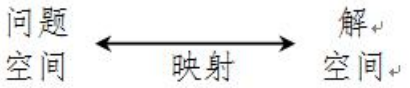
第 1 章　　对象导论

面向对象程序设计（Object-oriented Programming，OOP）

1.1 抽象过程



面向对象语言的五个基本特性：

（1）万物皆为对象

（2）程序是对象的集合，它们通过发送消息来告知彼此所要做的

（3）每个对象都有自己的由其他对象所构成的存储

（4）每个对象都有其类型

（5）某一特特定类型的所有对象都可以接收同样的消息

对象具有**状态、行为和标识**

1.2 每个对象都有一个接口

对象都是唯一的，但同时具有相同的特性和行为的对象所归属的类的一部分。

类描述了具有相同特性（数据元素）和行为（功能）的对象集合。类实际上是一个数据类型。

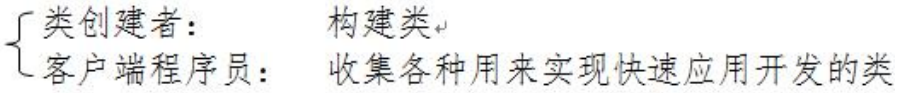
程序设计挑战就是在问题空间的元素和解空间的对象之间创建一对一的映射。

接口确定了对某一特定对象所能发出的请求，决定接口的便是类型。过程概括为：向某个对象“发送消息”(产生请求)，这个对象便知道此消息的目的，然后执行对应的程序代码。

1.3 每个对象都提供服务

对象即为“服务提供者”。(好处：提高内聚性)

1.4 被隐藏的具体实现



访问控制存在原因：

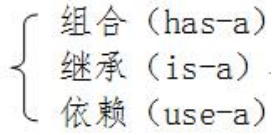
（1）让客户端程序员无法触及不应该触及的部分

（2）允许类设计者改变内部的工作方式而不用担心影响到客户端程序员

Java 用于类内部设定边界：(访问指定词（access specifier))

public、private、protected 和默认的访问控制（包访问权限）

1.5 复用具体实现



首先考虑使用组合。

1.6 继承

基类（超类或父类）

导出类（继承类或子类）

继承不仅包括现有类型的所有成员（private 成员被隐藏，不可访问），而且复制了基类的接口。

基类和导出类产生差异的方法：一是添加新方法；二是覆盖（overriding）基类方法

1.6.1 “是一个”与“像是一个”关系

“是一个”(is-a）视为纯粹替代，称为“替代关系”

“像是一个”(is-like-a）具有旧类的接口，还包含其他方法

1.7 伴随多态的可互换对象

前期绑定：编译器将产生对具体函数名字的调用，而运行时，将这个调用解析到将要被执行代码的绝对地址。

后期绑定：Java 使用一小段特殊的代码来替代绝对地址调用，这段代码中存储的信息来计算方法体的地址。(默认行为)

向上转型（upcasting）把导出类看做是它的基类的过程。

1.8 单根继承结构

终极基类的名字是 Object

单根继承的优点：一是保证所有对象具有某些功能（Object 所有方法）；二是所有对象可以很容易地在堆上创建；三是简化参数传递。

1.9 容器

容器是在任何需要的时候都可以扩充自己以容纳你置于其中的所有东西。(List、Map、Set 以及诸如队列、树、堆栈等)

使用不同容器的原因：一是不同容器提供了不同的接口和外部行为；二是不同容器对于某些操作具有不同的效率。

1.9.1 参数化类型

Java SE5 之前通过存储 Object 的容器可以存储任何东西，使得容器被复用，缺点是：当从容器取出对象时，还是必须要以某种方式记住这些对象究竟是什么类型，这样才能执行向下转型（除非确切知道所要处理的对象的类型，否则向下转型是不安全的）。

解决方案称为参数化类型机制，就是一个编译器可以自动定制作用于特定类型上的类。

1.10 对象的创建和生命期

使用对象时，最关键问题之一便是它们的生成和销毁方式。

对象数据存储位置：一是将对象置于堆栈或静态存储区域（效率高，灵活性差）；二是堆（heap）的内存池中动态创建对象（存储空间运行时动态管理，需要大量时间在堆中分配空间）。Java 采取动态内存分配方式。

对象的生命周期：Java 提供了被称为“垃圾回收器”的机制，可以自动发现对象何时不再被使用，并继而销毁它。

1.11 异常处理：处理错误

异常是一种对象，它从出错地点被“抛出”，并被专门设计用来处理特定类型错误的相应的异常处理器“捕获”。

异常作用就是提供了一种从错误状态进行可靠恢复的途径。

1.12 并发编程

并发的概念就是把问题切分成多个可独立运行的部分(任务),从而提高程序的响应能力。在程序中，这种彼此独立运行的部分称之为线程。

并发存在的隐患：共享资源，解决方案：锁定资源，完成任务后，释放资源锁。

1.13 Java 和 Internet

1.13.1 Web 是什么

1、客户/服务器计算技术

客户/服务器系统的核心思想是：系统具有一个中央信息存储池（central repository of information），用来存储某种数据，它通常存在于数据库中，可以根据需要将它分发给某些人或机器集群。(关键在于信息存储池的位置集中于中央，使得它可以被修改，并且这些修改将被传播给信息消费者)

信息存储池、用于分发信息的软件以及信息与软件所驻留的机器或机群被总称为服务器。

驻留在用户机器上的软件与服务器进行通信，以获取信息、处理信息，然后将它们显示在被称为客户机的用户机器上。

事务处理：保证一个客户插入的新数据不会覆盖另一个客户插入的新数据，也不会将其添加到数据库的过程中丢失。

客户/服务器计算技术大概占了所有程序设计行为的一半。

2、Web 就是一台巨型服务器

Web 发展过程：1、简单单向过程（对服务器产生一个请求，它返回一个文件，浏览器软件根据本地机器的格式解读文件）；2、客户可以将信息反馈给服务器；3、客户机执行运算任务

客户端浏览器运行程序的能力，这被称为“客户端编程”

1.13.2 客户端编程

Web 最初的“服务器-浏览器”设计是为了能够提供交互性的内容，但是交互性完全由服务器提供。服务器产生静态页面，提供给只能解释并显示它们的客户端浏览器。HTML 包含有简单的数据收集机制：文本输入框、复选框、单选框、列表和下拉式列表以及按钮——它只能被编程来实现复位表单上的数据或提交表单上的数据给服务器。这种提动作通过所有的Web 服务器都提供的通用网关接口（common gateway interface，CGI）传递。提交内容会告诉 CGI 应该如何处理它。

可以通过 CGI 做任何事情。CGI 程序复杂而难以维护，并同时响应时间过长的问题。响应时间依赖于所必须发送的数据量的大小，以及服务器和 Internet 的负载。

问题的解决方法就是客户端编程。意味着：Web 浏览器能用来执行任何它可以完成的工作，使得返回给用户的结果各加迅捷，而且使得网站更加具有交互性。

客户端编程的问题是：它与通常意义上的编程十分不同，参数几乎相同，而平台却不同。Web 浏览器就像一个功能受限的操作系统。

1、插件（plug-in）

插件的价值在于：它允许专家级的程序员不需经过浏览器生产厂家的许可，就可以开发某种语言的扩展，并将它们添加到服务器中。提供了一个“后门”，使得可以创建新的客户端编程语言。

2、脚本语言（scripting language）

通过使用脚本语言，可以将客户端程序的源代码直接嵌入到 HTML 页面中，解释这种语言的插件在 HTML 页面被显示时自动激活。优点是易于理解，因为它只是作为 HTML 页面一部分的简单文本，当服务器收到要获取该页面的请求时，它们可以被快速加载。缺点是代码会暴露给任何人去浏览（或窃取）。

JavaScript 语言优缺点。用于创建更丰富、更具有交互性的图形化用户界面（graphic user interface，GUI）。

3、Java

Java 是通过 applet 以及使用 Java Web Start 来进行客户端编程。

4、备选方案

Macromedia 的 Flex，允许创建基于 Flash 的与 applet 相当的应用。ActionScript 语言是基于 ECMAScript 的，Flex 使得在编程时无需担心浏览器的相关性，因此远比 JavaScript 要吸引人得多。值得考虑的备选方案。

5、.NET 和 C#

它要求客户端必须运行 Windows 平台。

6、Internet 和 Intranet

1.13.3 服务器端编程

基于 Java 的 Web 服务器，它让你用 Java 编写的被称为 servlet 的程序来实现服务器端编程。Servlet 及其衍生物 JSP，消除了处理具有不同能力的浏览器所遇到的问题。

第 2 章　一切都是对象

2.1 用引用操纵对象

引用（reference）

对象（object）

2.2 必须由你创建所有对象

引用与对象关联通过 new 操作符

2.2.1 存储到什么地方

1）寄存器：最快的存储区，数量极其有限，根据需求分配，不能直接控制

2）堆栈：位于通用 RAM（随机访问存储）中，通过堆栈指针可以从处理器获得直接支持，Java 对象引用、基本类型存储于堆栈，对象并不处于其中

3）堆：通用内存池（也位于 RAM），灵活，需更多时间分配和清理

4）常量存储：存放在程序代码内部或 ROM

5）非 RAM 存储：例子是流对象和持久化对象（磁盘等永久存储空间）

2.2.2 特例：基本类型

基本类型不用 new 创建变量，而是创建一个并非引用的“自动”变量。这个变量直接存储“值”，并置于堆栈中，因此更加高效。

boolean　char　byte　short　int　long　float　double

包装器类：可以在堆中创建一个非基本对象，用来表示对应的基本类型。

自动装箱、自动拆箱

高精度计算的类：BigInteger、BigDecimal

2.2.3 Java 中的数组

创建一个数组对象，实际上就是创建了一个引用数组，自动初始化为 null。

2.3 永远不需要销毁对象

变量的生命周期

2.3.1 作用域（scope）（基本类型）

作用域由花括号的位置决定，决定了其内定义的变量名的可见性和生命周期。

2.2.2 对象的作用域

Java 对象不具备和基本类型一样的生命周期。当 new 创建一个 Java 对象时，它可以存活在作用域之外，当引用在作用域终点消失了，引用所指向的对象仍然继续占据内存空间，作用域之后，无法访问该对象，因为它的唯一引用超出了作用域范围。

Java 垃圾回收器，用于监视 new 创建的所有对象，辨别不会再被引用的对象，释放内存空间，供其他对象使用。

2.4 创建新的数据类型：类

关键字 class 用于引入一种新的类型，决定了某一类对象的外观和行为。

2.4.1 字段和方法

字段可以是任何类型的对象。

2.5 方法、参数和返回值

Java 方法决定对象能够接收什么样消息（提供什么样的服务）。

基本组成部分：名称、参数、返回值和方法体

方法签名（唯一地标识出某个方法） = 方法名 + 参数列表

调用方法的行为通常称为发送消息给对象。

2.5.1 参数列表

方法的参数列表指定要传递给方法什么样的消息。在参数列表中必须指定每个所传递对象的类型及名字。传递的实际是对象的引用。

return 关键字含义：一是已经做完了离开此方法；二是返回方法计算结果。

2.6 构建一个 Java 程序

2.6.1 名字可见性

Java 通过包即类库解决名字冲突，反转域名，句点代表子目录的划分。包名都是小写。

2.6.2 运用其他构件

import 指示编译器导入一个包，也就是一个类库。

通配符“*”

2.6.3 static 关键字

执行 new 来创建对象，存储空间才被分配，方法才供外界调用。

声明为 static，就意味着域和方法不会与包含它的那个类的任何对象实例关联在一起。

static 字段只有一份存储空间，为所有对象共享。

2.7 你的第一个 Java 程序

2.7.1 编译和运行

java 和 javac 命令

2.8 注释和嵌入式文档

段注释　　/*……*/

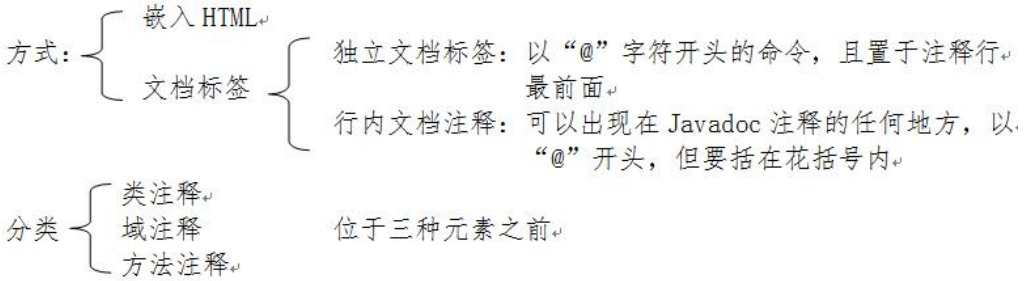
行注释　　//

2.8.1 注释文档

javadoc 用于提取注释文档的工具，在文件中查找特殊注释标签。

输出为 html 文件

2.8.2 语法



Javadoc 只能为 public 和 protected 成员进行文档注释，private 和包内成员注释自动被忽略（可以通过-private 标记，以便把 private 成员的注释包含在内）

2.8.3 嵌入式 HTML

Javadoc 通过生成的 HTML 文档传送 HTML 命令，使得充分利用 HTML 对代码进行格式化

所有类型的注释文档都支持嵌入式 HTML

2.8.4 一些标签示例

@see　{@link package.class # nemeber label}　{@docRoot}　　{@inheritDoc}

@version　@author　　@since　　@param　　@return　　@throws　　@deprecated

2.8.5 文档示例

2.9 编码风格

驼峰风格

第 3 章　操作符

3.1 更简单的打印语句

静态导入（static import）

3.2 使用 Java 操作符

几乎所有操作符只能操作“基本类型”，例外的操作符是“=”、“==”和“!=”，能操作所有对象；String 类型支持“+”和“+=”。

3.3 优先级

优先级决定了各部分的计算顺序，括号明确规定计算顺序。

3.4 赋值 （=）

基本类型：取右边的值，复制给左边；

引用：操作的是对象的引用，实际是将“引用”从一个地方复制到另一个地方。

对象赋值的“别名现象”（容易引起混乱）

3.4.1 方法调用中别名问题

原因：传递给方法的参数实际只是传递了一个引用。

3.5 算术操作符 （+、-、*、/、%、+=、-=、*=、/=、%=）

整数除法会直接去掉结果的小数位，而不是四舍五入的结果。

3.5.1 一元加、一元减（+、-）

3.6 自动递增和递减（++、--）（前缀式、后缀式）

3.7 关系操作符（==、>=、<=、>、<、!=）（equals）

3.7.1 测试对象的等价性

关系操作符==和!=适用于所有对象，除基本类型外的其他对象，比较的是对象的引用是否相同（即是否指向同一个对象）。比较对象的实际内容是否相同，必须使用所有对象都使用的特殊方法 equals()。

equals()方法默认行为是比较引用,如果希望得到希望的行为,必须在新类中覆盖 equal()方法。

3.8 逻辑操作符（&&、||、!）

根据参数的逻辑关系，生成一个布尔值（true 或 false）。“&&”、“||”、“!”操作只可应用于布尔值。

3.8.1 短路

是指一旦能够明确无误地确定整个表达式的值，就不再计算表达式余下部分了。

3.9 直接常量

后缀字符标志：L(Long)、F(Float)、D(Double)

前缀字符标志：0(八进制)、0X(十六进制)

3.9.1 指数计数法

1.39e-43f　　47e47d

3.10 按位操作符(&、|、^、~)

用来操作整数基本数据类型中的单个“比特”(bit)。按位操作符会对两个参数中对应的位执行布尔代数运算，最终生成结果。

3.11 移位操作符(<<、>>、>>>、<<=、>>=、>>>=)

<<　(低位补 0)

>>　(符号为正，高位补 0，符号为负，高位补 1)

>>>　(高位补 0)

3.12 三元操作符 if-else （? : ）

boolean-exp ? value0 : value1

3.13 字符串操作符+和+=

操作符重载（operator overloading），用于连接不同的字符串

3.14 使用操作符时常犯的错误

= 和 ==　&&和&　||和|

3.15 类型转换操作符

类型转换（cast）

窄化转化（narrowing conversion 显式，面临信息丢失危险）

扩展转化（widening conversion 不显式，不会造成信息丢失）

3.15.1 截尾和舍入

float 和 double 转型为整型值时，对数字实行截尾，舍入使用 Java.lang.Math.round() 方法。

3.16 Java 中没有 sizeof

第 4 章　控制执行流程

4.1 true 和 false

所有条件语句都利用条件表达式的真或假来决定执行路径。Java 不允许将一个数字作为布尔值使用。

4.2 if-else

控制流程的最基本形式。

4.3 迭代

while、do-while、for 用来控制循环，也称为迭代语句（iteration statement）。

Math.random()用于产生 0 到 1 之间的一个 double 值。

4.3.1 do-while

与 while 唯一区别是 do-while 的语句至少执行一次。

4.3.2 for

4.3.3 逗号操作符

Java 唯一用到逗号操作符的地方就是 for 循环的控制表达式，在控制表达式的初始化和步进控制部分，可以使用由逗号分隔的语句，而且那些语句均会独立执行。

4.4 Foreach 语法

用于数组和容器，自动产生每一项。

4.5 return

return 关键字的用途：一是指定一个方法返回什么值，二是导致当前的方法退出。

4.6 break 和 continue

break：用于强行退出循环，不执行循环剩余部分

continue：用于停止执行当前迭代，然后退出到循环起始处。

4.7 臭名昭著的 goto

标签是指后面跟冒号的标识符，起作用的唯一地方是在迭代语句之前（在标签和迭代之间置入任何语句都不好）。

```
label1:
outer-iteration {
   inner-iteration {
   //…
   break;     //(1)
   //…
   continue;  //(2)
   //…
   continue label1;  //(3)
//…
break label1;     //(4)
}
}
```

(1)break 终止内部迭代，回到外部迭代

(2)continue 使执行点移回到内部迭代起始处

(3)continue label1 同时中断内部及外部迭代，直接转到 label1 处，随后，继续迭代过程，却从外部迭代开始

(4)break label1 中断所有迭代，并回到 label1 处，但不重新进入迭代

重点：在 Java 里需要使用标签的唯一理由就是因为由循环嵌套存在，而且想从嵌套中 break 或 continue。

4.8 switch

switch 语句根据整数表达式的值，从一系列代码中选出一段去执行。
选择因子必须是 int 或 char 或 enum 那样的整数型。
case 均以 break 结尾，可使执行流程跳转到 switch 主体的尾部。

第 5 章　初始化和清理

5.1 用构造器确保初始化

Java 中通过提供构造器，类设计者可确保每个对象都会得到初始化。

构造器采用与类相同的名称。

new 创建对象，将会为对象分配存储空间，并调用相应构造器，确保正确初始化。

无参数构造器（默认构造器）是不接受任何参数的构造器。

Java 中，“初始化”和“创建”捆绑在一起，两者不能分离。

5.2 方法重载

方法名相同而形式参数不同的方法就是方法重载。

5.2.1 区分重载方法（方法签名）

每个重载的方法必须有一个独一无二的参数类型列表。(方法名相同，参数不同或参数顺序不同)

5.2.2 涉及基本类型的重载

如传入的实际参数类型小于（扩展转化）方法中声明的形式参数类型，实际类型就会被自动提升。

如传入的实际参数类型大于（窄化转化）方法中声明的形式参数类型，就必须通过显式的类型转换。

5.2.3 以返回值区分重载方法

行不通

5.3 默认构造器

默认构造器（又名“无参”构造器）是没有形式参数的，作用就是创建一个“默认对象”。如类中无构造器，编译器自动创建一个默认构造器，如已定义一个构造器(无论是否有参数)，编译器都不自动创建默认构造器。

5.4 this 关键字

“发送消息给对象”，编译器暗自把“所操作对象的引用”作为第一参数传递给方法。

this 关键字只能在方法内部使用，表示对“调用方法的那个对象”的引用。

5.4.1 在构造器中调用构造器

this 是指“当前对象”，而且它本身表示对当前对象的引用。在构造器中，如为 this 添加了参数列表，将产生对符合此参数列表的某个构造器的明确调用。this 只能调用一个构造器，必须将构造器置于最起始处（只能调用一个），否则编译器报错。除构造器之外，编译器禁止其他任何方法中调用构造器。

5.4.2 static 的含义

static 方法就是没有 this 的方法。通过类本身来调用 static 方法，实际上正是 static 方法的主要途径。

5.5 清理：终结处理和垃圾回收

Java 有垃圾回收器负责回收无用对象（必须是经由 new 分配）占据的内存资源。

对于“特殊”内存（并非使用 new），java 允许类中定义一个名为 finalize()方法。工作原理“假定”是这样的：一旦垃圾回收器准备好释放对象占用的存储空间，首先调用其 finalize()方法，并且在下一次垃圾回收动作发生时，才会真正回收对象占有的内存。编程陷阱：Java 里的对象并非总是被垃圾回收。即：

1、对象可能不被垃圾回收。

2、垃圾回收并不等于“析构”。

意味着不再需要某个对象前，如果必须执行某些动作，那么得自己去做。

5.5.1 finalize()的用途何在

3、垃圾回收至于内存有关。

使用垃圾回收器的唯一原因是为了回收程序不再使用的内存。对 finalize()的需求限制到一种特殊情况，即通过某种创建对象方式以外的方式对对象分配存储空间。不是进行普通清理工作的合适场合。

5.5.2 你必须实施清理

要清理一个对象，用户必须在需要清理的时刻调用执行清理动作的方法。Java 虚拟机并未面临内存耗尽，它不会浪费时间去执行垃圾回收以恢复内存。

5.5.3 终结条件

System.gc()用于强制进行终结动作。finalize 有趣用法是对象终结条件的验证。

5.5.4 垃圾回收器如何工作

*引用计数

该算法在 Java 虚拟机没被使用过，主要是循环使用问题，因为引用计数并不记录谁指向他，无法发现这些交互自引用的对象。

当引用连接到对象时，对象计数加 1，当引用离开作用域或被置为 null 时减 1。

遍历对象列表，计数为 0 就释放。

存在循环引用问题，A 引用 B，B 引用 A，那么 A，B 永远都不会被释放。

*标记算法

该算法的思想是从堆栈和静态存储区的对象开始，遍历所有引用，标记活的对象（对于活的对象一定能追溯到其存活在堆栈或静态存储区之中的引用)。对于标记后有两种处理方式：

（1）停止-复制（stop-and-copy）

所谓停止，就是停止在运行的程序，进行垃圾回收；所谓复制，就是将活得对象复制到另一个堆上，以使内存更紧凑。停止-复制的时机是内存较低的时候。

优点是：当大内存释放时，有利于整个内存重新分配。

问题是：停止干扰正常程序的运行；复制耗费大量的时间；如程序稳定，垃圾比较少，每次重新复制量是非常大的，非常不合算。

（2）标记-清扫（mark-and-sweep）

清除就是讲标记为非活得对象释放,不会发生任何复制动作，也必须暂停程序运行。优点就是程序比较稳定，垃圾比较少时，速度比较快。

问题是：停止程序运行是一个问题，只清除也会造成很多内存碎片。

两种方法都需要暂停程序是因为如果不暂停，标记会被运行的程序弄乱。

5.6 成员初始化

Java 尽力保证：所有变量在使用前都能得到恰当的初始化。

方法的局部变量：Java 以编译时错误的形式来保证。(如采取默认值容易掩盖错误)

类的数据成员（即字段)：类的基本类型数据成员会有一个初始值；类的对象引用会获得特殊值 null。

5.6.1 指定初始化

指定初始化：在定义类成员变量的地方为其赋值或通过方法提供初值，方法可以带有参数，前提是参数必须已经被初始化。

5.7 构造器初始化

可以通过构造器来进行初始化。但无法阻止自动进行初始化的进行，它在构造器被调用之前发生。

5.7.1 初始化顺序

在类的内部，变量定义的先后顺序决定了初始化的顺序。即使变量定义散布于方法定义之间，它们仍旧会在任何方法（包括构造器）被调用之前得到初始化。

5.7.2 静态数据的初始化

静态数据只占有一份存储区域，static 关键字不能应用于局部变量，只能作用于域。默

认初始化和指定初始化同于非静态数据。

★对象的创建过程：

1、即使没有显式地使用 static 关键字，构造器实际上也是静态方法。因此，当首次创建对象时，或者类的静态方法被首次访问时，Java 解释器必须查找类路径，以定位.class 文件。

2、载入.class 文件，有关静态初始化的所有动作都会执行。因此，静态初始化只在 Class 对象首次加载的时候进行一次。

3、当用 new 创建对象的时候，首先在堆上为对象分配足够的存储空间。

4、这块存储空间会被清零，这就自动地将对象中所有基本类型数据设置成了默认值，而引用则被设置成了 null。

5、执行所有出现于字段定义处的初始化动作。

6、执行构造器。

5.7.3 显式的静态初始化

“静态子句”(也叫“静态块”)。实际只是一段跟在 static 关键字后面的代码。与初始化动作一样，仅执行一次，当首次生成类的一个对象时，或是首次访问属于类的静态数据成员时。

5.7.4 非静态实例初始化

实例初始化如同静态块，只不过少了 static 关键字。是在构造器之前执行的。

5.8 数组初始化

数组是相同类型，用一个标识名称封装到一起的一个对象序列或基本类型数据序列。

要定义一个数组，只需在类型名后加上一对空方括号（int []），编译器不允许指定数组大小。此时，只是数组的一个引用，没有分配任何空间。

①int[] a1 = {1,2,3,4,5};特殊的初始化表达式，必须在创建数组的地方出现。存储空间的分配（等价于使用 new）将由编译器负责。

②基本类型数组，int[] a1 = new int[5];数组元素中的基本数据类型会自动初始化为空值。

③非基本类型数组，实际就是创建了一个引用数组。直到通过创建对象并把对象赋值给引用，初始化才算结束。否则，试图使用数组中的空引用，就会在运行时产生异常。

初始化方式： Integer[] a = {new Integer(1),new Integer(2),3};

String[] a = new String[]{“fiddle”, “de”, “dum”};

5.8.1 可变参数列表

形式：①void printArray(Object[] args)

②void printArray(Object…args)

5.9 枚举类型

enum 关键字，可使我们在需要群组并使用枚举型集时，可以很方便地处理。

创建枚举类型，它具有具名值，由于枚举类型实例是常量，因此都用大写字母表示。

使用 enum，创建该类型的引用，并将其赋值给某个实例。创建 enum，编译器自动添加一些特性，如创建 toString()方法及 static values()方法。

第 6 章　访问权限控制

面向对象需要考虑的基本问题：“如何把变动的事物与保持不变的事物区分开来”。

Java 提供访问权限修饰词，以供类库开发人员向客户端程序员指明哪些是可用的，哪些是不可用的，访问权限控制等级，从最大权限到最小权限依次为：public、protected、包访问权限（没有关键字）和 private。

6.1 包：库单元

包内包含有一组类，它们在单一的名字空间之下被组织在一起。

import 关键字导入，就是要提供一个管理名字空间的机制。

类存于单一文件之中，并专为本地使用（local use），实际类位于未命名包或默认包。

Java 源代码文件通常被称为编译单元。编译单元必须有一个后缀名.Java，在编译单元内则可以有一个 public 类，该类的名称必须与文件的名称相同。每个编译单元只能有一个 public 类，否则编译器不接受。编译单元 public 类之外的类，包之外无法看到，主要为 public 类提供支持。

6.1.1 代码组织

编译一个.Java 文件时，在.Java 文件中的每个类都会有一个输出文件，输出文件的名称与.Java 文件中每个类的名称相同，只是后缀名为.class。

Java 可运行程序是一组可以打包并压缩为一个 Java 文档文件 （Jar，使用 Java 的 Jar 文档生成器）的.class 文件。Java 解释器负责这些文件的查找、装载和解释。

类库实际上是一组类文件。其中每个文件都有一个 public 类，以及任意数量的非 public 类。因此每个文件都有一个构件。如希望这些构件从属于同一个群组，就可以使用关键字 package。

package 语句，必须是文件中除注释以外的第一句程序代码。在文件起始处写：package access; 就表示声明该编译单元是名为 access 的类库的一部分。或者说，正在声明该编译单元中的 public 类名称是位于 access 名称的保护伞下。任何想使用该名称的人必须使用前面给出的选择，指定全名或者与 access 结合使用关键字 import。

牢记：package 与 import 关键字允许做的，是将单一的全局名字空间分割开。

6.1.2 创建独一无二的包名

Java 解释器的运行过程如下：首先，找出环境变量 CLASSPATH。CLASSPATH 包含一个或多个目录，用来查找.class 文件的根目录。从根目录开始，解释器获取包的名称并将每个句点替换成反斜杠，以从 CLASSPATH 根中产生一个路径名称。得到的路径会与 CLASSPATH 中的各个不同的项相连接，解释器就在这些目录中查找与你所要创建的类名称相关的.class 文件。

冲突

将两个含有相同名称的类库以“*”形式同时导入，编译器报错，强制你明确指出。

6.1.3 定制工具库

创建工具库来减少或消除重复的程序代码。

6.1.4 用 import 改变行为

6.1.5 对使用包的忠告

创建包，在给定包的名称的时候隐含地指定了目录结构。

6.2 Java 访问权限修饰词

访问权限修饰词置于类中每个成员的定义之前。仅控制它所修饰的特定定义的访问权。

6.2.1 包访问权限

包访问权限，处于同一个编译单元中的所有类彼此之间都是自动可访问的。

取得成员的访问权的途径是：

1、使该成员成为 public。

2、通过不加访问权限修饰词并将其他类放置于同一个包内的方式给成员赋予包访问权。

3、继承而来的类既可以访问 public 成员也可以访问 protected 成员。

4、提供访问器（accessor）和变异器（mutator）方法，以读取和改变数值。

6.2.2 public：接口访问权限

默认包

处于相同目录并且没有设定任何包名称。Java 将这样的文件自动看作是隶属于该目录的默认包之中。

6.2.3 private：你无法访问

除了包含该成员的类之外，其他任何类都无法访问这个成员。

6.2.4 protected：继承访问权限

既提供继承访问权限，也提供包访问权限。

6.3 接口和实现

访问权限的控制被称为是具体实现的**隐藏**。**把数据和方法包装进类中，以及具体实现的隐藏**，常共同被称作是**封装**。其结果是一个同时带有特征和行为的数据类型。

访问权限控制将权限的边界划在了数据类型的内部原因是：一是要设定客户端程序员可以使用和不可以使用的界限。二是将接口和具体实现进行分离。

6.4 类的访问权限

访问权限修饰词也可以用于确定库中哪些类对于该库的试用者是可用的。

额外的限制：

1、每个编译单元（文件）都只能有一个 public 类。

2、public 类的名称必须完全与含有该编译单元的文件名相匹配，包括大小写。

3、编译单元不带 public 类也是可以的。

应尽可能地总是将域指定为私有的，但通常来说，将与类（包访问权限）相同的访问权限赋予方法也是很合理的。

类的访问权限只有两种选择：包访问权限或 public。

如不希望其他人对类拥有访问权限，可以将所有构造器指定为 private，从而组织任何人创建给类的对象，例外，在该类的 static 成员内部可以创建。

单例（singleton）设计模式

```
class Soup2
{
   private Soup2()
   {
      System.out.println("Create Soup2");
   }
   //(2)Create a static object and return a reference
   //upon request.(The "Singleton" pattern)
   private static Soup2 ps1 = new Soup2();
   public static Soup2 access()
   {
      return ps1;
   }
}
```

第 7 章　复 用 类

复用类的方法：

1、组合：在新类中产生现有类的对象。复用了现有程序代码的功能，而非它的形式。

2、继承：按照现有类的类型来创建新类。采用现有类的形式并在其中添加新代码。

7.1 组合语法

对于非基本类型，只需将对象引用置于新类中即可。

特殊方法：toString()，编译器需要 String 而只是一个对象时，该方法便会被调用。

类中引用会被自动初始化为 null。引用的初始化位置：

1、在定义对象的地方。(指定初始化)；

2、在类的构造器中；

3、使用对象之前（惰性初始化)；

4、使用实例初始化。

7.2 继承语法

创建一个类时，除非明确指出要从其他类中继承，否则隐式地从 Java 的标准根类 Object 进行继承。声明通过关键字 extends 实现，会自动得到基类中所有的域和方法。

一般规则：为了继承，将所有的数据成员指定为 private，将所有的方法指定为 public。

7.2.1 初始化基类

继承并不只是复制基类接口。当创建导出类的对象时，该对象包含了一个基类的子对象。这个子对象与用基类直接创建的对象是一样的。二者区别在于，后者来自外部，而基类的子对象被包装在导出类对象内部。

通过在构造器中调用基类构造器来保证基类子对象的正确初始化。Java 会自动在导出类的构造器中插入对基类构造器的调用。

带参数的构造器

如没有基类没有默认构造器或想调用一个带参数的基类构造器，就必须用关键字 super 显式编写调用基类构造器的语句，并配以适当的参数列表。

7.3 代理

代理，Java 没有提供对它的直接支持。是继承和组合之间的中庸之道，将一个对象置于所要构造的类中（就像组合)，同时在新类中暴露该成员对象的所有方法（就像继承)。

7.4 结合使用组合和继承

编译器强制初始化基类，并要求在构造器起始处就如此，但它并不监督必须将成员对象也初始化。

7.4.1 保证正确清理

Java 习惯只是忘掉而不是销毁对象，并让垃圾回收器在必要时释放其内存。

执行类的所有特定的清理动作，其顺序同生成顺序相反，首先调用基类清理方法。

垃圾回收器可能永远也无法被调用，即使被调用，也可能以任何它想要的顺序来回收对象。最好的办法是除了内存以外，不能依赖垃圾回收器做任何事。

7.4.2 名称屏蔽

Java 的基类拥有已被多次重载的方法名称，那么在导出类中重新定义该方法名称并不会屏蔽掉其在基类中的任何版本。

使用与基类完全相同的特征签名及返回类型来覆盖具有相同名称的方法。

@Override 注解使用

7.5 在组合与继承之间选择

组合和继承都允许在新的类中放置子对象，组合是显式地这样做，继承则是隐式地做。

组合用于在新类中使用现有类的功能而非它的接口。(has-a)

继承用于使用某个现有类，并开发一个特殊版本。(is-a)

7.6 protected 关键字

protected 指明“就类用户而言，这是 private 的，但对于任何继承于该类的导出类或其他任何位于同一个包内的类来说，它他却是可以访问的”。

7.7 向上转型

“导出类是基类的一种类型。”用来表现导出类和基类之间的关系。

将导出类引用转换为基类引用的动作，称之为向上转型。

7.7.1 为什么称为向上转型

以传统的类继承图的绘制方法为基础的：将根置于页面的顶端，然后逐渐向下。

导出类是基类的一个超集。它比基类含有更多的方法，在向上转型过程中，唯一可能发生的事情是丢失方法，而不是获取它们。

7.7.2 再论组合与继承

选择继承或组合，一个清晰的判断方法就是问一问自己是否需要从新类向基类进行向上转型。

7.8 final 关键字

final 的三种情况：数据、方法和类

7.8.1 final 数据

编译期常量：由 final 和 static 表示的基本数据类型。定义时，必须进行赋值。

final 用于基本类型：使数值恒定不变

final 用于引用：使引用恒定不变（无法改变引用指向，但对象其自身是可以被修改的）

空白 final

空白 final 是指被声明为 final 但又未给定初值的域。必须保证使用前被初始化（在构造器中进行初始化）。

final 参数

final 参数意味着无法在方法中更改参数引用所指向的对象。

7.8.2 final 方法

使用 final 方法的原因：一是把方法锁定，以防任何继承类修改它的含义；二是效率（不建议作为因素考虑）。

final 和 private 关键字

类中所有的 private 方法都隐式地指定为是 final 的。（因为无法取用 private 方法，也就无法覆盖它）

“覆盖”只有在某方法是基类的接口的一部分才会出现。如方法为 private，它就是不是接口的一部分。它仅是一些隐藏于类中的程序代码，只不过是具有相同的名称而已。

7.8.3 final 类

将类定义为 final 类，就表示不打算继承该类。

7.8.4 有关 final 的忠告

7.9 初始化及类的加载

Java 类的编译代码在独立的文件中，在初次使用时才加载。通常是指加载发生于创建类的第一个对象之时，或访问 static 域或方法时才加载。

初次使用之处也是 static 初始化发生之处。所有的 static 对象和 static 代码段都会在加载时依程序中的顺序（即定义类的书写顺序）而依次初始化。定义为 static 的东西只会被初始化一次。

7.9.1 继承与初始化

加载过程：对类进行加载过程中，编译器注意到它有一个基类（由关键字 extends 得知），于是对基类继续进行加载，不管是否打算产生一个基类的对象，这都要发生。递归加载基类，根基类中 static 初始化即会被执行，然后是下一个导出类，以此类推。原因是导出类的 static 初始化可能会依赖于基类成员能否正确初始化。

加载完成，对象就可以被**创建**：

1、对象中所有的基本类型都被设成默认值，对象引用被设为 null—通过将对象内存设为二进制零值而一举生成的。

2、基类的构造器被调用（可以自动调用或 super 指定对基类构造器调用）

3、基类的构造器和导出类的构造器一样，以相同的顺序经历相同的过程，基类构造器完成之后，实例变量按照其次序被初始化。

4、构造器其余部分被执行。

第8章 多　　态

在面向对象的程序设计语言中，多态是继数据抽象和继承之后的第三种基本特征。

“封装”通过合并特征和行为来创建新的数据类型。“实现隐藏”则通过细节“私有化”把接口和实现分离开来。多态的**作用**则是消除类型之间的**耦合**关系。

8.1 再论向上转型

把对某个对象的引用视为对其基本类型的引用的做法被称作向上转型。

8.1.1 忘记对象类型

方法仅接收基类作为参数，而不是特殊的导出类。这真是多态所允许的。

8.2 转机

绑定理解

8.2.1 方法调用绑定

将一个方法调用同一个方法主体关联起来被称为绑定。若在程序执行前进行绑定（如果有的话，由编译器和连接程序实现），叫做前期绑定。

在允许时根据对象的类型进行绑定，就是后期绑定（动态绑定或运行时绑定）。要实现后期绑定，就必须具有某种机制，以便在运行时能判断对象的类型（“类型信息”）。

Java 中除了 static 方法和 final 方法（private 方法属于 final 方法）之外，其他所有的方法都是后期绑定。根据设计来决定是否使用 final。

8.2.2 产生正确的行为

发送消息给对象，让对象而不是引用去断定应该做什么。

8.2.3 可扩展性

方法只与基类接口通信，这样的程序是可扩展的。因为可以从通用的基类继承出新的数据类型，从而新添一些功能。那些操纵基类接口的方法不需任何改动就可以应用于新类。

多态是一项让程序员“将改变的事物与未变的事物分离开来”的重要技术。

8.2.4 缺陷：“覆盖”私有方法

导出类方法与基类私有方法签名相同,无法通过基类引用指向导出类对象调用该方法(编译报错)。

8.2.5 缺陷：域与静态方法

域是在编译器进行解析，由引用的断定访问的域。

静态方法是与类，而并非与单个的对象相关联的。

8.3 构造器和多态

构造器并不具有多态性（实际上是 static 方法，只是隐式的）。

8.3.1 构造器的调用顺序

基类的构造器总是在导出类的构造过程中被调用，而且按照继承层次逐渐向上链接，以使每个基类的构造器都能得到调用。这样的意义在于：构造器具有一项特殊任务:检查对象是否被正确地构造。导出类只能访问自己的成员，不能访问基类中的成员（基类成员通常是 private)。只有基类的构造器才具有恰当的知识和权限来对自己的元素进行初始化。(调用基类构造器进行初始化原因)

在导出类的构造器主体中，如没明确指定某个基类构造器，它就会调用默认构造器。如不存在默认构造器，编译器报错。

复杂对象的调用顺序：

1、调用基类构造器。反复递归，直到最低层的导出类。

2、按声明顺序调用成员的初始化方法。

3、调用导出类构造器的主体。

8.3.2 继承和清理

销毁的顺序应该和初始化顺序相反，首先对导出类进行清理，然后才是基类。

8.3.3 构造器内部的多态方法的行为

初始化的实际过程：

1、在其他任何事物发生之前，将分配给对象的存储空间初始化成二进制零。

2、调用基类构造器。此时，如果调用导出类覆盖基类的方法，将调用覆盖后的方法。

3、按照声明的顺序调用成员的初始化方法。

4、调用导出类的构造器主体。

编写构造器有效的准则：“用尽可能简单的方法使对象进入正常状态，如果可以的话，避免调用其他方法。”

8.4 协变返回类型

协变返回类型表示导出类中的被覆盖方法可以返回基类方法的返回类型的某种导出类型。

8.5 用继承进行设计

状态模式：允许一个对象在其内部状态改变时改变它的行为。对象看起来视乎修改了它的类。

通用的准则：“用继承表达行为间的差异，并用字段表达状态上的变化”。

8.5.1 纯继承与扩展

纯继承是基类和导出类的接口完全相同。(is-a)

扩展接口（is-like-a）

8.5.2 向下转型与运行时类型识别

Java 语言中，所有转型都会得到检查，如果不是希望的类型转型，就会返回一个 ClassCastException 异常。这种在运行期间对类型进行检查的行为称作“运行时类型识别”（RTTI)。

RTTI 还可以在试图向下转型之前，查看你所要处理的类型。

第9章 接　　口

抽象类是普通类与接口之间的一种中庸之道。

9.1 抽象类和抽象方法

通用接口

建立通用接口唯一理由是不同子类可以用不同的方式表示接口。通用接口建立起一种基本形式，以此表示所有导出类的共同部分。

创建抽象类是希望通过这个通用接口操纵一系列类。

Java 提供一个叫做抽象方法的机制，抽象方法声明的语法：abstract void f();包含抽象方法的类叫做抽象类。

试图产生抽象类的对象，编译器报错。

从抽象类继承，并创建该新类的对象，那么就必须为基类中所有抽象方法提供方法定义。如果不如此，那么导出类便也是抽象类，且编译器将会强制用 abstract 关键字限定这个类。

抽象类是很有用的重构工具，因为它使得我们可以很容易地将公共方法沿着继承层次结构向上移动。

9.2 接口

interface 关键字产生一个完全抽象的类，它根本没有提供任何具体实现。

一个接口表示：“所有实现了该特定接口的类看起来都像这样”。因此，任何使用某特定接口的代码都知道可以调用该接口的哪些方法，而且仅需知道这些。因此，接口被用来建立类与类之间的协议（protocol）。

接口也可以包含域，但是隐式地是 static 和 final 的。

要让一个类遵循某个特定接口（或者是一组接口），需要使用 implements 关键字，它表示：“interface 只是它的外貌，但是现在我要声明它是如何工作的”。

接口中的方法都为 public。在方法的继承过程中，其可访问权限不可被降低。

9.3 完全解耦

方法操作的是接口，可以编写复用性更好的代码。

策略设计模式：创建一个能够根据所传递的参数对象的不同而具有不同行为的方法。这类方法包含所要执行的算法中固定不变的部分，而“策略”包含变化的部分。策略就是传递进去的参数对象，它包含要执行的代码。

适配器设计模式：适配器中的代码将接受你所拥有的接口，并产生你所需要的接口。

```
class FilterAdapter implements Processor {
   Filter filter;
   public FilterAdapter(Filter filter) {
   this.filter = filter;
}
public String name() { return filter.name(); }
public Waveform process(Object input) {
   return filter.process((Waveform)input);
}
}
public class FilterProcessor {
   public static void main(String[] args) {
   Waveform w = new Waveform();
   Apply.process(new FilterAdapter(new LowPass(1.0)),w);
}
}
```

在这种使用适配器的方式中，FIlterAdapter 的构造器接受你所拥有的接口 Filter，然后生成你所需要的 Processor 接口的对象。在 FilterAdapter 类中用到了代理。

9.4 Java 中的多重继承

具体类必须放在前面，后面跟着才是接口（否则编译器会报错）。

当想要创建对象时，所有的定义首先必须都存在。

使用接口的核心原因：一是能够向上转型为多个基类型（以及由此带来的灵活性）；二是与使用抽象基类相同：防止客户端程序员创建该类的对象，并确保这仅仅是建立一个接口。

9.5 通过继承来扩展接口

通过继承在新接口中组合数个接口。使用关键字 extends，逗号分隔多个接口名。

9.5.1 组合接口时的名字冲突

相同的方法不会有什么问题，但如果他们的签名相同返回类型不同，则会编译报错。

9.6 适配接口

策略设计模式：编写一个执行某些操作的方法，而该方法将接受一个同样是你指定的接口。主要就是声明:“你可以用任何你想要的对象来调用我的方法,只要你对象遵循我的接口”

适配器设计模式：

9.7 接口中的域

接口中的任何域都自动是 static 和 final 的。

9.7.1 初始化接口中的域

接口中定义的域不能是“空 final”，但是可以被非常量表达式初始化。在类第一次被加载时初始化。

9.8 嵌套接口

接口可以嵌套在类或其他接口中。

实现一个 private 接口只是一种方式，它可以强制该接口中的方法定义不要添加任何类型信息（也就是说，不允许向上转型）。

9.9 接口和工厂

工厂方法设计模式：生成遵循某个接口的对象的典型方式。在工厂对象上调用的是创建方法，而该工厂对象将生成接口的某个实现的对象。通过这种方式，我们的代码完全与接口的实现分离，这就使得我们可以透明地将某个实现替换为另一个实现。

```
//:interface/Factories.java

interface Service
{
   void method1();
   void method2();
}
interface ServiceFactory
{
   Service getService();
}
class Implementation1 implements Service
{
   Implementation1() {}  //Package access
   public void method1() { System.out.println("Implementation1 method1"); }
   public void method2() { System.out.println("Implementation1 method2"); }
}
class Implementation1Factory implements ServiceFactory
```

```
{
   public Service getService()
   {
      return new Implementation1();
   }
}
class Implementation2 implements Service
{
   Implementation2() {}
   public void method1() { System.out.println("Implementation2 method1"); }
   public void method2() { System.out.println("Implementation2 method2"); }
}
class Implementation2Factory implements ServiceFactory
{
   public Service getService()
   {
      return new Implementation2();
   }
}
public class Factories
{
   public static void serviceConsumer(ServiceFactory fact)
   {
      Service s = fact.getService();
      s.method1();
      s.method2();
   }
   public static void main(String[] args)
   {
      serviceConsumer(new Implementation1Factory());
      //Implementations are completely interchangeable:
      serviceConsumer(new Implementation2Factory());
   }
}
```

9.10 总结

任何抽象性都应该是应真正的需求而产生的。

恰当的原则应该是优先选择类而不是接口。从类开始，如果接口的必需性变得非常明确，那么就进行重构。

第 10 章　内　部　类

可以将一个类的定义放在另一个类的定义内部，这就是内部类。

10.1 创建内部类

从外部类的非静态方法之外的任意位置创建某个内部类的对象，那么必须具体指明这个对象的类型：OuterClassName.InnerClassName。

10.2 链接到外部类

内部类自动拥有外部类的所有元素的访问权。这是如何做到的呢？当某个外部类的对象创建了一个内部类对象时，此内部类对象必定会秘密地捕获一个指向那个外部类对象的引用（编译器自动生成一个外部类附加实例于 this$0）。然后，在你访问外部类的成员时，就是用那个引用来选择外部类的成员。

内部类的对象只能在与外部类对象相关联的情况下才能被创建。构建内部类对象时需要一个指向其外部类对象的引用。

迭代器设计模式

10.3 使用.this 与.new

需要生成对外部类对象的引用，可以使用外部类的名字后面紧跟圆点和 this。

创建内部类的对象，必须在 new 表达式中提供对其他外部类对象的引用，使用 .new 语法。

OuterClassName.this 生成对外部类对象的引用。

OuterClassName outer = new OuterCLassName();

OuterClassName.InnerClassName inter = outer.new InnerClassName();必须使用外部类对象来创建内部类对象。

在拥有外部类对象之前是不可能创建内部类对象的，这是因为内部类对象会暗暗地连接到创建它的外部类对象上。

10.4 内部类与向上转型

内部类向上转型为其基类或接口的时候，内部类就有了用武之地。因为此内部类（某个接口的实现）能够完全不可见，并且不可用。所得到的只是指向基类或接口的引用，方便隐藏实现细节。

从客户端程序员的角度，由于不能访问任何新增加的、原本不属于公共接口的方法，所有扩展接口是没有价值的，也给 Java 编译器提供了生成更高效代码的机会。

10.5 在方法和作用域内的内部类

在方法里面或任意的作用域内定义内部类的理由：一是实现了某类型的接口，于是可以创建并返回对其的引用；二是要解决复杂问题，想创建一个类来辅助你的解决方案，又不希望类是公共可用的。

在方法的作用域内创建一个完整的类，这被称为**局部内部类**。

局部内部类是方法的一部分，而不是类（OuterClass）的一部分，因此在方法之外，不能访问局部内部类。出现在 return 语句的向上转型——返回的是局部内部类的基类的引用。当然在方法中定义了内部类，并不意味着一旦方法执行完毕，该类就不可用了。

10.6 匿名内部类

“创建一个继承自 Contents 的匿名内部类的对象。”实际通过 new 表达式返回的引用被自动向上转型为对 Contents 的引用。

基类需要一个有参数的构造器，只需简单传递合适的参数给基类的构造器即可。此时，不要求变量一定是 final 的，因为变量被传递给匿名类的基类的构造器，并不会在匿名类内部被直接使用。

如果定义了一个匿名内部类，并且希望它使用一个在其外部定义的对象，编译器会要求其参数引用是 final 的。

匿名内部类无命名构造器（因为它根本没名字），只能通过实例初始化，达到匿名内部类创建一个构造器的效果。

匿名内部类既可以扩展类，也可以实现接口，但不能两者兼备，实现接口，也只能实现一个接口。

10.6.1 再访工厂方法

```
//:innerclasses/Factories.java

interface Service{
   void method1();
   void method2();
}
interface ServiceFactory{
   Service getService();
}
class Implementatian1 implements Service{
   private Implementatian1() {}
   public void method1() { System.out.println("Implementation1 method1"); }
   public void method2() { System.out.println("Implementation1 method2"); }
   public static ServiceFactory factory =
       new ServiceFactory(){
           public Service getService(){
               return new Implementatian1();
           }
       };
}
class Implementatian2 implements Service{
   private Implementatian2() {}
   public void method1() { System.out.println("Implementation2 method1"); }
   public void method2() { System.out.println("Implementation2 method2"); }
   public static ServiceFactory factory =
       new ServiceFactory(){
           public Service getService(){
               return new Implementatian2();
           }
       };
}
public class Factories{
   public static void serviceConsumer(ServiceFactory fact){
       Service s = fact.getService();
       s.method1();
       s.method2();
   }
   public static void main(String[] args){
       serviceConsumer(Implementatian1.factory);
       serviceConsumer(Implementatian2.factory);
   }
```

}

优先使用类而不是接口。

10.7 嵌套类

如不需要内部类对象与外围类对象之间有联系，那么可以将内部类声明为 static。通常称为嵌套类。

嵌套类意味着：

1）要创建嵌套类的对象，并不需要其外部类的对象。

2）不能从嵌套类的对象中访问非静态的外部类对象。

嵌套类与普通内部类区别：普通内部类的字段与方法，只能放在类的外部层次上，所以普通的内部类不能有 static 数据和 static 字段，也不能包含嵌套类。但是嵌套类可以包含这些东西。

10.7.1 接口内部类

放置到接口中的任何类都自动地是 public 和 static 的。因为类是 static 的，只是将嵌套类置于接口的命名空间内，并不违反接口的规则。

如想要创建某些公共代码，使得它们可以被某个接口的所有不同实现所共有，那么使用接口内部的嵌套类会显得很方便。

10.7.2 从多层嵌套类中访问外部类的成员

一个内部类被嵌套多少层并不重要——它能透明地访问所有它嵌入的外部类的所有成员。

10.8 为什么需要内部类

内部类继承自某个类或实现某个接口，内部类的代码操作创建它的外部类的对象，所以可以认为内部类提供了某种进入其外部类的窗口。

如果只是需要一个对接口的引用，为什么不通过外部类实现那个接口呢？答案是：**“如果这能满足需求，那么就应该这样做。”**那么内部类实现一个接口与外部类实现这个接口有什么区别？答案是：后者不是总能享用到接口带来的方便，有时需要用到接口的实现。所以使用内部类最吸引人的原因：每个内部类都能独立地继承自一个（接口的）实现，所以无论外部类是否已经继承了某个（接口的）实现，对于内部类都没有影响。

内部类使得多重继承解决方案变得完整，允许继承多个非接口类型。

使用内部类，可以获得其他一些特性：

1）内部类可以有多个实例，每个实例都有自己的状态信息，并且与其外部类对象的信息相互独立。（iterator 和 iteratable 接口）

2）在单个外部类中，可以让多个内部类以不同的方式实现同一个接口，或继承同一个类。

3）创建内部类对象的时刻并不依赖于外部类对象的创建。

4）内部类并没有令人迷惑的“is-a”关系；它就是一个独立的实体。

10.8.1 闭包与回调

闭包（closure）是一个可调用的对象，它记录了一些信息，这些信息来自于创建它的作用域。通过定义，可以看出内部类是面向对象的闭包，因为它不仅包含外部类对象（创建内部类的作用域）的信息，还自动拥有一个指向此外部类对象的引用，在此作用域内，内部类有权操作所有的成员，包括 private。

Java 最引人争议的问题之一就是，人们认为 Java 应该包含某种类似指针的机制，以允许回调（callback）。通过回调，对象能够携带一些信息，这些信息允许它在稍后的某个时刻调用初始的对象。

通过内部类提供闭包的功能是优良的解决方案，它比指针更灵活、更安全。

回调的价值在于它的灵活性——可以在运行时动态地决定需要调用什么方法。

```
//:innerclasses/CallBacks.java
//Using inner classes for callbacks
```

```
//package innerclasses;

interface Incrementable {
   void increment();
}

//Very simple to just implement the interface:
class Callee1 implements Incrementable {
   private int i = 0;
   public void increment() {
       i++;
       System.out.println(i);
   }
}

class MyIncrement {
   public void increment() { System.out.println("Other operation"); }
   static void f(MyIncrement mi) { mi.increment(); }
}

//If your class must implement increment() in
//some other way,you must use an inner class:
class Callee2 extends MyIncrement {
   private int i = 0;
   public void increment() {
       super.increment();
       i++;
       System.out.println(i);
   }
   private class Closure implements Incrementable {
       public void increment() {
           //Specify outer-class method.otherwise
           //you'd get an infinite recursion:
           Callee2.this.increment();
       }
   }
   Incrementable getCallbackReference() {
       return new Closure();
   }
}

class Caller {
   private Incrementable callbackReference;
   Caller(Incrementable cbh) { callbackReference = cbh; }
   void go() { callbackReference.increment(); }
}
```

```
public class CallBacks {
   public static void main(String[] args) {
      Callee1 c1 = new Callee1();
      Callee2 c2 = new Callee2();
      MyIncrement.f(c2);
      Caller caller1 = new Caller(c1);
      Caller caller2 = new Caller(c2.getCallbackReference());
      caller1.go();
      caller1.go();
      caller2.go();
      caller2.go();

   }
}
```

外部类实现一个接口与内部类实现此接口之间的区别。Callee2 继承自 MyIncreament，后者已经有了一个不同的 increment()方法，并且与 Incrementable 接口期望的 increment()方法完全不想关。于是只能使用内部类独立地实现 Incrementable。

内部类 Closure 实现了 Incrementable，以提供一个返回 Callee2 的“钩子”(hook)——而且是一个安全的钩子。

Caller 的构造器需要一个 Incrementable 的引用作为参数（虽然可以在任意时刻捕获回调引用），然后在以后的某个时刻，Caller 对象可以使用此引用回调 Callee 类。

10.8.2 内部类与控制框架

控制框架（control framework）

应用程序框架（application framework）就是被设计用以解决某类特定问题的一个类或一组类。要运用某个应用程序框架，通常是继承一个或多个类，并覆盖某些方法。在覆盖后的方法中，编写代码定制应用程序框架提供的通用解决方案，以解决你的特定问题（这是设计模式中模板方法的例子）。模板方法包含算法的基本结构，并且会调用一个或多个可覆盖的方法，以完成算法动作。**设计模式总是将变化的事物与保持不变的事物分离开，**在这个模式中，**模板方法**是保持不变的事物，而可覆盖的方法是变化的事物。

控制框架是一类特殊的应用程序框架，它用来响应事件的需求。主要用来响应事件的系统被称为事件驱动系统。

1、Event 接口描述了要控制的事件。因为其默认的行为是基于时间去执行控制，所以用抽象类代替实际的接口。

2、Controller 用于管理并触发事件的实际控制框架。Event 对象被保存在 List<Event>类型的容器对象中。

3、在目前的设计中并不知道 Event 到底做了什么。这是设计的关键所在，“使变化的事物与不变的事物相互分离”。“变化向量”就是各种不同的 Event 对象所具有的不同行为，通过创建不同的 Event 子类来表现不同的行为。

这正是内部类要做的事，内部类允许：

1）控制框架的完整实现是由单个的类创建的，从而使得实现的细节被封装了起来。内部类用来表示解决问题所必须的各种不同的 action();

2）内部类能够容易地访问外围类的任意成员，所以可以避免这种实现变得笨拙。

10.9 内部类的继承

内部类的构造器必须连接到指向其外部类对象的引用，所以在继承内部类的时候，那个指向外部类对象的“秘密的”引用必须被初始化。必须在导出类构造器内使用如下语法：

enclosingClassReference.super();

这样才能提供了必要的引用，然后程序才能编译通过。

10.10 内部类可以被覆盖码

继承外部类并重新定义内部类，两个内部类是完全独立的两个实体，各自在自己的命名空间内。

10.11 局部内部类

局部内部类不能有访问说明符，因为它不是外部类的一部分；它可以访问当前代码块内的常量，以及此外部类的所有成员。

使用局部内部类而非匿名内部类的唯一理由是：我们需要一个已命名的构造器，或者需要重载构造器，而匿名内部类只能用于实例初始化。

10.12 内部类标识符

LocalInnerClass$1.class	匿名内部类
LocalInnerClass$1LocalCounter.class	局部内部类（类名：LocalCounter）
LocalInnerClass$InnerClass.class	内部类（类名：InnerClass）
LocalInnerClass.class	外部类（类名：LocalInnerClass）

第 11 章　持 有 对 象

Java 实用类库提供一套完整的容器类来解决数组尺寸固定所带来的限制。基本类型是 List、Set、Queue 和 Map。这些对象类型也称为集合类。

11.1 泛型和类型安全的容器

@SuppressWarning 注解及其参数表示只有有关“不受检查的异常”的警告信息应该被抑制。

Java SE5 之前的容器存在主要问题是编译器允许向容器中插入不正确的类型（容器通过持有 Object 引用持有不同类型对象）。使用 Java 泛型可以在编译器防止错误类型的对象放置到容器中。语法：ArrayList<Apple> apples = new ArrayList<Apple>();

通过泛型，不仅知道编译器将会检查你放置在容器中的对象类型，而且使用容器的对象时，可以使用更加清晰的语法。

指定了某个类型作为泛型参数时，向上转型也可以作用于泛型。

11.2 基本概念

Java 容器类库的用途是“保存对象”，将其划分为两个不同的概念：

1）Collection。一个独立元素的序列，这些元素服从一条或多条规则。List 必须按插入顺序保存元素。而 Set 不能有重复的元素，Queue 按照排队规则来确定对象产生的顺序。

2）Map 一组成对的“键值对”对象，允许你使用键来查找值。也被称为“关联数组”。

理想情况下，编写的大部分代码都是在与接口打交道，并且唯一需要指定所使用的精确类型的地方是在创建的时候。

使用接口的目的在于如果决定修改实现，所需要的只是在创建处修改它。如需使用类具有的额外功能，就不能将它们向上转型为更通用的接口。

11.3 添加一组元素

java.util 包中 Arrays 类和 Collections 类可以在一个 Collection 中添加一组元素。

Arrays.asList()方法接受一个数组或是一个用逗号分隔的元素列表（实用可变参数）并将其转换为一个 List 对象（底层表示仍为数组，因此不能调整尺寸。如果试图使用 add()或 delete()方法在列表中添加和删除元素可能会引发去改变数组尺寸的尝试，获得“Unsupported Operation(不支持的操作)”错误）。存在一个限制是它对所产生的 List 类型做出最理想（自动）的假设，而并没有注意你对它赋予什么样的类型。（可以在中间插入一条“线索”，以告诉编译器对于 Arrays.asList()产生的 List 类型，实际的目标类型应该是什么。称为显式参数类型说明。例子：Arrays.<Snow>asList()）

Collections.addAll()方法接受一个 Collection 对象，以及一个数组或是一个用逗号分割的列表，将元素添加到 Collection 中。

Arrays（数组工具类）主要方法有：

asList():返回一个受指定数组支持的固定大小的列表

binarySearch():使用二分搜索法来搜索指定的类型数组，以获得指定的值。重载支持指定搜索范围，通过重载支持 byte、char、double、float、int、long、short、Object、T。

copyOf():复制指定的数组，截取或用 0 填充（如有必要），以使副本具有指定的长度，重载支持 boolean、byte、char、double、float、int、long、short、T

copyOfRange():将指定数组的指定范围复制到一个新数组。重载支持 boolean、byte、char、double、float、int、long、short、T

deepEquals(): 如果两个指定数组彼此是深层相等的，则返回 true。

deepHashCode():基于指定数组的“深层内容”返回哈希码

deepToString():返回指定数组“深层内容”的字符串表示形式

equals():如果两个指定的数组彼此相等，则返回 true。重载支持 boolean、byte、char、double、float、int、long、short、Object、T

fill():将指定的值分配给指定的类型数组或指定范围中的每个元素,重载支持boolean、byte、char、double、float、int、long、short、Object、T

hashCode()：基于指定数组的内容返回哈希码，重载支持 boolean、byte、char、double、float、int、long、short、Object、T

sort():对指定的类型数组按数字升序进行排序。重载支持 boolean、byte、char、double、float、int、long、short、Object、T

sort()：对指定的类型数组的指定范围按数字升序进行排序，重载支持 boolean、byte、char、double、float、int、long、short、Object、T

toString():返回指定数组内容的字符串表示形式。

Collections（集合工具类）主要方法有：

11.4 容器的打印

默认打印行为（使用容器提供的 toString()方法）即可生成可读性很好的结果。

List 以特定顺序保存一组元素；Set 元素不能重复；Queue 只允许容器的一端插入元素，从另一端移除对象。

Map 使得可以用键来查找对象，就像一个简单的数据库。对于每个键，Map 只接受存储一次。(方法：Map.put(key,value)将增加一个值，并将它与某个键关联起来；Map.get(key)将产生与这个键相关联的值。)

11.5 List

List 可以将元素维护在特定的序列中。

有两种类型的 List:

◇ArrayList，长于随机访问元素，在 List 的中间插入删除元素较慢。

◇LinkedList，在 List 中间进行的插入和删除操作代价较低，提供了优化的顺序访问。随机访问比较慢，它的特性集较 ArrayList 更大。

List 类：

void add(int index,E element)：在列表的指定位置插入指定元素（可选操作）

boolean addAll(Collection<? Extends E> c)：添加指定 Collection 中的所有元素到此列表的尾部，顺序是指定 collection 的迭代器返回这些元素的顺序（可选操作）

boolean contains(Object o):如果列表包含指定的元素，则返回 true

boolean containsAll(Collection<?> c):如果列表包含指定 collection 的所有元素，则返回 true

boolean equals(Object o):比较指定的对象与列表是否相等

E get(int index)：返回列表中指定位置的元素

int hashCode():返回列表的哈希码

int indexOf(Object o):返回此列表中第一次出现的指定元素的索引，如果此列表不包含该元素，则返回-1

boolean isEmpty():如果列表不包含元素，则返回 true

Iterator<E> iterator():返回按适当顺序在列表的元素上进行迭代的迭代器

int lastIndexOf(Object o):返回此列表中最后出现的指定元素的索引，如列表不包含此元素，则返回-1

ListIterator<E> listIterator():返回此列表元素的列表迭代器（按适当顺序）

ListIterator<E> listIterator(int index):返回此列表中元素的迭代器(按适当顺序)，从列表指定位置开始

E remove(int index):移除列表中指定位置的元素（可选操作）

boolean remove(Object o)：从此列表中移除第一次出现指定元素（如果存在）(可选操作)

boolean removeAll(Collection<?> c):从列表中移除指定 collection 中包含的其所有元素（可选操作）

boolean retainAll(Collection<?> c):仅在列表中保留指定 collection 中所包含的元素（可选操作）

E set(int index,E element):用指定元素替换列表中指定位置的元素（可选操作）

int size():返回列表中的元素数

List<E> subList(int fromIndex,int toIndex):返回列表中指定的 fromIndex（包含）和 toIndex（不包含）之间的部分视图

Object[] toArray(): 返回适当顺序包含列表中所有元素的数组

<T> T[] toArray(T[] a): 返回按适当顺序包含列表中所有元素的数组，返回数组的运行时类型是指定数组的运行时类型。

11.6 迭代器

迭代器设计模式：迭代器是一个对象，它的工作是遍历并选择序列中对象，而客户端程序员不必知道或关心该序列低层的结构。通常被称为轻量级对象，创建它的代价小。Java 的 Iterator 只能单向移动， Iterator 接口包含的方法：

1)next():获得序列中的下一个元素； 2)hasNext()检查序列中是否还有元素；

3)remove():将迭代器新近返回的元素删除。

调用 remove()之前必须先调用 next()。

接受对象容器并传递它，从而在每个对象上都执行操作。迭代器统一了对容器的访问方式。

11.6.1 ListIterator

ListIterator 是一个更加强大的 Iterator 的子类型，只能用于各种 List 类的访问。

void add(E e):将指定的元素插入列表（可选操作）

boolean hasNext(): 以正向遍历列表时，如果列表迭代器有多个元素，则返回 true

boolean hasPrevious():以逆向遍历列表，列表迭代器有多个元素，则返回 true

E next():返回列表中的下一个元素

int nextIndex():返回对 next 的后续调用所返回元素的索引

E previous():返回列表中的前一个元素

int previousIndex(): 返回 previous 的后续调用返回元素的索引

void remove():从列表中移除由 next 或 previous 返回的最后一个元素（可选操作）

void set(): 用指定元素替换由 next 或 previous 返回的最后一个元素（可选操作）

11.7 LinkedList

LinkedList 实现了基本的 List 接口，它执行插入和移除比 ArrayList 效率高，但在随机访问操作方面却要逊色一些。

ListList 添加了可以使其用作栈、队列或双端队列的方法。

getFirst()、element()和 peek()都返回列表的头，而并不移除它； removeFirst()、remove()和 poll()都移除并返回列表的头，如果列表为空则抛出 NotSuchElementException，peek()、poll()则返回 null; offer()与 add()和 addLast()相同，都将某个元素插入到列表的尾部; removeLast()移除并返回列表的最后一个元素。

Queue 接口就是在 LinkedList 的基础上添加了 element()、offer()、peek()、poll()和 remove()方法，以使其称为一个 Queue 的实现。

11.8 Stack

“栈”通常是指“后进先出”(LIFO)的容器。最后“压入”栈的元素，第一个“弹出”栈。

使用组合而非继承可以避免 LinkedList 的其他所有方法的类。创建实例需要使用完整指定包名，避免与 java.util 包中的 Stack 发生冲突。

11.9 Set

Set 不保存重复的元素。常被用于测试归属性（基于对象的值来确定归属性），可以很容易地询问某个对象是否在某个 Set 中。因此查找是 Set 最重要的操作，通常选用 HashSet 的实现，专门对快速查找进行了优化。

HashSet 使用了散列函数，存储没有任何插入顺序规律可循；TreeSet 将元素的存储在红-黑树数据结构中，排列按照字典序进行插入顺序；LinkedList 查询速度因为使用了散列，速度快，使用链表来维护元素的插入顺序。

11.10 Map

将对象映射到其他对象的能力是一种解决编程问题的杀手锏。

Map 接口包含方法：

boolean containsKey(Object key):如果此映射包含指定键的映射关系，则返回 true

boolean containsValue(Object value):如果此映射将一个或多个映射到指定值，则返回 true

Set<Map.Entry<K,V>>entrySet():返回此映射包含的映射关系的 Set 视图

boolean equals(Object o):比较指定对象与此映射是否相等

V get(Object key):返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null

Set<K> keySet():返回此映射中所包含的键的 Set 视图

V put(K key,V value):将指定的值与此映射中的指定键关联（可选操作）

void putAll(Map<? extends K,? extends V> m):从指定映射中将所有映射关系复制到此映射中（可选操作）

V remove(Object key):如果存在一个键的映射关系，则将其从此映射中移除（可选操作）

int size():返回此映射中的键—值映射关系数

Collection<V> values():返回此映射中包含值的 Collection 视图

11.11 Queue

队列是一个典型的先进先出（FIFO）的容器。LinkedList 实现了 Queue 接口，因此 LinkedList 可以用作 Queue 的一种实现。通过将 LinkedList 向上转型为 Queue。

offer():将一个元素插入到队尾；

peek()和 element()都将在不移除的情况下返回队头；

poll()和 remove()将移除并返回队头。

11.11.1 PriorityQueue

优先级队列声明下一个弹出元素是最需要的元素(具有最高的优先级)。在 PriorityQueue 上调用 offer()方法来插入一个对象时，这个对象会在队列中被排序。默认使用对象在队列中的自然顺序，可以通过提供自己的 Comparator（构造函数提供）来修改这个顺序。Collection.reverseOrder()产生反序的 Comparator。

11.12 Collection 和 Iterator

Collection 是描述所有序列容器的共性的根接口，java.util.AbstractCollection 类提供了 Collection 的默认实现，使得可以创建 AbstractCollection 的子类型，而其中没有不必要的代码重复。

使用接口可以创建更通用的代码。通过针对接口而非具体实现编写代码，可以使得代码应用于更多的对象类型。

通过迭代器而非 Collection 表示容器之间的共性。要实现 Collection 就必须实现该接口中的所有方法。此时，继承并提供创建迭代器的能力就会显得容易很多。

生成 Iterator 是将队列与消费队列的方法连接在一起耦合度最小的方式，并且与实现 Collection 相比，它在类上所施加的约束也少很多。

11.13 Foreach 与迭代器

foreach 之所以能正常工作，是因为实现了 Iterable 的接口，该接口包含一个能够产生 Iterator 的 iterator()方法，并且 Iterable 接口被 foreach 用来在序列中移动。因此创建了任何实现 Iterable 的类，都可以将它用于 foreach 语句中。

foreach 语句可以用于数组或其他任何 Iterable。

11.13.1 适配器方法惯用法

适配器方法的惯用法。“适配器”部分来自设计模式，因为提供特定接口以满足 foreach 语句。当你有一个接口并需要另一个接口时，编写适配器（添加一个产生 Iterable 对象的方法，该对象可以用于 foreach 语句）就可以解决问题。

第 12 章　通过异常处理错误

12.1 概念

用强制规定的形式来消除错误处理过程中随心所欲的因素。异常只是在当前的环境中还没有足够的信息来解决这个问题，所以把这个问题提交到一个更高级别的环境中，在这里（异常处理程序）作出正确的决定。

12.2 基本异常

异常情形（exceptional condition）是指阻止当前方法或作用域继续执行的问题。

首先，同 Java 中其他对象的创建一样，将使用 new 在堆上创建异常对象；然后，当前的执行路径被终止，并且同当前环境中弹出对异常对象的引用。异常处理机制接管程序，并开始寻找一个恰当的地方继续执行程序。恰当的地方就是指异常处理程序。

异常使得我们将每件事都当作一个事务来考虑，而异常可以看护着这些事务的底线。

12.2.1 异常参数

标准异常类都有两个构造器：一个是默认构造器；另一个是接受字符串作为参数，以便把相关信息放入异常对象的构造器。

Throwable 对象，是异常类型的根类。

12.3 捕获异常

监控区域（guarded region）

12.3.1 try 块

在这个块里“尝试”各种方法调用，所以称为 try 块（监控区域）。

12.3.2 异常处理程序

抛出的异常必须在某处得到处理的“地点”就是异常处理程序。异常处理程序紧跟在 try 块之后，以关键字 catch 表示。

异常处理机制将负责搜寻参数与异常类型相匹配的第一个处理程序。进入 catch 子句执行，异常得到了处理。

终止与恢复

Java 支持终止模式。

另一种称为恢复模型。不实用，因为它所致的耦合：恢复性的处理程序需要了解异常抛出的地点，势必包含依赖于抛出位置的非通用性代码。

12.4 创建自己的异常

要定义自己的异常，必须从已有的异常类继承，最好是选择意思相近的异常类继承。对异常类来说，最重要的部分是类名。

printStackTrace()方法默认输出为 System.err，可以重定向输出 System.out。

12.4.1 异常与记录日志

java.util.logging 工具将输出记录到日志中。

12.5 异常说明

异常说明就是 Java 提供相应的语法（并强制使用这个语法），使得能够以礼貌的方式告知客户端程序员某个方法能会抛出的异常类型，然后客户端程序员就可以进行相应的处理。它属于方法声明的一部分，紧跟在形式参数列表之后。

使用附加关键字 throws，后面接着一个所有潜在异常类型的列表。RuntimeException 继承的异常可以在没有异常说明的情况下被抛出。

代码必须与异常说明保持一致。即如代码产生了异常却没有进行处理，编译器会提醒：要么处理这个异常，要么就在异常说明中表明此方法将产生异常。

这种在编译时被强制检查的异常称为被检查的异常。

12.6 捕获所有异常

通过捕获异常类型的基类 Exception，就可以捕获所有类型的异常。所以最好把它放在

处理程序列表的末尾，以防它抢在其他处理程序之前先把异常捕获了。

Exception 可以调用它从 Throwable 继承的方法：

String getMessage()

String getLocalizedMessage()

String toString()

void printStackTrace()

void printStackTrace(PrintStream)

void printStackTrace(java.io.PrintWriter)

打印 Throwable 和 Throwable 的调用栈轨迹。调用栈显示了“把你带到异常抛出地点”的方法调用序列。

Throwable fillInStackTrace()用于在 Throwable 对象的内部记录栈帧的当前状态。这个程序重新抛处错误或异常时很有用。

12.6.1 栈轨迹

printStackTrace()方法所提供的信息可以通过 getStackTrace()方法来直接访问，该方法返回一个由栈轨迹的元素所构成的数组，其中每个元素都表示栈中的一帧。

12.6.2 重新抛出异常

如果只是把当前异常对象重新抛出，那么 printStackTrace()方法显示的将是原来异常抛出点的调用栈信息，而并非重新抛出点的信息。要想更新这个信息，可以调用 fillInStaceTrace()方法，将返回一个 Throwable 对象，它通过把当前调用栈信息填入原来那个异常对象而建立的。

在捕获异常之后抛出另一种异常，得到的效果类似于使用 fillInStackTrace()，有关原来异常发生点的信息会丢失，剩下的是与新的抛出点有关的信息。

12.6.3 异常链

想要在捕获一个异常后，抛出另一个异常，并且希望把原始异常的信息保存下来，这被称为异常链。Throwable 的子类在构造器中可以接受一个 cause（因由）对象作为参数。这 cause 就用来表示原始异常，通过把原始异常传递给新的异常，使得即使在当前位置创建并抛出了新的异常，也能通过异常链追踪到异常最初发生的位置。

在 Throwable 的子类中，只有三种基本的异常类 Error、Exception、RuntimeException 提供了带 cause 参数的构造器。如果要把其他类型的异常链接起来，应该使用 initCause()方法而不是构造器。

12.7 Java 标准异常

Throwable 对象分为两种类型：Error 用来表示编译时和系统错误（除特殊情况外，一般不用关心）；Exception 是可以被抛出的基本类型。

12.7.1 特例：RuntimeException

运行时异常都是从 RuntimeException 类继承而来，不需要在异常说明中声明方法将抛出 RuntimeException 类型的异常，也被称为“不受检查异常”。此类异常会穿越所有的执行路径直达 main()方法，而不会被捕获。

记住：只能在代码中忽略 RuntimeException（及其子类）类型的异常，其他类型异常的处理都是由编译器强制实施的。究其原因 RuntimeException 代表的是编程错误：

1）无法预料的错误。

2）作为程序员，应该在代码中进行检查的错误。

12.8 使用 finally 进行清理

无论是否抛出异常，finally 子句总能被执行。

12.8.1 finally 用来做什么

当要把内存之外的资源恢复到它们的初始状态时，就要用到 finally 子句。

12.8.2 在 return 中使用 finally

在 finally 类内部，从何处返回无关紧要。

12.8.3 缺憾：异常丢失

异常在 finally 子句里被其他异常取代。

12.9 异常的限制

当覆盖方法的时候，只能抛出在基类方法的异常说明里列出的那些异常。意味着当基类使用的代码应用到其派生类对象的时候，一样能够工作。

异常限制对构造器不起作用。派生类构造器的异常说明必须包含基类构造器的异常说明。派生类构造器不能捕获基类构造器抛出的异常。

通过强制派生类遵守基类方法的异常说明，对象的可替换性得到了保证。

12.10 构造器

Java 的缺陷：除了内存的清理之外，所有的清理都不会自动发生。通用的清理惯用法在构造器不抛出异常时也应该运用，其基本规则是：在创建需要清理的对象之后，立即进入一个 try-finally 语句块。

12.11 异常匹配

异常处理系统会按照代码的书写顺序找出“最近”的处理程序，找到匹配的处理程序后，他就认为异常得到了处理，然后就不再继续查找。

如因基类异常位于前面，派生类的异常的 catch 子句永远得不到执行，那么他会报告错误。

12.12 其他可选方式

异常处理的一个重要原则是“只有在你知道如何处理的情况下才捕获异常”。异常处理的一个重要目标就是把错误处理的代码同错误发生地点相分离。

“被检查的异常”强制在可能没准备好处理错误的时候被迫加上 catch 子句，这就导致了吞食则有害（harmful if swallowed）的问题。

12.12.1 历史

12.12.2 观点

12.12.3 把异常传递给控制台

main()作为方法也可以由异常说明，这里异常的类型是 Exception，它也是所有“被检查的异常”的基类。通过把它传递到控制台，就不必在 main()里写 try-catch 子句了。

12.12.4 把“被检查的异常”转换为“不检查的异常”

直接把“被检查的异常”包装进 RuntimeException 里面，异常链保证不会丢失任何原始异常的信息。

12.13 异常使用指南

应当在下列情况下使用异常：

1）恰当的级别处理问题

2）解决问题并且重新调用产生异常的方法

3）进行少许修补，然后绕过异常产生的地方继续执行

4）用别的数据进行计算，以代替方法预计会返回的值

5）把当前运行环境下能做的事情尽量完成，然后把相同的异常重新抛出到更高层

6）把当前运行环境下能做的事情尽量完成，然后把不同的异常抛到更高层

7）终止程序

8）进行简化

9）让类库和程序更安全

第13章 字 符 串

13.1 不可变String

String对象具有只读特性，所以指向它的任何引用都不可能改变它的值。

13.2 重载“+”与StringBuilder

javap反编译命令

编译器自动引入了java.lang.StringBuilder类。

已经知道最终的字符串大概多长，可以预先指定StringBuilder的大小可以避免多次重新分配缓冲。

13.3 无意识的递归

希望toString()打印出对象的内存地址，使用this关键字，导致递归调用。解决方法调用Object.toString()。

13.4 String上的操作

1、构造器：重载版本:默认版本String、StringBuilder、StringBuffer、char数组、byte数组；

2、length():String中字符的个数；

3、charAt(int index):取得String中索引位置上的char；

4、getChars()、getBytes():复制char或byte到一个目标数组中；

5、toCharArray():生成一个char[]，包含String的所有字符；

6、equals()、equalsIgnoreCase():比较两个String的内容是否相同；

7、compareTo():按照字典顺序比较String的内容，比较结果为负数、零或正数；

8、contains():如果该String对象包含参数的内容，则返回true

9、contentEquals():如果该String与参数CharSequence的内容完全一致,则返回true；

10、regionMatcher():测试两个字符串区域是否相等，重载增加了“忽略大小写”功能；

11、startsWith():测试String是否以参数为起始；

12、endsWith():测试String是否以参数为后缀；

13、indexOf()、lastIndexOf():如果该String并不包含此参数，就返回-1；否则返回此参数在String中的起始索引。lastIndexOf()从后向前搜索。

14、substring()、subsequence():返回一个新的String，以包含参数指定的子字符串；

15、concat():返回一个新的String对象，内容为原始String连接上参数String；

16、replace():返回替换字符后的新String对象。如果没有替换发生，则返回原始的String对象；

17、toLowerCase()、toUpperCase():将字符的大小写改变后，返回一个新String对象。如果改变没有发生，则返回原始的String对象；

18、trim():将String两端的空白字符删除后，返回一个新的String对象。如果改变没有发生，则返回原始的String对象；

19、valueOf():返回一个表示参数内容的String；

20、intern():为每个唯一的字符序列生成一个且仅生成一个String引用。

13.5 格式化输出

13.5.1 printf()

格式修饰符

13.5.2 System.out.format()

format方法可用于PrintStram或PrintWriter对象，其中包括System.out对象。与printf()是等价的。

13.5.3 Formatter类

Java中，所有新的格式化功能都由java.util.Formatter类处理。创建一个Formatter

对象的时候，需要向其构造器传递一些信息（OutPutStream、File、PrintStream、String fileName），告诉它最终的结果将向哪里输出。

close():关闭此 Formatter

flush():刷新此 Formatter

format(String format,Object…args): 使用指定格式字符串和参数将一个格式化字符串写入此对象的目标文件中。

toString():返回对输出的目标文件调用 toString() 的结果。

13.5.4 格式化说明符

格式化修饰符语法：

%[argument_index$][flags][width][.precision]conversion

argument_index:十进制整数，用于表示参数在参数列表中的位置；

flags:是修改输出格式的字符集，有效标志集取决于转换类型；

width: 是非负十进制整数，表明要向输出中写入的最少字符数；

precision:是一个非负十进制整数，通常用于限制字符数（只用于 String、浮点数）；

conversion:是一个表明应该如何格式参数的字符。

Flags: y 表示该标志受指示参数类型支持。

标志	常规	字符	整数	浮点	日期时间	说明
'-'	y	y	y	y	y	结果将是左对齐的。
'#'	y[1]	-	y[3]	y	-	结果应该使用依赖于转换类型的替换形式
'+'	-	-	y[4]	y	-	结果总是包括一个符号
' '	-	-	y[4]	y	-	对于正值，结果中将包括一个前导空格
'0'	-	-	y	y	-	结果将用零来填充
','	-	-	y[2]	y[5]	-	结果将包括特定于语言环境的组分隔符
'('	-	-	y[4]	y[5]	-	结果将是用圆括号括起来的负数

1 取决于 Formattable 的定义。
2 只适用于 'd' 转换。
3 只适用于 'o'、'x' 和 'X' 转换。
4 对 BigInteger 应用 'd'、'o'、'x' 和 'X' 转换时，或者对 byte 及 Byte、short 及 Short、int 及 Integer、long 及 Long 分别应用 'd' 转换时适用。
5 只适用于 'e'、'E'、'f'、'g' 和 'G' 转换。

13.5.5 Formatter 转换

类型转换字符			
d	整数型（十进制）	e	浮点数（科学计数）
c	Unicode 字符	x	整数（十六进制）
b	Boolean 值	h	散列码（十六进制）
s	String	%	字符“%”
f	浮点数（十进制）		

13.5.6 String.format()

String.format()是一个 static 方法，接受与 Formatter.format()方法一样的参数，返回一个 String 对象。

13.6 正则表达式

正则表达式优点

13.6.1 基础

正则表达式就是以某种方式来描述字符串，因此，可以说：“如果一个字符串含有这些东西，那么它就是我正在找的东西。”

Java 中，\\的意思是“我要插入一个正则表达式的反斜线，所以其后的字符具有特殊的意义。”

应用正则表达式的最简单的途径，利用 String 内建的功能。

boolean matches(String regex):对字符串匹配正则表达式;

String[] split(String regex)或 String[] split(String regex,int limit):将字符串从正则表达式匹配的地方切开，重载限制字符串分割的次数。

String replaceAll(String regex,String replacement): 使用给定的 replacement 替换此字符串所有匹配给定的正则表达式的子字符串。

String replaceFirst(String regex, String replacement): 使用给定的 replacement 替换此字符串匹配给定的正则表达式的第一个子字符串。

13.6.2 创建正则表达式

字符			
x	字符 x	\xhh	带有十六进制值 0x 的字符 hh
\t	制表符 ('\u0009')	\uhhhh	带有十六进制值 0x 的字符 hhhh
\r	回车符 ('\u000D')	\n	新行（换行）符 ('\u000A')
\f	换页符 ('\u000C')	\e	转义符 ('\u001B')

字符类	
[abc]	a、b 或 c（简单类）
[^abc]	任何字符，除了 a、b 或 c（否定）
[a-zA-Z]	a 到 z 或 A 到 Z，两头的字母包括在内（范围）
[a-d[m-p]]	a 到 d 或 m 到 p：[a-dm-p]（并集）
[a-z&&[def]]	d、e 或 f（交集）
[a-z&&[^bc]]	a 到 z，除了 b 和 c：[ad-z]（减去）
[a-z&&[^m-p]]	a 到 z，而非 m 到 p：[a-lq-z]（减去）

预定义字符类			
.	任何字符	\d	数字:[0-9]
\D	非数字:[^0-9]	\s	空白字符(空格、tab、换行、回车)
\S	非空白字符:[^\s]	\w	单词字符：[a-zA-Z0-9]
\W	非单词字符：[^\w]		

边界匹配器			
^	行的开头	$	行的结尾
\b	单词边界	\B	非单词边界
\A	输入的开头	\G	上一个匹配的结尾
\Z	输入的结尾	\z	输入的结尾

Logical 运算符	
XY	*X* 后跟 *Y*
X|Y	*X* 或 *Y*
(X)	X，作为捕获组

13.6.3 量词

贪婪型	勉强型	占有型	如何匹配
X?	X??	X?+	一个或零个 X
X*	X*?	X*+	零个或多个 X
X+	X+?	X++	一个或多个 X
X{n}	X{n}?	X{n}+	恰好 n 次 X
X{n,}	X{n,}?	X{n,}+	至少 n 次 X
X{n,m}	X{n,m}?	X{n,m}+	X 至少 n 次，且不超过 m 次

接口 CharSequence 从 CharBuffer、String、StringBuffer、StringBuilder 类中抽象出了字符序列的一般化定义：

```
interface CharSequence {
   charAt(int i);
   length();
   subsequence(int start,int end);
   toString();
}
```

13.6.4 Pattern 和 Matcher

java.util.regex 包 Pattern 类和 Matcher 类

Pattern 类：

static Pattern compile(String regex)将给定的正则表达式编译到模式中；

static Pattern compile(String regex,int flags)将给定的正则表达式编译到具有给定标志的模式中；

Matcher matcher(CharSequence input)创建匹配给定输入与此模式的匹配器；

int flags()返回此模式的匹配标志；

static boolean matches(String regex, CharSequence input)编译给定正则表达式并尝试将给定输入与其匹配；

String pattern()返回在其中编译过此模式的正则表达式；

String[] split(CharSequence input)围绕此模式的匹配拆分给定输入序列；

String[] split(CharSequence input,int limit)围绕此模式的匹配拆分给定输入序列。

Matcher 类：

int **start**()返回以前匹配的初始索引；

int **start**(int group)返回在以前的匹配操作期间，由给定组所捕获的子序列的初始索引；

int **end**()返回最后匹配字符之后的偏移量；

int **end**(int group)返回在以前的匹配操作期间，由给定组所捕获子序列的最后字符之后的偏移量；

boolean **find**()尝试查找与该模式匹配的输入序列的下一个子序列；

boolean **find**(int start)重置此匹配器，然后尝试查找匹配该模式、从指定索引开始的输入序列的下一个子序列；

String **group**()返回由以前匹配操作所匹配的输入子序列；

String **group**(int group)返回在以前匹配操作期间由给定组捕获的输入子序列。

int **groupCount**()返回此匹配器模式中的捕获组数；

boolean **lookingAt**()尝试将从区域开头开始的输入序列与该模式匹配；

boolean **matches**()尝试将整个区域与模式匹配；

Pattern pattern()返回由此匹配器解释的模式；

String **replaceAll**(String replacement)替换模式与给定替换字符串相匹配的输入序列的每个子序列；

String **replaceFirst**(String replacement)替换模式与给定替换字符串匹配的输入序列的第一个子序列；

Matcher **appendReplacement**(StringBuffer sb,String replacement)实现非终端添加和替换步骤；

StringBuffer **appendTail**(StringBuffer sb)实现终端添加和替换步骤；

Matcher **reset**()重置匹配器；

Matcher **reset**(CharSequence input)重置此具有新输入序列的匹配器；

String toString()返回匹配器的字符串表示形式。

Pattern 标记：

编译标记	效　果
Pattern.CANON_EQ	两个字符当且仅当它们的完全规范分解相匹配时，就认为它们是匹配的。默认情况不考虑规范的等价性。
☆ Pattern.CASE_INSENSITIVE(?i)	标记允许模式匹配不必考虑大小写，基于 Unicode 的大小写不敏感的匹配开启。
☆Pattern.COMMENTS(?x)	此模式空格符被忽略，并且以#开始直到行末的注释也会被忽略掉。
Pattern.DOTALL(?s)	在 dotall 模式中，表达式“.”匹配所有字符，包括行终结符。默认情况不匹配行结束符。
☆ Pattern.MULTILINE(?m)	在多行模式下，表达式^和$分别匹配一行的开始和结束。^还匹配输入字符串的开始，而$还匹配输入字符串的结尾。默认情况下，这些表达式仅匹配输入的完整字符串的开始和结束。
Pattern.UNICODE_CASE(?u)	当制定这个标记，并且开启 CASE_INSENSITIVE 时，大小写不敏感的匹配将按照与 Unicode 标准相一致的方式进行。
Pattern.UNIX_LINES(?d)	此模式下，在.^和 $行为中，只识别行终结符\n。

通过或(|)操作符组合多个标记

Matcher m = Pattern.compile(“(?m)(\\S+)\\s+((\\S+)\\s+(\\S+))$”).matcher(POEM);

13.6.5 split()

由 Pattern 对象确定。

13.6.6 替换操作

replaceFirst(String replacement)、replaceAll(String replacement)

appendReplacement(StringBuffer sbuf, String replacement)执行渐进式的替换，它允许调用其他方法来生成或处理 repalcement(replaceFirst 和 replaceAll 则只能使用一个固定的字符串)，使你能够以编程的方式将目标分隔成组，从而具备强大的替换功能。

appendTail(StringBuffer sbuf)在执行了一次或多次 appendReplace()之后，调用此方法可以将输入字符串余下的部分复制到 sbuf 中。

13.6.7 reset()

将现有的 Matcher 对象应用于新的字符序列。不带参数，将 Matcher 对象重新设置到当前字符序列的起始位置。

13.6.8 正则表达式与 Java I/O

13.7 扫描输入

Java SE5 新增了 Scanner 类，减轻了扫描输入的工作负担。

Scanner 的构造器接受 File 对象、InputStream、String 或者 Readable 输入对象。Readable 接口，表示“具有 Read()方法的某种东西”。

boolean **hasNext**()如果此扫描器的输入中有另一个标记，则返回 true;

boolean **hasNext**(**Pattern pattern**)如果下一个完整标记与指定模式匹配,则返回 true;

boolean **hasNext**(String pattern)如果下一个标记与从指定字符串构造的模式匹配，则返回 true;

hasNextBigDecimal()、hasNextBigInteger()、hasNextBigInteger(int radix)、hasNextBoolean()、hasNextByte()、hasNextByte(int radix)、hasNextDouble()、hasNextFloat()、hasNextInt()、hasNextInt(int radix)、hasNextLine()、hasNextLong()、hasNextLong(int radix)、hasNextShort()、hasNextShort(int radix);

IOException ioException()返回此 Scanner 的底层 Readable 最后抛出的 IOException;

MatchResult match()返回此扫描器所执行的最后扫描操作的匹配结果;

String **next**()查找并返回来自此扫描器的下一个完整标记;

String **next**(Pattern pattern)如果下一个标记与指定模式匹配，则返回下一个标记;

String **next**(String pattern)如果下一个标记与从指定字符串构造的模式匹配，则返回下一个标记;

nextBigDecimal()、nextBigInteger()、nextBigInteger(int radix)、nextBoolean()、nextByte()、nextByte(int radix)、nextDouble()、nextFloat()、nextInt()、nextInt(int radix)、nextLine()、nextLong()、nextLong(int radix)、nextShort()、nextShort(int radix);

Scanner reset()重置此扫描器;

Scanner **skip**(Pattern pattern)在忽略分隔符的情况下跳过与指定模式匹配的输入信息;

Scanner **skip**(String pattern)跳过与从指定字符串构造的模式匹配的输入信息;

Scanner **useDelimiter**(Pattern pattern)将此扫描器的分隔模式设置为指定模式;

Scanner **useDelimiter**(String pattern)将此扫描器的分隔模式设置为从指定 String 构造的模式;

Scanner useRadix(int radix)将此扫描器的默认基数设置为指定基数。

接口 MatchResult:

int end()返回最后匹配字符之后的偏移量;

int end(int group)返回在匹配期间由给定组所捕获子序列的最后字符之后的偏移量;

String group()返回由以前匹配所匹配的输入子序列;

String group(int group)返回在以前匹配操作期间由给定组捕获的输入子序列。

int groupCount()返回此匹配结果的模式中的捕获组数。

int start()返回匹配的初始索引。

int start(int group)返回在匹配期间由给定组捕获的子序列的初始索引。

第 14 章　类型信息

运行时类型信息使得可以在程序运行时发现和使用类型信息。

分为 RTTI 和“反射”机制

14.1 为什么需要 RTTI

基本目的：让代码只操纵对基类的引用。

RTTI 名字的含义：在运行时，识别一个对象的类型。

List<Shape>实际将所有事物都当坐 Object 持有——会自动将结果转型为 Shape。在编译时，将由容器和 Java 泛型来强制确保这一点，而在运行时，由类型转换操作来确保这一点。

多态机制，基类引用实际执行什么样的代码，是由引用所指向的具体对象而决定的。

14.2 Class 对象

类型信息在运行是由称为 Class 对象完成的，包含了与类有关的信息。事实上，Class 对象就是用来创建类的所有的“常规”对象的。Java 使用 Class 对象来执行其 RTTI，即使正在执行的是类型转换这样的操作。

类是程序的一部分。每个类都有一个 Class 对象。编写并编译一个新类，就会产生一个 Class 对象（更恰当地说，是被保存在一个一个同名的.class 文件中）。为了生成这个类的对象，运行这个程序的 Java 虚拟机(JVM)将使用被称为“类加载器”的子系统。

原生类加载器、可信类

所有的类都是对其第一次使用时，动态加载到 JVM 中的。因此 Java 程序在它开始运行之前并非被完全加载，各个部分在必需时才加载的。

类加载器首先检查类的 Class 对象是否已经加载。如尚未加载，默认的类加载器就会根据类名查找.class 文件。在类的字节码加载时，会接受验证，确保未被破坏，或包含不良代码。

一旦某个类的 Class 对象被载入内存，就被用来创建这个类的所有对象。

forName(String className)返回 Class 对象的引用，如果类没有被加载就加载它。必须使用全限定名（包含包名）。

Class 类：

<U> Class<? extends U> asSubclass(Class<U> clazz)强制转换该 Class 对象，以表示指定的 Class 对象所表示的类的一个子类。

T **cast**(Object obj)将一个对象强制转换成此 Class 对象所表示的类或接口。

static Class<?> forName(String className)返回与带有给定字符串名的类或接口相关联的 Class 对象。

static Class<?> forName(String name, boolean initialize, ClassLoader loader)使用给定的类加载器，返回与带有给定字符串名的类或接口相关联的 Class 对象。

Annotation[] getAnnotations()返回此元素上存在的所有注释。

ClassLoader getClassLoader()返回该类的类加载器。

Class<?> getComponentType()返回表示数组组件类型的 Class。

Constructor<?>[] **getConstructors**()返回一个包含某些 Constructor 对象的数组，这些对象反映此 Class 对象所表示的类的所有**公共**构造方法。

Constructor<?>[] **getDeclaredConstructors**()返回 Constructor 对象的一个数组，这些对象反映此 Class 对象表示的类声明的所有构造方法。

Constructor<T> **getDeclaredConstructor**(Class<?>… parameterTypes)返回一个 Constructor 对象，该对象反映此 Class 对象所表示的类或接口的指定构造方法。

Class<?>[] getClasses()返回一个包含某些 Class 对象的数组，这些对象表示属于此 Class 对象所表示的类的成员的所有公共类和接口。

Class<?>[] getDeclaredClasses()返回 Class 对象的一个数组，这些对象反映声明为此 Class 对象所表示的类的成员的所有类和接口。

Field **getDeclaredField**(String name)返回一个 Field 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明字段。

Field **getField**(String name)返回一个 Field 对象，它反映此 Class 对象所表示的类或接口的指定公共成员字段。

Field[] **getFields**()返回一个包含某些 Field 对象的数组，这些对象反映此 Class 对象所表示的类或接口的所有可访问公共字段。

Field[] **getDeclaredFields**()返回 Field 对象的一个数组，这些对象反映此 Class 对象所表示的类或接口所声明的所有字段。

Method getEnclosingMethod()如果此 Class 对象表示某一方法中的一个本地或匿名类，则返回 Method 对象，它表示底层类的立即封闭方法。

Method **getDeclaredMethod**(String name, Class<?>... parameterTypes)返回一个 Method 对象，该对象反映此 Class 对象所表示的类或接口的指定已声明方法。

Method getMethod(String name, Class<?>... parameterTypes)返回一个 Method 对象，它反映此 Class 对象所表示的类或接口的指定公共成员方法。

Method[] getDeclaredMethods()返回 Method 对象的一个数组，这些对象反映此 Class 对象表示的类或接口声明的所有方法，包括公共、保护、默认（包）访问和私有方法，但不包括继承的方法。

Method[] getMethods()返回一个包含某些 Method 对象的数组，这些对象反映此 Class 对象所表示的类或接口（包括那些由该类或接口声明的以及从超类和超接口继承的那些的类或接口）的公共 *member* 方法。

T[] getEnumConstants()如果此 Class 对象不表示枚举类型，则返回枚举类的元素或 null。

Type[] getGenericInterfaces()返回表示某些接口的 Type，这些接口由此对象所表示的类或接口直接实现。

Type getGenericSuperclass()返回表示此 Class 所表示的实体（类、接口、基本类型或 void）的**直接超类**的 Type。

Class<? super T> getSuperclass()返回表示此 Class 所表示的实体（类、接口、基本类型或 void）的超类的 Class。

Class<?>[] getInterfaces()确定此对象所表示的类或接口实现的接口。

int **getModifiers**()返回此类或接口以整数编码的 Java 语言修饰符。

String getName()以 String 的形式返回此 Class 对象所表示的实体（类、接口、数组类、基本类型或 void）名称。

String **getSimpleName**()返回源代码中给出的底层类的简称。

Package getPackage()获取此类的包。

URL getResource(String name)查找带有给定名称的资源。

T **newInstance**() 创建此 Class 对象所表示的类的一个新实例（必须有默认构造器）。

boolean **isAssignableFrom**(Class<?> cls)判定此**Class 对象**所表示的类或接口与指定的 Class 参数所表示的类或接口是否相同，或是否是其超类或超接口。

boolean **isInstance**(Object obj)判定指定的**Object** 是否与此 Class 所表示的对象赋值兼容。

isArray()、isEnum()、isInterface()、isLocalClass()、isAnnotation()、isLocalClass()、isMemberClass()、isPrimitive()、isSynthetic()

14.2.1 类字面常量

类字面常量是生成对 Class 对象的引用方法。表示为 ClassName.class。

使用“.class”来创建对 Class 对象的引用时，不会自动地初始化该 Class 对象。

使用类做的准备工作：

1、加载。由类加载器执行。查找字节码（通常在 classpath 所指定的路径中查找，但并非必需），并从这些字节码中创建一个 Class 对象；

2、链接。验证类中的字节码，为静态域分配存储空间，并且如果必需的话，将解析这个类创建的对其他类的引用；

3、初始化。如果该类有超类，则对其初始化，执行静态初始化器和静态初始化块。

一个 static final 值是“编译器常量”，不需要对类进行初始化就可以被读取。但不是每个 static final 域能确保这种行为。

14.2.2 泛化的 Class 引用

Class 引用表示的是它所指向的对象的确切类型，而该对象便是 Class 类的一个对象。

泛型语法通过允许你对 Class 引用所指向的 Class 对象的类型进行限定而实现。通过泛

型语法，可以让编译器强制执行额外的类型检查。

为了在使用泛化的 Class 引用时放松限制，可以使用通配符。通配符就是“?”表示“任何事物”。

通配符结合 extends 和 super，创建一个范围。newInstance()返回值不是精确类型，而只是 Object。

泛型语法用于 Class 对象，newInstance()将返回该对象的确切类型，而不是基本的 Object。

14.2.3 新的转型语法

Cast 引用的转型语法，即 Cast()方法。

14.3 类型转换前先做检查

RTTI 形式包括:

1）传统的类型转换，由 RTTI 确保类型转换的正确性;

2）代表对象类型的 Class 对象。通过查询 Class 对象可以获取运行时所需的信息。

3）第三种形式就是关键字 instanceof，返回一个布尔值，告诉我们对象是不是某个特定类型的实例。

Java 执行类型检查，如果不使用显式的类型转换，编译器不允许执行向下转型赋值，以告知编译器你拥有额外的信息，这些信息使用你知道该类型是特定类型。编译器将检查向下转型是否合理，因此它不允许向下转型到实际上不是待转型类的子类的类型上。

instance 的限制：只可将其与命名类型进行比较，而不能与 Class 对象作比较。

14.3.1 使用类字面常量

14.3.2 动态的 instanceof

isInstance()方法可以使我们不再需要 instanceof 表达式。

14.3.3 递归计数

14.4 注册工厂

这里我们做的其他修改就是使用工厂方法设计模式，将对象的创建工作交给类自己去完成。

```
//:typeinfo/factory/Factory.java
package typeinfo.factory;
public interface Factory<T> { T create(); }

//:typeinfo/RegisteredFactories.java
//Registering CLass Factories in the base class.
import typeinfo.factory.*;
import java.util.*;

class Part {
    public String toString() {
        return getClass().getSimpleName();
    }
    static List<Factory<? extends Part>> partFactories =
        new ArrayList<Factory<? extends Part>>();
    static {
        //Collextions.addAll() gives an "unchecked generic
        //array creation ... for varargs parameter" warning.
        partFactories.add(new FuelFilter.Factory());
        partFactories.add(new AirFilter.Factory());
        partFactories.add(new CabinAirFilter.Factory());
        partFactories.add(new OilFilter.Factory());
        partFactories.add(new FanBelt.Factory());
```

```
        partFactories.add(new PowerSteeringBelt.Factory());
        partFactories.add(new GeneratorBelt.Factory());
    }
    private static Random rand = new Random(47);
    public static Part createRandom() {
        int n = rand.nextInt(partFactories.size());
        return partFactories.get(n).create();
    }
}

class Filter extends Part { }

class FuelFilter extends Filter {
    //Create a Class Factory for each specific type:
    public static class Factory implements
        typeinfo.factory.Factory<FuelFilter> {
        public FuelFilter create() { return new FuelFilter(); }
    }
}

class AirFilter extends Filter {
    //Create a Class Factory for each specific type:
    public static class Factory implements
        typeinfo.factory.Factory<AirFilter> {
        public AirFilter create() { return new AirFilter(); }
    }
}

class CabinAirFilter extends Filter {
    //Create a Class Factory for each specific type:
    public static class Factory implements
        typeinfo.factory.Factory<CabinAirFilter> {
        public CabinAirFilter create() { return new CabinAirFilter(); }
    }
}

class OilFilter extends Filter {
    //Create a Class Factory for each specific type:
    public static class Factory implements
        typeinfo.factory.Factory<OilFilter> {
        public OilFilter create() { return new OilFilter(); }
    }
}

class Belt extends Part {}

class FanBelt extends Belt {
    //Create a Class Factory for each specific type:
    public static class Factory implements
        typeinfo.factory.Factory<FanBelt> {
        public FanBelt create() { return new FanBelt(); }
    }
}
```

```
class PowerSteeringBelt extends Belt {
    //Create a Class Factory for each specific type:
    public static class Factory implements
        typeinfo.factory.Factory<PowerSteeringBelt> {
        public PowerSteeringBelt create() { return new PowerSteeringBelt(); }
    }
}

class GeneratorBelt extends Belt {
    //Create a Class Factory for each specific type:
    public static class Factory implements
        typeinfo.factory.Factory<GeneratorBelt> {
        public GeneratorBelt create() { return new GeneratorBelt(); }
    }
}

public class RegisteredFactories {
    public static void main(String[] args) {
        for(int i = 0;i < 10;i ++)
            System.out.println(Part.createRandom());
    }
}
```

14.5 instanceof 与 Class 的等价性

instanceof、isInstance()保持了类型的概念，它指的是否是这个类或这个类的派生类；==比较实际的 Class 对象，就没有考虑继承，它或者是确切的类型或者不是。

14.6 反射：运行时的类信息

Class 类与 java.lang.reflect 类库一起对反射的概念进行了支持,该类库包含了 Field、Method 以及 Constructor 类。这些类型的对象是由 JVM 在运行时创建的，用以表示未知类里对应的成员。这样，匿名对象的类信息就能在运行时被完全确定下来，而在编译时不需要知道任何事情。

RTTI 和反射之间真正的区别在于，对 RTTI 来说，编译器在编译时打开和检查.class 文件；而对于反射机制来说，.class 文件在编译时是不可获取的，所以是运行时打开和检查.class 文件。

Constructor<T>类：

boolean equals(Object obj)将此 Constructor 对象与指定的对象进行比较。

<T extends Annotation> T getAnnotation(Class<T> annotationClass)如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null。

Annotation[] getDeclaredAnnotations()返回直接存在于此元素上的所有注释。

Class<T> **getDeclaringClass**()返回 Class 对象，该对象表示声明由此 Constructor 对象表示的构造方法的类。

Class<?>[] **getExceptionTypes**()返回一组表示声明要抛出的异常类型的 Class 对象，这些异常是由此 Constructor 对象表示的底层构造方法抛出的。

Type[] getGenericExceptionTypes()返回一组 Type 对象，这些对象表示声明要由此 Constructor 对象抛出的异常。

Type[] getGenericParameterTypes()按照声明顺序返回一组 Type 对象，这些对象表示此 Constructor 对象所表示的方法的形参类型。

int **getModifiers**()以整数形式返回此 Constructor 对象所表示构造方法的 Java 语言修饰符。

String **getName**()以字符串形式返回此构造方法的名称。

Annotation[][] getParameterAnnotations()按照声明顺序返回一组数组，这些数组表示通过此 Constructor 对象表示的方法的形参上的注释。

Class<?>[] **getParameterTypes()**按照声明顺序返回一组 Class 对象，这些对象表示此 Constructor 对象所表示构造方法的形参类型。

TypeVariable<Constructor<T>>[] getTypeParameters()按照声明顺序返回一组 TypeVariable 对象，这些对象表示通过此 GenericDeclaration 对象所表示的一般声明来声明的类型变量。

int hashCode()返回此 Constructor 的哈希码。

T **newInstance**(Object... initargs)使用此 Constructor 对象表示的构造方法来创建该构造方法的声明类的新实例，并用指定的初始化参数初始化该实例。

String toGenericString()返回描述此 Constructor 的字符串，其中包括类型参数。

String toString()返回描述此 Constructor 的字符串。

Method 类：

boolean equals(Object obj)将此 Method 与指定对象进行比较。

<T extends Annotation> T getAnnotation(Class<T> annotationClass)如果存在该元素的指定类型的注释，则返回这些注释，否则返回 null。

Annotation[]getDeclaredAnnotations()返回直接存在于此元素上的所有注释。

Class<?> getDeclaringClass()返回表示声明由此 Method 对象表示的方法的类或接口的 Class 对象。

Object getDefaultValue()返回由此 Method 实例表示的注释成员的默认值。

Class<?>[] getExceptionTypes()返回 Class 对象的数组，这些对象描述了声明将此 Method 对象表示的底层方法抛出的异常类型。

Type[] getGenericExceptionTypes()返回 Type 对象数组，这些对象描述了声明由此 Method 对象抛出的异常。

Type[] getGenericParameterTypes()按照声明顺序返回 Type 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型的。

Type getGenericReturnType()返回表示由此 Method 对象所表示方法的正式返回类型的 Type 对象。

int **getModifiers**()以整数形式返回此 Method 对象所表示方法的 Java 语言修饰符。

String getName()以 String 形式返回此 Method 对象表示的方法名称。

Annotation[][] getParameterAnnotations()返回表示按照声明顺序对此 Method 对象所表示方法的形参进行注释的那个数组的数组。

Class<?>[] getParameterTypes()按照声明顺序返回 Class 对象的数组，这些对象描述了此 Method 对象所表示的方法的形参类型。

Class<?> getReturnType()返回一个 Class 对象，该对象描述了此 Method 对象所表示的方法的正式返回类型。

TypeVariable<Method>[] getTypeParameters()返回 TypeVariable 对象的数组，这些对象描述了由 GenericDeclaration 对象表示的一般声明按声明顺序来声明的类型变量。

int hashCode()返回此 Method 的哈希码。

Object **invoke**(Object obj, Object... args)对带有指定参数的指定对象调用由此 Method 对象表示的底层方法。

boolean isBridge()如果此方法是 bridge 方法，则返回 true；否则，返回 false。

boolean isSynthetic()如果此方法为复合方法，则返回 true；否则，返回 false。

boolean isVarArgs()如果将此方法声明为带有可变数量的参数，则返回 true；否则，返回 false。

String toGenericString()返回描述此 Method 的字符串，包括类型参数。

String **toString**()返回描述此 Method 的字符串。

Field 类：

boolean equals(Object obj)将此 Field 与指定对象比较。

Object get(Object obj)返回指定对象上此 Field 表示的字段的值。

boolean getBoolean(Object obj)获取一个静态或实例 boolean 字段的值。

byte getByte(Object obj)获取一个静态或实例 byte 字段的值。

char getChar(Object obj)获取 char 类型或另一个通过扩展转换可以转换为 char 类型的基本类型的静态或实例字段的值。

double getDouble(Object obj)、float getFloat(Object obj)、 int getInt(Object obj)、long getLong(Object obj)、short getShort(Object obj)

int getModifiers()以整数形式返回由此 Field 对象表示的字段的 Java 语言修饰符。

Type getGenericType()返回一个 Type 对象，它表示此 Field 对象所表示字段的声明类型。

Class<?> getType()返回一个 Class 对象，它标识了此 Field 对象所表示字段的声明类型。

Class<?> getDeclaringClass()返回表示类或接口的 Class 对象，该类或接口声明由此 Field 对象表示的字段。

String getName()返回此 Field 对象表示的字段的名称。

void set(Object obj, Object value)将指定对象变量上此 Field 对象表示的字段设置为指定的新值。

setBoolean(Object obj, boolean z)、setByte(Object obj, byte b)、setChar(Object obj, char c)、setDouble(Object obj, double d)、setFloat(Object obj, float f)、setInt(Object obj, int i)、setLong(Object obj, long l)、setShort(Object obj, short s)

String toGenericString()返回一个描述此 Field（包括其一般类型）的字符串。

String **toString**()返回一个描述此 Field 的字符串。

Constructor<T>、Method、Field 从类 java.lang.reflect.AccessibleObject 继承的方法：getAnnotations,isAccessible,isAnnotationPresent,**setAccessible**,setAccessible

Modifier 类：

static boolean isAbstract(int mod)如果整数参数包括 abstract 修饰符，则返回 true，否则返回 false。

isFinal(int mod)、isInterface(int mod)、isNative(int mod)、isPrivate(int mod)、isProtected(int mod)、isPublic(int mod)、isStatic(int mod)

static boolean isStrict(int mod)如果整数参数包括 strictfp 修饰符，则返回 true，否则返回 false。

static boolean isSynchronized(int mod)如果整数参数包括 synchronized 修饰符，则返回 true，否则返回 false。

static boolean isTransient(int mod)如果整数参数包括 transient 修饰符，则返回 true，否则返回 false。

static boolean isVolatile(int mod)如果整数参数包括 volatile 修饰符，则返回 true，否则返回 false。

static String **toString**(int mod)返回描述指定修饰符中的访问修饰符标志的字符串。

14.7 动态代理

代理是基本的设计模式之一，它是你为了提供额外的或不同的操作，而插入的用来代替“实际”对象的对象。这些操作通常设计与“实际”对象的通信，因此代理通常充当着中间人的角色。

Java 的动态代理可以动态地创建代理并动态地处理对所代理方法的调用。

InvocationHandler 接口：

Object **invoke**(Object proxy, Method method, Object[] args)在代理实例上处理方法调用并返回结果。(参数 proxy，一般是指代理类；method 是对应于在代理实例上调用的接口方法的 Method 实例。Method 对象的声明类将是在其中声明方法的接口，该接口可以是代理类赖以继承方法的代理接口的超接口；args 为该方法的参数数组。这个抽象方法在代理类中动态实现)

Proxy 类：

protected Proxy(InvocationHandler h)使用其调用处理程序的指定值从子类（通常为动态代理类）构建新的 Proxy 实例。

static InvocationHandler getInvocationHandler(Object proxy)返回指定代理实例的调用处理程序。

static Class<?> **getProxyClass**(ClassLoader loader, Class<?>… interfaces)返回代理类的 java.lang.Class 对象，并向其提供类加载器和接口数组。(用于产生代理类，参数要提供 interface 数组，他会产生这些 interface 的“虚拟实现”，用来冒充真实的对象)

static boolean isProxyClass(Class<?> cl)当且仅当指定的类通过 getProxyClas 方法或 newProxyInstance 方法动态生成为代理类时，返回 true。

static Object **newProxyInstance**(ClassLoader loader,Class<?>[]interfaces,InvocationHandler h)返回一个指定接口的代理类实例，该接口可以将方法调用指派到指定的调用处理程序。 (loader—定义代理类的类加载器；interfaces—代理类要实现的接口列表；h—指派方法调用的调用处理程序)

动态代理可以将所有调用重定向到调用处理器，因此通常会向调用处理器的构造器传递一个“实际”对象的引用，从而使得调用处理器在执行其中介任务时，可以将请求转发。

invoke()方法传递进来代理对象，以防你需要区分请求的来源，在很多情况下不关注此。然而，在 invoke()内部，在代理上调用方法时需要当心，因为对接口的调用将被重定向为对代理的调用。

通常，你会执行被代理的操作，然后使用 Method.invoke()将请求转发给被代理对象，并传入必要的参数。

14.8 空对象

“空对象”是通过假设所有对象都是有效的，而不必浪费时间去检查 null。

空对象最有用之处在于它更靠近数据，因为对象表示的是问题空间内的实体。

14.9 接口与类型信息

interface 关键字允许程序员隔离构件，进而降低耦合性。但是通过使用 RTTI 使得代码的耦合程度超过期望。

解决方案一是直接声明；二是对实现使用包访问权限（向下转型被禁止）。

Class.getDeclaredMethod、Method.setAccessible(true)、Field.invoke()调用所有方法，甚至 private。

Class.getDeclaredField、Field. setAccessible(true)、Field.set***()方法对域进行设置。

final 域实际上在遭遇修改时是安全的。运行时系统会在不抛异常的情况下接受任何修改的尝试，但是**实际上不会发生任何修改**。

第 15 章　泛　　型

Java SE5 的重大变化：泛型的概念。泛型实现了参数化类型的概念，使代码可以应用于多种类型。意思是“适用于许多许多的类型”。创建参数化类型的一个实例时，**编译器**负责转型操作，并且保证类型的正确性。

15.1 与 C++的比较

15.2 简单泛型

泛型的主要目的之一就是为了创造容器类。Java 泛型的核心概念：告诉编译器向想使用什么类型，然后编译器帮你处理一切细节。泛型的主要目的之一就是用来指定容器要持有什么类型的对象，而且由**编译器**来保证类型的正确性。

15.2.1 一个元组类库

元组（tuple）是将一组对象直接打包存储在其中的一个单一对象。这个容器对象允许读取其中元素，但是不允许向其中存放新的对象。（这个概念也称为数据传送对象，或信使）

final 声明确保能够保护 public 元素，在对象被构造出来后，声明为 final 的元素不能被再赋予其他值。

元组可以具有任意长度，元组中的对象可以使任意不同类型的。

15.2.2 一个堆栈类

传统的下推堆栈。使用末端哨兵（end sentinel）来判断堆栈何时为空。

15.2.3 RandomList

持有特定类型对象的列表，每次调用其上的 select()方法，可以随机地取一个元素。

15.3 泛型接口

泛型可以应用于接口。例如生成器（generator），这是专门负责创建对象的类，实际上，这是工厂方法设计模式的一种应用。生成器无需额外的信息就知道如何创建新对象。

接口使用泛型与类使用泛型没有什么区别。

基本类型无法作为参数类型。Java 具备了自动打包和自动拆包的功能，可以方便地进行基本类型和其相应的包装器类型之间进行转换。

15.4 泛型方法

可以在类中包含参数化方法，而与是否是泛型类没有关系。

基本指导原则：无论何时，只要你能做到，就应该尽量使用泛型方法。static 方法无法访问泛型类的类型参数，因此，static 方法需要使用泛型能力，就必须使其成为泛型方法。

定义泛型方法，只需将泛型参数列表置于返回值之前。

类型参数推断（type argument inference）指不必指明参数类型，编译器会为我们找到具体的类型。

15.4.1 杠杆利用类型参数推断

类型参数推断避免了重复的泛型参数列表。只对赋值操作有效，其他时候不起作用。

显式类型说明

要显式地指明类型，必须在点操作符与方法名之间插入尖括号，然后把类型置于尖括号内。如果在定义该方法类的内部，必须在点操作符之前使用 this 关键字，如果是使用 static 的方法，必须在点操作符之前加上类名。

15.4.2 可变参数与泛型方法

15.4.3 用于 Generator 的泛型方法

15.4.4 一个通用的 Generator

15.4.5 简化元组的使用

15.4.6 一个 Set 实用工具

15.5 匿名内部类

泛型可以应用于内部类以及匿名内部类。

15.6 构建复杂模型

15.7 擦除的神秘之处

ArrayList<String>和 ArrayList<Integer>会被程序（运行时）认为它们是相同的类型。

在泛型代码内部，无法获得任何有关泛型参数类型的信息。

Java 泛型是使用擦除来实现的，任何具体的类型信息都被擦除，唯一知道的就是在使用一个对象。因此 List<String>和 List<Integer>在运行时事实上是相同的类型。这两种形式都被擦除成它们的“原生”类型，即 List。

15.7.1 C++的方式

Manipulator<T extends HasF>泛型类型参数将擦除到它的第一个边界（限定类型），我们还提到了类型参数的擦除。编译器实际上会把类型参数替换为它的擦除，T 擦除到了 HasF，就好像在类的声明中用 HasF 替代了 T 一样。

泛型使用时机：只有希望使用的类型参数比某个具体类型（以及它的所有子类型）更加“泛化”时——也就是说，当希望代码能够**跨**多个类工作时，使用泛型才有所帮助。

必须查看所有的代码，并确定它是否“足够复杂”到必须使用泛型的程序。

15.7.2 迁移兼容性

泛型类型被当作第二类型处理，即不能再某些重要的上下文环境中使用的类型。泛型类型只有在静态类型检查期间才出现，在此之后，程序中的所有泛型类型都被擦除，替换为他们的非泛型上界。

擦除的核心动机是它使得泛化的客户端可以用非泛化的类库来使用，被称为“迁移兼容性”。

15.7.3 擦除的问题

擦除主要的正当理由是从非泛化代码到泛化代码的转变过程，以及在不破坏现有类库的情况下，将泛型融入 Java 语言。

擦除的代价是显著的。①泛型不能用于显式地引用运行时类型的操作之中，例如转型、instanceof 操作和 new 表达式；②使用泛型并不是强制的；③当将类型参数不要仅仅当作 Object 处理时，就需要付出额外努力来管理边界。

@SuppressWarnings(“unchecked”)

15.7.4 边界处的动作

泛型困惑之处即可以表示没有任何意义的事物。

即使擦除在方法或类内部移除了有关实际类型的信息，编译器仍旧可以确保在方法或类中使用的类型的内部一致性。

因为擦除在方法体中移除了类型信息，所以在运行时的问题就是边界，即对象进入和离开方法的地点。也是编译器在编译器执行类型检查并插入转换代码的地点。

15.8 擦除的补偿

任何在运行时需要知道确切类型信息的操作都将无法工作，但是可以通过引入类型标签对擦除进行补偿。意味着需要显式地传递类型的 Class 对象。

instanceof 的尝试失败，是因为类型信息被擦除，如引入类型标签，就可以转而使用动态的 isInstance()。编译器将确保类型标签可以匹配泛型参数。

15.8.1 创建类型实例

对创建一个 new T()的尝试将无法实现，部分原因是因为擦除，而另一部分原因是因为编译器不能验证 T 具有默认（无参）构造器。

Java 解决方案是传递一个工厂对象，并使用它来创建新的实例。最便利的工厂对象就是①Class 对象。使用 newInstance 来创建类型的对象。问题是需要创建的对象必须具有默认的构造器，而这个错误不是编译期捕获的。

因而建议②使用显式的工厂，并将限制其类型，使得只能接受实现了这个工厂的类。

另一种方式是③模板方法设计模式。在子类中定义的、用来产生子类类型的对象。

15.8.2 泛型数组

不能创建泛型类型的数组。解决方案：①在任何想要创建泛型数组的地方都使用 ArrayList；②可以创建泛型类型数组的引用，但是永远都不能创建这个确切类型的数组（包括类型参数）；③创建一个 Object 数组，并将其转型为所希望的数组类型，可以编译，运行产生 ClassCaseException，原因是数组将跟踪它们的实际类型，而这个类型在数组被创建时确定，因此，即使数组已经被转型为泛型数组，这个信息只存在于编译器，在运行时，仍旧是 Object 数组，而这将引发问题。④成功创建泛型数组的唯一方式就是创建一个被擦除类型的新数组，然后对其转型。⑤Java SE 7 使用@SafeVarargs 标注来消除创建泛型数组的有关限制，方法如下：@SafeVarargs static <E> E[] array(E…array) { return array; } Pair<String> pair1 = …; Pair<String>pair2 = …; Pair<String>[] table = array(pair1,pair2);【Java 核心技术 P540】

不能声明泛型数组，T[] array = new T[sz]，因此，可以①创建一个对象（Object）数组，然后将其转型。因为有了擦除，数组的运行时类型就只能是 Object[]，如果⑴立即将其转型为 T[]，那么在编译期该数组的实际类型将丢失，而编译器可能会错过某些潜在的错误检查。⑵在集合内部使用 Object[]，然后使用数组元素时，添加一个对 T 的转型。如果试着将 Object[]转型为 T[]，仍旧是不正确的，将在编译器产生警告，在运行时产生异常。因此，没有任何方式可以推翻低层的数组类型，它只能是 Object[]。②传递一个**类型标记**（Class<T>），以便从擦除中恢复，创建需要的实际类型的数组。该数组运行时的类型是确切类型 T[]。

15.9 边界

边界可以在用于泛型的参数类型上设置限制条件。其潜在的一个更重要的效果是可以按照自己的边界类型来调用方法。

Java 泛型重用了 extends 关键字。

15.10 通配符（PECS 原则）

通配符即是泛型参数表达式中的问号。所代表的其实是一组类型，但具体类型是未知的。

数组具有内建的协变类型，数组对象可以保留有关它们包含的对象类型的规则。因此会在向数组中放置异构类型时抛出异常。

不能将一个设计子类的泛型赋值给一个涉及父类的泛型。是因为泛型没有内建的协变类型。编译器和运行时系统都不知道想用类型做些什么，以及应该采取什么样的规则。

通配符在两个类型之间建立某种类型的向上转型关系。

flist 类型现在是 List<? extends Fruit>，可以读作“具有任何从 Fruit 继承的类型的列表”。实际上并不意味着这个 List 将持有任何类型的 Fruit。通配符引用的是明确的类型，因此它意味着“某种 flist 引用没有指定的具体类型”。 List<? extends Fruit>可以合法地指向一个 List<Apple>和 List<Orange>，一旦执行这种类型的向上转型，就将丢失掉向其中传递任何对象的能力，甚至传递 Object 也不行。另一方面，调用一个返回 Fruit 的方法，则是安全的，因为这个 List 中的任何对象至少具有 Fruit 类型，编译器将允许这么做。

15.10.1 编译器有多聪明

泛型类的设计者决定哪些调用是“安全的”（PECS 原则），并使用 Object 类型（不涉及通配符）作为其参数类型。

15.10.2 逆变

超类型通配符即可以声明通配符是由某个特定类的任何基类来界定的，方法是指定<? super MyClass>，甚至或者使用类型参数<? super T>。

参数是List<? super T>，因此这个List将持有从T导出的某种具体类型，这样可以安全地将一个T类型的对象或者从T导出的任何对象作为参数传递给List的方法。

15.10.3 无界通配符

无界通配符<?>实际是声明:“我是想用Java泛型来编写这段代码，在这里并不是原生类型，但是在当前这种情况下，泛型参数可以持有任何类型。”

区分List list、List<?> list和List<? extends Object>

15.10.4 捕获转换

15.11 问题

15.11.1 任何基本类型都不能作为参数类型

不能将基本类型作为参数，解决方法：自动包装机制。自动包装机制不能用于数组。

15.11.2 实现参数化接口

一个类不能实现同一个泛型接口的两种变体（非泛型版本是合法的），因为由于擦除的原因，这两个变体会成为相同的接口。(错误：无法使用以下不同的参数继承Payable:<Hourly>和<Employee>)

15.11.3 转型和警告

使用带有泛型类型参数的转型或instanceof不会有任何效果。实质是将Object转型为Object。

泛型没有消除对转型的需要，这就会由编译器产生警告，而这个警告是不恰当的。

Java SE5中引入的新的转型形式，既通过泛型类来转型（cast（）方法），但是不能转型到实际类型。只能声明为（List<Widget>）List.class.cast（in.readObject（）），而无法声明为List<Widget>.class.cast（in.readObject（））。

15.11.4 重载

void f(List<T> v)和void f(List<W> v)由于擦除的原因，重载方法将产生相同的类型签名。(错误：名称冲突：f(List<W>)和f(List<T>)具有相同类型签名)

15.11.5 基类劫持了接口

子类（Cat）和父类（ComparablePet）同时实现Comparable<T>接口，导致出现编译（错误：无法使用不同参数继承Comparable：<Cat>和<ComparablePet>）。

15.12 自限定的类型

15.12.1 古怪的循环泛型

不能直接继承一个泛型参数，但是，可以继承在其自己的定义中使用这个泛型参数的类。含义是:“我在创建一个新类，它继承自一个泛型类型，这个泛型类型接受我的类的名字作为其参数”。(class **Subtype** extends BasicHolder<**Subtype**> { })

基类用导出类替代其参数。这意味着泛型基类变成了一种其所有导出类的公共功能的模板，但是这些功能对于其所有参数和返回值，将使用导出类型。

15.12.2 自限定

自限定采取额外的步骤，强制泛型当作其自己的边界参数来使用。自限定的参数意义是它可以保证类型参数必须与正在被定义的类相同。

class SelfBounded<T extends SelfBounded<T>> { }

自限定惯用法不是可强制执行的。class F extends SelfBounded { }

自限定限制只能强制作用于继承关系。如果使用自限定，就应该了解这个类所用的类型参数将与使用这个参数的类具有相同的基类型。

自限定用于泛型方法可以防止方法被用于除上述形式的自限定参数之外的任何事物上。

15.12.3 参数协变

自限定类型的价值在于它们可以产生协变参数类型——方法参数类型会随子类而变化。

Java SE5 支持协变返回类型。然而，在非泛型代码中，参数类型不能随子类型发生变化。set(derived)和 set(base)都是合法的，没有覆盖，而是重载了方法。(桥方法被合成来保持多态；为保持类型安全性，必要时插入强制类型转换)【Java 核心技术 P537】

15.13 动态类型安全

可以向 Java SE5 之前的代码传递泛型容器，因而旧式代码仍旧有可能破坏容器。Java SE5 的 java.util.Collections 中工具，可以解决在这种情况下的类型检查问题，它们是：静态方法 checkedCollection()、checkedList()、checkedMap()、checkedSet()、checkedSortedMap()和 checkedSortedSet()。这些方法每一个都会将希望检查的容器当作第一个参数接受，并将你希望强制要求的类型作为第二个参数。受检查的容器在试图插入类型不正确的对象时抛出 ClassCastException。

```
List<Dog> dogs2 = Collections.checkedList(new ArrayList<Dog>(),Dog.class);
```

15.14 异常

由于擦除的原因，将泛型应用于异常是非常受限的。catch 语句不能捕获泛型类型的异常，因为编译器和运行时必须知道异常的确切类型。

但是，类型参数能在一个方法的 throws 子句中用到。这使得可以编写随检查型异常的类型而发生变化的泛型代码。

```
interface Processor<T,E extends Exception> {
    void process(List<T> resultCollector) throws E;
}
```

15.15 混型

混型是混合多个类的能力，以产生一个可以表示混型中所有类型的类。使组装多个类变得简单易行。

混型的价值之一是它们可以将特性和行为一致地应用于多个类之上。

15.15.1 C++中的混型

C++使用多重继承的最大理由，就是为了使用混型。更优雅的方式是使用参数化类型，因为混型就是继承自其类型参数的类。

Java 不允许如此，擦除会忘记基类类型，因此泛型类不能直接继承自一个泛型参数。

15.15.2 与接口混合

就是使用代理，每个混入类型都要求在 Mixin 类中有一个相应的域，而必须在 Mixin 类中编写所有需要的方法，将方法调用**转发**给恰当的对象。使用复杂混型时，代码数量急速增加。

15.15.3 使用装饰器模式

装饰器用于满足各种可能的组合，而直接子类化会产生过多的类。装饰器模式使用分层对象来动态透明地向单个对象中添加责任。装饰器指定包装在最初的对象周围的所有对象都具有相同的基本接口。某些事物是可装饰的，可以通过将其他类包装在这个装饰对象的四周，来将功能分层。这使得装饰的使用时透明的——无论对象是否被装饰，都拥有一个可以向对象发送的公共消息集。装饰器可以添加新方法，这将是受限的。

装饰器是通过使用组合和形式化结构（可装饰物/装饰器层次结构）来实现的，混型是基于继承的。因此可以将基于参数化类型的混型当作一种泛型装饰器机制，而这种机制不需要装饰器设计模式的继承结构。

装饰器的缺陷是它只能有效地工作于装饰中的一层（最后一层），是对混型提出的问题的一种局限的解决方案。

15.15.4 与动态代理混合

可以使用动态代理来创建一种比装饰器更贴近混型模型的机制。通过使用动态代理，所

产生的类的动态类型将会是已经混入的组合类型。

动态代理的限制，每个被混入的类都必须是某个接口的实现。

只有动态类型而不是非静态类型才包含所有的混入类型。

15.16 潜在类型机制

编写能够尽可能广泛地应用的代码，为了实现这一点，需要各种途径来放松我们的代码将要作用的类型所作的限制，同时不丢失静态类型检查的好处。

解决方案：潜在类型机制或结构化类型机制（鸭子类型机制，即如果它走起来像鸭子，并且叫起来也像鸭子，那么就可以把它当作鸭子对待）。Java 没有对这种特性的支持。

潜在类型机制使得可以横跨类继承结构，调用不属于某个公共接口的方法。

15.17 对缺乏潜在类型机制的补偿

需要额外的努力：强制要求使用一个类或者接口，并在边界表达式中指定它。

15.17.1 反射

通过反射，能够动态地确定所需要的方法是否可用并调用他们。

15.17.2 将一个方法应用于序列

反射将所有类型检查都转移到了运行时。如何解决实现编译器类型检查和潜在类型机制？

示例存在 FilledList 为了使某种类型可用，必须有默认（无参）构造器。解决方法：定义一个工厂接口，它有一个可以生成对象的方法，然后 FilledList 将接受这个接口而不是这个类型标记的“原生工厂”，这样做问题是 FilledList 中使用的所有类都必须实现这个工厂接口。

类型标记技术

15.17.3 当你并未碰巧拥有正确的接口时

潜在类型机制的参数化类型机制的价值所在，因为不会受任何特定类库的创建者过去所作的设计决策的支配，因此不需要在每次碰到一个没有考虑到你的具体情况的新类库时，都去重写代码。

示例限制在 Collection 继承层次结构之内，即便 SimpleQueue 有 add()方法，也不能工作。

15.17.4 用适配器仿真潜在类型机制

潜在类型机制意味着：“我不关心我在这里使用的类型，只要它具有这些方法即可。”实际上，潜在类型机制创建了一个包含所需方法的隐式接口。

可以使用适配器来适配已有的接口，以产生想要的接口。

Fill2 只需实现 Addable 的对象，而 Addable 已经为 Fill 编写了——它是我希望编译器帮我创建的潜在类型的一种体现。

使用这样的适配器看起来是对缺乏潜在类型机制的一种补偿，因此允许编写出真正的泛化代码。但，这是一个额外的步骤，并且是类库的创建者和消费者都必须理解的事物。潜在类型机制通过移除额外的步骤，使得泛化代码更容易应用。

15.18 将函数对象用作策略

策略设计模式将“变化的事物”完全隔离到了一个函数对象中。函数对象就是在某种程度上行为像函数的对象——一般地，会有一个相关的方法。函数对象的价值就在于，与普通方法不同，它们可以传递出去，并且还可以拥有在多个调用之间持久化的状态。

在 C++中，潜在类型机制将在你调用函数时负责协调各个操作，但在 Java 中，需要编写函数对象来将泛型方法适配为我们特定的需求。

15.19 总结：转型真的如此糟糕吗？

第16章 数 组

16.1 数组为什么特殊

区 分	数 组	容 器
☆效率	效率最高的存储和随机访问对象引用的方式,代价是大小固定,在其生命周期中不可改变	空间自动分配,弹性需要开销,因此,效率比数组低很多
类型	优于泛型之前的数组,可以通过编译期检查,防止插入错误类型和抽取不当类型	泛型之前,处理对象时,将他们视为没有任何具体类型,即将对象当作根类Object处理
保存基本类型	可以持有基本类型	自动包装,可以方便地用于基本类型

16.2 数组是第一级对象

数组标识符只是一个引用,指向在堆中创建的一个真实的对象,这个(数组)对象用以保存指向其他对象的引用。可以作为数组初始化语法的一部分隐式地创建此对象,或者用new表达式显示的创建。只读成员length是数组对象的一部分,表示此数组对象可以存储多少元素,“[]”语法是访问数组对象唯一的方式。

对象数组保存的是引用,基本类型数组直接保存基本类型的值。

自动初始化:新生成基本类型数值型初始化为0,字符型(char)初始化为(char)0,布尔型(boolean)初始化为false;新生成数组对象,其所有的引用被自动初始化为null。

动态的聚集初始化

16.3 返回一个数组

Java直接“返回一个数组”,而无需担心为数组负责,只要需要,一直存在,使用完后,垃圾回收会清理掉它。

16.4 多维数组

创建多维数组,可以通过使用花括号将每个向量分隔开,每对花括

号将每个向量分隔开,每对花括号括起来的集合都会把你带到下一级数组。

数组中构成矩形的每个向量都可具有任意的长度,称为粗糙数组。

Arrays.deepToString()可以将多维数组转换为多个String。

16.5 数组与泛型

不能实例化具有参数化类型的数组,擦除会移除参数类型信息,而数组必须知道它所持有的确切类型,以强制保证类型安全。可以参数化数组本身的类型。

```
public T[] f(T[] arg) { return arg; }
```

编译允许创建具有参数化类型数组的引用,可以创建非泛型数组,然后将其转型。一旦拥有了对List<String>[]引用,将会得到某些编译器检查。问题是数组是协变类型的,List<String>也是Object[],利用这个可以将ArrayList<Integer>赋值到数组,而不会由任何编译期或运行时错误。

泛型容器是比泛型数组更好的选择。

泛型在类或方法的边界处有效,而在类或方法的内部,擦除会使得泛型变得不适用。

16.6 创建测试数据

16.6.1 Arrays.fill()

用同一个值填充各个位置,而针对对象而言,就是复制同一个引用进行填充。

16.6.2 数据生成器

通过选择Generator的类型来创建任何类型的数据。

16.6.3 从Generator中创建数组

16.7 Arrays实用功能

16.7.1 复制数组

Java标准类库提供static方法System.arraycopy()，用于复制数组比用for循环复制快很多。

static void arraycopy(Object src, int srcPos, Object dest, int destPos, int length)从指定源数组中复制一个数组，复制从指定的位置开始，到目标数组的指定位置结束。对数组越界会导致异常。

16.7.2 数组的比较

Arrays提供了重载后的equals()方法，用来比较整个数组。此方法针对所有基本类型与Object都做了重载。数组相等的条件是元素个数必须相同，并且对应位置的元素也相等，通过每个元素使用equals()作比较判断。

16.7.3 数组元素的比较

排序必须根据对象的实际类型执行比较操作。Java有两种方式提供比较功能。第一种是实现java.lang.Comparable接口，使得类具有“天生”的比较能力。此接口只有CompareTo<T>方法，此方法接收另一个Object为参数，如果当前对象小于参数则返回负值，如果相等则返回零，如果当前对象大于参数则返回正值。第二种创建一个实现了Comparator<T>接口的单独的类。这个类有compare()和eauqls()两个方法，Object自带equals()方法，所以只需要默认的Object的equals()方法。

Collections类包含一个reverseOrder()方法，该方法可以产生一个Comparator，它可以反转自然的排序顺序。

16.7.4 数组排序

使用内置的排序方法，就可以对任意的基本类型数组排序；也可以对任意的对象数组进行排序，只要该对象实现了Comparable接口或具有相关联的Comparator。

String排序算法依据字典编排顺序排序，所以大写字母开头的词都放在前面输出，然后才是小写字母开头的词。忽略大小写字母将单词都放在一起排序，使用String.CASE_INSENSITIVE_ORDER。

16.7.5 在已排序的数组中查找

在已排序的数组中，使用Arrays.binarySearch()执行快速查找。如未排序，将产生不可预料的结果。

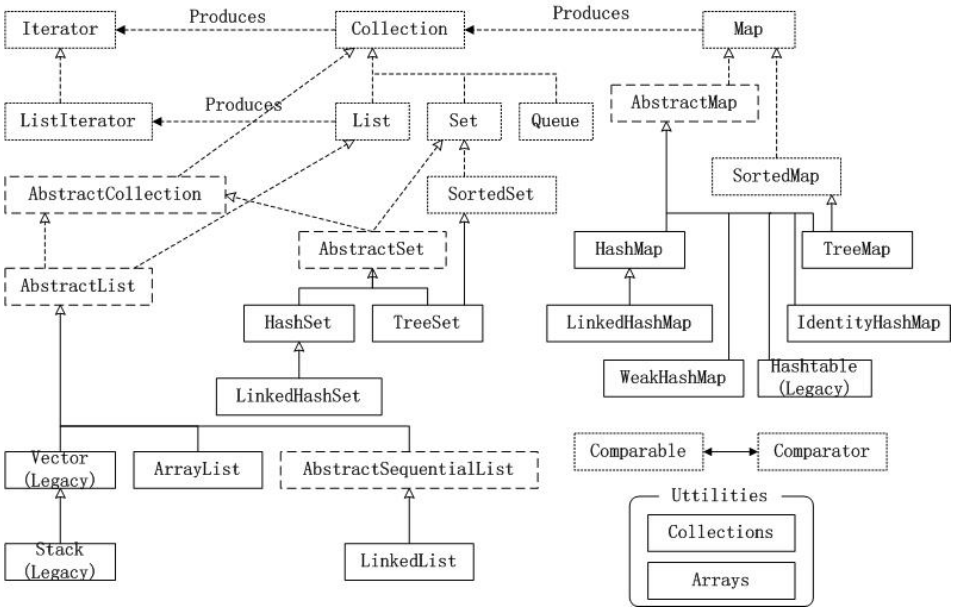
Arrays.binarySearch()产生的返回值等于或大于0，否则，产生负返回值，表示若要保持数组的排序状态此目标要素应该插入的位置。这个负值的计算方式是：-（插入点）-1

“插入点”是指第一个大于查找对象的元素在数组中的位置。

如果数组包含重复的元素，搜索算法无法保证找到的是这些副本中的哪一个。搜索算法确实不是专门为包含重复元素的数组而设计的，不过仍然可用。

第 17 章　容器深入研究

17.1 完整的容器分类法



Java SE5 添加了：

Queue 接口及其实现 PriorityQueue 和各种风格的 BlockingQueue;

ConcurrentMap 接口及其实现 ConcurrentHashMap，它们也是用于多线程机制的;

CopyOnWriteArrayList 和 CopyOnWriteArraySet,也是用于多线程机制的;

EnumSet 和 EnumMap，为使用 enum 而设计的 Set 和 Map 的特殊实现;

在 Collections 类中的多个便利方法。

17.2 填充容器

第一种是使用 Collections.nCopies() 创建传递给构造器的 List；第二种 Collections.fill()替换已经在 List 中存在的元素，而不能添加新的元素。

17.2.1 一种 Generator 解决方案

构建接收 Generator 和 quantity 数值并将它们作为构造器参数的类。

17.2.2 Map 生成器

17.2.3 使用 Abstract 类

每个 java.util 容器都有自己的 Abstract 类，它们提供了该容器的部分实现，因此必须做的只是去实现那些产生想要的容器所必须的方法。

享元设计模式：在解决方案需要过多对象，或产生普通对象太占用空间时使用享元。该模式使得对象的一部分可以被具体化，因此，与对象中的所有事物都包含在对象内部不同，可以在更加高效的外部表中查找对象的一部分或整体。

17.3 Collection 的功能方法

boolean add(E e)确保此 collection 包含指定的元素（可选操作）。

boolean addAll(Collection<? extends E> c)将指定 collection 中的所有元素都添加到此 collection 中（可选操作）。

void clear()移除此 collection 中的所有元素（可选操作）。

boolean contains(Object o)如果此 collection 包含指定的元素，则返回 true。

boolean containsAll(Collection<?> c)如果此 collection 包含指定 collection 中的所有元素，则返回 true。

boolean equals(Object o)比较此 collection 与指定对象是否相等。

int hashCode()返回此 collection 的哈希码值。

boolean isEmpty()如果此 collection 不包含元素，则返回 true。

Iterator<E> iterator()返回在此 collection 的元素上进行迭代的迭代器。

boolean remove(Object o)从此 collection 中移除指定元素的单个实例，如果存在的话（可选操作）。

boolean removeAll(Collection<?> c)移除此 collection 中那些也包含在指定 collection 中的所有元素（可选操作）。

boolean retainAll(Collection<?> c)仅保留此 collection 中那些也包含在指定 collection 的元素（可选操作）。

int size()返回此 collection 中的元素数。

Object[] toArray()返回包含此 collection 中所有元素的数组。

<T> T[] toArray(T[] a)返回包含此 collection 中所有元素的数组；返回数组的运行时类型与指定数组的运行时类型相同。

17.4 可选操作

执行各种不同的添加和移除的方法在Collection接口中都是可选操作。这意味着实现类并不需要为这些方法提供功能定义。

将方法定义为可选的可以防止设计中出现接口爆炸的情况。未获支持的操作是一种特例，可以延迟到需要时再实现。但为了让这种方式能够工作：

1、UnsupportedOperationException必须是一种罕见事件。

2、如果一个操作是未获支持的，那么实现接口的时候可能会导致UnsupportedOperationException异常，而不是将程序交给客户后才出现此异常。

未获支持的操作只有在运行时才能探测到，因此它们表示动态类型检查。

17.4.1 未获支持的操作

最常见的未获支持的操作，都来源于背后由固定尺寸的数据结构支持的容器。

Arrays .asList()会生成一个List，它基于一个固定大小的数组，仅支持那些不会改变数组大小的操作，任何会引起对底层数据结构的尺寸进行修改的方法都会产生一个UnsupportedOperationException异常，以表示对未或支持操作的调用。

Collections.unmodifiableList()产生不可修改的列表。

17.5 List的功能方法

boolean add(E e)向列表的尾部添加指定的元素（可选操作）。

void add(int index, E element)在列表的指定位置插入指定元素（可选操作）。

boolean addAll(Collection<? extends E> c)添加指定 collection 中的所有元素到此列表的结尾，顺序是指定 collection 的迭代器返回这些元素的顺序（可选操作）。Boolean addAll(int index, Collection<? extends E> c)将指定 collection 中的所有元素都插入到列表中的指定位置（可选操作）。

void clear()从列表中移除所有元素（可选操作）。

boolean contains(Object o)如果列表包含指定的元素，则返回 true。

boolean containsAll(Collection<?> c)如果列表包含指定 collection 的所有元素，则返回 true。

boolean equals(Object o)比较指定的对象与列表是否相等。

E get(int index)返回列表中指定位置的元素。

int hashCode()返回列表的哈希码值。

int indexOf(Object o)返回此列表中第一次出现的指定元素的索引；如果此列表不包含该元素，则返回 -1。

boolean isEmpty()如果列表不包含元素，则返回 true。

Iterator<E> iterator()返回按适当顺序在列表的元素上进行迭代的迭代器。

int lastIndexOf(Object o)返回此列表中最后出现的指定元素的索引；如果列表不包含此元素，则返回 -1。

ListIterator<E> listIterator()返回此列表元素的列表迭代器（按适当顺序）。ListIterator<E> listIterator(int index)返回列表中元素的列表迭代器（按适当顺序），从列表的指定位置开始。

E remove(int index)移除列表中指定位置的元素（可选操作）。

boolean remove(Object o)从此列表中移除第一次出现的指定元素（如果存在）（可选操作）。Boolean removeAll(Collection<?> c)从列表中移除指定 collection 中包含的其所有元素（可选操作）。

boolean retainAll(Collection<?> c)仅在列表中保留指定 collection 中所包含的元素（可选操作）。

E set(int index, E element)用指定元素替换列表中指定位置的元素（可选操作）。

int size()返回列表中的元素数。

List<E> subList(int fromIndex, int toIndex)返回列表中指定的 fromIndex（包括 ）和 toIndex（不包括）之间的部分视图。

Object[] toArray()返回按适当顺序包含列表中的所有元素的数组（从第一个元素到最后一个元素）

<T> T[] toArray(T[] a)返回按适当顺序（从第一个元素到最后一个元素）包含列表中所有元素的数组；返回数组的运行时类型是指定数组的运行时类型。

17.6 Set 和存储顺序

Set(interface)	存在 Set 的每个元素必须是唯一的，因为 Set 不保存重复元素。加入 Set 必须定义 equals()方法以确保对象的唯一性。Set 接口不保证维护元素的次序。
HashSet*	为快速查找而设计的 Set。存入 HashSet 的元素必须定义 hashCode()。
TreeSet	保存次序的 Set，低层为树结构。使用它可以从 Set 中提取有序的序列。元素必须实现 Comparable 接口。
LinkedHashSet	具有 HashSet 的查询速度，且内部使用链表维护元素的顺序（插入的次序）。元素必须实现 hashCode()方法。

星号表示，没有其他的限制，应该是默认的选择，它对速度进行了优化。

良好的编程风格而言，在覆盖 equals()方法时，总是同时覆盖 hashCode()方法。

① CopyOnWirteArraySet 使用 equals 方法来判断元素是否相同；

② TreeSet 使用 comparableTo 方法来判断元素是否相同；

③ HashSet 使用 hashCode 方法判断元素是否相同。

在 TreeSet 中使用没有实现 Comparable 的类型，将抛出异常。

Set 接口：

boolean add(E e)如果 set 中尚未存在指定的元素，则添加此元素（可选操作）。

boolean addAll(Collection<? extends E> c)如果 set 中没有指定 collection 中的所有元素，则将其添加到此 set 中（可选操作）。

void clear()移除此 set 中的所有元素（可选操作）。

boolean contains(Object o)如果 set 包含指定的元素，则返回 true。

boolean containsAll(Collection<?> c)如果此 set 包含指定 collection 的所有元素，则返回 true。

boolean equals(Object o)比较指定对象与此 set 的相等性。

int hashCode()返回 set 的哈希码值。

boolean isEmpty()如果 set 不包含元素，则返回 true。

Iterator<E> iterator()返回在此 set 中的元素上进行迭代的迭代器。

boolean remove(Object o)如果 set 中存在指定的元素，则将其移除（可选操作）。

boolean removeAll(Collection<?> c)移除 set 中那些包含在指定 collection 中的元素（可选操作）。

boolean retainAll(Collection<?> c)仅保留 set 中那些包含在指定 collection 中的元素（可选操作）。

int size()返回 set 中的元素数（其容量）。

Object[] toArray()返回一个包含 set 中所有元素的数组。

<T> T[] toArray(T[] a)返回一个包含此 set 中所有元素的数组；返回数组的运行时类型是指定数组的类型。

17.6.1 SortedSet

SortedSet 中元素可以保证处于排序状态（升序）。

SortedSet 接口：

Comparator<? super E> comparator()返回对此 set 中的元素进行排序的比较器；如果此 set 使用

其元素的自然顺序，则返回 null。

E first()返回此 set 中当前第一个（最低）元素。

SortedSet<E> headSet(E toElement)返回此 set 的部分视图，其元素严格小于 toElement。

E last()返回此 set 中当前最后一个（最高）元素。

SortedSet<E> subSet(E fromElement, E toElement)返回此 set 的部分视图，其元素从 fromElement（包括）到 toElement（不包括）。

SortedSet<E> tailSet(E fromElement)返回此 set 的部分视图，其元素大于等于 fromElement。

17.7 队列

除了并发应用，Queue 在 Java SE5 仅有两个实现是 LinkedList 和 PriorityQueue，它们的差异在于排序行为而不是性能。

Queue 接口：

boolean add(E e)将指定的元素插入此队列（如果立即可行且不会违反容量限制），在成功时返回 true，如果当前没有可用的空间，则抛出 IllegalStateException。

E element()获取，但是不移除此队列的头。

boolean offer(E e)将指定的元素插入此队列（如果立即可行且不会违反容量限制），当使用有容量限制的队列时，此方法通常要优于 add(E)，后者可能无法插入元素，而只是抛出一个异常。

E peek()获取但不移除此队列的头；如果此队列为空，则返回 null。

E poll()获取并移除此队列的头，如果此队列为空，则返回 null。

E remove()获取并移除此队列的头。

17.7.1 优先级队列

PriorityQueue 按照其自然顺序进行排序，或者根据构造队列时提供的 Comparator 进行排序，具体取决于所使用的构造方法。优先级队列不允许使用 null 元素。

boolean add(E e)将指定的元素插入此优先级队列。

void clear()从此优先级队列中移除所有元素。

Comparator<? super E> **comparator**()返回用来对此队列中的元素进行排序的比较器；如果此队列根据其元素的自然顺序进行排序，则返回 null。

boolean contains(Object o)如果此队列包含指定的元素，则返回 true。

Iterator<E> iterator()返回在此队列中的元素上进行迭代的迭代器。

boolean offer(E e)将指定的元素插入此优先级队列。

E peek()获取但不移除此队列的头；如果此队列为空，则返回 null。

E poll()获取并移除此队列的头，如果此队列为空，则返回 null。

boolean remove(Object o)从此队列中移除指定元素的单个实例（如果存在）。

int size()返回此 collection 中的元素数。

Object[] toArray()返回一个包含此队列所有元素的数组。

<T> T[] toArray(T[] a)返回一个包含此队列所有元素的数组；返回数组的运行时类型是指定数组的类型。

17.7.2 双向队列

双向队列可以在任何一端添加或移除元素。Java 没有显示的用于双向队列的接口。可以通过组合创建一个双向接口类，并暴露 LinkedList 相关的方法。

17.8 理解 Map

映射表（也称为关联数组）的基本思想是它维护的是键-值关联，因此可以通过使用键来查找值。

Map 接口：

void clear()从此映射中移除所有映射关系（可选操作）。

boolean containsKey(Object key)如果此映射包含指定键的映射关系，则返回 true。

boolean containsValue(Object value)如果此映射将一个或多个键映射到指定值，则返回 true。

Set<Map.Entry<K,V>> **entrySet**()返回此映射中包含的映射关系的 Set 视图。

boolean equals(Object o)比较指定的对象与此映射是否相等。

V **get**(Object key)返回指定键所映射的值；如果此映射不包含该键的映射关系，则返回 null。

int hashCode()返回此映射的哈希码值。

boolean isEmpty()如果此映射未包含键-值映射关系，则返回 true。

Set<K> keySet()返回此映射中包含的键的 Set 视图。

V **put**(K key, V value)将指定的值与此映射中的指定键关联（可选操作）。

void putAll(Map<? extends K,? extends V> m)从指定映射中将所有映射关系复制到此映射中（可选操作）。

V remove(Object key)如果存在一个键的映射关系，则将其从此映射中移除（可选操作）。

int size()返回此映射中的键-值映射关系数。

Collection<V> values()返回此映射中包含的值的 Collection 视图。

示例是基于数组实现 Map，equals()比较键。

Map.Entry<K,V>接口：

boolean equals(Object o)比较指定对象与此项的相等性。

K getKey()返回与此项对应的键。

V getValue()返回与此项对应的值。

int hashCode()返回此映射项的哈希码值。

V setValue(V value)用指定的值替换与此项对应的值（可选操作）。

17.8.1 性能

HashMap 使用散列码来取代对键的缓慢搜索。散列码是“相对唯一”的、用以代表对象的 int 值，通过对象的某些信息进行转换而生成的。hashCode()是根类 Object 中的方法。

HashMap*	基于散列实现(取代 Hashtable)。插入和查询“键值对”的开销是固定的，可以通过构造器设置容量和负载因子，以调整容器的性能
LinkedHashMap	类似于 HashMap，但迭代遍历时，取得“键值对”的顺序是其插入顺序，或者是最近最少使用（LRU）的次序。只比 HashMap 慢一点，而在迭代访问时反而更快，因为它使用链表维护内部次序。
TreeMap	基于红黑树的实现。查看“键”或“键值对”时，它们会排序（次序由 Comparable 或 Comparator 决定）。是唯一带有 subMap()方法的 Map，返回一个子树。
WeakHashMap	弱键（weak key）映射，允许释放映射所指向的对象；这是为解决某些特殊问题而设计的，如果映射之外没有引用指向某个“键”，则此“键”可以被垃圾收集器回收。
ConcurrentHashMap	一种线程安全的 Map，不涉及同步加锁。
IdentityHashMap	使用==代替 equals()对“键”进行比较的散列映射。

对 Map 中使用的键的要求与对 Set 中的元素的要求一样，任何键必须具有一个 equals()方法，散列 Map 必须具有 hashCode()方法；TreeMap 必须实现 Comparable 接口。

17.8.2 SortedMap

使用 SortedSet，可以确保键处于排序状态。

SortedSet 接口：

Comparator<? super K> comparator()返回对此映射中的键进行排序的比较器；如果此映射使用键的自然顺序，则返回 null。

Set<Map.Entry<K,V>> entrySet()返回在此映射中包含的映射关系的 Set 视图。

K firstKey()返回此映射中当前第一个（最低）键。

SortedMap<K,V> headMap(K toKey)返回此映射的部分视图，其键值严格小于 toKey。

Set<K> keySet()返回在此映射中所包含键的 Set 视图。

K lastKey()返回映射中当前最后一个（最高）键。

SortedMap<K,V> subMap(K fromKey, K toKey)返回此映射的部分视图，其键值的范围从 fromKey（包括）到 toKey（不包括）。

SortedMap<K,V> tailMap(K fromKey)返回此映射的部分视图，其键大于等于 fromKey。

Collection<V> values()返回在此映射中所包含值的 Collection 视图。

17.8.3 LinkedHashMap

LinkedHashMap()构造一个带默认初始容量 (16) 和加载因子 (0.75) 的空插入顺序 LinkedHashMap 实例。

LinkedHashMap(int initialCapacity)构造一个带指定初始容量和默认加载因子 (0.75) 的空插入顺序 LinkedHashMap 实例。

LinkedHashMap(int initialCapacity, float loadFactor)构造一个带指定初始容量和加载因子的空插入顺序 LinkedHashMap 实例。

LinkedHashMap(int initialCapacity, float loadFactor, boolean accessOrder)构造一个带指定初始容量、加载因子和排序模式的空 LinkedHashMap 实例。

LinkedHashMap(Map<? extends K,? extends V> m)构造一个映射关系与指定映射相同的插入顺序 LinkedHashMap 实例。

void clear()从该映射中移除所有映射关系。

boolean containsValue(Object value)如果此映射将一个或多个键映射到指定值，则返回 true。

V get(Object key)返回此映射到指定键的值。

protected boolean removeEldestEntry(Map.Entry<K,V> eldest)如果此映射移除其最旧的条目，则返回 true。

17.9 散列与散列码

HashMap 使用 equals()判断当前键是否与表中存在的键相同，使用散列值进行查找。

正确的 equals()方法满足条件：

1、自反性。　　2、对称性　　3、传递性　　4、一致性

5、对任何不是 null 的 x，x.equals(null)一定为 false

17.9.1 理解 hashCode()

如果不为键覆盖 hashCode()和 equals()，那么使用散列的数据结构（HashSet、HashMap、LinkedHashSet 或 LinkedHashMap）就无法正确处理键。

使用散列目的在于：想要使用一个对象查找另一个对象。

示例使用一对 ArrayLists 实现了一个 Map，继承 AbstractMap。必须实现 entrySet()方法，必须产生一个 Map.Entry 对象集。因此创建自己的 Map 类型，就必须同时定义 Map.Entry 的实现。

17.9.2 为速度而散列

散列的价值在于速度：散列使得查询得以快速进行。查询速度解决方案一是保持键的排序状态，然后使用 Collections.binarySearch()进行查询；二是使用散列。

散列使用使用数组存储键的信息，而不是键本身。通过键对象生成一个数字，将其作为数组的下标。这个数字就是散列码，由 hashCode()方法生成。

数组容量固定，

查询过程首先是计算散列码，然后使用散列码查询数组。如没有冲突，那就有了一个完美的散列函数（特例）。冲突由外部链接处理：数组并不直接保存值，而是保存值的 list。然后对 list 中的值使用 equals()方法进行线性的查询。

散列表中的“槽位”（slot）通常称为桶位（bucket），即散列表的数组命名为 bucket。为使散列分布均匀，桶的数量通常使用质数。

示例：LinkedList<MapEntry<K,V>>[] buckets = new LinkedList[SIZE];

17.9.3 覆盖 hashCode()

编写 hashCode()的意义：无法控制 bucket 数组的下标值产生。这个值依赖于 HashMap 对象的容量，而容量的改变与容器的充满程度和负载因子有关。hashCode()生成的结果，经过处理后成为桶位的下标。

设计 hashCode 最重要的因素：无论何时，对同一个对象调用 hashCode()都应该生成同样的值。另一个因素：好的 hashCode()产生分布均匀的散列码。

hashCode()指导：

域类型	计算
boolean	c = (f?0:1)
byte、char、short 和 int	c = (int)f
long	c = (int)(f^(f>>>32))
float	c = Float.floatToIntBits(f);
double	long l = Double.doubleToLongBits(f); c = (int)(l^(l>>>32));
Object 、其 equals()调用这个域的 equals()	c = f.hashCode();
数组	每个元素都应用上述规则

1、给 int 变量 result 赋予某个非零值常量，

2、为对象内每个有意义的域 f 计算出一个 int 散列码 c;

3、合并计算得到的散列码：result = 37 * result + c;

4、返回 result;

5、检查 hashCode()最后生成的结果，确保相同的对象有相同的散列码。

17.10 选择接口的不同实现

只有四种容器：Map、List、Set 和 Queue，如何选择每种接口的实现版本？

容器之间的区别在于由什么样的数据结构实现的。可以根据所需的行为来选择不同的接口实现。

17.10.1 性能测试框架

遵循典型的模板方法模式，覆盖每个特定测试的 Test.test()方法，其核心代码在一个单独的 Tester 类中，待测容器类型是泛型参数 C:

每个 Test 对象都存储了该测试的名字。当调用 test()方法时，必须给出待测容器，以及“信使”或“数据传输对象”(TestParam)，它们保存有用于该特定测试的各种参数（元素数量 size，迭代次数 loops)。

TestParam 包含静态方法 array()，第一个版本接受的是可变参数列表，第二个版本接受 String 数组，返回 TestParam 数组。

为了使用框架，需要将待测容器以及 Test 对象列表传递给 Tester.run()方法（这些都是重载的泛型便利方法，可以减少在使用它们时所必需的类型检查)。Tester.run()调用适当的重载构造器，然后调用 timedTest(),它会执行针对该容器的列表中的每一个测试。TimedTest()会使用 paramList 中的每一个 TestParam 对象进行重复测试。

17.10.2 对 List 的选择

对于背后有数组支撑的 List 和 ArrayList，无论列表大小如何，get 和 set 方法访问都快速一致；而对于 LinkedList，访问时间对于较大的列表将明显增加。

对于 LinkedList，iteradd、insert 和 remove 方法，无论列表尺寸如何变化，其代价大致相同。

最佳的做法是将 ArrayList 作为默认首选，只有需要使用额外的功能，或者当程序的性能因为经常从表中间进行插入和删除而变差的时候，采取选择 LinkedList。如果使用的是固定数量的元素，那么既可以选择背后有数组支持的 List，也可以选择数组。

17.10.3 微基准测试的危险

17.10.4 对 Set 的选择

HashSet 的性能基本上总是比 TreeSet 好，特别是在添加和查询元素时。TreeSet 存在的唯一原因是它可以维持元素的排序状态；因为其内部结构支持排序，并用因为迭代是我们更有可能执行的操作，所以，用 TreeSet 迭代通常比用 HashSet 要快。对插入操作，所以，用 TreeSet 迭代通常比 HashSet 要快。

对于插入操作，LinkedHashSet 比 HashSet 代价更高，是由于维护链表所带来的额外开销造成的。

17.10.5 对 Map 的选择

所有 Map 实现的插入操作都会随 Map 尺寸的变大而明显变慢。查找的代价通常比插入要小很多。

Hashtable 的性能大体上与 HashMap 相当。TreeMap 通常比 HashMap 要慢，是一种创建有序列表的方式。

LinkedHashMap 在插入时比 HashMap 慢一点，因为它维护散列数据结构的同时还要维护链表（以保证插入顺序）。

IdentityHashMap 使用==而不是 equals()来比较元素。

HashMap 的性能因子

容量：表中的桶位数。

初始容量：表在创建时所拥有的桶位数。HashMap 和 HashSet 都有允许指定初始容量的构造器。

尺寸：表中当前存储的项数。

负载因子：尺寸/容量。HashMap 和 HashSet 都具有允许指定负载因子的构造器，表示负载因子达到该负载因子的水平时，容器将自动增加其容量（桶位数），实现方式是使容量大致加倍，并重新将现有对象分布到新的桶位集中（称为再散列）

17.11 实用方法

java.util.Collections 类：

static <T> boolean addAll(Collection<? super T> c, T... elements)将所有指定元素添加到指定 collection 中。

static <T> Queue<T> asLifoQueue(Deque<T> deque)以后进先出 (Lifo) Queue 的形式返回某个 Deque 的视图。

static <T> int **binarySearch**(List<? extends Comparable<? super T>> list, T key)使用二分搜索法搜索指定列表，以获得指定对象。

static <T> int binarySearch(List<? extends T> list, T key, Comparator<? super T> c)使用二分搜索法搜索指定列表，以获得指定对象。

static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type)返回指定 collection 的一个动态类型安全视图。

static <E> List<E> checkedList(List<E> list, Class<E> type)返回指定列表的一个动态类型安全视图。

static <K,V> Map<K,V> checkedMap(Map<K,V> m, Class<K> keyType, Class<V> valueType)返回指定映射的一个动态类型安全视图。

static <E> Set<E> checkedSet(Set<E> s, Class<E> type)返回指定 set 的一个动态类型安全视图。

static <K,V> SortedMap<K,V> checkedSortedMap(SortedMap<K,V> m, Class<K> keyType, Class<V> valueType)返回指定有序映射的一个动态类型安全视图。

static <E> SortedSet<E> checkedSortedSet(SortedSet<E> s, Class<E> type)返回指定有序 set 的一个动态类型安全视图。

static <T> void **copy**(List<? super T> dest, List<? extends T> src)将所有元素从一个列表复制到另一个列表。

static boolean **disjoint**(Collection<?> c1, Collection<?> c2)如果两个指定 collection 中没有相同的元素，则返回 true。

static <T> List<T> emptyList()返回空的列表（不可变的）。

static <K,V> Map<K,V> emptyMap()返回空的映射（不可变的）。
static <T> Set<T> emptySet()返回空的 set（不可变的）。
static <T> Enumeration<T> **enumeration**(Collection<T> c)返回一个指定 collection 上的枚举。
static <T> void **fill**(List<? super T> list, T obj)使用指定元素替换指定列表中的所有元素。
static int **frequency**(Collection<?> c, Object o)返回指定 collection 中等于指定对象的元素数。
static int **indexOfSubList**(List<?> source, List<?> target)返回指定源列表中第一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。
static int **lastIndexOfSubList**(List<?> source, List<?> target)返回指定源列表中最后一次出现指定目标列表的起始位置；如果没有出现这样的列表，则返回 -1。
static <T> ArrayList<T> list(Enumeration<T> e)返回一个数组列表，它按返回顺序包含指定枚举返回的元素。
static <T extends Object & Comparable<? super T>> T **max**(Collection<? extends T> coll)根据元素的自然顺序，返回给定 collection 的最大元素。
static <T> T **max**(Collection<? extends T> coll, Comparator<? super T> comp)根据指定比较器产生的顺序，返回给定 collection 的最大元素。
static <T extends Object & Comparable<? super T>> T **min**(Collection<? extends T> coll)根据元素的自然顺序 返回给定 collection 的最小元素。
static <T> T **min**(Collection<? extends T> coll, Comparator<? super T> comp)根据指定比较器产生的顺序，返回给定 collection 的最小元素。
static <T> List<T> **nCopies**(int n, T o)返回由指定对象的 n 个副本组成的不可变列表。
static <E> Set<E> newSetFromMap(Map<E,Boolean> map)返回指定映射支持的 set。
static <T> boolean **replaceAll**(List<T> list, T oldVal, T newVal)使用另一个值替换列表中出现的所有某一指定值。
static void **reverse**(List<?> list)反转指定列表中元素的顺序。
static <T> Comparator<T> reverseOrder()返回一个比较器，它强行逆转实现了 Comparable 接口的对象 collection 的自然顺序。
static <T> Comparator<T> reverseOrder(Comparator<T> cmp)返回一个比较器，它强行逆转指定比较器的顺序。
static void **rotate**(List<?> list, int distance)根据指定的距离轮换指定列表中的元素。
static void **shuffle**(List<?> list)使用默认随机源对指定列表进行置换。
static void **shuffle**(List<?> list, Random rnd)使用指定的随机源对指定列表进行置换。
static <T> Set<T> **singleton**(T o)返回一个只包含指定对象的不可变 set。
static <T> List<T> **singletonList**(T o)返回一个只包含指定对象的不可变列表。
static <K,V> Map<K,V> **singletonMap**(K key, V value)返回一个不可变的映射，它只将指定键映射到指定值。
static <T extends Comparable<? super T>> void **sort**(List<T> list)根据元素的自然顺序 对指定列表按升序进行排序。
static <T> void **sort**(List<T> list, Comparator<? super T> c)根据指定比较器产生的顺序对指定列表进行排序。
static void **swap**(List<?> list, int i, int j)在指定列表的指定位置处交换元素。
static <T> Collection<T> **synchronizedCollection**(Collection<T> c)返回指定 collection 支持的同步（线程安全的）collection。
static <T> List<T> **synchronizedList**(List<T> list)返回指定列表支持的同步（线程安全的）列表。
static <K,V> Map<K,V> **synchronizedMap**(Map<K,V> m)返回由指定映射支持的同步（线程安全的）映射。
static <T> Set<T> **synchronizedSet**(Set<T> s)返回指定 set 支持的同步（线程安全的）set。
static <K,V> SortedMap<K,V> **synchronizedSortedMap**(SortedMap<K,V> m)返回指定有序映射支持的同步（线程安全的）有序映射。
static <T> SortedSet<T> **synchronizedSortedSet**(SortedSet<T> s)返回指定有序 set 支持的同步（线程安全的）有序 set。
static <T> Collection<T> **unmodifiableCollection**(Collection<? extends T> c)返回指定

collection 的不可修改视图。

static <T> List<T> **unmodifiableList**(List<? extends T> list)返回指定列表的不可修改视图。

static <K,V> Map<K,V> **unmodifiableMap**(Map<? extends K,? extends V> m)返回指定映射的不可修改视图。

static <T> Set<T> **unmodifiableSet**(Set<? extends T> s)返回指定 set 的不可修改视图。

static <K,V> SortedMap<K,V> **unmodifiableSortedMap**(SortedMap<K,? extends V> m)返回指定有序映射的不可修改视图。

static <T> SortedSet<T> **unmodifiableSortedSet**(SortedSet<T> s)返回指定有序 set 的不可修改视图。

17.11.1 List 的排序和查询

List 排序与查询所使用的方法与对象数组所实用的相应方法有相同的名字与语法，使用 Collections 的 sort()方法。如果使用 Comparator 进行排序，那么 binarySearch()必须使用相同的 Comparator。

17.11.2 设定 Collection 或 Map 为不可修改

对于特定类型的“不可修改的”方法的调用并不会产生编译时的检查，使用 Collections 的 unmodifiable***()方法。但是转换完成后，任何会改变容器内容的操作都会引起 UnsupportedOperationException 异常。

17.11.3 Collection 或 Map 的同步控制

语法与“不可修改的”方法相似。使用 Collectiongs 的 synchronized***()方法。

快速报错

Java 容器类快速报错机制，能够防止多个进程同时修改同一个容器的内容。在你迭代遍历某个容器的过程中，另一个进程介入，并且插入、删除或修改此容器内的某个对象。Java 容器类类库采用“快速报错”（fail-fast）机制：会探查容器上的任何除了你的进程所进行的操作以外的所有变化，一旦它发现其他进行修改了容器，就会立即抛出 ConcurrentModificationException 异常。

“快速报错”机制的工作原理：只需创建一个迭代器，然后向迭代器所指向的 Collection 添加点什么，程序运行时发生了异常。

17.12 持有引用（java.lang.ref）

第 18 章　Java I/O 系统

18.1 File 类

File(文件)类既能代表特定文件的名称，又能代表一个目录下的一组文件的名称。如果它指的是一个文件集，就可以对此集合调用 list()方法，返回一个字符数组。

18.1.1 目录列表器

File 类：

字段摘要：

static String pathSeparator 与系统有关的路径分隔符，为了方便，它被表示为一个字符串。

static char pathSeparatorChar 与系统有关的路径分隔符。

static String separator 与系统有关的默认名称分隔符，为了方便，它被表示为一个字符串。

static char separatorChar 与系统有关的默认名称分隔符。

构造方法摘要：

File(File parent, String child)根据 parent 抽象路径名和 child 路径名字符串创建一个新 File 实例。

File(String pathname)通过将给定路径名字符串转换为抽象路径名来创建一个新 File 实例。

File(String parent, String child)根据 parent 路径名字符串和 child 路径名字符串创建一个新 File 实例。

File(URI uri)通过将给定的 file: URI 转换为一个抽象路径名来创建一个新的 File 实例。

方法摘要：

boolean canExecute()测试应用程序是否可以执行此抽象路径名表示的文件。

boolean canRead()测试应用程序是否可以读取此抽象路径名表示的文件。

boolean canWrite()测试应用程序是否可以修改此抽象路径名表示的文件。

int compareTo(File pathname)按字母顺序比较两个抽象路径名。

boolean createNewFile()当且仅当不存在具有此抽象路径名指定名称的文件时，不可分地创建一个新的空文件。

static File createTempFile(String prefix, String suffix)在默认临时文件目录中创建一个空文件，使用给定前缀和后缀生成其名称。

static File createTempFile(String prefix, String suffix, File directory)在指定目录中创建一个新的空文件，使用给定的前缀和后缀字符串生成其名称。

boolean delete()除此抽象路径名表示的文件或目录。

void deleteOnExit()在虚拟机终止时，请求删除此抽象路径名表示的文件或目录。

boolean equals(Object obj)测试此抽象路径名与给定对象是否相等。

boolean exists()测试此抽象路径名表示的文件或目录是否存在。

File getAbsoluteFile()返回此抽象路径名的绝对路径名形式。

String getAbsolutePath()返回此抽象路径名的绝对路径名字符串。

File getCanonicalFile()返回此抽象路径名的规范形式。

String getCanonicalPath()返回此抽象路径名的规范路径名字符串。

long getFreeSpace()返回此抽象路径名指定的分区中未分配的字节数。

String getName()返回由此抽象路径名表示的文件或目录的名称。

String getParent()返回此抽象路径名父目录的路径名字符串；如果此路径名没有指定父目录，则返回 null。

File getParentFile()返回此抽象路径名父目录的抽象路径名；如果此路径名没有指定父目录，则返回 null。

String getPath()将此抽象路径名转换为一个路径名字符串。

long getTotalSpace()返回此抽象路径名指定的分区大小。

long getUsableSpace()返回此抽象路径名指定的分区上可用于此虚拟机的字节数。

int hashCode()计算此抽象路径名的哈希码。

boolean isAbsolute()测试此抽象路径名是否为绝对路径名。

boolean isDirectory()测试此抽象路径名表示的文件是否是一个目录。

boolean isFile()测试此抽象路径名表示的文件是否是一个标准文件。

boolean isHidden()测试此抽象路径名指定的文件是否是一个隐藏文件。

long lastModified()返回此抽象路径名表示的文件最后一次被修改的时间。

long length()返回由此抽象路径名表示的文件的长度。

String[] list()返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中的文件和目录。

String[] list(FilenameFilter filter)返回一个字符串数组，这些字符串指定此抽象路径名表示的目录中满足指定过滤器的文件和目录。

File[] listFiles()返回一个抽象路径名数组，这些路径名表示此抽象路径名表示的目录中的文件。

File[] listFiles(FileFilter filter)返回抽象路径名数组，这些路径名表示此抽象路径名表示的目录中满足指定过滤器的文件和目录。

File[] listFiles(FilenameFilter filter)返回抽象路径名数组，这些路径名表示此抽象路径名表示的目录中满足指定过滤器的文件和目录。

static File[] listRoots()列出可用的文件系统根。

boolean mkdir()创建此抽象路径名指定的目录。

boolean mkdirs()创建此抽象路径名指定的目录，包括所有必需但不存在的父目录。

boolean renameTo(File dest)重新命名此抽象路径名表示的文件。

boolean setExecutable(boolean executable)设置此抽象路径名所有者执行权限的一个便捷方法。

boolean setExecutable(boolean executable, boolean ownerOnly)设置此抽象路径名的所有者或所有用户的执行权限。

boolean setLastModified(long time)设置此抽象路径名指定的文件或目录的最后一次修改时间。

boolean setReadable(boolean readable)设置此抽象路径名所有者读权限的一个便捷方法。

boolean setReadable(boolean readable, boolean ownerOnly)设置此抽象路径名的所有者或所有用户的读权限。

boolean setReadOnly()标记此抽象路径名指定的文件或目录，从而只能对其进行读操作。

boolean setWritable(boolean writable)设置此抽象路径名所有者写权限的一个便捷方法。

boolean setWritable(boolean writable, boolean ownerOnly)设置此抽象路径名的所有者或所有用户的写权限。

String toString()返回此抽象路径名的路径名字符串。

URI toURI()构造一个表示此抽象路径名的 file: URI。

FilenameFilter 接口：

boolean accept(File dir, String name)测试指定文件是否应该包含在某一文件列表中。

FileFilter 接口：

boolean accept(File pathname)测试指定抽象路径名是否应该包含在某个路径名列表中。

18.1.2 目录实用工具

通过方法产生给定目录下的由整个目录树中所有文件构成的List<File>(File 对象比文件名更有用，因为File 对象包含更多信息)

18.1.3 目录的检查及创建

18.2 输入和输出

流代表任何有能力产出数据的数据源对象或是有能力接收数据的接收端对象。

输入流：InputStream 或Reader 派生，含有read()基本方法

输出流：OutputStream 或Writer 派生，含有write()基本方法

装饰器涉及模式：叠合多个对象来提供所期望的功能。

18.2.1 InputStream 类型

InputStream 的作用是用来表示那些从不同数据源产生输入的类。

数据源包括：字节数组、String 对象、文件、管道、一个由其他种类的流组成的序列、其他数据源（Internet 连接）

InputStream 类型

类	功能	构造器参数
		任何使用

ByteArray InputStream	允许将内存的缓冲区当作InputStream使用	缓冲区，字节将从中取出 作为一种数据源：将其与FilterInputStream对象相连以提供有用接口
StringBuffer InputStream	将String转换成InputStream(已过时)	字符串，底层实现实际使用StringBuffer 作为一种数据源：将其与FilterInputStream对象相连以提供有用接口
File InputStream	用于从文件中读取信息	字符串，表示文件名、文件或FileDescriptor 作为一种数据源：将其与FilterInputStream对象相连以提供有用接口
Piped InputStream	产生用于写入相关Piped-OutputStream的数据	PipedOutputStream 作为多线程中数据源：将其与FilterInput-Stream对象相连以提供有用接口
Sequence InputStream	将两个或多个InputStream对象转换成单一InputStream	两个InputStream对象或一个容纳Input-Stream对象的容器Enumeration 作为一种资源：将其与FilterInputStream对象相连以提供有用接口

18.2.2 OutputStream类型

OutputStream类型

类	功能	构造器参数 任何使用
ByteArray OutputStream	在内存中创建缓冲区。所有送往“流”的数据都要放置在此缓冲区。	缓冲区初始化尺寸（可选的） 用于指定数据的目的地：将其与Filter-OutputStream对象相连以提供有用接口
File OutputStream	用于将信息写至文件	字符串，表示文件名、文件或FileDescriptor 指定数据的目的地：将其与FilterOutputStream对象相连以提供有用接口
Piped OutputStream	任何写入其中的信息都会自动作为相关PipedInput-Stream的输入	PipedOutputStream 指定用于多线程的数据目的地：将其与Fileter-OutputStream对象相连以提供有用接口

18.3 添加属性和有用的接口

Java I/O类库使用装饰器模式实现多种不同功能的组合。抽象类filter是所有装饰器类的基类。装饰器必须具有和它所装饰的对象相同的接口，也可以扩展接口，而这种情况只

发生在个别 filter 类中。

装饰器类优点：提供了相当多的灵活性，缺点：增加了代码的复杂性。不便在于必须创建许多类——“核心”I/O 类型加上所有的装饰器，才能得到所希望的单个 I/O 对象。

FilterInputStream 和 FilterOutputStream 是用来提供装饰器类接口以控制特定特定输入流（InputStream）和输出流（OutputStream）的两个类。

18.3.1 通过 FilterInputStream 从 InputStream 读取数据

FilterInputStream 类型

类	功能	构造器参数 任何使用
Data InputStream	与 DataOutputStream 搭配使用，因此可以按照可移植方式从流读取基本数据类型（int、char、long 等）	InputStream 包含用于读取基本类型数据的全部接口
Buffered InputStream	使用它可以防止每次读取时都得进行实际写操作。代表“使用缓冲区”	InputStream，可以指定缓冲区大小（可选） 本质上不提供接口，只不过是向进程中添加缓冲区所必需的。与接口对象搭配
LineNumber InputStream	根据输入流中的行号；可以调用 getLineNumber()和 setLineNumber(int)	InputStream 仅增加了行号，因此可能要与接口对象搭配使用
Pushback InputStream	具有“能弹出一个字节的缓冲区”。因此可以将读到的最后一个字符回退	InputStream 通常作为编译器的扫描器，之所以包含在内是因为 Java 编译器的需要，我们可能永远用不到

18.3.2 通过 FilterOutputStream 从 OutputStream 读取数据

FilterOutputStream 类型

类	功能	构造器参数 任何使用
Data OutputStream	与 DataOutputStream 搭配使用，因此可以按照可移植方式向流中写入基本类型数据（int、char、long 等）	OutputStream 包含用于写入基本类型数据的全部接口
PrintStream	用于产生格式化输出。其中 DataOutputStream 处理数据的存储，PrintStream 处理显示	OutputStream，可以用 boolean 值指示是否在每次换行时清空缓冲区（可选的）应该是对 OutputStream 对象的“final”封装。可能会经常使用到它

Buffered OutputStream	使用它避免每次发送数据都要进行实际的写操作。代表“使用缓冲区”，可以调用 flush()清空缓冲区	OutputStream，可以指定缓冲区大小（可选） 本质上并不是提供接口，只不过是向进程中添加缓冲区所必需的。与接口对象搭配

18.4 Reader 和 Writer

Reader 和 Writer 提供兼容 Unicode 与面向字符的 I/O 功能。而不是替代 InputStream 和 OutputStream。

InputStreamReader、OutputStreamReader 作为“适配器”(adapter)类，将 InputStream 转换为 Reader，OutputStream 转换为 Writer。

18.4.1 数据的来源和去处

尽量尝试使用 Reader 和 Writer，一旦程序无法成功编译，再使用面向字节的类库。

来源与去处：Java 1.0 类	相应的 Java 1.1 类
InputStream	Reader 适配器：InputStreamReader
OutputStream	Writer 适配器：OutputStreamWriter
FileInputStream	FileReader
FileOutputStream	FileWriter
StringBufferedInputStream(已弃用)	StringReader
（无相应的类）	StringWriter
ByteArrayInputStream	CharArrayReader
ByteArrayOutputStream	CharArrayWriter
PipedInputStream	PipedReader
PipedOutputStream	PipedWriter

18.4.2 更改流的行为

FilterWriter 和 FilterReader 作为一个占位符，没有任何子类。

过滤器：Java 1.0 类	相应的 Java 1.1 类
FilterInputStream	FilterReader
FilterOutputStream	FilterWriter（抽象类，没有子类）
BufferedInputStream	BufferedReader（也有 readLine()）
BufferedOutputStream	BufferedWriter
DataInputStream	使用 DataInputStream(除了当需要使用 readLine()时以外，这时应该使用 BufferedReader)
PrintStream	PrintWriter
LineNumberInputStream(已弃用)	LineNumberReader
StreamTokenizer	StreamTokenizer（使用接受 Reader 的构造器）
PushbackInputStream	PushbackReader

无论何时使用 readLine()，都不应使用 DataInputStream（已过时，编译器强烈反对），而应该使用 BufferedReader。除了这一点，DataInputStream 仍是 I/O 类库的首选成员。

PrintWriter 提供了一个既能接受 Writer 对象又能接受任何 OutputStream 对象的构造器，PrintWriter 的格式化接口实际上与 PrintStream 相同。PrintWriter 构造器提供“自动

执行清空”选项，如果设置此选项，则在每次执行 println()之后，便会自动清空。

18.5 自我独立的类：RandomAccessFile

RandomAccessFile 适用于由大小已知的记录组成的文件，因此可以使用 seek()将记录从一处转移到另一处，然后读取或者修改记录。文件中记录的大小不一定都相同，只要我们能够确定那些记录有多大以及他们在文件中的位置即可。

实现 DataInput 和 DataOutput 接口，工作方式类似于把 DataInputStream 和 DataOutputStream 组合起来使用，还添加了一些方法。

RamdomAccessFile 类：

构造方法摘要

RandomAccessFile(File file, String mode)创建从中读取和向其中写入（可选）的随机访问文件流，该文件由 File 参数指定。

RandomAccessFile(String name, **String mode**)创建从中读取和向其中写入（可选）的随机访问文件流，该文件具有指定名称。

方法摘要

void close()关闭此随机访问文件流并释放与该流关联的所有系统资源。

FileChannel getChannel()返回与此文件关联的唯一 FileChannel 对象。

FileDescriptor getFD()返回与此流关联的不透明文件描述符对象。

long **getFilePointer**()返回此文件中的当前偏移量。

long **length**()返回此文件的长度。

int read()从此文件中读取一个数据字节。

int **read**(byte[] b)将最多 b.length 个数据字节从此文件读入 byte 数组。

int read(byte[] b, int off, int len)将最多 len 个数据字节从此文件读入 byte 数组。

boolean readBoolean()从此文件读取一个 boolean。

byte readByte()从此文件读取一个有符号的八位值。

char readChar()从此文件读取一个字符。

double readDouble()从此文件读取一个 double。

float readFloat()从此文件读取一个 float。

void readFully(byte[] b)将 b.length 个字节从此文件读入 byte 数组，并从当前文件指针开始。

void readFully(byte[] b, int off, int len)将正好 len 个字节从此文件读入 byte 数组，并从当前文件指针开始。

int readInt()从此文件读取一个有符号的 32 位整数。

String readLine()从此文件读取文本的下一行。

long readLong()从此文件读取一个有符号的 64 位整数。

short readShort()从此文件读取一个有符号的 16 位数。

int readUnsignedByte()从此文件读取一个无符号的八位数。

int readUnsignedShort()从此文件读取一个无符号的 16 位数。

String readUTF()从此文件读取一个字符串。

void **seek**(long pos)设置到此文件开头测量到的文件指针偏移量，在该位置发生下一个读取或写入操作。

void setLength(long newLength)设置此文件的长度。

int skipBytes(int n)尝试跳过输入的 n 个字节以丢弃跳过的字节。

void **write**(byte[] b)将 b.length 个字节从指定 byte 数组写入到此文件，并从当前文件指针开始。

void write(byte[] b, int off, int len)将 len 个字节从指定 byte 数组写入到此文件，并从偏移量 off 处开始。

void write(int b)向此文件写入指定的字节。

void writeBoolean(boolean v)按单字节值将 boolean 写入该文件。

void writeByte(int v)按单字节值将 byte 写入该文件。

void writeBytes(String s)按字节序列将该字符串写入该文件。

void writeChar(int v)按双字节值将 char 写入该文件，先写高字节。

void writeChars(String s)按字符序列将一个字符串写入该文件。

void writeDouble(double v)使用 Double 类中的 doubleToLongBits 方法将双精度参数转换为一个

long，然后按八字节数量将该 long 值写入该文件，先定高字节。

void writeFloat(float v)使用 Float 类中的 floatToIntBits 方法将浮点参数转换为一个 int，然后按四字节数量将该 int 值写入该文件，先写高字节。

void writeInt(int v)按四个字节将 int 写入该文件，先写高字节。

void writeLong(long v)按八个字节将 long 写入该文件，先写高字节。

void writeShort(int v)按两个字节将 short 写入该文件，先写高字节。

void writeUTF(String str)使用 modified UTF-8 编码以与机器无关的方式将一个字符串写入该文件。

只有 RandomAccessFile 支持搜寻方法，并且只适用于文件。BufferedInputStream 却能允许标注（mark()）位置（其值存储于内部某个简单变量内）和重新设定位置（reset()）

18.6 I/O 流的典型使用方式

18.6.1 缓冲输入文件（BufferedInputFile）

```
BufferedReader in = new BufferedReader(new FileReader(filename));
in.readLine();   //删除换行符
```

18.6.2 从内存输入

```
StringReader in = new StringReader(BufferedInputFile.read(“MemInput.java”));
in.read();    //以 int 形式返回下一字节，必须转型为 char 才能打印
```

18.6.3 格式化的内存输入

```
DataInputStream in = new DataInputStream(
   new ByteArrayInputStream(
      BufferedInputFile.read(“FormattedMemoryInput.java”).getBytes()));
(char)in.readByte();
in.available();  //可以不受阻塞地从此输入流中读取（或跳过）的估计字节数
```

18.6.4 基本的文件输出

```
BufferedReader in = new BufferedReader(
   new StringReader(BufferedInputFile.read(“BasicFileOutput.java”)));
PrintWriter out = new PrintWriter(
   new BufferedWriter(new FileWriter(file)));
```

18.6.5 存储和恢复数据

```
DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(new FileOutputStream(“Data.txt”)));
out.writeDouble(3.14159);
DataInputStream in = new DataInputStream(
   new BufferedInputStream(new FileInputStream(“Data.txt”)));
in.readDouble();
```

DataOutputStream 写入数据，DataInputStream 恢复数据

字符串写入、读出使用 UTF-8 编码，用 writeUTF()和 writeUTF()来实现，如果使用非 Java 程序读取用 writeUTF 所写的字符串时，必须编写特殊代码才能正确读取字符串。

必须：要么为文件中的数据采用固定的格式；要么将额外的信息保存到文件中，以便能够对其进行解析以确定数据的存放位置。

18.6.6 读取随机访问文件

```
RandomAccessFile rf = new RandomAccessFile(file,”r”);
rf.readDouble();
rf = new RandomAccessFile(file,”rw”);
rf.seek(5 * 8);
rf.writeDouble(47.0001);
```

RandomAccessFile 类似于组合使用了 DataInputStream 和 DataOutputStream，seek()

可以在文件中到处移动，并**修改**文件中的某个值。

18.6.7 管道流

管道流用于任务之间的通信。

18.7 文件读写的实用工具

一个常见的程序化任务就是读取文件，修改，然后再写出。Java I/O 类库的问题之一就是：编写相当多代码执行这些常用操作——没有任何基本的帮助功能可以为我们做这一切。装饰器会使得要记住如何打开文件变成一件相当困难的事。

18.7.1 读取二进制文件

```
BufferedInputStream bf = new BufferedInputStream(
   new FileInputStream(bFile));
byte[] data = new byte[bf.available()];
bf.read(data);
```

18.8 标准 I/O

18.8.1 从标准输入中读取

Java 提供了 System.in（是一个没有被包装过的未经加工的 InputStream）、System.out（事先包装成了 PrintStream）、System.err（同样是 PrintStream）。这意味着在读取 System.in 之前必须对其进行包装。

```
BufferedReader stdin = new BufferedReader(
   new InputStreamReader(System.in));
String s = stdin.readLine();
```

18.8.2 将 System.out 转换成 PrintWriter

```
PrintWriter out = new PrintWriter(System.out,true);
```

18.8.3 标准 I/O 重定向

Java 的 System 类提供了一些简单的静态方法调用，以允许对标准输入、输出和错误 I/O 流进行重定向：setIn(InputStream)、setOut(PrintStream)、setErr(PrintStream)

I/O 重定向操纵的是字节流，而不是字符流。

18.9 进程控制

18.10 新 I/O

JDK 1.4 的 java.nio.*包中引入了新的 Java I/O 类库，使用：**通道**和**缓冲器**，目的在于提高速度。

通道要么从缓冲器获得数据，要么向缓冲器发送数据。**唯一直接**和通道交互的缓冲器是 ByteBuffer——也就是说，可以存储未加工字节的缓冲器。

旧 I/O 类库中 FileInputStream、FileOutputStream、RandomAccessFile 被修改用以产生 FileChannel。

```
FileChannel fc = new FileOutputStream(“data.txt”).getChannel();
fc.write(ByteBuffer.wrap(“Some text”).getBytes());
fc.close();
fc = new FileInputStream(“data.txt”).getChannel();
ByteBuffer buff = ByteBuffer.allocate(BSIZE);
fc.read(buff);
buff.flip(); //反转此缓冲区。首先将限制设置为当前位置，然后将位置设置为 0。
while(buff.hasRemaining())
   System.out.print((char)buff.get());
```

通道是一种相当基础的东西：可以向它传送用于读写的 ByteBuffer，并且可以锁定文件的某些区域用于独占式访问。

allocateDirect()、flip()、clear()

transferTo()、transferForm()将一个通道与另一个通道相连。

18.10.1 转换数据

缓冲器容纳的是普通的字节，为了把它们转换成字符，要么在输入的时候对其进行**编码**，要么将其从缓冲器输出时对它们进行**解码**（使用 java.nio.charset.Charset 类实现这些功能）。

Charset 类：16 位的 Unicode 代码单元序列和字节序列之间的指定映射关系。此类定义了用于创建解码器和编码器以及获取与 charset 关联的各种名称的方法。此类的实例是不可变的。

一是使用 System.getProperty("file.encoding")发现默认字符集，会产生代表字符集名称的字符串，把该字符串传送给 Charset.forName()用以产生 Charset 对象，调用 decode(buff)方法对字符串进行解码。

二是在读文件时，使用能够产生可打印的输出的字符集进行 encode()，ByteBuffer.wrap("Some text".getBytes("UTF-16BE"))，读取时只需把它转换成 CharBuffer（buff.asCharBuffer）即可。

三是通过 CharBuffer 向 ByteBuffer **写入**(buff.asCharBuffer().put("Some text"))，剩余的内容为零的字节仍出现在由它的 toString()所产生的 CharBuffer 的表示中，

18.10.2 获取基本类型

向 ByteBuffer 插入基本类型数据的最简单方法是：利用 asCharBuffer()、asShortBuffer()等获得该缓冲器上的视图，然后使用视图的 put()方法，此方法适用于所有基本数据类型。

18.10.3 视图缓冲器

试图缓冲器（view buffer）可以通过某个特定的基本数据类型的视窗查看其底层的 ByteBuffer。ByteBuffer 依然是实际存储数据的地方，"支持"着前面的视图。对视图的任何修改都会**映射**成为对 ByteBuffer 插入数据。视图还允许从 ByteBuffer 一次一个地或者成批地（放入数组中）读取基本类型值。

```
ByteBuffer bb = ByteBuffer.allocate(BSIZE);
IntBuffer ib = bb.asIntBuffer();
ib.put(new int[]{11,42,47,99,143,811,1016});
ib.put(3,1811);  //替换
```

Buffer 类：一个用于特定基本类型数据的容器。缓冲区是特定基本类型元素的线性有限序列。除内容外，缓冲区的基本属性还包括容量、限制和位置：缓冲区的容量 是它所包含的元素的数量。缓冲区的容量不能为负并且不能更改。 缓冲区的限制 是第一个不应该读取或写入的元素的索引。缓冲区的限制不能为负，并且不能大于其容量。缓冲区的位置 是下一个要读取或写入的元素的索引。缓冲区的位置不能为负，并且不能大于其限制。

方法摘要

abstract Object array()返回此缓冲区的底层实现数组（可选操作）。

abstract int arrayOffset()返回此缓冲区的底层实现数组中第一个缓冲区元素的偏移量（可选操作）。

int **capacity**()返回此缓冲区的容量。

Buffer **clear**()清除此缓冲区。将 position 设置为 0.

Buffer **flip**()反转此缓冲区。将 limit 设置为 position，position 设置为 0.

abstract boolean hasArray()告知此缓冲区是否具有可访问的底层实现数组。

boolean **hasRemaining**()告知在当前位置和限制之间是否有元素。

abstract boolean isDirect()告知此缓冲区是否为直接缓冲区。

abstract boolean isReadOnly()告知此缓冲区是否为只读缓冲区。

int **limit**()返回此缓冲区的限制。

Buffer **limit**(int newLimit)设置此缓冲区的限制。

Buffer **mark**()在此缓冲区的位置设置标记。将 mark 设置为 position 值
int **position**()返回此缓冲区的位置。
Buffer **position**(int newPosition)设置此缓冲区的位置。
int **remaining**()返回当前位置与限制之间的元素数。
Buffer **reset**()将此缓冲区的位置重置为以前标记的位置。将 position 设置为 mark 值
Buffer **rewind**()重绕此缓冲区。

ByteBuffer 类：字节缓冲区。

方法摘要

static ByteBuffer **allocate**(int capacity)分配一个新的字节缓冲区。
static ByteBuffer **allocateDirect**(int capacity)分配新的直接字节缓冲区。
byte[] **array**()返回实现此缓冲区的 byte 数组（可选操作）。
int arrayOffset()返回此缓冲区中的第一个元素在缓冲区的底层实现数组中的偏移量（可选操作）。
abstract CharBuffer asCharBuffer()创建此字节缓冲区的视图，作为 char 缓冲区。
abstract DoubleBuffer asDoubleBuffer()创建此字节缓冲区的视图，作为 double 缓冲区。
abstract FloatBuffer asFloatBuffer()创建此字节缓冲区的视图，作为 float 缓冲区。
abstract IntBuffer asIntBuffer()创建此字节缓冲区的视图，作为 int 缓冲区。
abstract LongBuffer asLongBuffer()创建此字节缓冲区的视图，作为 long 缓冲区。
abstract ByteBuffer asReadOnlyBuffer()创建共享此缓冲区内容的新的只读字节缓冲区。
abstract ShortBuffer asShortBuffer()创建此字节缓冲区的视图，作为 short 缓冲区。
abstract ByteBuffer compact()压缩此缓冲区（可选操作）。
int compareTo(ByteBuffer that)将此缓冲区与另一个缓冲区进行比较。
abstract ByteBuffer duplicate()创建共享此缓冲区内容的新的字节缓冲区。
boolean equals(Object ob)判断此缓冲区是否与另一个对象相同。
abstract byte get()相对 get 方法。
ByteBuffer get(byte[] dst)相对批量 get 方法。
ByteBuffer get(byte[] dst, int offset, int length)相对批量 get 方法。
abstract byte get(int index)绝对 get 方法。
abstract char getChar()用于读取 char 值的相对 get 方法。
abstract char getChar(int index)用于读取 char 值的绝对 get 方法。
abstract double getDouble()用于读取 double 值的相对 get 方法。
abstract double getDouble(int index)用于读取 double 值的绝对 get 方法。
abstract float getFloat()用于读取 float 值的相对 get 方法。
abstract float getFloat(int index)用于读取 float 值的绝对 get 方法。
abstract int getInt()用于读取 int 值的相对 get 方法。
abstract int getInt(int index)用于读取 int 值的绝对 get 方法。
abstract long getLong()用于读取 long 值的相对 get 方法。
abstract long getLong(int index)用于读取 long 值的绝对 get 方法。
abstract short getShort()用于读取 short 值的相对 get 方法。
abstract short getShort(int index)用于读取 short 值的绝对 get 方法。
boolean hasArray()判断是否可通过一个可访问的 byte 数组实现此缓冲区。
int hashCode()返回此缓冲区的当前哈希码。
Abstract boolean isDirect()判断此字节缓冲区是否为直接的。
ByteOrder **order**()获取此缓冲区的字节顺序。(**BIG_ENDIAN、LITTLE_ENDIAN**)
ByteBuffer **order**(ByteOrder bo)修改此缓冲区的字节顺序。
abstract ByteBuffer put(byte b)相对 put 方法（可选操作）。
ByteBuffer put(byte[] src)相对批量 put 方法（可选操作）。
ByteBuffer put(byte[] src, int offset, int length)相对批量 put 方法（可选操作）。
ByteBuffer put(ByteBuffer src)相对批量 put 方法（可选操作）。
abstract ByteBuffer put(int index, byte b)绝对 put 方法（可选操作）。
abstract ByteBuffer putChar(char value)用来写入 char 值的相对 put 方法（可选操作）。
abstract ByteBuffer putChar(int index, char value)用于写入 char 值的绝对 put 方法（可选操作）。

abstract ByteBuffer putDouble(double value)用于写入 double 值的相对 put 方法（可选操作）。

abstract ByteBuffer putDouble(int index, double value)用于写入 double 值的绝对 put 方法(可选操作)。

abstract ByteBuffer putFloat(float value)用于写入 float 值的相对 put 方法（可选操作）。

abstract ByteBuffer putFloat(int index, float value)用于写入 float 值的绝对 put 方法（可选操作）。

abstract ByteBuffer putInt(int value)用于写入 int 值的相对 put 方法（可选操作）。

abstract ByteBuffer putInt(int index, int value)用于写入 int 值的绝对 put 方法（可选操作）。

abstract ByteBuffer putLong(int index, long value)用于写入 long 值的绝对 put 方法（可选操作）。

abstract ByteBuffer putLong(long value)用于写入 long 值（可先操作） 的相对 put 方法。

abstract ByteBuffer putShort(int index, short value)用于写入 short 值的绝对 put 方法（可选操作）。

abstract ByteBuffer putShort(short value)用于写入 short 值的相对 put 方法（可选操作）。

abstract ByteBuffer **slice**()创建新的字节缓冲区，其内容是此缓冲区内容的**共享**子序列。(子缓冲区就像是父缓冲区的一个窗口，共享一部分底层数组位置，新缓冲区的位置将为零，其容量和界限将为此缓冲区中所剩余的字节数量，其标记是不确定的)

String toString()返回汇总了此缓冲区状态的字符串。

static ByteBuffer **wrap**(byte[] array)将 byte 数组包装到缓冲区中。

static ByteBuffer **wrap**(byte[] array, int offset, int length)将 byte 数组包装到缓冲区中。

字节存放序列

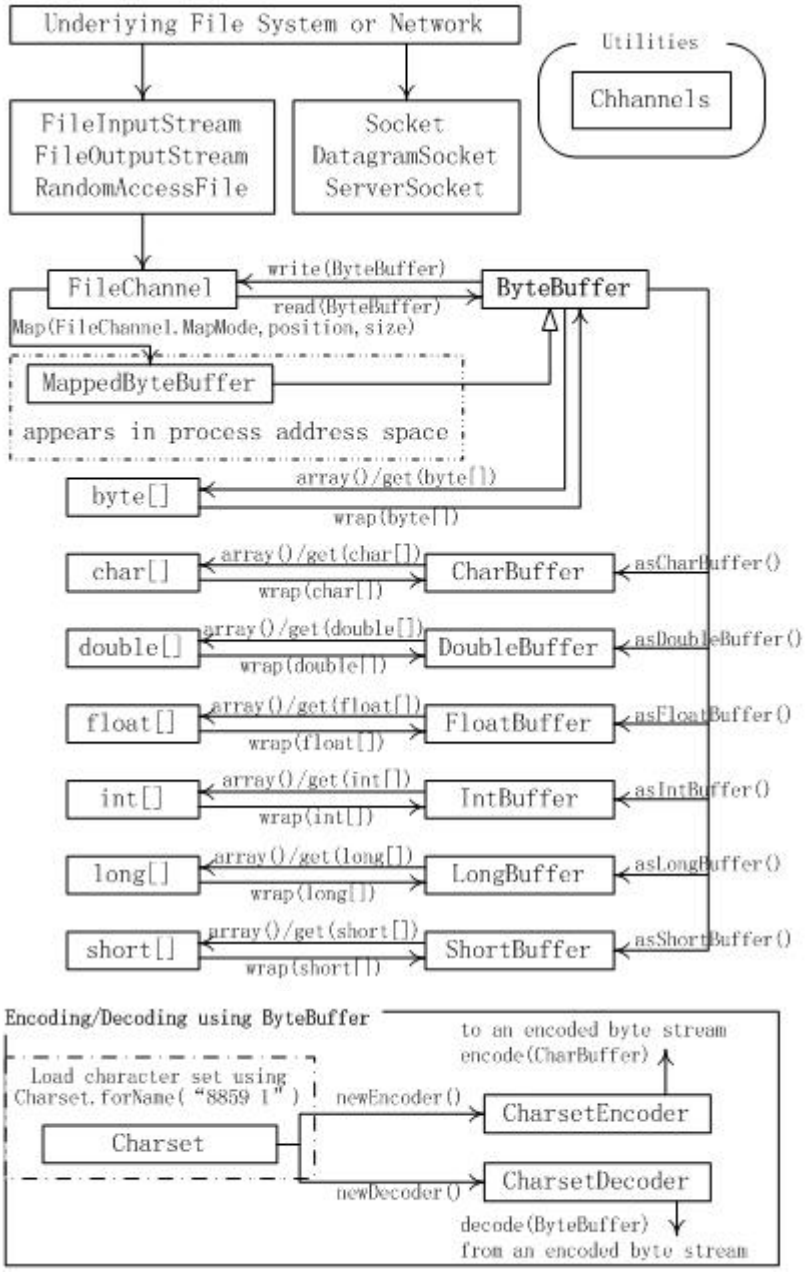
“big endian”(高位优先)将最重要的字节存放在地址最低的存储单元;“little endian”（低位优先）则是将最重要的字节放在地址最高的存储单元。ByteBuffer 默认以高位优先的形式存储数据，可以使用 order 方法改变字节排序方式。

18.10.4 用缓冲区操纵数据

ByteBuffer 是将数据移进通道的唯一方式，并且只能创建一个独立的基本类型缓冲器，或者使用“as”方法从 ByteBuffer 中获得。也就是说不能将基本类型的缓冲器转换为 ByteBuffer。然而，可以经由视图缓冲器将基本类型数据移进移出 ByteBuffer。

18.10.5 缓冲器的细节

Buffer 数据的四个索引：mark（标记）、position（位置）、limit（界限）和 capacity（容量）。



18.10.6 内存映射文件

内存映射文件允许创建和修改那些因为太大而不能放入内存的文件。有了内存映射文件就可以假定整个文件都放在内存中，而且可以完全把它当作非常大的**数组**来访问。

```
MappedByteBuffer out =
   new RandomAccessFile("test.dat", "rw").getChannel()
   .map(FileChannel.MapMode.READ_WRITE,0,length);
for(int i = 0; i < length; i++)
   out.put((byte)' x' );
```

MappedByteBuffer 由 ByteBuffer 继承而来，因此它具有 ByteBuffer 所有方法。

底层操作系统文件映射工具是用来最大化地提高性能。

性能

模板方法

18.10.7 文件加锁

JDK 1.4 引入了文件加锁机制，它允许**同步**访问某个作为共享资源的文件。Java 的文件

加锁**直接映射**到本地操作系统的加锁工具。

通过对FileChannel调用tryLock()或lock(),就可获得整个文件的FileLock。tryLock()是非阻塞式的，它设法获取锁，如果不能获得，将直接从方法调用返回。lock()是阻塞式的，阻塞进程直至锁可以获得。使用FIleLock.release()可以释放锁。

可以对文件的一部分上锁：

tyrLock(long position,long size,boolean shared)

或者

lock(long position,long size,boolean shared)

加锁区域由size-position决定，第三个参数指定是否是共享锁。

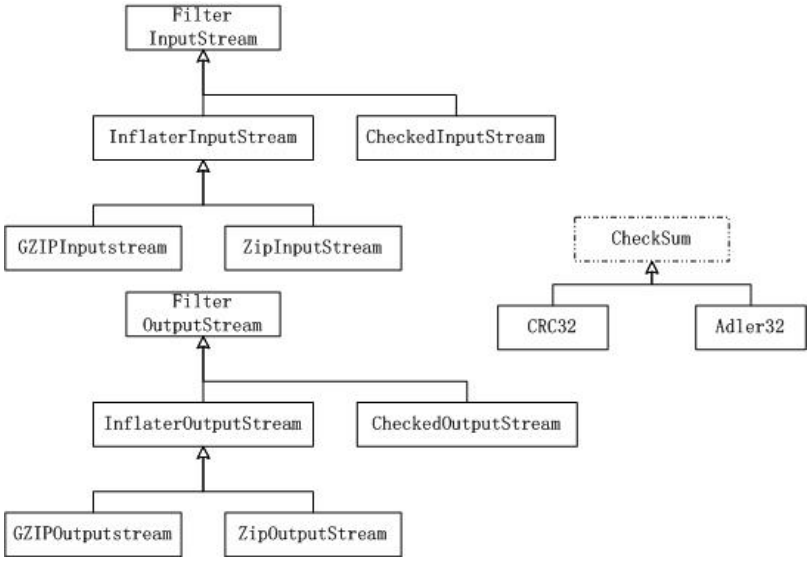
共享锁和独占锁

对映射文件的部分加锁

线程类LockAndModify创建了缓冲区和用于修改的slice()，然后在run()中，获得文件通道上的锁。lock()调用类似于获得一个对象的线程锁——处于“临界区”，即对该部分的文件具有独占访问权。

18.11 压缩（java.util.zip.*）

Java I/O类库中的类支持读写压缩格式的数据流。可以对其他的I/O类进行封装，以提供压缩功能。是属于InputStream和OutPutStream继承层次结构的一部分。



Zip和GZIP是最常用的

18.11.1 用GZIP进行简单压缩

GZIP接口简单，只对单个数据流（而不是一系列互异数据）进行压缩，比较合适。

```
BufferedOutputStream out = new BufferedOutputStream(
   new GZIPOutputStream(
      new FileOutputStream(“test.gz”)));
BufferedReader in2 = new BufferedReader(
   new InputStreamReader(new GZIPInputStream(
      new FileInputStream(“test.gz”))));
```

压缩类的使用——直接将输出流封装成GZIPOutputStream或ZipOutputStream，并将输入流封装成GZIPInputStream或ZipInputStream即可，其他全部操作就是通常的I/O读写。

18.11.2 用Zip进行文件保存

CheckedInputStream：需要维护所读取数据校验和的输入流。校验和可用于验证输入数据的完整性。

CheckedOutputStream：需要维护所读取数据校验和的输出流。校验和可用于验证输出数据的完整性。

GZIPInputStream：此类为读取 GZIP 文件格式的压缩数据实现流过滤器。

GZIPOutputStream：此类为使用 GZIP 文件格式写入压缩数据实现流过滤器。

ZipInputStream：此类为读取 ZIP 文件格式的文件实现输入流过滤器。包括对已压缩和未压缩条目的支持。

ZipOutputStream：此类为以 ZIP 文件格式写入文件实现输出流过滤器。包括对已压缩和未压缩条目的支持。

Adler32：可用于计算数据流的 Adler-32 校验和的类。Adler-32 校验和几乎与 CRC-32 一样可靠，但是能够更快地计算出来。(快一些)

CRC32：可用于计算数据流的 CRC-32 的类。(慢一些，但更准确)

ZipEntry：此类用于表示 ZIP 文件条目。

ZipFile：此类用于从 ZIP 文件读取条目。

```
FileOutputStream f = new FileOutputStream("test.zip");
CheckedOutputStream csum = new CheckedOutputStream(f,new Adler32());
ZipOutputStream zos = new ZipOutputStream(csum);
BufferedOutputStream out = new BufferedOutputStream(zos);
zos.setComment("A test of Java Zipping");
BufferedReader in = new BufferedReader(new FileReader(arg));
zos.putNextEntry(new ZipEntry(arg));
c = in.read();
out.write(c);

FileInputStream fi = new FileInputStream("test.zip");
CheckedInputStream csumi = new CheckedInputStream(fi,new Adler32());
ZipInputStream in2 = new ZipInputStream(csumi);
BufferedInputStream bis = new BufferedInputStream(in2);
ZipEntry ze;
while((ze = in2.getNextEntry()) != null) {
   print("Reading file " + ze);
   int x ;
   while((x = bis.read()) != -1)
       System.out.write(x);
}
```

对于要加入压缩档案的文件，都必须调用 putNextEntry()，并将其传递给一个 ZipEntry 对象。Java 的 Zip 类库不提供密码支持。CheckedInputStream 和 CheckedOutputStream 都支持 Adler32 和 CRC32 两种类型的校验和，但 ZipEntry 类只有一个支持 CRC 接口。

为了能够解压缩文件,ZipInputStream 提供了一个 getNextEntry()返回下一个 ZipEntry (如果存在的话)。解压缩文件有一个简便的方法——利用 ZipFile 对象读取文件。该对象有一个 entries()方法用来向 ZipEntries 返回一个 Enumeration (枚举)。

18.11.3 Java 档案文件

JAR 文件由一组压缩文件构成，同时还有一张描述了所有文件的“文件清单”(可自行创建文件清单，也可以由 jar 程序自动生成)

通过命令行的形式调用 jar，如下：

```
jar [options] destination [manifest] inputfile(s)
```

其中 options 只是一个字母集合(不必输入任何“-”或其它任何标识符，这些选项字符意义如下：

c	创建一个新的或空的压缩文档

t	列出目录表
x	解压所有文件
x file	解压该文件
f	意指:“我打算指定一个文件名。”如果没有用这个选项，jar 假设所有的输入都来自于标准输入；或者在创建一个文件时，输出对象也假设为标准输出
m	表示第一个参数将是用户自建的清单文件的名字
v	产生详细输出，描述 jar 所做的工作
o	只存储文件，不压缩文件
M	不自动创建文件清单

18.12 对象序列化

Java 的对象序列化将那些实现了 Serializable 接口的对象转换成一个字节序列，并能够在以后将这个字节序列完全恢复为原来的对象。

轻量级持久（lightweight persistence)。“持久性”意味着一个对象的生存周期并部取决于程序是否正在执行，它可以生存于程序的调用**之间**。之所以称为**“轻量级”**是因为不能用某种“persistent”（持久）关键字来简单地定义一个对象，并让系统自动维护其他细节问题，相反，对象必须在程序中显示的序列化（serialize）和反序列化还原（deserialize)。

对象序列化作用是为了支持两种主要特性：一是 Java 的远程方法调用（Remote Method Invocation，RMI)；二是支持 Java Beans。

序列化对象：首先创建某些 OutputStream 对象，然后将其封装在 ObjectOutputStream 对象内。只需调用 writeObject()即可将对象序列化。还原对象，需要将一个 InputStream 封装在 ObjectInputStream 内，然后调用 readObject()。

对象序列化保存对象的“全景图”，而且能跟踪对象内所包含的所有引用，并保存对象；又能对对象内包含的每个引用进行跟踪；建立“对象网”，因而尽量不要自己动手，让它用优化算法自动维护整个对象网。

```
ObjectOutputStream out = new
   ObjectOutputStream(new FileOutputStream("worm.out"));
ObjectInputStream in = new
   ObjectInputStream(new FileInputStream("Worm.out"));
```

序列化将对象读写到任何 InputStream 或 OutputStream，设置包括网络。

对一个 Serializable 对象进行还原的过程中，无需调用任何构造器，整个对象都是从 InputStream 中取得数据恢复而来的。

18.12.1 寻找类

必须保证 Java 虚拟机能找到相关的.class 文件，如果找不到，那么就会抛出一个 ClassNotFoundException 异常。

18.12.2 序列化的控制

通道实现 Externalizable 接口代替实现 Serializable 接口，对序列化过程进行控制。

恢复对象会**先**调用对象的默认构造器（如果构造不是公共的（public),那么就会在恢复时造成异常），**后**调用 readExternal()或 writeExternal()。

为了正常运行，不仅需要在 writeExternal()方法中将来自对象的重要信息写入，还必须在 readExternal()中恢复数据。

transient（瞬时）关键字

防止敏感部分被序列化，一是将类实现为 Externalizable(没有任何东西可以自动序列化)；二是用 transient（瞬时）关键字逐个字段**关闭**序列化（只能和 Serializable 对象一起使用)。

Externalizable 的替代方法

可以实现 Serializable 接口，并**添加**名为 writeObject()和 readObject()的**私有**方法。这样一旦对象被序列化或被反序列化还原,就会自动地分别调用这两个方法。必须是 private，因此不会是接口的一部分，因为必须完全遵循方法特征签名，所有效果和实现了接口一样，在方法内部可以调用 defaultWriteObject()和 defaultReadObject()来选择执行默认的 writeObject()和 readObject()。

```
private void writeObject(ObjectOutputStream stream)
throws IOException {
   stream.defaultWriteObject();
   stream.writeObject(b);
}
private void readObject(ObjectInputStream stream)
throws IOException,ClassNotFoundException {
   stream.defaultReadObject();
   b = (String)stream.readObject();
}
ByteArrayOutputStream buf = new ByteArrayOutputStream();
ObjectOutputStream o = new ObjectOutputStream(buf);
o.writeObject(sc);
```

writeObject 必须检查 sc，判断它是否拥有自己的 writeObject 方法，如果有，就使用它。

18.12.3 使用“持久性”

可以实现对任何可 Serializable 对象的“深度复制”（deep copy）——深度复制意味着复制是整个对象网，而不仅仅是基本对象及其引用。

只要将任何对象序列化到单一流中，就可以恢复出与我们写出时一样的对象网，并且没有任何意外重复复制出的对象。

如果向保存系统状态，最安全的做法是将其作为“原子”操作进行序列化，将构成系统状态的所有对象都置入单一容器内，并在一个操作中将该容器直接写出。

static 字段序列化

序列化 Class 对象无法自动序列化 static 字段。必须①在需要需要序列化 static 字段的类中添加 serializeStaticState()和 desserializeStaticState()方法；②调用序列化和反序列化还原静态方法。

```
public static void serializeStaticState(ObjectOutputStream os)
   throws IOException { os.writeInt(color); }
public static void deserializeStaticState(ObjectInputStream os)
   throws IOException { color = os.readInt(); }

Line.serializeStaticState(out);

Line.deserializeStaticState(in);
```

安全问题

序列化也会将 private 数据保存下来，如考虑安全问题，应将其标记成 transient，还必须设计一种安全的保存信息的方法，以便在执行恢复时可以复位 private 变量。

18.13 XML

将数据转换为 XML 格式，可以使其被各种平台和语言使用。

18.14 Preferences

Preferences API 可以自动存储和读取信息。不过它只能用于小的、受限的数据集合（只能存储基本类型和字符串），并且字符串长度不能超过 8K。Preferences API 用于存储和读取用户的偏好（preferences）以及程序配置项的设置。

```
Preferences prefs = Preferences
   .userNodeForPackage(PreferencesDemo.class);
int usageCount = prefs.getInt("UsageCount",0);
usageCount++;
prefs.putInt("UsageCount",usageCount);
```

userNodeForPackage()最好用于个别用户的偏好，systemNodeForPackage()用于通用的安装配置。PreferencesDemo.class 用来标识节点，但通常使用 getClass()方法。

一旦创建了节点，就可以用它加载或者读取数据了。向节点载入了各种不同类型的数据项，然后获取其 keys()，它是以 String[]的形式返回的，get()的第二个参数，如果某个关键字下没有任何条目，那么这个参数就是所产生的默认值。

第一次运行程序时，UsageCount 的值是 0，在随后引用中，它将是非零值，每次运行程序,UsageCount 的值都会增加 1,Preferences API 利用合适的系统资源完成这个任务(Windows 就使用注册表)

第 19 章　枚举类型

关键字 enum 可以将一组具名的值的有限集合创建为一种新的类型，而这些具名的值可以作为常规的程序组件使用。

19.1 基本 enum 特性

创建 enum，编译器会为你生成一个相关的类，这个类继承自 java.lang.Enum。

Enum<E extends Enum<E>>类：

构造方法摘要

protected Enum(String name, int ordinal)单独的构造方法。

方法摘要

protected Object clone()抛出 CloneNotSupportedException。

int compareTo(E o)比较此枚举与指定对象的顺序。

boolean equals(Object other)当指定对象等于此枚举常量时，返回 true。

protected void finalize()枚举类不能有 finalize 方法。

Class<E> getDeclaringClass()返回与此枚举常量的枚举类型相对应的 Class 对象。

int hashCode()返回枚举常量的哈希码。

String name()返回此枚举常量的名称，在其枚举声明中对其进行声明。

int ordinal()返回枚举常量的序数（它在枚举声明中的位置，其中初始常量序数为零）。

String toString()回枚举常量的名称，它包含在声明中。

static <T extends Enum<T>> T valueOf(Class<T> enumType, String name)返回带指定名称的指定枚举类型的枚举常量。

19.1.1 将静态导入用于 enum

使用 import static 能够将 enum 实例的标识符带入当前的命名空间，所以无需再用 enum 类型来修饰 enum 实例。

19.2 向 enum 中添加新方法

除了不能继承自一个 enum 之外，基本可以将 enum 看作一个常规的类。也就是说可以向 enum 中添加方法。

如自定义自己的方法，那么必须在 enum 实例序列的**最后添加一个分号**。Java 要求必须**先定义 enum 实例。**在定义 enum 实例之前定义了任何方法或属性，编译器就会得到错误信息。

enum 中构造器与方法和普通的类没有区别， enum 构造器声明为 private，因为只能在 enum 定义的内部使用其构造器创建 enum 实例。一旦 enum 的定义结束，编译器就不允许再使用其构造器来创建任何实例。

19.2.1 覆盖 enum 的方法

覆盖 enum 的方法与覆盖一般类的方法没有区别。

19.3 switch 语句中的 enum

枚举实例具备整数值的次序(通过 ordinal()获取次序),编译器对 switch 中没有 default 语句或去掉某个 case，并无告警。如 case 语句中调用 return，那么编译器就会警告没有 default 语句。

19.4 values()的神秘之处

编译器创建的 enum 类都继承自 Enum 类，然而 Enum 类并没有 values()方法。values()是由编译器添加的 static 方法，在创建 enum 的过程中，编译器还为其添加了 valueOf()（一个参数），与 Enum 类（两个参数）签名不一样。

编译器将 enum 类标记为 final 类，所以无法继承自 enum，其中还有一个 static 的初始化子句。

由于擦除效应。反编译无法得到 Enum 的完整信息，所以它展示的父类只是原始的 Enum，而非事实上的 Enum<Explore>。如将 enum 实例向上转型为 Enum，那么 values()就不可访问了。Class 中有一个 getEnumConstants()，不对枚举的类使用该方法，那么该方法返回 null。

19.5 实现，而非继承

创建一个新的 enum 时，可以同时实现一个或多个接口。

19.6 随机选取

语法<T extends Enum<T>>表示 T 是一个 enum 实例。Class<T>作为参数，可以利用 Class 对象得到 enum 实例的数组【Class 的 getEnumConstants()方法】，重载后的 random()方法只需使用 T[]作为参数，因为它不会调用 Enum 上的任何操作，只需从数组中随机选择一个元素即可，这样，最终返回类型正是 enum 的类型。

19.7 使用接口组织枚举

在一个接口的内部，创建实现该接口的枚举【static final】，以此将元素进行分组，可以达到将枚举元素分类组织的目的。

对 enum 而言，实现接口是使其子类化的唯一办法，如 enum 实现了 Food 接口，就可以将其实例向上转型为 Food。

```
public interface Food {
    enum Appetizer implements Food {
        SALAD, SOUP, SPRING_ROLLS;
    }
    ……
}
Food food = Appetizer.SALAD;
```

创建一个“枚举的枚举”,可以创建一个新的 enum,然后用其实例包装 Food 中的每一 enum 类（APPETIZER(Food.Appetizer.class)）。

```
public enum Course {
    APPETIZER(Food.Appetizer.class),
    ……
    private Food[] values;
    private Course(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
}
```

每个实例都将其对应的 Class 对象作为构造器的参数。通过 getEnumConstants()方法，可以从该 Class 对象中获取某个 Food 子类的所有 enum 实例。

另外一种更简洁的管理枚举的办法，就是将一个 enum 嵌套在另一个 enum 内。Secutity 接口的作用是将其所包含的 enum 组合成一个公共类型，然后，SecurityCategory 才能将 Secutity 中的 enum 作为其构造器的参数使用，以起到组合的效果。

```
public enum SecurityCategory {
    STOCK(Security.Stock.class), BOND(Security.Bond.class);
    Security[] values;
    SecurityCategory(Class<? extends Security> kind) {
        values = kind.getEnumConstants();
    }
    interface Security {
        enum Stock implements Security {SHORT, LONG, MARGIN}
        enum Bond implements Security {MUNICIPAL, JUNK}
    }
    public Security randomSelection() {
        return Enums.random(values);
    }
```

```
    public static void main(String[] args) {
        for (int i = 0; i < 10; i++) {
            SecurityCategory category = Enums.random(SecurityCategory.class);
            System.out.println(category + ": " + category.randomSelection());
        }
    }
}
```

19.8 使用 EnumSet 替代标志

Set 是一种集合，只能向其中添加不重复的对象。enum 成员都是唯一的，但不能从 enum 中删除或添加元素。

Java SE5 引入 EnumSet，是为了通过 enum 创建一种替代品，替代传统基于 int 的“位标志”，这种标志可以用来表示某种“开/关”信息。

EnumSet 的设计充分考虑到了速度因素，就其内部是将一个 long 值作为比特向量，所以非常高效。

EnumSet<E extends Enum<E>>类：

static <E extends Enum<E>>EnumSet<E> **allOf**(Class<E> elementType)创建一个包含指定元素类型的所有元素的枚举 set。

EnumSet<E> clone()返回 set 的副本。

static <E extends Enum<E>> EnumSet<E> **complementOf**(EnumSet<E> s)创建一个其元素类型与指定枚举 set 相同的枚举 set，最初包含指定 set 中所不包含的此类型的所有元素。

static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)创建一个从指定 collection 初始化的枚举 set。

static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)创建一个其元素类型与指定枚举 set 相同的枚举 set，最初包含相同的元素（如果有的话）。

static <E extends Enum<E>> EnumSet<E> **noneOf**(Class<E> elementType)创建一个具有指定元素类型的空枚举 set。

static <E extends Enum<E>> EnumSet<E> of(E e)创建一个最初包含指定元素的枚举 set。

static <E extends Enum<E>> EnumSet<E> of(E first, E... rest)创建一个最初包含指定元素的枚举 set。

static <E extends Enum<E>> EnumSet<E> of(E e1, E e2)创建一个最初包含指定元素的枚举 set。

static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3)创建一个最初包含指定元素的枚举 set。

static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4)创建一个最初包含指定元素的枚举 set。

static <E extends Enum<E>> EnumSet<E> of(E e1, E e2, E e3, E e4, E e5)创建一个最初包含指定元素的枚举 set。

static <E extends Enum<E>> EnumSet<E> **range**(E from, E to)创建一个最初包含由两个指定端点所定义范围内的所有元素的枚举 set。

从类 java.util.AbstractCollection 继承的方法

add, addAll, clear, contains, containsAll, isEmpty, iterator, remove, retainAll, size, toArray, toArray, toString

EnumSet 中的元素必须来自一个 enum。

使用 static import 可以简化 enum 常量的使用。

19.9 使用 EnumMap

EnumMap 是一种特殊的 Map，它要求其中的键（key）必须来自一个 enum。在内部可由数组实现。因此 EnumMap 速度很快。

EnumMap<K extends Enum<K>,V>类：

构造方法摘要

EnumMap(Class<K> keyType)创建一个具有指定键类型的空枚举映射。

EnumMap(EnumMap<K,? extends V> m)创建一个其键类型与指定枚举映射相同的枚举映射，最初包含

相同的映射关系（如果有的话）。

EnumMap(Map<K,? extends V> m)创建一个枚举映射，从指定映射对其初始化。

方法摘要

void clear()从此映射中移除所有映射关系。

EnumMap<K,V> clone()返回此枚举映射的浅表副本。

boolean containsKey(Object key)如果此映射包含指定键的映射关系，则返回 true。

boolean containsValue(Object value)如果此映射将一个或多个键映射到指定值，则返回 true。

Set<Map.Entry<K,V>> **entrySet**()返回此映射中所包含映射关系的 Set 视图。

boolean equals(Object o)比较指定对象与此映射的相等性。

V get(Object key)返回指定键所映射的值，如果此映射不包含此键的映射关系，则返回 null。

Set<K> keySet()返回此映射中所包含键的 Set 视图。

V put(K key, V value)将指定值与此映射中指定键关联。

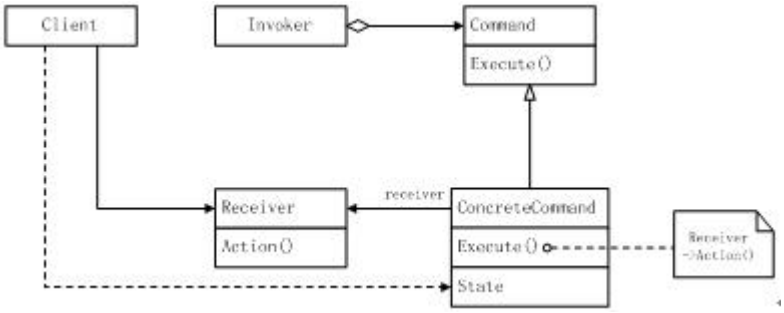
Void putAll(Map<? extends K,? extends V> m)将指定映射中所有映射关系复制到此映射中。

V remove(Object key)从此映射中移除该键的映射关系（如果存在）。

int size()返回此映射中的键-值映射关系数。

Collection<V> values()返回此映射中所包含值的 Collection 视图。

命令模式首先需要一个只有单一方法的接口，然后从该接口实现具有各自不同的行为的多个子类。接下来，就可以构造命令对象，并在需要的时候使用它们。



与 EnumSet 一样，enum 实例**定义时的次序决定了其在 EnumMap 中的顺序**。

如没有为键调用 put()方法来存入相应的值的话，其对应的值就是 null。

19.10 常量相关的方法

为 enum 实例编写方法，从而为每个 enum 实例赋予各自不同的行为。要实现常量相关的方法，需要为 enum 定义一个或多个 abstract 方法，然后为每个 enum 实例实现该抽象方法。

通过相应的 enum 实例，可以调用其上的方法。也称为表驱动的代码（table driven code，与命令模式有相似之处）。

通过常量相关的方法，每个 enum 实例可以具备自己独特的行为，但并不能将 enum 实例作为一个类型使用。

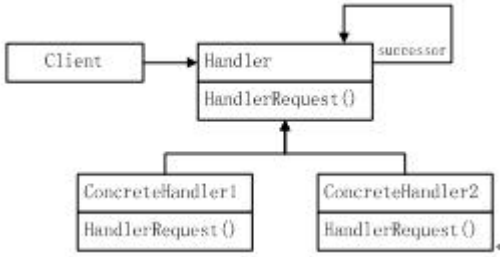
每个 enum 元素都是 enum 类型的 static final 实例，因此无法访问外部类的非 static 元素或方法。

与使用匿名内部类相比较，定义常量相关方法更高效、简洁。

除了实现 abstract 方法以外，也可以覆盖常量相关的方法。

19.10.1 使用 enum 职责链

职责链设计模式中，程序员以多种不同的方式来解决一个问题，然后将它们链接在一起，当一个请求到来时，遍历这个链，直到链中的某个解决方案能够处理该请求。



Mail 中有一个 randomMail()方法，负责随机创建用于测试的邮件。而 generator()方法生成一个 Iterable 对象，该对象在调用 next()方法时，在其内部使用 randomMail()来创建 Mail 对象。这样通过调用 Mail.generator()方法，容易地构造出一个 foreach 循环。

职责链由 enum MailHandler 实现，而 enum 定义的次序决定了各个解决策略在应用时的次序。对每一封邮件，都要按此顺序尝试每个解决策略，直到其中一个能够成功地处理该邮件，如所有策略失败了，那么该邮件将被判定为一封死信。

19.10.2 使用 enum 的状态机

一个状态机可以具有有限个特定的状态，它通常根据输入，从一个状态转移到下一个状态，不过也可能存在瞬时状态（transient states），而一旦任务执行结束，状态机就会离开瞬时状态。

每个状态都具有某些可接受的输入，不同的输入会使状态机从当前转移到不同的新状态。由于 enum 对其实例有严格限制，非常适合用来表现不同的状态的输入。一般而言，每个状态都具有一些相关的输入。

设计缺陷，它要求 enum State 实例访问的 VendingMachine 属性必须声明为 static，这意味着，只能有一个 VendingMachine 实例。不过实际的嵌入式 Java 应用，这也许并不是大问题，因为一台机器上，只有一个应用程序。

19.11 多路分发

Java 只支持单路分发。也就是说如要执行操作包含了不止一个类型未知的对象时，那么 Java 的动态绑定机制只能处理其中一个的类型。

解决问题的办法就是多路分发。多态只能发生在方法调用时，所以想使用两路分发，就必须两个方法调用：第一个方法调用决定第一个未知类型，第二个方法调用决定第二个未知的类型。一般而言，程序员需要有设定好的某种配置，以便一个方法调用能够引出更多的方法调用，从而能够处理多种类型。(示例对应的方法时 compete()和 eval())

Item 是几种类型的接口，将会被用于多路分发。RoShamBo1.match()有两个 Item 参数，通过调用 Item.compete()方法开始两路分发。要判定 a 的类型，分发机制会在 a 的实际类型 compete()内部起到分发的作用。compete()方法通过调用 eval()来为另一个类型实现第二次分发。将自身(this)作为参数调用 eval()，能够调用重载过的 eval()方法，这能够保留第一次分发的类型信息。当二次分发完成时，就能够知道两个 Item 的具体类型了。

19.11.1 使用 enum 分发

使用构造器来初始化每个 enum 示例，并以“一组”结果作为参数。两者放在一块，形成了类似查询表的结构。

仍使用两路分发判定两个对象的类型，第一次分发是实际的方法调用，第二次分发使用的是 switch，不过这样做是安全的，因为 enum 限制了 switch 语句的选择分支。

定义两个 static 方法（static 可以避免显式地指明参数类型）。第一个是 match()方法，它会为一个 Competitor 对象调用 compete()方法,并与另一个 Competitor 对象比较。在 paly()方法中，类型参数必须同时是 Enum<T>类型和 Competitor<T>类型。

19.11.2 使用常量相关的方法

常量相关的方法允许为每个 enum 实例提供方法的不同实现，不过，通过这种方式，enum 实例虽然可以具有不同的行为，但仍然不是类型，不能将其作为方法签名中参数类型来使用。最好将 enum 用在 switch 语句中。

对一个大型系统而言，难以理解的代码将导致整个系统不够健壮。

19.11.3 使用 EnumMap 分发

程序在一个 static 子句中初始化 EnumMap 对象，compete()方法，一行语句中发生了两次分发。

19.11.4 使用二维数组

每个 enum 实例都有一个固定的值（基于其声明的次序），并且通过 ordinal()方法取得

该值。使用二维数组，将竞争者映射到竞争结果。

第 20 章　注　　解

注解（也被称为元数据）在代码中添加信息提供了一种形式化的方法，使得可以在稍后某个时刻（编译、运行等）使用这些数据。

注解作用：提供用来完整地描述程序所需的信息，而这些信息无法用 Java 来表达。注解可以用来生成描述符文件，甚至或是新的类定义，并且有助于减轻编写“样板”代码的负担，通过注解，可以将元数据保存在 Java 源代码中，并利用 annotation API 为注解构造处理工具。

注解语法比较简单，除了@符号的使用之外，基本与 Java 固有的语法一致。

Java SE5 内置了三种，定义在 java.lang 中的注解：@Override，表示当前的方法定义将覆盖超类中的方法；@Deprecated，如果使用了注解为它的元素，编译器会发出告警信息（已过时）；@SuppressWarnings，关闭不当的编译器警告信息。

20.1 基本语法

从语法的角度来看，注解的使用方式与修饰符的使用是一模一样的。

20.1.1 定义注解

与其他任何 Java 接口一样，注解也会编译成 class 文件。除了@ 符号，注解类似于空的接口。定义注解时，会需要**元注解**（meta-annotation）。

在注解中，包含一些元素以表示某些值。当分析处理注解时，程序或工具可以利用这些值。注解元素类似接口的方式，唯一的区别可以为其**指定默认值**。

没有元素的注解称为**标记注解**（marker annotation）。

注解的元素在使用时表现为名-值对的形式，并且需要置于注解声明之后的括号内。

20.1.2 元注解

Java 至内置三种标准注解，以及四种元注解。

@Target	表示该注解可以用于什么地方。可能的 ElementType 参数包括： CONSTRUCTOR:构造器的声明 FIELD:域声明（包括 enum 实例） LOCAL_VARIABLE:局部变量声明 METHOD:方法声明 PACKAGE:包声明 PARAMETER:参数声明 TYPE:类、接口（包括注解类型）或 enum 声明
@Retention	表示需要在什么级别保持该注解信息。可选的 RetentionPolicy 参数包括： SOURCE:注解将被编译器丢弃 CLASS:注解在 class 文件中使用，但会被 VM 丢弃 RUNTIME:VM 将在运行期也保留注解，因此可以通过反射机制读取注解的信息。
@Documented	将此注解包含在 JavaDoc 中
@Inherited	允许子类继承父类中的注解

20.2 编写注解处理器

Java SE5 扩展了反射机制的 API，解析带有注解的 Java 源代码。

20.2.1 注解元素

注解元素可用的类型如下：所有基本类型（int，float，boolean 等）、String、Class、enum、Annotation 以及以上类型的数组。

20.2.2 默认值限制

编译器对元素的默认值比较严格。首先，元素不能有不确定的值，即元素必须要有默认值，要么在使用注解时提供元素的值；其次，对于非基本类型，都不能以 null 作为其值。为绕开约束，只能定义一些特殊的值，如空字符串或负数，以此表示某个元素不存在。

20.2.3 生成外部文件

注解可以将描述信息保存在 JavaBean 源文件中。

【在注解中定义注解】

快捷方式：如果注解中定义了名为 **value 的元素**，并且再应用该注解的时候，如果该元素是**唯一需要赋值**的元素，那么此时无需使用**名-值对**这种语法，只需在括号内给出 value 元素所需的值即可。

变通之道

可以使用多个注解替代在注解中定义注解。注意，使用多个注解时，同一个注解不能重复使用。

20.2.4 注解不支持继承

注解不支持 extends 继承一个注解。

20.2.5 实现处理器

20.3 使用 apt 处理注解

注解处理工具 apt

20.4 将观察者模式用于 apt

20.5 基于注解的单元测试

单元测试时对类中的每个方法提供一个或多个测试的一种实践，其目的是为了有规律地测试一个类的各个部分是否具备正确的行为。在 Java 中最著名的单元测试是 JUnit。JUnit4 支持注解。

@Unit 测试框架，@Test 标记测试方法。

第21章 并　　发

并发“具有可论证的确定性，但是实际上具有不可确定性”。

21.1 并发的多面性

21.1.1 更快的执行

并发提高运行在单处理器上的程序的性能，在于**阻塞**。常见示例是**事件驱动的编程**。产生具有可响应的用户界面。

Java 所使用的并发会共享诸如内存和 I/O 这样的资源，因此编写多线程程序最基本的**困难**在于协调不同线程驱动的任务之间这些资源的使用，以使得这些资源不会同时被多个任务访问。

Java 采取的方式在**顺序语言的基础上**提供对**线程**的支持。**是由执行程序表示的单一进程中创建任务**。好处是操作系统的透明性（系统移植性）。

21.1.2 改进代码设计

Java 线程机制是**抢占式**，这表示**调度机制会周期性地中断线程**，将上下文切换到另一个线程，从而为每个线程都提供时间片，使得每个线程都会分配到数量合理的时间去驱动它的任务。

并发需要付出代价，包含复杂性代价。

21.2 基本的线程机制

并发编程使得可以将程序划分为多个分离的、独立运行的任务，通过使用多线程机制，这些独立任务（也被称为子任务）中的每一个都将由**执行线程**来**驱动**。**一个线程就是进程中的一个单一的顺序控制流**。

底层机制是切分 CPU 时间，通常不需考虑。

21.2.1 定义任务

线程可以**驱动任务**，可以由 Runnable 接口来提供，并编写 run()方法，使得该任务可以执行你的命令。

21.2.2 Thread 类

将 Runnable 对象转变为工作任务的传统方式是把它提交给一个 Thread 构造器。调用 Thread 对象的 start()方法为该线程执行必需的初始化操作，然后调用 Runnable 的 run()方法，以便在这个新线程中启动该任务。

线程调度器在处理器之间**自动**分发线程。

创建 Thread 对象时，可以不捕获对象的引用【意思是没有引用指向 Thread 的实例】，在任务退出其 run()并死亡之前，垃圾回收期无法清除它。

Thread 类：

嵌套类摘要

static class Thread.State 线程状态。

static interface Thread.UncaughtExceptionHandler 当 Thread 因未捕获的异常而突然终止时，调用处理程序的接口。

字段摘要

static int MAX_PRIORITY 线程可以具有的最高优先级。

static int MIN_PRIORITY 线程可以具有的最低优先级。

static int NORM_PRIORITY 分配给线程的默认优先级。

方法摘要

static int activeCount()返回当前线程的线程组中活动线程的数目。

void checkAccess()判定当前运行的线程是否有权修改该线程。

static Thread currentThread()返回对当前正在执行的线程对象的引用。

static void dumpStack()将当前线程的堆栈跟踪打印至标准错误流。

static int enumerate(Thread[] tarray)将当前线程的线程组及其子组中的每一个活动线程复制到指

定的数组中。

static Map<Thread,StackTraceElement[]> getAllStackTraces()返回所有活动线程的堆栈跟踪的一个映射。

ClassLoader getContextClassLoader()返回该线程的上下文 ClassLoader。

static Thread.UncaughtExceptionHandler getDefaultUncaughtExceptionHandler()返回线程由于未捕获到异常而突然终止时调用的默认处理程序。

long getId()返回该线程的标识符。

String getName()返回该线程的名称。

int getPriority()返回线程的优先级。

StackTraceElement[] getStackTrace()返回一个表示该线程堆栈转储的堆栈跟踪元素数组。

Thread.State getState()返回该线程的状态。

ThreadGroup getThreadGroup()返回该线程所属的线程组。

Thread.UncaughtExceptionHandler getUncaughtExceptionHandler()返回该线程由于未捕获到异常而突然终止时调用的处理程序。

static boolean holdsLock(Object obj)当且仅当当前线程在指定的对象上保持监视器锁时，才返回 true。

void interrupt()中断线程。

static boolean interrupted()测试**当前线程**是否已经中断。【并将中断标志位置为 true】

boolean isAlive()测试线程是否处于活动状态。

boolean isDaemon()测试该线程是否为守护线程。

boolean isInterrupted()测试线程是否已经中断。

void join()等待该线程终止。

void join(long millis)等待该线程终止的时间最长为 millis 毫秒。

void join(long millis, int nanos)等待该线程终止的时间最长为 millis 毫秒 + nanos 纳秒。

void run()如果该线程是使用独立的 Runnable 运行对象构造的，则调用该 Runnable 对象的 run 方法；否则，该方法不执行任何操作并返回。

void setContextClassLoader(ClassLoader cl)设置该线程的上下文 ClassLoader。

void setDaemon(boolean on)将该线程标记为守护线程或用户线程。

static void setDefaultUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)设置当线程由于未捕获到异常而突然终止，并且没有为该线程定义其他处理程序时所调用的**默认处理程序**。

void setName(String name)改变线程名称，使之与参数 name 相同。

void **setPriority**(int newPriority)更改线程的优先级。

void setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler eh)设置**该线程**由于未捕获到异常而突然终止时调用的处理程序。

static void sleep(long millis)在指定的毫秒数内让**当前正在执行的线程**休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。

static void sleep(long millis, int nanos)在指定的毫秒数加指定的纳秒数内让**当前正在执行的线程**休眠（暂停执行），此操作受到系统计时器和调度程序精度和准确性的影响。

void start()使该线程开始执行；Java 虚拟机调用该线程的 run 方法。

String toString()返回该线程的字符串表示形式，包括线程名称、优先级和线程组。

static void yield()暂停当前正在执行的线程对象，并执行其他线程。

21.2.3 使用 Executor

Java SE5 的 java.util.concurrent 包中的执行器（Excutor）将管理 Thread 对象。从而简化了并发编程。Executor 在客户端和任务执行之间提供了一个**间接层**；与客户端直接执行任务不同，这个中介对象将执行任务。Executor 允许管理异步任务的执行，而无须显式地管理线程的生命周期。

Executors 类：

static Callable<Object> callable(PrivilegedAction<?> action)返回 Callable 对象，调用它时可运行给定特权的操作并返回其结果。

static Callable<Object> callable(PrivilegedExceptionAction<?> action)返回 Callable 对象，调用它时可运行给定特权的异常操作并返回其结果。

static Callable<Object> callable(Runnable task)返回 Callable 对象，调用它时可运行给定的任

务并返回 null。

static <T> Callable<T> callable(Runnable task, T result)返回 Callable 对象，调用它时可运行给定的任务并返回给定的结果。

static ThreadFactory defaultThreadFactory()返回用于创建新线程的默认线程工厂。

static ExecutorService newCachedThreadPool()创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们。

static ExecutorService newCachedThreadPool(ThreadFactory threadFactory)创建一个可根据需要创建新线程的线程池，但是在以前构造的线程可用时将重用它们，并在需要时使用提供的 ThreadFactory 创建新线程。

static ExecutorService newFixedThreadPool(int nThreads)创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程。

static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory)创建一个可重用固定线程数的线程池，以共享的无界队列方式来运行这些线程，在需要时使用提供的 ThreadFactory 创建新线程。

static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

static ScheduledExecutorService newScheduledThreadPool(int corePoolSize, ThreadFactory threadFactory)创建一个线程池，它可安排在给定延迟后运行命令或者定期地执行。

static ExecutorService newSingleThreadExecutor()创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程。

static ExecutorService newSingleThreadExecutor(ThreadFactory threadFactory)创建一个使用单个 worker 线程的 Executor，以无界队列方式来运行该线程，并在需要时使用提供的 ThreadFactory 创建新线程。

static ScheduledExecutorService newSingleThreadScheduledExecutor()创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期地执行。

static ScheduledExecutorService newSingleThreadScheduledExecutor(ThreadFactory threadFactory)创建一个单线程执行程序，它可安排在给定延迟后运行命令或者定期地执行。

static <T> Callable<T> privilegedCallable(Callable<T> callable)返回 Callable 对象，调用它时可在当前的访问控制上下文中执行给定的 callable 对象。

static <T> Callable<T> privilegedCallableUsingCurrentClassLoader(Callable<T> callable)返回 Callable 对象，调用它时可在当前的访问控制上下文中，使用当前上下文类加载器作为上下文类加载器来执行给定的 callable 对象。

static ThreadFactory privilegedThreadFactory()返回用于创建新线程的线程工厂，这些新线程与当前线程具有相同的权限。

static ExecutorService unconfigurableExecutorService(ExecutorService executor)返回一个将所有已定义的 ExecutorService 方法委托给指定执行程序的对象，但是使用强制转换可能无法访问其他方法。

static ScheduledExecutorService unconfigurableScheduledExecutorService (ScheduledExecutorService executor)返回一个将所有已定义的 ExecutorService 方法委托给指定执行程序的对象，但是使用强制转换可能无法访问其他方法。

ExecutorService 接口：继承自 Executor 接口

方法摘要

boolean **awaitTermination**(long timeout, TimeUnit unit)请求关闭、发生超时或者当前线程中断，无论哪一个首先发生之后，都将导致阻塞，直到所有任务完成执行。

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)执行给定的任务，当所有任务完成时，返回保持任务状态和结果的 Future 列表。

<T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)执行给定的任务，当所有任务完成或超时期满时（无论哪个首先发生），返回保持任务状态和结果的 Future 列表。

<T> T invokeAny(Collection<? extends Callable<T>> tasks)执行给定的任务，如果某个任务已成功完成（也就是未抛出异常），则返回其结果。

<T> T invokeAny(Collection<? extends Callable<T>> tasks, long timeout, TimeUnit unit)执行

给定的任务，如果在给定的超时期满前某个任务已成功完成（也就是未抛出异常），则返回其结果。

boolean isShutdown()如果此执行程序已关闭，则返回 true。

boolean isTerminated()如果关闭后所有任务都已完成，则返回 true。

void shutdown()启动一次顺序关闭，**执行以前提交的任务，但不接受新任务**。

List<Runnable> shutdownNow()试图停止所有正在执行的活动任务，暂停处理正在等待的任务，并返回等待执行的任务列表。

<T> Future<T> submit(Callable<T> task)提交一个返回值的任务用于执行，返回一个表示任务的未决结果的 Future。

Future<?> submit(Runnable task)提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。

<T> Future<T> submit(Runnable task, T result)提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。

21.2.4 从任务中产生返回值

可以实现 Callable 接口，是具有类型参数的泛型，它的类型参数表示从 call()方法中返回的值，并且必须使用 ExecutorService.submit()方法调用它。

Callable<V>接口：

V call()计算结果，如果无法计算结果，则抛出一个异常。

Future<V>接口：

方法摘要

boolean cancel(boolean mayInterruptIfRunning)试图取消对此任务的执行。

V get()如有必要，等待计算完成，然后获取其结果。

V get(long timeout, TimeUnit unit)如有必要，最多等待为使计算完成所给定的时间之后，获取其结果（如果结果可用）。

boolean isCancelled()如果在任务正常完成前将其取消，则返回 true。

boolean isDone()如果任务已完成，则返回 true。

21.2.5 休眠

TimeUnit 枚举

枚举常量摘要：DAYS、HOURS、MICROSECONDS、MILLISECONDS、MINUTES、NANOSECONDS、SECONDS

方法摘要

long convert(long sourceDuration, TimeUnit sourceUnit)将给定单元的时间段转换到此单元。

void sleep(long timeout)使用此单元执行 Thread.sleep.这是将时间参数转换为 Thread.sleep 方法所需格式的便捷方法。

void timedJoin(Thread thread, long timeout)使用此时间单元执行计时的 Thread.join。

void timedWait(Object obj, long timeout)使用此时间单元执行计时的 Object.wait。

long toDays(long duration)等效于 DAYS.convert(duration, this)。

long toHours(long duration)等效于 HOURS.convert(duration, this)。

long toMicros(long duration)等效于 MICROSECONDS.convert(duration, this)。

long toMillis(long duration)等效于 MILLISECONDS.convert(duration, this)。

long toMinutes(long duration)等效于 MINUTES.convert(duration, this)。

long toNanos(long duration)等效于 NANOSECONDS.convert(duration, this)。

long toSeconds(long duration)等效于 SECONDS.convert(duration, this)。

static TimeUnit valueOf(String name)返回带有指定名称的该类型的枚举常量。

static TimeUnit[] values()Returns an array containing the constants of this enum type, in the order they are declared.

21.2.6 优先级

调度器倾向于让优先级最高的线程先执行，优先级低的线程仅仅是执行的频率较低。所有线程都应该以默认的优先级允许。

JDK 有 10 个优先级，但与操作系统映射得不是很好，唯一的可移植的方法是当调整优先级的时候，只使用 MAX_PRIORITY、MIN_PRIORITY 和 NORM_PRIORITY 三种级别。

21.2.7 让步

yield()方法是主动放弃 CPU 使用权，由“运行状态”进入“就绪状态”，并与其他线程**参加 CPU 竞争**【可能马上重新获得 CPU 使用权】。

21.2.8 后台线程

后台（daemon）线程，是指在程序运行的时候在后台提供一种通用服务的线程，并且这种线程不属于程序中不可或缺的部分。因此，当所有的非后台线程结束时，会杀死进程中的所有后台线程。

必须在线程启动之前调用 setDaemon()方法，才能把它设置为后台线程。

ThreadFactory 接口：

Thread newThread(Runnable r)构造一个新 Thread。

通过编写定制的 ThreadFactory 可以定制由 Executor 创建的线程的属性(后台、优先级、名称)。

通过使用 Executors.newCachedThreadPool(ThreadFactory factory)方法,每个静态的 ExecutorService 创建方法都被**重载**为接受一个 **ThreadFactory 对象**，而这个对象将**被用来创建新的线程**。

线程被设置为后台线程，那么它派生出的子线程，也自动为后台线程。

一旦最后一个非后台线程终止，JVM 就会立即关闭所有的后台进程，而不会有任何确认形式。因此，后台进程在不执行 finally 字句的情况下就会终止其 run()方法。

21.2.9 编码的变体

任务类直接从 Thread 继承这种可替换实现了 Rnnable 的方式。

自管理的 Runnable：实现 Runnable 接口，Thread 对象作为私有成员，在**构造器中调用 start()方法**。在构造器中启动线程存在的问题：另一个任务可能会在构造器结束之前开始执行，意味着该任务能够访问处于不稳定状态的对象。

可以通过使用内部类来将线程代码隐藏在类中。

21.2.10 术语

Thread 自身不执行任何操作，它只是驱动赋予它的**任务**。在描述将要执行的工作使用术语**“任务”**，只有在引用到驱动任务的具体机制时，才使用**“线程”**。

Java 的线程机制基于来自 C 的低级的 p 线程方式。

21.2.11 加入一个线程

一个线程在其他线程上调用 join()方法，效果是让“主线程”等待“子线程”结束后才能继续执行。

join()方法的调用可以被中断，做法是“子线程”上调用 interrupt()方法，需要 try-catch 字句。

21.2.12 创建有响应的用户界面

计算作为后台程序运行，同时还在等待用户输入。把计算程序放在 run()方法中，这样它就能让出处理器给别的程序。

21.2.13 线程组

“最好把线程组看成一次不成功的尝试，只要忽略它就好了”

21.2.14 捕获异常

不能捕获从线程中逃逸的异常。一旦异常逃出任务的 run()方法，它就会向外**传播到控制台**，除非采取特殊步骤捕获这种错误的异常。(把 main 主体放到 try-catch 语句块中是没有作用的)

捕获异常方式： Thread.UncaughtExceptionHandler 是 Java SE5 的新接口，它允许在每个 Thread 对象上都附着一个异常处理器。uncaughtException()会在线程因未捕获的异常而临近死亡时被调用。

Thread.UncaughtExceptionHandler 接口

void uncaughtException(Thread t,Throwable e)当给定线程因给定的未捕获异常而终止时，调用该

方法。

为了使用该接口，创建一个新类型的 ThreadFactory，通过调用 Thread 上的 setUncaughtExceptionHandler(Thread.UncaughtExceptionHandler)方法它**将每个新创建的 Thread 对象上附着一个 Thread.UncaughtExceptionHandler。我们将这个工厂传递给 Executors 创建新的 ExecutorService 的方法。**

更简单的方式是在 Thread 类中设置一个静态域，并将这个处理器设置为默认的未捕获异常处理器。Thread.setDefaultUncaughtExceptionHandler()。

21.3 共享受限资源

21.3.1 不正确访问资源

一个任务有可能在另一个任务执行**第一个**对 currentEvenValue 的递增操作之后，但没有执行第二个操作之前，调用 next()方法。这将使这个值处于“不恰当”的状态。

21.3.2 解决共享资源竞争

防止冲突的方法就是当一个资源被一个任务使用时，在其上加锁。采用**序列化访问共享资源**的方案，意味着给定时刻只允许一个任务访问共享资源。通常是通过在代码前面加上一条锁语句来实现，这使得一段时间内只有一个任务可以运行这段代码。因为锁语句产生了互相排斥的效果，所以这种机制通常称为**互斥量（mutex）**。

Java 以关键字 synchronized 的形式，为防止资源冲突提供了内置支持。

共享资源一般是以对象形式存在的内存片段，要控制对**共享资源**的访问，得先把它**包装进一个对象**，然后**所有要访问**这个**资源的方法标记为 synchronized**。如果某个任务处于一个对标记为 synchronized 的方法调用中，那么这个线程从该方法返回之前，其他所有要调用类中**任何标记为 synchronized 的线程**都会被阻塞。

所有对象都自动含有单一的**锁**（也称为**监视器**）。当在对象上调用其任意 synchronized 方法，**此对象都被加锁，**该对象上的其他 synchronized 方法只有等到前一个方法调用完毕并释放了锁之后才能被调用。对于某个对象而言，其**所有 synchronized 方法共享同一个锁**，可以被用来防止多个任务同时访问被编码为对象内存。

使用并发，**将域设置为 private**，否则，synchronized 关键字就不能防止其他任务直接访问域，这样就会产生冲突。

一个任务可以多次获得对象的锁。JVM 负责跟踪对象被**加锁的次数**，每当任务离开 synchronized 方法，计数递减，当计数为零，其他任务可以使用此资源。

针对每个类，也有一个锁（作为类的 Class 对象的一部分），所以 **synchronized static 方法可以在类的范围内防止对 static 数据的并发访问。**

Brian 的同步规则：如果你正在写一个变量，它可能接下来将被另一个线程读取，或者正在读取一个上一次已经被另一个线程写过的变量，那么必须使用同步，并且，读写线程都必须使用相同的监视器同步。

每个访问临界共享资源的方法都必须被同步，否则他们就不会正确地工作。

使用显式的 Lock 对象

java.util.concurrent.locks 中显式的互斥机制，Lock 对象必须显示地创建、锁定和释放。和内建锁形式相比，代码缺乏优雅性，但是解决某些类型的问题，更加灵活。

使用 Lock 对象时:**紧接着对 lock()的调用,必须放置 try-finally 语句中,并在 finally 子句中带有 unlock()的调用**。注意，return 语句必须在 try 字句中出现，以确保 unlock()不会过早发生，从而将数据暴露给第二个任务。

```
Lock l = ...;
l.lock();
try {
        // access the resource protected by this lock
} finally {
```

```
            l.unlock();
        }
```

优点：使用 finally 字句将系统维护在正确的状态。

特殊情况才使用显式的 Lock 对象。如，用 synchronized 关键字不能尝试着获取锁且最终获取锁会失败，或者尝试着获取锁一段时间，然后放弃它。

Lock 接口

void lock()获取锁。

void **lockInterruptibly**()如果当前线程未被中断，则获取锁。【如果锁可用，则获取锁，并立即返回。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下两种情况之一以前，该线程将一直处于休眠状态：锁由当前线程获得；或者其他某个线程中断当前线程，并且支持对锁获取的中断。】

Condition newCondition()返回绑定到此 Lock 实例的新 Condition 实例。

boolean tryLock()仅在调用时锁为空闲状态才获取该锁。

boolean tryLock(long time, TimeUnit unit)如果锁在给定的等待时间内空闲，并且当前线程未被中断，则获取锁。

void unlock()释放锁。

ReentrantLock 类：允许尝试着获取但最终未获取锁，如果其他人已经获取了这个锁，那么可以离开执行其他事情，而不是一直等待锁被释放。

方法摘要

int getHoldCount()查询当前线程保持此锁的次数。

protected Thread getOwner()返回目前拥有此锁的线程，如果此锁不被任何线程拥有，则返回 null。

protected Collection<Thread> getQueuedThreads()返回一个 collection，它包含可能正等待获取此锁的线程。

int getQueueLength()返回正等待获取此锁的线程估计数。

protected Collection<Thread> getWaitingThreads(Condition condition)返回一个 collection，它包含可能正在等待与此锁相关给定条件的那些线程。

int getWaitQueueLength(Condition condition)返回等待与此锁相关的给定条件的线程估计数。

boolean hasQueuedThread(Thread thread)查询给定线程是否正在等待获取此锁。

boolean hasQueuedThreads()查询是否有些线程正在等待获取此锁。

boolean hasWaiters(Condition condition)查询是否有些线程正在等待与此锁有关的给定条件。

boolean isFair()如果此锁的公平设置为 true，则返回 true。

boolean isHeldByCurrentThread()查询当前线程是否保持此锁。

boolean **isLocked**()查询此锁是否由任意线程保持。

void **lock**()获取锁。

void **lockInterruptibly**()如果当前线程未被中断，则获取锁。【如果当前线程未被中断，则获取锁。如果锁可用，则获取锁，并立即返回。如果锁不可用，出于线程调度目的，将禁用当前线程，并且在发生以下两种情况之一以前，该线程将一直处于休眠状态：①锁由当前线程获得；②或者其他某个线程中断当前线程，并且支持对锁获取的中断。如果当前线程：在进入此方法时已经设置了该线程的中断状态；或者在获取锁时被中断，并且支持对锁获取的中断，则将抛出 InterruptedException，并清除当前线程的已中断状态。】

Condition newCondition()返回用来与此 Lock 实例一起使用的 Condition 实例。

String toString()返回标识此锁及其锁定状态的字符串。

boolean tryLock()仅在调用时锁未被另一个线程保持的情况下，才获取该锁。

boolean tryLock(long timeout, TimeUnit unit)如果锁在给定等待时间内没有被另一个线程保持，且当前线程未被中断，则获取该锁。

void unlock()试图释放此锁。

21.3.3 原子性与易变性

原子操作是不能被线程调度机制中断的操作。依赖于原子性很棘手且很危险的。

Goetz 测试：如果你可以编写用于现代微处理器的高性能 JVM，那么就有资格去考虑是否可以避免同步。

JVM 将 64 位的读取和写入当作两个分离的 32 位操作执行，这就产生了一个读取和写入操作中间发生上下文切换，从而导致不同的任务可以看到不正确结果的可能性（也称为**字撕**

裂)。原子操作可由线程机制来保证其不可中断，专家级程序员可以利用这一点来编写无锁的代码。

可视性问题远比原子性问题多得多。一个任务作出的修改，即使在不中断的意义上讲是原子性的，对其他任务也可能是不可视(如，修改只是暂时性地存储在本地处理器的缓存中)，因此不同的任务对应用的状态有不同的视图。

volatile 关键字确保了应用中的可视性，volatile 域会立即被写入到主存中，而读取操作发生在主存中。

在非 volatile 域上的原子性操作不必刷新到主存中(任何写入操作对这个任务都是可视的，因此如果只需要在这个任务内部可视，就不需要将其设置为 volatile)，因此其他读取该域的任务也不必看到这个新值。

①volatile 变量的写操作必须不依赖当前值（如递增一个计数器，volatile 无法工作）②变量没有包含在具有其他变量的不变式中，只有状态真正独立于程序内其他内容时才能使用 volatile。

使用 volatile 而不是 synchronized 的唯一安全的情况是类中只有一个可变的域。

如果一个域可能被多个任务同时访问，或者这些任务至少有一个写入任务，应该将这个域设置为 volatile，让编译器不要执行任何移除读取和写入操作的优化，这些操作的目的是用线程中的局部变量维护对这个域的精确同步。

21.3.4 原子类

Java SE5 引入了 AtomicInteger、AtomicLong、AtomicReference 等特殊的原子性变量类。只有在特殊情况下才在自己的代码使用它们，即便使用了也需要确保不存在其他可能出现的问题，通常依赖于锁会更安全一些。

21.3.5 临界区

希望**多个线程同时访问方法内部的部分代码**而不是防止访问整个方法，通过这种方式分离出来的代码段被称为**临界区** (critical section)，也是使用 synchronized 关键字建立。

```
synchronized(syncObject) {
   //This code can be accessed
   //by only one task at a time
}
```

也被称为**同步控制块**；在进入此段代码前，必须得到 **syncObject 对象的锁**。如果其他线程得到这个锁，那么就得等到锁被释放后，才能进入临界区。

可以提高访问对象的时间性能得到提高（对象的不加锁时间更长)。

可以显式使用 Lock 对象来创建临界区。

21.3.6 在其他对象上同步

Synchronized 块必须给定一个在其上进行同步的对象，并且最合理的方式是，使用其方法正在被调用的当前对象：synchronized(this)，临界区的效果就会直接缩小在同步的范围内。有时必须在另一个对象上同步，就必须**确保所有相关的任务都是在同一个对象上同步**。

在不同对象上的同步锁之间是相互独立的，哪怕在同一个对象内部。

21.3.7 线程本地存储

防止任务在共享资源上产生冲突的第二种方式是**根除对变量的共享**。线程本地存储是一种自动化机制，可以为使用变量的每个不同的线程都**创建不同的存储**。

创建和管理线程本地存储可以由 java.lang.ThreadLocal 类来实现。

ThreadLocal<T>类：

T get()返回此线程局部变量的当前线程副本中的值。

protected T **initialValue**()返回此线程局部变量的当前线程的“初始值”。

void remove()移除此线程局部变量当前线程的值。

void set(T value)将此线程局部变量的当前线程副本中的值设置为指定值。

ThreadLocal 对象通常当作**静态域存储**，只能通过 get()和 set()方法来访问该对象的内

容。

```
private static ThreadLocal<Integer> value =
    new ThreadLocal<Integer>() {
        private Random rand = new Random(47);
        protected synchronized Integer initialValue() {
            return rand.nextInt(1000);
        }
};
```

21.4 终结任务

21.4.1 装饰性花园

使用单个 Count 对象跟踪参观者的主计数值，并且将其当做 Entrance 类中的一个静态域进行存储。Count.increment()和 Count.value()都是 synchonized 的，用来控制对 count 域的访问。increment()方法使用 Random 对象，目的是在从把 count 读取到 temp 中，到递增 temp 并将其存储回 count，大约一半的时间产生让步。如果将 increment()上的 synchronized 关键字注释掉，那么马上将出问题。

每个 Entrance 任务都维护着一个本地 number，它包含通过某个特定入口进入参观者的数量。提供了对 count 对象的双重检查，以确保其记录的参观者数量是正确的。Entrance.run()只是递增 number 和 count 对象，然后休眠 100 毫秒。

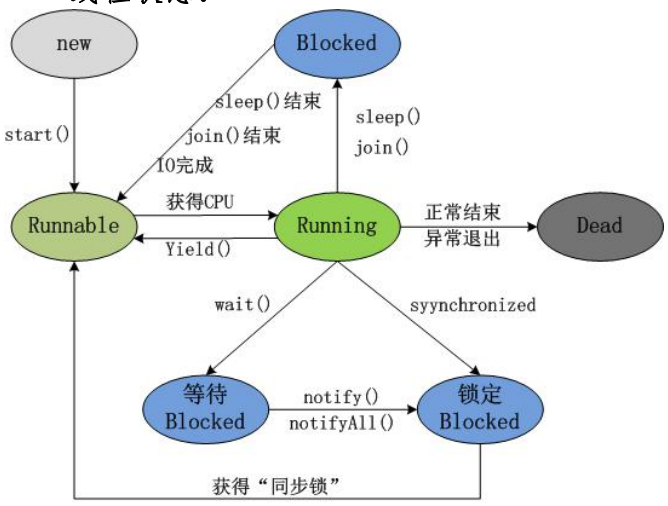
通过因为 Entrance.canceled 是一个 volatile 布尔标志，而它只会被读取和赋值，所以不需要同步对其访问，就可以安全地操作它。

Entrance 发送 static cancel()消息，然后调用 exec 对象的 shutdown()方法，之后调用 exec 上的 awaitTermination()方法。ExecutorService.awaitTermination()等待每个任务结束，如果所有的任务在超时时间达到之前全部结束，则返回 true，否则返回 false，表示不是所有的任务都已经结束。

控制任务终结。（在 run()方法中使用 while 循环，直到 cancel 标志位改变，并退出循环）

21.4.2 阻塞时终结

线程状态：



1）新建（new）：当线程被创建时，只会短暂地处于这种状态。此时它已经分配了必需的系统资源，并执行了初始化。此时线程已经有资格获得 CPU 时间了，之后调度器将把这个线程转变为可运行状态或阻塞状态。

2）就绪（Runnable）：只要调度器把时间片分配给线程，线程就可以运行。也就是说，在任意时刻，线程可以运行也可以不运行。只要调度器能分配时间片给线程，它就可以运行；

这不同于死亡和阻塞状态。

3）阻塞（Blocked）：线程能够运行，但有某个条件阻止它的运行。当线程处于阻塞状态时，调度器将忽略线程，不会分配给线程任何 CPU 时间。直到线程重新进入了就绪状态，才可以执行操作。

4）死亡（Dead）：处于死亡或终止状态的线程将不再是可调度的，并且再也不会得到 CPU 时间，它的任务已结束，或不再是可运行的。任务死亡的通常方式是从 run()方法返回，但是任务的线程还可以被中断。

进入阻塞状态

可能原因如下：

1）通过调用 sleep()使任务进入休眠状态；

2）调用 wait()使线程挂起，直到线程得到 notify()或 notifyAll()消息，线程才会进入就绪状态；

3）任务等待某个输入/输出完成；

4）任务试图在某个对象上调用其同步控制方法，但另一个任务已经获取了这个锁。

21.4.3 中断

在 Runnable.run()中间打断它，当打断被阻塞的任务时，可能需要清理资源。因此在 Java 线程中这种类型的异常中断中用到了异常。为了在以这种方式终止任务时，返回良好状态，必须仔细考虑代码的执行路径，并仔细编写 catch 字句以正确清除所有事物。

Thread 类包含 interrupt()方法，因此可以终止被阻塞的任务，该方法将**设置线程的中断状态**。如一个线程已被阻塞，或试图执行一个阻塞操作，那么设置这个线程的中断状态将**抛出 InterruptedException**。当**抛出异常或者该任务调用 Thread.interrupted()**时，**中断状态**将被复位。

尽量通过 Executor 来执行所有操作，在 Executor 上调用 **shutdownNow()**，那么它发送一个 interrupt()调用给它启动的**所有**线程。

中断某个单一任务：

```
ExecutorService exec = Executors.newCachedThreadPool();
Runnable r = …;
Future<?> f = exec.submit(r);
f.cancel(true);//中断特定任务 r
```

I/O 和 **synchronized** 块上的等待是不可中断的。I/O 等待确实行之有效的解决方案，即关闭任务在其上发生阻塞的底层资源。nio 类提供了更人性化的 I/O 中断，被阻塞的 nio 通道会自动地响应中断（ClosedByInterruptException）。

被互斥所阻塞【解决锁定阻塞无法中断的问题**】**

同一个互斥可以能被同一个任务多次获得，因为一个任务应该能够调用在同一个对象中的其他的 synchronized 方法，而这个任务已经持有锁了。

无论在任何时刻，只要任务以不可中断的方式被阻塞，那么都有潜在的会锁住程序的可能。

Java SE5 并发类库中添加了一个特性，即在 ReentrantLock 上阻塞的任务具备可以被中断的能力。【lockInterruptibly()方法：如果当前线程，在进入此方法时已经设置了该线程的**中断状态**；或者在等待**获取锁的同时被中断**。则抛出 **InterruptedException**，并且**清除**当前线程的已中断状态。在此实现中，因为此方法是一个显式中断点，所以**要优先考虑响应中断，而不是响应锁的普通获取或重入获取。】**

21.4.4 检查中断

当在线程上调用 interrupt()时，**中断发生的唯一时刻是任务要进入到阻塞操作中，或者已经在阻塞操作内部时**（除了不可中断的 I/O 或被阻塞的 synchronized 方法之外），可以通过在阻塞调用上**抛出异常**来退出，

如 run()循环碰巧没有产生任何阻塞调用的情况下，任务需要**第二种方式来退出**。由中断状态（通过 interrupt()来设置）来表示的，通过调用 **interrupted()来检查中断状态，这不仅可以确定 interrupt()是否被调用过，而且还可以清除中断状态。**清除中断状态可以确保并发结构不会就某个任务被中断这个问题通知两次。

```
try {
   while(!Thread.interrupted()) {
      NeedCleanup n1 = new NeedCleanup();
      try {
         //…
      }finally {
         n1.cleanup();//确保无论 run()如何退出，清理都会发生
      }
   }
} catch(InterruptedException e) {
   //…
}
```

被设计用于响应 interrupt()的类必须建立一种策略，来确保它将保持一致的状态，通常意味着所有需要清理的对象创建操作的后面，都必须紧跟 try-finally 字句，从而使得无论 run()循环如何退出，清理都会发生。

21.5 线程之间的协作

当任务协作时，关键问题是这些任务之间的握手。为了实现握手，使用相同的基础特性：互斥。互斥确保只有一个任务可以响应某个信号，可以根除任何可能的竞争条件。在互斥之上，为任务添加了一种**途径**（握手），可以将其自身挂起，直至某些外部条件发生变化，表示是时候让这个任务向前开动了为止。这种握手通过 Object 的方法 wait()和 notify()来安全实现。Java SE5 并发类库提供了具有 await()和 signal()方法的 Condition 对象。

21.5.1 wait()与 notifyAll()

wait()使得可以等待某个条件发生变化，而改变这个条件超出了当前方法的控制能力。通常，这种条件将由另一个任务来改变。因此 wait()会在等待外部条件变化时将任务**挂起**，并且只有在 notify()和 notifyAll()发生时，即表示某些条件发生变化，任务才被唤醒并去检查所产生的变化。因此 wait()提供了一种在任务之间对活动同步的方式。

线程的执行被挂起，对象上的锁被释放。因为 wait()将释放锁，意味着另一个任务可以获得这个锁，因此在该对象中的其他 synchronized 方法可以在 wait()期间被调用。

wait()在其他线程调用**此对象（执行挂起对象）**的 notify()方法或 notifyAll()方法前，导致**当前线程**挂起。**当前线程**必须**拥有此对象监视器**。该线程发布对此监视器的所有权并等待，直到其他线程通过调用 notify 方法，或 notifyAll 方法通知在**此对象**的监视器上等待的线程醒来。然后该线程将等到重新获得**对监视器的所有权**后才能继续执行。

对于某一个参数的版本，实现**中断和虚假唤醒**是可能的，而且**此方法应始终在循环中使用**：

```
synchronized (obj) {
   while (<condition does not hold>)
      obj.wait();
   // Perform action appropriate to condition
}
```

此方法只应由作为此对象监视器的所有者的线程来调用。

有两种形式的 wait()。

第一种版本接受毫秒数作为参数，含义与 sleep()方法里参数的意思相同，都是指“在

此期间暂停”。与 sleep()不同的是:1)在 wait()期间对象锁是释放的;2)可以通过 notify()、notifyAll()，或令时间到期，从 wait()中恢复执行。

第二种，wait()不接受任何参数。这种 wait()无限制等待下去，直到接收到 notify()、notifyAll()消息。

wait()、notify()、notifyAll()是基类 Object 的一部分，因为**这些方法操作的锁也是所有对象的一部分**。所以可以把 wait()放进任何同步控制方法里，而不用考虑这个类是继承自 Thread 还是实现了 Runnable 接口。实际上，**只能在同步控制方法或同步控制块里调用 wait()、notify()和 notifyAll()**（因为不用操作锁，所以 sleep()可以在非同步块控制方法里调用)。如果在非同步控制方法里调用这些方法，得到 IllegalMonitorStateException 异常。

使用 while 循环包围 wait()是因为:

（1）可能多个任务出于相同的原因在等待同一个锁，而第一个唤醒任务可能会改变这种状况。如果属于这种情况，那么这个任务应该被再次挂起，直至其感兴趣的条件发生变化;【虚假唤醒】

（2）在这个任务从其 wait()中被唤醒的时刻，有可能会有某个其他的任务已经做出了改变，从而使得这个任务在此时不能执行，或者执行其操作已显得无关紧要。此时，应该通过再次调用 wait()来将其重新挂起;

（3）也有可能某些任务处于不同的原因等待你的对象上的锁（这种情况必须等待 notifyAll()）。这种情况下，你需要检查是否已经由正确的原因唤醒，如果不是，就再次调用 wait()。

其本质是要检查所感兴趣的特定条件，并在条件不满足的情况下返回 wait()中。

错失的信号

T1:	T2:

synchronized(sharedMonitor)
{
<setup condition for T2>
sharedMonitor.notify();
}
	while(someCondition) {
	//point1
	synchronized(sharedMonitor)
	sharedMonitor.wait()
	}

在 point1，线程调度器可能切换到 T1，而 T1 执行其设置，然后调用 notify()，T2 得以继续执行时，此时，对 T2 来说，时机已经太晚了，以至于不能意识到这个条件已经发生了变化，因此盲目进入了 wait()。此时 notify()将错失，而 T2 也将无限等待这个已经发送过的信号，从而产生死锁。

解决方案防止在 someCondition 变量上产生竞争条件。

```
synchronized(sharedMonitor) {
   While(someCondition)
       sharedMonitor.wait();
}
```

21.5.2 notify()与 notifyAll()

调用 notifyAll()比 notify()更安全。

使用 notify()而不是 notifyAll()是一种优化，使用 notify()时，在众多等待同一个锁的任务中只有一个被唤醒，因此使用 notify()必须确保唤醒的是恰当的任务。为了使用 notify()，所有任务必须等待相同的条件，因为如果你有多个任务等待不同的条件，那么就不会知道是否唤醒了恰当的任务。

notifyAll()**因某个特定锁而被调用**时，只有**等待这个锁**的任务才会被唤醒。

类 Timer：一种工具，线程用其安排以后在后台线程中执行的任务。可安排任务执行一次，或者定期重复执行。

与每个 Timer 对象相对应的是单个后台线程，用于顺序地执行所有计时器任务。计时器任务应该迅速完成。如果完成某个计时器任务的时间太长，那么它会“独占”计时器的任务执行线程。因此，这就可能延迟后续任务的执行，而这些任务就可能“堆在一起”，并且在上述不友好的任务最终完成时才能够被快速连续地执行。

void cancel() 终止此计时器，丢弃所有当前已安排的任务。

int purge()从此计时器的任务队列中移除所有已取消的任务。

void schedule(TimerTask task, Date time)安排在指定的时间执行指定的任务。

void schedule(TimerTask task, Date firstTime, long period)安排指定的任务在指定的时间开始进行重复的固定延迟执行。

void schedule(TimerTask task, long delay)安排在指定延迟后执行指定的任务。

void schedule(TimerTask task, long delay, long period)安排指定的任务从指定的延迟后开始进行重复的固定延迟执行。

void scheduleAtFixedRate(TimerTask task, Date firstTime, long period)安排指定的任务在指定的时间开始进行重复的固定速率执行。

void scheduleAtFixedRate(TimerTask task, long delay, long period)安排指定的任务在指定的延迟后开始进行重复的固定速率执行。

21.5.3 生产者与消费者

例子：一个饭店，有一个厨师和一个服务员。这个服务员必须等待厨师准备好膳食，当厨师准备好时，会通知服务员，之后服务员上菜，然后返回继续等待。其中：厨师代表生产者，而服务员代表消费者。两个任务必须在膳食被生产和消费时进行握手，而系统必须以有序的方式关闭。

shutdownNow()将向所有 ExecutorService 启动的任务发送 interrupt()，任务并没有因为 interrupt()之后立即关闭，因为当任务试图进入一个（可中断的）【sleep】阻塞操作时，这个中断只能抛出 InterruptedException。如果没有可中断阻塞操作,那么任务将回到 run()循环的顶部，并由于 Thread.interrupted()测试而退出。

使用显式的 Lock 和 Condition

Java SE5 的 java.util.concurrent 类库中使用互斥并允许任务挂起的基本类是 Condition,可以通过在 Condition 上调用 await()来挂起一个任务。当外部条件发生变化，意味着某个任务应该继续执行，通过调用 signal()或 signalAll()来唤醒任务。

Lock 和 Condition 对象只有在更加困难的多线程问题中才是必需的。

Condition 接口：

Condition 将 Object 监视器方法（wait、notify 和 notifyAll）分解成截然不同的对象，以便通过将这些对象与任意 Lock 实现组合使用，为每个对象提供多个等待 set（wait-set）。其中,Lock 替代了 synchronized 方法和语句的使用,Condition 替代了 Object 监视器方法的使用。Condition 实例实质上被绑定到一个锁上。

void await()造成当前线程在接到信号或被中断之前一直处于等待状态。

boolean await(long time, TimeUnit unit)造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。

long awaitNanos(long nanosTimeout)造成当前线程在接到信号、被中断或到达指定等待时间之前一直处于等待状态。

void awaitUninterruptibly()造成当前线程在接到信号之前一直处于等待状态。

boolean awaitUntil(Date deadline)造成当前线程在接到信号、被中断或到达指定最后期限之前一直处于等待状态。

void signal()唤醒一个等待线程。

void signalAll()唤醒所有等待线程。

21.5.4 生产者-消费者队列

使用阻塞队列来解决任务协作问题，阻塞队列（BlockingQueue）是一个支持两个附加操作的队列，附加操作是：获取元素时等待队列变为非空，以及存储元素时等待空间变得可用。常用于生产者-消费者的场景，**阻塞队列就是生产者存放元素的容器，而消费者也只从容器内拿元素。**

阻塞队列提供的四种处理方式：

方法\ 处理方式	抛出异常	返回 特殊值	一直 阻塞	超时退出
插入方法	add(e)	offer(e)	put(e)	offer(e,time,unit)
移除方法	remove(e)	poll()	take()	poll(time,unit)
检查方法	element(e)	peek()	不可用	不可用

抛出异常：是指阻塞队列满时候,再往队列中插入元素,就会抛出IllegalStateException异常，当队列为空时，从队列获取元素时会抛出NoSuchElementException异常；

返回特殊值：插入方法会返回是否成功，成功为true，移除方法，则是从队列里拿出一个元素，如果没有则返回null；

一直阻塞：当阻塞队列满时，如果生产者线程往队列里put元素，队列会一直阻塞生产者线程，直到拿到数据，或响应中断退出。当队列为空时，消费者试图从队列里take元素，队列也会阻塞线程，直到队列可用；

超时退出：当阻塞队列满时，队列会阻塞生产者线程一段时间，如果超过一定时间，生产者线程就会退出，当阻塞队列空时，同样。

BlockingQueue接口：

boolean **add**(E e)将指定元素插入此队列中（如果立即可行且不会违反容量限制），成功时返回 true，如果当前没有可用的空间，则抛出IllegalStateException。

boolean contains(Object o)如果此队列包含指定元素，则返回true。

int drainTo(Collection<? super E> c)移除此队列中所有可用的元素，并将它们添加到给定collection中。

int drainTo(Collection<? super E> c, int maxElements)最多从此队列中移除给定数量的可用元素，并将这些元素添加到给定collection中。

boolean **offer**(E e)将指定元素插入此队列中（如果立即可行且不会违反容量限制），成功时返回true，如果当前没有可用的空间，则返回false。

boolean **offer**(E e, long timeout, TimeUnit unit)将指定元素插入此队列中，在到达指定的等待时间前等待可用的空间（如果有必要）。

E **poll**(long timeout, TimeUnit unit)获取并移除此队列的头部，在指定的等待时间前等待可用的元素（如果有必要）。

void **put**(E e)将指定元素插入此队列中，**将等待可用的空间**（如果有必要）。

int remainingCapacity()返回在无阻塞的理想情况下（不存在内存或资源约束）此队列能接受的附加元素数量；如果没有内部限制，则返回Integer.MAX_VALUE。

boolean **remove**(Object o)从此队列中移除指定元素的单个实例（如果存在）。

E **take**()获取并移除此队列的头部，**在元素变得可用之前一直等待**（如果有必要）。

JDK7提供了7个阻塞队列。分别是：

1、ArrayBlockingQueue：一个由数组结构组成的有界阻塞队列

2、LinkedBlockingQueue：一个由链表结构组成的有界阻塞队列

3、PriorityBlockingQueue：一个支持优先级排序的无界阻塞队列

4、DelayQueue：一个使用优先级队列实现的无界阻塞队列

5、SynchronousQueue：一个不存储元素的阻塞队列；一种阻塞队列，其中每个插入操作

必须等待另一个线程的对应移除操作。

6、LinkedTransferQueue：一个由链表结构组成的无界阻塞队列

7、LinkedBlockingDeque：一个由链表结构组成的双向阻塞队列

吐司 BlockingQueue

有一台机器具有三个任务：一个制作吐司、一个给吐司抹黄油，另一个给摸了黄油的吐司上涂果酱。

enum Status{ DRY,BUTTERED,JAMMED }

ToastQueue dryQueue = new ToastQueue(), bufferedQueue = new ToastQueue(), finishedQueue = new ToastQueue();

示例没有任何显式的同步（即 Lock 对象或 synchronized 关键字的同步），因为同步由队列（因为其内部是同步的）和系统的设计隐式地管理了——每片 Toast 在任何时刻都只由一个任务在操作。因为队列的阻塞，使得处理过程将被自动的挂起和恢复。在使用显式的 wait()和 notifyAll()时存在的类和类之间的耦合被消除了，每个类都只和它的 BolockingQueue 通信。

21.5.5 任务间使用管道进行输入/输出

通过 PipedWriter 类和 PipedReader 类以管道的形式对线程间的输入/输出提供了支持。这个模型可以看成是“生产者—消费者”问题的变体，管道就是一个封装好的解决方案，管道基本上是一个阻塞队列。

管道流是可中断的。

Sender 和 Receiver 代表了需要互相通信两个任务。Sender 创建了一个 PipedWriter，它是一个单独的对象；但是对于 Receiver，PipedReader 的建立必须在构造器中与一个 PipedWriter 相关联。Sender 把数据放进 Writer，然后休眠一段时间(随机数)，然而，Receiver 没有 sleep()和 wait()。但当它调用 read()时，如果没有更多的数据，**管道自动阻塞**。

sender 和 receiver 是在 main()中启动的，即对象构造彻底完毕以后。如果启动了一个没有构造完毕的对象，在不同的平台上管道可能会产生不一致的行为，BlockingQueue 使用起来更加健壮而容易。

21.6 死锁

任务之间**互相等待**.的**连续循环**，没有哪个线程能继续，这被称之为**死锁**。

真正的问题是程序可能看起来运行良好，但具有潜在死锁危险，缺陷潜伏在程序里，以一种可以肯定是很难重现的方式发生。

满足以下四个条件，就会发生死锁：

1）互斥条件。任务使用的资源中至少有一个是不能共享的。

2）至少有一个任务它必须持有一个资源且正在等待获取一个当前被别的任务持有的资源。

3）资源不能被任务抢占，任务必须把资源释放当作普通事件。

4）必须有循环等待，一个任务等待其他任务所持有的资源。

防止死锁最容易的方法是破坏第 4 个条件。

Java 死锁并没有提供语言层面上的支持，能否避免死锁，取决于自己。

21.7 新类库中的构件

21.7.1 CountDownLatch

被用来同步一个或多个任务，强制它们等待由其他任务执行的一组操作完成。

可以向 CountDownLatch 对象设置一个初始计数值，任何在这个对象上调用 wait()的方法都将阻塞，直至这个计数值到达 0.其他任务在结束其工作时，可以在该对象上调用 countDown()来减小这个计数值。CountDownLatch 被设计只能触发一次，计数值不能被重置。

CountDownLatch 的典型用法是将一个程序分为 n 个互相独立的可解决任务，并创建值为 0 的 CountDownLatch。每个任务完成时，都会在这个**锁存器**上调用 countDown()。等待问题

被解决的任务在这个锁存器上调用 await()，将他们拦住，直至锁存器计数结束。

CountDownLatch 类：

一个同步辅助类，在完成一组正在其他线程中执行的操作之前，它允许一个或多个线程一直等待。用给定的计数初始化 CountDownLatch。由于调用了 countDown()方法，所以在当前计数到达零之前，await 方法会一直受阻塞。之后，会释放所有等待的线程，await 的所有后续调用都将立即返回。这种现象只出现一次——计数无法被重置。

CountDownLatch 是一个通用同步工具，它有很多用途。将计数 1 初始化的 CountDownLatch 用作一个简单的开/关锁存器，或入口：在通过调用 countDown()的线程打开入口前，所有调用 await 的线程都一直在入口处等待。用 *N* 初始化的 CountDownLatch 可以使一个线程在 *N* 个线程完成某项操作之前一直等待，或者使其在某项操作完成 N 次之前一直等待。

CountDownLatch 的典型用法是将一个程序分为 n 个互相独立的可解决任务，并创建值为 0 的 CountDownLatch。当每个任务完成时，都会在这个锁存器上调用 countDown()。等待问题被解决的任务在这个锁存器上调用 await()，将它们拦住，直到锁存器结束。

构造方法摘要：

CountDownLatch(int count)构造一个用给定计数初始化的 CountDownLatch。

方法摘要：

void await()使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断。

boolean await(long timeout, TimeUnit unit)使当前线程在锁存器倒计数至零之前一直等待，除非线程被中断或超出了指定的等待时间。

void countDown()递减锁存器的计数，如果计数到达零，则释放所有等待的线程。

long getCount()返回当前计数。

String toString()返回标识此锁存器及其状态的字符串。

21.7.2 CyclicBarrier

创建一组任务，并行地执行工作，然后在进行下一步骤之前等待，直至所有任务都完成。使得所有任务在栅栏处列队，因此可以一致地向前移动。与 CountDownLatch 类似，只是 CountDownLatch 只触发一次，而 CyclicBarrier 可以多次重用。

一个同步辅助类,它允许一组线程互相等待,直到到达某个公共屏障点【栅栏处】(common barrier point)。在涉及一组固定大小的线程的程序中，这些线程必须不时地互相等待，此时 CyclicBarrier 很有用。因为该 barrier 在释放等待线程后可以重用，所以称它为**循环**的 barrier。

CyclicBarrier 支持一个可选的 Runnable 命令，在一组线程中的最后一个线程到达之后（但在释放所有线程之前），该命令只在每个屏障点运行一次。若在继续所有参与线程之前更新共享状态，此*屏障操作*很有用。

CyclicBarrier 类：

构造方法摘要

CyclicBarrier(int parties)创建一个新的 CyclicBarrier，它将在给定数量的参与者（线程）处于等待状态时启动，但它不会在启动 barrier 时执行预定义的操作。

CyclicBarrier(int parties, **Runnable** barrierAction)创建一个新的 CyclicBarrier，它将在**给定数量的参与者**（线程）处于等待状态时启动，并在**启动 barrier 时执行给定的屏障操作**，该操作由最后一个进入 barrier 的线程执行。

方法摘要

int await()在所有参与者都已经在此 barrier 上调用 await 方法之前，将一直等待。

int await(long timeout, TimeUnit unit)在所有参与者都已经在此屏障上调用 await 方法之前将一直等待,或者超出了指定的等待时间。

int getNumberWaiting()返回当前在屏障处等待的参与者数目。

int getParties()返回要求启动此 barrier 的参与者数目。

boolean isBroken()查询此屏障是否处于损坏状态。

void reset()将屏障重置为其初始状态。

21.7.3 DelayQueue

Delayed 接口：extends Comparable<Delayed>

一种混合风格的接口，用来标记那些应该在给定延迟时间之后执行的对象。

long **getDelay**(TimeUnit unit)返回与此对象相关的剩余延迟时间，以给定的时间单位表示。【getDelay()中，希望使用的单位是作为 unit 参数传递进来的，你使用它将当前时间与触发时间之间的差转换为调用者要求的单位，而无需知道这些单位是什么，策略模式简单示例】

DelayQueue<E extends Delayed>类：

Delayed 元素的一个**无界阻塞队列**，用于放置实现了 Delayed 接口的对象，其中的对象只能在其到期时才能从队列中取走。该队列的**头部**是延迟期满后保存时间最长的 Delayed 元素。如果延迟都还没有期满，则队列没有头部，并且 poll 将返回 null。当一个元素的 getDelay(TimeUnit.NANOSECONDS)方法返回一个小于等于 0 的值时，将发生到期。即使无法使用 take 或 poll 移除未到期的元素，也不会将这些元素作为正常元素对待。

构造方法摘要

DelayQueue()创建一个最初为空的新 DelayQueue。

DelayQueue(Collection<? extends E> c)创建一个最初包含 Delayed 实例的给定 collection 元素的 DelayQueue。

方法摘要

boolean add(E e)将指定元素插入此延迟队列中。

void clear()自动移除此延迟队列的所有元素。

int drainTo(Collection<? super E> c)移除此队列中所有可用的元素，并将它们添加到给定 collection 中。

int drainTo(Collection<? super E> c, int maxElements)最多从此队列中移除给定数量的可用元素，并将这些元素添加到给定 collection 中。

Iterator<E> iterator()返回在此队列所有元素（既包括到期的，也包括未到期的）上进行迭代的迭代器。

boolean offer(E e)将指定元素插入此延迟队列。

Boolean offer(E e,long timeout,TimeUnit unit)将指定元素插入此延迟队列中。

E peek()获取但不移除此队列的头部；如果此队列为空，则返回 null。

E **poll**()获取并移除此队列的头，如果此队列不包含具有已到期延迟时间的元素，则返回 null。

E **poll**(long timeout,TimeUnit unit)获取并移除此队列的头部，在可从此队列获得到期延迟的元素，或者到达指定的等待时间之前一直等待（如有必要）。

void put(E e)将指定元素插入此延迟队列。

int remainingCapacity()因为 DelayQueue 没有容量限制，所以它总是返回 Integer.MAX_VALUE。

boolean remove(Object o)从此队列中移除指定元素的单个实例（如果存在），无论它是否到期。

int size()返回此 collection 中的元素数。

E take()获取并移除此队列的头部，在可从此队列获得到期延迟的元素之前一直等待（如有必要）。

Object[] toArray()返回包含此队列所有元素的数组。

<T> T[] toArray(T[] a)返回一个包含此队列所有元素的数组；返回数组的运行时类型是指定数组的类型。

21.7.4 PriorityBlockingQueue

一个无界阻塞队列，它使用与类 PriorityQueue 相同的顺序规则，并且提供了阻塞获取操作。虽然**此队列逻辑上是无界的，但是资源被耗尽时试图执行 add 操作也将失败**（导致 OutOfMemoryError）。此类不允许使用 null 元素。依赖自然顺序的优先级队列也不允许插入不可比较的对象（这样做会导致抛出 ClassCastException）。此类及其迭代器可以实现 Collection 和 Iterator 接口的所有可选方法。iterator()方法中提供的迭代器并不保证以特定的顺序遍历 PriorityBlockingQueue 的元素。如果需要有序地进行遍历，则应考虑使用 Arrays.sort(pq.toArray())。

PriorityBlockingQueue<E>类：

构造方法摘要

PriorityBlockingQueue()用默认的**初始容量(11)**创建一个 PriorityBlockingQueue，并根据自然顺序对元素进行排序。

PriorityBlockingQueue (Collection<? extends E> c) 创建一个包含指定 collection 元素的 PriorityBlockingQueue。

PriorityBlockingQueue(int initialCapacity) 使用指定的初始容量创建一个 PriorityBlockingQueue，并根据元素的自然顺序对其元素进行排序。

PriorityBlockingQueue (int initialCapacity, Comparator<? super E> comparator) 使用指定的初始容量创建一个 PriorityBlockingQueue，并根据指定的比较器对其元素进行排序。

方法摘要

boolean add(E e)将指定元素插入此优先级队列。

void clear()完全移除此队列中的所有元素。

Comparator<? super E> comparator()返回用于对此队列元素进行排序的比较器；如果此队列使用其元素的自然顺序，则返回 null。

boolean contains(Object o)如果队列包含指定的元素，则返回 true。

int drainTo(Collection<? super E> c)移除此队列中所有可用的元素，并将它们添加到给定 collection 中。

int drainTo(Collection<? super E> c, int maxElements)最多从此队列中移除给定数量的可用元素，并将这些元素添加到给定 collection 中。

Iterator<E> iterator()返回在此队列元素上进行迭代的迭代器。

boolean offer(E e)将指定元素插入此优先级队列。

boolean offer(E e, long timeout, TimeUnit unit)将指定元素插入此优先级队列。

E peek()获取但不移除此队列的头；如果此队列为空，则返回 null。

E poll()获取并移除此队列的头，如果此队列为空，则返回 null。

E poll(long timeout, TimeUnit unit)获取并移除此队列的头部，在指定的等待时间前等待可用的元素（如果有必要）。

void put(E e)将指定元素插入此优先级队列。

int remainingCapacity()总是返回 Integer.MAX_VALUE，因为 PriorityBlockingQueue 没有容量限制。

boolean remove(Object o)从队列中移除指定元素的单个实例（如果存在）。

int size()返回此 collection 中的元素数。

E take()获取并移除此队列的头部，在元素变得可用之前一直等待（如果有必要）。

Object[] toArray()返回包含此队列所有元素的数组。

<T> T[] toArray(T[] a)返回一个包含此队列所有元素的数组；返回数组的运行时类型是指定数组的类型。

String toString()返回此 collection 的字符串表示形式。

21.7.5 使用 ScheduleExecutor 的温室控制器

ScheduledThreadPoolExecutor 类:

ThreadPoolExecutor，它可另行安排在给定的延迟后运行命令，或者定期执行命令。需要多个辅助线程时，或者要求 ThreadPoolExecutor 具有额外的灵活性或功能时，此类要优于 Timer。

一旦启用已延迟的任务就执行它，但是有关何时启用，启用后何时执行则没有任何实时保证。按照提交的先进先出 (FIFO) 顺序来启用那些被安排在同一执行时间的任务。

虽然此类继承自 ThreadPoolExecutor，但是几个继承的调整方法对此类并无作用。特别是，因为它作为一个使用 corePoolSize 线程和一个无界队列的固定大小的池，所以调整 maximumPoolSize 没有什么效果。

构造方法摘要

ScheduledThreadPoolExecutor(int corePoolSize)使用给定核心池大小创建一个新 ScheduledThreadPoolExecutor。

ScheduledThreadPoolExecutor(int corePoolSize, RejectedExecutionHandler handler)使用给定初始参数创建一个新 ScheduledThreadPoolExecutor。

ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory)使用给定的初始参数创建一个新 ScheduledThreadPoolExecutor。

ScheduledThreadPoolExecutor(int corePoolSize, ThreadFactory threadFactory, RejectedExecutionHandler handler) 使用给定初始参数创建一个新 ScheduledThreadPoolExecutor。

方法摘要

protected <V> RunnableScheduledFuture<V> decorateTask(Callable<V> callable, RunnableScheduledFuture<V> task)修改或替换用于执行 callable 的任务。

protected <V> RunnableScheduledFuture<V> decorateTask(Runnable runnable, RunnableScheduledFuture<V> task)修改或替换用于执行 runnable 的任务。

void execute(Runnable command)使用所要求的零延迟执行命令。

boolean getContinueExistingPeriodicTasksAfterShutdownPolicy()获取有关在此执行程序已 shutdown 的情况下、是否继续执行现有定期任务的策略。

boolean getExecuteExistingDelayedTasksAfterShutdownPolicy()获取有关在此执行程序已 shutdown 的情况下是否继续执行现有延迟任务的策略。

BlockingQueue<Runnable> getQueue()返回此执行程序使用的任务队列。

boolean remove(Runnable task)从执行程序的内部队列中移除此任务（如果存在），从而如果尚未开始，则其不再运行。

<V> ScheduledFuture<V> **schedule**(Callable<V> callable, long delay, TimeUnit unit)创建并执行在给定延迟后启用的 ScheduledFuture。

ScheduledFuture<?> **schedule**(Runnable command, long delay, TimeUnit unit)创建并执行在给定延迟后启用的一次性操作。

ScheduledFuture<?> **scheduleAtFixedRate**(Runnable command, long initialDelay, long period, TimeUnit unit)创建并执行一个在给定初始延迟后首次启用的定期操作，后续操作具有给定的周期;也就是将在 initialDelay 后开始执行,然后在 initialDelay+period 后执行，接着在 initialDelay + 2 * period 后执行，依此类推。

ScheduledFuture<?> **scheduleWithFixedDelay**(Runnable command, long initialDelay, long delay, TimeUnit unit)创建并执行一个在给定初始延迟后首次启用的定期操作，随后，在每一次执行终止和下一次执行开始之间都存在给定的延迟。

void setContinueExistingPeriodicTasksAfterShutdownPolicy(boolean value)设置有关在此执行程序已 shutdown 的情况下是否继续执行现有定期任务的策略。

void setExecuteExistingDelayedTasksAfterShutdownPolicy(boolean value)设置有关

在此执行程序已 shutdown 的情况下是否继续执行现有延迟任务的策略。

void shutdown()在以前已提交任务的执行中发起一个有序的关闭，但是不接受新任务。

List<Runnable> shutdownNow()尝试停止所有正在执行的任务、暂停等待任务的处理，并返回等待执行的任务列表。

<T> Future<T> submit(Callable<T> task)提交一个返回值的任务用于执行，返回一个表示任务的未决结果的 Future。

Future<?> submit(Runnable task)提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。

<T> Future<T> submit(Runnable task, T result)提交一个 Runnable 任务用于执行，并返回一个表示该任务的 Future。