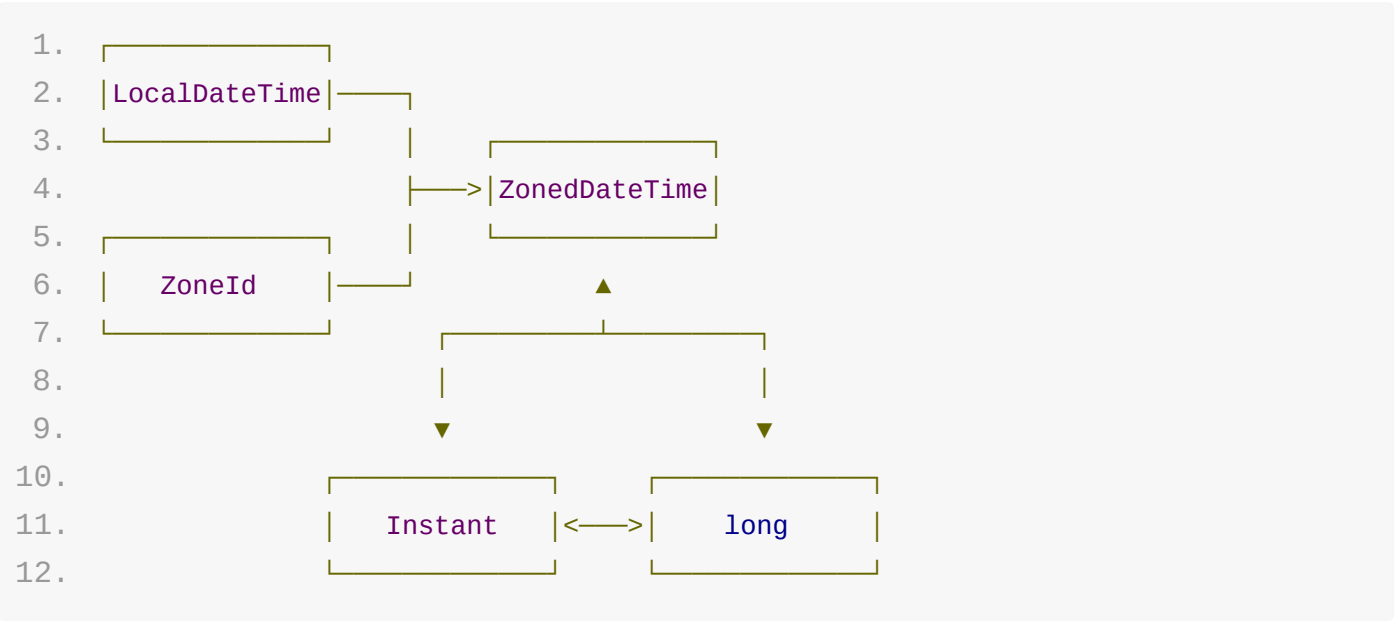
一个是以秒为单位的时间戳，一个是更精确的纳秒精度。它和 `System.currentTimeMillis()` 返回的 `long` 相比，只是多了更高精度的纳秒。

既然 `Instant` 就是时间戳，那么，给它附加上一个时区，就可以创建出 `ZonedDateTime` ：

```
// 以指定时间戳创建Instant:
Instant ins = Instant.ofEpochSecond(1568568760);
ZonedDateTime zdt = ins.atZone(ZoneId.systemDefault());
System.out.println(zdt); // 2019-09-16T01:32:40+08:00[Asia/Shanghai]
```

可见，对于某一个时间戳，给它关联上指定的 `ZoneId` ，就得到了 `ZonedDateTime` ，继而可以获得了对应时区的 `LocalDateTime` 。

所以， `LocalDateTime` ， `ZoneId` ， `Instant` ， `ZonedDateTime` 和 `long` 都可以互相转换：



转换的时候，只需要留意 `long` 类型以毫秒还是秒为单位即可。

小结

`Instant` 表示高精度时间戳，它可以和 `ZonedDateTime` 以及 `long` 互相转换。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





最佳实践

由于Java提供了新旧两套日期和时间的API，除非涉及到遗留代码，否则我们应该坚持使用新的API。

如果需要与遗留代码打交道，如何在新旧API之间互相转换呢？

旧API转新API

如果要把旧式的 `Date` 或 `Calendar` 转换为新API对象，可以通过 `toInstant()` 方法转换为 `Instant` 对象，再继续转换为 `ZonedDateTime` ：

```
// Date -> Instant:
Instant ins1 = new Date().toInstant();

// Calendar -> Instant -> ZonedDateTime:
Calendar calendar = Calendar.getInstance();
Instant ins2 = Calendar.getInstance().toInstant();
ZonedDateTime zdt = ins2.atZone(calendar.getTimeZone().toZoneId());
```

从上面的代码还可以看到，旧的 `TimeZone` 提供了一个 `toZoneId()` ，可以把自己变成新的 `ZoneId` 。

新API转旧API

如果要把新的 `ZonedDateTime` 转换为旧的API对象，只能借助 `long` 型时间戳做一个“中转”：

```
// ZonedDateTime -> long:
ZonedDateTime zdt = ZonedDateTime.now();
long ts = zdt.toEpochSecond() * 1000;

// long -> Date:
Date date = new Date(ts);

// long -> Calendar:
Calendar calendar = Calendar.getInstance();
calendar.clear();
calendar.setTimeZone(TimeZone.getTimeZone(zdt.getZone().getId()));
calendar.setTimeInMillis(zdt.toEpochSecond() * 1000);
```

从上面的代码还可以看到，新的 `ZoneId` 转换为旧的 `TimeZone` ，需要借助 `ZoneId.getId()` 返回的 `String` 完成。

在数据库中存储日期和时间

除了旧式的 `java.util.Date` ，我们还可以找到另一个 `java.sql.Date` ，它继承自 `java.util.Date` ，但会自动忽略所有时间相关信息。这个奇葩的设计原因要追溯到数据库的日期与时间类型。

在数据库中，也存在几种日期和时间类型：

- `DATETIME` ：表示日期和时间；
- `DATE` ：仅表示日期；
- `TIME` ：仅表示时间；
- `TIMESTAMP` ：和 `DATETIME` 类似，但是数据库会在创建或者更新记录的时候同时修改 `TIMESTAMP` 。

在使用Java程序操作数据库时，我们需要把数据库类型与Java类型映射起来。下表是数据库类型与Java新旧API的映射关系：

数据库	对应Java类（旧）	对应Java类（新）
DATETIME	java.util.Date	LocalDateTime
DATE	java.sql.Date	LocalDate
TIME	java.sql.Time	LocalTime
TIMESTAMP	java.sql.Timestamp	LocalDateTime

实际上，在数据库中，我们需要存储的最常用的是时刻（ `Instant` ），因为有了时刻信息，就可以根据用户自己选择的时区，显示出正确的本地时间。所以，最好的方法是直接用长整数 `long` 表示，在数据库中存储为 `BIGINT` 类型。

通过存储一个 `long` 型时间戳，我们可以编写一个 `timestampToString()` 的方法，非常简单地为不同用户以不同的偏好来显示不同的本地时间：

```
import java.time.*;
import java.time.format.*;
import java.util.Locale;
```

```
public class Main {
    public static void main(String[] args) {
        long ts = 1574208900000L;
        System.out.println(timestampToString(ts, Locale.CHINA, "Asia/Shanghai"));
        System.out.println(timestampToString(ts, Locale.US, "America/New_York"));
    }

    static String timestampToString(long epochMilli, Locale lo, String zoneId) {
        Instant ins = Instant.ofEpochMilli(epochMilli);
        DateTimeFormatter f = DateTimeFormatter.ofLocalizedDateTime(FormatStyle.MEDIUM,
FormatStyle.SHORT);
        return f.withLocale(lo).format(ZonedDateTime.ofInstant(ins, ZoneId.of(zoneId)));
    }
}
```

▶ Run

对上述方法进行调用，结果如下：

```
2019年11月20日 上午8:15
Nov 19, 2019, 7:15 PM
```

小结

处理日期和时间时，尽量使用新的 `java.time` 包；

在数据库中存储时间戳时，尽量使用 `long` 型时间戳，它具有省空间，效率高，不依赖数据库的优点。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





单元测试

本节我们介绍Java平台最常用的测试框架JUnit，并详细介绍如何编写单元测试。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





编写JUnit测试

什么是单元测试呢？单元测试就是针对最小的功能单元编写测试代码。Java程序最小的功能单元是方法，因此，对Java程序进行单元测试就是针对单个Java方法的测试。

单元测试有什么好处呢？在学习单元测试前，我们可以先了解一下测试驱动开发。

所谓测试驱动开发，是指先编写接口，紧接着编写测试。编写完测试后，我们才开始真正编写实现代码。在编写实现代码的过程中，一边写，一边测，什么时候测试全部通过了，那就表示编写的实现完成了：

```
    编写接口
     |
     ▼
    编写测试
     |
     ▼
┌─> 编写实现
|    |
| N  ▼
└── 运行测试
     | Y
     ▼
    任务完成
```

这就是传说中的……



当然，这是一种理想情况。大部分情况是我们已经编写了实现代码，需要对已有的代码进行测试。

我们先通过一个示例来看如何编写测试。假定我们编写了一个计算阶乘的类，它只有一个静态方法来计算阶乘：

$n! = 1 \times 2 \times 3 \times ... \times n$n!=1×2×3×…×n

代码如下：

```
public class Factorial {
```

```
    public static long fact(long n) {
        long r = 1;
        for (long i = 1; i <= n; i++) {
            r = r * i;
        }
        return r;
    }
}
```

要测试这个方法，一个很自然的想法是编写一个 `main()` 方法，然后运行一些测试代码：

```
public class Test {
    public static void main(String[] args) {
        if (fact(10) == 3628800) {
            System.out.println("pass");
        } else {
            System.out.println("fail");
        }
    }
}
```

这样我们就可以通过运行 `main()` 方法来运行测试代码。

不过，使用 `main()` 方法测试有很多缺点：

一是只能有一个 `main()` 方法，不能把测试代码分离，二是没有打印出测试结果和期望结果，例如， `expected: 3628800, but actual: 123456` ，三是很难编写一组通用的测试代码。

因此，我们需要一种测试框架，帮助我们编写测试。

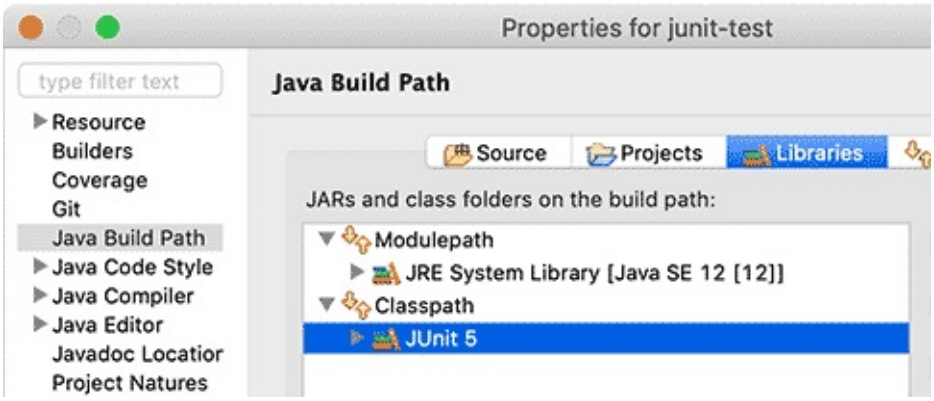
JUnit

JUnit是一个开源的Java语言的单元测试框架，专门针对Java设计，使用最广泛。JUnit是事实上的单元测试的标准框架，任何Java开发者都应当学习并使用JUnit编写单元测试。

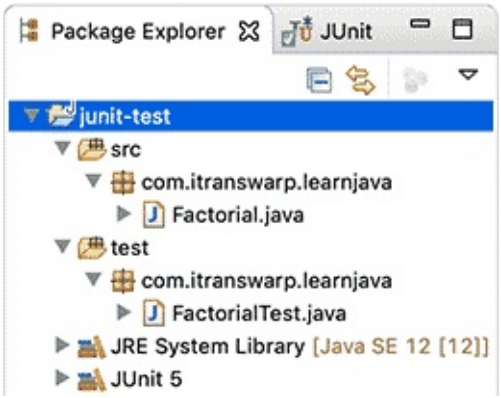
使用JUnit编写单元测试的好处在于，我们可以非常简单地组织测试代码，并随时运行它们，JUnit就会给出成功的测试和失败的测试，还可以生成测试报告，不仅包含测试的成功率，还可以统计测试的代码覆盖率，即被测试的代码本身有多少经过了测试。对于高质量的代码来说，测试覆盖率应该在80%以上。

此外，几乎所有的IDE工具都集成了JUnit，这样我们就可以直接在IDE中编写并运行JUnit测试。JUnit目前最新版本是5。

以Eclipse为例，当我们已经编写了一个 `Factorial.java` 文件后，我们想对其进行测试，需要编写一个对应的 `FactorialTest.java` 文件，以 `Test` 为后缀是一个惯例，并分别将其放入 `src` 和 `test` 目录中。最后，在 `Project` - `Properties` - `Java Build Path` - `Libraries` 中添加 `JUnit 5` 的库：



整个项目结构如下：



我们来看一下 `FactorialTest.java` 的内容：

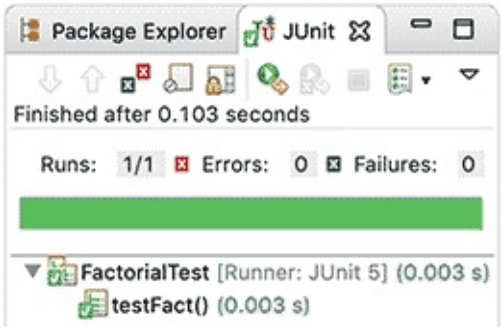
```
1.  package com.itranswarp.learnjava;
2.
3.  import static org.junit.jupiter.api.Assertions.*;
4.  import org.junit.jupiter.api.Test;
5.
6.  public class FactorialTest {
7.
8.      @Test
9.      void testFact() {
10.         assertEquals(1, Factorial.fact(1));
11.         assertEquals(2, Factorial.fact(2));
12.         assertEquals(6, Factorial.fact(3));
13.         assertEquals(3628800, Factorial.fact(10));
14.         assertEquals(2432902008176640000L, Factorial.fact(20));
15.     }
```

```
16. }
```

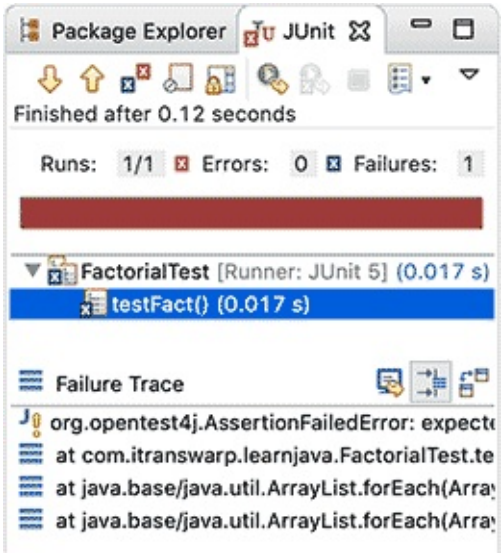
核心测试方法 `testFact()` 加上了 `@Test` 注解，这是JUnit要求的，它会把带有 `@Test` 的方法识别为测试方法。在测试方法内部，我们用 `assertEquals(1, Factorial.fact(1))` 表示，期望 `Factorial.fact(1)` 返回 `1` 。 `assertEquals(expected, actual)` 是最常用的测试方法，它在 `Assertion` 类中定义。 `Assertion` 还定义了其他断言方法，例如：

- `assertTrue()` ： 期待结果为 `true`
- `assertFalse()` ： 期待结果为 `false`
- `assertNotNull()` ： 期待结果为非 `null`
- `assertArrayEquals()` ： 期待结果为数组并与期望数组每个元素的值均相等
- …

运行单元测试非常简单。选中 `Factorial.java` 文件，点击 `Run` - `Run As` - `JUnit Test` ，Eclipse会自动运行这个JUnit测试，并显示结果：



如果测试结果与预期不符， `assertEquals()` 会抛出异常，我们就会得到一个测试失败的结果：



在Failure Trace中，JUnit会告诉我们详细的错误结果：

```
1. org.opentest4j.AssertionFailedError: expected: <3628800> but was: <362880>
2.     at org.junit.jupiter.api.AssertionUtils.fail(AssertionUtils.java:55)
```

```
    at org.junit.jupiter.api.AssertEquals.failNotEqual(AssertEquals.java:195)
    at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:168)
    at org.junit.jupiter.api.AssertEquals.assertEquals(AssertEquals.java:163)
    at org.junit.jupiter.api.Assertions.assertEquals(Assertions.java:611)
    at com.itranswarp.learnjava.FactorialTest.testFact(FactorialTest.java:14)
    at java.base/jdk.internal.reflect.NativeMethodAccessorImpl.invoke0(Native
Method)
    at ...
```

第一行的失败信息的意思是期待结果 `3628800` 但是实际返回是 `362880` ，此时，我们要么修正实现代码，要么修正测试代码，直到测试通过为止。

使用浮点数时，由于浮点数无法精确地进行比较，因此，我们需要调用 `assertEquals(double expected, double actual, double delta)` 这个重载方法，指定一个误差值：

```
assertEquals(0.1, Math.abs(1 - 9 / 10.0), 0.0000001);
```

单元测试的好处

单元测试可以确保单个方法按照正确预期运行，如果修改了某个方法的代码，只需确保其对应的单元测试通过，即可认为改动正确。此外，测试代码本身就可以作为示例代码，用来演示如何调用该方法。

使用JUnit进行单元测试，我们可以使用断言（ `Assertion` ）来测试期望结果，可以方便地组织和运行测试，并方便地查看测试结果。此外，JUnit既可以直接在IDE中运行，也可以方便地集成到Maven这些自动化工具中运行。

在编写单元测试的时候，我们要遵循一定的规范：

一是单元测试代码本身必须非常简单，能一下看明白，决不能再为测试代码编写测试；

二是每个单元测试应当互相独立，不依赖运行的顺序；

三是测试时不但要覆盖常用测试用例，还要特别注意测试边界条件，例如输入为 `0` ， `null` ，空字符串 `""` 等情况。

练习

下载练习：JUnit测试 （推荐使用IDE练习插件快速下载）

小结

JUnit是一个单元测试框架，专门用于运行我们编写的单元测试：

一个JUnit测试包含若干 `@Test` 方法，并使用 `Assertions` 进行断言，注意浮点数 `assertEquals()` 要指定 `delta` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Fixture

在一个单元测试中，我们经常编写多个 `@Test` 方法，来分组、分类对目标代码进行测试。

在测试的时候，我们经常遇到一个对象需要初始化，测试完可能还需要清理的情况。如果每个 `@Test` 方法都写一遍这样的重复代码，显然比较麻烦。

JUnit提供了编写测试前准备、测试后清理的固定代码，我们称之为Fixture。

我们来看一个具体的 `Calculator` 的例子：

```
1.  public class Calculator {
2.      private long n = 0;
3.
4.      public long add(long x) {
5.          n = n + x;
6.          return n;
7.      }
8.
9.      public long sub(long x) {
10.         n = n - x;
11.         return n;
12.     }
13. }
```

这个类的功能很简单，但是测试的时候，我们要先初始化对象，我们不必在每个测试方法中都写上初始化代码，而是通过 `@BeforeEach` 来初始化，通过 `@AfterEach` 来清理资源：

```
1.  public class CalculatorTest {
2.
3.      Calculator calculator;
4.
5.      @BeforeEach
6.      public void setUp() {
7.          this.calculator = new Calculator();
8.      }
9.
10.     @AfterEach
11.     public void tearDown() {
12.         this.calculator = null;
13.     }
14.
```

```
    @Test
    void testAdd() {
        assertEquals(100, this.calculator.add(100));
        assertEquals(150, this.calculator.add(50));
        assertEquals(130, this.calculator.add(-20));
    }

    @Test
    void testSub() {
        assertEquals(-100, this.calculator.sub(100));
        assertEquals(-150, this.calculator.sub(50));
        assertEquals(-130, this.calculator.sub(-20));
    }
}
```

在 `CalculatorTest` 测试中，有两个标记为 `@BeforeEach` 和 `@AfterEach` 的方法，它们会在运行每个 `@Test` 方法前后自动运行。

上面的测试代码在JUnit中运行顺序如下：

```
for (Method testMethod : findTestMethods(CalculatorTest.class)) {
    var test = new CalculatorTest(); // 创建Test实例
    invokeBeforeEach(test);
        invokeTestMethod(test, testMethod);
    invokeAfterEach(test);
}
```

可见， `@BeforeEach` 和 `@AfterEach` 会“环绕”在每个 `@Test` 方法前后。

还有一些资源初始化和清理可能更加繁琐，而且会耗费较长的时间，例如初始化数据库。JUnit还提供了 `@BeforeAll` 和 `@AfterAll` ，它们在运行所有@Test前后运行，顺序如下：

```
invokeBeforeAll(CalculatorTest.class);
for (Method testMethod : findTestMethods(CalculatorTest.class)) {
    var test = new CalculatorTest(); // 创建Test实例
    invokeBeforeEach(test);
        invokeTestMethod(test, testMethod);
    invokeAfterEach(test);
}
invokeAfterAll(CalculatorTest.class);
```

因为 `@BeforeAll` 和 `@AfterAll` 在所有 `@Test` 方法运行前后仅运行一次，因此，它们只能初始

化静态变量，例如：

```
public class DatabaseTest {
    static Database db;

    @BeforeAll
    public static void initDatabase() {
        db = createDb(...);
    }

    @AfterAll
    public static void dropDatabase() {
        ...
    }
}
```

事实上，`@BeforeAll` 和 `@AfterAll` 也只能标注在静态方法上。

因此，我们总结出编写Fixture的套路如下：

- 对于实例变量，在 `@BeforeEach` 中初始化，在 `@AfterEach` 中清理，它们在各个 `@Test` 方法中互不影响，因为是不同的实例；
- 对于静态变量，在 `@BeforeAll` 中初始化，在 `@AfterAll` 中清理，它们在各个 `@Test` 方法中均是唯一实例，会影响各个 `@Test` 方法。

大多数情况下，使用 `@BeforeEach` 和 `@AfterEach` 就足够了。只有某些测试资源初始化耗费时间太长，以至于我们不得不尽量“复用”时才会用到 `@BeforeAll` 和 `@AfterAll` 。

最后，注意到每次运行一个 `@Test` 方法前，JUnit首先创建一个 `XxxTest` 实例，因此，每个 `@Test` 方法内部的成员变量都是独立的，不能也无法把成员变量的状态从一个 `@Test` 方法带到另一个 `@Test` 方法。

练习

下载练习：使用Fixture （推荐使用IDE练习插件快速下载）

小结

编写Fixture是指针对每个 `@Test` 方法，编写 `@BeforeEach` 方法用于初始化测试资源，编写 `@AfterEach` 用于清理测试资源；

必要时，可以编写 `@BeforeAll` 和 `@AfterAll` ，使用静态变量来初始化耗时的资源，并且在所

有 `@Test` 方法的运行前后仅执行一次。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





异常测试

在Java程序中，异常处理是非常重要的。

我们自己编写的方法，也经常抛出各种异常。对于可能抛出的异常进行测试，本身就是测试的重要环节。

因此，在编写JUnit测试的时候，除了正常的输入输出，我们还要特别针对可能导致异常的情况进行测试。

我们仍然用 `Factorial` 举例：

```
1.  public class Factorial {
2.      public static long fact(long n) {
3.          if (n < 0) {
4.              throw new IllegalArgumentException();
5.          }
6.          long r = 1;
7.          for (long i = 1; i <= n; i++) {
8.              r = r * i;
9.          }
10.         return r;
11.     }
12. }
```

在方法入口，我们增加了对参数 `n` 的检查，如果为负数，则直接抛出 `IllegalArgumentException` 。

现在，我们希望对异常进行测试。在JUnit测试中，我们可以编写一个 `@Test` 方法专门测试异常：

```
1.  @Test
2.  void testNegative() {
3.      assertThrows(IllegalArgumentException.class, new Executable() {
4.          @Override
5.          public void execute() throws Throwable {
6.              Factorial.fact(-1);
7.          }
8.      });
9.  }
```

JUnit提供 `assertThrows()` 来期望捕获一个指定的异常。第二个参数 `Executable` 封装了我们要执行的会产生异常的代码。当我们执行 `Factorial.fact(-1)` 时，必定抛

出 `IllegalArgumentException` 。 `assertThrows()` 在捕获到指定异常时表示通过测试，未捕获到异常，或者捕获到的异常类型不对，均表示测试失败。

有些童鞋会觉得编写一个 `Executable` 的匿名类太繁琐了。实际上，Java 8开始引入了函数式编程，所有单方法接口都可以简写如下：

```
@Test
void testNegative() {
    assertThrows(IllegalArgumentException.class, () -> {
        Factorial.fact(-1);
    });
}
```

上述奇怪的 `->` 语法就是函数式接口的实现代码，我们会在后面详细介绍。现在，我们只需要通过这种固定的代码编写能抛出异常的语句即可。

练习

观察 `Factorial.fact()` 方法，注意到由于 `long` 型整数有范围限制，当我们传入参数 `21` 时，得到的结果是 `-4249290049419214848` ，而不是期望的 `51090942171709440000` ，因此，当传入参数大于 `20` 时， `Factorial.fact()` 方法应当抛出 `ArithmeticException` 。请编写测试并修改实现代码，确保测试通过。

下载练习：异常测试 （推荐使用IDE练习插件快速下载）

小结

测试异常可以使用 `assertThrows()` ，期待捕获到指定类型的异常；

对可能发生的每种类型的异常都必须进行测试。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





条件测试

在运行测试的时候，有些时候，我们需要排出某些 `@Test` 方法，不要让它运行，这时，我们就可以给它标记一个 `@Disabled` ：

```
@Disabled
@Test
void testBug101() {
    // 这个测试不会运行
}
```

为什么我们不直接注释掉 `@Test` ，而是要加一个 `@Disabled` ？这是因为注释掉 `@Test` ，JUnit就不知道这是个测试方法，而加上 `@Disabled` ，JUnit仍然识别出这是个测试方法，只是暂时不运行。它会在测试结果中显示：

```
Tests run: 68, Failures: 2, Errors: 0, Skipped: 5
```

类似 `@Disabled` 这种注解就称为条件测试，JUnit根据不同的条件注解，决定是否运行当前的 `@Test` 方法。

我们来看一个例子：

```
public class Config {
    public String getConfigFile(String filename) {
        String os = System.getProperty("os.name").toLowerCase();
        if (os.contains("win")) {
            return "C:\\" + filename;
        }
        if (os.contains("mac") || os.contains("linux") || os.contains("unix"))
{
            return "/usr/local/" + filename;
        }
        throw new UnsupportedOperationException();
    }
}
```

我们想要测试 `getConfigFile()` 这个方法，但是在Windows上跑，和在Linux上跑的代码路径不同，因此，针对两个系统的测试方法，其中一个只能在Windows上跑，另一个只能在Mac/Linux上跑：

```
@Test
```

```
    void testWindows() {
        assertEquals("C:\\test.ini", config.getConfigFile("test.ini"));
    }

    @Test
    void testLinuxAndMac() {
        assertEquals("/usr/local/test.cfg", config.getConfigFile("test.cfg"));
    }
```

因此，我们给上述两个测试方法分别加上条件如下：

```
@Test
@EnabledOnOs(OS.WINDOWS)
void testWindows() {
    assertEquals("C:\\test.ini", config.getConfigFile("test.ini"));
}

@Test
@EnabledOnOs({ OS.LINUX, OS.MAC })
void testLinuxAndMac() {
    assertEquals("/usr/local/test.cfg", config.getConfigFile("test.cfg"));
}
```

`@EnableOnOs` 就是一个条件测试判断。

我们来看一些常用的条件测试：

不在Windows平台执行的测试，可以加上 `@DisabledOnOs(OS.WINDOWS)` ：

```
@Test
@DisabledOnOs(OS.WINDOWS)
void testOnNonWindowsOs() {
    // TODO: this test is disabled on windows
}
```

只能在Java 9或更高版本执行的测试，可以加上 `@DisabledOnJre(JRE.JAVA_8)` ：

```
@Test
@DisabledOnJre(JRE.JAVA_8)
void testOnJava9OrAbove() {
    // TODO: this test is disabled on java 8
}
```

只能在64位操作系统上执行的测试，可以用 `@EnabledIfSystemProperty` 判断：

```
@Test
@EnabledIfSystemProperty(named = "os.arch", matches = ".*64.*")
void testOnlyOn64bitSystem() {
    // TODO: this test is only run on 64 bit system
}
```

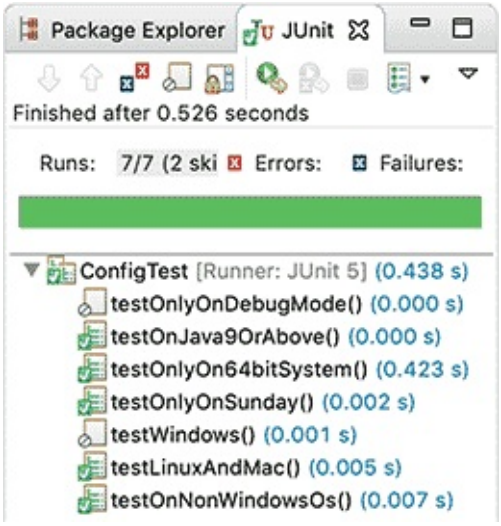
需要传入环境变量 `DEBUG=true` 才能执行的测试，可以用 `@EnabledIfEnvironmentVariable` ：

```
@Test
@EnabledIfEnvironmentVariable(named = "DEBUG", matches = "true")
void testOnlyOnDebugMode() {
    // TODO: this test is only run on DEBUG=true
}
```

最后，万能的 `@EnableIf` 可以执行任意Java语句并根据返回的 `boolean` 决定是否执行测试。下面的代码演示了一个只能在星期日执行的测试：

```
@Test
@EnabledIf("java.time.LocalDate.now().getDayOfWeek()==java.time.DayOfWeek.SUNDAY")
void testOnlyOnSunday() {
    // TODO: this test is only run on Sunday
}
```

当我们在JUnit中运行所有测试的时候，JUnit会给出执行的结果。在IDE中，我们能很容易地看到没有执行的测试：



带有⊘标记的测试方法表示没有执行。

练习

下载练习：条件测试 （推荐使用IDE练习插件快速下载）。

小结

条件测试是根据某些注解在运行期让JUnit自动忽略某些测试。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





参数化测试

如果待测试的输入和输出是一组数据：可以把测试数据组织起来用不同的测试数据调用相同的测试方法

参数化测试和普通测试稍微不同的地方在于，一个测试方法需要接收至少一个参数，然后，传入一组参数反复运行。

JUnit提供了一个 `@ParameterizedTest` 注解，用来进行参数化测试。

假设我们想对 `Math.abs()` 进行测试，先用一组正数进行测试：

```
@ParameterizedTest
@ValueSource(ints = { 0, 1, 5, 100 })
void testAbs(int x) {
    assertEquals(x, Math.abs(x));
}
```

再用一组负数进行测试：

```
@ParameterizedTest
@ValueSource(ints = { -1, -5, -100 })
void testAbsNegative(int x) {
    assertEquals(-x, Math.abs(x));
}
```

注意到参数化测试的注解是 `@ParameterizedTest` ，而不是普通的 `@Test` 。

实际的测试场景往往没有这么简单。假设我们自己编写了一个 `StringUtils.capitalize()` 方法，它会把字符串的第一个字母变为大写，后续字母变为小写：

```
public class StringUtils {
    public static String capitalize(String s) {
        if (s.length() == 0) {
            return s;
        }
        return Character.toUpperCase(s.charAt(0)) +
s.substring(1).toLowerCase();
    }
}
```

要用参数化测试的方法来测试，我们不但要给出输入，还要给出预期输出。因此，测试方法至少需要接收两个参数：

```
@ParameterizedTest
void testCapitalize(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}
```

现在问题来了：参数如何传入？

最简单的方法是通过 `@MethodSource` 注解，它允许我们编写一个同名的静态方法来提供测试参数：

```
@ParameterizedTest
@MethodSource
void testCapitalize(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}

static List<Arguments> testCapitalize() {
    return List.of( // arguments:
            Arguments.arguments("abc", "Abc"), //
            Arguments.arguments("APPLE", "Apple"), //
            Arguments.arguments("gooD", "Good"));
}
```

上面的代码很容易理解：静态方法 `testCapitalize()` 返回了一组测试参数，每个参数都包含两个 `String` ，正好作为测试方法的两个参数传入。

如果静态方法和测试方法的名称不同，@MethodSource也允许指定方法名。但使用默认同名方法最方便。

另一种传入测试参数的方法是使用 `@CsvSource` ，它的每一个字符串表示一行，一行包含的若干参数用 `,` 分隔，因此，上述测试又可以改写如下：

```
@ParameterizedTest
@CsvSource({ "abc, Abc", "APPLE, Apple", "gooD, Good" })
void testCapitalize(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}
```

如果有成百上千的测试输入，那么，直接写 `@CsvSource` 就很不方便。这个时候，我们可以把测试数据提到一个独立的CSV文件中，然后标注上 `@CsvFileSource` ：

```
@ParameterizedTest
@CsvFileSource(resources = { "/test-capitalize.csv" })
```

```
3.  void testCapitalizeUsingCsvFile(String input, String result) {
4.      assertEquals(result, StringUtils.capitalize(input));
5.  }
```

JUnit只在classpath中查找指定的CSV文件，因此，`test-capitalize.csv`这个文件要放到`test`目录下，内容如下：

```
1.  apple, Apple
2.  HELLO, Hello
3.  JUnit, Junit
4.  reSource, Resource
```

练习

下载练习：参数化测试StringUtils （推荐使用IDE练习插件快速下载）

小结

使用参数化测试，可以提供一组测试数据，对一个测试方法反复测试。

参数既可以在测试代码中写死，也可以通过`@CsvFileSource`放到外部的CSV文件中。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





正则表达式

正则表达式是一种用来匹配字符串的强有力的武器。Java内置了强大的正则表达式的支持。

本章我们会详细介绍如何在Java程序中使用正则表达式。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





正则表达式简介

在了解正则表达式之前，我们先看几个非常常见的问题：

- 如何判断字符串是否是有效的电话号码？例如：`010-1234567`，`123ABC456`，`13510001000`等；
- 如何判断字符串是否是有效的电子邮件地址？例如：`test@example.com`，`test#example`等；
- 如何判断字符串是否是有效的时间？例如：`12:34`，`09:60`，`99:99`等。

一种直观的想法是通过程序判断，这种方法需要为每种用例创建规则，然后用代码实现。下面是判断手机号的代码：

```
boolean isValidMobileNumber(String s) {
    // 是否是11位？
    if (s.length() != 11) {
        return false;
    }
    // 每一位都是0~9：
    for (int i=0; i<s.length(); i++) {
        char c = s.charAt(i);
        if (c < '0' || c > '9') {
            return false;
        }
    }
    return true;
}
```

上述代码仅仅做了非常粗略的判断，并未考虑首位数字不能为`0`等更详细的情况。

除了判断手机号，我们还需要判断电子邮件地址、电话、邮编等等：

- `boolean isValidMobileNumber(String s) { … }`
- `boolean isValidEmail(String s) { … }`
- `boolean isValidPhoneNumber(String s) { … }`
- `boolean isValidZipCode(String s) { … }`
- …

为每一种判断逻辑编写代码实在是太繁琐了。有没有更简单的方法？

有！用正则表达式！

正则表达式可以用字符串来描述规则，并用来匹配字符串。例如，判断手机号，我们用正则表达式 `\d{11}` ：

```
boolean isValidMobileNumber(String s) {
    return s.matches("\\d{11}");
}
```

使用正则表达式的好处有哪些？一个正则表达式就是一个描述规则的字符串，所以，只需要编写正确的规则，我们就可以让正则表达式引擎去判断目标字符串是否符合规则。

正则表达式是一套标准，它可以用于任何语言。Java标准库的 `java.util.regex` 包内置了正则表达式引擎，在Java程序中使用正则表达式非常简单。

举个例子：要判断用户输入的年份是否是 `20##` 年，我们先写出规则如下：

一共有4个字符，分别是： `2` ， `0` ， `0~9任意数字` ， `0~9任意数字` 。

对应的正则表达式就是： `20\d\d` ，其中 `\d` 表示任意一个数字。

把正则表达式转换为Java字符串就变成了 `20\d\d` ，注意Java字符串用 `\` 表示 `\` 。

最后，用正则表达式匹配一个字符串的代码如下：

```
// regex
public class Main {
    public static void main(String[] args) {
        String regex = "20\\d\\d";
        System.out.println("2019".matches(regex)); // true
        System.out.println("2100".matches(regex)); // false
    }
}
```

▶ Run

可见，使用正则表达式，不必编写复杂的代码来判断，只需给出一个字符串表达的正则规则即可。

小结

正则表达式是用字符串描述的一个匹配规则，使用正则表达式可以快速判断给定的字符串是否符合匹配规则。Java标准库 `java.util.regex` 内建了正则表达式引擎。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





匹配规则

正则表达式的匹配规则是从左到右按规则匹配。我们首先来看如何使用正则表达式来做精确匹配。

对于正则表达式 `abc` 来说，它只能精确地匹配字符串 `"abc"` ，不能匹配 `"ab"` ， `"Abc"` ， `"abcd"` 等其他任何字符串。

如果正则表达式有特殊字符，那就需要用 `\` 转义。例如，正则表达式 `a\&c` ，其中 `\&` 是用来匹配特殊字符 `&` 的，它能精确匹配字符串 `"a&c"` ，但不能匹配 `"ac"` 、 `"a-c"` 、 `"a&&c"` 等。

要注意正则表达式在Java代码中也是一个字符串，所以，对于正则表达式 `a\&c` 来说，对应的Java字符串是 `"a\&c"` ，因为 `\` 也是Java字符串的转义字符，两个 `\` 实际上表示的是一个 `\` ：

```
// regex
public class Main {
    public static void main(String[] args) {
        String re1 = "abc";
        System.out.println("abc".matches(re1));
        System.out.println("Abc".matches(re1));
        System.out.println("abcd".matches(re1));

        String re2 = "a\\&c"; // 对应的正则是a\&c
        System.out.println("a&c".matches(re2));
        System.out.println("a-c".matches(re2));
        System.out.println("a&&c".matches(re2));
    }
}
```

▶ Run

如果想匹配非ASCII字符，例如中文，那就用 `\u####` 的十六进制表示，例如： `a\u548cc` 匹配字符串 `"a和c"` ，中文字符 `和` 的Unicode编码是 `548c` 。

匹配任意字符

精确匹配实际上用处不大，因为我们直接用 `String.equals()` 就可以做到。大多数情况下，我们想要的匹配规则更多的是模糊匹配。我们可以用 `.` 匹配一个任意字符。

例如，正则表达式 `a.c` 中间的 `.` 可以匹配一个任意字符，例如，下面的字符串都可以被匹配：

- `"abc"` ，因为 `.` 可以匹配字符 `b` ；
- `"a&c"` ，因为 `.` 可以匹配字符 `&` ；
- `"acc"` ，因为 `.` 可以匹配字符 `c` 。

但它不能匹配 `"ac"` 、 `"a&&c"` ，因为 `.` 匹配一个字符且仅限一个字符。

匹配数字

用 `.` 可以匹配任意字符，这个口子开得有点大。如果我们只想匹配 `0` ~ `9` 这样的数字，可以用 `\d` 匹配。例如，正则表达式 `00\d` 可以匹配：

- `"007"`，因为 `\d` 可以匹配字符 `7`；
- `"008"`，因为 `\d` 可以匹配字符 `8`。

它不能匹配 `"00A"`，`"0077"`，因为 `\d` 仅限单个数字字符。

匹配常用字符

用 `\w` 可以匹配一个字母、数字或下划线，w的意思是word。例如，`java\w` 可以匹配：

- `"javac"`，因为 `\w` 可以匹配英文字符 `c`；
- `"java9"`，因为 `\w` 可以匹配数字字符 `9`；。
- `"java”`，因为 `\w` 可以匹配下划线 `_`。

它不能匹配 `"java#"`，`"java "`，因为 `\w` 不能匹配 `#`、空格等字符。

匹配空格字符

用 `\s` 可以匹配一个空格字符，注意空格字符不但包括空格 ` `，还包括tab字符（在Java中用 `\t` 表示）。例如，`a\sc` 可以匹配：

- `"a c"`，因为 `\s` 可以匹配空格字符 ` `；
- `"a c"`，因为 `\s` 可以匹配tab字符 `\t`。

它不能匹配 `"ac"`，`"abc"` 等。

匹配非数字

用 `\d` 可以匹配一个数字，而 `\D` 则匹配一个非数字。例如，`00\D` 可以匹配：

- `"00A"`，因为 `\D` 可以匹配非数字字符 `A`；
- `"00#"`，因为 `\D` 可以匹配非数字字符 `#`。

`00\d` 可以匹配的字符串 `"007"`，`"008"` 等，`00\D` 是不能匹配的。

类似的，`\W` 可以匹配 `\w` 不能匹配的字符，`\S` 可以匹配 `\s` 不能匹配的字符，这几个正好是反着来的。

```
// regex
public class Main {
    public static void main(String[] args) {
        String re1 = "java\\d"; // 对应的正则是java\d
        System.out.println("java9".matches(re1));
        System.out.println("java10".matches(re1));
        System.out.println("javac".matches(re1));

        String re2 = "java\\D";
        System.out.println("javax".matches(re2));
        System.out.println("java#".matches(re2));
        System.out.println("java5".matches(re2));
    }
}
```

▶ Run

重复匹配

我们用 `\d` 可以匹配一个数字，例如， `A\d` 可以匹配 `"A0"` ， `"A1"` ，如果要匹配多个数字，比如 `"A380"` ，怎么办？

修饰符 `*` 可以匹配任意个字符，包括0个字符。我们用 `A\d*` 可以匹配：

- `A` ：因为 `\d*` 可以匹配0个数字；
- `A0` ：因为 `\d*` 可以匹配1个数字 `0` ；
- `A380` ：因为 `\d*` 可以匹配多个数字 `380` 。

修饰符 `+` 可以匹配至少一个字符。我们用 `A\d+` 可以匹配：

- `A0` ：因为 `\d+` 可以匹配1个数字 `0` ；
- `A380` ：因为 `\d+` 可以匹配多个数字 `380` 。

但它无法匹配 `"A"` ，因为修饰符 `+` 要求至少一个字符。

修饰符 `?` 可以匹配0个或一个字符。我们用 `A\d?` 可以匹配：

- `A` ：因为 `\d?` 可以匹配0个数字；
- `A0` ：因为 `\d+` 可以匹配1个数字 `0` 。

但它无法匹配 `"A33"` ，因为修饰符 `?` 超过1个字符就不能匹配了。

如果我们想精确指定n个字符怎么办？用修饰符 `{n}` 就可以。 `A\d{3}` 可以精确匹配：

- `A380` ：因为 `\d{3}` 可以匹配3个数字 `380` 。

如果我们想指定匹配n~m个字符怎么办？用修饰符 `{n,m}` 就可以。 `A\d{3,5}` 可以精确匹配：

- `A380`：因为`\d{3,5}`可以匹配3个数字`380`；
- `A3800`：因为`\d{3,5}`可以匹配4个数字`3800`；
- `A38000`：因为`\d{3,5}`可以匹配5个数字`38000`。

如果没有上限，那么修饰符`{n,}`就可以匹配至少n个字符。

练习

请编写一个正则表达式匹配国内的电话号码规则：3~4位区号加7~8位电话，中间用`-`连接，例如：`010-12345678`。

```
// regex
import java.util.*;

public class Main {
    public static void main(String[] args) throws Exception {
        String re = "\\d";
        for (String s : List.of("010-12345678", "020-9999999", "0755-7654321")) {
            if (!s.matches(re)) {
                System.out.println("测试失败: " + s);
                return;
            }
        }
        for (String s : List.of("010 12345678", "A20-9999999", "0755-7654.321")) {
            if (s.matches(re)) {
                System.out.println("测试失败: " + s);
                return;
            }
        }
        System.out.println("测试成功!");
    }
}
```

▶ Run

下载练习：电话匹配练习 （推荐使用IDE练习插件快速下载）

进阶：国内区号必须以0开头，而电话号码不能以0开头，试修改正则表达式，使之能更精确地匹配。

提示：`\d`和`\D`这种简单的规则暂时做不到，我们需要更复杂规则，后面会详细讲解。

小结

单个字符的匹配规则如下：

正则表达式	规则	可以匹配
`A`	指定字符	`A`
`\u548c`	指定Unicode字符	`和`

.	任意字符	a，b，&，0
\d	数字0~9	0 ~ 9
\w	大小写字母，数字和下划线	a ~ z，A ~ Z，0 ~ 9，_
\s	空格、Tab键	空格，Tab
\D	非数字	a，A，&，_，……
\W	非\w	&，@，中，……
\S	非\s	a，A，&，_，……

多个字符的匹配规则如下：

正则表达式	规则	可以匹配
A*	任意个数字符	空，A，AA，AAA，……
A+	至少1个字符	A，AA，AAA，……
A?	0个或1个字符	空，A
A{3}	指定个数字符	AAA
A{2,3}	指定范围个数字符	AA，AAA
A{2,}	至少n个字符	AA，AAA，AAAA，……
A{0,3}	最多n个字符	空，A，AA，AAA

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





复杂匹配规则

匹配开头和结尾

用正则表达式进行多行匹配时，我们用 `^` 表示开头， `$` 表示结尾。例如， `^A\d{3}$` ，可以匹配 `"A001"` 、 `"A380"` 。

匹配指定范围

如果我们规定一个7~8位数字的电话号码不能以 `0` 开头，应该怎么写匹配规则呢？ `\d{7,8}` 是不行的，因为第一个 `\d` 可以匹配到 `0` 。

使用 `[…]` 可以匹配范围内的字符，例如， `[123456789]` 可以匹配 `1` ~ `9` ，这样就可以写出上述电话号码的规则： `[123456789]\d{6,7}` 。

把所有字符全列出来太麻烦， `[…]` 还有一种写法，直接写 `[1-9]` 就可以。

要匹配大小写不限的十六进制数，比如 `1A2b3c` ，我们可以这样写： `[0-9a-fA-F]` ，它表示一共可以匹配以下任意范围的字符：

- `0-9` ：字符 `0` ~ `9` ；
- `a-f` ：字符 `a` ~ `f` ；
- `A-F` ：字符 `A` ~ `F` 。

如果要匹配6位十六进制数，前面讲过的 `{n}` 仍然可以继续配合使用： `[0-9a-fA-F]{6}` 。

`[…]` 还有一种排除法，即不包含指定范围的字符。假设我们要匹配任意字符，但不包括数字，可以写 `[^1-9]{3}` ：

- 可以匹配 `"ABC"` ，因为不包含字符 `1` ~ `9` ；
- 可以匹配 `"A00"` ，因为不包含字符 `1` ~ `9` ；
- 不能匹配 `"A01"` ，因为包含字符 `1` ；
- 不能匹配 `"A05"` ，因为包含字符 `5` 。

或规则匹配

用 `|` 连接的两个正则规则是或规则，例如， `AB|CD` 表示可以匹配 `AB` 或 `CD` 。

我们来看这个正则表达式 `java|php` ：

```
// regex
public class Main {
    public static void main(String[] args) {
        String re = "java|php";
        System.out.println("java".matches(re));
        System.out.println("php".matches(re));
        System.out.println("go".matches(re));
    }
}
```

▶ Run

它可以匹配 `"java"` 或 `"php"` ，但无法匹配 `"go"` 。

要把 `go` 也加进来匹配，可以改写为 `java|php|go` 。

使用括号

现在我们想要匹配字符串 `learn java` 、 `learn php` 和 `learn go` 怎么办？一个最简单的规则是 `learn\sjava|learn\sphp|learn\sgo` ，但是这个规则太复杂了，可以把公共部分提出来，然后用 `(…)` 把子规则括起来表示成 `learn\s(java|php|go)` 。

```
// regex
public class Main {
    public static void main(String[] args) {
        String re = "learn\\s(java|php|go)";
        System.out.println("learn java".matches(re));
        System.out.println("learn Java".matches(re));
        System.out.println("learn php".matches(re));
        System.out.println("learn Go".matches(re));
    }
}
```

▶ Run

上面的规则仍然不能匹配 `learn Java` 、 `learn Go` 这样的字符串。试修改正则，使之能匹配大写字母开头的 `learn Java` 、 `learn Php` 、 `learn Go` 。

小结

复杂匹配规则主要有：

正则表达式	规则	可以匹配
^	开头	字符串开头
$	结尾	字符串结束

[ABC]	[…]内任意字符	A，B，C
[A-F0-9xy]	指定范围的字符	`A`，……，`F`，`0`，……，`9`，`x`，`y`
[^A-F]	指定范围外的任意字符	非`A`~`F`
AB|CD|EF	AB或CD或EF	`AB`，`CD`，`EF`

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





分组匹配

我们前面讲到的 `(…)` 可以用来把一个子规则括起来，这样写 `learn\s(java|php|go)` 就可以更方便地匹配长字符串了。

实际上 `(…)` 还有一个重要作用，就是分组匹配。

我们来看一下如何用正则匹配 `区号-电话号` 码这个规则。利用前面讲到的匹配规则，写出来很容易：

```
1.  \d{3,4}\-\d{6,8}
```

虽然这个正则匹配规则很简单，但是往往匹配成功后，下一步是提取区号和电话号码，分别存入数据库。于是问题来了：如何提取匹配的子串？

当然可以用 `String` 提供的 `indexOf()` 和 `substring()` 这些方法，但它们从正则匹配的字符串中提取子串没有通用性，下一次要提取 `learn\s(java|php)` 还得改代码。

正确的方法是用 `(…)` 先把要提取的规则分组，把上述正则表达式变为 `(\d{3,4})-(\d{6,8})` 。

现在问题又来了：匹配后，如何按括号提取子串？

现在我们没办法用 `String.matches()` 这样简单的判断方法了，必须引入 `java.util.regex` 包，用 `Pattern` 对象匹配，匹配后获得一个 `Matcher` 对象，如果匹配成功，就可以直接从 `Matcher.group(index)` 返回子串：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        Pattern p = Pattern.compile("(\\d{3,4})\\-(\\d{7,8})");
        Matcher m = p.matcher("010-12345678");
        if (m.matches()) {
            String g1 = m.group(1);
            String g2 = m.group(2);
            System.out.println(g1);
            System.out.println(g2);
        } else {
            System.out.println("匹配失败!");
        }
    }
}
```

▶ Run

运行上述代码，会得到两个匹配上的子串 `010` 和 `12345678` 。

要特别注意， `Matcher.group(index)` 方法的参数用1表示第一个子串，2表示第二个子串。如果我

们传入0会得到什么呢？答案是 `010-12345678` ，即整个正则匹配到的字符串。

Pattern

我们在前面的代码中用到的正则表达式代码是 `String.matches()` 方法，而我们在分组提取的代码中用的是 `java.util.regex` 包里面的 `Pattern` 类和 `Matcher` 类。实际上这两种代码本质上是一样的，因为 `String.matches()` 方法内部调用的就是 `Pattern` 和 `Matcher` 类的方法。

但是反复使用 `String.matches()` 对同一个正则表达式进行多次匹配效率较低，因为每次都会创建出一样的 `Pattern` 对象。完全可以先创建出一个 `Pattern` 对象，然后反复使用，就可以实现编译一次，多次匹配：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("(\\d{3,4})\\-(\\d{7,8})");
        pattern.matcher("010-12345678").matches(); // true
        pattern.matcher("021-123456").matches(); // true
        pattern.matcher("022#1234567").matches(); // false
        // 获得Matcher对象:
        Matcher matcher = pattern.matcher("010-12345678");
        if (matcher.matches()) {
            String whole = matcher.group(0); // "010-12345678", 0表示匹配的整个字符串
            String area = matcher.group(1); // "010", 1表示匹配的第1个子串
            String tel = matcher.group(2); // "12345678", 2表示匹配的第2个子串
            System.out.println(area);
            System.out.println(tel);
        }
    }
}
```

▶ Run

使用 `Matcher` 时，必须首先调用 `matches()` 判断是否匹配成功，匹配成功后，才能调用 `group()` 提取子串。

利用提取子串的功能，我们轻松获得了区号和号码两部分。

练习

利用分组匹配，从字符串 `"23:01:59"` 提取时、分、秒。

下载练习：分组匹配 （推荐使用IDE练习插件快速下载）

小结

正则表达式用 `(…)` 分组可以通过 `Matcher` 对象快速提取子串：

- `group(0)` 表示匹配的整个字符串；
- `group(1)` 表示第1个子串，`group(2)` 表示第2个子串，以此类推。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





非贪婪匹配

在介绍非贪婪匹配前，我们先看一个简单的问题：

给定一个字符串表示的数字，判断该数字末尾 `0` 的个数。例如：

- `"123000"`：3个 `0`
- `"10100"`：2个 `0`
- `"1001"`：0个 `0`

可以很容易地写出该正则表达式：`(\d+)(0*)`，Java代码如下：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("(\\d+)(0*)");
        Matcher matcher = pattern.matcher("1230000");
        if (matcher.matches()) {
            System.out.println("group1=" + matcher.group(1)); // "1230000"
            System.out.println("group2=" + matcher.group(2)); // ""
        }
    }
}
```

▶ Run

然而打印的第二个子串是空字符串 `""`。

实际上，我们期望分组匹配结果是：

input	`\d+`	`0*`
123000	"123"	"000"
10100	"101"	"00"
1001	"1001"	""

但实际的分组匹配结果是这样的：

input	`\d+`	`0*`
123000	"123000"	""
10100	"10100"	""
1001	"1001"	""

仔细观察上述实际匹配结果，实际上它是完全合理的，因为 `\d+` 确实可以匹配后面任意个 `0`。

这是因为正则表达式默认使用贪婪匹配：任何一个规则，它总是尽可能多地向后匹配，因此，`\d+` 总

是会把后面的 `0` 包含进来。

要让 `\d+` 尽量少匹配，让 `0*` 尽量多匹配，我们就必须让 `\d+` 使用非贪婪匹配。在规则 `\d+` 后面加个 `?` 即可表示非贪婪匹配。我们改写正则表达式如下：

```
import java.util.regex.*;

public class Main {
    public static void main(String[] args) {
        Pattern pattern = Pattern.compile("(\\d+?)(0*)");
        Matcher matcher = pattern.matcher("1230000");
        if (matcher.matches()) {
            System.out.println("group1=" + matcher.group(1)); // "123"
            System.out.println("group2=" + matcher.group(2)); // "0000"
        }
    }
}
```

▶ Run

因此，给定一个匹配规则，加上 `?` 后就变成了非贪婪匹配。

我们再来看这个正则表达式 `(\d??)(9*)` ，注意 `\d?` 表示匹配0个或1个数字，后面第二个 `?` 表示非贪婪匹配，因此，给定字符串 `"9999"` ，匹配到的两个子串分别是 `""` 和 `"9999"` ，因为对于 `\d?` 来说，可以匹配1个 `9` ，也可以匹配0个 `9` ，但是因为后面的 `?` 表示非贪婪匹配，它就会尽可能少的匹配，结果是匹配了0个 `9` 。

小结

正则表达式匹配默认使用贪婪匹配，可以使用 `?` 表示对某一规则进行非贪婪匹配。

注意区分 `?` 的含义： `\d??` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





搜索和替换

分割字符串

使用正则表达式分割字符串可以实现更加灵活的功能。`String.split()`方法传入的正是正则表达式。我们来看下面的代码：

```
"a b c".split("\\s"); // { "a", "b", "c" }
"a b  c".split("\\s"); // { "a", "b", "", "c" }
"a, b ;; c".split("[\\,\\;\\s]+"); // { "a", "b", "c" }
```

如果我们想让用户输入一组标签，然后把标签提取出来，因为用户的输入往往是不规范的，这时，使用合适的正则表达式，就可以消除多个空格、混合`,`和`;`这些不规范的输入，直接提取出规范的字符串。

搜索字符串

使用正则表达式还可以搜索字符串，我们来看例子：

```
import java.util.regex.*;
public class Main {
    public static void main(String[] args) {
        String s = "the quick brown fox jumps over the lazy dog.";
        Pattern p = Pattern.compile("\\wo\\w");
        Matcher m = p.matcher(s);
        while (m.find()) {
            String sub = s.substring(m.start(), m.end());
            System.out.println(sub);
        }
    }
}
```

▶ Run

我们获取到`Matcher`对象后，不需要调用`matches()`方法（因为匹配整个串肯定返回false），而是反复调用`find()`方法，在整个串中搜索能匹配上`\wo\w`规则的子串，并打印出来。这种方式比`String.indexOf()`要灵活得多，因为我们搜索的规则是3个字符：中间必须是`o`，前后两个必须是字符`[A-Za-z0-9_]`。

替换字符串

使用正则表达式替换字符串可以直接调用`String.replaceAll()`，它的第一个参数是正则表达式，第二个参数是待替换的字符串。我们还是来看例子：

```
// regex
public class Main {
    public static void main(String[] args) {
        String s = "The     quick\t\t brown   fox  jumps   over the  lazy dog.";
        String r = s.replaceAll("\\s+", " ");
        System.out.println(r); // "The quick brown fox jumps over the lazy dog."
    }
}
```

▶ Run

上面的代码把不规范的连续空格分隔的句子变成了规范的句子。可见，灵活使用正则表达式可以大大降低代码量。

反向引用

如果我们要把搜索到的指定字符串按规则替换，比如前后各加一个 `<b>xxxx</b>` ，这个时候，使用 `replaceAll()` 的时候，我们传入的第二个参数可以使用 `$1` 、 `$2` 来反向引用匹配到的子串。例如：

```
// regex
public class Main {
    public static void main(String[] args) {
        String s = "the quick brown fox jumps over the lazy dog.";
        String r = s.replaceAll("\\s([a-z]{4})\\s", " <b>$1</b> ");
        System.out.println(r);
    }
}
```

▶ Run

上述代码的运行结果是：

```
1.  the quick brown fox jumps <b>over</b> the <b>lazy</b> dog.
```

它实际上把任何4字符单词的前后用 `<b>xxxx</b>` 括起来。实现替换的关键就在于 `" <b>$1</b> "` ，它用匹配的分组子串 `([a-z]{4})` 替换了 `$1` 。

练习

模板引擎是指，定义一个字符串作为模板：

```
Hello, ${name}! You are learning ${lang}!
```

其中，以 `${key}` 表示的是变量，也就是将要被替换的内容

当传入一个 `Map<String, String>` 给模板后，需要把对应的key替换为Map的value。

例如，传入 `Map` 为：

```
{
    "name": "Bob",
    "lang": "Java"
}
```

然后， `${name}` 被替换为 `Map` 对应的值"Bob”， `${lang}` 被替换为 `Map` 对应的值"Java"，最终输出的结果为：

```
Hello, Bob! You are learning Java!
```

请编写一个简单的模板引擎，利用正则表达式实现这个功能。

提示：参考Matcher.appendReplacement())方法。

下载练习：模板引擎练习 （推荐使用IDE练习插件快速下载）

小结

使用正则表达式可以：

- 分割字符串：`String.split()`
- 搜索子串：`Matcher.find()`
- 替换字符串：`String.replaceAll()`

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





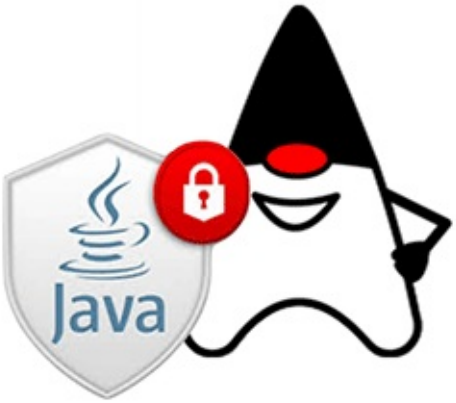
加密与安全

在计算机系统中，什么是加密与安全呢？

我们举个栗子：假设Bob要给Alice发一封邮件，在邮件传送的过程中，黑客可能会窃取到邮件的内容，所以需要防窃听。黑客还可能会篡改邮件的内容，Alice必须有能力识别出邮件有没有被篡改。最后，黑客可能假冒Bob给Alice发邮件，Alice必须有能力识别出伪造的邮件。

所以，应对潜在的安全威胁，需要做到三防：

- 防窃听
- 防篡改
- 防伪造



计算机加密技术就是为了实现上述目标，而现代计算机密码学理论是建立在严格的数学理论基础上的，密码学已经逐渐发展成一门科学。对于绝大多数开发者来说，设计一个安全的加密算法非常困难，验证一个加密算法是否安全更加困难，当前被认为安全的加密算法仅仅是迄今为止尚未被攻破。因此，要编写安全的计算机程序，我们要做到：

- 不要自己设计山寨的加密算法；
- 不要自己实现已有的加密算法；
- 不要自己修改已有的加密算法。

本章我们会介绍最常用的加密算法，以及如何通过Java代码实现。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





编码算法

要学习编码算法，我们先来看一看什么是编码。

ASCII码就是一种编码，字母 `A` 的编码是十六进制的 `0x41` ，字母 `B` 是 `0x42` ，以此类推：

字母	ASCII编码
A	0x41
B	0x42
C	0x43
D	0x44
…	…

因为ASCII编码最多只能有127个字符，要想对更多的文字进行编码，就需要用Unicode。而中文的中使用Unicode编码就是 `0x4e2d` ，使用UTF-8则需要3个字节编码：

汉字	Unicode编码	UTF-8编码
中	0x4e2d	0xe4b8ad
文	0x6587	0xe69687
编	0x7f16	0xe7bc96
码	0x7801	0xe7a081
…	…	…

因此，最简单的编码是直接给每个字符指定一个若干字节表示的整数，复杂一点的编码就需要根据一个已有的编码推算出来。

比如UTF-8编码，它是一种不定长编码，但可以从给定字符的Unicode编码推算出来。

URL编码

URL编码是浏览器发送数据给服务器时使用的编码，它通常附加在URL的参数部分，例如：

https://www.baidu.com/s?wd=%E4%B8%AD%E6%96%87

之所以需要URL编码，是因为出于兼容性考虑，很多服务器只识别ASCII字符。但如果URL中包含中文、日文这些非ASCII字符怎么办？不要紧，URL编码有一套规则：

- 如果字符是 `A` ~ `Z` ， `a` ~ `z` ， `0` ~ `9` 以及 `-` 、 `_` 、 `.` 、 `*` ，则保持不变；
- 如果是其他字符，先转换为UTF-8编码，然后对每个字节以 `%XX` 表示。

例如：字符 `中` 的UTF-8编码是 `0xe4b8ad` ，因此，它的URL编码是 `%E4%B8%AD` 。URL编码总是大写。

Java标准库提供了一个 `URLEncoder` 类来对任意字符串进行URL编码：

```
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;

public class Main {
    public static void main(String[] args) {
        String encoded = URLEncoder.encode("中文!", StandardCharsets.UTF_8);
        System.out.println(encoded);
    }
}
```

▶ Run

上述代码的运行结果是 `%E4%B8%AD%E6%96%87%21` ， `中` 的URL编码是 `%E4%B8%AD` ， `文` 的URL编码是 `%E6%96%87` ， `!` 虽然是ASCII字符，也要对其编码为 `%21` 。

和标准的URL编码稍有不同，URLEncoder把空格字符编码成 `+` ，而现在的URL编码标准要求空格被编码为 `%20` ，不过，服务器都可以处理这两种情况。

如果服务器收到URL编码的字符串，就可以对其进行解码，还原成原始字符串。Java标准库的 `URLDecoder` 就可以解码：

```
import java.net.URLDecoder;
import java.nio.charset.StandardCharsets;

public class Main {
    public static void main(String[] args) {
        String decoded = URLDecoder.decode("%E4%B8%AD%E6%96%87%21", StandardCharsets.UTF_8);
        System.out.println(decoded);
    }
}
```

▶ Run

要特别注意：URL编码是编码算法，不是加密算法。URL编码的目的是把任意文本数据编码为 `%` 前缀表示的文本，编码后的文本仅包含 `A` ~ `Z` ， `a` ~ `z` ， `0` ~ `9` ， `-` ， `_` ， `.` ， `*` 和 `%` ，便于浏览器和服务器处理。

Base64编码

URL编码是对字符进行编码，表示成 `%xx` 的形式，而Base64编码是对二进制数据进行编码，表示成

文本格式。

Base64编码可以把任意长度的二进制数据变为纯文本，且只包含 `A` ~ `Z` 、 `a` ~ `z` 、 `0` ~ `9` 、 `+` 、 `/` 、 `=` 这些字符。它的原理是把3字节的二进制数据按6bit一组，用4个int整数表示，然后查表，把int整数用索引对应到字符，得到编码后的字符串。

举个例子：3个byte数据分别是 `e4` 、 `b8` 、 `ad` ，按6bit分组得到 `39` 、 `0b` 、 `22` 和 `2d` ：

```
┌──────────┬──────────┬──────────┐
│    e4    │    b8    │    ad    │
└──────────┴──────────┴──────────┘
┌─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┬─┐
|1|1|1|0|0|1|0|0|1|0|1|1|1|0|0|0|1|0|1|0|1|1|0|1|
└─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┴─┘
┌──────────┬──────────┬──────────┬──────────┐
│    39    │    0b    │    22    │    2d    │
└──────────┴──────────┴──────────┴──────────┘
```

因为6位整数的范围总是 `0` ~ `63` ，所以，能用64个字符表示：字符 `A` ~ `Z` 对应索引 `0` ~ `25` ，字符 `a` ~ `z` 对应索引 `26` ~ `51` ，字符 `0` ~ `9` 对应索引 `52` ~ `61` ，最后两个索引 `62` 、 `63` 分别用字符 `+` 和 `/` 表示。

在Java中，二进制数据就是 `byte[]` 数组。Java标准库提供了 `Base64` 来对 `byte[]` 数组进行编解码：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        byte[] input = new byte[] { (byte) 0xe4, (byte) 0xb8, (byte) 0xad };
        String b64encoded = Base64.getEncoder().encodeToString(input);
        System.out.println(b64encoded);
    }
}
```

▶ Run

编码后得到 `5Lit` 4个字符。要对 `Base64` 解码，仍然用 `Base64` 这个类：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        byte[] output = Base64.getDecoder().decode("5Lit");
        System.out.println(Arrays.toString(output)); // [-28, -72, -83]
    }
}
```

▶ Run

有的童鞋会问：如果输入的 `byte[]` 数组长度不是3的整数倍肿么办？这种情况下，需要对输入的末尾补一个或两个 `0x00` ，编码后，在结尾加一个 `=` 表示补充了1个 `0x00` ，加两个 `=` 表示补充了2个 `0x00` ，解码的时候，去掉末尾补充的一个或两个 `0x00` 即可。

实际上，因为编码后的长度加上 `=` 总是4的倍数，所以即使不加 `=` 也可以计算出原始输入的 `byte[]` 。Base64编码的时候可以用 `withoutPadding()` 去掉 `=` ，解码出来的结果是一样的：

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        byte[] input = new byte[] { (byte) 0xe4, (byte) 0xb8, (byte) 0xad, 0x21 };
        String b64encoded = Base64.getEncoder().encodeToString(input);
        String b64encoded2 = Base64.getEncoder().withoutPadding().encodeToString(input);
        System.out.println(b64encoded);
        System.out.println(b64encoded2);
        byte[] output = Base64.getDecoder().decode(b64encoded2);
        System.out.println(Arrays.toString(output));
    }
}
```

▶ Run

因为标准的Base64编码会出现 `+` 、 `/` 和 `=` ，所以不适合把Base64编码后的字符串放到URL中。一种针对URL的Base64编码可以在URL中使用的Base64编码，它仅仅是把 `+` 变成 `-` ， `/` 变成 `_` ：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        byte[] input = new byte[] { 0x01, 0x02, 0x7f, 0x00 };
        String b64encoded = Base64.getUrlEncoder().encodeToString(input);
        System.out.println(b64encoded);
        byte[] output = Base64.getUrlDecoder().decode(b64encoded);
        System.out.println(Arrays.toString(output));
    }
}
```

▶ Run

Base64编码的目的是把二进制数据变成文本格式，这样在很多文本中就可以处理二进制数据。例如，电子邮件协议就是文本协议，如果要在电子邮件中添加一个二进制文件，就可以用Base64编码，然后以文本的形式传送。

Base64编码的缺点是传输效率会降低，因为它把原始数据的长度增加了1/3。

和URL编码一样，Base64编码是一种编码算法，不是加密算法。

如果把Base64的64个字符编码表换成32个、48个或者58个，就可以使用Base32编码，Base48编码和Base58编码。字符越少，编码的效率就会越低。

小结

URL编码和Base64编码都是编码算法，它们不是加密算法；

URL编码的目的是把任意文本数据编码为%前缀表示的文本，便于浏览器和服务器处理；

Base64编码的目的是把任意二进制数据编码为文本，但编码后数据量会增加1/3。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





哈希算法

哈希算法（Hash）又称摘要算法（Digest），它的作用是：对任意一组输入数据进行计算，得到一个固定长度的输出摘要。

哈希算法最重要的特点就是：

- 相同的输入一定得到相同的输出；
- 不同的输入大概率得到不同的输出。

哈希算法的目的就是为了验证原始数据是否被篡改。

Java字符串的 `hashCode()` 就是一个哈希算法，它的输入是任意字符串，输出是固定的4字节 `int` 整数：

```
"hello".hashCode(); // 0x5e918d2
"hello, java".hashCode(); // 0x7a9d88e8
"hello, bob".hashCode(); // 0xa0dbae2f
```

两个相同的字符串永远会计算出相同的 `hashCode` ，否则基于 `hashCode` 定位的 `HashMap` 就无法正常工作。这也是为什么当我们自定义一个class时，覆写 `equals()` 方法时我们必须正确覆写 `hashCode()` 方法。

哈希碰撞

哈希碰撞是指，两个不同的输入得到了相同的输出：

```
"AaAaAa".hashCode(); // 0x7460e8c0
"BBAaBB".hashCode(); // 0x7460e8c0
```

有童鞋会问：碰撞能不能避免？答案是不能。碰撞是一定会出现的，因为输出的字节长度是固定的， `String` 的 `hashCode()` 输出是4字节整数，最多只有4294967296种输出，但输入的数据长度是不固定的，有无数种输入。所以，哈希算法是把一个无限的输入集合映射到一个有限的输出集合，必然会产生碰撞。

碰撞不可怕，我们担心的不是碰撞，而是碰撞的概率，因为碰撞概率的高低关系到哈希算法的安全性。一个安全的哈希算法必须满足：

- 碰撞概率低；
- 不能猜测输出。

不能猜测输出是指，输入的任意一个bit的变化会造成输出完全不同，这样就很难从输出反推输入（只

能依靠暴力穷举）。假设一种哈希算法有如下规律：

```
1. hashA("java001") = "123456"
2. hashA("java002") = "123457"
3. hashA("java003") = "123458"
```

那么很容易从输出 `123459` 反推输入，这种哈希算法就不安全。安全的哈希算法从输出是看不出任何规律的：

```
1. hashB("java001") = "123456"
2. hashB("java002") = "580271"
3. hashB("java003") = ???
```

常用的哈希算法有：

算法	输出长度（位）	输出长度（字节）
MD5	128 bits	16 bytes
SHA-1	160 bits	20 bytes
RipeMD-160	160 bits	20 bytes
SHA-256	256 bits	32 bytes
SHA-512	512 bits	64 bytes

根据碰撞概率，哈希算法的输出长度越长，就越难产生碰撞，也就越安全。

Java标准库提供了常用的哈希算法，并且有一套统一的接口。我们以MD5算法为例，看看如何对输入计算哈希：

```
import java.math.BigInteger;
import java.security.MessageDigest;

public class Main {
    public static void main(String[] args) throws Exception {
        // 创建一个MessageDigest实例:
        MessageDigest md = MessageDigest.getInstance("MD5");
        // 反复调用update输入数据:
        md.update("Hello".getBytes("UTF-8"));
        md.update("World".getBytes("UTF-8"));
        byte[] result = md.digest(); // 16 bytes: 68e109f0f40ca72a15e05cc22786f8e6
        System.out.println(new BigInteger(1, result).toString(16));
    }
}
```

▶ Run

使用 `MessageDigest` 时，我们首先根据哈希算法获取一个 `MessageDigest` 实例，然后，反复调

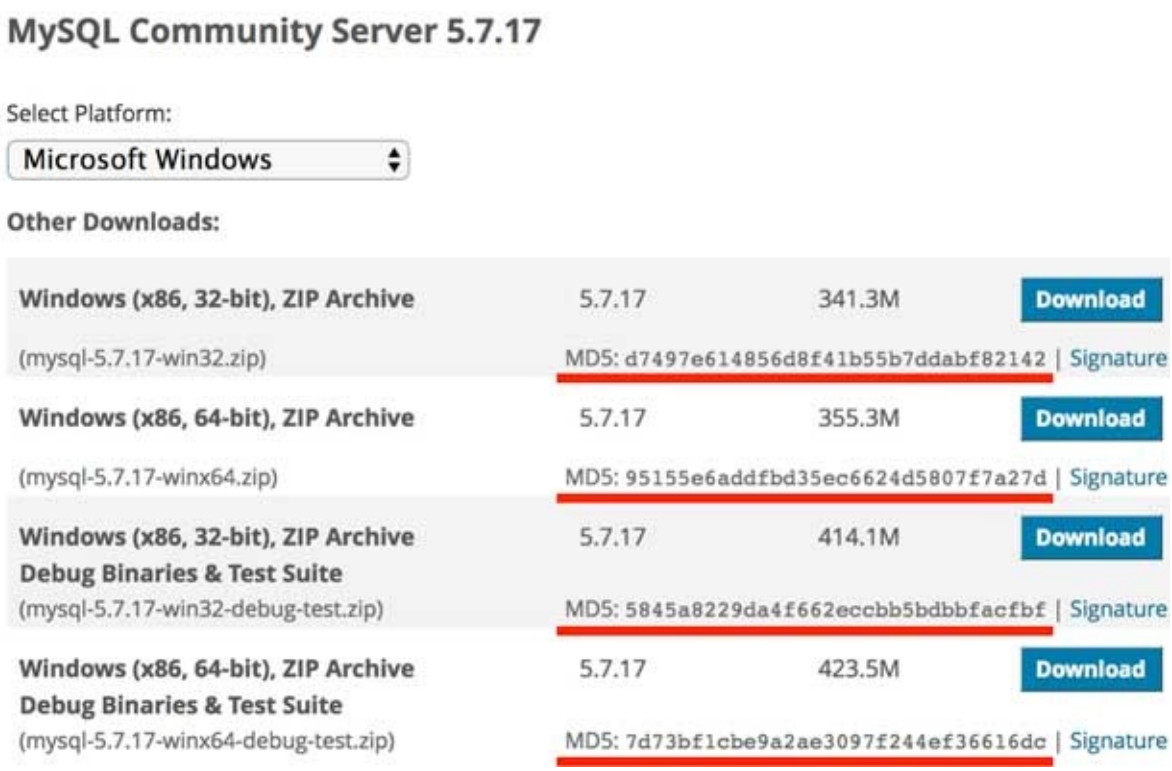
用 `update(byte[])` 输入数据。当输入结束后，调用 `digest()` 方法获得byte[]数组表示的摘要，最后，把它转换为十六进制的字符串。

运行上述代码，可以得到输入 `HelloWorld` 的MD5是 `68e109f0f40ca72a15e05cc22786f8e6` 。

哈希算法的用途

因为相同的输入永远会得到相同的输出，因此，如果输入被修改了，得到的输出就会不同。

我们在网站上下载软件的时候，经常看到下载页显示的哈希：



如何判断下载到本地的软件是原始的、未经篡改的文件？我们只需要自己计算一下本地文件的哈希值，再与官网公开的哈希值对比，如果相同，说明文件下载正确，否则，说明文件已被篡改。

哈希算法的另一个重要用途是存储用户口令。如果直接将用户的原始口令存放到数据库中，会产生极大的安全风险：

- 数据库管理员能够看到用户明文口令；
- 数据库数据一旦泄漏，黑客即可获取用户明文口令。

不存储用户的原始口令，那么如何对用户进行认证？

方法是存储用户口令的哈希，例如，MD5。

在用户输入原始口令后，系统计算用户输入的原始口令的MD5并与数据库存储的MD5对比，如果一致，说明口令正确，否则，口令错误。

因此，数据库存储用户名和口令的表内容应该像下面这样：

username	password
bob	f30aa7a662c728b7407c54ae6bfd27d1
alice	25d55ad283aa400af464c76d713c07ad
tim	5f4dcc3b5aa765d61d8327deb882cf99

这样一来，数据库管理员看不到用户的原始口令。即使数据库泄漏，黑客也无法拿到用户的原始口令。想要拿到用户的原始口令，必须用暴力穷举的方法，一个口令一个口令地试，直到某个口令计算的MD5恰好等于指定值。

使用哈希口令时，还要注意防止彩虹表攻击。

什么是彩虹表呢？上面讲到了，如果只拿到MD5，从MD5反推明文口令，只能使用暴力穷举的方法。

然而黑客并不笨，暴力穷举会消耗大量的算力和时间。但是，如果有一个预先计算好的常用口令和它们的MD5的对照表：

常用口令	MD5
hello123	f30aa7a662c728b7407c54ae6bfd27d1
12345678	25d55ad283aa400af464c76d713c07ad
passw0rd	bed128365216c019988915ed3add75fb
19700101	570da6d5277a646f6552b8832012f5dc
…	…
20201231	6879c0ae9117b50074ce0a0d4c843060

这个表就是彩虹表。如果用户使用了常用口令，黑客从MD5一下就能反查到原始口令：

bob的MD5：`f30aa7a662c728b7407c54ae6bfd27d1`，原始口令：`hello123`；

alice的MD5：`25d55ad283aa400af464c76d713c07ad`，原始口令：`12345678`；

tim的MD5：`bed128365216c019988915ed3add75fb`，原始口令：`passw0rd`。

这就是为什么不要使用常用密码，以及不要使用生日作为密码的原因。

即使用户使用了常用口令，我们也可以采取措施来抵御彩虹表攻击，方法是对每个口令额外添加随机数，这个方法称之为加盐（salt）：

```
1.  digest = md5(salt+inputPassword)
```

经过加盐处理的数据库表，内容如下：

username	salt	password

bob	H1r0a	a5022319ff4c56955e22a74abcc2c210
alice	7$p2w	e5de688c99e961ed6e560b972dab8b6a
tim	z5Sk9	1eee304b92dc0d105904e7ab58fd2f64

加盐的目的在于使黑客的彩虹表失效，即使用户使用常用口令，也无法从MD5反推原始口令。

SHA-1

SHA-1也是一种哈希算法，它的输出是160 bits，即20字节。SHA-1是由美国国家安全局开发的，SHA算法实际上是一个系列，包括SHA-0（已废弃）、SHA-1、SHA-256、SHA-512等。

在Java中使用SHA-1，和MD5完全一样，只需要把算法名称改为 `"SHA-1"` ：

```
import java.math.BigInteger;
import java.security.MessageDigest;

public class Main {
    public static void main(String[] args) throws Exception {
        // 创建一个MessageDigest实例:
        MessageDigest md = MessageDigest.getInstance("SHA-1");
        // 反复调用update输入数据:
        md.update("Hello".getBytes("UTF-8"));
        md.update("World".getBytes("UTF-8"));
        byte[] result = md.digest(); // 20 bytes: 6f44e49f848dd8ed27f73f59ab5bd4631b3f6b0d
        System.out.println(new BigInteger(1, result).toString(16));
    }
}
```

▶ Run

类似的，计算SHA-256，我们需要传入名称 `"SHA-256"` ，计算SHA-512，我们需要传入名称 `"SHA-512"` 。Java标准库支持的所有哈希算法可以在这里查到。

注意：MD5因为输出长度较短，短时间内破解是可能的，目前已经不推荐使用。

小结

哈希算法可用于验证数据完整性，具有防篡改检测的功能；

常用的哈希算法有MD5、SHA-1等；

用哈希存储口令时要考虑彩虹表攻击。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





BouncyCastle

我们知道，Java标准库提供了一系列常用的哈希算法。

但如果我们要用的某种算法，Java标准库没有提供怎么办？

方法一：自己写一个，难度很大；

方法二：找一个现成的第三方库，直接使用。

BouncyCastle就是一个提供了很多哈希算法和加密算法的第三方库。它提供了Java标准库没有的一些算法，例如，RipeMD160哈希算法。

我们来看一下如何使用BouncyCastle这个第三方提供的算法。

首先，我们必须把BouncyCastle提供的jar包放到classpath中。这个jar包就是 `bcprov-jdk15on-xxx.jar` ，可以从官方网站下载。

Java标准库的 `java.security` 包提供了一种标准机制，允许第三方提供商无缝接入。我们要使用BouncyCastle提供的RipeMD160算法，需要先把BouncyCastle注册一下：

```
1.  public class Main {
2.      public static void main(String[] args) throws Exception {
3.          // 注册BouncyCastle:
4.          Security.addProvider(new BouncyCastleProvider());
5.          // 按名称正常调用:
6.          MessageDigest md = MessageDigest.getInstance("RipeMD160");
7.          md.update("HelloWorld".getBytes("UTF-8"));
8.          byte[] result = md.digest();
9.          System.out.println(new BigInteger(1, result).toString(16));
10.     }
11. }
```

其中，注册BouncyCastle是通过下面的语句实现的：

```
1.  Security.addProvider(new BouncyCastleProvider());
```

注册只需要在启动时进行一次，后续就可以使用BouncyCastle提供的所有哈希算法和加密算法。

练习

下载练习：使用BouncyCastle提供的RipeMD160 （推荐使用IDE练习插件快速下载）

小结

BouncyCastle是一个开源的第三方算法提供商；

BouncyCastle提供了很多Java标准库没有提供的哈希算法和加密算法；

使用第三方算法前需要通过 `Security.addProvider()` 注册。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Hmac算法

在前面讲到哈希算法时，我们说，存储用户的哈希口令时，要加盐存储，目的就在于抵御彩虹表攻击。

我们回顾一下哈希算法：

```
digest = hash(input)
```

正是因为相同的输入会产生相同的输出，我们加盐的目的就在于，使得输入有所变化：

```
digest = hash(salt + input)
```

这个salt可以看作是一个额外的“认证码”，同样的输入，不同的认证码，会产生不同的输出。因此，要验证输出的哈希，必须同时提供“认证码”。

Hmac算法就是一种基于密钥的消息认证码算法，它的全称是Hash-based Message Authentication Code，是一种更安全的消息摘要算法。

Hmac算法总是和某种哈希算法配合起来用的。例如，我们使用MD5算法，对应的就是HmacMD5算法，它相当于“加盐”的MD5：

```
HmacMD5 ≈ md5(secure_random_key, input)
```

因此，HmacMD5可以看作带有一个安全的key的MD5。使用HmacMD5而不是用MD5加salt，有如下好处：

- HmacMD5使用的key长度是64字节，更安全；
- Hmac是标准算法，同样适用于SHA-1等其他哈希算法；
- Hmac输出和原有的哈希算法长度一致。

可见，Hmac本质上就是把key混入摘要的算法。验证此哈希时，除了原始的输入数据，还要提供key。

为了保证安全，我们不会自己指定key，而是通过Java标准库的KeyGenerator生成一个安全的随机的key。下面是使用HmacMD5的代码：

```
import java.math.BigInteger;
import javax.crypto.*;

public class Main {
    public static void main(String[] args) throws Exception {
        KeyGenerator keyGen = KeyGenerator.getInstance("HmacMD5");
        SecretKey key = keyGen.generateKey();
        // 打印随机生成的key:
        byte[] skey = key.getEncoded();
        System.out.println(new BigInteger(1, skey).toString(16));
        Mac mac = Mac.getInstance("HmacMD5");
        mac.init(key);
        mac.update("HelloWorld".getBytes("UTF-8"));
        byte[] result = mac.doFinal();
        System.out.println(new BigInteger(1, result).toString(16));
    }
}
```

▶ Run

和MD5相比，使用HmacMD5的步骤是：

- 通过名称 `HmacMD5` 获取 `KeyGenerator` 实例；
- 通过 `KeyGenerator` 创建一个 `SecretKey` 实例；
- 通过名称 `HmacMD5` 获取 `Mac` 实例；
- 用 `SecretKey` 初始化 `Mac` 实例；
- 对 `Mac` 实例反复调用 `update(byte[])` 输入数据；
- 调用 `Mac` 实例的 `doFinal()` 获取最终的哈希值。我们可以用Hmac算法取代原有的自定义的加盐算法，因此，存储用户名和口令的数据库结构如下：

username	secret_key (64 bytes)	password
bob	a8c06e05f92e…5e16	7e0387872a57c85ef6dddbaa12f376de
alice	e6a343693985…f4be	c1f929ac2552642b302e739bc0cdbaac
tim	f27a973dfdc0…6003	af57651c3a8a73303515804d4af43790

有了Hmac计算的哈希和 `SecretKey` ，我们想要验证怎么办？这时， `SecretKey` 不能从 `KeyGenerator` 生成，而是从一个 `byte[]` 数组恢复：

```
import java.util.Arrays;
import javax.crypto.*;
import javax.crypto.spec.*;
```

```
public class Main {
    public static void main(String[] args) throws Exception {
        byte[] hkey = new byte[] { 106, 70, -110, 125, 39, -20, 52, 56, 85, 9, -19, -72, 52, -53, 52,
-45, -6, 119, -63,
                30, 20, -83, -28, 77, 98, 109, -32, -76, 121, -106, 0, -74, -107, -114, -45, 104,
-104, -8, 2, 121, 6,
                97, -18, -13, -63, -30, -125, -103, -80, -46, 113, -14, 68, 32, -46, 101, -116, -104,
-81, -108, 122,
                89, -106, -109 };

        SecretKey key = new SecretKeySpec(hkey, "HmacMD5");
        Mac mac = Mac.getInstance("HmacMD5");
        mac.init(key);
        mac.update("HelloWorld".getBytes("UTF-8"));
        byte[] result = mac.doFinal();
        System.out.println(Arrays.toString(result));
        // [126, 59, 37, 63, 73, 90, 111, -96, -77, 15, 82, -74, 122, -55, -67, 54]
    }
}
```

▶ Run

恢复 `SecretKey` 的语句就是 `new SecretKeySpec(hkey, "HmacMD5")` 。

小结

Hmac算法是一种标准的基于密钥的哈希算法，可以配合MD5、SHA-1等哈希算法，计算的摘要长度和原摘要算法长度相同。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





对称加密算法

对称加密算法就是传统的用一个密码进行加密和解密。例如，我们常用的WinZIP和WinRAR对压缩包的加密和解密，就是使用对称加密算法：



从程序的角度看，所谓加密，就是这样一个函数，它接收密码和明文，然后输出密文：

```
secret = encrypt(key, message);
```

而解密则相反，它接收密码和密文，然后输出明文：

```
plain = decrypt(key, secret);
```

在软件开发中，常用的对称加密算法有：

算法	密钥长度	工作模式	填充模式
DES	56/64	ECB/CBC/PCBC/CTR/…	NoPadding/PKCS5Padding/…
AES	128/192/256	ECB/CBC/PCBC/CTR/…	NoPadding/PKCS5Padding/PKCS7Padding/…
IDEA	128	ECB	PKCS5Padding/PKCS7Padding/…

密钥长度直接决定加密强度，而工作模式和填充模式可以看成是对称加密算法的参数和格式选择。Java标准库提供的算法实现并不包括所有的工作模式和所有填充模式，但是通常我们只需要挑选常用的使用就可以了。

最后注意，DES算法由于密钥过短，可以在短时间内被暴力破解，所以现在已经不安全了。

使用AES加密

AES算法是目前应用最广泛的加密算法。我们先用ECB模式加密并解密：

```
import java.security.*;
import java.util.Base64;

import javax.crypto.*;
import javax.crypto.spec.*;

public class Main {
    public static void main(String[] args) throws Exception {
        // 原文:
        String message = "Hello, world!";
        System.out.println("Message: " + message);
        // 128位密钥 = 16 bytes Key:
        byte[] key = "1234567890abcdef".getBytes("UTF-8");
        // 加密:
        byte[] data = message.getBytes("UTF-8");
        byte[] encrypted = encrypt(key, data);
        System.out.println("Encrypted: " + Base64.getEncoder().encodeToString(encrypted));
        // 解密:
        byte[] decrypted = decrypt(key, encrypted);
        System.out.println("Decrypted: " + new String(decrypted, "UTF-8"));
    }

    // 加密:
    public static byte[] encrypt(byte[] key, byte[] input) throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        SecretKey keySpec = new SecretKeySpec(key, "AES");
        cipher.init(Cipher.ENCRYPT_MODE, keySpec);
        return cipher.doFinal(input);
    }

    // 解密:
    public static byte[] decrypt(byte[] key, byte[] input) throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("AES/ECB/PKCS5Padding");
        SecretKey keySpec = new SecretKeySpec(key, "AES");
        cipher.init(Cipher.DECRYPT_MODE, keySpec);
        return cipher.doFinal(input);
    }
}
```

▶ Run

Java标准库提供的对称加密接口非常简单，使用时按以下步骤编写代码：

- 根据算法名称/工作模式/填充模式获取Cipher实例；
- 根据算法名称初始化一个SecretKey实例，密钥必须是指定长度；
- 使用SerectKey初始化Cipher实例，并设置加密或解密模式；
- 传入明文或密文，获得密文或明文。ECB模式是最简单的AES加密模式，它只需要一个固定长度的密钥，固定的明文会生成固定的密文，这种一对一的加密方式会导致安全性降低，更好的方式是通过CBC模式，它需要一个随机数作为IV参数，这样对于同一份明文，每次生成的密文都不同：

```
import java.security.*;
import java.util.Base64;
import javax.crypto.*;
import javax.crypto.spec.*;
```

```
public class Main {
    public static void main(String[] args) throws Exception {
        // 原文:
        String message = "Hello, world!";
        System.out.println("Message: " + message);
        // 256位密钥 = 32 bytes Key:
        byte[] key = "1234567890abcdef1234567890abcdef".getBytes("UTF-8");
        // 加密:
        byte[] data = message.getBytes("UTF-8");
        byte[] encrypted = encrypt(key, data);
        System.out.println("Encrypted: " + Base64.getEncoder().encodeToString(encrypted));
        // 解密:
        byte[] decrypted = decrypt(key, encrypted);
        System.out.println("Decrypted: " + new String(decrypted, "UTF-8"));
    }

    // 加密:
    public static byte[] encrypt(byte[] key, byte[] input) throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        SecretKeySpec keySpec = new SecretKeySpec(key, "AES");
        // CBC模式需要生成一个16 bytes的initialization vector:
        SecureRandom sr = SecureRandom.getInstanceStrong();
        byte[] iv = sr.generateSeed(16);
        IvParameterSpec ivps = new IvParameterSpec(iv);
        cipher.init(Cipher.ENCRYPT_MODE, keySpec, ivps);
        byte[] data = cipher.doFinal(input);
        // IV不需要保密，把IV和密文一起返回:
        return join(iv, data);
    }

    // 解密:
    public static byte[] decrypt(byte[] key, byte[] input) throws GeneralSecurityException {
        // 把input分割成IV和密文:
        byte[] iv = new byte[16];
        byte[] data = new byte[input.length - 16];
        System.arraycopy(input, 0, iv, 0, 16);
        System.arraycopy(input, 16, data, 0, data.length);
        // 解密:
        Cipher cipher = Cipher.getInstance("AES/CBC/PKCS5Padding");
        SecretKeySpec keySpec = new SecretKeySpec(key, "AES");
        IvParameterSpec ivps = new IvParameterSpec(iv);
        cipher.init(Cipher.DECRYPT_MODE, keySpec, ivps);
        return cipher.doFinal(data);
    }

    public static byte[] join(byte[] bs1, byte[] bs2) {
        byte[] r = new byte[bs1.length + bs2.length];
        System.arraycopy(bs1, 0, r, 0, bs1.length);
        System.arraycopy(bs2, 0, r, bs1.length, bs2.length);
        return r;
    }
}
```

▶ Run

在CBC模式下，需要一个随机生成的16字节IV参数，必须使用 `SecureRandom` 生成。因为多了一个 `IvParameterSpec` 实例，因此，初始化方法需要调用 `Cipher` 的一个重载方法并传入 `IvParameterSpec` 。

观察输出，可以发现每次生成的IV不同，密文也不同。

小结

对称加密算法使用同一个密钥进行加密和解密，常用算法有DES、AES和IDEA等；

密钥长度由算法设计决定，AES的密钥长度是128/192/256位；

使用对称加密算法需要指定算法名称、工作模式和填充模式。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





口令加密算法

上一节我们讲的AES加密，细心的童鞋可能会发现，密钥长度是固定的128/192/256位，而不是我们用WinZip/WinRAR那样，随便输入几位都可以。

这是因为对称加密算法决定了口令必须是固定长度，然后对明文进行分块加密。又因为安全需求，口令长度往往都是128位以上，即至少16个字符。

但是我们平时使用的加密软件，输入6位、8位都可以，难道加密方式不一样？

实际上用户输入的口令并不能直接作为AES的密钥进行加密（除非长度恰好是128/192/256位），并且用户输入的口令一般都有规律，安全性远远不如安全随机数产生的随机口令。因此，用户输入的口令，通常还需要使用PBE算法，采用随机数杂凑计算出真正的密钥，再进行加密。

PBE就是Password Based Encryption的缩写，它的作用如下：

```
key = generate(userPassword, secureRandomPassword);
```

PBE的作用就是把用户输入的口令和一个安全随机的口令采用杂凑后计算出真正的密钥。以AES密钥为例，我们让用户输入一个口令，然后生成一个随机数，通过PBE算法计算出真正的AES口令，再进行加密，代码如下：

```
public class Main {
    public static void main(String[] args) throws Exception {
        // 把BouncyCastle作为Provider添加到java.security:
        Security.addProvider(new BouncyCastleProvider());
        // 原文:
        String message = "Hello, world!";
        // 加密口令:
        String password = "hello12345";
        // 16 bytes随机Salt:
        byte[] salt = SecureRandom.getInstanceStrong().generateSeed(16);
        System.out.printf("salt: %032x\n", new BigInteger(1, salt));
        // 加密:
        byte[] data = message.getBytes("UTF-8");
        byte[] encrypted = encrypt(password, salt, data);
        System.out.println("encrypted: " +
Base64.getEncoder().encodeToString(encrypted));
        // 解密:
        byte[] decrypted = decrypt(password, salt, encrypted);
        System.out.println("decrypted: " + new String(decrypted, "UTF-8"));
    }
```

```
20.
21.     // 加密:
        public static byte[] encrypt(String password, byte[] salt, byte[] input)
22. throws GeneralSecurityException {
23.         PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
            SecretKeyFactory skeyFactory =
24. SecretKeyFactory.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
25.         SecretKey skey = skeyFactory.generateSecret(keySpec);
26.         PBEParameterSpec pbeps = new PBEParameterSpec(salt, 1000);
27.         Cipher cipher = Cipher.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
28.         cipher.init(Cipher.ENCRYPT_MODE, skey, pbeps);
29.         return cipher.doFinal(input);
30.     }
31.
32.     // 解密:
        public static byte[] decrypt(String password, byte[] salt, byte[] input)
33. throws GeneralSecurityException {
34.         PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
            SecretKeyFactory skeyFactory =
35. SecretKeyFactory.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
36.         SecretKey skey = skeyFactory.generateSecret(keySpec);
37.         PBEParameterSpec pbeps = new PBEParameterSpec(salt, 1000);
38.         Cipher cipher = Cipher.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
39.         cipher.init(Cipher.DECRYPT_MODE, skey, pbeps);
40.         return cipher.doFinal(input);
41.     }
42. }
```

使用PBE时，我们还需要引入BouncyCastle，并指定算法是 `PBEwithSHA1and128bitAES-CBC-BC` 。观察代码，实际上真正的AES密钥是调用 `Cipher` 的 `init()` 方法时同时传入 `SecretKey` 和 `PBEParameterSpec` 实现的。在创建 `PBEParameterSpec` 的时候，我们还指定了循环次数 `1000` ，循环次数越多，暴力破解需要的计算量就越大。

如果我们把salt和循环次数固定，就得到了一个通用的“口令”加密软件。如果我们把随机生成的salt存储在U盘，就得到了一个“口令”加USB Key的加密软件，它的好处在于，即使用户使用了一个非常弱的口令，没有USB Key仍然无法解密，因为USB Key存储的随机数密钥安全性非常高。

小结

PBE算法通过用户口令和安全的随机salt计算出Key，然后再进行加密；

Key通过口令和安全的随机salt计算得出，大大提高了安全性；

PBE算法内部使用的仍然是标准对称加密算法（例如AES）。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





密钥交换算法

对称加密算法解决了数据加密的问题。我们以AES加密为例，在现实世界中，小明要向路人甲发送一个加密文件，他可以先生成一个AES密钥，对文件进行加密，然后把加密文件发送给对方。因为对方要解密，就必须需要小明生成的密钥。

现在问题来了：如何传递密钥？

在不安全的信道上传递加密文件是没有问题的，因为黑客拿到加密文件没有用。但是，如何如何在不安全的信道上安全地传输密钥？

要解决这个问题，密钥交换算法即DH算法：Diffie-Hellman算法应运而生。

DH算法解决了密钥在双方不直接传递密钥的情况下完成密钥交换，这个神奇的交换原理完全由数学理论支持。

我们来看DH算法交换密钥的步骤。假设甲乙双方需要传递密钥，他们之间可以这么做：

- 甲首选选择一个素数 `p`，例如509，底数 `g`，任选，例如5，随机数 `a`，例如123，然后计算 `A=g^a mod p`，结果是215，然后，甲发送 `p=509`，`g=5`，`A=215` 给乙；
- 乙方收到后，也选择一个随机数 `b`，例如，456，然后计算 `B=g^b mod p`，结果是181，乙再同时计算 `s=A^b mod p`，结果是121；
- 乙把计算的 `B=181` 发给甲，甲计算 `s=B^a mod p` 的余数，计算结果与乙算出的结果一样，都是121。所以最终双方协商出的密钥 `s` 是121。注意到这个密钥 `s` 并没有在网络上传输。而通过网络传输的 `p`，`g`，`A` 和 `B` 是无法推算出 `s` 的，因为实际算法选择的素数是非常大的。

所以，更确切地说，DH算法是一个密钥协商算法，双方最终协商出一个共同的密钥，而这个密钥不会通过网络传输。

如果我们把 `a` 看成甲的私钥，`A` 看成甲的公钥，`b` 看成乙的私钥，`B` 看成乙的公钥，DH算法的本质就是双方各自生成自己的私钥和公钥，私钥仅对自己可见，然后交换公钥，并根据自己的私钥和对方的公钥，生成最终的密钥 `secretKey`，DH算法通过数学定律保证了双方各自计算出的 `secretKey` 是相同的。

使用Java实现DH算法的代码如下：

```
import java.math.BigInteger;
import java.security.*;
import java.security.spec.*;

import javax.crypto.KeyAgreement;
```

```
public class Main {
    public static void main(String[] args) {
        // Bob和Alice:
        Person bob = new Person("Bob");
        Person alice = new Person("Alice");

        // 各自生成KeyPair:
        bob.generateKeyPair();
        alice.generateKeyPair();

        // 双方交换各自的PublicKey:
        // Bob根据Alice的PublicKey生成自己的本地密钥:
        bob.generateSecretKey(alice.publicKey.getEncoded());
        // Alice根据Bob的PublicKey生成自己的本地密钥:
        alice.generateSecretKey(bob.publicKey.getEncoded());

        // 检查双方本地密钥是否相同:
        bob.printKeys();
        alice.printKeys();
        // 双方的SecretKey相同，后续通信将使用SecretKey作为密钥进行AES加解密...
    }
}

class Person {
    public final String name;

    public PublicKey publicKey;
    private PrivateKey privateKey;
    private byte[] secretKey;

    public Person(String name) {
        this.name = name;
    }

    // 生成本地KeyPair:
    public void generateKeyPair() {
        try {
            KeyPairGenerator kpGen = KeyPairGenerator.getInstance("DH");
            kpGen.initialize(512);
            KeyPair kp = kpGen.generateKeyPair();
            this.privateKey = kp.getPrivate();
            this.publicKey = kp.getPublic();
        } catch (GeneralSecurityException e) {
            throw new RuntimeException(e);
        }
    }

    public void generateSecretKey(byte[] receivedPubKeyBytes) {
        try {
            // 从byte[]恢复PublicKey:
            X509EncodedKeySpec keySpec = new X509EncodedKeySpec(receivedPubKeyBytes);
            KeyFactory kf = KeyFactory.getInstance("DH");
            PublicKey receivedPublicKey = kf.generatePublic(keySpec);
            // 生成本地密钥:
            KeyAgreement keyAgreement = KeyAgreement.getInstance("DH");
            keyAgreement.init(this.privateKey); // 自己的PrivateKey
            keyAgreement.doPhase(receivedPublicKey, true); // 对方的PublicKey
            // 生成SecretKey密钥:
            this.secretKey = keyAgreement.generateSecret();
        } catch (GeneralSecurityException e) {
            throw new RuntimeException(e);
        }
    }

    public void printKeys() {
        System.out.printf("Name: %s\n", this.name);
        System.out.printf("Private key: %x\n", new BigInteger(1, this.privateKey.getEncoded()));
        System.out.printf("Public key: %x\n", new BigInteger(1, this.publicKey.getEncoded()));
        System.out.printf("Secret key: %x\n", new BigInteger(1, this.secretKey));
    }
}
```

▶ Run

但是DH算法并未解决中间人攻击，即甲乙双方并不能确保与自己通信的是否真的是对方。消除中间人攻击需要其他方法。

练习

下载练习：密钥交换算法 （推荐使用IDE练习插件快速下载）

小结

DH算法是一种密钥交换协议，通信双方通过不安全的信道协商密钥，然后进行对称加密传输。

DH算法没有解决中间人攻击。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





非对称加密算法

从DH算法我们可以看到，公钥-私钥组成的密钥对是非常有用的加密方式，因为公钥是可以公开的，而私钥是完全保密的，由此奠定了非对称加密的基础。

非对称加密就是加密和解密使用的不是相同的密钥：只有同一个公钥-私钥对才能正常加解密。

因此，如果小明要加密一个文件发送给小红，他应该首先向小红索取她的公钥，然后，他用小红的公钥加密，把加密文件发送给小红，此文件只能由小红的私钥解开，因为小红的私钥在她自己手里，所以，除了小红，没有任何人能解开此文件。

非对称加密的典型算法就是RSA算法，它是由Ron Rivest，Adi Shamir，Leonard Adleman这三个哥们一起发明的，所以用他们仨的姓的首字母缩写表示。

非对称加密相比对称加密的显著优点在于，对称加密需要协商密钥，而非对称加密可以安全地公开各自的公钥，在N个人之间通信的时候：使用非对称加密只需要N个密钥对，每个人只管理自己的密钥对。而使用对称加密需要则需要 `N*(N-1)/2` 个密钥，因此每个人需要管理 `N-1` 个密钥，密钥管理难度大，而且非常容易泄漏。

既然非对称加密这么好，那我们抛弃对称加密，完全使用非对称加密行不行？也不行。因为非对称加密的缺点就是运算速度非常慢，比对称加密要慢很多。

所以，在实际应用的时候，非对称加密总是和对称加密一起使用。假设小明需要给小红需要传输加密文件，他俩首先交换了各自的公钥，然后：

- 小明生成一个随机的AES口令，然后用小红的公钥通过RSA加密这个口令，并发给小红；
- 小红用自己的RSA私钥解密得到AES口令；
- 双方使用这个共享的AES口令用AES加密通信。 可见非对称加密实际上应用在第一步，即加密“AES口令”。这也是我们在浏览器中常用的HTTPS协议的做法，即浏览器和服务器先通过RSA交换AES口令，接下来双方通信实际上采用的是速度较快的AES对称加密，而不是缓慢的RSA非对称加密。

Java标准库提供了RSA算法的实现，示例代码如下：

```
import java.math.BigInteger;
import java.security.*;
import javax.crypto.Cipher;

public class Main {
    public static void main(String[] args) throws Exception {
        // 明文:
        byte[] plain = "Hello, encrypt use RSA".getBytes("UTF-8");
        // 创建公钥/私钥对:
        Person alice = new Person("Alice");
        // 用Alice的公钥加密:
        byte[] pk = alice.getPublicKey();
        System.out.println(String.format("public key: %x", new BigInteger(1, pk)));
        byte[] encrypted = alice.encrypt(plain);
        System.out.println(String.format("encrypted: %x", new BigInteger(1, encrypted)));
        // 用Alice的私钥解密:
        byte[] sk = alice.getPrivateKey();
        System.out.println(String.format("private key: %x", new BigInteger(1, sk)));
        byte[] decrypted = alice.decrypt(encrypted);
        System.out.println(new String(decrypted, "UTF-8"));
    }
}

class Person {
    String name;
    // 私钥:
    PrivateKey sk;
    // 公钥:
    PublicKey pk;

    public Person(String name) throws GeneralSecurityException {
        this.name = name;
        // 生成公钥/私钥对:
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("RSA");
        kpGen.initialize(1024);
        KeyPair kp = kpGen.generateKeyPair();
        this.sk = kp.getPrivate();
        this.pk = kp.getPublic();
    }

    // 把私钥导出为字节
    public byte[] getPrivateKey() {
        return this.sk.getEncoded();
    }

    // 把公钥导出为字节
    public byte[] getPublicKey() {
        return this.pk.getEncoded();
    }

    // 用公钥加密:
    public byte[] encrypt(byte[] message) throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.ENCRYPT_MODE, this.pk);
        return cipher.doFinal(message);
    }

    // 用私钥解密:
    public byte[] decrypt(byte[] input) throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance("RSA");
        cipher.init(Cipher.DECRYPT_MODE, this.sk);
        return cipher.doFinal(input);
    }
}
```

▶ Run

RSA的公钥和私钥都可以通过 `getEncoded()` 方法获得以 `byte[]` 表示的二进制数据，并根据需要保存到文件中。要从 `byte[]` 数组恢复公钥或私钥，可以这么写：

```
byte[] pkData = ...
byte[] skData = ...
KeyFactory kf = KeyFactory.getInstance("RSA");
// 恢复公钥:
X509EncodedKeySpec pkSpec = new X509EncodedKeySpec(pkData);
PublicKey pk = kf.generatePublic(pkSpec);
// 恢复私钥:
PKCS8EncodedKeySpec skSpec = new PKCS8EncodedKeySpec(skData);
PrivateKey sk = kf.generatePrivate(skSpec);
```

以RSA算法为例，它的密钥有256/512/1024/2048/4096等不同的长度。长度越长，密码强度越大，当然计算速度也越慢。

如果修改待加密的 `byte[]` 数据的大小，可以发现，使用512bit的RSA加密时，明文长度不能超过53字节，使用1024bit的RSA加密时，明文长度不能超过117字节，这也是为什么使用RSA的时候，总是配合AES一起使用，即用AES加密任意长度的明文，用RSA加密AES口令。

此外，只使用非对称加密算法不能防止中间人攻击。

练习

下载练习：RSA加密 （推荐使用IDE练习插件快速下载）

小结

非对称加密就是加密和解密使用的不是相同的密钥，只有同一个公钥-私钥对才能正常加解密；

只使用非对称加密算法不能防止中间人攻击。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





签名算法

我们使用非对称加密算法的时候，对于一个公钥-私钥对，通常是用公钥加密，私钥解密。

如果使用私钥加密，公钥解密是否可行呢？实际上是完全可行的。

不过我们再仔细想一想，私钥是保密的，而公钥是公开的，用私钥加密，那相当于所有人都可以用公钥解密。这个加密有什么意义？

这个加密的意义在于，如果小明用自己的私钥加密了一条消息，比如 `小明喜欢小红` ，然后他公开了加密消息，由于任何人都可以用小明的公钥解密，从而使得任何人都可以确认 `小明喜欢小红` 这条消息肯定是小明发出的，其他人不能伪造这个消息，小明也不能抵赖这条消息不是自己写的。

因此，私钥加密得到的密文实际上就是数字签名，要验证这个签名是否正确，只能用私钥持有者的公钥进行解密验证。使用数字签名的目的是为了确认某个信息确实是由某个发送方发送的，任何人都不可能伪造消息，并且，发送方也不能抵赖。

在实际应用的时候，签名实际上并不是针对原始消息，而是针对原始消息的哈希进行签名，即：

```
signature = encrypt(privateKey, sha256(message))
```

对签名进行验证实际上就是用公钥解密：

```
hash = decrypt(publicKey, signature)
```

然后把解密后的哈希与原始消息的哈希进行对比。

因为用户总是使用自己的私钥进行签名，所以，私钥就相当于用户身份。而公钥用来给外部验证用户身份。

常用数字签名算法有：

- MD5withRSA
- SHA1withRSA
- SHA256withRSA

它们实际上就是指定某种哈希算法进行RSA签名的方式。

```
import java.math.BigInteger;
import java.nio.charset.StandardCharsets;
import java.security.*;
```

```
public class Main {
    public static void main(String[] args) throws GeneralSecurityException {
        // 生成RSA公钥/私钥:
        KeyPairGenerator kpGen = KeyPairGenerator.getInstance("RSA");
        kpGen.initialize(1024);
        KeyPair kp = kpGen.generateKeyPair();
        PrivateKey sk = kp.getPrivate();
        PublicKey pk = kp.getPublic();

        // 待签名的消息:
        byte[] message = "Hello, I am Bob!".getBytes(StandardCharsets.UTF_8);

        // 用私钥签名:
        Signature s = Signature.getInstance("SHA1withRSA");
        s.initSign(sk);
        s.update(message);
        byte[] signed = s.sign();
        System.out.println(String.format("signature: %x", new BigInteger(1, signed)));

        // 用公钥验证:
        Signature v = Signature.getInstance("SHA1withRSA");
        v.initVerify(pk);
        v.update(message);
        boolean valid = v.verify(signed);
        System.out.println("valid? " + valid);
    }
}
```

▶ Run

使用其他公钥，或者验证签名的时候修改原始信息，都无法验证成功。

DSA签名

除了RSA可以签名外，还可以使用DSA算法进行签名。DSA是Digital Signature Algorithm的缩写，它使用ElGamal数字签名算法。

DSA只能配合SHA使用，常用的算法有：

- SHA1withDSA
- SHA256withDSA
- SHA512withDSA

和RSA数字签名相比，DSA的优点是更快。

ECDSA签名

椭圆曲线签名算法ECDSA：Elliptic Curve Digital Signature Algorithm也是一种常用的签

名算法，它的特点是可以从私钥推出公钥。比特币的签名算法就采用了ECDSA算法，使用标准椭圆曲线secp256k1。BouncyCastle提供了ECDSA的完整实现。

练习

下载练习：签名算法练习 （推荐使用IDE练习插件快速下载）

小结

数字签名就是用发送方的私钥对原始数据进行签名，只有用发送方公钥才能通过签名验证。

数字签名用于：

- 防止伪造；
- 防止抵赖；
- 检测篡改。

常用的数字签名算法包括：MD5withRSA / SHA1withRSA / SHA256withRSA / SHA1withDSA / SHA256withDSA / SHA512withDSA / ECDSA等。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





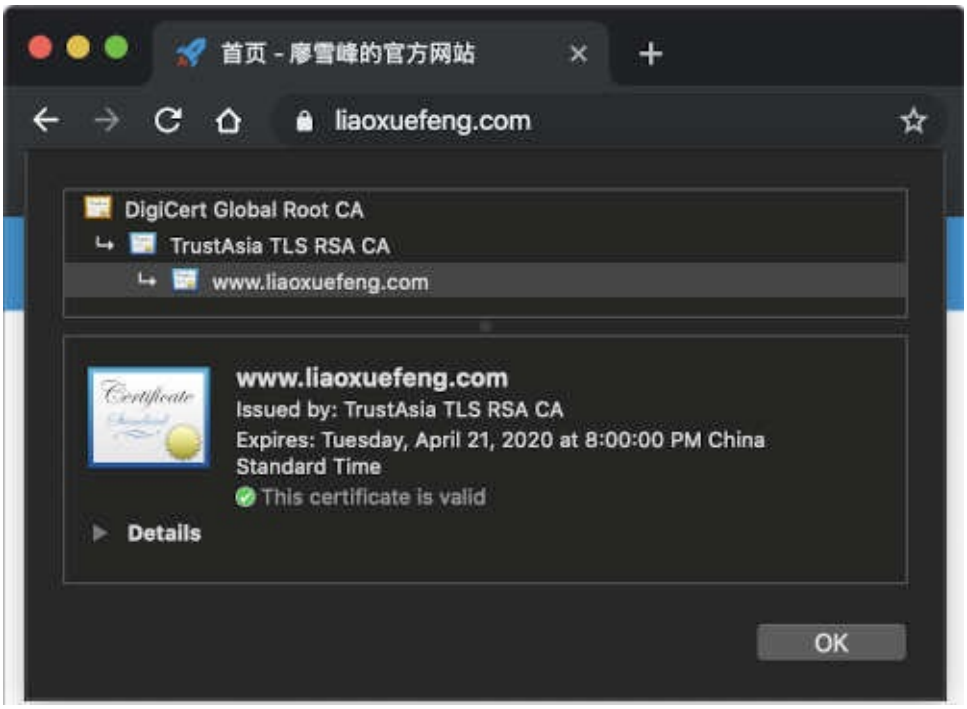
数字证书

我们知道，摘要算法用来确保数据没有被篡改，非对称加密算法可以对数据进行加解密，签名算法可以确保数据完整性和抗否认性，把这些算法集合到一起，并搞一套完善的标准，这就是数字证书。

因此，数字证书就是集合了多种密码学算法，用于实现数据加解密、身份认证、签名等多种功能的一种安全标准。

数字证书可以防止中间人攻击，因为它采用链式签名认证，即通过根证书（Root CA）去签名下一级证书，这样层层签名，直到最终的用户证书。而Root CA证书内置于操作系统中，所以，任何经过CA认证的数字证书都可以对其本身进行校验，确保证书本身不是伪造的。

我们在上网时常用的HTTPS协议就是数字证书的应用。浏览器会自动验证证书的有效性：



要使用数字证书，首先需要创建证书。正常情况下，一个合法的数字证书需要经过CA签名，这需要认证域名并支付一定的费用。开发的时候，我们可以使用自签名的证书，这种证书可以正常开发调试，但不能对外作为服务使用，因为其他客户端并不认可未经CA签名的证书。

在Java程序中，数字证书存储在一种Java专用的key store文件中，JDK提供了一系列命令来创建和管理key store。我们用下面的命令创建一个key store，并设定口令123456：

```
keytool -storepass 123456 -genkeypair -keyalg RSA -keysize 1024 -sigalg
SHA1withRSA -validity 3650 -alias mycert -keystore my.keystore -dname
"CN=www.sample.com, OU=sample, O=sample, L=BJ, ST=BJ, C=CN"
```

几个主要的参数是：

- keyalg: 指定RSA加密算法;
- sigalg: 指定SHA1withRSA签名算法;
- validity: 指定证书有效期3650天;
- alias: 指定证书在程序中引用的名称;
- dname: 最重要的 `CN=www.sample.com` 指定了 `Common Name`，如果证书用在HTTPS中，这个名称必须与域名完全一致。

执行上述命令，JDK会在当前目录创建一个 `my.keystore` 文件，并存储创建成功的一个私钥和一个证书，它的别名是 `mycert` 。

有了key store存储的证书，我们就可以通过数字证书进行加解密和签名:

```
import java.io.InputStream;
import java.math.BigInteger;
import java.security.*;
import java.security.cert.*;
import javax.crypto.Cipher;

public class Main {
    public static void main(String[] args) throws Exception {
        byte[] message = "Hello, use X.509 cert!".getBytes("UTF-8");
        // 读取KeyStore:
        KeyStore ks = loadKeyStore("/my.keystore", "123456");
        // 读取私钥:
        PrivateKey privateKey = (PrivateKey) ks.getKey("mycert",
"123456".toCharArray());
        // 读取证书:
        X509Certificate certificate = (X509Certificate)
ks.getCertificate("mycert");
        // 加密:
        byte[] encrypted = encrypt(certificate, message);
        System.out.println(String.format("encrypted: %x", new BigInteger(1,
encrypted)));
        // 解密:
        byte[] decrypted = decrypt(privateKey, encrypted);
        System.out.println("decrypted: " + new String(decrypted, "UTF-8"));
        // 签名:
        byte[] sign = sign(privateKey, certificate, message);
        System.out.println(String.format("signature: %x", new BigInteger(1,
sign)));
        // 验证签名:
        boolean verified = verify(certificate, message, sign);
        System.out.println("verify: " + verified);
```

```
28.     }
29.
30.     static KeyStore loadKeyStore(String keyStoreFile, String password) {
            try (InputStream input = Main.class.getResourceAsStream(keyStoreFile))
31. {
32.             if (input == null) {
                    throw new RuntimeException("file not found in classpath: " +
33. keyStoreFile);
34.             }
35.             KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
36.             ks.load(input, password.toCharArray());
37.             return ks;
38.         } catch (Exception e) {
39.             throw new RuntimeException(e);
40.         }
41.     }
42.
        static byte[] encrypt(X509Certificate certificate, byte[] message) throws
43. GeneralSecurityException {
            Cipher cipher =
44. Cipher.getInstance(certificate.getPublicKey().getAlgorithm());
45.         cipher.init(Cipher.ENCRYPT_MODE, certificate.getPublicKey());
46.         return cipher.doFinal(message);
47.     }
48.
        static byte[] decrypt(PrivateKey privateKey, byte[] data) throws
49. GeneralSecurityException {
50.         Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
51.         cipher.init(Cipher.DECRYPT_MODE, privateKey);
52.         return cipher.doFinal(data);
53.     }
54.
        static byte[] sign(PrivateKey privateKey, X509Certificate certificate,
55. byte[] message)
56.             throws GeneralSecurityException {
            Signature signature =
57. Signature.getInstance(certificate.getSigAlgName());
58.         signature.initSign(privateKey);
59.         signature.update(message);
60.         return signature.sign();
61.     }
62.
```

```
        static boolean verify(X509Certificate certificate, byte[] message, byte[]
63. sig) throws GeneralSecurityException {
            Signature signature =
64. Signature.getInstance(certificate.getSigAlgName());
65.         signature.initVerify(certificate);
66.         signature.update(message);
67.         return signature.verify(sig);
68.     }
69. }
```

在上述代码中，我们从key store直接读取了私钥-公钥对，私钥以 `PrivateKey` 实例表示，公钥以 `X509Certificate` 表示，实际上数字证书只包含公钥，因此，读取证书并不需要口令，只有读取私钥才需要。如果部署到Web服务器上，例如Nginx，需要把私钥导出为Private Key格式，把证书导出为X509Certificate格式。

以HTTPS协议为例，浏览器和服务器建立安全连接的步骤如下：

- 浏览器向服务器发起请求，服务器向浏览器发送自己的数字证书；
- 浏览器用操作系统内置的Root CA来验证服务器的证书是否有效，如果有效，就使用该证书加密一个随机的AES口令并发送给服务器；
- 服务器用自己的私钥解密获得AES口令，并在后续通讯中使用AES加密。上述流程只是一种最常见的单向验证。如果服务器还要验证客户端，那么客户端也需要把自己的证书发送给服务器验证，这种场景常见于网银等。

注意：数字证书存储的是公钥，以及相关的证书链和算法信息。私钥必须严格保密，如果数字证书对应的私钥泄漏，就会造成严重的安全威胁。如果CA证书的私钥泄漏，那么该CA证书签发的所有证书将不可信。数字证书服务商DigiNotar就发生过私钥泄漏导致公司破产的事故。

练习

下载练习：使用数字证书 （推荐使用IDE练习插件快速下载）

小结

数字证书就是集合了多种密码学算法，用于实现数据加解密、身份认证、签名等多种功能的一种安全标准。

数字证书采用链式签名管理，顶级的Root CA证书已内置在操作系统中。

数字证书存储的是公钥，可以安全公开，而私钥必须严格保密。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





多线程

多线程是Java最基本的一种并发模型，本章我们将详细介绍Java多线程编程。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





多线程基础

现代操作系统（Windows，macOS，Linux）都可以执行多任务。多任务就是同时运行多个任务，例如：

CPU执行代码都是一条一条顺序执行的，但是，即使是单核cpu，也可以同时运行多个任务。因为操作系统执行多任务实际上就是让CPU对多个任务轮流交替执行。

例如，假设我们有语文、数学、英语3门作业要做，每个作业需要30分钟。我们把这3门作业看成是3个任务，可以做1分钟语文作业，再做1分钟数学作业，再做1分钟英语作业：



这样轮流做下去，在某些人眼里看来，做作业的速度就非常快，看上去就像同时在做3门作业一样



类似的，操作系统轮流让多个任务交替执行，例如，让浏览器执行0.001秒，让QQ执行0.001秒，再让音乐播放器执行0.001秒，在人看来，CPU就是在同时执行多个任务。

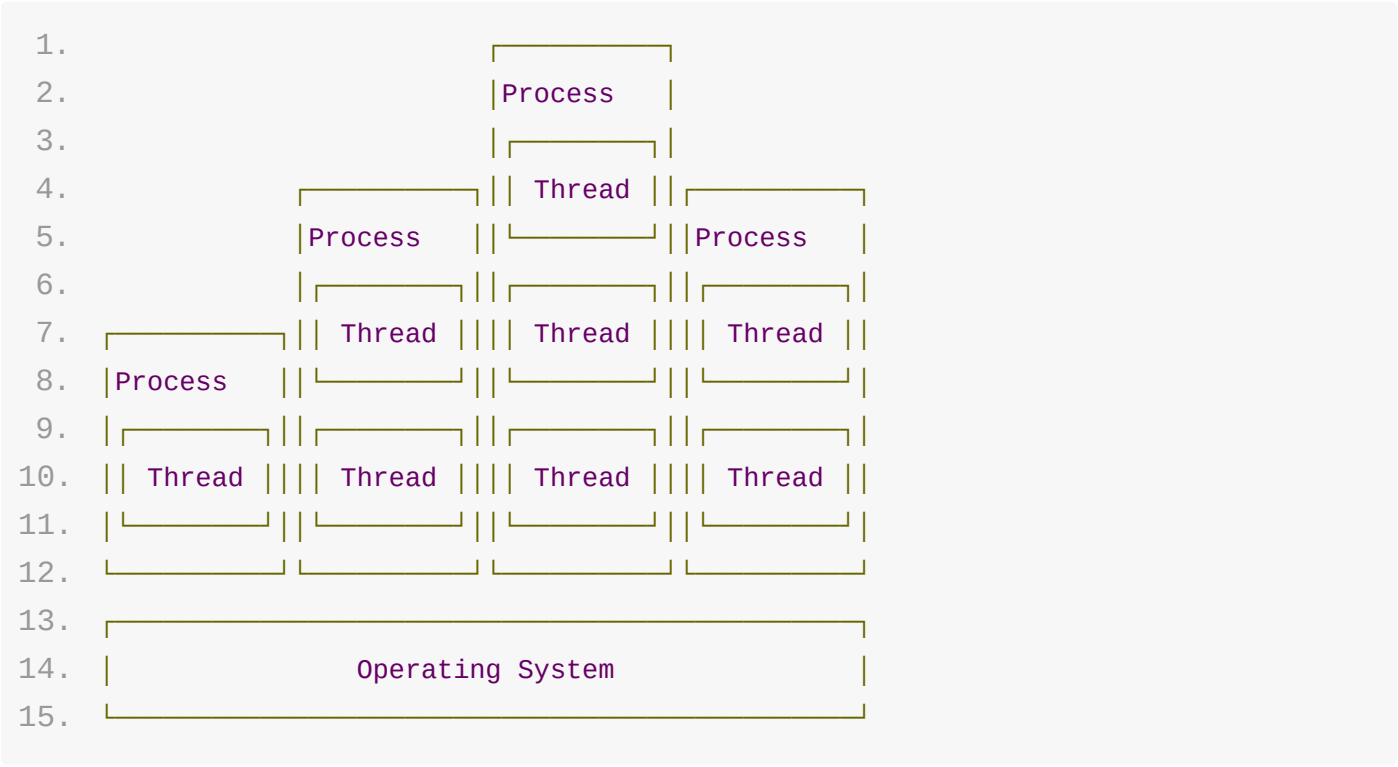
即使是多核CPU，因为通常任务的数量远远多于CPU的核数，所以任务也是交替执行的。

进程

在计算机中，我们把一个任务称为一个进程，浏览器就是一个进程，视频播放器是另一个进程，类似的，音乐播放器和Word都是进程。

某些进程内部还需要同时执行多个子任务。例如，我们在使用Word时，Word可以让我们一边打字，一边进行拼写检查，同时还可以在后台进行打印，我们把子任务称为线程。

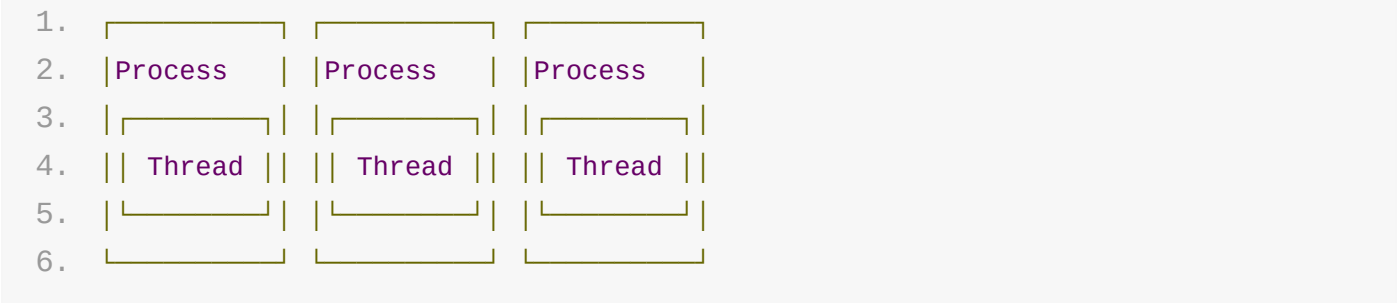
进程和线程的关系就是：一个进程可以包含一个或多个线程，但至少会有一个线程。



操作系统调度的最小任务单位其实不是进程，而是线程。常用的Windows、Linux等操作系统都采用抢占式多任务，如何调度线程完全由操作系统决定，程序自己不能决定什么时候执行，以及执行多长时间。

因为同一个应用程序，既可以有多个进程，也可以有多个线程，因此，实现多任务的方法，有以下几种：

多进程模式（每个进程只有一个线程）：



多线程模式（一个进程有多个线程）：

```
┌──────────────────────┐
│Process               │
│┌──────────┐┌──────────┐│
|| Thread || Thread ||
│└──────────┘└──────────┘│
│┌──────────┐┌──────────┐│
|| Thread || Thread ||
│└──────────┘└──────────┘│
└──────────────────────┘
```

多进程+多线程模式（复杂度最高）：

```
┌──────────┐┌──────────┐┌──────────┐
|Process   ||Process   ||Process   |
|┌──────────┐||┌──────────┐||┌──────────┐|
|| Thread |||| Thread |||| Thread ||
|└──────────┘||└──────────┘||└──────────┘|
|┌──────────┐||┌──────────┐||┌──────────┐|
|| Thread |||| Thread |||| Thread ||
|└──────────┘||└──────────┘||└──────────┘|
└──────────┘└──────────┘└──────────┘
```

进程 vs 线程

进程和线程是包含关系，但是多任务既可以由多进程实现，也可以由单进程内的多线程实现，还可以混合多进程+多线程。

具体采用哪种方式，要考虑到进程和线程的特点。

和多线程相比，多进程的缺点在于：

- 创建进程比创建线程开销大，尤其是在Windows系统上；
- 进程间通信比线程间通信要慢，因为线程间通信就是读写同一个变量，速度很快。

而多进程的优点在于：

多进程稳定性比多线程高，因为在多进程的情况下，一个进程崩溃不会影响其他进程，而在多线程的情况下，任何一个线程崩溃会直接导致整个进程崩溃。

多线程

Java语言内置了多线程支持：一个Java程序实际上是一个JVM进程，JVM进程用一个主线程来执

行 `main()` 方法，在 `main()` 方法内部，我们又可以启动多个线程。此外，JVM还有负责垃圾回收的其他工作线程等。

因此，对于大多数Java程序来说，我们说多任务，实际上是说如何使用多线程实现多任务。

和单线程相比，多线程编程的特点在于：多线程经常需要读写共享数据，并且需要同步。例如，播放电影时，就必须由一个线程播放视频，另一个线程播放音频，两个线程需要协调运行，否则画面和声音就不同步。因此，多线程编程的复杂度高，调试更困难。

Java多线程编程的特点又在于：

- 多线程模型是Java程序最基本的并发模型；
- 后续读写网络、数据库、Web开发等都依赖Java多线程模型。

因此，必须掌握Java多线程编程才能继续深入学习其他内容。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





创建新线程

Java语言内置了多线程支持。当Java程序启动的时候，实际上是启动了一个JVM进程，然后，JVM启动主线程来执行 `main()` 方法。在 `main()` 方法中，我们又可以启动其他线程。

要创建一个新线程非常容易，我们需要实例化一个 `Thread` 实例，然后调用它的 `start()` 方法：

```
// 多线程
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread();
        t.start(); // 启动新线程
    }
}
```

▶ Run

但是这个线程启动后实际上什么也不做就立刻结束了。我们希望新线程能执行指定的代码，有以下几种方法：

方法一：从 `Thread` 派生一个自定义类，然后覆写 `run()` 方法：

```
// 多线程
public class Main {
    public static void main(String[] args) {
        Thread t = new MyThread();
        t.start(); // 启动新线程
    }
}

class MyThread extends Thread {
    @Override
    public void run() {
        System.out.println("start new thread!");
    }
}
```

▶ Run

执行上述代码，注意到 `start()` 方法会在内部自动调用实例的 `run()` 方法。

方法二：创建 `Thread` 实例时，传入一个 `Runnable` 实例：

```
// 多线程
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(new MyRunnable());
        t.start(); // 启动新线程
    }
}

class MyRunnable implements Runnable {
    @Override
    public void run() {
        System.out.println("start new thread!");
    }
}
```

▶ Run

或者用Java8引入的lambda语法进一步简写为：

```
// 多线程
public class Main {
    public static void main(String[] args) {
        Thread t = new Thread(() -> {
            System.out.println("start new thread!");
        });
        t.start(); // 启动新线程
    }
}
```

▶ Run

有童鞋会问，使用线程执行的打印语句，和直接在 `main()` 方法执行有区别吗？

区别大了去了。我们看以下代码：

```
1.  public class Main {
2.      public static void main(String[] args) {
3.          System.out.println("main start...");
4.          Thread t = new Thread() {
5.              public void run() {
6.                  System.out.println("thread run...");
7.                  System.out.println("thread end.");
8.              }
9.          };
10.         t.start();
11.         System.out.println("main end...");
```

```
12.     }
13. }
```

我们用蓝色表示主线程，也就是 `main` 线程， `main` 线程执行的代码有4行，首先打印 `main start` ，然后创建 `Thread` 对象，紧接着调用 `start()` 启动新线程。当 `start()` 方法被调用时，JVM就创建了一个新线程，我们通过实例变量 `t` 来表示这个新线程对象，并开始执行。

接着， `main` 线程继续执行打印 `main end` 语句，而 `t` 线程在 `main` 线程执行的同时会并发执行，打印 `thread run` 和 `thread end` 语句。

当 `run()` 方法结束时，新线程就结束了。而 `main()` 方法结束时，主线程也结束了。

我们再来看线程的执行顺序：

- `main` 线程肯定是先打印 `main start` ，再打印 `main end` ；
- `t` 线程肯定是先打印 `thread run` ，再打印 `thread end` 。但是，除了可以肯定， `main start` 会先打印外， `main end` 打印在 `thread run` 之前、 `thread end` 之后或者之间，都无法确定。因为从 `t` 线程开始运行以后，两个线程就开始同时运行了，并且由操作系统调度，程序本身无法确定线程的调度顺序。

要模拟并发执行的效果，我们可以在线程中调用 `Thread.sleep()` ，强迫当前线程暂停一段时间：

```
// 多线程
public class Main {
    public static void main(String[] args) {
        System.out.println("main start...");
        Thread t = new Thread() {
            public void run() {
                System.out.println("thread run...");
                try {
                    Thread.sleep(10);
                } catch (InterruptedException e) {}
                System.out.println("thread end.");
            }
        };
        t.start();
        try {
            Thread.sleep(20);
        } catch (InterruptedException e) {}
        System.out.println("main end...");
    }
}
```

▶ Run

`sleep()` 传入的参数是毫秒。调整暂停时间的大小，我们可以看到 `main` 线程和 `t` 线程执行的先后顺序。

要特别注意：直接调用 `Thread` 实例的 `run()` 方法是无效的：

```
1.  public class Main {
2.      public static void main(String[] args) {
3.          Thread t = new MyThread();
4.          t.run();
5.      }
6.  }
7.
8.  class MyThread extends Thread {
9.      public void run {
10.         System.out.println("hello");
11.     }
12. }
```

直接调用 `run()` 方法，相当于调用了一个普通的Java方法，当前线程并没有任何改变，也不会启动新线程。上述代码实际上是在 `main()` 方法内部又调用了 `run()` 方法，打印 `hello` 语句是在 `main` 线程中执行的，没有任何新线程被创建。

必须调用 `Thread` 实例的 `start()` 方法才能启动新线程，如果我们查看 `Thread` 类的源代码，会看到 `start()` 方法内部调用了一个 `private native void start0()` 方法， `native` 修饰符表示这个方法是由JVM虚拟机内部的C代码实现的，不是由Java代码实现的。

线程的优先级

可以对线程设定优先级，设定优先级的方法是：

```
1.  Thread.setPriority(int n) // 1~10, 默认值5
```

优先级高的线程被操作系统调度的优先级较高，操作系统对高优先级线程可能调度更频繁，但我们决不能通过设置优先级来确保高优先级的线程一定会先执行。

练习

下载练习：创建新线程 （推荐使用IDE练习插件快速下载）

小结

Java用 `Thread` 对象表示一个线程，通过调用 `start()` 启动一个新线程；

一个线程对象只能调用一次 `start()` 方法；

线程的执行代码写在 `run()` 方法中；

线程调度由操作系统决定，程序本身无法决定调度顺序；

`Thread.sleep()` 可以把当前线程暂停一段时间。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论



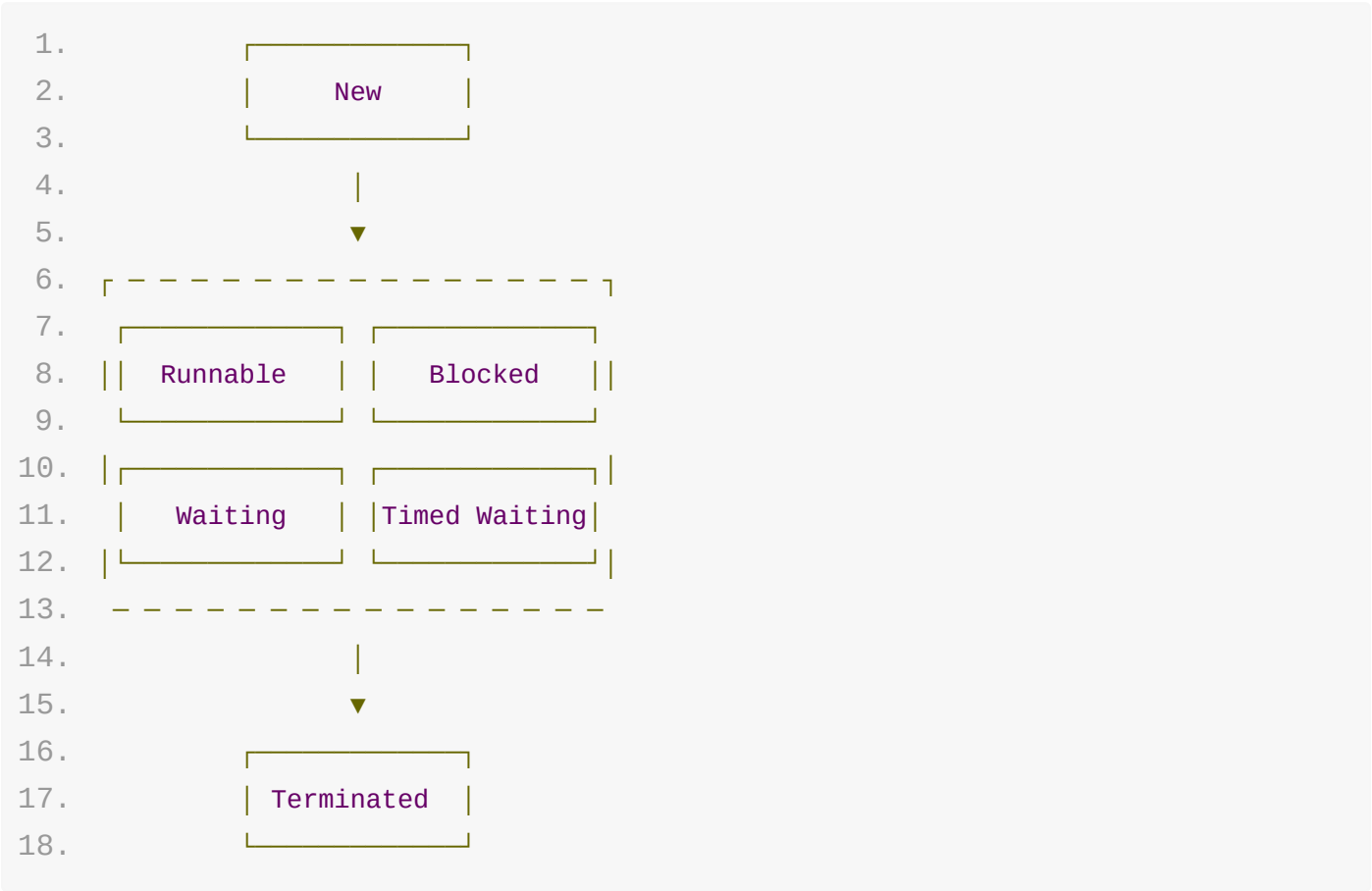


线程的状态

在Java程序中，一个线程对象只能调用一次 `start()` 方法启动新线程，并在新线程中执行 `run()` 方法。一旦 `run()` 方法执行完毕，线程就结束了。因此，Java线程的状态有以下几种：

- New：新创建的线程，尚未执行；
- Runnable：运行中的线程，正在执行 `run()` 方法的Java代码；
- Blocked：运行中的线程，因为某些操作被阻塞而挂起；
- Waiting：运行中的线程，因为某些操作在等待中；
- Timed Waiting：运行中的线程，因为执行 `sleep()` 方法正在计时等待；
- Terminated：线程已终止，因为 `run()` 方法执行完毕。

用一个状态转移图表示如下：



当线程启动后，它可以在 `Runnable` 、 `Blocked` 、 `Waiting` 和 `Timed Waiting` 这几个状态之间切换，直到最后变成 `Terminated` 状态，线程终止。

线程终止的原因有：

- 线程正常终止： `run()` 方法执行到 `return` 语句返回；
- 线程意外终止： `run()` 方法因为未捕获的异常导致线程终止；
- 对某个线程的 `Thread` 实例调用 `stop()` 方法强制终止（强烈不推荐使用）。

一个线程还可以等待另一个线程直到其运行结束。例如， `main` 线程在启动 `t` 线程后，可以通过 `t.join()` 等待 `t` 线程结束后再继续运行：

```
// 多线程
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new Thread(() -> {
            System.out.println("hello");
        });
        System.out.println("start");
        t.start();
        t.join();
        System.out.println("end");
    }
}
```

▶ Run

当 `main` 线程对线程对象 `t` 调用 `join()` 方法时，主线程将等待变量 `t` 表示的线程运行结束，即 `join` 就是指等待该线程结束，然后才继续往下执行自身线程。所以，上述代码打印顺序可以肯定是 `main` 线程先打印 `start` ， `t` 线程再打印 `hello` ， `main` 线程最后再打印 `end` 。

如果 `t` 线程已经结束，对实例 `t` 调用 `join()` 会立刻返回。此外， `join(long)` 的重载方法也可以指定一个等待时间，超过等待时间后就不再继续等待。

小结

Java线程对象 `Thread` 的状态包括： `New` 、 `Runnable` 、 `Blocked` 、 `Waiting` 、 `Timed Waiting` 和 `Terminated` ；

通过对另一个线程对象调用 `join()` 方法可以等待其执行结束；

可以指定等待时间，超过等待时间线程仍然没有结束就不再等待；

对已经运行结束的线程调用 `join()` 方法会立刻返回。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





中断线程

如果线程需要执行一个长时间任务，就可能需要能中断线程。中断线程就是其他线程给该线程发一个信号，该线程收到信号后结束执行 `run()` 方法，使得自身线程能立刻结束运行。

我们举个栗子：假设从网络下载一个100M的文件，如果网速很慢，用户等得不耐烦，就可能在下载过程中点“取消”，这时，程序就需要中断下载线程的执行。

中断一个线程非常简单，只需要在其他线程中对目标线程调用 `interrupt()` 方法，目标线程需要反复检测自身状态是否是interrupted状态，如果是，就立刻结束运行。

我们还是看示例代码：

```
// 中断线程
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new MyThread();
        t.start();
        Thread.sleep(1); // 暂停1毫秒
        t.interrupt(); // 中断t线程
        t.join(); // 等待t线程结束
        System.out.println("end");
    }
}

class MyThread extends Thread {
    public void run() {
        int n = 0;
        while (! isInterrupted()) {
            n ++;
            System.out.println(n + " hello!");
        }
    }
}
```

▶ Run

仔细看上述代码， `main` 线程通过调用 `t.interrupt()` 方法中断 `t` 线程，但是要注意， `interrupt()` 方法仅仅向 `t` 线程发出了“中断请求”，至于 `t` 线程是否能立刻响应，要看具体代码。而 `t` 线程的 `while` 循环会检测 `isInterrupted()` ，所以上述代码能正确响应 `interrupt()` 请求，使得自身立刻结束运行 `run()` 方法。

如果线程处于等待状态，例如， `t.join()` 会让 `main` 线程进入等待状态，此时，如果对 `main` 线程调用 `interrupt()` ， `join()` 方法会立刻抛出 `InterruptedException` ，因此，目标线程只要捕获到 `join()` 方法抛出的 `InterruptedException` ，就说明有其他线程对其调用了 `interrupt()` 方法，通常情况下该线程应该立刻结束运行。

我们来看下面的示例代码：

```
// 中断线程
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Thread t = new MyThread();
        t.start();
        Thread.sleep(1000);
        t.interrupt(); // 中断t线程
        t.join(); // 等待t线程结束
        System.out.println("end");
    }
}

class MyThread extends Thread {
    public void run() {
        Thread hello = new HelloThread();
        hello.start(); // 启动hello线程
        try {
            hello.join(); // 等待hello线程结束
        } catch (InterruptedException e) {
            System.out.println("interrupted!");
        }
        hello.interrupt();
    }
}

class HelloThread extends Thread {
    public void run() {
        int n = 0;
        while (!isInterrupted()) {
            n++;
            System.out.println(n + " hello!");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                break;
            }
        }
    }
}
```

▶ Run

`main` 线程通过调用 `t.interrupt()` 从而通知 `t` 线程中断，而此时 `t` 线程正位于 `hello.join()` 的等待中，此方法会立刻结束等待并抛出 `InterruptedException` 。由于我们在 `t` 线程中捕获了 `InterruptedException` ，因此，就可以准备结束该线程。在 `t` 线程结束前，对 `hello` 线程也进行了 `interrupt()` 调用通知其中断。如果去掉这一行代码，可以发现 `hello` 线程仍然会继续运行，且JVM不会退出。

另一个常用的中断线程的方法是设置标志位。我们通常会用一个 `running` 标志位来标识线程是否应该继续运行，在外部线程中，通过把 `HelloThread.running` 置为 `false` ，就可以让线程结束：

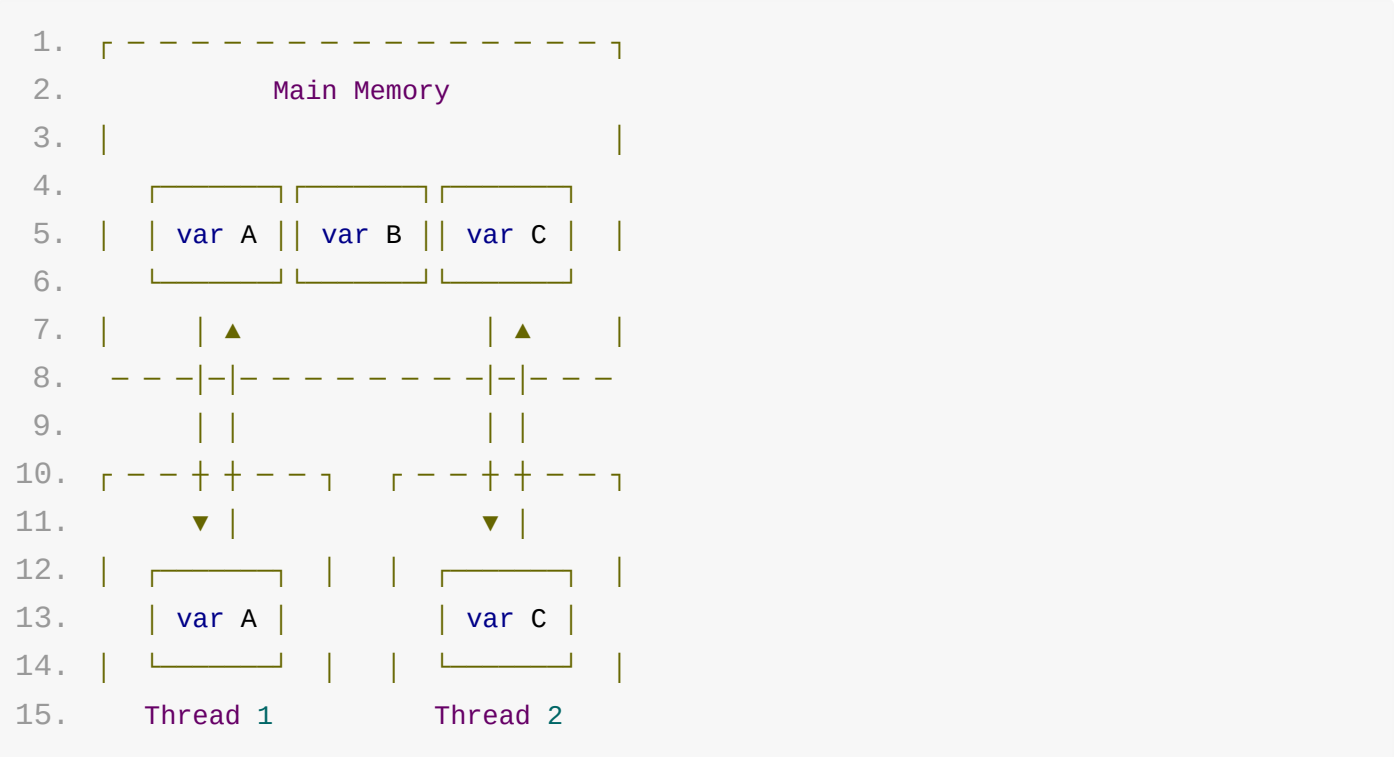
```
// 中断线程
public class Main {
    public static void main(String[] args)  throws InterruptedException {
        HelloThread t = new HelloThread();
        t.start();
        Thread.sleep(1);
        t.running = false; // 标志位置为false
    }
}

class HelloThread extends Thread {
    public volatile boolean running = true;
    public void run() {
        int n = 0;
        while (running) {
            n ++;
            System.out.println(n + " hello!");
        }
        System.out.println("end!");
    }
}
```

▶ Run

注意到 `HelloThread` 的标志位 `boolean running` 是一个线程间共享的变量。线程间共享变量需要使用 `volatile` 关键字标记，确保每个线程都能读取到更新后的变量值。

为什么要对线程间共享的变量用关键字 `volatile` 声明？这涉及到Java的内存模型。在Java虚拟机中，变量的值保存在主内存中，但是，当线程访问变量时，它会先获取一个副本，并保存在自己的工作内存中。如果线程修改了变量的值，虚拟机会在某个时刻把修改后的值回写到主内存，但是，这个时间是不确定的！



```
16. └ _ _ _ _ _ _ ┘   └ _ _ _ _ _ _ ┘
```

这会导致如果一个线程更新了某个变量，另一个线程读取的值可能还是更新前的。例如，主内存的变量 `a = true` ，线程1执行 `a = false` 时，它在此刻仅仅是把变量 `a` 的副本变成了 `false` ，主内存的变量 `a` 还是 `true` ，在JVM把修改后的 `a` 回写到主内存之前，其他线程读取到的 `a` 的值仍然是 `true` ，这就造成了多线程之间共享的变量不一致。

因此， `volatile` 关键字的目的是告诉虚拟机：

- 每次访问变量时，总是获取主内存的最新值；
- 每次修改变量后，立刻回写到主内存。

`volatile` 关键字解决的是可见性问题：当一个线程修改了某个共享变量的值，其他线程能够立刻看到修改后的值。

如果我们去掉 `volatile` 关键字，运行上述程序，发现效果和带 `volatile` 差不多，这是因为在x86的架构下，JVM回写主内存的速度非常快，但是，换成ARM的架构，就会有显著的延迟。

小结

对目标线程调用 `interrupt()` 方法可以请求中断一个线程，目标线程通过检测 `isInterrupted()` 标志获取自身是否已中断。如果目标线程处于等待状态，该线程会捕获到 `InterruptedException` ；

目标线程检测到 `isInterrupted()` 为 `true` 或者捕获了 `InterruptedException` 都应该立刻结束自身线程；

通过标志位判断需要正确使用 `volatile` 关键字；

`volatile` 关键字解决了共享变量在线程间的可见性问题。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





守护线程

Java程序入口就是由JVM启动 `main` 线程， `main` 线程又可以启动其他线程。当所有线程都运行结束时，JVM退出，进程结束。

如果有一个线程没有退出，JVM进程就不会退出。所以，必须保证所有线程都能及时结束。

但是有一种线程的目的就是无限循环，例如，一个定时触发任务的线程：

```
class TimerThread extends Thread {
    @Override
    public void run() {
        while (true) {
            System.out.println(LocalTime.now());
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                break;
            }
        }
    }
}
```

如果这个线程不结束，JVM进程就无法结束。问题是，由谁负责结束这个线程？

然而这类线程经常没有负责人来负责结束它们。但是，当其他线程结束时，JVM进程又必须要结束，怎么办？

答案是使用守护线程（Daemon Thread）。

守护线程是指为其他线程服务的线程。在JVM中，所有非守护线程都执行完毕后，无论有没有守护线程，虚拟机都会自动退出。

因此，JVM退出时，不必关心守护线程是否已结束。

如何创建守护线程呢？方法和普通线程一样，只是在调用 `start()` 方法前，调用 `setDaemon(true)` 把该线程标记为守护线程：

```
Thread t = new MyThread();
t.setDaemon(true);
t.start();
```

在守护线程中，编写代码要注意：守护线程不能持有任何需要关闭的资源，例如打开文件等，因为虚拟

机退出时，守护线程没有任何机会来关闭文件，这会导致数据丢失。

练习

下载练习：使用守护线程 （推荐使用IDE练习插件快速下载）

小结

守护线程是为其他线程服务的线程；

所有非守护线程都执行完毕后，虚拟机退出；

守护线程不能持有需要关闭的资源（如打开文件等）。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





线程同步

当多个线程同时运行时，线程的调度由操作系统决定，程序本身无法决定。因此，任何一个线程都有可能在任何指令处被操作系统暂停，然后在某个时间段后继续执行。

这个时候，有个单线程模型下不存在的问题就来了：如果多个线程同时读写共享变量，会出现数据不一致的问题。

我们来看一个例子：

```
// 多线程
public class Main {
    public static void main(String[] args) throws Exception {
        var add = new AddThread();
        var dec = new DecThread();
        add.start();
        dec.start();
        add.join();
        dec.join();
        System.out.println(Counter.count);
    }
}

class Counter {
    public static int count = 0;
}

class AddThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) { Counter.count += 1; }
    }
}

class DecThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) { Counter.count -= 1; }
    }
}
```

▶ Run

上面的代码很简单，两个线程同时对一个 `int` 变量进行操作，一个加10000次，一个减10000次，最后结果应该是0，但是，每次运行，结果实际上都是不一样的。

这是因为对变量进行读取和写入时，结果要正确，必须保证是原子操作。原子操作是指不能被中断的一个或一系列操作。

例如，对于语句：

```
1.  n = n + 1;
```

看上去是一行语句，实际上对应了3条指令：

```
ILOAD
IADD
ISTORE
```

我们假设 `n` 的值是 `100` ，如果两个线程同时执行 `n = n + 1` ，得到的结果很可能不是 `102` ，而是 `101` ，原因在于：

```
┌──────┐    ┌──────┐
|Thread1|    |Thread2|
└──┬───┘    └──┬───┘
   |           |
   |ILOAD (100)|
   |           |ILOAD (100)
   |           |IADD
   |           |ISTORE (101)
   |IADD       |
   |ISTORE (101)|
   ▼           ▼
```

如果线程1在执行 `ILOAD` 后被操作系统中断，此刻如果线程2被调度执行，它执行 `ILOAD` 后获取的值仍然是 `100` ，最终结果被两个线程的 `ISTORE` 写入后变成了 `101` ，而不是期待的 `102` 。

这说明多线程模型下，要保证逻辑正确，对共享变量进行读写时，必须保证一组指令以原子方式执行：即某一个线程执行时，其他线程必须等待：

```
┌──────┐    ┌──────┐
|Thread1|    |Thread2|
└──┬───┘    └──┬───┘
   |           |
   |-- lock -- |
   |ILOAD (100)|
   |IADD       |
   |ISTORE (101)|
   |-- unlock -- |
   |           |-- lock --
   |           |ILOAD (101)
   |           |IADD
   |           |ISTORE (102)
   |           |-- unlock --
```

```
15.     ▼               ▼
```

通过加锁和解锁的操作，就能保证3条指令总是在一个线程执行期间，不会有其他线程会进入此指令区间。即使在执行期线程被操作系统中断执行，其他线程也会因为无法获得锁导致无法进入此指令区间。只有执行线程将锁释放后，其他线程才有机会获得锁并执行。这种加锁和解锁之间的代码块我们称之为临界区（Critical Section），任何时候临界区最多只有一个线程能执行。

可见，保证一段代码的原子性就是通过加锁和解锁实现的。Java程序使用 `synchronized` 关键字对一个对象进行加锁：

```
1.  synchronized(lock) {
2.      n = n + 1;
3.  }
```

`synchronized` 保证了代码块在任意时刻最多只有一个线程能执行。我们把上面的代码用 `synchronized` 改写如下：

```
// 多线程
public class Main {
    public static void main(String[] args) throws Exception {
        var add = new AddThread();
        var dec = new DecThread();
        add.start();
        dec.start();
        add.join();
        dec.join();
        System.out.println(Counter.count);
    }
}

class Counter {
    public static final Object lock = new Object();
    public static int count = 0;
}

class AddThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.count += 1;
            }
        }
    }
}

class DecThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.count -= 1;
            }
        }
    }
}
```

▶ Run

注意到代码：

```
synchronized(Counter.lock) { // 获取锁
    ...
} // 释放锁
```

它表示用 `Counter.lock` 实例作为锁，两个线程在执行各自的 `synchronized(Counter.lock) { … }` 代码块时，必须先获得锁，才能进入代码块进行。执行结束后，在 `synchronized` 语句块结束会自动释放锁。这样一来，对 `Counter.count` 变量进行读写就不可能同时进行。上述代码无论运行多少次，最终结果都是0。

使用 `synchronized` 解决了多线程同步访问共享变量的正确性问题。但是，它的缺点是带来了性能下

降。因为 `synchronized` 代码块无法并发执行。此外，加锁和解锁需要消耗一定的时间，所以， `synchronized` 会降低程序的执行效率。

我们来概括一下如何使用 `synchronized` ：

- 找出修改共享变量的线程代码块；
- 选择一个共享实例作为锁；
- 使用 `synchronized(lockObject) { … }` 。在使用 `synchronized` 的时候，不必担心抛出异常。因为无论是否有异常，都会在 `synchronized` 结束处正确释放锁：

```
public void add(int m) {
    synchronized (obj) {
        if (m < 0) {
            throw new RuntimeException();
        }
        this.value += m;
    } // 无论有无异常，都会在此释放锁
}
```

我们再来看一个错误使用 `synchronized` 的例子：

```
// 多线程
public class Main {
    public static void main(String[] args) throws Exception {
        var add = new AddThread();
        var dec = new DecThread();
        add.start();
        dec.start();
        add.join();
        dec.join();
        System.out.println(Counter.count);
    }
}

class Counter {
    public static final Object lock1 = new Object();
    public static final Object lock2 = new Object();
    public static int count = 0;
}

class AddThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock1) {
                Counter.count += 1;
            }
        }
    }
}

class DecThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock2) {
                Counter.count -= 1;
            }
        }
    }
}
```

▶ Run

结果并不是0，这是因为两个线程各自的 `synchronized` 锁住的不是同一个对象！这使得两个线程各自都可以同时获得锁：因为JVM只保证同一个锁在任意时刻只能被一个线程获取，但两个不同的锁在同一时刻可以被两个线程分别获取。

因此，使用 `synchronized` 的时候，获取到的是哪个锁非常重要。锁对象如果不对，代码逻辑就不对。

我们再看一个例子：

```
// 多线程
public class Main {
    public static void main(String[] args) throws Exception {
        var ts = new Thread[] { new AddStudentThread(), new DecStudentThread(), new
AddTeacherThread(), new DecTeacherThread() };
        for (var t : ts) {
            t.start();
        }
        for (var t : ts) {
            t.join();
        }
        System.out.println(Counter.studentCount);
        System.out.println(Counter.teacherCount);
    }
}

class Counter {
    public static final Object lock = new Object();
    public static int studentCount = 0;
    public static int teacherCount = 0;
}

class AddStudentThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.studentCount += 1;
            }
        }
    }
}

class DecStudentThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.studentCount -= 1;
            }
        }
    }
}

class AddTeacherThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.teacherCount += 1;
            }
        }
    }
}

class DecTeacherThread extends Thread {
    public void run() {
        for (int i=0; i<10000; i++) {
            synchronized(Counter.lock) {
                Counter.teacherCount -= 1;
            }
        }
    }
}
```

▶ Run

上述代码的4个线程对两个共享变量分别进行读写操作，但是使用的锁都是 `Counter.lock` 这一个对象，这就造成了原本可以并发执行的 `Counter.studentCount += 1` 和 `Counter.teacherCount += 1` ，现在无法并发执行了，执行效率大大降低。实际上，需要同步的线程可以分成两组： `AddStudentThread` 和 `DecStudentThread` ， `AddTeacherThread` 和 `DecTeacherThread` ，组之间不存在竞争，因此，应该使用两个不同的锁，即：

`AddStudentThread` 和 `DecStudentThread` 使用 `lockStudent` 锁：

```
synchronized(Counter.lockStudent) {
    ...
}
```

`AddTeacherThread` 和 `DecTeacherThread` 使用 `lockTeacher` 锁：

```
synchronized(Counter.lockTeacher) {
    ...
}
```

这样才能最大化地提高执行效率。

不需要synchronized的操作

JVM规范定义了几种原子操作：

- 基本类型（ `long` 和 `double` 除外）赋值，例如： `int n = m` ；
- 引用类型赋值，例如： `List<String> list = anotherList` 。

`long` 和 `double` 是64位数据，JVM没有明确规定64位赋值操作是不是一个原子操作，不过在x64平台的JVM是把 `long` 和 `double` 的赋值作为原子操作实现的。

单条原子操作的语句不需要同步。例如：

```
public void set(int m) {
    synchronized(lock) {
        this.value = m;
    }
}
```

就不需要同步。

对引用也是类似。例如：

```
public void set(String s) {
    this.value = s;
}
```

上述赋值语句并不需要同步。

但是，如果是多行赋值语句，就必须保证是同步操作，例如：

```
class Pair {
    int first;
    int last;
    public void set(int first, int last) {
        synchronized(this) {
            this.first = first;
            this.last = last;
        }
    }
}
```

有些时候，通过一些巧妙的转换，可以把非原子操作变为原子操作。例如，上述代码如果改造成：

```
class Pair {
    int[] pair;
    public void set(int first, int last) {
        int[] ps = new int[] { first, last };
        this.pair = ps;
    }
}
```

就不再需要同步，因为 `this.pair = ps` 是引用赋值的原子操作。而语句：

```
int[] ps = new int[] { first, last };
```

这里的 `ps` 是方法内部定义的局部变量，每个线程都会有各自的局部变量，互不影响，并且互不可见，并不需要同步。

小结

多线程同时读写共享变量时，会造成逻辑错误，因此需要通过 `synchronized` 同步；

同步的本质就是给指定对象加锁，加锁后才能继续执行后续代码；

注意加锁对象必须是同一个实例；

对JVM定义的单个原子操作不需要同步。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





同步方法

我们知道Java程序依靠 `synchronized` 对线程进行同步，使用 `synchronized` 的时候，锁住的是哪个对象非常重要。

让线程自己选择锁对象往往会使得代码逻辑混乱，也不利于封装。更好的方法是把 `synchronized` 逻辑封装起来。例如，我们编写一个计数器如下：

```
public class Counter {
    private int count = 0;

    public void add(int n) {
        synchronized(this) {
            count += n;
        }
    }

    public void dec(int n) {
        synchronized(this) {
            count += n;
        }
    }

    public int get() {
        return count;
    }
}
```

这样一来，线程调用 `add()` 、 `dec()` 方法时，它不必关心同步逻辑，因为 `synchronized` 代码块在 `add()` 、 `dec()` 方法内部。并且，我们注意到， `synchronized` 锁住的对象是 `this` ，即当前实例，这又使得创建多个 `Counter` 实例的时候，它们之间互不影响，可以并发执行：

```
var c1 = Counter();
var c2 = Counter();

// 对c1进行操作的线程:
new Thread(() -> {
    c1.add();
}).start();
new Thread(() -> {
    c1.dec();
```

```
}).start();

// 对c2进行操作的线程:
new Thread(() -> {
    c2.add();
}).start();
new Thread(() -> {
    c2.dec();
}).start();
```

现在，对于 `Counter` 类，多线程可以正确调用。

如果一个类被设计为允许多线程正确访问，我们就说这个类就是“线程安全”的（thread-safe），上面的 `Counter` 类就是线程安全的。Java标准库的 `java.lang.StringBuffer` 也是线程安全的。

还有一些不变类，例如 `String` ， `Integer` ， `LocalDate` ，它们的所有成员变量都是 `final` ，多线程同时访问时只能读不能写，这些不变类也是线程安全的。

最后，类似 `Math` 这些只提供静态方法，没有成员变量的类，也是线程安全的。

除了上述几种少数情况，大部分类，例如 `ArrayList` ，都是非线程安全的类，我们不能在多线程中修改它们。但是，如果所有线程都只读取，不写入，那么 `ArrayList` 是可以安全地在线程间共享的。

没有特殊说明时，一个类默认是非线程安全的。

我们再观察 `Counter` 的代码：

```
public class Counter {
    public void add(int n) {
        synchronized(this) {
            count += n;
        }
    }
    ...
}
```

当我们锁住的是 `this` 实例时，实际上可以用 `synchronized` 修饰这个方法。下面两种写法是等价的：

```
public void add(int n) {
    synchronized(this) { // 锁住this
        count += n;
```

```
    } // 解锁
}
```

```
public synchronized void add(int n) { // 锁住this
    count += n;
} // 解锁
```

因此，用 `synchronized` 修饰的方法就是同步方法，它表示整个方法都必须用 `this` 实例加锁。

我们再思考一下，如果对一个静态方法添加 `synchronized` 修饰符，它锁住的是哪个对象？

```
public synchronized static void test(int n) {
    ...
}
```

对于 `static` 方法，是没有 `this` 实例的，因为 `static` 方法是针对类而不是实例。但是我们注意到任何一个类都有一个由JVM自动创建的 `Class` 实例，因此，对 `static` 方法添加 `synchronized` ，锁住的是该类的 `class` 实例。上述 `synchronized static` 方法实际上相当于：

```
public class Counter {
    public static void test(int n) {
        synchronized(Counter.class) {
            ...
        }
    }
}
```

我们再考察 `Counter` 的 `get()` 方法：

```
public class Counter {
    private int count;

    public int get() {
        return count;
    }
    ...
}
```

它没有同步，因为读一个 `int` 变量不需要同步。

然而，如果我们把代码稍微改一下，返回一个包含两个 `int` 的对象：

```
public class Counter {
    private int first;
    private int last;

    public Pair get() {
        Pair p = new Pair();
        p.first = first;
        p.last = last;
        return p;
    }
    ...
}
```

就必须要同步了。

小结

用 `synchronized` 修饰方法可以把整个方法变为同步代码块， `synchronized` 方法加锁对象是 `this` ；

通过合理的设计和数据封装可以让一个类变为“线程安全”；

一个类没有特殊说明，默认不是thread-safe;

多线程能否安全访问某个非线程安全的实例，需要具体问题具体分析。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





死锁

Java的线程锁是可重入的锁。

什么是可重入的锁？我们还是来看例子：

```
1.  public class Counter {
2.      private int count = 0;
3.
4.      public synchronized void add(int n) {
5.          if (n < 0) {
6.              dec(-n);
7.          } else {
8.              count += n;
9.          }
10.     }
11.
12.     public synchronized void dec(int n) {
13.         count += n;
14.     }
15. }
```

观察 `synchronized` 修饰的 `add()` 方法，一旦线程执行到 `add()` 方法内部，说明它已经获取了当前实例的 `this` 锁。如果传入的 `n < 0` ，将在 `add()` 方法内部调用 `dec()` 方法。由于 `dec()` 方法也需要获取 `this` 锁，现在问题来了：

对同一个线程，能否在获取到锁以后继续获取同一个锁？

答案是肯定的。JVM允许同一个线程重复获取同一个锁，这种能被同一个线程反复获取的锁，就叫做可重入锁。

由于Java的线程锁是可重入锁，所以，获取锁的时候，不但要判断是否是第一次获取，还要记录这是第几次获取。每获取一次锁，记录+1，每退出 `synchronized` 块，记录-1，减到0的时候，才会真正释放锁。

死锁

一个线程可以获取一个锁后，再继续获取另一个锁。例如：

```
1.  public void add(int m) {
2.      synchronized(lockA) { // 获得lockA的锁
3.          this.value += m;
```

```
        synchronized(lockB) { // 获得lockB的锁
            this.another += m;
        } // 释放lockB的锁
    } // 释放lockA的锁
}

public void dec(int m) {
    synchronized(lockB) { // 获得lockB的锁
        this.another -= m;
        synchronized(lockA) { // 获得lockA的锁
            this.value -= m;
        } // 释放lockA的锁
    } // 释放lockB的锁
}
```

在获取多个锁的时候，不同线程获取多个不同对象的锁可能导致死锁。对于上述代码，线程1和线程2如果分别执行 `add()` 和 `dec()` 方法时：

- 线程1：进入 `add()` ，获得 `lockA` ；
- 线程2：进入 `dec()` ，获得 `lockB` 。

随后：

- 线程1：准备获得 `lockB` ，失败，等待中；
- 线程2：准备获得 `lockA` ，失败，等待中。

此时，两个线程各自持有不同的锁，然后各自试图获取对方手里的锁，造成了双方无限等待下去，这就是死锁。

死锁发生后，没有任何机制能解除死锁，只能强制结束JVM进程。

因此，在编写多线程应用时，要特别注意防止死锁。因为死锁一旦形成，就只能强制结束进程。

那么我们应该如何避免死锁呢？答案是：线程获取锁的顺序要一致。即严格按照先获取 `lockA` ，再获取 `lockB` 的顺序，改写 `dec()` 方法如下：

```
public void dec(int m) {
    synchronized(lockA) { // 获得lockA的锁
        this.value -= m;
        synchronized(lockB) { // 获得lockB的锁
            this.another -= m;
        } // 释放lockB的锁
    } // 释放lockA的锁
}
```

练习

请观察死锁的代码输出，然后修复。

下载练习：死锁 （推荐使用IDE练习插件快速下载）

小结

Java的 `synchronized` 锁是可重入锁；

死锁产生的条件是多线程各自持有不同的锁，并互相试图获取对方已持有的锁，导致无限等待；

避免死锁的方法是多线程获取锁的顺序要一致。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用wait和notify

在Java程序中，`synchronized`解决了多线程竞争的问题。例如，对于一个任务管理器，多个线程同时往队列中添加任务，可以用`synchronized`加锁：

```
class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
    }
}
```

但是`synchronized`并没有解决多线程协调的问题。

仍然以上面的`TaskQueue`为例，我们再编写一个`getTask()`方法取出队列的第一个任务：

```
class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
    }

    public synchronized String getTask() {
        while (queue.isEmpty()) {
        }
        return queue.remove();
    }
}
```

上述代码看上去没有问题：`getTask()`内部先判断队列是否为空，如果为空，就循环等待，直到另一个线程往队列中放入了一个任务，`while()`循环退出，就可以返回队列的元素了。

但实际上`while()`循环永远不会退出。因为线程在执行`while()`循环时，已经在`getTask()`入口获取了`this`锁，其他线程根本无法调用`addTask()`，因为`addTask()`执行条件也是获取`this`锁。

因此，执行上述代码，线程会在`getTask()`中因为死循环而100%占用CPU资源。

如果深入思考一下，我们想要的执行效果是：

- 线程1可以调用 `addTask()` 不断往队列中添加任务；
- 线程2可以调用 `getTask()` 从队列中获取任务。如果队列为空，则 `getTask()` 应该等待，直到队列中至少有一个任务时再返回。

因此，多线程协调运行的原则就是：当条件不满足时，线程进入等待状态；当条件满足时，线程被唤醒，继续执行任务。

对于上述 `TaskQueue` ，我们先改造 `getTask()` 方法，在条件不满足时，线程进入等待状态：

```
1. public synchronized String getTask() {
2.     while (queue.isEmpty()) {
3.         this.wait();
4.     }
5.     return queue.remove();
6. }
```

当一个线程执行到 `getTask()` 方法内部的 `while` 循环时，它必定已经获取到了 `this` 锁，此时，线程执行 `while` 条件判断，如果条件成立（队列为空），线程将执行 `this.wait()` ，进入等待状态。

这里的关键是： `wait()` 方法必须在当前获取的锁对象上调用，这里获取的是 `this` 锁，因此调用 `this.wait()` 。

调用 `wait()` 方法后，线程进入等待状态， `wait()` 方法不会返回，直到将来某个时刻，线程从等待状态被其他线程唤醒后， `wait()` 方法才会返回，然后，继续执行下一条语句。

有些仔细的童鞋会指出：即使线程在 `getTask()` 内部等待，其他线程如果拿不到 `this` 锁，照样无法执行 `addTask()` ，肿么办？

这个问题的关键就在于 `wait()` 方法的执行机制非常复杂。首先，它不是一个普通的Java方法，而是定义在 `Object` 类的一个 `native` 方法，也就是由JVM的C代码实现的。其次，必须在 `synchronized` 块中才能调用 `wait()` 方法，因为 `wait()` 方法调用时，会释放线程获得的锁， `wait()` 方法返回后，线程又会重新试图获得锁。

因此，只能在锁对象上调用 `wait()` 方法。因为在 `getTask()` 中，我们获得了 `this` 锁，因此，只能在 `this` 对象上调用 `wait()` 方法：

```
1. public synchronized String getTask() {
2.     while (queue.isEmpty()) {
3.         // 释放this锁:
4.         this.wait();
5.         // 重新获取this锁
6.     }
```

```
7.      return queue.remove();
8.  }
```

当一个线程在 `this.wait()` 等待时，它就会释放 `this` 锁，从而使得其他线程能够在 `addTask()` 方法获得 `this` 锁。

现在我们面临第二个问题：如何让等待的线程被重新唤醒，然后从 `wait()` 方法返回？答案是在相同的锁对象上调用 `notify()` 方法。我们修改 `addTask()` 如下：

```
1.  public synchronized void addTask(String s) {
2.      this.queue.add(s);
3.      this.notify(); // 唤醒在this锁等待的线程
4.  }
```

注意到在往队列中添加了任务后，线程立刻对 `this` 锁对象调用 `notify()` 方法，这个方法会唤醒一个正在 `this` 锁等待的线程（就是在 `getTask()` 中位于 `this.wait()` 的线程），从而使得等待线程从 `this.wait()` 方法返回。

我们来看一个完整的例子：

```
import java.util.*;
public class Main {
    public static void main(String[] args) throws InterruptedException {
        var q = new TaskQueue();
        var ts = new ArrayList<Thread>();
        for (int i=0; i<5; i++) {
            var t = new Thread() {
                public void run() {
                    // 执行task:
                    while (true) {
                        try {
                            String s = q.getTask();
                            System.out.println("execute task: " + s);
                        } catch (InterruptedException e) {
                            return;
                        }
                    }
                }
            };
            t.start();
            ts.add(t);
        }
        var add = new Thread(() -> {
            for (int i=0; i<10; i++) {
                // 放入task:
                String s = "t-" + Math.random();
                System.out.println("add task: " + s);
                q.addTask(s);
                try { Thread.sleep(100); } catch(InterruptedException e) {}
            }
        });
        add.start();
        add.join();
        Thread.sleep(100);
        for (var t : ts) {
            t.interrupt();
        }
    }
}

class TaskQueue {
    Queue<String> queue = new LinkedList<>();

    public synchronized void addTask(String s) {
        this.queue.add(s);
        this.notifyAll();
    }

    public synchronized String getTask() throws InterruptedException {
        while (queue.isEmpty()) {
            this.wait();
        }
        return queue.remove();
    }
}
```

▶ Run

这个例子中，我们重点关注 `addTask()` 方法，内部调用了 `this.notifyAll()` 而不

是 `this.notify()` ，使用 `notifyAll()` 将唤醒所有当前正在 `this` 锁等待的线程，而 `notify()` 只会唤醒其中一个（具体哪个依赖操作系统，有一定的随机性）。这是因为可能有多个线程正在 `getTask()` 方法内部的 `wait()` 中等待，使用 `notifyAll()` 将一次性全部唤醒。通常来说， `notifyAll()` 更安全。有些时候，如果我们的代码逻辑考虑不周，用 `notify()` 会导致只唤醒了一个线程，而其他线程可能永远等待下去醒不过来了。

但是，注意到 `wait()` 方法返回时需要重新获得 `this` 锁。假设当前有3个线程被唤醒，唤醒后，首先要等待执行 `addTask()` 的线程结束此方法后，才能释放 `this` 锁，随后，这3个线程中只能有一个获取到 `this` 锁，剩下两个将继续等待。

再注意到我们在 `while()` 循环中调用 `wait()` ，而不是 `if` 语句：

```
public synchronized String getTask() throws InterruptedException {
    if (queue.isEmpty()) {
        this.wait();
    }
    return queue.remove();
}
```

这种写法实际上是错误的，因为线程被唤醒时，需要再次获取 `this` 锁。多个线程被唤醒后，只有一个线程能获取 `this` 锁，此刻，该线程执行 `queue.remove()` 可以获取到队列的元素，然而，剩下的线程如果获取 `this` 锁后执行 `queue.remove()` ，此刻队列可能已经没有任何元素了，所以，要始终在 `while` 循环中 `wait()` ，并且每次被唤醒后拿到 `this` 锁就必须再次判断：

```
while (queue.isEmpty()) {
    this.wait();
}
```

所以，正确编写多线程代码是非常困难的，需要仔细考虑的条件非常多，任何一个地方考虑不周，都会导致多线程运行时不正常。



小结

`wait` 和 `notify` 用于多线程协调运行：

- 在 `synchronized` 内部可以调用 `wait()` 使线程进入等待状态；
- 必须在已获得的锁对象上调用 `wait()` 方法；
- 在 `synchronized` 内部可以调用 `notify()` 或 `notifyAll()` 唤醒其他等待线程；
- 必须在已获得的锁对象上调用 `notify()` 或 `notifyAll()` 方法；
- 已唤醒的线程还需要重新获得锁后才能继续执行。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用ReentrantLock

从Java 5开始，引入了一个高级的处理并发的 `java.util.concurrent` 包，它提供了大量更高级的并发功能，能大大简化多线程程序的编写。

我们知道Java语言直接提供了 `synchronized` 关键字用于加锁，但这种锁一是很重，二是获取时必须一直等待，没有额外的尝试机制。

`java.util.concurrent.locks` 包提供的 `ReentrantLock` 用于替代 `synchronized` 加锁，我们来看一下传统的 `synchronized` 代码：

```
public class Counter {
    private int count;

    public void add(int n) {
        synchronized(this) {
            count += n;
        }
    }
}
```

如果用 `ReentrantLock` 替代，可以把代码改造为：

```
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int count;

    public void add(int n) {
        lock.lock();
        try {
            count += n;
        } finally {
            lock.unlock();
        }
    }
}
```

因为 `synchronized` 是Java语言层面提供的语法，所以我们不需要考虑异常，而 `ReentrantLock` 是Java代码实现的锁，我们就必须先获取锁，然后在 `finally` 中正确释放锁。

顾名思义， `ReentrantLock` 是可重入锁，它和 `synchronized` 一样，一个线程可以多次获取同一个锁。

和 `synchronized` 不同的是， `ReentrantLock` 可以尝试获取锁：

```
if (lock.tryLock(1, TimeUnit.SECONDS)) {
    try {
        ...
    } finally {
        lock.unlock();
    }
}
```

上述代码在尝试获取锁的时候，最多等待1秒。如果1秒后仍未获取到锁， `tryLock()` 返回 `false` ，程序就可以做一些额外处理，而不是无限等待下去。

所以，使用 `ReentrantLock` 比直接使用 `synchronized` 更安全，线程在 `tryLock()` 失败的时候不会导致死锁。

小结

`ReentrantLock` 可以替代 `synchronized` 进行同步；

`ReentrantLock` 获取锁更安全；

必须先获取到锁，再进入 `try {…}` 代码块，最后使用 `finally` 保证释放锁；

可以使用 `tryLock()` 尝试获取锁。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Condition

使用 `ReentrantLock` 比直接使用 `synchronized` 更安全，可以替代 `synchronized` 进行线程同步。

但是， `synchronized` 可以配合 `wait` 和 `notify` 实现线程在条件不满足时等待，条件满足时唤醒，用 `ReentrantLock` 我们怎么编写 `wait` 和 `notify` 的功能呢？

答案是使用 `Condition` 对象来实现 `wait` 和 `notify` 的功能。

我们仍然以 `TaskQueue` 为例，把前面用 `synchronized` 实现的功能通过 `ReentrantLock` 和 `Condition` 来实现：

```
class TaskQueue {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private Queue<String> queue = new LinkedList<>();

    public void addTask(String s) {
        lock.lock();
        try {
            queue.add(s);
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public String getTask() {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                condition.await();
            }
            return queue.remove();
        } finally {
            lock.unlock();
        }
    }
}
```

可见，使用 `Condition` 时，引用的 `Condition` 对象必须从 `Lock` 实例的 `newCondition()` 返

回，这样才能获得一个绑定了 `Lock` 实例的 `Condition` 实例。

`Condition` 提供的 `await()` 、 `signal()` 、 `signalAll()` 原理和 `synchronized` 锁对象的 `wait()` 、 `notify()` 、 `notifyAll()` 是一致的，并且其行为也是一样的：

- `await()` 会释放当前锁，进入等待状态；
- `signal()` 会唤醒某个等待线程；
- `signalAll()` 会唤醒所有等待线程；
- 唤醒线程从 `await()` 返回后需要重新获得锁。

此外，和 `tryLock()` 类似， `await()` 可以在等待指定时间后，如果还没有被其他线程通过 `signal()` 或 `signalAll()` 唤醒，可以自己醒来：

```
1. if (condition.await(1, TimeUnit.SECOND)) {
2.     // 被其他线程唤醒
3. } else {
4.     // 指定时间内没有被其他线程唤醒
5. }
```

可见，使用 `Condition` 配合 `Lock` ，我们可以实现更灵活的线程同步。

小结

`Condition` 可以替代 `wait` 和 `notify` ；

`Condition` 对象必须从 `Lock` 对象获取。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用ReadWriteLock

前面讲到的 `ReentrantLock` 保证了只有一个线程可以执行临界区代码：

```
1.  public class Counter {
2.      private final Lock lock = new ReentrantLock();
3.      private int[] counts = new int[10];
4.
5.      public void inc(int index) {
6.          lock.lock();
7.          try {
8.              counts[index] += 1;
9.          } finally {
10.             lock.unlock();
11.         }
12.     }
13.
14.     public int[] get() {
15.         lock.lock();
16.         try {
17.             return Arrays.copyOf(counts, counts.length);
18.         } finally {
19.             lock.unlock();
20.         }
21.     }
22. }
```

但是有些时候，这种保护有点过头。因为我们发现，任何时刻，只允许一个线程修改，也就是调用 `inc()` 方法是必须获取锁，但是， `get()` 方法只读取数据，不修改数据，它实际上允许多个线程同时调用。

实际上我们想要的是：允许多个线程同时读，但只要有一个线程在写，其他线程就必须等待：

	读	写
读	允许	不允许
写	不允许	不允许

使用 `ReadWriteLock` 可以解决这个问题，它保证：

- 只允许一个线程写入（其他线程既不能写入也不能读取）；
- 没有写入时，多个线程允许同时读（提高性能）。

用 `ReadWriteLock` 实现这个功能十分容易。我们需要创建一个 `ReadWriteLock` 实例，然后分别获取读锁和写锁：

```
public class Counter {
    private final ReadWriteLock rwlock = new ReentrantReadWriteLock();
    private final Lock rlock = rwlock.readLock();
    private final Lock wlock = rwlock.writeLock();
    private int[] counts = new int[10];

    public void inc(int index) {
        wlock.lock(); // 加写锁
        try {
            counts[index] += 1;
        } finally {
            wlock.unlock(); // 释放写锁
        }
    }

    public int[] get() {
        rlock.lock(); // 加读锁
        try {
            return Arrays.copyOf(counts, counts.length);
        } finally {
            rlock.unlock(); // 释放读锁
        }
    }
}
```

把读写操作分别用读锁和写锁来加锁，在读取时，多个线程可以同时获得读锁，这样就大大提高了并发读的执行效率。

使用 `ReadWriteLock` 时，适用条件是同一个数据，有大量线程读取，但仅有少数线程修改。

例如，一个论坛的帖子，回复可以看做写入操作，它是不频繁的，但是，浏览可以看做读取操作，是非常频繁的，这种情况就可以使用 `ReadWriteLock` 。

小结

使用 `ReadWriteLock` 可以提高读取效率：

- `ReadWriteLock` 只允许一个线程写入；
- `ReadWriteLock` 允许多个线程在没有写入时同时读取；

- `ReadWriteLock` 适合读多写少的场景。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用StampedLock

前面介绍的 `ReadWriteLock` 可以解决多线程同时读，但只有一个线程能写的问题。

如果我们深入分析 `ReadWriteLock` ，会发现它有个潜在的问题：如果有线程正在读，写线程需要等待读线程释放锁后才能获取写锁，即读的过程中不允许写，这是一种悲观的读锁。

要进一步提升并发执行效率，Java 8引入了新的读写锁： `StampedLock` 。

`StampedLock` 和 `ReadWriteLock` 相比，改进之处在于：读的过程中也允许获取写锁后写入！这样一来，我们读的数据就可能不一致，所以，需要一点额外的代码来判断读的过程中是否有写入，这种读锁是一种乐观锁。

乐观锁的意思就是乐观地估计读的过程中大概率不会有写入，因此被称为乐观锁。反过来，悲观锁则是读的过程中拒绝有写入，也就是写入必须等待。显然乐观锁的并发效率更高，但一旦有小概率的写入导致读取的数据不一致，需要能检测出来，再读一遍就行。

我们来看例子：

```
1.  public class Point {
2.      private final StampedLock stampedLock = new StampedLock();
3.
4.      private double x;
5.      private double y;
6.
7.      public void move(double deltaX, double deltaY) {
8.          long stamp = stampedLock.writeLock(); // 获取写锁
9.          try {
10.             x += deltaX;
11.             y += deltaY;
12.         } finally {
13.             stampedLock.unlockWrite(stamp); // 释放写锁
14.         }
15.     }
16.
17.     public double distanceFromOrigin() {
18.         long stamp = stampedLock.tryOptimisticRead(); // 获得一个乐观读锁
19.         // 注意下面两行代码不是原子操作
20.         // 假设x,y = (100,200)
21.         double currentX = x;
22.         // 此处已读取到x=100，但x,y可能被写线程修改为(300,400)
23.         double currentY = y;
24.         // 此处已读取到y，如果没有写入，读取是正确的(100,200)
```

```
25.         // 如果有写入，读取是错误的(100,400)
26.         if (!stampedLock.validate(stamp)) { // 检查乐观读锁后是否有其他写锁发生
27.             stamp = stampedLock.readLock(); // 获取一个悲观读锁
28.             try {
29.                 currentX = x;
30.                 currentY = y;
31.             } finally {
32.                 stampedLock.unlockRead(stamp); // 释放悲观读锁
33.             }
34.         }
35.         return Math.sqrt(currentX * currentX + currentY * currentY);
36.     }
37. }
```

和 `ReadWriteLock` 相比，写入的加锁是完全一样的，不同的是读取。注意到首先我们通过 `tryOptimisticRead()` 获取一个乐观读锁，并返回版本号。接着进行读取，读取完成后，我们通过 `validate()` 去验证版本号，如果在读取过程中没有写入，版本号不变，验证成功，我们就可以放心地继续后续操作。如果在读取过程中有写入，版本号会发生变化，验证将失败。在失败的时候，我们再通过获取悲观读锁再次读取。由于写入的概率不高，程序在绝大部分情况下可以通过乐观读锁获取数据，极少数情况下使用悲观读锁获取数据。

可见， `StampedLock` 把读锁细分为乐观读和悲观读，能进一步提升并发效率。但这也是有代价的：一是代码更加复杂，二是 `StampedLock` 是不可重入锁，不能在一个线程中反复获取同一个锁。

`StampedLock` 还提供了更复杂的将悲观读锁升级为写锁的功能，它主要使用在if-then-update的场景：即先读，如果读的数据满足条件，就返回，如果读的数据不满足条件，再尝试写。

小结

`StampedLock` 提供了乐观读锁，可取代 `ReadWriteLock` 以进一步提升并发性能；

`StampedLock` 是不可重入锁。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Concurrent集合

我们在前面已经通过 `ReentrantLock` 和 `Condition` 实现了一个 `BlockingQueue` ：

```
public class TaskQueue {
    private final Lock lock = new ReentrantLock();
    private final Condition condition = lock.newCondition();
    private Queue<String> queue = new LinkedList<>();

    public void addTask(String s) {
        lock.lock();
        try {
            queue.add(s);
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }

    public String getTask() {
        lock.lock();
        try {
            while (queue.isEmpty()) {
                condition.await();
            }
            return queue.remove();
        } finally {
            lock.unlock();
        }
    }
}
```

`BlockingQueue` 的意思就是说，当一个线程调用这个 `TaskQueue` 的 `getTask()` 方法时，该方法内部可能会让线程变成等待状态，直到队列条件满足不为空，线程被唤醒后， `getTask()` 方法才会返回。

因为 `BlockingQueue` 非常有用，所以我们不必自己编写，可以直接使用Java标准库的 `java.util.concurrent` 包提供的线程安全的集合： `ArrayBlockingQueue` 。

除了 `BlockingQueue` 外，针对 `List` 、 `Map` 、 `Set` 、 `Deque` 等， `java.util.concurrent` 包也提供了对应的并发集合类。我们归纳一下：

interface	non-thread-safe	thread-safe
List	ArrayList	CopyOnWriteArrayList
Map	HashMap	ConcurrentHashMap
Set	HashSet / TreeSet	CopyOnWriteArraySet
Queue	ArrayDeque / LinkedList	ArrayBlockingQueue / LinkedBlockingQueue
Deque	ArrayDeque / LinkedList	LinkedBlockingDeque

使用这些并发集合与使用非线程安全的集合类完全相同。我们以 `ConcurrentHashMap` 为例：

```
Map<String, String> map = ConcurrentHashMap<>();
// 在不同的线程读写:
map.put("A", "1");
map.put("B", "2");
map.get("A", "1");
```

因为所有的同步和加锁的逻辑都在集合内部实现，对外部调用者来说，只需要正常按接口引用，其他代码和原来的非线程安全代码完全一样。即当我们需要多线程访问时，把：

```
Map<String, String> map = HashMap<>();
```

改为：

```
Map<String, String> map = ConcurrentHashMap<>();
```

就可以了。

`java.util.Collections` 工具类还提供了一个旧的线程安全集合转换器，可以这么用：

```
Map unsafeMap = new HashMap();
Map threadSafeMap = Collections.synchronizedMap(unsafeMap);
```

但是它实际上是用一个包装类包装了非线程安全的 `Map` ，然后对所有读写方法都用 `synchronized` 加锁，这样获得的线程安全集合的性能比 `java.util.concurrent` 集合要低很多，所以不推荐使用。

小结

使用 `java.util.concurrent` 包提供的线程安全的并发集合可以大大简化多线程编程：

多线程同时读写并发集合是安全的；

尽量使用Java标准库提供的并发集合，避免自己编写同步代码。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Atomic

Java的 `java.util.concurrent` 包除了提供底层锁、并发集合外，还提供了一组原子操作的封装类，它们位于 `java.util.concurrent.atomic` 包。

我们以 `AtomicInteger` 为例，它提供的主要操作有：

- 增加值并返回新值： `int addAndGet(int delta)`
- 加1后返回新值： `int incrementAndGet()`
- 获取当前值： `int get()`
- 用CAS方式设置： `int compareAndSet(int expect, int update)`

Atomic类是通过无锁（lock-free）的方式实现的线程安全（thread-safe）访问。它的主要原理是利用了CAS：Compare and Set。

如果我们自己通过CAS编写 `incrementAndGet()` ，它大概长这样：

```
1. public int incrementAndGet(AtomicInteger var) {
2.     int prev, next;
3.     do {
4.         prev = var.get();
5.         next = prev + 1;
6.     } while ( ! var.compareAndSet(prev, next));
7.     return prev;
8. }
```

CAS是指，在这个操作中，如果 `AtomicInteger` 的当前值是 `prev` ，那么就更新为 `next` ，返回 `true` 。如果 `AtomicInteger` 的当前值不是 `prev` ，就什么也不干，返回 `false` 。通过CAS操作并配合 `do … while` 循环，即使其他线程修改了 `AtomicInteger` 的值，最终的结果也是正确的。

我们利用 `AtomicLong` 可以编写一个多线程安全的全局唯一ID生成器：

```
1. class IdGenerator {
2.     AtomicLong var = new AtomicLong(0);
3.
4.     public long getNextId() {
5.         return var.incrementAndGet();
6.     }
7. }
```

通常情况下，我们并不需要直接用 `do … while` 循环调用 `compareAndSet` 实现复杂的并发操作，

而是用 `incrementAndGet()` 这样的封装好的方法，因此，使用起来非常简单。

在高度竞争的情况下，还可以使用Java 8提供的 `LongAdder` 和 `LongAccumulator` 。

小结

使用 `java.util.concurrent.atomic` 提供的原子操作可以简化多线程编程：

- 原子操作实现了无锁的线程安全；
- 适用于计数器，累加器等。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用线程池

Java语言虽然内置了多线程支持，启动一个新线程非常方便，但是，创建线程需要操作系统资源（线程资源，栈空间等），频繁创建和销毁大量线程需要消耗大量时间。

如果可以复用一组线程：

```
┌────┐ execute  ┌─────────────────────┐
|Task1|─────────>|ThreadPool           |
├────┤          |┌──────┐ ┌──────┐    |
|Task2|          ||Thread1||Thread2||   |
├────┤          |└──────┘ └──────┘    |
|Task3|          |┌──────┐ ┌──────┐    |
├────┤          ||Thread3||Thread4||   |
|Task4|          |└──────┘ └──────┘    |
├────┤          └─────────────────────┘
|Task5|
├────┤
|Task6|
└────┘
  ...
```

那么我们就可以把很多小任务让一组线程来执行，而不是一个任务对应一个新线程。这种能接收大量小任务并进行分发处理的就是线程池。

简单地说，线程池内部维护了若干个线程，没有任务的时候，这些线程都处于等待状态。如果有新任务，就分配一个空闲线程执行。如果所有线程都处于忙碌状态，新任务要么放入队列等待，要么增加一个新线程进行处理。

Java标准库提供了 `ExecutorService` 接口表示线程池，它的典型用法如下：

```
// 创建固定大小的线程池:
ExecutorService executor = Executors.newFixedThreadPool(3);
// 提交任务:
executor.submit(task1);
executor.submit(task2);
executor.submit(task3);
executor.submit(task4);
executor.submit(task5);
```

因为 `ExecutorService` 只是接口，Java标准库提供的几个常用实现类有：

- FixedThreadPool：线程数固定的线程池；
- CachedThreadPool：线程数根据任务动态调整的线程池；
- SingleThreadExecutor：仅单线程执行的线程池。

创建这些线程池的方法都被封装到 `Executors` 这个类中。我们以 `FixedThreadPool` 为例，看看线程池的执行逻辑：

```
// thread-pool
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) {
        // 创建一个固定大小的线程池:
        ExecutorService es = Executors.newFixedThreadPool(4);
        for (int i = 0; i < 6; i++) {
            es.submit(new Task("" + i));
        }
        // 关闭线程池:
        es.shutdown();
    }
}

class Task implements Runnable {
    private final String name;

    public Task(String name) {
        this.name = name;
    }

    @Override
    public void run() {
        System.out.println("start task " + name);
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        System.out.println("end task " + name);
    }
}
```

Run

我们观察执行结果，一次性放入6个任务，由于线程池只有固定的4个线程，因此，前4个任务会同时执行，等到有线程空闲后，才会执行后面的两个任务。

线程池在程序结束的时候要关闭。使用 `shutdown()` 方法关闭线程池的时候，它会等待正在执行的任务先完成，然后再关闭。 `shutdownNow()` 会立刻停止正在执行的任务， `awaitTermination()` 则会等待指定的时间让线程池关闭。

如果我们把线程池改为 `CachedThreadPool` ，由于这个线程池的实现会根据任务数量动态调整线程池的大小，所以6个任务可一次性全部同时执行。

如果我们想把线程池的大小限制在4～10个之间动态调整怎么办？我们查看 `Executors.newCachedThreadPool()` 方法的源码：

```
public static ExecutorService newCachedThreadPool() {
    return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
                                  60L, TimeUnit.SECONDS,
                                  new SynchronousQueue<Runnable>());
}
```

因此，想创建指定动态范围的线程池，可以这么写：

```
int min = 4;
int max = 10;
ExecutorService es = new ThreadPoolExecutor(min, max,
        60L, TimeUnit.SECONDS, new SynchronousQueue<Runnable>());
```

ScheduledThreadPool

还有一种任务，需要定期反复执行，例如，每秒刷新证券价格。这种任务本身固定，需要反复执行的，可以使用 `ScheduledThreadPool` 。放入 `ScheduledThreadPool` 的任务可以定期反复执行。

创建一个 `ScheduledThreadPool` 仍然是通过 `Executors` 类：

```
ScheduledExecutorService ses = Executors.newScheduledThreadPool(4);
```

我们可以提交一次性任务，它会在指定延迟后只执行一次：

```
// 1秒后执行一次性任务:
ses.schedule(new Task("one-time"), 1, TimeUnit.SECONDS);
```

如果任务以固定的每3秒执行，我们可以这样写：

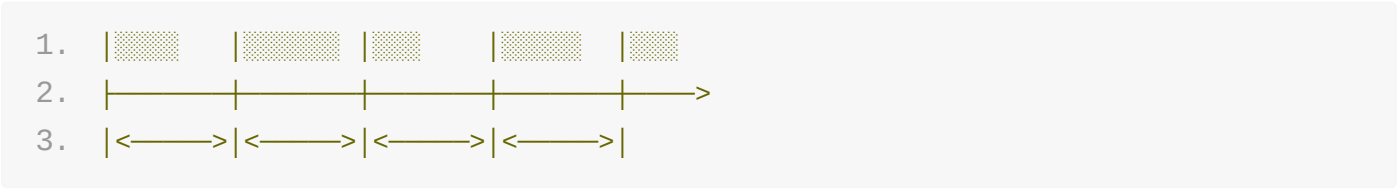
```
// 2秒后开始执行定时任务，每3秒执行:
ses.scheduleAtFixedRate(new Task("fixed-rate"), 2, 3, TimeUnit.SECONDS);
```

如果任务以固定的3秒为间隔执行，我们可以这样写：

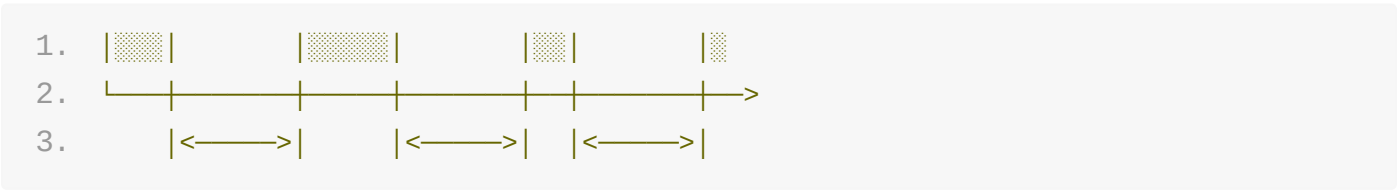
```
// 3秒后开始执行定时任务，以3秒为间隔执行:
ses.scheduleWithFixedDelay(new Task("fixed-delay"), 2, 3, TimeUnit.SECONDS);
```

注意FixedRate和FixedDelay的区别。FixedRate是指任务总是以固定时间间隔触发，不管任务执

行多长时间：



而FixedDelay是指，上一次任务执行完毕后，等待固定的时间间隔，再执行下一次任务：



因此，使用 `ScheduledThreadPool` 时，我们要根据需要选择执行一次、FixedRate执行还是FixedDelay执行。

细心的童鞋还可以思考下面的问题：

- 在FixedRate模式下，假设每秒触发，如果某次任务执行时间超过1秒，后续任务会不会并发执行？
- 如果任务抛出了异常，后续任务是否继续执行？

Java标准库还提供了一个 `java.util.Timer` 类，这个类也可以定期执行任务，但是，一个 `Timer` 会对应一个 `Thread` ，所以，一个 `Timer` 只能定期执行一个任务，多个定时任务必须启动多个 `Timer` ，而一个 `ScheduledThreadPool` 就可以调度多个定时任务，所以，我们完全可以用 `ScheduledThreadPool` 取代旧的 `Timer` 。

练习

下载练习：使用线程池 （推荐使用IDE练习插件快速下载）

小结

JDK提供了 `ExecutorService` 实现了线程池功能：

- 线程池内部维护一组线程，可以高效执行大量小任务；
- `Executors` 提供了静态方法创建不同类型的 `ExecutorService` ；
- 必须调用 `shutdown()` 关闭 `ExecutorService` ；
- `ScheduledThreadPool` 可以定期调度多个任务。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Future

在执行多个任务的时候，使用Java标准库提供的线程池是非常方便的。我们提交的任务只需要实现 `Runnable` 接口，就可以让线程池去执行：

```
class Task implements Runnable {
    public String result;

    public void run() {
        this.result = longTimeCalculation();
    }
}
```

`Runnable` 接口有个问题，它的方法没有返回值。如果任务需要一个返回结果，那么只能保存到变量，还要提供额外的方法读取，非常不便。所以，Java标准库还提供了一个 `Callable` 接口，和 `Runnable` 接口比，它多了一个返回值：

```
class Task implements Callable<String> {
    public String call() throws Exception {
        return longTimeCalculation();
    }
}
```

并且 `Callable` 接口是一个泛型接口，可以返回指定类型的结果。

现在的问题是，如何获得异步执行的结果？

如果仔细看 `ExecutorService.submit()` 方法，可以看到，它返回了一个 `Future` 类型，一个 `Future` 类型的实例代表一个未来能获取结果的对象：

```
ExecutorService executor = Executors.newFixedThreadPool(4);
// 定义任务:
Callable<String> task = new Task();
// 提交任务并获得Future:
Future<String> future = executor.submit(task);
// 从Future获取异步执行返回的结果:
String result = future.get(); // 可能阻塞
```

当我们提交一个 `Callable` 任务后，我们会同时获得一个 `Future` 对象，然后，我们在主线程某个时刻调用 `Future` 对象的 `get()` 方法，就可以获得异步执行的结果。在调用 `get()` 时，如果异步任务已经完成，我们就直接获得结果。如果异步任务还没有完成，那么 `get()` 会阻塞，直到任务完

成后才返回结果。

一个 `Future<V>` 接口表示一个未来可能会返回的结果，它定义的方法有：

- `get()`：获取结果（可能会等待）
- `get(long timeout, TimeUnit unit)`：获取结果，但只等待指定的时间；
- `cancel(boolean mayInterruptIfRunning)`：取消当前任务；
- `isDone()`：判断任务是否已完成。

练习

下载练习：使用Future （推荐使用IDE练习插件快速下载）

小结

对线程池提交一个 `Callable` 任务，可以获得一个 `Future` 对象；

可以用 `Future` 在将来某个时刻获取结果。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用CompletableFuture

使用 `Future` 获得异步执行结果时，要么调用阻塞方法 `get()` ，要么轮询看 `isDone()` 是否为 `true` ，这两种方法都不是很好，因为主线程也会被迫等待。

从Java 8开始引入了 `CompletableFuture` ，它针对 `Future` 做了改进，可以传入回调对象，当异步任务完成或者发生异常时，自动调用回调对象的回调方法。

我们以获取股票价格为例，看看如何使用 `CompletableFuture` ：

```
// CompletableFuture
import java.util.concurrent.CompletableFuture;

public class Main {
    public static void main(String[] args) throws Exception {
        // 创建异步执行任务:
        CompletableFuture<Double> cf = CompletableFuture.supplyAsync(Main::fetchPrice);
        // 如果执行成功:
        cf.thenAccept((result) -> {
            System.out.println("price: " + result);
        });
        // 如果执行异常:
        cf.exceptionally((e) -> {
            e.printStackTrace();
            return null;
        });
        // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭:
        Thread.sleep(2000);
    }

    static Double fetchPrice() {
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        if (Math.random() < 0.3) {
            throw new RuntimeException("fetch price failed!");
        }
        return 5 + Math.random() * 20;
    }
}
```

▶ Run

创建一个 `CompletableFuture` 是通过 `CompletableFuture.supplyAsync()` 实现的，它需要一个实现了 `Supplier` 接口的对象：

```
1. public interface Supplier<T> {
2.     T get();
3. }
```

这里我们用lambda语法简化了一下，直接传入 `Main::fetchPrice` ，因为 `Main.fetchPrice()` 静态方法的签名符合 `Supplier` 接口的定义（除了方法名外）。

紧接着， `CompletableFuture` 已经被提交给默认的线程池执行了，我们需要定义的是 `CompletableFuture` 完成时和异常时需要回调的实例。完成时， `CompletableFuture` 会调用 `Consumer` 对象：

```
1. public interface Consumer<T> {
2.     void accept(T t);
3. }
```

异常时， `CompletableFuture` 会调用 `Function` 对象：

```
1. public interface Function<T, R> {
2.     R apply(T t);
3. }
```

这里我们都用lambda语法简化了代码。

可见 `CompletableFuture` 的优点是：

- 异步任务结束时，会自动回调某个对象的方法；
- 异步任务出错时，会自动回调某个对象的方法；
- 主线程设置好回调后，不再关心异步任务的执行。

如果只是实现了异步回调机制，我们还看不出 `CompletableFuture` 相比 `Future` 的优势。 `CompletableFuture` 更强大的功能是，多个 `CompletableFuture` 可以串行执行，例如，定义两个 `CompletableFuture` ，第一个 `CompletableFuture` 根据证券名称查询证券代码，第二个 `CompletableFuture` 根据证券代码查询证券价格，这两个 `CompletableFuture` 实现串行操作如下：

```
// CompletableFuture
import java.util.concurrent.CompletableFuture;
```

```
public class Main {
    public static void main(String[] args) throws Exception {
        // 第一个任务:
        CompletableFuture<String> cfQuery = CompletableFuture.supplyAsync(() -> {
            return queryCode("中国石油");
        });
        // cfQuery成功后继续执行下一个任务:
        CompletableFuture<Double> cfFetch = cfQuery.thenApplyAsync((code) -> {
            return fetchPrice(code);
        });
        // cfFetch成功后打印结果:
        cfFetch.thenAccept((result) -> {
            System.out.println("price: " + result);
        });
        // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭:
        Thread.sleep(2000);
    }

    static String queryCode(String name) {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        }
        return "601857";
    }

    static Double fetchPrice(String code) {
        try {
            Thread.sleep(500);
        } catch (InterruptedException e) {
        }
        return 5 + Math.random() * 20;
    }
}
```

▶ Run

除了串行执行外，多个 `CompletableFuture` 还可以并行执行。例如，我们考虑这样的场景：

同时从新浪和网易查询证券代码，只要任意一个返回结果，就进行下一步查询价格，查询价格也同时从新浪和网易查询，只要任意一个返回结果，就完成操作：

```
// CompletableFuture
import java.util.concurrent.CompletableFuture;

public class Main {
    public static void main(String[] args) throws Exception {
        // 两个CompletableFuture执行异步查询:
        CompletableFuture<String> cfQueryFromSina = CompletableFuture.supplyAsync(() -> {
            return queryCode("中国石油", "https://finance.sina.com.cn/code/");
        });
        CompletableFuture<String> cfQueryFrom163 = CompletableFuture.supplyAsync(() -> {
            return queryCode("中国石油", "https://money.163.com/code/");
        });

        // 用anyOf合并为一个新的CompletableFuture:
        CompletableFuture<Object> cfQuery = CompletableFuture.anyOf(cfQueryFromSina, cfQueryFrom163);

        // 两个CompletableFuture执行异步查询:
        CompletableFuture<Double> cfFetchFromSina = cfQuery.thenApplyAsync((code) -> {
            return fetchPrice((String) code, "https://finance.sina.com.cn/price/");
        });
        CompletableFuture<Double> cfFetchFrom163 = cfQuery.thenApplyAsync((code) -> {
            return fetchPrice((String) code, "https://money.163.com/price/");
        });

        // 用anyOf合并为一个新的CompletableFuture:
        CompletableFuture<Object> cfFetch = CompletableFuture.anyOf(cfFetchFromSina, cfFetchFrom163);

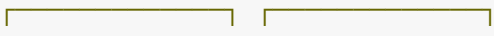
        // 最终结果:
        cfFetch.thenAccept((result) -> {
            System.out.println("price: " + result);
        });
        // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭:
        Thread.sleep(2000);
    }

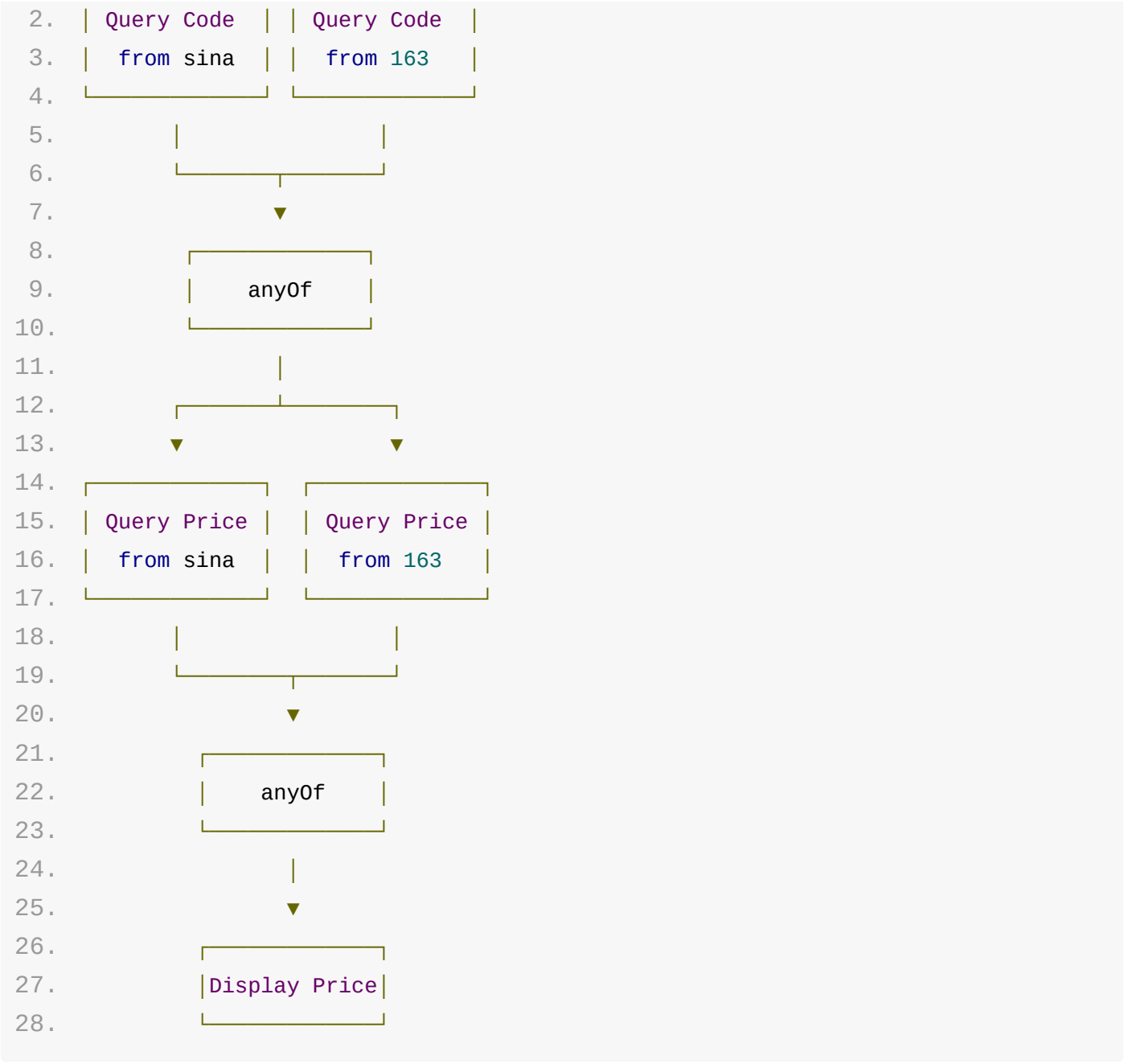
    static String queryCode(String name, String url) {
        System.out.println("query code from " + url + "...");
        try {
            Thread.sleep((long) Math.random() * 1000);
        } catch (InterruptedException e) {
        }
        return "601857";
    }

    static Double fetchPrice(String code, String url) {
        System.out.println("query price from " + url + "...");
        try {
            Thread.sleep((long) Math.random() * 1000);
        } catch (InterruptedException e) {
        }
        return 5 + Math.random() * 20;
    }
}
```

▶ Run

上述逻辑实现的异步查询规则实际上是：

1. 



除了 `anyOf()` 可以实现“任意个 `CompletableFuture` 只要一个成功”，`allOf()` 可以实现“所有 `CompletableFuture` 都必须成功”，这些组合操作可以实现非常复杂的异步流程控制。

最后我们注意 `CompletableFuture` 的命名规则：

- `xxx()`：表示该方法将继续在已有的线程中执行；
- `xxxAsync()`：表示将异步在线程池中执行。

练习

下载练习：使用CompletableFuture （推荐使用IDE练习插件快速下载）

小结

`CompletableFuture` 可以指定异步处理流程：

- `thenAccept()` 处理正常结果；
- `exceptional()` 处理异常结果；
- `thenApplyAsync()` 用于串行化另一个 `CompletableFuture` ；
- `anyOf()` 和 `allOf()` 用于并行化多个 `CompletableFuture` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





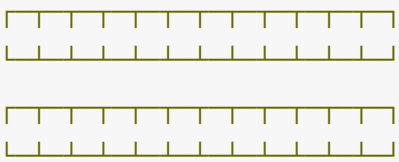
使用ForkJoin

Java 7开始引入了一种新的Fork/Join线程池，它可以执行一种特殊的任务：把一个大任务拆成多个小任务并行执行。

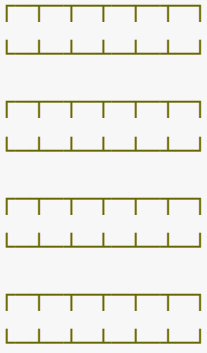
我们举个例子：如果要计算一个超大数组的和，最简单的做法是用一个循环在一个线程内完成：



还有一种方法，可以把数组拆成两部分，分别计算，最后加起来就是最终结果，这样可以用两个线程并行执行：



如果拆成两部分还是很大，我们还可以继续拆，用4个线程并行执行：



这就是Fork/Join任务的原理：判断一个任务是否足够小，如果是，直接计算，否则，就分拆成几个小任务分别计算。这个过程可以反复“裂变”成一系列小任务。

我们来看如何使用Fork/Join对大数据进行并行求和：

```
import java.util.Random;
import java.util.concurrent.*;

public class Main {
    public static void main(String[] args) throws Exception {
        // 创建2000个随机数组成的数组:
        long[] array = new long[2000];
        long expectedSum = 0;
        for (int i = 0; i < array.length; i++) {
            array[i] = random();
            expectedSum += array[i];
        }
        System.out.println("Expected sum: " + expectedSum);
        // fork/join:
        ForkJoinTask<Long> task = new SumTask(array, 0, array.length);
        long startTime = System.currentTimeMillis();
        Long result = ForkJoinPool.commonPool().invoke(task);
        long endTime = System.currentTimeMillis();
        System.out.println("Fork/join sum: " + result + " in " + (endTime - startTime) + " ms.");
    }

    static Random random = new Random(0);

    static long random() {
        return random.nextInt(10000);
    }
}

class SumTask extends RecursiveTask<Long> {
    static final int THRESHOLD = 500;
    long[] array;
    int start;
    int end;

    SumTask(long[] array, int start, int end) {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    @Override
    protected Long compute() {
        if (end - start <= THRESHOLD) {
            // 如果任务足够小,直接计算:
            long sum = 0;
            for (int i = start; i < end; i++) {
                sum += this.array[i];
                // 故意放慢计算速度:
                try {
                    Thread.sleep(1);
                } catch (InterruptedException e) {
                }
            }
            return sum;
        }
        // 任务太大,一分为二:
        int middle = (end + start) / 2;
        System.out.println(String.format("split %d-%d ==> %d-%d, %d-%d", start, end, start, middle,
middle, end));
        SumTask subtask1 = new SumTask(this.array, start, middle);
        SumTask subtask2 = new SumTask(this.array, middle, end);
        invokeAll(subtask1, subtask2);
        Long subresult1 = subtask1.join();
        Long subresult2 = subtask2.join();
        Long result = subresult1 + subresult2;
        System.out.println("result = " + subresult1 + " + " + subresult2 + " ==> " + result);
        return result;
    }
}
```

▶ Run

观察上述代码的执行过程，一个大的计算任务0~2000首先分裂为两个小任务0~1000和1000~2000，这两个小任务仍然太大，继续分裂为更小的0~500，500~1000，1000~1500，1500~2000，最后，计算结果被依次合并，得到最终结果。

因此，核心代码 `SumTask` 继承自 `RecursiveTask` ，在 `compute()` 方法中，关键是如何“分裂”出子任务并且提交子任务：

```
class SumTask extends RecursiveTask<Long> {
    protected Long compute() {
        // “分裂”子任务:
        SumTask subtask1 = new SumTask(...);
        SumTask subtask2 = new SumTask(...);
        // invokeAll会并行运行两个子任务:
        invokeAll(subtask1, subtask2);
        // 获得子任务的结果:
        Long result1 = fork1.join();
        Long result2 = fork2.join();
        // 汇总结果:
        return result1 + result2;
    }
}
```

Fork/Join线程池在Java标准库中就有应用。Java标准库提供的 `java.util.Arrays.parallelSort(array)` 可以进行并行排序，它的原理就是内部通过Fork/Join对大数组分拆进行并行排序，在多核CPU上就可以大大提高排序的速度。

练习

下载练习：使用Fork/Join （推荐使用IDE练习插件快速下载）

小结

Fork/Join是一种基于“分治”的算法：通过分解任务，并行执行，最后合并结果得到最终结果。

`ForkJoinPool` 线程池可以把一个大任务分拆成小任务并行执行，任务类必须继承自 `RecursiveTask` 或 `RecursiveAction` 。

使用Fork/Join模式可以进行并行计算以提高效率。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用ThreadLocal

多线程是Java实现多任务的基础，`Thread`对象代表一个线程，我们可以在代码中调用`Thread.currentThread()`获取当前线程。例如，打印日志时，可以同时打印出当前线程的名字：

```
// Thread
public class Main {
    public static void main(String[] args) throws Exception {
        log("start main...");
        new Thread(() -> {
            log("run task...");
        }).start();
        new Thread(() -> {
            log("print...");
        }).start();
        log("end main.");
    }

    static void log(String s) {
        System.out.println(Thread.currentThread().getName() + ": " + s);
    }
}
```

▶ Run

对于多任务，Java标准库提供的线程池可以方便地执行这些任务，同时复用线程。Web应用程序就是典型的多任务应用，每个用户请求页面时，我们都会创建一个任务，类似：

```
1.  public void process(User user) {
2.      checkPermission();
3.      doWork();
4.      saveStatus();
5.      sendResponse();
6.  }
```

然后，通过线程池去执行这些任务。

观察`process()`方法，它内部需要调用若干其他方法，同时，我们遇到一个问题：如何在一个线程内传递状态？

`process()`方法需要传递的状态就是`User`实例。有的童鞋会想，简单地传入`User`就可以了：

```
1.  public void process(User user) {
2.      checkPermission(user);
3.      doWork(user);
```

```
    saveStatus(user);
    sendResponse(user);
}
```

但是往往一个方法又会调用其他很多方法，这样会导致 `User` 传递到所有地方：

```
void doWork(User user) {
    queryStatus(user);
    checkStatus();
    setNewStatus(user);
    log();
}
```

这种在一个线程中，横跨若干方法调用，需要传递的对象，我们通常称之为上下文（Context），它是一种状态，可以是用户身份、任务信息等。

给每个方法增加一个context参数非常麻烦，而且有些时候，如果调用链有无法修改源码的第三方库， `User` 对象就传不进去了。

Java标准库提供了一个特殊的 `ThreadLocal` ，它可以在一个线程中传递同一个对象。

`ThreadLocal` 实例通常总是以静态字段初始化如下：

```
static ThreadLocal<String> threadLocalUser = new ThreadLocal<>();
```

它的典型使用方式如下：

```
void processUser(user) {
    try {
        threadLocalUser.set(user);
        step1();
        step2();
    } finally {
        threadLocalUser.remove();
    }
}
```

通过设置一个 `User` 实例关联到 `ThreadLocal` 中，在移除之前，所有方法都可以随时获取到该 `User` 实例：

```
void step1() {
    User u = threadLocalUser.get();
```

```
    log();
    printUser();
}

void log() {
    User u = threadLocalUser.get();
    println(u.name);
}

void step2() {
    User u = threadLocalUser.get();
    checkUser(u.id);
}
```

注意到普通的方法调用一定是同一个线程执行的，所以，`step1()`、`step2()`以及`log()`方法内，`threadLocalUser.get()`获取的`User`对象是同一个实例。

实际上，可以把`ThreadLocal`看成一个全局`Map<Thread, Object>`：每个线程获取`ThreadLocal`变量时，总是使用`Thread`自身作为key：

```
Object threadLocalValue = threadLocalMap.get(Thread.currentThread());
```

因此，`ThreadLocal`相当于给每个线程都开辟了一个独立的存储空间，各个线程的`ThreadLocal`关联的实例互不干扰。

最后，特别注意`ThreadLocal`一定要在`finally`中清除：

```
try {
    threadLocalUser.set(user);
    ...
} finally {
    threadLocalUser.remove();
}
```

这是因为当前线程执行完相关代码后，很可能会被重新放入线程池中，如果`ThreadLocal`没有被清除，该线程执行其他代码时，会把上一次的状态带进去。

为了保证能释放`ThreadLocal`关联的实例，我们可以通过`AutoCloseable`接口配合`try (resource) {…}`结构，让编译器自动为我们关闭。例如，一个保存了当前用户名的`ThreadLocal`可以封装为一个`UserContext`对象：

```
public class UserContext implements AutoCloseable {
```

```
    static final ThreadLocal<String> ctx = new ThreadLocal<>();

    public UserContext(String user) {
        ctx.set(user);
    }

    public static String currentUser() {
        return ctx.get();
    }

    @Override
    public void close() {
        ctx.remove();
    }
}
```

使用的时候，我们借助 `try (resource) {…}` 结构，可以这么写：

```
try (var ctx = new UserContext("Bob")) {
    // 可任意调用UserContext.currentUser():
    String currentUser = UserContext.currentUser();
} // 在此自动调用UserContext.close()方法释放ThreadLocal关联对象
```

这样就在 `UserContext` 中完全封装了 `ThreadLocal` ，外部代码在 `try (resource) {…}` 内部可以随时调用 `UserContext.currentUser()` 获取当前线程绑定的用户名。

练习

下载练习：ThreadLocal练习 （推荐使用IDE练习插件快速下载）

小结

`ThreadLocal` 表示线程的“局部变量”，它确保每个线程的 `ThreadLocal` 变量都是各自独立的；

`ThreadLocal` 适合在一个线程的处理流程中保持上下文（避免了同一参数在所有方法中传递）；

使用 `ThreadLocal` 要用 `try … finally` 结构，并在 `finally` 中清除。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论

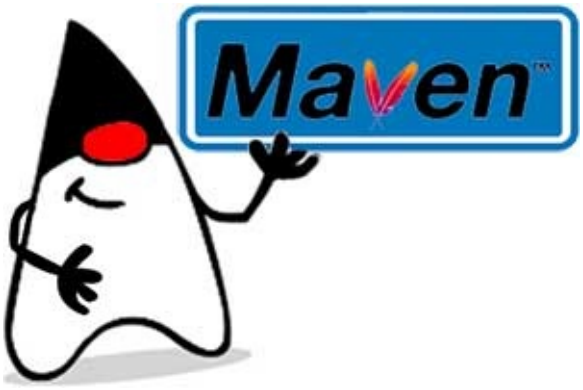




Maven基础

Maven是一个Java项目管理和构建工具，它可以定义项目结构、项目依赖，并使用统一的方式进行自动化构建，是Java项目不可缺少的工具。

本章我们详细介绍如何使用Maven。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





Maven介绍

在了解Maven之前，我们先来看看一个Java项目需要的东西。首先，我们需要确定引入哪些依赖包。例如，如果我们需要用到commons logging，我们就必须把commons logging的jar包放入classpath。如果我们还需要log4j，就需要把log4j相关的jar包都放到classpath中。这些就是依赖包的管理。

其次，我们要确定项目的目录结构。例如，`src`目录存放Java源码，`resources`目录存放配置文件，`bin`目录存放编译生成的`.class`文件。

此外，我们还需要配置环境，例如JDK的版本，编译打包的流程，当前代码的版本号。

最后，除了使用Eclipse这样的IDE进行编译外，我们还必须能通过命令行工具进行编译，才能够让项目在一个独立的服务器上编译、测试、部署。

这些工作难度不大，但是非常琐碎且耗时。如果每一个项目都自己搞一套配置，肯定会一团糟。我们需要的是一个标准化的Java项目管理和构建工具。

Maven就是是专门为Java项目打造的管理和构建工具，它的主要功能有：

- 提供了一套标准化的项目结构；
- 提供了一套标准化的构建流程（编译，测试，打包，发布……）；
- 提供了一套依赖管理机制。

Maven项目结构

一个使用Maven管理的普通的Java项目，它的目录结构默认如下：

```
a-maven-project
├── pom.xml
├── src
│   ├── main
│   │   ├── java
│   │   └── resources
│   └── test
│       ├── java
│       └── resources
└── target
```

项目的根目录`a-maven-project`是项目名，它有一个项目描述文件`pom.xml`，存放Java源码的目录是`src/main/java`，存放资源文件的目录是`src/main/resources`，存放测试源码的目录是`src/test/java`，存放测试资源的目录是`src/test/resources`，最后，所有编译、打包生成

的文件都放在 `target` 目录里。这些就是一个Maven项目的标准目录结构。

所有的目录结构都是约定好的标准结构，我们千万不要随意修改目录结构。使用标准结构不需要做任何配置，Maven就可以正常使用。

我们再来看最关键的一个项目描述文件 `pom.xml` ，它的内容长得像下面：

```
<project ...>
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>hello</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>
    <properties>
        ...
    </properties>
    <dependencies>
        <dependency>
            <groupId>commons-logging</groupId>
            <artifactId>commons-logging</artifactId>
            <version>1.2</version>
        </dependency>
    </dependencies>
</project>
```

其中， `groupId` 类似于Java的包名，通常是公司或组织名称， `artifactId` 类似于Java的类名，通常是项目名称，再加上 `version` ，一个Maven工程就是由 `groupId` ， `artifactId` 和 `version` 作为唯一标识。我们在引用其他第三方库的时候，也是通过这3个变量确定。例如，依赖 `commons-logging` ：

```
<dependency>
    <groupId>commons-logging</groupId>
    <artifactId>commons-logging</artifactId>
    <version>1.2</version>
</dependency>
```

使用 `<dependency>` 声明一个依赖后，Maven就会自动下载这个依赖包并把它放到classpath中。

安装Maven

要安装Maven，可以从Maven官网下载最新的Maven 3.6.x，然后在本地解压，设置几个环境变量：

```
M2_HOME=/path/to/maven-3.6.x
PATH=$PATH:$M2_HOME/bin
```

Windows可以把 `%M2_HOME%\bin` 添加到系统Path变量中。

然后，打开命令行窗口，输入 `mvn -version` ，应该看到Maven的版本信息：

```
┌──────────────────────────────────────────────────────────┐
|Command Prompt                                  - □ x |
├──────────────────────────────────────────────────────────┤
|Microsoft Windows [Version 10.0.0]                       |
|(c) 2015 Microsoft Corporation. All rights reserved.     |
|                                                         |
|C:\> mvn -version                                        |
|Apache Maven 3.6.0 (97c98ec64a1fdfee7767ce5ffb20918...) |
|Maven home: C:\Users\liaoxuefeng\maven                   |
|Java version: ...                                        |
|...                                                      |
|C:\> _                                                   |
|                                                         |
|                                                         |
└──────────────────────────────────────────────────────────┘
```

如果提示命令未找到，说明系统PATH路径有误，需要修复后再运行。

小结

Maven是一个Java项目的管理和构建工具：

- Maven使用 `pom.xml` 定义项目内容，并使用预设的目录结构；
- 在Maven中声明一个依赖项可以自动下载并导入classpath；
- Maven使用 `groupId` ， `artifactId` 和 `version` 唯一定位一个依赖。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





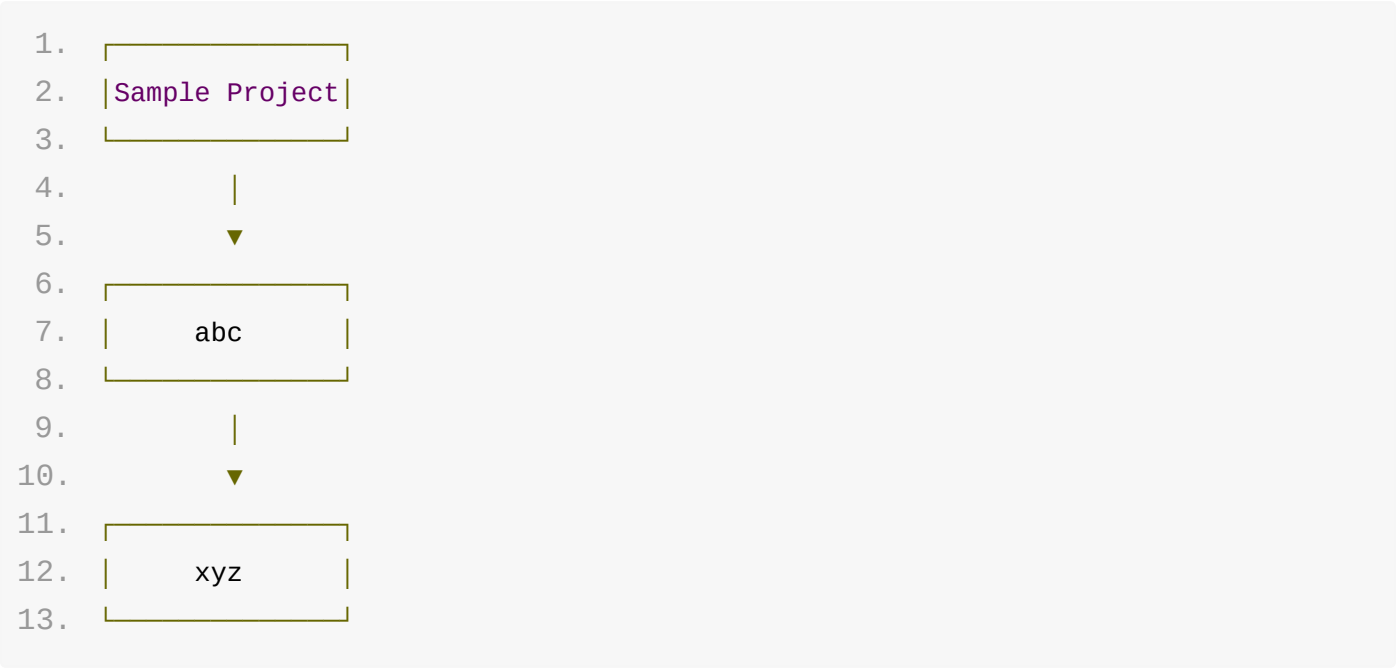
依赖管理

如果我们的项目依赖第三方的jar包，例如commons logging，那么问题来了：commons logging发布的jar包在哪下载？

如果我们还希望依赖log4j，那么使用log4j需要哪些jar包？

类似的依赖还包括：JUnit，JavaMail，MySQL驱动等等，一个可行的方法是通过搜索引擎搜索到项目的官网，然后手动下载zip包，解压，放入classpath。但是，这个过程非常繁琐。

Maven解决了依赖管理问题。例如，我们的项目依赖 `abc` 这个jar包，而 `abc` 又依赖 `xyz` 这个jar包：



当我们声明了 `abc` 的依赖时，Maven自动把 `abc` 和 `xyz` 都加入了我们的项目依赖，不需要我们自己去研究 `abc` 是否需要依赖 `xyz` 。

因此，Maven的第一个作用就是解决依赖管理。我们声明了自己的项目需要 `abc` ，Maven会自动导入 `abc` 的jar包，再判断出 `abc` 需要 `xyz` ，又会自动导入 `xyz` 的jar包，这样，最终我们的项目会依赖 `abc` 和 `xyz` 两个jar包。

我们来看一个复杂依赖示例：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>1.4.2.RELEASE</version>
</dependency>
```

当我们声明一个 `spring-boot-starter-web` 依赖时，Maven会自动解析并判断最终需要大概二三十个其他依赖：

```
spring-boot-starter-web
  spring-boot-starter
    spring-boot
    sprint-boot-autoconfigure
    spring-boot-starter-logging
      logback-classic
        logback-core
        slf4j-api
      jcl-over-slf4j
        slf4j-api
      jul-to-slf4j
        slf4j-api
      log4j-over-slf4j
        slf4j-api
    spring-core
    snakeyaml
  spring-boot-starter-tomcat
    tomcat-embed-core
    tomcat-embed-el
    tomcat-embed-websocket
      tomcat-embed-core
  jackson-databind
  ...
```

如果我们自己去手动管理这些依赖是非常费时费力的，而且出错的概率很大。

依赖关系

Maven定义了几种依赖关系，分别是 `compile` 、 `test` 、 `runtime` 和 `provided` ：

scope	说明	示例
compile	编译时需要用到该jar包（默认）	commons-logging
test	编译Test时需要用到该jar包	junit
runtime	编译时不需要，但运行时需要用到	mysql
provided	编译时需要用到，但运行时由JDK或某个服务器提供	servlet-api

其中，默认的 `compile` 是最常用的，Maven会把这种类型的依赖直接放入classpath。

`test` 依赖表示仅在测试时使用，正常运行时并不需要。最常用的 `test` 依赖就是JUnit：

```
<dependency>
    <groupId>org.junit.jupiter</groupId>
    <artifactId>junit-jupiter-api</artifactId>
    <version>5.3.2</version>
    <scope>test</scope>
</dependency>
```

`runtime` 依赖表示编译时不需要，但运行时需要。最典型的 `runtime` 依赖是JDBC驱动，例如MySQL驱动：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.48</version>
    <scope>runtime</scope>
</dependency>
```

`provided` 依赖表示编译时需要，但运行时不需要。最典型的 `provided` 依赖是Servlet API，编译的时候需要，但是运行时，Servlet服务器内置了相关的jar，所以运行期不需要：

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.0</version>
    <scope>provided</scope>
</dependency>
```

最后一个问题是，Maven如何知道从何处下载所需的依赖？也就是相关的jar包？答案是Maven维护了一个中央仓库（repo1.maven.org），所有第三方库将自身的jar以及相关信息上传至中央仓库，Maven就可以从中央仓库把所需依赖下载到本地。

Maven并不会每次都从中央仓库下载jar包。一个jar包一旦被下载过，就会被Maven自动缓存在本地目录（用户主目录的 `.m2` 目录），所以，除了第一次编译时因为下载需要时间会比较慢，后续过程因为有本地缓存，并不会重复下载相同的jar包。

唯一ID

对于某个依赖，Maven只需要3个变量即可唯一确定某个jar包：

- groupId：属于组织的名称，类似Java的包名；
- artifactId：该jar包自身的名称，类似Java的类名；

- version：该jar包的版本。

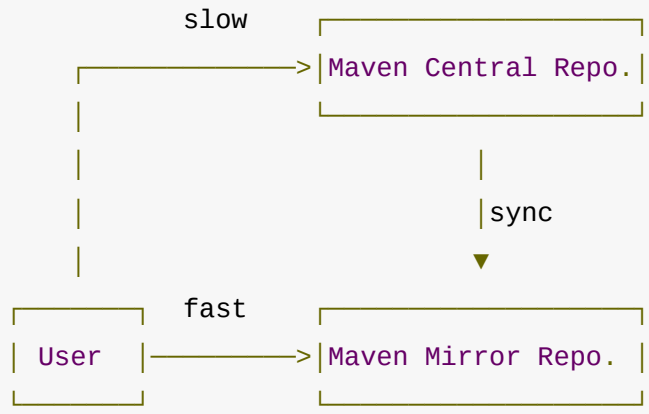
通过上述3个变量，即可唯一确定某个jar包。Maven通过对jar包进行PGP签名确保任何一个jar包一经发布就无法修改。修改已发布jar包的唯一方法是发布一个新版本。

因此，某个jar包一旦被Maven下载过，即可永久地安全缓存在本地。

注：只有以 `SNAPSHOT-` 开头的版本号会被Maven视为开发版本，开发版本每次都会重复下载，这种SNAPSHOT版本只能用于内部私有的Maven repo，公开发布的版本不允许出现SNAPSHOT。

Maven镜像

除了可以从Maven的中央仓库下载外，还可以从Maven的镜像仓库下载。如果访问Maven的中央仓库非常慢，我们可以选择一个速度较快的Maven的镜像仓库。Maven镜像仓库定期从中央仓库同步：



中国区用户可以使用阿里云提供的Maven镜像仓库。使用Maven镜像仓库需要一个配置，在用户主目录下进入 `.m2` 目录，创建一个 `settings.xml` 配置文件，内容如下：

```
<settings>
    <mirrors>
        <mirror>
            <id>aliyun</id>
            <name>aliyun</name>
            <mirrorOf>central</mirrorOf>
            <!-- 国内推荐阿里云的Maven镜像 -->
            <url>http://maven.aliyun.com/nexus/content/groups/public/</url>
        </mirror>
    </mirrors>
</settings>
```

配置镜像仓库后，Maven的下载速度就会非常快。

搜索第三方组件

最后一个问题：如果我们要引用一个第三方组件，比如 `okhttp` ，如何确切地获得它的 `groupId` 、 `artifactId` 和 `version` ？方法是通过search.maven.org搜索关键字，找到对应的组件后，直接复制：



练习

下载练习：使用Maven编译hello项目 （推荐使用IDE练习插件快速下载）

小结

Maven通过解析依赖关系确定项目所需的jar包，常用的4种 `scope` 有： `compile` （默认）， `test` ， `runtime` 和 `provided` ；

Maven从中央仓库下载所需的jar包并缓存在本地；

可以通过镜像仓库加速下载。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





构建流程

构建流程

Maven不但有标准化的项目结构，而且还有一套标准化的构建流程，可以自动化实现编译，打包，发布，等等。

Lifecycle和Phase

使用Maven时，我们首先要了解什么是Maven的生命周期（lifecycle）。

Maven的生命周期由一系列阶段（phase）构成，以内置的生命周期 `default` 为例，它包含以下phase:

- validate
- initialize
- generate-sources
- process-sources
- generate-resources
- process-resources
- compile
- process-classes
- generate-test-sources
- process-test-sources
- generate-test-resources
- process-test-resources
- test-compile
- process-test-classes
- test
- prepare-package
- package
- pre-integration-test
- integration-test
- post-integration-test
- verify
- install
- deploy

如果我们运行 `mvn package` ，Maven就会执行 `default` 生命周期，它会从开始一直运行到 `package` 这个phase为止:

- validate
- …
- package

如果我们运行 `mvn compile` ，Maven也会执行 `default` 生命周期，但这次它只会运行到 `compile` ，即以下几个phase:

- validate
- …
- compile

Maven另一个常用的生命周期是 `clean` ，它会执行3个phase:

- pre-clean
- clean （注意这个clean不是lifecycle而是phase）
- post-clean

所以，我们使用 `mvn` 这个命令时，后面的参数是phase，Maven自动根据生命周期运行到指定的phase。

更复杂的例子是指定多个phase，例如，运行 `mvn clean package` ，Maven先执行 `clean` 生命周期并运行到 `clean` 这个phase，然后执行 `default` 生命周期并运行到 `package` 这个phase，实际执行的phase如下:

- pre-clean
- clean （注意这个clean是phase）
- validate
- …
- package

在实际开发过程中，经常使用的命令有:

`mvn clean` ：清理所有生成的class和jar;

`mvn clean compile` ：先清理，再执行到 `compile` ;

`mvn clean test` ：先清理，再执行到 `test` ，因为执行 `test` 前必须执行 `compile` ，所以这里不必指定 `compile` ;

`mvn clean package` ：先清理，再执行到 `package` 。

大多数phase在执行过程中，因为我们通常没有在 `pom.xml` 中配置相关的设置，所以这些phase什么事情都不做。

经常用到的phase其实只有几个:

- clean：清理
- compile：编译
- test：运行测试
- package：打包

Goal

执行一个phase又会触发一个或多个goal：

执行的Phase	对应执行的Goal
compile	compiler:compile
test	compiler:testCompile surefile:test

goal的命名总是 `abc:xyz` 这种形式。

看到这里，相信大家对lifecycle、phase和goal已经明白了吧？



其实我们类比一下就明白了：

- lifecycle相当于Java的package，它包含一个或多个phase；
- phase相当于Java的class，它包含一个或多个goal；
- goal相当于class的method，它其实才是真正干活的。

大多数情况，我们只要指定phase，就默认执行这些phase默认绑定的goal，只有少数情况，我们可以直接指定运行一个goal，例如，启动Tomcat服务器：

```
1. mvn tomcat:run
```

小结

Maven通过lifecycle、phase和goal来提供标准的构建流程。

最常用的构建命令是指定phase，然后让Maven执行到指定的phase：

- mvn clean
- mvn clean compile
- mvn clean test
- mvn clean package

通常情况，我们总是执行phase默认绑定的goal，因此不必指定goal。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用插件

我们在前面介绍了Maven的lifecycle，phase和goal：使用Maven构建项目就是执行lifecycle，执行到指定的phase为止。每个phase会执行自己默认的一个或多个goal。goal是最小任务单元。

我们以 `compile` 这个phase为例，如果执行：

```
1. mvn compile
```

Maven将执行 `compile` 这个phase，这个phase会调用 `compiler` 插件执行关联的 `compiler:compile` 这个goal。

实际上，执行每个phase，都是通过某个插件（plugin）来执行的，Maven本身其实并不知道如何执行 `compile` ，它只是负责找到对应的 `compiler` 插件，然后执行默认的 `compiler:compile` 这个goal来完成编译。

所以，使用Maven，实际上就是配置好需要使用的插件，然后通过phase调用它们。

Maven已经内置了一些常用的标准插件：

插件名称	对应执行的phase
clean	clean
compiler	compile
surefire	test
jar	package

如果标准插件无法满足需求，我们还可以使用自定义插件。使用自定义插件的时候，需要声明。例如，使用 `maven-shade-plugin` 可以创建一个可执行的jar，要使用这个插件，需要在 `pom.xml` 中声明它：

```
1. <project>
2.     ...
3.     <build>
4.         <plugins>
5.             <plugin>
6.                 <groupId>org.apache.maven.plugins</groupId>
7.                 <artifactId>maven-shade-plugin</artifactId>
8.                 <version>3.2.1</version>
9.                 <executions>
10.                    <execution>
11.                        <phase>package</phase>
12.                        <goals>
```

```
                            <goal>shade</goal>
                        </goals>
                        <configuration>
                            ...
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

自定义插件往往需要一些配置，例如， `maven-shade-plugin` 需要指定Java程序的入口，它的配置是：

```
<configuration>
    <transformers>
        <transformer
implementation="org.apache.maven.plugins.shade.resource.ManifestResourceTransform
            <mainClass>com.itranswarp.learnjava.Main</mainClass>
        </transformer>
    </transformers>
</configuration>
```

注意，Maven自带的标准插件例如 `compiler` 是无需声明的，只有引入其它的插件才需要声明。

下面列举了一些常用的插件：

- maven-shade-plugin：打包所有依赖包并生成可执行jar；
- cobertura-maven-plugin：生成单元测试覆盖率报告；
- findbugs-maven-plugin：对Java源码进行静态分析以找出潜在问题。

练习

下载练习：使用maven-shade-plugin创建可执行jar （推荐使用IDE练习插件快速下载）

小结

Maven通过自定义插件可以执行项目构建时需要的额外功能，使用自定义插件必须在pom.xml中声明插件及配置；

插件会在某个phase被执行时执行；

插件的配置和用法需参考插件的官方文档。

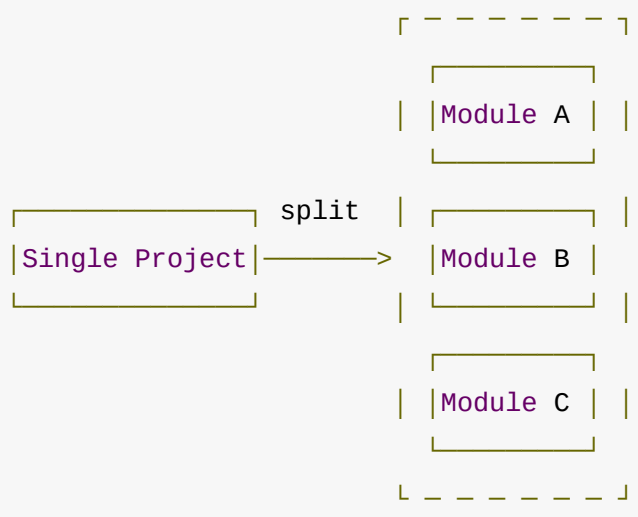
读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





模块管理

在软件开发中，把一个大项目分拆为多个模块是降低软件复杂度的有效方法：



对于Maven工程来说，原来是一个大项目：

```
single-project
├── pom.xml
└── src
```

现在可以分拆成3个模块：

```
single-project
├── module-a
│   ├── pom.xml
│   └── src
├── module-b
│   ├── pom.xml
│   └── src
└── module-c
    ├── pom.xml
    └── src
```

Maven可以有效地管理多个模块，我们只需要把每个模块当作一个独立的Maven项目，它们有各自独立的pom.xml。例如，模块A的pom.xml：

模块B的pom.xml：

可以看出来，模块A和模块B的pom.xml高度相似，因此，我们可以提取出共同部分作为parent：

注意到parent的packaging是pom而不是jar，因为parent本身不含任何java代码。编写parent的pom.xml只是为了在各个模块中减少重复的配置。现在我们的整个工程结构如下：

```
single-project
├── parent
│   └── pom.xml
├── module-a
│   ├── pom.xml
│   └── src
├── module-b
│   ├── pom.xml
│   └── src
└── module-c
    ├── pom.xml
    └── src
```

如果模块A依赖模块B，则模块A需要模块B的jar包才能正常编译：

中央仓库

其实我们使用的大多数第三方模块都是这个用法，例如，我们使用commons logging、log4j这些第三方模块，就是第三方模块的开发者自己把编译好的jar包发布到maven的中央仓库中。

私有仓库

本地仓库

但是我们不推荐把自己的模块安装到maven的本地仓库，因为每次修改模块b的源码，都需要重新安装，容易出现版本不一致的情况

推荐的做法是模块化编译，在编译的时候，告诉maven几个模块之间存在依赖关系，需要一块编译，maven就会自动按依赖顺序编译这些模块

```
<modules>
    <module>模块A</module>
    <module>模块B</module>
    <module>模块C</module>
</modules>
```

Maven支持模块化管理，可以把一个大项目拆成几个模块可以通过继承在parent的pom.xml统一定义重复配置可以通过 `<modules>` 编译多个模块

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用mvnw

我们使用Maven时，基本上只会用到 `mvn` 这一个命令。有些童鞋可能听说过 `mvnw` ，这个是啥？

`mvnw` 是Maven Wrapper的缩写。因为我们安装Maven时，默认情况下，系统所有项目都会使用全局安装的这个Maven版本。但是，对于某些项目来说，它可能必须使用某个特定的Maven版本，这个时候，就可以使用Maven Wrapper，它可以负责给这个特定的项目安装指定版本的Maven，而其他项目不受影响。

简单地说，Maven Wrapper就是给一个项目提供一个独立的，指定版本的Maven给它使用。

安装Maven Wrapper

安装Maven Wrapper最简单的方式是在项目的根目录（即 `pom.xml` 所在的目录）下运行安装命令：

```
mvn -N io.takari:maven:0.7.6:wrapper
```

它会自动使用最新版本的Maven。注意 `0.7.6` 是Maven Wrapper的版本。最新的Maven Wrapper版本可以去官方网站查看。

如果要指定使用的Maven版本，使用下面的安装命令指定版本，例如 `3.3.3` ：

```
mvn -N io.takari:maven:0.7.6:wrapper -Dmaven=3.3.3
```

安装后，查看项目结构：

```
my-project
├── .mvn
│   └── wrapper
│       ├── MavenWrapperDownloader.java
│       ├── maven-wrapper.jar
│       └── maven-wrapper.properties
├── mvnw
├── mvnw.cmd
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   └── resources
    └── test
        ├── java
        └── resources
```

发现多了 `mvnw` 、 `mvnw.cmd` 和 `.mvn` 目录，我们只需要把 `mvn` 命令改成 `mvnw` 就可以使用跟项目关联的Maven。例如：

```
1.  mvnw clean package
```

在Linux或macOS下运行时需要加上 `./` ：

```
1.  ./mvnw clean package
```

Maven Wrapper的另一个作用是把项目的 `mvnw` 、 `mvnw.cmd` 和 `.mvn` 提交到版本库中，可以使所有开发人员使用统一的Maven版本。

练习

下载练习：使用mvnw编译hello项目 （推荐使用IDE练习插件快速下载）

小结

使用Maven Wrapper，可以为一个项目指定特定的Maven版本。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





网络编程

网络编程是Java最擅长的方向之一，使用Java进行网络编程时，由虚拟机实现了底层复杂的网络协议，Java程序只需要调用Java标准库提供的接口，就可以简单高效地编写网络程序。

本章我们详细介绍如何使用Java进行网络编程。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





网络编程基础

在学习Java网络编程之前，我们先来了解什么是计算机网络。

计算机网络是指两台或更多的计算机组成的网络，在同一个网络中，任意两台计算机都可以直接通信，因为所有计算机都需要遵循同一种网络协议。

那什么是互联网呢？互联网是网络的网络（internet），即把很多计算机网络连接起来，形成一个全球统一的互联网。

对某个特定的计算机网络来说，它可能使用网络协议ABC，而另一个计算机网络可能使用网络协议XYZ。如果计算机网络各自的通讯协议不统一，就没法把不同的网络连接起来形成互联网。因此，为了把计算机网络接入互联网，就必须使用TCP/IP协议。

TCP/IP协议泛指互联网协议，其中最重要的两个协议是TCP协议和IP协议。只有使用TCP/IP协议的计算机才能够联入互联网，使用其他网络协议（例如NetBIOS、AppleTalk协议等）是无法联入互联网的。

IP地址

在互联网中，一个IP地址用于唯一标识一个网络接口（Network Interface）。一台联入互联网的计算机肯定有一个IP地址，但也可能有多个IP地址。

IP地址分为IPv4和IPv6两种。IPv4采用32位地址，类似 `101.202.99.12` ，而IPv6采用128位地址，类似 `2001:0DA8:100A:0000:0000:1020:F2F3:1428` 。IPv4地址总共有2^{32}个（大约42亿），而IPv6地址则总共有2^{128}个（大约340万亿亿亿亿），IPv4的地址目前已耗尽，而IPv6的地址是根本用不完的。

IP地址又分为公网IP地址和内网IP地址。公网IP地址可以直接被访问，内网IP地址只能在内网访问。内网IP地址类似于：

- 192.168.x.x
- 10.x.x.x

有一个特殊的IP地址，称之为本机地址，它总是 `127.0.0.1` 。

IPv4地址实际上是一个32位整数。例如：

```
106717964 = 0x65ca630c
          = 65  ca  63 0c
          = 101.202.99.12
```

如果一台计算机只有一个网卡，并且接入了网络，那么，它有一个本机地址 `127.0.0.1` ，还有一个

IP地址，例如 `101.202.99.12` ，可以通过这个IP地址接入网络。

如果一台计算机有两块网卡，那么除了本机地址，它可以有两个IP地址，可以分别接入两个网络。通常连接两个网络的设备是路由器或者交换机，它至少有两个IP地址，分别接入不同的网络，让网络之间连接起来。

如果两台计算机位于同一个网络，那么他们之间可以直接通信，因为他们的IP地址前段是相同的，也就是网络号是相同的。网络号是IP地址通过子网掩码过滤后得到的。例如：

某台计算机的IP是 `101.202.99.2` ，子网掩码是 `255.255.255.0` ，那么计算该计算机的网络号是：

```
IP = 101.202.99.2
Mask = 255.255.255.0
Network = IP & Mask = 101.202.99.0
```

每台计算机都需要正确配置IP地址和子网掩码，根据这两个就可以计算网络号，如果两台计算机计算出的网络号相同，说明两台计算机在同一个网络，可以直接通信。如果两台计算机计算出的网络号不同，那么两台计算机不在同一个网络，不能直接通信，它们之间必须通过路由器或者交换机这样的网络设备间接通信，我们把这种设备称为网关。

网关的作用就是连接多个网络，负责把来自一个网络的数据包发到另一个网络，这个过程叫路由。

所以，一台计算机的一个网卡会有3个关键配置：



- IP地址，例如： `10.0.2.15`

- 子网掩码，例如：`255.255.255.0`
- 网关的IP地址，例如：`10.0.2.2`

域名

因为直接记忆IP地址非常困难，所以我们通常使用域名访问某个特定的服务。域名解析服务器DNS负责把域名翻译成对应的IP，客户端再根据IP地址访问服务器。

用`nslookup`可以查看域名对应的IP地址：

```
$ nslookup www.liaoxuefeng.com
Server:  xxx.xxx.xxx.xxx
Address: xxx.xxx.xxx.xxx#53

Non-authoritative answer:
Name:    www.liaoxuefeng.com
Address: 47.98.33.223
```

有一个特殊的本机域名`localhost`，它对应的IP地址总是本机地址`127.0.0.1`。

网络模型

由于计算机网络从底层的传输到高层的软件设计十分复杂，要合理地设计计算机网络模型，必须采用分层模型，每一层负责处理自己的操作。OSI（Open System Interconnect）网络模型是ISO组织定义的一个计算机互联的标准模型，注意它只是一个定义，目的是为了简化网络各层的操作，提供标准接口便于实现和维护。这个模型从上到下依次是：

- 应用层，提供应用程序之间的通信；
- 表示层：处理数据格式，加解密等等；
- 会话层：负责建立和维护会话；
- 传输层：负责提供端到端的可靠传输；
- 网络层：负责根据目标地址选择路由来传输数据；
- 链路层和物理层负责把数据进行分片并且真正通过物理网络传输，例如，无线网、光纤等。

互联网实际使用的TCP/IP模型并不是对应到OSI的7层模型，而是大致对应OSI的5层模型：

OSI	TCP/IP
应用层	应用层
表示层	
会话层	
传输层	传输层
网络层	IP层

链路层	网络接口层
物理层	

常用协议

IP协议是一个分组交换，它不保证可靠传输。而TCP协议是传输控制协议，它是面向连接的协议，支持可靠传输和双向通信。TCP协议是建立在IP协议之上的，简单地说，IP协议只负责发数据包，不保证顺序和正确性，而TCP协议负责控制数据包传输，它在传输数据之前需要先建立连接，建立连接后才能传输数据，传输完后还需要断开连接。TCP协议之所以能保证数据的可靠传输，是通过接收确认、超时重传这些机制实现的。并且，TCP协议允许双向通信，即通信双方可以同时发送和接收数据。

TCP协议也是应用最广泛的协议，许多高级协议都是建立在TCP协议之上的，例如HTTP、SMTP等。

UDP协议（User Datagram Protocol）是一种数据报文协议，它是无连接协议，不保证可靠传输。因为UDP协议在通信前不需要建立连接，因此它的传输效率比TCP高，而且UDP协议比TCP协议要简单得多。

选择UDP协议时，传输的数据通常是能容忍丢失的，例如，一些语音视频通信的应用会选择UDP协议。

小结

计算机网络的基本概念主要有：

- 计算机网络：由两台或更多计算机组成的网络；
- 互联网：连接网络的网络；
- IP地址：计算机的网络接口（通常是网卡）在网络中的唯一标识；
- 网关：负责连接多个网络，并在多个网络之间转发数据的计算机，通常是路由器或交换机；
- 网络协议：互联网使用TCP/IP协议，它泛指互联网协议簇；
- IP协议：一种分组交换传输协议；
- TCP协议：一种面向连接，可靠传输的协议；
- UDP协议：一种无连接，不可靠传输的协议。

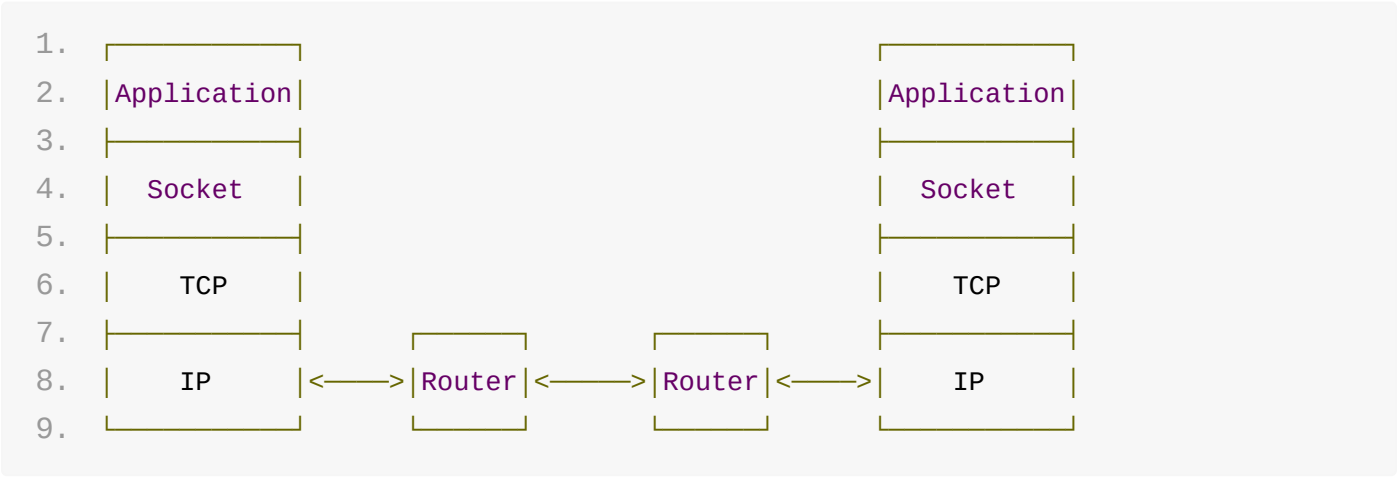
读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





TCP编程

在开发网络应用程序的时候，我们又会遇到Socket这个概念。Socket是一个抽象概念，一个应用程序通过一个Socket来建立一个远程连接，而Socket内部通过TCP/IP协议把数据传输到网络：



Socket、TCP和部分IP的功能都是由操作系统提供的，不同的编程语言只是提供了对操作系统调用的简单的封装。例如，Java提供的几个Socket相关的类就封装了操作系统提供的接口。

为什么需要Socket进行网络通信？因为仅仅通过IP地址进行通信是不够的，同一台计算机同一时间会运行多个网络应用程序，例如浏览器、QQ、邮件客户端等。当操作系统接收到一个数据包的时候，如果只有IP地址，它没法判断应该发给哪个应用程序，所以，操作系统抽象出Socket接口，每个应用程序需要各自对应到不同的Socket，数据包才能根据Socket正确地发到对应的应用程序。

一个Socket就是由IP地址和端口号（范围是0～65535）组成，可以把Socket简单理解为IP地址加端口号。端口号总是由操作系统分配，它是一个0～65535之间的数字，其中，小于1024的端口属于特权端口，需要管理员权限，大于1024的端口可以由任意用户的应用程序打开。

- 101.202.99.2:1201
- 101.202.99.2:1304
- 101.202.99.2:15000

使用Socket进行网络编程时，本质上就是两个进程之间的网络通信。其中一个进程必须充当服务器端，它会主动监听某个指定的端口，另一个进程必须充当客户端，它必须主动连接服务器的IP地址和指定端口，如果连接成功，服务器端和客户端就成功地建立了一个TCP连接，双方后续就可以随时发送和接收数据。

因此，当Socket连接成功地在服务器端和客户端之间建立后：

- 对服务器端来说，它的Socket是指定的IP地址和指定的端口号；
- 对客户端来说，它的Socket是它所在计算机的IP地址和一个由操作系统分配的随机端口号。

服务器端

要使用Socket编程，我们首先要编写服务器端程序。Java标准库提供了 `ServerSocket` 来实现对指定IP和指定端口的监听。 `ServerSocket` 的典型实现代码如下：

```
1.  public class Server {
2.      public static void main(String[] args) throws IOException {
3.          ServerSocket ss = new ServerSocket(6666); // 监听指定端口
4.          System.out.println("server is running...");
5.          for (;;) {
6.              Socket sock = ss.accept();
                System.out.println("connected from " +
7.  sock.getRemoteSocketAddress());
8.              Thread t = new Handler(sock);
9.              t.start();
10.         }
11.     }
12. }
13.
14. class Handler extends Thread {
15.     Socket sock;
16.
17.     public Handler(Socket sock) {
18.         this.sock = sock;
19.     }
20.
21.     @Override
22.     public void run() {
23.         try (InputStream input = this.sock.getInputStream()) {
24.             try (OutputStream output = this.sock.getOutputStream()) {
25.                 handle(input, output);
26.             }
27.         } catch (Exception e) {
28.             try {
29.                 this.sock.close();
30.             } catch (IOException ioe) {
31.             }
32.             System.out.println("client disconnected.");
33.         }
34.     }
35.
        private void handle(InputStream input, OutputStream output) throws
36. IOException {
```

```
            var writer = new BufferedWriter(new OutputStreamWriter(output,
37. StandardCharsets.UTF_8));
            var reader = new BufferedReader(new InputStreamReader(input,
38. StandardCharsets.UTF_8));
39.         writer.write("hello\n");
40.         writer.flush();
41.         for (;;) {
42.             String s = reader.readLine();
43.             if (s.equals("bye")) {
44.                 writer.write("bye\n");
45.                 writer.flush();
46.                 break;
47.             }
48.             writer.write("ok: " + s + "\n");
49.             writer.flush();
50.         }
51.     }
52. }
```

服务器端通过代码：

```
1. ServerSocket ss = new ServerSocket(6666);
```

在指定端口 `6666` 监听。这里我们没有指定IP地址，表示在计算机的所有网络接口上进行监听。

如果 `ServerSocket` 监听成功，我们就使用一个无限循环来处理客户端的连接：

```
1. for (;;) {
2.     Socket sock = ss.accept();
3.     Thread t = new Handler(sock);
4.     t.start();
5. }
```

注意到代码 `ss.accept()` 表示每当有新的客户端连接进来后，就返回一个 `Socket` 实例，这个 `Socket` 实例就是用来和刚连接的客户端进行通信的。由于客户端很多，要实现并发处理，我们就必须为每个新的 `Socket` 创建一个新线程来处理，这样，主线程的作用就是接收新的连接，每当收到新连接后，就创建一个新线程进行处理。

我们在多线程编程的章节中介绍过线程池，这里也完全可以利用线程池来处理客户端连接，能大大提高运行效率。

如果没有客户端连接进来， `accept()` 方法会阻塞并一直等待。如果有多个客户端同时连接进

来，`ServerSocket`会把连接扔到队列里，然后一个一个处理。对于Java程序而言，只需要通过循环不断调用`accept()`就可以获取新的连接。

客户端

相比服务器端，客户端程序就要简单很多。一个典型的客户端程序如下：

```
1.  public class Client {
2.      public static void main(String[] args) throws IOException {
3.          Socket sock = new Socket("localhost", 6666); // 连接指定服务器和端口
4.          try (InputStream input = sock.getInputStream()) {
5.              try (OutputStream output = sock.getOutputStream()) {
6.                  handle(input, output);
7.              }
8.          }
9.          sock.close();
10.         System.out.println("disconnected.");
11.     }
12.
        private static void handle(InputStream input, OutputStream output) throws
13. IOException {
            var writer = new BufferedWriter(new OutputStreamWriter(output,
14. StandardCharsets.UTF_8));
            var reader = new BufferedReader(new InputStreamReader(input,
15. StandardCharsets.UTF_8));
16.         Scanner scanner = new Scanner(System.in);
17.         System.out.println("[server] " + reader.readLine());
18.         for (;;) {
19.             System.out.print(">>> "); // 打印提示
20.             String s = scanner.nextLine(); // 读取一行输入
21.             writer.write(s);
22.             writer.newLine();
23.             writer.flush();
24.             String resp = reader.readLine();
25.             System.out.println("<<< " + resp);
26.             if (resp.equals("bye")) {
27.                 break;
28.             }
29.         }
30.     }
31. }
```

客户端程序通过：

```
Socket sock = new Socket("localhost", 6666);
```

连接到服务器端，注意上述代码的服务器地址是 `"localhost"` ，表示本机地址，端口号是 `6666` 。如果连接成功，将返回一个 `Socket` 实例，用于后续通信。

Socket流

当Socket连接创建成功后，无论是服务器端，还是客户端，我们都使用 `Socket` 实例进行网络通信。因为TCP是一种基于流的协议，因此，Java标准库使用 `InputStream` 和 `OutputStream` 来封装Socket的数据流，这样我们使用Socket的流，和普通IO流类似：

```
// 用于读取网络数据:
InputStream in = sock.getInputStream();
// 用于写入网络数据:
OutputStream out = sock.getOutputStream();
```

最后我们重点来看看，为什么写入网络数据时，要调用 `flush()` 方法。

如果不调用 `flush()` ，我们很可能会发现，客户端和服务器都收不到数据，这并不是Java标准库的设计问题，而是我们以流的形式写入数据的时候，并不是一写入就立刻发送到网络，而是先写入内存缓冲区，直到缓冲区满了以后，才会一次性真正发送到网络，这样设计的目的是为了提高传输效率。如果缓冲区的数据很少，而我们又想强制把这些数据发送到网络，就必须调用 `flush()` 强制把缓冲区数据发送出去。

练习

下载练习：使用Socket实现服务器和客户端通信 （推荐使用IDE练习插件快速下载）

小结

使用Java进行TCP编程时，需要使用Socket模型：

- 服务器端用 `ServerSocket` 监听指定端口；
- 客户端使用 `Socket(InetAddress, port)` 连接服务器；
- 服务器端用 `accept()` 接收连接并返回 `Socket` ；
- 双方通过 `Socket` 打开 `InputStream` / `OutputStream` 读写数据；
- 服务器端通常使用多线程同时处理多个客户端连接，利用线程池可大幅提升效率；
- `flush()` 用于强制输出缓冲区到网络。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





UDP编程

和TCP编程相比，UDP编程就简单得多，因为UDP没有创建连接，数据包也是一次收发一个，所以没有流的概念。

在Java中使用UDP编程，仍然需要使用Socket，因为应用程序在使用UDP时必须指定网络接口（IP）和端口号。注意：UDP端口和TCP端口虽然都使用0~65535，但他们是两套独立的端口，即一个应用程序用TCP占用了端口1234，不影响另一个应用程序用UDP占用端口1234。

服务器端

在服务器端，使用UDP也需要监听指定的端口。Java提供了 `DatagramSocket` 来实现这个功能，代码如下：

```
DatagramSocket ds = new DatagramSocket(6666); // 监听指定端口
for (;;) { // 无限循环
    // 数据缓冲区:
    byte[] buffer = new byte[1024];
    DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
    ds.receive(packet); // 收取一个UDP数据包
    // 收取到的数据存储在buffer中，由packet.getOffset(), packet.getLength()指定起始位置和长度
    // 将其按UTF-8编码转换为String:
    String s = new String(packet.getData(), packet.getOffset(), packet.getLength(), StandardCharsets.UTF_8);
    // 发送数据:
    byte[] data = "ACK".getBytes(StandardCharsets.UTF_8);
    packet.setData(data);
    ds.send(packet);
}
```

服务器端首先使用如下语句在指定的端口监听UDP数据包：

```
DatagramSocket ds = new DatagramSocket(6666);
```

如果没有其他应用程序占据这个端口，那么监听成功，我们就使用一个无限循环来处理收到的UDP数据包：

```
for (;;) {
    ...
}
```

要接收一个UDP数据包，需要准备一个 `byte[]` 缓冲区，并通过 `DatagramPacket` 实现接收：

```
byte[] buffer = new byte[1024];
DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
ds.receive(packet);
```

假设我们收取到的是一个 `String` ，那么，通过 `DatagramPacket` 返回的 `packet.getOffset()` 和 `packet.getLength()` 确定数据在缓冲区的起止位置：

```
String s = new String(packet.getData(), packet.getOffset(), packet.getLength(),
StandardCharsets.UTF_8);
```

当服务器收到一个DatagramPacket后，通常必须立刻回复一个或多个UDP包，因为客户端地址在DatagramPacket中，每次收到的DatagramPacket可能是不同的客户端，如果不回复，客户端就收不到任何UDP包。

发送UDP包也是通过 `DatagramPacket` 实现的，发送代码非常简单：

```
byte[] data = ...
packet.setData(data);
ds.send(packet);
```

客户端

和服务器端相比，客户端使用UDP时，只需要直接向服务器端发送UDP包，然后接收返回的UDP包：

```
DatagramSocket ds = new DatagramSocket();
ds.setSoTimeout(1000);
ds.connect(InetAddress.getByName("localhost"), 6666); // 连接指定服务器和端口
// 发送:
byte[] data = "Hello".getBytes();
DatagramPacket packet = new DatagramPacket(data, data.length);
ds.send(packet);
// 接收:
byte[] buffer = new byte[1024];
packet = new DatagramPacket(buffer, buffer.length);
ds.receive(packet);
String resp = new String(packet.getData(), packet.getOffset(),
packet.getLength());
ds.disconnect();
```

客户端打开一个 `DatagramSocket` 使用以下代码：

```
DatagramSocket ds = new DatagramSocket();
ds.setSoTimeout(1000);
ds.connect(InetAddress.getByName("localhost"), 6666);
```

客户端创建 `DatagramSocket` 实例时并不需要指定端口，而是由操作系统自动指定一个当前未使用的端口。紧接着，调用 `setSoTimeout(1000)` 设定超时1秒，意思是后续接收UDP包时，等待时间最多不会超过1秒，否则在没有收到UDP包时，客户端会无限等待下去。这一点和服务器端不一样，服务器端可以无限等待，因为它本来就被设计成长时间运行。

注意到客户端的 `DatagramSocket` 还调用了一个 `connect()` 方法“连接”到指定的服务器端。不是说UDP是无连接的协议吗？为啥这里需要 `connect()` ？

这个 `connect()` 方法不是真连接，它是为了在客户端的 `DatagramSocket` 实例中保存服务器端的IP和端口号，确保这个 `DatagramSocket` 实例只能往指定的地址和端口发送UDP包，不能往其他地址和端口发送。这么做不是UDP的限制，而是Java内置了安全检查。

如果客户端希望向两个不同的服务器发送UDP包，那么它必须创建两个 `DatagramSocket` 实例。

后续的收发数据和服务器端是一致的。通常来说，客户端必须先发UDP包，因为客户端不发UDP包，服务器端就根本不知道客户端的地址和端口号。

如果客户端认为通信结束，就可以调用 `disconnect()` 断开连接：

```
ds.disconnect();
```

注意到 `disconnect()` 也不是真正地断开连接，它只是清除了客户端 `DatagramSocket` 实例记录的远程服务器地址和端口号，这样， `DatagramSocket` 实例就可以连接另一个服务器端。

练习

下载练习：使用UDP实现服务器和客户端通信 （推荐使用IDE练习插件快速下载）

小结

使用UDP协议通信时，服务器和客户端双方无需建立连接：

- 服务器端用 `DatagramSocket(port)` 监听端口；
- 客户端使用 `DatagramSocket.connect()` 指定远程地址和端口；
- 双方通过 `receive()` 和 `send()` 读写数据；
- `DatagramSocket` 没有IO流接口，数据被直接写入 `byte[]` 缓冲区。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论



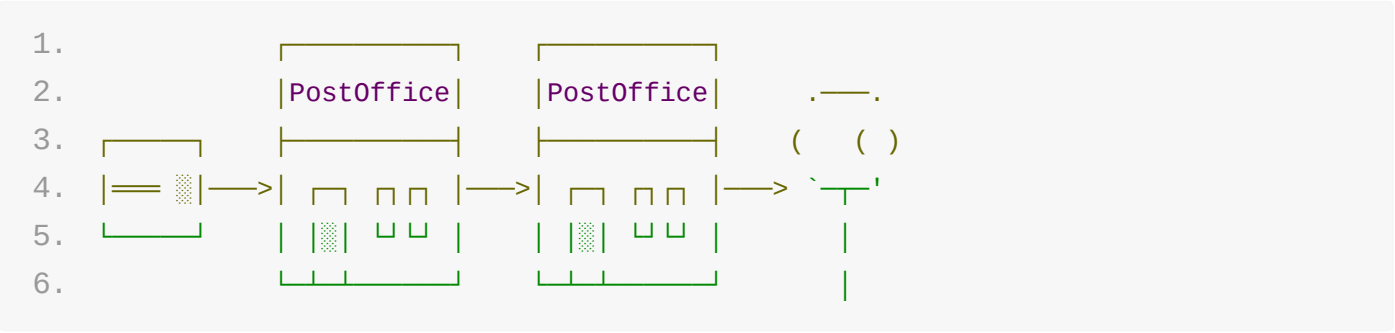


发送Email

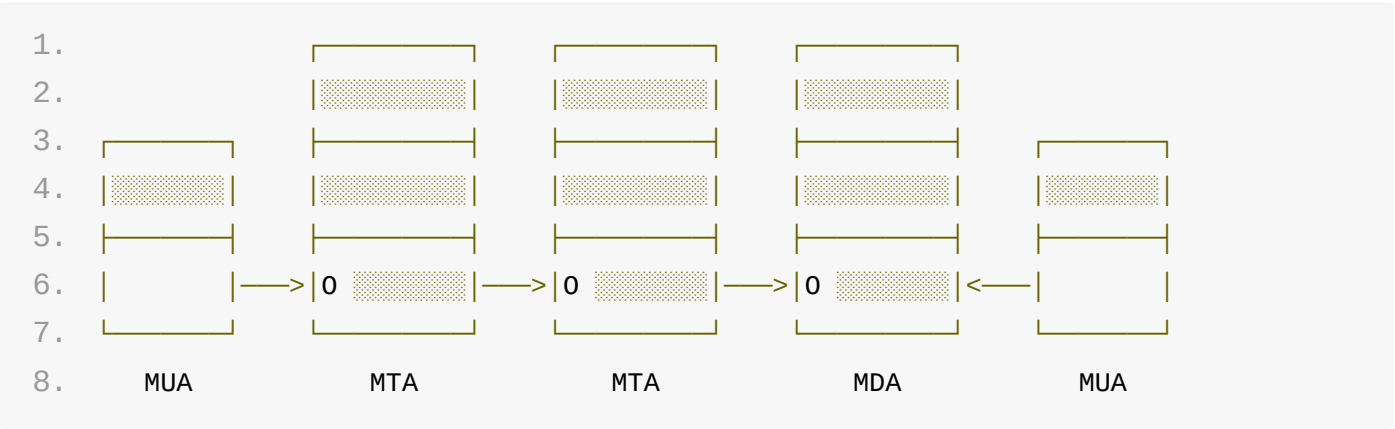
Email就是电子邮件。电子邮件的应用已经有几十年的历史了，我们熟悉的邮箱地址比如 `abc@example.com` ，邮件软件比如Outlook都是用来收发邮件的。

使用Java程序也可以收发电子邮件。我们先来看一下传统的邮件是如何发送的。

传统的邮件是通过邮局投递，然后从一个邮局到另一个邮局，最终到达用户的邮箱：



电子邮件的发送过程也是类似的，只不过是电子邮件是从用户电脑的邮件软件，例如Outlook，发送到邮件服务器上，可能经过若干个邮件服务器的中转，最终到达对方邮件服务器上，收件方就可以用软件接收邮件：



我们把类似Outlook这样的邮件软件称为MUA: Mail User Agent，意思是给用户服务的邮件代理；邮件服务器则称为MTA: Mail Transfer Agent，意思是邮件中转的代理；最终到达的邮件服务器称为MDA: Mail Delivery Agent，意思是邮件到达的代理。电子邮件一旦到达MDA，就不再动了。实际上，电子邮件通常就存储在MDA服务器的硬盘上，然后等收件人通过软件或者登陆浏览器查看邮件。

MTA和MDA这样的服务器软件通常是现成的，我们不关心这些服务器内部是如何运行的。要发送邮件，我们关心的是如何编写一个MUA的软件，把邮件发送到MTA上。

MUA到MTA发送邮件的协议就是SMTP协议，它是Simple Mail Transport Protocol的缩写，使用标准端口25，也可以使用加密端口465或587。

SMTP协议是一个建立在TCP之上的协议，任何程序发送邮件都必须遵守SMTP协议。使用Java程序发送邮件时，我们无需关心SMTP协议的底层原理，只需要使用JavaMail这个标准API就可以直接发送邮

件。

准备SMTP登录信息

假设我们准备使用自己的邮件地址 `me@example.com` 给小明发送邮件，已知小明的邮件地址是 `xiaoming@somewhere.com` ，发送邮件前，我们首先要确定作为MTA的邮件服务器地址和端口号。邮件服务器地址通常是 `smtp.example.com` ，端口号由邮件服务商确定使用25、465还是587。以下是一些常用邮件服务商的SMTP信息：

- QQ邮箱：SMTP服务器是smtp.qq.com，端口是465/587；
- 163邮箱：SMTP服务器是smtp.163.com，端口是465；
- Gmail邮箱：SMTP服务器是smtp.gmail.com，端口是465/587。

有了SMTP服务器的域名和端口号，我们还需要SMTP服务器的登录信息，通常是使用自己的邮件地址作为用户名，登录口令是用户口令或者一个独立设置的SMTP口令。

我们来看看如何使用JavaMail发送邮件。

首先，我们需要创建一个Maven工程，并把JavaMail相关的两个依赖加入进来：

```
<dependencies>
    <dependency>
        <groupId>javax.mail</groupId>
        <artifactId>javax.mail-api</artifactId>
        <version>1.6.2</version>
    </dependency>
    <dependency>
        <groupId>com.sun.mail</groupId>
        <artifactId>javax.mail</artifactId>
        <version>1.6.2</version>
    </dependency>
    ...
```

然后，我们通过JavaMail API连接到SMTP服务器上：

```
// 服务器地址:
String smtp = "smtp.office365.com";
// 登录用户名:
String username = "jxsmtp101@outlook.com";
// 登录口令:
String password = "********";
// 连接到SMTP服务器587端口:
Properties props = new Properties();
```

```
props.put("mail.smtp.host", smtp); // SMTP主机名
props.put("mail.smtp.port", "587"); // 主机端口号
props.put("mail.smtp.auth", "true"); // 是否需要用户认证
props.put("mail.smtp.starttls.enable", "true"); // 启用TLS加密
// 获取Session实例:
Session session = Session.getInstance(props, new Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication(username, password);
    }
});
// 设置debug模式便于调试:
session.setDebug(true);
```

以587端口为例，连接SMTP服务器时，需要准备一个Properties对象，填入相关信息。最后获取Session实例时，如果服务器需要认证，还需要传入一个Authenticator对象，并返回指定的用户名和口令。

当我们获取到Session实例后，打开调试模式可以看到SMTP通信的详细内容，便于调试。

发送邮件

发送邮件时，我们需要构造一个 `Message` 对象，然后调用 `Transport.send(Message)` 即可完成发送：

```
MimeMessage message = new MimeMessage(session);
// 设置发送方地址:
message.setFrom(new InternetAddress("me@example.com"));
// 设置接收方地址:
message.setRecipient(Message.RecipientType.TO, new
InternetAddress("xiaoming@somewhere.com"));
// 设置邮件主题:
message.setSubject("Hello", "UTF-8");
// 设置邮件正文:
message.setText("Hi Xiaoming...", "UTF-8");
// 发送:
Transport.send(message);
```

绝大多数邮件服务器要求发送方地址和登录用户名必须一致，否则发送将失败。

填入真实的地址，运行上述代码，我们可以在控制台看到JavaMail打印的调试信息：

```
这是JavaMail打印的调试信息:
```

```
DEBUG: setDebug: JavaMail version 1.6.2
DEBUG: getProvider() returning javax.mail.Provider[TRANSPORT,smtp,com.sun.mail.smtp.SMTPTransport,Oracle]
DEBUG SMTP: need username and password for authentication
DEBUG SMTP: protocolConnect returning false, host=smtp.office365.com, ...
DEBUG SMTP: useEhlo true, useAuth true
开始尝试连接smtp.office365.com:
DEBUG SMTP: trying to connect to host "smtp.office365.com", port 587, ...
DEBUG SMTP: connected to host "smtp.office365.com", port: 587
发送命令EHLO:
EHLO localhost
SMTP服务器响应250:
250-SG3P274CA0024.outlook.office365.com Hello
250-SIZE 157286400
...
DEBUG SMTP: Found extension "SIZE", arg "157286400"
发送命令STARTTLS:
STARTTLS
SMTP服务器响应220:
220 2.0.0 SMTP server ready
EHLO localhost
250-SG3P274CA0024.outlook.office365.com Hello [111.196.164.63]
250-SIZE 157286400
250-PIPELINING
250-...
DEBUG SMTP: Found extension "SIZE", arg "157286400"
...
尝试登录:
DEBUG SMTP: protocolConnect login, host=smtp.office365.com, user=********, password=********
DEBUG SMTP: Attempt to authenticate using mechanisms: LOGIN PLAIN DIGEST-MD5 NTLM XOAUTH2
DEBUG SMTP: Using mechanism LOGIN
DEBUG SMTP: AUTH LOGIN command trace suppressed
登录成功:
DEBUG SMTP: AUTH LOGIN succeeded
DEBUG SMTP: use8bit false
开发发送邮件，设置FROM:
MAIL FROM:<********@outlook.com>
250 2.1.0 Sender OK
设置TO:
RCPT TO:<********@sina.com>
250 2.1.5 Recipient OK
```

```
发送邮件数据:
DATA
服务器响应354:
354 Start mail input; end with <CRLF>.<CRLF>
真正的邮件数据:
Date: Mon, 2 Dec 2019 09:37:52 +0800 (CST)
From: ********@outlook.com
To: ********001@sina.com
Message-ID: <1617791695.0.1575250672483@localhost>
邮件主题是编码后的文本:
Subject: =?UTF-8?Q?JavaMail=E9=82=AE=E4=BB=B6?=
MIME-Version: 1.0
Content-Type: text/plain; charset=UTF-8
Content-Transfer-Encoding: base64

邮件正文是Base64编码的文本:
SGVsbG8sIOi/meaYr+S4gOWwgeadpeiHqmphdmFtYWls55qE6YKu5Lu277yB
.
邮件数据发送完成后，以\r\n.\r\n结束，服务器响应250表示发送成功:
250 2.0.0 OK <HK0PR03MB4961.apcprd03.prod.outlook.com>
[Hostname=HK0PR03MB4961.apcprd03.prod.outlook.com]
DEBUG SMTP: message successfully delivered to mail server
发送QUIT命令:
QUIT
服务器响应221结束TCP连接:
221 2.0.0 Service closing transmission channel
```

从上面的调试信息可以看出，SMTP协议是一个请求-响应协议，客户端总是发送命令，然后等待服务器响应。服务器响应总是以数字开头，后面的信息才是用于调试的文本。这些响应码已经被定义在SMTP协议中了，查看具体的响应码就可以知道出错原因。

如果一切顺利，对方将收到一封文本格式的电子邮件：



发送HTML邮件

发送HTML邮件和文本邮件是类似的，只需要把：

```
message.setText(body, "UTF-8");
```

改为：

```
message.setText(body, "UTF-8", "html");
```

传入的 `body` 是类似 `<h1>Hello</h1><p>Hi, xxx</p>` 这样的HTML字符串即可。

HTML邮件可以在邮件客户端直接显示为网页格式：



发送附件

要在电子邮件中携带附件，我们就不能直接调用 `message.setText()` 方法，而是要构造一个 `Multipart` 对象：

```
Multipart multipart = new MimeMultipart();
// 添加text:
BodyPart textpart = new MimeBodyPart();
textpart.setContent(body, "text/html;charset=utf-8");
multipart.addBodyPart(textpart);
// 添加image:
BodyPart imagepart = new MimeBodyPart();
imagepart.setFileName(fileName);
imagepart.setDataHandler(new DataHandler(new ByteArrayDataSource(input,
"application/octet-stream")));
multipart.addBodyPart(imagepart);
// 设置邮件内容为multipart:
message.setContent(multipart);
```

一个 `Multipart` 对象可以添加若干个 `BodyPart` ，其中第一个 `BodyPart` 是文本，即邮件正文，后面的BodyPart是附件。 `BodyPart` 依靠 `setContent()` 决定添加的内容，如果添加文本，用 `setContent("…", "text/plain;charset=utf-8")` 添加纯文本，或者用 `setContent("…", "text/html;charset=utf-8")` 添加HTML文本。如果添加附件，需要设置文件名（不一定和真实文件名一致），并且添加一个 `DataHandler()` ，传入文件的MIME类型。二进制文件可以

用 `application/octet-stream` ，Word文档则是 `application/msword` 。

最后，通过 `setContent()` 把 `Multipart` 添加到 `Message` 中，即可发送。

带附件的邮件在客户端会被提示下载：



发送内嵌图片的HTML邮件

有些童鞋可能注意到，HTML邮件中可以内嵌图片，这是怎么做到的？

如果给一个 `<img src="http://example.com/test.jpg">` ，这样的外部图片链接通常会被邮件客户端过滤，并提示用户显示图片并不安全。只有内嵌的图片才能正常在邮件中显示。

内嵌图片实际上也是一个附件，即邮件本身也是 `Multipart` ，但需要做一点额外的处理：

```
Multipart multipart = new MimeMultipart();
// 添加text:
BodyPart textpart = new MimeBodyPart();
textpart.setContent("<h1>Hello</h1><p><img src=\"cid:img01\"></p>",
"text/html;charset=utf-8");
multipart.addBodyPart(textpart);
// 添加image:
BodyPart imagepart = new MimeBodyPart();
imagepart.setFileName(fileName);
imagepart.setDataHandler(new DataHandler(new ByteArrayDataSource(input,
"image/jpeg")));
// 与HTML的<img src="cid:img01">关联:
imagepart.setHeader("Content-ID", "<img01>");
multipart.addBodyPart(imagepart);
```

在HTML邮件中引用图片时，需要设定一个ID，用类似 `<img src=\"cid:img01\">` 引用，然后，在添加图片作为BodyPart时，除了要正确设置MIME类型（根据图片类型使

用 `image/jpeg` 或 `image/png` ），还需要设置一个Header：

```
1.  imagepart.setHeader("Content-ID", "<img01>");
```

这个ID和HTML中引用的ID对应起来，邮件客户端就可以正常显示内嵌图片：



常见问题

如果用户名或口令错误，会导致 `535` 登录失败：

```
1.  DEBUG SMTP: AUTH LOGIN failed
    Exception in thread "main" javax.mail.AuthenticationFailedException: 535 5.7.3
2.  Authentication unsuccessful [HK0PR03CA0105.apcprd03.prod.outlook.com]
```

如果登录用户和发件人不一致，会导致 `554` 拒绝发送错误：

```
1.  DEBUG SMTP: MessagingException while sending, THROW:
```

```
    com.sun.mail.smtp.SMTPSendFailedException: 554 5.2.0
2.  STOREDRV.Submission.Exception:SendAsDeniedException.MapiExceptionSendAsDenied;
```

有些时候，如果邮件主题和正文过于简单，会导致 `554` 被识别为垃圾邮件的错误：

```
1.  DEBUG SMTP: MessagingException while sending, THROW:
2.  com.sun.mail.smtp.SMTPSendFailedException: 554 DT:SPM
```

练习

下载练习：使用SMTP发送邮件 （推荐使用IDE练习插件快速下载）

小结

使用JavaMail API发送邮件本质上是一个MUA软件通过SMTP协议发送邮件至MTA服务器；

打开调试模式可以看到详细的SMTP交互信息；

某些邮件服务商需要开启SMTP，并需要独立的SMTP登录密码。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





接收Email

发送Email的过程我们在上一节已经讲过了，客户端总是通过SMTP协议把邮件发送给MTA。

接收Email则相反，因为邮件最终到达收件人的MDA服务器，所以，接收邮件是收件人用自己的客户端把邮件从MDA服务器上抓取到本地的过程。

接收邮件使用最广泛的协议是POP3：Post Office Protocol version 3，它也是一个建立在TCP连接之上的协议。POP3服务器的标准端口是110，如果整个会话需要加密，那么使用加密端口995。

另一种接收邮件的协议是IMAP：Internet Mail Access Protocol，它使用标准端口143和加密端口993。IMAP和POP3的主要区别是，IMAP协议在本地的所有操作都会自动同步到服务器上，并且，IMAP可以允许用户在邮件服务器的收件箱中创建文件夹。

JavaMail也提供了IMAP协议的支持。因为POP3和IMAP的使用方式非常类似，因此我们只介绍POP3的用法。

使用POP3收取Email时，我们无需关心POP3协议底层，因为JavaMail提供了高层接口。首先需要连接到Store对象：

```
// 准备登录信息:
String host = "pop3.example.com";
int port = 995;
String username = "bob@example.com";
String password = "password";

Properties props = new Properties();
props.setProperty("mail.store.protocol", "pop3"); // 协议名称
props.setProperty("mail.pop3.host", host);// POP3主机名
props.setProperty("mail.pop3.port", String.valueOf(port)); // 端口号
// 启动SSL:
props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
props.put("mail.smtp.socketFactory.port", String.valueOf(port));

// 连接到Store:
URLName url = new URLName("pop3", host, post, "", username, password);
Session session = Session.getInstance(props, null);
session.setDebug(true); // 显示调试信息
Store store = new POP3SSLStore(session, url);
store.connect();
```

一个 `Store` 对象表示整个邮箱的存储，要收取邮件，我们需要通过 `Store` 访问指定

的 `Folder` （文件夹），通常是 `INBOX` 表示收件箱：

```
// 获取收件箱:
Folder folder = store.getFolder("INBOX");
// 以读写方式打开:
folder.open(Folder.READ_WRITE);
// 打印邮件总数/新邮件数量/未读数量/已删除数量:
System.out.println("Total messages: " + folder.getMessageCount());
System.out.println("New messages: " + folder.getNewMessageCount());
System.out.println("Unread messages: " + folder.getUnreadMessageCount());
System.out.println("Deleted messages: " + folder.getDeletedMessageCount());
// 获取每一封邮件:
Message[] messages = folder.getMessages();
for (Message message : messages) {
    // 打印每一封邮件:
    printMessage((MimeMessage) message);
}
```

当我们获取到一个 `Message` 对象时，可以强制转型为MimeMessage，然后打印出邮件主题、发件人、收件人等信息：

```
void printMessage(MimeMessage msg) throws IOException, MessagingException {
    // 邮件主题:
    System.out.println("Subject: " + MimeUtility.decodeText(msg.getSubject()));
    // 发件人:
    Address[] froms = msg.getFrom();
    InternetAddress address = (InternetAddress) froms[0];
    String personal = address.getPersonal();
    String from = personal == null ? address.getAddress() :
(MimeUtility.decodeText(personal) + " <" + address.getAddress() + ">");
    System.out.println("From: " + from);
    // 继续打印收件人:
    ...
}
```

比较麻烦的是获取邮件的正文。一个 `MimeMessage` 对象也是一个 `Part` 对象，它可能只包含一个文本，也可能是一个 `Multipart` 对象，即由几个 `Part` 构成，因此，需要递归地解析出完整的正文：

```
String getBody(Part part) throws MessagingException, IOException {
    if (part.isMimeType("text/*")) {
        // Part是文本:
        return part.getContent().toString();
```

```
    }
    if (part.isMimeType("multipart/*")) {
        // Part是一个Multipart对象:
        Multipart multipart = (Multipart) part.getContent();
        // 循环解析每个子Part:
        for (int i = 0; i < multipart.getCount(); i++) {
            BodyPart bodyPart = multipart.getBodyPart(i);
            String body = getBody(bodyPart);
            if (!body.isEmpty()) {
                return body;
            }
        }
    }
    return "";
}
```

最后记得关闭 `Folder` 和 `Store` ：

```
folder.close(true); // 传入true表示删除操作会同步到服务器上（即删除服务器收件箱的邮件）
store.close();
```

练习

下载练习：使用POP3接收邮件 （推荐使用IDE练习插件快速下载）

小结

使用Java接收Email时，可以用POP3协议或IMAP协议。

使用POP3协议时，需要用Maven引入JavaMail依赖，并确定POP3服务器的域名 / 端口 / 是否使用SSL等，然后，调用相关API接收Email。

设置debug模式可以查看通信详细内容，便于排查错误。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





HTTP编程

什么是HTTP？HTTP就是目前使用最广泛的Web应用程序使用的基础协议，例如，浏览器访问网站，手机App访问后台服务器，都是通过HTTP协议实现的。

HTTP是HyperText Transfer Protocol的缩写，翻译为超文本传输协议，它是基于TCP协议之上的一种请求-响应协议。

我们来看一下浏览器请求访问某个网站时发送的HTTP请求-响应。当浏览器希望访问某个网站时，浏览器和网站服务器之间首先建立TCP连接，且服务器总是使用 `80` 端口和加密端口 `443` ，然后，浏览器向服务器发送一个HTTP请求，服务器收到后，返回一个HTTP响应，并且在响应中包含了HTML的网页内容，这样，浏览器解析HTML后就可以给用户显示网页了。一个完整的HTTP请求-响应如下：

```
                GET / HTTP/1.1
                Host: www.sina.com.cn
                User-Agent: Mozilla/5 MSIE
                Accept: */*                          ┌──────────┐
┌──────────┐    Accept-Language: zh-CN,en             │▒▒▒▒▒▒▒▒▒▒│
│O ▒▒▒▒▒▒▒▒▒│──────────────────────────────>├──────────┤
├──────────┤<──────────────────────────────│▒▒▒▒▒▒▒▒▒▒│
│          │ HTTP/1.1 200 OK              ├──────────┤
│          │ Content-Type: text/html      │▒▒▒▒▒▒▒▒▒▒│
└──────────┘ Content-Length: 133251       └──────────┘
  Browser    <!DOCTYPE html>                 Server
             <html><body>
             <h1>Hello</h1>
             ...
```

HTTP请求的格式是固定的，它由HTTP Header和HTTP Body两部分构成。第一行总是 `请求方法 路径 HTTP版本` ，例如， `GET / HTTP/1.1` 表示使用 `GET` 请求，路径是 `/` ，版本是 `HTTP/1.1` 。

后续的每一行都是固定的 `Header: Value` 格式，我们称为HTTP Header，服务器依靠某些特定的Header来识别客户端请求，例如：

- Host：表示请求的域名，因为一台服务器上可能有多个网站，因此有必要依靠Host来识别用于请求；
- User-Agent：表示客户端自身标识信息，不同的浏览器有不同的标识，服务器依靠User-Agent判断客户端类型；
- Accept：表示客户端能处理的HTTP响应格式， `/` 表示任意格式， `text/*` 表示任意文本， `image/png` 表示PNG格式的图片；
- Accept-Language：表示客户端接收的语言，多种语言按优先级排序，服务器依靠该字段给用

户返回特定语言的网页版本。

如果是 `GET` 请求，那么该HTTP请求只有HTTP Header，没有HTTP Body。如果是 `POST` 请求，那么该HTTP请求带有Body，以一个空行分隔。一个典型的带Body的HTTP请求如下：

```
POST /login HTTP/1.1
Host: www.example.com
Content-Type: application/x-www-form-urlencoded
Content-Length: 30

username=hello&password=123456
```

`POST` 请求通常要设置 `Content-Type` 表示Body的类型， `Content-Length` 表示Body的长度，这样服务器就可以根据请求的Header和Body做出正确的响应。

此外， `GET` 请求的参数必须附加在URL上，并以URLEncode方式编码，例如： `http://www.example.com/?a=1&b=K%26R` ，参数分别是 `a=1` 和 `b=K&R` 。因为URL的长度限制， `GET` 请求的参数不能太多，而 `POST` 请求的参数就没有长度限制，因为 `POST` 请求的参数必须放到Body中。并且， `POST` 请求的参数不一定是URL编码，可以按任意格式编码，只需要在 `Content-Type` 中正确设置即可。常见的发送JSON的 `POST` 请求如下：

```
POST /login HTTP/1.1
Content-Type: application/json
Content-Length: 38

{"username":"bob","password":"123456"}
```

HTTP响应也是由Header和Body两部分组成，一个典型的HTTP响应如下：

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 133251

<!DOCTYPE html>
<html><body>
<h1>Hello</h1>
...
```

响应的第一行总是 `HTTP版本 响应代码 响应说明` ，例如， `HTTP/1.1 200 OK` 表示版本是 `HTTP/1.1` ，响应代码是 `200` ，响应说明是 `OK` 。客户端只依赖响应代码判断HTTP响应是否成功。HTTP有固定的响应代码：

- 1xx：表示一个提示性响应，例如101表示将切换协议，常见于WebSocket连接；
- 2xx：表示一个成功的响应，例如200表示成功，206表示只发送了部分内容；
- 3xx：表示一个重定向的响应，例如301表示永久重定向，303表示客户端应该按指定路径重新发送请求；
- 4xx：表示一个因为客户端问题导致的错误响应，例如400表示因为Content-Type等各种原因导致的无效请求，404表示指定的路径不存在；
- 5xx：表示一个因为服务器问题导致的错误响应，例如500表示服务器内部故障，503表示服务器暂时无法响应。

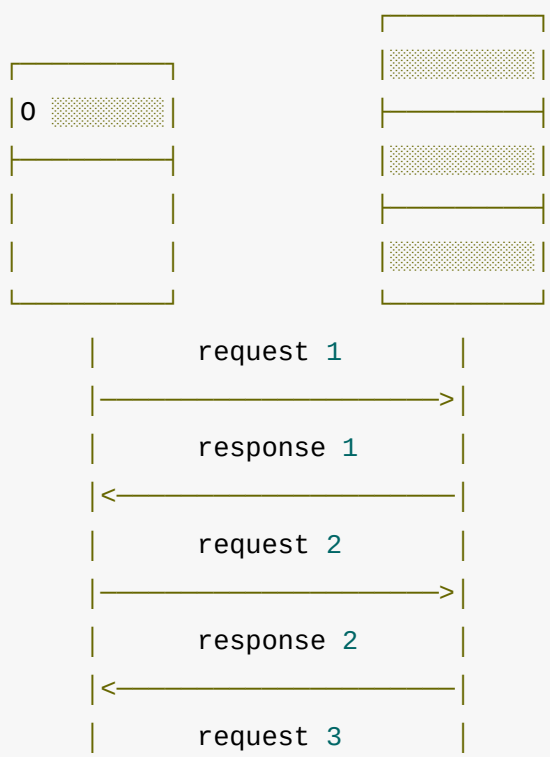
当浏览器收到第一个HTTP响应后，它解析HTML后，又会发送一系列HTTP请求，例如，`GET /logo.jpg HTTP/1.1` 请求一个图片，服务器响应图片请求后，会直接把二进制内容的图片发送给浏览器：

```
HTTP/1.1 200 OK
Content-Type: image/jpeg
Content-Length: 18391

????JFIFHH??XExifMM?i&??X?...(二进制的JPEG图片)
```

因此，服务器总是被动地接收客户端的一个HTTP请求，然后响应它。客户端则根据需要发送若干个HTTP请求。

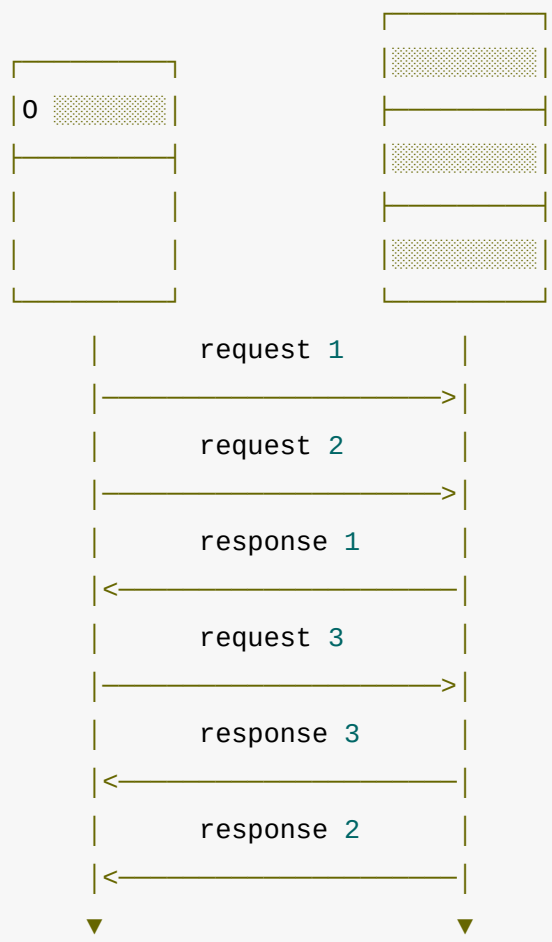
对于最早期的HTTP/1.0协议，每次发送一个HTTP请求，客户端都需要先创建一个新的TCP连接，然后，收到服务器响应后，关闭这个TCP连接。由于建立TCP连接就比较耗时，因此，为了提高效率，HTTP/1.1协议允许在一个TCP连接中反复发送-响应，这样就能大大提高效率：



```
      |──────────────────────>|
      |      response 3       |
      |<──────────────────────|
      ▼                       ▼
```

因为HTTP协议是一个请求-响应协议，客户端在发送了一个HTTP请求后，必须等待服务器响应后，才能发送下一个请求，这样一来，如果某个响应太慢，它就会堵住后面的请求。

所以，为了进一步提速，HTTP/2.0允许客户端在没有收到响应的时候，发送多个HTTP请求，服务器返回响应的时候，不一定按顺序返回，只要双方能识别出哪个响应对应哪个请求，就可以做到并行发送和接收：



可见，HTTP/2.0进一步提高了效率。

HTTP编程

既然HTTP涉及到客户端和服务器端，和TCP类似，我们也需要针对客户端编程和针对服务器端编程。

本节我们不讨论服务器端的HTTP编程，因为服务器端的HTTP编程本质上就是编写Web服务器，这是一个非常复杂的体系，也是JavaEE开发的核心内容，我们在后面的章节再仔细研究。

本节我们只讨论作为客户端的HTTP编程。

因为浏览器也是一种HTTP客户端，所以，客户端的HTTP编程，它的行为本质上和浏览器是一样的，即发送一个HTTP请求，接收服务器响应后，获得响应内容。只不过浏览器进一步把响应内容解析后渲染并展示给了用户，而我们使用Java进行HTTP客户端编程仅限于获得响应内容。

我们来看一下Java如果使用HTTP客户端编程。

Java标准库提供了基于HTTP的包，但是要注意，早期的JDK版本是通过 `HttpURLConnection` 访问HTTP，典型代码如下：

```
URL url = new URL("http://www.example.com/path/to/target?a=1&b=2");
HttpURLConnection conn = (HttpURLConnection) url.openConnection();
conn.setRequestMethod("GET");
conn.setUseCaches(false);
conn.setConnectTimeout(5000); // 请求超时5秒
// 设置HTTP头:
conn.setRequestProperty("Accept", "*/*");
conn.setRequestProperty("User-Agent", "Mozilla/5.0 (compatible; MSIE 11;
Windows NT 5.1)");
// 连接并发送HTTP请求:
conn.connect();
// 判断HTTP响应是否200:
if (conn.getResponseCode() != 200) {
    throw new RuntimeException("bad response");
}
// 获取所有响应Header:
Map<String, List<String>> map = conn.getHeaderFields();
for (String key : map.keySet()) {
    System.out.println(key + ": " + map.get(key));
}
// 获取响应内容:
InputStream input = conn.getInputStream();
...
```

上述代码编写比较繁琐，并且需要手动处理 `InputStream` ，所以用起来很麻烦。

从Java 11开始，引入了新的 `HttpClient` ，它使用链式调用的API，能大大简化HTTP的处理。

我们来看一下如何使用新版的 `HttpClient` 。首先需要创建一个全局 `HttpClient` 实例，因为 `HttpClient` 内部使用线程池优化多个HTTP连接，可以复用：

```
static HttpClient httpClient = HttpClient.newBuilder().build();
```

使用 `GET` 请求获取文本内容代码如下：

```
import java.net.URI;
import java.net.http.*;
import java.net.http.HttpClient.Version;
import java.time.Duration;
import java.util.*;
```

```
public class Main {
    // 全局HttpClient:
    static HttpClient httpClient = HttpClient.newBuilder().build();

    public static void main(String[] args) throws Exception {
        String url = "https://www.sina.com.cn/";
        HttpRequest request = HttpRequest.newBuilder(new URI(url))
            // 设置Header:
            .header("User-Agent", "Java HttpClient").header("Accept", "*/*")
            // 设置超时:
            .timeout(Duration.ofSeconds(5))
            // 设置版本:
            .version(Version.HTTP_2).build();
        HttpResponse<String> response = httpClient.send(request,
HttpResponse.BodyHandlers.ofString());
        // HTTP允许重复的Header，因此一个Header可对应多个Value:
        Map<String, List<String>> headers = response.headers().map();
        for (String header : headers.keySet()) {
            System.out.println(header + ": " + headers.get(header).get(0));
        }
        System.out.println(response.body().substring(0, 1024) + "...");
    }
}
```

▶ Run

如果我们要获取图片这样的二进制内容，只需要把 `HttpResponse.BodyHandlers.ofString()` 换成 `HttpResponse.BodyHandlers.ofByteArray()` ，就可以获得一个 `HttpResponse<byte[]>` 对象。如果响应的内容很大，不希望一次性全部加载到内存，可以使用 `HttpResponse.BodyHandlers.ofInputStream()` 获取一个 `InputStream` 流。

要使用 `POST` 请求，我们要准备好发送的Body数据并正确设置 `Content-Type` ：

```
String url = "http://www.example.com/login";
String body = "username=bob&password=123456";
HttpRequest request = HttpRequest.newBuilder(new URI(url))
    // 设置Header:
    .header("Accept", "*/*")
    .header("Content-Type", "application/x-www-form-urlencoded")
    // 设置超时:
    .timeout(Duration.ofSeconds(5))
    // 设置版本:
    .version(Version.HTTP_2)
    // 使用POST并设置Body:
    .POST(BodyPublishers.ofString(body, StandardCharsets.UTF_8)).build();
```

```
    HttpResponse<String> response = httpClient.send(request,
13. HttpResponse.BodyHandlers.ofString());
14. String s = response.body();
```

可见发送 `POST` 数据也十分简单。

练习

下载练习：使用HttpClient （推荐使用IDE练习插件快速下载）

小结

Java提供了 `HttpClient` 作为新的HTTP客户端编程接口用于取代老的 `HttpURLConnection` 接口；

`HttpClient` 使用链式调用并通过内置的 `BodyPublishers` 和 `BodyHandlers` 来更方便地处理数据。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





XML与JSON

XML和JSON是两种经常在网络使用的数据表示格式，本章我们介绍如何使用Java读写XML和JSON。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





XML简介

XML是可扩展标记语言（eXtensible Markup Language）的缩写，它是是一种数据表示格式，可以描述非常复杂的数据结构，常用于传输和存储数据。

例如，一个描述书籍的XML文档可能如下：

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE note SYSTEM "book.dtd">
<book id="1">
    <name>Java核心技术</name>
    <author>Cay S. Horstmann</author>
    <isbn lang="CN">1234567</isbn>
    <tags>
        <tag>Java</tag>
        <tag>Network</tag>
    </tags>
    <pubDate/>
</book>
```

XML有几个特点：一是纯文本，默认使用UTF-8编码，二是可嵌套，适合表示结构化数据。如果把XML内容存为文件，那么它就是一个XML文件，例如 `book.xml` 。此外，XML内容经常通过网络作为消息传输。

XML的结构

XML有固定的结构，首行必定是 `<?xml version="1.0"?>` ，可以加上可选的编码。紧接着，如果以类似 `<!DOCTYPE note SYSTEM "book.dtd">` 声明的是文档定义类型（DTD：Document Type Definition），DTD是可选的。接下来是XML的文档内容，一个XML文档有且仅有一个根元素，根元素可以包含任意个子元素，元素可以包含属性，例如， `<isbn lang="CN">1234567</isbn>` 包含一个属性 `lang="CN"` ，且元素必须正确嵌套。如果是空元素，可以用 `<tag/>` 表示。

由于使用了 `<` 、 `>` 以及引号等标识符，如果内容出现了特殊符号，需要使用 `&???;` 表示转义。例如， `Java<tm>` 必须写成：

```
<name>Java&lt;tm&gt;</name>
```

常见的特殊字符如下：

字符	表示
<	<

>	>
&	&
"	"
'	'

格式正确的XML（Well Formed）是指XML的格式是正确的，可以被解析器正常读取。而合法的XML是指，不但XML格式正确，而且它的数据结构可以被DTD或者XSD验证。

DTD文档可以指定一系列规则，例如：

- 根元素必须是`book`
- `book`元素必须包含`name`，`author`等指定元素
- `isbn`元素必须包含属性`lang`
- …

如何验证XML文件的正确性呢？最简单的方式是通过浏览器验证。可以直接把XML文件拖拽到浏览器窗口，如果格式错误，浏览器会报错。

和结构类似的HTML不同，浏览器对HTML有一定的“容错性”，缺少关闭标签也可以被解析，但XML要求严格的格式，任何没有正确嵌套的标签都会导致错误。

XML是一个技术体系，除了我们经常用到的XML文档本身外，XML还支持：

- DTD和XSD：验证XML结构和数据是否有效；
- Namespace：XML节点和属性的名字空间；
- XSLT：把XML转化为另一种文本；
- XPath：一种XML节点查询语言；
- …

实际上，XML的这些相关技术实现起来非常复杂，在实际应用中很少用到，通常了解一下就可以了。

小结

XML使用嵌套结构的数据表示方式，支持格式验证；

XML常用于配置文件、网络消息传输等。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用DOM

因为XML是一种树形结构的文档，它有两种标准的解析API：

- DOM：一次性读取XML，并在内存中表示为树形结构；
- SAX：以流的形式读取XML，使用事件回调。

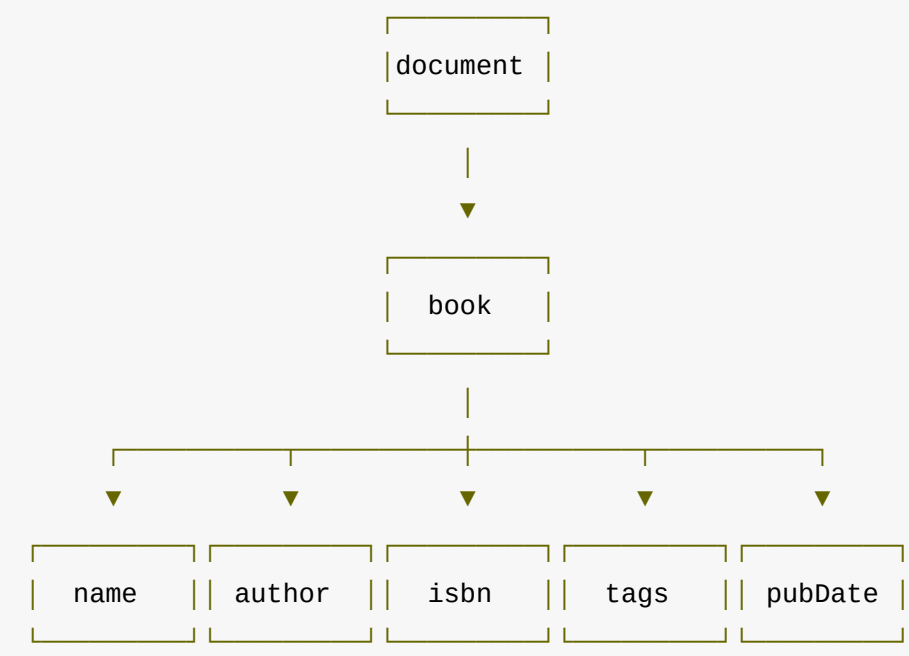
我们先来看如何使用DOM来读取XML。

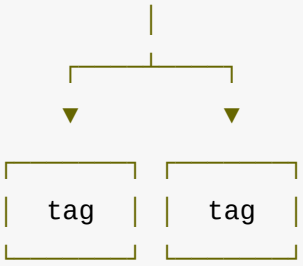
DOM是Document Object Model的缩写，DOM模型就是把XML结构作为一个树形结构处理，从根节点开始，每个节点都可以包含任意个子节点。

我们以下面的XML为例：

```
<?xml version="1.0" encoding="UTF-8" ?>
<book id="1">
    <name>Java核心技术</name>
    <author>Cay S. Horstmann</author>
    <isbn lang="CN">1234567</isbn>
    <tags>
        <tag>Java</tag>
        <tag>Network</tag>
    </tags>
    <pubDate/>
</book>
```

如果解析为DOM结构，它大概长这样：





注意到最顶层的document代表XML文档，它是真正的“根”，而 `<book>` 虽然是根元素，但它是 `document` 的一个子节点。

Java提供了DOM API来解析XML，它使用下面的对象来表示XML的内容：

- Document：代表整个XML文档；
- Element：代表一个XML元素；
- Attribute：代表一个元素的某个属性。

使用DOM API解析一个XML文档的代码如下：

```
InputStream input = Main.class.getResourceAsStream("/book.xml");
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document doc = db.parse(input);
```

`DocumentBuilder.parse()` 用于解析一个XML，它可以接收InputStream，File或者URL，如果解析无误，我们将获得一个Document对象，这个对象代表了整个XML文档的树形结构，需要遍历以便读取指定元素的值：

```
void printNode(Node n, int indent) {
    for (int i = 0; i < indent; i++) {
        System.out.print(' ');
    }
    switch (n.getNodeType()) {
    case Node.DOCUMENT_NODE: // Document节点
        System.out.println("Document: " + n.getNodeName());
        break;
    case Node.ELEMENT_NODE: // 元素节点
        System.out.println("Element: " + n.getNodeName());
        break;
    case Node.TEXT_NODE: // 文本
        System.out.println("Text: " + n.getNodeName() + " = " +
n.getNodeValue());
        break;
```

```
15.      case Node.ATTRIBUTE_NODE: // 属性
             System.out.println("Attr: " + n.getNodeName() + " = " +
16.  n.getNodeValue());
17.          break;
18.      default: // 其他
             System.out.println("NodeType: " + n.getNodeType() + ", NodeName: " +
19.  n.getNodeName());
20.      }
         for (Node child = n.getFirstChild(); child != null; child =
21.  child.getNextSibling()) {
22.          printNode(child, indent + 1);
23.      }
24.  }
```

解析结构如下：

```
1.  Document: #document
2.   Element: book
3.    Text: #text =
4.
5.    Element: name
6.     Text: #text = Java核心技术
7.    Text: #text =
8.
9.    Element: author
10.    Text: #text = Cay S. Horstmann
11.   Text: #text =
12.   ...
```

对于DOM API解析出来的结构，我们从根节点Document出发，可以遍历所有子节点，获取所有元素、属性、文本数据，还可以包括注释，这些节点被统称为Node，每个Node都有自己的Type，根据Type来区分一个Node到底是元素，还是属性，还是文本，等等。

使用DOM API时，如果要读取某个元素的文本，需要访问它的Text类型的子节点，所以使用起来还是比较繁琐的。

练习

下载练习：使用DOM解析XML （推荐使用IDE练习插件快速下载）

小结

Java提供的DOM API可以将XML解析为DOM结构，以Document对象表示；

DOM可在内存中完整表示XML数据结构；

DOM解析速度慢，内存占用大。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用SAX

使用DOM解析XML的优点是用起来省事，但它的主要缺点是内存占用太大。

另一种解析XML的方式是SAX。SAX是Simple API for XML的缩写，它是一种基于流的解析方式，边读取XML边解析，并以事件回调的方式让调用者获取数据。因为是一边读一边解析，所以无论XML有多大，占用的内存都很小。

SAX解析会触发一系列事件：

- startDocument：开始读取XML文档；
- startElement：读取到了一个元素，例如 `<book>` ；
- characters：读取到了字符；
- endElement：读取到了一个结束的元素，例如 `</book>` ；
- endDocument：读取XML文档结束。

如果我们用SAX API解析XML，Java代码如下：

```
InputStream input = Main.class.getResourceAsStream("/book.xml");
SAXParserFactory spf = SAXParserFactory.newInstance();
SAXParser saxParser = spf.newSAXParser();
saxParser.parse(input, new MyHandler());
```

关键代码 `SAXParser.parse()` 除了需要传入一个 `InputStream` 外，还需要传入一个回调对象，这个对象要继承自 `DefaultHandler` ：

```
class MyHandler extends DefaultHandler {
    public void startDocument() throws SAXException {
        print("start document");
    }

    public void endDocument() throws SAXException {
        print("end document");
    }

    public void startElement(String uri, String localName, String qName,
Attributes attributes) throws SAXException {
        print("start element:", localName, qName);
    }

    public void endElement(String uri, String localName, String qName) throws
SAXException {
```

```
        print("end element:", localName, qName);
    }

    public void characters(char[] ch, int start, int length) throws
SAXException {
        print("characters:", new String(ch, start, length));
    }

    public void error(SAXParseException e) throws SAXException {
        print("error:", e);
    }

    void print(Object... objs) {
        for (Object obj : objs) {
            System.out.print(obj);
            System.out.print(" ");
        }
        System.out.println();
    }
}
```

运行SAX解析代码，可以打印出下面的结果：

```
start document
start element:  book
characters:

start element:  name
characters: Java核心技术
end element:  name
characters:

start element:  author
...
```

如果要读取 `<name>` 节点的文本，我们就必须在解析过程中根据 `startElement()` 和 `endElement()` 定位当前正在读取的节点，可以使用栈结构保存，每遇到一个 `startElement()` 入栈，每遇到一个 `endElement()` 出栈，这样，读到 `characters()` 时我们才知道当前读取的文本是哪个节点的。可见，使用SAX API仍然比较麻烦。

练习

下载练习：使用SAX解析XML （推荐使用IDE练习插件快速下载）

小结

SAX是一种流式解析XML的API；

SAX通过事件触发，读取速度快，消耗内存少；

调用方必须通过回调方法获得解析过程中的数据。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用Jackson

前面我们介绍了DOM和SAX两种解析XML的标准接口。但是，无论是DOM还是SAX，使用起来都不直观。

观察XML文档的结构：

```
<?xml version="1.0" encoding="UTF-8" ?>
<book id="1">
    <name>Java核心技术</name>
    <author>Cay S. Horstmann</author>
    <isbn lang="CN">1234567</isbn>
    <tags>
        <tag>Java</tag>
        <tag>Network</tag>
    </tags>
    <pubDate/>
</book>
```

我们发现，它完全可以对应到一个定义好的JavaBean中：

```
public class Book {
    public long id;
    public String name;
    public String author;
    public String isbn;
    public List<String> tags;
    public String pubDate;
}
```

如果能直接从XML文档解析成一个JavaBean，那比DOM或者SAX不知道容易到哪里去了。

幸运的是，一个名叫Jackson的开源的第三方库可以轻松做到XML到JavaBean的转换。我们要使用Jackson，先添加两个Maven的依赖：

```
<dependency>
    <groupId>com.fasterxml.jackson.dataformat</groupId>
    <artifactId>jackson-dataformat-xml</artifactId>
    <version>2.10.1</version>
</dependency>
<dependency>
    <groupId>org.codehaus.woodstox</groupId>
    <artifactId>woodstox-core-asl</artifactId>
```

```
    <version>4.4.1</version>
</dependency>
```

然后，定义好JavaBean，就可以用下面几行代码解析：

```
InputStream input = Main.class.getResourceAsStream("/book.xml");
JacksonXmlModule module = new JacksonXmlModule();
XmlMapper mapper = new XmlMapper(module);
Book book = mapper.readValue(input, Book.class);
System.out.println(book.id);
System.out.println(book.name);
System.out.println(book.author);
System.out.println(book.isbn);
System.out.println(book.tags);
System.out.println(book.pubDate);
```

注意到`XmlMapper`就是我们需要创建的核心对象，可以用`readValue(InputStream, Class)`直接读取XML并返回一个JavaBean。运行上述代码，就可以直接从Book对象中拿到数据：

```
1
Java核心技术
Cay S. Horstmann
1234567
[Java, Network]
null
```

如果要解析的数据格式不是Jackson内置的标准格式，那么需要编写一点额外的扩展来告诉Jackson如何自定义解析。这里我们不做深入讨论，可以参考Jackson的官方文档。

练习

下载练习：使用Jackson解析XML （推荐使用IDE练习插件快速下载）

小结

使用Jackson解析XML，可以直接把XML解析为JavaBean，十分方便。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





使用JSON

前面我们讨论了XML这种数据格式。XML的特点是功能全面，但标签繁琐，格式复杂。在Web上使用XML现在越来越少，取而代之的是JSON这种数据结构。

JSON是JavaScript Object Notation的缩写，它去除了所有JavaScript执行代码，只保留JavaScript的对象格式。一个典型的JSON如下：

```
{
    "id": 1,
    "name": "Java核心技术",
    "author": {
        "firstName": "Abc",
        "lastName": "Xyz"
    },
    "isbn": "1234567",
    "tags": ["Java", "Network"]
}
```

JSON作为数据传输的格式，有几个显著的优点：

- JSON只允许使用UTF-8编码，不存在编码问题；
- JSON只允许使用双引号作为key，特殊字符用 `\` 转义，格式简单；
- 浏览器内置JSON支持，如果把数据用JSON发送给浏览器，可以用JavaScript直接处理。

因此，JSON适合表示层次结构，因为它格式简单，仅支持以下几种数据类型：

- 键值对：`{"key": value}`
- 数组：`[1, 2, 3]`
- 字符串：`"abc"`
- 数值（整数和浮点数）：`12.34`
- 布尔值：`true` 或 `false`
- 空值：`null`

浏览器直接支持使用JavaScript对JSON进行读写：

```
// JSON string to JavaScript object:
jsObj = JSON.parse(jsonStr);

// JavaScript object to JSON string:
jsonStr = JSON.stringify(jsObj);
```

所以，开发Web应用的时候，使用JSON作为数据传输，在浏览器端非常方便。因为JSON天生适合JavaScript处理，所以，绝大多数REST API都选择JSON作为数据传输格式。

现在问题来了：使用Java如何对JSON进行读写？

在Java中，针对JSON也有标准的JSR 353 API，但是我们在前面讲XML的时候发现，如果能直接在XML和JavaBean之间互相转换是最好的。类似的，如果能直接在JSON和JavaBean之间转换，那么用起来就简单多了。

常用的用于解析JSON的第三方库有：

- Jackson
- Gson
- Fastjson
- …

注意到上一节提到的那个可以解析XML的浓眉大眼的Jackson也可以解析JSON！因此我们只需要引入以下Maven依赖：

```
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.0</version>
</dependency>
```

就可以使用下面的代码解析一个JSON文件：

```
InputStream input = Main.class.getResourceAsStream("/book.json");
ObjectMapper mapper = new ObjectMapper();
// 反序列化时忽略不存在的JavaBean属性:
mapper.configure(DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES,
false);
Book book = mapper.readValue(input, Book.class);
```

核心代码是创建一个 `ObjectMapper` 对象。关闭 `DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES` 功能使得解析时如果JavaBean不存在该属性时解析不会报错。

把JSON解析为JavaBean的过程称为反序列化。如果把JavaBean变为JSON，那就是序列化。要实现JavaBean到JSON的序列化，只需要一行代码：

```
String json = mapper.writeValueAsString(book);
```

要把JSON的某些值解析为特定的Java对象，例如 `LocalDate` ，也是完全可以的。例如：

```
{
    "name": "Java核心技术",
    "pubDate": "2016-09-01"
}
```

要解析为：

```
public class Book {
    public String name;
    public LocalDate pubDate;
}
```

只需要引入标准的JSR 310关于JavaTime的数据格式定义：

```
<dependency>
    <groupId>com.fasterxml.jackson.datatype</groupId>
    <artifactId>jackson-datatype-jsr310</artifactId>
    <version>2.10.0</version>
</dependency>
```

然后，在创建 `ObjectMapper` 时，注册一个新的 `JavaTimeModule` ：

```
ObjectMapper mapper = new ObjectMapper().registerModule(new
JavaTimeModule());
```

有些时候，内置的解析规则和扩展的解析规则如果都不满足我们的需求，还可以自定义解析。

举个例子，假设 `Book` 类的 `isbn` 是一个 `BigInteger` ：

```
public class Book {
    public String name;
    public BigInteger isbn;
}
```

但JSON数据并不是标准的整形格式：

```
{
    "name": "Java核心技术",
    "isbn": "978-7-111-54742-6"
}
```

直接解析，肯定报错。这时，我们需要自定义一个 `IsbnDeserializer` ，用于解析含有非数字的字符串：

```
public class IsbnDeserializer extends JsonDeserializer<BigInteger> {
    public BigInteger deserialize(JsonParser p, DeserializationContext
ctxt) throws IOException, JsonProcessingException {
        // 读取原始的JSON字符串内容:
        String s = p.getValueAsString();
        if (s != null) {
            try {
                return new BigInteger(s.replace("-", ""));
            } catch (NumberFormatException e) {
                throw new JsonParseException(p, s, e);
            }
        }
        return null;
    }
}
```

然后，在 `Book` 类中使用注解标注：

```
public class Book {
    public String name;
    // 表示反序列化isbn时使用自定义的IsbnDeserializer:
    @JsonDeserialize(using = IsbnDeserializer.class)
    public BigInteger isbn;
}
```

类似的，自定义序列化时我们需要自定义一个 `IsbnSerializer` ，然后在 `Book` 类中标注 `@JsonSerialize(using = …)` 即可。

练习

下载练习：使用Jackson解析JSON （推荐使用IDE练习插件快速下载）

小结

JSON是轻量级的数据表示方式，常用于Web应用；

Jackson可以实现JavaBean和JSON之间的转换；

可以通过Module扩展Jackson能处理的数据类型；

可以自定义 `JsonSerializer` 和 `JsonDeserializer` 来定制序列化和反序列化。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论

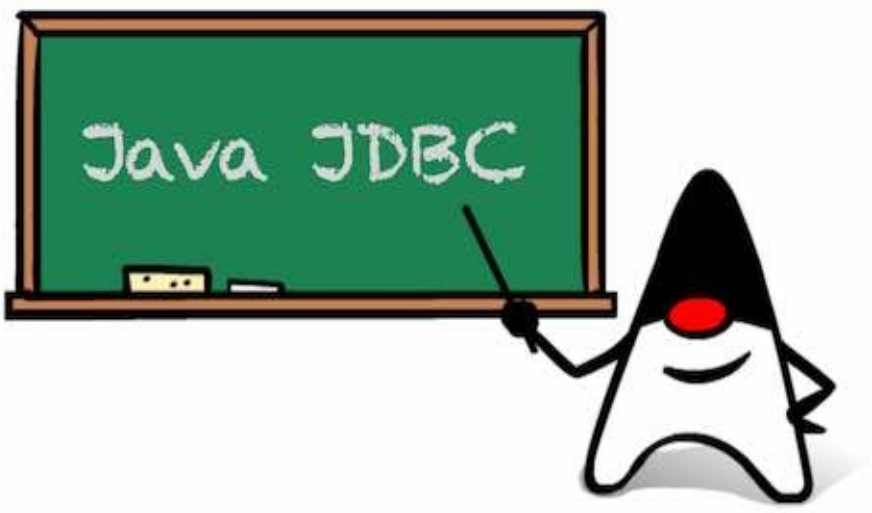




JDBC编程

程序运行的时候，往往需要存取数据。现代应用程序最基本，也是使用最广泛的数据存储就是关系数据库。

Java为关系数据库定义了一套标准的访问接口：JDBC（Java Database Connectivity），本章我们介绍如何在Java程序中使用JDBC。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





JDBC简介

在介绍JDBC之前，我们先简单介绍一下关系数据库。

程序运行的时候，数据都是在内存中的。当程序终止的时候，通常都需要将数据保存到磁盘上，无论是保存到本地磁盘，还是通过网络保存到服务器上，最终都会将数据写入磁盘文件。

而如何定义数据的存储格式就是一个大问题。如果我们自己来定义存储格式，比如保存一个班级所有学生的成绩单：

名字	成绩
Michael	99
Bob	85
Bart	59
Lisa	87

你可以用一个文本文件保存，一行保存一个学生，用 `,` 隔开：

```
Michael,99
Bob,85
Bart,59
Lisa,87
```

你还可以用JSON格式保存，也是文本文件：

```
[
    {"name":"Michael","score":99},
    {"name":"Bob","score":85},
    {"name":"Bart","score":59},
    {"name":"Lisa","score":87}
]
```

你还可以定义各种保存格式，但是问题来了：

存储和读取需要自己实现，JSON还是标准，自己定义的格式就各式各样了；

不能做快速查询，只有把数据全部读到内存中才能自己遍历，但有时候数据的大小远远超过了内存（比如蓝光电影，40GB的数据），根本无法全部读入内存。

为了便于程序保存和读取数据，而且，能直接通过条件快速查询到指定的数据，就出现了数据库（Database）这种专门用于集中存储和查询的软件。

数据库软件诞生的历史非常久远，早在1950年数据库就诞生了。经历了网状数据库，层次数据库，我们现在广泛使用的关系数据库是20世纪70年代基于关系模型的基础上诞生的。

关系模型有一套复杂的数学理论，但是从概念上是十分容易理解的。举个学校的例子：

假设某个XX省YY市ZZ县第一实验小学有3个年级，要表示出这3个年级，可以在Excel中用一个表格画出来：

Grade_ID	Name
1	一年级
2	二年级
3	三年级

每个年级又有若干个班级，要把所有班级表示出来，可以在Excel中再画一个表格：

Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

这两个表格有个映射关系，就是根据Grade_ID可以在班级表中查找到对应的所有班级：

Grade_ID	Name
1	一年级
2	二年级
3	三年级



Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

也就是Grade表的每一行对应Class表的多行，在关系数据库中，这种基于表（Table）的一对多的关系就是关系数据库的基础。

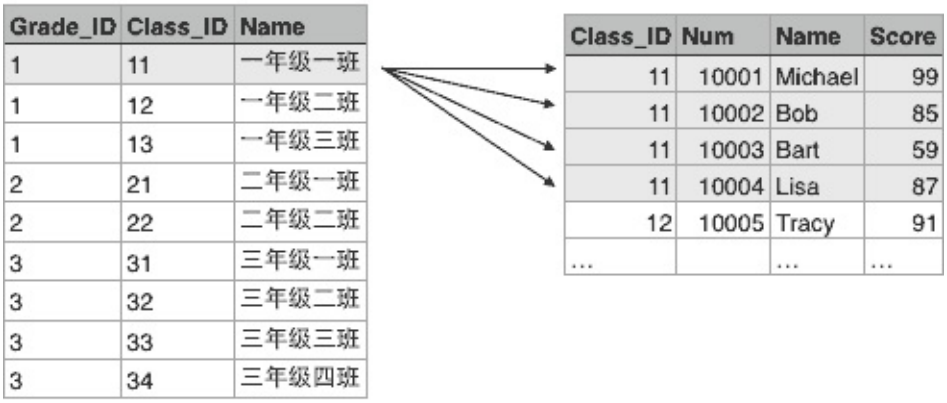
根据某个年级的ID就可以查找所有班级的行，这种查询语句在关系数据库中称为SQL语句，可以写成：

```
SELECT * FROM classes WHERE grade_id = '1';
```

结果也是一个表：

```
---------+----------+----------
grade_id | class_id | name
---------+----------+----------
1        | 11       | 一年级一班
---------+----------+----------
1        | 12       | 一年级二班
---------+----------+----------
1        | 13       | 一年级三班
---------+----------+----------
```

类似的，Class表的一行记录又可以关联到Student表的多行记录：



Grade_ID	Class_ID	Name
1	11	一年级一班
1	12	一年级二班
1	13	一年级三班
2	21	二年级一班
2	22	二年级二班
3	31	三年级一班
3	32	三年级二班
3	33	三年级三班
3	34	三年级四班

Class_ID	Num	Name	Score
11	10001	Michael	99
11	10002	Bob	85
11	10003	Bart	59
11	10004	Lisa	87
12	10005	Tracy	91
...		...	...

由于本教程不涉及到关系数据库的详细内容，如果你想从零学习关系数据库和基本的SQL语句，请参考SQL课程。

NoSQL

你也许还听说过NoSQL数据库，很多NoSQL宣传其速度和规模远远超过关系数据库，所以很多同学觉得有了NoSQL是否就不需要SQL了呢？千万不要被他们忽悠了，连SQL都不明白怎么可能搞明白NoSQL呢？

数据库类别

既然我们要使用关系数据库，就必须选择一个关系数据库。目前广泛使用的关系数据库也就这么几种：

付费的商用数据库：

- Oracle，典型的高富帅；

- SQL Server，微软自家产品，Windows定制专款；
- DB2，IBM的产品，听起来挺高端；
- Sybase，曾经跟微软是好基友，后来关系破裂，现在家境惨淡。

这些数据库都是不开源而且付费的，最大的好处是花了钱出了问题可以找厂家解决，不过在Web的世界里，常常需要部署成千上万的数据库服务器，当然不能把大把大把的银子扔给厂家，所以，无论是Google、Facebook，还是国内的BAT，无一例外都选择了免费的开源数据库：

- MySQL，大家都在用，一般错不了；
- PostgreSQL，学术气息有点重，其实挺不错，但知名度没有MySQL高；
- sqlite，嵌入式数据库，适合桌面和移动应用。

作为一个Java工程师，选择哪个免费数据库呢？当然是MySQL。因为MySQL普及率最高，出了错，可以很容易找到解决方法。而且，围绕MySQL有一大堆监控和运维的工具，安装和使用很方便。

安装MySQL

为了能继续后面的学习，你需要从MySQL官方网站下载并安装MySQL Community Server 5.6，这个版本是免费的，其他高级版本是要收钱的（请放心，收钱的功能我们用不上）。MySQL是跨平台的，选择对应的平台下载安装文件，安装即可。

安装时，MySQL会提示输入 `root` 用户的口令，请务必记清楚。如果怕记不住，就把口令设置为 `password` 。

在Windows上，安装时请选择 `UTF-8` 编码，以便正确地处理中文。

在Mac或Linux上，需要编辑MySQL的配置文件，把数据库默认的编码全部改为UTF-8。MySQL的配置文件默认存放在 `/etc/my.cnf` 或者 `/etc/mysql/my.cnf` ：

```
[client]
default-character-set = utf8

[mysqld]
default-storage-engine = INNODB
character-set-server = utf8
collation-server = utf8_general_ci
```

重启MySQL后，可以通过MySQL的客户端命令行检查编码：

```
$ mysql -u root -p
```

```
Enter password:
Welcome to the MySQL monitor...
...

mysql> show variables like '%char%';
+--------------------------+-----------------------------------------------------
-----+
| Variable_name            | Value
|
+--------------------------+-----------------------------------------------------
-----+
| character_set_client     | utf8
|
| character_set_connection | utf8
|
| character_set_database   | utf8
|
| character_set_filesystem | binary
|
| character_set_results    | utf8
|
| character_set_server     | utf8
|
| character_set_system     | utf8
|
| character_sets_dir       | /usr/local/mysql-5.1.65-osx10.6-
x86_64/share/charsets/ |
+--------------------------+-----------------------------------------------------
-----+
8 rows in set (0.00 sec)
```

看到 `utf8` 字样就表示编码设置正确。

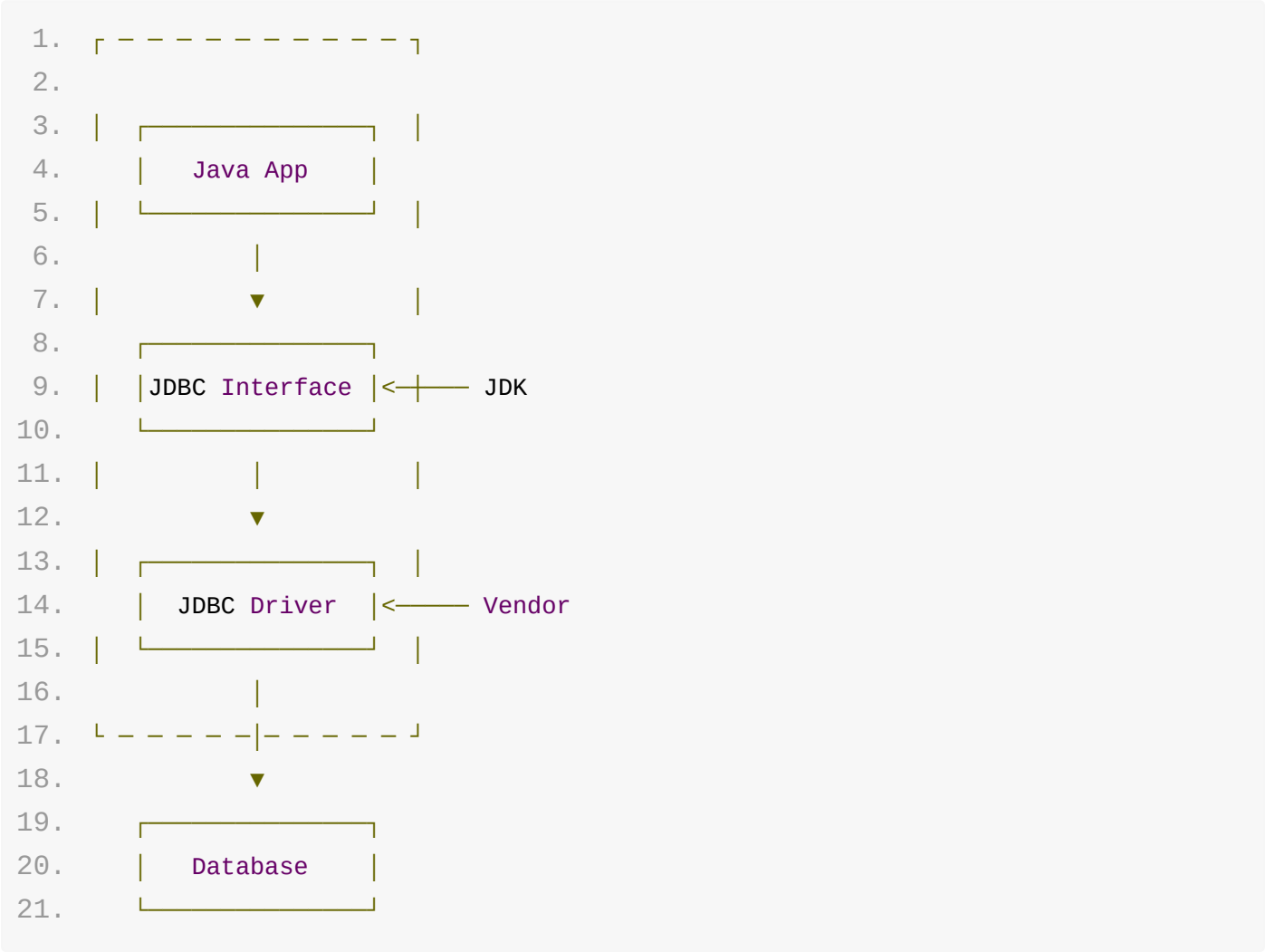
注：如果MySQL的版本≥5.5.3，可以把编码设置为 `utf8mb4` ， `utf8mb4` 和 `utf8` 完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

JDBC

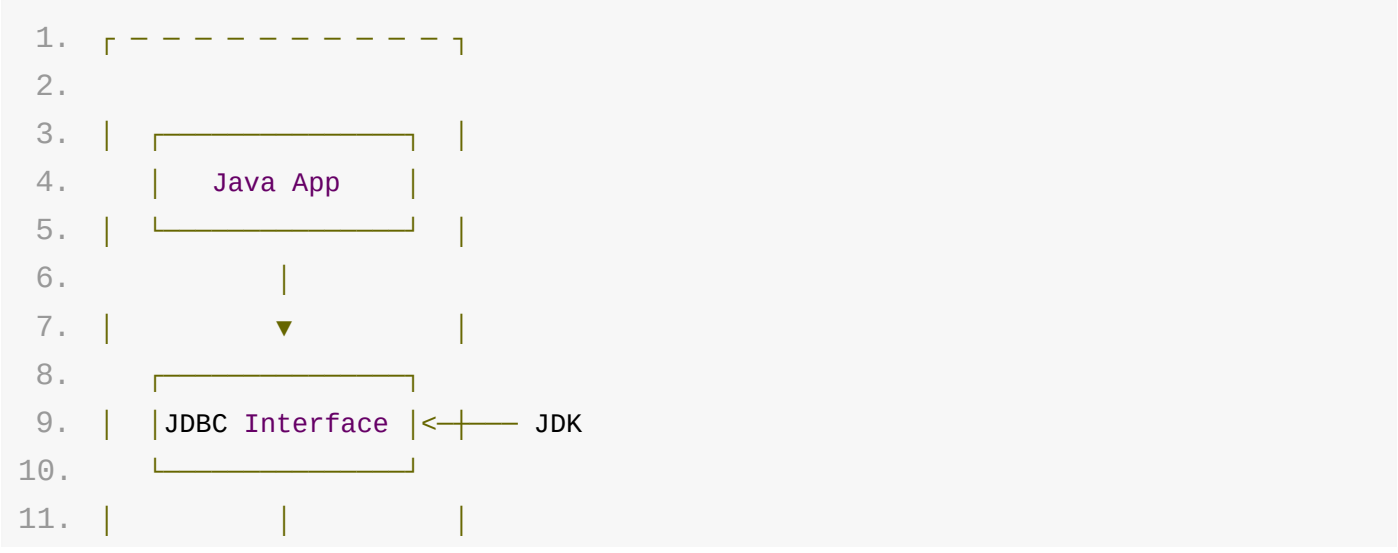
什么是JDBC？JDBC是Java DataBase Connectivity的缩写，它是Java程序访问数据库的标准接口。

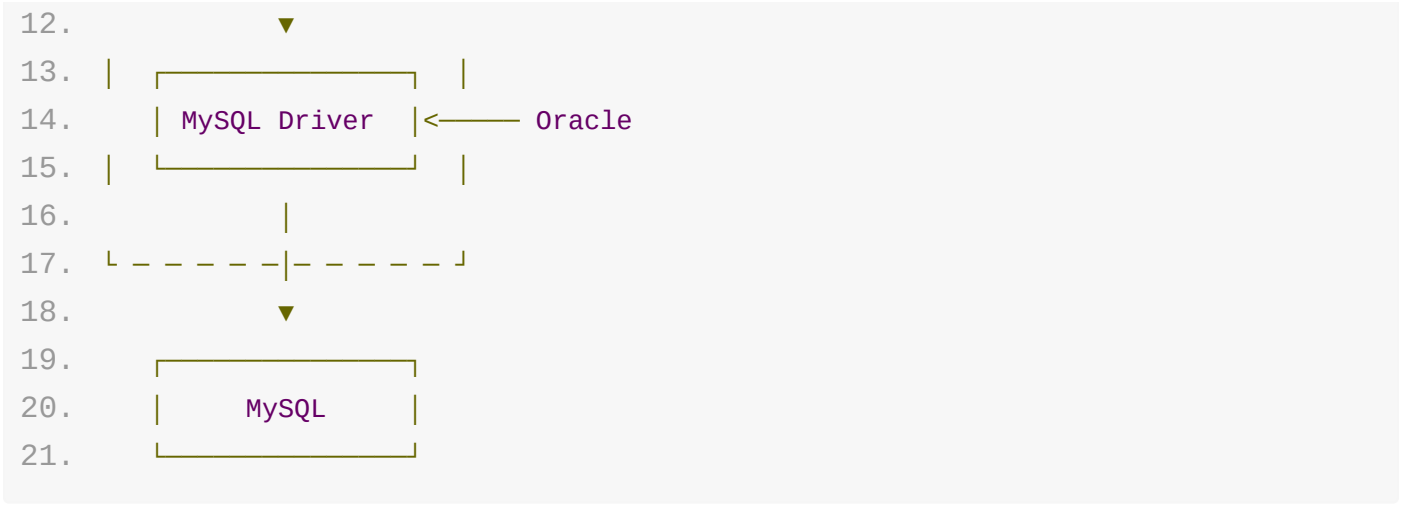
使用Java程序访问数据库时，Java代码并不是直接通过TCP连接去访问数据库，而是通过JDBC接口来访问，而JDBC接口则通过JDBC驱动来实现真正对数据库的访问。

例如，我们在Java代码中如果要访问MySQL，那么必须编写代码操作JDBC接口。注意到JDBC接口是Java标准库自带的，所以可以直接编译。而具体的JDBC驱动是由数据库厂商提供的，例如，MySQL的JDBC驱动由Oracle提供。因此，访问某个具体的数据库，我们只需要引入该厂商提供的JDBC驱动，就可以通过JDBC接口来访问，这样保证了Java程序编写的是一套数据库访问代码，却可以访问各种不同的数据库，因为他们都提供了标准的JDBC驱动：

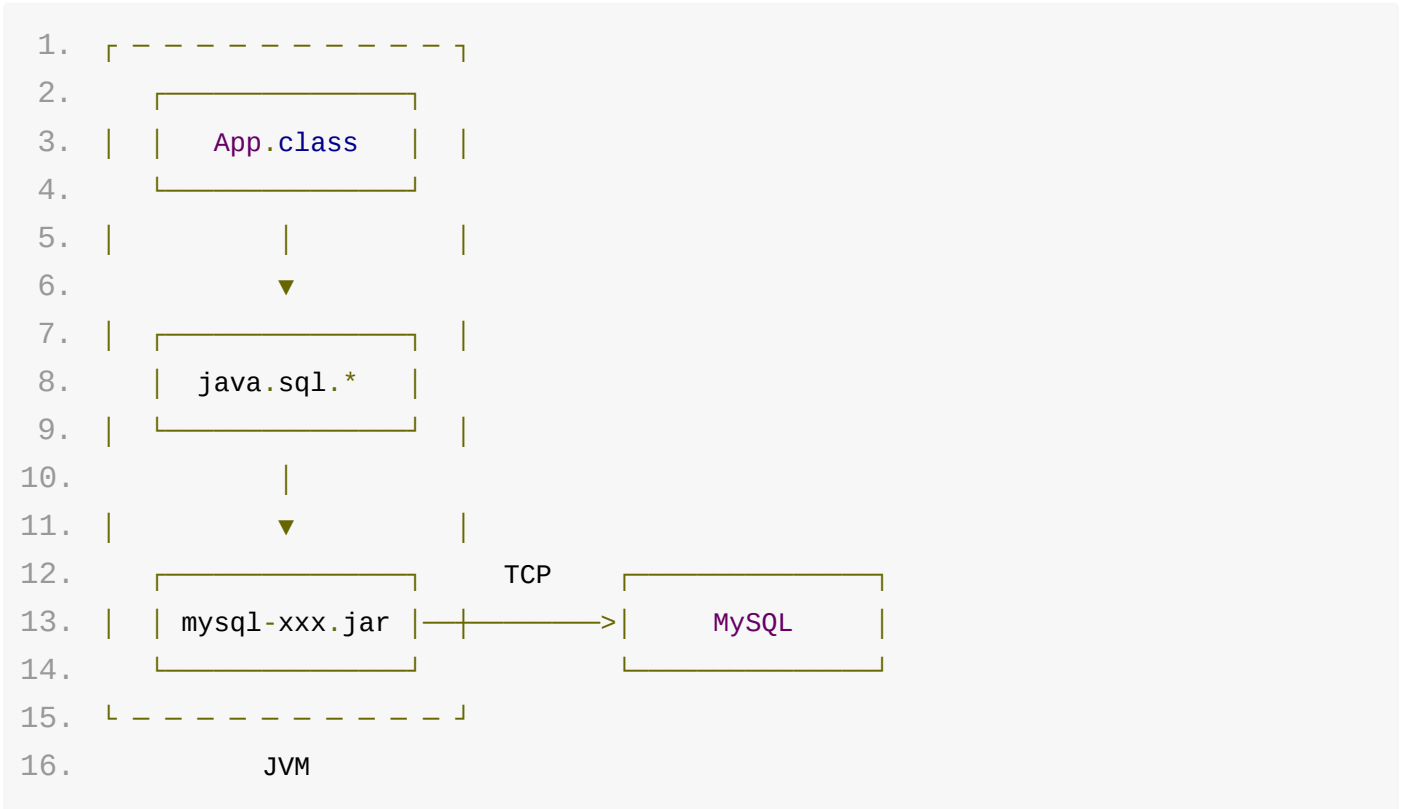


从代码来看，Java标准库自带的JDBC接口其实就是定义了一组接口，而某个具体的JDBC驱动其实就是实现了这些接口的类：





实际上，一个MySQL的JDBC的驱动就是一个jar包，它本身也是纯Java编写的。我们自己编写的代码只需要引用Java标准库提供的java.sql包下面的相关接口，由此再间接地通过MySQL驱动的jar包通过网络访问MySQL服务器，所有复杂的网络通讯都被封装到JDBC驱动中，因此，Java程序本身只需要引入一个MySQL驱动的jar包就可以正常访问MySQL服务器：



小结

使用JDBC的好处是：

- 各数据库厂商使用相同的接口，Java代码不需要针对不同数据库分别开发；
- Java程序编译期仅依赖java.sql包，不依赖具体数据库的jar包；
- 可随时替换底层数据库，访问数据库的Java代码基本不变。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





JDBC查询

前面我们讲了Java程序要通过JDBC接口来查询数据库。JDBC是一套接口规范，它在哪呢？就在Java的标准库 `java.sql` 里放着，不过这里面大部分都是接口。接口并不能直接实例化，而是必须实例化对应的实现类，然后通过接口引用这个实例。那么问题来了：JDBC接口的实现类在哪？

因为JDBC接口并不知道我们要使用哪个数据库，所以，用哪个数据库，我们就去使用哪个数据库的“实现类”，我们把某个数据库实现了JDBC接口的jar包称为JDBC驱动。

因为我们选择了MySQL 5.x作为数据库，所以我们首先得找一个MySQL的JDBC驱动。所谓JDBC驱动，其实就是一个第三方jar包，我们直接添加一个Maven依赖就可以了：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.47</version>
    <scope>runtime</scope>
</dependency>
```

注意到这里添加依赖的 `scope` 是 `runtime` ，因为编译Java程序并不需要MySQL的这个jar包，只有在运行期才需要使用。如果把 `runtime` 改成 `compile` ，虽然也能正常编译，但是在IDE里写程序的时候，会多出来一大堆类似 `com.mysql.jdbc.Connection` 这样的类，非常容易与Java标准库的JDBC接口混淆，所以坚决不要设置为 `compile` 。

有了驱动，我们还要确保MySQL在本机正常运行，并且还需要准备一点数据。这里我们用一个脚本创建数据库和表，然后插入一些数据：

```
-- 创建数据库learjdbc:
DROP DATABASE IF EXISTS learnjdbc;
CREATE DATABASE learnjdbc;

-- 创建登录用户learn/口令learnpassword
CREATE USER IF NOT EXISTS learn@'%' IDENTIFIED BY 'learnpassword';
GRANT ALL PRIVILEGES ON learnjdbc.* TO learn@'%' WITH GRANT OPTION;
FLUSH PRIVILEGES;

-- 创建表students:
USE learnjdbc;
CREATE TABLE students (
  id BIGINT AUTO_INCREMENT NOT NULL,
  name VARCHAR(50) NOT NULL,
  gender TINYINT(1) NOT NULL,
```

```
  grade INT NOT NULL,
  score INT NOT NULL,
  PRIMARY KEY(id)
) Engine=INNODB DEFAULT CHARSET=UTF8;

-- 插入初始数据:
INSERT INTO students (name, gender, grade, score) VALUES ('小明', 1, 1, 88);
INSERT INTO students (name, gender, grade, score) VALUES ('小红', 1, 1, 95);
INSERT INTO students (name, gender, grade, score) VALUES ('小军', 0, 1, 93);
INSERT INTO students (name, gender, grade, score) VALUES ('小白', 0, 1, 100);
INSERT INTO students (name, gender, grade, score) VALUES ('小牛', 1, 2, 96);
INSERT INTO students (name, gender, grade, score) VALUES ('小兵', 1, 2, 99);
INSERT INTO students (name, gender, grade, score) VALUES ('小强', 0, 2, 86);
INSERT INTO students (name, gender, grade, score) VALUES ('小乔', 0, 2, 79);
INSERT INTO students (name, gender, grade, score) VALUES ('小青', 1, 3, 85);
INSERT INTO students (name, gender, grade, score) VALUES ('小王', 1, 3, 90);
INSERT INTO students (name, gender, grade, score) VALUES ('小林', 0, 3, 91);
INSERT INTO students (name, gender, grade, score) VALUES ('小贝', 0, 3, 77);
```

在控制台输入 `mysql -u root -p` ，输入 `root` 口令后以 `root` 身份，把上述SQL贴到控制台执行一遍就行。如果你运行的是最新版MySQL 8.x，需要调整一下 `CREATE USER` 语句。

JDBC连接

使用JDBC时，我们先了解什么是Connection。Connection代表一个JDBC连接，它相当于Java程序到数据库的连接（通常是TCP连接）。打开一个Connection时，需要准备URL、用户名和口令，才能成功连接到数据库。

URL是由数据库厂商指定的格式，例如，MySQL的URL是：

```
jdbc:mysql://<hostname>:<port>/<db>?key1=value1&key2=value2
```

假设数据库运行在本机 `localhost` ，端口使用标准的 `3306` ，数据库名称是 `learnjdbc` ，那么URL如下：

```
jdbc:mysql://localhost:3306/learnjdbc?useSSL=false&characterEncoding=utf8
```

后面的两个参数表示不使用SSL加密，使用UTF-8作为字符编码（注意MySQL的UTF-8是 `utf8` ）。

要获取数据库连接，使用如下代码：

```
// JDBC连接的URL, 不同数据库有不同的格式:
```

```
String JDBC_URL = "jdbc:mysql://localhost:3306/test";
String JDBC_USER = "root";
String JDBC_PASSWORD = "password";
// 获取连接:
Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD);
// TODO: 访问数据库...
// 关闭连接:
conn.close();
```

核心代码是 `DriverManager` 提供的静态方法 `getConnection()` 。 `DriverManager` 会自动扫描classpath，找到所有的JDBC驱动，然后根据我们传入的URL自动挑选一个合适的驱动。

因为JDBC连接是一种昂贵的资源，所以使用后要及时释放。使用 `try (resource)` 来自动释放JDBC连接是一个好方法：

```
try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD)) {
    ...
}
```

JDBC查询

获取到JDBC连接后，下一步我们就可以查询数据库了。查询数据库分以下几步：

第一步，通过 `Connection` 提供的 `createStatement()` 方法创建一个 `Statement` 对象，用于执行一个查询；

第二步，执行 `Statement` 对象提供的 `executeQuery("SELECT * FROM students")` 并传入SQL语句，执行查询并获得返回的结果集，使用 `ResultSet` 来引用这个结果集；

第三步，反复调用 `ResultSet` 的 `next()` 方法并读取每一行结果。

完整查询代码如下：

```
try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
    try (Statement stmt = conn.createStatement()) {
        try (ResultSet rs = stmt.executeQuery("SELECT id, grade, name, gender
FROM students WHERE gender=\'M\'")) {
            while (rs.next()) {
                long id = rs.getLong(1); // 注意：索引从1开始
                long grade = rs.getLong(2);
```

```
                String name = rs.getString(3);
                String gender = rs.getString(4);
            }
        }
    }
}
```

注意要点：

`Statment` 和 `ResultSet` 都是需要关闭的资源，因此嵌套使用 `try (resource)` 确保及时关闭；

`rs.next()` 用于判断是否有下一行记录，如果有，将自动把当前行移动到下一行（一开始获得 `ResultSet` 时当前行不是第一行）；

`ResultSet` 获取列时，索引从 `1` 开始而不是 `0` ；

必须根据 `SELECT` 的列的对应位置来调用 `getLong(1)` ， `getString(2)` 这些方法，否则对应位置的数据类型不对，将报错。

SQL注入

使用 `Statement` 拼字符串非常容易引发SQL注入的问题，这是因为SQL参数往往是从方法参数传入的。

我们来看一个例子：假设用户登录的验证方法如下：

```
User login(String name, String pass) {
    ...
    stmt.executeQuery("SELECT * FROM user WHERE login='" + name + "' AND
pass='" + pass + "'");
    ...
}
```

其中，参数 `name` 和 `pass` 通常都是Web页面输入后由程序接收到的。

如果用户的输入是程序期待的值，就可以拼出正确的SQL。例如：name = `"bob"` , pass = `"1234"` ：

```
SELECT * FROM user WHERE login='bob' AND pass='1234'
```

但是，如果用户的输入是一个精心构造的字符串，就可以拼出意想不到的SQL，这个SQL也是正确的，但它查询的条件不是程序设计的意图。例如：name = `"bob' OR pass="` , pass = `" OR`

`pass=''` :

```
SELECT * FROM user WHERE login='bob' OR pass=' AND pass=' OR pass=''
```

这个SQL语句执行的时候，根本不用判断口令是否正确，这样一来，登录就形同虚设。

要避免SQL注入攻击，一个办法是针对所有字符串参数进行转义，但是转义很麻烦，而且需要在任何使用SQL的地方增加转义代码。

还有一个办法就是使用 `PreparedStatement` 。使用 `PreparedStatement` 可以完全避免*SQL*注入的问题，因为 `PreparedStatement` 始终使用 `?` 作为占位符，并且把数据连同SQL本身传给数据库，这样可以保证每次传给数据库的SQL语句是相同的，只是占位符的数据不同，还能高效利用数据库本身对查询的缓存。上述登录SQL如果用 `PreparedStatement` 可以改写如下：

```
User login(String name, String pass) {
    ...
    String sql = "SELECT * FROM user WHERE login=? AND pass=?";
    PreparedStatement ps = conn.prepareStatement(sql);
    ps.setObject(1, name);
    ps.setObject(2, pass);
    ...
}
```

所以， `PreparedStatement` 比 `Statement` 更安全，而且更快。

使用Java对数据库进行操作时，必须使用PreparedStatement，严禁任何通过参数拼字符串的代码！

我们把上面使用 `Statement` 的代码改为使用 `PreparedStatement` ：

```
    try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
JDBC_PASSWORD) {
        try (PreparedStatement ps = conn.prepareStatement("SELECT id, grade, name,
gender FROM students WHERE gender=? AND grade=?")) {
            ps.setObject(1, "M"); // 注意：索引从1开始
            ps.setObject(2, 3);
            try (ResultSet rs = stmt.executeQuery()) {
                while (rs.next()) {
                    long id = rs.getLong("id");
                    long grade = rs.getLong("grade");
                    String name = rs.getString("name");
                    String gender = rs.getString("gender");
                }
```

```
        }
    }
}
```

使用 `PreparedStatement` 和 `Statement` 稍有不同，必须首先调用 `setObject()` 设置每个占位符 `?` 的值，最后获取的仍然是 `ResultSet` 对象。

另外注意到从结果集读取列时，使用 `String` 类型的列名比索引要易读，而且不易出错。

注意到JDBC查询的返回值总是 `ResultSet` ，即使我们写这样的聚合查询 `SELECT SUM(score) FROM …` ，也需要按结果集读取：

```
ResultSet rs = ...
if (rs.next()) {
    double sum = rs.getDouble(1);
}
```

数据类型

有的童鞋可能注意到了，使用JDBC的时候，我们需要在Java数据类型和SQL数据类型之间进行转换。JDBC在 `java.sql.Types` 定义了一组常量来表示如何映射SQL数据类型，但是平时我们使用的类型通常也就以下几种：

SQL数据类型	Java数据类型
BIT, BOOL	boolean
INTEGER	int
BIGINT	long
REAL	float
FLOAT, DOUBLE	double
CHAR, VARCHAR	String
DECIMAL	BigDecimal
DATE	java.sql.Date, LocalDate
TIME	java.sql.Time, LocalTime

注意：只有最新的JDBC驱动才支持 `LocalDate` 和 `LocalTime` 。

练习

下载练习：使用JDBC查询数据库 （推荐使用IDE练习插件快速下载）

小结

JDBC接口的 `Connection` 代表一个JDBC连接;

使用JDBC查询时，总是使用 `PreparedStatement` 进行查询而不是 `Statement` ;

查询结果总是 `ResultSet` ，即使使用聚合查询也不例外。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





JDBC更新

数据库操作总结起来就四个字：增删改查，行话叫CRUD：Create，Retreive，Update和Delete。

查就是查询，我们已经讲过了，就是使用 `PreparedStatement` 进行各种 `SELECT` ，然后处理结果集。现在我们来看看如何使用JDBC进行增删改。

插入

插入操作是 `INSERT` ，即插入一条新记录。通过JDBC进行插入，本质上也是用 `PreparedStatement` 执行一条SQL语句，不过最后执行的不是 `executeQuery()` ，而是 `executeUpdate()` 。示例代码如下：

```
    try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
1.  JDBC_PASSWORD) {
2.      try (PreparedStatement ps = conn.prepareStatement(
                "INSERT INTO students (id, grade, name, gender) VALUES (?,?,?,?)"))
3.  {
4.          ps.setObject(1, 999); // 注意：索引从1开始
5.          ps.setObject(2, 1); // grade
6.          ps.setObject(3, "Bob"); // name
7.          ps.setObject(4, "M"); // gender
8.          int n = ps.executeUpdate(); // 1
9.      }
10. }
```

设置参数与查询是一样的，有几个 `?` 占位符就必须设置对应的参数。虽然 `Statement` 也可以执行插入操作，但我们仍然要严格遵循绝不能手动拼*SQL*字符串的原则，以避免安全漏洞。

当成功执行 `executeUpdate()` 后，返回值是 `int` ，表示插入的记录数量。此处总是 `1` ，因为只插入了一条记录。

插入并获取主键

如果数据库的表设置了自增主键，那么在执行 `INSERT` 语句时，并不需要指定主键，数据库会自动分配主键。对于使用自增主键的程序，有个额外的步骤，就是如何获取插入后的自增主键的值。

要获取自增主键，不能先插入，再查询。因为两条SQL执行期间可能有别的程序也插入了同一个表。获取自增主键的正确写法是在创建 `PreparedStatement` 的时候，指定一个 `RETURN_GENERATED_KEYS` 标志位，表示JDBC驱动必须返回插入的自增主键。示例代码如下：

```
    try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
1.  JDBC_PASSWORD) {
2.      try (PreparedStatement ps = conn.prepareStatement(
3.              "INSERT INTO students (grade, name, gender) VALUES (?,?,?)",
4.              Statement.RETURN_GENERATED_KEYS)) {
5.          ps.setObject(1, 1); // grade
6.          ps.setObject(2, "Bob"); // name
7.          ps.setObject(3, "M"); // gender
8.          int n = ps.executeUpdate(); // 1
9.          try (ResultSet rs = ps.getGeneratedKeys()) {
10.             if (rs.next()) {
11.                 long id = rs.getLong(1); // 注意：索引从1开始
12.             }
13.         }
14.     }
15. }
```

观察上述代码，有两点注意事项：

一是调用 `prepareStatement()` 时，第二个参数必须传入常量 `Statement.RETURN_GENERATED_KEYS` ，否则JDBC驱动不会返回自增主键；

二是执行 `executeUpdate()` 方法后，必须调用 `getGeneratedKeys()` 获取一个 `ResultSet` 对象，这个对象包含了数据库自动生成的主键的值，读取该对象的每一行来获取自增主键的值。如果一次插入多条记录，那么这个 `ResultSet` 对象就会有多行返回值。如果插入时有多列自增，那么 `ResultSet` 对象的每一行都会对应多个自增值（自增列不一定必须是主键）。

更新

更新操作是 `UPDATE` 语句，它可以一次更新若干列的记录。更新操作和插入操作在JDBC代码的层面上实际上没有区别，除了SQL语句不同：

```
    try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
1.  JDBC_PASSWORD) {
        try (PreparedStatement ps = conn.prepareStatement("UPDATE students SET
2.  name=? WHERE id=?")) {
3.          ps.setObject(1, "Bob"); // 注意：索引从1开始
4.          ps.setObject(2, 999);
5.          int n = ps.executeUpdate(); // 返回更新的行数
6.      }
7.  }
```

`executeUpdate()` 返回数据库实际更新的行数。返回结果可能是正数，也可能是0（表示没有任何记录更新）。

删除

删除操作是 `DELETE` 语句，它可以一次删除若干列。和更新一样，除了SQL语句不同外，JDBC代码都是相同的：

```
    try (Connection conn = DriverManager.getConnection(JDBC_URL, JDBC_USER,
1.  JDBC_PASSWORD) {
        try (PreparedStatement ps = conn.prepareStatement("DELETE FROM students
2.  WHERE id=?")) {
3.          ps.setObject(1, 999); // 注意：索引从1开始
4.          int n = ps.executeUpdate(); // 删除的行数
5.      }
6.  }
```

练习

下载练习：使用JDBC更新数据库 （推荐使用IDE练习插件快速下载）

小结

使用JDBC执行 `INSERT` 、 `UPDATE` 和 `DELETE` 都可视为更新操作；

更新操作使用 `PreparedStatement` 的 `executeUpdate()` 进行，返回受影响的行数。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论





JDBC事务

数据库事务（Transaction）是由若干个SQL语句构成的一个操作序列，有点类似于Java的 `synchronized` 同步。数据库系统保证在一个事务中的所有SQL要么全部执行成功，要么全部不执行，即数据库事务具有ACID特性：

- Atomicity：原子性
- Consistency：一致性
- Isolation：隔离性
- Durability：持久性

数据库事务可以并发执行，而数据库系统从效率考虑，对事务定义了不同的隔离级别。SQL标准定义了4种隔离级别，分别对应可能出现的数据不一致的情况：

Isolation Level	脏读（Dirty Read）	不可重复读（Non Repeatable Read）	幻读（Phantom Read）
Read Uncommitted	Yes	Yes	Yes
Read Committed	-	Yes	Yes
Repeatable Read	-	-	Yes
Serializable	-	-	-

对应用程序来说，数据库事务非常重要，很多运行着关键任务的应用程序，都必须依赖数据库事务保证程序的结果正常。

举个例子：假设小明准备给小红支付100，两人在数据库中的记录主键分别是 `123` 和 `456` ，那么用两条SQL语句操作如下：

```
1. UPDATE accounts SET balance = balance - 100 WHERE id=123 AND balance >= 100;
2. UPDATE accounts SET balance = balance + 100 WHERE id=456;
```

这两条语句必须以事务方式执行才能保证业务的正确性，因为一旦第一条SQL执行成功而第二条SQL失败的话，系统的钱就会凭空减少100，而有了事务，要么这笔转账成功，要么转账失败，双方账户的钱都不变。

这里我们不讨论详细的SQL事务，如果对SQL事务不熟悉，请参考SQL事务。

要在JDBC中执行事务，本质上就是如何把多条SQL包裹在一个数据库事务中执行。我们来看JDBC的事务代码：

```
1. Connection conn = openConnection();
```

```
try {
    // 关闭自动提交:
    conn.setAutoCommit(false);
    // 执行多条SQL语句:
    insert(); update(); delete();
    // 提交事务:
    conn.commit();
} catch (SQLException e) {
    // 回滚事务:
    conn.rollback();
} finally {
    conn.setAutoCommit(true);
    conn.close();
}
```

其中，开启事务的关键代码是 `conn.setAutoCommit(false)` ，表示关闭自动提交。提交事务的代码在执行完指定的若干条SQL语句后，调用 `conn.commit()` 。要注意事务不是总能成功，如果事务提交失败，会抛出SQL异常（也可能在执行SQL语句的时候就抛出了），此时我们必须捕获并调用 `conn.rollback()` 回滚事务。最后，在 `finally` 中通过 `conn.setAutoCommit(true)` 把 `Connection` 对象的状态恢复到初始值。

实际上，默认情况下，我们获取到 `Connection` 连接后，总是处于“自动提交”模式，也就是每执行一条SQL都是作为事务自动执行的，这也是为什么前面几节我们的更新操作总能成功的原因：因为默认有这种“隐式事务”。只要关闭了 `Connection` 的 `autoCommit` ，那么就可以在一个事务中执行多条语句，事务以 `commit()` 方法结束。

如果要设定事务的隔离级别，可以使用如下代码：

```
// 设定隔离级别为READ COMMITTED:
conn.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);
```

如果没有调用上述方法，那么会使用数据库的默认隔离级别。MySQL的默认隔离级别是 `REPEATABLE READ` 。

练习

从  下载练习：使用数据库事务 （推荐使用IDE练习插件快速下载）

小结

数据库事务（Transaction）具有ACID特性：

- Atomicity: 原子性
- Consistency: 一致性
- Isolation: 隔离性
- Durability: 持久性

JDBC提供了事务的支持，使用Connection可以开启、提交或回滚事务。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





JDBC Batch

使用JDBC操作数据库的时候，经常会执行一些批量操作。

例如，一次性给会员增加可用优惠券若干，我们可以执行以下SQL代码：

```
INSERT INTO coupons (user_id, type, expires) VALUES (123, 'DISCOUNT', '2030-12-31');
INSERT INTO coupons (user_id, type, expires) VALUES (234, 'DISCOUNT', '2030-12-31');
INSERT INTO coupons (user_id, type, expires) VALUES (345, 'DISCOUNT', '2030-12-31');
INSERT INTO coupons (user_id, type, expires) VALUES (456, 'DISCOUNT', '2030-12-31');
...
```

实际上执行JDBC时，因为只有占位符参数不同，所以SQL实际上是一样的：

```
for (var params : paramsList) {
    PreparedStatement ps = conn.preparedStatement("INSERT INTO coupons (user_id, type, expires) VALUES (?,?,?)");
    ps.setLong(params.get(0));
    ps.setString(params.get(1));
    ps.setString(params.get(2));
    ps.executeUpdate();
}
```

类似的还有，给每个员工薪水增加10%~30%：

```
UPDATE employees SET salary = salary * ? WHERE id = ?
```

通过一个循环来执行每个 `PreparedStatement` 虽然可行，但是性能很低。SQL数据库对SQL语句相同，但只有参数不同的若干语句可以作为batch执行，即批量执行，这种操作有特别优化，速度远远快于循环执行每个SQL。

在JDBC代码中，我们可以利用SQL数据库的这一特性，把同一个SQL但参数不同的若干次操作合并为一个batch执行。我们以批量插入为例，示例代码如下：

```
try (PreparedStatement ps = conn.prepareStatement("INSERT INTO students (name, gender, grade, score) VALUES (?, ?, ?, ?)")) {
    // 对同一个PreparedStatement反复设置参数并调用addBatch():
```

```
    for (String name : names) {
        ps.setString(1, name);
        ps.setBoolean(2, gender);
        ps.setInt(3, grade);
        ps.setInt(4, score);
        ps.addBatch(); // 添加到batch
    }
    // 执行batch:
    int[] ns = ps.executeBatch();
    for (int n : ns) {
        System.out.println(n + " inserted."); // batch中每个SQL执行的结果数量
    }
}
```

执行batch和执行一个SQL不同点在于，需要对同一个 `PreparedStatement` 反复设置参数并调用 `addBatch()` ，这样就相当于给一个SQL加上了多组参数，相当于变成了“多行”SQL。

第二个不同点是调用的不是 `executeUpdate()` ，而是 `executeBatch()` ，因为我们设置了多组参数，相应地，返回结果也是多个 `int` 值，因此返回类型是 `int[]` ，循环 `int[]` 数组即可获取每组参数执行后影响的结果数量。

练习

使用Batch操作

小结

使用JDBC的batch操作会大大提高执行效率，对内容相同，参数不同的SQL，要优先考虑batch操作。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





JDBC连接池

我们在讲多线程的时候说过，创建线程是一个昂贵的操作，如果有大量的小任务需要执行，并且频繁地创建和销毁线程，实际上会消耗大量的系统资源，往往创建和消耗线程所耗费的时间比执行任务的时间还长，所以，为了提高效率，可以用线程池。

类似的，在执行JDBC的增删改查的操作时，如果每一次操作都来一次打开连接，操作，关闭连接，那么创建和销毁JDBC连接的开销就太大了。为了避免频繁地创建和销毁JDBC连接，我们可以通过连接池（Connection Pool）复用已经创建好的连接。

JDBC连接池有一个标准的接口 `javax.sql.DataSource` ，注意这个类位于Java标准库中，但仅仅是接口。要使用JDBC连接池，我们必须选择一个JDBC连接池的实现。常用的JDBC连接池有：

- HikariCP
- C3P0
- BoneCP
- Druid

目前使用最广泛的是HikariCP。我们以HikariCP为例，要使用JDBC连接池，先添加HikariCP的依赖如下：

```
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>2.7.1</version>
</dependency>
```

紧接着，我们需要创建一个 `DataSource` 实例，这个实例就是连接池：

```
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:mysql://localhost:3306/test");
config.setUsername("root");
config.setPassword("password");
config.addDataSourceProperty("connectionTimeout", "1000"); // 连接超时：1秒
config.addDataSourceProperty("idleTimeout", "60000"); // 空闲超时：60秒
config.addDataSourceProperty("maximumPoolSize", "10"); // 最大连接数：10
DataSource ds = new HikariDataSource(config);
```

注意创建 `DataSource` 也是一个非常昂贵的操作，所以通常 `DataSource` 实例总是作为一个全局变量存储，并贯穿整个应用程序的生命周期。

有了连接池以后，我们如何使用它呢？和前面的代码类似，只是获取 `Connection` 时，把 `DriverManage.getConnection()` 改为 `ds.getConnection()` ：

```
try (Connection conn = ds.getConnection()) { // 在此获取连接
    ...
} // 在此“关闭”连接
```

通过连接池获取连接时，并不需要指定JDBC的相关URL、用户名、口令等信息，因为这些信息已经存储在连接池内部了（创建 `HikariDataSource` 时传入的 `HikariConfig` 持有这些信息）。一开始，连接池内部并没有连接，所以，第一次调用 `ds.getConnection()` ，会迫使连接池内部先创建一个 `Connection` ，再返回给客户端使用。当我们调用 `conn.close()` 方法时（ `在try(resource){…}` 结束处），不是真正“关闭”连接，而是释放到连接池中，以便下次获取连接时能直接返回。

因此，连接池内部维护了若干个 `Connection` 实例，如果调用 `ds.getConnection()` ，就选择一个空闲连接，并标记它为“正在使用”然后返回，如果对 `Connection` 调用 `close()` ，那么就把连接再次标记为“空闲”从而等待下次调用。这样一来，我们就通过连接池维护了少量连接，但可以频繁地执行大量的SQL语句。

通常连接池提供了大量的参数可以配置，例如，维护的最小、最大活动连接数，指定一个连接在空闲一段时间后自动关闭等，需要根据应用程序的负载合理地配置这些参数。此外，大多数连接池都提供了详细的实时状态以便进行监控。

练习

从 码云 下载练习：使用HikariCP连接池 （推荐使用IDE练习插件快速下载）

小结

数据库连接池是一种复用 `Connection` 的组件，它可以避免反复创建新连接，提高JDBC代码的运行效率；

可以配置连接池的详细参数并监控连接池。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





函数式编程

本章我们介绍Java的函数式编程。

我们先看看什么是函数。函数是一种最基本的任务，一个大型程序就是一个顶层函数调用若干底层函数，这些被调用的函数又可以调用其他函数，即大任务被一层层拆解并执行。所以函数就是面向过程的程序设计的基本单元。

Java不支持单独定义函数，但可以把静态方法视为独立的函数，把实例方法视为自带 `this` 参数的函数。

而函数式编程（请注意多了一个“式”字）——Functional Programming，虽然也可以归结到面向过程的程序设计，但其思想更接近数学计算。

我们首先要搞明白计算机（Computer）和计算（Compute）的概念。

在计算机的层次上，CPU执行的是加减乘除的指令代码，以及各种条件判断和跳转指令，所以，汇编语言是最贴近计算机的语言。

而计算则指数学意义上的计算，越是抽象的计算，离计算机硬件越远。

对应到编程语言，就是越低级的语言，越贴近计算机，抽象程度低，执行效率高，比如C语言；越高级的语言，越贴近计算，抽象程度高，执行效率低，比如Lisp语言。

函数式编程就是一种抽象程度很高的编程范式，纯粹的函数式编程语言编写的函数没有变量，因此，任意一个函数，只要输入是确定的，输出就是确定的，这种纯函数我们称之为没有副作用。而允许使用变量的程序设计语言，由于函数内部的变量状态不确定，同样的输入，可能得到不同的输出，因此，这种函数是有副作用的。

函数式编程的一个特点就是，允许把函数本身作为参数传入另一个函数，还允许返回一个函数！

函数式编程最早是数学家阿隆佐·邱奇研究的一套函数变换逻辑，又称Lambda Calculus（λ-Calculus），所以也经常把函数式编程称为Lambda计算。

Java平台从Java 8开始，支持函数式编程。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





Lambda基础

在了解Lambda之前，我们先回顾一下Java的方法。

Java的方法分为实例方法，例如 `Integer` 定义的 `equals()` 方法：

```
public final class Integer {
    boolean equals(Object o) {
        ...
    }
}
```

以及静态方法，例如 `Integer` 定义的 `parseInt()` 方法：

```
public final class Integer {
    public static int parseInt(String s) {
        ...
    }
}
```

无论是实例方法，还是静态方法，本质上都相当于过程式语言的函数。例如C函数：

```
char* strcpy(char* dest, char* src)
```

只不过Java的实例方法隐含地传入了一个 `this` 变量，即实例方法总是有一个隐含参数 `this` 。

函数式编程（Functional Programming）是把函数作为基本运算单元，函数可以作为变量，可以接收函数，还可以返回函数。历史上研究函数式编程的理论是Lambda演算，所以我们经常把支持函数式编程的编码风格称为Lambda表达式。

Lambda表达式

在Java程序中，我们经常遇到一大堆单方法接口，即一个接口只定义了一个方法：

- Comparator
- Runnable
- Callable

以 `Comparator` 为例，我们想要调用 `Arrays.sort()` 时，可以传入一个 `Comparator` 实例，以匿名类方式编写如下：

```
String[] array = ...
Arrays.sort(array, new Comparator<String>() {
    public int compare(String s1, String s2) {
        return s1.compareTo(s2);
    }
});
```

上述写法非常繁琐。从Java 8开始，我们可以用Lambda表达式替换单方法接口。改写上述代码如下：

```
// Lambda
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };
        Arrays.sort(array, (s1, s2) -> {
            return s1.compareTo(s2);
        });
        System.out.println(String.join(", ", array));
    }
}
```

▶ Run

观察Lambda表达式的写法，它只需要写出方法定义：

```
(s1, s2) -> {
    return s1.compareTo(s2);
}
```

其中，参数是 `(s1, s2)` ，参数类型可以省略，因为编译器可以自动推断出 `String` 类型。 `-> { … }` 表示方法体，所有代码写在内部即可。Lambda表达式没有 `class` 定义，因此写法非常简洁。

如果只有一行 `return xxx` 的代码，完全可以用更简单的写法：

```
Arrays.sort(array, (s1, s2) -> s1.compareTo(s2));
```

返回值的类型也是由编译器自动推断的，这里推断出的返回值是 `int` ，因此，只要返回 `int` ，编译器就不会报错。

FunctionalInterface

我们把只定义了单方法的接口称之为 `FunctionalInterface` ，用注解 `@FunctionalInterface` 标记。例如， `Callable` 接口：

```
@FunctionalInterface
public interface Callable<V> {
    V call() throws Exception;
}
```

再来看 `Comparator` 接口：

```
@FunctionalInterface
public interface Comparator<T> {

    int compare(T o1, T o2);

    boolean equals(Object obj);

    default Comparator<T> reversed() {
        return Collections.reverseOrder(this);
    }

    default Comparator<T> thenComparing(Comparator<? super T> other) {
        ...
    }
    ...
}
```

虽然 `Comparator` 接口有很多方法，但只有一个抽象方法 `int compare(T o1, T o2)` ，其他的方法都是 `default` 方法或 `static` 方法。另外注意到 `boolean equals(Object obj)` 是 `Object` 定义的方法，不算在接口方法内。因此， `Comparator` 也是一个 `FunctionalInterface` 。

练习

从码云下载练习：使用Lambda表达式实现忽略大小写排序 （推荐使用IDE练习插件快速下载）

小结

单方法接口被称为 `FunctionalInterface` 。

接收 `FunctionalInterface` 作为参数的时候，可以把实例化的匿名类改写为Lambda表达式，能大大简化代码。

Lambda表达式的参数和返回值均可由编译器自动推断。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





方法引用

使用Lambda表达式，我们就可以不必编写 `FunctionalInterface` 接口的实现类，从而简化代码：

```
Arrays.sort(array, (s1, s2) -> {
    return s1.compareTo(s2);
});
```

实际上，除了Lambda表达式，我们还可以直接传入方法引用。例如：

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };
        Arrays.sort(array, Main::cmp);
        System.out.println(String.join(", ", array));
    }

    static int cmp(String s1, String s2) {
        return s1.compareTo(s2);
    }
}
```

Run

上述代码在 `Arrays.sort()` 中直接传入了静态方法 `cmp` 的引用，用 `Main::cmp` 表示。

因此，所谓方法引用，是指如果某个方法签名和接口恰好一致，就可以直接传入方法引用。

因为 `Comparator<String>` 接口定义的方法是 `int compare(String, String)` ，和静态方法 `int cmp(String, String)` 相比，除了方法名外，方法参数一致，返回类型相同，因此，我们说两者的方法签名一致，可以直接把方法名作为Lambda表达式传入：

```
Arrays.sort(array, Main::cmp);
```

注意：在这里，方法签名只看参数类型和返回类型，不看方法名称，也不看类的继承关系。

我们再看看如何引用实例方法。如果我们把代码改写如下：

```
import java.util.Arrays;
public class Main {
    public static void main(String[] args) {
        String[] array = new String[] { "Apple", "Orange", "Banana", "Lemon" };
        Arrays.sort(array, String::compareTo);
        System.out.println(String.join(", ", array));
    }
}
```

Run

不但可以编译通过，而且运行结果也是一样的，这说明 `String.compareTo()` 方法也符合Lambda定义。

观察 `String.compareTo()` 的方法定义：

```
public final class String {
    public int compareTo(String o) {
        ...
    }
}
```

这个方法的签名只有一个参数，为什么和 `int Comparator<String>.compare(String, String)` 能匹配呢？

因为实例方法有一个隐含的 `this` 参数， `String` 类的 `compareTo()` 方法在实际调用的时候，第一个隐含参数总是传入 `this` ，相当于静态方法：

```
public static int compareTo(this, String o);
```

所以， `String.compareTo()` 方法也可作为方法引用传入。

构造方法引用

除了可以引用静态方法和实例方法，我们还可以引用构造方法。

我们来看一个例子：如果要把一个 `List<String>` 转换为 `List<Person>` ，应该怎么办？

```
class Person {
    String name;
    public Person(String name) {
        this.name = name;
```

```
    }
}

List<String> names = List.of("Bob", "Alice", "Tim");
List<Person> persons = ???
```

传统的做法是先定义一个 `ArrayList<Person>` ，然后用 `for` 循环填充这个 `List` ：

```
List<String> names = List.of("Bob", "Alice", "Tim");
List<Person> persons = new ArrayList<>();
for (String name : names) {
    persons.add(new Person(name));
}
```

要更简单地实现 `String` 到 `Person` 的转换，我们可以引用 `Person` 的构造方法：

```
// 引用构造方法
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        List<String> names = List.of("Bob", "Alice", "Tim");
        List<Person> persons = names.stream().map(Person::new).collect(Collectors.toList());
        System.out.println(persons);
    }
}

class Person {
    String name;
    public Person(String name) {
        this.name = name;
    }
    public String toString() {
        return "Person:" + this.name;
    }
}
```

Run

后面我们会讲到 `Stream` 的 `map()` 方法。现在我们看到，这里的 `map()` 需要传入的FunctionalInterface的定义是：

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);
}
```

把泛型对应上就是方法签名 `Person apply(String)` ，即传入参数 `String` ，返回类型 `Person` 。而 `Person` 类的构造方法恰好满足这个条件，因为构造方法的参数是 `String` ，而构造方法虽然没有 `return` 语句，但它会隐式地返回 `this` 实例，类型就是 `Person` ，因此，此处可以引用构造方法。构造方法的引用写法是 `类名::new` ，因此，此处传入 `Person::new` 。

练习

从  下载练习：使用方法引用实现忽略大小写排序 （推荐使用IDE练习插件快速下载）

小结

`FunctionalInterface` 允许传入：

- 接口的实现类（传统写法，代码较繁琐）；
- Lambda表达式（只需列出参数名，由编译器推断类型）；
- 符合方法签名的静态方法；
- 符合方法签名的实例方法（实例类型被看做第一个参数类型）；
- 符合方法签名的构造方法（实例类型被看做返回类型）。

`FunctionalInterface` 不强制继承关系，不需要方法名称相同，只要求方法参数（类型和数量）与方法返回类型相同，即认为方法签名相同。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





使用Stream

Java从8开始，不但引入了Lambda表达式，还引入了一个全新的流式API：Stream API。它位于 `java.util.stream` 包中。

划重点：这个 `Stream` 不同于 `java.io` 的 `InputStream` 和 `OutputStream` ，它代表的是任意Java对象的序列。两者对比如下：

	java.io	java.util.stream
存储	顺序读写的 `byte` 或 `char`	顺序输出的任意Java对象实例
用途	序列化至文件或网络	内存计算 / 业务逻辑

有同学会问：一个顺序输出的Java对象序列，不就是一个 `List` 容器吗？

再次划重点：这个 `Stream` 和 `List` 也不一样， `List` 存储的每个元素都是已经存储在内存中的某个Java对象，而 `Stream` 输出的元素可能并没有预先存储在内存中，而是实时计算出来的。

换句话说， `List` 的用途是操作一组已存在的Java对象，而 `Stream` 实现的是惰性计算，两者对比如下：

	java.util.List	java.util.stream
元素	已分配并存储在内存	可能未分配，实时计算
用途	操作一组已存在的Java对象	惰性计算

`Stream` 看上去有点不好理解，但我们举个例子就明白了。

如果我们要表示一个全体自然数的集合，显然，用 `List` 是不可能写出来的，因为自然数是无限的，内存再大也没法放到 `List` 中：

```
1.  List<BigInteger> list = ??? // 全体自然数?
```

但是，用 `Stream` 可以做到。写法如下：

```
1.  Stream<BigInteger> naturals = createNaturalStream(); // 全体自然数
```

我们先不考虑 `createNaturalStream()` 这个方法是如何实现的，我们看看如何使用这个 `Stream` 。

首先，我们可以对每个自然数做一个平方，这样我们就把这个 `Stream` 转换成了另一个 `Stream` ：

```
1.  Stream<BigInteger> naturals = createNaturalStream(); // 全体自然数
```

```
    Stream<BigInteger> streamNxN = naturals.map(n -> n.multiply(n)); // 全体自然数的
2.  平方
```

因为这个 `streamNxN` 也有无限多个元素，要打印它，必须首先把无限多个元素变成有限个元素，可以用 `limit()` 方法截取前100个元素，最后用 `forEach()` 处理每个元素，这样，我们就打印出了前100个自然数的平方：

```
1.  Stream<BigInteger> naturals = createNaturalStream();
2.  naturals.map(n -> n.multiply(n)) // 1, 4, 9, 16, 25...
3.          .limit(100)
4.          .forEach(System.out::println);
```

我们总结一下 `Stream` 的特点：它可以“存储”有限个或无限个元素。这里的存储打了个引号，是因为元素有可能已经全部存储在内存中，也有可能是根据需要实时计算出来的。

`Stream` 的另一个特点是，一个 `Stream` 可以轻易地转换为另一个 `Stream` ，而不是修改原 `Stream` 本身。

最后，真正的计算通常发生在最后结果的获取，也就是惰性计算。

```
1.  Stream<BigInteger> naturals = createNaturalStream(); // 不计算
2.  Stream<BigInteger> s2 = naturals.map(BigInteger::multiply); // 不计算
3.  Stream<BigInteger> s3 = s2.limit(100); // 不计算
4.  s3.forEach(System.out::println); // 计算
```

惰性计算的特点是：一个 `Stream` 转换为另一个 `Stream` 时，实际上只存储了转换规则，并没有任何计算发生。

例如，创建一个全体自然数的 `Stream` ，不会进行计算，把它转换为上述 `s2` 这个 `Stream` ，也不会进行计算。再把 `s2` 这个无限 `Stream` 转换为 `s3` 这个有限的 `Stream` ，也不会进行计算。只有最后，调用 `forEach` 确实需要 `Stream` 输出的元素时，才进行计算。我们通常把 `Stream` 的操作写成链式操作，代码更简洁：

```
1.  createNaturalStream()
2.      .map(BigInteger::multiply)
3.      .limit(100)
4.      .forEach(System.out::println);
```

因此，Stream API的基本用法就是：创建一个 `Stream` ，然后做若干次转换，最后调用一个求值方法获取真正计算的结果：

```
1.  int result = createNaturalStream() // 创建Stream
```

```
2.              .filter(n -> n % 2 == 0) // 任意个转换
3.              .map(n -> n * n) // 任意个转换
4.              .limit(100) // 任意个转换
5.              .sum(); // 最终计算结果
```

小结

Stream API的特点是:

- Stream API提供了一套新的流式处理的抽象序列;
- Stream API支持函数式编程和链式操作;
- Stream可以表示无限序列，并且大多数情况下是惰性求值的。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论:





创建Stream

要使用 `Stream` ，就必须现创建它。创建 `Stream` 有很多种方法，我们来一一介绍。

Stream.of()

创建 `Stream` 最简单的方式是直接用 `Stream.of()` 静态方法，传入可变参数即创建了一个能输出确定元素的 `Stream` ：

```
import java.util.stream.Stream;
public class Main {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("A", "B", "C", "D");
        // forEach()方法相当于内部循环调用，
        // 可传入符合Consumer接口的void accept(T t)的方法引用：
        stream.forEach(System.out::println);
    }
}
```

▶ Run

虽然这种方式基本上没啥实质性用途，但测试的时候很方便。

基于数组或Collection

第二种创建 `Stream` 的方法是基于一个数组或者 `Collection` ，这样该 `Stream` 输出的元素就是数组或者 `Collection` 持有的元素：

```
import java.util.*;
import java.util.stream.*;
public class Main {
    public static void main(String[] args) {
        Stream<String> stream1 = Arrays.stream(new String[] { "A", "B", "C" });
        Stream<String> stream2 = List.of("X", "Y", "Z").stream();
        stream1.forEach(System.out::println);
        stream2.forEach(System.out::println);
    }
}
```

▶ Run

把数组变成 `Stream` 使用 `Arrays.strem()` 方法。对于 `Collection` （ `List` 、 `Set` 、 `Queue` 等），直接调用 `stream()` 方法就可以获

得 `Stream` 。

上述创建 `Stream` 的方法都是把一个现有的序列变为 `Stream` ，它的元素是固定的。

基于Supplier

创建 `Stream` 还可以通过 `Stream.generate()` 方法，它需要传入一个 `Supplier` 对象：

```
1.  Stream<String> s = Stream.generate(Supplier<String> sp);
```

基于 `Supplier` 创建的 `Stream` 会不断调用 `Supplier.get()` 方法来不断产生下一个元素，这种 `Stream` 保存的不是元素，而是算法，它可以用来表示无限序列。

例如，我们编写一个能不断生成自然数的 `Supplier` ，它的代码非常简单，每次调用 `get()` 方法，就生成下一个自然数：

```
import java.util.function.*;
import java.util.stream.*;

public class Main {
        public static void main(String[] args) {
                Stream<Integer> natual = Stream.generate(new NatualSupplier());
                // 注意：无限序列必须先变成有限序列再打印:
                natual.limit(20).forEach(System.out::println);
        }
}

class NatualSupplier implements Supplier<Integer> {
        int n = 0;
        public Integer get() {
                n++;
                return n;
        }
}
```

▶ Run

上述代码我们用一个 `Supplier<Integer>` 模拟了一个无限序列（当然受 `int` 范围限制不是真的无限大）。如果用 `List` 表示，即便在 `int` 范围内，也会占用巨大的内存，而 `Stream` 几乎不占用空间，因为每个元素都是实时计算出来的，用的时候再算。

对于无限序列，如果直接调用 `forEach()` 或者 `count()` 这些最终求值操作，会进入死循环，因为永远无法计算完这个序列，所以正确的方法是先把无限序列变成有限序列，例如，用 `limit()` 方法可以截取前面若干个元素，这样就变成了一个有限序列，对这个有限序列调用 `forEach()` 或者 `count()` 操作就没有问题。

其他方法

创建 `Stream` 的第三种方法是通过一些API提供的接口，直接获得 `Stream` 。

例如， `Files` 类的 `lines()` 方法可以把一个文件变成一个 `Stream` ，每个元素代表文件的一行内容：

```
try (Stream<String> lines = Files.lines(Paths.get("/path/to/file.txt"))) {
    ...
}
```

此方法对于按行遍历文本文件十分有用。

另外，正则表达式的 `Pattern` 对象有一个 `splitAsStream()` 方法，可以直接把一个长字符串分割成 `Stream` 序列而不是数组：

```
Pattern p = Pattern.compile("\\s+");
Stream<String> s = p.splitAsStream("The quick brown fox jumps over the lazy dog");
s.forEach(System.out::println);
```

基本类型

因为Java的范型不支持基本类型，所以我们无法用 `Stream<int>` 这样的类型，会发生编译错误。为了保存 `int` ，只能使用 `String<Integer>` ，但这样会产生频繁的装箱、拆箱操作。为了提高效率，Java标准库提供了 `IntStream` 、 `LongStream` 和 `DoubleStream` 这三种使用基本类型的 `Stream` ，它们的使用方法和范型 `Stream` 没有大的区别，设计这三个 `Stream` 的目的是提高运行效率：

```
// 将int[]数组变为IntStream:
IntStream is = Arrays.stream(new int[] { 1, 2, 3 });
// 将Stream<String>转换为LongStream:
LongStream ls = List.of("1", "2", "3").stream().mapToLong(Long::parseLong);
```

练习

编写一个能输出斐波拉契数列（Fibonacci）的 `LongStream` ：

```
1, 1, 2, 3, 5, 8, 13, 21, 34, ...
```

从码云下载练习：FibonacciStream练习 （推荐使用IDE练习插件快速下载）

小结

创建 `Stream` 的方法有 ：

- 通过指定元素、指定数组、指定 `Collection` 创建 `Stream` ；
- 通过 `Supplier` 创建 `Stream` ，可以是无限序列；
- 通过其他类的相关方法创建。

基本类型的 `Stream` 有 `IntStream` 、 `LongStream` 和 `DoubleStream` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





使用map

`Stream.map()` 是 `Stream` 最常用的一个转换方法，它把一个 `Stream` 转换为另一个 `Stream` 。

所谓 `map` 操作，就是把一种操作运算，映射到一个序列的每一个元素上。例如，对 `x` 计算它的平方，可以使用函数 `f(x) = x * x` 。我们把这个函数映射到一个序列1，2，3，4，5上，就得到了另一个序列1，4，9，16，25：

```
            f(x) = x * x

                  |
                  |
   ┌──┬──┬──┬──┬──┬──┬──┬──┬──┐
   │  │  │  │  │  │  │  │  │  │
   ▼  ▼  ▼  ▼  ▼  ▼  ▼  ▼  ▼

[ 1  2  3  4  5  6  7  8  9 ]

   │  │  │  │  │  │  │  │  │
   │  │  │  │  │  │  │  │  │
   ▼  ▼  ▼  ▼  ▼  ▼  ▼  ▼  ▼

[ 1  4  9 16 25 36 49 64 81 ]
```

可见， `map` 操作，把一个 `Stream` 的每个元素一一对应到应用了目标函数的结果上。

```
Stream<Integer> s = Stream.of(1, 2, 3, 4, 5);
Stream<Integer> s2 = s.map(n -> n * n);
```

如果我们查看 `Stream` 的源码，会发现 `map()` 方法接收的对象是 `Function` 接口对象，它定义了一个 `apply()` 方法，负责把一个 `T` 类型转换成 `R` 类型：

```
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
```

其中， `Function` 的定义是：

```
@FunctionalInterface
public interface Function<T, R> {
    // 将T类型转换为R:
```

```
    R apply(T t);
}
```

利用 `map()` ，不但能完成数学计算，对于字符串操作，以及任何Java对象都是非常有用的。例如：

```
import java.util.*;
import java.util.stream.*;
public class Main {
    public static void main(String[] args) {
        List.of("  Apple ", " pear ", " ORANGE", " BaNaNa ")
                .stream()
                .map(String::trim) // 去空格
                .map(String::toLowerCase) // 变小写
                .forEach(System.out::println); // 打印
    }
}
```

▶ Run

通过若干步 `map` 转换，可以写出逻辑简单、清晰的代码。

练习

使用 `map()` 把一组 `String` 转换为 `LocalDate` 并打印。

从 码云 下载练习：map练习 （推荐使用IDE练习插件快速下载）

小结

`map()` 方法用于将一个 `Stream` 的每个元素映射成另一个元素并转换成一个新的 `Stream` ；

可以将一种元素类型转换成另一种元素类型。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





使用filter

`Stream.filter()` 是 `Stream` 的另一个常用转换方法。

所谓 `filter()` 操作，就是对一个 `Stream` 的所有元素一一进行测试，不满足条件的就被“滤掉”了，剩下的满足条件的元素就构成了一个新的 `Stream` 。

例如，我们对1，2，3，4，5这个 `Stream` 调用 `filter()` ，传入的测试函数 `f(x) = x % 2 != 0` 用来判断元素是否是奇数，这样就过滤掉偶数，只剩下奇数，因此我们得到了另一个序列1，3，5：

```
             f(x) = x % 2 != 0

                   |
                   |
    ┌──┬──┬──┬──┬──┬──┬──┬──┬──┐
    │  │  │  │  │  │  │  │  │  │
    ▼  ▼  ▼  ▼  ▼  ▼  ▼  ▼  ▼

[ 1  2  3  4  5  6  7  8  9 ]

    │  X  │  X  │  X  │  X  │
    │     │     │     │     │
    ▼     ▼     ▼     ▼     ▼

[ 1     3     5     7     9 ]
```

用IntStream写出上述逻辑，代码如下：

```
import java.util.stream.IntStream;
public class Main {
    public static void main(String[] args) {
        IntStream.of(1, 2, 3, 4, 5, 6, 7, 8, 9)
                .filter(n -> n % 2 != 0)
                .forEach(System.out::println);
    }
}
```

▶ Run

从结果可知，经过 `filter()` 后生成的 `Stream` 元素可能变少。

`filter()` 方法接收的对象是 `Predicate` 接口对象，它定义了一个 `test()` 方法，负责判断元素

是否符合条件：

```
@FunctionalInterface
public interface Predicate<T> {
    // 判断元素t是否符合条件：
    boolean test(T t);
}
```

`filter()` 除了常用于数值外，也可应用于任何Java对象。例如，从一组给定的 `LocalDate` 中过滤掉工作日，以便得到休息日：

```
import java.time.*;
import java.util.function.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Stream.generate(new LocalDateSupplier())
                .limit(31)
                .filter(ldt -> ldt.getDayOfWeek() == DayOfWeek.SATURDAY || ldt.getDayOfWeek() ==
DayOfWeek.SUNDAY)
                .forEach(System.out::println);
    }
}

class LocalDateSupplier implements Supplier<LocalDate> {
    LocalDate start = LocalDate.of(2020, 1, 1);
    int n = -1;
    public LocalDate get() {
        n++;
        return start.plusDays(n);
    }
}
```

▶ Run

练习

请使用 `filter()` 过滤出成绩及格的同学，并打印出名字。

从  下载练习：filter练习 （推荐使用IDE练习插件快速下载）

小结

使用 `filter()` 方法可以对一个 `Stream` 的每个元素进行测试，通过测试的元素被过滤后生成一个新的 `Stream` 。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





使用reduce

`map()` 和 `filter()` 都是 `Stream` 的转换方法，而 `Stream.reduce()` 则是 `Stream` 的一个聚合方法，它可以把一个 `Stream` 的所有元素按照聚合函数聚合成一个结果。

我们来看一个简单的聚合方法：

```
import java.util.stream.*;
public class Main {
    public static void main(String[] args) {
        int sum = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).reduce(0, (acc, n) -> acc + n);
        System.out.println(sum); // 45
    }
}
```

▶ Run

`reduce()` 方法传入的对象是 `BinaryOperator` 接口，它定义了一个 `apply()` 方法，负责把上次累加的结果和本次的元素进行运算，并返回累加的结果：

```
@FunctionalInterface
public interface BinaryOperator<T> {
    // Bi操作：两个输入，一个输出
    T apply(T t, T u);
}
```

上述代码看上去不好理解，但我们用 `for` 循环改写一下，就容易理解了：

```
Stream<Integer> stream = ...
int sum = 0;
for (n : stream) {
    sum = (sum, n) -> sum + n;
}
```

可见， `reduce()` 操作首先初始化结果为指定值（这里是0），紧接着， `reduce()` 对每个元素依次调用 `(acc, n) -> acc + n` ，其中， `acc` 是上次计算的结果：

```
// 计算过程:
acc = 0 // 初始化为指定值
acc = acc + n = 0 + 1 = 1 // n = 1
```

```
acc = acc + n = 1 + 2 = 3 // n = 2
acc = acc + n = 3 + 3 = 6 // n = 3
acc = acc + n = 6 + 4 = 10 // n = 4
acc = acc + n = 10 + 5 = 15 // n = 5
acc = acc + n = 15 + 6 = 21 // n = 6
acc = acc + n = 21 + 7 = 28 // n = 7
acc = acc + n = 28 + 8 = 36 // n = 8
acc = acc + n = 36 + 9 = 45 // n = 9
```

因此，实际上这个 `reduce()` 操作是一个求和。

如果去掉初始值，我们会得到一个 `Optional<Integer>` ：

```
Optional<Integer> opt = stream.reduce((acc, n) -> acc + n);
if (opt.isPresent) {
    System.out.println(opt.get());
}
```

这是因为 `Stream` 的元素有可能是0个，这样就没法调用 `reduce()` 的聚合函数了，因此返回 `Optional` 对象，需要进一步判断结果是否存在。

利用reduce()，我们可以把求和改成求积，代码也十分简单：

```
import java.util.stream.*;
public class Main {
    public static void main(String[] args) {
        int s = Stream.of(1, 2, 3, 4, 5, 6, 7, 8, 9).reduce(1, (acc, n) -> acc * n);
        System.out.println(s); // 362880
    }
}
```

▶ Run

注意：计算求积时，初始值必须设置为 `1` 。

除了可以对数值进行累积计算外，灵活运用 `reduce()` 也可以对Java对象进行操作。下面的代码演示了如何将配置文件的每一行配置通过 `map()` 和 `reduce()` 操作聚合成一个 `Map<String, String>` ：

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        // 按行读取配置文件:
        List<String> props = List.of("profile=native", "debug=true", "logging=warn", "interval=500");
        Map<String, String> map = props.stream()
                // 把k=v转换为Map[k]=v:
                .map(kv -> {
                    String[] ss = kv.split("\\=", 2);
                    return Map.of(ss[0], ss[1]);
                })
                // 把所有Map聚合到一个Map:
                .reduce(new HashMap<String, String>(), (m, kv) -> {
                    m.putAll(kv);
                    return m;
                });
        // 打印结果:
        map.forEach((k, v) -> {
            System.out.println(k + " = " + v);
        });
    }
}
```

▶ Run

小结

`reduce()` 方法将一个 `Stream` 的每个元素依次作用于 `BinaryOperator` ，并将结果合并。

`reduce()` 是聚合方法，聚合方法会立刻对 `Stream` 进行计算。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





输出集合

我们介绍了 `Stream` 的几个常见操作： `map()` 、 `filter()` 、 `reduce()` 。这些操作对 `Stream` 来说可以分为两类，一类是转换操作，即把一个 `Stream` 转换为另一个 `Stream` ，例如 `map()` 和 `filter()` ，另一类是聚合操作，即对 `Stream` 的每个元素进行计算，得到一个确定的结果，例如 `reduce()` 。

区分这两种操作是非常重要的，因为对于 `Stream` 来说，对其进行转换操作并不会触发任何计算！我们可以做个实验：

```
import java.util.function.Supplier;
import java.util.stream.Stream;

public class Main {
    public static void main(String[] args)     {
        Stream<Long> s1 = Stream.generate(new NatualSupplier());
        Stream<Long> s2 = s1.map(n -> n * n);
        Stream<Long> s3 = s2.map(n -> n - 1);
        System.out.println(s3); // java.util.stream.ReferencePipeline$3@49476842
    }
}

class NatualSupplier implements Supplier<Long> {
    long n = 0;
    public Long get() {
        n++;
        return n;
    }
}
```

▶ Run

因为 `s1` 是一个 `Long` 类型的序列，它的元素高达922亿个，但执行上述代码，既不会有任何内存增长，也不会有任何计算，因为转换操作只是保存了转换规则，无论我们对一个 `Stream` 转换多少次，都不会有任何实际计算发生。

而聚合操作则不一样，聚合操作会立刻促使 `Stream` 输出它的每一个元素，并依次纳入计算，以获得最终结果。所以，对一个 `Stream` 进行聚合操作，会触发一系列连锁反应：

```
1. Stream<Long> s1 = Stream.generate(new NatualSupplier());
2. Stream<Long> s2 = s1.map(n -> n * n);
3. Stream<Long> s3 = s2.map(n -> n - 1);
4. Stream<Long> s4 = s3.limit(10);
5. s4.reduce(0, (acc, n) -> acc + n);
```

我们对 `s4` 进行 `reduce()` 聚合计算，会不断请求 `s4` 输出它的每一个元素。因为 `s4` 的上游

是 `s3` ，它又会向 `s3` 请求元素，导致 `s3` 向 `s2` 请求元素， `s2` 向 `s1` 请求元素，最终， `s1` 从 `Supplier` 实例中请求到真正的元素，并经过一系列转换，最终被 `reduce()` 聚合出结果。

可见，聚合操作是真正需要从 `Stream` 请求数据的，对一个 `Stream` 做聚合计算后，结果就不是一个 `Stream` ，而是一个其他的Java对象。

输出为List

`reduce()` 只是一种聚合操作，如果我们希望把 `Stream` 的元素保存到集合，例如 `List` ，因为 `List` 的元素是确定的Java对象，因此，把 `Stream` 变为 `List` 不是一个转换操作，而是一个聚合操作，它会强制 `Stream` 输出每个元素。

下面的代码演示了如何将一组 `String` 先过滤到空字符串，然后把非空字符串保存到 `List` 中：

```
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("Apple", "", null, "Pear", "  ", "Orange");
        List<String> list = stream.filter(s -> s != null &&
!s.isBlank()).collect(Collectors.toList());
        System.out.println(list);
    }
}
```

▶ Run

把 `Stream` 的每个元素收集到 `List` 的方法是调用 `collect()` 并传入 `Collectors.toList()` 对象，它实际上是一个 `Collector` 实例，通过类似 `reduce()` 的操作，把每个元素添加到一个收集器中（实际上是 `ArrayList` ）。

类似的， `collect(Collectors.toSet())` 可以把 `Stream` 的每个元素收集到 `Set` 中。

输出为数组

把Stream的元素输出为数组和输出为List类似，我们只需要调用 `toArray()` 方法，并传入数组的“构造方法”：

```
1. List<String> list = List.of("Apple", "Banana", "Orange");
2. String[] array = list.stream().toArray(String[]::new);
```

注意到传入的“构造方法”是 `String[]::new` ，它的签名实际上是 `IntFunction<String[]>` 定义的 `String[] apply(int)` ，即传入 `int` 参数，获得 `String[]` 数组的返回值。

输出为Map

如果我们要把Stream的元素收集到Map中，就稍微麻烦一点。因为对于每个元素，添加到Map时需要key和value，因此，我们要指定两个映射函数，分别把元素映射为key和value：

```
import java.util.*;
import java.util.stream.*;
public class Main {
    public static void main(String[] args) {
        Stream<String> stream = Stream.of("APPL:Apple", "MSFT:Microsoft");
        Map<String, String> map = stream
                .collect(Collectors.toMap(
                        // 把元素s映射为key:
                        s -> s.substring(0, s.indexOf(':')),
                        // 把元素s映射为value:
                        s -> s.substring(s.indexOf(':') + 1)));
        System.out.println(map);
    }
}
```

▶ Run

分组输出

`Stream` 还有一个强大的分组功能，可以按组输出。我们看下面的例子：

```
import java.util.*;
import java.util.stream.*;
public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("Apple", "Banana", "Blackberry", "Coconut", "Avocado", "Cherry",
"Apricots");
        Map<String, List<String>> groups = list.stream()
                .collect(Collectors.groupingBy(s -> s.substring(0, 1), Collectors.toList()));
        System.out.println(groups);
    }
}
```

▶ Run

分组输出使用 `Collectors.groupingBy()` ，它需要提供两个函数：一个是分组的key，这里使用 `s -> s.substring(0, 1)` ，表示只要首字母相同的 `String` 分到一组，第二个是分组的value，这里直接使用 `Collectors.toList()` ，表示输出为 `List` ，上述代码运行结果如下：

```
{
    A=[Apple, Avocado, Apricots],
    B=[Banana, Blackberry],
```

```
    C=[Coconut, Cherry]
}
```

可见，结果一共有3组，按 `"A"` ， `"B"` ， `"C"` 分组，每一组都是一个 `List` 。

假设有这样一个 `Student` 类，包含学生姓名、班级和成绩：

```
class Student {
    int gradeId; // 年级
    int classId; // 班级
    String name; // 名字
    int score; // 分数
}
```

如果我们有一个 `Stream<Student>` ，利用分组输出，可以非常简单地按年级或班级把 `Student` 归类。

小结

`Stream` 可以输出为集合：

`Stream` 通过 `collect()` 方法可以方便地输出为 `List` 、 `Set` 、 `Map` ，还可以分组输出。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





其他操作

我们把 `Stream` 提供的操作分为两类：转换操作和聚合操作。除了前面介绍的常用操作外， `Stream` 还提供了一系列非常有用的方法。

排序

对 `Stream` 的元素进行排序十分简单，只需调用 `sorted()` 方法：

```
import java.util.*;
import java.util.stream.*;

public class Main {
    public static void main(String[] args) {
        List<String> list = List.of("Orange", "apple", "Banana")
            .stream()
            .sorted()
            .collect(Collectors.toList());
        System.out.println(list);
    }
}
```

▶ Run

此方法要求 `Stream` 的每个元素必须实现 `Comparable` 接口。如果要自定义排序，传入指定的 `Comparator` 即可：

```
List<String> list = List.of("Orange", "apple", "Banana")
    .stream()
    .sorted(String::compareToIgnoreCase)
    .collect(Collectors.toList());
```

注意 `sorted()` 只是一个转换操作，它会返回一个新的 `Stream` 。

去重

对一个 `Stream` 的元素进行去重，没必要先转换为 `Set` ，可以直接用 `distinct()` ：

```
List.of("A", "B", "A", "C", "B", "D")
    .stream()
    .distinct()
    .collect(Collectors.toList()); // [A, B, C, D]
```

截取

截取操作常用于把一个无限的 `Stream` 转换成有限的 `Stream` ， `skip()` 用于跳过当前 `Stream` 的前N个元素， `limit()` 用于截取当前 `Stream` 最多前N个元素：

```
List.of("A", "B", "C", "D", "E", "F")
    .stream()
    .skip(2) // 跳过A, B
    .limit(3) // 截取C, D, E
    .collect(Collectors.toList()); // [C, D, E]
```

截取操作也是一个转换操作，将返回新的 `Stream` 。

合并

将两个 `Stream` 合并为一个 `Stream` 可以使用 `Stream` 的静态方法 `concat()` ：

```
Stream<String> s1 = List.of("A", "B", "C").stream();
Stream<String> s2 = List.of("D", "E").stream();
// 合并:
Stream<String> s = Stream.concat(s1, s2);
System.out.println(s.collect(Collectors.toList())); // [A, B, C, D, E]
```

flatMap

如果 `Stream` 的元素是集合：

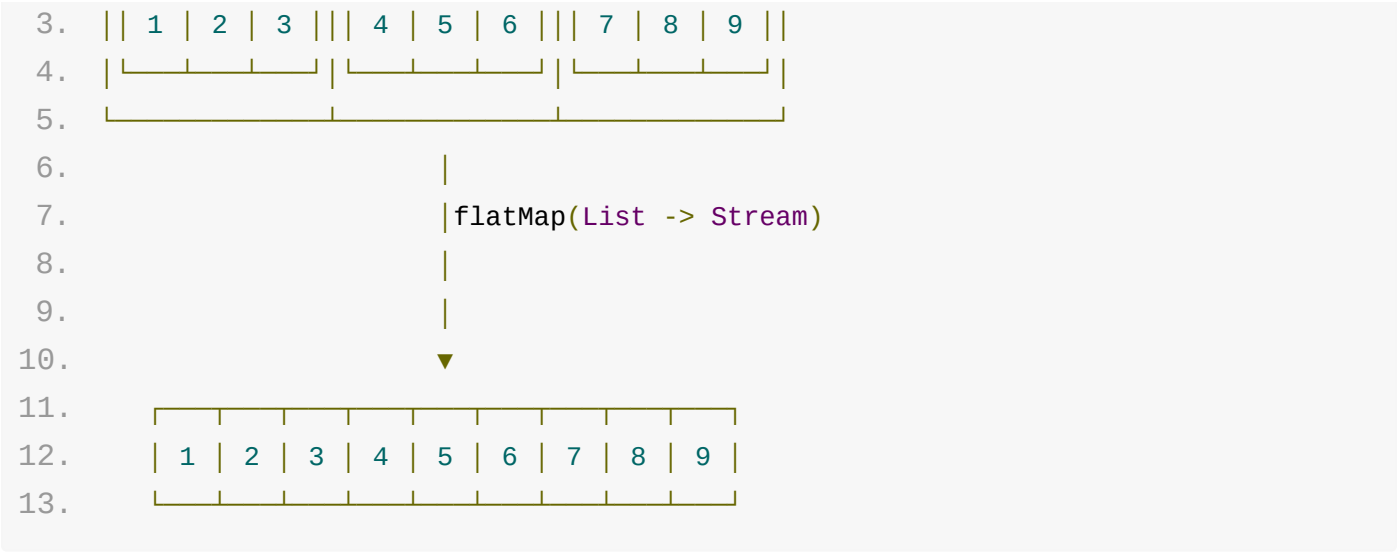
```
Stream<List<Integer>> s = Stream.of(
        Arrays.asList(1, 2, 3),
        Arrays.asList(4, 5, 6),
        Arrays.asList(7, 8, 9));
```

而我们希望把上述 `Stream` 转换为 `Stream<Integer>` ，就可以使用 `flatMap()` ：

```
Stream<Integer> i = s.flatMap(list -> list.stream());
```

因此，所谓 `flatMap()` ，是指把 `Stream` 的每个元素（这里是 `List` ）映射为 `Stream` ，然后合并成一个新的 `Stream` ：

```
┌─────────────┬─────────────┬─────────────┐
│┌─┬─┬─┐    │┌─┬─┬─┐    │┌─┬─┬─┐    │
```



并行

通常情况下，对 `Stream` 的元素进行处理是单线程的，即一个一个元素进行处理。但是很多时候，我们希望可以并行处理 `Stream` 的元素，因为在元素数量非常大的情况，并行处理可以大大加快处理速度。

把一个普通 `Stream` 转换为可以并行处理的 `Stream` 非常简单，只需要用 `parallel()` 进行转换：

```
Stream<String> s = ...
String[] result = s.parallel() // 变成一个可以并行处理的Stream
                   .sorted() // 可以进行并行排序
                   .toArray(String[]::new);
```

经过 `parallel()` 转换后的 `Stream` 只要可能，就会对后续操作进行并行处理。我们不需要编写任何多线程代码就可以享受到并行处理带来的执行效率的提升。

其他聚合方法

除了 `reduce()` 和 `collect()` 外， `Stream` 还有一些常用的聚合方法：

- `count()` ：用于返回元素个数；
- `max(Comparator<? super T> cp)` ：找出最大元素；
- `min(Comparator<? super T> cp)` ：找出最小元素。

针对 `IntStream` 、 `LongStream` 和 `DoubleStream` ，还额外提供了以下聚合方法：

- `sum()` ：对所有元素求和；
- `average()` ：对所有元素求平均数。

还有一些方法，用来测试 `Stream` 的元素是否满足以下条件：

- `boolean allMatch(Predicate<? super T>)`：测试是否所有元素均满足测试条件；
- `boolean anyMatch(Predicate<? super T>)`：测试是否至少有一个元素满足测试条件。

最后一个常用的方法是 `forEach()`，它可以循环处理 `Stream` 的每个元素，我们经常传入 `System.out::println` 来打印 `Stream` 的元素：

```
Stream<String> s = ...
s.forEach(str -> {
    System.out.println("Hello, " + str);
});
```

小结

`Stream` 提供的常用操作有：

转换操作：`map()`，`filter()`，`sorted()`，`distinct()`；

合并操作：`concat()`，`flatMap()`；

并行处理：`parallel()`；

聚合操作：`reduce()`，`collect()`，`count()`，`max()`，`min()`，`sum()`，`average()`；

其他操作：`allMatch()`，`anyMatch()`，`forEach()`。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





设计模式

设计模式，即Design Patterns，是指在软件设计中，被反复使用的一种代码设计经验。使用设计模式的目的是为了可重用代码，提高代码的可扩展性和可维护性。

设计模式这个术语是上个世纪90年代由Erich Gamma、Richard Helm、Raplh Johnson和Jonhn Vlissides四个人总结提炼出来的，并且写了一本Design Patterns的书。这四人也被称为四人帮（GoF）。

为什么要使用设计模式？根本原因还是软件开发要实现可维护、可扩展，就必须尽量复用代码，并且降低代码的耦合度。设计模式主要是基于OOP编程提炼的，它基于以下几个原则：

开闭原则

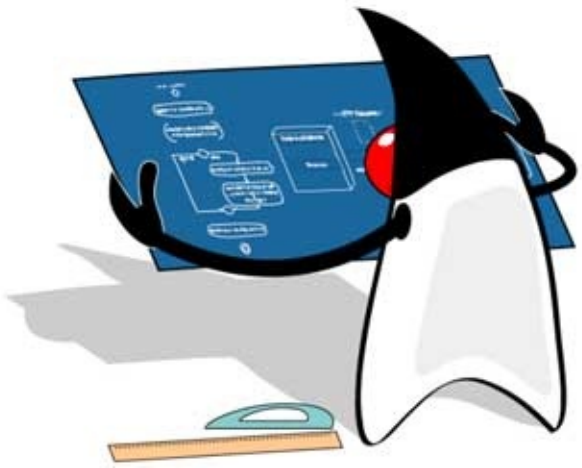
由Bertrand Meyer提出的开闭原则（Open Closed Principal）是指，软件应该对扩展开放，而对修改关闭。这里的意思是在增加新功能的时候，能不改代码就尽量不要改，如果只增加代码就完成了新功能，那是最好的。

里氏替换原则

里氏替换原则是Barbara Liskov提出的，这是一种面向对象的设计原则，即如果我们调用一个父类的方法可以成功，那么替换成子类调用也应该完全可以运行。

设计模式把一些常用的设计思想提炼出一个个模式，然后给每个模式命名，这样在使用的时候更方便交流。GoF把23个常用模式分为创建型模式、结构型模式和行为型模式三类，我们后续会一一讲解。

学习设计模式，关键是学习设计思想，不能简单地生搬硬套，也不能为了使用设计模式而过度设计，要合理平衡设计的复杂度和灵活性，并意识到设计模式也并不是万能的。



读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





创建型模式

创建型模式关注点是如何创建对象，其核心思想是要把对象的创建和使用相分离，这样使得两者能相对独立地变换。

创建型模式包括：

- 工厂方法：Factory Method
- 抽象工厂：Abstract Factory
- 建造者：Builder
- 原型：Prototype
- 单例：Singleton

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





工厂方法

> 定义一个用于创建对象的接口，让子类决定实例化哪一个类。Factory Method使一个类的实例化延迟到其子类。

工厂方法即Factory Method，是一种对象创建型模式。

工厂方法的目的是使得创建对象和使用对象是分离的，并且客户端总是引用抽象工厂和抽象产品：

```
┌──────────────┐      ┌──────────────┐
│   Product    │      │   Factory    │
└──────────────┘      └──────────────┘
       ▲                     ▲
       │                     │
┌──────────────┐      ┌──────────────┐
│ ProductImpl  │<- - -| FactoryImpl  |
└──────────────┘      └──────────────┘
```

我们以具体的例子来说：假设我们希望实现一个解析字符串到 `Number` 的 `Factory` ，可以定义如下：

```
public interface Factory {
    Number parse(String s);
}
```

有了工厂接口，再编写一个工厂的实现类：

```
public class NumberFactoryImpl implements NumberFactory {
    public Number parse(String s) {
        return new BigDecimal(s);
    }
}
```

而产品接口是 `Number` ， `NumberFactoryImpl` 返回的实际产品是 `BigDecimal` 。

那么客户端如何创建 `NumberFactoryImpl` 呢？通常我们会在接口 `Factory` 中定义一个静态方法 `getFactory()` 来返回真正的子类：

```
public interface NumberFactory {
    // 创建方法:
    Number parse(String s);
```

```
4.
5.      // 获取工厂实例:
6.      static NumberFactory getFactory() {
7.          return impl;
8.      }
9.
10.     static NumberFactory impl = new NumberFactoryImpl();
11. }
```

在客户端中，我们只需要和工厂接口 `NumberFactory` 以及抽象产品 `Number` 打交道：

```
1.  NumberFactory factory = NumberFactory.getFactory();
2.  Number result = factory.parse("123.456");
```

调用方可以完全忽略真正的工厂 `NumberFactoryImpl` 和实际的产品 `BigDecimal` ，这样做的好处是允许创建产品的代码独立地变换，而不会影响到调用方。

有的童鞋会问：一个简单的 `parse()` 需要写这么复杂的工厂吗？实际上大多数情况下我们并不需要抽象工厂，而是通过静态方法直接返回产品，即：

```
1.  public class NumberFactory {
2.      public static Number parse(String s) {
3.          return new BigDecimal(s);
4.      }
5.  }
```

这种简化的使用静态方法创建产品的方式称为静态工厂方法（Static Factory Method）。静态工厂方法广泛地应用在Java标准库中。例如：

```
1.  Integer n = Integer.valueOf(100);
```

`Integer` 既是产品又是静态工厂。它提供了静态方法 `valueOf()` 来创建 `Integer` 。那么这种方式和直接写 `new Integer(100)` 有何区别呢？我们观察 `valueOf()` 方法：

```
1.  public final class Integer {
2.      public static Integer valueOf(int i) {
3.          if (i >= IntegerCache.low && i <= IntegerCache.high)
4.              return IntegerCache.cache[i + (-IntegerCache.low)];
5.          return new Integer(i);
6.      }
7.      ...
```

```
}
```

它的好处在于，`valueOf()` 内部可能会使用 `new` 创建一个新的 `Integer` 实例，但也可能直接返回一个缓存的 `Integer` 实例。对于调用方来说，没必要知道 `Integer` 创建的细节。

工厂方法可以隐藏创建产品的细节，且不一定每次都会真正创建产品，完全可以返回缓存的产品，从而提升速度并减少内存消耗。

如果调用方直接使用 `Integer n = new Integer(100)`，那么就失去了使用缓存优化的可能性。

我们经常使用的另一个静态工厂方法是 `List.of()`：

```
List<String> list = List.of("A", "B", "C");
```

这个静态工厂方法接收可变参数，然后返回 `List` 接口。需要注意的是，调用方获取的产品总是 `List` 接口，而且并不关心它的实际类型。即使调用方知道 `List` 产品的实际类型是 `java.util.ImmutableCollections$ListN`，也不要去强制转型为子类，因为静态工厂方法 `List.of()` 保证返回 `List`，但也完全可以修改为返回 `java.util.ArrayList`。这就是里氏替换原则：返回实现接口的任意子类都可以满足该方法的要求，且不影响调用方。

总是引用接口而非实现类，能允许变换子类而不影响调用方，即尽可能面向抽象编程。

和 `List.of()` 类似，我们使用 `MessageDigest` 时，为了创建某个摘要算法，总是使用静态工厂方法 `getInstance(String)`：

```
MessageDigest md5 = MessageDigest.getInstance("MD5");
MessageDigest sha1 = MessageDigest.getInstance("SHA-1");
```

调用方通过产品名称获得产品实例，不但调用简单，而且获得的引用仍然是 `MessageDigest` 这个抽象类。

练习

使用静态工厂方法实现一个类似 `20200202` 的整数转换为 `LocalDate`：

```
public class LocalDateFactory {
    public static LocalDate fromInt(int yyyyMMdd) {
        ...
    }
}
```

从码云下载练习：静态工厂方法练习 （推荐使用IDE练习插件快速下载）

小结

工厂方法是指定义工厂接口和产品接口，但如何创建实际工厂和实际产品被推迟到子类实现，从而使调用方只和抽象工厂与抽象产品打交道。

实际更常用的是更简单的静态工厂方法，它允许工厂内部对创建产品进行优化。

调用方尽量持有接口或抽象类，避免持有具体类型的子类，以便工厂方法能随时切换不同的子类返回，却不影响调用方代码。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：



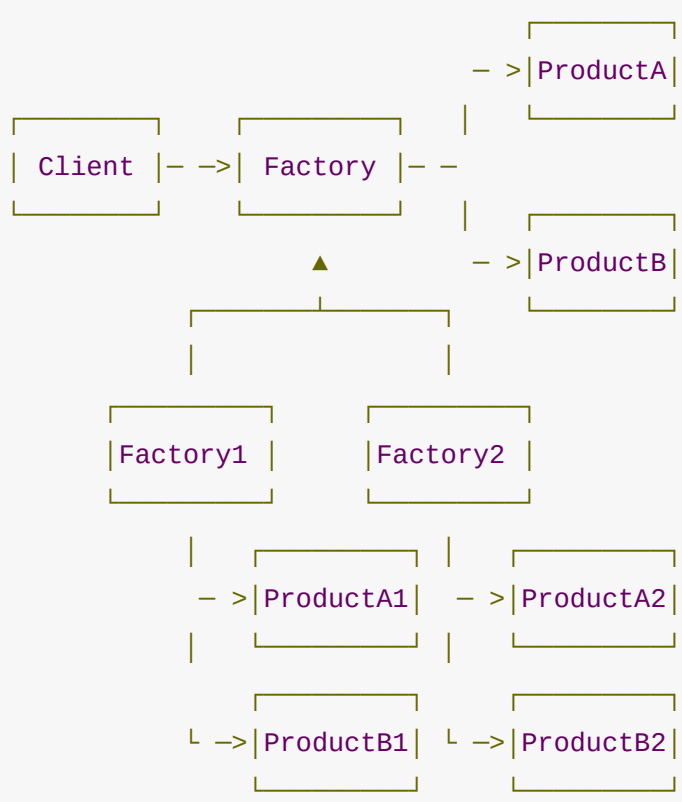


抽象工厂

> 提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。

抽象工厂模式（Abstract Factory）是一个比较复杂的创建型模式。

抽象工厂模式和工厂方法不太一样，它要解决的问题比较复杂，不但工厂是抽象的，产品是抽象的，而且有多个产品需要创建，因此，这个抽象工厂会对应到多个实际工厂，每个实际工厂负责创建多个实际产品：



这种模式有点类似于多个供应商负责提供一系列类型的产品。我们举个例子：

假设我们希望为用户提供一个Markdown文本转换为HTML和Word的服务，它的接口定义如下：

```
public interface AbstractFactory {
    // 创建Html文档:
    HtmlDocument createHtml(String md);
    // 创建Word文档:
    WordDocument createWord(String md);
}
```

注意到上面的抽象工厂仅仅是一个接口，没有任何代码。同样的，因为 `HtmlDocument` 和 `WordDocument` 都比较复杂，现在我们并不知道如何实现它们，所以只有接

口：

```
// Html文档接口:
public interface HtmlDocument {
    String toHtml();
    void save(Path path) throws IOException;
}

// Word文档接口:
public interface WordDocument {
    void save(Path path) throws IOException;
}
```

这样，我们就定义好了抽象工厂（`AbstractFactory`）以及两个抽象产品（`HtmlDocument`和`WordDocument`）。因为实现它们比较困难，我们决定让供应商来完成。

现在市场上有两家供应商：FastDoc Soft的产品便宜，并且转换速度快，而GoodDoc Soft的产品贵，但转换效果好。我们决定同时使用这两家供应商的产品，以便给免费用户和付费用户提供不同的服务。

我们先看看FastDoc Soft的产品是如何实现的。首先，FastDoc Soft必须要有实际的产品，即`FastHtmlDocument`和`FastWordDocument`：

```
public class FastHtmlDocument implements HtmlDocument {
    public String toHtml() {
        ...
    }
    public void save(Path path) throws IOException {
        ...
    }
}

public class FastWordDocument implements WordDocument {
    public void save(Path path) throws IOException {
        ...
    }
}
```

然后，FastDoc Soft必须提供一个实际的工厂来生产这两种产品，即`FastFactory`：

```
public class FastFactory implements AbstractFactory {
    public HtmlDocument createHtml(String md) {
```

```
        return new FastHtmlDocument(md);
    }
    public WordDocument createWord(String md) {
        return new FastWordDocument(md);
    }
}
```

这样，我们就可以使用FastDoc Soft的服务了。客户端编写代码如下：

```
// 创建AbstractFactory，实际类型是FastFactory:
AbstractFactory factory = new FastFactory();
// 生成Html文档:
HtmlDocument html = factory.createHtml("#Hello\nHello, world!");
html.save(Paths.get(".", "fast.html"));
// 生成Word文档:
WordDocument word = fastFactory.createWord("#Hello\nHello, world!");
word.save(Paths.get(".", "fast.doc"));
```

如果我们要同时使用GoodDoc Soft的服务怎么办？因为用了抽象工厂模式，GoodDoc Soft只需要根据我们定义的抽象工厂和抽象产品接口，实现自己的实际工厂和实际产品即可：

```
// 实际工厂:
public class GoodFactory implements AbstractFactory {
    public HtmlDocument createHtml(String md) {
        return new GoodHtmlDocument(md);
    }
    public WordDocument createWord(String md) {
        return new GoodWordDocument(md);
    }
}

// 实际产品:
public class GoodHtmlDocument implements HtmlDocument {
    ...
}

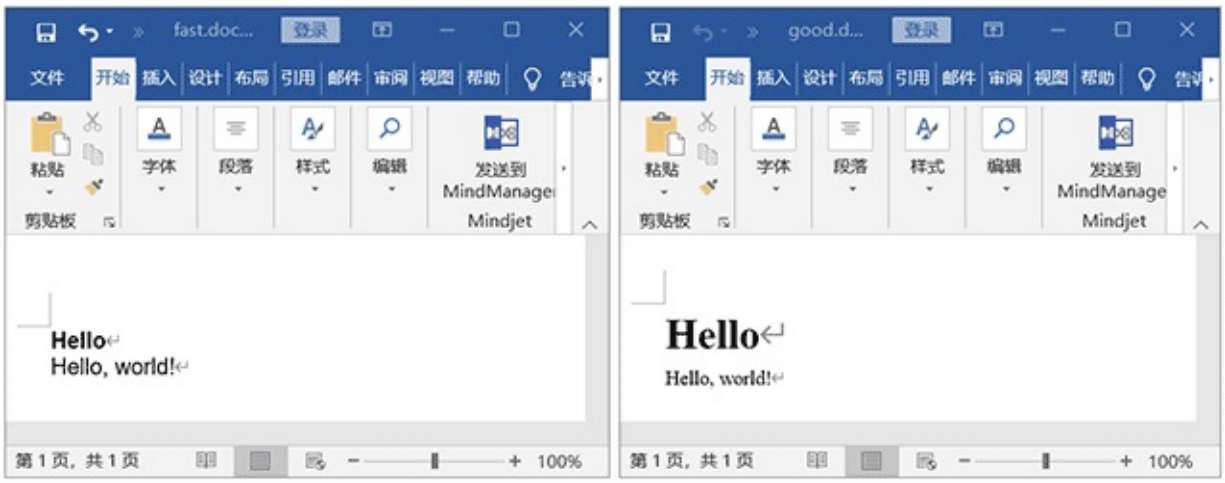
public class GoodWordDocument implements HtmlDocument {
    ...
}
```

客户端要使用GoodDoc Soft的服务，只需要把原来的 `new FastFactory()` 切换为 `new GoodFactory()` 即可。

注意到客户端代码除了通过 `new` 创建了 `FastFactory` 或 `GoodFactory` 外，其余代码只引用了产品接口，并未引用任何实际产品（例如， `FastHtmlDocument` ），如果把创建工厂的代码放到 `AbstractFactory` 中，就可以连实际工厂也屏蔽了：

```
1.  public interface AbstractFactory {
2.      public static AbstractFactory createFactory(String name) {
3.          if (name.equalsIgnoreCase("fast")) {
4.              return new FastFactory();
5.          } else if (name.equalsIgnoreCase("good")) {
6.              return new GoodFactory();
7.          } else {
8.              throw new IllegalArgumentException("Invalid factory name");
9.          }
10.     }
11. }
```

我们来看看 `FastFactory` 和 `GoodFactory` 创建的 `WordDocument` 的实际效果：



注意：出于简化代码的目的，我们只支持两种Markdown语法：以 `#` 开头的标题以及普通正文。

练习

使用Abstract Factory模式实现 `HtmlDocument` 和 `WordDocument` 。

从码云下载练习：Abstract Factory模式 （推荐使用IDE练习插件快速下载）

小结

抽象工厂模式是为了让创建工厂和一组产品与使用相分离，并可以随时切换到另一个工厂以及另一组产品；

抽象工厂模式实现的关键点是定义工厂接口和产品接口，但如何实现工厂与产品本身需要留给具体的子类实现，客户端只和抽象工厂与抽象产品打交道。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





生成器

> 将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。

生成器模式（Builder）是使用多个“小型”工厂来最终创建出一个完整对象。

当我们使用Builder的时候，一般来说，是因为创建这个对象的步骤比较多，每个步骤都需要一个零部件，最终组合成一个完整的对象。

我们仍然以Markdown转HTML为例，因为直接编写一个完整的转换器比较困难，但如果针对类似下面的一行文本：

```
# this is a heading
```

转换成HTML就很简单：

```
<h1>this is a heading</h1>
```

因此，我们把Markdown转HTML看作一行一行的转换，每一行根据语法，使用不同的转换器：

- 如果以 `#` 开头，使用 `HeadingBuilder` 转换；
- 如果以 `>` 开头，使用 `QuoteBuilder` 转换；
- 如果以 `---` 开头，使用 `HrBuilder` 转换；
- 其余使用 `ParagraphBuilder` 转换。

这个 `HtmlBuilder` 写出来如下：

```
public class HtmlBuilder {
    private HeadingBuilder headingBuilder = new HeadingBuilder();
    private HrBuilder hrBuilder = new HrBuilder();
    private ParagraphBuilder paragraphBuilder = new ParagraphBuilder();
    private QuoteBuilder quoteBuilder = new QuoteBuilder();

    public String toHtml(String markdown) {
        StringBuilder buffer = new StringBuilder();
        markdown.lines().forEach(line -> {
            if (line.startsWith("#")) {
                buffer.append(headingBuilder.buildHeading(line)).append('\n');
            } else if (line.startsWith(">")) {
                buffer.append(quoteBuilder.buildQuote(line)).append('\n');
            } else if (line.startsWith("---")) {
```

```
                buffer.append(hrBuilder.buildHr(line)).append('\n');
            } else {

buffer.append(paragraphBuilder.buildParagraph(line)).append('\n');
            }
        });
        return buffer.toString();
    }
}
```

注意观察上述代码，`HtmlBuilder`并不是一次性把整个Markdown转换为HTML，而是一行一行转换，并且，它自己并不会将某一行转换为特定的HTML，而是根据特性把每一行都“委托”给一个`XxxBuilder`去转换，最后，把所有转换的结果组合起来，返回给客户端。

这样一来，我们只需要针对每一种类型编写不同的Builder。例如，针对以`#`开头的行，需要`HeadingBuilder`：

```
public class HeadingBuilder {
    public String buildHeading(String line) {
        int n = 0;
        while (line.charAt(0) == '#') {
            n++;
            line = line.substring(1);
        }
        return String.format("<h%d>%s</h%d>", n, line.strip(), n);
    }
}
```

注意：实际解析Markdown是带有状态的，即下一行的语义可能与上一行相关。这里我们简化了语法，把每一行视为可以独立转换。

可见，使用Builder模式时，适用于创建的对象比较复杂，最好一步一步创建出“零件”，最后再装配起来。

JavaMail的`MimeMessage`就可以看作是一个Builder模式，只不过Builder和最终产品合二为一，都是`MimeMessage`：

```
Multipart multipart = new MimeMultipart();
// 添加text:
BodyPart textpart = new MimeBodyPart();
textpart.setContent(body, "text/html;charset=utf-8");
multipart.addBodyPart(textpart);
```

```
// 添加image:
BodyPart imagepart = new MimeBodyPart();
imagepart.setFileName(fileName);
imagepart.setDataHandler(new DataHandler(new ByteArrayDataSource(input,
"application/octet-stream")));
multipart.addBodyPart(imagepart);

MimeMessage message = new MimeMessage(session);
// 设置发送方地址:
message.setFrom(new InternetAddress("me@example.com"));
// 设置接收方地址:
message.setRecipient(Message.RecipientType.TO, new
InternetAddress("xiaoming@somewhere.com"));
// 设置邮件主题:
message.setSubject("Hello", "UTF-8");
// 设置邮件内容为multipart:
message.setContent(multipart);
```

很多时候，我们可以简化Builder模式，以链式调用的方式来创建对象。例如，我们经常编写这样的代码：

```
StringBuilder builder = new StringBuilder();
builder.append(secure ? "https://" : "http://")
        .append("www.liaoxuefeng.com")
        .append("/")
        .append("?t=0");
String url = builder.toString();
```

由于我们经常需要构造URL字符串，可以使用Builder模式编写一个URLBuilder，调用方式如下：

```
String url = URLBuilder.builder() // 创建Builder
         .setDomain("www.liaoxuefeng.com") // 设置domain
         .setScheme("https") // 设置scheme
         .setPath("/") // 设置路径
         .setQuery(Map.of("a", "123", "q", "K&R")) // 设置query
         .build(); // 完成build
```

练习

从码云下载练习：使用Builder模式实现一个URLBuilder （推荐使用IDE练习插件快速下

载）

小结

Builder模式是为了创建一个复杂的对象，需要多个步骤完成创建，或者需要多个零件组装的场景，且创建过程中可以灵活调用不同的步骤或组件。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





原型

> 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。

原型模式，即Prototype，是指创建新对象的时候，根据现有的一个原型来创建。

我们举个例子：如果我们已经有了一个 `String[]` 数组，想再创建一个一模一样的 `String[]` 数组，怎么写？

实际上创建过程很简单，就是把现有数组的元素复制到新数组。如果我们把这个创建过程封装一下，就成了原型模式。用代码实现如下：

```
// 原型:
String[] original = { "Apple", "Pear", "Banana" };
// 新对象:
String[] copy = Arrays.copyOf(original, original.length);
```

对于普通类，我们如何实现原型拷贝？Java的 `Object` 提供了一个 `clone()` 方法，它的意图就是复制一个新的对象出来，我们需要实现一个 `Cloneable` 接口来标识一个对象是“可复制”的：

```
public class Student implements Cloneable {
    private int id;
    private String name;
    private int score;

    // 复制新对象并返回:
    public Object clone() {
        Student std = new Student();
        std.id = this.id;
        std.name = this.name;
        std.score = this.score;
        return std;
    }
}
```

使用的时候，因为 `clone()` 的方法签名是定义在 `Object` 中，返回类型也是 `Object` ，所以要强制转型，比较麻烦：

```
Student std1 = new Student();
std1.setId(123);
std1.setName("Bob");
```

```
std1.setScore(88);
// 复制新对象:
Student std2 = (Student) std1.clone();
System.out.println(std1);
System.out.println(std2);
System.out.println(std1 == std2); // false
```

实际上，使用原型模式更好的方式是定义一个 `copy()` 方法，返回明确的类型：

```
public class Student {
    private int id;
    private String name;
    private int score;

    public Student copy() {
        Student std = new Student();
        std.id = this.id;
        std.name = this.name;
        std.score = this.score;
        return std;
    }
}
```

原型模式应用不是很广泛，因为很多实例会持有类似文件、Socket这样的资源，而这些资源是无法复制给另一个对象共享的，只有存储简单类型的“值”对象可以复制。

练习

给 `Student` 类增加 `clone()` 方法。

从 gitee 下载练习：原型模式练习 （推荐使用IDE练习插件快速下载）

小结

原型模式是根据一个现有对象实例复制出一个新的实例，复制出的类型和属性与原实例相同。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





单例

> 保证一个类仅有一个实例，并提供一个访问它的全局访问点。

单例模式（Singleton）的目的是为了保证在一个进程中，某个类有且仅有一个实例。

因为这个类只有一个实例，因此，自然不能让调用方使用 `new Xyz()` 来创建实例了。所以，单例的构造方法必须是 `private` ，这样就防止了调用方自己创建实例，但是在类的内部，是可以用一个静态字段来引用唯一创建的实例的：

```
public class Singleton {
    // 静态字段引用唯一实例:
    private static final Singleton INSTANCE = new Singleton();

    // private构造方法保证外部无法实例化:
    private Singleton() {
    }
}
```

那么问题来了，外部调用方如何获得这个唯一实例？

答案是提供一个静态方法，直接返回实例：

```
public class Singleton {
    // 静态字段引用唯一实例:
    private static final Singleton INSTANCE = new Singleton();

    // 通过静态方法返回实例:
    public static Singleton getInstance() {
        return INSTANCE;
    }

    // private构造方法保证外部无法实例化:
    private Singleton() {
    }
}
```

或者直接把 `static` 变量暴露给外部：

```
public class Singleton {
    // 静态字段引用唯一实例:
```

```
    public static final Singleton INSTANCE = new Singleton();

    // private构造方法保证外部无法实例化:
    private Singleton() {
    }
}
```

所以，单例模式的实现方式很简单：

- 只有 `private` 构造方法，确保外部无法实例化；
- 通过 `private static` 变量持有唯一实例，保证全局唯一性；
- 通过 `public static` 方法返回此唯一实例，使外部调用方能获取到实例。 Java标准库有一些类就是单例，例如 `Runtime` 这个类：

```
Runtime runtime = Runtime.getRuntime();
```

有些童鞋可能听说过延迟加载，即在调用方第一次调用 `getInstance()` 时才初始化全局唯一实例，类似这样：

```
public class Singleton {
    private static Singleton INSTANCE = null;

    public static Singleton getInstance() {
        if (INSTANCE == null) {
            INSTANCE = new Singleton();
        }
        return INSTANCE;
    }

    private Singleton() {
    }
}
```

遗憾的是，这种写法在多线程中是错误的，在竞争条件下会创建出多个实例。必须对整个方法进行加锁：

```
public synchronized static Singleton getInstance() {
    if (INSTANCE == null) {
        INSTANCE = new Singleton();
    }
    return INSTANCE;
}
```

但加锁会严重影响并发性能。还有些童鞋听说过双重检查，类似这样：

```
public static Singleton getInstance() {
    if (INSTANCE == null) {
        synchronized (Singleton.class) {
            if (INSTANCE == null) {
                INSTANCE = new Singleton();
            }
        }
    }
    return INSTANCE;
}
```

然而，由于Java的内存模型，双重检查在这里不成立。要真正实现延迟加载，只能通过Java的ClassLoader机制完成。如果没有特殊的需求，使用Singleton模式的时候，最好不要延迟加载，这样会使代码更简单。

另一种实现Singleton的方式是利用Java的 `enum` ，因为Java保证枚举类的每个枚举都是单例，所以我们只需要编写一个只有一个枚举的类即可：

```
public enum World {
    // 唯一枚举:
    INSTANCE;

    private String name = "world";

    public String getName() {
        return this.name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

枚举类也完全可以像其他类那样定义自己的字段、方法，这样上面这个 `World` 类在调用方看来就可以这么用：

```
String name = World.INSTANCE.getName();
```

使用枚举实现Singleton还避免了第一种方式实现Singleton的一个潜在问题：即序列化和反序列化会绕过普通类的 `private` 构造方法从而创建出多个实例，而枚举类就没有这个问题。

那我们什么时候应该用Singleton呢？实际上，很多程序，尤其是Web程序，大部分服务类都应该被视作Singleton，如果全部按Singleton的写法写，会非常麻烦，所以，通常是通过约定让框架（例如Spring）来实例化这些类，保证只有一个实例，调用方自觉通过框架获取实例而不是 `new` 操作符：

```
@Component // 表示一个单例组件
public class MyService {
    ...
}
```

因此，除非确有必要，否则Singleton模式一般以“约定”为主，不会刻意实现它。

练习

使用两种Singleton模式实现单例：

从 gitee 下载练习：Singleton练习 （推荐使用IDE练习插件快速下载）

小结

Singleton模式是为了保证一个程序的运行期间，某个类有且只有一个全局唯一实例；

Singleton模式既可以严格实现，也可以以约定的方式把普通类视作单例。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





结构型模式

结构型模式主要涉及如何组合各种对象以便获得更好、更灵活的结构。虽然面向对象的继承机制提供了最基本的子类扩展父类的功能，但结构型模式不仅仅简单地使用继承，而更多地通过组合与运行期的动态组合来实现更灵活的功能。

结构型模式有：

- 适配器
- 桥接
- 组合
- 装饰器
- 外观
- 享元
- 代理

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





适配器

> 将一个类的接口转换成客户希望的另外一个接口，使得原本由于接口不兼容而不能一起工作的那些类可以一起工作。

适配器模式是Adapter，也称Wrapper，是指如果一个接口需要B接口，但是待传入的对象却是A接口，怎么办？

我们举个例子。如果去美国，我们随身带的电器是无法直接使用的，因为美国的插座标准和中国不同，所以，我们需要一个适配器：



在程序设计中，适配器也是类似的。我们已经有一个 `Task` 类，实现了 `Callable` 接口：

```
public class Task implements Callable<Long> {
    private long num;
    public Task(long num) {
        this.num = num;
    }

    public Long call() throws Exception {
        long r = 0;
        for (long n = 1; n <= this.num; n++) {
            r = r + n;
        }
        System.out.println("Result: " + r);
        return r;
    }
}
```

现在，我们想通过一个线程去执行它：

```
Callable<Long> callable = new Task(123450000L);
Thread thread = new Thread(callable); // compile error!
thread.start();
```

发现编译不过！因为 `Thread` 接收 `Runnable` 接口，但不接收 `Callable` 接口，肿么办？

一个办法是改写 `Task` 类，把实现的 `Callable` 改为 `Runnable` ，但这样做不好，因为 `Task` 很可能在其他地方作为 `Callable` 被引用，改写 `Task` 的接口，会导致其他正常工作的代码无法编译。

另一个办法不用改写 `Task` 类，而是用一个Adapter，把这个 `Callable` 接口“变成” `Runnable` 接口，这样，就可以正常编译：

```
Callable<Long> callable = new Task(123450000L);
Thread thread = new Thread(new RunnableAdapter(callable));
thread.start();
```

这个 `RunnableAdapter` 类就是Adapter，它接收一个 `Callable` ，输出一个 `Runnable` 。怎么实现这个 `RunnableAdapter` 呢？我们先看完整的代码：

```
public class RunnableAdapter implements Runnable {
    // 引用待转换接口:
    private Callable<?> callable;

    public RunnableAdapter(Callable<?> callable) {
        this.callable = callable;
    }

    // 实现指定接口:
    public void run() {
        // 将指定接口调用委托给转换接口调用:
        try {
            callable.call();
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
}
```

编写一个Adapter的步骤如下：

- 实现目标接口，这里是 `Runnable` ；
- 内部持有一个待转换接口的引用，这里是通过字段持有 `Callable` 接口；
- 在目标接口的实现方法内部，调用待转换接口的方法。 这样一来，Thread就可以接收这个 `RunnableAdapter` ，因为它实现了 `Runnable` 接口。 `Thread` 作为调用方，它会调用 `RunnableAdapter` 的 `run()` 方法，在这个 `run()` 方法内部，又调用

了 `Callable` 的 `call()` 方法，相当于 `Thread` 通过一层转换，间接调用了 `Callable` 的 `call()` 方法。

适配器模式在Java标准库中有广泛应用。比如我们持有数据类型是 `String[]` ，但是需要 `List` 接口时，可以用一个Adapter：

```
String[] exist = new String[] {"Good", "morning", "Bob", "and", "Alice"};
Set<String> set = new HashSet<>(Arrays.asList(exist));
```

注意到 `List<T> Arrays.asList(T[])` 就相当于一个转换器，它可以把数组转换为 `List` 。

我们再看一个例子：假设我们持有一个 `InputStream` ，希望调用 `readText(Reader)` 方法，但它的参数类型是 `Reader` 而不是 `InputStream` ，怎么办？

当然是使用适配器，把 `InputStream` “变成” `Reader` ：

```
InputStream input = Files.newInputStream(Paths.get("/path/to/file"));
Reader reader = new InputStreamReader(input, "UTF-8");
readText(reader);
```

`InputStreamReader` 就是Java标准库提供的 `Adapter` ，它负责把一个 `InputStream` 适配为 `Reader` 。类似的还有 `OutputStreamWriter` 。

如果我们把 `readText(Reader)` 方法参数从 `Reader` 改为 `FileReader` ，会有什么问题？这个时候，因为我们需要一个 `FileReader` 类型，就必须把 `InputStream` 适配为 `FileReader` ：

```
FileReader reader = new InputStreamReader(input, "UTF-8"); // compile error!
```

直接使用 `InputStreamReader` 这个Adapter是不行的，因为它只能转换出 `Reader` 接口。事实上，要把 `InputStream` 转换为 `FileReader` 也不是不可能，但需要花费十倍以上的功夫。这时，面向抽象编程这一原则就体现出了威力：持有高层接口不但代码更灵活，而且把各种接口组合起来也更容易。一旦持有某个具体的子类类型，要想做一些改动就非常困难。

练习

使用Adapter模式将 `Callable` 接口适配为 `Runnable` 。

从 gitee 下载练习：Adapter模式 （推荐使用IDE练习插件快速下载）

小结

Adapter模式可以将一个A接口转换为B接口，使得新的对象符合B接口规范。

编写Adapter实际上就是编写一个实现了B接口，并且内部持有A接口的类：

```
public BAdapter implements B {
    private A a;
    public BAdapter(A a) {
        this.a = a;
    }
    public void b() {
        a.a();
    }
}
```

在Adapter内部将B接口的调用“转换”为对A接口的调用。

只有A、B接口均为抽象接口时，才能非常简单地实现Adapter模式。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：



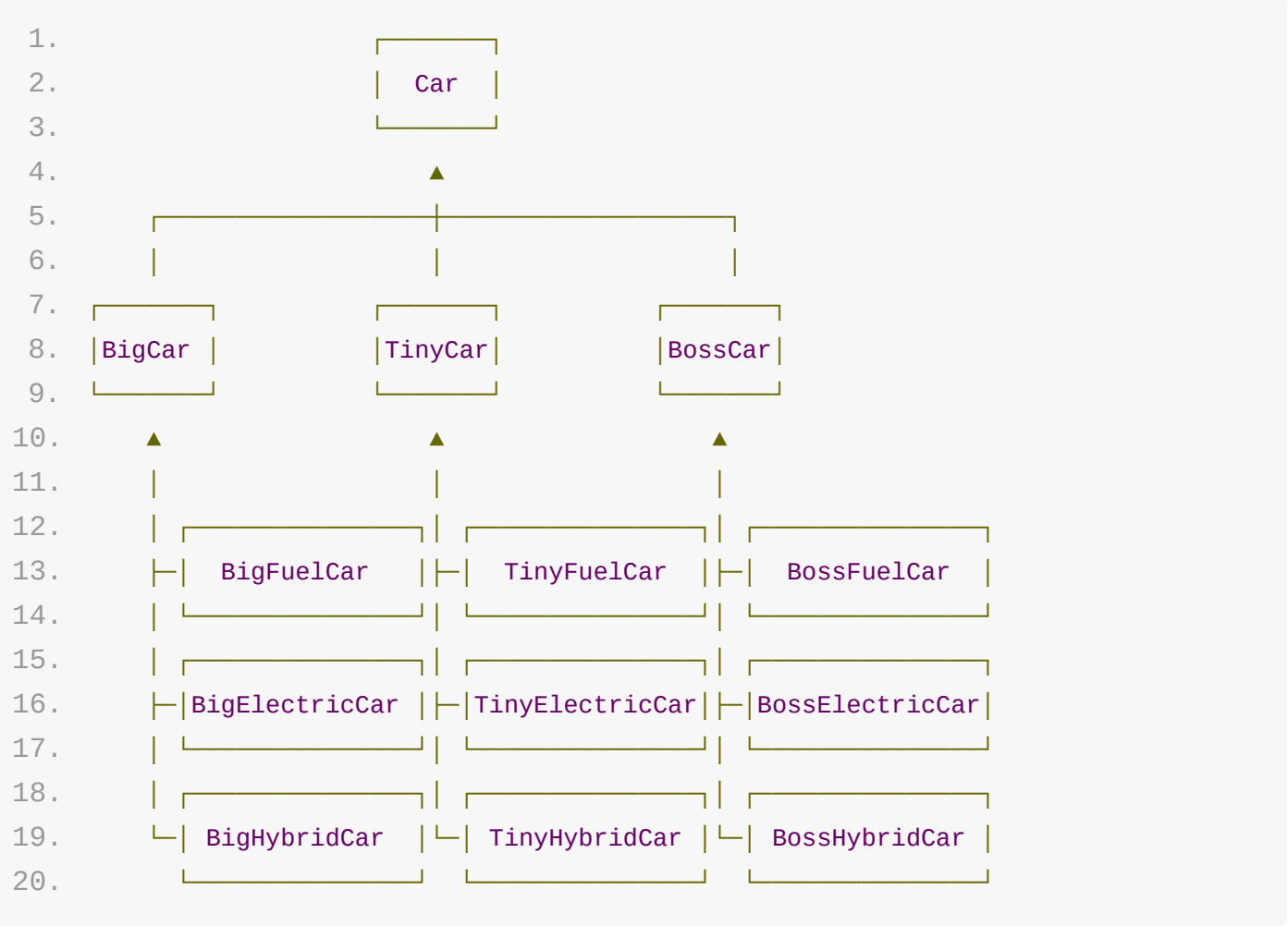


桥接

> 将抽象部分与它的实现部分分离，使它们都可以独立地变化。

桥接模式的定义非常玄乎，直接理解不太容易，所以我们还是举例子。

假设某个汽车厂商生产三种品牌的汽车：Big、Tiny和Boss，每种品牌又可以选择燃油、纯电和混合动力。如果用传统的继承来表示各个最终车型，一共有3个抽象类加9个最终子类：



如果要新增一个品牌，或者加一个新的引擎（比如核动力），那么子类的数量增长更快。

所以，桥接模式就是为了避免直接继承带来的子类爆炸。

我们来看看桥接模式如何解决上述问题。

在桥接模式中，首先把 `Car` 按品牌进行子类化，但是，每个品牌选择什么发动机，不再使用子类扩充，而是通过一个抽象的“修正”类，以组合的形式引入。我们来看看具体的实现。

首先定义抽象类 `Car` ，它引用一个 `Engine` ：

```
public abstract class Car {
```

```
    // 引用Engine:
    protected Engine engine;

    public Car(Engine engine) {
        this.engine = engine;
    }

    public abstract void drive();
}
```

`Engine` 的定义如下：

```
public interface Engine {
    void start();
}
```

紧接着，在一个“修正”的抽象类 `RefinedCar` 中定义一些额外操作：

```
public abstract class RefinedCar extends Car {
    public RefinedCar(Engine engine) {
        super(engine);
    }

    public void drive() {
        this.engine.start();
        System.out.println("Drive " + getBrand() + " car...");
    }

    public abstract String getBrand();
}
```

这样一来，最终的不同品牌继承自 `RefinedCar` ，例如 `BossCar` ：

```
public class BossCar extends RefinedCar {
    public BossCar(Engine engine) {
        super(engine);
    }

    public String getBrand() {
        return "Boss";
    }
}
```

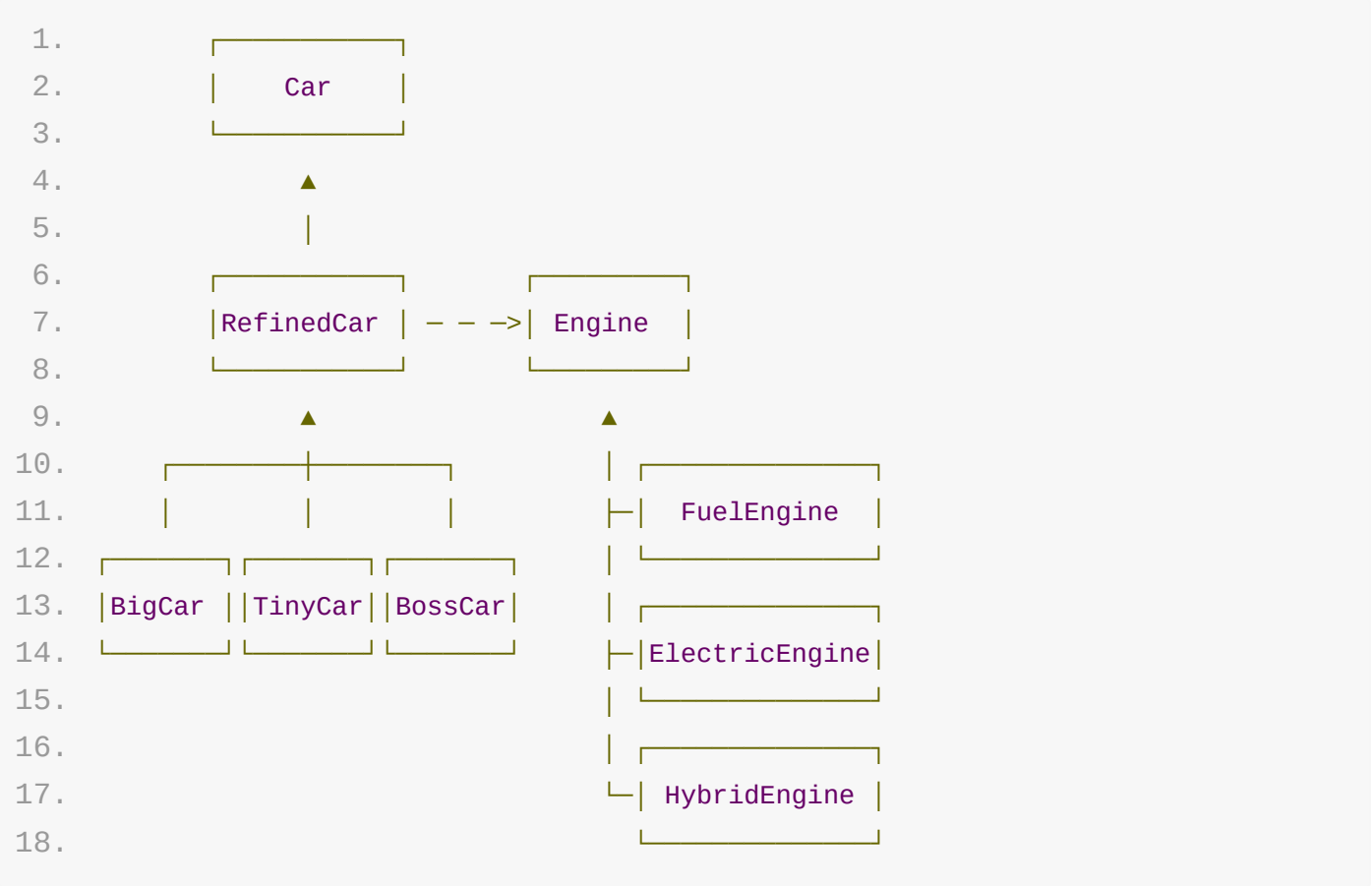
而针对每一种引擎，继承自 `Engine` ，例如 `HybridEngine` ：

```
public class HybridEngine implements Engine {
    public void start() {
        System.out.println("Start Hybrid Engine...");
    }
}
```

客户端通过自己选择一个品牌，再配合一种引擎，得到最终的Car：

```
RefinedCar car = new BossCar(new HybridEngine());
car.drive();
```

使用桥接模式的好处在于，如果要增加一种引擎，只需要针对 `Engine` 派生一个新的子类，如果要增加一个品牌，只需要针对 `RefinedCar` 派生一个子类，任何 `RefinedCar` 的子类都可以和任何一种 `Engine` 自由组合，即一辆汽车的两个维度：品牌和引擎都可以独立地变化。



桥接模式实现比较复杂，实际应用也非常少，但它提供的设计思想值得借鉴，即不要过度使用继承，而是优先拆分某些部件，使用组合的方式来扩展功能。

练习

使用桥接模式扩展一种新的品牌和新的核动力引擎。

从 gitee 下载练习：桥接模式练习 （推荐使用IDE练习插件快速下载）

小结

桥接模式通过分离一个抽象接口和它的实现部分，使得设计可以按两个维度独立扩展。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





组合

> 将对象组合成树形结构以表示“部分-整体”的层次结构，使得用户对单个对象和组合对象的使用具有一致性。

组合模式（Composite）经常用于树形结构，为了简化代码，使用Composite可以把一个叶子节点与一个父节点统一起来处理。

我们来看一个具体的例子。在XML或HTML中，从根节点开始，每个节点都可能包含任意个其他节点，这些层层嵌套的节点就构成了一颗树。

要以树的结构表示XML，我们可以先抽象出节点类型 `Node` ：

```
public interface Node {
    // 添加一个节点为子节点:
    Node add(Node node);
    // 获取子节点:
    List<Node> children();
    // 输出为XML:
    String toXml();
}
```

对于一个 `<abc>` 这样的节点，我们称之为 `ElementNode` ，它可以作为容器包含多个子节点：

```
public class ElementNode implements Node {
    private String name;
    private List<Node> list = new ArrayList<>();

    public ElementNode(String name) {
        this.name = name;
    }

    public Node add(Node node) {
        list.add(node);
        return this;
    }

    public List<Node> children() {
        return list;
    }

    public String toXml() {
```

```
        String start = "<" + name + ">\n";
        String end = "</" + name + ">\n";
        StringJoiner sj = new StringJoiner("", start, end);
        list.forEach(node -> {
            sj.add(node.toXml() + "\n");
        });
        return sj.toString();
    }
}
```

对于普通文本，我们把它看作 `TextNode` ，它没有子节点：

```
public class TextNode implements Node {
    private String text;

    public TextNode(String text) {
        this.text = text;
    }

    public Node add(Node node) {
        throw new UnsupportedOperationException();
    }

    public List<Node> children() {
        return List.of();
    }

    public String toXml() {
        return text;
    }
}
```

此外，还可以有注释节点：

```
public class CommentNode implements Node {
    private String text;

    public CommentNode(String text) {
        this.text = text;
    }

    public Node add(Node node) {
```

```
        throw new UnsupportedOperationException();
    }

    public List<Node> children() {
        return List.of();
    }

    public String toXml() {
        return "<!-- " + text + " -->";
    }
}
```

通过 `ElementNode` 、 `TextNode` 和 `CommentNode` ，我们就可以构造出一颗树：

```
Node root = new ElementNode("school");
root.add(new ElementNode("classA")
        .add(new TextNode("Tom"))
        .add(new TextNode("Alice")));
root.add(new ElementNode("classB")
        .add(new TextNode("Bob"))
        .add(new TextNode("Grace"))
        .add(new CommentNode("comment...")));
System.out.println(root.toXml());
```

最后通过 `root` 节点输出的XML如下：

```
<school>
<classA>
Tom
Alice
</classA>
<classB>
Bob
Grace
<!-- comment... -->
</classB>
</school>
```

可见，使用Composite模式时，需要先统一单个节点以及“容器”节点的接口：

```
            ┌──────────┐
            │   Node   │
```

```
                └──────────┘
                     ▲
          ┌──────────┼──────────┐
          │          │          │
┌─────────┐ ┌─────────┐ ┌─────────┐
|ElementNode| | TextNode | |CommentNode|
└─────────┘ └─────────┘ └─────────┘
```

作为容器节点的 `ElementNode` 又可以添加任意个 `Node` ，这样就可以构成层级结构。

类似的，像文件夹和文件、GUI窗口的各种组件，都符合Composite模式的定义，因为它们的结构天生就是层级结构。

练习

从 gitee 下载练习：使用Composite模式构造XML （推荐使用IDE练习插件快速下载）

小结

Composite模式使得叶子对象和容器对象具有一致性，从而形成统一的树形结构，并用一致的方式去处理它们。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





装饰器

> 动态地给一个对象添加一些额外的职责。就增加功能来说，相比生成子类更为灵活。

装饰器（Decorator）模式，是一种在运行期动态给某个对象的实例增加功能的方法。

我们在IO的Filter模式一节中其实已经讲过装饰器模式了。在Java标准库中，`InputStream`是抽象类，`FileInputStream`、`ServletInputStream`、`Socket.getInputStream()`这些`InputStream`都是最终数据源。

现在，如果要给不同的最终数据源增加缓冲功能、计算签名功能、加密解密功能，那么，3个最终数据源、3种功能一共需要9个子类。如果继续增加最终数据源，或者增加新功能，子类会爆炸式增长，这种设计方式显然是不可取的。

Decorator模式的目的就是把一个一个的附加功能，用Decorator的方式给一层一层地累加到原始数据源上，最终，通过组合获得我们想要的功能。

例如：给`FileInputStream`增加缓冲和解压缩功能，用Decorator模式写出来如下：

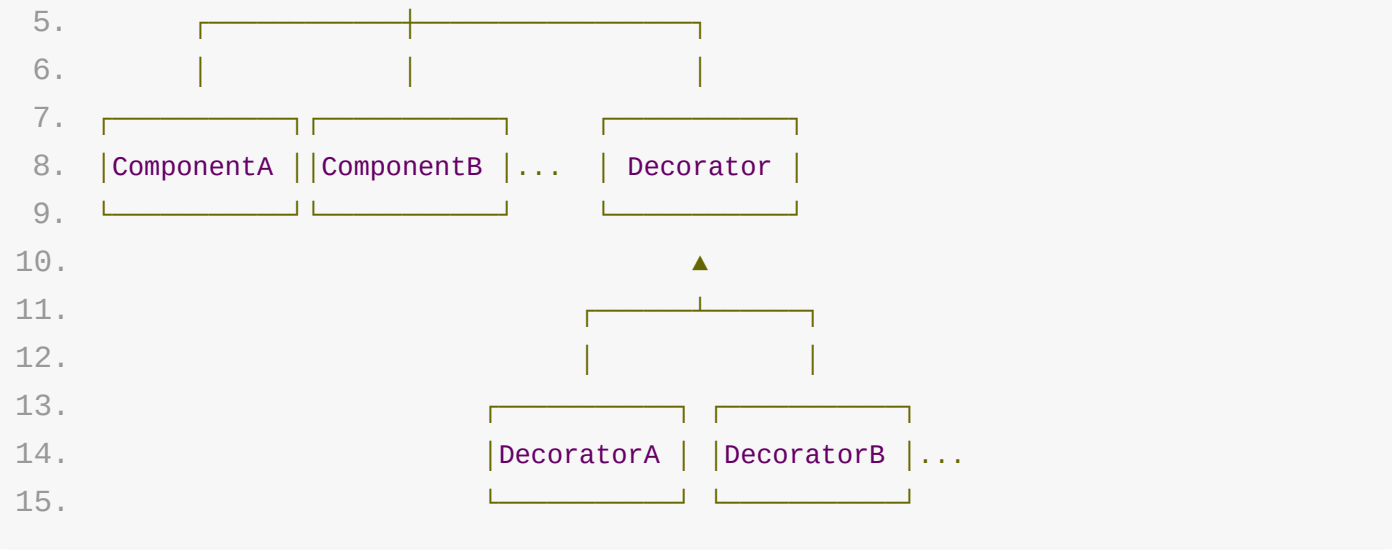
```
// 创建原始的数据源:
InputStream fis = new FileInputStream("test.gz");
// 增加缓冲功能:
InputStream bis = new BufferedInputStream(fis);
// 增加解压缩功能:
InputStream gis = new GZIPInputStream(bis);
```

或者一次性写成这样：

```
InputStream input = new GZIPInputStream( // 第二层装饰
                        new BufferedInputStream( // 第一层装饰
                            new FileInputStream("test.gz") // 核心功能
                        ));
```

观察`BufferedInputStream`和`GZIPInputStream`，它们实际上都是从`FilterInputStream`继承的，这个`FilterInputStream`就是一个抽象的Decorator。我们用图把Decorator模式画出来如下：

```
            ┌─────────────┐
            │ Component │
            └─────────────┘
                   ▲
```



最顶层的Component是接口，对应到IO的就是 `InputStream` 这个抽象类。ComponentA、ComponentB是实际的子类，对应到IO的就是 `FileInputStream` 、 `ServletInputStream` 这些数据源。Decorator是用于实现各个附加功能的抽象装饰器，对应到IO的就是 `FilterInputStream` 。而从Decorator派生的就是一个一个的装饰器，它们每个都有独立的功能，对应到IO的就是 `BufferedInputStream` 、 `GZIPInputStream` 等。

Decorator模式有什么好处？它实际上把核心功能和附加功能给分开了。核心功能指 `FileInputStream` 这些真正读数据的源头，附加功能指加缓冲、压缩、解密这些功能。如果我们要新增核心功能，就增加Component的子类，例如 `ByteInputStream` 。如果我们要增加附加功能，就增加Decorator的子类，例如 `CipherInputStream` 。两部分都可以独立地扩展，而具体如何附加功能，由调用方自由组合，从而极大地增强了灵活性。

如果我们要自己设计完整的Decorator模式，应该如何设计？

我们还是举个栗子：假设我们需要渲染一个HTML的文本，但是文本还可以附加一些效果，比如加粗、变斜体、加下划线等。为了实现动态附加效果，可以采用Decorator模式。

首先，仍然需要定义顶层接口 `TextNode` ：

```
public interface TextNode {
    // 设置text:
    void setText(String text);
    // 获取text:
    String getText();
}
```

对于核心节点，例如 `<span>` ，它需要从 `TextNode` 直接继承：

```
public class SpanNode implements TextNode {
    private String text;
```

```
3.
4.      public void setText(String text) {
5.          this.text = text;
6.      }
7.
8.      public String getText() {
9.          return "<span>" + text + "</span>";
10.     }
11. }
```

紧接着，为了实现Decorator模式，需要有一个抽象的Decorator类：

```
1.  public abstract class NodeDecorator implements TextNode {
2.      protected final TextNode target;
3.
4.      protected NodeDecorator(TextNode target) {
5.          this.target = target;
6.      }
7.
8.      public void setText(String text) {
9.          this.target.setText(text);
10.     }
11. }
```

这个 `NodeDecorator` 类的核心是持有一个 `TextNode` ，即将要把功能附加到的 `TextNode` 实例。接下来就可以写一个加粗功能：

```
1.  public class BoldDecorator extends NodeDecorator {
2.      public BoldDecorator(TextNode target) {
3.          super(target);
4.      }
5.
6.      public String getText() {
7.          return "<b>" + target.getText() + "</b>";
8.      }
9.  }
```

类似的，可以继续加 `ItalicDecorator` 、 `UnderlineDecorator` 等。客户端可以自由组合这些Decorator：

```
1.  TextNode n1 = new SpanNode();
```

```
TextNode n2 = new BoldDecorator(new UnderlineDecorator(new SpanNode()));
TextNode n3 = new ItalicDecorator(new BoldDecorator(new SpanNode()));
n1.setText("Hello");
n2.setText("Decorated");
n3.setText("World");
System.out.println(n1.getText());
// 输出<span>Hello</span>

System.out.println(n2.getText());
// 输出<b><u><span>Decorated</span></u></b>

System.out.println(n3.getText());
// 输出<i><b><span>World</span></b></i>
```

练习

使用Decorator添加一个 `<del>` 标签表示删除。

从 gitee 下载练习：Decorator练习 （推荐使用IDE练习插件快速下载）

小结

使用Decorator模式，可以独立增加核心功能，也可以独立增加附加功能，二者互不影响；

可以在运行期动态地给核心功能增加任意个附加功能。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





外观

> 为子系统中的一组接口提供一个一致的界面。Facade模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。

外观模式，即Facade，是一个比较简单的模式。它的基本思想如下：

如果客户端要跟许多子系统打交道，那么客户端需要了解各个子系统的接口，比较麻烦。如果有一个统一的“中介”，让客户端只跟中介打交道，中介再去跟各个子系统打交道，对客户端来说就比较简单。所以Facade就相当于搞了一个中介。

我们以注册公司为例，假设注册公司需要三步：

- 向工商局申请公司营业执照；
- 在银行开设账户；
- 在税务局开设纳税号。 以下是三个系统的接口：

```
// 工商注册:
public class AdminOfIndustry {
    public Company register(String name) {
        ...
    }
}

// 银行开户:
public class Bank {
    public String openAccount(String companyId) {
        ...
    }
}

// 纳税登记:
public class Taxation {
    public String applyTaxCode(String companyId) {
        ...
    }
}
```

如果子系统比较复杂，并且客户对流程也不熟悉，那就把这些流程全部委托给中介：

```
public class Facade {
```

```
    public Company openCompany(String name) {
        Company c = this.admin.register(name);
        String bankAccount = this.bank.openAccount(c.getId());
        c.setBankAccount(bankAccount);
        String taxCode = this.taxation.applyTaxCode(c.getId());
        c.setTaxCode(taxCode);
        return c;
    }
}
```

这样，客户端只跟Facade打交道，一次完成公司注册的所有繁琐流程：

```
Company c = facade.openCompany("Facade Software Ltd.");
```

很多Web程序，内部有多个子系统提供服务，经常使用一个统一的Facade入口，例如一个 `RestApiController` ，使得外部用户调用的时候，只关心Facade提供的接口，不用管内部到底是哪个子系统处理的。

更复杂的Web程序，会有多个Web服务，这个时候，经常会使用一个统一的网关入口来自动转发到不同的Web服务，这种提供统一入口的网关就是Gateway，它本质上也是一个Facade，但可以附加一些用户认证、限流限速的额外服务。

练习

使用Facade模式实现一个注册公司的“中介”服务。

从 gitee 下载练习：Facade模式练习 （推荐使用IDE练习插件快速下载）

小结

Facade模式是为了给客户端提供一个统一入口，并对外屏蔽内部子系统的调用细节。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





享元

> 运用共享技术有效地支持大量细粒度的对象。

享元（Flyweight）的核心思想很简单：如果一个对象实例一经创建就不可变，那么反复创建相同的实例就没有必要，直接向调用方返回一个共享的实例就行，这样即节省内存，又可以减少创建对象的过程，提高运行速度。

享元模式在Java标准库中有很多应用。我们知道，包装类型如 `Byte`、`Integer` 都是不变类，因此，反复创建同一个值相同的包装类型是没有必要的。以 `Integer` 为例，如果我们通过 `Integer.valueOf()` 这个静态工厂方法创建 `Integer` 实例，当传入的 `int` 范围在 `-128` ~ `+127` 之间时，会直接返回缓存的 `Integer` 实例：

```
// 享元模式
public class Main {
    public static void main(String[] args) throws InterruptedException {
        Integer n1 = Integer.valueOf(100);
        Integer n2 = Integer.valueOf(100);
        System.out.println(n1 == n2); // true
    }
}
```

▶ Run

对于 `Byte` 来说，因为它一共只有256个状态，所以，通过 `Byte.valueOf()` 创建的 `Byte` 实例，全部都是缓存对象。

因此，享元模式就是通过工厂方法创建对象，在工厂方法内部，很可能返回缓存的实例，而不是新创建实例，从而实现不可变实例的复用。

总是使用工厂方法而不是new操作符创建实例，可获得享元模式的好处。

在实际应用中，享元模式主要应用于缓存，即客户端如果重复请求某些对象，不必每次查询数据库或者读取文件，而是直接返回内存中缓存的数据。

我们以 `Student` 为例，设计一个静态工厂方法，它在内部可以返回缓存的对象：

```
1. public class Student {
2.     // 持有缓存:
3.     private static final Map<String, Student> cache = new HashMap<>();
4.
5.     // 静态工厂方法:
```

```
    public static Student create(int id, String name) {
        String key = id + "\n" + name;
        // 先查找缓存:
        Student std = cache.get(key);
        if (std == null) {
            // 未找到,创建新对象:
            System.out.println(String.format("create new Student(%s, %s)", id, name));
            std = new Student(id, name);
            // 放入缓存:
            cache.put(key, std);
        } else {
            // 缓存中存在:
            System.out.println(String.format("return cached Student(%s, %s)", std.id, std.name));
        }
        return std;
    }

    private final int id;
    private final String name;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
    }
}
```

在实际应用中，我们经常使用成熟的缓存库，例如Guava的Cache，因为它提供了最大缓存数量限制、定时过期等实用功能。

练习

从 gitee 下载练习：使用享元模式实现缓存 （推荐使用IDE练习插件快速下载）

小结

享元模式的设计思想是尽量复用已创建的对象，常用于工厂方法内部的优化。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





代理

> 为其他对象提供一种代理以控制对这个对象的访问。

代理模式，即Proxy，它和Adapter模式很类似。我们先回顾Adapter模式，它用于把A接口转换为B接口：

```
public BAdapter implements B {
    private A a;
    public BAdapter(A a) {
        this.a = a;
    }
    public void b() {
        a.a();
    }
}
```

而Proxy模式不是把A接口转换成B接口，它还是转换成A接口：

```
public AProxy implements A {
    private A a;
    public AProxy(A a) {
        this.a = a;
    }
    public void a() {
        this.a.a();
    }
}
```

合着Proxy就是为了给A接口再包一层，这不是脱了裤子放屁吗？

当然不是。我们观察Proxy的实现A接口的方法：

```
public void a() {
    this.a.a();
}
```

这样写当然没啥卵用。但是，如果我们在调用 `a.a()` 的前后，加一些额外的代码：

```
public void a() {
```

```
    if (getCurrentUser().isRoot()) {
        this.a.a();
    } else {
        throw new SecurityException("Forbidden");
    }
}
```

这样一来，我们就实现了权限检查，只有符合要求的用户，才会真正调用目标方法，否则，会直接抛出异常。

有的童鞋会问，为啥不把权限检查的功能直接写到目标实例A的内部？

因为我们编写代码的原则有：

- 职责清晰：一个类只负责一件事；
- 易于测试：一次只测一个功能。

用Proxy实现这个权限检查，我们可以获得更清晰、更简洁的代码：

- A接口：只定义接口；
- ABusiness类：只实现A接口的业务逻辑；
- APermissionProxy类：只实现A接口的权限检查代理。

如果我们希望编写其他类型的代理，可以继续增加类似ALogProxy，而不必对现有的A接口、ABusiness类进行修改。

实际上权限检查只是代理模式的一种应用。Proxy还广泛应用在：

远程代理

远程代理即Remote Proxy，本地的调用者持有的接口实际上是一个代理，这个代理负责把对接口的方法访问转换成远程调用，然后返回结果。Java内置的RMI机制就是一个完整的远程代理模式。

虚代理

虚代理即Virtual Proxy，它让调用者先持有一个代理对象，但真正的对象尚未创建。如果没有必要，这个真正的对象是不会被创建的，直到客户端需要真的必须调用时，才创建真正的对象。JDBC的连接池返回的JDBC连接（Connection对象）就可以是一个虚代理，即获取连接时根本没有任何实际的数据库连接，直到第一次执行JDBC查询或更新操作时，才真正创建实际的JDBC连接。

保护代理

保护代理即Protection Proxy，它用代理对象控制对原始对象的访问，常用于鉴权。

智能引用

智能引用即Smart Reference，它也是一种代理对象，如果有很多客户端对它进行访问，通过内部的计数器可以在外部调用者都不使用后自动释放它。

我们来看一下如何应用代理模式编写一个JDBC连接池（`DataSource`）。我们首先来编写一个虚代理，即如果调用者获取到`Connection`后，并没有执行任何SQL操作，那么这个Connection Proxy实际上并不会真正打开JDBC连接。调用者代码如下：

```
    DataSource lazyDataSource = new LazyDataSource(jdbcUrl, jdbcUsername,
1.  jdbcPassword);
2.  System.out.println("get lazy connection...");
3.  try (Connection conn1 = lazyDataSource.getConnection()) {
4.      // 并没有实际打开真正的Connection
5.  }
6.  System.out.println("get lazy connection...");
7.  try (Connection conn2 = lazyDataSource.getConnection()) {
        try (PreparedStatement ps = conn2.prepareStatement("SELECT * FROM
8.  students")) { // 打开了真正的Connection
9.          try (ResultSet rs = ps.executeQuery()) {
10.             while (rs.next()) {
11.                 System.out.println(rs.getString("name"));
12.             }
13.         }
14.     }
15. }
```

现在我们来思考如何实现这个`LazyConnectionProxy`。为了简化代码，我们首先针对`Connection`接口做一个抽象的代理类：

```
1.  public abstract class AbstractConnectionProxy implements Connection {
2.
3.      // 抽象方法获取实际的Connection:
4.      protected abstract Connection getRealConnection();
5.
6.      // 实现Connection接口的每一个方法:
7.      public Statement createStatement() throws SQLException {
8.          return getRealConnection().createStatement();
9.      }
10.
11.     public PreparedStatement prepareStatement(String sql) throws SQLException {
12.         return getRealConnection().prepareStatement(sql);
```

```
    }

    ...其他代理方法...
}
```

这个 `AbstractConnectionProxy` 代理类的作用是把 `Connection` 接口定义的方法全部实现一遍，因为 `Connection` 接口定义的方法太多了，后面我们要编写的 `LazyConnectionProxy` 只需要继承 `AbstractConnectionProxy` ，就不必再把 `Connection` 接口方法挨个实现一遍。

`LazyConnectionProxy` 实现如下：

```
public class LazyConnectionProxy extends AbstractConnectionProxy {
    private Supplier<Connection> supplier;
    private Connection target = null;

    public LazyConnectionProxy(Supplier<Connection> supplier) {
        this.supplier = supplier;
    }

    // 覆写close方法：只有target不为null时才需要关闭:
    public void close() throws SQLException {
        if (target != null) {
            System.out.println("Close connection: " + target);
            super.close();
        }
    }

    @Override
    protected Connection getRealConnection() {
        if (target == null) {
            target = supplier.get();
        }
        return target;
    }
}
```

如果调用者没有执行任何SQL语句，那么 `target` 字段始终为 `null` 。只有第一次执行SQL语句时（即调用任何类似 `prepareStatement()` 方法时，触发 `getRealConnection()` 调用），才会真正打开实际的JDBC Connection。

最后，我们还需要编写一个 `LazyDataSource` 来支持这个 `LazyConnecitonProxy` ：

```
public class LazyDataSource implements DataSource {
    private String url;
    private String username;
    private String password;

    public LazyDataSource(String url, String username, String password) {
        this.url = url;
        this.username = username;
        this.password = password;
    }

    public Connection getConnection(String username, String password) throws
SQLException {
        return new LazyConnectionProxy(() -> {
            try {
                Connection conn = DriverManager.getConnection(url, username,
password);
                System.out.println("Open connection: " + conn);
                return conn;
            } catch (SQLException e) {
                throw new RuntimeException(e);
            }
        });
    }
    ...
}
```

我们执行代码，输出如下：

```
get lazy connection...
get lazy connection...
Open connection: com.mysql.jdbc.JDBC4Connection@7a36aefa
小明
小红
小军
小白
...
Close connection: com.mysql.jdbc.JDBC4Connection@7a36aefa
```

可见第一个 `getConnection()` 调用获取到的 `LazyConnectionProxy` 并没有实际打开真正的JDBC Connection。

使用连接池的时候，我们更希望能重复使用连接。如果调用方编写这样的代码：

```
    DataSource pooledDataSource = new PooledDataSource(jdbcUrl, jdbcUsername,
1.  jdbcPassword);
2.  try (Connection conn = pooledDataSource.getConnection()) {
3.  }
4.  try (Connection conn = pooledDataSource.getConnection()) {
5.      // 获取到的是同一个Connection
6.  }
7.  try (Connection conn = pooledDataSource.getConnection()) {
8.      // 获取到的是同一个Connection
9.  }
```

调用方并不关心是否复用了 `Connection` ，但从 `PooledDataSource` 获取的 `Connection` 确实自带这个优化功能。如何实现可复用 `Connection` 的连接池？答案仍然是使用代理模式。

```
1.  public class PooledConnectionProxy extends AbstractConnectionProxy {
2.      // 实际的Connection:
3.      Connection target;
4.      // 空闲队列:
5.      Queue<PooledConnectionProxy> idleQueue;
6.
        public PooledConnectionProxy(Queue<PooledConnectionProxy> idleQueue,
7.  Connection target) {
8.          this.idleQueue = idleQueue;
9.          this.target = target;
10.     }
11.
12.     public void close() throws SQLException {
            System.out.println("Fake close and released to idle queue for future
13. reuse: " + target);
14.         // 并没有调用实际Connection的close()方法,
15.         // 而是把自己放入空闲队列:
16.         idleQueue.offer(this);
17.     }
18.
19.     protected Connection getRealConnection() {
20.         return target;
21.     }
22. }
```

复用连接的关键在于覆写 `close()` 方法，它并没有真正关闭底层JDBC连接，而是把自己放回一个空

闲队列，以便下次使用。

空闲队列由 `PooledDataSource` 负责维护：

```
public class PooledDataSource implements DataSource {
    private String url;
    private String username;
    private String password;

    // 维护一个空闲队列:
    private Queue<PooledConnectionProxy> idleQueue = new ArrayBlockingQueue<>(100);

    public PooledDataSource(String url, String username, String password) {
        this.url = url;
        this.username = username;
        this.password = password;
    }

    public Connection getConnection(String username, String password) throws SQLException {
        // 首先试图获取一个空闲连接:
        PooledConnectionProxy conn = idleQueue.poll();
        if (conn == null) {
            // 没有空闲连接时，打开一个新连接:
            conn = openNewConnection();
        } else {
            System.out.println("Return pooled connection: " + conn.target);
        }
        return conn;
    }

    private PooledConnectionProxy openNewConnection() throws SQLException {
        Connection conn = DriverManager.getConnection(url, username, password);
        System.out.println("Open new connection: " + conn);
        return new PooledConnectionProxy(idleQueue, conn);
    }
    ...
}
```

我们执行调用方代码，输出如下：

```
Open new connection: com.mysql.jdbc.JDBC4Connection@61ca2dfa
Fake close and released to idle queue for future reuse:
com.mysql.jdbc.JDBC4Connection@61ca2dfa
Return pooled connection: com.mysql.jdbc.JDBC4Connection@61ca2dfa
Fake close and released to idle queue for future reuse:
com.mysql.jdbc.JDBC4Connection@61ca2dfa
Return pooled connection: com.mysql.jdbc.JDBC4Connection@61ca2dfa
Fake close and released to idle queue for future reuse:
com.mysql.jdbc.JDBC4Connection@61ca2dfa
```

除了第一次打开了一个真正的JDBC Connection，后续获取的 `Connection` 实际上是同一个JDBC Connection。但是，对于调用方来说，完全不需要知道底层做了哪些优化。

我们实际使用的DataSource，例如HikariCP，都是基于代理模式实现的，原理同上，但增加了更多的如动态伸缩的功能（一个连接空闲一段时间后自动关闭）。

有的童鞋会发现Proxy模式和Decorator模式有些类似。确实，这两者看起来很像，但区别在于：Decorator模式让调用者自己创建核心类，然后组合各种功能，而Proxy模式决不能让调用者自己创建再组合，否则就失去了代理的功能。Proxy模式让调用者认为获取到的是核心类接口，但实际上是代理类。

练习

从  下载练习：使用代理模式编写一个JDBC连接池 （推荐使用IDE练习插件快速下载）

小结

代理模式通过封装一个已有接口，并向调用方返回相同的接口类型，能让调用方在不改变任何代码的前提下增强某些功能（例如，鉴权、延迟加载、连接池复用等）。

使用Proxy模式要求调用方持有接口，作为Proxy的类也必须实现相同的接口类型。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





行为型模式

行为型模式主要涉及算法和对象间的职责分配。通过使用对象组合，行为型模式可以描述一组对象应该如何协作来完成一个整体任务。

行为型模式有：

- 责任链
- 命令
- 解释器
- 迭代器
- 中介
- 备忘录
- 观察者
- 状态
- 策略
- 模板方法
- 访问者

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：

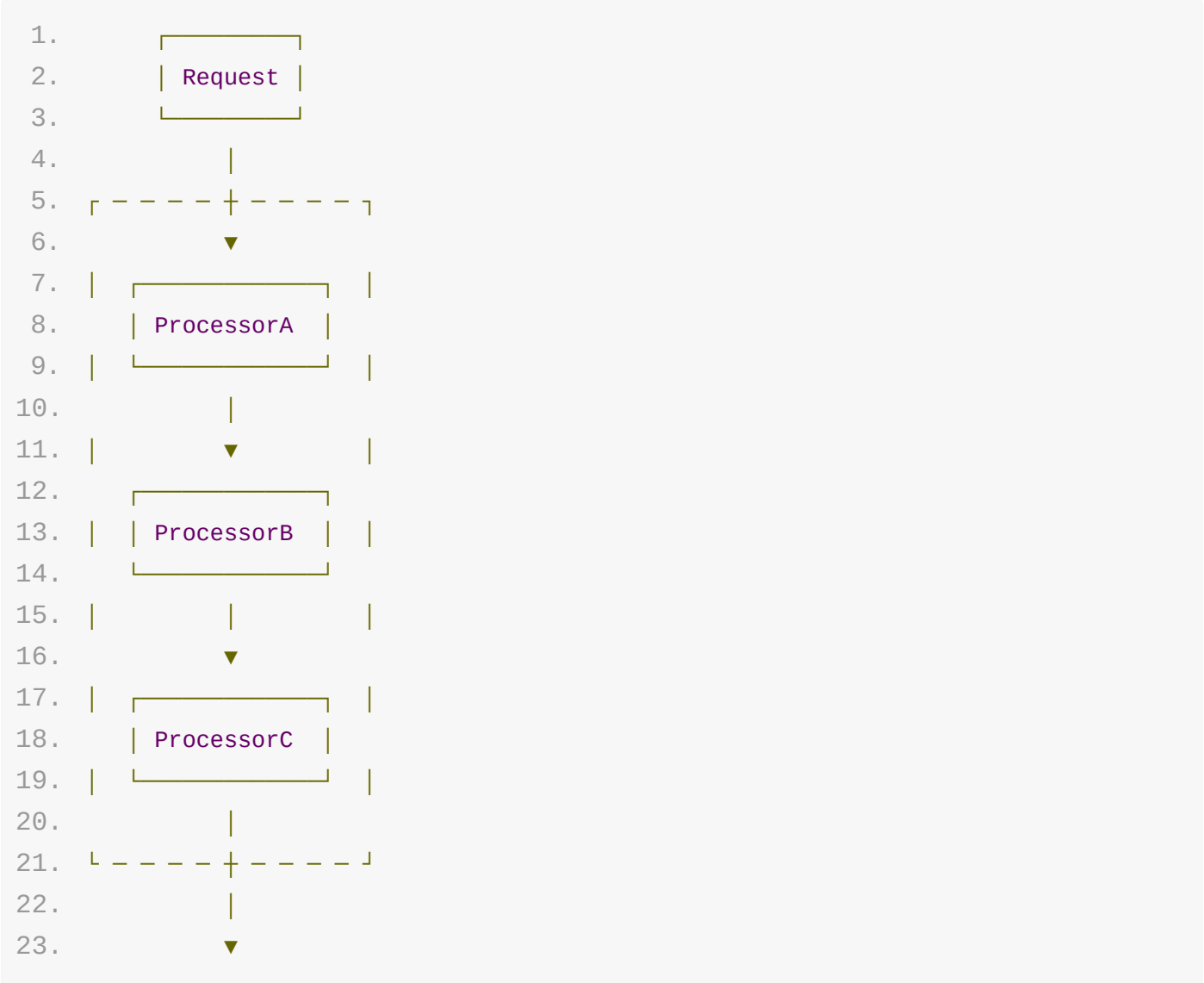




责任链

> 使多个对象都有机会处理请求，从而避免请求的发送者和接收者之间的耦合关系。将这些对象连成一条链，并沿着这条链传递该请求，直到有一个对象处理它为止。

责任链模式（Chain of Responsibility）是一种处理请求的模式，它让多个处理器都有机会处理该请求，直到其中某个处理成功为止。责任链模式把多个处理器串成链，然后让请求在链上传递：



在实际场景中，财务审批就是一个责任链模式。假设某个员工需要报销一笔费用，审核者可以分为：

- Manager：只能审核1000元以下的报销；
- Director：只能审核10000元以下的报销；
- CEO：可以审核任意额度。

用责任链模式设计此报销流程时，每个审核者只关心自己责任范围内的请求，并且处理它。对于超出自己责任范围的，扔给下一个审核者处理，这样，将来继续添加审核者的时候，不用改动现有逻辑。

我们来看看如何实现责任链模式。

首先，我们要抽象出请求对象，它将在责任链上传递：

```
public class Request {
    private String name;
    private BigDecimal amount;

    public Request(String name, BigDecimal amount) {
        this.name = name;
        this.amount = amount;
    }

    public String getName() {
        return name;
    }

    public BigDecimal getAmount() {
        return amount;
    }
}
```

其次，我们要抽象出处理器：

```
public interface Handler {
    // 返回Boolean.TRUE = 成功
    // 返回Boolean.FALSE = 拒绝
    // 返回null = 交下一个处理
    Boolean process(Request request);
}
```

并且做好约定：如果返回 `Boolean.TRUE` ，表示处理成功，如果返回 `Boolean.FALSE` ，表示处理失败（请求被拒绝），如果返回 `null` ，则交由下一个 `Handler` 处理。

然后，依次编写ManagerHandler、DirectorHandler和CEOHandler。以ManagerHandler为例：

```
public class ManagerHandler implements Handler {
    public Boolean process(Request request) {
        // 如果超过1000元，处理不了，交下一个处理:
        if (request.getAmount().compareTo(BigDecimal.valueOf(1000)) > 0) {
            return null;
```

```
        }
        // 对Bob有偏见:
        return !request.getName().equalsIgnoreCase("bob");
    }
}
```

有了不同的 `Handler` 后，我们还要把这些 `Handler` 组合起来，变成一个链，并通过一个统一入口处理：

```
public class HandlerChain {
    // 持有所有Handler:
    private List<Handler> handlers = new ArrayList<>();

    public void addHandler(Handler handler) {
        this.handlers.add(handler);
    }

    public boolean process(Request request) {
        // 依次调用每个Handler:
        for (Handler handler : handlers) {
            Boolean r = handler.process(request);
            if (r != null) {
                // 如果返回TRUE或FALSE，处理结束:
                System.out.println(request + " " + (r ? "Approved by " :
"Denied by ") + handler.getClass().getSimpleName());
                return r;
            }
        }
        throw new RuntimeException("Could not handle request: " + request);
    }
}
```

现在，我们就可以在客户端组装出责任链，然后用责任链来处理请求：

```
// 构造责任链:
HandlerChain chain = new HandlerChain();
chain.addHandler(new ManagerHandler());
chain.addHandler(new DirectorHandler());
chain.addHandler(new CEOHandler());
// 处理请求:
chain.process(new Request("Bob", new BigDecimal("123.45")));
chain.process(new Request("Alice", new BigDecimal("1234.56")));
```

```
chain.process(new Request("Bill", new BigDecimal("12345.67")));
chain.process(new Request("John", new BigDecimal("123456.78")));
```

责任链模式本身很容易理解，需要注意的是，`Handler` 添加的顺序很重要，如果顺序不对，处理的结果可能就不是符合要求的。

此外，责任链模式有很多变种。有些责任链的实现方式是通过某个 `Handler` 手动调用下一个 `Handler` 来传递 `Request` ，例如：

```
public class AHandler implements Handler {
    private Handler next;
    public void process(Request request) {
        if (!canProcess(request)) {
            // 手动交给下一个Handler处理:
            next.process(request);
        } else {
            ...
        }
    }
}
```

还有一些责任链模式，每个 `Handler` 都有机会处理 `Request` ，通常这种责任链被称为拦截器（Interceptor）或者过滤器（Filter），它的目的不是找到某个 `Handler` 处理掉 `Request` ，而是每个 `Handler` 都做一些工作，比如：

- 记录日志；
- 检查权限；
- 准备相关资源；
- …

例如，JavaEE的Servlet规范定义的 `Filter` 就是一种责任链模式，它不但允许每个 `Filter` 都有机会处理请求，还允许每个 `Filter` 决定是否将请求“放行”给下一个 `Filter` ：

```
public class AuditFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse resp, FilterChain
chain) throws IOException, ServletException {
        log(req);
        if (check(req)) {
            // 放行:
            chain.doFilter(req, resp);
        } else {
            // 拒绝:
```

```
            sendError(resp);
        }
    }
}
```

这种模式不但允许一个 `Filter` 自行决定处理 `ServletRequest` 和 `ServletResponse` ，还可以“伪造” `ServletRequest` 和 `ServletResponse` 以便让下一个 `Filter` 处理，能实现非常复杂的功能。

练习

从 gitee 下载练习：使用责任链模式实现审批 （推荐使用IDE练习插件快速下载）

小结

责任链模式是一种把多个处理器组合在一起，依次处理请求的模式；

责任链模式的好处是添加新的处理器或者重新排列处理器非常容易；

责任链模式经常用在拦截、预处理请求等。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：





命令

> 将一个请求封装为一个对象，从而使你可用不同的请求对客户进行参数化，对请求排队或记录请求日志，以及支持可撤销的操作。

命令模式（Command）是指，把请求封装成一个命令，然后执行该命令。

在使用命令模式前，我们先以一个编辑器为例子，看看如何实现简单的编辑操作：

```
1.  public class TextEditor {
2.      private StringBuilder buffer = new StringBuilder();
3.
4.      public void copy() {
5.          ...
6.      }
7.
8.      public void paste() {
9.          String text = getFromClipBoard();
10.         add(text);
11.     }
12.
13.     public void add(String s) {
14.         buffer.append(s);
15.     }
16.
17.     public void delete() {
18.         if (buffer.length() > 0) {
19.             buffer.deleteCharAt(buffer.length() - 1);
20.         }
21.     }
22.
23.     public String getState() {
24.         return buffer.toString();
25.     }
26. }
```

我们用一个 `StringBuilder` 模拟一个文本编辑器，它支持 `copy()` 、 `paste()` 、 `add()` 、 `delete()` 等方法。

正常情况，我们像这样调用 `TextEditor` ：

```
TextEditor editor = new TextEditor();
editor.add("Command pattern in text editor.\n");
editor.copy();
editor.paste();
System.out.println(editor.getState());
```

这是直接调用方法，调用方需要了解 `TextEditor` 的所有接口信息。

如果改用命令模式，我们就要把调用方发送命令和执行方执行命令分开。怎么分？

解决方案是引入一个 `Command` 接口：

```
public interface Command {
    void execute();
}
```

调用方创建一个对应的 `Command` ，然后执行，并不关心内部是如何具体执行的。

为了支持 `CopyCommand` 和 `PasteCommand` 这两个命令，我们从 `Command` 接口派生：

```
public class CopyCommand implements Command {
    // 持有执行者对象:
    private TextEditor receiver;

    public CopyCommand(TextEditor receiver) {
        this.receiver = receiver;
    }

    public void execute() {
        receiver.copy();
    }
}

public class PasteCommand implements Command {
    private TextEditor receiver;

    public PasteCommand(TextEditor receiver) {
        this.receiver = receiver;
    }

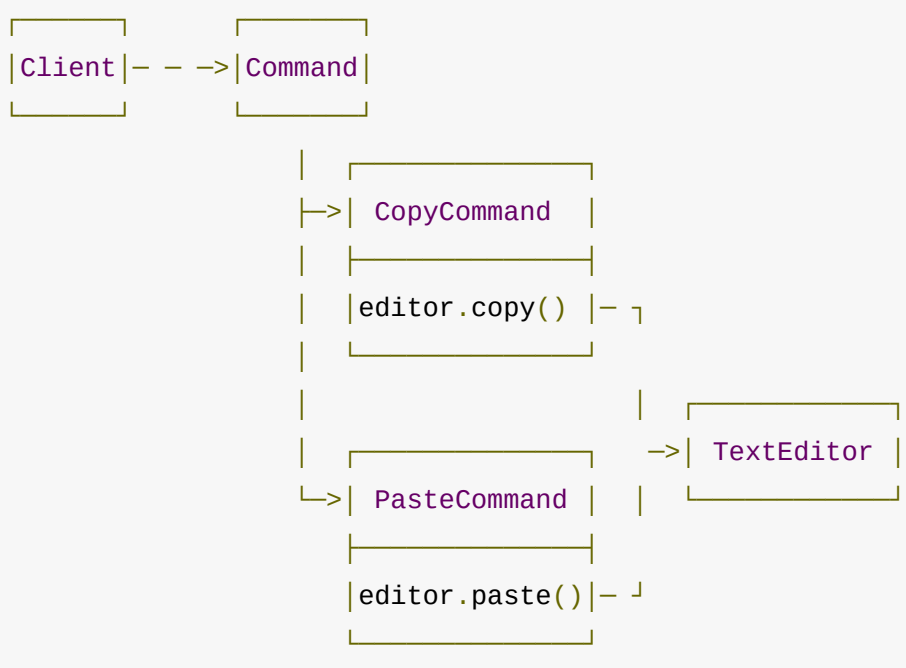
    public void execute() {
        receiver.paste();
```

```
    }
}
```

最后我们把 `Command` 和 `TextEditor` 组装一下，客户端这么写：

```
TextEditor editor = new TextEditor();
editor.add("Command pattern in text editor.\n");
// 执行一个CopyCommand:
Command copy = new CopyCommand(editor);
copy.execute();
editor.add("----\n");
// 执行一个PasteCommand:
Command paste = new PasteCommand(editor);
paste.execute();
System.out.println(editor.getState());
```

这就是命令模式的结构：



有的童鞋会有疑问：搞了一大堆 `Command` ，多了好几个类，还不如直接这么写简单：

```
TextEditor editor = new TextEditor();
editor.add("Command pattern in text editor.\n");
editor.copy();
editor.paste();
```

实际上，使用命令模式，确实增加了系统的复杂度。如果需求很简单，那么直接调用显然更直观而且更

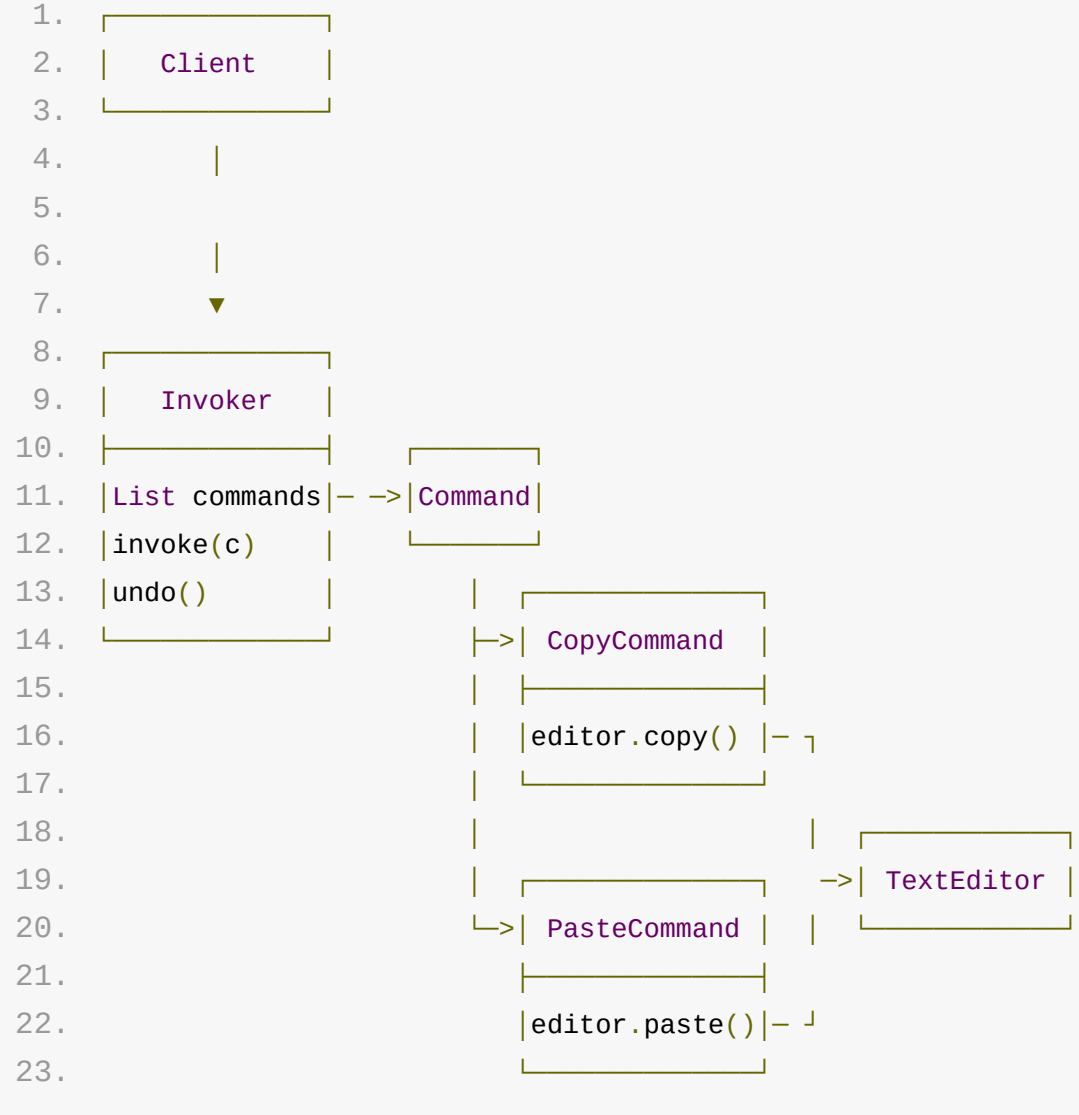
简单。

那么我们还需要命令模式吗？

答案是视需求而定。如果 `TextEditor` 复杂到一定程度，并且需要支持Undo、Redo的功能时，就需要使用命令模式，因为我们可以给每个命令增加 `undo()` ：

```
public interface Command {
    void execute();
    void undo();
}
```

然后把执行的一系列命令用 `List` 保存起来，就既能支持Undo，又能支持Redo。这个时候，我们又需要一个 `Invoker` 对象，负责执行命令并保存历史命令：



可见，模式带来的设计复杂度的增加是随着需求而增加的，它减少的是系统各组件的耦合度。

练习

给命令模式新增Add和Delete命令并支持Undo、Redo操作。

从 gitee 下载练习：命令模式练习 （推荐使用IDE练习插件快速下载）

小结

命令模式的设计思想是把命令的创建和执行分离，使得调用者无需关心具体的执行过程。

通过封装 `Command` 对象，命令模式可以保存已执行的命令，从而支持撤销、重做等操作。

读后有收获可以支付宝请作者喝咖啡，读后有疑问请加微信群讨论：



