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 17!

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！

关于作者

廖雪峰，十年软件开发经验，业余产品经理，精通Java/Python/Ruby/Scheme/Objective C等，对开源框架有深入研究，著有《Spring 2.0核心技术与最佳实践》一书，多个业余开源项目托管在GitHub，欢迎微博交流：

使用窄屏手机的童鞋，请点击左上角“目录”查看教程：

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。

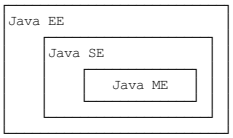
Java最早是由SUN公司（已被Oracle收购）的詹姆斯·高斯林（高司令，人称Java之父）在上个世纪90年代初开发的一种编程语言，最初被命名为Oak，目标是针对小型家电设备的嵌入式应用，结果市场没啥反响。谁料到互联网的崛起，让Oak重新焕发了生机，于是SUN公司改造了Oak，在1995年以Java的名称正式发布，原因是Oak已经被人注册了，因此SUN注册了Java这个商标。随着互联网的高速发展，Java逐渐成为最重要的网络编程语言。

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：

1. 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；
2. 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；
3. 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；
4. 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 17：

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
2020/3	14
2020/9	15
2021/3	16
2021/9	17

本教程使用的Java版本是最新版的**Java 17**。

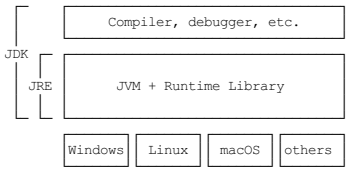
名词解释

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

```
请问Java之父是：
----
    James Bond
[x] James Gosling
    James Simons
```

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 17，确保从Oracle的官网下载最新的稳定版JDK：

```
Java SE 17
Java SE 17 is the latest release for the Java SE Platform

Documentation                Oracle JDK
Installation Instructions      JDK Download
Release Notes                  ═══════════
Oracle License                 Documentation Download
```

找到Java SE 17的下载链接`JDK Download`，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-17
```

在Mac下，它在~/.bash_profile或~/.zprofile里，它是：

```
export JAVA_HOME=`/usr/libexec/java_home -v 17`
```

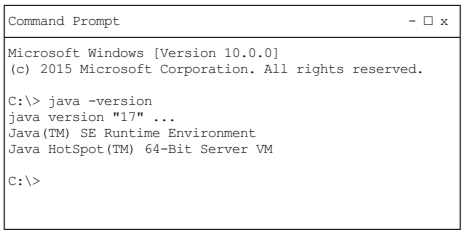
然后，把JAVA_HOME的bin目录附加到系统环境变量PATH上。在Windows下，它长这样：

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

在Mac下，它在~/.bash_profile或~/.zprofile里，长这样：

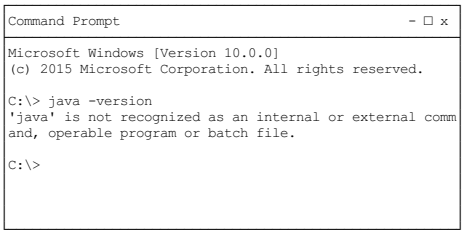
```
export PATH=$JAVA_HOME/bin:$PATH
```

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：



如果你看到的版本号不是17，而是15、1.8之类，说明系统存在多个JDK，且默认JDK不是JDK 17，需要把JDK 17提到PATH前面。

如果你得到一个错误输出：



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。

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。

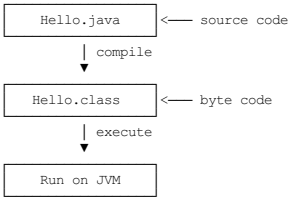
Java规定，某个类定义的public static void main(String[] args)是Java程序的固定入口方法，因此，Java程序总是从main方法开始执行。

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：



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。

Java代码运行助手可以让你在线输入Java代码，然后远程运行后，在网页显示代码执行结果：

试试效果

需要支持HTML5的浏览器：

- IE >= 9
- Edge
- Firefox
- Chrome
- Sarafi

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

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

使用IDE的好处在于，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16，本教程的所有代码均使用Java 16的语法，并且编译到Java 16的版本。

去掉“Use default compliance settings”并钩上“Enable preview features for Java 16”，这样我们就可以使用Java 16的预览功能。

如果Compiler compliance level没有16这个选项，请更新到最新版Eclipse。如果更新后还是没有16，打开Help - Eclipse Marketplace，搜索Java 16 Support安装后重启即可。

Java > Installed JREs

在Installed JREs中应该看到Java SE 16，如果还有其他的JRE，可以删除，以确保Java SE 16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 16：

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。

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

点击下载IDE练习插件，保存到本地硬盘。

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

点击“Add”，选择“Archive...”，从本地选择下载的`java-practice-update-site.zip`文件，对Name填写一个任意的名称，例如“Java Practice”，然后点击“Add”添加：

去掉勾选“Group items by category”，选择列表中的“LearnJava”插件，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？

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

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。

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
```

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：

```
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
    }
}
```

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
    }
}
```

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;。

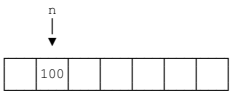
变量不但可以重新赋值，还可以赋值给其他变量。让我们来看一个例子：

```
// 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
```

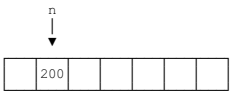
```
        x = x + 100; // 变量x赋值为x+100（x的值为200，因此赋值后x的值是200+100=300）
        System.out.println("x = " + x); // 打印x的值
        System.out.println("n = " + n); // 再次打印n的值，n应该是200还是300?
    }
}
```

我们一行一行地分析代码执行流程：

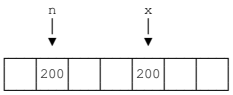
执行`int n = 100;`，该语句定义了变量`n`，同时赋值为`100`，因此，JVM在内存中为变量`n`分配一个“存储单元”，填入值`100`：



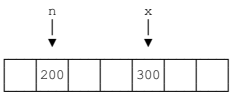
执行`n = 200;`时，JVM把`200`写入变量`n`的存储单元，因此，原有的值被覆盖，现在`n`的值为`200`：



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`：



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`：



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

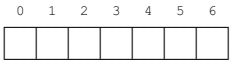
基本数据类型是CPU可以直接进行运算的类型。Java定义了以下几种基本数据类型：

- 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`。

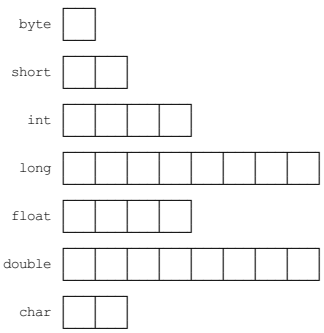
内存单元从0开始编号，称为内存地址。每个内存单元可以看作一间房间，内存地址就是门牌号。



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`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
    }
}
```

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);
    }
}
```

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。

引用类型

除了上述基本类型的变量，剩下的都是引用类型。例如，引用类型最常用的就是`String`字符串：

```
String s = "hello";
```

引用类型的变量类似于C语言的指针，它内部存储一个“地址”，指向某个对象在内存的位置，后续我们介绍类的概念时会详细讨论。

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。

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);
    }
}
```

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：

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

求余运算使用%：

```
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
    }
}
```

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);
    }
}
```

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;  // 11111000 00000000 00000000 00000000 = -134217728
int c = n >> 28; // 11111111 11111111 11111111 11111110 = -2
int d = n >> 29; // 11111111 11111111 11111111 11111111 = -1
```

还有一种无符号的右移运算，使用>>>，它的特点是不管符号位，右移后高位总是补0，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：

```
n = 0 ^ 0; // 0
n = 0 ^ 1; // 1
n = 1 ^ 0; // 1
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
    }
}
```

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!
    }
}
```

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：

```
int i = 12345;
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);
    }
}
```

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式：

```
\frac{(1+N)\times N}2
```

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");
    }
}
```

计算前N个自然数的和

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。

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);
    }
}
```

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);
    }
}
```

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：

```
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：

```
double d1 = 0.0 / 0; // NaN
double d2 = 1.0 / 0; // Infinity
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);
    }
}
```

练习

根据一元二次方程$ax^2+bx+c=0$的求根公式：

```
\frac{\displaystyle-b\pm\sqrt{b^2-4ac}}{\displaystyle2a}
```

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");
    }
}
```

计算一元二次方程的两个解

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差的绝对值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。

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);
    }
}
```

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：

```
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);
    }
}
```

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");
----
    }
}
```

判断指定年龄是否是小学生

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。

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：

```
char c1 = 'A';
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：

```
int n1 = 'A'; // 字母“A”的Unicodde编码是65
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：

```
// 注意是十六进制:
char c3 = '\u0041'; // 'A'，因为十六进制0041 = 十进制65
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：

```
String s = ""; // 空字符串，包含0个字符
String s1 = "A"; // 包含一个字符
String s2 = "ABC"; // 包含3个字符
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\：

```
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);
    }
}
```

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);
    }
}
```

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);
    }
}
```

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。

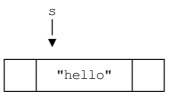
不可变特性

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：

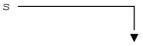
```
// 字符串不可变
----
public class Main {
    public static void main(String[] args) {
        String s = "hello";
        System.out.println(s); // 显示 hello
        s = "world";
        System.out.println(s); // 显示 world
    }
}
```

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：



	"hello"		"world"	

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"?
    }
}
```

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);
---
    }
}
```

Unicode值拼接字符串

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""。

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;
    }
}
```

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
    }
}
```

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
    }
}
```

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
    }
}
```

还可以进一步简写为：

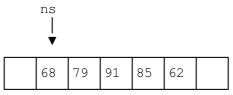
```
int[] ns = { 68, 79, 91, 85, 62 };
```

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：

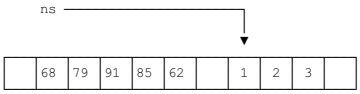
```
// 数组
----
public class Main {
    public static void main(String[] args) {
        // 5位同学的成绩:
        int[] ns;
        ns = new int[] { 68, 79, 91, 85, 62 };
        System.out.println(ns.length); // 5
        ns = new int[] { 1, 2, 3 };
        System.out.println(ns.length); // 3
    }
}
```

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。

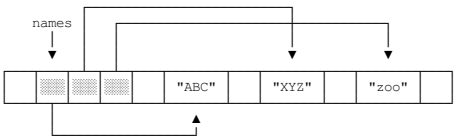
字符串数组

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？

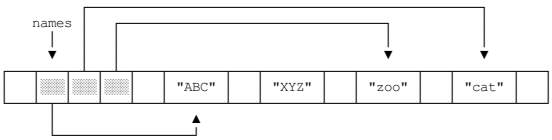
字符串是引用类型，因此我们先定义一个字符串数组：

```
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"?
    }
}
```

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；

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。

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");
    }
}
```

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
    }
}
```

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
    }
}
```

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
    }
}
```

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：

```
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%）。

输入和输出练习

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()` / ...

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

串联的效果其实相当于：

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

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");
        }
    }
}
```

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");
        }
    }
}
```

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");
        }
    }
}
```

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");
        }
    }
}
```

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");
        }
    }
}
```

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");
        }
    }
}
```

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");
        }
    }
}
```

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

BMI练习

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。

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：

1. 单人模式
2. 多人模式
3. 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;
        }
    }
}
```

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;
        }
    }
}
```

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");
        }
    }
}
```

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;
        }
    }
}
```

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

```
switch (option) {
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;
        }
    }
}
```

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");
        }
    }
}
```

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);
    }
}
```

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);
    }
}
```

练习

使用switch实现一个简单的石头、剪子、布游戏。

switch练习

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 14开始，switch语句正式升级为表达式，不再需要break，并且允许使用yield返回值。

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 + 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：

```
while (条件表达式) {
    循环语句
}
// 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
    }
}
```

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);
    }
}
```

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);
    }
}
```

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);
 }
}
```

while练习

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。

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);
    }
}
```

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);
 }
}
```

do while练习

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。

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);
    }
}
```

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);
    }
}
```

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;
        }
    }
}
```

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：

```
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);
        }
    }
}
```

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]);
        }
    }
}
```

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
    }
}
```

练习3

圆周率π可以使用公式计算：

```
\frac{\mathrm\pi}4=1-\frac13+\frac15-\frac17+\frac19-\dots
```

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);
    }
}
```

for循环计算π练习

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；

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);
    }
}
```

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");
        }
    }
}
```

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
    }
}
```

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。

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。

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);
        }
    }
}
```

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);
        }
    }
}
```

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));
    }
}
```

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(???);
        }
    }
}
```

倒序遍历数组练习

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。

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));
    }
}
```

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
```

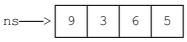
实际上，Java的标准库已经内置了排序功能，我们只需要调用JDK提供的`Arrays.sort()`就可以排序：

```
// 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));
    }
}
```

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：



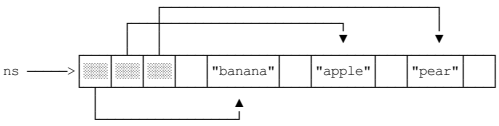
当我们调用`Arrays.sort(ns);`后，这个整型数组在内存中变为：

即变量`ns`指向的数组内容已经被改变了。

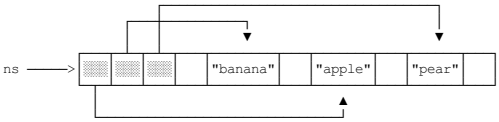
如果对一个字符串数组进行排序，例如：

```
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");
        }
    }
}
```

降序排序练习

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；

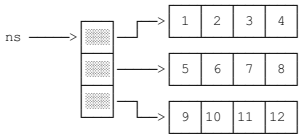
对数组排序会直接修改数组本身。

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
    }
}
```

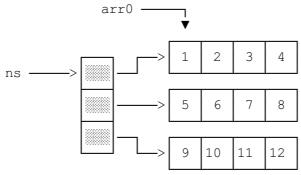
因为`ns`包含3个数组，因此，`ns.length`为`3`。实际上`ns`在内存中的结构如下：



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
    }
}
```

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：



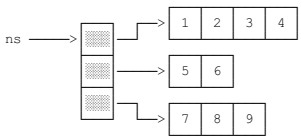
访问二维数组的某个元素需要使用`array[row][col]`，例如：

```
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 }
```

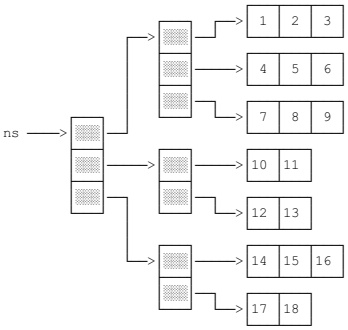
```
        };
        System.out.println(Arrays.deepToString(ns));
    }
}
```

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");
        }
    }
}
```

计算平均分

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]`。

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。

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：

1. 读取文件；
2. 编写TODO；
3. 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：

```
GirlFriend gf = new GirlFriend();
gf.name = "Alice";
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的基本思想，但不保证能找到对象。

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();
```

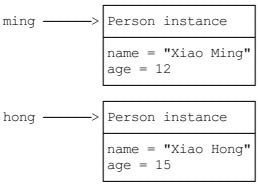
上述代码创建了一个Person类型的实例，并通过变量ming指向它。

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：



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。

一个Java源文件可以包含多个类的定义，但只能定义一个public类，且public类名必须与文件名一致。如果要定义多个public类，必须拆到多个Java源文件中。

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
----
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 {
    ???
}
```

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。

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;
}
```

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;
    }
}
```

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;
    }
}
```

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;
    }
}
```

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;
    }
}
```

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;
    }
}
```

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;
    }
```

```
}
```

给Person类增加getAge/setAge方法

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。

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：

```
Person ming = new Person();
ming.setName("小明");
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;
    }
}
```

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：

```
class Person {
    public Person() {
    }
}
```

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;
    }
}
```

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;
    }
}
```

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：

1. 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`；
2. 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;
    }
}
```

给Person类增加(String, int)的构造方法

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。

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);
    }
}
```

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;
    }
}
```

给Person增加重载方法

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。

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) { … }
}
```

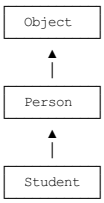
可见，通过继承，Student只需要编写额外的功能，不再需要重复代码。

注意：子类自动获得了父类的所有字段，严禁定义与父类重名的字段！

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）。

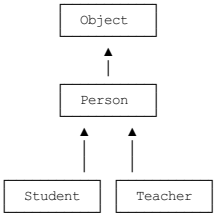
继承树

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：



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：



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;
    }
}
```

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。

阻止继承

正常情况下，只要某个class没有final修饰符，那么任何类都可以从该class继承。

从Java 15开始，允许使用sealed修饰class，并通过permits明确写出能够从该class继承的子类名称。

例如，定义一个Shape类：

```
public sealed class Shape permits Rect, Circle, Triangle {
    ...
}
```

上述Shape类就是一个sealed类，它只允许指定的3个类继承它。如果写：

```
public final class Rect extends Shape {...}
```

是没问题的，因为Rect出现在Shape的permits列表中。但是，如果定义一个Ellipse就会报错：

```
public final class Ellipse extends Shape {...}
// Compile error: class is not allowed to extend sealed class: Shape
```

原因是Ellipse并未出现在Shape的permits列表中。这种sealed类主要用于一些框架，防止继承被滥用。

sealed类在Java 15中目前是预览状态，要启用它，必须使用参数--enable-preview和--source 15。

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
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：

```
Person p1 = new Student(); // upcasting, ok
Person p2 = new Person();
Student s1 = (Student) p1; // ok
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：

```
Person p = new Person();
System.out.println(p instanceof Person); // true
System.out.println(p instanceof Student); // false

Student s = new Student();
System.out.println(s instanceof Person); // true
System.out.println(s instanceof Student); // true

Student n = null;
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
}
```

从Java 14开始，判断instanceof后，可以直接转型为指定变量，避免再次强制转型。例如，对于以下代码：

```
Object obj = "hello";
if (obj instanceof String) {
    String s = (String) obj;
    System.out.println(s.toUpperCase());
}
```

可以改写如下：

```
// instanceof variable:
----
public class Main {
    public static void main(String[] args) {
        Object obj = "hello";
        if (obj instanceof String s) {
            // 可以直接使用变量s:
            System.out.println(s.toUpperCase());
        }
    }
}
```

这种使用instanceof的写法更加简洁。

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
}
```

继承练习

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。

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) {}
}
```

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：

```
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");
    }
}
```

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：

```
Person p = new Student();
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：

```
public void runTwice(Person p) {
    p.run();
    p.run();
}
```

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：

```
class Income {
    protected double income;
    public double getTax() {
        return income * 0.1; // 税率10%
    }
}
```

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;
    }
}
```

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。

计算所得税

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。

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");
    }
}
```

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。

用抽象类算税

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。

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

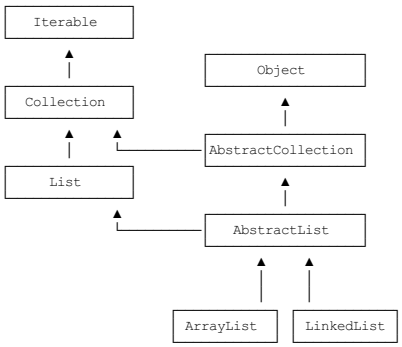
一个interface可以继承自另一个interface。interface继承自interface使用extends，它相当于扩展了接口的方法。例如：

```
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;
    }
}
```

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。

用接口算税

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）。

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。

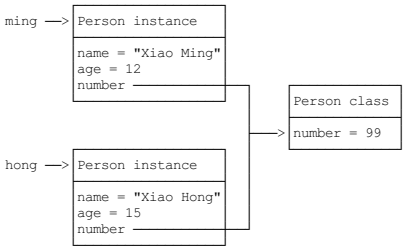
实例字段在每个实例中都有自己的一个独立“空间”，但是静态字段只有一个共享“空间”，所有实例都会共享该字段。举个例子：

```
class Person {
    public String name;
    public int age;
    // 定义静态字段number:
    public static int number;
}
```

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;
    }
}
```

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;
    }
}
```

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：

```
public interface Person {
    public static final int MALE = 1;
    public static final int FEMALE = 2;
}
```

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：

```
public interface Person {
    // 编译器会自动加上public statc final:
    int MALE = 1;
    int FEMALE = 2;
}
```

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。

静态字段和静态方法

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。

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：

```
package ming; // 申明包名ming

public class Person {
}
```

小军的Arrays.java文件：

```
package mr.jun; // 申明包名mr.jun

public class Arrays {
}
```

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：

```
// Main.java
package test;

import java.text.Format;

public class Main {
    public static void main(String[] args) {
        java.util.List list; // ok，使用完整类名 -> java.util.List
        Format format = null; // ok，使用import的类 -> java.text.Format
        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 -> java.lang.System
        MessageFormat mf = null; // 编译错误：无法找到MessageFormat: MessageFormat cannot be resolved to a type
    }
}
```

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
```

Package结构

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`。

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：

```
package abc;

public class Hello {
    public void hi() {
    }
}
```

上面的`Hello`是`public`，因此，可以被其他包的类访问：

```
package xyz;

class Main {
    void foo() {
        // Main可以访问Hello
        Hello h = new Hello();
    }
}
```

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：

```
package abc;

public class Hello {
    public void hi() {
    }
}
```

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：

```
package xyz;

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();
        }
    }
}
```

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：

```
package abc;

public class Hello {
    // protected方法:
    protected void hi() {
    }
}
```

上面的protected方法可以被继承的类访问：

```
package xyz;

class Main extends Hello {
    void foo() {
        // 可以访问protected方法:
        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。

在Java程序中，通常情况下，我们把不同的类组织在不同的包下面，对于一个包下面的类来说，它们是在同一层次，没有父子关系：

```
java.lang
├── Math
├── Runnable
├── String
└── ...
```

还有一种类，它被定义在另一个类的内部，所以称为内部类（Nested Class）。Java的内部类分为好几种，通常情况用得不多，但也需要了解它们是如何使用的。

Inner Class

如果一个类定义在另一个类的内部，这个类就是Inner Class：

```
class Outer {
    class Inner {
        // 定义了一个Inner Class
    }
}
```

上述定义的Outer是一个普通类，而Inner是一个Inner Class，它与普通类有个最大的不同，就是Inner Class的实例不能单独存在，必须依附于一个Outer Class的实例。示例代码如下：

```
// inner class
----
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer("Nested"); // 实例化一个Outer
        Outer.Inner inner = outer.new Inner(); // 实例化一个Inner
        inner.hello();
    }
}

class Outer {
    private String name;

    Outer(String name) {
        this.name = name;
    }

    class Inner {
        void hello() {
```

```
            System.out.println("Hello, " + Outer.this.name);
        }
    }
}
```

观察上述代码，要实例化一个Inner，我们必须首先创建一个Outer的实例，然后，调用Outer实例的new来创建Inner实例：

```
Outer.Inner inner = outer.new Inner();
```

这是因为Inner Class除了有一个this指向它自己，还隐含地持有一个Outer Class实例，可以用Outer.this访问这个实例。所以，实例化一个Inner Class不能脱离Outer实例。

Inner Class和普通Class相比，除了能引用Outer实例外，还有一个额外的“特权”，就是可以修改Outer Class的private字段，因为Inner Class的作用域在Outer Class内部，所以能访问Outer Class的private字段和方法。

观察Java编译器编译后的.class文件可以发现，Outer类被编译为Outer.class，而Inner类被编译为Outer$Inner.class。

Anonymous Class

还有一种定义Inner Class的方法，它不需要在Outer Class中明确地定义这个Class，而是在方法内部，通过匿名类（Anonymous Class）来定义。示例代码如下：

```
// Anonymous Class
----
public class Main {
    public static void main(String[] args) {
        Outer outer = new Outer("Nested");
        outer.asyncHello();
    }
}

class Outer {
    private String name;

    Outer(String name) {
        this.name = name;
    }

    void asyncHello() {
        Runnable r = new Runnable() {
            @Override
            public void run() {
                System.out.println("Hello, " + Outer.this.name);
            }
        };
        new Thread(r).start();
    }
}
```

观察asyncHello()方法，我们在方法内部实例化了一个Runnable。Runnable本身是接口，接口是不能实例化的，所以这里实际上是定义了一个实现了Runnable接口的匿名类，并且通过new实例化该匿名类，然后转型为Runnable。在定义匿名类的时候就必须实例化它，定义匿名类的写法如下：

```
Runnable r = new Runnable() {
    // 实现必要的抽象方法...
};
```

匿名类和Inner Class一样，可以访问Outer Class的private字段和方法。之所以我们要定义匿名类，是因为在这里我们通常不关心类名，比直接定义Inner Class可以少写很多代码。

观察Java编译器编译后的.class文件可以发现，Outer类被编译为Outer.class，而匿名类被编译为Outer$1.class。如果有多个匿名类，Java编译器会将每个匿名类依次命名为Outer$1、Outer$2、Outer$3……

除了接口外，匿名类也完全可以继承自普通类。观察以下代码：

```
// Anonymous Class
----
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, String> map1 = new HashMap<>();
        HashMap<String, String> map2 = new HashMap<>() {}; // 匿名类!
        HashMap<String, String> map3 = new HashMap<>() {
            {
                put("A", "1");
                put("B", "2");
            }
        };
        System.out.println(map3.get("A"));
    }
}
```

map1是一个普通的HashMap实例，但map2是一个匿名类实例，只是该匿名类继承自HashMap。map3也是一个继承自HashMap的匿名类实例，并且添加了static代码块来初始化数据。观察编译输出可发现Main$1.class和Main$2.class两个匿名类文件。

Static Nested Class

最后一种内部类和Inner Class类似，但是使用static修饰，称为静态内部类（Static Nested Class）：

```
// Static Nested Class
----
public class Main {
    public static void main(String[] args) {
        Outer.StaticNested sn = new Outer.StaticNested();
        sn.hello();
    }
}

class Outer {
    private static String NAME = "OUTER";

    private String name;

    Outer(String name) {
        this.name = name;
    }

    static class StaticNested {
        void hello() {
            System.out.println("Hello, " + Outer.NAME);
        }
    }
}
```

用static修饰的内部类和Inner Class有很大的不同，它不再依附于Outer的实例，而是一个完全独立的类，因此无法引用Outer.this，但它可以访问Outer的private静态字段和静态方法。如果把StaticNested移到Outer之外，就失去了访问private的权限。

小结

Java的内部类可分为Inner Class、Anonymous Class和Static Nested Class三种：

- Inner Class和Anonymous Class本质上是相同的，都必须依附于Outer Class的实例，即隐含地持有`Outer.this`实例，并拥有Outer Class的`private`访问权限；
- Static Nested Class是独立类，但拥有Outer Class的`private`访问权限。

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：

```
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：

```
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。

假设我们有一个编译后的`Hello.class`，它的包名是`com.example`，当前目录是`C:\work`，那么，目录结构必须如下：

```
C:\work
└─ com
   └─ example
      └─ Hello.class
```

运行这个`Hello.class`必须在当前目录下使用如下命令：

```
C:\work> java -cp . com.example.Hello
```

JVM根据classpath设置的`.`在当前目录下查找`com.example.Hello`，即实际搜索文件必须位于`com/example/Hello.class`。如果指定的`.class`文件不存在，或者目录结构和包名对不上，均会报错。

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：

```
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：

```
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。

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：

```
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。

[打包模块和JRE]

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。

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

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：

```
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：

```
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);
    }
}
```

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));
    }
}
```

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));
    }
}
```

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
"Hello".startsWith("He"); // true
"Hello".endsWith("lo"); // true
```

提取子串的例子：

```
"Hello".substring(2); // "llo"
"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：

```
"  \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：

```
"\u3000Hello\u3000".strip(); // "Hello"
" Hello ".stripLeading(); // "Hello "
" 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：

```
"".isEmpty(); // true，因为字符串长度为0
"  ".isEmpty(); // false，因为字符串长度不为0
"  \n".isBlank(); // true，因为只包含空白字符
" 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：

```
String s = "hello";
s.replace('l', 'w'); // "hewwo"，所有字符'l'被替换为'w'
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"
```

格式化字符串

字符串提供了`formatted()`方法和`format()`静态方法，可以传入其他参数，替换占位符，然后生成新的字符串：

```
// String
----
public class Main {
    public static void main(String[] args) {
        String s = "Hi %s, your score is %d!";
        System.out.println(s.formatted("Alice", 80));
        System.out.println(String.format("Hi %s, your score is %.2f!", "Bob", 59.5));
    }
}
```

有几个占位符，后面就传入几个参数。参数类型要和占位符一致。我们经常用这个方法来格式化信息。常用的占位符有：

- `%s`：显示字符串；
- `%d`：显示整数；
- `%x`：显示十六进制整数；
- `%f`：显示浮点数。

占位符还可以带格式，例如`%.2f`表示显示两位小数。如果你不确定用啥占位符，那就始终用`%s`，因为`%s`可以显示任何数据类型。要查看完整的格式化语法，请参考JDK文档。

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`：

```
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：

```
char[] cs = "Hello".toCharArray(); // String -> char[]
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);
    }
}
```

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));
    }
}
```

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。

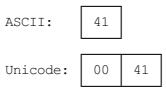
如果要把汉字也纳入计算机编码，很显然一个字节是不够的。GB2312标准使用两个字节表示一个汉字，其中第一个字节的最高位始终为1，以便和ASCII编码区分开。例如，汉字'中'的GB2312编码是0xd6d0。

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。

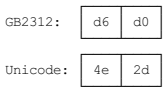
Unicode编码需要两个或者更多字节表示，我们可以比较中英文字符在ASCII、GB2312和Unicode的编码：

英文字符'A'的ASCII编码和Unicode编码：



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：



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(); // 按系统默认编码转换，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。

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());
    }
}
```

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;
    }
}
```

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 "";
    }
----
}
```

StringBuilder练习

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。

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());
    }
}
```

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());
    }
}
```

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());
    }
}
```

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 "";
    }
---
}
```

StringJoiner练习

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”。

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

基本类型	对应的引用类型
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());
    }
}
```

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;
    }
}
```

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：

```
public final class Integer {
    private final int value;
}
```

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
    }
}
```

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
    }
}
```

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
    }
}
```

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。

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：

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

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;
    }
}
```

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。

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;
}
```

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;
    }
}
```

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;
    }
}
```

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;
}
```

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。

使用String、Integer等类型的时候，这些类型都是不变类，一个不变类具有以下特点：

1. 定义class时使用final，无法派生子类；
2. 每个字段使用final，保证创建实例后无法修改任何字段。

假设我们希望定义一个Point类，有x、y两个变量，同时它是一个不变类，可以这么写：

```
public final class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int x() {
        return this.x;
    }

    public int y() {
        return this.y;
    }
}
```

为了保证不变类的比较，还需要正确覆写equals()和hashCode()方法，这样才能在集合类中正常使用。后续我们会详细讲解正确覆写equals()和hashCode()，这里演示Point不变类的写法目的是，这些代码写起来都非常简单，但是很繁琐。

record

从Java 14开始，引入了新的Record类。我们定义Record类时，使用关键字record。把上述Point类改写为Record类，代码如下：

```
// Record
----
public class Main {
    public static void main(String[] args) {
        Point p = new Point(123, 456);
        System.out.println(p.x());
        System.out.println(p.y());
        System.out.println(p);
    }
}

public record Point(int x, int y) {}
```

仔细观察Point的定义：

```
public record Point(int x, int y) {}
```

把上述定义改写为class，相当于以下代码：

```
public final class Point extends Record {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    public int x() {
        return this.x;
    }

    public int y() {
        return this.y;
    }

    public String toString() {
        return String.format("Point[x=%s, y=%s]", x, y);
    }

    public boolean equals(Object o) {
        ...
    }
    public int hashCode() {
        ...
    }
}
```

除了用final修饰class以及每个字段外，编译器还自动为我们创建了构造方法，和字段名同名的方法，以及覆写toString()、equals()和hashCode()方法。

换句话说，使用record关键字，可以一行写出一个不变类。

和enum类似，我们自己不能直接从Record派生，只能通过record关键字由编译器实现继承。

构造方法

编译器默认按照record声明的变量顺序自动创建一个构造方法，并在方法内给字段赋值。那么问题来了，如果我们要检查参数，应该怎么办？

假设Point类的x、y不允许负数，我们就得给Point的构造方法加上检查逻辑：

```
public record Point(int x, int y) {
```

```
    public Point {
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException();
        }
    }
}
```

注意到方法`public Point {...}`被称为Compact Constructor，它的目的是让我们编写检查逻辑，编译器最终生成的构造方法如下：

```
public final class Point extends Record {
    public Point(int x, int y) {
        // 这是我们编写的Compact Constructor:
        if (x < 0 || y < 0) {
            throw new IllegalArgumentException();
        }
        // 这是编译器继续生成的赋值代码：
        this.x = x;
        this.y = y;
    }
    ...
}
```

作为`record`的`Point`仍然可以添加静态方法。一种常用的静态方法是`of()`方法，用来创建`Point`：

```
public record Point(int x, int y) {
    public static Point of() {
        return new Point(0, 0);
    }
    public static Point of(int x, int y) {
        return new Point(x, y);
    }
}
```

这样我们可以写出更简洁的代码：

```
var z = Point.of();
var p = Point.of(123, 456);
```

小结

从Java 14开始，提供新的`record`关键字，可以非常方便地定义Data Class：

- 使用`record`定义的是不变类；
- 可以编写Compact Constructor对参数进行验证；
- 可以定义静态方法。

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)); // 2867971860299718107233761438093672048294900000
```

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（3.4×10^{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);
    }
}
```

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。

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);
    }
}
```

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：

```
BigDecimal d1 = new BigDecimal("123.456");
BigDecimal d2 = new BigDecimal("23.456789");
BigDecimal d3 = d1.divide(d2, 10, RoundingMode.HALF_UP); // 保留10位小数并四舍五入
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
    }
}
```

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：

```
BigDecimal n = new BigDecimal("12.75");
BigDecimal m = new BigDecimal("0.15");
BigDecimal[] dr = n.divideAndRemainder(m);
if (dr[1].signum() == 0) {
    // 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()。

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：

```
Math.abs(-100); // 100
Math.abs(-7.8); // 7.8
```

取最大或最小值：

```
Math.max(100, 99); // 100
Math.min(1.2, 2.3); // 1.2
```

计算x^y次方：

```
Math.pow(2, 10); // 2的10次方=1024
```

计算√x：

```
Math.sqrt(2); // 1.414...
```

计算e^x次方：

```
Math.exp(2); // 7.389...
```

计算以e为底的对数：

```
Math.log(4); // 1.386...
```

计算以10为底的对数：

```
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);
    }
}
```

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...
    }
}
```

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));
    }
}
```

`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

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。

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。

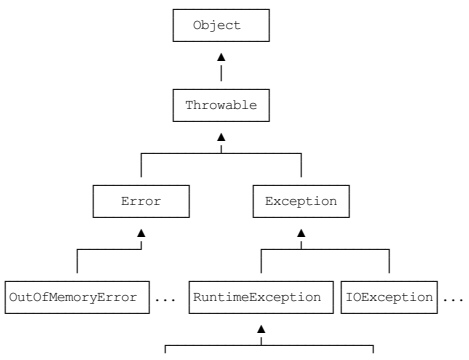
方法二：在语言层面上提供一个异常处理机制。

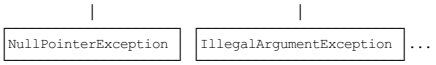
Java内置了一套异常处理机制，总是使用异常来表示错误。

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：

1. RuntimeException以及它的子类；
2. 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。

注意：编译器对RuntimeException及其子类不做强制捕获要求，不是指应用程序本身不应该捕获并处理RuntimeException。是否需要捕获，具体问题具体分析。

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
        }
    }
}
```

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");
    }
}
```

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 {
    ...
}
```

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");
    }
}
```

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");
    }
}
```

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");
    }
}
```

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。

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：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println(e);
    } catch (NumberFormatException e) {
        System.out.println(e);
    }
}
```

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：

```
public static void main(String[] args) {
    try {
        process1();
        process2();
        process3();
    } catch (IOException e) {
        System.out.println("IO error");
    } catch (UnsupportedEncodingException e) { // 永远捕获不到
        System.out.println("Bad encoding");
    }
}
```

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：

1. finally语句不是必须的，可写可不写；
2. 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。

捕获异常练习

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。

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
    }
}
```

通过printStackTrace()可以打印出方法的调用栈，类似：

```
java.lang.NumberFormatException: null
    at java.base/java.lang.Integer.parseInt(Integer.java:614)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    at Main.process2(Main.java:16)
    at Main.process1(Main.java:12)
    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，从下往上看，调用层次依次是：

1. main()调用process1()；
2. process1()调用process2()；
3. process2()调用Integer.parseInt(String)；
4. 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：

1. 创建某个Exception的实例；
2. 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();
    }
}
```

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();
    }
}
```

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。

捕获到异常并再次抛出时，一定要留住原始异常，否则很难定位第一案发现场！

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");
        }
    }
}
```

上述代码执行结果如下：

```
catched
finally
Exception in thread "main" java.lang.RuntimeException: java.lang.NumberFormatException: For input string: "abc"
    at Main.main(Main.java:8)
Caused by: java.lang.NumberFormatException: For input string: "abc"
    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();
        }
    }
}
```

执行上述代码，发现异常信息如下：

```
catched
finally
Exception in thread "main" java.lang.IllegalArgumentException
    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;
        }
    }
}
```

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：

```
Exception in thread "main" java.lang.IllegalArgumentException
    at Main.main(Main.java:11)
Suppressed: java.lang.NumberFormatException: For input string: "abc"
    at java.base/java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
    at java.base/java.lang.Integer.parseInt(Integer.java:652)
    at java.base/java.lang.Integer.parseInt(Integer.java:770)
    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。

提问时贴出异常

异常打印的详细的栈信息是找出问题的关键，许多初学者在提问时只贴代码，不贴异常，相当于只报案不给线索，福尔摩斯也无能为力。

还有的童鞋只贴部分异常信息，最关键的Caused by: xxx给省略了，这都属于不正确的提问方式，得改。

练习

如果传入的参数为负，则抛出IllegalArgumentException。

抛出异常练习

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。

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

从BaseException派生自定义异常

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。

在所有的RuntimeException异常中，Java程序员最熟悉的恐怕就是NullPointerException了。

NullPointerException即空指针异常，俗称NPE。如果一个对象为null，调用其方法或访问其字段就会产生NullPointerException，这个异常通常是由JVM抛出的，例如：

```
// NullPointerException
----
public class Main {
    public static void main(String[] args) {
        String s = null;
        System.out.println(s.toLowerCase());
    }
}
```

指针这个概念实际上源自C语言，Java语言中并无指针。我们定义的变量实际上是引用，Null Pointer更确切地说是Null Reference，不过两者区别不大。

处理NullPointerException

如果遇到NullPointerException，我们应该如何处理？首先，必须明确，NullPointerException是一种代码逻辑错误，遇到NullPointerException，遵循原则是早暴露，早修复，严禁使用catch来隐藏这种编码错误：

```
// 错误示例：捕获NullPointerException
try {
    transferMoney(from, to, amount);
} catch (NullPointerException e) {
}
```

好的编码习惯可以极大地降低NullPointerException的产生，例如：

成员变量在定义时初始化：

```
public class Person {
    private String name = "";
}
```

使用空字符串""而不是默认的null可避免很多NullPointerException，编写业务逻辑时，用空字符串""表示未填写比null安全得多。

返回空字符串""、空数组而不是null：

```
public String[] readLinesFromFile(String file) {
    if (getFileSize(file) == 0) {
        // 返回空数组而不是null:
        return new String[0];
    }
    ...
}
```

这样可以使得调用方无需检查结果是否为null。

如果调用方一定要根据null判断，比如返回null表示文件不存在，那么考虑返回Optional<T>：

```
public Optional<String> readFromFile(String file) {
    if (!fileExist(file)) {
        return Optional.empty();
    }
    ...
}
```

这样调用方必须通过Optional.isPresent()判断是否有结果。

定位NullPointerException

如果产生了NullPointerException，例如，调用a.b.c.x()时产生了NullPointerException，原因可能是：

- a是null；
- a.b是null；
- a.b.c是null；

确定到底是哪个对象是null以前只能打印这样的日志：

```
System.out.println(a);
System.out.println(a.b);
System.out.println(a.b.c);
```

从Java 14开始，如果产生了NullPointerException，JVM可以给出详细的信息告诉我们null对象到底是谁。我们来看例子：

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        System.out.println(p.address.city.toLowerCase());
```

```
    }
}

class Person {
    String[] name = new String[2];
    Address address = new Address();
}

class Address {
    String city;
    String street;
    String zipcode;
}
```

可以在NullPointerException的详细信息中看到类似... because "<local1>.address.city" is null，意思是city字段为null，这样我们就能快速定位问题所在。

这种增强的NullPointerException详细信息是Java 14新增的功能，但默认是关闭的，我们可以给JVM添加一个-XX:+ShowCodeDetailsInExceptionMessages参数启用它：

```
java -XX:+ShowCodeDetailsInExceptionMessages Main.java
```

小结

NullPointerException是Java代码常见的逻辑错误，应当早暴露，早修复；

可以启用Java 14的增强异常信息来查看NullPointerException的详细错误信息。

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：

```
public static void main(String[] args) {
    double x = Math.abs(-123.45);
    assert x >= 0;
    System.out.println(x);
}
```

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：

```
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：

```
void sort(int[] arr) {
    assert arr != null;
}
```

应该抛出异常并在上层捕获：

```
void sort(int[] arr) {
    if (arr == null) {
        throw new IllegalArgumentException("array cannot be null");
    }
}
```

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);
    }
}
```

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。

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：

1. 可以设置输出样式，避免自己每次都写"ERROR: " + var；
2. 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；
3. 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；
4. 可以按包名控制日志级别，只输出某些包打的日志；
5. 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...");
    }
}
```

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.");
----
    }
}
```

打印异常

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。

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.");
```

```
    }
}
```

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);
}
```

练习

使用`log.error(String, Throwable)`打印异常。

Commons Logging练习

小结

Commons Logging是使用最广泛的日志模块；

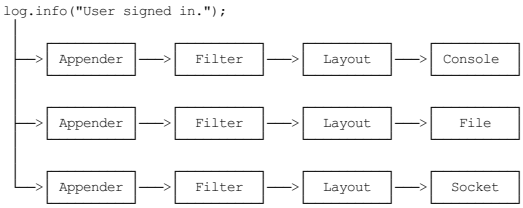
Commons Logging的API非常简单；

Commons Logging可以自动检测并使用其他日志模块。

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：



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：

```
03-03 12:09:45.880 [main] INFO  com.itranswarp.learnjava.Main
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。

commons logging + log4j

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）。

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：

```
int score = 99;
p.setScore(score);
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：

```
int score = 99;
p.setScore(score);
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：

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

class Main {
    final Logger logger = LoggerFactory.getLogger(getClass());
}
```

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。

slf4j+logback

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。

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。

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：

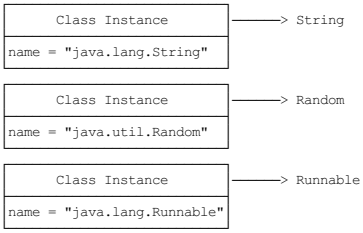
```
public final class Class {
    private Class() {}
}
```

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：

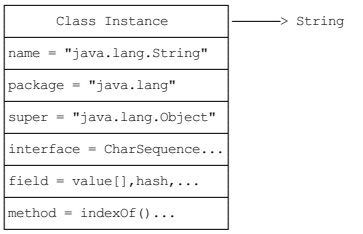
```
Class cls = new Class(String);
```

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）：



一个Class实例包含了该class的所有完整信息：



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());
    }
}
```

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。

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;
}
```

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：

```
public int Student.score
public java.lang.String Person.name
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;
    }
}
```

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：

```
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;
    }
}
```

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：reflect-field

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。

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";
    }
}
```

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：

```
public int Student.getScore(java.lang.String)
public java.lang.String Person.getName()
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：

```
String s = "Hello world";
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);
    }
}
```

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);
    }
}
```

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;
    }
}
```

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");
    }
}
```

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：

```
Method m = Person.class.getMethod("hello");
m.invoke(new Student());
```

实际上相当于：

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

练习

利用反射调用方法：reflect-method

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。

我们通常使用new操作符创建新的实例：

```
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：

```
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);
    }
}
```

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。

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：

```
Class cls = String.class; // 获取到String的Class
```

还可以用实例的`getClass()`方法获取：

```
String s = "";
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：

```
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());
    }
}
```

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);
        }
    }
}
```

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);
        }
    }
}
```

`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()); // java.io.FilterInputStream，因为DataInputStream继承自FilterInputStream
System.out.println(java.io.Closeable.class.getSuperclass()); // null，对接口调用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：

```
Object n = Integer.valueOf(123);
boolean isDouble = n instanceof Double; // false
boolean isInteger = n instanceof Integer; // true
boolean isNumber = n instanceof Number; // true
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()：

```
// Integer i = ?
Integer.class.isAssignableFrom(Integer.class); // true，因为Integer可以赋值给Integer
// Number n = ?
Number.class.isAssignableFrom(Integer.class); // true，因为Integer可以赋值给Number
// Object o = ?
Object.class.isAssignableFrom(Integer.class); // true，因为Integer可以赋值给Object
// Integer i = ?
Integer.class.isAssignableFrom(Number.class); // false，因为Number不能赋值给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。

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某个实例向上转型并赋值给接口类型变量的：

```
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：

```
public interface Hello {
    void morning(String name);
}
```

编写实现类：

```
public class HelloWorld implements Hello {
    public void morning(String name) {
        System.out.println("Good morning, " + name);
    }
}
```

创建实例，转型为接口并调用：

```
Hello hello = new HelloWorld();
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);
}
```

在运行期动态创建一个interface实例的方法如下：

1. 定义一个InvocationHandler实例，它负责实现接口的方法调用；
2. 通过Proxy.newProxyInstance()创建interface实例，它需要3个参数：
    1. 使用的ClassLoader，通常就是接口类的ClassLoader；
    2. 需要实现的接口数组，至少需要传入一个接口进去；
    3. 用来处理接口方法调用的InvocationHandler实例。
3. 将返回的Object强制转型为接口。

动态代理实际上是JVM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：

```
public class HelloDynamicProxy implements Hello {
    InvocationHandler handler;
    public HelloDynamicProxy(InvocationHandler handler) {
        this.handler = handler;
    }
    public void morning(String name) {
        handler.invoke(
           this,
           Hello.class.getMethod("morning", String.class),
           new Object[] { name });
    }
}
```

其实就是JVM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。

本节我们将介绍Java程序的一种特殊“注释”——注解（Annotation）。

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、Class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：

```
public class Hello {
    @Check(min=0, max=100, value=55)
    public int n;

    @Check(value=99)
    public int p;

    @Check(99) // @Check(value=99)
    public int x;

    @Check
    public int y;
}
```

`@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。

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(Reports.class)
@Target(ElementType.TYPE)
public @interface Report {
    int type() default 0;
    String level() default "info";
    String value() default "";
}

@Target(ElementType.TYPE)
public @interface Reports {
    Report[] value();
}
```

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`。

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;
    }
```

```
}
```

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：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.FIELD)
public @interface Range {
    int min() default 0;
    int max() default 255;
}
```

在某个JavaBean中，我们可以使用该注解：

```
public class Person {
    @Range(min=1, max=20)
    public String name;

    @Range(max=10)
    public String city;
}
```

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, ReflectiveOperationException {
    // 遍历所有Field:
    for (Field field : person.getClass().getFields()) {
        // 获取Field定义的@Range:
        Range range = field.getAnnotation(Range.class);
        // 如果@Range存在:
        if (range != null) {
            // 获取Field的值:
            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。

annotation-range-check

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。

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。

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）。

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> {
    /**
     * 返回负数：当前实例比参数o小
     * 返回0：当前实例与参数o相等
     * 返回正数：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));
----
    }
}
```

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;
    }
}
```

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;
    }
}
```

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。

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>`。

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：

```
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;
    }
}
```

而虚拟机根本不知道泛型。这是虚拟机执行的代码：

```
public class Pair {
    private Object first;
    private Object last;
    public Pair(Object first, Object last) {
        this.first = first;
        this.last = last;
    }
    public Object getFirst() {
        return first;
    }
    public Object getLast() {
        return last;
    }
}
```

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;
    }
}
```

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：无法判断带泛型的类型：

```
Pair<Integer> p = new Pair<>(123, 456);
// Compile error:
if (p instanceof Pair<String>) {
}
```

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：

```
public class Pair<T> {
    public boolean equals(T t) {
        return this == t;
    }
}
```

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：

```
public class Pair<T> {
    public boolean same(T t) {
        return this == t;
    }
}
```

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：

```
public class IntPair extends Pair<Integer> {
}
```

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：

```
IntPair ip = new IntPair(1, 2);
```

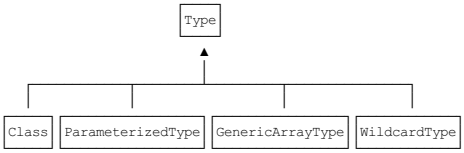
前面讲了，我们无法获取Pair<T>的T类型，即给定一个变量Pair<Integer> p，无法从p中获取到Integer类型。

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);
    }
}
```

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>`。

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>`：

```
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>`：

```
public class PairHelper {
    static int add(Pair<Number> p) {
        Number first = p.getFirst();
        Number last = p.getLast();
        return first.intValue() + last.intValue();
    }
}
```

上述代码是可以正常编译的。使用的时候，我们传入：

```
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;
    }
}
```

直接运行，会得到一个编译错误：

```
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：

```
Number first = p.getFirst();
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;
    }
}
```

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：

```
<? 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：

```
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：

```
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;
    }
}
```

不出意外，我们会得到一个编译错误：

```
incompatible types: Integer cannot be converted to CAP#1
where CAP#1 is a fresh type-variable:
    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的子类型给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：

```
public class Pair<T extends Number> { ... }
```

现在，我们只能定义：

```
Pair<Number> p1 = null;
Pair<Integer> p2 = new Pair<>(1, 2);
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：

```
Pair<String> p1 = null; // compile error!
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。

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：

```
void set(Pair<Integer> p, Integer first, Integer last) {
    p.setFirst(first);
    p.setLast(last);
}
```

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：

```
void set(Pair<? super Integer> p, Integer first, Integer last) {
    p.setFirst(first);
    p.setLast(last);
}
```

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;
    }
}
```

考察`Pair<? super Integer>`的`setFirst()`方法，它的方法签名实际上是：

```
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：

```
? 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) {
    return p.getFirst() == null || p.getLast() == null;
}
```

<?>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;
    }
}
```

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。

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[]：

```
public class ArrayHelper {
    @SafeVarargs
    static <T> T[] asArray(T... objs) {
        return objs;
    }
}

String[] ss = ArrayHelper.asArray("a", "b", "c");
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：

```
static <T> T[] asArray(T... objs) {
    return objs;
}
```

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;
    }
----
}
```

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。

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。

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：

```
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。

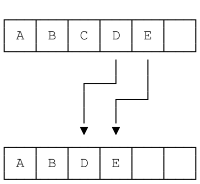
小结

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。

在集合类中，`List`是最基础的一种集合：它是一种有序列表。

`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。

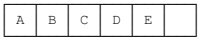
数组和`List`类似，也是有序结构，如果我们使用数组，在添加和删除元素的时候，会非常不方便。例如，从一个已有的数组`{'A', 'B', 'C', 'D', 'E'}`中删除索引为`2`的元素：



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）：

```
size=5
```



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：

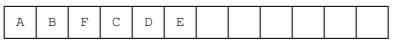
size=5

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

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



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：

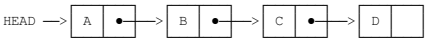
size=7

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：boolean add(E e)
- 在指定索引添加一个元素：boolean add(int index, E e)
- 删除指定索引的元素：E remove(int index)
- 删除某个元素：boolean 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：



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());
    }
}
```

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);
    }
}
```

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：

```
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);
```

```
        }
    }
}
```

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);
        }
    }
}
```

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);
        }
    }
}
```

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);
        }
    }
}
```

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);
        }
    }
}
```

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);
        }
    }
}
```

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
    }
}
```

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;
    }
----
}
```

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shuffle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;
    }
----
}
```

找出缺失的数字

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。

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
    }
}
```

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?
    }
}
```

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：

```
public class ArrayList {
    Object[] elementData;
    public boolean contains(Object o) {
        for (int i = 0; i < elementData.length; i++) {
```

```
            if (o.equals(elementData[i])) {
                return true;
            }
        }
        return false;
    }
}
```

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;
    }
}
```

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：

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

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：

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        return this.name.equals(p.name) && this.age == p.age;
    }
    return false;
}
```

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：

```
public boolean equals(Object o) {
    if (o instanceof Person) {
        Person p = (Person) o;
        boolean nameEquals = false;
        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：

1. 先确定实例“相等”的逻辑，即哪些字段相等，就认为实例相等；
2. 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；
3. 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;
import java.util.Objects;
----
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;
    }
}
```

覆写equals方法

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。

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;
    }
}
```

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
    }
}
```

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：

```
Map<String, Integer> map = new HashMap<>();
map.put("apple", 123);
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);
        }
    }
}
```

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);
        }
    }
}
```

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;
    }
}
```

find-student-score

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。

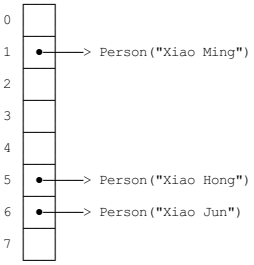
最常用的一种Map实现是HashMap。

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
    }
}
```

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：

1. 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；
2. 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}）。

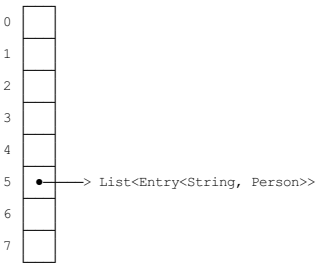
最后一个问题：如果不同的两个key，例如"a"和"b"，它们的hashCode()恰好是相同的（这种情况是完全可能的，因为不相等的两个实例，只要求hashCode()尽量不相等），那么，当我们放入：

```
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：



在查找的时候，例如：

```
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。

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));
    }
}
```

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。

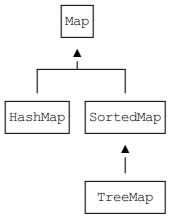
小结

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。

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
    }
}
```

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 + "}";
    }
}
```

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);
    }
}
```

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：

```
public int compare(Student p1, Student p2) {
```

```
    return p1.score > p2.score ? -1 : 1;
}
```

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：

```
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。

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：

1. 创建Properties实例；
2. 调用load()读取文件；
3. 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：

```
Properties props = new Properties();
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"));
    }
}
```

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：

```
Properties props = new Properties();
props.load(getClass().getResourceAsStream("/common/setting.properties"));
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。

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
    }
}
```

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。

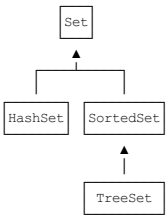
最常用的Set实现类是HashSet，实际上，HashSet仅仅是对HashMap的一个简单封装，它的核心代码如下：

```
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();

    public boolean add(E e) {
        return map.put(e, PRESENT) == null;
    }

    public boolean contains(Object o) {
        return map.containsKey(o);
    }

    public boolean remove(Object o) {
        return map.remove(o) == PRESENT;
    }
}
```

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);
        }
    }
}
```

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);
        }
    }
}
```

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;
    }
}
```

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。

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
    }
}
```

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
    }
}
```

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。

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
    }
}
```

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;
    }
}
```

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）。

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"); // A <- B
        deque.offerFirst("C"); // C <- A <- B
        System.out.println(deque.pollFirst()); // C, 剩下A <- B
        System.out.println(deque.pollLast()); // B, 剩下A
        System.out.println(deque.pollFirst()); // A
        System.out.println(deque.pollFirst()); // null
    }
}
```

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。

```
// 不推荐的写法:
LinkedList<String> d1 = new LinkedList<>();
d1.offerLast("z");
// 推荐的写法:
Deque<String> d2 = new LinkedList<>();
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。

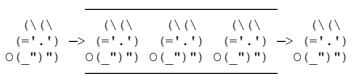
小结

Deque实现了一个双端队列（Double Ended Queue），它可以：

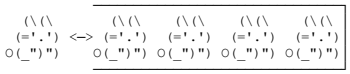
- 将元素添加到队尾或队首：addLast()/offerLast()/addFirst()/offerFirst()；
- 从队首 / 队尾获取元素并删除：removeFirst()/pollFirst()/removeLast()/pollLast()；
- 从队首 / 队尾获取元素但不删除：getFirst()/peekFirst()/getLast()/peekLast()；
- 总是调用xxxFirst()/xxxLast()以便与Queue的方法区分开；
- 避免把null添加到队列。

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：



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：



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()；
- 取栈顶元素但不弹出：peek()。

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()/removeFirst()；
- 取栈顶元素但不弹出：peek()/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：

```
static void main(String[] args) {
    foo(123);
}

static String foo(x) {
    return "F-" + bar(x + 1);
}

static int bar(int x) {
    return x << 2;
}
```

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;
    }
}
```

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：

然后计算12500÷16=781…4，余数是4，把余数4压栈：

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：

然后计算48÷16=3…0，余数是0，把余数0压栈：

最后计算3÷16=0…3，余数是3，把余数3压栈：

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)`。

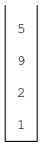
但是计算机执行表达式的时候，它并不能直接计算中缀表达式，而是通过编译器把中缀表达式转换为后缀表达式，例如：`1 2 9 5 - * +`。

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：

紧接着，遇到数字2，9，5，也扔到栈里：



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：

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：

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：

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 "";
    }
}
```

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;
    }
}
```

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;
    }
}
```

Stack练习

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`。

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);
        }
    }
}
```

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。

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：

```
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：

```
List<String> list1 = List.of();
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.singletonList("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);
    }
}
```

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);
    }
}
```

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!
    }
}
```

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);
    }
}
```

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);
    }
}
```

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。

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。

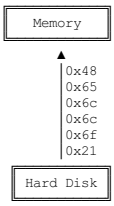
从Java代码来看，输入实际上就是从外部，例如，硬盘上的某个文件，把内容读到内存，并且以Java提供的某种数据类型表示，例如，byte[]，String，这样，后续代码才能处理这些数据。

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。

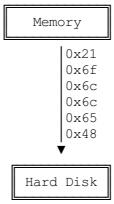
InputStream / OutputStream

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个字节，就相当于读入了6个字节的数据：



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：



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做了解码，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。

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);
    }
}
```

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：

```
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：

```
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：

```
// 假设当前目录是C:\Docs
File f1 = new File("sub\\javac"); // 绝对路径是C:\Docs\sub\javac
File f3 = new File(".\\sub\\javac"); // 绝对路径是C:\Docs\sub\javac
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());
    }
}
```

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：

```
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());
    }
}
```

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());
    }
}
```

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("==========");
    }
}
```

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);
        }
    }
}
```

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。

io-file

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。

`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);
            }
        }
    }
}
```

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();
    }
}
```

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。

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：

```
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"));
    }
}
```

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。

同时操作多个AutoCloseable资源时，在try(resource) { ... }语句中可以同时写出多个资源，用;隔开。例如，同时读写两个文件：

```
// 读取input.txt，写入output.txt:
try (InputStream input = new FileInputStream("input.txt");
     OutputStream output = new FileOutputStream("output.txt"))
{
    input.transferTo(output); // transferTo的作用是?
}
```

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：

```
java CopyFile.java source.txt copy.txt
```

CopyFile练习

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。

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：

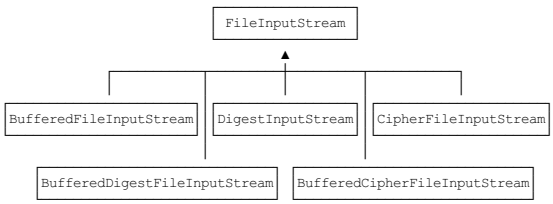
```
DigestFileInputStream extends FileInputStream
```

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：



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：

```
InputStream file = new FileInputStream("test.gz");
```

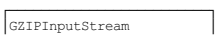
紧接着，我们希望FileInputStream能提供缓冲的功能来提高读取的效率，因此我们用BufferedInputStream包装这个InputStream，得到的包装类型是BufferedInputStream，但它仍然被视为一个InputStream：

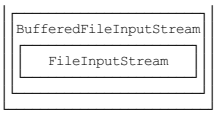
```
InputStream buffered = new BufferedInputStream(file);
```

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：

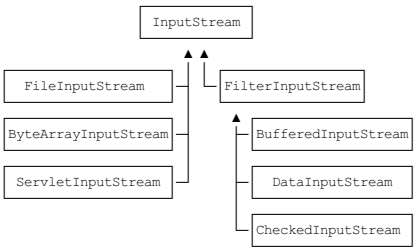
```
InputStream gzip = new GZIPInputStream(buffered);
```

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：

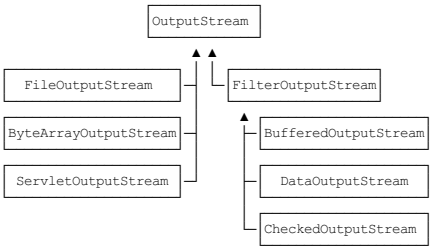




上述这种通过一个“基础”组件再叠加各种“附加”功能组件的模式，称之为Filter模式（或者装饰器模式：Decorator）。它可以让我们通过少量的类来实现各种功能的组合：



类似的，OutputStream也是以这种模式来提供各种功能：



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);
        if (n != -1) {
            this.count += n;
        }
        return n;
    }
}
```

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。

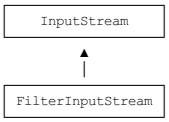
小结

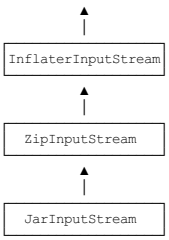
Java的IO标准库使用Filter模式为InputStream和OutputStream增加功能：

- 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）。

ZipInputStream是一种FilterInputStream，它可以直接读取zip包的内容：





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。

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。

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：

```
public interface Serializable {
}
```

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()));
    }
}
```

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。

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")) {
}
```

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：

```
// 持有InputStream:
InputStream input = new FileInputStream("src/readme.txt");
// 变换为Reader:
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 FileInputStream("src/readme.txt"), "UTF-8")) {
    // TODO:
}
```

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。

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"), "UTF-8")) {
    // TODO:
}
```

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。

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：

```
System.out.print(12345); // 输出12345
System.out.print(new Object()); // 输出类似java.lang.Object@3c7a835a
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());
    }
}
```

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。

从Java 7开始，提供了Files和Paths这两个工具类，能极大地方便我们读写文件。

虽然Files和Paths是java.nio包里面的类，但他俩封装了很多读写文件的简单方法，例如，我们要把一个文件的全部内容读取为一个byte[]，可以这么写：

```
byte[] data = Files.readAllBytes(Paths.get("/path/to/file.txt"));
```

如果是文本文件，可以把一个文件的全部内容读取为String：

```
// 默认使用UTF-8编码读取:
String content1 = Files.readString(Paths.get("/path/to/file.txt"));
// 可指定编码:
String content2 = Files.readString(Paths.get("/path/to/file.txt"), StandardCharsets.ISO_8859_1);
// 按行读取并返回每行内容:
List<String> lines = Files.readAllLines(Paths.get("/path/to/file.txt"));
```

写入文件也非常方便：

```
// 写入二进制文件:
byte[] data = ...
Files.write(Paths.get("/path/to/file.txt"), data);
// 写入文本并指定编码:
Files.writeString(Paths.get("/path/to/file.txt"), "文本内容...", StandardCharsets.ISO_8859_1);
// 按行写入文本:
List<String> lines = ...
Files.write(Paths.get("/path/to/file.txt"), lines);
```

此外，Files工具类还有copy()、delete()、exists()、move()等快捷方法操作文件和目录。

最后需要特别注意的是，Files提供的读写方法，受内存限制，只能读写小文件，例如配置文件等，不可一次读入几个G的大文件。读写大型文件仍然要使用文件流，每次只读写一部分文件内容。

小结

对于简单的小文件读写操作，可以使用Files工具类简化代码。

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。

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

中国用户	美国用户
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。

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));
    }
}
```

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：

```
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：

```
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：

```
public class Date implements Serializable, Cloneable, Comparable<Date> {

    private transient long fastTime;

    ...
}
```

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());
    }
}
```

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));
    }
}
```

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));
    }
}
```

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);
    }
}
```

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
    }
}
```

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
    }
}
```

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
    }
}
```

可见，利用`Calendar`进行时区转换的步骤是：

1. 清除所有字段；
2. 设定指定时区；
3. 设定日期和时间；
4. 创建`SimpleDateFormat`并设定目标时区；
5. 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
    }
}
```

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。

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
    }
}
```

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);
    }
}
```

`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
    }
}
```

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
    }
}
```

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);
    }
}
```

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")));
    }
}
```

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
    }
}
```

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”。

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);
    }
}
```

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);
    }
}
```

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);
    }
}
```

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：

```
ZonedDateTime zdt = ...
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;
    }
}
```

提示：ZonedDateTime仍然提供了plusDays()等加减操作。

flight-time练习

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。

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：

```
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 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));
    }
}
```

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。

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
    }
}
```

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;
}
```

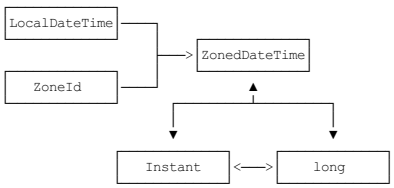
一个是以秒为单位的时间戳，一个是更精确的纳秒精度。它和System.currentTimeMillis()返回的long相比，只是多了更高精度的纳秒。

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。

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.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)));
    }
}
```

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；

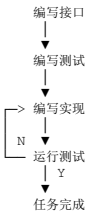
在数据库中存储时间戳时，尽量使用long型时间戳，它具有省空间，效率高，不依赖数据库的优点。

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。

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：



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\times2\times3\times...\times 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：

```
package com.itranswarp.learnjava;

import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class FactorialTest {

    @Test
    void testFact() {
        assertEquals(1, Factorial.fact(1));
        assertEquals(2, Factorial.fact(2));
        assertEquals(6, Factorial.fact(3));
        assertEquals(3628800, Factorial.fact(10));
        assertEquals(2432902008176640000L, Factorial.fact(20));
    }
}
```

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`FactorialTest.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：

```
org.opentest4j.AssertionFailedError: expected: <3628800> but was: <362880>
 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

JUnit测试

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。

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：

```
public class Calculator {
    private long n = 0;

    public long add(long x) {
        n = n + x;
        return n;
    }

    public long sub(long x) {
        n = n - x;
        return n;
    }
}
```

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：

```
public class CalculatorTest {

    Calculator calculator;

    @BeforeEach
    public void setUp() {
        this.calculator = new Calculator();
    }

    @AfterEach
    public void tearDown() {
        this.calculator = null;
    }

    @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：

1. 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；
2. 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

使用Fixture

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。

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：

```
public class Factorial {
    public static long fact(long n) {
        if (n < 0) {
            throw new IllegalArgumentException();
        }
        long r = 1;
        for (long i = 1; i <= n; i++) {
            r = r * i;
        }
        return r;
    }
}
```

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：

```
@Test
void testNegative() {
    assertThrows(IllegalArgumentException.class, new Executable() {
        @Override
        public void execute() throws Throwable {
            Factorial.fact(-1);
        }
    });
}
```

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。

异常测试

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。

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
}
```

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

条件测试。

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。

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" })
void testCapitalizeUsingCsvFile(String input, String result) {
    assertEquals(result, StringUtils.capitalize(input));
}
```

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：

```
apple, Apple
HELLO, Hello
JUnit, Junit
reSource, Resource
```

练习

参数化测试StringUtils

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。

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。

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
    }
}
```

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。

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));
    }
}
```

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));
    }
}
```

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!");
    }
}
```

电话匹配练习

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

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));
    }
}
```

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));
    }
}
```

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

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：

```
\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!");
        }
    }
}
```

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);
        }
    }
}
```

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。

分组匹配

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。

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)); // ""
        }
    }
}
```

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"
        }
    }
}
```

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??。

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);
        }
    }
}
```

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."
    }
}
```

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xxxx，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);
    }
}
```

上述代码的运行结果是：

```
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xxxx括起来。实现替换的关键就在于" $1 "，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。

模板引擎练习

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()

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。

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);
    }
}
```

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);
    }
}
```

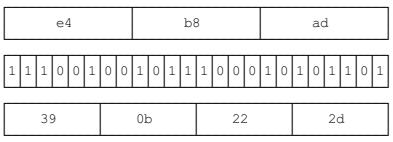
要特别注意：URL编码是编码算法，不是加密算法。URL编码的目的是把任意文本数据编码为%前缀表示的文本，编码后的文本仅包含A~Z，a~z，0~9，-，_，.，*和%，便于浏览器和服务器处理。

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：



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);
    }
}
```

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]
    }
}
```

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));
    }
}
```

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));
    }
}
```

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。

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：

```
hashA("java001") = "123456"
hashA("java002") = "123457"
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：

```
hashB("java001") = "123456"
hashB("java002") = "580271"
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));
    }
}
```

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	bed128365216c019988915ed3add75fb

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	570da6d5277a646ff6552b8832012f5dc
…	…
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）：

```
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: db8ac1c259eb89d4a131b253bacfca5f319d54f2
        System.out.println(new BigInteger(1, result).toString(16));
    }
}
```

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。

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：

```
public class Main {
    public static void main(String[] args) throws Exception {
        // 注册BouncyCastle:
        Security.addProvider(new BouncyCastleProvider());
        // 按名称正常调用:
        MessageDigest md = MessageDigest.getInstance("RipeMD160");
        md.update("HelloWorld".getBytes("UTF-8"));
        byte[] result = md.digest();
        System.out.println(new BigInteger(1, result).toString(16));
    }
}
```

其中，注册BouncyCastle是通过下面的语句实现的：

```
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

使用BouncyCastle提供的RipeMD160

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。

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));
    }
}
```

和MD5相比，使用HmacMD5的步骤是：

1. 通过名称`HmacMD5`获取`KeyGenerator`实例；
2. 通过`KeyGenerator`创建一个`SecretKey`实例；
3. 通过名称`HmacMD5`获取`Mac`实例；
4. 用`SecretKey`初始化`Mac`实例；
5. 对`Mac`实例反复调用`update(byte[])`输入数据；
6. 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]
    }
}
```

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。

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);
    }
}
```

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：

1. 根据算法名称/工作模式/填充模式获取Cipher实例；
2. 根据算法名称初始化一个SecretKey实例，密钥必须是指定长度；
3. 使用SerectKey初始化Cipher实例，并设置加密或解密模式；
4. 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;
    }
}
```

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。

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"));
    }

    // 加密:
    public static byte[] encrypt(String password, byte[] salt, byte[] input) throws GeneralSecurityException {
        PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
        SecretKeyFactory skeyFactory = SecretKeyFactory.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
        SecretKey skey = skeyFactory.generateSecret(keySpec);
        PBEParameterSpec pbeps = new PBEParameterSpec(salt, 1000);
        Cipher cipher = Cipher.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
        cipher.init(Cipher.ENCRYPT_MODE, skey, pbeps);
        return cipher.doFinal(input);
    }

    // 解密:
    public static byte[] decrypt(String password, byte[] salt, byte[] input) throws GeneralSecurityException {
        PBEKeySpec keySpec = new PBEKeySpec(password.toCharArray());
        SecretKeyFactory skeyFactory = SecretKeyFactory.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
        SecretKey skey = skeyFactory.generateSecret(keySpec);
        PBEParameterSpec pbeps = new PBEParameterSpec(salt, 1000);
        Cipher cipher = Cipher.getInstance("PBEwithSHA1and128bitAES-CBC-BC");
        cipher.init(Cipher.DECRYPT_MODE, skey, pbeps);
        return cipher.doFinal(input);
    }
}
```

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）。

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：

1. 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；
2. 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；
3. 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());

        // 检查双方的本地密钥是否相同:
        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));
    }
}
```

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

[密钥交换算法]

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。

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：

1. 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；
2. 小红用自己的RSA私钥解密得到AES口令；
3. 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);
    }
}
```

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

RSA加密

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。

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);
    }
}
```

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

签名算法练习

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。

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
```

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);
    }

    static KeyStore loadKeyStore(String keyStoreFile, String password) {
        try (InputStream input = Main.class.getResourceAsStream(keyStoreFile)) {
            if (input == null) {
                throw new RuntimeException("file not found in classpath: " + keyStoreFile);
            }
            KeyStore ks = KeyStore.getInstance(KeyStore.getDefaultType());
            ks.load(input, password.toCharArray());
            return ks;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    static byte[] encrypt(X509Certificate certificate, byte[] message) throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance(certificate.getPublicKey().getAlgorithm());
        cipher.init(Cipher.ENCRYPT_MODE, certificate.getPublicKey());
        return cipher.doFinal(message);
```

```
    }

    static byte[] decrypt(PrivateKey privateKey, byte[] data) throws GeneralSecurityException {
        Cipher cipher = Cipher.getInstance(privateKey.getAlgorithm());
        cipher.init(Cipher.DECRYPT_MODE, privateKey);
        return cipher.doFinal(data);
    }

    static byte[] sign(PrivateKey privateKey, X509Certificate certificate, byte[] message)
            throws GeneralSecurityException {
        Signature signature = Signature.getInstance(certificate.getSigAlgName());
        signature.initSign(privateKey);
        signature.update(message);
        return signature.sign();
    }

    static boolean verify(X509Certificate certificate, byte[] message, byte[] sig) throws GeneralSecurityException {
        Signature signature = Signature.getInstance(certificate.getSigAlgName());
        signature.initVerify(certificate);
        signature.update(message);
        return signature.verify(sig);
    }
}
```

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：

1. 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；
2. 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；
3. 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

使用数字证书

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。

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。

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。

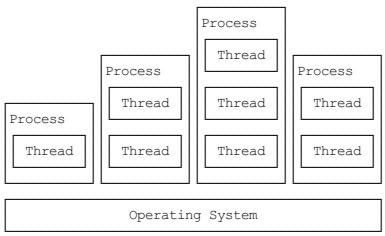
即使是多核CPU，因为通常任务的数量远远多于CPU的核数，所以任务也是交替执行的。

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。

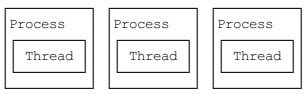
进程和线程的关系就是：一个进程可以包含一个或多个线程，但至少会有一个线程。



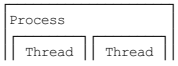
操作系统调度的最小任务单位其实不是进程，而是线程。常用的Windows、Linux等操作系统都采用抢占式多任务，如何调度线程完全由操作系统决定，程序自己不能决定什么时候执行，以及执行多长时间。

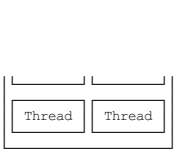
因为同一个应用程序，既可以有多个进程，也可以有多个线程，因此，实现多任务的方法，有以下几种：

多进程模式（每个进程只有一个线程）：

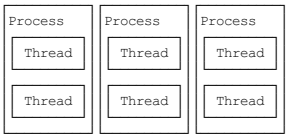


多线程模式（一个进程有多个线程）：





多进程＋多线程模式（复杂度最高）：



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。

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
    }
}
```

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!");
    }
}
```

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!");
    }
}
```

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
    }
}
```

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：

```
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...");
                System.out.println("thread end.");
            }
        };
        t.start();
        System.out.println("main end...");
    }
}
```

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：

1. main线程肯定是先打印main start，再打印main end；
2. 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...");
    }
}
```

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：

```
public class Main {
    public static void main(String[] args) {
        Thread t = new MyThread();
        t.run();
    }
}

class MyThread extends Thread {
    public void run() {
        System.out.println("hello");
    }
}
```

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：

```
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

创建新线程

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；

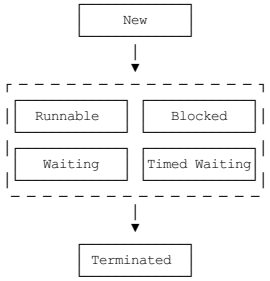
线程调度由操作系统决定，程序本身无法决定调度顺序；

Thread.sleep()可以把当前线程暂停一段时间。

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");
    }
}
```

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。

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!");
        }
    }
}
```

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;
            }
        }
    }
}
```

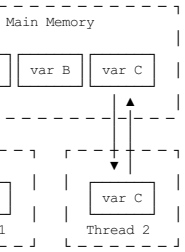
main线程通过调用t.interrupt()从而通知t线程中断，而此时t线程正位于hello.join()的等待中，此方法会立刻结束等待并抛出InterruptedException。由于我们在t线程中捕获了InterruptedException，因此，就可以准备结束该线程。在t线程结束前，对hello线程也进行了interrupt()调用通知其中断。如果去掉这一行代码，可以发现hello线程仍然会继续运行，且JVM不会退出。

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!");
    }
}
```

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！



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。

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

使用守护线程

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）。

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; }
    }
}
```

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。

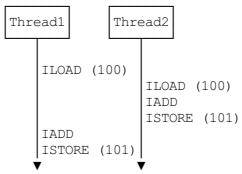
例如，对于语句：

```
n = n + 1;
```

看上去是一行语句，实际上对应了3条指令：

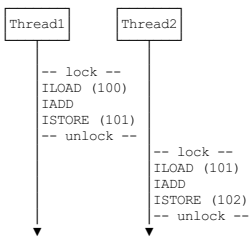
```
ILOAD
IADD
ISTORE
```

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：



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：



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：

```
synchronized(lock) {
    n = n + 1;
}
```

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;
            }
        }
    }
}
```

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：

1. 找出修改共享变量的线程代码块；
2. 选择一个共享实例作为锁；
3. 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;
            }
        }
    }
}
```

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;
            }
        }
    }
}
```

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。

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。

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：

```
public class Counter {
    private int count = 0;

    public synchronized void add(int n) {
        if (n < 0) {
            dec(-n);
        } else {
            count += n;
        }
    }

    public synchronized void dec(int n) {
        count += n;
    }
}
```

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：

```
public void add(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
}

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。

死锁

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。

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：

```
public synchronized String getTask() {
    while (queue.isEmpty()) {
        this.wait();
    }
    return queue.remove();
}
```

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：

```
public synchronized String getTask() {
    while (queue.isEmpty()) {
        // 释放this锁:
        this.wait();
        // 重新获取this锁
    }
    return queue.remove();
}
```

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：

```
public synchronized void addTask(String s) {
```

```
    this.queue.add(s);
    this.notify(); // 唤醒在this锁等待的线程
}
```

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();
    }
}
```

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。

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。

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：

```
if (condition.await(1, TimeUnit.SECOND)) {
    // 被其他线程唤醒
} else {
    // 指定时间内没有被其他线程唤醒
}
```

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。

前面讲到的ReentrantLock保证了只有一个线程可以执行临界区代码：

```
public class Counter {
    private final Lock lock = new ReentrantLock();
    private int[] counts = new int[10];

    public void inc(int index) {
        lock.lock();
        try {
            counts[index] += 1;
        } finally {
            lock.unlock();
        }
    }

    public int[] get() {
        lock.lock();
        try {
            return Arrays.copyOf(counts, counts.length);
        } finally {
            lock.unlock();
        }
    }
}
```

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。

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：

```
public class Point {
    private final StampedLock stampedLock = new StampedLock();

    private double x;
    private double y;

    public void move(double deltaX, double deltaY) {
        long stamp = stampedLock.writeLock(); // 获取写锁
        try {
            x += deltaX;
            y += deltaY;
        } finally {
            stampedLock.unlockWrite(stamp); // 释放写锁
        }
    }

    public double distanceFromOrigin() {
        long stamp = stampedLock.tryOptimisticRead(); // 获得一个乐观读锁
        // 注意下面两行代码不是原子操作
```

```
        // 假设x,y = (100,200)
        double currentX = x;
        // 此处已读取到x=100，但x,y可能被写线程修改为(300,400)
        double currentY = y;
        // 此处已读取到y，如果没有写入，读取是正确的(100,200)
        // 如果有写入，读取是错误的(100,400)
        if (!stampedLock.validate(stamp)) { // 检查乐观读锁后是否有其他写锁发生
            stamp = stampedLock.readLock(); // 获取一个悲观读锁
            try {
                currentX = x;
                currentY = y;
            } finally {
                stampedLock.unlockRead(stamp); // 释放悲观读锁
            }
        }
        return Math.sqrt(currentX * currentX + currentY * currentY);
    }
}
```

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。

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 = new 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 = new HashMap<>();
```

改为：

```
Map<String, String> map = new 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。

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：

```
public int incrementAndGet(AtomicInteger var) {
    int prev, next;
    do {
        prev = var.get();
        next = prev + 1;
    } while ( ! var.compareAndSet(prev, next));
    return next;
}
```

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：

```
class IdGenerator {
    AtomicLong var = new AtomicLong(0);

    public long getNextId() {
        return var.incrementAndGet();
    }
}
```

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`。

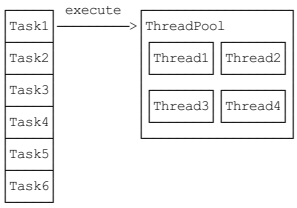
小结

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。

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：



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);
    }
}
```

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);
```

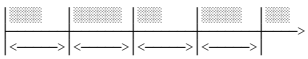
如果任务以固定的每3秒执行，我们可以这样写：

```
// 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);
```

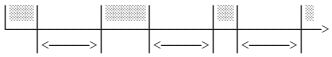
如果任务以固定的3秒为间隔执行，我们可以这样写：

```
// 2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

使用线程池

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。

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

使用Future

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。

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(200);
    }

    static Double fetchPrice() {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
        if (Math.random() < 0.3) {
            throw new RuntimeException("fetch price failed!");
        }
        return 5 + Math.random() * 20;
    }
}
```

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：

```
public interface Supplier<T> {
    T get();
}
```

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：

```
public interface Consumer<T> {
    void accept(T t);
}
```

异常时，CompletableFuture会调用Function对象：

```
public interface Function<T, R> {
    R apply(T t);
}
```

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) -> {
```

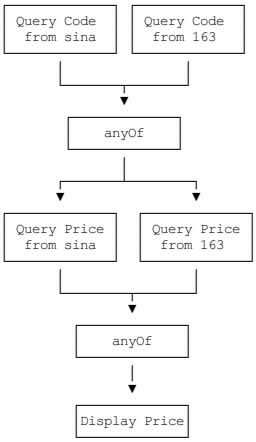
```
            System.out.println("price: " + result);
        });
        // 主线程不要立刻结束，否则CompletableFuture默认使用的线程池会立刻关闭:
        Thread.sleep(2000);
    }

    static String queryCode(String name) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
        return "601857";
    }

    static Double fetchPrice(String code) {
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
        }
        return 5 + Math.random() * 20;
    }
}
```

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(200);
    }

    static String queryCode(String name, String url) {
        System.out.println("query code from " + url + "...");
        try {
            Thread.sleep((long) (Math.random() * 100));
        } 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() * 100));
        } catch (InterruptedException e) {
        }
        return 5 + Math.random() * 20;
    }
}
```

上述逻辑实现的异步查询规则实际上是：



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

使用CompletableFuture

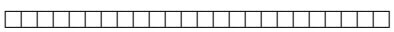
小结

CompletableFuture可以指定异步处理流程：

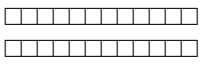
- thenAccept()处理正常结果；
- exceptional()处理异常结果；
- thenApplyAsync()用于串行化另一个CompletableFuture；
- anyOf()和allOf()用于并行化多个CompletableFuture。

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。

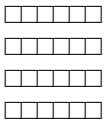
我们举个例子：如果要计算一个超大数组的和，最简单的做法是用一个循环在一个线程内完成：



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;
    }
}
```

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 subresult1 = subtask1.join();
        Long subresult2 = subtask2.join();
        // 汇总结果:
        return subresult1 + subresult2;
    }
}
```

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

使用Fork/Join

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。

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);
    }
}
```

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：

```
public void process(User user) {
    checkPermission();
    doWork();
    saveStatus();
    sendResponse();
}
```

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：

```
public void process(User user) {
    checkPermission(user);
    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<User> 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

ThreadLocal练习

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。

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。

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：

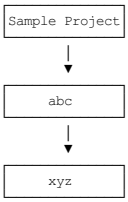
- Maven使用pom.xml定义项目内容，并使用预设的目录结构；
- 在Maven中声明一个依赖项可以自动下载并导入classpath；
- Maven使用groupId，artifactId和version唯一定位一个依赖。

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。

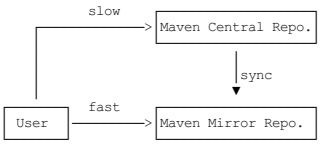
通过上述3个变量，即可唯一确定某个jar包。Maven通过对jar包进行PGP签名确保任何一个jar包一经发布就无法修改。修改已发布jar包的唯一方法是发布一个新版本。

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结尾的版本号会被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>https://maven.aliyun.com/repository/central</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：

命令行编译

在命令中，进入到pom.xml所在目录，输入以下命令：

```
$ mvn clean package
```

如果一切顺利，即可在target目录下获得编译后自动打包的jar。

在IDE中使用Maven

几乎所有的IDE都内置了对Maven的支持。在Eclipse中，可以直接创建或导入Maven项目。如果导入后的Maven项目有错误，可以尝试选择项目后点击右键，选择Maven - Update Project...更新：

练习

使用Maven编译hello项目

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。

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 surefire: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：

```
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。

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：

```
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：

```
<project>
    ...
 <build>
  <plugins>
   <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-shade-plugin</artifactId>
                <version>3.2.1</version>
    <executions>
     <execution>
      <phase>package</phase>
      <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.ManifestResourceTransformer">
            <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

使用maven-shade-plugin创建可执行jar

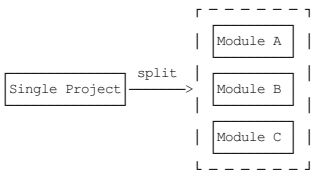
小结

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。

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：

```
mutiple-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：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>module-a</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <name>module-a</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.28</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.3</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.5.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

模块B的pom.xml：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>module-b</artifactId>
    <version>1.0</version>
    <packaging>jar</packaging>

    <name>module-b</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.28</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.3</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.5.2</version>
```

```
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

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：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>parent</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>

    <name>parent</name>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-api</artifactId>
            <version>1.7.28</version>
        </dependency>
        <dependency>
            <groupId>ch.qos.logback</groupId>
            <artifactId>logback-classic</artifactId>
            <version>1.2.3</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.junit.jupiter</groupId>
            <artifactId>junit-jupiter-engine</artifactId>
            <version>5.5.2</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```

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：

```
multiple-project
├── pom.xml
├── parent
│   └── pom.xml
├── module-a
│   ├── pom.xml
│   └── src
├── module-b
│   ├── pom.xml
│   └── src
└── module-c
    ├── pom.xml
    └── src
```

这样模块A就可以简化为：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <parent>
        <groupId>com.itranswarp.learnjava</groupId>
        <artifactId>parent</artifactId>
        <version>1.0</version>
        <relativePath>../parent/pom.xml</relativePath>
    </parent>

    <artifactId>module-a</artifactId>
    <packaging>jar</packaging>
    <name>module-a</name>
</project>
```

模块B、模块C都可以直接从parent继承，大幅简化了pom.xml的编写。

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，我们需要在模块A中引入模块B：

```
    ...
    <dependencies>
        <dependency>
            <groupId>com.itranswarp.learnjava</groupId>
            <artifactId>module-b</artifactId>
            <version>1.0</version>
        </dependency>
    </dependencies>
```

最后，在编译的时候，需要在根目录创建一个pom.xml统一编译：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>build</artifactId>
    <version>1.0</version>
    <packaging>pom</packaging>
    <name>build</name>

    <modules>
        <module>parent</module>
        <module>module-a</module>
        <module>module-b</module>
        <module>module-c</module>
    </modules>
</project>
```

这样，在根目录执行mvn clean package时，Maven根据根目录的pom.xml找到包括parent在内的共4个<module>，一次性全部编译。

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

私有仓库是指公司内部如果不希望把源码和jar包放到公网上，那么可以搭建私有仓库。私有仓库总是在公司内部使用，它只需要在本地的`~/.m2/settings.xml`中配置好，使用方式和中央仓位没有任何区别。

本地仓库

本地仓库是指把本地开发的项目“发布”在本地，这样其他项目可以通过本地仓库引用它。但是我们不推荐把自己的模块安装到Maven的本地仓库，因为每次修改某个模块的源码，都需要重新安装，非常容易出现版本不一致的情况。更好的方法是使用模块化编译，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。

小结

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。

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：

```
mvnw clean package
```

在Linux或macOS下运行时需要加上`./`：

```
./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

使用mvnw编译hello项目

小结

使用Maven Wrapper，可以为一个项目指定特定的Maven版本。

当我们使用`commons-logging`这些第三方开源库的时候，我们实际上是通过Maven自动下载它的jar包，并根据其`pom.xml`解析依赖，自动把相关依赖包都下载后加入到classpath。

那么问题来了：当我们自己写了一个牛逼的开源库时，非常希望别人也能使用，总不能直接放个jar包的链接让别人下载吧？

如果我们把自己的开源库放到Maven的repo中，那么，别人只需按标准引用`groupId:artifactId:version`，即可自动下载jar包以及相关依赖。因此，本节我们介绍如何发布一个库到Maven的repo中。

把自己的库发布到Maven的repo中有好几种方法，我们介绍3种最常用的方法。

以静态文件发布

如果我们观察一个中央仓库的Artifact结构，例如Commons Math，它的groupId是`org.apache.commons`，artifactId是`commons-math3`，以版本`3.6.1`为例，发布在中央仓库的文件夹路径就是https://repo1.maven.org/maven2/org/apache/commons/commons-math3/3.6.1/，在此文件夹下，`commons-math3-3.6.1.jar`就是发布的jar包，`commons-math3-3.6.1.pom`就是它的`pom.xml`描述文件，`commons-math3-3.6.1-sources.jar`是源代码，`commons-math3-3.6.1-javadoc.jar`是文档。其它以`.asc`、`.md5`、`.sha1`结尾的文件分别是GPG签名、MD5摘要和SHA-1摘要。

我们只要按照这种目录结构组织文件，它就是一个有效的Maven仓库。

我们以广受好评的开源项目how-to-become-rich为例，先创建Maven工程目录结构如下：

```
how-to-become-rich
├── maven-repo          <-- Maven本地文件仓库
├── pom.xml             <-- 项目文件
├── src
│   ├── main
│   │   ├── java        <-- 源码目录
│   │   └── resources   <-- 资源目录
│   └── test
│       ├── java        <-- 测试源码目录
│       └── resources   <-- 测试资源目录
└── target              <-- 编译输出目录
```

在`pom.xml`中添加如下内容：

```
<project ...>
    ...
    <distributionManagement>
        <repository>
```

```
            <id>local-repo-release</id>
            <name>GitHub Release</name>
            <url>file://${project.basedir}/maven-repo</url>
        </repository>
    </distributionManagement>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-source-plugin</artifactId>
                <executions>
                    <execution>
                        <id>attach-sources</id>
                        <phase>package</phase>
                        <goals>
                            <goal>jar-no-fork</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <artifactId>maven-javadoc-plugin</artifactId>
                <executions>
                    <execution>
                        <id>attach-javadocs</id>
                        <phase>package</phase>
                        <goals>
                            <goal>jar</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

注意到`<distributionManagement>`，它指示了发布的软件包的位置，这里的`<url>`是项目根目录下的`maven-repo`目录，在`<build>`中定义的两个插件`maven-source-plugin`和`maven-javadoc-plugin`分别用来创建源码和javadoc，如果不想发布源码，可以把对应的插件去掉。

我们直接在项目根目录下运行Maven命令`mvn clean package deploy`，如果一切顺利，我们就可以在`maven-repo`目录下找到部署后的所有文件如下：

```
maven-repo
└── com
    └── itranswarp
        └── rich
            └── how-to-become-rich
                ├── 1.0.0
                │   ├── how-to-become-rich-1.0.0-javadoc.jar
                │   ├── how-to-become-rich-1.0.0-javadoc.jar.md5
                │   ├── how-to-become-rich-1.0.0-javadoc.jar.sha1
                │   ├── how-to-become-rich-1.0.0-sources.jar
                │   ├── how-to-become-rich-1.0.0-sources.jar.md5
                │   ├── how-to-become-rich-1.0.0-sources.jar.sha1
                │   ├── how-to-become-rich-1.0.0.jar
                │   ├── how-to-become-rich-1.0.0.jar.md5
                │   ├── how-to-become-rich-1.0.0.jar.sha1
                │   ├── how-to-become-rich-1.0.0.pom
                │   ├── how-to-become-rich-1.0.0.pom.md5
                │   └── how-to-become-rich-1.0.0.pom.sha1
                ├── maven-metadata.xml
                ├── maven-metadata.xml.md5
                └── maven-metadata.xml.sha1
```

最后一步，是把这个工程推到GitHub上，并选择`Settings-GitHub Pages`，选择`master branch`启用Pages服务：

这样，把全部内容推送至GitHub后，即可作为静态网站访问Maven的repo，它的地址是https://michaelliao.github.io/how-to-become-rich/maven-repo/。版本`1.0.0`对应的jar包地址是：

```
https://michaelliao.github.io/how-to-become-rich/maven-repo/com/itranswarp/rich/how-to-become-rich/1.0.0/how-to-become-rich-1.0.0.jar
```

现在，如果其他人希望引用这个Maven包，我们可以告知如下依赖即可：

```
<dependency>
    <groupId>com.itranswarp.rich</groupId>
    <artifactId>how-to-become-rich</artifactId>
    <version>1.0.0</version>
</dependency>
```

但是，除了正常导入依赖外，对方还需要再添加一个`<repository>`的声明，即使用方完整的`pom.xml`如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>example</groupId>
    <artifactId>how-to-become-rich-usage</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>
    </properties>

    <repositories>
        <repository>
            <id>github-rich-repo</id>
            <name>The Maven Repository on Github</name>
            <url>https://michaelliao.github.io/how-to-become-rich/maven-repo/</url>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>com.itranswarp.rich</groupId>
            <artifactId>how-to-become-rich</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
</project>
```

在`<repository>`中，我们必须声明发布的Maven的repo地址，其中`<id>`和`<name>`可以任意填写，`<url>`填入GitHub Pages提供的地址+`/maven-repo/`后缀。现在，即可正常引用这个库并编写代码如下：

```
Millionaire millionaire = new Millionaire();
System.out.println(millionaire.howToBecomeRich());
```

有的童鞋会问，为什么使用`commons-logging`等第三方库时，并不需要声明repo地址？这是因为这些库都是发布到Maven中央仓库的，发布到中央仓库后，不需要告诉Maven仓库地址，因为它知道中央仓库的地址

默认是https://repo1.maven.org/maven2/，也可以通过~/.m2/settings.xml指定一个代理仓库地址以替代中央仓库来提高速度（参考依赖管理的Maven镜像）。

因为GitHub Pages并不会把我们发布的Maven包同步到中央仓库，所以自然使用方必须手动添加一个我们提供的仓库地址。

此外，通过GitHub Pages发布Maven repo时需要注意一点，即不要改动已发布的版本。因为Maven的仓库是不允许修改任何版本的，对一个库进行修改的唯一方法是发布一个新版本。但是通过静态文件的方式发布repo，实际上我们是可以修改jar文件的，但最好遵守规范，不要修改已发布版本。

通过Nexus发布到中央仓库

有的童鞋会问，能不能把自己的开源库发布到Maven的中央仓库，这样用户就不需要声明repo地址，可以直接引用，显得更专业。

当然可以，但我们不能直接发布到Maven中央仓库，而是通过曲线救国的方式，发布到central.sonatype.org，它会定期自动同步到Maven的中央仓库。Nexus是一个支持Maven仓库的软件，由Sonatype开发，有免费版和专业版两个版本，很多大公司内部都使用Nexus作为自己的私有Maven仓库，而这个central.sonatype.org相当于面向开源的一个Nexus公共服务。

所以，第一步是在central.sonatype.org上注册一个账号，注册链接非常隐蔽，可以自己先找找，找半小时没找到点这里查看攻略。

如果注册顺利并审核通过，会得到一个登录账号，然后，通过这个页面一步一步操作就可以成功地将自己的Artifact发布到Nexus上，再耐心等待几个小时后，你的Artifact就会出现在Maven的中央仓库中。

这里简单提一下发布重点与难点：

- 必须正确创建GPG签名，Linux和Mac下推荐使用gnupg2；
- 必须在~/.m2/settings.xml中配置好登录用户名和口令，以及GPG口令：

```
<settings ...>
    ...
    <servers>
        <server>
            <id>ossrh</id>
            <username>OSSRH-USERNAME</username>
            <password>OSSRH-PASSWORD</password>
        </server>
    </servers>
    <profiles>
        <profile>
            <id>ossrh</id>
            <activation>
                <activeByDefault>true</activeByDefault>
            </activation>
            <properties>
                <gpg.executable>gpg2</gpg.executable>
                <gpg.passphrase>GPG-PASSWORD</gpg.passphrase>
            </properties>
        </profile>
    </profiles>
</settings>
```

在待发布的Artifact的pom.xml中添加OSS的Maven repo地址，以及maven-jar-plugin、maven-source-plugin、maven-javadoc-plugin、maven-gpg-plugin、nexus-staging-maven-plugin：

```
<project ...>
    ...
    <distributionManagement>
        <snapshotRepository>
            <id>ossrh</id>
            <url>https://oss.sonatype.org/content/repositories/snapshots</url>
        </snapshotRepository>

        <repository>
            <id>ossrh</id>
            <name>Nexus Release Repository</name>
            <url>http://oss.sonatype.org/service/local/staging/deploy/maven2/</url>
        </repository>
    </distributionManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <executions>
                    <execution>
                        <goals>
                            <goal>jar</goal>
                            <goal>test-jar</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-source-plugin</artifactId>
                <executions>
                    <execution>
                        <id>attach-sources</id>
                        <goals>
                            <goal>jar-no-fork</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-javadoc-plugin</artifactId>
                <executions>
                    <execution>
                        <id>attach-javadocs</id>
                        <goals>
                            <goal>jar</goal>
                        </goals>
                        <configuration>
                            <additionalOption>
                                <additionalOption>-Xdoclint:none</additionalOption>
                            </additionalOption>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-gpg-plugin</artifactId>
                <executions>
                    <execution>
                        <id>sign-artifacts</id>
                        <phase>verify</phase>
                        <goals>
                            <goal>sign</goal>
                        </goals>
                    </execution>
                </executions>
```

```
            </plugin>
            <plugin>
                <groupId>org.sonatype.plugins</groupId>
                <artifactId>nexus-staging-maven-plugin</artifactId>
                <version>1.6.3</version>
                <extensions>true</extensions>
                <configuration>
                    <serverId>ossrh</serverId>
                    <nexusUrl>https://oss.sonatype.org/</nexusUrl>
                    <autoReleaseAfterClose>true</autoReleaseAfterClose>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

最后执行命令`mvn clean package deploy`即可发布至central.sonatype.org。

此方法前期需要复杂的申请账号和项目的流程，后期需要安装调试GPG，但只要跑通流程，后续发布都只需要一行命令。

发布到私有仓库

通过`nexus-staging-maven-plugin`除了可以发布到central.sonatype.org外，也可以发布到私有仓库，例如，公司内部自己搭建的Nexus服务器。

如果没有私有Nexus服务器，还可以发布到GitHub Packages。GitHub Packages是GitHub提供的仓库服务，支持Maven、NPM、Docker等。使用GitHub Packages时，无论是发布Artifact，还是引用已发布的Artifact，都需要明确的授权Token，因此，GitHub Packages只能作为私有仓库使用。

在发布前，我们必须首先登录后在用户的`Settings-Developer settings-Personal access tokens`中创建两个Token，一个用于发布，一个用于使用。发布Artifact的Token必须有`repo、write:packages`和`read:packages`权限：

使用Artifact的Token只需要`read:packages`权限。

在发布端，把GitHub的用户名和发布Token写入`~/.m2/settings.xml`配置中：

```
<settings ...>
    ...
    <servers>
        <server>
            <id>github-release</id>
            <username>GITHUB-USERNAME</username>
            <password>f052...c21f</password>
        </server>
    </servers>
</settings>
```

然后，在需要发布的Artifact的`pom.xml`中，添加一个`<repository>`声明：

```
<project ...>
    ...
    <distributionManagement>
        <repository>
            <id>github-release</id>
            <name>GitHub Release</name>
            <url>https://maven.pkg.github.com/michaelliao/complex</url>
        </repository>
    </distributionManagement>
</project>
```

注意到`<id>`和`~/.m2/settings.xml`配置中的`<id>`要保持一致，因为发布时Maven根据id找到用于登录的用户名和Token，才能成功上传文件到GitHub。我们直接通过命令`mvn clean package deploy`部署，成功后，在GitHub用户页面可以看到该Artifact：

完整的配置请参考complex项目，这是一个非常简单的支持复数运算的库。

使用该Artifact时，因为GitHub的Package只能作为私有仓库使用，所以除了在使用方的`pom.xml`中声明`<repository>`外：

```
<project ...>
    ...
    <repositories>
        <repository>
            <id>github-release</id>
            <name>GitHub Release</name>
            <url>https://maven.pkg.github.com/michaelliao/complex</url>
        </repository>
    </repositories>

    <dependencies>
        <dependency>
            <groupId>com.itranswarp</groupId>
            <artifactId>complex</artifactId>
            <version>1.0.0</version>
        </dependency>
    </dependencies>
    ...
</project>
```

还需要把有读权限的Token配置到`~/.m2/settings.xml`文件中。

练习

使用maven-deploy-plugin把Artifact发布到本地。

小结

使用Maven发布一个Artifact时：

- 可以发布到本地，然后推送到远程Git库，由静态服务器提供基于网页的repo服务，使用方必须声明repo地址；
- 可以发布到central.sonatype.org，并自动同步到Maven中央仓库，需要前期申请账号以及本地配置；
- 可以发布到GitHub Packages作为私有仓库使用，必须提供Token以及正确的权限才能发布和使用。

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。

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	
表示层	应用层
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。

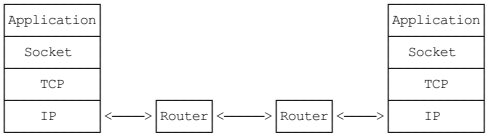
选择UDP协议时，传输的数据通常是能容忍丢失的，例如，一些语音视频通信的应用会选择UDP协议。

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，它泛指互联网协议族；
- 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。

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：

```
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(6666); // 监听指定端口
        System.out.println("server is running...");
        for (;;) {
            Socket sock = ss.accept();
            System.out.println("connected from " + sock.getRemoteSocketAddress());
            Thread t = new Handler(sock);
            t.start();
        }
    }
}

class Handler extends Thread {
    Socket sock;

    public Handler(Socket sock) {
        this.sock = sock;
    }

    @Override
    public void run() {
        try (InputStream input = this.sock.getInputStream()) {
            try (OutputStream output = this.sock.getOutputStream()) {
                handle(input, output);
            }
        } catch (Exception e) {
            try {
                this.sock.close();
            } catch (IOException ioe) {
            }
            System.out.println("client disconnected.");
        }
    }

    private void handle(InputStream input, OutputStream output) throws IOException {
        var writer = new BufferedWriter(new OutputStreamWriter(output, StandardCharsets.UTF_8));
        var reader = new BufferedReader(new InputStreamReader(input, StandardCharsets.UTF_8));
        writer.write("hello\n");
        writer.flush();
        for (;;) {
            String s = reader.readLine();
            if (s.equals("bye")) {
                writer.write("bye\n");
                writer.flush();
                break;
            }
            writer.write("ok: " + s + "\n");
            writer.flush();
        }
    }
}
```

服务器端通过代码：

```
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：

```
for (;;) {
    Socket sock = ss.accept();
    Thread t = new Handler(sock);
    t.start();
```

```
}
```

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：

```
public class Client {
    public static void main(String[] args) throws IOException {
        Socket sock = new Socket("localhost", 6666); // 连接指定服务器和端口
        try (InputStream input = sock.getInputStream()) {
            try (OutputStream output = sock.getOutputStream()) {
                handle(input, output);
            }
        }
        sock.close();
        System.out.println("disconnected.");
    }

    private static void handle(InputStream input, OutputStream output) throws IOException {
        var writer = new BufferedWriter(new OutputStreamWriter(output, StandardCharsets.UTF_8));
        var reader = new BufferedReader(new InputStreamReader(input, StandardCharsets.UTF_8));
        Scanner scanner = new Scanner(System.in);
        System.out.println("[server] " + reader.readLine());
        for (;;) {
            System.out.print(">>> "); // 打印提示
            String s = scanner.nextLine(); // 读取一行输入
            writer.write(s);
            writer.newLine();
            writer.flush();
            String resp = reader.readLine();
            System.out.println("<<< " + resp);
            if (resp.equals("bye")) {
                break;
            }
        }
    }
}
```

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

使用Socket实现服务器和客户端通信

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。

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

使用UDP实现服务器和客户端通信

小结

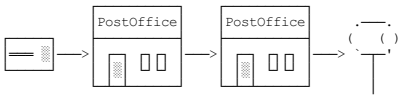
使用UDP协议通信时，服务器和客户端双方无需建立连接：

- 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。

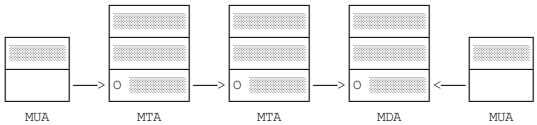
Email就是电子邮件。电子邮件的应用已经有几十年的历史了，我们熟悉的邮箱地址比如`abc@example.com`，邮件软件比如Outlook都是用来收发邮件的。

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> Hi, xxx 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：

```
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：

```
DEBUG SMTP: AUTH LOGIN failed
Exception in thread "main" javax.mail.AuthenticationFailedException: 535 5.7.3 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：

```
DEBUG SMTP: MessagingException while sending, THROW:
com.sun.mail.smtp.SMTPSendFailedException: 554 5.2.0 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：

```
DEBUG SMTP: MessagingException while sending, THROW:
com.sun.mail.smtp.SMTPSendFailedException: 554 DT:SPM
```

练习

使用SMTP发送邮件

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。

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

使用POP3接收邮件

小结

使用Java接收Email时，可以用POP3协议或IMAP协议。

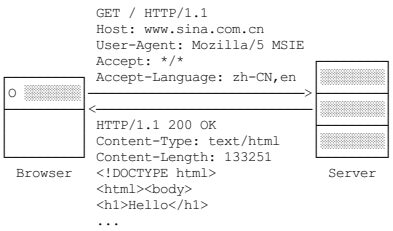
使用POP3协议时，需要用Maven引入JavaMail依赖，并确定POP3服务器的域名／端口／是否使用SSL等，然后，调用相关API接收Email。

设置debug模式可以查看通信详细内容，便于排查错误。

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：



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来识别请求是发给哪个网站的；
- 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判断客户端类型是IE还是Chrome，是Firefox还是一个Python爬虫；
- 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：

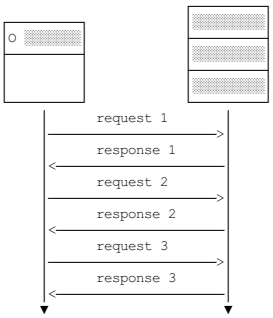
- 1xx：表示一个提示性响应，例如101表示将切换协议，常见于WebSocket连接；
- 2xx：表示一个成功的响应，例如200表示成功，206表示只发送了部分内容；
- 3xx：表示一个重定向的响应，例如301表示永久重定向，303表示客户端应该按指定路径重新发送请求；
- 4xx：表示一个因为客户端问题导致的错误响应，例如400表示因为Content-Type等各种原因导致的无效请求，404表示指定的路径不存在；
- 5xx：表示一个因为服务器问题导致的错误响应，例如500表示服务器内部故障，503表示服务器暂时无法响应。

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)
```

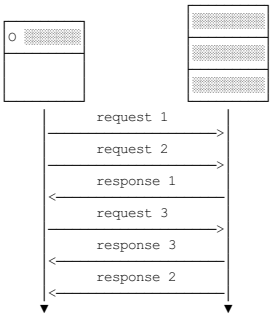
因此，服务器总是被动地接收客户端的一个HTTP请求，然后响应它。客户端则根据需要发送若干个HTTP请求。

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：



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如何使用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) + "...");
    }
}
```

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, HttpResponse.BodyHandlers.ofString());
String s = response.body();
```

可见发送POST数据也十分简单。

练习

使用HttpClient

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。

Java的RMI远程调用是指，一个JVM中的代码可以通过网络实现远程调用另一个JVM的某个方法。RMI是Remote Method Invocation的缩写。

提供服务的一方我们称之为服务器，而实现远程调用的一方我们称之为客户端。

我们先来实现一个最简单的RMI：服务器会提供一个WorldClock服务，允许客户端获取指定时区的时间，即允许客户端调用下面的方法：

```
LocalDateTime getLocalDateTime(String zoneId);
```

要实现RMI，服务器和客户端必须共享同一个接口。我们定义一个WorldClock接口，代码如下：

```
public interface WorldClock extends Remote {
    LocalDateTime getLocalDateTime(String zoneId) throws RemoteException;
}
```

Java的RMI规定此接口必须派生自java.rmi.Remote，并在每个方法声明抛出RemoteException。

下一步是编写服务器的实现类，因为客户端请求的调用方法getLocalDateTime()最终会通过这个实现类返回结果。实现类WorldClockService代码如下：

```
public class WorldClockService implements WorldClock {
    @Override
    public LocalDateTime getLocalDateTime(String zoneId) throws RemoteException {
        return LocalDateTime.now(ZoneId.of(zoneId)).withNano(0);
    }
}
```

现在，服务器端的服务相关代码就编写完毕。我们需要通过Java RMI提供的一系列底层支持接口，把上面编写的服务以RMI的形式暴露在网络上，客户端才能调用：

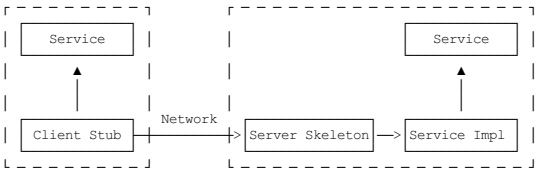
```
public class Server {
    public static void main(String[] args) throws RemoteException {
        System.out.println("create World clock remote service...");
        // 实例化一个WorldClock:
        WorldClock worldClock = new WorldClockService();
        // 将此服务转换为远程服务接口:
        WorldClock skeleton = (WorldClock) UnicastRemoteObject.exportObject(worldClock, 0);
        // 将RMI服务注册到1099端口:
        Registry registry = LocateRegistry.createRegistry(1099);
        // 注册此服务，服务名为"WorldClock":
        registry.rebind("WorldClock", skeleton);
    }
}
```

上述代码主要目的是通过RMI提供的相关类，将我们自己的WorldClock实例注册到RMI服务上。RMI的默认端口是1099，最后一步注册服务时通过rebind()指定服务名称为"WorldClock"。

下一步我们就可以编写客户端代码。RMI要求服务器和客户端共享同一个接口，因此我们要把WorldClock.java这个接口文件复制到客户端，然后在客户端实现RMI调用：

```
public class Client {
    public static void main(String[] args) throws RemoteException, NotBoundException {
        // 连接到服务器localhost，端口1099:
        Registry registry = LocateRegistry.getRegistry("localhost", 1099);
        // 查找名称为"WorldClock"的服务并强制转型为WorldClock接口:
        WorldClock worldClock = (WorldClock) registry.lookup("WorldClock");
        // 正常调用接口方法:
        LocalDateTime now = worldClock.getLocalDateTime("Asia/Shanghai");
        // 打印调用结果:
        System.out.println(now);
    }
}
```

先运行服务器，再运行客户端。从运行结果可知，因为客户端只有接口，并没有实现类，因此，客户端获得的接口方法返回值实际上是通过网络从服务器端获取的。整个过程实际上非常简单，对客户端来说，客户端持有的WorldClock接口实际上对应了一个“实现类”，它是由Registry内部动态生成的，并负责把方法调用通过网络传递到服务器端。而服务器端接收网络调用的服务并不是我们自己编写的WorldClockService，而是Registry自动生成的代码。我们把客户端的“实现类”称为stub，而服务器端的网络服务类称为skeleton，它会真正调用服务器端的WorldClockService，获取结果，然后把结果通过网络传递给客户端。整个过程由RMI底层负责实现序列化和反序列化：



Java的RMI严重依赖序列化和反序列化，而这种情况下可能会造成严重的安全漏洞，因为Java的序列化和反序列化不但涉及到数据，还涉及到二进制的字节码，即使使用白名单机制也很难保证100%排除恶意构造的字节码。因此，使用RMI时，双方必须是内网互相信任的机器，不要把1099端口暴露在公网上作为对外服务。

此外，Java的RMI调用机制决定了双方必须是Java程序，其他语言很难调用Java的RMI。如果要使用不同语言进行RPC调用，可以选择更通用的协议，例如gRPC。

练习

使用RMI远程调用

小结

Java提供了RMI实现远程方法调用：

RMI通过自动生成stub和skeleton实现网络调用，客户端只需要查找服务并获得接口实例，服务器端只需要编写实现类并注册为服务；

RMI的序列化和反序列化可能会造成安全漏洞，因此调用双方必须是内网互相信任的机器，不要把1099端口暴露在公网上作为对外服务。

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。

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。

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。

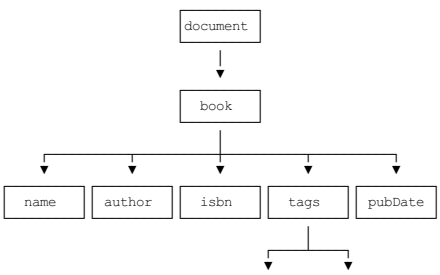
我们先来看如何使用DOM来读取XML。

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;
    case Node.ATTRIBUTE_NODE: // 属性
        System.out.println("Attr: " + n.getNodeName() + " = " + n.getNodeValue());
        break;
    default: // 其他
        System.out.println("NodeType: " + n.getNodeType() + ", NodeName: " + n.getNodeName());
    }
    for (Node child = n.getFirstChild(); child != null; child = child.getNextSibling()) {
        printNode(child, indent + 1);
    }
}
```

解析结构如下：

```
Document: #document
 Element: book
  Text: #text =

  Element: name
   Text: #text = Java核心技术
  Text: #text =

  Element: author
   Text: #text = Cay S. Horstmann
  Text: #text =
  ...
```

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

使用DOM解析XML

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。

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

使用SAX解析XML

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。

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：

- com.fasterxml.jackson.dataformat:jackson-dataformat-xml:2.10.1
- org.codehaus.woodstox:woodstox-core-asl:4.4.1

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

使用Jackson解析XML

小结

使用Jackson解析XML，可以直接把XML解析为JavaBean，十分方便。

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：

- com.fasterxml.jackson.core:jackson-databind:2.10.0

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的数据格式定义至Maven：

- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:2.10.0

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

使用Jackson解析JSON

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。

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。

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：

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：

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：

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：

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。

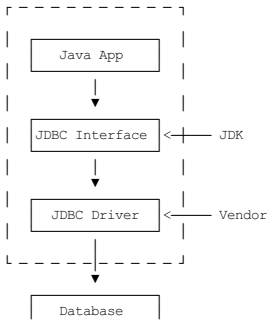
注：如果MySQL的版本≥5.5.3，可以把编码设置为`utf8mb4`，`utf8mb4`和`utf8`完全兼容，但它支持最新的Unicode标准，可以显示emoji字符。

JDBC

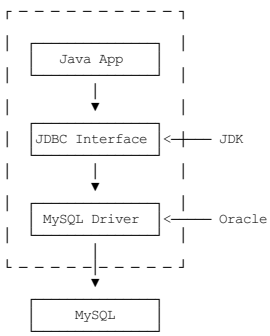
什么是JDBC？JDBC是Java DataBase Connectivity的缩写，它是Java程序访问数据库的标准接口。

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。

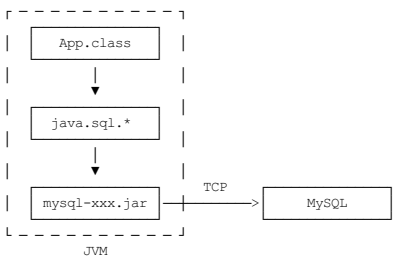
例如，我们在Java代码中如果要访问MySQL，那么必须编写代码操作JDBC接口。注意到JDBC接口是Java标准库自带的，所以可以直接编译。而具体的JDBC驱动是由数据库厂商提供的，例如，MySQL的JDBC驱动由Oracle提供。因此，访问某个具体的数据库，我们只需要引入该厂商提供的JDBC驱动，就可以通过JDBC接口来访问，这样保证了Java程序编写的是一套数据库访问代码，却可以访问各种不同的数据库，因为他们都提供了标准的JDBC驱动：



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。

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=1")) {
            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);
                int gender = rs.getInt(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 = ps.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

使用JDBC查询数据库

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。

数据库操作总结起来就四个字：增删改查，行话叫CRUD：Create，Retrieve，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, JDBC_PASSWORD)) {
    try (PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO students (id, grade, name, gender) VALUES (?,?,?,?)")) {
        ps.setObject(1, 999); // 注意：索引从1开始
        ps.setObject(2, 1); // grade
        ps.setObject(3, "Bob"); // name
        ps.setObject(4, "M"); // gender
        int n = ps.executeUpdate(); // 1
    }
}
```

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, JDBC_PASSWORD)) {
    try (PreparedStatement ps = conn.prepareStatement(
            "INSERT INTO students (grade, name, gender) VALUES (?,?,?)",
            Statement.RETURN_GENERATED_KEYS)) {
        ps.setObject(1, 1); // grade
        ps.setObject(2, "Bob"); // name
        ps.setObject(3, "M"); // gender
        int n = ps.executeUpdate(); // 1
        try (ResultSet rs = ps.getGeneratedKeys()) {
            if (rs.next()) {
                long id = rs.getLong(1); // 注意：索引从1开始
            }
        }
    }
}
```

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, JDBC_PASSWORD)) {
    try (PreparedStatement ps = conn.prepareStatement("UPDATE students SET name=? WHERE id=?")) {
        ps.setObject(1, "Bob"); // 注意：索引从1开始
        ps.setObject(2, 999);
        int n = ps.executeUpdate(); // 返回更新的行数
    }
}
```

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, JDBC_PASSWORD)) {
    try (PreparedStatement ps = conn.prepareStatement("DELETE FROM students WHERE id=?")) {
        ps.setObject(1, 999); // 注意：索引从1开始
        int n = ps.executeUpdate(); // 删除的行数
    }
}
```

练习

使用JDBC更新数据库

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。

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：

```
UPDATE accounts SET balance = balance - 100 WHERE id=123 AND balance >= 100;
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：

```
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

使用数据库事务

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。

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 (Student s : students) {
        ps.setString(1, s.name);
        ps.setBoolean(2, s.gender);
        ps.setInt(3, s.grade);
        ps.setInt(4, s.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。

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

使用HikariCP连接池

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。

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。

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));
    }
}
```

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

使用Lambda表达式实现忽略大小写排序

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。

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);
    }
}
```

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));
    }
}
```

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;
    }
}
```

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

使用方法引用实现忽略大小写排序

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。

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：

```
List<BigInteger> list = ??? // 全体自然数?
```

但是，用Stream可以做到。写法如下：

```
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：

```
Stream<BigInteger> naturals = createNaturalStream(); // 全体自然数
Stream<BigInteger> streamNxN = naturals.map(n -> n.multiply(n)); // 全体自然数的平方
```

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：

```
Stream<BigInteger> naturals = createNaturalStream();
naturals.map(n -> n.multiply(n)) // 1, 4, 9, 16, 25...
        .limit(100)
        .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。

```
Stream<BigInteger> naturals = createNaturalStream(); // 不计算
Stream<BigInteger> s2 = naturals.map(BigInteger::multiply); // 不计算
Stream<BigInteger> s3 = s2.limit(100); // 不计算
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：

```
createNaturalStream()
    .map(BigInteger::multiply)
    .limit(100)
    .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：

```
int result = createNaturalStream() // 创建Stream
             .filter(n -> n % 2 == 0) // 任意个转换
             .map(n -> n * n) // 任意个转换
             .limit(100) // 任意个转换
             .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。

要使用Stream，就必须先创建它。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);
    }
}
```

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);
    }
}
```

把数组变成Stream使用Arrays.stream()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：

```
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;
    }
}
```

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`Stream<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, ...
```

FibonacciStream练习

小结

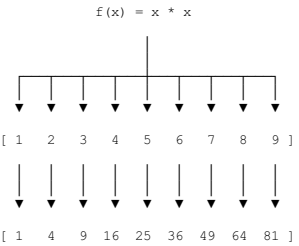
创建`Stream`的方法有：

- 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`。

`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：



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
    }
}
```

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。

map练习

小结

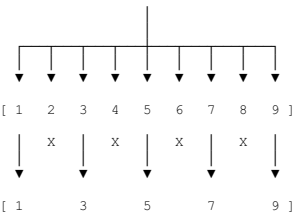
`map()`方法用于将一个`Stream`的每个元素映射成另一个元素并转换成一个新的`Stream`：

可以将一种元素类型转换成另一种元素类型。

`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：



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);
    }
}
```

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);
    }
}
```

练习

请使用filter()过滤出成绩及格的同学，并打印出名字。

filter练习

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。

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
    }
}
```

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
    }
}
```

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);
        });
    }
}
```

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。

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;
    }
}
```

因为s1是一个Long类型的序列，它的元素高达922亿亿个，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：

```
Stream<Long> s1 = Stream.generate(new NatualSupplier());
Stream<Long> s2 = s1.map(n -> n * n);
Stream<Long> s3 = s2.map(n -> n - 1);
Stream<Long> s4 = s3.limit(10);
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先过滤掉空字符串，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);
    }
}
```

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”：

```
List<String> list = List.of("Apple", "Banana", "Orange");
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);
    }
}
```

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);
    }
}
```

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。

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);
    }
}
```

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

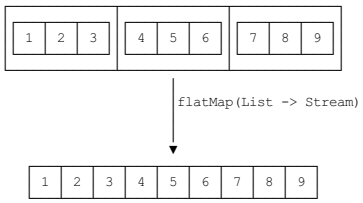
如果Stream的元素是集合：

```
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()。

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 Principle）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。

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。

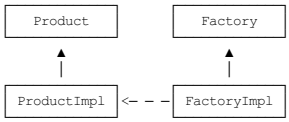
创建型模式包括：

- 工厂方法：Factory Method
- 抽象工厂：Abstract Factory
- 建造者：Builder
- 原型：Prototype
- 单例：Singleton

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：



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 NumberFactory {
    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);

    // 获取工厂实例:
    static NumberFactory getFactory() {
        return impl;
    }

    static NumberFactory impl = new NumberFactoryImpl();
}
```

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：

```
NumberFactory factory = NumberFactory.getFactory();
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：

```
public class NumberFactory {
    public static Number parse(String s) {
        return new BigDecimal(s);
    }
}
```

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：

```
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：

```
public final class Integer {
    public static Integer valueOf(int i) {
        if (i >= IntegerCache.low && i <= IntegerCache.high)
            return IntegerCache.cache[i + (-IntegerCache.low)];
        return new Integer(i);
    }
    ...
}
```

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) {
        ...
    }
}
```

静态工厂方法练习

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。

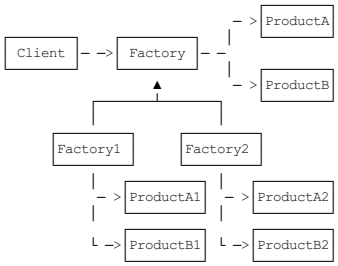
实际更常用的是更简单的静态工厂方法，它允许工厂内部对创建产品进行优化。

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。

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 = factory.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：

```
public interface AbstractFactory {
    public static AbstractFactory createFactory(String name) {
        if (name.equalsIgnoreCase("fast")) {
            return new FastFactory();
        } else if (name.equalsIgnoreCase("good")) {
            return new GoodFactory();
        } else {
            throw new IllegalArgumentException("Invalid factory name");
        }
    }
}
```

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`。

Abstract Factory模式

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。

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

使用Builder模式实现一个URLBuilder

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。

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。

原型模式练习

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。

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：

1. 只有private构造方法，确保外部无法实例化；
2. 通过private static变量持有唯一实例，保证全局唯一性；
3. 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：

Singleton练习

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。

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

> 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：

1. 实现目标接口，这里是`Runnable`：

2. 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；
3. 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。

Adapter模式

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();
    }
}
```

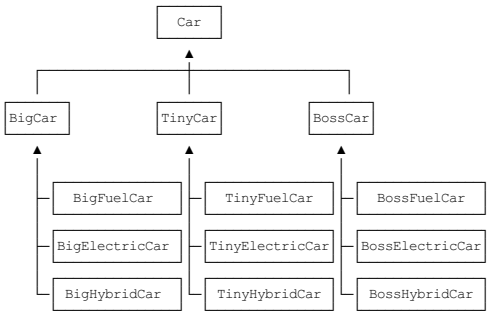
在Adapter内部将B接口的调用“转换”为对A接口的调用。

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。

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";
    }
}
```

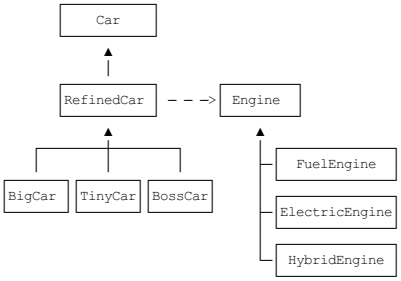
而针对每一种引擎，继承自`Engine`，例如`HybridEngine`：

```
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。

桥接模式练习

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。

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 + " -->";
 }
}
```

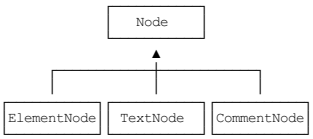
通过ElementNode、TextNode和CommentNode，我们就可以构造出一颗树：

```
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：



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

使用Composite模式构造XML

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。

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。

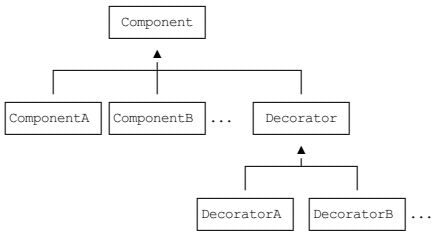
例如：给FileInputStream增加缓冲和解压缩功能，用Decorator模式写出来如下：

```
// 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：



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，它需要从TextNode直接继承：

```
public class SpanNode implements TextNode {
    private String text;

    public void setText(String text) {
        this.text = text;
    }

    public String getText() {
        return "<span>" + text + "</span>";
    }
}
```

紧接着，为了实现Decorator模式，需要有一个抽象的Decorator类：

```
public abstract class NodeDecorator implements TextNode {
    protected final TextNode target;

    protected NodeDecorator(TextNode target) {
        this.target = target;
    }

    public void setText(String text) {
        this.target.setText(text);
    }
}
```

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：

```
public class BoldDecorator extends NodeDecorator {
    public BoldDecorator(TextNode target) {
        super(target);
    }

    public String getText() {
        return "<b>" + target.getText() + "</b>";
    }
}
```

类似的，可以继续加ItalicDecorator、UnderlineDecorator等。客户端可以自由组合这些Decorator：

```
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。

Decorator练习

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。

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：

1. 向工商局申请公司营业执照；
2. 在银行开设账户；
3. 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。

Facade模式练习

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。

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
    }
}
```

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：

```
public class Student {
    // 持有缓存:
    private static final Map<String, Student> cache = new HashMap<>();

    // 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

使用享元模式实现缓存

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。

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, jdbcPassword);
System.out.println("get lazy connection...");
try (Connection conn1 = lazyDataSource.getConnection()) {
    // 并没有实际打开真正的Connection
}
System.out.println("get lazy connection...");
try (Connection conn2 = lazyDataSource.getConnection()) {
    try (PreparedStatement ps = conn2.prepareStatement("SELECT * FROM students")) { // 打开了真正的Connection
        try (ResultSet rs = ps.executeQuery()) {
            while (rs.next()) {
```

```
                System.out.println(rs.getString("name"));
            }
        }
    }
}
```

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：

```
public abstract class AbstractConnectionProxy implements Connection {

    // 抽象方法获取实际的Connection:
    protected abstract Connection getRealConnection();

    // 实现Connection接口的每一个方法:
    public Statement createStatement() throws SQLException {
        return getRealConnection().createStatement();
    }

    public PreparedStatement prepareStatement(String sql) throws SQLException {
        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, jdbcPassword);
try (Connection conn = pooledDataSource.getConnection()) {
}
try (Connection conn = pooledDataSource.getConnection()) {
    // 获取到的是同一个Connection
}
try (Connection conn = pooledDataSource.getConnection()) {
    // 获取到的是同一个Connection
}
```

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。

```
public class PooledConnectionProxy extends AbstractConnectionProxy {
    // 实际的Connection:
    Connection target;
    // 空闲队列:
    Queue<PooledConnectionProxy> idleQueue;
```

```
    public PooledConnectionProxy(Queue<PooledConnectionProxy> idleQueue, Connection target) {
        this.idleQueue = idleQueue;
        this.target = target;
    }

    public void close() throws SQLException {
        System.out.println("Fake close and released to idle queue for future reuse: " + target);
        // 并没有调用实际Connection的close()方法,
        // 而是把自己放入空闲队列:
        idleQueue.offer(this);
    }

    protected Connection getRealConnection() {
        return target;
    }
}
```

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

使用代理模式编写一个JDBC连接池

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。

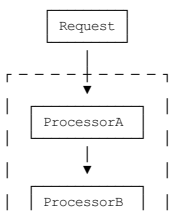
行为型模式主要涉及算法和对象间的职责分配。通过使用对象组合，行为型模式可以描述一组对象应该如何协作来完成一个整体任务。

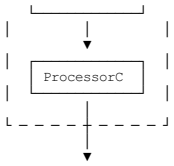
行为型模式有：

- 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

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

使用责任链模式实现审批

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。

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：

```
public class TextEditor {
    private StringBuilder buffer = new StringBuilder();

    public void copy() {
        ...
    }

    public void paste() {
        String text = getFromClipBoard();
        add(text);
    }

    public void add(String s) {
        buffer.append(s);
    }

    public void delete() {
        if (buffer.length() > 0) {
            buffer.deleteCharAt(buffer.length() - 1);
        }
    }

    public String getState() {
        return buffer.toString();
    }
}
```

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：

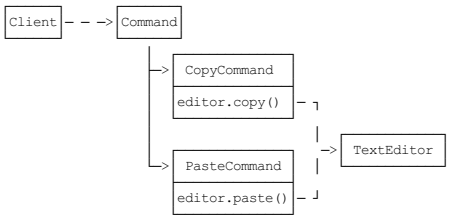
```
public class CopyCommand implements Command {
    // 持有执行者对象:
    private TextEditor receiver;

    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();
```

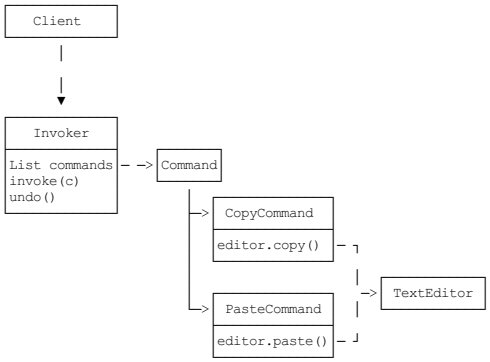
实际上，使用命令模式，确实增加了系统的复杂度。如果需求很简单，那么直接调用显然更直观而且更简单。

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。

命令模式练习

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。

给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

解释器模式（Interpreter）是一种针对特定问题设计的一种解决方案。例如，匹配字符串的时候，由于匹配条件非常灵活，使得通过代码来实现非常不灵活。举个例子，针对以下的匹配条件：

- 以+开头的数字表示的区号和电话号码，如+861012345678；
- 以英文开头，后接英文和数字，并以.分隔的域名，如www.liaoxuefeng.com；
- 以/开头的文件路径，如/path/to/file.txt；
- ...

因此，需要一种通用的表示方法——正则表达式来进行匹配。正则表达式就是一个字符串，但要把正则表达式解析为语法树，然后再匹配指定的字符串，就需要一个解释器。

实现一个完整的正则表达式的解释器非常复杂，但是使用解释器模式却很简单：

```
String s = "+861012345678";
System.out.println(s.matches("^\\+\\d+$"));
```

类似的，当我们使用JDBC时，执行的SQL语句虽然是字符串，但最终需要数据库服务器的SQL解释器来把SQL“翻译”成数据库服务器能执行的代码，这个执行引擎也非常复杂，但对于使用者来说，仅仅需要写出SQL字符串即可。

练习

请实现一个简单的解释器，它可以以SLF4J的日志格式输出字符串：

```
log("[{}] start {} at {}...", LocalTime.now().withNano(0), "engine", LocalDate.now());
// [11:02:18] start engine at 2020-02-21...
```

解释器模式练习

小结

解释器模式通过抽象语法树实现对用户输入的解释执行。

解释器模式的实现通常非常复杂，且一般只能解决一类特定问题。

> 提供一种方法顺序访问一个聚合对象中的各个元素，而又不需要暴露该对象的内部表示。

迭代器模式（Iterator）实际上在Java的集合类中已经广泛使用了。我们以`List`为例，要遍历`ArrayList`，即使我们知道它的内部存储了一个`Object[]`数组，也不应该直接使用数组索引去遍历，因为这样需要了解集合内部的存储结构。如果使用`Iterator`遍历，那么，`ArrayList`和`LinkedList`都可以以一种统一的接口来遍历：

```
List<String> list = ...
for (Iterator<String> it = list.iterator(); it.hasNext(); ) {
    String s = it.next();
}
```

实际上，因为Iterator模式十分有用，因此，Java允许我们直接把任何支持`Iterator`的集合对象用`foreach`循环写出来：

```
List<String> list = ...
for (String s : list) {

}
```

然后由Java编译器完成Iterator模式的所有循环代码。

虽然我们对如何使用Iterator有了一定了解，但如何实现一个Iterator模式呢？我们以一个自定义的集合为例，通过Iterator模式实现倒序遍历：

```
public class ReverseArrayCollection<T> implements Iterable<T> {
    // 以数组形式持有集合:
    private T[] array;

    public ReverseArrayCollection(T... objs) {
        this.array = Arrays.copyOfRange(objs, 0, objs.length);
    }

    public Iterator<T> iterator() {
        return ???;
    }
}
```

实现Iterator模式的关键是返回一个`Iterator`对象，该对象知道集合的内部结构，因为它可以实现倒序遍历。我们使用Java的内部类实现这个`Iterator`：

```
public class ReverseArrayCollection<T> implements Iterable<T> {
    private T[] array;

    public ReverseArrayCollection(T... objs) {
        this.array = Arrays.copyOfRange(objs, 0, objs.length);
    }

    public Iterator<T> iterator() {
        return new ReverseIterator();
    }

    class ReverseIterator implements Iterator<T> {
        // 索引位置:
        int index;

        public ReverseIterator() {
            // 创建Iterator时,索引在数组末尾:
            this.index = ReverseArrayCollection.this.array.length;
        }

        public boolean hasNext() {
            // 如果索引大于0,那么可以移动到下一个元素(倒序往前移动):
            return index > 0;
        }

        public T next() {
            // 将索引移动到下一个元素并返回(倒序往前移动):
            index--;
            return array[index];
        }
    }
}
```

使用内部类的好处是内部类隐含地持有一个它所在对象的`this`引用，可以通过`ReverseArrayCollection.this`引用到它所在的集合。上述代码实现的逻辑非常简单，但是实际应用时，如果考虑到多线程访问，当一个线程正在迭代某个集合，而另一个线程修改了集合的内容时，是否能继续安全地迭代，还是抛出`ConcurrentModificationException`，就需要更仔细地设计。

练习

使用Iterator模式实现集合的倒序遍历

小结

Iterator模式常用于遍历集合，它允许集合提供一个统一的`Iterator`接口来遍历元素，同时保证调用者对集合内部的数据结构一无所知，从而使得调用者总是以相同的接口遍历各种不同类型的集合。

> 用一个中介对象来封装一系列的对象交互。中介者使各个对象不需要显式地相互引用，从而使其耦合松散，而且可以独立地改变它们之间的交互。

中介模式（Mediator）又称调停者模式，它的目的是把多方会谈变成双方会谈，从而实现多方的松耦合。

有些童鞋听到中介立刻想到房产中介，立刻气不打一处来。这个中介模式与房产中介还真有点像，所以消消气，先看例子。

考虑一个简单的点餐输入：

 汉堡
 鸡块
 薯条
 咖啡

选择全部 取消所有 反选

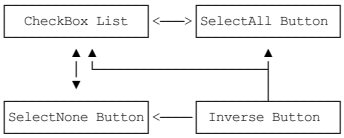
这个小系统有4个参与对象：

- 多选框；

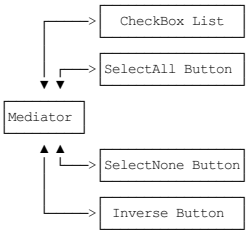
- “选择全部”按钮；
- “取消所有”按钮；
- “反选”按钮。

它的复杂性在于，当多选框变化时，它会影响“选择全部”和“取消所有”按钮的状态（是否可点击），当用户点击某个按钮时，例如“反选”，除了会影响多选框的状态，它又可能影响“选择全部”和“取消所有”按钮的状态。

所以这是一个多方会谈，逻辑写起来很复杂：



如果我们引入一个中介，把多方会谈变成多个双方会谈，虽然多了一个对象，但对象之间的关系就变简单了：



下面我们用中介模式来实现各个UI组件的交互。首先把UI组件给画出来：

```
public class Main {
    public static void main(String[] args) {
        new OrderFrame("Hanburger", "Nugget", "Chip", "Coffee");
    }
}

class OrderFrame extends JFrame {
    public OrderFrame(String... names) {
        setTitle("Order");
        setSize(460, 200);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        Container c = getContentPane();
        c.setLayout(new FlowLayout(FlowLayout.LEADING, 20, 20));
        c.add(new JLabel("Use Mediator Pattern"));
        List<JCheckBox> checkboxList = addCheckBox(names);
        JButton selectAll = addButton("Select All");
        JButton selectNone = addButton("Select None");
        selectNone.setEnabled(false);
        JButton selectInverse = addButton("Inverse Select");
        new Mediator(checkBoxList, selectAll, selectNone, selectInverse);
        setVisible(true);
    }

    private List<JCheckBox> addCheckBox(String... names) {
        JPanel panel = new JPanel();
        panel.add(new JLabel("Menu:"));
        List<JCheckBox> list = new ArrayList<>();
        for (String name : names) {
            JCheckBox checkbox = new JCheckBox(name);
            list.add(checkbox);
            panel.add(checkbox);
        }
        getContentPane().add(panel);
        return list;
    }

    private JButton addButton(String label) {
        JButton button = new JButton(label);
        getContentPane().add(button);
        return button;
    }
}
```

然后，我们设计一个Mediator类，它引用4个UI组件，并负责跟它们交互：

```
public class Mediator {
    // 引用UI组件:
    private List<JCheckBox> checkBoxList;
    private JButton selectAll;
    private JButton selectNone;
    private JButton selectInverse;

    public Mediator(List<JCheckBox> checkBoxList, JButton selectAll, JButton selectNone, JButton selectInverse) {
        this.checkBoxList = checkBoxList;
        this.selectAll = selectAll;
        this.selectNone = selectNone;
        this.selectInverse = selectInverse;
        // 绑定事件:
        this.checkBoxList.forEach(checkBox -> {
            checkBox.addChangeListener(this::onCheckBoxChanged);
        });
        this.selectAll.addActionListener(this::onSelectAllClicked);
        this.selectNone.addActionListener(this::onSelectNoneClicked);
        this.selectInverse.addActionListener(this::onSelectInverseClicked);
    }

    // 当checkbox有变化时:
    public void onCheckBoxChanged(ChangeEvent event) {
        boolean allChecked = true;
        boolean allUnchecked = true;
        for (var checkBox : checkBoxList) {
            if (checkBox.isSelected()) {
                allUnchecked = false;
            } else {
                allChecked = false;
            }
        }
        selectAll.setEnabled(!allChecked);
        selectNone.setEnabled(!allUnchecked);
    }
```

```
    // 当点击select all:
    public void onSelectAllClicked(ActionEvent event) {
        checkBoxList.forEach(checkBox -> checkBox.setSelected(true));
        selectAll.setEnabled(false);
        selectNone.setEnabled(true);
    }

    // 当点击select none:
    public void onSelectNoneClicked(ActionEvent event) {
        checkBoxList.forEach(checkBox -> checkBox.setSelected(false));
        selectAll.setEnabled(true);
        selectNone.setEnabled(false);
    }

    // 当点击select inverse:
    public void onSelectInverseClicked(ActionEvent event) {
        checkBoxList.forEach(checkBox -> checkBox.setSelected(!checkBox.isSelected()));
        onCheckBoxChanged(null);
    }
}
```

运行一下看看效果：

使用Mediator模式后，我们得到了以下好处：

- 各个UI组件互不引用，这样就减少了组件之间的耦合关系；
- Mediator用于当一个组件发生状态变化时，根据当前所有组件的状态决定更新某些组件；
- 如果新增一个UI组件，我们只需要修改Mediator更新状态的逻辑，现有的其他UI组件代码不变。

Mediator模式经常用在有众多交互组件的UI上。为了简化UI程序，MVC模式以及MVVM模式都可以看作是Mediator模式的扩展。

练习

使用Mediator模式

小结

中介模式是通过引入一个中介对象，把多边关系变成多个双边关系，从而简化系统组件的交互耦合度。

在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态。

备忘录模式（Memento），主要用于捕获一个对象的内部状态，以便在将来的某个时候恢复此状态。

其实我们使用的几乎所有软件都用到了备忘录模式。最简单的备忘录模式就是保存到文件，打开文件。对于文本编辑器来说，保存就是把`TextEditor`类的字符串存储到文件，打开就是恢复`TextEditor`类的状态。对于图像编辑器来说，原理是一样的，只是保存和恢复的数据格式比较复杂而已。Java的序列化也可以看作是备忘录模式。

在使用文本编辑器的时候，我们还经常使用Undo、Redo这些功能。这些其实也可以用备忘录模式实现，即不定期地把`TextEditor`类的字符串复制一份存起来，这样就可以Undo或Redo。

标准的备忘录模式有这么几种角色：

- Memonto：存储的内部状态；
- Originator：创建一个备忘录并设置其状态；
- Caretaker：负责保存备忘录。

实际上我们在使用备忘录模式的时候，不必设计得这么复杂，只需要对类似`TextEditor`的类，增加`getState()`和`setState()`就可以了。

我们以一个文本编辑器`TextEditor`为例，它内部使用`StringBuilder`允许用户增删字符：

```
public class TextEditor {
    private StringBuilder buffer = new StringBuilder();

    public void add(char ch) {
        buffer.append(ch);
    }

    public void add(String s) {
        buffer.append(s);
    }

    public void delete() {
        if (buffer.length() > 0) {
            buffer.deleteCharAt(buffer.length() - 1);
        }
    }
}
```

为了支持这个`TextEditor`能保存和恢复状态，我们增加`getState()`和`setState()`两个方法：

```
public class TextEditor {
    ...

    // 获取状态:
    public String getState() {
        return buffer.toString();
    }

    // 恢复状态:
    public void setState(String state) {
        this.buffer.delete(0, this.buffer.length());
        this.buffer.append(state);
    }
}
```

对这个简单的文本编辑器，用一个`String`就可以表示其状态，对于复杂的对象模型，通常我们会使用JSON、XML等复杂格式。

练习

给TextEditor添加备忘录模式

小结

备忘录模式是为了保存对象的内部状态，并在将来恢复，大多数软件提供的保存、打开，以及编辑过程中的Undo、Redo都是备忘录模式的应用。

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。

观察者模式（Observer）又称发布-订阅模式（Publish-Subscribe：Pub/Sub）。它是一种通知机制，让发送通知的一方（被观察方）和接收通知的一方（观察者）能彼此分离，互不影响。

要理解观察者模式，我们还是看例子。

假设一个电商网站，有多种`Product`（商品），同时，`Customer`（消费者）和`Admin`（管理员）对商品上架、价格改变都感兴趣，希望能第一时间获得通知。于是，`Store`（商场）可以这么写：

```
public class Store {
    Customer customer;
    Admin admin;

    private Map<String, Product> products = new HashMap<>();

    public void addNewProduct(String name, double price) {
        Product p = new Product(name, price);
        products.put(p.getName(), p);
        // 通知用户:
        customer.onPublished(p);
        // 通知管理员:
        admin.onPublished(p);
    }

    public void setProductPrice(String name, double price) {
        Product p = products.get(name);
        p.setPrice(price);
        // 通知用户:
        customer.onPriceChanged(p);
        // 通知管理员:
        admin.onPriceChanged(p);
    }
}
```

我们观察上述Store类的问题：它直接引用了Customer和Admin。先不考虑多个Customer或多个Admin的问题，上述Store类最大的问题是，如果要加一个新的观察者类型，例如工商局管理员，Store类就必须继续改动。

因此，上述问题的本质是Store希望发送通知给那些关心Product的对象，但Store并不想知道这些人是谁。观察者模式就是要分离被观察者和观察者之间的耦合关系。

要实现这一目标也很简单，Store不能直接引用Customer和Admin，相反，它引用一个ProductObserver接口，任何人想要观察Store，只要实现该接口，并且把自己注册到Store即可：

```
public class Store {
    private List<ProductObserver> observers = new ArrayList<>();
    private Map<String, Product> products = new HashMap<>();

    // 注册观察者:
    public void addObserver(ProductObserver observer) {
        this.observers.add(observer);
    }

    // 取消注册:
    public void removeObserver(ProductObserver observer) {
        this.observers.remove(observer);
    }

    public void addNewProduct(String name, double price) {
        Product p = new Product(name, price);
        products.put(p.getName(), p);
        // 通知观察者:
        observers.forEach(o -> o.onPublished(p));
    }

    public void setProductPrice(String name, double price) {
        Product p = products.get(name);
        p.setPrice(price);
        // 通知观察者:
        observers.forEach(o -> o.onPriceChanged(p));
    }
}
```

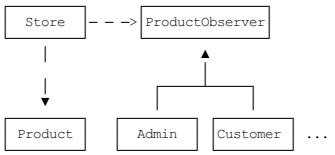
就是这么一个小小的改动，使得观察者类型就可以无限扩充，而且，观察者的定义可以放到客户端：

```
// observer:
Admin a = new Admin();
Customer c = new Customer();
// store:
Store store = new Store();
// 注册观察者:
store.addObserver(a);
store.addObserver(c);
```

甚至可以注册匿名观察者：

```
store.addObserver(new ProductObserver() {
    public void onPublished(Product product) {
        System.out.println("[Log] on product published: " + product);
    }

    public void onPriceChanged(Product product) {
        System.out.println("[Log] on product price changed: " + product);
    }
});
```

用一张图画出观察者模式：



观察者模式也有很多变体形式。有的观察者模式把被观察者也抽象出接口：

```
public interface ProductObservable { // 注意此处拼写是Observable不是Observer!
    void addObserver(ProductObserver observer);
    void removeObserver(ProductObserver observer);
}
```

对应的实体被观察者就要实现该接口：

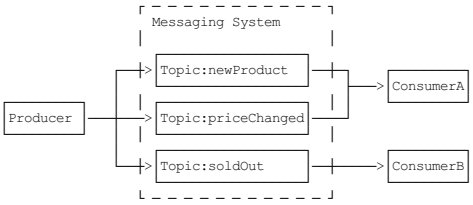
```
public class Store implements ProductObservable {
    ...
}
```

有些观察者模式把通知变成一个Event对象，从而不再有多种方法通知，而是统一成一种：

```
public interface ProductObserver {
    void onEvent(ProductEvent event);
}
```

让观察者自己从Event对象中读取通知类型和通知数据。

广义的观察者模式包括所有消息系统。所谓消息系统，就是把观察者和被观察者完全分离，通过消息系统本身来通知：



消息发送方称为Producer，消息接收方称为Consumer，Producer发送消息的时候，必须选择发送到哪个Topic。Consumer可以订阅自己感兴趣的Topic，从而只获得特定类型的消息。

使用消息系统实现观察者模式时，Producer和Consumer甚至经常不在同一台机器上，并且双方对对方完全一无所知，因为注册观察者这个动作本身都在消息系统中完成，而不是在Producer内部完成。

此外，注意到我们在编写观察者模式的时候，通知Observer是依靠语句：

```
observers.forEach(o -> o.onPublished(p));
```

这说明各个观察者是依次获得的同步通知，如果上一个观察者处理太慢，会导致下一个观察者不能及时获得通知。此外，如果观察者在处理通知的时候，发生了异常，还需要被观察者处理异常，才能保证继续通知下一个观察者。

思考：如何改成异步通知，使得所有观察者可以并发同时处理？

有的童鞋可能发现Java标准库有个`java.util.Observable`类和一个`Observer`接口，用来帮助我们实现观察者模式。但是，这个类非常不！好！用！实现观察者模式的时候，也不推荐借助这两个东东。

练习

给`Store`增加一种类型的观察者，并把通知改为异步。

观察者模式练习

小结

观察者模式，又称发布-订阅模式，是一种一对多的通知机制，使得双方无需关心对方，只关心通知本身。

> 允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它的类。

状态模式（State）经常用在带有状态的对象中。

什么是状态？我们以QQ聊天为例，一个用户的QQ有几种状态：

- 离线状态（尚未登录）；
- 正在登录状态；
- 在线状态；
- 忙状态（暂时离开）。

如何表示状态？我们定义一个`enum`就可以表示不同的状态。但不同的状态需要对应不同的行为，比如收到消息时：

```
if (state == ONLINE) {
    // 闪烁图标
} else if (state == BUSY) {
    reply("现在忙，稍后回复");
} else if ...
```

状态模式的目的是为了把上述一大串`if...else...`的逻辑给分拆到不同的状态类中，使得将来增加状态比较容易。

例如，我们设计一个聊天机器人，它有两个状态：

- 未连线；
- 已连线。

对于未连线状态，我们收到消息也不回复：

```
public class DisconnectedState implements State {
    public String init() {
        return "Bye!";
    }

    public String reply(String input) {
        return "";
    }
}
```

对于已连线状态，我们回应收到的消息：

```
public class ConnectedState implements State {
    public String init() {
        return "Hello, I'm Bob.";
    }

    public String reply(String input) {
        if (input.endsWith("?")) {
            return "Yes. " + input.substring(0, input.length() - 1) + "!";
        }
        if (input.endsWith(".")) {
            return input.substring(0, input.length() - 1) + "!";
        }
        return input.substring(0, input.length() - 1) + "?";
    }
}
```

状态模式的关键设计思想在于状态切换，我们引入一个`BotContext`完成状态切换：

```
public class BotContext {
 private State state = new DisconnectedState();

 public String chat(String input) {
  if ("hello".equalsIgnoreCase(input)) {
            // 收到hello切换到在线状态:
   state = new ConnectedState();
   return state.init();
  } else if ("bye".equalsIgnoreCase(input)) {
            /  收到bye切换到离线状态:
   state = new DisconnectedState();
   return state.init();
  }
  return state.reply(input);
 }
}
```

这样，一个价值千万的AI聊天机器人就诞生了：

```
Scanner scanner = new Scanner(System.in);
BotContext bot = new BotContext();
for (;;) {
    System.out.print("> ");
    String input = scanner.nextLine();
    String output = bot.chat(input);
    System.out.println(output.isEmpty() ? "(no reply)" : "< " + output);
}
```

试试效果：

```
> hello
< Hello, I'm Bob.
> Nice to meet you.
< Nice to meet you!
> Today is cold?
< Yes. Today is cold!
> bye
< Bye!
```

练习

新增BusyState状态表示忙碌

小结

状态模式的设计思想是把不同状态的逻辑分离到不同的状态类中，从而使得增加新状态更容易；

状态模式的实现关键在于状态转换。简单的状态转换可以直接由调用方指定，复杂的状态转换可以在内部根据条件触发完成。

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。

策略模式：Strategy，是指，定义一组算法，并把其封装到一个对象中。然后在运行时，可以灵活的使用其中的一个算法。

策略模式在Java标准库中应用非常广泛，我们以排序为例，看看如何通过`Arrays.sort()`实现忽略大小写排序：

```
import java.util.Arrays;
----
public class Main {
    public static void main(String[] args) throws InterruptedException {
        String[] array = { "apple", "Pear", "Banana", "orange" };
        Arrays.sort(array, String::compareToIgnoreCase);
        System.out.println(Arrays.toString(array));
    }
}
```

如果我们想忽略大小写排序，就传入`String::compareToIgnoreCase`，如果我们想倒序排序，就传入`(s1, s2) -> -s1.compareTo(s2)`，这个比较两个元素大小的算法就是策略。

我们观察`Arrays.sort(T[] a, Comparator<? super T> c)`这个排序方法，它在内部实现了TimSort排序，但是，排序算法在比较两个元素大小的时候，需要借助我们传入的`Comparator`对象，才能完成比较。因此，这里的策略是指比较两个元素大小的策略，可以是忽略大小写比较，可以是倒序比较，也可以根据字符串长度比较。

因此，上述排序使用到了策略模式，它实际上指，在一个方法中，流程是确定的，但是，某些关键步骤的算法依赖调用方传入的策略，这样，传入不同的策略，即可获得不同的结果，大大增强了系统的灵活性。

如果我们自己实现策略模式的排序，用冒泡法编写如下：

```
import java.util.*;
----
public class Main {
    public static void main(String[] args) throws InterruptedException {
        String[] array = { "apple", "Pear", "Banana", "orange" };
        sort(array, String::compareToIgnoreCase);
        System.out.println(Arrays.toString(array));
    }

    static <T> void sort(T[] a, Comparator<? super T> c) {
        for (int i = 0; i < a.length - 1; i++) {
            for (int j = 0; j < a.length - 1 - i; j++) {
                if (c.compare(a[j], a[j + 1]) > 0) { // 注意这里比较两个元素的大小依赖传入的策略
                    T temp = a[j];
                    a[j] = a[j + 1];
                    a[j + 1] = temp;
                }
            }
        }
    }
}
```

一个完整的策略模式要定义策略以及使用策略的上下文。我们以购物车结算为例，假设网站针对普通会员、Prime会员有不同的折扣，同时活动期间还有一个满100减20的活动，这些就可以作为策略实现。先定义打折策略接口：

```
public interface DiscountStrategy {
    // 计算折扣额度:
    BigDecimal getDiscount(BigDecimal total);
}
```

接下来，就是实现各种策略。普通用户策略如下：

```
public class UserDiscountStrategy implements DiscountStrategy {
    public BigDecimal getDiscount(BigDecimal total) {
        // 普通会员打九折:
        return total.multiply(new BigDecimal("0.1")).setScale(2, RoundingMode.DOWN);
    }
}
```

满减策略如下：

```
public class OverDiscountStrategy implements DiscountStrategy {
    public BigDecimal getDiscount(BigDecimal total) {
        // 满100减20优惠:
        return total.compareTo(BigDecimal.valueOf(100)) >= 0 ? BigDecimal.valueOf(20) : BigDecimal.ZERO;
    }
}
```

最后，要应用策略，我们需要一个`DiscountContext`：

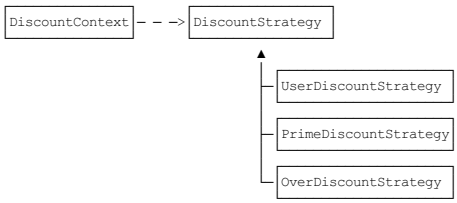
```
public class DiscountContext {
    // 持有某个策略:
    private DiscountStrategy strategy = new UserDiscountStrategy();

    // 允许客户端设置新策略:
    public void setStrategy(DiscountStrategy strategy) {
        this.strategy = strategy;
    }
```

```
    public BigDecimal calculatePrice(BigDecimal total) {
        return total.subtract(this.strategy.getDiscount(total)).setScale(2);
    }
}
```

调用方必须首先创建一个DiscountContext，并指定一个策略（或者使用默认策略），即可获得折扣后的价格：

```
DiscountContext ctx = new DiscountContext();

// 默认使用普通会员折扣:
BigDecimal pay1 = ctx.calculatePrice(BigDecimal.valueOf(105));
System.out.println(pay1);

// 使用满减折扣:
ctx.setStrategy(new OverDiscountStrategy());
BigDecimal pay2 = ctx.calculatePrice(BigDecimal.valueOf(105));
System.out.println(pay2);

// 使用Prime会员折扣:
ctx.setStrategy(new PrimeDiscountStrategy());
BigDecimal pay3 = ctx.calculatePrice(BigDecimal.valueOf(105));
System.out.println(pay3);
```

上述完整的策略模式如下图所示：



策略模式的核心思想是在一个计算方法中把容易变化的算法抽出来作为“策略”参数传进去，从而使得新增策略不必修改原有逻辑。

练习

使用策略模式新增一种策略，允许在满100减20的基础上对Prime会员再打七折。

策略模式练习

小结

策略模式是为了允许调用方选择一个算法，从而通过不同策略实现不同的计算结果。

通过扩展策略，不必修改主逻辑，即可获得新策略的结果。

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中，使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

模板方法（Template Method）是一个比较简单的模式。它的主要思想是，定义一个操作的一系列步骤，对于某些暂时确定不下来的步骤，就留给子类去实现好了，这样不同的子类就可以定义出不同的步骤。

因此，模板方法的核心在于定义一个“骨架”。我们还是举例说明。

假设我们开发了一个从数据库读取设置的类：

```
public class Setting {
    public final String getSetting(String key) {
        String value = readFromDatabase(key);
        return value;
    }

 private String readFromDatabase(String key) {
        // TODO: 从数据库读取
    }
}
```

由于从数据库读取数据较慢，我们可以考虑把读取的设置缓存起来，这样下一次读取同样的key就不必再访问数据库了。但是怎么实现缓存，暂时没想好，但不妨碍我们先写出使用缓存的代码：

```
public class Setting {
    public final String getSetting(String key) {
        // 先从缓存读取:
        String value = lookupCache(key);
        if (value == null) {
            // 在缓存中未找到,从数据库读取:
            value = readFromDatabase(key);
            System.out.println("[DEBUG] load from db: " + key + " = " + value);
            // 放入缓存:
            putIntoCache(key, value);
        } else {
            System.out.println("[DEBUG] load from cache: " + key + " = " + value);
        }
        return value;
    }
}
```

整个流程没有问题，但是，lookupCache(key)和putIntoCache(key, value)这两个方法还根本没实现，怎么编译通过？这个不要紧，我们声明抽象方法就可以：

```
public abstract class AbstractSetting {
    public final String getSetting(String key) {
        String value = lookupCache(key);
        if (value == null) {
            value = readFromDatabase(key);
            putIntoCache(key, value);
        }
        return value;
    }

    protected abstract String lookupCache(String key);

    protected abstract void putIntoCache(String key, String value);
}
```

因为声明了抽象方法，自然整个类也必须是抽象类。如何实现lookupCache(key)和putIntoCache(key, value)这两个方法就交给子类了。子类其实并不关心核心代码getSetting(key)的逻辑，它只需要关心如何完成两个小小的子任务就可以了。

假设我们希望用一个Map做缓存，那么可以写一个LocalSetting：

```
public class LocalSetting extends AbstractSetting {
    private Map<String, String> cache = new HashMap<>();
```

```
    protected String lookupCache(String key) {
        return cache.get(key);
    }

    protected void putIntoCache(String key, String value) {
        cache.put(key, value);
    }
}
```

如果我们要使用Redis做缓存，那么可以再写一个RedisSetting：

```
public class RedisSetting extends AbstractSetting {
    private RedisClient client = RedisClient.create("redis://localhost:6379");

    protected String lookupCache(String key) {
        try (StatefulRedisConnection<String, String> connection = client.connect()) {
            RedisCommands<String, String> commands = connection.sync();
            return commands.get(key);
        }
    }

    protected void putIntoCache(String key, String value) {
        try (StatefulRedisConnection<String, String> connection = client.connect()) {
            RedisCommands<String, String> commands = connection.sync();
            commands.set(key, value);
        }
    }
}
```

客户端代码使用本地缓存的代码这么写：

```
AbstractSetting setting1 = new LocalSetting();
System.out.println("test = " + setting1.getSetting("test"));
System.out.println("test = " + setting1.getSetting("test"));
```

要改成Redis缓存，只需要把LocalSetting替换为RedisSetting：

```
AbstractSetting setting2 = new RedisSetting();
System.out.println("autosave = " + setting2.getSetting("autosave"));
System.out.println("autosave = " + setting2.getSetting("autosave"));
```

可见，模板方法的核心思想是：父类定义骨架，子类实现某些细节。

为了防止子类重写父类的骨架方法，可以在父类中对骨架方法使用final。对于需要子类实现的抽象方法，一般声明为protected，使得这些方法对外部客户端不可见。

Java标准库也有很多模板方法的应用。在集合类中，AbstractList和AbstractQueuedSynchronizer都定义了很多通用操作，子类只需要实现某些必要方法。

练习

使用模板方法增加一个使用Guava Cache的子类。

模板方法练习

思考：能否将readFromDatabase()作为模板方法，使得子类可以选择从数据库读取还是从文件读取。

再思考如果既可以扩展缓存，又可以扩展底层存储，会不会出现子类数量爆炸的情况？如何解决？

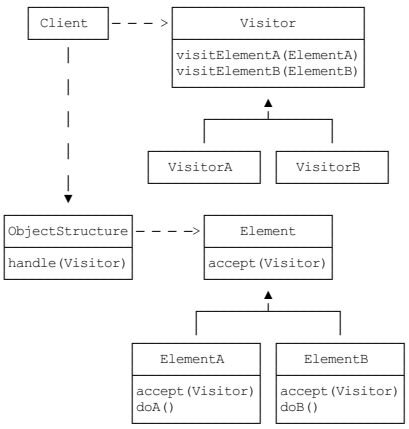
小结

模板方法是一种高层定义骨架，底层实现细节的设计模式，适用于流程固定，但某些步骤不确定或可替换的情况。

表示一个作用于某对象结构中的各元素的操作。它使你可以在不改变各元素的类的前提下定义作用于这些元素的新操作。

访问者模式（Visitor）是一种操作一组对象的操作，它的目的是不改变对象的定义，但允许新增不同的访问者，来定义新的操作。

访问者模式的设计比较复杂，如果我们查看GoF原始的访问者模式，它是这么设计的：



上述模式的复杂之处在于上述访问者模式为了实现所谓的“双重分派”，设计了一个回调再回调的机制。因为Java只支持基于多态的单分派模式，这里强行模拟出“双重分派”反而加大了代码的复杂性。

这里我们只介绍简化的访问者模式。假设我们要递归遍历某个文件夹的所有子文件夹和文件，然后找出.java文件，正常的做法是写个递归：

```
void scan(File dir, List<File> collector) {
    for (File file : dir.listFiles()) {
        if (file.isFile() && file.getName().endsWith(".java")) {
            collector.add(file);
        } else if (file.isDir()) {
            // 递归调用:
            scan(file, collector);
        }
    }
}
```

上述代码的问题在于，扫描目录的逻辑和处理.java文件的逻辑混在了一起。如果下次需要增加一个清理.class文件的功能，就必须再重复写扫描逻辑。

因此，访问者模式先把数据结构（这里是文件夹和文件构成的树型结构）和对其的操作（查找文件）分离开，以后如果要新增操作（例如清理.class文件），只需要新增访问者，不需要改变现有逻辑。

用访问者模式改写上述代码步骤如下：

首先，我们需要定义访问者接口，即该访问者能够干的事情：

```
public interface Visitor {
    // 访问文件夹:
    void visitDir(File dir);
    // 访问文件:
    void visitFile(File file);
}
```

紧接着，我们要定义能持有文件夹和文件的数据结构`FileStructure`：

```
public class FileStructure {
    // 根目录:
    private File path;
    public FileStructure(File path) {
        this.path = path;
    }
}
```

然后，我们给`FileStructure`增加一个`handle()`方法，传入一个访问者：

```
public class FileStructure {
    ...

    public void handle(Visitor visitor) {
  scan(this.path, visitor);
 }

 private void scan(File file, Visitor visitor) {
  if (file.isDirectory()) {
            // 让访问者处理文件夹:
   visitor.visitDir(file);
   for (File sub : file.listFiles()) {
               // 递归处理子文件夹:
    scan(sub, visitor);
   }
  } else if (file.isFile()) {
            // 让访问者处理文件:
   visitor.visitFile(file);
  }
 }
}
```

这样，我们就把访问者的行为抽象出来了。如果我们要实现一种操作，例如，查找`.java`文件，就传入`JavaFileVisitor`：

```
FileStructure fs = new FileStructure(new File("."));
fs.handle(new JavaFileVisitor());
```

这个`JavaFileVisitor`实现如下：

```
public class JavaFileVisitor implements Visitor {
    public void visitDir(File dir) {
        System.out.println("Visit dir: " + dir);
    }

    public void visitFile(File file) {
        if (file.getName().endsWith(".java")) {
            System.out.println("Found java file: " + file);
        }
    }
}
```

类似的，如果要清理`.class`文件，可以再写一个`ClassFileClearnerVisitor`：

```
public class ClassFileCleanerVisitor implements Visitor {
 public void visitDir(File dir) {
 }

 public void visitFile(File file) {
  if (file.getName().endsWith(".class")) {
   System.out.println("Will clean class file: " + file);
  }
 }
}
```

可见，访问者模式的核心思想是为了访问比较复杂的数据结构，不去改变数据结构，而是把对数据的操作抽象出来，在“访问”的过程中以回调形式在访问者中处理操作逻辑。如果要新增一组操作，那么只需要增加一个新的访问者。

实际上，Java标准库提供的`Files.walkFileTree()`已经实现了一个访问者模式：

```
import java.io.*;
import java.nio.file.*;
import java.nio.file.attribute.*;
----
public class Main {
    public static void main(String[] args) throws IOException {
        Files.walkFileTree(Paths.get("."), new MyFileVisitor());
    }
}

// 实现一个FileVisitor:
class MyFileVisitor extends SimpleFileVisitor<Path> {
    // 处理Directory:
    public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs) throws IOException {
        System.out.println("pre visit dir: " + dir);
        // 返回CONTINUE表示继续访问:
        return FileVisitResult.CONTINUE;
    }

    // 处理File:
    public FileVisitResult visitFile(Path file, BasicFileAttributes attrs) throws IOException {
        System.out.println("visit file: " + file);
        // 返回CONTINUE表示继续访问:
        return FileVisitResult.CONTINUE;
    }
}
```

`Files.walkFileTree()`允许访问者返回`FileVisitResult.CONTINUE`以便继续访问，或者返回`FileVisitResult.TERMINATE`停止访问。

类似的，对XML的SAX处理也是一个访问者模式，我们需要提供一个SAX Handler作为访问者处理XML的各个节点。

练习

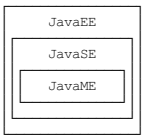
使用访问者模式递归遍历文件夹

小结

访问者模式是为了抽象出作用于一组复杂对象的操作，并且后续可以新增操作而不必对现有的对象结构做任何改动。

从本章开始，我们就正式进入到JavaEE的领域。

什么是JavaEE？JavaEE是Java Platform Enterprise Edition的缩写，即Java企业平台。我们前面介绍的所有基于标准JDK的开发都是JavaSE，即Java Platform Standard Edition。此外，还有一个小众不太常用的JavaME：Java Platform Micro Edition，是Java移动开发平台（非Android），它们三者关系如下：

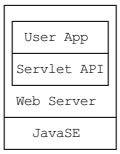


JavaME是一个裁剪后的“微型版”JDK，现在使用很少，我们不用管它。JavaEE也不是凭空冒出来的，它实际上是完全基于JavaSE，只是多了一大堆服务器相关的库以及API接口。所有的JavaEE程序，仍然是运行在标准的JavaSE的虚拟机上的。

最早的JavaEE的名称是J2EE：Java 2 Platform Enterprise Edition，后来改名为JavaEE。由于Oracle将JavaEE移交给Eclipse开源组织时，不允许他们继续使用Java商标，所以JavaEE再次改名为Jakarta EE。因为这个拼写比较复杂而且难记，所以我们后面还是用JavaEE这个缩写。

JavaEE并不是一个软件产品，它更多的是一种软件架构和设计思想。我们可以把JavaEE看作是在JavaSE的基础上，开发的一系列基于服务器的组件、API标准和通用架构。

JavaEE最核心的组件就是基于Servlet标准的Web服务器，开发者编写的应用程序是基于Servlet API并运行在Web服务器内部的：



此外，JavaEE还有一系列技术标准：

- EJB：Enterprise JavaBean，企业级JavaBean，早期经常用于实现应用程序的业务逻辑，现在基本被轻量级框架如Spring所取代；
- JAAS：Java Authentication and Authorization Service，一个标准的认证和授权服务，常用于企业内部，Web程序通常使用更轻量级的自定义认证；
- JCA：JavaEE Connector Architecture，用于连接企业内部的EIS系统等；
- JMS：Java Message Service，用于消息服务；
- JTA：Java Transaction API，用于分布式事务；
- JAX-WS：Java API for XML Web Services，用于构建基于XML的Web服务；
- ...

目前流行的基于Spring的轻量级JavaEE开发架构，使用最广泛的是Servlet和JMS，以及一系列开源组件。本章我们将详细介绍基于Servlet的Web开发。

今天我们访问网站，使用App时，都是基于Web这种Browser/Server模式，简称BS架构，它的特点是，客户端只需要浏览器，应用程序的逻辑和数据都存储在服务器端。浏览器只需要请求服务器，获取Web页面，并把Web页面展示给用户即可。

Web页面具有极强的交互性。由于Web页面是用HTML编写的，而HTML具备超强的表现力，并且，服务器端升级后，客户端无需任何部署就可以使用到新的版本，因此，BS架构升级非常容易。

HTTP协议

在Web应用中，浏览器请求一个URL，服务器就把生成的HTML网页发送给浏览器，而浏览器和服务器之间的传输协议是HTTP，所以：

- HTML是一种用来定义网页的文本，会HTML，就可以编写网页；
- HTTP是在网络上传输HTML的协议，用于浏览器和服务器的通信。

HTTP协议是一个基于TCP协议之上的请求-响应协议，它非常简单，我们先使用Chrome浏览器查看新浪首页，然后选择View - Developer - Inspect Elements就可以看到HTML：

切换到Network，重新加载页面，可以看到浏览器发出的每一个请求和响应：

使用Chrome浏览器可以方便地调试Web应用程序。

对于Browser来说，请求页面的流程如下：

1. 与服务器建立TCP连接；
2. 发送HTTP请求；
3. 收取HTTP响应，然后把网页在浏览器中显示出来。

浏览器发送的HTTP请求如下：

```
GET / HTTP/1.1
Host: www.sina.com.cn
User-Agent: Mozilla/5.0 xxx
Accept: */*
Accept-Language: zh-CN,zh;q=0.9,en-US;q=0.8
```

其中，第一行表示使用GET请求获取路径为/的资源，并使用HTTP/1.1协议，从第二行开始，每行都是以Header: Value形式表示的HTTP头，比较常用的HTTP Header包括：

- Host: 表示请求的主机名，因为一个服务器上可能运行着多个网站，因此，Host表示浏览器正在请求的域名；
- User-Agent: 标识客户端本身，例如Chrome浏览器的标识类似Mozilla/5.0 ... Chrome/79，IE浏览器的标识类似Mozilla/5.0 (Windows NT ...) like Gecko；
- Accept：表示浏览器能接收的资源类型，如text/*，image/*或者*/*表示所有；
- Accept-Language：表示浏览器偏好的语言，服务器可以据此返回不同语言的网页；
- Accept-Encoding：表示浏览器可以支持的压缩类型，例如gzip, deflate, br。

服务器的响应如下：

```
HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 21932
Content-Encoding: gzip
Cache-Control: max-age=300

<html>...网页数据...
```

服务器响应的第一行总是版本号+空格+数字+空格+文本，数字表示响应代码，其中2xx表示成功，3xx表示重定向，4xx表示客户端引发的错误，5xx表示服务器端引发的错误。数字是给程序识别，文本则是给开发者调试使用的。常见的响应代码有：

- 200 OK：表示成功；
- 301 Moved Permanently：表示该URL已经永久重定向；

- 302 Found：表示该URL需要临时重定向；
- 304 Not Modified：表示该资源没有修改，客户端可以使用本地缓存的版本；
- 400 Bad Request：表示客户端发送了一个错误的请求，例如参数无效；
- 401 Unauthorized：表示客户端因为身份未验证而不允许访问该URL；
- 403 Forbidden：表示服务器因为权限问题拒绝了客户端的请求；
- 404 Not Found：表示客户端请求了一个不存在的资源；
- 500 Internal Server Error：表示服务器处理时内部出错，例如因为无法连接数据库；
- 503 Service Unavailable：表示服务器此刻暂时无法处理请求。

从第二行开始，服务器每一行均返回一个HTTP头。服务器经常返回的HTTP Header包括：

- Content-Type：表示该响应内容的类型，例如`text/html`，`image/jpeg`；
- Content-Length：表示该响应内容的长度（字节数）；
- Content-Encoding：表示该响应压缩算法，例如`gzip`；
- Cache-Control：指示客户端应如何缓存，例如`max-age=300`表示可以最多缓存300秒。

HTTP请求和响应都由HTTP Header和HTTP Body构成，其中HTTP Header每行都以`\r\n`结束。如果遇到两个连续的`\r\n`，那么后面就是HTTP Body。浏览器读取HTTP Body，并根据Header信息中指示的`Content-Type`、`Content-Encoding`等解压后显示网页、图像或其他内容。

通常浏览器获取的第一个资源是HTML网页，在网页中，如果嵌入了JavaScript、CSS、图片、视频等其他资源，浏览器会根据资源的URL再次向服务器请求对应的资源。

关于HTTP协议的详细内容，请参考HTTP权威指南一书，或者Mozilla开发者网站。

我们在前面介绍的HTTP编程是以客户端的身份去请求服务器资源。现在，我们需要以服务器的身份响应客户端请求，编写服务器程序来处理客户端请求通常就称之为Web开发。

编写HTTP Server

我们来看一下如何编写HTTP Server。一个HTTP Server本质上是一个TCP服务器，我们先用TCP编程的多线程实现的服务器端框架：

```
public class Server {
    public static void main(String[] args) throws IOException {
        ServerSocket ss = new ServerSocket(8080); // 监听指定端口
        System.out.println("server is running...");
        for (;;) {
            Socket sock = ss.accept();
            System.out.println("connected from " + sock.getRemoteSocketAddress());
            Thread t = new Handler(sock);
            t.start();
        }
    }
}

class Handler extends Thread {
    Socket sock;

    public Handler(Socket sock) {
        this.sock = sock;
    }

    public void run() {
        try (InputStream input = this.sock.getInputStream()) {
            try (OutputStream output = this.sock.getOutputStream()) {
                handle(input, output);
            }
        } catch (Exception e) {
            try {
                this.sock.close();
            } catch (IOException ioe) {
            }
            System.out.println("client disconnected.");
        }
    }

    private void handle(InputStream input, OutputStream output) throws IOException {
        var reader = new BufferedReader(new InputStreamReader(input, StandardCharsets.UTF_8));
        var writer = new BufferedWriter(new OutputStreamWriter(output, StandardCharsets.UTF_8));
        // TODO: 处理HTTP请求
    }
}
```

只需要在`handle()`方法中，用Reader读取HTTP请求，用Writer发送HTTP响应，即可实现一个最简单的HTTP服务器。编写代码如下：

```
private void handle(InputStream input, OutputStream output) throws IOException {
    System.out.println("Process new http request...");
    var reader = new BufferedReader(new InputStreamReader(input, StandardCharsets.UTF_8));
    var writer = new BufferedWriter(new OutputStreamWriter(output, StandardCharsets.UTF_8));
    // 读取HTTP请求:
    boolean requestOk = false;
    String first = reader.readLine();
    if (first.startsWith("GET / HTTP/1.")) {
        requestOk = true;
    }
    for (;;) {
        String header = reader.readLine();
        if (header.isEmpty()) { // 读取到空行时, HTTP Header读取完毕
            break;
        }
        System.out.println(header);
    }
    System.out.println(requestOk ? "Response OK" : "Response Error");
    if (!requestOk) {
        // 发送错误响应:
        writer.write("HTTP/1.0 404 Not Found\r\n");
        writer.write("Content-Length: 0\r\n");
        writer.write("\r\n");
        writer.flush();
    } else {
        // 发送成功响应:
        String data = "<html><body><h1>Hello, world!</h1></body></html>";
        int length = data.getBytes(StandardCharsets.UTF_8).length;
        writer.write("HTTP/1.0 200 OK\r\n");
        writer.write("Connection: close\r\n");
        writer.write("Content-Type: text/html\r\n");
        writer.write("Content-Length: " + length + "\r\n");
        writer.write("\r\n"); // 空行标识Header和Body的分隔
        writer.write(data);
        writer.flush();
    }
}
```

这里的核心代码是，先读取HTTP请求，这里我们只处理`GET /`的请求。当读取到空行时，表示已读到连续两个`\r\n`，说明请求结束，可以发送响应。发送响应的时候，首先发送响应代码`HTTP/1.0 200 OK`表示一个成功的200响应，使用`HTTP/1.0`协议，然后，依次发送Header，发送完Header后，再发送一个空行标识Header结束，紧接着发送HTTP Body，在浏览器输入`http://local.liaoxuefeng.com:8080/`就可以看到响应页面：

HTTP目前有多个版本，1.0是早期版本，浏览器每次建立TCP连接后，只发送一个HTTP请求并接收一个HTTP响应，然后就关闭TCP连接。由于创建TCP连接本身就需要消耗一定的时间，因此，HTTP 1.1允许浏览器和服务器在同一个TCP连接上反复发送、接收多个HTTP请求和响应，这样就大大提高了传输效率。

我们注意到HTTP协议是一个请求-响应协议，它总是发送一个请求，然后接收一个响应。能不能一次性发送多个请求，然后再接收多个响应呢？HTTP 2.0可以支持浏览器同时发出多个请求，但每个请求需要唯一标识，服务器可以不按请求的顺序返回多个响应，由浏览器自己把收到的响应和请求对应起来。可见，HTTP 2.0进一步提高了传输效率，因为浏览器发出一个请求后，不必等待响应，就可以继续发下一个请求。

HTTP 3.0为了进一步提高速度，将抛弃TCP协议，改为使用无需创建连接的UDP协议，目前HTTP 3.0仍然处于实验阶段。

练习

编写一个简单的HTTP服务器

小结

使用B/S架构时，总是通过HTTP协议实现通信；

Web开发通常是指开发服务器端的Web应用程序。

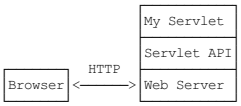
在上一节中，我们看到，编写HTTP服务器其实是非常简单的，只需要先编写基于多线程的TCP服务，然后在一个TCP连接中读取HTTP请求，发送HTTP响应即可。

但是，要编写一个完善的HTTP服务器，以HTTP/1.1为例，需要考虑的包括：

- 识别正确和错误的HTTP请求；
- 识别正确和错误的HTTP头；
- 复用TCP连接；
- 复用线程；
- IO异常处理；
- ...

这些基础工作需要耗费大量的时间，并且经过长期测试才能稳定运行。如果我们只需要输出一个简单的HTML页面，就不得不编写上千行底层代码，那就根本无法做到高效而可靠地开发。

因此，在JavaEE平台上，处理TCP连接，解析HTTP协议这些底层工作统统扔给现成的Web服务器去做，我们只需要把自己的应用程序跑在Web服务器上。为了实现这一目的，JavaEE提供了Servlet API，我们使用Servlet API编写自己的Servlet来处理HTTP请求，Web服务器实现Servlet API接口，实现底层功能：



我们来实现一个最简单的Servlet：

```
// WebServlet注解表示这是一个Servlet，并映射到地址/:
@WebServlet(urlPatterns = "/")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // 设置响应类型:
        resp.setContentType("text/html");
        // 获取输出流:
        PrintWriter pw = resp.getWriter();
        // 写入响应:
        pw.write("<h1>Hello, world!</h1>");
        // 最后不要忘记flush强制输出:
        pw.flush();
    }
}
```

一个Servlet总是继承自HttpServlet，然后覆写doGet()或doPost()方法。注意到doGet()方法传入了HttpServletRequest和HttpServletResponse两个对象，分别代表HTTP请求和响应。我们使用Servlet API时，并不直接与底层TCP交互，也不需要解析HTTP协议，因为HttpServletRequest和HttpServletResponse就已经封装好了请求和响应。以发送响应为例，我们只需要设置正确的响应类型，然后获取PrintWriter，写入响应即可。

现在问题来了：Servlet API是谁提供？

Servlet API是一个jar包，我们需要通过Maven来引入它，才能正常编译。编写pom.xml文件如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>web-servlet-hello</artifactId>
    <packaging>war</packaging>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
            <version>4.0.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
        <finalName>hello</finalName>
    </build>
</project>
```

注意到这个pom.xml与前面我们讲到的普通Java程序有个区别，打包类型不是jar，而是war，表示Java Web Application Archive：

```
<packaging>war</packaging>
```

引入的Servlet API如下：

```
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>4.0.0</version>
    <scope>provided</scope>
</dependency>
```

注意到<scope>指定为provided，表示编译时使用，但不会打包到.war文件中，因为运行期Web服务器本身已经提供了Servlet API相关的jar包。

我们还需要在工程目录下创建一个web.xml描述文件，放到src/main/webapp/WEB-INF目录下（固定目录结构，不要修改路径，注意大小写）。文件内容可以固定如下：

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
  <display-name>Archetype Created Web Application</display-name>
</web-app>
```

整个工程结构如下：

```
web-servlet-hello
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── itranswarp
        │           └── learnjava
        │               └── servlet
        │                   └── HelloServlet.java
        ├── resources
        └── webapp
            └── WEB-INF
                └── web.xml
```

运行Maven命令mvn clean package，在target目录下得到一个hello.war文件，这个文件就是我们编译打包后的Web应用程序。

现在问题又来了：我们应该如何运行这个war文件？

普通的Java程序是通过启动JVM，然后执行main()方法开始运行。但是Web应用程序有所不同，我们无法直接运行war文件，必须先启动Web服务器，再由Web服务器加载我们编写的HelloServlet，这样就可以让HelloServlet处理浏览器发送的请求。

因此，我们首先要找一个支持Servlet API的Web服务器。常用的服务器有：

- Tomcat：由Apache开发的开源免费服务器；
- Jetty：由Eclipse开发的开源免费服务器；
- GlassFish：一个开源的全功能JavaEE服务器。

还有一些收费的商用服务器，如Oracle的WebLogic，IBM的WebSphere。

无论使用哪个服务器，只要它支持Servlet API 4.0（因为我们引入的Servlet版本是4.0），我们的war包都可以在上面运行。这里我们选择使用最广泛的开源免费的Tomcat服务器。

要运行我们的hello.war，首先要下载Tomcat服务器，解压后，把hello.war复制到Tomcat的webapps目录下，然后切换到bin目录，执行startup.sh或startup.bat启动Tomcat服务器：

```
$ ./startup.sh
Using CATALINA_BASE:   .../apache-tomcat-9.0.30
Using CATALINA_HOME:   .../apache-tomcat-9.0.30
Using CATALINA_TMPDIR: .../apache-tomcat-9.0.30/temp
Using JRE_HOME:        .../jdk-11.jdk/Contents/Home
Using CLASSPATH:       .../apache-tomcat-9.0.30/bin/bootstrap.jar:...
Tomcat started.
```

在浏览器输入http://localhost:8080/hello/即可看到HelloServlet的输出：

细心的童鞋可能会问，为啥路径是/hello/而不是/？因为一个Web服务器允许同时运行多个Web App，而我们的Web App叫hello，因此，第一级目录/hello表示Web App的名字，后面的/才是我们在HelloServlet中映射的路径。

那能不能直接使用/而不是/hello/？毕竟/比较简洁。

答案是肯定的。先关闭Tomcat（执行shutdown.sh或shutdown.bat），然后删除Tomcat的webapps目录下的所有文件夹和文件，最后把我们的hello.war复制过来，改名为ROOT.war，文件名为ROOT的应用程序将作为默认应用，启动后直接访问http://localhost:8080/即可。

实际上，类似Tomcat这样的服务器也是Java编写的，启动Tomcat服务器实际上是启动Java虚拟机，执行Tomcat的main()方法，然后由Tomcat负责加载我们的.war文件，并创建一个HelloServlet实例，最后以多线程的模式来处理HTTP请求。如果Tomcat服务器收到的请求路径是/（假定部署文件为ROOT.war），就转发到HelloServlet并传入HttpServletRequest和HttpServletResponse两个对象。

因为我们编写的Servlet并不是直接运行，而是由Web服务器加载后创建实例运行，所以，类似Tomcat这样的Web服务器也称为Servlet容器。

在Servlet容器中运行的Servlet具有如下特点：

- 无法在代码中直接通过new创建Servlet实例，必须由Servlet容器自动创建Servlet实例；
- Servlet容器只会给每个Servlet类创建唯一实例；
- Servlet容器会使用多线程执行doGet()或doPost()方法。

复习一下Java多线程的内容，我们可以得出结论：

- 在Servlet中定义的实例变量会被多个线程同时访问，要注意线程安全；
- HttpServletRequest和HttpServletResponse实例是由Servlet容器传入的局部变量，它们只能被当前线程访问，不存在多个线程访问的问题；
- 在doGet()或doPost()方法中，如果使用了ThreadLocal，但没有清理，那么它的状态很可能会影响到下次的某个请求，因为Servlet容器很可能用线程池实现线程复用。

因此，正确编写Servlet，要清晰理解Java的多线程模型，需要同步访问的必须同步。

练习

给HelloServlet增加一个URL参数，例如传入http://localhost:8080/?name=Bob，能够输出Hello, Bob!。

HelloServlet练习

提示：根据HttpServletRequest文档，调用合适的方法获取URL参数。

小结

编写Web应用程序就是编写Servlet处理HTTP请求；

Servlet API提供了HttpServletRequest和HttpServletResponse两个高级接口来封装HTTP请求和响应；

Web应用程序必须按固定结构组织并打包为.war文件；

需要启动Web服务器来加载我们的war包来运行Servlet。

在上一节中，我们看到，一个完整的Web应用程序的开发流程如下：

1. 编写Servlet；
2. 打包为war文件；
3. 复制到Tomcat的webapps目录下；
4. 启动Tomcat。

这个过程是不是很繁琐？如果我们想在IDE中断点调试，还需要打开Tomcat的远程调试端口并且连接上去。

许多初学者经常卡在如何在IDE中启动Tomcat并加载webapp，更不要说断点调试了。

我们需要一种简单可靠，能直接在IDE中启动并调试webapp的方法。

因为Tomcat实际上也是一个Java程序，我们看看Tomcat的启动流程：

1. 启动JVM并执行Tomcat的`main()`方法；
2. 加载war并初始化Servlet；
3. 正常服务。

启动Tomcat无非就是设置好classpath并执行Tomcat某个jar包的`main()`方法，我们完全可以把Tomcat的jar包全部引入进来，然后自己编写一个`main()`方法，先启动Tomcat，然后让它加载我们的webapp就行。

我们新建一个`web-servlet-embedded`工程，编写`pom.xml`如下：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>web-servlet-embedded</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>
        <tomcat.version>9.0.26</tomcat.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-core</artifactId>
            <version>${tomcat.version}</version>
            <scope>provided</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-jasper</artifactId>
            <version>${tomcat.version}</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>
</project>
```

其中，`<packaging>`类型仍然为`war`，引入依赖`tomcat-embed-core`和`tomcat-embed-jasper`，引入的Tomcat版本`<tomcat.version>`为`9.0.26`。

不必引入Servlet API，因为引入Tomcat依赖后自动引入了Servlet API。因此，我们可以正常编写Servlet如下：

```
@WebServlet(urlPatterns = "/")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("text/html");
        String name = req.getParameter("name");
        if (name == null) {
            name = "world";
        }
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Hello, " + name + "!</h1>");
        pw.flush();
    }
}
```

然后，我们编写一个`main()`方法，启动Tomcat服务器：

```
public class Main {
    public static void main(String[] args) throws Exception {
        // 启动Tomcat:
        Tomcat tomcat = new Tomcat();
        tomcat.setPort(Integer.getInteger("port", 8080));
        tomcat.getConnector();
        // 创建webapp:
        Context ctx = tomcat.addWebapp("", new File("src/main/webapp").getAbsolutePath());
        WebResourceRoot resources = new StandardRoot(ctx);
        resources.addPreResources(
                new DirResourceSet(resources, "/WEB-INF/classes", new File("target/classes").getAbsolutePath(), "/"));
        ctx.setResources(resources);
        tomcat.start();
        tomcat.getServer().await();
    }
}
```

这样，我们直接运行`main()`方法，即可启动嵌入式Tomcat服务器，然后，通过预设的`tomcat.addWebapp("", new File("src/main/webapp"))`，Tomcat会自动加载当前工程作为根webapp，可直接在浏览器访问`http://localhost:8080/`：

通过`main()`方法启动Tomcat服务器并加载我们自己的webapp有如下好处：

1. 启动简单，无需下载Tomcat或安装任何IDE插件；
2. 调试方便，可在IDE中使用断点调试；
3. 使用Maven创建war包后，也可以正常部署到独立的Tomcat服务器中。

对SpringBoot有所了解的童鞋可能知道，SpringBoot也支持在`main()`方法中一行代码直接启动Tomcat，并且还能方便地更换成Jetty等其他服务器。它的启动方式和我们介绍的是基本一样的，后续涉及到SpringBoot的部分我们还会详细讲解。

练习

使用嵌入式Tomcat运行Servlet

注意：引入的Tomcat的scope为`provided`，在Idea下运行时，需要设置`Run/Debug Configurations`，选择`Application - Main`，钩上`Include dependencies with "Provided" scope`，这样才能让Idea在运行时把Tomcat相关依赖包自动添加到classpath中。

小结

开发Servlet时，推荐使用`main()`方法启动嵌入式Tomcat服务器并加载当前工程的webapp，便于开发调试，且不影响打包部署，能极大地提升开发效率。

一个Web App就是由一个或多个Servlet组成的，每个Servlet通过注解说明自己能处理的路径。例如：

```
@WebServlet(urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    ...
}
```

上述`HelloServlet`能处理`/hello`这个路径的请求。

早期的`Servlet`需要在`web.xml`中配置映射路径，但最新`Servlet`版本只需要通过注解就可以完成映射。

因为浏览器发送请求的时候，还会有请求方法（HTTP Method）：即GET、POST、PUT等不同类型的请求。因此，要处理GET请求，我们要覆写`doGet()`方法：

```
@WebServlet(urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        ...
    }
}
```

类似的，要处理POST请求，就需要覆写`doPost()`方法。

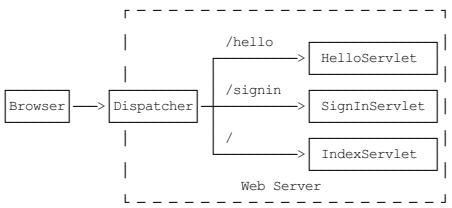
如果没有覆写`doPost()`方法，那么`HelloServlet`能不能处理`POST /hello`请求呢？

我们查看一下`HttpServlet`的`doPost()`方法就一目了然了：它会直接返回405或400错误。因此，一个Servlet如果映射到`/hello`，那么所有请求方法都会由这个Servlet处理，至于能不能返回200成功响应，要看有没有覆写对应的请求方法。

一个Webapp完全可以有多个Servlet，分别映射不同的路径。例如：

```
@WebServlet(urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    ...
}

@WebServlet(urlPatterns = "/signin")
public class SignInServlet extends HttpServlet {
    ...
}

@WebServlet(urlPatterns = "/")
public class IndexServlet extends HttpServlet {
    ...
}
```

浏览器发出的HTTP请求总是由Web Server先接收，然后，根据Servlet配置的映射，不同的路径转发到不同的Servlet：



这种根据路径转发的功能我们一般称为Dispatch。映射到`/`的`IndexServlet`比较特殊，它实际上会接收所有未匹配的路径，相当于`/*`，因为Dispatcher的逻辑可以用伪代码实现如下：

```
String path = ...
if (path.equals("/hello")) {
    dispatchTo(helloServlet);
} else if (path.equals("/signin")) {
    dispatchTo(signinServlet);
} else {
    // 所有未匹配的路径均转发到"/"
    dispatchTo(indexServlet);
}
```

所以我们在浏览器输入一个`http://localhost:8080/abc`也会看到`IndexServlet`生成的页面。

HttpServletRequest

`HttpServletRequest`封装了一个HTTP请求，它实际上是从`ServletRequest`继承而来。最早设计Servlet时，设计者希望Servlet不仅能处理HTTP，也能处理类似SMTP等其他协议，因此，单独抽出了`ServletRequest`接口，但实际上除了HTTP外，并没有其他协议会用Servlet处理，所以这是一个过度设计。

我们通过`HttpServletRequest`提供的接口方法可以拿到HTTP请求的几乎全部信息，常用的方法有：

- getMethod()：返回请求方法，例如，`"GET"`，`"POST"`；
- getRequestURI()：返回请求路径，但不包括请求参数，例如，`"/hello"`；
- getQueryString()：返回请求参数，例如，`"name=Bob&a=1&b=2"`；
- getParameter(name)：返回请求参数，GET请求从URL读取参数，POST请求从Body中读取参数；
- getContentType()：获取请求Body的类型，例如，`"application/x-www-form-urlencoded"`；
- getContextPath()：获取当前Webapp挂载的路径，对于ROOT来说，总是返回空字符串`""`；
- getCookies()：返回请求携带的所有Cookie；
- getHeader(name)：获取指定的Header，对Header名称不区分大小写；
- getHeaderNames()：返回所有Header名称；
- getInputStream()：如果该请求带有HTTP Body，该方法将打开一个输入流用于读取Body；
- getReader()：和getInputStream()类似，但打开的是Reader；
- getRemoteAddr()：返回客户端的IP地址；
- getScheme()：返回协议类型，例如，`"http"`，`"https"`；

此外，`HttpServletRequest`还有两个方法：`setAttribute()`和`getAttribute()`，可以给当前`HttpServletRequest`对象附加多个Key-Value，相当于把`HttpServletRequest`当作一个`Map<String, Object>`使用。

调用`HttpServletRequest`的方法时，注意务必阅读接口方法的文档说明，因为有的方法会返回`null`，例如`getQueryString()`的文档就写了：

```
... This method returns null if the URL does not have a query string...
```

HttpServletResponse

`HttpServletResponse`封装了一个HTTP响应。由于HTTP响应必须先发送Header，再发送Body，所以，操作`HttpServletResponse`对象时，必须先调用设置Header的方法，最后调用发送Body的方法。

常用的设置Header的方法有：

- setStatus(sc)：设置响应代码，默认是200；
- setContentType(type)：设置Body的类型，例如，"text/html"；
- setCharacterEncoding(charset)：设置字符编码，例如，"UTF-8"；
- setHeader(name, value)：设置一个Header的值；
- addCookie(cookie)：给响应添加一个Cookie；
- addHeader(name, value)：给响应添加一个Header，因为HTTP协议允许有多个相同的Header；

写入响应时，需要通过getOutputStream()获取写入流，或者通过getWriter()获取字符流，二者只能获取其中一个。

写入响应前，无需设置setContentLength()，因为底层服务器会根据写入的字节数自动设置，如果写入的数据量很小，实际上会先写入缓冲区，如果写入的数据量很大，服务器会自动采用Chunked编码让浏览器能识别数据结束符而不需要设置Content-Length头。

但是，写入完毕后调用flush()却是必须的，因为大部分Web服务器都基于HTTP/1.1协议，会复用TCP连接。如果没有调用flush()，将导致缓冲区的内容无法及时发送到客户端。此外，写入完毕后千万不要调用close()，原因同样是因为会复用TCP连接，如果关闭写入流，将关闭TCP连接，使得Web服务器无法复用此TCP连接。

写入完毕后对输出流调用flush()而不是close()方法！

有了HttpServletRequest和HttpServletResponse这两个高级接口，我们就不需要直接处理HTTP协议。注意到具体的实现类是由各服务器提供的，而我们编写的Web应用程序只关心接口方法，并不需要关心具体实现的子类。

Servlet多线程模型

一个Servlet类在服务器中只有一个实例，但对于每个HTTP请求，Web服务器会使用多线程执行请求。因此，一个Servlet的doGet()、doPost()等处理请求的方法是多线程并发执行的。如果Servlet中定义了字段，要注意多线程并发访问的问题：

```
public class HelloServlet extends HttpServlet {
    private Map<String, String> map = new ConcurrentHashMap<>();

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 注意读写map字段是多线程并发的:
        this.map.put(key, value);
    }
}
```

对于每个请求，Web服务器会创建唯一的HttpServletRequest和HttpServletResponse实例，因此，HttpServletRequest和HttpServletResponse实例只有在当前处理线程中有效，它们总是局部变量，不存在多线程共享的问题。

小结

一个Webapp中的多个Servlet依靠路径映射来处理不同的请求；

映射为/的Servlet可处理所有“未匹配”的请求；

如何处理请求取决于Servlet覆写的对应方法；

Web服务器通过多线程处理HTTP请求，一个Servlet的处理方法可以由多线程并发执行。

Redirect

重定向是指当浏览器请求一个URL时，服务器返回一个重定向指令，告诉浏览器地址已经变了，麻烦使用新的URL再重新发送新请求。

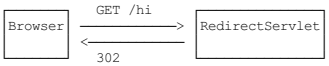
例如，我们已经编写了一个能处理/hello的HelloServlet，如果收到的路径为/hi，希望能重定向到/hello，可以再编写一个RedirectServlet：

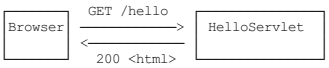
```
@WebServlet(urlPatterns = "/hi")
public class RedirectServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 构造重定向的路径:
        String name = req.getParameter("name");
        String redirectToUrl = "/hello" + (name == null ? "" : "?name=" + name);
        // 发送重定向响应:
        resp.sendRedirect(redirectToUrl);
    }
}
```

如果浏览器发送GET /hi请求，RedirectServlet将处理此请求。由于RedirectServlet在内部又发送了重定向响应，因此，浏览器会收到如下响应：

```
HTTP/1.1 302 Found
Location: /hello
```

当浏览器收到302响应后，它会立刻根据Location的指示发送一个新的GET /hello请求，这个过程就是重定向：





观察Chrome浏览器的网络请求，可以看到两次HTTP请求：

并且浏览器的地址栏路径自动更新为/hello。

重定向有两种：一种是302响应，称为临时重定向，一种是301响应，称为永久重定向。两者的区别是，如果服务器发送301永久重定向响应，浏览器会缓存/hi到/hello这个重定向的关联，下次请求/hi的时候，浏览器就直接发送/hello请求了。

重定向有什么作用？重定向的目的是当Web应用升级后，如果请求路径发生了变化，可以将原来的路径重定向到新路径，从而避免浏览器请求原路径找不到资源。

HttpServletResponse提供了快捷的redirect()方法实现302重定向。如果要实现301永久重定向，可以这么写：

```
resp.setStatus(HttpServletResponse.SC_MOVED_PERMANENTLY); // 301
resp.setHeader("Location", "/hello");
```

Forward

Forward是指内部转发。当一个Servlet处理请求的时候，它可以决定自己不继续处理，而是转发给另一个Servlet处理。

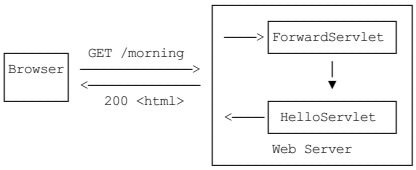
例如，我们已经编写了一个能处理/hello的HelloServlet，继续编写一个能处理/morning的ForwardServlet：

```
@WebServlet(urlPatterns = "/morning")
public class ForwardServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        req.getRequestDispatcher("/hello").forward(req, resp);
    }
}
```

ForwardServlet在收到请求后，它并不自己发送响应，而是把请求和响应都转发给路径为/hello的Servlet，即下面的代码：

```
req.getRequestDispatcher("/hello").forward(req, resp);
```

后续请求的处理实际上是由HelloServlet完成的。这种处理方式称为转发（Forward），我们用流程图画出来如下：



转发和重定向的区别在于，转发是在Web服务器内部完成的，对浏览器来说，它只发出了一个HTTP请求：

注意到使用转发的时候，浏览器的地址栏路径仍然是/morning，浏览器并不知道该请求在Web服务器内部实际上做了一次转发。

练习

使用重定向和转发

小结

使用重定向时，浏览器知道重定向规则，并且会自动发起新的HTTP请求；

使用转发时，浏览器并不知道服务器内部的转发逻辑。

在Web应用程序中，我们经常要跟踪用户身份。当一个用户登录成功后，如果他继续访问其他页面，Web程序如何才能识别出该用户身份？

因为HTTP协议是一个无状态协议，即Web应用程序无法区分收到的两个HTTP请求是否是同一个浏览器发出的。为了跟踪用户状态，服务器可以向浏览器分配一个唯一ID，并以Cookie的形式发送到浏览器，浏览器在后续访问时总是附带此Cookie，这样，服务器就可以识别用户身份。

Session

我们把这种基于唯一ID识别用户身份的机制称为Session。每个用户第一次访问服务器后，会自动获得一个Session ID。如果用户在一段时间内没有访问服务器，那么Session会自动失效，下次即使带着上次分配的Session ID访问，服务器也认为这是一个新用户，会分配新的Session ID。

JavaEE的Servlet机制内建了对Session的支持。我们以登录为例，当一个用户登录成功后，我们就可以把这个用户的名字放入一个HttpSession对象，以便后续访问其他页面的时候，能直接从HttpSession取出用户名：

```
@WebServlet(urlPatterns = "/signin")
public class SignInServlet extends HttpServlet {
    // 模拟一个数据库:
    private Map<String, String> users = Map.of("bob", "bob123", "alice", "alice123", "tom", "tomcat");

    // GET请求时显示登录页:
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("text/html");
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Sign In</h1>");
        pw.write("<form action=\"/signin\" method=\"post\">");
        pw.write("<p>Username: <input name=\"username\"></p>");
        pw.write("<p>Password: <input name=\"password\" type=\"password\"></p>");
        pw.write("<p><button type=\"submit\">Sign In</button> <a href=\"/\">Cancel</a></p>");
        pw.write("</form>");
        pw.flush();
    }

    // POST请求时处理用户登录:
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String name = req.getParameter("username");
        String password = req.getParameter("password");
        String expectedPassword = users.get(name.toLowerCase());
        if (expectedPassword != null && expectedPassword.equals(password)) {
            // 登录成功:
            req.getSession().setAttribute("user", name);
            resp.sendRedirect("/");
        } else {
            resp.sendError(HttpServletResponse.SC_FORBIDDEN);
        }
    }
}
```

上述SignInServlet在判断用户登录成功后，立刻将用户名放入当前HttpSession中：

```
HttpSession session = req.getSession();
session.setAttribute("user", name);
```

在IndexServlet中，可以从HttpSession取出用户名：

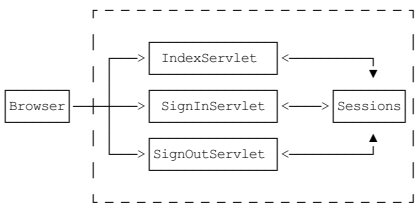
```
@WebServlet(urlPatterns = "/")
public class IndexServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 从HttpSession获取当前用户名:
        String user = (String) req.getSession().getAttribute("user");
        resp.setContentType("text/html");
        resp.setCharacterEncoding("UTF-8");
        resp.setHeader("X-Powered-By", "JavaEE Servlet");
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Welcome, " + (user != null ? user : "Guest") + "</h1>");
        if (user == null) {
            // 未登录，显示登录链接:
            pw.write("<p><a href=\"/signin\">Sign In</a></p>");
        } else {
            // 已登录，显示登出链接:
            pw.write("<p><a href=\"/signout\">Sign Out</a></p>");
        }
        pw.flush();
    }
}
```

如果用户已登录，可以通过访问/signout登出。登出逻辑就是从HttpSession中移除用户相关信息：

```
@WebServlet(urlPatterns = "/signout")
public class SignOutServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 从HttpSession移除用户名:
        req.getSession().removeAttribute("user");
        resp.sendRedirect("/");
```

```
    }
}
```

对于Web应用程序来说，我们总是通过HttpSession这个高级接口访问当前Session。如果要深入理解Session原理，可以认为Web服务器在内存中自动维护了一个ID到HttpSession的映射表，我们可以用下图表示：



而服务器识别Session的关键就是依靠一个名为JSESSIONID的Cookie。在Servlet中第一次调用req.getSession()时，Servlet容器自动创建一个Session ID，然后通过一个名为JSESSIONID的Cookie发送给浏览器：

这里要注意的几点是：

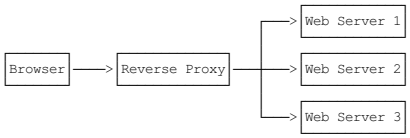
- JSESSIONID是由Servlet容器自动创建的，目的是维护一个浏览器会话，它和我们的登录逻辑没有关系；
- 登录和登出的业务逻辑是我们自己根据HttpSession是否存在一个"user"的Key判断的，登出后，Session ID并不会改变；
- 即使没有登录功能，仍然可以使用HttpSession追踪用户，例如，放入一些用户配置信息等。

除了使用Cookie机制可以实现Session外，还可以通过隐藏表单、URL末尾附加ID来追踪Session。这些机制很少使用，最常用的Session机制仍然是Cookie。

使用Session时，由于服务器把所有用户的Session都存储在内存中，如果遇到内存不足的情况，就需要把部分不活动的Session序列化到磁盘上，这会大大降低服务器的运行效率，因此，放入Session的对象要小，通常我们放入一个简单的User对象就足够了：

```
public class User {
    public long id; // 唯一标识
    public String email;
    public String name;
}
```

在使用多台服务器构成集群时，使用Session会遇到一些额外的问题。通常，多台服务器集群使用反向代理作为网站入口：



如果多台Web Server采用无状态集群，那么反向代理总是以轮询方式将请求依次转发给每台Web Server，这会造成一个用户在Web Server 1存储的Session信息，在Web Server 2和3上并不存在，即从Web Server 1登录后，如果后续请求被转发到Web Server 2或3，那么用户看到的仍然是未登录状态。

要解决这个问题，方案一是在所有Web Server之间进行Session复制，但这样会严重消耗网络带宽，并且，每个Web Server的内存均存储所有用户的Session，内存使用率很低。

另一个方案是采用粘滞会话（Sticky Session）机制，即反向代理在转发请求的时候，总是根据JSESSIONID的值判断，相同的JSESSIONID总是转发到固定的Web Server，但这需要反向代理的支持。

无论采用何种方案，使用Session机制，会使得Web Server的集群很难扩展，因此，Session适用于中小型Web应用程序。对于大型Web应用程序来说，通常需要避免使用Session机制。

Cookie

实际上，Servlet提供的HttpSession本质上就是通过一个名为JSESSIONID的Cookie来跟踪用户会话的。除了这个名称外，其他名称的Cookie我们可以任意使用。

如果我们想要设置一个Cookie，例如，记录用户选择的语言，可以编写一个LanguageServlet：

```
@WebServlet(urlPatterns = "/pref")
public class LanguageServlet extends HttpServlet {

    private static final Set<String> LANGUAGES = Set.of("en", "zh");

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        String lang = req.getParameter("lang");
        if (LANGUAGES.contains(lang)) {
            // 创建一个新的Cookie:
            Cookie cookie = new Cookie("lang", lang);
            // 该Cookie生效的路径范围:
            cookie.setPath("/");
            // 该Cookie有效期:
            cookie.setMaxAge(8640000); // 8640000秒=100天
            // 将该Cookie添加到响应:
            resp.addCookie(cookie);
        }
        resp.sendRedirect("/");
    }
}
```

创建一个新Cookie时，除了指定名称和值以外，通常需要设置setPath("/")，浏览器根据此前缀决定是否发送Cookie。如果一个Cookie调用了setPath("/user/")，那么浏览器只有在请求以/user/开头的路径时才会附加此Cookie。通过setMaxAge()设置Cookie的有效期，单位为秒，最后通过resp.addCookie()把它添加到响应。

如果访问的是https网页，还需要调用setSecure(true)，否则浏览器不会发送该Cookie。

因此，务必注意：浏览器在请求某个URL时，是否携带指定的Cookie，取决于Cookie是否满足以下所有要求：

- URL前缀是设置Cookie时的Path；
- Cookie在有效期内；
- Cookie设置了secure时必须以https访问。

我们可以在浏览器看到服务器发送的Cookie：

如果我们要读取Cookie，例如，在IndexServlet中，读取名为lang的Cookie以获取用户设置的语言，可以写一个方法如下：

```
private String parseLanguageFromCookie(HttpServletRequest req) {
    // 获取请求附带的所有Cookie:
    Cookie[] cookies = req.getCookies();
    // 如果获取到Cookie:
    if (cookies != null) {
        // 循环每个Cookie:
        for (Cookie cookie : cookies) {
            // 如果Cookie名称为lang:
            if (cookie.getName().equals("lang")) {
```

```
                // 返回Cookie的值:
                return cookie.getValue();
            }
        }
    }
    // 返回默认值:
    return "en";
}
```

可见，读取Cookie主要依靠遍历`HttpServletRequest`附带的所有Cookie。

练习

使用Session和Cookie

小结

Servlet容器提供了Session机制以跟踪用户；

默认的Session机制是以Cookie形式实现的，Cookie名称为`JSESSIONID`；

通过读写Cookie可以在客户端设置用户偏好等。

我们从前面的章节可以看到，Servlet就是一个能处理HTTP请求，发送HTTP响应的小程序，而发送响应无非就是获取`PrintWriter`，然后输出HTML：

```
PrintWriter pw = resp.getWriter();
pw.write("<html>");
pw.write("<body>");
pw.write("<h1>Welcome, " + name + "!</h1>");
pw.write("</body>");
pw.write("</html>");
pw.flush();
```

只不过，用PrintWriter输出HTML比较痛苦，因为不但要正确编写HTML，还需要插入各种变量。如果想在Servlet中输出一个类似新浪首页的HTML，写对HTML基本上不太可能。

那有没有更简单的输出HTML的办法？

有！

我们可以使用JSP。

JSP是Java Server Pages的缩写，它的文件必须放到`/src/main/webapp`下，文件名必须以`.jsp`结尾，整个文件与HTML并无太大区别，但需要插入变量，或者动态输出的地方，使用特殊指令`<% ... %>`。

我们来编写一个`hello.jsp`，内容如下：

```
<html>
<head>
    <title>Hello World - JSP</title>
</head>
<body>
    <%-- JSP Comment --%>
    <h1>Hello World!</h1>
    <p>
    <%
        out.println("Your IP address is ");
    %>
    <span style="color:red">
        <%= request.getRemoteAddr() %>
    </span>
    </p>
</body>
</html>
```

整个JSP的内容实际上是一个HTML，但是稍有不同：

- 包含在`<%--`和`--%>`之间的是JSP的注释，它们会被完全忽略；
- 包含在`<%`和`%>`之间的是Java代码，可以编写任意Java代码；
- 如果使用`<%= xxx %>`则可以快捷输出一个变量的值。

JSP页面内置了几个变量：

- out：表示HttpServletResponse的PrintWriter；
- session：表示当前HttpSession对象；
- request：表示HttpServletRequest对象。

这几个变量可以直接使用。

访问JSP页面时，直接指定完整路径。例如，`http://localhost:8080/hello.jsp`，浏览器显示如下：

JSP和Servlet有什么区别？其实它们没有任何区别，因为JSP在执行前首先被编译成一个Servlet。在Tomcat的临时目录下，可以找到一个`hello_jsp.java`的源文件，这个文件就是Tomcat把JSP自动转换成的Servlet源码：

```
package org.apache.jsp;
import ...

public final class hello_jsp extends org.apache.jasper.runtime.HttpJspBase
    implements org.apache.jasper.runtime.JspSourceDependent,
               org.apache.jasper.runtime.JspSourceImports {

    ...

    public void _jspService(final javax.servlet.http.HttpServletRequest request, final javax.servlet.http.HttpServletResponse response)
        throws java.io.IOException, javax.servlet.ServletException {
        ...
        out.write("<html>\n");
        out.write("<head>\n");
        out.write("    <title>Hello World - JSP</title>\n");
        out.write("</head>\n");
        out.write("<body>\n");
        ...
    }
    ...
}
```

可见JSP本质上就是一个Servlet，只不过无需配置映射路径，Web Server会根据路径查找对应的`.jsp`文件，如果找到了，就自动编译成Servlet再执行。在服务器运行过程中，如果修改了JSP的内容，那么服务器会自动重新编译。

JSP高级功能

JSP的指令非常复杂，除了`<% ... %>`外，JSP页面本身可以通过`page`指令引入Java类：

```
<%@ page import="java.io.*" %>
<%@ page import="java.util.*" %>
```

这样后续的Java代码才能引用简单类名而不是完整类名。

使用include指令可以引入另一个JSP文件：

```
<html>
<body>
    <%@ include file="header.jsp"%>
    <h1>Index Page</h1>
    <%@ include file="footer.jsp"%>
</body>
```

JSP Tag

JSP还允许自定义输出的tag，例如：

```
<c:out value = "${sessionScope.user.name}"/>
```

JSP Tag需要正确引入taglib的jar包，并且还需要正确声明，使用起来非常复杂，对于页面开发来说，*不推荐*使用JSP Tag，因为我们后续会介绍更简单的模板引擎，这里我们不再介绍如何使用taglib。

练习

编写一个简单的JSP文件，输出当前日期和时间。

JSP练习

小结

JSP是一种在HTML中嵌入动态输出的文件，它和Servlet正好相反，Servlet是在Java代码中嵌入输出HTML；

JSP可以引入并使用JSP Tag，但由于其语法复杂，不推荐使用；

JSP本身目前已经很少使用，我们只需要了解其基本用法即可。

我们通过前面的章节可以看到：

- Servlet适合编写Java代码，实现各种复杂的业务逻辑，但不适合输出复杂的HTML；
- JSP适合编写HTML，并在其中插入动态内容，但不适合编写复杂的Java代码。

能否将两者结合起来，发挥各自的优点，避免各自的缺点？

答案是肯定的。我们来看一个具体的例子。

假设我们已经编写了几个JavaBean：

```
public class User {
    public long id;
    public String name;
    public School school;
}

public class School {
    public String name;
    public String address;
}
```

在UserServlet中，我们可以从数据库读取User、School等信息，然后，把读取到的JavaBean先放到HttpServletRequest中，再通过forward()传给user.jsp处理：

```
@WebServlet(urlPatterns = "/user")
public class UserServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 假装从数据库读取:
        School school = new School("No.1 Middle School", "101 South Street");
        User user = new User(123, "Bob", school);
        // 放入Request中:
        req.setAttribute("user", user);
        // forward给user.jsp:
        req.getRequestDispatcher("/WEB-INF/user.jsp").forward(req, resp);
    }
}
```

在user.jsp中，我们只负责展示相关JavaBean的信息，不需要编写访问数据库等复杂逻辑：

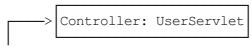
```
<%@ page import="com.itranswarp.learnjava.bean.*"%>
<%
    User user = (User) request.getAttribute("user");
%>
<html>
<head>
    <title>Hello World - JSP</title>
</head>
<body>
    <h1>Hello <%= user.name %>!</h1>
    <p>School Name:
    <span style="color:red">
        <%= user.school.name %>
    </span>
    </p>
    <p>School Address:
    <span style="color:red">
        <%= user.school.address %>
    </span>
    </p>
</body>
</html>
```

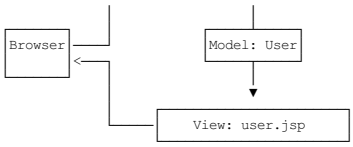
请注意几点：

- 需要展示的User被放入HttpServletRequest中以便传递给JSP，因为一个请求对应一个HttpServletRequest，我们也无需清理它，处理完该请求后HttpServletRequest实例将被丢弃；
- 把user.jsp放到/WEB-INF/目录下，是因为WEB-INF是一个特殊目录，Web Server会阻止浏览器对WEB-INF目录下任何资源的访问，这样就防止用户通过/user.jsp路径直接访问到JSP页面；
- JSP页面首先从request变量获取User实例，然后在页面中直接输出，此处未考虑HTML的转义问题，有潜在安全风险。

我们在浏览器访问http://localhost:8080/user，请求首先由UserServlet处理，然后交给user.jsp渲染：

我们把UserServlet看作业务逻辑处理，把User看作模型，把user.jsp看作渲染，这种设计模式通常被称为MVC：Model-View-Controller，即UserServlet作为控制器（Controller），User作为模型（Model），user.jsp作为视图（View），整个MVC架构如下：





使用MVC模式的好处是，Controller专注于业务处理，它的处理结果就是Model。Model可以是一个JavaBean，也可以是一个包含多个对象的Map，Controller只负责把Model传递给View，View只负责把Model给“渲染”出来，这样，三者职责明确，且开发更简单，因为开发Controller时无需关注页面，开发View时无需关心如何创建Model。

MVC模式广泛地应用在Web页面和传统的桌面程序中，我们在这里通过Servlet和JSP实现了一个简单的MVC模型，但它还不够简洁和灵活，后续我们会介绍更简单的Spring MVC开发。

练习

使用MVC开发

小结

MVC模式是一种分离业务逻辑和显示逻辑的设计模式，广泛应用在Web和桌面应用程序。

通过结合Servlet和JSP的MVC模式，我们可以发挥二者各自的优点：

- Servlet实现业务逻辑；
- JSP实现展示逻辑。

但是，直接把MVC搭在Servlet和JSP之上还是不太好，原因如下：

- Servlet提供的接口仍然偏底层，需要实现Servlet调用相关接口；
- JSP对页面开发不友好，更好的替代品是模板引擎；
- 业务逻辑最好由纯粹的Java类实现，而不是强迫继承自Servlet。

能不能通过普通的Java类实现MVC的Controller？类似下面的代码：

```
public class UserController {
    @GetMapping("/signin")
    public ModelAndView signin() {
        ...
    }

    @PostMapping("/signin")
    public ModelAndView doSignin(SignInBean bean) {
        ...
    }

    @GetMapping("/signout")
    public ModelAndView signout(HttpSession session) {
        ...
    }
}
```

上面的这个Java类每个方法都对应一个GET或POST请求，方法返回值是`ModelAndView`，它包含一个View的路径以及一个Model，这样，再由MVC框架处理后返回给浏览器。

如果是GET请求，我们希望MVC框架能直接把URL参数按方法参数对应起来然后传入：

```
@GetMapping("/hello")
public ModelAndView hello(String name) {
    ...
}
```

如果是POST请求，我们希望MVC框架能直接把Post参数变成一个JavaBean后通过方法参数传入：

```
@PostMapping("/signin")
public ModelAndView doSignin(SignInBean bean) {
    ...
}
```

为了增加灵活性，如果Controller的方法在处理请求时需要访问`HttpServletRequest`、`HttpServletResponse`、`HttpSession`这些实例时，只要方法参数有定义，就可以自动传入：

```
@GetMapping("/signout")
public ModelAndView signout(HttpSession session) {
    ...
}
```

以上就是我们在设计MVC框架时，上层代码所需要的一切信息。

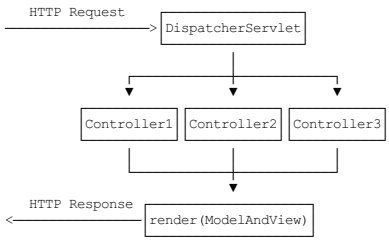
设计MVC框架

如何设计一个MVC框架？在上文中，我们已经定义了上层代码编写Controller的一切接口信息，并且并不要求实现特定接口，只需返回`ModelAndView`对象，该对象包含一个`View`和一个`Model`。实际上`View`就是模板的路径，而`Model`可以用一个`Map<String, Object>`表示，因此，`ModelAndView`定义非常简单：

```
public class ModelAndView {
    Map<String, Object> model;
    String view;
}
```

比较复杂的是我们需要在MVC框架中创建一个接收所有请求的`Servlet`，通常我们把它命名为`DispatcherServlet`，它总是映射到`/`，然后，根据不同的Controller的方法定义的`@Get`或`@Post`的Path决定调用哪个方法，最后，获得方法返回的`ModelAndView`后，渲染模板，写入`HttpServletResponse`，即完成了整个MVC的处理。

这个MVC的架构如下：



其中，`DispatcherServlet`以及如何渲染均由MVC框架实现，在MVC框架之上只需要编写每一个Controller。

我们来看看如何编写最复杂的`DispatcherServlet`。首先，我们需要存储请求路径到某个具体方法的映射：

```
@WebServlet(urlPatterns = "/")
```

```
public class DispatcherServlet extends HttpServlet {
    private Map<String, GetDispatcher> getMappings = new HashMap<>();
    private Map<String, PostDispatcher> postMappings = new HashMap<>();
}
```

处理一个GET请求是通过`GetDispatcher`对象完成的，它需要如下信息：

```
class GetDispatcher {
    Object instance; // Controller实例
    Method method; // Controller方法
    String[] parameterNames; // 方法参数名称
    Class<?>[] parameterClasses; // 方法参数类型
}
```

有了以上信息，就可以定义`invoke()`来处理真正的请求：

```
class GetDispatcher {
    ...
    public ModelAndView invoke(HttpServletRequest request, HttpServletResponse response) {
        Object[] arguments = new Object[parameterClasses.length];
        for (int i = 0; i < parameterClasses.length; i++) {
            String parameterName = parameterNames[i];
            Class<?> parameterClass = parameterClasses[i];
            if (parameterClass == HttpServletRequest.class) {
                arguments[i] = request;
            } else if (parameterClass == HttpServletResponse.class) {
                arguments[i] = response;
            } else if (parameterClass == HttpSession.class) {
                arguments[i] = request.getSession();
            } else if (parameterClass == int.class) {
                arguments[i] = Integer.valueOf(getOrDefault(request, parameterName, "0"));
            } else if (parameterClass == long.class) {
                arguments[i] = Long.valueOf(getOrDefault(request, parameterName, "0"));
            } else if (parameterClass == boolean.class) {
                arguments[i] = Boolean.valueOf(getOrDefault(request, parameterName, "false"));
            } else if (parameterClass == String.class) {
                arguments[i] = getOrDefault(request, parameterName, "");
            } else {
                throw new RuntimeException("Missing handler for type: " + parameterClass);
            }
        }
        return (ModelAndView) this.method.invoke(this.instance, arguments);
    }

    private String getOrDefault(HttpServletRequest request, String name, String defaultValue) {
        String s = request.getParameter(name);
        return s == null ? defaultValue : s;
    }
}
```

上述代码比较繁琐，但逻辑非常简单，即通过构造某个方法需要的所有参数列表，使用反射调用该方法后返回结果。

类似的，`PostDispatcher`需要如下信息：

```
class PostDispatcher {
    Object instance; // Controller实例
    Method method; // Controller方法
    Class<?>[] parameterClasses; // 方法参数类型
    ObjectMapper objectMapper; // JSON映射
}
```

和GET请求不同，POST请求严格地来说不能有URL参数，所有数据都应当从Post Body中读取。这里我们为了简化处理，*只支持*JSON格式的POST请求，这样，把Post数据转化为JavaBean就非常容易。

```
class PostDispatcher {
    ...
    public ModelAndView invoke(HttpServletRequest request, HttpServletResponse response) {
        Object[] arguments = new Object[parameterClasses.length];
        for (int i = 0; i < parameterClasses.length; i++) {
            Class<?> parameterClass = parameterClasses[i];
            if (parameterClass == HttpServletRequest.class) {
                arguments[i] = request;
            } else if (parameterClass == HttpServletResponse.class) {
                arguments[i] = response;
            } else if (parameterClass == HttpSession.class) {
                arguments[i] = request.getSession();
            } else {
                // 读取JSON并解析为JavaBean:
                BufferedReader reader = request.getReader();
                arguments[i] = this.objectMapper.readValue(reader, parameterClass);
            }
        }
        return (ModelAndView) this.method.invoke(instance, arguments);
    }
}
```

最后，我们来实现整个`DispatcherServlet`的处理流程，以`doGet()`为例：

```
public class DispatcherServlet extends HttpServlet {
    ...
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("text/html");
        resp.setCharacterEncoding("UTF-8");
        String path = req.getRequestURI().substring(req.getContextPath().length());
        // 根据路径查找GetDispatcher:
        GetDispatcher dispatcher = this.getMappings.get(path);
        if (dispatcher == null) {
            // 未找到返回404:
            resp.sendError(404);
            return;
        }
        // 调用Controller方法获得返回值:
        ModelAndView mv = dispatcher.invoke(req, resp);
        // 允许返回null:
        if (mv == null) {
            return;
        }
        // 允许返回`redirect:`开头的view表示重定向:
        if (mv.view.startsWith("redirect:")) {
            resp.sendRedirect(mv.view.substring(9));
            return;
        }
        // 将模板引擎渲染的内容写入响应:
        PrintWriter pw = resp.getWriter();
        this.viewEngine.render(mv, pw);
        pw.flush();
    }
}
```

这里有几个小改进：

- 允许Controller方法返回`null`，表示内部已自行处理完毕；
- 允许Controller方法返回以`redirect:`开头的view名称，表示一个重定向。

这样使得上层代码编写更灵活。例如，一个显示用户资料的请求可以这样写：

```
@GetMapping("/user/profile")
public ModelAndView profile(HttpServletResponse response, HttpSession session) {
    User user = (User) session.getAttribute("user");
    if (user == null) {
        // 未登录，跳转到登录页:
        return new ModelAndView("redirect:/signin");
    }
    if (!user.isManager()) {
        // 权限不够，返回403:
        response.sendError(403);
        return null;
    }
    return new ModelAndView("/profile.html", Map.of("user", user));
}
```

最后一步是在`DispatcherServlet`的`init()`方法中初始化所有Get和Post的映射，以及用于渲染的模板引擎：

```
public class DispatcherServlet extends HttpServlet {
    private Map<String, GetDispatcher> getMappings = new HashMap<>();
    private Map<String, PostDispatcher> postMappings = new HashMap<>();
    private ViewEngine viewEngine;

    @Override
    public void init() throws ServletException {
        this.getMappings = scanGetInControllers();
        this.postMappings = scanPostInControllers();
        this.viewEngine = new ViewEngine(getServletContext());
    }
    ...
}
```

如何扫描所有Controller以获取所有标记有`@GetMapping`和`@PostMapping`的方法？当然是使用反射了。虽然代码比较繁琐，但我们相信各位童鞋可以轻松实现。

这样，整个MVC框架就搭建完毕。

实现渲染

有的童鞋对如何使用模板引擎进行渲染有疑问，即如何实现上述的`ViewEngine`？其实`ViewEngine`非常简单，只需要实现一个简单的`render()`方法：

```
public class ViewEngine {
    public void render(ModelAndView mv, Writer writer) throws IOException {
        String view = mv.view;
        Map<String, Object> model = mv.model;
        // 根据view找到模板文件:
        Template template = getTemplateByPath(view);
        // 渲染并写入Writer:
        template.write(writer, model);
    }
}
```

Java有很多开源的模板引擎，常用的有：

- Thymeleaf
- FreeMarker
- Velocity

他们的用法都大同小异。这里我们推荐一个使用Jinja语法的模板引擎Pebble，它的特点是语法简单，支持模板继承，编写出来的模板类似：

```
<html>
<body>
  <ul>
  {% for user in users %}
    <li><a href="{{ user.url }}">{{ user.username }}</a></li>
  {% endfor %}
  </ul>
</body>
</html>
```

即变量用`{{ xxx }}`表示，控制语句用`{% xxx %}`表示。

使用Pebble渲染只需要如下几行代码：

```
public class ViewEngine {
    private final PebbleEngine engine;

    public ViewEngine(ServletContext servletContext) {
        // 定义一个ServletLoader用于加载模板:
        ServletLoader loader = new ServletLoader(servletContext);
        // 模板编码:
        loader.setCharset("UTF-8");
        // 模板前缀，这里默认模板必须放在`/WEB-INF/templates`目录:
        loader.setPrefix("/WEB-INF/templates");
        // 模板后缀:
        loader.setSuffix("");
        // 创建Pebble实例:
        this.engine = new PebbleEngine.Builder()
            .autoEscaping(true) // 默认打开HTML字符转义，防止XSS攻击
            .cacheActive(false) // 禁用缓存使得每次修改模板可以立刻看到效果
            .loader(loader).build();
    }

    public void render(ModelAndView mv, Writer writer) throws IOException {
        // 查找模板:
        PebbleTemplate template = this.engine.getTemplate(mv.view);
        // 渲染:
        template.evaluate(writer, mv.model);
    }
}
```

最后我们来看看整个工程的结构：

```
web-mvc
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── itranswarp
        │           └── learnjava
        │               ├── Main.java
        │               ├── bean
        │               │   ├── SignInBean.java
        │               │   └── User.java
```

```
        ├── controller
        │   ├── IndexController.java
        │   └── UserController.java
        └── framework
            ├── DispatcherServlet.java
            ├── FileServlet.java
            ├── GetMapping.java
            ├── ModelAndView.java
            ├── PostMapping.java
            └── ViewEngine.java
    └── webapp
        ├── WEB-INF
        │   ├── templates
        │   │   ├── _base.html
        │   │   ├── hello.html
        │   │   ├── index.html
        │   │   ├── profile.html
        │   │   └── signin.html
        │   └── web.xml
        └── static
            ├── css
            │   └── bootstrap.css
            └── js
                ├── bootstrap.js
                └── jquery.js
```

其中，`framework`包是MVC的框架，完全可以单独编译后作为一个Maven依赖引入，`controller`包才是我们需要编写的业务逻辑。

我们还硬性规定模板必须放在`webapp/WEB-INF/templates`目录下，静态文件必须放在`webapp/static`目录下，因此，为了便于开发，我们还顺带实现一个`FileServlet`来处理静态文件：

```
@WebServlet(urlPatterns = { "/favicon.ico", "/static/*" })
public class FileServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 读取当前请求路径:
        ServletContext ctx = req.getServletContext();
        // RequestURI包含ContextPath,需要去掉:
        String urlPath = req.getRequestURI().substring(ctx.getContextPath().length());
        // 获取真实文件路径:
        String filepath = ctx.getRealPath(urlPath);
        if (filepath == null) {
            // 无法获取到路径:
            resp.sendError(HttpServletResponse.SC_NOT_FOUND);
            return;
        }
        Path path = Paths.get(filepath);
        if (!path.toFile().isFile()) {
            // 文件不存在:
            resp.sendError(HttpServletResponse.SC_NOT_FOUND);
            return;
        }
        // 根据文件名猜测Content-Type:
        String mime = Files.probeContentType(path);
        if (mime == null) {
            mime = "application/octet-stream";
        }
        resp.setContentType(mime);
        // 读取文件并写入Response:
        OutputStream output = resp.getOutputStream();
        try (InputStream input = new BufferedInputStream(new FileInputStream(filepath))) {
            input.transferTo(output);
        }
        output.flush();
    }
}
```

运行代码，在浏览器中输入URL`http://localhost:8080/hello?name=Bob`可以看到如下页面：

为了把方法参数的名称编译到class文件中，以便处理`@GetMapping`时使用，我们需要打开编译器的一个参数，在Eclipse中勾选`Preferences-Java-Compiler-Store information about method parameters (usable via reflection)`；在Idea中选择`Preferences-Build, Execution, Deployment-Compiler-Java Compiler-Additional command line parameters`，填入`-parameters`；在Maven的`pom.xml`添加一段配置如下：

```
<project ...>
    <modelVersion>4.0.0</modelVersion>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <compilerArgs>
                        <arg>-parameters</arg>
                    </compilerArgs>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

有些用过Spring MVC的童鞋会发现，本节实现的这个MVC框架，上层代码使用的公共类如`GetMapping`、`PostMapping`和`ModelAndView`都和Spring MVC非常类似。实际上，我们这个MVC框架主要参考就是Spring MVC，通过实现一个“简化版”MVC，可以掌握Java Web MVC开发的核心思想与原理，对将来直接使用Spring MVC是非常有帮助的。

练习

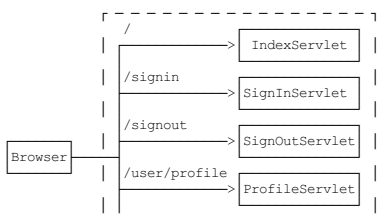
实现一个MVC框架

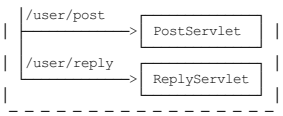
小结

一个MVC框架是基于Servlet基础抽象出更高级的接口，使得上层基于MVC框架的开发可以不涉及Servlet相关的`HttpServletRequest`等接口，处理多个请求更加灵活，并且可以使用任意模板引擎，不必使用JSP。

在一个比较复杂的Web应用程序中，通常都有很多URL映射，对应的，也会有多个Servlet来处理URL。

我们考察这样一个论坛应用程序：





各个Servlet设计功能如下：

- IndexServlet：浏览帖子；
- SignInServlet：登录；
- SignOutServlet：登出；
- ProfileServlet：修改用户资料；
- PostServlet：发帖；
- ReplyServlet：回复。

其中，ProfileServlet、PostServlet和ReplyServlet都需要用户登录后才能操作，否则，应当直接跳转到登录页面。

我们可以直接把判断登录的逻辑写到这3个Servlet中，但是，同样的逻辑重复3次没有必要，并且，如果后续继续加Servlet并且也需要验证登录时，还需要继续重复这个检查逻辑。

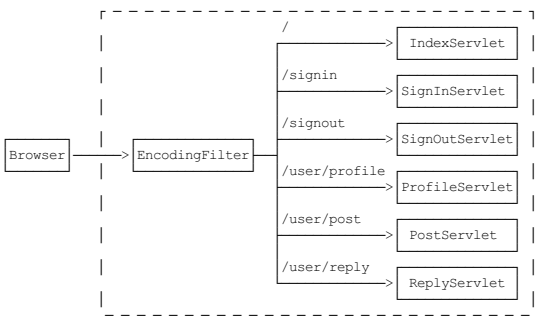
为了把一些公用逻辑从各个Servlet中抽离出来，JavaEE的Servlet规范还提供了一种Filter组件，即过滤器，它的作用是，在HTTP请求到达Servlet之前，可以被一个或多个Filter预处理，类似打印日志、登录检查等逻辑，完全可以放到Filter中。

例如，我们编写一个最简单的EncodingFilter，它强制把输入和输出的编码设置为UTF-8：

```
@WebFilter(urlPatterns = "/*")
public class EncodingFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        System.out.println("EncodingFilter:doFilter");
        request.setCharacterEncoding("UTF-8");
        response.setCharacterEncoding("UTF-8");
        chain.doFilter(request, response);
    }
}
```

编写Filter时，必须实现Filter接口，在doFilter()方法内部，要继续处理请求，必须调用chain.doFilter()。最后，用@WebFilter注解标注该Filter需要过滤的URL。这里的/*表示所有路径。

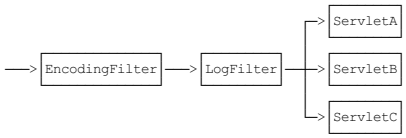
添加了Filter之后，整个请求的处理架构如下：



还可以继续添加其他Filter，例如LogFilter：

```
@WebFilter("/*")
public class LogFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        System.out.println("LogFilter: process " + ((HttpServletRequest) request).getRequestURI());
        chain.doFilter(request, response);
    }
}
```

多个Filter会组成一个链，每个请求都被链上的Filter依次处理：



有些细心的童鞋会问，有多个Filter的时候，Filter的顺序如何指定？多个Filter按不同顺序处理会造成处理结果不同吗？

答案是Filter的顺序确实对处理的结果有影响。但遗憾的是，Servlet规范并没有对@WebFilter注解标注的Filter规定顺序。如果一定要给每个Filter指定顺序，就必须在web.xml文件中对这些Filter再配置一遍。

注意到上述两个Filter的过滤路径都是/*，即它们会对所有请求进行过滤。也可以编写只对特定路径进行过滤的Filter，例如AuthFilter：

```
@WebFilter("/user/*")
public class AuthFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        System.out.println("AuthFilter: check authentication");
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        if (req.getSession().getAttribute("user") == null) {
            // 未登录，自动跳转到登录页:
            System.out.println("AuthFilter: not signin!");
            resp.sendRedirect("/signin");
        } else {
            // 已登录，继续处理:
            chain.doFilter(request, response);
        }
    }
}
```

注意到AuthFilter只过滤以/user/开头的路径，因此：

- 如果一个请求路径类似/user/profile，那么它会被上述3个Filter依次处理；
- 如果一个请求路径类似/test，那么它会被上述2个Filter依次处理（不会被AuthFilter处理）。

再注意观察AuthFilter，当用户没有登录时，在AuthFilter内部，直接调用resp.sendRedirect()发送重定向，且没有调用chain.doFilter()，因此，当用户没有登录时，请求到达AuthFilter后，不再继续处理，即后续的Filter和任何Servlet都没有机会处理该请求了。

可见，Filter可以有针对性地拦截或者放行HTTP请求。

如果一个Filter在当前请求中生效，但什么都没有做：

```
@WebFilter("/*")
public class MyFilter implements Filter {
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        // TODO
    }
}
```

那么，用户将看到一个空白页，因为请求没有继续处理，默认响应是200+空白输出。

如果Filter要使请求继续被处理，就一定要调用chain.doFilter()！

如果我们使用上一节介绍的MVC模式，即一个统一的DispatcherServlet入口，加上多个Controller，这种模式下Filter仍然是正常工作的。例如，一个处理/user/*的Filter实际上作用于那些处理/user/开头的Controller方法之前。

小结

Filter是一种对HTTP请求进行预处理的组件，它可以构成一个处理链，使得公共处理代码能集中到一起；

Filter适用于日志、登录检查、全局设置等；

设计合理的URL映射可以让Filter链更清晰。

Filter可以对请求进行预处理，因此，我们可以把很多公共预处理逻辑放到Filter中完成。

考察这样一种需求：我们在Web应用中经常需要处理用户上传文件，例如，一个UploadServlet可以简单地编写如下：

```
@WebServlet(urlPatterns = "/upload/file")
public class UploadServlet extends HttpServlet {
    protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        // 读取Request Body:
        InputStream input = req.getInputStream();
        ByteArrayOutputStream output = new ByteArrayOutputStream();
        byte[] buffer = new byte[1024];
        for (;;) {
            int len = input.read(buffer);
            if (len == -1) {
                break;
            }
            output.write(buffer, 0, len);
        }
        // TODO: 写入文件:
        // 显示上传结果:
        String uploadedText = output.toString(StandardCharsets.UTF_8);
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Uploaded:</h1>");
        pw.write("<pre><code>");
        pw.write(uploadedText);
        pw.write("</code></pre>");
        pw.flush();
    }
}
```

但是要保证文件上传的完整性怎么办？在哈希算法一节中，我们知道，如果在上传文件的同时，把文件的哈希也传过来，服务器端做一个验证，就可以确保用户上传的文件一定是完整的。

这个验证逻辑非常适合写在ValidateUploadFilter中，因为它可以复用。

我们先写一个简单的版本，快速实现ValidateUploadFilter的逻辑：

```
@WebFilter("/upload/*")
public class ValidateUploadFilter implements Filter {

    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        // 获取客户端传入的签名方法和签名:
        String digest = req.getHeader("Signature-Method");
        String signature = req.getHeader("Signature");
        if (digest == null || digest.isEmpty() || signature == null || signature.isEmpty()) {
            sendErrorPage(resp, "Missing signature.");
            return;
        }
        // 读取Request的Body并验证签名:
        MessageDigest md = getMessageDigest(digest);
        InputStream input = new DigestInputStream(request.getInputStream(), md);
        byte[] buffer = new byte[1024];
        for (;;) {
            int len = input.read(buffer);
            if (len == -1) {
                break;
            }
        }
        String actual = toHexString(md.digest());
        if (!signature.equals(actual)) {
            sendErrorPage(resp, "Invalid signature.");
            return;
        }
        // 验证成功后继续处理:
        chain.doFilter(request, response);
    }

    // 将byte[]转换为hex string:
    private String toHexString(byte[] digest) {
        StringBuilder sb = new StringBuilder();
        for (byte b : digest) {
            sb.append(String.format("%02x", b));
        }
        return sb.toString();
    }

    // 根据名称创建MessageDigest:
    private MessageDigest getMessageDigest(String name) throws ServletException {
        try {
            return MessageDigest.getInstance(name);
        } catch (NoSuchAlgorithmException e) {
            throw new ServletException(e);
        }
    }

    // 发送一个错误响应:
    private void sendErrorPage(HttpServletResponse resp, String errorMessage) throws IOException {
        resp.setStatus(HttpServletResponse.SC_BAD_REQUEST);
```

```
        PrintWriter pw = resp.getWriter();
        pw.write("<html><body><h1>");
        pw.write(errorMessage);
        pw.write("</h1></body></html>");
        pw.flush();
    }
}
```

这个ValidateUploadFilter的逻辑似乎没有问题，我们可以用curl命令测试：

```
$ curl http://localhost:8080/upload/file -v -d 'test-data' \
  -H 'Signature-Method: SHA-1' \
  -H 'Signature: 7115e9890f5b5cc6914bdfa3b7c011db1cdafedb' \
  -H 'Content-Type: application/octet-stream'
*   Trying ::1...
* TCP_NODELAY set
* Connected to localhost (::1) port 8080 (#0)
> POST /upload/file HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
> Signature-Method: SHA-1
> Signature: 7115e9890f5b5cc6914bdfa3b7c011db1cdafedb
> Content-Type: application/octet-stream
> Content-Length: 9
>
* upload completely sent off: 9 out of 9 bytes
< HTTP/1.1 200
< Transfer-Encoding: chunked
< Date: Thu, 30 Jan 2020 13:56:39 GMT
<
* Connection #0 to host localhost left intact
<h1>Uploaded:</h1><pre><code></code></pre>
* Closing connection 0
```

ValidateUploadFilter对签名进行验证的逻辑是没有问题的，但是，细心的童鞋注意到，UploadServlet并未读取到任何数据！

这里的原因是对HttpServletRequest进行读取时，只能读取一次。如果Filter调用getInputStream()读取了一次数据，后续Servlet处理时，再次读取，将无法读到任何数据。怎么办？

这个时候，我们需要一个“伪造”的HttpServletRequest，具体做法是使用代理模式，对getInputStream()和getReader()返回一个新的流：

```
class ReReadableHttpServletRequest extends HttpServletRequestWrapper {
    private byte[] body;
    private boolean open = false;

    public ReReadableHttpServletRequest(HttpServletRequest request, byte[] body) {
        super(request);
        this.body = body;
    }

    // 返回InputStream:
    public ServletInputStream getInputStream() throws IOException {
        if (open) {
            throw new IllegalStateException("Cannot re-open input stream!");
        }
        open = true;
        return new ServletInputStream() {
            private int offset = 0;

            public boolean isFinished() {
                return offset >= body.length;
            }

            public boolean isReady() {
                return true;
            }

            public void setReadListener(ReadListener listener) {
            }

            public int read() throws IOException {
                if (offset >= body.length) {
                    return -1;
                }
                int n = body[offset] & 0xff;
                offset++;
                return n;
            }
        };
    }

    // 返回Reader:
    public BufferedReader getReader() throws IOException {
        if (open) {
            throw new IllegalStateException("Cannot re-open reader!");
        }
        open = true;
        return new BufferedReader(new InputStreamReader(new ByteArrayInputStream(body), "UTF-8"));
    }
}
```

注意观察ReReadableHttpServletRequest的构造方法，它保存了ValidateUploadFilter读取的byte[]内容，并在调用getInputStream()时通过byte[]构造了一个新的ServletInputStream。

然后，我们在ValidateUploadFilter中，把doFilter()调用时传给下一个处理者的HttpServletRequest替换为我们自己“伪造”的ReReadableHttpServletRequest：

```
public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
        throws IOException, ServletException {
    ...
    chain.doFilter(new ReReadableHttpServletRequest(req, output.toByteArray()), response);
}
```

再注意到我们编写ReReadableHttpServletRequest时，是从HttpServletRequestWrapper继承，而不是直接实现HttpServletRequest接口。这是因为，Servlet的每个新版本都会对接口增加一些新方法，从HttpServletRequestWrapper继承可以确保新方法被正确地覆写了，因为HttpServletRequestWrapper是由Servlet的jar包提供的，目的就是为了让我们方便地实现对HttpServletRequest接口的代理。

我们总结一下对HttpServletRequest接口进行代理的步骤：

1. 从HttpServletRequestWrapper继承一个XxxHttpServletRequest，需要传入原始的HttpServletRequest实例；
2. 覆写某些方法，使得新的XxxHttpServletRequest实例看上去“改变”了原始的HttpServletRequest实例；
3. 在doFilter()中传入新的XxxHttpServletRequest实例。

虽然整个Filter的代码比较复杂，但它的好处在于：这个Filter在整个处理链中实现了灵活的“可插拔”特性，即是否启用对Web应用程序的其他组件（Filter、Servlet）完全没有影响。

练习

使用Filter修改请求

小结

借助HttpServletRequestWrapper，我们可以在Filter中实现对原始HttpServletRequest的修改。

既然我们能通过Filter修改HttpServletRequest，自然也能修改HttpServletResponse，因为这两者都是接口。

我们来看一下在什么情况下我们需要修改HttpServletResponse。

假设我们编写了一个Servlet，但由于业务逻辑比较复杂，处理该请求需要耗费很长的时间：

```
@WebServlet(urlPatterns = "/slow/hello")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("text/html");
        // 模拟耗时1秒:
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Hello, world!</h1>");
        pw.flush();
    }
}
```

好消息是每次返回的响应内容是固定的，因此，如果我们能使用缓存将结果缓存起来，就可以大大提高Web应用程序的运行效率。

缓存逻辑最好不要在Servlet内部实现，因为我们希望能复用缓存逻辑，所以，编写一个CacheFilter最合适：

```
@WebFilter("/slow/*")
public class CacheFilter implements Filter {
    // Path到byte[]的缓存:
    private Map<String, byte[]> cache = new ConcurrentHashMap<>();

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        // 获取Path:
        String url = req.getRequestURI();
        // 获取缓存内容:
        byte[] data = this.cache.get(url);
        resp.setHeader("X-Cache-Hit", data == null ? "No" : "Yes");
        if (data == null) {
            // 缓存未找到,构造一个伪造的Response:
            CachedHttpServletResponse wrapper = new CachedHttpServletResponse(resp);
            // 让下游组件写入数据到伪造的Response:
            chain.doFilter(request, wrapper);
            // 从伪造的Response中读取写入的内容并放入缓存:
            data = wrapper.getContent();
            cache.put(url, data);
        }
        // 写入到原始的Response:
        ServletOutputStream output = resp.getOutputStream();
        output.write(data);
        output.flush();
    }
}
```

实现缓存的关键在于，调用doFilter()时，我们不能传入原始的HttpServletResponse，因为这样就会写入Socket，我们也就无法获取下游组件写入的内容。如果我们传入的是“伪造”的HttpServletResponse，让下游组件写入到我们预设的ByteArrayOutputStream，我们就“截获”了下游组件写入的内容，于是，就可以把内容缓存起来，再通过原始的HttpServletResponse实例写入到网络。

这个CachedHttpServletResponse实现如下：

```
class CachedHttpServletResponse extends HttpServletResponseWrapper {
    private boolean open = false;
    private ByteArrayOutputStream output = new ByteArrayOutputStream();

    public CachedHttpServletResponse(HttpServletResponse response) {
        super(response);
    }

    // 获取Writer:
    public PrintWriter getWriter() throws IOException {
        if (open) {
            throw new IllegalStateException("Cannot re-open writer!");
        }
        open = true;
        return new PrintWriter(output, false, StandardCharsets.UTF_8);
    }

    // 获取OutputStream:
    public ServletOutputStream getOutputStream() throws IOException {
        if (open) {
            throw new IllegalStateException("Cannot re-open output stream!");
        }
        open = true;
        return new ServletOutputStream() {
            public boolean isReady() {
                return true;
            }

            public void setWriteListener(WriteListener listener) {
            }

            // 实际写入ByteArrayOutputStream:
            public void write(int b) throws IOException {
                output.write(b);
            }
        };
    }

    // 返回写入的byte[]:
    public byte[] getContent() {
        return output.toByteArray();
    }
}
```

可见，如果我们想要修改响应，就可以通过HttpServletResponseWrapper构造一个“伪造”的HttpServletResponse，这样就能拦截到写入的数据。

修改响应时，最后不要忘记把数据写入原始的HttpServletResponse实例。

这个CacheFilter同样是一个“可插拔”组件，它是否启用不影响Web应用程序的其他组件（Filter、Servlet）。

练习

通过Filter修改响应

小结

借助HttpServletResponseWrapper，我们可以在Filter中实现对原始HttpServletResponse的修改。

除了Servlet和Filter外，JavaEE的Servlet规范还提供了第三种组件：Listener。

Listener顾名思义就是监听器，有好几种Listener，其中最常用的是ServletContextListener，我们编写一个实现了ServletContextListener接口的类如下：

```
@WebListener
public class AppListener implements ServletContextListener {
    // 在此初始化WebApp,例如打开数据库连接池等:
    public void contextInitialized(ServletContextEvent sce) {
        System.out.println("WebApp initialized.");
    }

    // 在此清理WebApp,例如关闭数据库连接池等:
    public void contextDestroyed(ServletContextEvent sce) {
        System.out.println("WebApp destroyed.");
    }
}
```

任何标注为@WebListener，且实现了特定接口的类会被Web服务器自动初始化。上述AppListener实现了ServletContextListener接口，它会在整个Web应用程序初始化完成后，以及Web应用程序关闭后获得回调通知。我们可以把初始化数据库连接池等工作放到contextInitialized()回调方法中，把清理资源的工作放到contextDestroyed()回调方法中，因为Web服务器保证在contextInitialized()执行后，才会接受用户的HTTP请求。

很多第三方Web框架都会通过一个ServletContextListener接口初始化自己。

除了ServletContextListener外，还有几种Listener：

- HttpSessionListener：监听HttpSession的创建和销毁事件；
- ServletRequestListener：监听ServletRequest请求的创建和销毁事件；
- ServletRequestAttributeListener：监听ServletRequest请求的属性变化事件（即调用ServletRequest.setAttribute()方法）；
- ServletContextAttributeListener：监听ServletContext的属性变化事件（即调用ServletContext.setAttribute()方法）；

ServletContext

一个Web服务器可以运行一个或多个WebApp，对于每个WebApp，Web服务器都会为其创建一个全局唯一的ServletContext实例，我们在AppListener里面编写的两个回调方法实际上对应的就是ServletContext实例的创建和销毁：

```
public void contextInitialized(ServletContextEvent sce) {
    System.out.println("WebApp initialized: ServletContext = " + sce.getServletContext());
}
```

ServletRequest、HttpSession等很多对象也提供getServletContext()方法获取到同一个ServletContext实例。ServletContext实例最大的作用就是设置和共享全局信息。

此外，ServletContext还提供了动态添加Servlet、Filter、Listener等功能，它允许应用程序在运行期间动态添加一个组件，虽然这个功能不是很常用。

练习

使用Listener监听WebApp

小结

通过Listener我们可以监听Web应用程序的生命周期，获取HttpSession等创建和销毁的事件；

ServletContext是一个WebApp运行期的全局唯一实例，可用于设置和共享配置信息。

对一个Web应用程序来说，除了Servlet、Filter这些逻辑组件，还需要JSP这样的视图文件，外加一堆静态资源文件，如CSS、JS等。

合理组织文件结构非常重要。我们以一个具体的Web应用程序为例：

```
webapp
├─ pom.xml
└─ src
   └─ main
      ├─ java
      │  └─ com
      │     └─ itranswarp
      │        └─ learnjava
      │           ├─ Main.java
      │           ├─ filter
      │           │  └─ EncodingFilter.java
      │           └─ servlet
      │              ├─ FileServlet.java
      │              └─ HelloServlet.java
      ├─ resources
      └─ webapp
         ├─ WEB-INF
         │  └─ web.xml
         ├─ favicon.ico
         └─ static
            └─ bootstrap.css
```

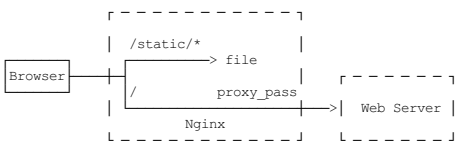
我们把所有的静态资源文件放入/static/目录，在开发阶段，有些Web服务器会自动为我们加一个专门负责处理静态文件的Servlet，但如果IndexServlet映射路径为/，会屏蔽掉处理静态文件的Servlet映射。因此，我们需要自己编写一个处理静态文件的FileServlet：

```
@WebServlet(urlPatterns = "/static/*")
public class FileServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        ServletContext ctx = req.getServletContext();
        // RequestURI包含ContextPath,需要去掉:
        String urlPath = req.getRequestURI().substring(ctx.getContextPath().length());
        // 获取真实文件路径:
        String filepath = ctx.getRealPath(urlPath);
        if (filepath == null) {
            // 无法获取到路径:
            resp.sendError(HttpServletResponse.SC_NOT_FOUND);
            return;
        }
        Path path = Paths.get(filepath);
        if (!path.toFile().isFile()) {
            // 文件不存在:
            resp.sendError(HttpServletResponse.SC_NOT_FOUND);
            return;
        }
        // 根据文件名猜测Content-Type:
        String mime = Files.probeContentType(path);
        if (mime == null) {
            mime = "application/octet-stream";
        }
        resp.setContentType(mime);
        // 读取文件并写入Response:
        OutputStream output = resp.getOutputStream();
        try (InputStream input = new BufferedInputStream(new FileInputStream(filepath))) {
            input.transferTo(output);
        }
```

```
        output.flush();
    }
}
```

这样一来，在开发阶段，我们就可以方便地高效开发。

类似Tomcat这样的Web服务器，运行的Web应用程序通常都是业务系统，因此，这类服务器也被称为应用服务器。应用服务器并不擅长处理静态文件，也不适合直接暴露给用户。通常，我们在生产环境部署时，总是使用类似Nginx这样的服务器充当反向代理和静态服务器，只有动态请求才会放行给应用服务器，所以，部署架构如下：



实现上述功能的Nginx配置文件如下：

```
server {
    listen 80;

    server_name www.local.liaoxuefeng.com;

    # 静态文件根目录:
    root /path/to/src/main/webapp;

    access_log /var/log/nginx/webapp_access_log;
    error_log  /var/log/nginx/webapp_error_log;

    # 处理静态文件请求:
    location /static {
    }

    # 处理静态文件请求:
    location /favicon.ico {
    }

    # 不允许请求/WEB-INF:
    location /WEB-INF {
        return 404;
    }

    # 其他请求转发给Tomcat:
    location / {
        proxy_pass       http://127.0.0.1:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
}
```

使用Nginx配合Tomcat服务器，可以充分发挥Nginx作为网关的优势，既可以高效处理静态文件，也可以把https、防火墙、限速、反爬虫等功能放到Nginx中，使得我们自己的WebApp能专注于业务逻辑。

练习

使用Nginx+Tomcat部署

小结

部署Web应用程序时，要设计合理的目录结构，同时考虑开发模式需要便捷性，生产模式需要高性能。

什么是Spring?

Spring是一个支持快速开发Java EE应用程序的框架。它提供了一系列底层容器和基础设施，并可以和大量常用的开源框架无缝集成，可以说是开发Java EE应用程序的必备。

Spring最早是由Rod Johnson这哥们在他的《Expert One-on-One J2EE Development without EJB》一书中提出的用来取代EJB的轻量级框架。随后这哥们又开始专心开发这个基础框架，并起名为Spring Framework。

随着Spring越来越受欢迎，在Spring Framework基础上，又诞生了Spring Boot、Spring Cloud、Spring Data、Spring Security等一系列基于Spring Framework的项目。本章我们只介绍Spring Framework，即最核心的Spring框架。后续章节我们还会涉及Spring Boot、Spring Cloud等其他框架。

Spring Framework

Spring Framework主要包括几个模块：

- 支持IoC和AOP的容器；
- 支持JDBC和ORM的数据访问模块；
- 支持声明式事务的模块；
- 支持基于Servlet的MVC开发；
- 支持基于Reactive的Web开发；
- 以及集成JMS、JavaMail、JMX、缓存等其他模块。

我们会依次介绍Spring Framework的主要功能。

在学习Spring框架时，我们遇到的第一个也是最核心的概念就是容器。

什么是容器？容器是一种为某种特定组件的运行提供必要支持的一个软件环境。例如，Tomcat就是一个Servlet容器，它可以为Servlet的运行提供运行环境。类似Docker这样的软件也是一个容器，它提供了必要的Linux环境以便运行一个特定的Linux进程。

通常来说，使用容器运行组件，除了提供一个组件运行环境之外，容器还提供了许多底层服务。例如，Servlet容器底层实现了TCP连接，解析HTTP协议等非常复杂的服务，如果没有容器来提供这些服务，我们就无法编写像Servlet这样代码简单，功能强大的组件。早期的JavaEE服务器提供的EJB容器最重要的功能就是通过声明式事务服务，使得EJB组件的开发人员不必自己编写冗长的事务处理代码，所以极大地简化了事务处理。

Spring的核心就是提供了一个IoC容器，它可以管理所有轻量级的JavaBean组件，提供的底层服务包括组件的生命周期管理、配置和组装服务、AOP支持，以及建立在AOP基础上的声明式事务服务等。

本章我们讨论的IoC容器，主要介绍Spring容器如何对组件进行生命周期管理和配置组装服务。

Spring提供的容器又称为IoC容器，什么是IoC？

IoC全称Inversion of Control，直译为控制反转。那么何谓IoC？在理解IoC之前，我们先看看通常的Java组件是如何协作的。

我们假定一个在线书店，通过`BookService`获取书籍：

```
public class BookService {
    private HikariConfig config = new HikariConfig();
    private DataSource dataSource = new HikariDataSource(config);
```

```
    public Book getBook(long bookId) {
        try (Connection conn = dataSource.getConnection()) {
            ...
            return book;
        }
    }
}
```

为了从数据库查询书籍，BookService持有一个DataSource。为了实例化一个HikariDataSource，又不得不实例化一个HikariConfig。

现在，我们继续编写UserService获取用户：

```
public class UserService {
    private HikariConfig config = new HikariConfig();
    private DataSource dataSource = new HikariDataSource(config);

    public User getUser(long userId) {
        try (Connection conn = dataSource.getConnection()) {
            ...
            return user;
        }
    }
}
```

因为UserService也需要访问数据库，因此，我们不得不也实例化一个HikariDataSource。

在处理用户购买的CartServlet中，我们需要实例化UserService和BookService：

```
public class CartServlet extends HttpServlet {
    private BookService bookService = new BookService();
    private UserService userService = new UserService();

    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        long currentUserId = getFromCookie(req);
        User currentUser = userService.getUser(currentUserId);
        Book book = bookService.getBook(req.getParameter("bookId"));
        cartService.addToCart(currentUser, book);
        ...
    }
}
```

类似的，在购买历史HistoryServlet中，也需要实例化UserService和BookService：

```
public class HistoryServlet extends HttpServlet {
    private BookService bookService = new BookService();
    private UserService userService = new UserService();
}
```

上述每个组件都采用了一种简单的通过new创建实例并持有的方式。仔细观察，会发现以下缺点：

1. 实例化一个组件其实很难，例如，BookService和UserService要创建HikariDataSource，实际上需要读取配置，才能先实例化HikariConfig，再实例化HikariDataSource。
2. 没有必要让BookService和UserService分别创建DataSource实例，完全可以共享同一个DataSource，但谁负责创建DataSource，谁负责获取其他组件已经创建的DataSource，不好处理。类似的，CartServlet和HistoryServlet也应当共享BookService实例和UserService实例，但也不好处理。
3. 很多组件需要销毁以便释放资源，例如DataSource，但如果该组件被多个组件共享，如何确保它的使用方都已经全部被销毁？
4. 随着更多的组件被引入，例如，书籍评论，需要共享的组件写起来会更困难，这些组件的依赖关系会越来越复杂。
5. 测试某个组件，例如BookService，是复杂的，因为必须要在真实的数据库环境下执行。

从上面的例子可以看出，如果一个系统有大量的组件，其生命周期和相互之间的依赖关系如果由组件自身来维护，不但大大增加了系统的复杂度，而且会导致组件之间极为紧密的耦合，继而给测试和维护带来了极大的困难。

因此，核心问题是：

1. 谁负责创建组件？
2. 谁负责根据依赖关系组装组件？
3. 销毁时，如何按依赖顺序正确销毁？

解决这一问题的核心方案就是IoC。

传统的应用程序中，控制权在程序本身，程序的控制流程完全由开发者控制，例如：

CartServlet创建了BookService，在创建BookService的过程中，又创建了DataSource组件。这种模式的缺点是，一个组件如果要使用另一个组件，必须先知道如何正确地创建它。

在IoC模式下，控制权发生了反转，即从应用程序转移到了IoC容器，所有组件不再由应用程序自己创建和配置，而是由IoC容器负责，这样，应用程序只需要直接使用已经创建好并且配置好的组件。为了能让组件在IoC容器中被“装配”出来，需要某种“注入”机制，例如，BookService自己并不会创建DataSource，而是等待外部通过setDataSource()方法来注入一个DataSource：

```
public class BookService {
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

不直接new一个DataSource，而是注入一个DataSource，这个小小的改动虽然简单，却带来了一系列好处：

1. BookService不再关心如何创建DataSource，因此，不必编写读取数据库配置之类的代码；
2. DataSource实例被注入到BookService，同样也可以注入到UserService，因此，共享一个组件非常简单；
3. 测试BookService更容易，因为注入的是DataSource，可以使用内存数据库，而不是真实的MySQL配置。

因此，IoC又称为依赖注入（DI：Dependency Injection），它解决了一个最主要的问题：将组件的创建+配置与组件的使用相分离，并且，由IoC容器负责管理组件的生命周期。

因为IoC容器要负责实例化所有的组件，因此，有必要告诉容器如何创建组件，以及各组件的依赖关系。一种最简单的配置是通过XML文件来实现，例如：

```
<beans>
    <bean id="dataSource" class="HikariDataSource" />
    <bean id="bookService" class="BookService">
        <property name="dataSource" ref="dataSource" />
    </bean>
    <bean id="userService" class="UserService">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

上述XML配置文件指示IoC容器创建3个JavaBean组件，并把id为dataSource的组件通过属性dataSource（即调用setDataSource()方法）注入到另外两个组件中。

在Spring的IoC容器中，我们把所有组件统称为JavaBean，即配置一个组件就是配置一个Bean。

依赖注入方式

我们从上面的代码可以看到，依赖注入可以通过`set()`方法实现。但依赖注入也可以通过构造方法实现。

很多Java类都具有带参数的构造方法，如果我们把`BookService`改造为通过构造方法注入，那么实现代码如下：

```
public class BookService {
    private DataSource dataSource;

    public BookService(DataSource dataSource) {
        this.dataSource = dataSource;
    }
}
```

Spring的IoC容器同时支持属性注入和构造方法注入，并允许混合使用。

无侵入容器

在设计上，Spring的IoC容器是一个高度可扩展的无侵入容器。所谓无侵入，是指应用程序的组件无需实现Spring的特定接口，或者说，组件根本不知道自己在Spring的容器中运行。这种无侵入的设计有以下好处：

1. 应用程序组件既可以在Spring的IoC容器中运行，也可以自己编写代码自行组装配置；
2. 测试的时候并不依赖Spring容器，可单独进行测试，大大提高了开发效率。

我们前面讨论了为什么要使用Spring的IoC容器，因为让容器来为我们创建并装配Bean能获得很大的好处，那么到底如何使用IoC容器？装配好的Bean又如何使用？

我们来看一个具体的用户注册登录的例子。整个工程的结构如下：

```
spring-ioc-appcontext
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── itranswarp
        │           └── learnjava
        │               ├── Main.java
        │               └── service
        │                   ├── MailService.java
        │                   ├── User.java
        │                   └── UserService.java
        └── resources
            └── application.xml
```

首先，我们用Maven创建工程并引入`spring-context`依赖：

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>spring-ioc-appcontext</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>

        <spring.version>5.2.3.RELEASE</spring.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-context</artifactId>
            <version>${spring.version}</version>
        </dependency>
    </dependencies>
</project>
```

我们先编写一个`MailService`，用于在用户登录和注册成功后发送邮件通知：

```
public class MailService {
    private ZoneId zoneId = ZoneId.systemDefault();

    public void setZoneId(ZoneId zoneId) {
        this.zoneId = zoneId;
    }

    public String getTime() {
        return ZonedDateTime.now(this.zoneId).format(DateTimeFormatter.ISO_ZONED_DATE_TIME);
    }

    public void sendLoginMail(User user) {
        System.err.println(String.format("Hi, %s! You are logged in at %s", user.getName(), getTime()));
    }

    public void sendRegistrationMail(User user) {
        System.err.println(String.format("Welcome, %s!", user.getName()));

    }
}
```

再编写一个`UserService`，实现用户注册和登录：

```
public class UserService {
    private MailService mailService;

    public void setMailService(MailService mailService) {
        this.mailService = mailService;
    }

    private List<User> users = new ArrayList<>(List.of( // users:
            new User(1, "bob@example.com", "password", "Bob"), // bob
            new User(2, "alice@example.com", "password", "Alice"), // alice
            new User(3, "tom@example.com", "password", "Tom"))); // tom

    public User login(String email, String password) {
        for (User user : users) {
            if (user.getEmail().equalsIgnoreCase(email) && user.getPassword().equals(password)) {
                mailService.sendLoginMail(user);
                return user;
            }
        }
        throw new RuntimeException("login failed.");
```

```
    }

    public User getUser(long id) {
        return this.users.stream().filter(user -> user.getId() == id).findFirst().orElseThrow();
    }

    public User register(String email, String password, String name) {
        users.forEach((user) -> {
            if (user.getEmail().equalsIgnoreCase(email)) {
                throw new RuntimeException("email exist.");
            }
        });
        User user = new User(users.stream().mapToLong(u -> u.getId()).max().getAsLong() + 1, email, password, name);
        users.add(user);
        mailService.sendRegistrationMail(user);
        return user;
    }
}
```

注意到`UserService`通过`setMailService()`注入了一个`MailService`。

然后，我们需要编写一个特定的`application.xml`配置文件，告诉Spring的IoC容器应该如何创建并组装Bean：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        https://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="userService" class="com.itranswarp.learnjava.service.UserService">
        <property name="mailService" ref="mailService" />
    </bean>

    <bean id="mailService" class="com.itranswarp.learnjava.service.MailService" />
</beans>
```

注意观察上述配置文件，其中与XML Schema相关的部分格式是固定的，我们只关注两个`<bean ...>`的配置：

- 每个`<bean ...>`都有一个`id`标识，相当于Bean的唯一ID；
- 在`userService`Bean中，通过`<property name="..." ref="..." />`注入了另一个Bean；
- Bean的顺序不重要，Spring根据依赖关系会自动正确初始化。

把上述XML配置文件用Java代码写出来，就像这样：

```
UserService userService = new UserService();
MailService mailService = new MailService();
userService.setMailService(mailService);
```

只不过Spring容器是通过读取XML文件后使用反射完成的。

如果注入的不是Bean，而是`boolean`、`int`、`String`这样的数据类型，则通过`value`注入，例如，创建一个`HikariDataSource`：

```
<bean id="dataSource" class="com.zaxxer.hikari.HikariDataSource">
    <property name="jdbcUrl" value="jdbc:mysql://localhost:3306/test" />
    <property name="username" value="root" />
    <property name="password" value="password" />
    <property name="maximumPoolSize" value="10" />
    <property name="autoCommit" value="true" />
</bean>
```

最后一步，我们需要创建一个Spring的IoC容器实例，然后加载配置文件，让Spring容器为我们创建并装配好配置文件中指定的所有Bean，这只需要一行代码：

```
ApplicationContext context = new ClassPathXmlApplicationContext("application.xml");
```

接下来，我们就可以从Spring容器中“取出”装配好的Bean然后使用它：

```
// 获取Bean:
UserService userService = context.getBean(UserService.class);
// 正常调用:
User user = userService.login("bob@example.com", "password");
```

完整的`main()`方法如下：

```
public class Main {
    public static void main(String[] args) {
        ApplicationContext context = new ClassPathXmlApplicationContext("application.xml");
        UserService userService = context.getBean(UserService.class);
        User user = userService.login("bob@example.com", "password");
        System.out.println(user.getName());
    }
}
```

ApplicationContext

我们从创建Spring容器的代码：

```
ApplicationContext context = new ClassPathXmlApplicationContext("application.xml");
```

可以看到，Spring容器就是`ApplicationContext`，它是一个接口，有很多实现类，这里我们选择`ClassPathXmlApplicationContext`，表示它会自动从classspath中查找指定的XML配置文件。

获得了`ApplicationContext`的实例，就获得了IoC容器的引用。从`ApplicationContext`中我们可以根据Bean的ID获取Bean，但更多的时候我们根据Bean的类型获取Bean的引用：

```
UserService userService = context.getBean(UserService.class);
```

Spring还提供另一种IoC容器叫`BeanFactory`，使用方式和`ApplicationContext`类似：

```
BeanFactory factory = new XmlBeanFactory(new ClassPathResource("application.xml"));
MailService mailService = factory.getBean(MailService.class);
```

`BeanFactory`和`ApplicationContext`的区别在于，`BeanFactory`的实现是按需创建，即第一次获取Bean时才创建这个Bean，而`ApplicationContext`会一次性创建所有的Bean。实际上，`ApplicationContext`接口是从`BeanFactory`接口继承而来的，并且，`ApplicationContext`提供了一些额外的功能，包括国际化支持、事件和通知机制等。通常情况下，我们总是使用`ApplicationContext`，很少会考虑使用`BeanFactory`。

练习

在上述示例的基础上，继续给`UserService`注入`DataSource`，并把注册和登录功能通过数据库实现。

使用ApplicationContext

小结

Spring的IoC容器接口是`ApplicationContext`，并提供了多种实现类；

通过XML配置文件创建IoC容器时，使用`ClassPathXmlApplicationContext`；

持有IoC容器后，通过getBean()方法获取Bean的引用。

使用Spring的IoC容器，实际上就是通过类似XML这样的配置文件，把我们自己的Bean的依赖关系描述出来，然后让容器来创建并装配Bean。一旦容器初始化完毕，我们就直接从容器中获取Bean使用它们。

使用XML配置的优点是所有的Bean都能一目了然地列出来，并通过配置注入能直观地看到每个Bean的依赖。它的缺点是写起来非常繁琐，每增加一个组件，就必须把新的Bean配置到XML中。

有没有其他更简单的配置方式呢？

有！我们可以使用Annotation配置，可以完全不需要XML，让Spring自动扫描Bean并组装它们。

我们把上一节的示例改造一下，先删除XML配置文件，然后，给UserService和MailService添加几个注解。

首先，我们给MailService添加一个@Component注解：

```
@Component
public class MailService {
    ...
}
```

这个@Component注解就相当于定义了一个Bean，它有一个可选的名称，默认是mailService，即小写开头的类名。

然后，我们给UserService添加一个@Component注解和一个@Autowired注解：

```
@Component
public class UserService {
    @Autowired
    MailService mailService;

    ...
}
```

使用@Autowired就相当于把指定类型的Bean注入到指定的字段中。和XML配置相比，@Autowired大幅简化了注入，因为它不但可以写在set()方法上，还可以直接写在字段上，甚至可以写在构造方法中：

```
@Component
public class UserService {
    MailService mailService;

    public UserService(@Autowired MailService mailService) {
        this.mailService = mailService;
    }
    ...
}
```

我们一般把@Autowired写在字段上，通常使用package权限的字段，便于测试。

最后，编写一个AppConfig类启动容器：

```
@Configuration
@ComponentScan
public class AppConfig {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        UserService userService = context.getBean(UserService.class);
        User user = userService.login("bob@example.com", "password");
        System.out.println(user.getName());
    }
}
```

除了main()方法外，AppConfig标注了@Configuration，表示它是一个配置类，因为我们创建ApplicationContext时：

```
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
```

使用的实现类是AnnotationConfigApplicationContext，必须传入一个标注了@Configuration的类名。

此外，AppConfig还标注了@ComponentScan，它告诉容器，自动搜索当前类所在的包以及子包，把所有标注为@Component的Bean自动创建出来，并根据@Autowired进行装配。

整个工程结构如下：

```
spring-ioc-annoconfig
├── pom.xml
└── src
    └── main
        └── java
            └── com
                └── itranswarp
                    └── learnjava
                        ├── AppConfig.java
                        └── service
                            ├── MailService.java
                            ├── User.java
                            └── UserService.java
```

使用Annotation配合自动扫描能大幅简化Spring的配置，我们只需要保证：

- 每个Bean被标注为@Component并正确使用@Autowired注入；
- 配置类被标注为@Configuration和@ComponentScan；
- 所有Bean均在指定包以及子包内。

使用@ComponentScan非常方便，但是，我们也要特别注意包的层次结构。通常来说，启动配置AppConfig位于自定义的顶层包（例如com.itranswarp.learnjava），其他Bean按类别放入子包。

思考

如果我们想给UserService注入HikariDataSource，但是这个类位于com.zaxxer.hikari包中，并且HikariDataSource也不可能有@Component注解，如何告诉IoC容器创建并配置HikariDataSource？或者换个说法，如何创建并配置一个第三方Bean？

练习

使用Annotation配置IoC容器

小结

使用Annotation可以大幅简化配置，每个Bean通过@Component和@Autowired注入；

必须合理设计包的层次结构，才能发挥@ComponentScan的威力。

Scope

对于Spring容器来说，当我们把一个Bean标记为@Component后，它就会自动为我们创建一个单例（Singleton），即容器初始化时创建Bean，容器关闭前销毁Bean。在容器运行期间，我们调用getBean(Class)获取到的Bean总是同一个实例。

还有一种Bean，我们每次调用getBean(Class)，容器都返回一个新的实例，这种Bean称为Prototype（原型），它的生命周期显然和Singleton不同。声明一个Prototype的Bean时，需要添加一个额外的@Scope注解：

```
@Component
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE) // @Scope("prototype")
public class MailSession {
    ...
}
```

注入List

有些时候，我们会有一系列接口相同，不同实现类的Bean。例如，注册用户时，我们要对email、password和name这3个变量进行验证。为了便于扩展，我们先定义验证接口：

```
public interface Validator {
    void validate(String email, String password, String name);
}
```

然后，分别使用3个Validator对用户参数进行验证：

```
@Component
public class EmailValidator implements Validator {
    public void validate(String email, String password, String name) {
        if (!email.matches("^[a-z0-9]+\\@[a-z0-9]+\\.[a-z]{2,10}$")) {
            throw new IllegalArgumentException("invalid email: " + email);
        }
    }
}

@Component
public class PasswordValidator implements Validator {
    public void validate(String email, String password, String name) {
        if (!password.matches("^.{6,20}$")) {
            throw new IllegalArgumentException("invalid password");
        }
    }
}

@Component
public class NameValidator implements Validator {
    public void validate(String email, String password, String name) {
        if (name == null || name.isBlank() || name.length() > 20) {
            throw new IllegalArgumentException("invalid name: " + name);
        }
    }
}
```

最后，我们通过一个Validators作为入口进行验证：

```
@Component
public class Validators {
    @Autowired
    List<Validator> validators;

    public void validate(String email, String password, String name) {
        for (var validator : this.validators) {
            validator.validate(email, password, name);
        }
    }
}
```

注意到Validators被注入了一个List<Validator>，Spring会自动把所有类型为Validator的Bean装配为一个List注入进来，这样一来，我们每新增一个Validator类型，就自动被Spring装配到Validators中了，非常方便。

因为Spring是通过扫描classpath获取到所有的Bean，而List是有序的，要指定List中Bean的顺序，可以加上@Order注解：

```
@Component
@Order(1)
public class EmailValidator implements Validator {
    ...
}

@Component
@Order(2)
public class PasswordValidator implements Validator {
    ...
}

@Component
@Order(3)
public class NameValidator implements Validator {
    ...
}
```

可选注入

默认情况下，当我们标记了一个@Autowired后，Spring如果没有找到对应类型的Bean，它会抛出NoSuchBeanDefinitionException异常。

可以给@Autowired增加一个required = false的参数：

```
@Component
public class MailService {
    @Autowired(required = false)
    ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

这个参数告诉Spring容器，如果找到一个类型为ZoneId的Bean，就注入，如果找不到，就忽略。

这种方式非常适合有定义就使用定义，没有就使用默认值的情况。

创建第三方Bean

如果一个Bean不在我们自己的package管理之内，例如ZoneId，如何创建它？

答案是我们自己在@Configuration类中编写一个Java方法创建并返回它，注意给方法标记一个@Bean注解：

```
@Configuration
@ComponentScan
public class AppConfig {
    // 创建一个Bean:
    @Bean
    ZoneId createZoneId() {
        return ZoneId.of("Z");
    }
}
```

Spring对标记为@Bean的方法只调用一次，因此返回的Bean仍然是单例。

初始化和销毁

有些时候，一个Bean在注入必要的依赖后，需要进行初始化（监听消息等）。在容器关闭时，有时候还需要清理资源（关闭连接池等）。我们通常会定义一个`init()`方法进行初始化，定义一个`shutdown()`方法进行清理，然后，引入JSR-250定义的Annotation：

```
<dependency>
    <groupId>javax.annotation</groupId>
    <artifactId>javax.annotation-api</artifactId>
    <version>1.3.2</version>
</dependency>
```

在Bean的初始化和清理方法上标记`@PostConstruct`和`@PreDestroy`：

```
@Component
public class MailService {
    @Autowired(required = false)
    ZoneId zoneId = ZoneId.systemDefault();

    @PostConstruct
    public void init() {
        System.out.println("Init mail service with zoneId = " + this.zoneId);
    }

    @PreDestroy
    public void shutdown() {
        System.out.println("Shutdown mail service");
    }
}
```

Spring容器会对上述Bean做如下初始化流程：

- 调用构造方法创建`MailService`实例；
- 根据`@Autowired`进行注入；
- 调用标记有`@PostConstruct`的`init()`方法进行初始化。

而销毁时，容器会首先调用标记有`@PreDestroy`的`shutdown()`方法。

Spring只根据Annotation查找*无参数*方法，对方法名不作要求。

使用别名

默认情况下，对一种类型的Bean，容器只创建一个实例。但有些时候，我们需要对一种类型的Bean创建多个实例。例如，同时连接多个数据库，就必须创建多个`DataSource`实例。

如果我们在`@Configuration`类中创建了多个同类型的Bean：

```
@Configuration
@ComponentScan
public class AppConfig {
    @Bean
    ZoneId createZoneOfZ() {
        return ZoneId.of("Z");
    }

    @Bean
    ZoneId createZoneOfUTC8() {
        return ZoneId.of("UTC+08:00");
    }
}
```

Spring会报`NoUniqueBeanDefinitionException`异常，意思是出现了重复的Bean定义。

这个时候，需要给每个Bean添加不同的名字：

```
@Configuration
@ComponentScan
public class AppConfig {
    @Bean("z")
    ZoneId createZoneOfZ() {
        return ZoneId.of("Z");
    }

    @Bean
    @Qualifier("utc8")
    ZoneId createZoneOfUTC8() {
        return ZoneId.of("UTC+08:00");
    }
}
```

可以用`@Bean("name")`指定别名，也可以用`@Bean`+`@Qualifier("name")`指定别名。

存在多个同类型的Bean时，注入`ZoneId`又会报错：

```
NoUniqueBeanDefinitionException: No qualifying bean of type 'java.time.ZoneId' available: expected single matching bean but found 2
```

意思是期待找到唯一的`ZoneId`类型Bean，但是找到两。因此，注入时，要指定Bean的名称：

```
@Component
public class MailService {
 @Autowired(required = false)
 @Qualifier("z") // 指定注入名称为"z"的ZoneId
 ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

还有一种方法是把其中某个Bean指定为`@Primary`：

```
@Configuration
@ComponentScan
public class AppConfig {
    @Bean
    @Primary // 指定为主要Bean
    @Qualifier("z")
    ZoneId createZoneOfZ() {
        return ZoneId.of("Z");
    }

    @Bean
    @Qualifier("utc8")
    ZoneId createZoneOfUTC8() {
        return ZoneId.of("UTC+08:00");
    }
}
```

这样，在注入时，如果没有指出Bean的名字，Spring会注入标记有`@Primary`的Bean。这种方式也很常用。例如，对于主从两个数据源，通常将主数据源定义为`@Primary`：

```
@Configuration
@ComponentScan
public class AppConfig {
    @Bean
```

```
    @Primary
    DataSource createMasterDataSource() {
        ...
    }

    @Bean
    @Qualifier("slave")
    DataSource createSlaveDataSource() {
        ...
    }
}
```

其他Bean默认注入的就是主数据源。如果要注入从数据源，那么只需要指定名称即可。

使用FactoryBean

我们在设计模式的工厂方法中讲到，很多时候，可以通过工厂模式创建对象。Spring也提供了工厂模式，允许定义一个工厂，然后由工厂创建真正的Bean。

用工厂模式创建Bean需要实现FactoryBean接口。我们观察下面的代码：

```
@Component
public class ZoneIdFactoryBean implements FactoryBean<ZoneId> {

    String zone = "Z";

    @Override
    public ZoneId getObject() throws Exception {
        return ZoneId.of(zone);
    }

    @Override
    public Class<?> getObjectType() {
        return ZoneId.class;
    }
}
```

当一个Bean实现了FactoryBean接口后，Spring会先实例化这个工厂，然后调用getObject()创建真正的Bean。getObjectType()可以指定创建的Bean的类型，因为指定类型不一定与实际类型一致，可以是接口或抽象类。

因此，如果定义了一个FactoryBean，要注意Spring创建的Bean实际上是这个FactoryBean的getObject()方法返回的Bean。为了和普通Bean区分，我们通常都以XxxFactoryBean命名。

练习

定制Bean

小结

Spring默认使用Singleton创建Bean，也可指定Scope为Prototype；

可将相同类型的Bean注入List；

可用@Autowired(required=false)允许可选注入；

可用带@Bean标注的方法创建Bean；

可使用@PostConstruct和@PreDestroy对Bean进行初始化和清理；

相同类型的Bean只能有一个指定为@Primary，其他必须用@Quanlifier("beanName")指定别名；

注入时，可通过别名@Quanlifier("beanName")指定某个Bean；

可以定义FactoryBean来使用工厂模式创建Bean。

在Java程序中，我们经常会读取配置文件、资源文件等。使用Spring容器时，我们也可以把“文件”注入进来，方便程序读取。

例如，AppService需要读取logo.txt这个文件，通常情况下，我们需要写很多繁琐的代码，主要是为了定位文件，打开InputStream。

Spring提供了一个org.springframework.core.io.Resource（注意不是javax.annotation.Resource），它可以像String、int一样使用@Value注入：

```
@Component
public class AppService {
    @Value("classpath:/logo.txt")
    private Resource resource;

    private String logo;

    @PostConstruct
    public void init() throws IOException {
        try (var reader = new BufferedReader(
                new InputStreamReader(resource.getInputStream(), StandardCharsets.UTF_8))) {
            this.logo = reader.lines().collect(Collectors.joining("\n"));
        }
    }
}
```

注入Resource最常用的方式是通过classpath，即类似classpath:/logo.txt表示在classpath中搜索logo.txt文件，然后，我们直接调用Resource.getInputStream()就可以获取到输入流，避免了自己搜索文件的代码。

也可以直接指定文件的路径，例如：

```
@Value("file:/path/to/logo.txt")
private Resource resource;
```

但使用classpath是最简单的方式。上述工程结构如下：

```
spring-ioc-resource
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── itranswarp
        │           └── learnjava
        │               ├── AppConfig.java
        │               └── AppService.java
        └── resources
            └── logo.txt
```

使用Maven的标准目录结构，所有资源文件放入src/main/resources即可。

练习

使用Spring的Resource注入app.properties文件，然后读取该配置文件。

使用Resource

小结

Spring提供了Resource类便于注入资源文件。

最常用的注入是通过classpath以`classpath:/path/to/file`的形式注入。

在开发应用程序时，经常需要读取配置文件。最常用的配置方法是以`key=value`的形式写在`.properties`文件中。

例如，`MailService`根据配置的`app.zone=Asia/Shanghai`来决定使用哪个时区。要读取配置文件，我们可以使用上一节讲到的`Resource`来读取位于classpath下的一个`app.properties`文件。但是，这样仍然比较繁琐。

Spring容器还提供了一个更简单的`@PropertySource`来自动读取配置文件。我们只需要在`@Configuration`配置类上再添加一个注解：

```
@Configuration
@ComponentScan
@PropertySource("app.properties") // 表示读取classpath的app.properties
public class AppConfig {
    @Value("${app.zone:Z}")
    String zoneId;

    @Bean
    ZoneId createZoneId() {
        return ZoneId.of(zoneId);
    }
}
```

Spring容器看到`@PropertySource("app.properties")`注解后，自动读取这个配置文件，然后，我们使用`@Value`正常注入：

```
@Value("${app.zone:Z}")
String zoneId;
```

注意注入的字符串语法，它的格式如下：

- `"${app.zone}"`表示读取key为`app.zone`的value，如果key不存在，启动将报错；
- `"${app.zone:Z}"`表示读取key为`app.zone`的value，但如果key不存在，就使用默认值`Z`。

这样一来，我们就可以根据`app.zone`的配置来创建`ZoneId`。

还可以把注入的注解写到方法参数中：

```
@Bean
ZoneId createZoneId(@Value("${app.zone:Z}") String zoneId) {
    return ZoneId.of(zoneId);
}
```

可见，先使用`@PropertySource`读取配置文件，然后通过`@Value`以`${key:defaultValue}`的形式注入，可以极大地简化读取配置的麻烦。

另一种注入配置的方式是先通过一个简单的JavaBean持有所有的配置，例如，一个`SmtpConfig`：

```
@Component
public class SmtpConfig {
    @Value("${smtp.host}")
    private String host;

    @Value("${smtp.port:25}")
    private int port;

    public String getHost() {
        return host;
    }

    public int getPort() {
        return port;
    }
}
```

然后，在需要读取的地方，使用`#{smtpConfig.host}`注入：

```
@Component
public class MailService {
    @Value("#{smtpConfig.host}")
    private String smtpHost;

    @Value("#{smtpConfig.port}")
    private int smtpPort;
}
```

注意观察`#{}`这种注入语法，它和`${key}`不同的是，`#{}`表示从JavaBean读取属性。`"#{smtpConfig.host}"`的意思是，从名称为`smtpConfig`的Bean读取`host`属性，即调用`getHost()`方法。一个Class名为`SmtpConfig`的Bean，它在Spring容器中的默认名称就是`smtpConfig`，除非用`@Qualifier`指定了名称。

使用一个独立的JavaBean持有所有属性，然后在其他Bean中以`#{bean.property}`注入的好处是，多个Bean都可以引用同一个Bean的某个属性。例如，如果`SmtpConfig`决定从数据库中读取相关配置项，那么`MailService`注入的`@Value("#{smtpConfig.host}")`仍然可以不修改正常运行。

练习

注入SMTP配置

小结

Spring容器可以通过`@PropertySource`自动读取配置，并以`@Value("${key}")`的形式注入；

可以通过`${key:defaultValue}`指定默认值；

以`#{bean.property}`形式注入时，Spring容器自动把指定Bean的指定属性值注入。

开发应用程序时，我们会使用开发环境，例如，使用内存数据库以便快速启动。而运行在生产环境时，我们会使用生产环境，例如，使用MySQL数据库。如果应用程序可以根据自身的环境做一些适配，无疑会更加灵活。

Spring为应用程序准备了Profile这一概念，用来表示不同的环境。例如，我们分别定义开发、测试和生产这3个环境：

- native
- test
- production

创建某个Bean时，Spring容器可以根据注解`@Profile`来决定是否创建。例如，以下配置：

```
@Configuration
@ComponentScan
public class AppConfig {
    @Bean
```

```
    @Profile("!test")
    ZoneId createZoneId() {
        return ZoneId.systemDefault();
    }

    @Bean
    @Profile("test")
    ZoneId createZoneIdForTest() {
        return ZoneId.of("America/New_York");
    }
}
```

如果当前的Profile设置为test，则Spring容器会调用createZoneIdForTest()创建ZoneId，否则，调用createZoneId()创建ZoneId。注意到@Profile("!test")表示非test环境。

在运行程序时，加上JVM参数-Dspring.profiles.active=test就可以指定以test环境启动。

实际上，Spring允许指定多个Profile，例如：

```
-Dspring.profiles.active=test,master
```

可以表示test环境，并使用master分支代码。

要满足多个Profile条件，可以这样写：

```
@Bean
@Profile({ "test", "master" }) // 同时满足test和master
ZoneId createZoneId() {
    ...
}
```

使用Conditional

除了根据@Profile条件来决定是否创建某个Bean外，Spring还可以根据@Conditional决定是否创建某个Bean。

例如，我们对SmtpMailService添加如下注解：

```
@Component
@Conditional(OnSmtpEnvCondition.class)
public class SmtpMailService implements MailService {
    ...
}
```

它的意思是，如果满足OnSmtpEnvCondition的条件，才会创建SmtpMailService这个Bean。OnSmtpEnvCondition的条件是什么呢？我们看一下代码：

```
public class OnSmtpEnvCondition implements Condition {
    public boolean matches(ConditionContext context, AnnotatedTypeMetadata metadata) {
        return "true".equalsIgnoreCase(System.getenv("smtp"));
    }
}
```

因此，OnSmtpEnvCondition的条件是存在环境变量smtp，值为true。这样，我们就可以通过环境变量来控制是否创建SmtpMailService。

Spring只提供了@Conditional注解，具体判断逻辑还需要我们自己实现。Spring Boot提供了更多使用起来更简单的条件注解，例如，如果配置文件中存在app.smtp=true，则创建MailService：

```
@Component
@ConditionalOnProperty(name="app.smtp", havingValue="true")
public class MailService {
    ...
}
```

如果当前classpath中存在类javax.mail.Transport，则创建MailService：

```
@Component
@ConditionalOnClass(name = "javax.mail.Transport")
public class MailService {
    ...
}
```

后续我们会介绍Spring Boot的条件装配。我们以文件存储为例，假设我们需要保存用户上传的头像，并返回存储路径，在本地开发运行时，我们总是存储到文件：

```
@Component
@ConditionalOnProperty(name = "app.storage", havingValue = "file", matchIfMissing = true)
public class FileUploader implements Uploader {
    ...
}
```

在生产环境运行时，我们会把文件存储到类似AWS S3上：

```
@Component
@ConditionalOnProperty(name = "app.storage", havingValue = "s3")
public class S3Uploader implements Uploader {
    ...
}
```

其他需要存储的服务则注入Uploader：

```
@Component
public class UserImageService {
    @Autowired
    Uploader uploader;
}
```

当应用程序检测到配置文件存在app.storage=s3时，自动使用S3Uploader，如果存在配置app.storage=file，或者配置app.storage不存在，则使用FileUploader。

可见，使用条件注解，能更灵活地装配Bean。

练习

使用@Profile进行条件装配

小结

Spring允许通过@Profile配置不同的Bean；

Spring还提供了@Conditional来进行条件装配，Spring Boot在此基础上进一步提供了基于配置、Class、Bean等条件进行装配。

AOP是Aspect Oriented Programming，即面向切面编程。

那什么是AOP？

我们先回顾一下OOP：Object Oriented Programming，OOP作为面向对象编程的模式，获得了巨大的成功，OOP的主要功能是数据封装、继承和多态。

而AOP是一种新的编程方式，它和OOP不同，OOP把系统看作多个对象的交互，AOP把系统分解为不同的关注点，或者称之为切面（Aspect）。

要理解AOP的概念，我们先用OOP举例，比如一个业务组件`BookService`，它有几个业务方法：

- createBook：添加新的Book；
- updateBook：修改Book；
- deleteBook：删除Book。

对每个业务方法，例如，`createBook()`，除了业务逻辑，还需要安全检查、日志记录和事务处理，它的代码像这样：

```
public class BookService {
    public void createBook(Book book) {
        securityCheck();
        Transaction tx = startTransaction();
        try {
            // 核心业务逻辑
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        }
        log("created book: " + book);
    }
}
```

继续编写`updateBook()`，代码如下：

```
public class BookService {
    public void updateBook(Book book) {
        securityCheck();
        Transaction tx = startTransaction();
        try {
            // 核心业务逻辑
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        }
        log("updated book: " + book);
    }
}
```

对于安全检查、日志、事务等代码，它们会重复出现在每个业务方法中。使用OOP，我们很难将这些四处分散的代码模块化。

考察业务模型可以发现，`BookService`关心的是自身的核心逻辑，但整个系统还要求关注安全检查、日志、事务等功能，这些功能实际上“横跨”多个业务方法，为了实现这些功能，不得不在每个业务方法上重复编写代码。

一种可行的方式是使用Proxy模式，将某个功能，例如，权限检查，放入Proxy中：

```
public class SecurityCheckBookService implements BookService {
    private final BookService target;

    public SecurityCheckBookService(BookService target) {
        this.target = target;
    }

    public void createBook(Book book) {
        securityCheck();
        target.createBook(book);
    }

    public void updateBook(Book book) {
        securityCheck();
        target.updateBook(book);
    }

    public void deleteBook(Book book) {
        securityCheck();
        target.deleteBook(book);
    }

    private void securityCheck() {
        ...
    }
}
```

这种方式的缺点是比较麻烦，必须先抽取接口，然后，针对每个方法实现Proxy。

另一种方法是，既然`SecurityCheckBookService`的代码都是标准的Proxy样板代码，不如把权限检查视作一种切面（Aspect），把日志、事务也视为切面，然后，以某种自动化的方式，把切面织入到核心逻辑中，实现Proxy模式。

如果我们以AOP的视角来编写上述业务，可以依次实现：

1. 核心逻辑，即BookService；
2. 切面逻辑，即：
3. 权限检查的Aspect；
4. 日志的Aspect；
5. 事务的Aspect。

然后，以某种方式，让框架来把上述3个Aspect以Proxy的方式“织入”到`BookService`中，这样一来，就不必编写复杂而冗长的Proxy模式。

AOP原理

如何把切面织入到核心逻辑中？这正是AOP需要解决的问题。换句话说，如果客户端获得了`BookService`的引用，当调用`bookService.createBook()`时，如何对调用方法进行拦截，并在拦截前后进行安全检查、日志、事务等处理，就相当于完成了所有业务功能。

在Java平台上，对于AOP的织入，有3种方式：

1. 编译期：在编译时，由编译器把切面调用编译进字节码，这种方式需要定义新的关键字并扩展编译器，AspectJ就扩展了Java编译器，使用关键字aspect来实现织入；
2. 类加载器：在目标类被装载到JVM时，通过一个特殊的类加载器，对目标类的字节码重新“增强”；
3. 运行期：目标对象和切面都是普通Java类，通过JVM的动态代理功能或者第三方库实现运行期动态织入。

最简单的方式是第三种，Spring的AOP实现就是基于JVM的动态代理。由于JVM的动态代理要求必须实现接口，如果一个普通类没有业务接口，就需要通过CGLIB或者Javassist这些第三方库实现。

AOP技术看上去比较神秘，但实际上，它本质就是一个动态代理，让我们把一些常用功能如权限检查、日志、事务等，从每个业务方法中剥离出来。

需要特别指出的是，AOP对于解决特定问题，例如事务管理非常有用，这是因为分散在各处的事务代码几乎是完全相同的，并且它们需要的参数（JDBC的Connection）也是固定的。另一些特定问题，如日志，就不那么容易实现，因为日志虽然简单，但打印日志的时候，经常需要捕获局部变量，如果使用AOP实现日志，我们只能输出固定格式的日志，因此，使用AOP时，必须适合特定的场景。

在AOP编程中，我们经常会遇到下面的概念：

- Aspect：切面，即一个横跨多个核心逻辑的功能，或者称之为系统关注点；
- Joinpoint：连接点，即定义在应用程序流程的何处插入切面的执行；
- Pointcut：切入点，即一组连接点的集合；
- Advice：增强，指特定连接点上执行的动作；

- Introduction：引介，指为一个已有的Java对象动态地增加新的接口；
- Weaving：织入，指将切面整合到程序的执行流程中；
- Interceptor：拦截器，是一种实现增强的方式；
- Target Object：目标对象，即真正执行业务的核心逻辑对象；
- AOP Proxy：AOP代理，是客户端持有的增强后的对象引用。

看完上述术语，是不是感觉对AOP有了进一步的困惑？其实，我们不用关心AOP创造的“术语”，只需要理解AOP本质上只是一种代理模式的实现方式，在Spring的容器中实现AOP特别方便。

我们以`UserService`和`MailService`为例，这两个属于核心业务逻辑，现在，我们准备给`UserService`的每个业务方法执行前添加日志，给`MailService`的每个业务方法执行前后添加日志，在Spring中，需要以下步骤：

首先，我们通过Maven引入Spring对AOP的支持：

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-aspects</artifactId>
    <version>${spring.version}</version>
</dependency>
```

上述依赖会自动引入AspectJ，使用AspectJ实现AOP比较方便，因为它的定义比较简单。

然后，我们定义一个`LoggingAspect`：

```
@Aspect
@Component
public class LoggingAspect {
    // 在执行UserService的每个方法前执行:
    @Before("execution(public * com.itranswarp.learnjava.service.UserService.*(..))")
    public void doAccessCheck() {
        System.err.println("[Before] do access check...");
    }

    // 在执行MailService的每个方法前后执行:
    @Around("execution(public * com.itranswarp.learnjava.service.MailService.*(..))")
    public Object doLogging(ProceedingJoinPoint pjp) throws Throwable {
        System.err.println("[Around] start " + pjp.getSignature());
        Object retVal = pjp.proceed();
        System.err.println("[Around] done " + pjp.getSignature());
        return retVal;
    }
}
```

观察`doAccessCheck()`方法，我们定义了一个`@Before`注解，后面的字符串是告诉AspectJ应该在何处执行该方法，这里写的意思是：执行`UserService`的每个`public`方法前执行`doAccessCheck()`代码。

再观察`doLogging()`方法，我们定义了一个`@Around`注解，它和`@Before`不同，`@Around`可以决定是否执行目标方法，因此，我们在`doLogging()`内部先打印日志，再调用方法，最后打印日志后返回结果。

在`LoggingAspect`类的声明处，除了用`@Component`表示它本身也是一个Bean外，我们再加上`@Aspect`注解，表示它的`@Before`标注的方法需要注入到`UserService`的每个`public`方法执行前，`@Around`标注的方法需要注入到`MailService`的每个`public`方法执行前后。

紧接着，我们需要给`@Configuration`类加上一个`@EnableAspectJAutoProxy`注解：

```
@Configuration
@ComponentScan
@EnableAspectJAutoProxy
public class AppConfig {
    ...
}
```

Spring的IoC容器看到这个注解，就会自动查找带有`@Aspect`的Bean，然后根据每个方法的`@Before`、`@Around`等注解把AOP注入到特定的Bean中。执行代码，我们可以看到以下输出：

```
[Before] do access check...
[Around] start void com.itranswarp.learnjava.service.MailService.sendRegistrationMail(User)
Welcome, test!
[Around] done void com.itranswarp.learnjava.service.MailService.sendRegistrationMail(User)
[Before] do access check...
[Around] start void com.itranswarp.learnjava.service.MailService.sendLoginMail(User)
Hi, Bob! You are logged in at 2020-02-14T23:13:52.167996+08:00[Asia/Shanghai]
[Around] done void com.itranswarp.learnjava.service.MailService.sendLoginMail(User)
```

这说明执行业务逻辑前后，确实执行了我们定义的Aspect（即`LoggingAspect`的方法）。

有些童鞋会问，`LoggingAspect`定义的方法，是如何注入到其他Bean的呢？

其实AOP的原理非常简单。我们以`LoggingAspect.doAccessCheck()`为例，要把它注入到`UserService`的每个`public`方法中，最简单的方法是编写一个子类，并持有原始实例的引用：

```
public UserServiceAopProxy extends UserService {
    private UserService target;
    private LoggingAspect aspect;

    public UserServiceAopProxy(UserService target, LoggingAspect aspect) {
        this.target = target;
        this.aspect = aspect;
    }

    public User login(String email, String password) {
        // 先执行Aspect的代码:
        aspect.doAccessCheck();
        // 再执行UserService的逻辑:
        return target.login(email, password);
    }

    public User register(String email, String password, String name) {
        aspect.doAccessCheck();
        return target.register(email, password, name);
    }

    ...
}
```

这些都是Spring容器启动时为我们自动创建的注入了Aspect的子类，它取代了原始的`UserService`（原始的`UserService`实例作为内部变量隐藏在`UserServiceAopProxy`中）。如果我们打印从Spring容器获取的`UserService`实例类型，它类似`UserService$$EnhancerBySpringCGLIB$$1f44e01c`，实际上是Spring使用CGLIB动态创建的子类，但对于调用方来说，感觉不到任何区别。

Spring对接口类型使用JDK动态代理，对普通类使用CGLIB创建子类。如果一个Bean的class是final，Spring将无法为其创建子类。

可见，虽然Spring容器内部实现AOP的逻辑比较复杂（需要使用AspectJ解析注解，并通过CGLIB实现代理类），但我们使用AOP非常简单，一共需要三步：

1. 定义执行方法，并在方法上通过AspectJ的注解告诉Spring应该在何处调用此方法；
2. 标记`@Component`和`@Aspect`；
3. 在`@Configuration`类上标注`@EnableAspectJAutoProxy`。

至于AspectJ的注入语法则比较复杂，请参考Spring文档。

Spring也提供其他方法来装配AOP，但都没有使用AspectJ注解的方式来得简洁明了，所以我们不再作介绍。

拦截器类型

顾名思义，拦截器有以下类型：

- @Before：这种拦截器先执行拦截代码，再执行目标代码。如果拦截器抛异常，那么目标代码就不执行了；
- @After：这种拦截器先执行目标代码，再执行拦截器代码。无论目标代码是否抛异常，拦截器代码都会执行；
- @AfterReturning：和@After不同的是，只有当目标代码正常返回时，才执行拦截器代码；
- @AfterThrowing：和@After不同的是，只有当目标代码抛出了异常时，才执行拦截器代码；
- @Around：能完全控制目标代码是否执行，并可以在执行前后、抛异常后执行任意拦截代码，可以说是包含了上面所有功能。

练习

使用AOP实现日志

小结

在Spring容器中使用AOP非常简单，只需要定义执行方法，并用AspectJ的注解标注应该在何处触发并执行。

Spring通过CGLIB动态创建子类等方式来实现AOP代理模式，大大简化了代码。

上一节我们讲解了使用AspectJ的注解，并配合一个复杂的`execution(* xxx.Xyz.*(..))`语法来定义应该如何装配AOP。

在实际项目中，这种写法其实很少使用。假设你写了一个`SecurityAspect`：

```
@Aspect
@Component
public class SecurityAspect {
    @Before("execution(public * com.itranswarp.learnjava.service.*.*(..))")
    public void check() {
        if (SecurityContext.getCurrentUser() == null) {
            throw new RuntimeException("check failed");
        }
    }
}
```

基本能实现无差别全覆盖，即某个包下面的所有Bean的所有方法都会被这个`check()`方法拦截。

还有的童鞋喜欢用方法名前缀进行拦截：

```
@Around("execution(public * update*(..))")
public Object doLogging(ProceedingJoinPoint pjp) throws Throwable {
    // 对update开头的方法切换数据源:
    String old = setCurrentDataSource("master");
    Object retVal = pjp.proceed();
    restoreCurrentDataSource(old);
    return retVal;
}
```

这种非精准打击误伤面更大，因为从方法前缀区分是否是数据库操作是非常不可取的。

我们在使用AOP时，要注意到虽然Spring容器可以把指定的方法通过AOP规则装配到指定的Bean的指定方法前后，但是，如果自动装配时，因为不恰当的范围，容易导致意想不到的结果，即很多不需要AOP代理的Bean也被自动代理了，并且，后续新增的Bean，如果不清楚现有的AOP装配规则，容易被强迫装配。

使用AOP时，被装配的Bean最好自己能清清楚楚地知道自己被安排了。例如，Spring提供的`@Transactional`就是一个非常好的例子。如果我们自己写的Bean希望在一个数据库事务中被调用，就标注上`@Transactional`：

```
@Component
public class UserService {
    // 有事务:
    @Transactional
    public User createUser(String name) {
        ...
    }

    // 无事务:
    public boolean isValidName(String name) {
        ...
    }

    // 有事务:
    @Transactional
    public void updateUser(User user) {
        ...
    }
}
```

或者直接在class级别注解，表示“所有public方法都被安排了”：

```
@Component
@Transactional
public class UserService {
    ...
}
```

通过`@Transactional`，某个方法是否启用了事务就一清二楚了。因此，装配AOP的时候，使用注解是最好的方式。

我们以一个实际例子演示如何使用注解实现AOP装配。为了监控应用程序的性能，我们定义一个性能监控的注解：

```
@Target(METHOD)
@Retention(RUNTIME)
public @interface MetricTime {
    String value();
}
```

在需要被监控的关键方法上标注该注解：

```
@Component
public class UserService {
    // 监控register()方法性能:
    @MetricTime("register")
    public User register(String email, String password, String name) {
        ...
    }
    ...
}
```

然后，我们定义`MetricAspect`：

```
@Aspect
@Component
public class MetricAspect {
    @Around("@annotation(metricTime)")
```

```
    public Object metric(ProceedingJoinPoint joinPoint, MetricTime metricTime) throws Throwable {
        String name = metricTime.value();
        long start = System.currentTimeMillis();
        try {
            return joinPoint.proceed();
        } finally {
            long t = System.currentTimeMillis() - start;
            // 写入日志或发送至JMX:
            System.err.println("[Metrics] " + name + ": " + t + "ms");
        }
    }
}
```

注意`metric()`方法标注了`@Around("@annotation(metricTime)")`，它的意思是，符合条件的目标方法是带有`@MetricTime`注解的方法，因为`metric()`方法参数类型是`MetricTime`（注意参数名是`metricTime`不是`MetricTime`），我们通过它获取性能监控的名称。

有了`@MetricTime`注解，再配合`MetricAspect`，任何Bean，只要方法标注了`@MetricTime`注解，就可以自动实现性能监控。运行代码，输出结果如下：

```
Welcome, Bob!
[Metrics] register: 16ms
```

练习

使用注解+AOP实现性能监控

小结

使用注解实现AOP需要先定义注解，然后使用`@Around("@annotation(name)")`实现装配；

使用注解既简单，又能明确标识AOP装配，是使用AOP推荐的方式。

无论是使用AspectJ语法，还是配合Annotation，使用AOP，实际上就是让Spring自动为我们创建一个Proxy，使得调用方能无感知地调用指定方法，但运行期却动态“织入”了其他逻辑，因此，AOP本质上就是一个代理模式。

因为Spring使用了CGLIB来实现运行期动态创建Proxy，如果我们没能深入理解其运行原理和实现机制，就极有可能遇到各种诡异的问题。

我们来看一个实际的例子。

假设我们定义了一个`UserService`的Bean：

```
@Component
public class UserService {
    // 成员变量:
    public final ZoneId zoneId = ZoneId.systemDefault();

    // 构造方法:
    public UserService() {
        System.out.println("UserService(): init...");
        System.out.println("UserService(): zoneId = " + this.zoneId);
    }

    // public方法:
    public ZoneId getZoneId() {
        return zoneId;
    }

    // public final方法:
    public final ZoneId getFinalZoneId() {
        return zoneId;
    }
}
```

再写个`MailService`，并注入`UserService`：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        String dt = ZonedDateTime.now(zoneId).toString();
        return "Hello, it is " + dt;
    }
}
```

最后用`main()`方法测试一下：

```
@Configuration
@ComponentScan
public class AppConfig {
    public static void main(String[] args) {
        ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
        MailService mailService = context.getBean(MailService.class);
        System.out.println(mailService.sendMail());
    }
}
```

查看输出，一切正常：

```
UserService(): init...
UserService(): zoneId = Asia/Shanghai
Hello, it is 2020-04-12T10:23:22.917721+08:00[Asia/Shanghai]
```

下一步，我们给`UserService`加上AOP支持，就添加一个最简单的`LoggingAspect`：

```
@Aspect
@Component
public class LoggingAspect {
    @Before("execution(public * com..*.UserService.*(..))")
    public void doAccessCheck() {
        System.err.println("[Before] do access check...");
    }
}
```

别忘了在`AppConfig`上加上`@EnableAspectJAutoProxy`。再次运行，不出意外的话，会得到一个`NullPointerException`：

```
Exception in thread "main" java.lang.NullPointerException: zone
    at java.base/java.util.Objects.requireNonNull(Objects.java:246)
    at java.base/java.time.Clock.system(Clock.java:203)
    at java.base/java.time.ZonedDateTime.now(ZonedDateTime.java:216)
    at com.itranswarp.learnjava.service.MailService.sendMail(MailService.java:19)
    at com.itranswarp.learnjava.AppConfig.main(AppConfig.java:21)
```

仔细跟踪代码，会发现`null`值出现在`MailService.sendMail()`内部的这一行代码：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        System.out.println(zoneId); // null
        ...
    }
}
```

我们还故意在UserService中特意用final修饰了一下成员变量：

```
@Component
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

用final标注的成员变量为null？逗我呢？

怎么肥四？

为什么加了AOP就报NPE，去了AOP就一切正常？final字段不执行，难道JVM有问题？为了解答这个诡异的问题，我们需要深入理解Spring使用CGLIB生成Proxy的原理：

第一步，正常创建一个UserService的原始实例，这是通过反射调用构造方法实现的，它的行为和我们预期的完全一致；

第二步，通过CGLIB创建一个UserService的子类，并引用了原始实例和LoggingAspect：

```
public UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target;
    LoggingAspect aspect;

    public UserService$$EnhancerBySpringCGLIB() {
    }

    public ZoneId getZoneId() {
        aspect.doAccessCheck();
        return target.getZoneId();
    }
}
```

如果我们观察Spring创建的AOP代理，它的类名总是类似UserService$$EnhancerBySpringCGLIB$$1c76af9d（你没看错，Java的类名实际上允许$字符）。为了让调用方获得UserService的引用，它必须继承自UserService。然后，该代理类会覆写所有public和protected方法，并在内部将调用委托给原始的UserService实例。

这里出现了两个UserService实例：

一个是我们代码中定义的*原始实例*，它的成员变量已经按照我们预期的方式被初始化完成：

```
UserService original = new UserService();
```

第二个UserService实例实际上类型是UserService$$EnhancerBySpringCGLIB，它引用了原始的UserService实例：

```
UserService$$EnhancerBySpringCGLIB proxy = new UserService$$EnhancerBySpringCGLIB();
proxy.target = original;
proxy.aspect = ...
```

注意到这种情况仅出现在启用了AOP的情况，此刻，从ApplicationContext中获取的UserService实例是proxy，注入到MailService中的UserService实例也是proxy。

那么最终的问题来了：proxy实例的成员变量，也就是从UserService继承的zoneId，它的值是null。

原因在于，UserService成员变量的初始化：

```
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    ...
}
```

在UserService$$EnhancerBySpringCGLIB中，并未执行。原因是，没必要初始化proxy的成员变量，因为proxy的目的是代理方法。

实际上，成员变量的初始化是在构造方法中完成的。这是我们看到的代码：

```
public class UserService {
    public final ZoneId zoneId = ZoneId.systemDefault();
    public UserService() {
    }
}
```

这是编译器实际编译的代码：

```
public class UserService {
    public final ZoneId zoneId;
    public UserService() {
        super(); // 构造方法的第一行代码总是调用super()
        zoneId = ZoneId.systemDefault(); // 继续初始化成员变量
    }
}
```

然而，对于Spring通过CGLIB动态创建的UserService$$EnhancerBySpringCGLIB代理类，它的构造方法中，并未调用super()，因此，从父类继承的成员变量，包括final类型的成员变量，统统都没有初始化。

有的童鞋会问：Java语言规定，任何类的构造方法，第一行必须调用super()，如果没有，编译器会自动加上，怎么Spring的CGLIB就可以搞特殊？

这是因为自动加super()的功能是Java编译器实现的，它发现你没加，就自动给加上，发现你加错了，就报编译错误。但实际上，如果直接构造字节码，一个类的构造方法中，不一定非要调用super()。Spring使用CGLIB构造的Proxy类，是直接生成字节码，并没有源码-编译-字节码这个步骤，因此：

Spring通过CGLIB创建的代理类，不会初始化代理类自身继承的任何成员变量，包括final类型的成员变量！

再考察MailService的代码：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        ZoneId zoneId = userService.zoneId;
        System.out.println(zoneId); // null
        ...
    }
}
```

如果没有启用AOP，注入的是原始的UserService实例，那么一切正常，因为UserService实例的zoneId字段已经被正确初始化了。

如果启动了AOP，注入的是代理后的UserService$$EnhancerBySpringCGLIB实例，那么问题大了：获取的UserService$$EnhancerBySpringCGLIB实例的zoneId字段，永远为null。

那么问题来了：启用了AOP，如何修复？

修复很简单，只需要把直接访问字段的代码，改为通过方法访问：

```
@Component
public class MailService {
    @Autowired
    UserService userService;

    public String sendMail() {
        // 不要直接访问UserService的字段:
        ZoneId zoneId = userService.getZoneId();
        ...
    }
}
```

无论注入的`UserService`是原始实例还是代理实例，`getZoneId()`都能正常工作，因为代理类会覆写`getZoneId()`方法，并将其委托给原始实例：

```
public UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target = ...
    ...

    public ZoneId getZoneId() {
        return target.getZoneId();
    }
}
```

注意到我们还给`UserService`添加了一个`public+final`的方法：

```
@Component
public class UserService {
    ...
    public final ZoneId getFinalZoneId() {
        return zoneId;
    }
}
```

如果在`MailService`中，调用的不是`getZoneId()`，而是`getFinalZoneId()`，又会出现`NullPointerException`，这是因为，代理类无法覆写`final`方法（这一点绕不过JVM的ClassLoader检查），该方法返回的是代理类的`zoneId`字段，即`null`。

实际上，如果我们加上日志，Spring在启动时会打印一个警告：

```
10:43:09.929 [main] DEBUG org.springframework.aop.framework.CglibAopProxy - Final method [public final java.time.ZoneId xxx.UserService.getFinalZoneId()] cannot get proxied via CGLIB:
```

上面的日志大意就是，因为被代理的`UserService`有一个`final`方法`getFinalZoneId()`，这会导致其他Bean如果调用此方法，无法将其代理到真正的原始实例，从而可能发生NPE异常。

因此，正确使用AOP，我们需要一个避坑指南：

1. 访问被注入的Bean时，总是调用方法而非直接访问字段；
2. 编写Bean时，如果可能会被代理，就不要编写`public final`方法。

这样才能保证有没有AOP，代码都能正常工作。

思考

为什么Spring刻意不初始化Proxy继承的字段？

如果一个Bean不允许任何AOP代理，应该怎么做来“保护”自己在运行期不会被代理？

练习

修复启用AOP导致的NPE

小结

由于Spring通过CGLIB实现代理类，我们要避免直接访问Bean的字段，以及由`final`方法带来的“未代理”问题。

遇到CglibAopProxy的相关日志，务必要仔细检查，防止因为AOP出现NPE异常。

数据库基本上是现代应用程序的标准存储，绝大多数程序都把自己的业务数据存储在关系数据库中，可见，访问数据库几乎是所有应用程序必备能力。

我们在前面已经介绍了Java程序访问数据库的标准接口JDBC，它的实现方式非常简洁，即：Java标准库定义接口，各数据库厂商以“驱动”的形式实现接口。应用程序要使用哪个数据库，就把该数据库厂商的驱动以jar包形式引入进来，同时自身仅使用JDBC接口，编译期并不需要特定厂商的驱动。

使用JDBC虽然简单，但代码比较繁琐。Spring为了简化数据库访问，主要做了以下几点工作：

- 提供了简化的访问JDBC的模板类，不必手动释放资源；
- 提供了一个统一的DAO类以实现Data Access Object模式；
- 把`SQLException`封装为`DataAccessException`，这个异常是一个`RuntimeException`，并且让我们能区分SQL异常的原因，例如，`DuplicateKeyException`表示违反了一个唯一约束；
- 能方便地集成Hibernate、JPA和MyBatis这些数据库访问框架。

本章我们将详细讲解在Spring中访问数据库的最佳实践。

我们在前面介绍JDBC编程时已经讲过，Java程序使用JDBC接口访问关系数据库的时候，需要以下几步：

- 创建全局`DataSource`实例，表示数据库连接池；
- 在需要读写数据库的方法内部，按如下步骤访问数据库：
  - 从全局`DataSource`实例获取`Connection`实例；
  - 通过`Connection`实例创建`PreparedStatement`实例；
  - 执行SQL语句，如果是查询，则通过`ResultSet`读取结果集，如果是修改，则获得`int`结果。

正确编写JDBC代码的关键是使用`try ... finally`释放资源，涉及到事务的代码需要正确提交或回滚事务。

在Spring使用JDBC，首先我们通过IoC容器创建并管理一个`DataSource`实例，然后，Spring提供了一个`JdbcTemplate`，可以方便地让我们操作JDBC，因此，通常情况下，我们会实例化一个`JdbcTemplate`。顾名思义，这个类主要使用了Template模式。

编写示例代码或者测试代码时，我们强烈推荐使用HSQLDB这个数据库，它是一个用Java编写的关系数据库，可以以内存模式或者文件模式运行，本身只有一个jar包，非常适合演示代码或者测试代码。

我们以实际工程为例，先创建Maven工程`spring-data-jdbc`，然后引入以下依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-context</artifactId>
        <version>5.2.0.RELEASE</version>
    </dependency>
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-jdbc</artifactId>
        <version>5.2.0.RELEASE</version>
    </dependency>
```

```
    <dependency>
        <groupId>javax.annotation</groupId>
        <artifactId>javax.annotation-api</artifactId>
        <version>1.3.2</version>
    </dependency>
    <dependency>
        <groupId>com.zaxxer</groupId>
        <artifactId>HikariCP</artifactId>
        <version>3.4.2</version>
    </dependency>
    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.5.0</version>
    </dependency>
</dependencies>
```

在AppConfig中，我们需要创建以下几个必须的Bean：

```
@Configuration
@ComponentScan
@PropertySource("jdbc.properties")
public class AppConfig {

    @Value("${jdbc.url}")
    String jdbcUrl;

    @Value("${jdbc.username}")
    String jdbcUsername;

    @Value("${jdbc.password}")
    String jdbcPassword;

    @Bean
    DataSource createDataSource() {
        HikariConfig config = new HikariConfig();
        config.setJdbcUrl(jdbcUrl);
        config.setUsername(jdbcUsername);
        config.setPassword(jdbcPassword);
        config.addDataSourceProperty("autoCommit", "true");
        config.addDataSourceProperty("connectionTimeout", "5");
        config.addDataSourceProperty("idleTimeout", "60");
        return new HikariDataSource(config);
    }

    @Bean
    JdbcTemplate createJdbcTemplate(@Autowired DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

在上述配置中：

1. 通过`@PropertySource("jdbc.properties")`读取数据库配置文件；
2. 通过`@Value("${jdbc.url}")`注入配置文件的相关配置；
3. 创建一个DataSource实例，它的实际类型是`HikariDataSource`，创建时需要用到注入的配置；
4. 创建一个JdbcTemplate实例，它需要注入`DataSource`，这是通过方法参数完成注入的。

最后，针对HSQLDB写一个配置文件`jdbc.properties`：

```
# 数据库文件名为testdb:
jdbc.url=jdbc:hsqldb:file:testdb

# Hsqldb默认的用户名是sa，口令是空字符串:
jdbc.username=sa
jdbc.password=
```

可以通过HSQLDB自带的工具来初始化数据库表，这里我们写一个Bean，在Spring容器启动时自动创建一个`users`表：

```
@Component
public class DatabaseInitializer {
    @Autowired
    JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        jdbcTemplate.update("CREATE TABLE IF NOT EXISTS users (" //
                + "id BIGINT IDENTITY NOT NULL PRIMARY KEY, " //
                + "email VARCHAR(100) NOT NULL, " //
                + "password VARCHAR(100) NOT NULL, " //
                + "name VARCHAR(100) NOT NULL, " //
                + "UNIQUE (email))");
    }
}
```

现在，所有准备工作都已完毕。我们只需要在需要访问数据库的Bean中，注入`JdbcTemplate`即可：

```
@Component
public class UserService {
    @Autowired
    JdbcTemplate jdbcTemplate;
    ...
}
```

JdbcTemplate用法

Spring提供的`JdbcTemplate`采用Template模式，提供了一系列以回调为特点的工具方法，目的是避免繁琐的`try...catch`语句。

我们以具体的示例来说明JdbcTemplate的用法。

首先我们看`T execute(ConnectionCallback<T> action)`方法，它提供了Jdbc的`Connection`供我们使用：

```
public User getUserById(long id) {
    // 注意传入的是ConnectionCallback:
    return jdbcTemplate.execute((Connection conn) -> {
        // 可以直接使用conn实例，不要释放它，回调结束后JdbcTemplate自动释放:
        // 在内部手动创建的PreparedStatement、ResultSet必须用try(...)释放:
        try (var ps = conn.prepareStatement("SELECT * FROM users WHERE id = ?")) {
            ps.setObject(1, id);
            try (var rs = ps.executeQuery()) {
                if (rs.next()) {
                    return new User( // new User object:
                            rs.getLong("id"), // id
                            rs.getString("email"), // email
                            rs.getString("password"), // password
                            rs.getString("name")); // name
                }
                throw new RuntimeException("user not found by id.");
```

```
            }
        }
    });
}
```

也就是说，上述回调方法允许获取Connection，然后做任何基于Connection的操作。

我们再看T execute(String sql, PreparedStatementCallback<T> action)的用法：

```
public User getUserByName(String name) {
    // 需要传入SQL语句，以及PreparedStatementCallback:
    return jdbcTemplate.execute("SELECT * FROM users WHERE name = ?", (PreparedStatement ps) -> {
        // PreparedStatement实例已经由JdbcTemplate创建，并在回调后自动释放:
        ps.setObject(1, name);
        try (var rs = ps.executeQuery()) {
            if (rs.next()) {
                return new User( // new User object:
                        rs.getLong("id"), // id
                        rs.getString("email"), // email
                        rs.getString("password"), // password
                        rs.getString("name")); // name
            }
            throw new RuntimeException("user not found by id.");
        }
    });
}
```

最后，我们看T queryForObject(String sql, Object[] args, RowMapper<T> rowMapper)方法：

```
public User getUserByEmail(String email) {
    // 传入SQL，参数和RowMapper实例:
    return jdbcTemplate.queryForObject("SELECT * FROM users WHERE email = ?", new Object[] { email },
            (ResultSet rs, int rowNum) -> {
                // 将ResultSet的当前行映射为一个JavaBean:
                return new User( // new User object:
                        rs.getLong("id"), // id
                        rs.getString("email"), // email
                        rs.getString("password"), // password
                        rs.getString("name")); // name
            });
}
```

在queryForObject()方法中，传入SQL以及SQL参数后，JdbcTemplate会自动创建PreparedStatement，自动执行查询并返回ResultSet，我们提供的RowMapper需要做的事情就是把ResultSet的当前行映射成一个JavaBean并返回。整个过程中，使用Connection、PreparedStatement和ResultSet都不需要我们手动管理。

RowMapper不一定返回JavaBean，实际上它可以返回任何Java对象。例如，使用SELECT COUNT(*)查询时，可以返回Long：

```
public long getUsers() {
    return jdbcTemplate.queryForObject("SELECT COUNT(*) FROM users", null, (ResultSet rs, int rowNum) -> {
        // SELECT COUNT(*)查询只有一列，取第一列数据:
        return rs.getLong(1);
    });
}
```

如果我们期望返回多行记录，而不是一行，可以用query()方法：

```
public List<User> getUsers(int pageIndex) {
    int limit = 100;
    int offset = limit * (pageIndex - 1);
    return jdbcTemplate.query("SELECT * FROM users LIMIT ? OFFSET ?", new Object[] { limit, offset },
            new BeanPropertyRowMapper<>(User.class));
}
```

上述query()方法传入的参数仍然是SQL、SQL参数以及RowMapper实例。这里我们直接使用Spring提供的BeanPropertyRowMapper。如果数据库表的结构恰好和JavaBean的属性名称一致，那么BeanPropertyRowMapper就可以直接把一行记录按列名转换为JavaBean。

如果我们执行的不是查询，而是插入、更新和删除操作，那么需要使用update()方法：

```
public void updateUser(User user) {
    // 传入SQL，SQL参数，返回更新的行数:
    if (1 != jdbcTemplate.update("UPDATE user SET name = ? WHERE id=?", user.getName(), user.getId())) {
        throw new RuntimeException("User not found by id");
    }
}
```

只有一种INSERT操作比较特殊，那就是如果某一列是自增列（例如自增主键），通常，我们需要获取插入后的自增值。JdbcTemplate提供了一个KeyHolder来简化这一操作：

```
public User register(String email, String password, String name) {
    // 创建一个KeyHolder:
    KeyHolder holder = new GeneratedKeyHolder();
    if (1 != jdbcTemplate.update(
        // 参数1:PreparedStatementCreator
        (conn) -> {
            // 创建PreparedStatement时，必须指定RETURN_GENERATED_KEYS:
            var ps = conn.prepareStatement("INSERT INTO users(email,password,name) VALUES(?,?,?)",
                    Statement.RETURN_GENERATED_KEYS);
            ps.setObject(1, email);
            ps.setObject(2, password);
            ps.setObject(3, name);
            return ps;
        },
        // 参数2:KeyHolder
        holder)
    ) {
        throw new RuntimeException("Insert failed.");
    }
    // 从KeyHolder中获取返回的自增值:
    return new User(holder.getKey().longValue(), email, password, name);
}
```

JdbcTemplate还有许多重载方法，这里我们不一一介绍。需要强调的是，JdbcTemplate只是对JDBC操作的一个简单封装，它的目的是尽量减少手动编写try(resource) {...}的代码，对于查询，主要通过RowMapper实现了JDBC结果集到Java对象的转换。

我们总结一下JdbcTemplate的用法，那就是：

- 针对简单查询，优选query()和queryForObject()，因为只需提供SQL语句、参数和RowMapper；
- 针对更新操作，优选update()，因为只需提供SQL语句和参数；
- 任何复杂的操作，最终也可以通过execute(ConnectionCallback)实现，因为拿到Connection就可以做任何JDBC操作。

实际上我们使用最多的仍然是各种查询。如果在设计表结构的时候，能够和JavaBean的属性一一对应，那么直接使用BeanPropertyRowMapper就很方便。如果表结构和JavaBean不一致怎么办？那就需要稍微改写一下查询，使结果集的结构和JavaBean保持一致。

例如，表的列名是office_address，而JavaBean属性是workAddress，就需要指定别名，改写查询如下：

```
SELECT id, email, office_address AS workAddress, name FROM users WHERE email = ?
```

练习

使用JdbcTemplate

小结

Spring提供了JdbcTemplate来简化JDBC操作：

使用JdbcTemplate时，根据需要优先选择高级方法；

任何JDBC操作都可以使用保底的execute(ConnectionCallback)方法。

使用Spring操作JDBC虽然方便，但是我们在前面讨论JDBC的时候，讲到过JDBC事务，如果要在Spring中操作事务，没必要手写JDBC事务，可以使用Spring提供的高级接口来操作事务。

Spring提供了一个PlatformTransactionManager来表示事务管理器，所有的事务都由它负责管理。而事务由TransactionStatus表示。如果手写事务代码，使用try...catch如下：

```
TransactionStatus tx = null;
try {
    // 开启事务:
    tx = txManager.getTransaction(new DefaultTransactionDefinition());
    // 相关JDBC操作:
    jdbcTemplate.update("...");
    jdbcTemplate.update("...");
    // 提交事务:
    txManager.commit(tx);
} catch (RuntimeException e) {
    // 回滚事务:
    txManager.rollback(tx);
    throw e;
}
```

Spring为啥要抽象出PlatformTransactionManager和TransactionStatus？原因是JavaEE除了提供JDBC事务外，它还支持分布式事务JTA（Java Transaction API）。分布式事务是指多个数据源（比如多个数据库，多个消息系统）要在分布式环境下实现事务的时候，应该怎么实现。分布式事务实现起来非常复杂，简单地说就是通过一个分布式事务管理器实现两阶段提交，但本身数据库事务就不快，基于数据库事务实现的分布式事务就慢得难以忍受，所以使用率不高。

Spring为了同时支持JDBC和JTA两种事务模型，就抽象出PlatformTransactionManager。因为我们的代码只需要JDBC事务，因此，在AppConfig中，需要再定义一个PlatformTransactionManager对应的Bean，它的实际类型是DataSourceTransactionManager：

```
@Configuration
@ComponentScan
@PropertySource("jdbc.properties")
public class AppConfig {
    ...
    @Bean
    PlatformTransactionManager createTxManager(@Autowired DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

使用编程的方式使用Spring事务仍然比较繁琐，更好的方式是通过声明式事务来实现。使用声明式事务非常简单，除了在AppConfig中追加一个上述定义的PlatformTransactionManager外，再加一个@EnableTransactionManagement就可以启用声明式事务：

```
@Configuration
@ComponentScan
@EnableTransactionManagement // 启用声明式
@PropertySource("jdbc.properties")
public class AppConfig {
    ...
}
```

然后，对需要事务支持的方法，加一个@Transactional注解：

```
@Component
public class UserService {
    // 此public方法自动具有事务支持:
    @Transactional
    public User register(String email, String password, String name) {
       ...
    }
}
```

或者更简单一点，直接在Bean的class处加上，表示所有public方法都具有事务支持：

```
@Component
@Transactional
public class UserService {
    ...
}
```

Spring对一个声明式事务的方法，如何开启事务支持？原理仍然是AOP代理，即通过自动创建Bean的Proxy实现：

```
public class UserService$$EnhancerBySpringCGLIB extends UserService {
    UserService target = ...
    PlatformTransactionManager txManager = ...

    public User register(String email, String password, String name) {
        TransactionStatus tx = null;
        try {
            tx = txManager.getTransaction(new DefaultTransactionDefinition());
            target.register(email, password, name);
            txManager.commit(tx);
        } catch (RuntimeException e) {
            txManager.rollback(tx);
            throw e;
        }
    }
    ...
}
```

注意：声明了@EnableTransactionManagement后，不必额外添加@EnableAspectJAutoProxy。

回滚事务

默认情况下，如果发生了RuntimeException，Spring的声明式事务将自动回滚。在一个事务方法中，如果程序判断需要回滚事务，只需抛出RuntimeException，例如：

```
@Transactional
public buyProducts(long productId, int num) {
    ...
    if (store < num) {
        // 库存不够，购买失败:
        throw new IllegalArgumentException("No enough products");
    }
    ...
}
```

如果要针对Checked Exception回滚事务，需要在@Transactional注解中写出来：

```
@Transactional(rollbackFor = {RuntimeException.class, IOException.class})
public buyProducts(long productId, int num) throws IOException {
    ...
}
```

上述代码表示在抛出RuntimeException或IOException时，事务将回滚。

为了简化代码，我们强烈建议业务异常体系从RuntimeException派生，这样就不必声明任何特殊异常即可让Spring的声明式事务正常工作：

```
public class BusinessException extends RuntimeException {
    ...
}

public class LoginException extends BusinessException {
    ...
}

public class PaymentException extends BusinessException {
    ...
}
```

事务边界

在使用事务的时候，明确事务边界非常重要。对于声明式事务，例如，下面的register()方法：

```
@Component
public class UserService {
    @Transactional
    public User register(String email, String password, String name) { // 事务开始
        ...
    } // 事务结束
}
```

它的事务边界就是register()方法开始和结束。

类似的，一个负责给用户增加积分的addBonus()方法：

```
@Component
public class BonusService {
    @Transactional
    public void addBonus(long userId, int bonus) { // 事务开始
        ...
    } // 事务结束
}
```

它的事务边界就是addBonus()方法开始和结束。

在现实世界中，问题总是要复杂一点点。用户注册后，能自动获得100积分，因此，实际代码如下：

```
@Component
public class UserService {
    @Autowired
    BonusService bonusService;

    @Transactional
    public User register(String email, String password, String name) {
        // 插入用户记录:
        User user = jdbcTemplate.insert("...");
        // 增加100积分:
        bonusService.addBonus(user.id, 100);
    }
}
```

现在问题来了：调用方（比如RegisterController）调用UserService.register()这个事务方法，它在内部又调用了BonusService.addBonus()这个事务方法，一共有几个事务？如果addBonus()抛出了异常需要回滚事务，register()方法的事务是否也要回滚？

问题的复杂度是不是一下子提高了10倍？

事务传播

要解决上面的问题，我们首先要定义事务的传播模型。

假设用户注册的入口是RegisterController，它本身没有事务，仅仅是调用UserService.register()这个事务方法：

```
@Controller
public class RegisterController {
    @Autowired
    UserService userService;

    @PostMapping("/register")
    public ModelAndView doRegister(HttpServletRequest req) {
        String email = req.getParameter("email");
        String password = req.getParameter("password");
        String name = req.getParameter("name");
        User user = userService.register(email, password, name);
        return ...
    }
}
```

因此，UserService.register()这个事务方法的起始和结束，就是事务的范围。

我们需要关心的问题是，在UserService.register()这个事务方法内，调用BonusService.addBonus()，我们期待的事务行为是什么：

```
@Transactional
public User register(String email, String password, String name) {
    // 事务已开启:
    User user = jdbcTemplate.insert("...");
    // ???:
    bonusService.addBonus(user.id, 100);
} // 事务结束
```

对于大多数业务来说，我们期待BonusService.addBonus()的调用，和UserService.register()应当融合在一起，它的行为应该如下：

UserService.register()已经开启了一个事务，那么在内部调用BonusService.addBonus()时，BonusService.addBonus()方法就没必要再开启一个新事务，直接加入到BonusService.register()的事务里就好了。

其实就相当于：

1. UserService.register()先执行了一条INSERT语句：INSERT INTO users ...
2. BonusService.addBonus()再执行一条INSERT语句：INSERT INTO bonus ...

因此，Spring的声明式事务为事务传播定义了几个级别，默认传播级别就是REQUIRED，它的意思是，如果当前没有事务，就创建一个新事务，如果当前有事务，就加入到当前事务中执行。

我们观察UserService.register()方法，它在RegisterController中执行，因为RegisterController没有事务，因此，UserService.register()方法会自动创建一个新事务。

在UserService.register()方法内部，调用BonusService.addBonus()方法时，因为BonusService.addBonus()检测到当前已经有事务了，因此，它会加入到当前事务中执行。

因此，整个业务流程的事务边界就清晰了：它只有一个事务，并且范围就是UserService.register()方法。

有的童鞋会问：把BonusService.addBonus()方法的@Transactional去掉，变成一个普通方法，那不就规避了复杂的传播模型吗？

去掉BonusService.addBonus()方法的@Transactional，会引来另一个问题，即其他地方如果调用BonusService.addBonus()方法，那就没法保证事务了。例如，规定用户登录时积分+5：

```
@Controller
public class LoginController {
    @Autowired
    BonusService bonusService;

    @PostMapping("/login")
    public ModelAndView doLogin(HttpServletRequest req) {
        User user = ...
        bonusService.addBonus(user.id, 5);
    }
}
```

可见，BonusService.addBonus()方法必须要有@Transactional，否则，登录后积分就无法添加了。

默认的事务传播级别是REQUIRED，它满足绝大部分的需求。还有一些其他的传播级别：

SUPPORTS：表示如果有事务，就加入到当前事务，如果没有，那也不开启事务执行。这种传播级别可用于查询方法，因为SELECT语句既可以在事务内执行，也可以不需要事务；

MANDATORY：表示必须要存在当前事务并加入执行，否则将抛出异常。这种传播级别可用于核心更新逻辑，比如用户余额变更，它总是被其他事务方法调用，不能直接由非事务方法调用；

REQUIRES_NEW：表示不管当前有没有事务，都必须开启一个新的事务执行。如果当前已经有事务，那么当前事务会挂起，等新事务完成后，再恢复执行；

NOT_SUPPORTED：表示不支持事务，如果当前有事务，那么当前事务会挂起，等这个方法执行完成后，再恢复执行；

NEVER：和NOT_SUPPORTED相比，它不但不支持事务，而且在监测到当前有事务时，会抛出异常拒绝执行；

NESTED：表示如果当前有事务，则开启一个嵌套级别事务，如果当前没有事务，则开启一个新事务。

上面这么多种事务的传播级别，其实默认的REQUIRED已经满足绝大部分需求，SUPPORTS和REQUIRES_NEW在少数情况下会用到，其他基本不会用到，因为把事务搞得越复杂，不仅逻辑跟着复杂，而且速度也会越慢。

定义事务的传播级别也是写在@Transactional注解里的：

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public Product createProduct() {
    ...
}
```

现在只剩最后一个问题了：Spring是如何传播事务的？

我们在JDBC中使用事务的时候，是这么个写法：

```
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();
}
```

Spring使用声明式事务，最终也是通过执行JDBC事务来实现功能的，那么，一个事务方法，如何获知当前是否存在事务？

答案是使用ThreadLocal。Spring总是把JDBC相关的Connection和TransactionStatus实例绑定到ThreadLocal。如果一个事务方法从ThreadLocal未取到事务，那么它会打开一个新的JDBC连接，同时开启一个新的事务，否则，它就直接使用从ThreadLocal获取的JDBC连接以及TransactionStatus。

因此，事务能正确传播的前提是，方法调用是在一个线程内才行。如果像下面这样写：

```
@Transactional
public User register(String email, String password, String name) { // BEGIN TX-A
    User user = jdbcTemplate.insert("...");
    new Thread(() -> {
        // BEGIN TX-B:
        bonusService.addBonus(user.id, 100);
        // END TX-B
    }).start();
} // END TX-A
```

在另一个线程中调用BonusService.addBonus()，它根本获取不到当前事务，因此，UserService.register()和BonusService.addBonus()两个方法，将分别开启两个完全独立的事务。

换句话说，事务只能在当前线程传播，无法跨线程传播。

那如果我们想实现跨线程传播事务呢？原理很简单，就是要想办法把当前线程绑定到ThreadLocal的Connection和TransactionStatus实例传递给新线程，但实现起来非常复杂，根据异常回滚更加复杂，不推荐自己去实现。

练习

使用声明式事务

小结

Spring提供的声明式事务极大地方便了在数据库中使用事务，正确使用声明式事务的关键在于确定好事务边界，理解事务传播级别。

在传统的多层应用程序中，通常是Web层调用业务层，业务层调用数据访问层。业务层负责处理各种业务逻辑，而数据访问层只负责对数据进行增删改查。因此，实现数据访问层就是用JdbcTemplate实现对数据库的操作。

编写数据访问层的时候，可以使用DAO模式。DAO即Data Access Object的缩写，它没有什么神秘之处，实现起来基本如下：

```
public class UserDao {

    @Autowired
    JdbcTemplate jdbcTemplate;

    User getById(long id) {
        ...
    }

    List<User> getUsers(int page) {
```

```
        ...
    }

    User createUser(User user) {
        ...
    }

    User updateUser(User user) {
        ...
    }

    void deleteUser(User user) {
        ...
    }
}
```

Spring提供了一个JdbcDaoSupport类，用于简化DAO的实现。这个JdbcDaoSupport没什么复杂的，核心代码就是持有一个JdbcTemplate：

```
public abstract class JdbcDaoSupport extends DaoSupport {

    private JdbcTemplate jdbcTemplate;

    public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        initTemplateConfig();
    }

    public final JdbcTemplate getJdbcTemplate() {
        return this.jdbcTemplate;
    }

    ...
}
```

它的意图是子类直接从JdbcDaoSupport继承后，可以随时调用getJdbcTemplate()获得JdbcTemplate的实例。那么问题来了：因为JdbcDaoSupport的jdbcTemplate字段没有标记@Autowired，所以，子类想要注入JdbcTemplate，还得自己想个办法：

```
@Component
@Transactional
public class UserDao extends JdbcDaoSupport {
    @Autowired
    JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        super.setJdbcTemplate(jdbcTemplate);
    }
}
```

有的童鞋可能看出来了：既然UserDao都已经注入了JdbcTemplate，那再把它放到父类里，通过getJdbcTemplate()访问岂不是多此一举？

如果使用传统的XML配置，并不需要编写@Autowired JdbcTemplate jdbcTemplate，但是考虑到现在基本上是使用注解的方式，我们可以编写一个AbstractDao，专门负责注入JdbcTemplate：

```
public abstract class AbstractDao extends JdbcDaoSupport {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @PostConstruct
    public void init() {
        super.setJdbcTemplate(jdbcTemplate);
    }
}
```

这样，子类的代码就非常干净，可以直接调用getJdbcTemplate()：

```
@Component
@Transactional
public class UserDao extends AbstractDao {
    public User getById(long id) {
        return getJdbcTemplate().queryForObject(
                "SELECT * FROM users WHERE id = ?",
                new BeanPropertyRowMapper<>(User.class),
                id
        );
    }
    ...
}
```

倘若肯再多写一点样板代码，就可以把AbstractDao改成泛型，并实现getById()，getAll()，deleteById()这样的通用方法：

```
public abstract class AbstractDao<T> extends JdbcDaoSupport {
    private String table;
    private Class<T> entityClass;
    private RowMapper<T> rowMapper;

    public AbstractDao() {
        // 获取当前类型的泛型类型:
        this.entityClass = getParameterizedType();
        this.table = this.entityClass.getSimpleName().toLowerCase() + "s";
        this.rowMapper = new BeanPropertyRowMapper<>(entityClass);
    }

    public T getById(long id) {
        return getJdbcTemplate().queryForObject("SELECT * FROM " + table + " WHERE id = ?", this.rowMapper, id);
    }

    public List<T> getAll(int pageIndex) {
        int limit = 100;
        int offset = limit * (pageIndex - 1);
        return getJdbcTemplate().query("SELECT * FROM " + table + " LIMIT ? OFFSET ?",
                new Object[] { limit, offset },
                this.rowMapper);
    }

    public void deleteById(long id) {
        getJdbcTemplate().update("DELETE FROM " + table + " WHERE id = ?", id);
    }
    ...
}
```

这样，每个子类就自动获得了这些通用方法：

```
@Component
@Transactional
public class UserDao extends AbstractDao<User> {
    // 已经有了:
    // User getById(long)
    // List<User> getAll(int)
    // void deleteById(long)
}
```

```
@Component
@Transactional
public class BookDao extends AbstractDao<Book> {
    // 已经有了:
    // Book getById(long)
    // List<Book> getAll(int)
    // void deleteById(long)
}
```

可见，DAO模式就是一个简单的数据访问模式，是否使用DAO，根据实际情况决定，因为很多时候，直接在Service层操作数据库也是完全没有问题的。

练习

使用DAO模式

小结

Spring提供了`JdbcDaoSupport`来便于我们实现DAO模式；

可以基于泛型实现更通用、更简洁的DAO模式。

使用`JdbcTemplate`的时候，我们用得最多的方法就是`List<T> query(String sql, Object[] args, RowMapper rowMapper)`。这个`RowMapper`的作用就是把`ResultSet`的一行记录映射为Java Bean。

这种把关系数据库的表记录映射为Java对象的过程就是ORM：Object-Relational Mapping。ORM既可以把记录转换成Java对象，也可以把Java对象转换为行记录。

使用`JdbcTemplate`配合`RowMapper`可以看作是最原始的ORM。如果要实现更自动化的ORM，可以选择成熟的ORM框架，例如Hibernate。

我们来看看如何在Spring中集成Hibernate。

Hibernate作为ORM框架，它可以替代`JdbcTemplate`，但Hibernate仍然需要JDBC驱动，所以，我们需要引入JDBC驱动、连接池，以及Hibernate本身。在Maven中，我们加入以下依赖项：

```
<!-- JDBC驱动，这里使用HSQLDB -->
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.5.0</version>
</dependency>

<!-- JDBC连接池 -->
<dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>3.4.2</version>
</dependency>

<!-- Hibernate -->
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-core</artifactId>
    <version>5.4.2.Final</version>
</dependency>

<!-- Spring Context和Spring ORM -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-orm</artifactId>
    <version>5.2.0.RELEASE</version>
</dependency>
```

在AppConfig中，我们仍然需要创建DataSource、引入JDBC配置文件，以及启用声明式事务：

```
@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() {
        ...
    }
}
```

为了启用Hibernate，我们需要创建一个`LocalSessionFactoryBean`：

```
public class AppConfig {
    @Bean
    LocalSessionFactoryBean createSessionFactory(@Autowired DataSource dataSource) {
        var props = new Properties();
        props.setProperty("hibernate.hbm2ddl.auto", "update"); // 生产环境不要使用
        props.setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect");
        props.setProperty("hibernate.show_sql", "true");
        var sessionFactoryBean = new LocalSessionFactoryBean();
        sessionFactoryBean.setDataSource(dataSource);
        // 扫描指定的package获取所有entity class:
        sessionFactoryBean.setPackagesToScan("com.itranswarp.learnjava.entity");
        sessionFactoryBean.setHibernateProperties(props);
        return sessionFactoryBean;
    }
}
```

注意我们在定制Bean中讲到过`FactoryBean`，`LocalSessionFactoryBean`是一个`FactoryBean`，它会再自动创建一个`SessionFactory`，在Hibernate中，`Session`是封装了一个JDBC `Connection`的实例，而`SessionFactory`是封装了JDBC `DataSource`的实例，即`SessionFactory`持有连接池，每次需要操作数据库的时候，`SessionFactory`创建一个新的`Session`，相当于从连接池获取到一个新的`Connection`。`SessionFactory`就是Hibernate提供的最核心的一个对象，但`LocalSessionFactoryBean`是Spring提供的为了让我们方便创建`SessionFactory`的类。

注意到上面创建`LocalSessionFactoryBean`的代码，首先用`Properties`持有Hibernate初始化`SessionFactory`时用到的所有设置，常用的设置请参考Hibernate文档，这里我们只定义了3个设置：

- `hibernate.hbm2ddl.auto=update`：表示自动创建数据库的表结构，注意不要在生产环境中启用；
- `hibernate.dialect=org.hibernate.dialect.HSQLDialect`：指示Hibernate使用的数据库是HSQLDB。Hibernate使用一种HQL的查询语句，它和SQL类似，但真正在“翻译”成SQL时，会根据设定的数据库“方言”来生成针对数据库优化的SQL；
- `hibernate.show_sql=true`：让Hibernate打印执行的SQL，这对于调试非常有用，我们可以方便地看到Hibernate生成的SQL语句是否符合我们的预期。

除了设置`DataSource`和`Properties`之外，注意到`setPackagesToScan()`我们传入了一个`package`名称，它指示Hibernate扫描这个包下面的所有Java类，自动找出能映射为数据库表记录的JavaBean。后面我们会仔细讨论如何编写符合Hibernate要求的JavaBean。

紧接着，我们还需要创建`HibernateTemplate`以及`HibernateTransactionManager`：

```
public class AppConfig {
    @Bean
    HibernateTemplate createHibernateTemplate(@Autowired SessionFactory sessionFactory) {
```

```
        return new HibernateTemplate(sessionFactory);
    }

    @Bean
    PlatformTransactionManager createTxManager(@Autowired SessionFactory sessionFactory) {
        return new HibernateTransactionManager(sessionFactory);
    }
}
```

这两个Bean的创建都十分简单。HibernateTransactionManager是配合Hibernate使用声明式事务所必须的，而HibernateTemplate则是Spring为了便于我们使用Hibernate提供的工具类，不是非用不可，但推荐使用以简化代码。

到此为止，所有的配置都定义完毕，我们来看看如何将数据库表结构映射为Java对象。

考察如下的数据库表：

```
CREATE TABLE user
    id BIGINT NOT NULL AUTO_INCREMENT,
    email VARCHAR(100) NOT NULL,
    password VARCHAR(100) NOT NULL,
    name VARCHAR(100) NOT NULL,
    createdAt BIGINT NOT NULL,
    PRIMARY KEY (`id`),
    UNIQUE KEY `email` (`email`)
);
```

其中，id是自增主键，email、password、name是VARCHAR类型，email带唯一索引以确保唯一性，createdAt存储整型类型的时间戳。用JavaBean表示如下：

```
public class User {
    private Long id;
    private String email;
    private String password;
    private String name;
    private Long createdAt;

    // getters and setters
    ...
}
```

这种映射关系十分易懂，但我们需要添加一些注解来告诉Hibernate如何把User类映射到表记录：

```
@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, updatable = false)
    public Long getId() { ... }

    @Column(nullable = false, unique = true, length = 100)
    public String getEmail() { ... }

    @Column(nullable = false, length = 100)
    public String getPassword() { ... }

    @Column(nullable = false, length = 100)
    public String getName() { ... }

    @Column(nullable = false, updatable = false)
    public Long getCreatedAt() { ... }
}
```

如果一个JavaBean被用于映射，我们就标记一个@Entity。默认情况下，映射的表名是user，如果实际的表名不同，例如实际表名是users，可以追加一个@Table(name="users")表示：

```
@Entity
@Table(name="users)
public class User {
    ...
}
```

每个属性到数据库列的映射用@Column()标识，nullable指示列是否允许为NULL，updatable指示该列是否允许被用在UPDATE语句，length指示String类型的列的长度（如果没有指定，默认是255）。

对于主键，还需要用@Id标识，自增主键再追加一个@GeneratedValue，以便Hibernate能读取到自增主键的值。

细心的童鞋可能还注意到，主键id定义的类型不是long，而是Long。这是因为Hibernate如果检测到主键为null，就不会在INSERT语句中指定主键的值，而是返回由数据库生成的自增值，否则，Hibernate认为我们的程序指定了主键的值，会在INSERT语句中直接列出。long型字段总是具有默认值0，因此，每次插入的主键值总是0，导致除第一次外后续插入都将失败。

createdAt虽然是整型，但我们并没有使用long，而是Long，这是因为使用基本类型会导致某种查询会添加意外的条件，后面我们会详细讨论，这里只需牢记，作为映射使用的JavaBean，所有属性都使用包装类型而不是基本类型。

使用Hibernate时，不要使用基本类型的属性，总是使用包装类型，如Long或Integer。

类似的，我们再定义一个Book类：

```
@Entity
public class Book {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, updatable = false)
    public Long getId() { ... }

    @Column(nullable = false, length = 100)
    public String getTitle() { ... }

    @Column(nullable = false, updatable = false)
    public Long getCreatedAt() { ... }
}
```

如果仔细观察User和Book，会发现它们定义的id、createdAt属性是一样的，这在数据库表结构的设计中很常见：对于每个表，通常我们会统一使用一种主键生成机制，并添加createdAt表示创建时间，updatedAt表示修改时间等通用字段。

不必在User和Book中重复定义这些通用字段，我们可以把它们提到一个抽象类中：

```
@MappedSuperclass
public abstract class AbstractEntity {

    private Long id;
    private Long createdAt;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(nullable = false, updatable = false)
    public Long getId() { ... }

    @Column(nullable = false, updatable = false)
    public Long getCreatedAt() { ... }

    @Transient
    public ZonedDateTime getCreatedDateTime() {
```

```
        return Instant.ofEpochMilli(this.createdAt).atZone(ZoneId.systemDefault());
    }

    @PrePersist
    public void preInsert() {
        setCreatedAt(System.currentTimeMillis());
    }
}
```

对于AbstractEntity来说，我们要标注一个@MappedSuperclass表示它用于继承。此外，注意到我们定义了一个@Transient方法，它返回一个“虚拟”的属性。因为getCreatedDateTime()是计算得出的属性，而不是从数据库表读出的值，因此必须要标注@Transient，否则Hibernate会尝试从数据库读取名为createdDateTime这个不存在的字段从而出错。

再注意到@PrePersist标识的方法，它表示在我们将一个JavaBean持久化到数据库之前（即执行INSERT语句），Hibernate会先执行该方法，这样我们就可以自动设置好createdAt属性。

有了AbstractEntity，我们就可以大幅简化User和Book：

```
@Entity
public class User extends AbstractEntity {

    @Column(nullable = false, unique = true, length = 100)
    public String getEmail() { ... }

    @Column(nullable = false, length = 100)
    public String getPassword() { ... }

    @Column(nullable = false, length = 100)
    public String getName() { ... }
}
```

注意到使用的所有注解均来自javax.persistence，它是JPA规范的一部分。这里我们只介绍使用注解的方式配置Hibernate映射关系，不再介绍传统的比较繁琐的XML配置。通过Spring集成Hibernate时，也不再需要hibernate.cfg.xml配置文件，用一句话总结：

使用Spring集成Hibernate，配合JPA注解，无需任何额外的XML配置。

类似User、Book这样的用于ORM的Java Bean，我们通常称之为Entity Bean。

最后，我们来看看如果对user表进行增删改查。因为使用了Hibernate，因此，我们要做的，实际上是对User这个JavaBean进行“增删改查”。我们编写一个UserService，注入HibernateTemplate以便简化代码：

```
@Component
@Transactional
public class UserService {
    @Autowired
    HibernateTemplate hibernateTemplate;
}
```

Insert操作

要持久化一个User实例，我们只需调用save()方法。以register()方法为例，代码如下：

```
public User register(String email, String password, String name) {
    // 创建一个User对象:
    User user = new User();
    // 设置好各个属性:
    user.setEmail(email);
    user.setPassword(password);
    user.setName(name);
    // 不要设置id，因为使用了自增主键
    // 保存到数据库:
    hibernateTemplate.save(user);
    // 现在已经自动获得了id:
    System.out.println(user.getId());
    return user;
}
```

Delete操作

删除一个User相当于从表中删除对应的记录。注意Hibernate总是用id来删除记录，因此，要正确设置User的id属性才能正常删除记录：

```
public boolean deleteUser(Long id) {
    User user = hibernateTemplate.get(User.class, id);
    if (user != null) {
        hibernateTemplate.delete(user);
        return true;
    }
    return false;
}
```

通过主键删除记录时，一个常见的用法是先根据主键加载该记录，再删除。load()和get()都可以根据主键加载记录，它们的区别在于，当记录不存在时，get()返回null，而load()抛出异常。

Update操作

更新记录相当于先更新User的指定属性，然后调用update()方法：

```
public void updateUser(Long id, String name) {
    User user = hibernateTemplate.load(User.class, id);
    user.setName(name);
    hibernateTemplate.update(user);
}
```

前面我们在定义User时，对有的属性标注了@Column(updatable=false)。Hibernate在更新记录时，它只会把@Column(updatable=true)的属性加入到UPDATE语句中，这样可以提供一层额外的安全性，即如果不小心修改了User的email、createdAt等属性，执行update()时并不会更新对应的数据库列。但也必须牢记：这个功能是Hibernate提供的，如果绕过Hibernate直接通过JDBC执行UPDATE语句仍然可以更新数据库的任意列的值。

最后，我们编写的大部分方法都是各种各样的查询。根据id查询我们可以直接调用load()或get()，如果要使用条件查询，有3种方法。

假设我们想执行以下查询：

```
SELECT * FROM user WHERE email = ? AND password = ?
```

我们来看看可以使用什么查询。

使用Example查询

第一种方法是使用findByExample()，给出一个User实例，Hibernate把该实例所有非null的属性拼成WHERE条件：

```
public User login(String email, String password) {
    User example = new User();
    example.setEmail(email);
    example.setPassword(password);
    List<User> list = hibernateTemplate.findByExample(example);
    return list.isEmpty() ? null : list.get(0);
}
```

因为example实例只有email和password两个属性为非null，所以最终生成的WHERE语句就是WHERE email = ? AND password = ?。

如果我们把User的createdAt的类型从Long改为long，findByExample()的查询将出问题，原因在于example实例的long类型字段有了默认值0，导致Hibernate最终生成的WHERE语句意外变成了WHERE email = ? AND password = ? AND createdAt = 0。显然，额外的查询条件将导致错误的查询结果。

使用findByExample()时，注意基本类型字段总是会加入到WHERE条件！

使用Criteria查询

第二种查询方法是使用Criteria查询，可以实现如下：

```
public User login(String email, String password) {
    DetachedCriteria criteria = DetachedCriteria.forClass(User.class);
    criteria.add(Restrictions.eq("email", email))
            .add(Restrictions.eq("password", password));
    List<User> list = (List<User>) hibernateTemplate.findByCriteria(criteria);
    return list.isEmpty() ? null : list.get(0);
}
```

DetachedCriteria使用链式语句来添加多个AND条件。和findByExample()相比，findByCriteria()可以组装出更灵活的WHERE条件，例如：

```
SELECT * FROM user WHERE (email = ? OR name = ?) AND password = ?
```

上述查询没法用findByExample()实现，但用Criteria查询可以实现如下：

```
DetachedCriteria criteria = DetachedCriteria.forClass(User.class);
criteria.add(
    Restrictions.and(
        Restrictions.or(
            Restrictions.eq("email", email),
            Restrictions.eq("name", email)
        ),
  Restrictions.eq("password", password)
    )
);
```

只要组织好Restrictions的嵌套关系，Criteria查询可以实现任意复杂的查询。

使用HQL查询

最后一种常用的查询是直接编写Hibernate内置的HQL查询：

```
List<User> list = (List<User>) hibernateTemplate.find("FROM User WHERE email=? AND password=?", email, password);
```

和SQL相比，HQL使用类名和属性名，由Hibernate自动转换为实际的表名和列名。详细的HQL语法可以参考Hibernate文档。

除了可以直接传入HQL字符串外，Hibernate还可以使用一种NamedQuery，它给查询起个名字，然后保存在注解中。使用NamedQuery时，我们要先在User类标注：

```
@NamedQueries(
    @NamedQuery(
        // 查询名称:
        name = "login",
        // 查询语句:
        query = "SELECT u FROM User u WHERE u.email=?0 AND u.password=?1"
    )
)
@Entity
public class User extends AbstractEntity {
    ...
}
```

注意到引入的NamedQuery是javax.persistence.NamedQuery，它和直接传入HQL有点不同的是，占位符使用?0、?1，并且索引是从0开始的（真乱）。

使用NamedQuery只需要引入查询名和参数：

```
public User login(String email, String password) {
    List<User> list = (List<User>) hibernateTemplate.findByNamedQuery("login", email, password);
    return list.isEmpty() ? null : list.get(0);
}
```

直接写HQL和使用NamedQuery各有优劣。前者可以在代码中直观地看到查询语句，后者可以在User类统一管理所有相关查询。

使用Hibernate原生接口

如果要使用Hibernate原生接口，但不知道怎么写，可以参考HibernateTemplate的源码。使用Hibernate的原生接口实际上总是从SessionFactory出发，它通常用全局变量存储，在HibernateTemplate中以成员变量被注入。有了SessionFactory，使用Hibernate用法如下：

```
void operation() {
    Session session = null;
    boolean isNew = false;
    // 获取当前Session或者打开新的Session:
    try {
        session = this.sessionFactory.getCurrentSession();
    } catch (HibernateException e) {
        session = this.sessionFactory.openSession();
        isNew = true;
    }
    // 操作Session:
    try {
        User user = session.load(User.class, 123L);
    }
    finally {
        // 关闭新打开的Session:
        if (isNew) {
            session.close();
        }
    }
}
```

练习

集成Hibernate

小结

在Spring中集成Hibernate需要配置的Bean如下：

- DataSource；
- LocalSessionFactory；
- HibernateTransactionManager；
- HibernateTemplate（推荐）。

推荐使用Annotation配置所有的Entity Bean。

上一节我们讲了在Spring中集成Hibernate。Hibernate是第一个被广泛使用的ORM框架，但是很多小伙伴还听说过JPA：Java Persistence API，这又是啥？

在讨论JPA之前，我们要注意到JavaEE早在1999年就发布了，并且有Servlet、JMS等诸多标准。和其他平台不同，Java世界早期非常热衷于标准先行，各家跟进：大家先坐下来把接口定了，然后，各自回家干活去实现接口，这样，用户就可以在不同的厂家提供的产品进行选择，还可以随意切换，因为用户编写代码的时候只需要引用接口，并不需要引用具体的底层实现（想想JDBC）。

JPA就是JavaEE的一个ORM标准，它的实现其实和Hibernate没啥本质区别，但是用户如果使用JPA，那么引用的就是`javax.persistence`这个“标准”包，而不是`org.hibernate`这样的第三方包。因为JPA只是接口，所以，还需要选择一个实现产品，跟JDBC接口和MySQL驱动一个道理。

我们使用JPA时也完全可以选择Hibernate作为底层实现，但也可以选择其它的JPA提供方，比如EclipseLink。Spring内置了JPA的集成，并支持选择Hibernate或EclipseLink作为实现。这里我们仍然以主流的Hibernate作为JPA实现为例子，演示JPA的基本用法。

和使用Hibernate一样，我们只需要引入如下依赖：

- org.springframework:spring-context:5.2.0.RELEASE
- org.springframework:spring-orm:5.2.0.RELEASE
- javax.annotation:javax.annotation-api:1.3.2
- org.hibernate:hibernate-core:5.4.2.Final
- com.zaxxer:HikariCP:3.4.2
- org.hsqldb:hsqldb:2.5.0

然后，在`AppConfig`中启用声明式事务管理，创建`DataSource`：

```
@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() { ... }
}
```

使用Hibernate时，我们需要创建一个`LocalSessionFactoryBean`，并让它再自动创建一个`SessionFactory`。使用JPA也是类似的，我们需要创建一个`LocalContainerEntityManagerFactoryBean`，并让它再自动创建一个`EntityManagerFactory`：

```
@Bean
LocalContainerEntityManagerFactoryBean createEntityManagerFactory(@Autowired DataSource dataSource) {
    var entityManagerFactoryBean = new LocalContainerEntityManagerFactoryBean();
    // 设置DataSource:
    entityManagerFactoryBean.setDataSource(dataSource);
    // 扫描指定的package获取所有entity class:
    entityManagerFactoryBean.setPackagesToScan("com.itranswarp.learnjava.entity");
    // 指定JPA的提供商是Hibernate:
    JpaVendorAdapter vendorAdapter = new HibernateJpaVendorAdapter();
    entityManagerFactoryBean.setJpaVendorAdapter(vendorAdapter);
    // 设定特定提供商自己的配置:
    var props = new Properties();
    props.setProperty("hibernate.hbm2ddl.auto", "update");
    props.setProperty("hibernate.dialect", "org.hibernate.dialect.HSQLDialect");
    props.setProperty("hibernate.show_sql", "true");
    entityManagerFactoryBean.setJpaProperties(props);
    return entityManagerFactoryBean;
}
```

观察上述代码，除了需要注入`DataSource`和设定自动扫描的`package`外，还需要指定JPA的提供商，这里使用Spring提供的一个`HibernateJpaVendorAdapter`，最后，针对Hibernate自己需要的配置，以`Properties`的形式注入。

最后，我们还需要实例化一个`JpaTransactionManager`，以实现声明式事务：

```
@Bean
PlatformTransactionManager createTxManager(@Autowired EntityManagerFactory entityManagerFactory) {
    return new JpaTransactionManager(entityManagerFactory);
}
```

这样，我们就完成了JPA的全部初始化工作。有些童鞋可能从网上搜索得知JPA需要`persistence.xml`配置文件，以及复杂的`orm.xml`文件。这里我们负责地告诉大家，使用Spring+Hibernate作为JPA实现，无需任何配置文件。

所有Entity Bean的配置和上一节完全相同，全部采用Annotation标注。我们现在只需关心具体的业务类如何通过JPA接口操作数据库。

还是以`UserService`为例，除了标注`@Component`和`@Transactional`外，我们需要注入一个`EntityManager`，但是不要使用`Autowired`，而是`@PersistenceContext`：

```
@Component
@Transactional
public class UserService {
    @PersistenceContext
    EntityManager em;
}
```

我们回顾一下JDBC、Hibernate和JPA提供的接口，实际上，它们的关系如下：

JDBC	Hibernate	JPA
DataSource	SessionFactory	EntityManagerFactory
Connection	Session	EntityManager

`SessionFactory`和`EntityManagerFactory`相当于`DataSource`，`Session`和`EntityManager`相当于`Connection`。每次需要访问数据库的时候，需要获取新的`Session`和`EntityManager`，用完后再关闭。

但是，注意到`UserService`注入的不是`EntityManagerFactory`，而是`EntityManager`，并且标注了`@PersistenceContext`。难道使用JPA可以允许多线程操作同一个`EntityManager`？

实际上这里注入的并不是真正的`EntityManager`，而是一个`EntityManager`的代理类，相当于：

```
public class EntityManagerProxy implements EntityManager {
    private EntityManagerFactory emf;
}
```

Spring遇到标注了`@PersistenceContext`的`EntityManager`会自动注入代理，该代理会在必要的时候自动打开`EntityManager`。换句话说，多线程引用的`EntityManager`虽然是同一个代理类，但该代理类内部针对不同线程会创建不同的`EntityManager`实例。

简单总结一下，标注了`@PersistenceContext`的`EntityManager`可以被多线程安全地共享。

因此，在`UserService`的每个业务方法里，直接使用`EntityManager`就很方便。以主键查询为例：

```
public User getUserById(long id) {
    User user = this.em.find(User.class, id);
    if (user == null) {
        throw new RuntimeException("User not found by id: " + id);
    }
    return user;
}
```

JPA同样支持Criteria查询，比如我们需要的查询如下：

```
SELECT * FROM user WHERE email = ?
```

使用Criteria查询的代码如下：

```
public User fetchUserByEmail(String email) {
    // CriteriaBuilder:
    var cb = em.getCriteriaBuilder();
    CriteriaQuery<User> q = cb.createQuery(User.class);
    Root<User> r = q.from(User.class);
    q.where(cb.equal(r.get("email"), cb.parameter(String.class, "e")));
    TypedQuery<User> query = em.createQuery(q);
    // 绑定参数:
    query.setParameter("e", email);
    // 执行查询:
    List<User> list = query.getResultList();
    return list.isEmpty() ? null : list.get(0);
}
```

一个简单的查询用Criteria写出来就像上面那样复杂，太恐怖了，如果条件多加几个，这种写法谁读得懂？

所以，正常人还是建议写JPQL查询，它的语法和HQL基本差不多：

```
public User getUserByEmail(String email) {
    // JPQL查询:
    TypedQuery<User> query = em.createQuery("SELECT u FROM User u WHERE u.email = :e", User.class);
    query.setParameter("e", email);
    List<User> list = query.getResultList();
    if (list.isEmpty()) {
        throw new RuntimeException("User not found by email.");
    }
    return list.get(0);
}
```

同样的，JPA也支持NamedQuery，即先给查询起个名字，再按名字创建查询：

```
public User login(String email, String password) {
    TypedQuery<User> query = em.createNamedQuery("login", User.class);
    query.setParameter("e", email);
    query.setParameter("p", password);
    List<User> list = query.getResultList();
    return list.isEmpty() ? null : list.get(0);
}
```

NamedQuery通过注解标注在`User`类上，它的定义和上一节的`User`类一样：

```
@NamedQueries(
    @NamedQuery(
        name = "login",
        query = "SELECT u FROM User u WHERE u.email=:e AND u.password=:p"
    )
)
@Entity
public class User {
    ...
}
```

对数据库进行增删改的操作，可以分别使用`persist()`、`remove()`和`merge()`方法，参数均为Entity Bean本身，使用非常简单，这里不再多述。

练习

使用JPA

小结

在Spring中集成JPA要选择一个实现，可以选择Hibernate或EclipseLink；

使用JPA与Hibernate类似，但注入的核心资源是带有`@PersistenceContext`注解的`EntityManager`代理类。

使用Hibernate或JPA操作数据库时，这类ORM干的主要工作就是把ResultSet的每一行变成Java Bean，或者把Java Bean自动转换到INSERT或UPDATE语句的参数中，从而实现ORM。

而ORM框架之所以知道如何把行数据映射到Java Bean，是因为我们在Java Bean的属性上给了足够的注解作为元数据，ORM框架获取Java Bean的注解后，就知道如何进行双向映射。

那么，ORM框架是如何跟踪Java Bean的修改，以便在`update()`操作中更新必要的属性？

答案是使用Proxy模式，从ORM框架读取的User实例实际上并不是User类，而是代理类，代理类继承自User类，但针对每个setter方法做了覆写：

```
public class UserProxy extends User {
    boolean _isNameChanged;

    public void setName(String name) {
        super.setName(name);
        _isNameChanged = true;
    }
}
```

这样，代理类可以跟踪到每个属性的变化。

针对一对多或多对一关系时，代理类可以直接通过getter方法查询数据库：

```
public class UserProxy extends User {
    Session _session;
    boolean _isNameChanged;

    public void setName(String name) {
        super.setName(name);
        _isNameChanged = true;
    }

    /**
     * 获取User对象关联的Address对象:
     */
    public Address getAddress() {
        Query q = _session.createQuery("from Address where userId = :userId");
        q.setParameter("userId", this.getId());
        List<Address> list = query.list();
        return list.isEmpty() ? null : list(0);
    }
}
```

为了实现这样的查询，UserProxy必须保存Hibernate的当前Session。但是，当事务提交后，Session自动关闭，此时再获取`getAddress()`将无法访问数据库，或者获取的不是事务一致的数据。因此，ORM框架总是引入了Attached/Detached状态，表示当前此Java Bean到底是在Session的范围内，还是脱离了Session变成了一个“游离”对象。很多初学者无法正确理解状态变化和事务边界，就会造成大量的`PersistentObjectException`异常。这种隐式状态使得普通Java Bean的生命周期变得复杂。

此外，Hibernate和JPA为了实现兼容多种数据库，它使用HQL或JPQL查询，经过一道转换，变成特定数据库的SQL，理论上这样可以做到无缝切换数据库，但这一层自动转换除了少许的性能开销外，给SQL级别的优化带来了麻烦。

最后，ORM框架通常提供了缓存，并且还分为一级缓存和二级缓存。一级缓存是指在一个Session范围内的缓存，常见的情景是根据主键查询时，两次查询可以返回同一实例：

```
User user1 = session.load(User.class, 123);
User user2 = session.load(User.class, 123);
```

二级缓存是指跨Session的缓存，一般默认关闭，需要手动配置。二级缓存极大的增加了数据的不一致性，原因在于SQL非常灵活，常常会导致意外的更新。例如：

```
// 线程1读取:
User user1 = session1.load(User.class, 123);
...
// 一段时间后，线程2读取:
User user2 = session2.load(User.class, 123);
```

当二级缓存生效的时候，两个线程读取的User实例是一样的，但是，数据库对应的行记录完全可能被修改，例如：

```
-- 给老用户增加100积分:
UPDATE users SET bonus = bonus + 100 WHERE createdAt <= ?
```

ORM无法判断`id=123`的用户是否受该`UPDATE`语句影响。考虑到数据库通常会支持多个应用程序，此UPDATE语句可能由其他进程执行，ORM框架就更不知道了。

我们把这种ORM框架称之为全自动ORM框架。

对比Spring提供的JdbcTemplate，它和ORM框架相比，主要有几点差别：

1. 查询后需要手动提供Mapper实例以便把ResultSet的每一行变为Java对象；
2. 增删改操作所需的参数列表，需要手动传入，即把User实例变为[user.id, user.name, user.email]这样的列表，比较麻烦。

但是JdbcTemplate的优势在于它的确定性：即每次读取操作一定是数据库操作而不是缓存，所执行的SQL是完全确定的，缺点就是代码比较繁琐，构造`INSERT INTO users VALUES (?,?,?)`更是复杂。

所以，介于全自动ORM如Hibernate和手写全部如JdbcTemplate之间，还有一种半自动的ORM，它只负责把ResultSet自动映射到Java Bean，或者自动填充Java Bean参数，但仍需自己写出SQL。MyBatis就是这样一种半自动化ORM框架。

我们来看看如何在Spring中集成MyBatis。

首先，我们要引入MyBatis本身，其次，由于Spring并没有像Hibernate那样内置对MyBatis的集成，所以，我们需要再引入MyBatis官方自己开发的一个与Spring集成的库：

- org.mybatis:mybatis:3.5.4
- org.mybatis:mybatis-spring:2.0.4

和前面一样，先创建`DataSource`是必不可少的：

```
@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() { ... }
}
```

再回顾一下Hibernate和JPA的`SessionFactory`与`EntityManagerFactory`，MyBatis与之对应的是`SqlSessionFactory`和`SqlSession`：

JDBC	Hibernate	JPA	MyBatis
DataSource	SessionFactory	EntityManagerFactory	SqlSessionFactory
Connection	Session	EntityManager	SqlSession

可见，ORM的设计套路都是类似的。使用MyBatis的核心就是创建`SqlSessionFactory`，这里我们需要创建的是`SqlSessionFactoryBean`：

```
@Bean
SqlSessionFactoryBean createSqlSessionFactoryBean(@Autowired DataSource dataSource) {
    var sqlSessionFactoryBean = new SqlSessionFactoryBean();
    sqlSessionFactoryBean.setDataSource(dataSource);
    return sqlSessionFactoryBean;
}
```

因为MyBatis可以直接使用Spring管理的声明式事务，因此，创建事务管理器和使用JDBC是一样的：

```
@Bean
PlatformTransactionManager createTxManager(@Autowired DataSource dataSource) {
    return new DataSourceTransactionManager(dataSource);
}
```

和Hibernate不同的是，MyBatis使用Mapper来实现映射，而且Mapper必须是接口。我们以User类为例，在User类和users表之间映射的UserMapper编写如下：

```
public interface UserMapper {
 @Select("SELECT * FROM users WHERE id = #{id}")
 User getById(@Param("id") long id);
}
```

注意：这里的Mapper不是JdbcTemplate的RowMapper的概念，它是定义访问users表的接口方法。比如我们定义了一个`User getById(long)`的主键查询方法，不仅要定义接口方法本身，还要明确写出查询的SQL，这里用注解`@Select`标记。SQL语句的任何参数，都与方法参数按名称对应。例如，方法参数id的名字通过注解`@Param()`标记为`id`，则SQL语句里将来替换的占位符就是`#{id}`。

如果有多个参数，那么每个参数命名后直接在SQL中写出对应的占位符即可：

```
@Select("SELECT * FROM users LIMIT #{offset}, #{maxResults}")
List<User> getAll(@Param("offset") int offset, @Param("maxResults") int maxResults);
```

注意：MyBatis执行查询后，将根据方法的返回类型自动把ResultSet的每一行转换为User实例，转换规则当然是按列名和属性名对应。如果列名和属性名不同，最简单的方式是编写SELECT语句的别名：

```
-- 列名是created_time，属性名是createdAt:
SELECT id, name, email, created_time AS createdAt FROM users
```

执行INSERT语句就稍微麻烦点，因为我们希望传入User实例，因此，定义的方法接口与`@Insert`注解如下：

```
@Insert("INSERT INTO users (email, password, name, createdAt) VALUES (#{user.email}, #{user.password}, #{user.name}, #{user.createdAt})")
void insert(@Param("user") User user);
```

上述方法传入的参数名称是`user`，参数类型是User类，在SQL中引用的时候，以`#{obj.property}`的方式写占位符。和Hibernate这样的全自动化ORM相比，MyBatis必须写出完整的INSERT语句。

如果`users`表的id是自增主键，那么，我们在SQL中不传入id，但希望获取插入后的主键，需要再加一个`@Options`注解：

```
@Options(useGeneratedKeys = true, keyProperty = "id", keyColumn = "id")
@Insert("INSERT INTO users (email, password, name, createdAt) VALUES (#{user.email}, #{user.password}, #{user.name}, #{user.createdAt})")
void insert(@Param("user") User user);
```

`keyProperty`和`keyColumn`分别指出JavaBean的属性和数据库的主键列名。

执行UPDATE和DELETE语句相对比较简单，我们定义方法如下：

```
@Update("UPDATE users SET name = #{user.name}, createdAt = #{user.createdAt} WHERE id = #{user.id}")
void update(@Param("user") User user);

@Delete("DELETE FROM users WHERE id = #{id}")
void deleteById(@Param("id") long id);
```

有了`UserMapper`接口，还需要对应的实现类才能真正执行这些数据库操作的方法。虽然可以自己写实现类，但我们除了编写`UserMapper`接口外，还有`BookMapper`、`BonusMapper`.....一个一个写太麻烦，因此，MyBatis提供了一个`MapperFactoryBean`来自动创建所有Mapper的实现类。可以用一个简单的注解来启用它：

```
@MapperScan("com.itranswarp.learnjava.mapper")
...其他注解...
public class AppConfig {
    ...
}
```

有了`@MapperScan`，就可以让MyBatis自动扫描指定包的所有Mapper并创建实现类。在真正的业务逻辑中，我们可以直接注入：

```
@Component
@Transactional
public class UserService {
    // 注入UserMapper:
    @Autowired
    UserMapper userMapper;

    public User getUserById(long id) {
        // 调用Mapper方法:
        User user = userMapper.getById(id);
        if (user == null) {
            throw new RuntimeException("User not found by id.");
        }
        return user;
    }
}
```

可见，业务逻辑主要就是通过`XxxMapper`定义的数据库方法来访问数据库。

XML配置

上述在Spring中集成MyBatis的方式，我们只需要用到注解，并没有任何XML配置文件。MyBatis也允许使用XML配置映射关系和SQL语句，例如，更新`User`时根据属性值构造动态SQL：

```
<update id="updateUser">
  UPDATE users SET
  <set>
    <if test="user.name != null"> name = #{user.name} </if>
    <if test="user.hobby != null"> hobby = #{user.hobby} </if>
    <if test="user.summary != null"> summary = #{user.summary} </if>
  </set>
  WHERE id = #{user.id}
</update>
```

编写XML配置的优点是可以组装出动态SQL，并且把所有SQL操作集中在一起。缺点是配置起来太繁琐，调用方法时如果想查看SQL还需要定位到XML配置中。这里我们不介绍XML的配置方式，需要了解的童鞋请自行阅读官方文档。

使用MyBatis最大的问题是所有SQL都需要全部手写，优点是执行的SQL就是我们自己写的SQL，对SQL进行优化非常简单，也可以编写任意复杂的SQL，或者使用数据库的特定语法，但切换数据库可能就不太容易。好消息是大部分项目并没有切换数据库的需求，完全可以针对某个数据库编写尽可能优化的SQL。

练习

集成MyBatis

小结

MyBatis是一个半自动化的ORM框架，需要手写SQL语句，没有自动加载一对多或多对一关系的功能。

我们从前几节可以看到，所谓ORM，也是建立在JDBC的基础上，通过ResultSet到JavaBean的映射，实现各种查询。有自动跟踪Entity修改的全自动化ORM如Hibernate和JPA，需要为每个Entity创建代理，也有完全自己映射，连INSERT和UPDATE语句都需要手动编写的MyBatis，但没有任何透明的Proxy。

而查询是涉及到数据库使用最广泛的操作，需要最大的灵活性。各种ORM解决方案各不相同，Hibernate和JPA自己实现了HQL和JPQL查询语法，用以生成最终的SQL，而MyBatis则完全手写，每增加一个查询都需要先编写SQL并增加接口方法。

还有一种Hibernate和JPA支持的Criteria查询，用Hibernate写出来类似：

```
DetachedCriteria criteria = DetachedCriteria.forClass(User.class);
criteria.add(Restrictions.eq("email", email))
        .add(Restrictions.eq("password", password));
List<User> list = (List<User>) hibernateTemplate.findByCriteria(criteria);
```

上述Criteria查询写法复杂，但和JPA相比，还是小巫见大巫了：

```
var cb = em.getCriteriaBuilder();
CriteriaQuery<User> q = cb.createQuery(User.class);
Root<User> r = q.from(User.class);
q.where(cb.equal(r.get("email"), cb.parameter(String.class, "e")));
TypedQuery<User> query = em.createQuery(q);
query.setParameter("e", email);
List<User> list = query.getResultList();
```

此外，是否支持自动读取一对多和多对一关系也是全自动化ORM框架的一个重要功能。

如果我们自己来设计并实现一个ORM，应该吸取这些ORM的哪些特色，然后高效实现呢？

设计ORM接口

任何设计，都必须明确设计目标。这里我们准备实现的ORM并不想要全自动ORM那种自动读取一对多和多对一关系的功能，也不想给Entity加上复杂的状态，因此，对于Entity来说，它就是纯粹的JavaBean，没有任何Proxy。

此外，ORM要兼顾易用性和适用性。易用性是指能覆盖95%的应用场景，但总有一些复杂的SQL，很难用ORM去自动生成，因此，也要给出原生的JDBC接口，能支持5%的特殊需求。

最后，我们希望设计的接口要易于编写，并使用流式API便于阅读。为了配合编译器检查，还应该支持泛型，避免强制转型。

以User类为例，我们设计的查询接口如下：

```
// 按主键查询: SELECT * FROM users WHERE id = ?
User u = db.get(User.class, 123);

// 条件查询唯一记录: SELECT * FROM users WHERE email = ? AND password = ?
User u = db.from(User.class)
          .where("email=? AND password=?", "bob@example.com", "bob123")
          .unique();

// 条件查询多条记录: SELECT * FROM users WHERE id < ? ORDER BY email LIMIT ?, ?
List<User> us = db.from(User.class)
                  .where("id < ?", 1000)
                  .orderBy("email")
                  .limit(0, 10)
                  .list();

// 查询特定列: SELECT id, name FROM users WHERE email = ?
User u = db.select("id", "name")
          .from(User.class)
          .where("email = ?", "bob@example.com")
          .unique();
```

这样的流式API便于阅读，也非常容易推导出最终生成的SQL。

对于插入、更新和删除操作，就相对比较简单：

```
// 插入User:
db.insert(user);

// 按主键更新更新User:
db.update(user);

// 按主键删除User:
db.delete(User.class, 123);
```

对于Entity来说，通常一个表对应一个。手动列出所有Entity是非常麻烦的，一定要传入package自动扫描。

最后，ORM总是需要元数据才能知道如何映射。我们不想编写复杂的XML配置，也没必要自己去定义一套规则，直接使用JPA的注解就行。

实现ORM

我们并不需要从JDBC底层开始编写，并且，还要考虑到事务，最好能直接使用Spring的声明式事务。实际上，我们可以设计一个全局`DbTemplate`，它注入了Spring的`JdbcTemplate`，涉及到数据库操作时，全部通过`JdbcTemplate`完成，自然天生支持Spring的声明式事务，因为这个ORM只是在`JdbcTemplate`的基础上做了一层封装。

在`AppConfig`中，我们初始化所有Bean如下：

```
@Configuration
@ComponentScan
@EnableTransactionManagement
@PropertySource("jdbc.properties")
public class AppConfig {
    @Bean
    DataSource createDataSource() { ... }

    @Bean
    JdbcTemplate createJdbcTemplate(@Autowired DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    DbTemplate createDbTemplate(@Autowired JdbcTemplate jdbcTemplate) {
        return new DbTemplate(jdbcTemplate, "com.itranswarp.learnjava.entity");
    }

    @Bean
    PlatformTransactionManager createTxManager(@Autowired DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

以上就是我们所需的所有配置。

编写业务逻辑，例如`UserService`，写出来像这样：

```
@Component
@Transactional
public class UserService {
    @Autowired
    DbTemplate db;

    public User getUserById(long id) {
        return db.get(User.class, id);
    }

    public User getUserByEmail(String email) {
        return db.from(User.class)
                 .where("email = ?", email)
                 .unique();
    }

    public List<User> getUsers(int pageIndex) {
        int pageSize = 100;
        return db.from(User.class)
                 .orderBy("id")
                 .limit((pageIndex - 1) * pageSize, pageSize)
                 .list();
    }

    public User register(String email, String password, String name) {
        User user = new User();
        user.setEmail(email);
        user.setPassword(password);
        user.setName(name);
        user.setCreatedAt(System.currentTimeMillis());
        db.insert(user);
        return user;
    }
    ...
}
```

上述代码给出了ORM的接口，以及如何在业务逻辑中使用ORM。下一步，就是如何实现这个`DbTemplate`。这里我们只给出框架代码，有兴趣的童鞋可以自己实现核心代码：

```
public class DbTemplate {
    private JdbcTemplate jdbcTemplate;

    // 保存Entity Class到Mapper的映射:
    private Map<Class<?>, Mapper<?>> classMapping;

    public <T> T fetch(Class<T> clazz, Object id) {
        Mapper<T> mapper = getMapper(clazz);
        List<T> list = (List<T>) jdbcTemplate.query(mapper.selectSQL, new Object[] { id }, mapper.rowMapper);
        if (list.isEmpty()) {
            return null;
        }
        return list.get(0);
    }

    public <T> T get(Class<T> clazz, Object id) {
        ...
    }

    public <T> void insert(T bean) {
        ...
    }

    public <T> void update(T bean) {
        ...
    }

    public <T> void delete(Class<T> clazz, Object id) {
        ...
```

```
    }
}
```

实现链式API的核心代码是第一步从DbTemplate调用select()或from()时实例化一个CriteriaQuery实例，并在后续的链式调用中设置它的字段：

```
public class DbTemplate {
    ...
    public Select select(String... selectFields) {
        return new Select(new Criteria(this), selectFields);
    }

    public <T> From<T> from(Class<T> entityClass) {
        Mapper<T> mapper = getMapper(entityClass);
        return new From<>(new Criteria<>(this), mapper);
    }
}
```

然后以此定义Select、From、Where、OrderBy、Limit等。在From中可以设置Class类型、表名等：

```
public final class From<T> extends CriteriaQuery<T> {
    From(Criteria<T> criteria, Mapper<T> mapper) {
        super(criteria);
        // from可以设置class、tableName:
        this.criteria.mapper = mapper;
        this.criteria.clazz = mapper.entityClass;
        this.criteria.table = mapper.tableName;
    }

    public Where<T> where(String clause, Object... args) {
        return new Where<>(this.criteria, clause, args);
    }
}
```

在Where中可以设置条件参数：

```
public final class Where<T> extends CriteriaQuery<T> {
    Where(Criteria<T> criteria, String clause, Object... params) {
        super(criteria);
        this.criteria.where = clause;
        this.criteria.whereParams = new ArrayList<>();
        // add:
        for (Object param : params) {
            this.criteria.whereParams.add(param);
        }
    }
}
```

最后，链式调用的尽头是调用list()返回一组结果，调用unique()返回唯一结果，调用first()返回首个结果。

在IDE中，可以非常方便地实现链式调用：

需要复杂查询的时候，总是可以使用JdbcTemplate执行任意复杂的SQL。

练习

设计并实现一个微型ORM

小结

ORM框架就是自动映射数据库表结构到JavaBean的工具，设计并实现一个简单高效的ORM框架并不困难。

在Web开发一章中，我们已经详细介绍了JavaEE中Web开发的基础：Servlet。具体地说，有以下几点：

1. Servlet规范定义了几种标准组件：Servlet、JSP、Filter和Listener；
2. Servlet的标准组件总是运行在Servlet容器中，如Tomcat、Jetty、WebLogic等。

直接使用Servlet进行Web开发好比直接在JDBC上操作数据库，比较繁琐，更好的方法是在Servlet基础上封装MVC框架，基于MVC开发Web应用，大部分时候，不需要接触Servlet API，开发省时省力。

我们在MVC开发和MVC高级开发已经由浅入深地介绍了如何编写MVC框架。当然，自己写的MVC主要是理解原理，要实现一个功能全面的MVC需要大量的工作以及广泛的测试。

因此，开发Web应用，首先要选择一个优秀的MVC框架。常用的MVC框架有：

- Struts：最古老的一个MVC框架，目前版本是2，和1.x有很大的区别；
- WebWork：一个比Struts设计更优秀的MVC框架，但不知道出于什么原因，从2.0开始把自己的代码全部塞给Struts 2了；
- Turbine：一个重度使用Velocity，强调布局的MVC框架；
- 其他100+MVC框架……（略）

Spring虽然都可以集成任何Web框架，但是，Spring本身也开发了一个MVC框架，就叫Spring MVC。这个MVC框架设计得足够优秀以至于我们已经不想再费劲去集成类似Struts这样的框架了。

本章我们会详细介绍如何基于Spring MVC开发Web应用。

我们在前面介绍Web开发时已经讲过了Java Web的基础：Servlet容器，以及标准的Servlet组件：

- Servlet：能处理HTTP请求并将HTTP响应返回；
- JSP：一种嵌套Java代码的HTML，将被编译为Servlet；
- Filter：能过滤指定的URL以实现拦截功能；
- Listener：监听指定的事件，如ServletContext、HttpSession的创建和销毁。

此外，Servlet容器为每个Web应用程序自动创建一个唯一的ServletContext实例，这个实例就代表了Web应用程序本身。

在MVC高级开发中，我们手撸了一个MVC框架，接口和Spring MVC类似。如果直接使用Spring MVC，我们写出来的代码类似：

```
@Controller
public class UserController {
    @GetMapping("/register")
    public ModelAndView register() {
        ...
    }

    @PostMapping("/signin")
    public ModelAndView signin(@RequestParam("email") String email, @RequestParam("password") String password) {
        ...
    }
    ...
}
```

但是，Spring提供的是一个IoC容器，所有的Bean，包括Controller，都在Spring IoC容器中被初始化，而Servlet容器由JavaEE服务器提供（如Tomcat），Servlet容器对Spring一无所知，他们之间到底依靠什么进行联系，又是以何种顺序初始化的？

在理解上述问题之前，我们先把基于Spring MVC开发的项目结构搭建起来。首先创建基于Web的Maven工程，引入如下依赖：

- org.springframework:spring-context:5.2.0.RELEASE
- org.springframework:spring-webmvc:5.2.0.RELEASE
- org.springframework:spring-jdbc:5.2.0.RELEASE
- javax.annotation:javax.annotation-api:1.3.2
- io.pebbletemplates:pebble-spring5:3.1.2
- ch.qos.logback:logback-core:1.2.3
- ch.qos.logback:logback-classic:1.2.3
- com.zaxxer:HikariCP:3.4.2
- org.hsqldb:hsqldb:2.5.0

以及`provided`依赖：

- org.apache.tomcat.embed:tomcat-embed-core:9.0.26
- org.apache.tomcat.embed:tomcat-embed-jasper:9.0.26

这个标准的Maven Web工程目录结构如下：

```
spring-web-mvc
├── pom.xml
└── src
    └── main
        ├── java
        │   └── com
        │       └── itranswarp
        │           └── learnjava
        │               ├── AppConfig.java
        │               ├── DatabaseInitializer.java
        │               ├── entity
        │               │   └── User.java
        │               ├── service
        │               │   └── UserService.java
        │               └── web
        │                   └── UserController.java
        ├── resources
        │   ├── jdbc.properties
        │   └── logback.xml
        └── webapp
            ├── WEB-INF
            │   ├── templates
            │   │   ├── _base.html
            │   │   ├── index.html
            │   │   ├── profile.html
            │   │   ├── register.html
            │   │   └── signin.html
            │   └── web.xml
            └── static
                ├── css
                │   └── bootstrap.css
                └── js
                    └── jquery.js
```

其中，`src/main/webapp`是标准web目录，`WEB-INF`存放`web.xml`，编译的class，第三方jar，以及不允许浏览器直接访问的View模版，`static`目录存放所有静态文件。

在`src/main/resources`目录中存放的是Java程序读取的classpath资源文件，除了JDBC的配置文件`jdbc.properties`外，我们又新增了一个`logback.xml`，这是Logback的默认查找的配置文件：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
 <appender name="STDOUT"
  class="ch.qos.logback.core.ConsoleAppender">
  <layout class="ch.qos.logback.classic.PatternLayout">
   <Pattern>%d{yyyy-MM-dd HH:mm:ss} %-5level %logger{36} - %msg%n</Pattern>
  </layout>
 </appender>

 <logger name="com.itranswarp.learnjava" level="info" additivity="false">
  <appender-ref ref="STDOUT" />
 </logger>

 <root level="info">
  <appender-ref ref="STDOUT" />
 </root>
</configuration>
```

上面给出了一个写入到标准输出的Logback配置，可以基于上述配置添加写入到文件的配置。

在`src/main/java`中就是我们编写的Java代码了。

配置Spring MVC

和普通Spring配置一样，我们编写正常的`AppConfig`后，只需加上`@EnableWebMvc`注解，就“激活”了Spring MVC：

```
@Configuration
@ComponentScan
@EnableWebMvc // 启用Spring MVC
@EnableTransactionManagement
@PropertySource("classpath:/jdbc.properties")
public class AppConfig {
    ...
}
```

除了创建`DataSource`、`JdbcTemplate`、`PlatformTransactionManager`外，`AppConfig`需要额外创建几个用于Spring MVC的Bean：

```
@Bean
WebMvcConfigurer createWebMvcConfigurer() {
    return new WebMvcConfigurer() {
        @Override
        public void addResourceHandlers(ResourceHandlerRegistry registry) {
            registry.addResourceHandler("/static/**").addResourceLocations("/static/");
        }
    };
}
```

`WebMvcConfigurer`并不是必须的，但我们在这里创建一个默认的`WebMvcConfigurer`，只覆写`addResourceHandlers()`，目的是让Spring MVC自动处理静态文件，并且映射路径为`/static/**`。

另一个必须要创建的Bean是`ViewResolver`，因为Spring MVC允许集成任何模板引擎，使用哪个模板引擎，就实例化一个对应的`ViewResolver`：

```
@Bean
ViewResolver createViewResolver(@Autowired ServletContext servletContext) {
    PebbleEngine engine = new PebbleEngine.Builder().autoEscaping(true)
            .cacheActive(false)
            .loader(new ServletLoader(servletContext))
            .extension(new SpringExtension())
            .build();
    PebbleViewResolver viewResolver = new PebbleViewResolver();
    viewResolver.setPrefix("/WEB-INF/templates/");
    viewResolver.setSuffix("");
```

```
    viewResolver.setPebbleEngine(engine);
    return viewResolver;
}
```

ViewResolver通过指定prefix和suffix来确定如何查找View。上述配置使用Pebble引擎，指定模板文件存放在/WEB-INF/templates/目录下。

剩下的Bean都是普通的@Component，但Controller必须标记为@Controller，例如：

```
// Controller使用@Controller标记而不是@Component:
@Controller
public class UserController {
    // 正常使用@Autowired注入:
    @Autowired
    UserService userService;

    // 处理一个URL映射:
    @GetMapping("/")
    public ModelAndView index() {
        ...
    }
    ...
}
```

如果是普通的Java应用程序，我们通过main()方法可以很简单地创建一个Spring容器的实例：

```
public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);
}
```

但是问题来了，现在是Web应用程序，而Web应用程序总是由Servlet容器创建，那么，Spring容器应该由谁创建？在什么时候创建？Spring容器中的Controller又是如何通过Servlet调用的？

在Web应用中启动Spring容器有很多种方法，可以通过Listener启动，也可以通过Servlet启动，可以使用XML配置，也可以使用注解配置。这里，我们只介绍一种*最简单*的启动Spring容器的方式。

第一步，我们在web.xml中配置Spring MVC提供的DispatcherServlet：

```
<!DOCTYPE web-app PUBLIC
 "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
 "http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.itranswarp.learnjava.AppConfig</param-value>
        </init-param>
        <load-on-startup>0</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

初始化参数contextClass指定使用注解配置的AnnotationConfigWebApplicationContext，配置文件的位置参数contextConfigLocation指向AppConfig的完整类名，最后，把这个Servlet映射到/*，即处理所有URL。

上述配置可以看作一个样板配置，有了这个配置，Servlet容器会首先初始化Spring MVC的DispatcherServlet，在DispatcherServlet启动时，它根据配置AppConfig创建了一个类型是WebApplicationContext的IoC容器，完成所有Bean的初始化，并将容器绑到ServletContext上。

因为DispatcherServlet持有IoC容器，能从IoC容器中获取所有@Controller的Bean，因此，DispatcherServlet接收到所有HTTP请求后，根据Controller方法配置的路径，就可以正确地把请求转发到指定方法，并根据返回的ModelAndView决定如何渲染页面。

最后，我们在AppConfig中通过main()方法启动嵌入式Tomcat：

```
public static void main(String[] args) throws Exception {
    Tomcat tomcat = new Tomcat();
    tomcat.setPort(Integer.getInteger("port", 8080));
    tomcat.getConnector();
    Context ctx = tomcat.addWebapp("", new File("src/main/webapp").getAbsolutePath());
    WebResourceRoot resources = new StandardRoot(ctx);
    resources.addPreResources(
            new DirResourceSet(resources, "/WEB-INF/classes", new File("target/classes").getAbsolutePath(), "/"));
    ctx.setResources(resources);
    tomcat.start();
    tomcat.getServer().await();
}
```

上述Web应用程序就是我们使用Spring MVC时的一个最小启动功能集。由于使用了JDBC和数据库，用户的注册、登录信息会被持久化：

编写Controller

有了Web应用程序的最基本的结构，我们的重点就可以放在如何编写Controller上。Spring MVC对Controller没有固定的要求，也不需要实现特定的接口。以UserController为例，编写Controller只需要遵循以下要点：

总是标记@Controller而不是@Component：

```
@Controller
public class UserController {
    ...
}
```

一个方法对应一个HTTP请求路径，用@GetMapping或@PostMapping表示GET或POST请求：

```
@PostMapping("/signin")
public ModelAndView doSignin(
        @RequestParam("email") String email,
        @RequestParam("password") String password,
        HttpSession session) {
    ...
}
```

需要接收的HTTP参数以@RequestParam()标注，可以设置默认值。如果方法参数需要传入HttpServletRequest、HttpServletResponse或者HttpSession，直接添加这个类型的参数即可，Spring MVC会自动按类型传入。

返回的ModelAndView通常包含View的路径和一个Map作为Model，但也可以没有Model，例如：

```
return new ModelAndView("signin.html"); // 仅View，没有Model
```

返回重定向时既可以写`new ModelAndView("redirect:/signin")`，也可以直接返回String：

```
public String index() {
    if (...) {
        return "redirect:/signin";
    } else {
        return "redirect:/profile";
    }
}
```

如果在方法内部直接操作`HttpServletResponse`发送响应，返回`null`表示无需进一步处理：

```
public ModelAndView download(HttpServletResponse response) {
    byte[] data = ...
    response.setContentType("application/octet-stream");
    OutputStream output = response.getOutputStream();
    output.write(data);
    output.flush();
    return null;
}
```

对URL进行分组，每组对应一个Controller是一种很好的组织形式，并可以在Controller的class定义出添加URL前缀，例如：

```
@Controller
@RequestMapping("/user")
public class UserController {
    // 注意实际URL映射是/user/profile
    @GetMapping("/profile")
    public ModelAndView profile() {
        ...
    }

    // 注意实际URL映射是/user/changePassword
    @GetMapping("/changePassword")
    public ModelAndView changePassword() {
        ...
    }
}
```

实际方法的URL映射总是前缀+路径，这种形式还可以有效避免不小心导致的重复的URL映射。

可见，Spring MVC允许我们编写既简单又灵活的Controller实现。

练习

在注册、登录等功能的基础上增加一个修改口令的页面。

使用Spring MVC

小结

使用Spring MVC时，整个Web应用程序按如下顺序启动：

1. 启动Tomcat服务器；
2. Tomcat读取web.xml并初始化DispatcherServlet；
3. DispatcherServlet创建IoC容器并自动注册到ServletContext中。

启动后，浏览器发出的HTTP请求全部由DispatcherServlet接收，并根据配置转发到指定Controller的指定方法处理。

使用Spring MVC开发Web应用程序的主要工作就是编写Controller逻辑。在Web应用中，除了需要使用MVC给用户显示页面外，还有一类API接口，我们称之为REST，通常输入输出都是JSON，便于第三方调用或者使用页面JavaScript与之交互。

直接在Controller中处理JSON是可以的，因为Spring MVC的`@GetMapping`和`@PostMapping`都支持指定输入和输出的格式。如果我们想接收JSON，输出JSON，那么可以这样写：

```
@PostMapping(value = "/rest",
             consumes = "application/json;charset=UTF-8",
             produces = "application/json;charset=UTF-8")
@ResponseBody
public String rest(@RequestBody User user) {
    return "{\"restSupport\":true}";
}
```

对应的Maven工程需要加入Jackson这个依赖：`com.fasterxml.jackson.core:jackson-databind:2.11.0`

注意到`@PostMapping`使用`consumes`声明能接收的类型，使用`produces`声明输出的类型，并且额外加了`@ResponseBody`表示返回的`String`无需额外处理，直接作为输出内容写入`HttpServletResponse`。输入的JSON则根据注解`@RequestBody`直接被Spring反序列化为`User`这个JavaBean。

使用curl命令测试一下：

```
$ curl -v -H "Content-Type: application/json" -d '{"email":"bob@example.com"}' http://localhost:8080/rest
> POST /rest HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 27
>
< HTTP/1.1 200
< Content-Type: application/json;charset=utf-8
< Content-Length: 20
< Date: Sun, 10 May 2020 09:56:01 GMT
<
{"restSupport":true}
```

输出正是我们写入的字符串。

直接用Spring的Controller配合一大堆注解写REST太麻烦了，因此，Spring还额外提供了一个`@RestController`注解，使用`@RestController`替代`@Controller`后，每个方法自动变成API接口方法。我们还是以实际代码举例，编写`ApiController`如下：

```
@RestController
@RequestMapping("/api")
public class ApiController {
    @Autowired
    UserService userService;

    @GetMapping("/users")
    public List<User> users() {
        return userService.getUsers();
    }

    @GetMapping("/users/{id}")
    public User user(@PathVariable("id") long id) {
        return userService.getUserById(id);
    }
```

```
    @PostMapping("/signin")
    public Map<String, Object> signin(@RequestBody SignInRequest signinRequest) {
        try {
            User user = userService.signin(signinRequest.email, signinRequest.password);
            return Map.of("user", user);
        } catch (Exception e) {
            return Map.of("error", "SIGNIN_FAILED", "message", e.getMessage());
        }
    }

    public static class SignInRequest {
        public String email;
        public String password;
    }
}
```

编写REST接口只需要定义`@RestController`，然后，每个方法都是一个API接口，输入和输出只要能被Jackson序列化或反序列化为JSON就没有问题。我们用浏览器测试GET请求，可直接显示JSON响应：

要测试POST请求，可以用curl命令：

```
$ curl -v -H "Content-Type: application/json" -d '{"email":"bob@example.com","password":"bob123"}' http://localhost:8080/api/signin
> POST /api/signin HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
> Content-Type: application/json
> Content-Length: 47
>
< HTTP/1.1 200
< Content-Type: application/json
< Transfer-Encoding: chunked
< Date: Sun, 10 May 2020 08:14:13 GMT
<
{"user":{"id":1,"email":"bob@example.com","password":"bob123","name":"Bob",...
```

注意观察上述JSON的输出，`User`能被正确地序列化为JSON，但暴露了`password`属性，这是我们不期望的。要避免输出`password`属性，可以把`User`复制到另一个`UserBean`对象，该对象只持有必要的属性，但这样做比较繁琐。另一种简单的方法是直接在`User`的`password`属性定义处加上`@JsonIgnore`表示完全忽略该属性：

```
public class User {
    ...

    @JsonIgnore
    public String getPassword() {
        return password;
    }

    ...
}
```

但是这样一来，如果写一个`register(User user)`方法，那么该方法的User对象也拿不到注册时用户传入的密码了。如果要允许输入`password`，但不允许输出`password`，即在JSON序列化和反序列化时，允许写属性，禁用读属性，可以更精细地控制如下：

```
public class User {
    ...

    @JsonProperty(access = Access.WRITE_ONLY)
    public String getPassword() {
        return password;
    }

    ...
}
```

同样的，可以使用`@JsonProperty(access = Access.READ_ONLY)`允许输出，不允许输入。

练习

使用REST实现API

小结

使用`@RestController`可以方便地编写REST服务，Spring默认使用JSON作为输入和输出。

要控制序列化和反序列化，可以使用Jackson提供的`@JsonIgnore`和`@JsonProperty`注解。

在Spring MVC中，`DispatcherServlet`只需要固定配置到`web.xml`中，剩下的工作主要是专注于编写Controller。

但是，在Servlet规范中，我们还可以使用Filter。如果要在Spring MVC中使用`Filter`，应该怎么做？

有的童鞋在上一节的Web应用中可能发现了，如果注册时输入中文会导致乱码，因为Servlet默认按非UTF-8编码读取参数。为了修复这一问题，我们可以简单地使用一个EncodingFilter，在全局范围类给`HttpServletRequest`和`HttpServletResponse`强制设置为UTF-8编码。

可以自己编写一个EncodingFilter，也可以直接使用Spring MVC自带的一个`CharacterEncodingFilter`。配置Filter时，只需在`web.xml`中声明即可：

```
<web-app>
    <filter>
        <filter-name>encodingFilter</filter-name>
        <filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>

    <filter-mapping>
        <filter-name>encodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

因为这种Filter和我们业务关系不大，注意到`CharacterEncodingFilter`其实和Spring的IoC容器没有任何关系，两者均互不知晓对方的存在，因此，配置这种Filter十分简单。

我们再考虑这样一个问题：如果允许用户使用Basic模式进行用户验证，即在HTTP请求中添加头`Authorization: Basic email:password`，这个需求如何实现？

编写一个`AuthFilter`是最简单的实现方式：

```
@Component
public class AuthFilter implements Filter {
    @Autowired
```

```
    UserService userService;

    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        // 获取Authorization头:
        String authHeader = req.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Basic ")) {
            // 从Header中提取email和password:
            String email = prefixFrom(authHeader);
            String password = suffixFrom(authHeader);
            // 登录:
            User user = userService.signin(email, password);
            // 放入Session:
            req.getSession().setAttribute(UserController.KEY_USER, user);
        }
        // 继续处理请求:
        chain.doFilter(request, response);
    }
}
```

现在问题来了：在Spring中创建的这个AuthFilter是一个普通Bean，Servlet容器并不知道，所以它不会起作用。

如果我们直接在web.xml中声明这个AuthFilter，注意到AuthFilter的实例将由Servlet容器而不是Spring容器初始化，因此，@Autowire根本不生效，用于登录的UserService成员变量永远是null。

所以，得通过一种方式，让Servlet容器实例化的Filter，间接引用Spring容器实例化的AuthFilter。Spring MVC提供了一个DelegatingFilterProxy，专门来干这个事情：

```
<web-app>
    <filter>
        <filter-name>authFilter</filter-name>
        <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    </filter>

    <filter-mapping>
        <filter-name>authFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

我们来看实现原理：

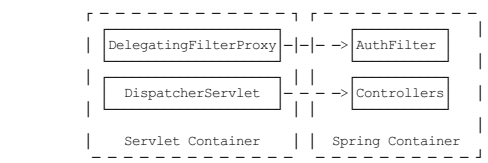
1. Servlet容器从web.xml中读取配置，实例化DelegatingFilterProxy，注意命名是authFilter；
2. Spring容器通过扫描@Component实例化AuthFilter。

当DelegatingFilterProxy生效后，它会自动查找注册在ServletContext上的Spring容器，再试图从容器中查找名为authFilter的Bean，也就是我们用@Component声明的AuthFilter。

DelegatingFilterProxy将请求代理给AuthFilter，核心代码如下：

```
public class DelegatingFilterProxy implements Filter {
    private Filter delegate;
    public void doFilter(...) throws ... {
        if (delegate == null) {
            delegate = findBeanFromSpringContainer();
        }
        delegate.doFilter(req, resp, chain);
    }
}
```

这就是一个代理模式的简单应用。我们画个图表示它们之间的引用关系如下：



如果在web.xml中配置的Filter名字和Spring容器的Bean的名字不一致，那么需要指定Bean的名字：

```
<filter>
    <filter-name>basicAuthFilter</filter-name>
    <filter-class>org.springframework.web.filter.DelegatingFilterProxy</filter-class>
    <!-- 指定Bean的名字 -->
    <init-param>
        <param-name>targetBeanName</param-name>
        <param-value>authFilter</param-value>
    </init-param>
</filter>
```

实际应用时，尽量保持名字一致，以减少不必要的配置。

要使用Basic模式的用户认证，我们可以使用curl命令测试。例如，用户登录名是tom@example.com，口令是tomcat，那么先构造一个使用URL编码的用户名:口令的字符串：

```
tom%40example.com:tomcat
```

对其进行Base64编码，最终构造出的Header如下：

```
Authorization: Basic dG9tJTQwZXhhbXBsZS5jb206dG9tY2F0
```

使用如下的curl命令并获得响应如下：

```
$ curl -v -H 'Authorization: Basic dG9tJTQwZXhhbXBsZS5jb206dG9tY2F0' http://localhost:8080/profile
> GET /profile HTTP/1.1
> Host: localhost:8080
> User-Agent: curl/7.64.1
> Accept: */*
> Authorization: Basic dG9tJTQwZXhhbXBsZS5jb206dG9tY2F0
>
< HTTP/1.1 200
< Set-Cookie: JSESSIONID=CE0F4BFC394816F717443397D4FEABBE; Path=/; HttpOnly
< Content-Type: text/html;charset=UTF-8
< Content-Language: en-CN
< Transfer-Encoding: chunked
< Date: Wed, 29 Apr 2020 00:15:50 GMT
<
<!doctype html>
...HTML输出...
```

上述响应说明AuthFilter已生效。

注意：Basic认证模式并不安全，本节只用来作为使用Filter的示例。

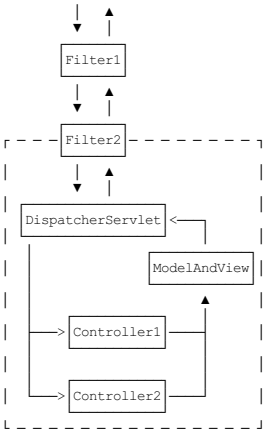
练习

使用DelegatingFilterProxy实现AuthFilter

小结

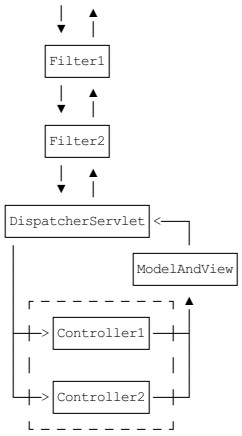
当一个Filter作为Spring容器管理的Bean存在时，可以通过`DelegatingFilterProxy`间接地引用它并使其生效。

在Web程序中，注意到使用Filter的时候，Filter由Servlet容器管理，它在Spring MVC的Web应用程序中作用范围如下：



上图虚线框就是Filter2的拦截范围，Filter组件实际上并不知道后续内部处理是通过Spring MVC提供的`DispatcherServlet`还是其他Servlet组件，因为Filter是Servlet规范定义的标准组件，它可以应用在任何基于Servlet的程序中。

如果只基于Spring MVC开发应用程序，还可以使用Spring MVC提供的一种功能类似Filter的拦截器：Interceptor。和Filter相比，Interceptor拦截范围不是后续整个处理流程，而是仅针对Controller拦截：



上图虚线框就是Interceptor的拦截范围，注意到Controller的处理方法一般都类似这样：

```
@Controller
public class Controller1 {
    @GetMapping("/path/to/hello")
    ModelAndView hello() {
        ...
    }
}
```

所以，Interceptor的拦截范围其实就是Controller方法，它实际上就相当于基于AOP的方法拦截。因为Interceptor只拦截Controller方法，所以要注意，返回`ModelAndView`后，后续对View的渲染就脱离了Interceptor的拦截范围。

使用Interceptor的好处是Interceptor本身是Spring管理的Bean，因此注入任意Bean都非常简单。此外，可以应用多个Interceptor，并通过简单的`@Order`指定顺序。我们先写一个`LoggerInterceptor`：

```
@Order(1)
@Component
public class LoggerInterceptor implements HandlerInterceptor {

    final Logger logger = LoggerFactory.getLogger(getClass());

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {
        logger.info("preHandle {}...", request.getRequestURI());
        if (request.getParameter("debug") != null) {
            PrintWriter pw = response.getWriter();
            pw.write("<p>DEBUG MODE</p>");
            pw.flush();
            return false;
        }
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        logger.info("postHandle {}.", request.getRequestURI());
        if (modelAndView != null) {
            modelAndView.addObject("__time__", LocalDateTime.now());
        }
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) throws Exception {
        logger.info("afterCompletion {}: exception = {}", request.getRequestURI(), ex);
    }
}
```

一个Interceptor必须实现`HandlerInterceptor`接口，可以选择实现`preHandle()`、`postHandle()`和`afterCompletion()`方法。`preHandle()`是Controller方法调用前执行，`postHandle()`是Controller方法正常返回后执行，而`afterCompletion()`无论Controller方法是否抛异常都会执行，参数`ex`就是Controller方法抛出的异常（未抛出异常是`null`）。

在preHandle()中，也可以直接处理响应，然后返回false表示无需调用Controller方法继续处理了，通常在认证或者安全检查失败时直接返回错误响应。在postHandle()中，因为捕获了Controller方法返回的ModelAndView，所以可以继续往ModelAndView里添加一些通用数据，很多页面需要的全局数据如Copyright信息等都可以放到这里，无需在每个Controller方法中重复添加。

我们再继续添加一个AuthInterceptor，用于替代上一节使用AuthFilter进行Basic认证的功能：

```
@Order(2)
@Component
public class AuthInterceptor implements HandlerInterceptor {

    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    UserService userService;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        logger.info("pre authenticate {}...", request.getRequestURI());
        try {
            authenticateByHeader(request);
        } catch (RuntimeException e) {
            logger.warn("login by authorization header failed.", e);
        }
        return true;
    }

    private void authenticateByHeader(HttpServletRequest req) {
        String authHeader = req.getHeader("Authorization");
        if (authHeader != null && authHeader.startsWith("Basic ")) {
            logger.info("try authenticate by authorization header...");
            String up = new String(Base64.getDecoder().decode(authHeader.substring(6)), StandardCharsets.UTF_8);
            int pos = up.indexOf(':');
            if (pos > 0) {
                String email = URLDecoder.decode(up.substring(0, pos), StandardCharsets.UTF_8);
                String password = URLDecoder.decode(up.substring(pos + 1), StandardCharsets.UTF_8);
                User user = userService.signin(email, password);
                req.getSession().setAttribute(UserController.KEY_USER, user);
                logger.info("user {} login by authorization header ok.", email);
            }
        }
    }
}
```

这个AuthInterceptor是由Spring容器直接管理的，因此注入UserService非常方便。

最后，要让拦截器生效，我们在WebMvcConfigurer中注册所有的Interceptor：

```
@Bean
WebMvcConfigurer createWebMvcConfigurer(@Autowired HandlerInterceptor[] interceptors) {
    return new WebMvcConfigurer() {
        public void addInterceptors(InterceptorRegistry registry) {
            for (var interceptor : interceptors) {
                registry.addInterceptor(interceptor);
            }
        }
        ...
    };
}
```

如果拦截器没有生效，请检查是否忘了在WebMvcConfigurer中注册。

处理异常

在Controller中，Spring MVC还允许定义基于@ExceptionHandler注解的异常处理方法。我们来看具体的示例代码：

```
@Controller
public class UserController {
    @ExceptionHandler(RuntimeException.class)
    public ModelAndView handleUnknowException(Exception ex) {
        return new ModelAndView("500.html", Map.of("error", ex.getClass().getSimpleName(), "message", ex.getMessage()));
    }
    ...
}
```

异常处理方法没有固定的方法签名，可以传入Exception、HttpServletRequest等，返回值可以是void，也可以是ModelAndView，上述代码通过@ExceptionHandler(RuntimeException.class)表示当发生RuntimeException的时候，就自动调用此方法处理。

注意到我们返回了一个新的ModelAndView，这样在应用程序内部如果发生了预料之外的异常，可以给用户显示一个出错页面，而不是简单的500 Internal Server Error或404 Not Found。例如B站的错误页：

可以编写多个错误处理方法，每个方法针对特定的异常。例如，处理LoginException使得页面可以自动跳转到登录页。

使用ExceptionHandler时，要注意它仅作用于当前的Controller，即ControllerA中定义的一个ExceptionHandler方法对ControllerB不起作用。如果我们有很多Controller，每个Controller都需要处理一些通用的异常，例如LoginException，思考一下应该怎么避免重复代码？

练习

使用Interceptor

小结

Spring MVC提供了Interceptor组件来拦截Controller方法，使用时要注意Interceptor的作用范围。

在开发REST应用时，很多时候，是通过页面的JavaScript和后端的REST API交互。

在JavaScript与REST交互的时候，有很多安全限制。默认情况下，浏览器按同源策略放行JavaScript调用API，即：

- 如果A站在域名a.com页面的JavaScript调用A站自己的API时，没有问题；
- 如果A站在域名a.com页面的JavaScript调用B站b.com的API时，将被浏览器拒绝访问，因为不满足同源策略。

同源要求域名要完全相同（a.com和www.a.com不同），协议要相同（http和https不同），端口要相同 。

那么，在域名a.com页面的JavaScript要调用B站b.com的API时，还有没有办法？

办法是有的，那就是CORS，全称Cross-Origin Resource Sharing，是HTML5规范定义的如何跨域访问资源。如果A站的JavaScript访问B站API的时候，B站能够返回响应头Access-Control-Allow-Origin: http://a.com，那么，浏览器就允许A站的JavaScript访问B站的API。

注意到跨域访问能否成功，取决于B站是否愿意给A站返回一个正确的Access-Control-Allow-Origin响应头，所以决定权永远在提供API的服务方手中。

关于CORS的详细信息可以参考MDN文档，这里不再详述。

使用Spring的@RestController开发REST应用时，同样会面对跨域问题。如果我们允许指定的网站通过页面JavaScript访问这些REST API，就必须正确地设置CORS。

有好几种方法设置CORS，我们来一一介绍。

使用@CrossOrigin

第一种方法是使用`@CrossOrigin`注解，可以在`@RestController`的class级别或方法级别定义一个`@CrossOrigin`，例如：

```
@CrossOrigin(origins = "http://local.liaoxuefeng.com:8080")
@RestController
@RequestMapping("/api")
public class ApiController {
    ...
}
```

上述定义在`ApiController`处的`@CrossOrigin`指定了只允许来自`local.liaoxuefeng.com`跨域访问，允许多个域访问需要写成数组形式，例如`origins = {"http://a.com", "https://www.b.com"})`。如果要允许任何域访问，写成`origins = "*"`即可。

如果有多个REST Controller都需要使用CORS，那么，每个Controller都必须标注`@CrossOrigin`注解。

使用CorsRegistry

第二种方法是在`WebMvcConfigurer`中定义一个全局CORS配置，下面是一个示例：

```
@Bean
WebMvcConfigurer createWebMvcConfigurer() {
    return new WebMvcConfigurer() {
        @Override
        public void addCorsMappings(CorsRegistry registry) {
            registry.addMapping("/api/**")
                    .allowedOrigins("http://local.liaoxuefeng.com:8080")
                    .allowedMethods("GET", "POST")
                    .maxAge(3600);
            // 可以继续添加其他URL规则:
            // registry.addMapping("/rest/v2/**")...
        }
    };
}
```

这种方式可以创建一个全局CORS配置，如果仔细地设计URL结构，那么可以一目了然地看到各个URL的CORS规则，推荐使用这种方式配置CORS。

使用CorsFilter

第三种方法是使用Spring提供的`CorsFilter`，我们在集成Filter中详细介绍了将Spring容器内置的Bean暴露为Servlet容器的Filter的方法，由于这种配置方式需要修改`web.xml`，也比较繁琐，所以推荐使用第二种方式。

测试

当我们配置好CORS后，可以在浏览器中测试一下规则是否生效。

我们先用`http://localhost:8080`在Chrome浏览器中打开首页，然后打开Chrome的开发者工具，切换到Console，输入一个JavaScript语句来跨域访问API：

```
$.getJSON( "http://local.liaoxuefeng.com:8080/api/users", (data) => console.log(JSON.stringify(data)));
```

上述源站的域是`http://localhost:8080`，跨域访问的是`http://local.liaoxuefeng.com:8080`，因为配置的CORS不允许`localhost`访问，所以不出意外地得到一个错误：

浏览题打印了错误原因就是`been blocked by CORS policy`。

我们再用`http://local.liaoxuefeng.com:8080`在Chrome浏览器中打开首页，在Console中执行JavaScript访问`localhost`：

```
$.getJSON( "http://localhost:8080/api/users", (data) => console.log(JSON.stringify(data)));
```

因为CORS规则允许来自`http://local.liaoxuefeng.com:8080`的访问，因此访问成功，打印出API的返回值：

练习

使用CORS控制跨域

小结

CORS可以控制指定域的页面JavaScript能否访问API。

在开发应用程序的时候，经常会遇到支持多语言的需求，这种支持多语言的功能称之为国际化，英文是internationalization，缩写为i18n（因为首字母i和末字母n中间有18个字母）。

还有针对特定地区的本地化功能，英文是localization，缩写为L10n，本地化是指根据地区调整类似姓名、日期的显示等。

也有把上面两者合称为全球化，英文是globalization，缩写为g11n。

在Java中，支持多语言和本地化是通过`MessageFormat`配合`Locale`实现的：

```
// MessageFormat
----
import java.text.MessageFormat;
import java.util.Locale;

public class Time {
    public static void main(String[] args) {
        double price = 123.5;
        int number = 10;
        Object[] arguments = { price, number };
        MessageFormat mfUS = new MessageFormat("Pay {0,number,currency} for {1} books.", Locale.US);
        System.out.println(mfUS.format(arguments));
        MessageFormat mfZH = new MessageFormat("{1}本书一共{0,number,currency}。", Locale.CHINA);
        System.out.println(mfZH.format(arguments));
    }
}
```

对于Web应用程序，要实现国际化功能，主要是渲染View的时候，要把各种语言的资源文件提出来，这样，不同的用户访问同一个页面时，显示的语言就是不同的。

我们来看看在Spring MVC应用程序中如何实现国际化。

获取Locale

实现国际化的第一步是获取到用户的`Locale`。在Web应用程序中，HTTP规范规定了浏览器会在请求中携带`Accept-Language`头，用来指示用户浏览器设定的语言顺序，如：

```
Accept-Language: zh-CN,zh;q=0.8,en;q=0.2
```

上述HTTP请求头表示优先选择简体中文，其次选择中文，最后选择英文。`q`表示权重，解析后我们可获得一个根据优先级排序的语言列表，把它转换为Java的`Locale`，即获得了用户的`Locale`。大多数框架通常

只返回权重最高的Locale。

Spring MVC通过LocaleResolver来自动从HttpServletRequest中获取Locale。有多种LocaleResolver的实现类，其中最常用的是CookieLocaleResolver：

```
@Bean
LocaleResolver createLocaleResolver() {
    var clr = new CookieLocaleResolver();
    clr.setDefaultLocale(Locale.ENGLISH);
    clr.setDefaultTimeZone(TimeZone.getDefault());
    return clr;
}
```

CookieLocaleResolver从HttpServletRequest中获取Locale时，首先根据一个特定的Cookie判断是否指定了Locale，如果没有，就从HTTP头获取，如果还没有，就返回默认的Locale。

当用户第一次访问网站时，CookieLocaleResolver只能从HTTP头获取Locale，即使用浏览器的默认语言。通常网站也允许用户自己选择语言，此时，CookieLocaleResolver就会把用户选择的语言存放到Cookie中，下一次访问时，就会返回用户上次选择的语言而不是浏览器默认语言。

提取资源文件

第二步是把写死在模板中的字符串以资源文件的方式存储在外部。对于多语言，主文件名如果命名为messages，那么资源文件必须按如下方式命名并放入classpath中：

- 默认语言，文件名必须为messages.properties；
- 简体中文，Locale是zh_CN，文件名必须为messages_zh_CN.properties；
- 日文，Locale是ja_JP，文件名必须为messages_ja_JP.properties；
- 其它更多语言……

每个资源文件都有相同的key，例如，默认语言是英文，文件messages.properties内容如下：

```
language.select=Language
home=Home
signin=Sign In
copyright=Copyright©{0,number,#}
```

文件messages_zh_CN.properties内容如下：

```
language.select=语言
home=首页
signin=登录
copyright=版权所有©{0,number,#}
```

创建MessageSource

第三步是创建一个Spring提供的MessageSource实例，它自动读取所有的.properties文件，并提供一个统一接口来实现“翻译”：

```
// code, arguments, locale:
String text = messageSource.getMessage("signin", null, locale);
```

其中，signin是我们在.properties文件中定义的key，第二个参数是Object[]数组作为格式化时传入的参数，最后一个参数就是获取的用户Locale实例。

创建MessageSource如下：

```
@Bean("i18n")
MessageSource createMessageSource() {
    var messageSource = new ResourceBundleMessageSource();
    // 指定文件是UTF-8编码:
    messageSource.setDefaultEncoding("UTF-8");
    // 指定主文件名:
    messageSource.setBasename("messages");
    return messageSource;
}
```

注意到ResourceBundleMessageSource会自动根据主文件名自动把所有相关语言的资源文件都读进来。

再注意到Spring容器会创建不只一个MessageSource实例，我们自己创建的这个MessageSource是专门给页面国际化使用的，因此命名为i18n，不会与其它MessageSource实例冲突。

实现多语言

要在View中使用MessageSource加上Locale输出多语言，我们通过编写一个MvcInterceptor，把相关资源注入到ModelAndView中：

```
@Component
public class MvcInterceptor implements HandlerInterceptor {
    @Autowired
    LocaleResolver localeResolver;

    // 注意注入的MessageSource名称是i18n:
    @Autowired
    @Qualifier("i18n")
    MessageSource messageSource;

    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {
        if (modelAndView != null) {
            // 解析用户的Locale:
            Locale locale = localeResolver.resolveLocale(request);
            // 放入Model:
            modelAndView.addObject("__messageSource__", messageSource);
            modelAndView.addObject("__locale__", locale);
        }
    }
}
```

不要忘了在WebMvcConfigurer中注册MvcInterceptor。现在，就可以在View中调用MessageSource.getMessage()方法来实现多语言：

```
<a href="/signin">{{ __messageSource__.getMessage('signin', null, __locale__) }}</a>
```

上述这种写法虽然可行，但格式太复杂了。使用View时，要根据每个特定的View引擎定制国际化函数。在Pebble中，我们可以封装一个国际化函数，名称就是下划线_，改造一下创建ViewResolver的代码：

```
@Bean
ViewResolver createViewResolver(@Autowired ServletContext servletContext, @Autowired @Qualifier("i18n") MessageSource messageSource) {
    PebbleEngine engine = new PebbleEngine.Builder()
            .autoEscaping(true)
            .cacheActive(false)
            .loader(new ServletLoader(servletContext))
            // 添加扩展:
            .extension(createExtension(messageSource))
            .build();
    PebbleViewResolver viewResolver = new PebbleViewResolver();
    viewResolver.setPrefix("/WEB-INF/templates/");
    viewResolver.setSuffix("");
    viewResolver.setPebbleEngine(engine);
    return viewResolver;
}

private Extension createExtension(MessageSource messageSource) {
    return new AbstractExtension() {
        @Override
```

```
        public Map<String, Function> getFunctions() {
            return Map.of("_", new Function() {
                public Object execute(Map<String, Object> args, PebbleTemplate self, EvaluationContext context, int lineNumber) {
                    String key = (String) args.get("0");
                    List<Object> arguments = this.extractArguments(args);
                    Locale locale = (Locale) context.getVariable("__locale__");
                    return messageSource.getMessage(key, arguments.toArray(), "???" + key + "???", locale);
                }
                private List<Object> extractArguments(Map<String, Object> args) {
                    int i = 1;
                    List<Object> arguments = new ArrayList<>();
                    while (args.containsKey(String.valueOf(i))) {
                        Object param = args.get(String.valueOf(i));
                        arguments.add(param);
                        i++;
                    }
                    return arguments;
                }
                public List<String> getArgumentNames() {
                    return null;
                }
            });
        }
    };
}
```

这样，我们可以把多语言页面改写为：

```
<a href="/signin">{{ _('signin') }}</a>
```

如果是带参数的多语言，需要把参数传进去：

```
<h5>{{ _('copyright', 2020) }}</h5>
```

使用其它View引擎时，也应当根据引擎接口实现更方便的语法。

切换Locale

最后，我们需要允许用户手动切换Locale，编写一个LocaleController来实现该功能：

```
@Controller
public class LocaleController {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    LocaleResolver localeResolver;

    @GetMapping("/locale/{lo}")
    public String setLocale(@PathVariable("lo") String lo, HttpServletRequest request, HttpServletResponse response) {
        // 根据传入的lo创建Locale实例:
        Locale locale = null;
        int pos = lo.indexOf('_');
        if (pos > 0) {
            String lang = lo.substring(0, pos);
            String country = lo.substring(pos + 1);
            locale = new Locale(lang, country);
        } else {
            locale = new Locale(lo);
        }
        // 设定此Locale:
        localeResolver.setLocale(request, response, locale);
        logger.info("locale is set to {}.", locale);
        // 刷新页面:
        String referer = request.getHeader("Referer");
        return "redirect:" + (referer == null ? "/" : referer);
    }
}
```

在页面设计中，通常在右上角给用户提供一个语言选择列表，来看看效果：

切换到中文：

练习

在MVC程序中实现国际化

小结

多语言支持需要从HTTP请求中解析用户的Locale，然后针对不同Locale显示不同的语言；

Spring MVC应用程序通过MessageSource和LocaleResolver，配合View实现国际化。

在Servlet模型中，每个请求都是由某个线程处理，然后，将响应写入IO流，发送给客户端。从开始处理请求，到写入响应完成，都是在同一个线程中处理的。

实现Servlet容器的时候，只要每处理一个请求，就创建一个新线程处理它，就能保证正确实现了Servlet线程模型。在实际产品中，例如Tomcat，总是通过线程池来处理请求，它仍然符合一个请求从头到尾都由某一个线程处理。

这种线程模型非常重要，因为Spring的JDBC事务是基于ThreadLocal实现的，如果在处理过程中，一会由线程A处理，一会又由线程B处理，那事务就全乱套了。此外，很多安全认证，也是基于ThreadLocal实现的，可以保证在处理请求的过程中，各个线程互不影响。

但是，如果一个请求处理的时间较长，例如几秒钟甚至更长，那么，这种基于线程池的同步模型很快就会把所有线程耗尽，导致服务器无法响应新的请求。如果把长时间处理的请求改为异步处理，那么线程池的利用率就会大大提高。Servlet从3.0规范开始添加了异步支持，允许对一个请求进行异步处理。

我们先来看看在Spring MVC中如何实现对请求进行异步处理的逻辑。首先建立一个Web工程，然后编辑web.xml文件如下：

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
    version="3.1">
    <display-name>Archetype Created Web Application</display-name>

    <servlet>
        <servlet-name>dispatcher</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextClass</param-name>
            <param-value>org.springframework.web.context.support.AnnotationConfigWebApplicationContext</param-value>
        </init-param>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <param-value>com.itranswarp.learnjava.AppConfig</param-value>
```

```
        </init-param>
        <load-on-startup>0</load-on-startup>
        <async-supported>true</async-supported>
    </servlet>

    <servlet-mapping>
        <servlet-name>dispatcher</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

和前面普通的MVC程序相比，这个`web.xml`主要有几点不同：

- 不能再使用`<!DOCTYPE ...web-app_2_3.dtd">`的DTD声明，必须用新的支持Servlet 3.1规范的XSD声明，照抄即可；
- 对`DispatcherServlet`的配置多了一个`<async-supported>`，默认值是`false`，必须明确写成`true`，这样Servlet容器才会支持async处理。

下一步就是在Controller中编写async处理逻辑。我们以`ApiController`为例，演示如何异步处理请求。

第一种async处理方式是返回一个`Callable`，Spring MVC自动把返回的`Callable`放入线程池执行，等待结果返回后再写入响应：

```
@GetMapping("/users")
public Callable<List<User>> users() {
    return () -> {
        // 模拟3秒耗时:
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
        }
        return userService.getUsers();
    };
}
```

第二种async处理方式是返回一个`DeferredResult`对象，然后在另一个线程中，设置此对象的值并写入响应：

```
@GetMapping("/users/{id}")
public DeferredResult<User> user(@PathVariable("id") long id) {
    DeferredResult<User> result = new DeferredResult<>(3000L); // 3秒超时
    new Thread(() -> {
        // 等待1秒:
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }
        try {
            User user = userService.getUserById(id);
            // 设置正常结果并由Spring MVC写入Response:
            result.setResult(user);
        } catch (Exception e) {
            // 设置错误结果并由Spring MVC写入Response:
            result.setErrorResult(Map.of("error", e.getClass().getSimpleName(), "message", e.getMessage()));
        }
    }).start();
    return result;
}
```

使用`DeferredResult`时，可以设置超时，超时会自动返回超时错误响应。在另一个线程中，可以调用`setResult()`写入结果，也可以调用`setErrorResult()`写入一个错误结果。

运行程序，当我们访问`http://localhost:8080/api/users/1`时，假定用户存在，则浏览器在1秒后返回结果：

访问一个不存在的User ID，则等待1秒后返回错误结果：

使用Filter

当我们使用async模式处理请求时，原有的Filter也可以工作，但我们必须在`web.xml`中添加`<async-supported>`并设置为`true`。我们用两个Filter：SyncFilter和AsyncFilter分别测试：

```
<web-app ...>
    ...
    <filter>
        <filter-name>sync-filter</filter-name>
        <filter-class>com.itranswarp.learnjava.web.SyncFilter</filter-class>
    </filter>

    <filter>
        <filter-name>async-filter</filter-name>
        <filter-class>com.itranswarp.learnjava.web.AsyncFilter</filter-class>
        <async-supported>true</async-supported>
    </filter>

    <filter-mapping>
        <filter-name>sync-filter</filter-name>
        <url-pattern>/api/version</url-pattern>
    </filter-mapping>

    <filter-mapping>
        <filter-name>async-filter</filter-name>
        <url-pattern>/api/*</url-pattern>
    </filter-mapping>
    ...
</web-app>
```

一个声明为支持`<async-supported>`的Filter既可以过滤async处理请求，也可以过滤正常的同步处理请求，而未声明`<async-supported>`的Filter无法支持async请求，如果一个普通的Filter遇到async请求时，会直接报错，因此，务必注意普通Filter的`<url-pattern>`不要匹配async请求路径。

在`logback.xml`配置文件中，我们把输出格式加上`[%thread]`，可以输出当前线程的名称：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>%d{yyyy-MM-dd HH:mm:ss} [%thread] %-5level %logger{36} - %msg%n</Pattern>
        </layout>
    </appender>
    ...
</configuration>
```

对于同步请求，例如`/api/version`，我们可以看到如下输出：

```
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO  c.i.learnjava.web.SyncFilter - start SyncFilter...
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO  c.i.learnjava.web.AsyncFilter - start AsyncFilter...
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO  c.i.learnjava.web.ApiController - get version...
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO  c.i.learnjava.web.AsyncFilter - end AsyncFilter.
2020-05-16 11:22:40 [http-nio-8080-exec-1] INFO  c.i.learnjava.web.SyncFilter - end SyncFilter.
```

可见，每个Filter和ApiController都是由同一个线程执行。

对于异步请求，例如/api/users，我们可以看到如下输出：

```
2020-05-16 11:23:49 [http-nio-8080-exec-4] INFO  c.i.learnjava.web.AsyncFilter - start AsyncFilter...
2020-05-16 11:23:49 [http-nio-8080-exec-4] INFO  c.i.learnjava.web.ApiController - get users...
2020-05-16 11:23:49 [http-nio-8080-exec-4] INFO  c.i.learnjava.web.AsyncFilter - end AsyncFilter.
2020-05-16 11:23:52 [MvcAsync1] INFO  c.i.learnjava.web.ApiController - return users...
```

可见，AsyncFilter和ApiController是由同一个线程执行的，但是，返回响应的是另一个线程。

对DeferredResult测试，可以看到如下输出：

```
2020-05-16 11:25:24 [http-nio-8080-exec-8] INFO  c.i.learnjava.web.AsyncFilter - start AsyncFilter...
2020-05-16 11:25:24 [http-nio-8080-exec-8] INFO  c.i.learnjava.web.AsyncFilter - end AsyncFilter.
2020-05-16 11:25:25 [Thread-2] INFO  c.i.learnjava.web.ApiController - deferred result is set.
```

同样，返回响应的是另一个线程。

在实际使用时，经常用到的就是DeferredResult，因为返回DeferredResult时，可以设置超时、正常结果和错误结果，易于编写比较灵活的逻辑。

使用async异步处理响应时，要时刻牢记，在另一个异步线程中的事务和Controller方法中执行的事务不是同一个事务，在Controller中绑定的ThreadLocal信息也无法在异步线程中获取。

此外，Servlet 3.0规范添加的异步支持是针对同步模型打了一个“补丁”，虽然可以异步处理请求，但高并发异步请求时，它的处理效率并不高，因为这种异步模型并没有用到真正的“原生”异步。Java标准库提供了封装操作系统的异步IO包java.nio，是真正的多路复用IO模型，可以用少量线程支持大量并发。使用NIO编程复杂度比同步IO高很多，因此我们很少直接使用NIO。相反，大部分需要高性能异步IO的应用程序会选择Netty这样的框架，它基于NIO提供了更易于使用的API，方便开发异步应用程序。

练习

使用Spring MVC实现异步处理请求

小结

在Spring MVC中异步处理请求需要正确配置web.xml，并返回Callable或DeferredResult对象。

WebSocket是一种基于HTTP的长链接技术。传统的HTTP协议是一种请求-响应模型，如果浏览器不发送请求，那么服务器无法主动给浏览器推送数据。如果需要定期给浏览器推送数据，例如股票行情，或者不定期给浏览器推送数据，例如在线聊天，基于HTTP协议实现这类需求，只能依靠浏览器的JavaScript定时轮询，效率很低且实时性不高。

因为HTTP本身是基于TCP连接的，所以，WebSocket在HTTP协议的基础上做了一个简单的升级，即建立TCP连接后，浏览器发送请求时，附带几个头：

```
GET /chat HTTP/1.1
Host: www.example.com
Upgrade: websocket
Connection: Upgrade
```

就表示客户端希望升级连接，变成长连接的WebSocket，服务器返回升级成功的响应：

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
```

收到成功响应后表示WebSocket“握手”成功，这样，代表WebSocket的这个TCP连接将不会被服务器关闭，而是一直保持，服务器可随时向浏览器推送消息，浏览器也可随时向服务器推送消息。双方推送的消息既可以是文本消息，也可以是二进制消息，一般来说，绝大部分应用程序会推送基于JSON的文本消息。

现代浏览器都已经支持WebSocket协议，服务器则需要底层框架支持。Java的Servlet规范从3.1开始支持WebSocket，所以，必须选择支持Servlet 3.1或更高规范的Servlet容器，才能支持WebSocket。最新版本的Tomcat、Jetty等开源服务器均支持WebSocket。

我们以实际代码演示如何在Spring MVC中实现对WebSocket的支持。首先，我们需要在pom.xml中加入以下依赖：

- org.apache.tomcat.embed:tomcat-embed-websocket:9.0.26
- org.springframework:spring-websocket:5.2.0.RELEASE

第一项是嵌入式Tomcat支持WebSocket的组件，第二项是Spring封装的支持WebSocket的接口。

接下来，我们需要在AppConfig中加入Spring Web对WebSocket的配置，此处我们需要创建一个WebSocketConfigurer实例：

```
@Bean
WebSocketConfigurer createWebSocketConfigurer(
        @Autowired ChatHandler chatHandler,
        @Autowired ChatHandshakeInterceptor chatInterceptor)
{
    return new WebSocketConfigurer() {
        public void registerWebSocketHandlers(WebSocketHandlerRegistry registry) {
            // 把URL与指定的WebSocketHandler关联，可关联多个:
            registry.addHandler(chatHandler, "/chat").addInterceptors(chatInterceptor);
        }
    };
}
```

此实例在内部通过WebSocketHandlerRegistry注册能处理WebSocket的WebSocketHandler，以及可选的WebSocket拦截器HandshakeInterceptor。我们注入的这两个类都是自己编写的业务逻辑，后面我们详细讨论如何编写它们，这里只需关注浏览器连接到WebSocket的URL是/chat。

处理WebSocket连接

和处理普通HTTP请求不同，没法用一个方法处理一个URL。Spring提供了TextWebSocketHandler和BinaryWebSocketHandler分别处理文本消息和二进制消息，这里我们选择文本消息作为聊天室的协议，因此，ChatHandler需要继承自TextWebSocketHandler：

```
@Component
public class ChatHandler extends TextWebSocketHandler {
    ...
}
```

当浏览器请求一个WebSocket连接后，如果成功建立连接，Spring会自动调用afterConnectionEstablished()方法，任何原因导致WebSocket连接中断时，Spring会自动调用afterConnectionClosed方法，因此，覆写这两个方法即可处理连接成功和结束后的业务逻辑：

```
@Component
public class ChatHandler extends TextWebSocketHandler {
    // 保存所有Client的WebSocket会话实例:
    private Map<String, WebSocketSession> clients = new ConcurrentHashMap<>();

    @Override
    public void afterConnectionEstablished(WebSocketSession session) throws Exception {
        // 新会话根据ID放入Map:
        clients.put(session.getId(), session);
        session.getAttributes().put("name", "Guest1");
    }

    @Override
    public void afterConnectionClosed(WebSocketSession session, CloseStatus status) throws Exception {
        clients.remove(session.getId());
    }
}
```

每个WebSocket会话以WebSocketSession表示，且已分配唯一ID。和WebSocket相关的数据，例如用户名称等，均可放入关联的getAttributes()中。

用实例变量clients持有当前所有的WebSocketSession是为了广播，即向所有用户推送同一消息时，可以这么写：

```
String json = ...
TextMessage message = new TextMessage(json);
for (String id : clients.keySet()) {
    WebSocketSession session = clients.get(id);
    session.sendMessage(message);
}
```

我们发送的消息是序列化后的JSON，可以用ChatMessage表示：

```
public class ChatMessage {
 public long timestamp;
 public String name;
    public String text;
}
```

每收到一个用户的消息后，我们就需要广播给所有用户：

```
@Component
public class ChatHandler extends TextWebSocketHandler {
    ...
    @Override
    protected void handleTextMessage(WebSocketSession session, TextMessage message) throws Exception {
        String s = message.getPayload();
        String r = ... // 根据输入消息构造待发送消息
        broadcastMessage(r); // 推送给所有用户
    }
}
```

如果要推送给指定的几个用户，那就需要在clients中根据条件查找出某些WebSocketSession，然后发送消息。

注意到我们在注册WebSocket时还传入了一个ChatHandshakeInterceptor，这个类实际上可以从HttpSessionHandshakeInterceptor继承，它的主要作用是在WebSocket建立连接后，把HttpSession的一些属性复制到WebSocketSession，例如，用户的登录信息等：

```
@Component
public class ChatHandshakeInterceptor extends HttpSessionHandshakeInterceptor {
    public ChatHandshakeInterceptor() {
        // 指定从HttpSession复制属性到WebSocketSession:
        super(List.of(UserController.KEY_USER));
    }
}
```

这样，在ChatHandler中，可以从WebSocketSession.getAttributes()中获取到复制过来的属性。

客户端开发

在完成了服务器端的开发后，我们还需要在页面编写一点JavaScript逻辑：

```
// 创建WebSocket连接:
var ws = new WebSocket('ws://' + location.host + '/chat');
// 连接成功时:
ws.addEventListener('open', function (event) {
    console.log('websocket connected.');
});
// 收到消息时:
ws.addEventListener('message', function (event) {
    console.log('message: ' + event.data);
    var msgs = JSON.parse(event.data);
    // TODO:
});
// 连接关闭时:
ws.addEventListener('close', function () {
    console.log('websocket closed.');
});
// 绑定到全局变量:
window.chatWs = ws;
```

用户可以在连接成功后任何时候给服务器发送消息：

```
var inputText = 'Hello, WebSocket.';
window.chatWs.send(JSON.stringify({text: inputText}));
```

最后，连调浏览器和服务器端，如果一切无误，可以开多个不同的浏览器测试WebSocket的推送和广播：

和上一节我们介绍的异步处理类似，Servlet的线程模型并不适合大规模的长链接。基于NIO的Netty等框架更适合处理WebSocket长链接，我们将在后面介绍。

练习

使用WebSocket编写一个聊天室

小结

在Servlet中使用WebSocket需要3.1及以上版本；

通过spring-websocket可以简化WebSocket的开发。

Spring框架不仅提供了标准的IoC容器、AOP支持、数据库访问以及WebMVC等标准功能，还可以非常方便地集成许多常用的第三方组件：

- 可以集成JavaMail发送邮件；
- 可以集成JMS消息服务；
- 可以集成Quartz实现定时任务；
- 可以集成Redis等服务。

本章我们介绍如何在Spring中简单快捷地集成这些第三方组件。

我们在发送Email和接收Email中已经介绍了如何通过JavaMail来收发电子邮件。在Spring中，同样可以集成JavaMail。

因为在服务器端，主要以发送邮件为主，例如在注册成功、登录时、购物付款后通知用户，基本上不会遇到接收用户邮件的情况，所以本节我们只讨论如何在Spring中发送邮件。

在Spring中，发送邮件最终也是需要JavaMail，Spring只对JavaMail做了一点简单的封装，目的是简化代码。为了在Spring中集成JavaMail，我们在pom.xml中添加以下依赖：

- org.springframework:spring-context-support:5.2.0.RELEASE
- javax.mail:javax.mail-api:1.6.2
- com.sun.mail:javax.mail:1.6.2

以及其他Web相关依赖。

我们希望用户在注册成功后能收到注册邮件，为此，我们先定义一个JavaMailSender的Bean：

```
@Bean
JavaMailSender createJavaMailSender(
        // smtp.properties:
        @Value("${smtp.host}") String host,
        @Value("${smtp.port}") int port,
        @Value("${smtp.auth}") String auth,
        @Value("${smtp.username}") String username,
        @Value("${smtp.password}") String password,
        @Value("${smtp.debug:true}") String debug)
{
    var mailSender = new JavaMailSenderImpl();
    mailSender.setHost(host);
    mailSender.setPort(port);
    mailSender.setUsername(username);
    mailSender.setPassword(password);
    Properties props = mailSender.getJavaMailProperties();
    props.put("mail.transport.protocol", "smtp");
    props.put("mail.smtp.auth", auth);
    if (port == 587) {
        props.put("mail.smtp.starttls.enable", "true");
    }
    if (port == 465) {
        props.put("mail.smtp.socketFactory.port", "465");
        props.put("mail.smtp.socketFactory.class", "javax.net.ssl.SSLSocketFactory");
    }
    props.put("mail.debug", debug);
    return mailSender;
}
```

这个`JavaMailSender`接口的实现类是`JavaMailSenderImpl`，初始化时，传入的参数与JavaMail是完全一致的。

另外注意到需要注入的属性是从`smtp.properties`中读取的，因此，`AppConfig`导入的就不止一个`.properties`文件，可以导入多个：

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties", "classpath:/smtp.properties" })
public class AppConfig {
    ...
}
```

下一步是封装一个`MailService`，并定义`sendRegistrationMail()`方法：

```
@Component
public class MailService {
    @Value("${smtp.from}")
    String from;

    @Autowired
    JavaMailSender mailSender;

    public void sendRegistrationMail(User user) {
        try {
            MimeMessage mimeMessage = mailSender.createMimeMessage();
            MimeMessageHelper helper = new MimeMessageHelper(mimeMessage, "utf-8");
            helper.setFrom(from);
            helper.setTo(user.getEmail());
            helper.setSubject("Welcome to Java course!");
            String html = String.format("<p>Hi, %s,</p><p>Welcome to Java course!</p><p>Sent at %s</p>", user.getName(), LocalDateTime.now());
            helper.setText(html, true);
            mailSender.send(mimeMessage);
        } catch (MessagingException e) {
            throw new RuntimeException(e);
        }
    }
}
```

观察上述代码，`MimeMessage`是JavaMail的邮件对象，而`MimeMessageHelper`是Spring提供的用于简化设置MimeMessage的类，比如我们设置HTML邮件就可以直接调用`setText(String text, boolean html)`方法，而不必再调用比较繁琐的JavaMail接口方法。

最后一步是调用`JavaMailSender.send()`方法把邮件发送出去。

在MVC的某个Controller方法中，当用户注册成功后，我们就启动一个新线程来异步发送邮件：

```
User user = userService.register(email, password, name);
logger.info("user registered: {}", user.getEmail());
// send registration mail:
new Thread(() -> {
    mailService.sendRegistrationMail(user);
}).start();
```

因为发送邮件是一种耗时的任务，从几秒到几分钟不等，因此，异步发送是保证页面能快速显示的必要措施。这里我们直接启动了一个新的线程，但实际上还有更优化的方法，我们在下一节讨论。

练习

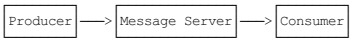
使用Spring发送邮件

小结

Spring可以集成JavaMail，通过简单的封装，能简化邮件发送代码。其核心是定义一个`JavaMailSender`的Bean，然后调用其`send()`方法。

JMS即Java Message Service，是JavaEE的消息服务接口。JMS主要有两个版本：1.1和2.0。2.0和1.1相比，主要是简化了收发消息的代码。

所谓消息服务，就是两个进程之间，通过消息服务器传递消息：



使用消息服务，而不是直接调用对方的API，它的好处是：

- 双方各自无需知晓对方的存在，消息可以异步处理，因为消息服务器会在Consumer离线的时候自动缓存消息；
- 如果Producer发送的消息频率高于Consumer的处理能力，消息可以积压在消息服务器，不至于压垮Consumer；
- 通过一个消息服务器，可以连接多个Producer和多个Consumer。

因为消息服务在各类应用程序中非常有用，所以JavaEE专门定义了JMS规范。注意到JMS是一组接口定义，如果我们要使用JMS，还需要选择一个具体的JMS产品。常用的JMS服务器有开源的ActiveMQ，商业服务器如WebLogic、WebSphere等也内置了JMS支持。这里我们选择开源的ActiveMQ作为JMS服务器，因此，在开发JMS之前我们必须首先安装ActiveMQ。

现在问题来了：从官网下载ActiveMQ时，蹦出一个页面，让我们选择ActiveMQ Classic或者ActiveMQ Artemis，这两个是什么关系，又有什么区别？

实际上ActiveMQ Classic原来就叫ActiveMQ，是Apache开发的基于JMS 1.1的消息服务器，目前稳定版本号是5.x，而ActiveMQ Artemis是由RedHat捐赠的HornetQ服务器代码的基础上开发的，目前稳定版本号是2.x。和ActiveMQ Classic相比，Artemis版的代码与Classic完全不同，并且，它支持JMS 2.0，使用基于Netty的异步IO，大大提升了性能。此外，Artemis不仅提供了JMS接口，它还提供了AMQP接口，STOMP接口和物联网使用的MQTT接口。选择Artemis，相当于一鱼四吃。

所以，我们这里直接选择ActiveMQ Artemis。从官网下载最新的2.x版本，解压后设置环境变量ARTEMIS_HOME，指向Artemis根目录，例如C:\Apps\artemis，然后，把ARTEMIS_HOME/bin加入PATH环境变量：

- Windows下添加%ARTEMIS_HOME%\bin到Path路径；
- Mac和Linux下添加$ARTEMIS_HOME/bin到PATH路径。

Artemis有个很好的设计，就是它把程序和数据完全分离了。我们解压后的ARTEMIS_HOME目录是程序目录，要启动一个Artemis服务，还需要创建一个数据目录。我们把数据目录直接设定在项目spring-integration-jms的jms-data目录下。执行命令artemis create jms-data：

```
$ pwd
/Users/liaoxuefeng/workspace/spring-integration-jms

$ artemis create jms-data
Creating ActiveMQ Artemis instance at: /Users/liaoxuefeng/workspace/spring-integration-jms/jms-data

--user: is a mandatory property!
Please provide the default username:
admin

--password: is mandatory with this configuration:
Please provide the default password:
********

--allow-anonymous | --require-login: is a mandatory property!
Allow anonymous access?, valid values are Y,N,True,False
N

Auto tuning journal ...
done! Your system can make 0.09 writes per millisecond, your journal-buffer-timeout will be 11392000

You can now start the broker by executing:

   "/Users/liaoxuefeng/workspace/spring-integration-jms/jms-data/bin/artemis" run

Or you can run the broker in the background using:

   "/Users/liaoxuefeng/workspace/spring-integration-jms/jms-data/bin/artemis-service" start
```

在创建过程中，会要求输入连接用户和口令，这里我们设定admin和password，以及是否允许匿名访问（这里选择N）。

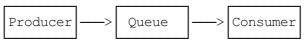
此数据目录jms-data不仅包含消息数据、日志，还自动创建了两个启动服务的命令bin/artemis和bin/artemis-service，前者在前台启动运行，按Ctrl+C结束，后者会一直在后台运行。

我们把目录切换到jms-data/bin，直接运行artemis run即可启动Artemis服务：

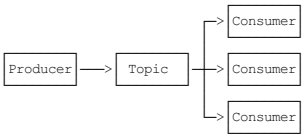
```
$ ./artemis run

  _        _               _
 / \  ____| |_  ___ __  __(_) _____
/ _ \|  _ \ __|/ _ \\ \/ /| |/ ___/
/ ___ \ | \/ |_/  __/ |\/| | |\___ \
/_/   \_\|   \__\____|_|  |_|_|/___ /
 Apache ActiveMQ Artemis 2.13.0

...

2020-06-02 07:50:21,718 INFO  [org.apache.activemq.artemis] AMQ241001: HTTP Server started at http://localhost:8161
2020-06-02 07:50:21,718 INFO  [org.apache.activemq.artemis] AMQ241002: Artemis Jolokia REST API available at http://localhost:8161/console/jolokia
2020-06-02 07:50:21,719 INFO  [org.apache.activemq.artemis] AMQ241004: Artemis Console available at http://localhost:8161/console
```

启动成功后，Artemis提示可以通过URLhttp://localhost:8161/console访问管理后台。注意*不要关闭命令行窗口*。

在编写JMS代码之前，我们首先得理解JMS的消息模型。JMS把生产消息的一方称为Producer，处理消息的一方称为Consumer。有两种类型的消息通道，一种是Queue：



一种是Topic：



它们的区别在于，Queue是一种一对一的通道，如果Consumer离线无法处理消息时，Queue会把消息存起来，等Consumer再次连接的时候发给它。设定了持久化机制的Queue不会丢失消息。如果有多个Consumer接入同一个Queue，那么它们等效于以集群方式处理消息，例如，发送方发送的消息是A，B，C，D，E，F，两个Consumer可能分别收到A，C，E和B，D，F，即每个消息只会交给其中一个Consumer处理。

Topic则是一种一对多通道。一个Producer发出的消息，会被多个Consumer同时收到，即每个Consumer都会收到一份完整的消息流。那么问题来了：如果某个Consumer暂时离线，过一段时间后又上线了，那么在它离线期间产生的消息还能不能收到呢？

这取决于消息服务器对Topic类型消息的持久化机制。如果消息服务器不存储Topic消息，那么离线的Consumer会丢失部分离线时期的消息，如果消息服务器存储了Topic消息，那么离线的Consumer可以收到自上次离线时刻开始后产生的所有消息。JMS规范通过Consumer指定一个持久化订阅可以在上线后收取所有离线期间的消息，如果指定的是非持久化订阅，那么离线期间的消息会全部丢失。

细心的童鞋可以看出来，如果一个Topic的消息全部都持久化了，并且只有一个Consumer，那么它和Queue其实是一样的。实际上，很多消息服务器内部都只有Topic类型的消息架构，Queue可以通过Topic“模拟”出来。

无论是Queue还是Topic，对Producer没有什么要求。多个Producer也可以写入同一个Queue或者Topic，此时消息服务器内部会自动排序确保消息总是有序的。

以上是消息服务的基本模型。具体到某个消息服务器时，Producer和Consumer通常是通过TCP连接消息服务器，在编写JMS程序时，又会遇到ConnectionFactory、Connection、Session等概念，其实这和JDBC连接是类似的：

- ConnectionFactory：代表一个到消息服务器的连接池，类似JDBC的DataSource；
- Connection：代表一个到消息服务器的连接，类似JDBC的Connection；
- Session：代表一个经过认证后的连接会话；
- Message：代表一个消息对象。

在JMS 1.1中，发送消息的典型代码如下：

```
try {
    Connection connection = null;
    try {
        // 创建连接:
        connection = connectionFactory.createConnection();
        // 创建会话:
        Session session = connection.createSession(false,Session.AUTO_ACKNOWLEDGE);
        // 创建一个Producer并关联到某个Queue:
        MessageProducer messageProducer = session.createProducer(queue);
        // 创建一个文本消息:
        TextMessage textMessage = session.createTextMessage(text);
        // 发送消息:
        messageProducer.send(textMessage);
    } finally {
```

```
        // 关闭连接:
        if (connection != null) {
            connection.close();
        }
    }
} catch (JMSException ex) {
    // 处理JMS异常
}
```

JMS 2.0改进了一些API接口，发送消息变得更简单：

```
try (JMSContext context = connectionFactory.createContext()) {
    context.createProducer().send(queue, text);
}
```

`JMSContext`实现了`AutoCloseable`接口，可以使用`try(resource)`语法，代码更简单。

有了以上预备知识，我们就可以开始开发JMS应用了。

首先，我们在`pom.xml`中添加如下依赖：

- org.springframework:spring-jms:5.2.0.RELEASE
- javax.jms:javax.jms-api:2.0.1
- org.apache.activemq:artemis-jms-client:2.13.0
- io.netty:netty-handler-proxy:4.1.45.Final

在AppConfig中，通过`@EnableJms`让Spring自动扫描JMS相关的Bean，并加载JMS配置文件`jms.properties`：

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableJms // 启用JMS
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties", "classpath:/jms.properties" })
public class AppConfig {
    ...
}
```

首先要创建的Bean是ConnectionFactory，即连接消息服务器的连接池：

```
@Bean
ConnectionFactory createJMSConnectionFactory(
    @Value("${jms.uri:tcp://localhost:61616}") String uri,
    @Value("${jms.username:admin}") String username,
    @Value("${jms.password:password}") String password)
{
    return new ActiveMQJMSConnectionFactory(uri, username, password);
}
```

因为我们使用的消息服务器是ActiveMQ Artemis，所以`ConnectionFactory`的实现类就是消息服务器提供的`ActiveMQJMSConnectionFactory`，它需要的参数均由配置文件读取后传入，并设置了默认值。

我们再创建一个`JmsTemplate`，它是Spring提供的一个工具类，和`JdbcTemplate`类似，可以简化发送消息的代码：

```
@Bean
JmsTemplate createJmsTemplate(@Autowired ConnectionFactory connectionFactory) {
    return new JmsTemplate(connectionFactory);
}
```

下一步要创建的是`JmsListenerContainerFactory`，

```
@Bean("jmsListenerContainerFactory")
DefaultJmsListenerContainerFactory createJmsListenerContainerFactory(@Autowired ConnectionFactory connectionFactory) {
    var factory = new DefaultJmsListenerContainerFactory();
    factory.setConnectionFactory(connectionFactory);
    return factory;
}
```

除了必须指定Bean的名称为`jmsListenerContainerFactory`外，这个Bean的作用是处理和Consumer相关的Bean。我们先跳过它的原理，继续编写`MessagingService`来发送消息：

```
@Component
public class MessagingService {
    @Autowired ObjectMapper objectMapper;
    @Autowired JmsTemplate jmsTemplate;

    public void sendMailMessage(MailMessage msg) throws Exception {
        String text = objectMapper.writeValueAsString(msg);
        jmsTemplate.send("jms/queue/mail", new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage(text);
            }
        });
    }
}
```

JMS的消息类型支持以下几种：

- TextMessage：文本消息；
- BytesMessage：二进制消息；
- MapMessage：包含多个Key-Value对的消息；
- ObjectMessage：直接序列化Java对象的消息；
- StreamMessage：一个包含基本类型序列的消息。

最常用的是发送基于JSON的文本消息，上述代码通过`JmsTemplate`创建一个`TextMessage`并发送到名称为`jms/queue/mail`的Queue。

注意：Artemis消息服务器默认配置下会自动创建Queue，因此不必手动创建一个名为`jms/queue/mail`的Queue，但不是所有的消息服务器都会自动创建Queue，生产环境的消息服务器通常会关闭自动创建功能，需要手动创建Queue。

再注意到`MailMessage`是我们自己定义的一个JavaBean，真正的JMS消息是创建的`TextMessage`，它的内容是JSON。

当用户注册成功后，我们就调用`MessagingService.sendMailMessage()`发送一条JMS消息，此代码十分简单，这里不再贴出。

下面我们要详细讨论的是如何处理消息，即编写Consumer。从理论上讲，可以创建另一个Java进程来处理消息，但对于我们这个简单的Web程序来说没有必要，直接在同一个Web应用中接收并处理消息即可。

处理消息的核心代码是编写一个Bean，并在处理方法上标注`@JmsListener`：

```
@Component
public class MailMessageListener {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired ObjectMapper objectMapper;
    @Autowired MailService mailService;

    @JmsListener(destination = "jms/queue/mail", concurrency = "10")
    public void onMailMessageReceived(Message message) throws Exception {
        logger.info("received message: " + message);
```

```
        if (message instanceof TextMessage) {
            String text = ((TextMessage) message).getText();
            MailMessage mm = objectMapper.readValue(text, MailMessage.class);
            mailService.sendRegistrationMail(mm);
        } else {
            logger.error("unable to process non-text message!");
        }
    }
}
```

注意到`@JmsListener`指定了Queue的名称，因此，凡是发到此Queue的消息都会被这个`onMailMessageReceived()`方法处理，方法参数是JMS的`Message`接口，我们通过强制转型为`TextMessage`并提取JSON，反序列化后获得自定义的JavaBean，也就获得了发送邮件所需的所有信息。

下面问题来了：Spring处理JMS消息的流程是什么？

如果我们直接调用JMS的API来处理消息，那么编写的代码大致如下：

```
// 创建JMS连接:
Connection connection = connectionFactory.createConnection();
// 创建会话:
Session session = connection.createSession(false, Session.AUTO_ACKNOWLEDGE);
// 创建一个Consumer:
MessageConsumer consumer = session.createConsumer(queue);
// 为Consumer指定一个消息处理器:
consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        // 在此处理消息...
    }
});
// 启动接收消息的循环:
connection.start();
```

我们自己编写的`MailMessageListener.onMailMessageReceived()`相当于消息处理器：

```
consumer.setMessageListener(new MessageListener() {
    public void onMessage(Message message) {
        mailMessageListener.onMailMessageReceived(message);
    }
});
```

所以，Spring根据`AppConfig`的注解`@EnableJms`自动扫描带有`@JmsListener`的Bean方法，并为其创建一个`MessageListener`把它包装起来。

注意到前面我们还创建了一个`JmsListenerContainerFactory`的Bean，它的作用就是为每个`MessageListener`创建`MessageConsumer`并启动消息接收循环。

再注意到`@JmsListener`还有一个`concurrency`参数，`10`表示可以最多同时并发处理10个消息，`5-10`表示并发处理的线程可以在5~10之间调整。

因此，Spring在通过`MessageListener`接收到消息后，并不是直接调用`mailMessageListener.onMailMessageReceived()`，而是用线程池调用，因此，要时刻牢记，`onMailMessageReceived()`方法可能被多线程并发执行，一定要保证线程安全。

我们总结一下Spring接收消息的步骤：

通过`JmsListenerContainerFactory`配合`@EnableJms`扫描所有`@JmsListener`方法，自动创建`MessageConsumer`、`MessageListener`以及线程池，启动消息循环接收处理消息，最终由我们自己编写的`@JmsListener`方法处理消息，可能会由多线程同时并发处理。

要验证消息发送和处理，我们注册一个新用户，可以看到如下日志输出：

```
2020-06-02 08:04:27 INFO  c.i.learnjava.web.UserController - user registered: bob@example.com
2020-06-02 08:04:27 INFO  c.i.l.service.MailMessageListener - received message: ActiveMQMessage[ID:9fc5...]:PERSISTENT/ClientMessageImpl[messageID=983, durable=true, address=jms/queue/
2020-06-02 08:04:27 INFO  c.i.learnjava.service.MailService - [send mail] sending registration mail to bob@example.com...
2020-06-02 08:04:30 INFO  c.i.learnjava.service.MailService - [send mail] registration mail was sent to bob@example.com.
```

可见，消息被成功发送到Artemis，然后在很短的时间内被接收处理了。

使用消息服务对发送Email进行改造的好处是，发送Email的能力通常是有限的，通过JMS消息服务，如果短时间内需要给大量用户发送Email，可以先把消息堆积在JMS服务器上慢慢发送，对于批量发送邮件、短信等尤其有用。

练习

使用JMS

小结

JMS是Java消息服务，可以通过JMS服务器实现消息的异步处理。

消息服务主要解决Producer和Consumer生产和处理速度不匹配的问题。

在很多应用程序中，经常需要执行定时任务。例如，每天或每月给用户发送账户汇总报表，定期检查并发送系统状态报告，等等。

定时任务我们在使用线程池一节中已经讲到了，Java标准库本身就提供了定时执行任务的功能。在Spring中，使用定时任务更简单，不需要手写线程池相关代码，只需要两个注解即可。

我们还是以实际代码为例，建立工程`spring-integration-schedule`，无需额外的依赖，我们可以直接在`AppConfig`中加上`@EnableScheduling`就开启了定时任务的支持：

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableScheduling
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties", "classpath:/task.properties" })
public class AppConfig {
    ...
}
```

接下来，我们可以直接在一个Bean中编写一个`public void`无参数方法，然后加上`@Scheduled`注解：

```
@Component
public class TaskService {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Scheduled(initialDelay = 60_000, fixedRate = 60_000)
    public void checkSystemStatusEveryMinute() {
        logger.info("Start check system status...");
    }
}
```

上述注解指定了启动延迟60秒，并以60秒的间隔执行任务。现在，我们直接运行应用程序，就可以在控制台看到定时任务打印的日志：

```
2020-06-03 18:47:32 INFO  [pool-1-thread-1] c.i.learnjava.service.TaskService - Start check system status...
2020-06-03 18:48:32 INFO  [pool-1-thread-1] c.i.learnjava.service.TaskService - Start check system status...
2020-06-03 18:49:32 INFO  [pool-1-thread-1] c.i.learnjava.service.TaskService - Start check system status...
```

如果没有看到定时任务的日志，需要检查：

- 是否忘记了在`AppConfig`中标注`@EnableScheduling`；
- 是否忘记了在定时任务的方法所在的class标注`@Component`。

除了可以使用fixedRate外，还可以使用fixedDelay，两者的区别我们已经在使用线程池一节中讲过，这里不再重复。

有的童鞋在实际开发中会遇到一个问题，因为Java的注解全部是常量，写死了fixedDelay=30000，如果根据实际情况要改成60秒怎么办，只能重新编译？

我们可以把定时任务的配置放到配置文件中，例如task.properties：

```
task.checkDiskSpace=30000
```

这样就可以随时修改配置文件而无需动代码。但是在代码中，我们需要用fixedDelayString取代fixedDelay：

```
@Component
public class TaskService {
    ...

    @Scheduled(initialDelay = 30_000, fixedDelayString = "${task.checkDiskSpace:30000}")
    public void checkDiskSpaceEveryMinute() {
        logger.info("Start check disk space...");
    }
}
```

注意到上述代码的注解参数fixedDelayString是一个属性占位符，并配有默认值30000，Spring在处理@Scheduled注解时，如果遇到String，会根据占位符自动用配置项替换，这样就可以灵活地修改定时任务的配置。

此外，fixedDelayString还可以使用更易读的Duration，例如：

```
@Scheduled(initialDelay = 30_000, fixedDelayString = "${task.checkDiskSpace:PT2M30S}")
```

以字符串PT2M30S表示的Duration就是2分30秒，请参考LocalDateTime一节的Duration相关部分。

多个@Scheduled方法完全可以放到一个Bean中，这样便于统一管理各类定时任务。

使用Cron任务

还有一类定时任务，它不是简单的重复执行，而是按时间触发，我们把这类任务称为Cron任务，例如：

- 每天凌晨2:15执行报表任务；
- 每个工作日12:00执行特定任务；
- ……

Cron源自Unix/Linux系统自带的crond守护进程，以一个简洁的表达式定义任务触发时间。在Spring中，也可以使用Cron表达式来执行Cron任务，在Spring中，它的格式是：

```
秒 分 小时 天 月份 星期 年
```

年是可以忽略的，通常不写。每天凌晨2:15执行的Cron表达式就是：

```
0 15 2 * * *
```

每个工作日12:00执行的Cron表达式就是：

```
0 0 12 * * MON-FRI
```

每个月1号，2号，3号和10号12:00执行的Cron表达式就是：

```
0 0 12 1-3,10 * *
```

在Spring中，我们定义一个每天凌晨2:15执行的任务：

```
@Component
public class TaskService {
    ...

    @Scheduled(cron = "${task.report:0 15 2 * * *}")
    public void cronDailyReport() {
        logger.info("Start daily report task...");
    }
}
```

Cron任务同样可以使用属性占位符，这样修改起来更加方便。

Cron表达式还可以表达每10分钟执行，例如：

```
0 */10 * * * *
```

这样，在每个小时的0:00，10:00，20:00，30:00，40:00，50:00均会执行任务，实际上它可以取代fixedRate类型的定时任务。

集成Quartz

在Spring中使用定时任务和Cron任务都十分简单，但是要注意到，这些任务的调度都是在每个JVM进程中的。如果在本机启动两个进程，或者在多台机器上启动应用，这些进程的定时任务和Cron任务都是独立运行的，互不影响。

如果一些定时任务要以集群的方式运行，例如每天23:00执行检查任务，只需要集群中的一台运行即可，这个时候，可以考虑使用Quartz。

Quartz可以配置一个JDBC数据源，以便存储所有的任务调度计划以及任务执行状态。也可以使用内存来调度任务，但这样配置就和使用Spring的调度没啥区别了，额外集成Quartz的意义就不大。

Quartz的JDBC配置比较复杂，Spring对其也有一定的支持。要详细了解Quartz的集成，请参考Spring的文档。

思考：如果不使用Quartz的JDBC配置，多个Spring应用同时运行时，如何保证某个任务只在某一台机器执行？

练习

使用Scheduler

小结

Spring内置定时任务和Cron任务的支持，编写调度任务十分方便。

在Spring中，可以方便地集成JMX。

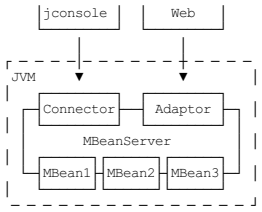
那么第一个问题来了：什么是JMX？

JMX是Java Management Extensions，它是一个Java平台的管理和监控接口。为什么要搞JMX呢？因为在所有的应用程序中，对运行中的程序进行监控都是非常重要的，Java应用程序也不例外。我们肯定希望知道Java应用程序当前的状态，例如，占用了多少内存，分配了多少内存，当前有多少活动线程，有多少休眠线程等等。如何获取这些信息呢？

为了标准化管理和监控，Java平台使用JMX作为管理和监控的标准接口，任何程序，只要按JMX规范访问这个接口，就可以获取所有管理与监控信息。

实际上，常用的运维监控如Zabbix、Nagios等工具对JVM本身的监控都是通过JMX获取的信息。

因为JMX是一个标准接口，不但可以用于管理JVM，还可以管理应用程序自身。下图是JMX的架构：



JMX把所有被管理的资源都称为MBean（Managed Bean），这些MBean全部由MBeanServer管理，如果要访问MBean，可以通过MBeanServer对外提供的访问接口，例如通过RMI或HTTP访问。

注意到使用JMX不需要安装任何额外组件，也不需要第三方库，因为MBeanServer已经内置在JavaSE标准库中了。JavaSE还提供了一个`jconsole`程序，用于通过RMI连接到MBeanServer，这样就可以管理整个Java进程。

除了JVM会把自身的各种资源以MBean注册到JMX中，我们自己的配置、监控信息也可以作为MBean注册到JMX，这样，管理程序就可以直接控制我们暴露的MBean。因此，应用程序使用JMX，只需要两步：

1. 编写MBean提供管理接口和监控数据；
2. 注册MBean。

在Spring应用程序中，使用JMX只需要一步：

1. 编写MBean提供管理接口和监控数据。

第二步注册的过程由Spring自动完成。我们以实际工程为例，首先在`AppConfig`中加上`@EnableMBeanExport`注解，告诉Spring自动注册MBean：

```
@Configuration
@ComponentScan
@EnableWebMvc
@EnableMBeanExport // 自动注册MBean
@EnableTransactionManagement
@PropertySource({ "classpath:/jdbc.properties" })
public class AppConfig {
    ...
}
```

剩下的全部工作就是编写MBean。我们以实际问题为例，假设我们希望给应用程序添加一个IP黑名单功能，凡是在黑名单中的IP禁止访问，传统的做法是定义一个配置文件，启动的时候读取：

```
# blacklist.txt
1.2.3.4
5.6.7.8
2.2.3.4
...
```

如果要修改黑名单怎么办？修改配置文件，然后重启应用程序。

但是每次都重启应用程序实在是太麻烦了，能不能不重启应用程序？可以自己写一个定时读取配置文件的功能，检测到文件改动时自动重新读取。

上述需求本质上是在应用程序运行期间对参数、配置等进行热更新并要求尽快生效。如果以JMX的方式实现，我们不必自己编写自动重新读取等任何代码，只需要提供一个符合JMX标准的MBean来存储配置即可。

还是以IP黑名单为例，JMX的MBean通常以MBean结尾，因此我们遵循标准命名规范，首先编写一个`BlacklistMBean`：

```
public class BlacklistMBean {
    private Set<String> ips = new HashSet<>();

    public String[] getBlacklist() {
        return ips.toArray(String[]::new);
    }

    public void addBlacklist(String ip) {
        ips.add(ip);
    }

    public void removeBlacklist(String ip) {
        ips.remove(ip);
    }

    public boolean shouldBlock(String ip) {
        return ips.contains(ip);
    }
}
```

这个MBean没什么特殊的，它的逻辑和普通Java类没有任何区别。

下一步，我们要使用JMX的客户端来实时热更新这个MBean，所以要给它加上一些注解，让Spring能根据注解自动把相关方法注册到MBeanServer中：

```
@Component
@ManagedResource(objectName = "sample:name=blacklist", description = "Blacklist of IP addresses")
public class BlacklistMBean {
    private Set<String> ips = new HashSet<>();

    @ManagedAttribute(description = "Get IP addresses in blacklist")
    public String[] getBlacklist() {
        return ips.toArray(String[]::new);
    }

    @ManagedOperation
    @ManagedOperationParameter(name = "ip", description = "Target IP address that will be added to blacklist")
    public void addBlacklist(String ip) {
        ips.add(ip);
    }

    @ManagedOperation
    @ManagedOperationParameter(name = "ip", description = "Target IP address that will be removed from blacklist")
    public void removeBlacklist(String ip) {
        ips.remove(ip);
    }

    public boolean shouldBlock(String ip) {
        return ips.contains(ip);
    }
}
```

观察上述代码，`BlacklistMBean`首先是一个标准的Spring管理的Bean，其次，添加了`@ManagedResource`表示这是一个MBean，将要被注册到JMX。objectName指定了这个MBean的名字，通常以`company:name=Xxx`来分类MBean。

对于属性，使用`@ManagedAttribute`注解标注。上述MBean只有get属性，没有set属性，说明这是一个只读属性。

对于操作，使用`@ManagedOperation`注解标准。上述MBean定义了两个操作：`addBlacklist()`和`removeBlacklist()`，其他方法如`shouldBlock()`不会被暴露给JMX。

使用MBean和普通Bean是完全一样的。例如，我们在`BlacklistInterceptor`对IP进行黑名单拦截：

```
@Order(1)
@Component
public class BlacklistInterceptor implements HandlerInterceptor {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    BlacklistMBean blacklistMBean;

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)
            throws Exception {
        String ip = request.getRemoteAddr();
        logger.info("check ip address {}...", ip);
        // 是否在黑名单中:
        if (blacklistMBean.shouldBlock(ip)) {
            logger.warn("will block ip {} for it is in blacklist.", ip);
            // 发送403错误响应:
            response.sendError(403);
            return false;
        }
        return true;
    }
}
```

下一步就是正常启动Web应用程序，不要关闭它，我们打开另一个命令行窗口，输入`jconsole`启动JavaSE自带的一个JMX客户端程序：

通过jconsole连接到一个Java进程最简单的方法是直接在Local Process中找到正在运行的`AppConfig`，点击Connect即可连接到我们当前正在运行的Web应用，在jconsole中可直接看到内存、CPU等资源的监控。

我们点击MBean，左侧按分类列出所有MBean，可以在`java.lang`查看内存等信息：

细心的童鞋可以看到HikariCP连接池也是通过JMX监控的。

在`sample`中可以看到我们自己的MBean，点击可查看属性`blacklist`：

点击`Operations-addBlacklist`，可以填入`127.0.0.1`并点击`addBlacklist`按钮，相当于jconsole通过JMX接口，调用了我们自己的`BlacklistMBean`的`addBlacklist()`方法，传入的参数就是填入的`127.0.0.1`：

再次查看属性`blacklist`，可以看到结果已经更新了：

我们可以在浏览器中测试一下黑名单功能是否已生效：

可见，`127.0.0.1`确实被添加到了黑名单，后台日志打印如下：

```
2020-06-06 20:22:12 INFO  c.i.l.web.BlacklistInterceptor - check ip address 127.0.0.1...
2020-06-06 20:22:12 WARN  c.i.l.web.BlacklistInterceptor - will block ip 127.0.0.1 for it is in blacklist.
```

注意：如果使用IPv6，那么需要把`0:0:0:0:0:0:0:1`这个本机地址加到黑名单。

如果从jconsole中调用`removeBlacklist`移除`127.0.0.1`，刷新浏览器可以看到又允许访问了。

使用jconsole直接通过Local Process连接JVM有个限制，就是jconsole和正在运行的JVM必须在同一台机器。如果要远程连接，首先要打开JMX端口。我们在启动`AppConfig`时，需要传入以下JVM启动参数：

- -Dcom.sun.management.jmxremote.port=19999
- -Dcom.sun.management.jmxremote.authenticate=false
- -Dcom.sun.management.jmxremote.ssl=false

第一个参数表示在19999端口监听JMX连接，第二个和第三个参数表示无需验证，不使用SSL连接，在开发测试阶段比较方便，生产环境必须指定验证方式并启用SSL。详细参数可参考Oracle官方文档。这样jconsole可以用`ip:19999`的远程方式连接JMX。连接后的操作是完全一样的。

许多JavaEE服务器如JBoss的管理后台都是通过JMX提供管理接口，并由Web方式访问，对用户更加友好。

练习

编写一个MBean统计当前注册用户数量，并在jconsole中查看：

编写MBean

小结

在Spring中使用JMX需要：

- 通过`@EnableMBeanExport`启用自动注册MBean;
- 编写MBean并实现管理属性和管理操作。

我们已经在前面详细介绍了Spring框架，它的主要功能包括IoC容器、AOP支持、事务支持、MVC开发以及强大的第三方集成功能等。

那么，Spring Boot又是什么？它和Spring是什么关系？

Spring Boot是一个基于Spring的套件，它帮我们预组装了Spring的一系列组件，以便以尽可能少的代码和配置来开发基于Spring的Java应用程序。

以汽车为例，如果我们想组装一辆汽车，我们需要发动机、传动、轮胎、底盘、外壳、座椅、内饰等各种部件，然后把它们装配起来。Spring就相当于提供了一系列这样的部件，但是要装好汽车上路，还需要我们自己动手。而Spring Boot则相当于已经帮我们预装好了一辆可以上路的汽车，如果有特殊的要求，例如把发动机从普通款换成涡轮增压款，可以通过修改配置或编写少量代码完成。

因此，Spring Boot和Spring的关系就是整车和零部件的关系，它们不是取代关系，试图跳过Spring直接学习Spring Boot是不可能的。

Spring Boot的目标就是提供一个开箱即用的应用程序架构，我们基于Spring Boot的预置结构继续开发，省时省力。

本章我们将详细介绍如何使用Spring Boot。

要了解Spring Boot，我们先来编写第一个Spring Boot应用程序，看看与前面我们编写的Spring应用程序有何异同。

我们新建一个`springboot-hello`的工程，创建标准的Maven目录结构如下：

```
springboot-hello
├── pom.xml
├── src
│   └── main
│       ├── java
│       └── resources
│           ├── application.yml
```

```
│           ├── logback-spring.xml
│           ├── static
│           └── templates
└── target
```

其中，在src/main/resources目录下，注意到几个文件：

application.yml

这是Spring Boot默认的配置文件，它采用YAML格式而不是.properties格式，文件名必须是application.yml而不是其他名称。

YAML格式比key=value格式的.properties文件更易读。比较一下两者的写法：

使用.properties格式：

```
# application.properties

spring.application.name=${APP_NAME:unnamed}

spring.datasource.url=jdbc:hsqldb:file:testdb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.dirver-class-name=org.hsqldb.jdbc.JDBCDriver

spring.datasource.hikari.auto-commit=false
spring.datasource.hikari.connection-timeout=3000
spring.datasource.hikari.validation-timeout=3000
spring.datasource.hikari.max-lifetime=60000
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.minimum-idle=1
```

使用YAML格式：

```
# application.yml

spring:
  application:
    name: ${APP_NAME:unnamed}
  datasource:
    url: jdbc:hsqldb:file:testdb
    username: sa
    password:
    driver-class-name: org.hsqldb.jdbc.JDBCDriver
    hikari:
      auto-commit: false
      connection-timeout: 3000
      validation-timeout: 3000
      max-lifetime: 60000
      maximum-pool-size: 20
      minimum-idle: 1
```

可见，YAML是一种层级格式，它和.properties很容易互相转换，它的优点是去掉了大量重复的前缀，并且更加易读。

也可以使用application.properties作为配置文件，但不如YAML格式简单。

使用环境变量

在配置文件中，我们经常使用如下的格式对某个key进行配置：

```
app:
  db:
    host: ${DB_HOST:localhost}
    user: ${DB_USER:root}
    password: ${DB_PASSWORD:password}
```

这种${DB_HOST:localhost}意思是，首先从环境变量查找DB_HOST，如果环境变量定义了，那么使用环境变量的值，否则，使用默认值localhost。

这使得我们在开发和部署时更加方便，因为开发时无需设定任何环境变量，直接使用默认值即本地数据库，而实际线上运行的时候，只需要传入环境变量即可：

```
$ DB_HOST=10.0.1.123 DB_USER=prod DB_PASSWORD=xxxx java -jar xxx.jar
```

logback-spring.xml

这是Spring Boot的logback配置文件名称（也可以使用logback.xml），一个标准的写法如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <include resource="org/springframework/boot/logging/logback/defaults.xml" />

    <appender name="CONSOLE" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>${CONSOLE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
    </appender>

    <appender name="APP_LOG" class="ch.qos.logback.core.rolling.RollingFileAppender">
        <encoder>
            <pattern>${FILE_LOG_PATTERN}</pattern>
            <charset>utf8</charset>
        </encoder>
          <file>app.log</file>
        <rollingPolicy class="ch.qos.logback.core.rolling.FixedWindowRollingPolicy">
            <maxIndex>1</maxIndex>
            <fileNamePattern>app.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="APP_LOG" />
    </root>
</configuration>
```

它主要通过<include resource="..." />引入了Spring Boot的一个缺省配置，这样我们就可以引用类似${CONSOLE_LOG_PATTERN}这样的变量。上述配置定义了一个控制台输出和文件输出，可根据需要修改。

static是静态文件目录，templates是模板文件目录，注意它们不再存放在src/main/webapp下，而是直接放到src/main/resources这个classpath目录，因为在Spring Boot中已经不需要专门的webapp目录了。

以上就是Spring Boot的标准目录结构，它完全是一个基于Java应用的普通Maven项目。

我们再来看源码目录结构：

```
src/main/java
```

```
└── com
    └── itranswarp
        └── learnjava
            ├── Application.java
            ├── entity
            │   └── User.java
            ├── service
            │   └── UserService.java
            └── web
                └── UserController.java
```

在存放源码的`src/main/java`目录中，Spring Boot对Java包的层级结构有一个要求。注意到我们的根package是`com.itranswarp.learnjava`，下面还有`entity`、`service`、`web`等子package。Spring Boot要求`main()`方法所在的启动类必须放到根package下，命名不做要求，这里我们以`Application.java`命名，它的内容如下：

```
@SpringBootApplication
public class Application {
    public static void main(String[] args) throws Exception {
        SpringApplication.run(Application.class, args);
    }
}
```

启动Spring Boot应用程序只需要一行代码加上一个注解`@SpringBootApplication`，该注解实际上又包含了：

- @SpringBootConfiguration
  - @Configuration
- @EnableAutoConfiguration
  - @AutoConfigurationPackage
- @ComponentScan

这样一个注解就相当于启动了自动配置和自动扫描。

我们再观察`pom.xml`，它的内容如下：

```
<project ...>
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.3.0.RELEASE</version>
    </parent>

    <modelVersion>4.0.0</modelVersion>
    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>springboot-hello</artifactId>
    <version>1.0-SNAPSHOT</version>

    <properties>
        <maven.compiler.source>11</maven.compiler.source>
        <maven.compiler.target>11</maven.compiler.target>
        <java.version>11</java.version>
        <pebble.version>3.1.2</pebble.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-jdbc</artifactId>
        </dependency>

        <!-- 集成Pebble View -->
        <dependency>
            <groupId>io.pebbletemplates</groupId>
            <artifactId>pebble-spring-boot-starter</artifactId>
            <version>${pebble.version}</version>
        </dependency>

        <!-- JDBC驱动 -->
        <dependency>
            <groupId>org.hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
        </dependency>
    </dependencies>
</project>
```

使用Spring Boot时，强烈推荐从`spring-boot-starter-parent`继承，因为这样就可以引入Spring Boot的预置配置。

紧接着，我们引入了依赖`spring-boot-starter-web`和`spring-boot-starter-jdbc`，它们分别引入了Spring MVC相关依赖和Spring JDBC相关依赖，无需指定版本号，因为引入的`<parent>`内已经指定了，只有我们自己引入的某些第三方jar包需要指定版本号。这里我们引入`pebble-spring-boot-starter`作为View，以及`hsqldb`作为嵌入式数据库。`hsqldb`已在`spring-boot-starter-jdbc`中预置了版本号`2.5.0`，因此此处无需指定版本号。

根据`pebble-spring-boot-starter`的文档，加入如下配置到`application.yml`：

```
pebble:
  # 默认为".pebble"，改为"":
  suffix:
  # 开发阶段禁用模板缓存:
  cache: false
```

对`Application`稍作改动，添加`WebMvcConfigurer`这个Bean：

```
@SpringBootApplication
public class Application {
    ...

    @Bean
    WebMvcConfigurer createWebMvcConfigurer(@Autowired HandlerInterceptor[] interceptors) {
        return new WebMvcConfigurer() {
            @Override
            public void addResourceHandlers(ResourceHandlerRegistry registry) {
                // 映射路径`/static/`到classpath路径:
                registry.addResourceHandler("/static/**")
                        .addResourceLocations("classpath:/static/");
            }
        };
    }
}
```

现在就可以直接运行`Application`，启动后观察Spring Boot的日志：

```
  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
```

```
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.0.RELEASE)

2020-06-08 08:47:23.152  INFO 32585 --- [           main] com.itranswarp.learnjava.Application     : Starting Application on xxx with PID 32585 (...)
2020-06-08 08:47:23.154  INFO 32585 --- [           main] com.itranswarp.learnjava.Application     : No active profile set, falling back to default profiles: default
2020-06-08 08:47:24.224  INFO 32585 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2020-06-08 08:47:24.235  INFO 32585 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2020-06-08 08:47:24.235  INFO 32585 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.35]
2020-06-08 08:47:24.309  INFO 32585 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2020-06-08 08:47:24.309  INFO 32585 --- [           main] o.s.web.context.ContextLoader            : Root WebApplicationContext: initialization completed in 1110 ms
2020-06-08 08:47:24.446  WARN 32585 --- [           main] com.zaxxer.hikari.HikariConfig           : HikariPool-1 - idleTimeout is close to or more than maxLifetime, disabling it.
2020-06-08 08:47:24.448  INFO 32585 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2020-06-08 08:47:24.753  INFO 32585 --- [           main] hsqldb.db.HSQLDB729157DF9B.ENGINE        : checkpointClose start
2020-06-08 08:47:24.754  INFO 32585 --- [           main] hsqldb.db.HSQLDB729157DF9B.ENGINE        : checkpointClose synched
2020-06-08 08:47:24.759  INFO 32585 --- [           main] hsqldb.db.HSQLDB729157DF9B.ENGINE        : checkpointClose script done
2020-06-08 08:47:24.763  INFO 32585 --- [           main] hsqldb.db.HSQLDB729157DF9B.ENGINE        : checkpointClose end
2020-06-08 08:47:24.767  INFO 32585 --- [           main] com.zaxxer.hikari.pool.PoolBase          : HikariPool-1 - Driver does not support get/set network timeout for connections. (fe
2020-06-08 08:47:24.770  INFO 32585 --- [           main] com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2020-06-08 08:47:24.971  INFO 32585 --- [           main] o.s.s.concurrent.ThreadPoolTaskExecutor  : Initializing ExecutorService 'applicationTaskExecutor'
2020-06-08 08:47:25.130  INFO 32585 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2020-06-08 08:47:25.138  INFO 32585 --- [           main] com.itranswarp.learnjava.Application     : Started Application in 2.68 seconds (JVM running for 3.097)
```

Spring Boot自动启动了嵌入式Tomcat，当看到`Started Application in xxx seconds`时，Spring Boot应用启动成功。

现在，我们在浏览器输入`localhost:8080`就可以直接访问页面。那么问题来了：

前面我们定义的数据源、声明式事务、JdbcTemplate在哪创建的？怎么就可以直接注入到自己编写的`UserService`中呢？

这些自动创建的Bean就是Spring Boot的特色：AutoConfiguration。

当我们引入`spring-boot-starter-jdbc`时，启动时会自动扫描所有的`XxxAutoConfiguration`：

- `DataSourceAutoConfiguration`：自动创建一个`DataSource`，其中配置项从`application.yml`的`spring.datasource`读取；
- `DataSourceTransactionManagerAutoConfiguration`：自动创建了一个基于JDBC的事务管理器；
- `JdbcTemplateAutoConfiguration`：自动创建了一个`JdbcTemplate`。

因此，我们自动得到了一个`DataSource`、一个`DataSourceTransactionManager`和一个`JdbcTemplate`。

类似的，当我们引入`spring-boot-starter-web`时，自动创建了：

- `ServletWebServerFactoryAutoConfiguration`：自动创建一个嵌入式Web服务器，默认是Tomcat；
- `DispatcherServletAutoConfiguration`：自动创建一个`DispatcherServlet`；
- `HttpEncodingAutoConfiguration`：自动创建一个`CharacterEncodingFilter`；
- `WebMvcAutoConfiguration`：自动创建若干与MVC相关的Bean。
- ...

引入第三方`pebble-spring-boot-starter`时，自动创建了：

- `PebbleAutoConfiguration`：自动创建了一个`PebbleViewResolver`。

Spring Boot大量使用`XxxAutoConfiguration`来使得许多组件被自动化配置并创建，而这些创建过程又大量使用了Spring的Conditional功能。例如，我们观察`JdbcTemplateAutoConfiguration`，它的代码如下：

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnClass({ DataSource.class, JdbcTemplate.class })
@ConditionalOnSingleCandidate(DataSource.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
@EnableConfigurationProperties(JdbcProperties.class)
@Import({ JdbcTemplateConfiguration.class, NamedParameterJdbcTemplateConfiguration.class })
public class JdbcTemplateAutoConfiguration {
}
```

当满足条件：

- `@ConditionalOnClass`：在classpath中能找到`DataSource`和`JdbcTemplate`；
- `@ConditionalOnSingleCandidate(DataSource.class)`：在当前Bean的定义中能找到唯一的`DataSource`；

该`JdbcTemplateAutoConfiguration`就会起作用。实际创建由导入的`JdbcTemplateConfiguration`完成：

```
@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(JdbcOperations.class)
class JdbcTemplateConfiguration {
    @Bean
    @Primary
    JdbcTemplate jdbcTemplate(DataSource dataSource, JdbcProperties properties) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        JdbcProperties.Template template = properties.getTemplate();
        jdbcTemplate.setFetchSize(template.getFetchSize());
        jdbcTemplate.setMaxRows(template.getMaxRows());
        if (template.getQueryTimeout() != null) {
            jdbcTemplate.setQueryTimeout((int) template.getQueryTimeout().getSeconds());
        }
        return jdbcTemplate;
    }
}
```

创建`JdbcTemplate`之前，要满足`@ConditionalOnMissingBean(JdbcOperations.class)`，即不存在`JdbcOperations`的Bean。

如果我们自己创建了一个`JdbcTemplate`，例如，在`Application`中自己写个方法：

```
@SpringBootApplication
public class Application {
    ...
    @Bean
    JdbcTemplate createJdbcTemplate(@Autowired DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
}
```

那么根据条件`@ConditionalOnMissingBean(JdbcOperations.class)`，Spring Boot就不会再创建一个重复的`JdbcTemplate`（因为`JdbcOperations`是`JdbcTemplate`的父类）。

可见，Spring Boot自动装配功能是通过自动扫描+条件装配实现的，这一套机制在默认情况下工作得很好，但是，如果我们要手动控制某个Bean的创建，就需要详细地了解Spring Boot自动创建的原理，很多时候还要跟踪`XxxAutoConfiguration`，以便设定条件使得某个Bean不会被自动创建。

练习

使用Spring Boot编写hello应用程序

小结

Spring Boot是一个基于Spring提供了开箱即用的一组套件，它可以让我们基于很少的配置和代码快速搭建出一个完整的应用程序。

Spring Boot有非常强大的AutoConfiguration功能，它是通过自动扫描+条件装配实现的。

在开发阶段，我们经常要修改代码，然后重启Spring Boot应用。经常手动停止再启动，比较麻烦。

Spring Boot提供了一个开发者工具，可以监控classpath路径上的文件。只要源码或配置文件发生修改，Spring Boot应用可以自动重启。在开发阶段，这个功能比较有用。

要使用这一开发者功能，我们只需添加如下依赖到pom.xml：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

然后，没有然后了。直接启动应用程序，然后试着修改源码，保存，观察日志输出，Spring Boot会自动重新加载。

默认配置下，针对/static、/public和/templates目录中的文件修改，不会自动重启，因为禁用缓存后，这些文件的修改可以实时更新。

练习

使用devtools检测修改并自动重启

小结

Spring Boot提供了一个开发阶段非常有用的spring-boot-devtools，能自动检测classpath路径上文件修改并自动重启。

我们在Maven的使用插件一节中介绍了如何使用maven-shade-plugin打包一个可执行的jar包。在Spring Boot应用中，打包更加简单，因为Spring Boot自带一个更简单的spring-boot-maven-plugin插件用来打包，我们只需要在pom.xml中加入以下配置：

```
<project ...>
    ...
    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

无需任何配置，Spring Boot的这款插件会自动定位应用程序的入口Class，我们执行以下Maven命令即可打包：

```
$ mvn clean package
```

以springboot-exec-jar项目为例，打包后我们在target目录下可以看到两个jar文件：

```
$ ls
classes
generated-sources
maven-archiver
maven-status
springboot-exec-jar-1.0-SNAPSHOT.jar
springboot-exec-jar-1.0-SNAPSHOT.jar.original
```

其中，springboot-exec-jar-1.0-SNAPSHOT.jar.original是Maven标准打包插件打的jar包，它只包含我们自己的Class，不包含依赖，而springboot-exec-jar-1.0-SNAPSHOT.jar是Spring Boot打包插件创建的包含依赖的jar，可以直接运行：

```
$ java -jar springboot-exec-jar-1.0-SNAPSHOT.jar
```

这样，部署一个Spring Boot应用就非常简单，无需预装任何服务器，只需要上传jar包即可。

在打包的时候，因为打包后的Spring Boot应用不会被修改，因此，默认情况下，spring-boot-devtools这个依赖不会被打包进去。但是要注意，使用早期的Spring Boot版本时，需要配置一下才能排除spring-boot-devtools这个依赖：

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <excludeDevtools>true</excludeDevtools>
    </configuration>
</plugin>
```

如果不喜欢默认的项目名+版本号作为文件名，可以加一个配置指定文件名：

```
<project ...>
    ...
    <build>
        <finalName>awesome-app</finalName>
        ...
    </build>
</project>
```

这样打包后的文件名就是awesome-app.jar。

练习

使用Spring Boot插件打包可执行jar

小结

Spring Boot提供了一个Maven插件用于打包所有依赖到单一jar文件，此插件十分易用，无需配置。

在上一节中，我们使用Spring Boot提供的spring-boot-maven-plugin打包Spring Boot应用，可以直接获得一个完整的可运行的jar包，把它上传到服务器上再运行就极其方便。

但是这种方式也不是没有缺点。最大的缺点就是包太大了，动不动几十MB，在网速不给力的情况下，上传服务器非常耗时。并且，其中我们引用到的Tomcat、Spring和其他第三方组件，只要版本号不变，这些jar就相当于每次都重复打进去，再重复上传了一遍。

真正经常改动的代码其实是我们自己编写的代码。如果只打包我们自己编写的代码，通常jar包也就几百KB。但是，运行的时候，classpath中没有依赖的jar包，肯定会报错。

所以问题来了：如何只打包我们自己编写的代码，同时又自动把依赖包下载到某处，并自动引入到classpath中。解决方案就是使用spring-boot-thin-launcher。

使用spring-boot-thin-launcher

我们先演示如何使用spring-boot-thin-launcher，再详细讨论它的工作原理。

首先复制一份上一节的Maven项目，并重命名为springboot-thin-jar：

```
<project ...>
    ...
    <groupId>com.itranswarp.learnjava</groupId>
    <artifactId>springboot-thin-jar</artifactId>
    <version>1.0-SNAPSHOT</version>
    ...
```

然后，修改<build>-<plugins>-<plugin>，给原来的spring-boot-maven-plugin增加一个<dependency>如下：

```
<project ...>
    ...
    <build>
        <finalName>awesome-app</finalName>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
                <dependencies>
                    <dependency>
                        <groupId>org.springframework.boot.experimental</groupId>
                        <artifactId>spring-boot-thin-layout</artifactId>
                        <version>1.0.27.RELEASE</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
    </build>
</project>
```

不需要任何其他改动了，我们直接按正常的流程打包，执行mvn clean package，观察target目录最终生成的可执行awesome-app.jar，只有79KB左右。

直接运行java -jar awesome-app.jar，效果和上一节完全一样。显然，79KB的jar肯定无法放下Tomcat和Spring这样的大块头。那么，运行时这个awesome-app.jar又是怎么找到它自己依赖的jar包呢？

实际上spring-boot-thin-launcher这个插件改变了spring-boot-maven-plugin的默认行为。它输出的jar包只包含我们自己代码编译后的class，一个很小的ThinJarWrapper，以及解析pom.xml后得到的所有依赖jar的列表。

运行的时候，入口实际上是ThinJarWrapper，它会先在指定目录搜索看看依赖的jar包是否都存在，如果不存在，先从Maven中央仓库下载到本地，然后，再执行我们自己编写的main()入口方法。这种方式有点类似很多在线安装程序：用户下载后得到的是一个很小的exe安装程序，执行安装程序时，会首先在线下载所需的若干巨大的文件，再进行真正的安装。

这个spring-boot-thin-launcher在启动时搜索的默认目录是用户主目录的.m2，我们也可以指定下载目录，例如，将下载目录指定为当前目录：

```
$ java -Dthin.root=. -jar awesome-app.jar
```

上述命令通过环境变量thin.root传入当前目录，执行后发现当前目录下自动生成了一个repository目录，这和Maven的默认下载目录~/.m2/repository的结构是完全一样的，只是它仅包含awesome-app.jar所需的运行期依赖项。

注意：只有首次运行时会自动下载依赖项，再次运行时由于无需下载，所以启动速度会大大加快。如果删除了repository目录，再次运行时就会再次触发下载。

预热

把79KB大小的awesome-app.jar直接扔到服务器执行，上传过程就非常快。但是，第一次在服务器上运行awesome-app.jar时，仍需要从Maven中央仓库下载大量的jar包，所以，spring-boot-thin-launcher还提供了一个dryrun选项，专门用来下载依赖项而不执行实际代码：

```
java -Dthin.dryrun=true -Dthin.root=. -jar awesome-app.jar
```

执行上述代码会在当前目录创建repository目录，并下载所有依赖项，但并不会运行我们编写的main()方法。此过程称之为“预热”（warm up）。

如果服务器由于安全限制不允许从外网下载文件，那么可以在本地预热，然后把awesome-app.jar和repository目录上传到服务器。只要依赖项没有变化，后续改动只需要上传awesome-app.jar即可。

练习

瘦身Spring Boot的可执行jar包

小结

利用spring-boot-thin-launcher可以给Spring Boot应用瘦身。其原理是记录app依赖的jar包，在首次运行时先下载依赖项并缓存到本地。

在生产环境中，需要对应用程序的状态进行监控。前面我们已经介绍了使用JMX对Java应用程序包括JVM进行监控，使用JMX需要把一些监控信息以MBean的形式暴露给JMX Server，而Spring Boot已经内置了一个监控功能，它叫Actuator。

使用Actuator非常简单，只需添加如下依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

然后正常启动应用程序，Actuator会把它能收集到的所有信息都暴露给JMX。此外，Actuator还可以通过URL/actuator/挂载一些监控点，例如，输入http://localhost:8080/actuator/health，我们可以查看应用程序当前状态：

```
{
    "status": "UP"
}
```

许多网关作为反向代理需要一个URL来探测后端集群应用是否存活，这个URL就可以提供给网关使用。

Actuator默认把所有访问点暴露给JMX，但处于安全原因，只有health和info会暴露给Web。Actuator提供的所有访问点均在官方文档列出，要暴露更多的访问点给Web，需要在application.yml中加上配置：

```
management:
  endpoints:
    web:
      exposure:
        include: info, health, beans, env, metrics
```

要特别注意暴露的URL的安全性，例如，/actuator/env可以获取当前机器的所有环境变量，不可暴露给公网。

练习

使用Actuator实现监控

小结

Spring Boot提供了一个Actuator，可以方便地实现监控，并可通过Web访问特定类型的监控。

Profile本身是Spring提供的功能，我们在使用条件装配中已经讲到了，Profile表示一个环境的概念，如开发、测试和生产这3个环境：

- native
- test
- production

或者按git分支定义master、dev这些环境：

- master
- dev

在启动一个Spring应用程序的时候，可以传入一个或多个环境，例如：

```
-Dspring.profiles.active=test,master
```

大多数情况下，使用一个环境就足够了。

Spring Boot对Profiles的支持在于，可以在application.yml中为每个环境进行配置。下面是一个示例配置：

```
spring:
  application:
    name: ${APP_NAME:unnamed}
  datasource:
    url: jdbc:hsqldb:file:testdb
    username: sa
    password:
    dirver-class-name: org.hsqldb.jdbc.JDBCDriver
    hikari:
      auto-commit: false
      connection-timeout: 3000
      validation-timeout: 3000
      max-lifetime: 60000
      maximum-pool-size: 20
      minimum-idle: 1

pebble:
  suffix:
  cache: false

server:
  port: ${APP_PORT:8080}

---

spring:
  profiles: test

server:
  port: 8000

---

spring:
  profiles: production

server:
  port: 80

pebble:
  cache: true
```

注意到分隔符---，最前面的配置是默认配置，不需要指定Profile，后面的每段配置都必须以spring.profiles: xxx开头，表示一个Profile。上述配置默认使用8080端口，但是在test环境下，使用8000端口，在production环境下，使用80端口，并且启用Pebble的缓存。

如果我们不指定任何Profile，直接启动应用程序，那么Profile实际上就是default，可以从Spring Boot启动日志看出：

```
2020-06-13 11:20:58.141  INFO 73265 --- [  restartedMain] com.itranswarp.learnjava.Application     : Starting Application on ... with PID 73265 ...
2020-06-13 11:20:58.144  INFO 73265 --- [  restartedMain] com.itranswarp.learnjava.Application     : No active profile set, falling back to default profiles: default
```

要以test环境启动，可输入如下命令：

```
$ java -Dspring.profiles.active=test -jar springboot-profiles-1.0-SNAPSHOT.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.0.RELEASE)

2020-06-13 11:24:45.020  INFO 73987 --- [           main] com.itranswarp.learnjava.Application     : Starting Application v1.0-SNAPSHOT on ... with PID 73987 ...
2020-06-13 11:24:45.022  INFO 73987 --- [           main] com.itranswarp.learnjava.Application     : The following profiles are active: test
...
2020-06-13 11:24:47.533  INFO 73987 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8000 (http) with context path ''
...
```

从日志看到活动的Profile是test，Tomcat的监听端口是8000。

通过Profile可以实现一套代码在不同环境启用不同的配置和功能。假设我们需要一个存储服务，在本地开发时，直接使用文件存储即可，但是，在测试和生产环境，需要存储到云端如S3上，如何通过Profile实现该功能？

首先，我们要定义存储接口StorageService：

```
public interface StorageService {

    // 根据URI打开InputStream:
    InputStream openInputStream(String uri) throws IOException;

    // 根据扩展名+InputStream保存并返回URI:
    String store(String extName, InputStream input) throws IOException;
}
```

本地存储可通过LocalStorageService实现：

```
@Component
@Profile("default")
public class LocalStorageService implements StorageService {
    @Value("${storage.local:/var/static}")
    String localStorageRootDir;

    final Logger logger = LoggerFactory.getLogger(getClass());

    private File localStorageRoot;

    @PostConstruct
    public void init() {
        logger.info("Intializing local storage with root dir: {}", this.localStorageRootDir);
        this.localStorageRoot = new File(this.localStorageRootDir);
    }

    @Override
    public InputStream openInputStream(String uri) throws IOException {
        File targetFile = new File(this.localStorageRoot, uri);
        return new BufferedInputStream(new FileInputStream(targetFile));
    }

    @Override
    public String store(String extName, InputStream input) throws IOException {
        String fileName = UUID.randomUUID().toString() + "." + extName;
        File targetFile = new File(this.localStorageRoot, fileName);
        try (OutputStream output = new BufferedOutputStream(new FileOutputStream(targetFile))) {
```

```
            input.transferTo(output);
        }
        return fileName;
    }
}
```

而云端存储可通过CloudStorageService实现：

```
@Component
@Profile("!default")
public class CloudStorageService implements StorageService {
    @Value("${storage.cloud.bucket:}")
    String bucket;

    @Value("${storage.cloud.access-key:}")
    String accessKey;

    @Value("${storage.cloud.access-secret:}")
    String accessSecret;

    final Logger logger = LoggerFactory.getLogger(getClass());

    @PostConstruct
    public void init() {
        // TODO:
        logger.info("Initializing cloud storage...");
    }

    @Override
    public InputStream openInputStream(String uri) throws IOException {
        // TODO:
        throw new IOException("File not found: " + uri);
    }

    @Override
    public String store(String extName, InputStream input) throws IOException {
        // TODO:
        throw new IOException("Unable to access cloud storage.");
    }
}
```

注意到LocalStorageService使用了条件装配@Profile("default")，即默认启用LocalStorageService，而CloudStorageService使用了条件装配@Profile("!default")，即非default环境时，自动启用CloudStorageService。这样，一套代码，就实现了不同环境启用不同的配置。

练习

使用Profile启动Spring Boot应用

小结

Spring Boot允许在一个配置文件中针对不同Profile进行配置；

Spring Boot在未指定Profile时默认为default。

使用Profile能根据不同的Profile进行条件装配，但是Profile控制比较糙，如果想要精细控制，例如，配置本地存储，AWS存储和阿里云存储，将来很可能会增加Azure存储等，用Profile就很难实现。

Spring本身提供了条件装配@Conditional，但是要自己编写比较复杂的Condition来做判断，比较麻烦。Spring Boot则为我们准备好了几个非常有用的条件：

- @ConditionalOnProperty：如果有指定的配置，条件生效；
- @ConditionalOnBean：如果有指定的Bean，条件生效；
- @ConditionalOnMissingBean：如果没有指定的Bean，条件生效；
- @ConditionalOnMissingClass：如果没有指定的Class，条件生效；
- @ConditionalOnWebApplication：在Web环境中条件生效；
- @ConditionalOnExpression：根据表达式判断条件是否生效。

我们以最常用的@ConditionalOnProperty为例，把上一节的StorageService改写如下。首先，定义配置storage.type=xxx，用来判断条件，默认为local：

```
storage:
  type: ${STORAGE_TYPE:local}
```

设定为local时，启用LocalStorageService：

```
@Component
@ConditionalOnProperty(value = "storage.type", havingValue = "local", matchIfMissing = true)
public class LocalStorageService implements StorageService {
    ...
}
```

设定为aws时，启用AwsStorageService：

```
@Component
@ConditionalOnProperty(value = "storage.type", havingValue = "aws")
public class AwsStorageService implements StorageService {
    ...
}
```

设定为aliyun时，启用AliyunStorageService：

```
@Component
@ConditionalOnProperty(value = "storage.type", havingValue = "aliyun")
public class AliyunStorageService implements StorageService {
    ...
}
```

注意到LocalStorageService的注解，当指定配置为local，或者配置不存在，均启用LocalStorageService。

可见，Spring Boot提供的条件装配使得应用程序更加具有灵活性。

练习

使用Spring Boot提供的条件装配

小结

Spring Boot提供了几个非常有用的条件装配注解，可实现灵活的条件装配。

加载配置文件可以直接使用注解@Value，例如，我们定义了一个最大允许上传的文件大小配置：

```
storage:
  local:
    max-size: 102400
```

在某个FileUploader里，需要获取该配置，可使用@Value注入：

```
@Component
public class FileUploader {
    @Value("${storage.local.max-size:102400}")
    int maxSize;

    ...
}
```

在另一个UploadFilter中，因为要检查文件的MD5，同时也要检查输入流的大小，因此，也需要该配置：

```
@Component
public class UploadFilter implements Filter {
    @Value("${storage.local.max-size:100000}")
    int maxSize;

    ...
}
```

多次引用同一个@Value不但麻烦，而且@Value使用字符串，缺少编译器检查，容易造成多处引用不一致（例如，UploadFilter把缺省值误写为100000）。

为了更好地管理配置，Spring Boot允许创建一个Bean，持有一组配置，并由Spring Boot自动注入。

假设我们在application.yml中添加了如下配置：

```
storage:
  local:
    # 文件存储根目录:
    root-dir: ${STORAGE_LOCAL_ROOT:/var/storage}
    # 最大文件大小，默认100K:
    max-size: ${STORAGE_LOCAL_MAX_SIZE:102400}
    # 是否允许空文件:
    allow-empty: false
    # 允许的文件类型:
    allow-types: jpg, png, gif
```

可以首先定义一个Java Bean，持有该组配置：

```
public class StorageConfiguration {

    private String rootDir;
    private int maxSize;
    private boolean allowEmpty;
    private List<String> allowTypes;

    // TODO: getters and setters
}
```

保证Java Bean的属性名称与配置一致即可。然后，我们添加两个注解：

```
@Configuration
@ConfigurationProperties("storage.local")
public class StorageConfiguration {
    ...
}
```

注意到@ConfigurationProperties("storage.local")表示将从配置项storage.local读取该项的所有子项配置，并且，@Configuration表示StorageConfiguration也是一个Spring管理的Bean，可直接注入到其他Bean中：

```
@Component
public class StorageService {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    StorageConfiguration storageConfig;

    @PostConstruct
    public void init() {
        logger.info("Load configuration: root-dir = {}", storageConfig.getRootDir());
        logger.info("Load configuration: max-size = {}", storageConfig.getMaxSize());
        logger.info("Load configuration: allowed-types = {}", storageConfig.getAllowTypes());
    }
}
```

这样一来，引入storage.local的相关配置就很容易了，因为只需要注入StorageConfiguration这个Bean，这样可以由编译器检查类型，无需编写重复的@Value注解。

练习

加载配置文件

小结

Spring Boot提供了@ConfigurationProperties注解，可以非常方便地把一段配置加载到一个Bean中。

Spring Boot大量使用自动配置和默认配置，极大地减少了代码，通常只需要加上几个注解，并按照默认规则设定一下必要的配置即可。例如，配置JDBC，默认情况下，只需要配置一个spring.datasource：

```
spring:
  datasource:
    url: jdbc:hsqldb:file:testdb
    username: sa
    password:
    dirver-class-name: org.hsqldb.jdbc.JDBCDriver
```

Spring Boot就会自动创建出DataSource、JdbcTemplate、DataSourceTransactionManager，非常方便。

但是，有时候，我们又必须要禁用某些自动配置。例如，系统有主从两个数据库，而Spring Boot的自动配置只能配一个，怎么办？

这个时候，针对DataSource相关的自动配置，就必须关掉。我们需要用exclude指定需要关掉的自动配置：

```
@SpringBootApplication
// 启动自动配置，但排除指定的自动配置:
@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)
public class Application {
    ...
}
```

现在，Spring Boot不再给我们自动创建DataSource、JdbcTemplate和DataSourceTransactionManager了，要实现主从数据库支持，怎么办？

让我们一步一步开始编写支持主从数据库的功能。首先，我们需要把主从数据库配置写到application.yml中，仍然按照Spring Boot默认的格式写，但datasource改为datasource-master和datasource-slave：

```
spring:
  datasource-master:
    url: jdbc:hsqldb:file:testdb
    username: sa
    password:
    dirver-class-name: org.hsqldb.jdbc.JDBCDriver
  datasource-slave:
```

```
    url: jdbc:hsqldb:file:testdb
    username: sa
    password:
    dirver-class-name: org.hsqldb.jdbc.JDBCDriver
```

注意到两个数据库实际上是同一个库。如果使用MySQL，可以创建一个只读用户，作为datasource-slave的用户来模拟一个从库。

下一步，我们分别创建两个HikariCP的DataSource：

```
public class MasterDataSourceConfiguration {
    @Bean("masterDataSourceProperties")
    @ConfigurationProperties("spring.datasource-master")
    DataSourceProperties dataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean("masterDataSource")
    DataSource dataSource(@Autowired @Qualifier("masterDataSourceProperties") DataSourceProperties props) {
        return props.initializeDataSourceBuilder().build();
    }
}

public class SlaveDataSourceConfiguration {
    @Bean("slaveDataSourceProperties")
    @ConfigurationProperties("spring.datasource-slave")
    DataSourceProperties dataSourceProperties() {
        return new DataSourceProperties();
    }

    @Bean("slaveDataSource")
    DataSource dataSource(@Autowired @Qualifier("slaveDataSourceProperties") DataSourceProperties props) {
        return props.initializeDataSourceBuilder().build();
    }
}
```

注意到上述class并未添加@Configuration和@Component，要使之生效，可以使用@Import导入：

```
@SpringBootApplication
@EnableAutoConfiguration(exclude = DataSourceAutoConfiguration.class)
@Import({ MasterDataSourceConfiguration.class, SlaveDataSourceConfiguration.class})
public class Application {
    ...
}
```

此外，上述两个DataSource的Bean名称分别为masterDataSource和slaveDataSource，我们还需要一个最终的@Primary标注的DataSource，它采用Spring提供的AbstractRoutingDataSource，代码实现如下：

```
class RoutingDataSource extends AbstractRoutingDataSource {
    @Override
    protected Object determineCurrentLookupKey() {
        // 从ThreadLocal中取出key:
        return RoutingDataSourceContext.getDataSourceRoutingKey();
    }
}
```

RoutingDataSource本身并不是真正的DataSource，它通过Map关联一组DataSource，下面的代码创建了包含两个DataSource的RoutingDataSource，关联的key分别为masterDataSource和slaveDataSource：

```
public class RoutingDataSourceConfiguration {
    @Primary
    @Bean
    DataSource dataSource(
            @Autowired @Qualifier("masterDataSource") DataSource masterDataSource,
            @Autowired @Qualifier("slaveDataSource") DataSource slaveDataSource) {
        var ds = new RoutingDataSource();
        // 关联两个DataSource:
        ds.setTargetDataSources(Map.of(
                "masterDataSource", masterDataSource,
                "slaveDataSource", slaveDataSource));
        // 默认使用masterDataSource:
        ds.setDefaultTargetDataSource(masterDataSource);
        return ds;
    }

    @Bean
    JdbcTemplate jdbcTemplate(@Autowired DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean
    DataSourceTransactionManager dataSourceTransactionManager(@Autowired DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }
}
```

仍然需要自己创建JdbcTemplate和PlatformTransactionManager，注入的是标记为@Primary的RoutingDataSource。

这样，我们通过如下的代码就可以切换RoutingDataSource底层使用的真正的DataSource：

```
RoutingDataSourceContext.setDataSourceRoutingKey("slaveDataSource");
jdbcTemplate.query(...);
```

只不过写代码切换DataSource即麻烦又容易出错，更好的方式是通过注解配合AOP实现自动切换，这样，客户端代码实现如下：

```
@Controller
public class UserController {
 @RoutingWithSlave // <-- 指示在此方法中使用slave数据库
 @GetMapping("/profile")
 public ModelAndView profile(HttpSession session) {
        ...
    }
}
```

实现上述功能需要编写一个@RoutingWithSlave注解，一个AOP织入和一个ThreadLocal来保存key。由于代码比较简单，这里我们不再详述。

如果我们想要确认是否真的切换了DataSource，可以覆写determineTargetDataSource()方法并打印出DataSource的名称：

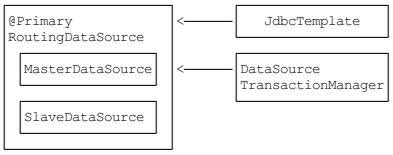
```
class RoutingDataSource extends AbstractRoutingDataSource {
    ...

    @Override
    protected DataSource determineTargetDataSource() {
        DataSource ds = super.determineTargetDataSource();
        logger.info("determin target datasource: {}", ds);
        return ds;
    }
}
```

访问不同的URL，可以在日志中看到两个DataSource，分别是HikariPool-1和hikariPool-2：

```
2020-06-14 17:55:21.676  INFO 91561 --- [nio-8080-exec-7] c.i.learnjava.config.RoutingDataSource   : determin target datasource: HikariDataSource (HikariPool-1)
```

```
2020-06-14 17:57:08.992  INFO 91561 --- [io-8080-exec-10] c.i.learnjava.config.RoutingDataSource   : determin target datasource: HikariDataSource (HikariPool-2)
```

我们用一个图来表示创建的DataSource以及相关Bean的关系：



注意到`DataSourceTransactionManager`和`JdbcTemplate`引用的都是`RoutingDataSource`，所以，这种设计的一个限制就是：在一个请求中，一旦切换了内部数据源，在同一个事务中，不能再切到另一个，否则，`DataSourceTransactionManager`和`JdbcTemplate`操作的就不是同一个数据库连接。

练习

禁用DataSourceAutoConfiguration并配置多数据源

小结

可以通过`@EnableAutoConfiguration(exclude = {...})`指定禁用的自动配置；

可以通过`@Import({...})`导入自定义配置。

我们在Spring中已经学过了集成Filter，本质上就是通过代理，把Spring管理的Bean注册到Servlet容器中，不过步骤比较繁琐，需要配置`web.xml`。

在Spring Boot中，添加一个`Filter`更简单了，可以做到零配置。我们来看看在Spring Boot中如何添加`Filter`。

Spring Boot会自动扫描所有的`FilterRegistrationBean`类型的Bean，然后，将它们返回的`Filter`自动注册到Servlet容器中，无需任何配置。

我们还是以`AuthFilter`为例，首先编写一个`AuthFilterRegistrationBean`，它继承自`FilterRegistrationBean`：

```
@Order(10)
@Component
public class AuthFilterRegistrationBean extends FilterRegistrationBean<Filter> {
    @Autowired
    UserService userService;

    @Override
    public Filter getFilter() {
        return new AuthFilter();
    }

    class AuthFilter implements Filter {
        ...
    }
}
```

`FilterRegistrationBean`本身不是`Filter`，它实际上是`Filter`的工厂。Spring Boot会调用`getFilter()`，把返回的`Filter`注册到Servlet容器中。因为我们可以在`FilterRegistrationBean`中注入需要的资源，然后，在返回的`AuthFilter`中，这个内部类可以引用外部类的所有字段，自然也包括注入的`UserService`，所以，整个过程完全基于Spring的IoC容器完成。

再注意到`AuthFilterRegistrationBean`标记了一个`@Order(10)`，因为Spring Boot支持给多个`Filter`排序，数字小的在前面，所以，多个`Filter`的顺序是可以固定的。

我们再编写一个`ApiFilter`，专门过滤`/api/*`这样的URL。首先编写一个`ApiFilterRegistrationBean`

```
@Order(20)
@Component
public class ApiFilterRegistrationBean extends FilterRegistrationBean<Filter> {
    @PostConstruct
    public void init() {
        setFilter(new ApiFilter());
        setUrlPatterns(List.of("/api/*"));
    }

    class ApiFilter implements Filter {
        ...
    }
}
```

这个`ApiFilterRegistrationBean`和`AuthFilterRegistrationBean`又有所不同。因为我们要过滤URL，而不是针对所有URL生效，因此，在`@PostConstruct`方法中，通过`setFilter()`设置一个`Filter`实例后，再调用`setUrlPatterns()`传入要过滤的URL列表。

练习

添加Filter并指定顺序

小结

在Spring Boot中添加`Filter`更加方便，并且支持对多个`Filter`进行排序。

和Spring相比，使用Spring Boot通过自动配置来集成第三方组件通常来说更简单。

我们将详细介绍如何通过Spring Boot集成常用的第三方组件，包括：

- Open API
- Redis
- Artemis
- RabbitMQ
- Kafka

Open API是一个标准，它的主要作用是描述REST API，既可以作为文档给开发者阅读，又可以让机器根据这个文档自动生成客户端代码等。

在Spring Boot应用中，假设我们编写了一堆REST API，如何添加Open API的支持？

我们只需要在`pom.xml`中加入以下依赖：

```
org.springdoc:springdoc-openapi-ui:1.4.0
```

然后呢？没有然后了，直接启动应用，打开浏览器输入`http://localhost:8080/swagger-ui.html`：

立刻可以看到自动生成的API文档，这里列出了3个API，来自`api-controller`（因为定义在`ApiController`这个类中），点击某个API还可以交互，即输入API参数，点“Try it out”按钮，获得运行结果。

是不是太方便了！

因为我们引入springdoc-openapi-ui这个依赖后，它自动引入Swagger UI用来创建API文档。可以给API加入一些描述信息，例如：

```
@RestController
@RequestMapping("/api")
public class ApiController {
    ...
    @Operation(summary = "Get specific user object by it's id.")
 @GetMapping("/users/{id}")
 public User user(@Parameter(description = "id of the user.") @PathVariable("id") long id) {
  return userService.getUserById(id);
 }
    ...
}
```

@Operation可以对API进行描述，@Parameter可以对参数进行描述，它们的目的是用于生成API文档的描述信息。添加了描述的API文档如下：

大多数情况下，不需要任何配置，我们就直接得到了一个运行时动态生成的可交互的API文档，该API文档总是和代码保持同步，大大简化了文档的编写工作。

要自定义文档的样式、控制某些API显示等，请参考springdoc文档。

配置反向代理

如果在服务器上，用户访问的域名是https://example.com，但内部是通过类似Nginx这样的反向代理访问实际的Spring Boot应用，比如http://localhost:8080，这个时候，在页面https://example.com/swagger-ui.html上，显示的URL仍然是http://localhost:8080，这样一来，就无法直接在页面执行API，非常不方便。

这是因为Spring Boot内置的Tomcat默认获取的服务器名称是localhost，端口是实际监听端口，而不是对外暴露的域名和80或443端口。要让Tomcat获取到对外暴露的域名等信息，必须在Nginx配置中传入必要的HTTP Header，常用的配置如下：

```
# Nginx配置
server {
    ...
    location / {
        proxy_pass http://localhost:8080;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    }
    ...
}
```

然后，在Spring Boot的application.yml中，加入如下配置：

```
server:
  # 实际监听端口:
  port: 8080
  # 从反向代理读取相关的HTTP Header:
  forward-headers-strategy: native
```

重启Spring Boot应用，即可在Swagger中显示正确的URL。

练习

利用springdoc实现API文档

小结

使用springdoc让其自动创建API文档非常容易，引入依赖后无需任何配置即可访问交互式API文档。

可以对API添加注解以便生成更详细的描述。

在Spring Boot中，要访问Redis，可以直接引入spring-boot-starter-data-redis依赖，它实际上是Spring Data的一个子项目——Spring Data Redis，主要用到了这几个组件：

- Lettuce：一个基于Netty的高性能Redis客户端；
- RedisTemplate：一个类似于JdbcTemplate的接口，用于简化Redis的操作。

因为Spring Data Redis引入的依赖项很多，如果只是为了使用Redis，完全可以只引入Lettuce，剩下的操作都自己来完成。

本节我们稍微深入一下Redis的客户端，看看怎么一步一步把一个第三方组件引入到Spring Boot中。

首先，我们添加必要的几个依赖项：

- io.lettuce:lettuce-core
- org.apache.commons:commons-pool2

注意我们并未指定版本号，因为在spring-boot-starter-parent中已经把常用组件的版本号确定下来了。

第一步是在配置文件application.yml中添加Redis的相关配置：

```
spring:
  redis:
    host: ${REDIS_HOST:localhost}
    port: ${REDIS_PORT:6379}
    password: ${REDIS_PASSWORD:}
    ssl: ${REDIS_SSL:false}
    database: ${REDIS_DATABASE:0}
```

然后，通过RedisConfiguration来加载它：

```
@ConfigurationProperties("spring.redis")
public class RedisConfiguration {
 private String host;
 private int port;
 private String password;
 private int database;

    // getters and setters...
}
```

再编写一个@Bean方法来创建RedisClient，可以直接放在RedisConfiguration中：

```
@ConfigurationProperties("spring.redis")
public class RedisConfiguration {
    ...

    @Bean
    RedisClient redisClient() {
        RedisURI uri = RedisURI.Builder.redis(this.host, this.port)
                .withPassword(this.password)
                .withDatabase(this.database)
                .build();
        return RedisClient.create(uri);
```

```
    }
}
```

在启动入口引入该配置：

```
@SpringBootApplication
@Import(RedisConfiguration.class) // 加载Redis配置
public class Application {
    ...
}
```

注意：如果在`RedisConfiguration`中标注`@Configuration`，则可通过Spring Boot的自动扫描机制自动加载，否则，使用`@Import`手动加载。

紧接着，我们用一个`RedisService`来封装所有的Redis操作。基础代码如下：

```
@Component
public class RedisService {
    @Autowired
    RedisClient redisClient;

    GenericObjectPool<StatefulRedisConnection<String, String>> redisConnectionPool;

    @PostConstruct
    public void init() {
        GenericObjectPoolConfig<StatefulRedisConnection<String, String>> poolConfig = new GenericObjectPoolConfig<>();
        poolConfig.setMaxTotal(20);
        poolConfig.setMaxIdle(5);
        poolConfig.setTestOnReturn(true);
        poolConfig.setTestWhileIdle(true);
        this.redisConnectionPool = ConnectionPoolSupport.createGenericObjectPool(() -> redisClient.connect(), poolConfig);
    }

    @PreDestroy
    public void shutdown() {
        this.redisConnectionPool.close();
        this.redisClient.shutdown();
    }
}
```

注意到上述代码引入了Commons Pool的一个对象池，用于缓存Redis连接。因为Lettuce本身是基于Netty的异步驱动，在异步访问时并不需要创建连接池，但基于Servlet模型的同步访问时，连接池是有必要的。连接池在`@PostConstruct`方法中初始化，在`@PreDestroy`方法中关闭。

下一步，是在`RedisService`中添加Redis访问方法。为了简化代码，我们仿照`JdbcTemplate.execute(ConnectionCallback)`方法，传入回调函数，可大幅减少样板代码。

首先定义回调函数接口`SyncCommandCallback`：

```
@FunctionalInterface
public interface SyncCommandCallback<T> {
    // 在此操作Redis:
    T doInConnection(RedisCommands<String, String> commands);
}
```

编写`executeSync`方法，在该方法中，获取Redis连接，利用callback操作Redis，最后释放连接，并返回操作结果：

```
public <T> T executeSync(SyncCommandCallback<T> callback) {
    try (StatefulRedisConnection<String, String> connection = redisConnectionPool.borrowObject()) {
        connection.setAutoFlushCommands(true);
        RedisCommands<String, String> commands = connection.sync();
        return callback.doInConnection(commands);
    } catch (Exception e) {
        logger.warn("executeSync redis failed.", e);
        throw new RuntimeException(e);
    }
}
```

有的童鞋觉得这样访问Redis的代码太复杂了，实际上我们可以针对常用操作把它封装一下，例如`set`和`get`命令：

```
public String set(String key, String value) {
    return executeSync(commands -> commands.set(key, value));
}

public String get(String key) {
    return executeSync(commands -> commands.get(key));
}
```

类似的，`hget`和`hset`操作如下：

```
public boolean hset(String key, String field, String value) {
    return executeSync(commands -> commands.hset(key, field, value));
}

public String hget(String key, String field) {
    return executeSync(commands -> commands.hget(key, field));
}

public Map<String, String> hgetall(String key) {
    return executeSync(commands -> commands.hgetall(key));
}
```

常用命令可以提供方法接口，如果要执行任意复杂的操作，就可以通过`executeSync(SyncCommandCallback<T>)`来完成。

完成了`RedisService`后，我们就可以使用Redis了。例如，在`UserController`中，我们在Session中只存放登录用户的ID，用户信息存放到Redis，提供两个方法用于读写：

```
@Controller
public class UserController {
    public static final String KEY_USER_ID = "__userid__";
    public static final String KEY_USERS = "__users__";

    @Autowired ObjectMapper objectMapper;
    @Autowired RedisService redisService;

    // 把User写入Redis:
    private void putUserIntoRedis(User user) throws Exception {
        redisService.hset(KEY_USERS, user.getId().toString(), objectMapper.writeValueAsString(user));
    }

    // 从Redis读取User:
    private User getUserFromRedis(HttpSession session) throws Exception {
        Long id = (Long) session.getAttribute(KEY_USER_ID);
        if (id != null) {
            String s = redisService.hget(KEY_USERS, id.toString());
            if (s != null) {
                return objectMapper.readValue(s, User.class);
            }
        }
        return null;
    }
    ...
}
```

用户登录成功后，把ID放入Session，把User实例放入Redis：

```
@PostMapping("/signin")
public ModelAndView doSignin(@RequestParam("email") String email, @RequestParam("password") String password, HttpSession session) throws Exception {
    try {
        User user = userService.signin(email, password);
        session.setAttribute(KEY_USER_ID, user.getId());
        putUserIntoRedis(user);
    } catch (RuntimeException e) {
        return new ModelAndView("signin.html", Map.of("email", email, "error", "Signin failed"));
    }
    return new ModelAndView("redirect:/profile");
}
```

需要获取User时，从Redis取出：

```
@GetMapping("/profile")
public ModelAndView profile(HttpSession session) throws Exception {
    User user = getUserFromRedis(session);
    if (user == null) {
        return new ModelAndView("redirect:/signin");
    }
    return new ModelAndView("profile.html", Map.of("user", user));
}
```

从Redis读写Java对象时，序列化和反序列化是应用程序的工作，上述代码使用JSON作为序列化方案，简单可靠。也可将相关序列化操作封装到RedisService中，这样可以提供更加通用的方法：

```
public <T> T get(String key, Class<T> clazz) {
    ...
}

public <T> T set(String key, T value) {
    ...
}
```

练习

在Spring Boot中访问Redis

小结

Spring Boot默认使用Lettuce作为Redis客户端，同步使用时，应通过连接池提高效率。

ActiveMQ Artemis是一个JMS服务器，在集成JMS一节中我们已经详细讨论了如何在Spring中集成Artemis，本节我们讨论如何在Spring Boot中集成Artemis。

我们还是以实际工程为例，创建一个springboot-jms工程，引入的依赖除了spring-boot-starter-web，spring-boot-starter-jdbc等以外，新增spring-boot-starter-artemis：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-artemis</artifactId>
</dependency>
```

同样无需指定版本号。

如何创建Artemis服务器我们已经在集成JMS一节中详细讲述了，此处不再重复。创建Artemis服务器后，我们在application.yml中加入相关配置：

```
spring:
  artemis:
    # 指定连接外部Artemis服务器，而不是启动嵌入式服务：
    mode: native
    # 服务器地址和端口号：
    host: 127.0.0.1
    port: 61616
    # 连接用户名和口令由创建Artemis服务器时指定：
    user: admin
    password: password
```

和Spring版本的JMS代码相比，使用Spring Boot集成JMS时，只要引入了spring-boot-starter-artemis，Spring Boot会自动创建JMS相关的ConnectionFactory、JmsListenerContainerFactory、JmsTemplate等，无需我们再手动配置了。

发送消息时只需要引入JmsTemplate：

```
@Component
public class MessagingService {
    @Autowired
    JmsTemplate jmsTemplate;

    public void sendMailMessage() throws Exception {
        String text = "...";
        jmsTemplate.send("jms/queue/mail", new MessageCreator() {
            public Message createMessage(Session session) throws JMSException {
                return session.createTextMessage(text);
            }
        });
    }
}
```

接收消息时只需要标注@JmsListener：

```
@Component
public class MailMessageListener {
    final Logger logger = LoggerFactory.getLogger(getClass());

    @JmsListener(destination = "jms/queue/mail", concurrency = "10")
    public void onMailMessageReceived(Message message) throws Exception {
        logger.info("received message: " + message);
    }
}
```

可见，应用程序收发消息的逻辑和Spring中使用JMS完全相同，只是通过Spring Boot，我们把工程简化到只需要设定Artemis相关配置。

练习

在Spring Boot中使用Artemis

小结

在Spring Boot中使用Artemis作为JMS服务时，只需引入spring-boot-starter-artemis依赖，即可直接使用JMS。

前面我们讲了ActiveMQ Artemis，它实现了JMS的消息服务协议。JMS是JavaEE的消息服务标准接口，但是，如果Java程序要和另一种语言编写的程序通过消息服务器进行通信，那么JMS就不太适合了。

AMQP是一种使用广泛的独立于语言的消息协议，它的全称是Advanced Message Queuing Protocol，即高级消息队列协议，它定义了一种二进制格式的消息流，任何编程语言都可以实现该协议。实际上，Artemis也支持AMQP，但实际应用最广泛的AMQP服务器是使用Erlang编写的RabbitMQ。

安装RabbitMQ

我们先从RabbitMQ的官网下载并安装RabbitMQ，安装和启动RabbitMQ请参考官方文档。要验证启动是否成功，可以访问RabbitMQ的管理后台http://localhost:15672，如能看到登录界面表示RabbitMQ启动成功：

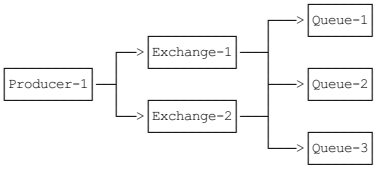
RabbitMQ后台管理的默认用户名和口令均为guest。

AMQP协议

AMQP协议和前面我们介绍的JMS协议有所不同。在JMS中，有两种类型的消息通道：

1. 点对点的Queue，即Producer发送消息到指定的Queue，接收方从Queue收取消息；
2. 一对多的Topic，即Producer发送消息到指定的Topic，任意多个在线的接收方均可从Topic获得一份完整的消息副本。

但是AMQP协议比JMS要复杂一点，它只有Queue，没有Topic，并且引入了Exchange的概念。当Producer想要发送消息的时候，它将消息发送给Exchange，由Exchange将消息根据各种规则投递到一个或多个Queue：



如果某个Exchange总是把消息发送到固定的Queue，那么这个消息通道就相当于JMS的Queue。如果某个Exchange把消息发送到多个Queue，那么这个消息通道就相当于JMS的Topic。和JMS的Topic相比，Exchange的投递规则更灵活，比如一个“登录成功”的消息被投递到Queue-1和Queue-2，而“登录失败”的消息则被投递到Queue-3。这些路由规则称之为Binding，通常都在RabbitMQ的管理后台设置。

我们以具体的业务为例子，在RabbitMQ中，首先创建3个Queue，分别用于发送邮件、短信和App通知：

创建Queue时注意到可配置为持久化（Durable）和非持久化（Transient），当Consumer不在线时，持久化的Queue会暂存消息，非持久化的Queue会丢弃消息。

紧接着，我们在Exchanges中创建一个Direct类型的Exchange，命名为registration，并添加如下两个Binding：

上述Binding的规则就是：凡是发送到registration这个Exchange的消息，均被发送到q_mail和q_sms这两个Queue。

我们再创建一个Direct类型的Exchange，命名为login，并添加如下Binding：

上述Binding的规则稍微复杂一点，当发送消息给login这个Exchange时，如果消息没有指定Routing Key，则被投递到q_app和q_mail，如果消息指定了Routing Key="login_failed"，那么消息被投递到q_sms。

配置好RabbitMQ后，我们就可以基于Spring Boot开发AMQP程序。

使用RabbitMQ

我们首先创建Spring Boot工程springboot-rabbitmq，并添加如下依赖引入RabbitMQ：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-amqp</artifactId>
</dependency>
```

然后在application.yml中添加RabbitMQ相关配置：

```
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

我们还需要在Application中添加一个MessageConverter：

```
import org.springframework.amqp.support.converter.MessageConverter;

@SpringBootApplication
public class Application {
    ...

    @Bean
    MessageConverter createMessageConverter() {
        return new Jackson2JsonMessageConverter();
    }
}
```

MessageConverter用于将Java对象转换为RabbitMQ的消息。默认情况下，Spring Boot使用SimpleMessageConverter，只能发送String和byte[]类型的消息，不太方便。使用Jackson2JsonMessageConverter，我们就可以发送JavaBean对象，由Spring Boot自动序列化为JSON并以文本消息传递。

因为引入了starter，所有RabbitMQ相关的Bean均自动装配，我们需要在Producer注入的是RabbitTemplate：

```
@Component
public class MessagingService {
    @Autowired
    RabbitTemplate rabbitTemplate;

    public void sendRegistrationMessage(RegistrationMessage msg) {
        rabbitTemplate.convertAndSend("registration", "", msg);
    }

    public void sendLoginMessage(LoginMessage msg) {
        String routingKey = msg.success ? "" : "login_failed";
        rabbitTemplate.convertAndSend("login", routingKey, msg);
    }
}
```

发送消息时，使用convertAndSend(exchange, routingKey, message)可以指定Exchange、Routing Key以及消息本身。这里传入JavaBean后会自动序列化为JSON文本。上述代码将RegistrationMessage发送到registration，将LoginMessage发送到login，并根据登录是否成功来指定Routing Key。

接收消息时，需要在消息处理的方法上标注@RabbitListener：

```
@Component
public class QueueMessageListener {
    final Logger logger = LoggerFactory.getLogger(getClass());
```

```
    static final String QUEUE_MAIL = "q_mail";
    static final String QUEUE_SMS = "q_sms";
    static final String QUEUE_APP = "q_app";

    @RabbitListener(queues = QUEUE_MAIL)
    public void onRegistrationMessageFromMailQueue(RegistrationMessage message) throws Exception {
        logger.info("queue {} received registration message: {}", QUEUE_MAIL, message);
    }

    @RabbitListener(queues = QUEUE_SMS)
    public void onRegistrationMessageFromSmsQueue(RegistrationMessage message) throws Exception {
        logger.info("queue {} received registration message: {}", QUEUE_SMS, message);
    }

    @RabbitListener(queues = QUEUE_MAIL)
    public void onLoginMessageFromMailQueue(LoginMessage message) throws Exception {
        logger.info("queue {} received message: {}", QUEUE_MAIL, message);
    }

    @RabbitListener(queues = QUEUE_SMS)
    public void onLoginMessageFromSmsQueue(LoginMessage message) throws Exception {
        logger.info("queue {} received message: {}", QUEUE_SMS, message);
    }

    @RabbitListener(queues = QUEUE_APP)
    public void onLoginMessageFromAppQueue(LoginMessage message) throws Exception {
        logger.info("queue {} received message: {}", QUEUE_APP, message);
    }
}
```

上述代码一共定义了5个Consumer，监听3个Queue。

启动应用程序，我们注册一个新用户，然后发送一条`RegistrationMessage`消息。此时，根据`registration`这个Exchange的设定，我们会在两个Queue收到消息：

```
... c.i.learnjava.service.UserService        : try register by bob@example.com...
... c.i.learnjava.web.UserController         : user registered: bob@example.com
... c.i.l.service.QueueMessageListener       : queue q_mail received registration message: [RegistrationMessage: email=bob@example.com, name=Bob, timestamp=1594559871495]
... c.i.l.service.QueueMessageListener       : queue q_sms received registration message: [RegistrationMessage: email=bob@example.com, name=Bob, timestamp=1594559871495]
```

当我们登录失败时，发送`LoginMessage`并设定Routing Key为`login_failed`，此时，只有`q_sms`会收到消息：

```
... c.i.learnjava.service.UserService        : try login by bob@example.com...
... c.i.l.service.QueueMessageListener       : queue q_sms received message: [LoginMessage: email=bob@example.com, name=(unknown), success=false, timestamp=1594559886722]
```

登录成功后，发送`LoginMessage`，此时，`q_mail`和`q_app`将收到消息：

```
... c.i.learnjava.service.UserService        : try login by bob@example.com...
... c.i.l.service.QueueMessageListener       : queue q_mail received message: [LoginMessage: email=bob@example.com, name=Bob, success=true, timestamp=1594559895251]
... c.i.l.service.QueueMessageListener       : queue q_app received message: [LoginMessage: email=bob@example.com, name=Bob, success=true, timestamp=1594559895251]
```

RabbitMQ还提供了使用Topic的Exchange（此Topic指消息的标签，并非JMS的Topic概念），可以使用`*`进行匹配并路由。可见，掌握RabbitMQ的核心是理解其消息的路由规则。

直接指定一个Queue并投递消息也是可以的，此时指定Routing Key为Queue的名称即可，因为RabbitMQ提供了一个`default exchange`用于根据Routing Key查找Queue并直接投递消息到指定的Queue。但是要实现一对多的投递就必须自己配置Exchange。

练习

使用RabbitMQ

小结

Spring Boot提供了AMQP的集成，默认使用RabbitMQ作为AMQP消息服务器。

使用RabbitMQ发送消息时，理解Exchange如何路由至一个或多个Queue至关重要。

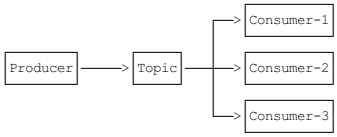
我们在前面已经介绍了JMS和AMQP，JMS是JavaEE的标准消息接口，Artemis是一个JMS实现产品，AMQP是跨语言的一个标准消息接口，RabbitMQ是一个AMQP实现产品。

Kafka也是一个消息服务器，它的特点一是快，二是有巨大的吞吐量，那么Kafka实现了什么标准消息接口呢？

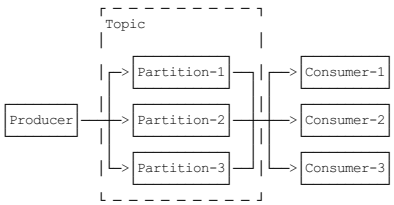
Kafka没有实现任何标准的消息接口，它自己提供的API就是Kafka的接口。

> 哥没有实现任何标准，哥自己就是标准。
>
> —— Kafka

Kafka本身是Scala编写的，运行在JVM之上。Producer和Consumer都通过Kafka的客户端使用网络来与之通信。从逻辑上讲，Kafka设计非常简单，它只有一种类似JMS的Topic的消息通道：



那么Kafka如何支持十万甚至百万的并发呢？答案是分区。Kafka的一个Topic可以有一个至多个Partition，并且可以分布到多台机器上：



Kafka只保证在一个Partition内部，消息是有序的，但是，存在多个Partition的情况下，Producer发送的3个消息会依次发送到Partition-1、Partition-2和Partition-3，Consumer从3个Partition接收的消息并不一定是Producer发送的顺序，因此，多个Partition只能保证接收消息大概率按发送时间有序，并不能保证完全按Producer发送的顺序。这一点在使用Kafka作为消息服务器时要特别注意，对发送顺序有严格要求的Topic只能有一个Partition。

Kafka的另一个特点是消息发送和接收都尽量使用批处理，一次处理几十甚至上百条消息，比一次一条效率要高很多。

最后要注意的是消息的持久性。Kafka总是将消息写入Partition对应的文件，消息保存多久取决于服务器的配置，可以按照时间删除（默认3天），也可以按照文件大小删除，因此，只要Consumer在离线期内的消息还没有被删除，再次上线仍然可以接收到完整的消息流。这一功能实际上是客户端自己实现的，客户端会存储它接收到的最后一个消息的offsetId，再次上线后按上次的offsetId查询。offsetId是Kafka标识某个Partion的每一条消息的递增整数，客户端通常将它存储在ZooKeeper中。

有了Kafka消息设计的基本概念，我们来看看如何在Spring Boot中使用Kafka。

安装Kafka

首先从Kafka官网下载最新版Kafaka，解压后在bin目录找到两个文件：

- zookeeper-server-start.sh：启动ZooKeeper（已内置在Kafka中）；
- kafka-server-start.sh：启动Kafka。

先启动ZooKeeper：

```
$ ./zookeeper-server-start.sh ../config/zookeeper.properties
```

再启动Kafka：

```
./kafka-server-start.sh ../config/server.properties
```

看到如下输出表示启动成功：

```
... INFO [KafkaServer id=0] started (kafka.server.KafkaServer)
```

如果要关闭Kafka和ZooKeeper，依次按Ctrl-C退出即可。注意这是在本地开发时使用Kafka的方式，线上Kafka服务推荐使用云服务厂商托管模式（AWS的MSK，阿里云的消息队列Kafka版）。

使用Kafka

在Spring Boot中使用Kafka，首先要引入依赖：

```
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
</dependency>
```

注意这个依赖是spring-kafka项目提供的。

然后，在application.yml中添加Kafka配置：

```
spring:
  kafka:
    bootstrap-servers: localhost:9092
    consumer:
      auto-offset-reset: latest
      max-poll-records: 100
      max-partition-fetch-bytes: 1000000
```

除了bootstrap-servers必须指定外，consumer相关的配置项均为调优选项。例如，max-poll-records表示一次最多抓取100条消息。配置名称去哪里看？IDE里定义一个KafkaProperties.Consumer的变量：

```
KafkaProperties.Consumer c = null;
```

然后按住Ctrl查看源码即可。

发送消息

Spring Boot自动为我们创建一个KafkaTemplate用于发送消息。注意到这是一个泛型类，而默认配置总是使用String作为Kafka消息的类型，所以注入KafkaTemplate<String, String>即可：

```
@Component
public class MessagingService {
    @Autowired ObjectMapper objectMapper;

    @Autowired KafkaTemplate<String, String> kafkaTemplate;

    public void sendRegistrationMessage(RegistrationMessage msg) throws IOException {
        send("topic_registration", msg);
    }

    public void sendLoginMessage(LoginMessage msg) throws IOException {
        send("topic_login", msg);
    }

    private void send(String topic, Object msg) throws IOException {
        ProducerRecord<String, String> pr = new ProducerRecord<>(topic, objectMapper.writeValueAsString(msg));
        pr.headers().add("type", msg.getClass().getName().getBytes(StandardCharsets.UTF_8));
        kafkaTemplate.send(pr);
    }
}
```

发送消息时，需指定Topic名称，消息正文。为了发送一个JavaBean，这里我们没有使用MessageConverter来转换JavaBean，而是直接把消息类型作为Header添加到消息中，Header名称为type，值为Class全名。消息正文是序列化的JSON。

接收消息

接收消息可以使用@KafkaListener注解：

```
@Component
public class TopicMessageListener {
    private final Logger logger = LoggerFactory.getLogger(getClass());

    @Autowired
    ObjectMapper objectMapper;

    @KafkaListener(topics = "topic_registration", groupId = "group1")
    public void onRegistrationMessage(@Payload String message, @Header("type") String type) throws Exception {
        RegistrationMessage msg = objectMapper.readValue(message, getType(type));
        logger.info("received registration message: {}", msg);
    }

    @KafkaListener(topics = "topic_login", groupId = "group1")
    public void onLoginMessage(@Payload String message, @Header("type") String type) throws Exception {
        LoginMessage msg = objectMapper.readValue(message, getType(type));
        logger.info("received login message: {}", msg);
    }

    @KafkaListener(topics = "topic_login", groupId = "group2")
    public void processLoginMessage(@Payload String message, @Header("type") String type) throws Exception {
        LoginMessage msg = objectMapper.readValue(message, getType(type));
        logger.info("process login message: {}", msg);
    }

    @SuppressWarnings("unchecked")
    private static <T> Class<T> getType(String type) {
        // TODO: use cache:
        try {
            return (Class<T>) Class.forName(type);
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
}
```

在接收消息的方法中，使用`@Payload`表示传入的是消息正文，使用`@Header`可传入消息的指定Header，这里传入`@Header("type")`，就是我们发送消息时指定的Class全名。接收消息时，我们需要根据Class全名来反序列化获得JavaBean。

上述代码一共定义了3个Listener，其中有两个方法监听的是同一个Topic，但它们的Group ID不同。假设Producer发送的消息流是A、B、C、D，Group ID不同表示这是两个不同的Consumer，它们将分别收取完整的消息流，即各自均收到A、B、C、D。Group ID相同的多个Consumer实际上被视作一个Consumer，即如果有两个Group ID相同的Consumer，那么它们各自收到的很可能是A、C和B、D。

运行应用程序，注册新用户后，观察日志输出：

```
... c.i.learnjava.service.UserService        : try register by bob@example.com...
... c.i.learnjava.web.UserController         : user registered: bob@example.com
... c.i.l.service.TopicMessageListener       : received registration message: [RegistrationMessage: email=bob@example.com, name=Bob, timestamp=1594637517458]
```

用户登录后，观察日志输出：

```
... c.i.learnjava.service.UserService        : try login by bob@example.com...
... c.i.l.service.TopicMessageListener       : received login message: [LoginMessage: email=bob@example.com, name=Bob, success=true, timestamp=1594637523470]
... c.i.l.service.TopicMessageListener       : process login message: [LoginMessage: email=bob@example.com, name=Bob, success=true, timestamp=1594637523470]
```

因为Group ID不同，同一个消息被两个Consumer分别独立接收。如果把Group ID改为相同，那么同一个消息只会被两者之一接收。

有细心的童鞋可能会问，在Kafka中是如何创建Topic的？又如何指定某个Topic的分区数量？

实际上开发使用的Kafka默认允许自动创建Topic，创建Topic时默认的分区数量是2，可以通过`server.properties`修改默认分区数量。

在生产环境中通常会关闭自动创建功能，Topic需要由运维人员先创建好。和RabbitMQ相比，Kafka并不提供网页版管理后台，管理Topic需要使用命令行，比较繁琐，只有云服务商通常会提供更友好的管理后台。

练习

使用Kafka

小结

Spring Boot通过`KafkaTemplate`发送消息，通过`@KafkaListener`接收消息；

配置Consumer时，指定Group ID非常重要。

Spring是JavaEE的一个轻量级开发框架，主营IoC和AOP，集成JDBC、ORM、MVC等功能便于开发。

Spring Boot是基于Spring，提供开箱即用的积木式组件，目的是提升开发效率。

那么Spring Cloud是啥？

Spring Cloud顾名思义是跟云相关的，云程序实际上就是指分布式应用程序，所以Spring Cloud就是为了让分布式应用程序编写更方便，更容易而提供的一组基础设施，它的核心是Spring框架，利用Spring Boot的自动配置，力图实现最简化的分布式应用程序开发。

Spring Cloud包含了一大堆技术组件，既有开源社区开发的组件，也有商业公司开发的组件，既有持续更新迭代的组件，也有即将退役不再维护的组件。

本章会介绍如何基于Spring Cloud创建分布式应用程序，但并不会面面俱到地介绍所有组件，而是挑选几个核心组件，演示如何构造一个基本的分布式应用程序。