

Chapter 1. 对象导论

1.1. 抽象过程

1、Smalltalk 五个基本特性

- 万物皆为对象
- 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
- 每个对象都有自己的由其他对象构成的存储
- 每个对象都拥有其类型
- 某一特定类型的所有对象都可以接受同样的消息

1.2. 每个对象都有一个接口

- 1、抽象数据类型的运行方式与内置类型几乎完全一致：你可以创建某一类型的变量，然后操作这些变量(称为发送消息或请求)
- 2、事实上，所有面向对象的程序设计语言都使用 `class` 这个关键词来表示数据类型
- 3、程序员通过定义类来适应问题，而不再被迫只能使用现有的用来表示机器中的存储单元的数据类型，可以根据需求，通过添加新的数据类型来扩展编程语言
- 4、接口确定了对某一特定对象所能发出的请求，在程序中必须有满足这些请求的代码，这些代码与隐藏的数据一起构成了实现

1.3. 每个对象都提供服务

- 1、将对象想象成“服务提供者”，程序本身将向用户提供服务，它通过调用其他对象提供的服务来实现这一目的

1.4. 被隐藏的具体实现

- 1、将程序开发人员按照角色分类为：类创建者(那些创建新数据类型的程序员)和客户端程序员(那些在其应用中使用数据类型的类消费者)
- 2、客户端程序员的目标是手机各种用来实现快速应用开发的类
- 3、类创建者的目标是构建类，暴露必须的部分，隐藏其他部分
- 4、访问控制的原因：
 - 让客户端程序员无法触及他们不应该触及的部分
 - 允许库设计者可以改变类内部的工作方式而不用担心会影响到客户端程序员
- 5、Java 用三个关键字在类的内部设定边界：**public private protected**，除此之外还有一个默认访问权限，即 **friendly**，**包访问权限**

1.5. 复用具体实现

- 1、代码复用是面向对象程序设计语言所提供的最了不起的优点之一
- 2、最简单的复用方式就是直接使用该类的一个对象，此外也可以将那个类的一个对象置于某个新类中

- 新的类可以由任意数量、任意类型的其他对象以任意可以实现新的类中想要的功能的方式所组成，这种概念被称为组合，如果组合是动态发生的，通常被称为聚合
 - 组合经常被视为拥有"has a(拥有)"关系
 - 新类的成员对象通常都被声明为 **private**，使得新类的客户端程序员不能访问它们
- 3、在建立新类时，首先考虑组合，因为它简单灵活，有充分的理由时才用继承

1. 6. 继承

Chapter 2. 一切都是对象

- 1、尽管 Java 是基于 C++ 的，但是相比之下，Java 是一种更纯粹的面向对象程序设计语言
- 2、C++之所以成为一种杂合型语言主要是因为它支持与 C 语言的向后兼容，因为 C++是 C 的一个超集，包括许多 C 语言不具备的特性，这些特性使得 C++在某些方面太过复杂

2.1. 用引用操纵对象

- 1、每种编程语言都有自己的操纵内存元素的方式
 - 直接操纵
 - 间接操纵(C++中的指针，引用)
- 2、所有的一切在 Java 这里得到了简化，一切都视为对象，因此可采用固定的语法
- 3、尽管一切都看做对象，但操纵的标识符实际上是对象的一个引用
- 4、与 C++不同，Java 中的引用可以独立存在，而 C++必须绑定到某个对象上

2.2. 必须由你创建所有对象

- 1、通常用 new 操作符来实现这一目的

2.2.1. 存储到什么地方

- 1、有五个不同的地方可以存储
 - **寄存器**: 最快的存储区，它位于不同于其他存储区的地方---处理器内部。我们不能直接控制寄存器，也不能在程序中感受到寄存器存在的迹象(C 和 C++允许向编译器建议寄存器的分配方式)
 - **堆栈**: 位于通用 RAM(随机访问存储器)中，通过堆栈指针可以从处理器哪里获得直接支持，堆栈指针若向下移动，则分配新的内存，若向上移动，释放那些内存，这是一种快速有效的分配存储的方式，仅次于寄存器
 - 创建程序时，Java 系统必须知道存储在堆栈内所有项的确切生命周期，以便上下移动堆栈指针
 - 这一约束限制了程序的灵活性，**所以虽然某些 Java 的数据存储在堆栈中---特别是对象的引用，但是 Java 的对象并不存储于其中**
 - **堆**: 通用的内存池(位于 RAM 区)，用于存放所有的 Java 对象
 - 不同于堆栈，编译器不需要知道存储的数据在堆里存活多长时间，因此在堆里分配有很大的灵活性
 - 这种灵活性是有代价的，用对进行存储分配和清理可能比用堆栈进行存储分配需要更多的时间
 - **常量存储**: 常量值通常直接存放在程序代码内部，这样做是安全的，因为它们永远不会改变
 - 在嵌入式系统中，常量本身会和其他部分离开，在这种情况下，可以选择将其放在 ROM(只读存储器)中
 - **非 RAM 存储**: 如果数据完全存活在程序之外，那么它可以不受程序的任何控制，在程序没有运行时也可以存在
 - 两个基本例子是：流对象和持久化对象
 - 在流对象中，对象转化成字节流，通常被发送给另一台机器

- 在持久化对象中，对象被存放于磁盘，因此即使程序终止，对象也能保持自己的状态，在需要时可以恢复成常规的基于 RAM 的对象

2.2.2. 特例：基本类型

1、用 new 创建一个对象---特别是小的简单的变量，往往不是很有效，**对于这些类型，Java 采用与 C 和 C++相同的方法，也就是说不用 new 来创建变量，而是创建一个并非是引用的“自动”变量，这些变量直接存储“值”，并置于堆栈中，因此更加高效**

2、Java 要确定每种基本类型所占存储空间的大小，它们的大小并不像其他大多数程序语言那样随机器硬件架构的变化而变化，这种所占存储空间大小不变性是 Java 程序比其他大多数语言编写的程序更具可移植性的原因

基本类型	大小	最小值	最大值	包装器类型
boolean	-	-	-	Boolean
char	16-bit	Unicode 0	Unicode 2 ¹⁶ -1	Character
byte	8 bits	-128	+127	Byte
short	16 bits	-2 ¹⁵	+2 ¹⁵ -1	Short
int	32 bits	-2 ³¹	+2 ³¹ -1	Integer
long	64 bits	-2 ⁶³	+2 ⁶³ -1	Long
float	32 bits	IEEE754	IEEE754	Float
double	64 bits	IEEE754	IEEE754	Double
void	-	-	-	Void

3、所有的类型都有正负号，所以不要去寻找无符号数值类型(C++有无符号)

4、boolean 类型所占存储空间的大小没有明确指定，仅定义为能够取字面值 true 和 false

5、高精度数字

- BigInteger: 支持任意精度的整数
- BigDouble: 支持任意精度的定点数
- 这两者属于包装类型，没有对应的基本类型

2.2.3. Java 中的数组

1、几乎所有的程序设计语言都支持数组

2、在 C 和 C++中使用数组是很危险的，**因为 C 和 C++中数组就是内存块**，如果程序要访问自身内存块之外的数组，或者在数组初始化前使用内存，都会产生难以预料的后果

3、Java 的主要目标之一就是安全性，所以许多在 C 和 C++里困扰程序员的问题在 Java 里不会再出现，**Java 确保数组会被初始化**，而且不能再它的范围之外被访问，**这种范围检查，是以每个数组上少量的内存开销以及运行时的下标检查为代价的**。但由此换来的是安全性和效率的提高

4、当创建一个数组对象时，实际上就是**创建了一个引用数组(C++中没有引用数组，只有数组的引用)**，并且每个引用都会自动被初始化为一个特定的值，该值拥有自己的关键字 null

5、还可以创建用来存放基本类型的数组，编译器也能确保这种数组的初始化，**将其所占内存全部置零**

2.3. 永远不需要销毁对象

2.3.1. 作用域

1、作用域决定了在其内定义的变量名可见性和声明周期

- 2、在 C 和 C++ 中，作用域由花括号的位置决定，在作用域里定义的变量只可用于作用域结束之前
- 3、在 C 和 C++ 中，内层作用域会隐藏外层作用域中的同名实体，但 Java 不允许这样做，因为 Java 设计者认为这样会导致程序混乱

2.3.2. 对象的作用域

- 1、Java 对象不具备和基本类型一样的生命周期，当用 new 创建一个 Java 对象时，它可以存活于作用域之外
- 2、事实证明，由 new 创建的对象，只要你需要，就会一直保留下去，这样许多 C++ 编程问题在 Java 中就完全消失了(C++ 中，你不仅必须要确保对象的保留时间与你需要这些对象的时间一样长，而且还必须在你使用完它们后将其销毁)
- 3、Java 有垃圾回收器，用来监视 new 创建的所有对象，并辨别那些不再会被引用的对象(看引用计数是否为 0???)。随后释放这些对象，以便供其他新的对象使用
- 4、垃圾回收器可以消除内存泄露的问题，程序员不必关心对象的销毁

2.4. 创建新的数据类型：类

- 1、大多数面向对象的程序设计语言习惯用关键字 class 来表示"我准备告诉你一种新类型的对象看起来像什么样子"
- 2、在定义它之前不能做太多的事情(不能发送消息)，但是可以创建这种类型的对象(比如在该类型的内部中有一个字段的类型就是该类型)。

2.4.1. 字段和方法

- 1、类中两种种类的元素：字段和方法
- 2、字段
 - 可以是任何类型的对象，可以通过其他引用与其通信，也可以是基本类型的一种
 - 如果是某个对象的引用，则必须初始化该引用，以便使其与一个实际的对象相关联
 - 每个对象都有用来存储其字段的空间，普通字段不能在对象间共享
- 3、基本成员默认值
 - 若某个成员是基本数据类型，即使没有进行初始化，Java 也会确保它获得一个默认值
 - 算数类型赋值为 0，boolean 赋值为 false 等等，就是内存为 0 的那个值
- 4、局部变量必须初始化，否则值是任意的(与 C 与 C++ 一样)，但是如果忘记赋初值，Java 会返回一个错误，告诉你变量没有初始化，这也是 Java 优于 C++ 的地方

2.5. 方法、参数和返回值

- 1、许多程序设计语言(像 C 和 C++)用函数这个术语来描述命名子程序
- 2、在 Java 中用方法这个术语来表示"做某些事情的方式"，实际上，看做函数也无妨
- 3、Java 的方法决定了一个对象能够接受什么样的消息，方法的基本组成部分包括：名称，参数，返回值和方法体
- 4、**Java 中的方法只能作为类的一部分来创建，非 static 方法只有通过对象才能被调用**，且这个对象必须能执行这个方法，如果试图在某个对象上调用它并不具备的方法，那么在编译时就会得到一条错误信息

2.5.1. 参数列表

1、像 Java 中任何传递对象的场合一样，参数列表中传递的也是引用，并且引用的类型必须正确

2、`return` 关键字包括两方面：

- 已经做完，离开此方法
- 如果产生了一个值，这个值要放在 `return` 的后面

3、对于 `void` 的返回类型，`return` 关键字的作用只是用来退出方法

2.6. 构建一个 Java 程序

2.6.1. 名字的可见性

1、C++通过几个关键字引入命名空间的概念

2、Java 采用了一种全新的方法来避免名字冲突的问题

- Java 设计者希望程序员反过来用自己的 Internet 域名，因为这样可以保证它们肯定是独一无二的
- 这种机制意味着所有的文件都能够存活于它们自己的名字空间内，而且同一个文件内的每个类都有唯一的标识符---Java 语言本身已经解决了这个问题

2.6.2. 运用其他构件

1、如果想要在自己的程序里使用预先定义好的类，那么编译器就必须知道怎么定位它们

- **这个类可能在发出调用的那个源文件中(.java???)，在这种情况下，就可以直接使用这个类，即使这个类在文件的后面才被定义(Java 消除了所谓的"前向引用"的问题)**

2、使用关键字 `import` 来准确的告诉编译器你想要的类是什么

- 导入一个名字： `import java.util.ArrayList;`
- 导入一个类群： `import java.util.*;`

2.6.3. `static` 关键字

1、希望某个方法不与包含它的任何对象关联在一起，即使没有创建对象也能调用这个方法，或者想为某特定域分配单一的存储空间，而不去考虑究竟要创建多少对象

2、`static` 关键字：

- 意味着这个域或方法不会与包含它的那个类的任何对象关联在一起
- 即使从未创建某个类的对象也可以调用其 `static` 方法或访问其 `static` 域
- 只需要将 `static` 关键字放在定义之前，就可以将字段或方法定义为 `static`

3、使用方法：

- 点运算符`.`
- 通过类名：首选
- 通过对象

4、和其他任何方法一样，`static` 方法可以创建或使用其类型相同的被命名对象，因此 `static` 方法常常拿来做"牧羊人"的角色，负责看护与其隶属同一类型的实例群

2.7. 你的第一个 Java 程序

1、每个程序的开头，必须声明 `import` 语句，以便引入在文件代码中需要用到的**额外类**

- 之所以说额外，是因为有一个特定类会自动导入到每一个 Java 文件中：java.lang

2.8. 注释和嵌入式文档

1、Java 里面有两种注释风格

- 这一种是传统的 C 语言风格的注释---C++也继承了这种风格，此种注释以'/*'开始，随后是注释内容，并可跨过多行，以'*/'结束
- 第二种风格也是源于 C++，单行注释，以一个'//'起头

2.8.1. 注释文档

1、代码文档撰写的最大问题，就是对文档的维护了，如果文档与代码是分离的，每次修改代码，就需要修改相应的文档，这会成为一件相当乏味的事情

2、解决方法：将代码同文档"链接起来"，最简单的方法就是将所有东西都放在同一个文件内

- 为了实现这一目的，必须使用一种特殊的注释语法来标记文档，此外还需要一个工具，用于提取那些注释，将其转换成有用的形式
- javadoc 便是用于提取注释的工具，它是 JDK 安装的一部分，采用了 Java 编译器的某些技术，查找程序内的特殊注释标签，它不仅解析由这些标签标记的信息，也将毗邻注释的类名或方法抽取出来
- javadoc 输出的是一个 HTML 文件，可以用 Web 浏览器查看

2.8.2. 语法

1、所有 javadoc 命令只能在'/**'注释中出现，和通常一样，注释结束于'*/'

2、使用 javadoc 的方式主要有两种：嵌入 HTML 或使用文档标签

3、独立文档标签是一些以@开头的命令，且要置于注释行的最前面(不算前导*之后的最前面)

4、行内文档标签则可以出现在 javadoc 注释中的任何地方，它们也是以@开头，但要在花括号内

5、javadoc 只能为 public 何 protected 成员进行文档注释，private 和包内可访问成员的注释会被忽略掉，因为只有 public 和 protected 成员才能在文件外被使用

2.8.3. 嵌入式 HTML

1、在文档注释中，位于每一行开头的星号和前导空格都会被 javadoc 丢弃，javadoc 会对所有内容重新格式化，使其与标准的文档外观一致

2、不要再嵌入式 HTML 中使用标题标签，例如<hl>或<hr>，因为 javadoc 会插入自己的标题，而你的标题可能与它们发生冲突

2.8.4. 一些标签示例

1、@see：引用其他类

2、{@linkpackage.class#member_label}：与@see 类似，只是作用于行内，并且是用"label"作为超链接文本而不用"See Also"

3、{@docRoot}：该标签产生到文档本目录的相对路径，用于文档树页面的显式超链接

4、{@inheritDoc}：该标签从当前这个类的最直接的基类中继承相关文档到当前的文档注释中

- 5、**@version:** 你认为适合包含在版本说明中的重要信息
- 6、**@author:** 作者
- 7、**@since:** 允许指定程序代码最早使用的版本
- 8、**@param:** 参数列表
- 9、**@return:** 返回值的含义
- 10、**@throws:** 异常说明
- 11、**@deprecated:** 指出一些旧特性已由改进的新特性所取代

2.9. 编码风格

- 1、类名首字母大写，如果由几个单词构成，将它们拼在一起，其中每个单词的首字母都采用大写形式
- 2、其他名字与类名相似，只是标识符第一字母小写

Chapter 3. 操作符

- 1、在最底层，Java 中的数据是通过使用操纵符来操作的
- 2、Java 是建立在 C++ 基础之上的

3. 1. 更简单的打印语句

- 1、`System.out.println()`
- 2、`System.out.print()`

3. 2. 使用 Java 操作符

- 1、几乎所有操作符都只能操作"基本类型"
- 2、"=","==","!="等操作符可以操作所有对象
- 3、`String` 类支持 "+" 与 "+="

3. 3. 优先级

- 1、成员访问运算符'.'
- 2、单目运算符
- 3、乘除模
- 4、加减
- 5、移位
- 6、关系运算符
- 7、相等关系运算符
- 8、位与
- 9、位异或
- 10、位或
- 11、逻辑与
- 12、逻辑或
- 13、条件
- 14、赋值
- 15、符合赋值
- 16、`throw`
- 17、逗号

3. 4. 赋值

1、概念

- 取右边的值，把它复制给左边的值
- 右值可以使任何常数，变量或表达式
- 左值必须是一个明确的，已命名的变量(有内存空间)

2、对基本类型的赋值

- **基本类型存储了实际的数值**，而不是指向对象的引用，赋值符合我们的预期

➤ "对象"赋值，其实是引用赋值，会导致资源的共享，会导致"别名现象"

3.4.1. 在方法中调用的别名问题

- 1、将一个对象传递给方法也会产生别名问题
- 2、在函数内部改变形参的指向不会导致外部引用的改变，但是在函数内部通过形参改变对象的状态，会导致外部引用指向的对象状态发生改变

3.5. 算数操作符

- 1、Java 的基本算数操作符与其他大多数程序设计语言是相同的

3.5.1. 一元加、减操作符

- 1、属于单目运算符，优先级大于双目运算符(包括乘除模加减)

3.6. 自动递增和递减

- 1、对于前缀递增和递减，会先执行运算，再生成值
- 2、对于后缀递增和递减，会先生成值，再执行运算
- 3、与 C++ 不同，前缀和后缀在 Java 中并没有左右值的区别

3.7. 关系运算符

- 1、关系操作符生成一个 boolean 结果

3.7.1. 测试对象的等价性

- 1、想要比较两个对象的内容是否相同，必须要使用 `obj1.equals(obj2)`
- 2、想要比较两个基本类型的內容是否相同，必须使用 `obj1==obj2`
- 3、对两个对象直接用`==`操作符，其意义是比较两个引用是否指向同一个对象

3.8. 逻辑操作符

- 1、`&&` `||` `!`
- 2、与 C++ 不同，不可将一个非布尔值当做布尔值在逻辑表达式中使用
- 3、如果在应该使用 `String` 值的地方使用了布尔值，布尔值会自动转换成适当的文本形式(`true`,`false`)

3.8.1. 短路

- 1、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
- 2、逻辑表达式靠后的部分有可能不会被运算

3.9. 直接常量

- 1、一般来说，如果在程序里使用了"直接常量"，编译器可以准确地知道要生成什么样的类型

2、有时候会产生模棱两可的意义，此时必须对编译器加以"指导"，附加额外的信息

➤ 后缀

- `I(L)`: 代表 `long`, 建议用大写, 小写 `I` 与 `1` 容易混淆
- `f(F)`: 代表 `float`
- `d(D)`: 代表 `double`

➤ 前缀

- `0x(0X)`: 十六进制
- `0`: 八进制

3、通过 `Integer` 与 `Long` 的静态方法 `toBinaryString()` 可以将十六进制与八进制的数以二进制的方式表示，当然这个结果是一个 `String`

3.9.1. 指数记数法

1、 $a \times 10^{+/-b}$

3.10. 按位操作符

1、按位操作符用来操作整数基本类型数据中单个 `bit`, 即二进制位

2、按位操作符会对两个参数中对应的位执行布尔代数运算，并最终生成一个结果

3、按位操作符来源于 C 语言面向底层的操作，在这种操作中经常要直接操纵硬件，设置硬件寄存器内的二进制位

3.11. 移位操作符

1、移位操作符的运算对象也是二进制的"位", `bit`

2、移位操作符只可用来处理整数类型

- 左移位操作符 "`<<`" 能按照操作符右侧指定的位数将操作符左边的操作数向左移动(在低位补 0), **并非循环移位哦**
- "有符号"右移位操作符 "`>>`" 则按照操作符右侧指定的位数将操作符左边的操作数向右移动
- **"有符号"右移位操作符使用"符号扩展": 若符号为正, 则在高位插入 0, 若符号为负, 则在高位插入 1**
- Java 增设了一种"无符号"右移位操作符, 无论正负, 都在高位插入 0

3、对 `char`、`byte`、`short` 类型的数值进行移位处理，在移位之前，它们会被转换为 `int` 类型

3.12. 三元操作符 `if-else`(条件语句)

1、`boolean-exp? value0:value1`

3.13. 字符串操作符`+`和`-`

1、C++引入操作符重载，以便 C++程序员可以为几乎所有操作符增加功能

2、Java 程序员不能像 C++ 和 C# 那样实现自己的重载操作符

3、如果表达式以字符串开头，那么后续所有操作数都必须是字符串类型

3.14. 使用操作符时常犯的错误

- 1、即使对表达式如何计算有点不确定，也不愿意使用括号
- 2、`==`写成了`=`

3.15. 类型转换操作符

- 1、在适当的时候，Java 会将一种数据类型自动转换成另一种
 - 为某浮点数赋以一个整数值，编译器会将 `int` 转换成 `float`
- 2、显式类型转换：将希望得到的**数据类型**置于圆括号内
- 3、相比于 C++，Java 中类型转换是一种比较安全的操作
 - 如果执行**窄化转换**的操作，就有可能面临信息丢失的危险，此时编译器会强制我们进行类型转换，表明：这可能是一件危险的事情，但无论如何要这么做，必须显式地进行类型转换
 - 如果执行**扩展转换**的操作，则不必进行显式的类型转换，因为新类型肯定能容纳原来类型的信息，不会造成信息丢失
- 4、Java 允许我们把任何基本数据类型转换成别的基本数据类型，但布尔型除外，布尔类型根本不允许进行任何类型的转换

3.15.1. 截尾和舍入

- 1、将 `float` 或 `double` 类型转换为整型值时，总是对该数字执行截尾
- 2、若要得到舍入的结果，就要使用 `java.lang.Math.round()`方法

3.15.2. 提升

- 1、如果对基本数据类型执行算数运算或按位运算，只要比 `int` 小的类型，在运算前，这些值会自动转换成 `int`，最终生成的结果就是 `int`
- 2、如果想把结果赋值给较小的类型，就必须使用类型转换，因为可能出现信息丢失
- 3、通常表达式中较大的数据类型决定了表达式最终结果的数据类型

3.16. Java 没有 `sizeof`

- 1、在 C 和 C++ 中，需要使用 `sizeof()` 的最大原因是为了“移植”，不同的数据类型在不同的机器上可能有不同的大小
- 2、Java 不需要 `sizeof()` 操作符来满足这方面的需要，因为所有数据类型在所有机器中的大小是相同的，我们不必考虑移植的问题，因为它已经被设计在语言中了

Chapter 4. 控制执行流程

1、就像有知觉的生物一样，程序必须在执行过程中控制它的世界，并做出选择，在 Java 中，要使用执行控制语句来做出选择

4. 1. true 和 false

1、不同于 C++，Java 不允许我们将一个数字作为布尔值用

4. 2. if-else

1、语法

```
if(Boolean-expression)
    statement
```

```
if(Boolean-expression)
    statement
else
    statement
```

4. 3. 迭代

1、while、do-while、for 用来控制循环，有时将它们划分为迭代语句，语句会重复执行，直到起控制作用的布尔表达式得到假的结果为止

2、while 语法

```
while(Boolean-expression)
    statement
```

4. 3. 1. do-while

1、语法

```
do
    statement
while(Boolean-expression);
```

4. 3. 2. for

1、语法

```
for(initialization;Boolean-expression;step)
    statement
```

4. 3. 3. 逗号操作符

1、注意，不是逗号分隔符，逗号用作分隔符时用来分隔函数的不同参数

2、Java 里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，可以使用一系列逗号分隔的语句，而且那些语句均会独立执行

3、在初始化部分可以拥有任意数量，相同类型的变量定义，注意类型名(即使相同)只能出现

一次

4. 4. Foreach 语法

1、语法

```
for(subClass i:SuperClassObj)  
    statement
```

2、String 类有一个 toCharArray() 的方法，将其转化为 char 数组，**可以对这个数组执行 foreach 语句，但不能直接对 String 对象执行 foreach 语句(与 C++不同)**

4. 5. return

1、Java 中有多个关键词表示无条件分支，它们只是表示这个分支无需任何测试即可发生，这些关键词包括 return、break、continue 和一种与其他语言的 goto 类似的跳转到标号语句的方式

2、return 有两方面的用途

- 一方面指定一个方法返回什么值
- 另一方面他会导致当前的方法退出，并返回那个值

4. 6. break 和 continue

1、在任何迭代语句的主体部分，都可以用 break 和 continue 来控制循环的流程

- break：用于强行退出循环，不执行循环中剩余的语句
- continue：停止当前执行的迭代，然后退回循环其实处，开始下一次迭代

4. 7. 臭名昭著的 goto

1、Java 编译器生成它自己的“汇编代码”，但是这个代码是运行在 Java 虚拟机上的，而不是直接运行在 CPU 硬件上

2、少数情况下，goto 还是组织控制流程的最佳手段

3、**goto 仍是 Java 中的一个保留字，但在语言中并未使用它，Java 没有 goto**

4、Java 能完成类似于跳转的操作，这与 break 和 continue 两个关键字有关

- continue label1;
- break label1;

5、在 Java 中，标签起作用的唯一的地方刚好是在迭代语句之前，即在标签和迭代之间插入任何语句都是非法的

6、break 和 continue 本身只作用于关键字所在的循环

4. 8. switch

1、语法

```
switch(integral-selector) {  
    case integral-value1:statement;break;  
    case integral-value2:statement;break;
```

```
...
default:statement;
}
```

- 2、`switch` 要求使用一个选择因子，必须是 `int` 或者 `char` 那样的整型
- 3、可以协助 `enum` 与 `switch` 实现非整数类型的选择语法

Chapter 5. 初始化与清理

- 1、随着计算机革命的发展，"不安全"的变成方式已经逐渐成为变成代价高昂的主因之一
- 2、初始化和清理正是设计安全的两个问题
- 3、C++一如了构造器(构造函数)的概念，这是一个在创建对象时被自动调用的特殊方法，Java 中也采用了构造器，并额外提供了"垃圾回收器"，对于不再使用的内存资源，垃圾回收器能自动将其释放

5.1. 用构造器确保初始化

- 1、Java 会在用户有能力操作对象之前自动调用相应的构造器，保证了初始化的进行
- 2、构造器的名称必须与类名完全相同，因此每个方法首字母小写的编码风格不适用于构造器
- 3、不接受任何参数的构造器叫做默认构造器，Java 文档中通常使用术语无参构造器
- 4、从概念上讲，初始化与创建时彼此独立的，在 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

- 1、大多数程序设计语言(尤其是 C)要求为每个方法都提供独一无二的标识符
- 2、在 Java(C++)里，构造器是强制重载方法名的另一个原因

5.2.1. 区分重载方法

- 1、规则：每个重载的方法都必须有一个独一无二的参数类型列表，甚至顺序的不同也足以区分两个方法

5.2.2. 涉及基本类型的重载

- 1、基本类型能从一个较小的类型自动提升至一个较大的类型
- 2、有精确匹配时，会选择精确匹配的函数
- 3、无精确匹配时，会提升基本数据类型来实现匹配

5.2.3. 不能以返回值区分重载方法

- 1、根据返回值来区分重载方法是行不通的

5.3. 默认构造器

- 1、默认构造器(无参构造器)，是没有形参的，它的作用是创建一个默认对象
- 2、如果你的类中没有构造器，编译器会自动帮你创建一个默认构造器

5.4. this 关键字

- 1、为了能用简便、面向对象的语法来编写代码，即---"发送消息给对象"，编译器做了一些幕后工作
- 2、this

- `this` 关键字只能在方法内部调用，表示对调用方法的那个对象的引用
- `this` 的用法和其他对象的引用并无不同
- 如果在方法内部调用同一个类的另一个方法，不必使用 `this`(但也可以加)，直接调用即可
- **不能对 `this` 赋值，即 `this` 只能指向调用它的那个对象(非静态方法)，在静态方法中不能出现 `this`**

5.4.1. 在构造器中调用构造器

- 1、在一个构造器中调用**该类**另一个构造器，可用 `this` 后跟参数列表
- 2、在一个构造器中调用超类的构造器，可用 `super` 后跟参数列表
- 3、**必须将构造器的调用放在最起始处，否则编译器会报错**
- 4、当形参中的名字与类中字段名字相同时(形参会隐藏字段的名字，即在该方法中使用该名字就是指的形参)，可以显式调用 `this.obj` 来表示引用类中定义的字段

5.4.2. static 的含义

- 1、`static` 的方法就是没有 `this` 的方法
- 2、**在 `static` 方法内部不能调用非静态方法**(不能直接写函数名的方式调用，但是可以创建一个该类的对象，然后通过这个对象调用)

5.5. 清理：终结处理和垃圾回收

- 1、Java 垃圾回收器负责回收无用对象占据的内存资源
- 2、垃圾回收器只知道释放那些由 `new` 分配的内存，所以它不知道如何释放特殊的内存(不是用 `new` 分配的内存)
- 3、Java 允许在类中定义一个**名为 `finalize()`的方法(属于 `Object` 的 `protected` 方法)**，它的工作原理：
 - 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用的内存
- 4、注意 `finalize()`并不是 C++中的析构函数，在 C++中，对象一定会被销毁(如果程序没有缺陷的话)，在 Java 里的对象并非总是被垃圾回收
 - 对象可能不被垃圾回收
 - 垃圾回收并不等于“析构”
- 5、也许，只要程序没有濒临存储空间用完的那一刻，对象占用的空间就总也得不到释放，如果程序执行结束，并且垃圾回收器一直都没有释放你创建的任何对象的存储空间，则随着程序的退出，那些资源也会全部交还给操作系统，这个策略是恰当的，因为垃圾回收器本身也有开销

5.5.1. `finalize()`的用途何在

- 1、垃圾回收器只与内存有关
 - 使用垃圾回收器的唯一原因就是回收程序不再使用的内存，所以对于垃圾回收期有关的任何行为来说(尤其是 `finalize()`方法)，它们也必须同内存及其回收有关
 - 将对 `finalize()`的需要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为对象分配了内存空间
 - 之所以要用 `finalize()`，是由于在分配内存时，可能采用了类似 C 语言中的做法，而

非 Java 中的通常做法

- 这种情况主要发生在使用"本地方法"的情况下
- 本地方法：一种在 Java 中调用非 Java 代码的方式，本地方法目前只支持 C 和 C++
但 C 和 C++ 可以调用其他原因的代码，所以实际上可以调用任何代码
- 在非 Java 代码中，也许会调用 C 的 malloc() 函数来分配存储空间，而且除非调用了 free() 函数，否则存储空间将得不到释放，从而造成内存泄露，因此需要在 finalize 中调用 free

5.5.2. 你必须实施清理

- 1、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
- 2、如果希望进行除释放存储空间之外的清理工作，还得明确调用某个恰当的 Java 方法
- 3、无论是"垃圾回收"还是"终结"，都不保证一定会发生，如果 Java 虚拟机并未面临内存耗尽的情况，它是不会浪费时间去执行垃圾回收以恢复内存的

5.5.3. 条件终结

- 1、通常不能指望 finalize()，因为它调用时刻不是你能控制的，除非你显式调用
- 2、必须创建其他的"清理"方法，并明确地调用它们
- 3、finalize() 可以使得缺陷被发现：
 - 比如有一个对象代表了一个打开的文件，可以在 finalize 中进行判断，看文件是否正确关闭
 - 因为当 finalize 被调用的时候意味着内存该回收了，因此清理动作必须在之前完成，如果没完成，说明存在缺陷

5.5.4. 垃圾回收器如何工作

- 1、垃圾回收器对于提高对象的创建速度，有明显的效果，Java 从堆分配空间的速度，可以和其他语言从栈上分配空间的速度相媲美
- 2、Java 的"堆指针"只是简单地移动到尚未分配的区域，其效率比得上 C++ 在堆栈上分配空间的效率，实际过程中，在簿记工作方面有少量额外开销，但比不上查找可用空间的开销大
- 3、垃圾回收器一面回收空间，一面使堆中对象紧凑排列，通过垃圾回收器对对象重新排列，实现了一种告诉的，有无限空间可供分配的堆模型
- 4、其他垃圾回收机制

- 引用计数：
 - 引用计数是一种简单但速度很慢的垃圾回收技术
 - 每个对象含有一个引用计数器，当有引用连接到对象时，引用计数+1，当引用离开作用域或被置位 null 时，引用计数-1
 - 虽然引用计数开销不大，但是垃圾回收器会在含有全部对象的列表上循环查找引用计数为 0 的对象，将其释放
 - 引用计数常用来说明垃圾收集的工作方式，但未应用于任何一种 Java 虚拟机中
- 停止-复制
 - 先暂停程序的运行，将所有存货的对象从当前堆复制到另一个堆，没有被复制的全部都是垃圾，当对象被复制到新堆时，它们是一个挨着一个的，
 - 把对象从一处移到另一处，引用必须修正
 - 这种复制式回收器，效率较低：维护的空间比实际需要多一倍；在程序进入稳定状态只会产生极少垃圾，但是仍会来回复制，很浪费

- 标记-清扫
 - 在停止-复制的基础上，一些 Java 虚拟机会进行检查，要是没有新垃圾产生，就会转换到另一种工作模式(标记-清扫)
 - 标记-清扫所依据的思路同样是从堆栈和静态存储区出发，遍历所有引用，进而找出所有存货对象，就会给对象设一个标记，只有全部标记工作完成后，清理动作才开始
- 代数
 - 内存分配以较大的"块"为单位，如果对象较大，它会占用单独的块
 - 停止-复制要求在释放旧有对象之前，必须把所有存活对象从旧堆复制到新堆，这将导致大量内存复制行为
 - 有了块后，垃圾回收器在回收的时候，就可以往废弃的块里拷贝对象
 - 每个块都有相应的代数来记录它是否存货

5、Java 虚拟机是自适应的、分代的、停止-复制、标记-清扫式垃圾回收器

6、JIT(Just-In-Time)

- 这种技术可以把程序全部或部分翻译成本地机器码，程序运行速度得以提升
- 当需要装载某个类时，编译器会先找到.class 文件，然后将该类的字节码装入内存
- 采用惰性评估：即时编译器只在必要的时候才编译代码，从不执行的代码将不会被 JIT 编译，代码每次执行都会做一些优化，所以执行的次数越多，速度越快

5. 6. 成员初始化

- 1、Java 尽力保证：所有变量在使用前都能得到恰当的初始化
- 2、对于方法的局部变量，Java 以编译时错误的形式来贯彻这种保证
- 3、类的数据成员
 - 基本类型，那么他们将被初始化为 0
 - 类类型，如果在定义时不将其初始化，那么该引用就会获得一个特殊值 null

5. 6. 1. 指定初始化

- 1、可以在定义类的数据成员的时候为其赋值

5. 7. 构造器初始化

- 1、可采用构造器来进行初始化，在运行时刻，可以调用方法或执行某些动作来确定初值
- 2、**无法阻止自动初始化的进行，它将在构造器被调用之前发生**

```
public class Counter{
    int i;
    Counter() {i=7;}
}
```

- 首先 i 会被置 0，然后变为 7
- 对于所有基本类型和对象引用，包括在定义时已经指定初值的变量，这种情况都是成立的，因此编译器不会强制你一定要在构造器某个地方火灾使用它们之前对元素进行初始化，因为初始化早已得到保证

5.7.1. 初始化顺序

1、在类的内部，变量定义的先后顺序决定了初始化的顺序，即使变量的定义散布于方法定义之间，它们仍然会在任何方法被调用之前得到初始化

2、步骤(对于类内的域)

- 首先，内存区域置零，自动初始化(对于空白 final，这个初始化是不作数的)
- 然后，用定义时的初始值初始化，指定初始化
- 最后，在构造器中初始化，构造器初始化
- C++不同，如果在构造函数初始化列表中有对一个成员进行显式初始化，那么类内初始值是没有用的，只有当初始化列表未有显式初始化时，类内初始值才会生效

5.7.2. 静态数据的初始化

1、无论创建多少个对象，静态数据都只占一份存储区域

2、与 C++ 不同，static 关键字不能应用于局部变量，因此他只能作用于域

3、如果一个域是静态的基本类型域，且没有对它进行初始化，那么它就会获得基本类型的标准处置，如果它是一个对象的引用，那么它的默认初始化就是 null

4、初始化的顺序

- 先静态对象
- 非静态对象

5、总结对象创建过程

- 首次创建类型为 Dog 的对象时(构造器可以看成静态方法)，或者 Dog 类的静态方法 /静态域首次被访问时，Java 解释器必须查找类路径，以定位 Dog.class 文件
- 载入 Dog.class(这将创建一个 Class 对象)，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。
- 当用 new Dog() 创建对象时，首先在堆上为 Dog 对象分配足够的存储空间
- 这块存储空间会清零，这就自动将 Dog 对象中的所有基本类型数据都设成了默认值(对数字来说就是 0，对于布尔类型和字符类型也相同)，而引用则被设置成了 null
- 执行所有出现于字段定义处的初始化动作
- 执行构造器

5.7.3. 显式的静态初始化

1、Java 允许将多个静态初始化动作组织成一个特殊的"静态子句"或"静态块"

```
static{  
    statement  
}
```

- 与其他静态初始化动作一样，这段代码仅执行一次：当首次生成这个类的对象，或者首次访问该类的静态数据成员时(所有的)
- 所有静态数据成员，以及静态子句，其执行的顺序就是代码书写顺序

5.7.4. 非静态实例初始化

1、Java 中也有被称为实例初始化的类似语法，相比于静态子句，少了 static 关键字

2、这种语法对支持匿名内部类的初始化是必须的

3、这种非静态子句，会在每次生成这个类的对象时被执行

5.8. 数组初始化

1、数组只是相同类型，用一个标识符名称封装到一起的一个对象序列或基本类型数据序列，数组是通过方括号下标操作[]来定义和使用的

2、定义一个数组只需要在类名后加上一对空方括号即可：T[]

3、定义数组时，编译器不允许指定数组的大小

4、数组初始化：

- 对于数组，初始化动作可以出现在代码的任何地方

```
int[] a; a=new int[5];  
int[] a1=new int[5];
```

- 也可以用一种特殊的初始化表达式，但它必须在创建数组的地方出现，这种特殊的初始化方式是由一对花括号括起来的值组成的，**初始化列表后最后一个逗号是可选的
(这一特性使维护长列表变得容易)**

```
int[] a2={1,2,3,4,5,};
```

- 可以指定数组大小为 0(C++用 new 分配数组也可以指定大小为 0)

5、与 C++不同，Java 中数组标识符是一个引用

6、所有数组都有一个固定数据成员(length)，可以通过它获知数组内包含了多少个元素，但不能对其修改

7、C 和 C++会接受越界访问，会产生无法预知的行为，但 Java 能保护我们免受这一问题的困扰，一旦访问下标过界，就会出现运行时错误(异常)

8、用 new 在数组里创建元素，尽管创建的是基本类型数组，new 仍然可以工作(**不同于 C++，new 不能用于创建单个基本数据类型**)

9、如果创建了一个非基本类型的数组，那么就创建了一个引用数组(C++不允许创建引用的数组，但存在数组的引用)

5.8.1. 可变参数列表

1、所有的类都是直接或者间接继承自 Object，因此可以用 Object 的数组作为参数

- 需要显式地传递一个数组，比如 new Object[]{arg1,arg2,...}

2、Java SE5 之后，可以这样写：func(Object...args)

- 编译器实际上会为你去填充数组，因此可以用 foreach 来迭代该数组

- 可以将 0 个参数传递给可变参数列表(数组形式或者...形式)

5.9. 枚举类型

1、Java SE5 加入了 enum 关键字，它使得我们在需要群组并使用枚举类型集时，可以很方便地处理，其功能比 C 与 C++完备得多

2、**创建 enum 时，编译器会自动添加一些有用的特性，例如，会创建 toString()方法，以便你可以显示某个 enum 实例的名字**(C++中，无法打印枚举类型的名字，而只能打印枚举类型对应的整型值)

Chapter 6. 访问权限控制

- 1、访问控制(或隐藏具体实现)与"最初的实现并不恰当"有关
- 2、重构代码即重写代码，以使得它更可读，更易理解，并因此更具可维护性
- 3、客户端程序员希望它们的代码保持不变，但是作为类创建者却希望改变某些代码，于是产生了一个问题，如何把变动的事物与不变的事物区分开来
- 4、类库的开发者必须有权限进行修改和改进，并确保客户端代码不会因为这些改变而受到影响
- 5、为了解决这一问题，Java 提供了访问权限修饰词，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，访问权限控制的等级从大到小为：public、protected、包访问权限(没有关键词)和 private。
- 6、作为类库设计员，你会尽可能将一切方法都设定为 private，而仅向客户端程序员公开你愿意让他们使用的方法

6.1. 包：库单元

- 1、包内有一组类，它们在单一的名字空间之下被组织在了一起
- 2、所有类成员的名称都是彼此隔离的，例如 A 类中的方法 f 与 B 类中的方法 f 具有相同的特征标记(参数列表)，它们也不会彼此冲突
- 3、但是如果类名相互冲突就不行了，必须提供一个管理名字空间的机制，在 Java 对名称空间进行完全控制并为每个类创建唯一的标识符组合就成为非常重要的事情
- 4、Java 源代码文件，此文件通常被称为编译单元(转译单元)。每个编译单元都必须有一个后缀名.java，在每个编译单元内最多仅能有一个 public 类(且类名必须与编译单元文件名相同)，如果在该编译单元之中还有额外的类的话，那么在包之外的世界是无法看到这些类的，因为他们不是 public 的，且他们主要为 public 类提供支持

6.1.1. 代码组织

- 1、当编译一个.java 文件时，在.java 文件中的每个类都会有一个输出文件，而该输出文件的名称与.java 文件中每个类的名称相同，只是多了一个.class 后缀。编译少量.java 文件会得到大量的.class 文件
- 2、Java 可运行程序是一组可打包并压缩为一个 Java 文档(JAR, 使用 Java 的 jar 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
- 3、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(每个类都有它们自己的独立的.java 和.class 文件)。如果希望这些构件从属同一个群，就要使用关键字 package
- 4、如果使用 package 语句，它必须是文件中除注释以外的第一句程序代码，以表明你在声明该编译单元为某类库的一部分。

6.1.2. 创建独一无二的包名

- 1、将特定包的所有.class 文件都置于一个目录下(对于 eclipse， 默认在项目文件夹的 bin 目录中)，也就是说，利用操作系统的层次化文件结构，这是 Java 解决混乱问题的一种方式，另一种方式是 jar 工具
- 2、将所有的文件收入一个子目录还可以解决另外两个问题
 - 怎样创建独一无二的名称

- 怎样查找有可能隐藏于目录结构中某处的类
- **这些任务是通过将.class 文件所在的路径位置编码成 package 的名称来实现的**

3、package 名称

- package 名称的**第一部分是类的创建者反序的 Internet 域名**，由此 Internet 域名应该是独一无二的，因此你的 package 名称也是独一无二的，就不会出现名字冲突问题
- package 名称的**第二部分是类库名称**
- 当 Java 程序运行并且需要加载.class 文件的时候，**package 名称分解为你机器上的一个目录**，结合 CLASSPATH 它就可以确定.class 文件在目录上所处的位置

4、Java 解释器运行过程：

- 首先，找出环境变量 CLASSPATH，CLASSPATH 包含一个或多个目录，用作查找.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(例如 foo.bar.bas 就变为 foo/bar/baz，具体看操作系统)
- 得到的路径会与 CLASSPATH 中的各个不同项相连接，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

5、举例说明：

- Internet 域名的反序，即全域名称： net.mindview
- 类库名： simple
- 包名为： net.mindview.simple
- 比如，这个类库的放置位置为： C:\DOC\JavaT\net\mindview\simple
- 结合 CLASSPATH=...;C:\DOC\JavaT
- CLASSPATH 可以包含多个可供选择的查询路径
- 对于 JAR 文件，必须将 JAR 文件的实际名称写清楚，而不仅仅指明它所在的目录

6、对于 eclipse

- 每个工程的 CLASSPATH 是独立的，如果想要调用外部的 JAR 包，需要在 Project-properties-Java Build Path-Add External JARs 里添加
- vim 工程路径/.classpath <==查看 classpath 的结构
 - 源文件的具体位置(kind="src") "/src"
 - 运行的系统环境(kind="con")
 - 工程的 library 的具体位置信息(kind="lib")
 - 在每个 lib 的 xml 子节点中，有关于它的其它配置信息(例如我配置的那个"javadoc_location")
 - 项目的输出目录(kind="output") "/bin"

6.1.3. 定制工具库

6.1.4. 用 import 改变行为???

- 1、Java 中没有 C 的条件编译功能，该功能可以使你不必更改任何代码，就能够切换开关并产生不同的行为。
- 2、Java 去掉此功能是因为 C 在绝大多数情况下是用此功能来解决跨平台问题的，即程序代码的不同部分是根据不同的平台来编译的。
- 3、由于 Java 自身可以自动跨越不同的平台，因此这个功能对于 Java 是没有必要的

6.1.5. 对使用包的忠告

- 1、无论何时创建包，都已经在给定包名的时候隐含的指定了目录结构，这个包必须

位于其名称所指定的目录之中，而该目录必须是在以 CLASSPATH 开始的目录中可以查询到的

2、编译过的代码通常放置在与源代码不同的目录中，但必须保证 JVM 能使用 CLASSPATH 可以找到该路径

6.2. Java 访问权限修饰词

1、public、protected 和 private 这几个 Java 访问权限修饰词在使用时，是置于类中每个成员定义之前的

2、与 C++不同，每个访问权限修饰词仅控制它所修饰的特定定义的访问权

6.2.1. 包访问权限

1、默认访问权限没有使用任何访问权限修饰词，通常指包访问权限(有时也表示成 friendly)

2、包访问权限意味着当前包中所有其他类对那个成员都有访问权限

3、一个编译单元只能隶属于一个包，所以经由包访问权限，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

4、包访问权限允许将包内所有相关的类组合起来，以使它们彼此之间可以轻松地相互作用，当把类组织起来放进一个包内之时，也就给它们的包访问权限的成员赋予了相互访问的权限

5、取得对某成员的访问权限的唯一途径

- 使该成员成为 public，无论谁，无论在哪里，都可以访问
- 通过不加访问权限修饰词，并将其他类置于同一个包内的方式，基于成员赋予包访问权限，于是包内其他类也就可以访问该成员了
- 继承，继承而来的类可以访问 public 成员也可以访问 protected 成员
- 提供访问器和编译器，也称作(get/set 方法)，以读取和改变数值

6.2.2. public: 接口访问权限

1、使用关键字 public，就意味着 public 之后的成员声明对所有人都可见

2、默认包：

- 在 CLASSPATH 中要有'.'
- 没有设置包名的文件隶属于默认包

6.2.3. private：你无法访问

1、关键字 private 的意思是，处理包含该成员的类外，其他任何类都无法访问这个成员

2、默认的包访问权限通常已经提供了充足的隐藏措施，使用该类的客户端程序员是无法访问包访问权限的

6.2.4. protected：继承访问权限

1、protected 同时也提供包访问权限

6.2.5. 访问权限总结：

1、访问权限：限制该成员的地方

2、private：只有该类的内部才能被访问

3、friendly：在包内可以被访问

- 4、**protected**: 在包内以及继承体系之内才能被访问
- 5、**public**: 在任何地方都可以访问
- 6、注意，客户端使用类库所处的位置就是最外层(包与继承体系之外)

6. 3. 接口和实现

- 1、访问权限的控制常被称为**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，常共同被称为**封装，其结果是一个同时带有特征和行为的数据类型**
- 2、出于两个很重要的原因，访问权限控制将权限的边界划在了数据类型的内部
 - 第一个原因是设定客户端程序员可以使用和不可以使用的界限，可以在结构中建立自己的内部机制，而不必担心客户端程序员会偶然地将内部机制当做是他们可以使用的接口的一部分
 - 第二个原因，将接口和实现分离。类库创建者可以随意更改不是 **public** 的东西，而不会破坏客户端代码
- 3、为了清楚起见，会采用一种将 **public** 成员置于开头，后跟 **protected**、包访问权限、**private** 成员的创建类的形式，这样的好处是可以从头读起，读到非 **public** 成员时停止阅读。仍能看到源代码--实现部分，因为它就在类中

6. 4. 类的访问权限

- 1、在 Java 中，访问权限修饰词也可以用于确定库中哪些类对于该库的使用者是可用的，可以用 **public** 作用域整个类的定义来达到目的
- 2、每个编译单元(.java 文件)都只能有一个 **public**，这表示每个编译单元都有单一的公共接口，用 **public** 类来表现
- 3、**public** 类的名称必须完全与含有该编译单元的文件名相匹配
- 4、编译单元内完全不带 **public** 类也是可能的，此时，可以随意命名文件名
- 5、将 **public** 拿掉，这个类就拥有了**包访问权限：该类的对象可以由包内任何其他类来创建，即创建该类的对象的地方必须在包之中**
- 6、**类不可以是 private 的，对于类的访问权限仅有两个：public 与包访问权限**
- 7、可以修饰类的关键字只有三个：final、abstract、public

Chapter 7. 复用类

- 1、复用代码是 Java 众多引人注目的功能之一
- 2、可以通过创建新类来复用代码，而不必再重头开始编写，可以使用别人已经开发并调试好的类
- 3、方法：组合与继承

7.1. 组合语法

- 1、只需要将对象引用置于新类中即可
- 2、编译器并不是简单地为每一个引用都创建用默认对象，这一点很有意义，因为这会增加不必要的负担，初始化这些引用可以在代码中的下列位置进行：
 - 在定义对象的地方，这意味着它们总是能够在构造器被调用之前被初始化(这与 C++ 是完全不同的，C++ 中初始化是在初始化列表中完成的)
 - 在类的构造器中
 - 在正要使用这些对象之前，这种方式被称为惰性初始化
 - 使用实例初始化(静态子句或非静态子句)

7.2. 继承语法

- 1、继承是所有 OPP 语言和 Java 语言不可缺少的组成部分，当创建一个类时，总是在继承，因此，除非已经明确指出要从其他类中继承，否则就是隐式地从 Java 的标准根类 Object 中继承
- 2、使用关键字 extends
- 3、继承会自动得到基类中所有域和方法(能不能访问是另一回事)
- 4、super 关键字：调用基类版本的方法

7.2.1. 初始化基类

- 1、对基类子对象的正确初始化时至关重要的，而且也仅有一种方法来保证这一点：在构造其中调用基类构造器来执行初始化，基类构造器具有执行基类初始化所需要的所有知识和能力
- 2、Java 会自动在导出类的构造器中插入对基类构造器的调用
- 3、带参数的构造器：
 - 如果没有默认的基类构造器，编译器不会自动插入基类构造器的调用
 - 必须使用关键字 super 显式地编写调用基类构造器的语句：super 后跟参数列表
 - 注意：对于其他非构造器的函数，调用基类的版本：super.func(args)，即必须加上方法名

7.3. 代理

- 1、第三种关系成为代理，Java 并没有提供对它的直接支持，这是继承与组合之间的中庸之道。给一个对象提供一个代理对象，并由代理对象来控制对原有对象的引用
 - 我们将一个成员对象置于所要构造的类中(就像组合)
 - 但与此同时我们在新类中暴露了该类成员对象的所有方法(就像继承)，即定义与该

- 类方法名称相同的方法
 - 在新类的方法中调用该对象的同名方法

7.4. 结合使用组合和继承

1、我们甚至不需要源代码(.java)就可以复用代码(即有.class 即可), 我们至多只需要导入一个包

7.4.1. 确保正确清理

1、Java 没有 C++ 中析构函数的概念, 析构函数是一种在对象被销毁时可以自动调用的函数, 在 Java 中, 我们只是习惯忘掉而不是销毁对象, 让垃圾回收器在必要的时候释放其内存
2、由于我们无法知道垃圾回收器将会何时被调动, 或者它是否被调用, 因此我们如果想要做一些清理的工作, 就必须显式地编写一个特殊的方法来做这件事, **注意不能使 finalize, finalize 只有在垃圾将要被释放内存时调用, 同样不能精准控制它的调用**
3、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情, **如果需要清理, 就编写自己的清理方法, 但不要使用 finalize**

7.4.2. 名称屏蔽

1、如果 Java 的基类拥有某个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其在基类中的任何版本(这一点与 C++ 不同, C++ 必须在基类中使用关键字 virtual 标记某一个函数才能让该函数可以被覆盖, 否则就会屏蔽基类的版本, 但 Java 不同, 默认都是可以覆盖的)
2、Java SE5 新增了 @Override 注解, 它并不是关键字, 但是可以把它当做关键字用, 要想覆盖某个方法时, 可以选择添加这个注解, **在你不小心重载而非覆写了该方法时, 编译器就会生成一条错误信息**

7.5. 在组合与继承之间选择

1、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, 而继承是隐式的
2、组合

- 组合技术通常用于想在新类中使用现有类的功能而非它的接口这种情形, 即在新类中嵌入某个对象, 让其实现所需的功能, 但新类的用户看到的只是为新类所定义的接口, 而非所嵌入的对象的接口, 为了取得此效果, 通常将该对象的引用设置为 private
- 有时允许用户直接访问新类中的组合成分是有意义的(如果成员对象自身都隐藏了具体实现), 会使得端口更加易于理解

3、继承

- 继承使用某个现有的类, 并开发它的一个特殊版本, 通常这意味着一个通用类, 并为了某种特殊需要而将其特殊化

7.6. protected 关键字

1、protected 关键字

- 就类用户而言, protected 成员是无法访问的, 与 private 成员一样

- 对于导出类或者位于同一个包中的类来说，它是可以访问的
- 2、尽管可以创建 `private` 域，但是最好的方式还是将域保持为 `private`

7.7. 向上转型

1、为新的类提供方法并不是继承技术中最重要的方面，**其最重要的方面是用来表现新类和基类之间的关系，这种关系可以用"新类是现有类的一种类型(is-a)"这句话加以概括**

2、继承可以确保基类中所有的方法都在导出类中同样有效，所以能够向基类发送的所有信息同样也可以向导出类发送

7.7.1. 为什么称为向上转型

1、由于历史原因，是以传统的类继承图的绘制方法为基础，将根置于页面的顶端，然后逐渐向下，由导出类向基类转型，在继承图上是向上移动的

2、**由于向上转型是从一个较为专用的类型向通用的类型转换，因此总是安全的(那如果以窄化转换来理解呢???)**

7.7.2. 再论组合与继承

1、在面向对象编程中，生成和使用程序代码最有可能采用的方法就是将数据和方法包装进一个类中，并使用该类的对象，也可以用组合技术使用现有类来开发新的类，而继承技术其实是不太常用的

2、最清晰的判断方法：是否需要从新类向基类进行向上转型，如果需要，那么继承是必须的

7.8. `final` 关键字

1、`final` 可能被用到的情况：数据、方法和类

2、`final` 数据

- 一个永远不改变的编译时常量
- 一个在运行时被初始化的值，而你不希望它被改变
- **两者只能初始化一次，可以在定义时初始化、(非)静态子句中初始化、构造器中初始化，只能三选一!!!，否则就会出现再次赋值的错误**
- 对于编译时常量，编译器可以将该常量值代入任何可能用到它的计算式中，也就是说，可以在编译时执行计算式，这减轻了一些运行时的负担，但这类常量必须是基本数据类型，并以 `final` 关键字表示，在对其定义时，必须对其进行赋值
- 一个 `static final` 域只占据一段不能改变的存储空间，用大写字母命名，字与字之间用下划线隔开
- 当对象的引用是 `final` 时，该引用将不能再引用其他对象(相当于 C++ 中的常量引用，C++ 还有一个指向常量的引用，即底层 `const`)，但是仍能操纵该引用改变它所指向的对象，数组也是如此，因为 Java 中数组是一个对象
- `final` 引用没有 `final` 基本类型的用处大

3、空白 `final`

- Java 允许生成空白 `final`，所谓空白 `final` 是指被声明为 `final` 但又未给定初值的域，即没有在定义时初始化的域
- 于是一个类中的 `final` 就可以做到根据对象而有所不同，却又保持其恒定的特性(C++)

中也可以定义没有初始化的 `const` 对象或基本数据类型，因为 C++ 的初始化是在构造函数初始化列表中完成的，如果构造列表中没有显式初始化那么会采用类内初始值来初始化，这一点与 Java 很不同)

4、final 参数

- Java 允许在参数列表中以声明的方式将参数指明为 `final`，这意味着你无法在方法中更改参数引用所指向的对象
- 当一个基本类型的参数被指定为 `final` 时，你可以读取参数，但无法修改参数，其实拷贝一下也就能改了，但是没意义

5、final 方法

- 使用 `final` 方法的原因有两个：
 - 第一个原因：把方法锁定，防止任何继承类修改它的含义，想要确保在继承中使方法行为保持不变，且不能被覆盖
 - 第二个原因：效率(过去有用，现在没有用)，将一个方法指定为 `final` 就是同意编译器针对该方法的所有调用都转为内嵌调用(有点 C++`inline` 的意思)

6、private 和 final 关键字

- 类中所有 `private` 方法都隐式指定为 `final` 的，由于无法取用 `private` 方法，所以也就无法覆盖它，可以对 `private` 方法添加 `final` 修饰词，但是没有任何额外的意义
- 覆盖只有在某方法是基类接口(`public protected`)的一部分时才会出现，即必须能将一个对象向上转型为它的基本类型并调用相同的方法，如果某方法是 `private`，那它就不是基类接口的一部分

7、final 类

- 当将某个类整体定义为 `final` 时，就表明了你不打算继承该类，而且也不允许别人这样做
- `final` 类中所有方法都是隐式 `final` 的

8、有关 final 的忠告

7.9. 初始化及类的加载

1、在许多传统语言中，程序是作为启动过程的一部分立刻被加载的，然后是初始化，紧接着程序开始运行

- 这些语言初始化时必须小心控制，以确保定义为 `static` 的东西，其初始化顺序不会造成麻烦
- 例如 C++，某个 `static` 期望另一个 `static` 在被初始化之前就能有效地使用，那么就会出现问题

2、Java 不会出现这个问题

- 因为采用了不同的加载方式，加载时众多变得更加容易的动作之一
- 每个类的编译代码都存在于它自己的独立文件中，该文件只有在需要使用程序代码时才会被加载，即类的代码在初次使用时才加载(初次通常是指创建类的第一个对象，或者访问 `static` 域或 `static` 方法时)
- 初次使用之处也是 `static` 初始化发生之处，所有的 `static` 对象和 `static` 代码都会在加载时依程序中的顺序(定义时书写顺序)而依次初始化，定义为 `static` 的东西只会被初始化依次

7.9.1. 继承与初始化

1、加载：在导出类第一次被使用时，首先会加载导出类的编译代码，然后加载该导出类的基类的编译代码(无论你是否访问该基类)，若该基类仍有基类，那么继续加载该基类，直到根基类。

2、初始化：

- 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的零
- 从基类到派生类的静态变量初始化
- 按照基类成员声明的顺序调用基类成员的初始化方法(指定初始化)
- 调用基类的构造器主体
- 按照导出类成员声明的顺序调用导出类成员的初始化方法(指定初始化)
- 调用导出类的构造器主体

Chapter 8. 多态

- 1、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
- 2、多态通过分离做什么和怎么做，从另一角度将接口和实现分离开
- 3、多态不但能够改善代码的组织结构和可读性，还能够创建可扩展的程序--即无论在项目最初创建时还是在需要添加新功能时都可以"生长"的程序
- 4、封装通过合并特征和行为来创建新的数据类型
- 5、实现隐藏通过将细节私有化把接口和实现分离开
- 6、多态则消除类型之间的耦合关系
- 7、多态方法调用允许一种类型表现出与其他相似类型之间的区别，只要他们都是从同一基类导出而来的，这种区别是根据方法行为的不同而表示出来的

8.1. 再论向上转型

- 1、向上转型可能会缩小接口，但是不会比基类的接口更窄

8.1.1. 忘记对象类型

- 1、如果以确切对象类型来创建对象，那么每添加一个新的导出类，就得编写特定类型的方法
- 2、我们不管导出类的存在，编写的代码只与基类打交道

8.2. 转机

- 1、绑定

8.2.1. 方法调用绑定

- 1、将一个方法调用同一个方法主体关联起来被称作绑定，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(静态绑定)**，例如 C 只有一种绑定，就是前期绑定

2、**后期绑定**也叫作**动态绑定**或**运行时绑定**

- 如果一种语言实现后期绑定，就必须具有某种机制，以便在运行时能判断对象的类型，从而调用恰当的方法
- 编译器一直不想知道对象的类型，但是方法调用机制能找到正确的方法体，并加以调用
- 后期绑定机制随编程语言不同而有所不通，但是不管怎样，都必须在对象中安置某种"类型信息"
- **Java 中除了 static 方法和 final 方法(private 方法属于 final 方法)，其他所有的方法都是后期绑定，这意味着，我们不必判定是否应该进行后期绑定--它会自动发生**
- 将某个方法声明为 final 可以关闭动态绑定，告诉编译器不需要动态绑定，这样编译器就能为 final 方法调用生成更有效的代码，但是大多数情况，对程序整体性能不会有什麼改观，因此最好根据设计来决定是否用 final

8.2.2. 产生正确的行为

- 1、我们可以编写只与基类打交道的代码，并且这些代码对所有的导出类都是可行的，即发

送消息给某个对象，让对象去判定该做什么事

8.2.3. 可扩展性

1、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，从而添加一些新的功能，那些操纵基类接口的方法不需要任何改动就能应用于新类

8.2.4. 缺陷：“覆盖”私有方法

1、只有非 `private` 方法才能被覆盖(`private` 方法是隐式 `final` 的)，但是还需要密切注意覆盖"private"方法的现象，因为编译器不会报错，但是也不会按照我们所期望(动态)的来执行
2、因此在导出类中，对于基类中的 `private` 方法，最好采用不同的名字
3、`private` 方法同样不能重载，因为在导出类中根本无法调用基类的 `private` 方法，何来重载

8.2.5. 缺陷：域与静态方法

1、只有普通的方法调用可以是多态的，访问某个静态或非静态域或者静态方法，这个访问将在编译器进行解析
2、注意，当一个方法如果是多态的，那么这个方法中访问的域就是动态的
2、可以在方法内部，调用 `super.field` 显式调用基类的域

8.3. 构造器和多态

1、构造器不具有多态性(它们实际上是 `static` 方法，只不过该 `static` 是隐式的)

8.3.1. 构造器的调用顺序

1、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，以使得每个基类的构造器都能够得到调用
2、这样做是有意义的，构造器有一项特殊的任务：检查对象是否被正确构造
3、导出类只能访问它自己的成员，而不能访问基类的成员(当基类域是 `private` 的时候)，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
4、如果没有明确地指明调用某个基类构造器，编译器就会调用默认构造器，如果不存在默认构造器，编译器就会报错
5、调用过程详解，初始化过程详见 7.9.1

8.3.2. 继承与清理

1、通过组合和继承方法来创建新类时，永远不必担心对象的清理问题，子对象通常会留给垃圾回收器进行处理
2、如果需要做清理动作，自定义方法来进行清理(不要用 `finalize`)
3、由于继承的缘故，如果我们需要做清理工作，必须在导出类中覆盖该自定义的方法，并且调用基类版本的该方法，否则基类的清理就不会发生
4、销毁的顺序应该和初始化的顺序相反：
➤ 与定义时的顺序相反
➤ 调用基类的版本放在最后，而不是开头

8.3.3. 构造器内部的多态方法的行为

1、如果在一个构造器的内部调用正在构造的对象的某个动态绑定的方法：
➤ 在一般的方法内部，动态绑定的调用时在运行时决定的，因为对象无法知道它是属

于方法所在的类还是属于那个类的导出类

- 如果要调用构造器内部的动态绑定的方法，就要用到哪个方法被覆盖后的定义，然而这个调用的效果可能相当难以预料，因为被覆盖的方法在对象被完全构造之前就会被调用，这可能会造成一些难于发现的隐藏错误

2、从概念上讲

- 构造器的工作实际上是创建对象(并非是一件平凡的工作)，在任何构造器内部，整个对象可能只是部分形成，我们只知道基类对象已经进行初始化
- 如果构造器只是在构建对象过程中的一个步骤，并且该对象所属的类是从这个构造器所属的类导出的，那么导出部分在当前构造器正在被调用的时刻仍旧是没有被初始化的
- 然而，一个动态绑定的方法调用却会向外深入到继承层次结构内部，它可以调用导出类里的方法
- 如果我们在构造器内部这样做，那么就可能会调用某个方法，而这个方法所操纵的成员可能还未进行初始化--这肯定招致灾难

3、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为二进制的零，这样做有一个优点，即所有东西至少初始化为 0，而不是仅仅留作垃圾(C++中，类内非静态域的值是未定义的)

4、上述所说的构造器中调用动态方法是可以编译通过的，而且会按照我们的想法发展

- C++不同，C++在函数体开始执行的时候，初始化已经完成了，因为初始化是在初始化列表中完成的
- 在函数体中调用虚函数的话，C++的策略是，执行与构造函数同一个类的版本
- 因此在 Java 中，尽量避免调用其他方法，在构造器中唯一安全调用的那些方法是 final 方法(private 方法也是自动属于 final 的)，在哪个类的构造器调用 final 方法，就执行那个构造器对应类的该方法，与 C++中调用虚函数效果一致

8.4. 协变返回类型

1、Java SE5 中添加了协变返回类型，它表示在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

2、协变返回类型允许返回更具体的类型

8.5. 用继承进行设计

1、当我们使用现成的类来建立新类时，首先考虑使用继承技术，反倒会集中我们的设计负担，让事情变得复杂

2、更好的方法是首选"组合"，尤其是不能十分确定应该使用哪一种方式时

8.5.1. 纯继承与扩展

1、纯继承被称作是纯粹的"is-a"关系，因为一个类的接口已经确定了它应该是什么，在纯继承下，导出类也将具有和基类一样的接口，并且可以认为是一种纯替代，因为导出类可以完全替代基类，而不需要知道关于子类任何额外消息

2、extends 关键字似乎在怂恿我们进行扩展而非纯继承

8.5.2. 向下转型与运行时类型识别

1、向上转型是安全的，因为基类不会具有大于导出类的接口，通过基类接口发送的消息保证都能被接受

2、必须有某种方法来保证向下转型的正确性，使我们不至于贸然转型到一种错误类型

- 在某些程序设计语言中，我们必须执行一个特殊的操作来获得安全的向下转型
- 在 Java 语言中，所有的转型都会得到检查，所以即我们只进行一次普通的加括弧形式的转型，在进入运行使其仍然会对其进行检查，以便保证它的确是我们希望的那种类型，如果不是，就会返回 `ClassCastException`，这种在运行期间对类型进行检查的行为被称为"RTTI"

Chapter 9. 接口

1、接口和内部类为我们提供了一种将接口与实现分离的更加结构化的方法

9.1. 抽象类和抽象方法

1、Java 提供了一种叫做"抽象方法"的机制，这种方法是不完整的，仅有声明而没有方法主体

```
abstract void f();
```

2、包含抽象方法的类叫做抽象类，**如果一个类包含一个或多个抽象方法，该类必须被限定为抽象的(用 abstract 关键字限定)**，否则编译器会报错

➤ C++只需要声明一个纯虚函数即可，不必在类前加关键字

3、编译器会确保抽象类是纯粹的，即不可创建抽象类的对象，不必担心误用它

4、如果从一个抽象类继承，并向创建该新类的对象，那么就必须为基类中的所有抽象方法提供定义，如果不这样做，那么导出类便也是抽象类，而且编译器会强制我们用 abstract 关键字来限定这个类

9.2. 接口

1、interface 关键字使抽象的概念更向前迈进了一步

➤ abstract 关键字允许人们在类中创建一个或多个没有任何定义的方法---提供了接口部分，但是没有提供任何相应的具体实现，这些实现是由此类的继承者创建的
➤ interface 这个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，它允许创建者确定方法名、形参列表和返回类型，但是没有任何方法体，接口只提供了形式，而未提供任何具体实现

● 相比于抽象方法，接口中的方法不需要 abstract 关键字，隐式抽象的

2、interface 不仅仅是一个极度抽象的类，因为它允许人们通过创建一个能够被向上转型为多种"基类"的类型，来实现某种类似多重继承变种的特性(Java 不允许多重继承)

3、要想创建一个接口，需要用 interface 关键字来代替 class 关键字，就像类一样，**可以在 interface 关键字前面添加关键字 public(还可以是 abstract，仅有这两种，与 class 关键字不同)**。若不添加关键字，就只具有包访问权限，它就只能在同一个包中可用

4、**接口也可以包含域，但是这些域是隐式 static 和 final 的**

5、要让类遵循某个特定接口(或者一组接口)，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，除此之外，它看起来很像继承

6、**可以选择在接口中显式地将方法声明为 public 的，但即使你不这么做，它们也是 public 的，当要实现一个接口时，在接口中被定义的方法必须定义为 public，否则编译器将报错**

9.3. 完全解耦

1、只要一个方法操作的是类而非接口，那么你就只能使用这个类及其子类，如果你想要将这个方法应用于不在此继承结构中的某个类，那么你就触霉头了，接口可以很大程度上放宽这种限制

2、复用代码

➤ 第一种方式是客户端程序员遵循该接口来编写他们自己的类

- 经常碰到无法修改想要使用的类，即想要实现接口的类是别人定义的，该类的接口与想要遵循的接口的接口不一致，而自己又无权修改该类
- 在此情况下可以使用**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，即适配器实现想要实现的接口**
- **定义一个新类，继承已有的类，实现所需要的接口**

9.4. Java 中的多重继承

1、接口不仅仅只是一种更纯粹形式的抽象类，它的目标比这要更高，因为接口根本没有任何具体实现，也就是说没有任何与接口相关的存储，因此无法阻止多个接口的组合

9.5. 通过继承来扩展接口

1、通过继承，可以很容易地在接口中添加新的方法声明，还可以通过继承在新接口中组合数个接口，使用 `extends` 关键字

2、**接口可以继承多个接口**

9.5.1. 组合接口时的名字冲突

1、在打算组合的不同接口中使用相同的方法名通常会造成代码可读性的混乱，尽量避免这种情况

9.6. 接口适配

1、接口最吸引人的原因之一就是允许同一个接口具有多个不同的具体实现

2、接口最常见的设计模式就是**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你指定的接口**
- 主要就是要声明：你可以用任何你想要的对象来调用我的方法，只要你的对象遵循我的接口

9.7. 接口中的域

1、接口中任何域自动都是 `final` 和 `static` 的

- **注意，`final` 代表不可更改**
- **`static` 代表该接口的实例共享该域**
- **并不是我之前认为的不能使用的意思(之前我认为实现不能继承接口的域)**

2、在 Java SE5 之前，接口是创建常量组的有效工具，产生于 C++ 和 C 中 `enum` 具有相同效果的类型的唯一途径

3、**但现在已经没有必要了，因为已经有了 `enum`**

9.7.1. 初始化接口中的域

1、接口中定义的域不能是“空 `final`”，可以被非常量表达式初始化

2、这些域不是接口的一部分，他们的值被存储在该接口的静态存储区域

9.8. 嵌套接口

- 1、接口可以嵌套在类或其他接口中
- 2、嵌套在接口中的接口自动是 `public` 的，而不能声明为 `private`
- 3、嵌套在类中的接口的访问权限关键字可以是 `public protected private` 以及 `friendly` 中的任意一种，但是该接口中的方法仍然是 `public` 的
 - 那么一个 `private` 的嵌套接口，其 `public` 方法如何调用呢？
 - 可以将该接口的实例传递给有权调用的对象

9.9. 接口与工厂

- 1、接口是实现多重继承的途径，而生成遵循某个接口的对象的典型方法就是工厂方法
- 2、与直接调用构造器不同，我们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
- 3、通过这种方式，我们的代码将完全与接口的实现分离

Chapter 10. 内部类

- 1、可以将一个类的定义放在另一个类的定义内部，称为内部类
- 2、内部类是一种非常有用特性，因为它允许你把一些逻辑相关的类组织在一起，并控制位于内部的类的可视性
- 3、内部类看起来就像是一种代码隐藏机制，将类置于其他类的内部，但内部类远不止如此，它了解外围类，并能与之通信

10.1. 创建内部类

- 1、把类的定义置于外围类的内部即可，有 `public`、`protected`、`friendly`、`private` 四种访问权限修饰关键字可用

10.2. 链接到外部类

- 1、当生成一个内部类的对象时，此对象与制造它的外围对象之前有一种联系，所以它能访问其外围对象的所有成员(包括 `private`)，而不需要任何特殊条件
- 2、内部类拥有其外围类的所有元素的访问权
- 3、当某个外围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围类的引用，在你访问外围类的成员时，就是利用这个引用来选择外围类的成员
- 4、内部类的对象只能在与其外围类的对象关联的情况下才能被创建
- 5、注意，内部类包含指向外围类对象的引用，并不能使得通过内部类的对象能访问外围类的成员，而是在内部类的实现中可以访问

10.3. 使用 `this` 与 `new`

- 1、如果你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`.this`，这样产生的引用具有正确的类型
- 2、外部类的实例后跟`.new` 来创建内部类，因此在拥有外部类对象之前，是不能创建内部类对象的，因为内部类对象必须连接到创建它的外部类对象上(**静态内部类除外**)
 - `Outer.Inner objInner=objOuter.new Inner();`

10.4. 内部类与向上转型

- 1、当将内部类向上转型为其基类，尤其是转型为一个接口的时候，内部类就有了用武之地
- 2、内部类能够使某个接口的实现完全不可见，所得到的只是指向基类或接口的引用，可以很方便的隐藏细节
- 3、`private` 内部类给类的设计者提供了一种途径，通过这种方式可以完全阻止任何依赖于类型的编码，并且完全隐藏了实现的细节

10.5. 在方法和作用域内的内部类

- 1、可以在一个方法里面或者任意作用域里定义内部类的理由
 - 实现了某类型的接口，于是可以创建并返回对其的引用

- 要解决一个复杂的问题，想要创建一个类来辅助你的解决方案，但是又不希望这个类是公共可用的

10.6. 匿名内部类

- 1、这种语法很奇怪：创建一个继承自某接口的匿名类的对象，通过 new 表达式返回的引用被自动向上转型为声明的那个接口类型
- 2、一般来说，在匿名内部类结束的分号，是用于 new 语句结束的，而非用于匿名内部类的定义
- 3、如果定义一个匿名内部类，并且希望它使用一个在其外部定义的对象，那么编译器会要求其参数引用是 final 的
- 4、匿名内部类是没有构造器的，因为根本没有名字，但是可以通过实例初始化，达到为匿名内部类创建一个构造器类似的效果，即{}括起来的语句

10.6.1. 再访工厂方法

- 1、本来需要定义工厂接口的不同实现的类，现在只需要用匿名内部类，将工厂定义为一个类的静态字段即可
- `public static Factory factory=new Factory()/*覆盖工厂方法*/;`

10.7. 嵌套类

- 1、如果不需要内部类对象与外围类对象有联系，那么可以将内部类声明为 static，这通常称为嵌套类
 - 创建嵌套类的对象，并不需要其外围类的对象
 - 不能从嵌套类的对象中访问非静态的外围类对象
- 2、普通内部类的字段和方法只能是非静态的，而嵌套类可以是静态或者非静态的
- 3、**与 C++ 嵌套类大致相似，但是 C++ 中嵌套类不能访问私有成员，而 Java 可以**

10.7.1. 接口内部的类

- 1、正常情况下，不能在接口内部放置任何代码，但嵌套类可以作为接口的一部分
- 2、放到接口中的任何类都自动地都是 public 和 static 的

10.7.2. 从多层嵌套类中访问外部类的成员

- 1、一个**普通内部**类被嵌套多少层并不重要--它能透明地访问所有它所嵌入的外围类的所有成员

10.8. 为什么需要内部类

- 1、一般来说，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围类的对象，所以可以认为内部类提供了某种进入其外围类的窗口
- 2、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，所以无论外围类是否已经继承了某个(接口的)实现，对于内部类都是没有影响的**
- 3、内部类使得多重继承的解决方案变得完整，接口解决了部分问题，而内部类有效地实现了“多重继承”

4、内部类的其他特性：

- 内部类可以有多个实例，每个实例都有自己的状态信息，并且与其外围类对象的信息相互独立
- 在单个外围类中，可以让多个内部类以不同的方式实现同一个接口，或继承同一类
- 创建内部类对象的时刻并不依赖于外围类对象的创建???
- 内部类并没有令人迷惑的"is-a"关系

10.8.1. 闭包与回调

1、闭包是一个可调用对象，它记录了一些信息，这些信息来自于创建它的作用域

- 通过这个定义，**可以看出内部类是面向对象的闭包**，因为它不仅包含外围类对象(创建内部类的作用域)的信息，还自动拥有一个指向此外围类对象的引用，在此作用域内，内部类有权操纵所有成员，包括 `private`
 - C++的 `lambda` 表达式也是一个闭包，[]捕获列表就会捕获一些信息
- 2、Java 最引人争议的问题之一：人们认为 Java 应该包含某种类似指针的机制，以允许回调。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，比指针更灵活，安全

10.8.2. 内部类与控制框架

- 1、应用程序框架就是被设计用以解决某类特定问题的一个类或一组类
- 2、要运用某个应用程序框架，通常是继承一个或多个类，并覆盖这些方法
- 3、设计模式总是将变化的事物与不变的事物分离开
- 4、控制框架是一类特殊的应用程序框架，它用来解决相应时间的需求，主要用来相应时间的系统被称作事件驱动系统

10.9. 内部类的继承

- 1、因为内部类的构造器必须连接到指向其外围类对象的引用，所在继承内部类的时候，事情会有点复杂，问题在于，那个指向其外围类的"秘密的"引用必须被初始化，而在导出类中不再存在可连接的默认对象，**要解决这个问题，必须使用特殊的语法来明确说清他们之间的关联(指明要关联的外部类的对象)**

10.10. 内部类可以被覆盖吗

- 1、如果一个类 A 创建了一个内部类 A.Inner，当有一个类 B 继承自 A，并重新定义该内部类会发生内部类的覆盖吗？不会
- 2、当继承了某个外围类的时候，内部类并没有发生什么特别神奇的变化，这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
- 3、当然，可以明确地继承某个内部类

10.11. 局部内部类

- 1、可以在代码块里创建内部类，典型的方式是在一个方法体内创建，**局部内部类不能有访问说明符，因为它不是外围类的一部分，但是它可以访问当前代码块内的常量，以及此外围**

类的所有成员

2、局部内部类和匿名内部类都可以实现返回一个接口实例的功能，如果需要局部内部类，那么唯一的理由就是需要一个已命名的构造器，**而匿名内部类只能用于实例初始化**

10.12. 内部类标识符

1、每个类都会产生一个.class 文件，其中包含了如何创建该类型的对象的全部信息(此信息产生一个"meta-class"，叫做 Class 对象)

2、内部类也有.class 文件用以包含它们的 Class 对象信息，这些文件的命名有严格的规则：外围类的名字，加上'\$'，再加上内部类的名字

Chapter 11. 持有对象

- 1、如果一个程序只包含固定数量的且其生命周期都是已知的对象，那么这是一个非常简单的程序
- 2、通常程序总是根据运行时才知道的某些条件去创建新对象，需要在任意时刻，任意位置创建任意数量的对象，因此不能依靠创建命名的引用来持有每一个对象
- 3、Java 除了提供数组来解决这个问题外还提供了一套相当完整的容器类来解决这个问题

11.1. 泛型和类型安全的容器

- 1、使用 Java SE5 之前的容器的一个主要问题就是：编译器允许你向容器中插入不正确的类型
- 2、可以使用 Java SE5 所特有的注解来抑制警告信息：
 - @SuppressWarnings: 表示有关"不受检查的异常"的警告信息应该被抑制
- 3、当你指定了某个类型作为泛型参数时，你并不仅限于只能将该确切类型的对象放置到容器中，向上转型也可以像作用域其他类型一样作用于泛型

11.2. 基本概念

- 1、Java 容器类库的用途是"保存对象"，并将其划分为两个不同的概念
 - Collection: 一个独立元素的序列，这些元素都服从一条或多条规则
 - Map: 一组成对的"键值对"对象，允许你使用键来查找值，也被称为关联数组或字典

11.3. 添加一组元素

- 1、java.util.Arrays 和 java.util.Collections 类中都有很多实用的方法
- 2、Collection 构造器可以接受另一个 Collection，用它来将自身初始化(**Java 中为数不多的拷贝，依次拷贝每个对象的引用，而不是拷贝容器的引用**)
- 3、Collection.addAll()运行速度更快，即创建一个不包含元素的 Collection，然后调用 addAll() 方法即可，**该方法只接受另一个 Collection 对象作为参数**
- 4、Arrays.asList() 与 Collections.addAll() 方法接受可变参数列表
 - 可以直接使用 Arrays.asList() 的输出，将其当做 List，Arrays.asList() 将一个数组转化为一个 List 对象，这个方法会返回一个 ArrayList 类型的对象，这个 ArrayList 类并非 java.util.ArrayList 类，而是 Arrays 类的静态内部类！其底层是数组，因此不能调整尺寸，试图用 add 或 delete 方法会引发改变数组尺寸的尝试，会获得"Unsupported Operation"的错误
 - Arrays.asList() 方法的限制是它对所产生的 List 的类型做出了最理想的假设，而且没有注意到你会赋予什么样的类型，但是可以提供泛型参数，形如
 - Arrays.<Integer>.asList()
- 5、**Collection 初始化的方式**
 - Collection<T> c=new ArrayList<T>(); //空的容器
 - Collection<T> c1=new ArrayList<T>(c); //用另一个容器来初始化
 - Collection<T> c2=new ArrayList<T>();c2.addAll(c1); 类似 c1，但效率更高

6、**Map** 更加复杂，除了用另一个 **Map** 之外，Java 标准库没有提供其他任何自动初始化的方式

11.4. 容器的打印

- 1、必须使用 `Arrays.toString()` 打印数组
- 2、打印容器无需任何帮助，容器提供了 `toString()` 方法

11.5. List

1、List 是接口

2、List 承诺可以将元素维护在特定序列中，List 接口在 Collection 的基础上添加了大量的方法，使得可以在 List 的中间插入和移除元素

3、两种类型的 List

- `ArrayList`: 它长于随机访问元素，但是在 List 的中间插入和移除元素时较慢
- `LinkedList`: 它通过代价较低的 List 中间进行的插入和删除操作，提供了优化的顺序访问，`LinkedList` 在随机访问方面相比 `ArrayList` 较慢，但是他的特性集较 `ArrayList` 大

4、List 常用接口

- `List.add(T t)`: 在末尾添加元素
- `List.add(int index, T t)`: 在指定位置添加元素
- `List.add(Collection<?> c)`: 在末尾添加元素
- `List.add(int index, Collection<?> c)`: 在指定位置添加元素
- `List.isEmpty()`: 判断是否为空
- `List.equals(Object obj)`: 比较**内容**是否相同，用"=="比较的是引用是否指向同一对象
- `List.contains(Object obj)`: 判断某个对象是否在列表中
- `List.containsAll(Collection<?> c)`: 判断列表中所有元素是否在当前列表中，顺序不重要
- `List.set(int index, T t)`: 替换
- `List.remove(Object obj)`: 删除某个指定的元素
- `List.remove(int dex)`: 删除指定位置的元素
- `List.indexOf(Object obj)`: 指出某个指定元素在 List 中所处的位置
- `List.subList(int fromIndex, int toIndex)`: 创建出一个片段(指定区域元素的引用拷贝)，因此通过引用改变片段中的元素，会改变原 List 中元素的状态，因为指向的是同一个对象
- `ListretainAll(Collection<?> c)`: 提供有效的"交集"操作，其行为依赖 `equals` 的实现
- `List.clear()`: 清空
- `List.toArray()`: 返回 Object 的数组
- `List.toArray(T[] a)`: 产生指定类型的数组，如果形参数组太小放不下所有元素，则会创建一个合适大小的数组
- **注意上述形参列表中，Object 说明与指定的泛型参数 T 无关**

11.6. 迭代器

- 1、任何容器类，都必须有某种方式可以插入元素并将它们再次取回，持有事物是容器最基

本的工作

2、从更高层次考虑，上述方法会有一个缺点：必须针对容器的确切类型编程

3、迭代器(也是一种设计模式)的概念可以用于达成此目的

- 迭代器是一个对象，它的工作是遍历并选择序列中的对象，而客户端程序员不必知道或关心改程序底层的结构

- 迭代器被称为轻量级对象，创建它的代价较小

4、Iterator

- 使用方法 `iterator()`要求容器返回一个 `Iterator`，`Iterator` 将准备好返回第一个元素
- 使用 `next()`获得序列中下一个元素
- 使用 `hasNext()`检查序列中是否还有元素
- 使用 `remove()`将迭代器新返回的元素从序列中删除(调用 `remove` 之前必须先调用 `next`)

11.6.1. ListIterator

1、`ListIterator` 是一个更强大的 `Iterator` 的子类型，它只能用于各种 `List` 类的访问

2、特性：

- `Iterator` 只能向前移动，`ListIterator` 可以双向移动
- `ListIterator` 还能产生相对于迭代器在列表中指向的当前位置的前一个和后一个元素的索引
- 可以使用 `ListIterator.set()`方法替换它访问过的最后一个元素
- 可以使用 `ListIterator.hasNext()`与 `ListIterator.hasPrevious()`根据位置来判断是否还有元素，即与之前是否访问过无关，只于迭代器所处的索引位置与边界位置有关
- 可以通过 `List.listIterator()`方法产生一个指向 `List` 开始出的 `ListIterator`，还能通过调用 `List.listIterator(int index)`方法创建一个指向列表索引为 `n` 的元素处的 `ListIterator`
 - 注意迭代器插入的位置，插入的位置为索引 `n-1` 与索引 `n` 之间(不指定索引同样适用，即-1 与 0 之间)
 - 调用 `ListIterator.next()`返回索引为 `n` 的元素
 - 调用 `ListIterator.previous()`返回索引为 `n-1` 的元素

11.7. LinkedList

1、`LinkedList` 也像 `ArrayList` 一样实现了基本的 `List` 接口，但它执行某些操作(在列表中间插入删除)比 `ArrayList` 要高效，在随机访问操作方面要逊色一点

2、`LinkedList` 中还添加了可以使其用作栈、队列、双端队列的方法

- 这些方法中有些彼此之间只是名称有些差异，或者只存在些许差异，使得这些名字在特定用法的上下文中更加适用
- 这么做是值得的，使方法与 `LinkedList` 向上转型所对应的接口(或者不转型，例如将 `LinkedList` 用作栈，那么 `push`, `pop` 相比于 `add` 与 `remove` 无疑是更好的名称)类型更加相配

11.8. Stack

1、栈通常是指先进后出"LILO(last input first output)"的容器，有时也被称为叠加栈，因为最后"压入"的元素总是第一个"弹出"

- 2、`LinkedList` 具有能够直接实现栈的所有功能的方法，因此可以直接将 `LinkedList` 作为栈使用
- 3、如果值需要栈的行为，可以利用适配器设计模式或者代理模式，显式指定你需要的接口，但是继承是不行的，因为我们所需要的类型的接口是要小于 `LinkedList` 的，使用继承，那么会暴露 `LinkedList` 的所有接口，没有意义(`java.util.Stack` 就犯了这个错误，并且继承的 `Vector` 是基于数组的，在某些特定情况下，压入弹出操作的执行时间并不如预期，因为可能有数组扩张和收缩的操作)

11.9. Set

- 1、**Set 是接口**
- 2、`Set` 不保存重复的元素，试图将相同对象的多个实例添加到 `Set` 中，那么他就会阻止这种重复现象(判断重复的行为依赖于其元素的 `hashCode` 与 `equals` 的实现)
 - 首先判断 `hashCode` 是否相同，若 `hashCode` 不同，则认为不重复
 - 若 `hashCode` 相同，接着判断 `equals` 是否相同，若 `equals` 相同，则认为重复，若 `equals` 不同，则认为不重复
 - 对于基本类型，只要值相同，其包装类型的 `hashCode` 的实现会保证其相同
 - 对于类类型，只要不是同一个对象，那么一般来说 `hashCode` 是不同的
 - 对于 `String`，内容相同的 `String`，其 `hashCode` 相同
- 3、`Set` 通常被用来测试属性，判断对象是否在某个 `Set` 中
- 4、`Set` 具有与 `Collection` 完全一样的接口，没有任何额外的功能
- 5、各种实现类型
 - `HashSet`: 使用了散列
 - `TreeSet`: 使用了红黑树
 - `LinkedHashSet`: 使用了散列，同时维护了插入时的顺序

11.10. Map

- 1、**Map 是接口**
- 2、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
- 3、常用方法:
 - `Map.get(Object key)`: 获取对应键的值
 - `Map.isEmpty()`: 检查是否为空
 - `Map.put(K key, V value)`: 插入或者更新对应键的值
 - `Map.containsKey(Object key)`: 判断指定关键字是否在 `Map` 中
 - `Map.containsValue(Object value)`: 判断指定值是否在 `Map` 中
 - `Map.keySet()`: 返回由键组成的 `Set`
 - `Map.entrySet()`: 返回元素类型为 `Map.Entry<K,V>` 的 `Set`，每个元素为一个键值对
 - `for(Map.Entry<K,V> pair:Map.entrySet()) {/* */}`
 - `for(K k:Map.keySet()) {/* */}`
- 4、多种实现
 - `HashMap`: 使用散列
 - `TreeMap`: 使用红黑树
 - `LinkedHashMap`: 使用散列，同时维护插入时的顺序

11.11. Queue

1、**Queue 是接口**

2、队列是一个先进先出(FIFO,first input first output)的容器

3、`LinkedList` 提供了方法以支持队列的行为，并且实现了 `Queue` 的接口，因此 `LinkedList` 可以作为 `Queue` 的一种实现

4、典型方法：

- `Queue.offer()`
- `Queue.poll()`
- `Queue.peek()`

11.11.1. PriorityQueue

1、先进先出描述了最典型的队列规则

2、**队列规则：指在给定一组队列中的元素的情况下，确定下一个弹出队列的元素的规则**，先进先出声明的是下一个元素应该是等待时间最长的元素

3、优先队列声明下一个弹出的元素是最需要的元素(最有最高的优先级)

4、当你在 `PriorityQueue` 上调用 `offer()` 方法插入一个对象时，这个对象会在队列中被“排序”(最大/小堆的形式对元素进行组织)。可以通过自己提供 `Comparator` 来修改这个顺序

11.12. Collection 和 Iterator

1、`Collection` 是描述所有序列容器的共性的根接口，它可能会被认为是一个“附属接口”

2、使用接口描述的一个理由是它可以使我们能够创建更通用的代码，通过针对接口而非具体实现来编写代码

3、C++类库中并没有容器的任何共基类，容器之间的共性是通过迭代器达成的

4、在 Java 中，迭代器和基类被捆绑在了一起，因为实现 `Collection` 就意味着必须要提供 `iterator()` 方法

5、选择：

- 当一个类属于 `Collection` 继承体系之中时，使用 `Collection` 来进行迭代会更方便一点，因为 `Collection` 是 `Iterable` 类型(`Collection` 接口继承自 `Iterable` 接口)
- 当一个类不属于 `Collection` 继承体系之中时，使用 `Iterator` 来进行迭代会更好，因为如果要使用 `Collection`，必须实现其所有的接口，而实现迭代器只需要实现 4 个方法

11.13. Foreach 与迭代器

1、`foreach` 语法主要用于数组，但是它也可以应用于任何 `Collection` 对象

2、`Iterable` 接口

- 该接口包含 `iterator()` 方法，用于产生一个 `Iterator`
- 任何实现了 `Iterable` 接口的类都能用于 `foreach` 中

11.13.1. 适配器方法惯用法

1、现在有一个 `Iterable` 类，想要添加一种或多种 `foreach` 语句中使用这个类的方法

2、继承原有类，然后添加新的接口

```

public Iterable<T> reversed(){
    return new Iterable<T>(){
        public Iterator<T> iterator(){
            return new Iterator<T>(){
                public boolean hasNext(){/*...*/}
                public T next(){/*...*/}
                public void remove(){/*...*/}
            }
        }
    }
}

```

11.14. 总结

- 1、数组将数字与对象联系起来
- 2、Collection 保存单一的元素，而 Map 保存相关联的键值对
- 3、容器不能持有基本类型，只能指定持有基本类型的包装类型
- 4、像数组一样，List 也建立数字索引与对象的关联，因此数组与 List 都是排好序的容器，List 能够自动扩充容量
- 5、若要进行大量的随机访问，用 ArrayList，若经常从中间插入删除元素，用 LinkedList
- 6、各种 Queue 以及栈行为，由 LinkedList 支持
- 7、Map 是一种将对象与对象相关联的设计
- 8、Set 不接受重复元素
- 9、新程序中不应该使用 Vector、Hashtable、Stack、StringBuffer
 - 为什么不用 Vector、StringBuffer、Hashtable
 - Vector、StringBuffer 是线程安全的，也就是为每一个方法都加了 synchronized 关键字
 - 但是这种线程安全是相对线程安全，在大多数时候增加了大量不必要的成本，并且未必达到目的
 - 比如说，一个线程打印 Vector，一个线程删除 Vector
 - ◆ 当绿色条件判断完后，该线程 1 交出执行权
 - ◆ 线程 2 获取执行权，当红色的语句执行完毕后，该线程交出执行权给线程 1
 - ◆ 线程 1 接过执行权开始执行的地方是打印语句，因此可能会出现越界的异常
 - 换句话说，Vector 所提供的所谓“线程安全”，并不包括“多个操作之间的原子性”支持。

```

private static Vector<Integer> vector=new Vector<Integer>();
public static void main(String[] args){
    while(true){
        for(int i=0;i<10;i++){
            vector.add(i);
        }
        Thread removeThread=new Thread(new Runnable(){
            @Override
            public void run(){
                for(int i=0;i<vector.size();i++){

```

```
vector.remove(i);
}
});
Thread printThread =new Thread(new Runnable(){
    @Override
    public void run(){
        for(int i=0;i<vector.size();i++){
            System.out.println((vector.get(i)));
        }
    }
});
removeThread.start();
printThread.start();
while(Thread.activeCount()>20);
}
}

private static Vector<Integer> vector=new Vector<Integer>();
public static void main (String[] args){
    Thread removeThread=new Thread(new Runnable(){
        @Override
        public void run(){
            synchronized (vector){
                for(int i=0;i<vector.size();i++){
                    vector.remove(i);
                }
            }
        }
    });
    Thread printThread=new Thread(new Runnable(){
        @Override
        public void run(){
            synchronized(vector){
                for(int i=0;i<vector.size();i++){
                    System.out.println((vector.get(i)));
                }
            }
        }
    });
}
});
```

- 为什么不用 Stack
 - 直接继承自 Vector，而 Vector 的底层实现是数组
 - 包含了大量不必要的接口，即具有 Vector 的所有方法
 - 在 API 设计中有一条规则是这样说的：一个类不应该自己实现同步，而应该把同步的工作留给用户，在绝大多数情况下，把同步的工作留给应用代码，而不是工具类

的代码

Chapter 12. 通过异常处理错误

- 1、Java 基本的理念是：结构不佳的代码不能运行
- 2、发现错误的理想时机是在编译阶段，余下的问题必须在运行期间解决，这就需要通过某种方式，把适当的信息传递给某个接受者，接受者将知道如何正确处理这个问题
- 3、错误恢复在我们所编写的每一个程序中都是基本要素，Java 的主要目标之一就是创建供他人使用的程序构件

12.1. 概念

12.2. 基本异常

- 1、异常情形：阻止当前方法或作用域继续执行的问题，把异常情形与普通问题相区分很重要
 - 所谓的普通问题是指，在当前环境下能得到足够的信息，总能处理这个错误
 - 对于异常情形，就不能继续下去了，因为在**当前环境**下无法获得必要的信息来解决问题，你所能做的就是从当前环境跳出，并且把问题提交给上一级环境
- 2、异常抛出后，发生的几件事情
 - 首先，同 Java 中其他对象的创建一样，使用 new 在堆上创建异常对象
 - 然后，当前的执行路径(它不能继续下去)被终止，并且从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来继续执行程序，这个**恰当的地方就是异常处理程序**，它的任务就是将程序从错误中恢复，以使程序要么换一种方式运行，要么继续运行下去
- 3、异常使我们可以将每件事都当做一个事务，而异常可以看护这些事务的底线
 - 事务的基本保障是我们所需的在分布式计算中的异常处理
 - 事务是计算机中的合同法，如果出了什么问题，我们只需要放弃整个计算
- 4、我们还可以将异常看做一种内建的恢复系统，我们在程序中可以拥有不同的恢复点
- 5、异常最重要的一方面就是：如果发生问题，它将不允许程序沿着其正常的路径继续走下去

12.2.1. 异常参数

- 1、与 Java 中其他对象一样，我们使用 new 在堆上创建异常对象
- 2、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，以便能把相关信息放入异常对象的构造器
- 3、异常抛出使用关键字 throw
- 4、能够抛出任意类型的 Throwable 对象，它是异常类型的根类

12.3. 捕获异常

- 1、要明白异常是如何捕获的，必须首先理解**监控区域**的概念，它是一个可能产生异常的代码，并且后面跟着处理这些异常的代码

12.3.1. try 块

1、如果在方法内部抛出异常(或者在方法内部调用的其他方法抛出了异常), 这个方法将在抛出异常的过程中结束

2、要是不希望方法就此结束, 可以在方法内部设置一个特殊的块来捕获异常

```
try{
    //code
}
```

3、对于不支持异常处理的程序语言, 要想仔细检查错误, 就得在每个方法调用的前后加上设置和错误检查的代码, 甚至在每次调用同一方法时也得这么做

4、有了异常处理机制, 可以把所有动作都放在 try 块内, 然后只需要在一个地方捕获所有异常, 这意味着代码将更容易编写和阅读, 因为完成任务的代码没有与错误检查的代码混在一起

12.3.2. 异常处理程序

1、抛出的异常必须在某处得到处理, 这个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

```
try{
    //code
}catch(Type1 id1){
    //Handle exceptions of Type1
}catch(Type1 id2){
    //Handle exceptions of Type2
}...
```

2、每个 catch 子句(异常处理程序)看起来就像接受一个且仅接受一个特殊类型参数的方法

3、异常处理程序必须紧跟在 try 块之后

- 当异常被抛出时, 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
- 然后进入 catch 子句执行, 此时认为异常得到了处理, 一旦 catch 子句结束, 则处理程序的查找过程结束

4、终止与恢复:

- 异常处理理论上有两种基本模型: 终止模型和恢复模型
- 终止模型
 - Java 与 C++ 支持终止模型
 - 在这种模型中, 将假设错误非常关键, 以至于程序无法返回到异常发生的地方继续执行
 - 一旦异常被抛出, 就表明错误已经无法挽回, 也不能回来继续执行
- 恢复模型:
 - 异常处理程序的工作是修正错误, 然后重新尝试调用出问题的方法, 并认为第二次能成功
 - 如果想要 Java 实现类似恢复的行为, 那么在遇到错误时就不能抛出异常, 而是调用方法来修正该错误, 或者把 try 放在 while 循环内, 这样就不断进入 try 块, 直到得到满意的结果
- 恢复模型开始显得很吸引人, 但不是很实用, 主要原因可能是它所导致的耦合: 恢复性的处理程序需要了解异常抛出的地点, 这势必要包含依赖于抛出位置的非通用

性代码，这增加了代码编写和维护的困难

12.4. 创建自定义异常

- 1、要定义异常类，必须从已有的异常类继承，最好是选择意思相近的异常类来继承
- 2、建立新的异常类型最简单的方法就是让编译器为你产生默认构造器

12.4.1. 异常与日志记录

- 1、可以使用 `java.util.logging` 工具将输出记录到日志中
- 2、<未完成>

12.5. 异常说明

- 1、Java 鼓励人们把方法可能会抛出的异常告知使用此方法的客户端程序员，它使得调用者能确切知道些什么样的代码可以捕获所有潜在的异常
- 2、异常说明使用了附加的关键字 `throws`，后面接一个所有潜在异常类型的列表

```
void f() throws TooBig, TooSmall, DivZero{/*...*/}
```

- 3、表示不会抛出任何异常(除了从 `RuntimeException` 继承的异常，它们可以在没有异常说明的情况下被抛出)

```
void f() {/*...*/}
```

- 该方法不能抛出了 `RuntimeException` 继承体系之外的任何异常
- 如果该方法会产生异常，那么编译器会强制要求你就地解决异常

- 4、**代码必须与异常说明保持一致**，如果方法里的代码产生了异常却没有进行处理，编译器会发现这个问题并提醒你：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

- 5、可以声明方法将抛出异常，但是不抛出

- 编译器相信了这个声明，并强制此方法的用户像真的抛出异常那样使用这个方法
- 为异常先占个位置，以后就可以抛出这种异常而不用修改已有的代码，定义抽象基类和接口时这种能力很重要，这样派生类或接口的实现就能够抛出这些预先声明的异常

12.6. 捕获所有异常

- 1、可以只写一个异常处理程序来捕获所有类型的异常，通过捕获异常类型的基类 `Exception` 就能做到这一点

- 2、这将捕获所有异常，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

- 3、`Throwable` 的方法

- `Throwable.getMessage()`
- `Throwable.getLocalizedMessage()`
- `Throwable.toString()`
- `Throwable.printStackTrace()`
- `Throwable.printStackTrace(PrintStream)`
- `Throwable.printStackTrace(java.io.PrintWriter)`

- `Throwable.fillInStackTrace()`: 在 `Throwable` 对象内部记录栈帧的当前状态，返回一个 `Throwable` 对象，在原来对象的基础上，插入当前调用信息

12.6.1. 栈轨迹

1、`printStackTrace()`方法所提供的信息可以通过 `getStackTrace()`方法来直接访问

- 这个方法将返回一个由栈轨迹中的元素所构成的数组
- 其中每一个元素都表示栈中的一帧，元素 0 是栈顶元素，并且是调用序列中的最后一个方法，数组中最后一个元素和栈底是调用序列中的第一个方法调用

12.6.2. 重新抛出异常

1、有时希望把刚捕获的异常重新抛出，尤其是在使用 `Exception` 捕获所有异常的时候

```
catch(Exception e){  
    throw e;  
}
```

2、重新抛出异常会把异常抛给上一级环境中的异常处理程序，同一个 `try` 块后续的 `catch` 子句将被忽略，此外异常对象的所有信息都得以保持，所以高一级环境中捕获此异常的处理程序可以从这个异常对象中得到信息

3、异常抛出点

- 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
- 想要更新这个信息，可以调用 `fillInStackTrace()`方法，这将返回一个 `Throwable` 对象，它是通过把当前调用栈信息填入原来那个异常对象建立的，调用 `fillInStackTrace()` 的那一行就变成了异常的新发生地

12.6.3. 异常链

1、常常会想要在捕获一个异常后抛出另一个异常，并且把原始异常的信息保存下来，这被称为异常链

- 在 JDK1.4 以前，程序员必须自己编写代码来保存原始异常的信息
- 现在所有 `Throwable` 的子类在构造器中都可以接受一个 `cause` 对象作为参数，这个 `cause` 就表示原始异常
- 这样通过把原始异常传递给新异常，使得即使在当前位置创建并抛出了新的异常，也能通过这个异常链追踪到异常最初发生的位置

12.7. Java 标准异常

1、`Throwable` 这个 Java 类用被用来表示任何可以作为异常被抛出的类

2、`Throwable` 可分为两种类型

- `Error` 用来表示编译时和系统错误(除特殊情况外，一般不用你关心)
- `Exception` 是可以被抛出的基本类型，在 Java 类库，用户方法以及运行时故障中都可能抛出 `Exception` 型异常，所以 Java 程序员关心的基本型通常是 `Exception`

12.7.1. 特例：`RuntimeException`

1、如果必须对传递给方法的每个引用都检查是否为 `null`，这确实吓人，幸运的是，它属于 Java 标准库运行时检测的一部分，如果对 `null` 引用进行调用，Java 会自动抛出 `NullPointerException`

ception 异常

2、属于运行时异常的类型有很多，它们会自动被 Java 虚拟机抛出，所以不必再异常说明中把它们列出来，这些异常都是从 `RuntimeException` 类继承而来的

3、`RuntimeException` 继承体系不需要再异常说明中声明方法将抛出 `RuntimeException`，它们被称为“**不受检查异常**”，这种异常属于错误，将被自动捕获

4、`RuntimeException` 类型的异常 **也许(因为我们也可以捕获 `RuntimeException`)** 会穿越所有执行路径直达 `main` 方法

5、`RuntimeException` 类型的异常，其输出被报告给了 `System.err`

6、**请务必记住：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` 代表编程错误**

- 无法预料的错误，必须从你控制范围之外传递进来的 `null` 引用
- 作为程序员，应该在代码中进行检查的错误

7、**`RuntimeException` 可以被 `catch(Exception e)`或 `catch(RuntimeException e)`捕获，但不要去捕获它**

12.8. 使用 `finally` 进行清理

1、对于一些代码，可能会希望无论 `try` 块中的异常是否抛出，它们都能得到执行，这通常适用于内存回收之外的情况，为了达到这个效果，可以在异常处理程序后面加上 `finally` 子句

```
try{
    //code
}catch(A a1){
    //Handler for situationA
}catch(B b1){
    //Handler for situationB
}finally{
    //Activities that happen every time
}
```

2、**无论异常是否被抛出，`finally` 子句总能执行**

3、**注意**

- 在 `try` 块内创建的对象无法在 `finally` 中引用，因为作用域不同!!!
- 在 `try` 块内创建的非异常类型的对象，无法在 `catch` 中引用

12.8.1. `finally` 用来做什么

1、**对于没有垃圾回收器和析构函数自动调用机制**的语言来说，`finally` 非常重要，它能使程序员保证：无论 `try` 里发生了什么，内存总能得到释放

2、对于 Java 这一类有垃圾回收器或析构函数机制的语言来说，`finally` 用来清理资源，包括已经打开的文件或网络连接，在屏幕上画的图形，甚至可以是外部世界某个开关

12.8.2. 在 `return` 中使用 `finally`

1、**因为 `finally` 子句总是会执行，所以在一个方法中，可以从多个点返回，并且可以保证重要的清理工作仍会执行**

12.8.3. 遗憾：异常丢失

- 1、在 finally 中抛出异常：如果在 try 中抛出异常，并且在紧跟的 catch 子句中没有被捕获，但在 finally 中又抛出了一个异常，那么原来抛出的异常将会丢失
- 2、在 finally 中返回：如果在 try 中抛出异常，并且在紧跟的 catch 子句中没有被捕获，但是在 finally 中对函数进行返回，那么整个程序将会吞噬该异常，导致“正常”运行

12.9. 异常的限制

- 1、当覆盖方式时，只能抛出在基类方法的异常说明列表中的那些异常，这意味着，当基类使用的代码应用到其派生类时，一样能工作
- 2、**异常限制对构造器不起作用**
 - 派生类构造器可以抛出任何异常，而不必理会基类构造器所抛出的异常
 - 基类构造器必须以这样或那样的方式被调用，因此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
- 3、派生类构造器不能捕获基类构造器抛出的异常
- 4、**异常说明本身并不属于方法类型的一部分**，方法类型是由方法名字以及参数的类型组成的，因此不能基于异常说明来重载方法
- 5、**一个出现在基类方法的异常说明中的异常，不一定会出现在派生类方法的异常说明里，即某个方法的“异常说明的接口”变小而不是变大了，这与类接口在继承时的情形相反**
- 6、**派生类覆盖的方法所要抛出的异常类型，必须自己在异常说明中指明，而不会继承基类的异常说明**

12.10. 构造器

- 1、构造器除了把对象设置成安全的初始状态，还会有别的动作，比如打开一个文件，这样的动作只有在对象使用完毕并且用户调用了特殊的清理方法后才能得以清理，如果在构造器内抛出了异常，这些清理行为就不能正常工作了
- 2、通常在构造器中使用 try 语句块，然后在 catch 子句中进行出现异常时的关闭工作，但是不要放在 finally 中，因为如果正常创建对象的话，是不能在构造器中关闭的
- 3、**对于在构造阶段可能会抛出的异常，并且要求清理的类，最安全的使用方法是使用嵌套的 try 子句**

```
public InputFile(String fname) throws Exception{
    try{
        in=new BufferedReader(new FileReader(fname));
    }catch(FileNotFoundException e){
        System.out.println("Could not open "+fname);
        throw e;//重新抛出，因为这两个 catch 子句的目的是做清理工作
    }catch(Exception e){
        try{
            in.close();
        }catch(IOException e2){
            System.out.println("in.close() unsuccessful");
            throw e;
        }finally{//不要在这关闭
```

```

    }
}

public static void main(String[] args){
    try{
        InputFile in=new InputFile("Cleanup.java");
        try{//只要运行到这，就说明创建没有异常，可以执行清理工作
            String s;
            int i=1;
            while((s=in.getLine())!=null);
        }catch(Exception e){
            System.out.println("Caught Exception in main");
            e.printStackTrace(System.out);
        }finally{
            in.dispose();//创建成功后，无论使用时是否有异常都该清理
        }
    }catch(Exception e){//捕获创建时的异常
        System.out.println("InputFile construction failed");
    }
}

```

- 这种通用的清理惯用法在构造器不抛出异常时也应运用，其基本规则是：在创建需要清理的对象后，立即进入一个 **try-finally** 语句块

12.11. 异常匹配

- 1、抛出异常后，异常处理系统会按照代码书写顺序找出"最近"的处理程序，找到匹配的处理程序之后，它就认为异常得到处理，然后就不再继续查找，即**匹配最近而不是最优**
- 2、查找的时候并不要求抛出的异常同处理程序所声明的异常完全匹配

12.12. 其他可选方式

- 1、异常处理的重要原则是：只有在你知道如何处理的情况下才捕获异常
- 2、异常处理的重要目标：把错误处理的代码同错误发生的地点相分离
- 3、被检查的异常：它强制你在可能还没有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的问题

12.12.1. 历史

- 1、<未完成>

12.12.2. 观点

- 1、<未完成>

12.12.3. 把异常传给控制台

- 1、<未完成>

12.12.4. 把"被检查的异常"转换为"不检查的异常"

1、<未完成>

12.13. 异常使用指南

- 1、在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，然后绕过异常发生的地方继续执行
- 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做完，然后把相同的异常抛到更高层
- 6、把当前运行环境下能做的事情尽量做完，然后把不同的异常抛到更高层
- 7、终止程序
- 8、进行简化
- 9、让类库和程序安全

Chapter 13. 字符串

1、字符串操作是计算机程序设计中最常见的行为

13.1. 不可变 String

2、**String 对象是不可变的，每个看起来会修改 String 值的方法，实际上都是创建了一个全新的 String 对象**，以包含修改后的字符串内容，而最初的 String 对象则丝毫未动

13.2. 重载 "+" 与 StringBuilder

1、String 对象具有只读特性，但不可变性会带来一定的效率问题

2、用于 String 的 "+" 与 "+=" 是 Java 中仅有的两个重载过的操作符，Java 不允许程序员重载任何操作符

3、编译器会在需要"改变"String 对象的时候自动引入 StringBuilder

4、如要在一个循环中累加一个 String 对象，最好使用 StringBuilder(在循环之外创建)避免每次调用 "+" 编译器都帮你创建一个 StringBuilder

5、StringBuilder 是 Java SE5 引入的，在这之前 Java 用的是 StringBuffer，StringBuffer 是线程安全的，因此开销也更大，但是 StringBuffer 在单线程中的性能要更好

6、StringBuilder 的常用方法

- `StringBuilder.append(T t)`//T 可以是所有基本类型以及 String、CharSequence、StringBuffer 等
- `StringBuilder.replace(int start, int end, String str)`
- `StringBuilder.substring(int start[, int end])`
- `StringBuilder.reverse()`//返回 StringBuilder，这是反转一个 String 的唯一方法，通过 StringBuilder 做桥梁
- `StringBuilder.delete(int start,int end)`
- `StringBuilder.toString()`
- `StringBuilder.setLength(int newLength)`

13.3. 无意识的递归

1、Java 每个类从根本上都是继承自 Object，标准库容器也不例外，容器都有 `toString()` 方法，并且覆写了 `Object` 的该方法，使得它生成的 String 结果能够表达容器自身

2、**如果希望 `toString` 方法打印出对象的内存地址，不要使用 `this` 关键字，这会导致无意识的递归。** `this` 引用指向的是一个该类型的对象，如果 `this` 出现在一串 String 的由 "+" 连接的表达式中，会自动调用 `toString()` 方法，然后该 `toString()` 方法内又包含了 `this`，于是会导致无限递归

```
public String toString(){ //错误
    return "address: "+this+"\n";
}
```

1、如果想要打印地址，显式调用 `Object` 的 `toString()` 方法：`super.toString()`

13.4. String 上的操作

1、构造器：

- `String()`
- `String(String)`
- `String(StringBuilder)`
- `String(StringBuffer)`
- `String(char[])`
- `String(byte[])`

2、基本方法

- `String.length()`
- `String.charAt(int index)`
- `String.toCharArray()`
- `String.equals(Object anObject)`
- `String.trim()//删除两边的空白，返回新对象，若无空白返回原对象`
- `String.indexOf(int ch[, int fromIndex])//返回 ASCII 码 ch 第一次出现的索引，没有则返回-1，第二个参数可选，指明从哪个位置开始查找，返回的索引仍是正常的索引`
- `String.indexOf(String str[, int fromIndex])//返回 str 第一次出现的索引，没有则返回-1，第二个参数可选，指明从哪个位置开始查找，返回的索引仍是正常的索引`
- `String.toLowerCase()`
- `String.toUpperCase()`
- `String.contains(CharSequence s)`
- `String.replace(char oldChar, char newChar)//将所有旧字符替换为新字符`
- `String.replaceFirst(String regex, String replacement)`
- `String.replaceAll(String regex, String replacement)`

13.5. 格式化输出

1、Java SE5 推出了 C 语言中 printf 风格的格式化输出这一功能

13.5.1. printf()

1、C 语言中的 printf() 并不能连接字符串

2、printf() 使用特殊的占位符来表示数据将来的位置，并且插入格式化字符串的参数，以逗号分隔，即 args

13.5.2. System.out.format()

1、Java SE5 引入的 format 方法可用于 PrintStream 或 PrintWriter 对象

2、或者使用 `System.out.printf()`，如果你比较怀旧的话，这两个方法是一样的

13.5.3. Formatter 类

1、在 Java 中，所有新的格式化功能都是由 `java.util.Formatter` 类处理，可以将 Formatter 看做一个翻译器，它将你的格式化字符串与数据翻译成想要的结果

2、`Formatter.format()//用法类似于 System.out.format()`

3、`Formatter(T t)//T 可以是 PrintStream、OutputStream、File 等等)`

13.5.4. 格式化说明符

1、语法：

- ```
%[argument_index$][flags][width][.precision]conversion
```
- 中括号表示可选参数
  - **argument\_index\$(占位符)**: 表示此处由后面参数的第几个来填，占位符从 1 开始计  
System.out.format("%2\$d\n%1\$d\n",i1,i2);
  - **flags(转换标志)**: -表示左对齐，+表示在转换值之前要加上正负号，"(空白字符)"表  
示正数之前保留空格，0 表示转换值若位数不够则用 0 填充
  - **width(最小字段宽度)**: 转换后的字符串至少应该具有该值指定的宽度
  - **.precision(精度)**: 小数点的位数，**若与 width 冲突，则以.precision 为准**
  - **conversion(转换类型)**:
    - d: 整数型(十进制)
    - c: Unicode 字符
    - b: Boolean 值
    - s: String
    - f: 浮点数(十进制)
    - e: 浮点数(科学计数)
    - x: 整数(十六进制)
    - h: 散列吗(十六进制)
    - %: 字符%

### 13.5.5. String.format()

- 1、用以生成格式化的 String 对象
- 2、这是一个 static 方法，接受与 Formatter.format()一样的参数，但返回一个 String 对象

## 13.6. 正则表达式

- 1、很久之前，正则表达式就整合到标准 UNIX 工具集之中，例如 sed 和 awk
- 2、正则表达式是一种强大而灵活的文本处理工具
- 3、正则表达式提供了一种完全通用的方式，能够解决各种字符串处理相关的问题：匹配、选择、编辑以及验证

### 13.6.1. 基础

- 1、<未完成>

### 13.6.2. 创建正则表达式

| 字符     |                             |
|--------|-----------------------------|
| B      | 指定字符 B                      |
| \xhh   | 十六进制值为 oxhh 的字符             |
| \uhhhh | 十六进制表示为 oxhhhh 的 Unicode 字符 |
| \t     | 制表符                         |
| \n     | 换行符                         |
| \r     | 回车                          |
| \f     | 换页                          |

|                         |            |
|-------------------------|------------|
| \e                      | 转义(Escape) |
| \'                      | 单引号        |
| \"                      | 双引号        |
| 以上这些含有'\'的为转义(可能还有没列出的) |            |

| 字符类                                                                       |                              |
|---------------------------------------------------------------------------|------------------------------|
| .                                                                         | 任意字符                         |
| [abc]                                                                     | 包含 a、b、c 的任何字符(与 a b c 作用相同) |
| [^abc]                                                                    | 除了 a、b、c 之外的任何字符             |
| [a-zA-Z]                                                                  | 所有大小写字母                      |
| [a-zA-Z&&[hij]]                                                           | 任意的 h、i、j                    |
| \s                                                                        | 空白符                          |
| \S                                                                        | 非空白符                         |
| \d                                                                        | 数字[0-9]                      |
| \D                                                                        | 非数字[^0-9]                    |
| \w                                                                        | 词字符[a-zA-Z0-9]               |
| \W                                                                        | 非词字符[^\\w]                   |
| 以上这些含有'\'的表达式，'\'并不代表转义，而是一个普通字符，因此在写入 String 时需要用"\\"s"、"\\"d"、"\\"W"... |                              |

| 逻辑操作符 |          |
|-------|----------|
| XY    | Y 在 X 后面 |
| X Y   | X 或 Y    |
| (X)   | 捕获组      |

| 边界匹配符         |          |
|---------------|----------|
| ^             | 一行的起始    |
| \$            | 一行的结束    |
| \b            | 词的边界     |
| \B            | 非词的边界    |
| \G            | 前一个匹配的结束 |
| 以上这些含有'\'的为转义 |          |

### 13.6.3. 量词

1、量词描述了一个模式吸收输入文本的方式

- **贪婪型：**量词总是贪婪的，除非有其他的选项被设定，贪婪表达式会为所有可能的模式发现尽可能多的匹配
- **勉强型：**用问号来指定，这个量词匹配满足模式所需的最少字符数，因此也称作懒惰的、最少匹配的、非贪婪型、或不贪婪的
- **占有型：**目前这种类型的量词只有在 Java 语言中才能使用，当正则表达式被应用于字符串时，它会产生相当多的状态，以便在匹配失败时可以回溯，而“占有的”的量词并不保存这些中间状态，因此它们可以防止回溯，常用于防止正则表达式失控，使得正则表达式执行起来更有效

| 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, 最多 m 次 |

注意：以上 X，在实际中一般是用"()"括起来的表达式，以便能够按照我们的期望执行

## 2、CharSequence

- 从 CharBuffer、String、StringBuffer、StringBuilder 类之中抽象出来的接口

```
interface Charsequence{
 char charAt(int index);
 int length();
 CharSequence subSequence(int start, int end);
 public String toString();
}
```

- 多数正则表达式都接受 CharSequence 类型的参数

### 13.6.4. Pattern 和 Matcher

#### 1、Pattern

- **Pattern.compile(String regex)**: 返回 Pattern 的对象，将给定的正则表达式编译并赋予该返回的 Pattern 对象
- **Pattern.matcher(CharSequence input)**: 返回一个 Matcher 对象
- **Pattern.split(CharSequence input)**: Pattern 对象将 CharSequence 对象按 regex(利用 Pattern.compile(regex)生成 Pattern 对象 p)分割成多个部分，返回 String[]
- **Pattern.split(CharSequence input,int limit)**: 限制分割数量最多为 limit，达到 limit 就不再分割
- String 也有 split 方法，功能与 Pattern 的 split 方法类似
  - **String.split(String regex)**
  - **String.split(String regex,int limit)**
- Pattern 标记：Pattern 的 compile()方法还有另一个版本，接受一个标记参数，以调整匹配的行为
  - **Pattern.compile(String regex, int flag)**
  - **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
  - **Pattern.CASE\_INSENSITIVE(?i)**: 默认情况下，大小写不敏感的假设只有 US-ASCII 字符集中的字符才能进行，这个标记允许模式不考虑大小写进行匹配。通过指定 UNICODE\_CASE 标记及结合此标记，基于 Unicode 的大小写不敏感的匹配就可以开启了
  - **Pattern.COMMENTS(?x)**: 在此模式下，空格符将被忽略掉，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，默认情况下'.'不匹配行终结符

- **`Pattern.MULTILINE(?M)`**: 在多行模式下，表达式`^`和`$`分别匹配一行的开始和结束， 默认情况下，这些表达式仅匹配输入的完整字符串的开始和结束
- **`Pattern.UNICODE_CASE(?u)`**: 当指定该标记，并且开启 `CASE_INSENSITIVE` 时，大小写不敏感的匹配将按照 `Unicode` 标准相一致的方式进行，默认情况下，大小写不敏感的匹配只能在 `US-ASCII` 字符集中才能进行
- **`Pattern.UNIX_LINES(?d)`**: 在这种模式下，`'.'`、`'^'`，`'$'`的行为只识别行终结符`\n`
- 可以通过`|`组合多个标记功能

## 2、Matcher

- 目标字符串就是指创建 `Matcher` 对象时所绑定的字符串 `Matcher m=p.matcher(s);`
- `Matcher.matches()`: 对整个目标字符串展开匹配检测，也就是只有目标字符串与模式字符串完全匹配时才返回真值
- `Matcher.find()`: 尝试在目标字符串里查找下一个匹配子串(能够像迭代器那样前向遍历输入字符串)
- `Matcher.find(int start)`: 接受整数作为参数，该整数表示目标字符串中字符的位置，并以其作为搜索的起点，可以不断重新设定搜索的起始位置
- `Matcher.lookingAt()`: 检测目标字符串(不必是整个字符串)的始部分是否能够匹配模式字符串
- 简而言之，组就是放置在圆括号之内的子模式，**组的序号取决于它左侧的括号数，组 0 就是整个模式**
- 例如：
  - `A(B(C))D`: 组 0: ABCD 组 1: BC 组 2: C
  - `(a)(b)(c)`: 组 0: abc 组 1: a 组 2: b 组 3: c
- `Matcher.groupCount()`: 返回该匹配器的模式中的分组数目，不包括第 0 组
- `Matcher.group()`: 返回前一次 `find` 所匹配的子串
- `Matcher.group(int i)`: 返回在前一次 `find` 所匹配的子串中匹配组 `i` 的子串
- `Matcher.start(int group)`: 返回在前一次匹配操作中寻找到的组的起始索引
- `Matcher.end(int group)`: 返回在前一次匹配操作中寻找到的组的最后一个字符索引加一，即右开区间的边界
- `Matcher.replaceFirst(String replacement)`: 用参数字符串替换掉第一个匹配的地方
- `Matcher.replaceAll(String replacement)`: 用参数字符串替换掉所有匹配的地方
- **以上这两种替换方式，参数 `replacement` 只能是固定字符串，不如直接利用 `String` 的同名方法**
- `Matcher.appendReplacement(StringBuffer sbuf, String replacement)`: 参数 `replacement` 可以是表达式
- 渐进式的替换：
  - 调用 `find()` 会记录上一次调用 `find()` 或 `find(int i)` 的记录
  - **调用 `find(int i)` 会清除之前的 `find()` 或 `find(int i)` 的调用，本次调用作为第一次调用**
  - 首先，会将本次调用 `find` 所匹配的子串与上一次 `find` 调用所匹配的子串之间的子串存入 `sbbuf`(若本次是第一次调用，那么会将本次调用 `find` 所匹配的子串之前的子串存入 `sbbuf`)
  - 然后，将本次调用 `find` 所匹配的子串用 `replacement` 替换后存入 `sbbuf`
- `appendTail(StringBuffer sbuf)`: 会将最近一次调用 `appendReplacement(StringBuffer sbuf, String replacement)` 之后的子串存入 `sbbuf`。
- 调用 `find(int i)` 会重置目标子串，以 `i` 作为起始位置，并且清除之前 `appendReplac`

- `ement(StringBuffer sbuf, String replacement)`调用的记录
- `reset()`: 可以将现有的 Matcher 应用于新的目标字符串

### 13.6.5. `split()`

- 1、`split()`方法将输入字符串断开成字符串对象数组
- 2、这是一个非常常用的方法

### 13.6.6. 替换操作

- 1、正则表达式特别便于替换文本，提供了许多方法，详见 13.6.4
- `replaceFirst(String replacement)`
- `replaceAll(String replacement)`

### 13.6.7. `reset()`

- 1、详见 13.6.4

## 13.6.8. 正则表达式与 Java I/O

## 13.7. 扫描输入

- 1、Java SE5 新增了 Scanner 类，它可以大大减轻扫描输入的工作负担
- 2、Scanner 的构造器可以接受任何类型的对象，包括 File 对象、InputStream、String、Readable 对象

### 13.7.1. Scanner 定界符

- 1、在默认情况下，Scanner 根据空白字符对输入进行分词，但是你可以用正则表达式指定自己所需的定界符
  - `Scanner.useDelimiter(String pattern)`
  - `Scanner.delimiter()`: 返回当前作为定界符使用的 Pattern 对象

### 13.7.2. 用正则表达式扫描

- 1、当 `next()`方法配合正则表达式使用时，找到下一个匹配该模式的输入部分，调用 `match()`方法就可以获得匹配的结果
  - 注意：它仅仅针对下一个输入分词(不包含定界符)进行匹配，如果你的正则表达式包含定界符，那么永远都不会匹配成功

## 13.8. StringTokenizer

- 1、在 Java 引入正则表达式(J2SE1.4)和 Scanner 类(Java SE5)之前，分隔字符串的唯一方法就是使用 StringTokenizer
- 2、**基本上，StringTokenizer 可以废弃了**

## Chapter 14. 类型信息

- 1、运行时类型信息使得你可以在程序运行时发现和使用类型信息
- 2、它使你从只能在编译时期执行面向类型的操作的禁锢中解脱出来，并且可以使用某些非常强大的程序
- 3、对 RTTI 的需要，解释了面向对象设计中许多有趣(并且复杂)的问题，同时也提出了如何组织程序的问题
- 4、Java 在运行时识别对象和类的信息，主要有两种方式
  - 一种是传统的 RTTI，它假定我们在编译时已经知道了所有的类型
  - 另一种是反射机制，它允许我们在运行时发现和使用类的信息

### 14.1.1. 为什么需要 RTTI

- 1、RTTI 名字的含义：在运行时，识别一个对象的类型
- 2、以 Shape 继承体系来说明
  - 把 Shape 对象(或者导出类的对象)放入 List<Shape>时就会向上转型，但在向上转型为 Shape 的时候也丢失了 Shape 对象的具体类型，对该容器而言，它们只是 Shape 类型
  - 当从容器中取出元素时，实际上容器将所有的事物都当做 Object 持有，**会自动将结果转型为 Shape，这是 RTTI 最基本的使用形式**
  - 在这个例子中，RTTI 类型转换并不彻底，Object 被转型为 Shape，而不是转型为 Circle、Square 或者 Triangle，因为目前我们只知道这个 List<Shape>保存的都是 Shape，在编译时，将由容器和 Java 的泛型系统来强制确保这一点，在运行时，由类型转换操作来确保这一点
  - 接下来就是多态的事情了，Shape 对象实际执行什么样的代码，是由引用所指向的具体对象 Circle、Square 或者 Triangle 决定的
- 3、**但有时候你需要明确知道该对象的确切类型而不是其基类或者接口的类型，来执行某些特殊的操作，这时就需要类型信息了**

## 14.2. Class 对象

- 1、要理解 RTTI 在 Java 中的工作原理，首先必须知道类型信息在运行时是如何表示的
  - 这项工作是由称为 **Class 对象** 的特殊对象完成的，它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“常规”对象的
  - Java 使用 Class 对象来执行 RTTI，即使你正在执行的是类似转型这样的操作
- 2、类是程序的一部分，每个类都有一个 Class 对象，每当编写了一个新类，就会产生一个 Class 对象(更恰当地说，是被保存在一个同名的.class 文件中)，为了生成这个类的对象(不是指 Class 对象，而是普通对象)，**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**
- 3、类加载器子系统实际上可以包含一条类加载器链，但是只有一个原生类加载器，它是 JVM 实现的一部分
  - 原生类加载器加载的是所谓的可信类，包括 Java API 类，它们通常是从本地盘加载的，在这条链中，通常不需要添加额外的类加载器
  - 若你有特殊需求(例如以某种特殊的方式加载类，以支持 Web 服务器应用，或者在网络中下载类)，那么你有一种方式可以挂接额外的类加载器

4、所有的类都是在对其第一次使用时，动态加载到 JVM 中的，当程序创建第一个对类的静态成员的引用时，就会加载这个类，这个证明构造器也是类的静态方法，即使在构造器之前并没有使用 static 关键字，**因此使用 new 操作符创建类的新对象也会被当做对类的静态成员的引用**

5、类加载器首先检查这个类的 Class 对象是否已经加载，如果尚未加载，默认的类加载器就会根据类名查找.class 文件(例如，某个附加类加载器可能会在数据库中查找字节码)，在这个类的字节码被加载时，它们会接受验证，以确保其没有被破坏，并且不包含不良 Java 代码

6、一旦某个类的 Class 对象被载入内存，它就用来创建这个类的所有对象

7、Class 对象仅在需要的时候才被加载，static 初始化是在类加载时进行的

8、无论何时，只要你想在运行时使用类型信息，就必须首先获得对恰当 Class 对象的引用

- `Class.forName(String className)`: 通过一个 String 表示的类名，获取 Class 对象，有一个副作用：如果该类没有加载就加载它
- `object.getClass()`: 通过实例获取 Class 对象
- `Object.class`: 通过类名获取 Class 对象

9、Class 的方法

- `Class.forName(String name)`: **获取 Class 对象的引用，静态方法，其中 name 必须包含包名**
  - 其副作用：如果类 name 还没有被加载就加载它。
  - **当找不到类 name 时，抛出 ClassNotFoundException 异常(动态的，而?.class 是静态的，不必用 try)**
- `Class.getSimpleName()`: 获取不含包名的类名
- `Class.getCanonicalName()`: 获取包含包名的类名(全限定的类名)
- `Class.getName()`: 获取包含包名的类名(全限定的类名)
- `Class.getSuperclass()`: **查询其直接基类，返回的是基类的 Class 对象(即使该 Class 对象为泛型：某类型的 Class 对象，该函数返回的对象也不能用泛型，只能使用普通 Class 或超类 Class<?super A>)**
- `Class.newInstance()`: 创建类的对象(**该类必须含有默认构造器**)**返回的是 Object 类型的引用**(当然实际指向的还是确切的类型的对象，在你进行转型之前，不能向该引用发送 Object 之外的消息)
  - 该方法是实现"虚拟构造器"的一种途径，虚拟构造器允许声明：我不知道你的确切类型，但是无论如何要正确地创建你自己
  - **泛型 Class 对象的 newInstance() 将返回该确切类型的对象**
  - **若泛型为 Class<? extends Pet>那么返回对象会自动转型为 Pet**
  - **会抛出两种异常 InstantiationException 和 IllegalAccessException**
- `Class.isAssignableFrom(Class a)`: 该函数检查传入的参数 a 是否为该类型或其导出类
- `Class.isInstance(obj)`: 检查实例 obj 是否属于该类型

#### 14.2.1. 类字面常量

1、Java 还提供了一种方法来生成对 Class 对象的引用，即使用类字面常量

- 这样做不仅更简单，而且更安全，因为它在编译时就会收到检查(因此不需要 try 语句块，而 `Class.forName()` 需要 try 语句块)
- **类字面常量不仅可以应用于普通的类，也可以应用于接口，数组以及基本数据类型**
  - 对于基本数据类型的包装类，还有一个标准字段 TYPE，TYPE 字段是一个引用，指向对应基本数据类型的 Class 对象

- Boolean.TYPE Integer.TYPE 等等
- 但是建议使用.class 的形式，保持与普通类的一致性

## 2、当使用.class 来创建 Class 对象的引用时，不会自动地初始化该 Class 对象

(`Class.forName()`会导致初始化该类)，为了使用类而做的准备工作实际包含三个步骤：

- **加载**：这是由类加载器执行的，该步骤将查找字节码，并从这些字节码中创建一个 Class 对象
- **链接**：在链接阶段将验证类中的字节码，为静态域分配存储空间，并且如果必须的话，将解析这个类创建的对其他类的所有引用
- **初始化**：如果该类具有超类，则对其初始化，执行静态初始化器和静态初始化块
- 初始化被延迟到了对静态方法(构造器隐式静态的)或者非常数静态域进行首次引用时才执行
- static final int staticFinal=47;对 staticFinal 进行访问时，不会导致该类被初始化，这是一个**编译器常量**

### 14.2.2. 泛化的 Class 引用

1、Class 引用总是指向某个 Class 对象，它可以制造类的实例，并包含可作用于这些实例的所有方法代码，它还包含该类的静态成员，因此 Class 引用表示的就是他所指向的对象的确切类型，而该对象便是 Class 类的一个对象

2、普通的 Class 引用可以被重新赋值为指向任何其他的 Class 对象，通过使用泛型语法，可以让编译器强制执行额外的类型检查

3、为了使用泛化的 Class 引用时放松限制，我们使用了通配符，它是 Java 泛型的一部分，通配符就是"?"，表示"任何事物"

- 在 Java SE5 中，`Class<?>`优于平凡的 Class，即便它们是等价的
- 平凡的 Class 不会产生编译器警告信息
- `Class<?>`的好处是它表示你并非是碰巧或者由于疏忽，而使用了一个非具体的类的引用，表明你就是选择非具体的版本
- 创建一个限定类型的 Class 对象，`Class<? extends A>`
- 向 Class 引用添加泛型语法的原因仅仅是为了提供编译器的类型检查，如果使用普通的 Class 引用，那么你直到运行时才会发现错误
- 如果你手头的是超类，即 `Class<? Super A>`，那么编译器只允许你声明超类引用是某个类<没看懂>P322

### 14.2.3. 新型的转型语法

1、Java SE5 添加了用于 Class 引用的转型语法，即 `cast()`

- `cast()`方法接受参数对象，并将其转型为 Class 引用的类型
- 在整个 Java SE5 类库，只有一处用到了 `cast()`

2、另一个不太有用的特性就是 `Class.asSubclass()`，该方法允许你将一个对象转型为更加具体的类型

## 14.3. 类型转换前先做检查

1、RTTI 的形式

- 传统的类型转换，如"(A)a"，由 RTTI 确保类型转换的正确性，如果执行了一个错误的类型转换，就会抛出一个 `ClassCastException` 异常

- C++中，经典的类型转换"(A)a"并不适用 RTTI，它只是简单地告诉编译器将这个对象作为新的类型对待，若要获取类型检查，则必须使用 `static_cast<A>(a)`
- Java 要执行类型检查，这个通常被称为"类型安全的向下转型"
  - ◆ 以 `Shape` 为例
  - ◆ 编译器无法知道对于给定 `Shape` 到底是什么 `Shape`---可能就是 `Shape`，或者 `Circle`、`Square`、`Triangle` 等，在编译器，编译器只能知道它是 `Shape`，如果不使用显式的类型转换，编译器就不允许你执行向下转型赋值
  - ◆ 通过显式的类型转换，告知编译器你拥有额外的信息，这些信息使你知道该类型是某种特定的类型
- 代表对象的类型的 `Class` 对象，通过查询 `Class` 对象可以获取运行时所需的信息
- RTTI 在 Java 中还有第三种形式：`instanceof`
  - 它返回一个布尔值，告诉我们对象是不是某个特定类型的实例
  - `instanceof` 有比较严格的限制：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`

1、`Class.isInstance` 方法提供了一种动态测试对象的途径

#### 14.3.3. 递归计数

### 14.4. 注册工厂

### 14.5. `instanceof` 与 `Class` 的等价性

- 1、在查询类型信息时，以 `instanceof` 的形式与直接比较 `Class` 对象有一个重要的差别
- `instanceof` 保持了类型的概念，它指的是：你是这个类吗？或你是这个类的派生类吗？
  - 如果用 `Class.equals()` 或 `==` 来比较 `Class` 对象，就没有考虑继承，它就只是指明确的该类型

### 14.6. 反射：运行时的类型信息

1、如果不知道某个对象的确切类型，RTTI 可以告诉你，但是有一个限制：**这个类在编译时必须已知，这样才能使用 RTTI 识别它，并利用这些信息做一些有用的事情，换句话说，在编译时，编译器必须知道所有要通过 RTTI 来处理的类**

- 基于构建的编程
  - 人们想要在运行时获取类的信息的另一个动机，希望提供在跨网络的远程平台上创建和运行对象的能力，这被称为**远程方法调用(RMI)**
- 2、`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`、`Constructor` 类(每个类都实现了 `Member` 接口)
  - 这些类型的对象是由 JVM 在运行时创建的，用以表示位置类里对应的成员
  - 可以使用 `Constructor` 创建新的对象

- 用 `get` 和 `set` 方法读取和修改与 `Field` 对象相关联的字段
- 用 `invoke()`方法调用与 `Method` 对象相关联的方法
- 可以调用 `getFields()`、`getMethods()`和 `getConstructors()`等很便利的方法，以返回表示字段、方法以及构造器的对象的数组，这样匿名对象的类型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
- 关于 `getMethod*()`和 `getDeclaredMethod*()`
  - `public Method[] getMethods()`返回某个类的所有公用(public)方法包括其继承类的公用方法，当然也包括它所实现接口的方法
  - `public Method getMethod(String name, Class<?>... parameterTypes)`同上
  - `public Method[] getDeclaredMethods()`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。当然也包括它所实现接口的方法
  - `public Method getDeclaredMethod(String name, Class<?>... parameterTypes)`同上

### 3、反射机制并没有什么神奇之处

- 当通过反射与一个未知类型的对象打交道时，JVM 只是简单地检查这个对象，看它属于哪个特定的类(就像 RTTI 那样)
- 在用它做其他事情之前必须先加载那个类的 `Class` 对象，因此类的 `.class` 文件对于 JVM 必须是可获取的
  - 要么在本地机器上
  - 要么通过网络取得
- RTTI 和反射之间真正的区别在于：
  - 对于 RTTI，编译器在编译时打开和检查 `.class` 文件，换句话说，我们可以用普通的方式调用对象的所有方法
  - 对于反射机制，`.class` 文件在编译时是不可获取的，所以是在运行时打开和检查 `.class` 文件

#### 14.6.1. 类方法提取器

- 1、通常你不需要直接使用反射工具，但是它们在你需要创建更加动态的代码中会很有用
- 反射在 Java 中是用来支持其他特性的，例如对象序列化和 JavaBean
  - 另外一个就是用于编写自动展示完整接口的简单工具，因为查找类定义的源代码和 JDK 文档会相当的乏味，而且可能很费时

#### 14.7. 动态代理

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

2、在任何时刻，只要你想要将额外的操作从“实际”对象中分离到不同的地方，特别是当你希望能够很容易地做出修改，从没有使用额外操作转为使用这些操作，或者反过来说，代理就显得很有用(设计模式的关键就是封装修改---因此你需要修改事务以证明这种模式的正确性)

3、动态代理比代理的思想更向前了一步，因为他可以动态地创建代理并动态的处理对所代理方法的调用

4、在动态代理上所做的所有调用都会被重定向到单一的调用处理器上，它的任务是揭示调

用的类型并确定相应的对策

## 5、通过调用静态方法 `Proxy.newProxyInstance()`可以创建动态代理

- 这个方法需要一个类加载器(通常可以从已经被加载的对象中获取其类加载器，然后传递给它)
- 一个你希望代理实现的接口列表(不是类或抽象类)
- `InvocationHandler` 接口的一个实现
  - 只含有一个方法(该方法会被自动调用)

```
public Object invoke(Object proxy, Method method, Object[] args) throws Throwable;
```

- `proxy`: 代理类的对象
- `method`: 会传入该代理类正在尝试调用的方法
- `args`: 调用该方法的参数列表
- 对接口的调用会被定向为对代理类的调用

- 例子:

```
interface Business{
 void trade();
}

class Client implements Business{
 public void trade(){
}
}

class Handler implements InvocationHandler{
 Object object;
 Handler(Object object){this.object=object;}
 public Object invoke(Object proxy, Method method, Object[] args){
 method.invoke(object, args) return null
 //或者
 //do something extra
 return method.invoke(proxy, args); //对接口的调用将被重定向为对被代理
 对象的调用
 }
}

public static void main(String[] args){
 Business obj=(Business) Proxy.newProxyInstance(
 Business.class.getClassLoader(),
 new Class[]{Business.class},
 new Handler(new Client));
 obj.trade();
}
```

## 6、动态代理可以将所有调用重定向到调用处理器，因此通常会像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()`方法中传递进来了代理对象，以防你需要区分请求的来源，但是在许多情况下，你并不关心这一点
- 在 `invoke()`内部，在代理上调用方法时需要格外小心，**因为对接口的调用将被重定向为对被代理对象的调用**

## 7、静态代理的优缺点

- 优点
  - 代理使客户端不需要知道实现类是什么，怎么做的，客户只需要知道代理即可(解耦和)
- 缺点
  - 代理类和委托类实现了相同的接口，代理类通过委托类实现了相同的方法。这样就出现了大量的代码重复。如果接口增加一个方法，除了所有实现类需要实现这个方法外，所有代理类也需要实现这个方法。增加了代码维护的复杂度
  - 代理对象只服务于一种类型的对象，如果要服务多类型的对象，必须要为每一种对象都进行代理，静态代理在程序规模稍大时就无法胜任了

## 8、动态代理的优缺点

- 优点：动态代理与静态代理相比较，最大的好处是接口中声明的所有方法都被转移到调用处理器---一个集中的方法中处理(`InvocationHandler.invoke`)。这样在接口方法比较多的时候，我们可以进行灵活处理，而不需要像静态代理那样重写每一个方法
- 缺点

## 9、总结

- 静态代理：一个代理类只能代理一个类
- 动态代理可以动态绑定不同的的委托类
- 代理模式
  - 抽象角色：声明真实对象和代理对象的共同接口
  - 代理角色：代理对象角色内部含有对真实对象的引用(也许是接口的引用？)，从而可以操作真实对象，同时代理对象提供了真实对象相同接口以便任何时候都能替代真实对象。同时，代理对象可以对可以在执行真实对象操作时，附加其他的操作，相当于对真实对象进行封装。
  - 真实角色：代理角色所代表的真实对象，是我们最终要引用的对象。
- 客户端通过代理来实现对真实角色的操作

## 14.8. 空对象

- 1、当使用内置的 `null` 表示缺少对象时，在每次使用引用时都必须测试其是否为 `null`，这显得枯燥，而且势必产生非常乏味的代码
- 2、`null` 除了在你试图它执行任何操作来产生 `NullPointerException` 之外，它自己没有其他任何行为
- 3、空对象的思想会很有用，他可以接受传递给它的消息，但是将返回表示为实际上并不存在任何“真实”对象的值，通过这种方式，可以假设所有的对象都是有效的，而不必浪费精力去检查 `null`
- 4、空对象可用作一种特殊的值，例如红黑树里的哨兵节点
- 5、<未完成>

### 14.8.1. 模拟对象和桩

- 1、<未完成>

## 14.9. 接口与类型信息

- 1、`interface` 关键字的一种重要目标是允许程序员隔离构件，进而降低耦合性，但是通过类型信息这种耦合性还是会传播出去---接口并非是对解耦的一种无懈可击的保障
- 2、通过反射可以访问私有的方法或域，以及(匿名)内部类

➤ `Method\Field\Constructor.setAccessible(true);`

## Chapter 15. 泛型

- 1、一般的类和方法，只能使用具体的类型，要么是基本类型，要么是自定义的类，如果要编写可以应用于多种类型的代码，这种刻板的限制对代码的束缚就会很大
- 2、泛型这个属于的意思是：适用于许多的类型

### 15.1. 与 C++ 的比较

- 1、Java 设计的灵感主要来自 C++

### 15.2. 简单泛型

- 1、泛型最引人注目的一个原因，就是创造容器类，容器是最具重用性的类库之一
- 2、泛型的主要目的之一就是用来指定容器要持有什么样类型的对象，而且由编译器来保证类型的正确性

#### 15.2.1. 一个元组类库

- 1、我们希望一次方法调用就能返回多个对象，但是 return 只允许返回单个对象，解决办法就是创建一个对象，用它来持有想要返回的多个对象
- 2、这个概念称为“元组”，它是将一组对象直接打包存储于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(这个概念也被称为数据传送对象或信使)

#### 15.2.2. 一个堆栈类

- 1、<未完成>

#### 15.2.3. RandomList

### 15.3. 泛型接口

- 1、泛型可以应用于接口，例如**生成器**，这是一种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
  - 当使用生成器创建新的对象时，它不需要任何参数，而工厂方法一般需要参数
  - 也就是说生成器无需额外的信息就知道如何创建对象
- 2、new.mingview.util.Generator 的实现
- 3、**Java 泛型的一个局限性：基本类型无法作为参数**

### 15.4. 泛型方法

- 1、定义泛型方法：只需将泛型参数列表置于返回值之前

```
public <T> void f(T x){}
```
- 2、在类中包含参数化方法，而这个方法所在的类可以是泛型类，也可以不是泛型类
- 3、泛型方法使得该方法能够独立于类而产生变化
- 4、对于一个 static 的方法，无法访问泛型类的类型参数，所以如果 static 方法需要使用泛型

能力，就必须使其成为泛型方法

5、使用泛型类时，必须在创建对象的时候指定类型参数的值，而使用泛型方法的时候，通常不必指明参数类型，因为编译器会为我们找出具体的类型，称为**类型参数推断**

6、**基本原则：无论何时，只要你能做到，就应该尽量使用泛型方法。也就是说，如果使用泛型方法可以取代整个类泛型化，那么久应该只是用泛型方法**

#### 15.4.1. 杠杆利用类型参数推断

1、人们对泛型有一个抱怨，使用泛型有时候需要向程序中加入更多的代码

- `ArrayList<String> ary=new ArrayList<String>();`
- 重复指定了 String

2、**类型推断只对赋值操作有效，其他时候并不起作用**

- 如果你将一个泛型方法调用的结果作为参数，传递给另一个方法，这时编译器并不会执行类型推断，在这种情况下，编译器认为：调用泛型方法后，其返回值被赋给一个 Object 类型的变量

3、泛型方法的显式类型说明

- 必须在**点操作符与方法名**之间插入尖括号，把类型置于尖括号内

4、该小结要说明的意思

- 使用泛型方法，利用编译器来推断参数可以帮助我们简化代码，但是在不了解具体类库的情况下会造成困惑
- 若显式指定了类型说明，那么泛型方法带来的好处就抵消了
- 需要进行权衡

#### 15.4.2. 可变参数与泛型方法

1、泛型方法和可变参数可以很好地共存

```
public static <T> List<T> makeList<T...args> {}
```

#### 15.4.3. 用于 Generator 的泛型方法

1、<未完成>

#### 15.4.4. 一个通用的 Generator

1、Java 泛型要求传入 Class 对象，以便可以在方法中用它进行类型推断

#### 15.4.5. 简化元组的使用

1、<未完成>

#### 15.4.6. 一个 Set 实用工具

1、<未完成>

### 15.5. 匿名内部类

1、泛型可以用于内部类和匿名内部类

2、**工厂方法，工厂域**的实现

## 15.6. 构建复杂模型

1、泛型的一个重要好处是能够简单而安全的创建复杂的模型

## 15.7. 擦除的神秘之处

1、当你开始深入地钻研泛型时，会发现有大量的东西初看起来没有意义

- 例如可以声明 `ArrayList.class`, 但是不能声明 `ArrayList<Integer>.class`

2、`Class.getTypeParameters()`将返回一个 `TypeVarialbe` 对象数组，表示泛型声明所声明的类型参数

- 注意，指的仅仅是 `T,U,V` 这样的占位符，而非真正的泛型参数

3、**在泛型代码内部，无法获得任何有关泛型参数类型的信息**

- 我们可以知道诸如**类型参数标识符(即 T 这样的东西)**和**泛型边界**这类的信息，但却无法知道用来创建某个特定实例的实际的类型参数

4、**Java 的泛型是用擦除来实现的**

- 这意味着当你在使用泛型时，任何具体的类型信息都被擦除了(**运行时被擦除???**)

- 唯一知道的就是你在使用一个对象

- 因此例如 `List<Integer>` 和 `List<String>` 在运行时事实上是相同的类型，这两种类型都被擦除成它们的“原生”类型，即 `List`

### 15.7.1. C++的方式

1、用 C++ 编写泛型代码很简单，因为当模板被实例化时，模板代码知道其模板参数的类型

- 例如可以编写如下代码

```
template <typename T>
class Manipulator{
 T obj
public:
 Manipulator(T x){obj=x;}
 void manipulate(){obj.f();}
};
```

- 编译器会在运行时验证，实例化参数是否具有 `f` 方法，若没有则会报错

- 但 Java 无法写出类似代码

2、**要在 Java 中写出上述代码，必须指定泛型边界**

- 利用 `extends` 关键字

- 泛型类型参数将擦除到它的**第一个**边界

- 编译器实际会把类型参数替换为它的擦除

3、只有当你希望使用的类型参数比某个类型更加泛化时---也就是说你希望代码能够跨多个类工作时，使用泛型才有所帮助

### 15.7.2. 迁移兼容性

1、Java 泛型是后来加入的，用擦除来实现泛型是折中的考虑

2、擦除减少了泛型的泛化性

3、泛型类型只有在静态类型检查期间才出现，在伺候，程序中的所有泛型类型都将被擦除，替换为它们的非泛型上界，**普通的类型变量在未指定边界时将被擦除为 Object**

4、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，这被称为迁移兼容性，通过允许非泛型代码与泛型代码共存，擦除使得这种向着泛型的迁移成为可能

### 15.7.3. 擦除的问题

1、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

- 擦除使得现有的非泛型客户端代码能够在不改变的情况下继续使用，直至客户端准备用泛型重写这些代码
- 这是一个崇高的动机，因为它不会突然间破坏所有现有的代码
- 比如之前的 `List` 存的就是 `Object`，**如果不用擦除实现泛型，那么以前的代码就全部是错了**

2、擦除的代价是显著的，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` 操作符
- `new` 表达式

### 15.7.4. 边界处的动作

1、擦除在方法或类内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

2、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，这些正是**编译器在编译期执行类型检查(进入)并插入转型代码的地点(离开)**

- 编译器如何知道该对象的实际类型？通过调用 `Object.getClass()` 即可获取对象实际的类型，而非引用的类型
  - `Fruit f=new Apple();`
  - `f.getClass()-->Apple`

3、以访问器和编译器为例(`T get();void set(T t)`)

- 转型是在调用 `get()` 的时候接受检查的
- 对进入 `set()` 的类型进行检查是不需要的，这将由编译器执行
- 从 `get()` 返回值进行转型仍旧是需要的，但这与你自己必须执行的操作是一样的---此处它将由编译器自动插入，因此你写入(和读取)的代码噪声更小

```
String s=obj.get();
String s=(String)obj.get();
```

4、在泛型中的所有动作都发生在边界处

- 对传进来的值进行额外的编译器检查(如 `set`)
- 对传出去的值进行类型转换(编译器自动帮我们插入类型转换，如 `get`)

5、对于以下类似的方法

```
class A<T>{
 Object obj;
 void set(T obj){this.obj=obj;}
 T get(){ return (T)obj;}//方法中的类型转换(T)是无用的，因为 T 被擦除到了
 //Object，实际上就是从 Object 转换为 Object
}
class B<T>{
 T obj;
```

```

 void set(T obj){this.obj=obj;}
 T get(){ return (T)obj;}//方法中的类型转换(T)仍然是无用的(同上), 但是, 方法返
 回时会由编译器插入自动地类型转换
}

```

## 15.8. 擦除的补偿

- 1、擦除丢失了在泛型代码中执行某些操作的能力，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
- 2、我们可以绕过这些问题来编程，但有时必须通过引入**类型标签**来对擦除进行补偿
  - 类型标签：Class 对象
  - 在类中设置一个 Class<T>的字段，用以存储类型 T 的实际类型的 Class 对象

### 15.8.1. 创建类型实例

- 1、在 Java 泛型中无法利用 new T() 来创建一个实例
  - 第一个原因：擦除
  - 第二个原因：编译器不能验证 T 具有默认(或无参)构造器
- 2、Java 的解决方案是**传递一个工厂对象**，并且用它来创建新的实例
  - 最便利的工厂对象就是 Class 对象，**Class 对象是内建的工厂**
    - 于是可以用 newInstance() 来创建这个类型的新对象(当然，前提是必须有默认(无参)构造器)，或者利用反射调用其他构造器
    - 但是如果出错，那么这个错误将不会在编译期间被捕获
  - **传递显式工厂**
    - 可以获得编译器检查
  - **模板方法设计模式**
    - ???

### 15.8.2. 泛型数组 (T[])

- 1、不能创建泛型数组，一般的解决方案是在任何想要创建泛型数组的地方都使用 ArrayList
  - ArrayList 将提供数组的行为，以及由泛型提供的编译期的类型安全
- 2、**可以声明一个泛型数组，但不能创建一个泛型数组的实例，即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**

```
ArrayList<Integer>[] ary=new ArrayList[2];
```
- 3、因为有了擦除，泛型数组的运行时类型就只能是 Object[]
- 4、<未完成>，没看懂???
- 5、在内部将 T[] 写成 Object[] 的好处：提示这个数组运行时的类型，即便写成 T[]，也等同于 Object[]，但是会让你感到迷惑(可能误以为 T 在运行时的类型不是 Object)

## 15.9. 边界

- 1、因为擦除移除了类型信息，所有，可以用无界泛型参数调用的方法只能是那些可以用 Object 调用的方法
- 2、若将这个参数限制为某个类型的子类，那么就可以用这些类型自己来调用方法，为了执行这种限制，Java 泛型重用了 extends 关键字

## 15.10. 通配符

1、数组的一种特殊行为：可以向导出类型的数组赋予基类型的数组引用(基类数组的引用可以引用导出类数组的对象实例)

```
Fruit[] fruit=new Apple[10];
```

➤ 一旦上述行为发生，那么类似下面的语句将在运行时抛出异常

```
fruit[0]=new Fruit[];
```

- 尽管在编译器，看起来很合法，基类数组的引用怎么不能持有基类呢，但是运行时会报错

- 数组对象可以保留有关它们包含的对象的类型的规则

```
fruit.getClass() --> Apple[]
```

2、数组可以持有实际的类型及其子类

3、泛型的主要目的就是将上述这种错误移动到编译期，即不允许类似上述 fruit 定义的语句发生，即不能将 Apple 的容器赋值给 Fruit 的容器

➤ Apple 的 List 不等价于 Fruit 的 List，即使 Apple 是一种 Fruit 的类型

```
List<Fruit> fruit2=new ArrayList<Apple>();
```

4、注意，以上讨论的重点是数组或容器的类型，而不是数组或容器持有的类型

5、如果要让某个 List 持有某种具体的 Fruit 或 Fruit 的子类型，但你不关心具体到底是哪种，那么可以使用通配符

```
List<? extends Fruit> flist=new ArrayList<Apple>()
```

- 一旦上述语句执行，那么该 flist 将丢失掉向其传递任何对象的能力，即使 Object 也不行，即 add 语句是非法的
- 但是一个返回 Fruit 的方法是安全的，因为在这个 List 中的任何对象至少具有 Fruit 类型

### 15.10.1. 编译器有多聪明

1、Collection 的 contains 与 indexOf，其参数是 Object

2、其实编译器没有这么聪明，尽管 add 接受一个具有泛型参数类型的参数，但是 contains() 和 indexOf() 将接受 Object 类型的参数

3、当指定 ArrayList<? extends Fruit> 时，add() 的参数就变成了 "? Extends Fruit"，编译器并不能了解这里需要 Fruit 的哪个具体子类型，因此它不会接受任何类型的 Fruit---编译器将直接拒绝对参数列表中涉及通配符的方法的调用

### 15.10.2. 逆变

1、超类型通配符 <? super MyClass>

2、超类型通配符允许向 Collection 写入，因为实例的类型必然是指定类型的基类，因此可以向其传递指定类型及其子类

3、超类型通配符 super 指定了下界，因此你可以知道向上转型是否是安全的，而 extends 指定了上界，我们无法保证向上转型的正确性，因此禁止向其传递对象

4、<未完成>，没太懂

### 15.10.3. 无界通配符

1、无界通配符 <?> 看起来意味着“任何事物”，因此使用无界通配符好像等价于原生类型

2、`<?>`被认为是一种装饰，**声明：想用 Java 的泛型来编写这段代码，在这里并不想要用原生类型**

3、使用确切类型来替代通配符类型的好处是：可以用泛型参数来做更多的事，但是使用通配符使得必须接受范围更宽的参数化类型作为参数

4、`<未完成>`

#### 15.10.4. 捕获转换

1、有一种情况特别需要使用`<?>`而不是原生类型，如果向一个使用`<?>`方法传递原生类型，那么对编译器来说，可能会推断出实际的类型参数，使得这个方法可以回转并调用另一个使用这个确切类型的方法(推断出具体类型并传入一个需要具体类型的方法)，这称为**捕获转换**

2、具体含义看例子 P399

3、捕获转换十分有趣，但非常受限

### 15.11. 问题

#### 15.11.1. 任何基本类型都不能作为类型参数

1、即不能声明 `ArrayList<int>`之类的东西

2、自动包装机制将自动实现 `int` 到 `Integer` 的双向转换

3、自动包装机制无法应用于数组，即 `Integer[]` 与 `int[]` 不会发生自动转换

#### 15.11.2. 实现参数化接口

1、一个类不能实现同一个泛型接口的两种变体，由于擦除的原因，这两个变体会成为相同的接口

#### 15.11.3. 转型和接口

1、使用带有泛型类型参数的转型或 `instanceof` 不会有任何效果

➤ 即在方法内部调用`(T)`或者 `obj instanceof T` 没有任何效果，因为 `T` 被擦除到了 `Object`

#### 15.11.4. 重载

1、当被擦除的参数不能产生唯一的参数列表时，必须提供明显有区别的方法名

2、因为擦除的原因，不能以泛型参数来区别方法

#### 15.11.5. 基类劫持了接口

1、`<未完成>`: 没懂

2、大概意思就是：

➤ 一个基类的 `ComparablePet` 实现了 `Comparable<ComparablePet>`

```
public class ComparablePet implements Comparable<ComparablePet>{...}
```

➤ 该基类的任何导出类，例如 `Cat` 无法再次实现 `Comparable<Cat>`

➤ 如果必须要实现，那么必须指定泛型参数为 `ComparablePet`，即与基类指定的参数一样

```
public class Cat implements Comparable<ComparablePet>{...}
```

## 15.12. 自限定类型

1、在 Java 中，有一个经常出现但是令人费解的惯用法

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

- 该语法强调的是：`extends` 关键字用于边界与用来创建子类明显是不同的

### 15.12.1. 古怪的循环泛型

1、简单的版本

```
class GenericType<T> {}
class CuriouslyRecurringGeneric extends GenericType<CuriouslyRecurringGeneric> {}
```

2、用 Jim Coplien 在 C++ 中的古怪的循环模板模式的命名方式，称为古怪的循环泛型(CRG)

3、CRG 的含义：

- 我在创建一个新类，它继承自一个泛型，这个泛型接受我的类名作为其泛型参数

4、CRG 的好处：

- 当给出导出类的名字时，这个泛型基类能够实现什么呢？
- **Java 中的泛型关乎参数和返回类型**
- **CRG 能够产生使用导出类作为其参数和返回类型的基类**
- **CRG 还能将导出类型用作其域类型，甚至那些将被擦除为 Object 的类型**

5、例子

```
class BasicHolder<T> {
 T element;
 void set(T arg) {element=arg;}
 T get() {return element;}
}
class Subtype extends BasicHolder<Subtype>{}
➤ 新类 Subtype 接受的参数和返回的值具有 Subtype 类型而不仅仅是基类 BasicHolder 类型
➤ CRG 本质：基类用导出类替代其参数
```

### 15.12.2. 自限定

1、自限定将采取额外的步骤，强制泛型当做其自己的边界参数来使用

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

- `SelfBounded` 的泛型参数 `T` 必须是一个自限定类型

```
class A extends SelfBounded<A>;//OK, A 是一个自限定类型
```

```
class B extends SelfBounded<A>;//OK, 普通继承而已，且 A 是一个自限定
```

```
class C extends SelfBounded;//False, B 不是自限定类型
```

2、自限定只会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
- 强制要求使用这个类的每个人都要遵循这种模式

### 15.12.3. 参数协变

1、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

- (当该函数没有接受基类类型的重载版本时)并且不可以传递基类类型，因为没有任何

方法具有这样的签名

- 2、自限定泛型将产生确切的导出类型作为其返回值
- 3、可以重载一个接受导出类类型的版本，调用时会选择该重载版本，而不会选择继承自基类并且协变了参数类型的版本

## 15.13. 动态类型安全

- 1、`java.util.Collections` 中有一组便利工具，可以解决类型检查问题，**返回一个相应的类型**

- `public static <E> Collection<E> checkedCollection(Collection<E> c, Class<E> type)`
- `public static <E> List<E> checkedList(List<E> list, Class<E> type)`
- `public static <K, V> Map<K, V> checkedMap(Map<K, V> m, Class<K> keyType, Class<V> valueType)`
- `public static <E> Set<E> checkedSet(Set<E> s, Class<E> type)`

- 2、受检查的容器在你试图插入类型不正确的对象时抛出 **ClassCastException**

- 这与泛型之前的(原生容器)形成了对比
- 对于后者，当你从对象从容器中取出时，才会通知你出现了问题，但无法知道问题出现在哪里
- 而受检查的容器将在你插入时就进行检查

## 15.14. 异常

- 1、由于擦除的原因，将泛型应用于异常时非常受限的

- `catch` 语句不能捕获泛型类的异常，**因为在编译期和运行时都必须知道异常的确切类型**
- 泛型类也不能直接或间接继承自 `Throwable`，这将进一步阻止你去定义不能捕获的泛型异常

- 2、类型参数可能会在一个方法的 `throws` 子句中用到，这使得你可以编写随检查型异常的类型而发生变化的泛型代码???

## 15.15. 混型

- 1、属于混型随时间的推移有了很多含义，但其最基本的概念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(**有点类似多重继承的意思**)

- 2、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，如果想在混型中修改某些东西，作为一种意外的好处，这些修改将会应用于混型所引用的所有类型之上

### 15.15.1. C++中的混型

- 1、在 C++中使用多重继承的最大理由，就是为了使用混型，**混形是基于继承的**
- 2、对于混型来说，更有趣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
- 3、在 C++中很容易创建混型，因为 C++能够记住其模板参数的类型

```
template<typename T> class A:public T{};
template<typename T> class B:public T{};
class C{}
```

A<B<C>> a;//于是 a 就拥有 ABC 中所有的功能(B 继承自 C, A 继承自 B)

4、遗憾的是，Java 的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

1、一种更常见的推荐解决方案是使用接口来产生混型效果

### 15.15.3. 使用装饰器模式

1、混型的概念**好像**与装饰器设计模式关系很近

- 装饰器经常用于满足各种可能的组合，而直接子类化会产生过多的类，因此是不实际的

2、装饰器模式

- **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可装饰对象的四周，来将功能分层(思考 InputStream, BufferedInputStream, DataInputStream)，这使得对装饰器的使用时透明的---无论对象是否被装饰，你都拥有一个可以向对象发送的公共消息集**
- **装饰器是通过使用组合和形式化接口(可装饰物/装饰器层次结构)来实现的，而混型是基于继承的，因此可以将基于参数化类型的混型当做是一种泛型装饰器机制，这种机制不需要装饰器设计模式的继承结构**

3、使用装饰器所产生的对象类型是最后被装饰的类型，例如

```
DataInputStream in=new DataInputStream(
 new BufferedInputStream(
 new FileInputStream("<path>")));
```

- 即最终产生的对象类型是 DataInputStream，而不是被包装的 FileInputStream

4、**装饰器尽管可以添加多个层，但是最后一层(最外面)才是实际的类型，因此只有最后一层的方法是可视的，而混型的类型是所有被混合到一起的类型，因此对于装饰器来说，其明显的缺陷就是它只能有效地工作于装饰中的一层(最后一层(最外面))，因此装饰器只是对由混型提出的问题的一种局限的解决方案(并不是装饰器模式不好，而是对于解决混型这个问题，装饰器的方案并不那么优秀)**

### 15.15.4. 与动态代理结合

1、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

2、由于动态代理的限制，每个被混入的类都必须是某个接口的实现

3、**因为只有动态类型而不是静态类型才包含所有的混入类型**，因此这仍不如 C++ 的方式好，因为可以在具有这些类型的对象上调用方法之前，你被强制要求必须先将这些类型向下转型到切当的类型，但是它明显更接近于正真的混型

```
class MixinProxy implements InvocationHandler{
 Map<String, Object> delegatesByMethod;//区别与一般的动态代理，这里用一个
 Map 来组织所有的被代理的对象，同时代理了很多对象，并且需要根据方法名来
 关联一个对象，方法名用 String 来存储
 public MixinProxy(TwoTuple<Object, Class<?>>...pairs){
 delegatesByMethod=new HashMap<String, Object>();
```

```

 for(TwoTuple<Object,Class<?>>pair:pairs){
 for(Method method:pair.second.getMethods()){
 String methodName=method.getName();
 if(!delegatesByMethod.containsKey(methodName))
 delegatesByMethod.put(methodName, pair.first);//将"方法名-对象"存储到 Map 中
 }
 }
 }

 public Object invoke(Object proxy,Method method,Object[] args) throws Throwable{
 String methodName=method.getName();
 Object delegate=delegatesByMethod.get(methodName);//根据方法名找到对应的对象
 return method.invoke(delegate, args);
 }

 @SuppressWarnings("unchecked")
//该方法用于创建一个动态代理对象
 public static Object newInstance(TwoTuple...pairs){
 Class[] interfaces=new Class[pairs.length];//代理需要实现的接口列表
 for(int i=0;i<pairs.length;i++){
 interfaces[i]=(Class)pairs[i].second;
 }
 ClassLoader cl=pairs[2].first.getClass().getClassLoader();//类加载器
 MixinProxy mixinProxy=new MixinProxy(pairs);//实现了 InvocationHandler 的类的对象
 return Proxy.newProxyInstance(cl, interfaces, mixinProxy);//
 }
}

public class DynamicProxyMixin {
 public static void main(String[] args){
 Object mixin=MixinProxy.newInstance(
 Tuple.tuple(new BasicImpl(),Basic.class),
 Tuple.tuple(new TimeStampedImpl(),TimeStamped.class),
 Tuple.tuple(new SerialNumberedImpl(),SerialNumbered.class));
 Basic b=(Basic)mixin;//需要向下转型到恰当的类型后才能使用
 TimeStamped t=(TimeStamped)mixin;
 SerialNumbered s=(SerialNumbered)mixin;
 b.set("Hello");
 System.out.println(b.get());
 System.out.println(t.getStamp());
 System.out.println(s.getSerialNumber());
 }
}

```

## 15.16. 潜在的类型机制

1、要编写能够尽可能广泛地应用的代码，我们需要各种途径来放松对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

2、Java 泛型看起来向这一方面迈进了一步，当你在编写只是持有对象的泛型时，这些代码将可以作用于任何类型，“持有器”泛型可以声明：我不关心你是什么类型，如果代码不关心它将要作用的类型，那么这种代码就可以真正地应用于任何地方，并因此相当泛化

3、当要在泛型类型上执行操作时，就会产生问题

- 因为擦除要求指定可能会用到的泛型类型的边界，以安全地调用代码中的泛型对象中的具体方法
- 这是对泛化概念的一种限制，因为必须限制你的泛型类型，使它们继承自特定的类，或者实现特定的接口
- 在某些情况下，你可能会使用普通的类或普通接口，因为界定边界的泛型可能会和指定类或接口没有任何区别

4、某些变成语言提供一种解决方案称为潜在类型机制或结构化类型机制，更古怪的术语称为鸭子类型机制

- 即“如果它走起来像鸭子，并且叫起来也像鸭子”，那么你就可以将它当做鸭子对待
- 具有潜在类型机制的语言只要求实现某个方法子集，而不是某个类或者接口，从而放松了这种限制(并且可以产生更加泛化的代码)
- 潜在类型机制使得你可以横跨类继承结构，调用不属于某个公共接口的方法
- 一段代码可以声明：我不关心你是什么类型，只要你可以 speak() 和 sit() 即可
- 两种支持潜在类型机制的语言实例是 Python 和 C++，Python 是动态类型语言，而 C++ 是静态类型语言

```
template<typename T> void perform(T anything){
 anything.speak();
 anything.sit();
}
```

```
def perform(anything):
 anything.speak()
 anything.sit()
```

- C++ 至少需要模板才能实现潜在类型机制，而 Python 完全不用
- C++ 确保了它实际上可以发送的那些消息，如果试图传递错误的类型，编译器就会给一个错误消息(这些错误消息从历史上看是相当可怕和冗长的，而主要原因是 C++ 模板名声欠佳)
- C++ 在编译期，Python 在运行时，两种语言都可以确保类型不会被误用，因此被认为是强类型的

5、泛型是 Java 在后期才添加的，因此没有任何机会去实现任何类型的潜在类型机制，因此 Java 没有对这种特性的支持

## 15.17. 对缺乏潜在类型机制的补偿

1、尽管 Java 不支持潜在类型机制，但这并不意味着有界泛型代码不能在不同的类型层次结构之间应用，也就是说，我们可以创建真正的泛型代码，但是这需要付出一点额外的努力

### 15.17.1. 反射

1、通过反射，我们能够动态地确定所需要的方法是否可用并调用它们

```
public static void perform(Object speaker){
 Class<?>spkr=speaker.getClass();
 try{
 try{
 Method speak=spkr.getMethod("speak");
 speak.invoke(speaker);
 }catch(NoSuchMethodException e){
 print(speaker+" cannot speak");
 }
 try{
 Method sit=spkr.getMethod("sit");
 sit.invoke(speaker);
 }catch(NoSuchMethodException e){
 print(speaker+" cannot sit");
 }
 }catch(Exception e){
 throw new RuntimeException(speaker.toString(),e);
 }
}
```

### 15.17.2. 将一个方法应用于序列

1、反射提供了一些有趣的可能性，但是它将所有的类型检查都转移到了运行时，因此在许多情况下并不是我们所希望的，如果能够实现编译期类型检查，这通常会更符合要求，但是 Java 能实现**编译期类型检查**和**潜在类型机制**吗

2、假设要创建一个 `apply()` 方法，它能够将任何方法应用于某个序列中的所有对象

- 这是接口看起来并不适合的情况，因为想要将任何方法应用于某个序列中的所有对象，这是接口看起来并不适合的情况，因为接口对于描述“任何方法”存在过多的限制
- 我的见解：
- **接口必须确定一个方法名，这与任何方法产生了矛盾**
- 对于参数列表的任意性：接口用可变参数 `Object...args` 来作为参数以产生一种通用的形式
- 对于返回类型的任意性：可用 `Object` 作为返回类型吗

3、我们可以用反射来解决这个问题，由于有了 Java SE5 的可变参数，这种方式很优雅

```
public static<T, S extends Iterable<? extends T>>
 void apply(S seq,Method f, Object...args){
 try{
 for(T t:seq){
 f.invoke(t, args); //重定向为对序列中每个对象调用该方法
 }
 }catch(Exception e){
```

```
 throw new RuntimeException(e);
 }
}
```

#### 4、本节想要说明什么？

- Apply 可以保证任何调用该方法的序列必须实现了 Iterable 接口，否则将会在编译期报错？

5、尽管该方案比较优雅，**但是使用反射可能比非反射的实现要慢一些**(尽管现在已经明显改善了)，因为有太多的动作都是在运行时发生的，**但这不该阻止你使用这种方案，至少这是一种马上能想到的方案**

### 15.17.3. 当你并未碰巧拥有正确的接口时

- 1、上一节 15.17.2，因为 Iterable 接口已经是内建的，而它正是我们需要的
- 2、例如我们需要创建一个参数化的方法 fill()，它接受一个序列，并使用 Generator 填充它
  - 当我们尝试用 Java 来编写时，就会陷入问题之中，因为没有任何像前面示例中的 Iterable 接口那样的"Addable"便利接口，因此你不能说：能在任何事物上调用 add()，而必须说可以在 Collection 的子类型上面调用 add()
  - 本节要实现的方法 fill()与上一节要实现的方法 apply()的区别
    - apply(): 接受一个序列，对序列中每个对象，调用指定的方法，**限制在于是否可用 foreach 遍历这个序列，即需要 Iterable 接口(而序列都是实现了 Iterable 接口的，因此说碰巧拥有了这个接口)**
    - fill(): 接受一个序列，并填充这个序列，即产生新对象填充进去，**限制在于是否可用 add()方法来添加对象实例，即需要 Collection 接口(而序列并非都实现了 Collection 接口，例如 Map)**
    - 那为什么不用反射判断一下是否有 add 方法???

```
public static<T>void fill(Collection<T> collection,Class<? extends T>classToken,int size){
 for(int i=0;i<size;i++)
 try{
 collection.add(classToken.newInstance());//为什么这里不用反射呢?
 }catch(Exception e){
 throw new RuntimeException(e);
 }
}
```

3、这正是潜在类型机制的参数化类型机制的价值所在，因为你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，因此不需要再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(这样的代码才是真正泛化的)

- 在上面的例子中，Java 设计者没有预见到对"Addable"接口的需要，**所以我们被限制在 Collection 继承层次结构之内**
- 即便有一个 add()方法，只要没有实现 Collection 接口，上述代码也不能工作

### 15.17.4. 用适配器仿真潜在类型机制

- 1、Java 泛型不是没有潜在类型机制，而我们需要像潜在类型机制这样的东西去编写能够跨类边界应用的代码(也就是"泛型"代码)，存在某种方式可以绕过这项限制吗
- 2、潜在类型机制将在这里实现什么？
  - 它意味着你可以编写代码声明：**我不关心我在这里使用的类型，只要它具有这些方法**

法即可

- 事实上，潜在类型机制创建了一个包含所需方法的隐式接口
- 它遵循这样的规则：如果我们手工编写了必须的接口(因为 Java 并没有为我们做这些事)，那么它就应该能够解决问题

### 3、解决方案

```
public static<T> void fill(Addable<T> addable,Class<? extends T> classToken,int size){
 for(int i=0;i<size;i++){
 try{
 addable.add(classToken.newInstance());
 }catch(Exception e){
 throw new RuntimeException(e);
 }
 }
}

public static <T> void fill(Addable<T> addable,Generator<T>generator,int size){
 for(int i=0;i<size;i++)
 addable.add(generator.next());
}
```

- 上述代码的要求与 15.17.3 的 fill() 方法要求不同
  - 它只需要实现了 Addable 接口的对象，而这个 Addable 接口正是我们为实现该方法而编写的接口
  - 第二个重载版本接受了一个 Generator 对象而不是 Class 类型的对象(标记)，Generator 在编译期是安全的，编译器将确保传递的是正确的 Generator
  - 为什么传递 Class<? extends T> classToken 是不安全的???我试着传入了错误的类型，在编译器就给我错误了啊???

```
class AddableCollectionAdapter<T> implements Addable<T>{
 private Collection<T> c;
 public AddableCollectionAdapter(Collection<T> c){
 this.c=c;
 }
 public void add(T item){c.add(item);}
}
```

- 上述适配器可以工作于基类型 Collection
  - 这意味着任何 Collection 的实现都能用
  - 这个版本直接存储 Collection 引用，并使用它来实现 add()

```
class AddableSimpleQueue<T> extends SimpleQueue<T> implements Addable<T>{
 public void add(T item){super.add(item);}
}
```

- 上述适配器工作于自定义的序列类型 SimpleQueue

## 15.18. 将函数对象用作策略

### 1、策略设计模式

- 这种模式可以产生更优雅的代码，因为它将“变化的事物”完全隔离到一个函数对象

中

- 函数对象就是在某种程度上行为像函数的对象，一般会有一个相关的方法(在支持操作符重载的语言中，可以创建对这个方法的调用，而这个调用看起来就和普通的方法调用一样，例如 C++ 重载函数调用运算符)
- 函数对象的概念是否是对 Java 没有函数指针的一种补偿???
- 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
- 当然，可以用类中任何方法来实现与此相似的操作，但是函数对象的主要是由其目的来区别的，这里的目地就是要创建某种事物，使它的行为就像是一个可以传递出去的单个方法一样

2、根据目前的理解，函数对象就是传入的一个对象，其主要目的是提供相应的方法

3、书上例子分析

- 定义了 4 个接口
  - Combiner: 将两个对象添加在一起
  - UnaryFunction: 接受一个参数，并产生一个结果，参数和结果可以是两个类型
  - Collector: 收集参数，当完成时，可以获得结果
  - UnaryPredicate: 产生一个 bool 类型的结果
- 泛型方法
  - 分别接受一个序列对象以及上述四种接口的一个对象
- 大量的静态内部类
  - 各个接口的不同实现，用于传入上面的泛型方法，作为函数对象

## 15.19. 总结：转型真的如此之糟吗？

1、使用泛型类机制的最吸引人的地方，就是在使用容器类的地方

- 在 Java SE5 前，你将一个对象放置到容器中，这个对象就会被向上转型为 Object，因此你会丢失类型信息；当你想要将这个对象从容器中取回，用它执行某些操作，必须向下转型回正确的类型
- 泛型出现之前的 Java 并不会让你误用放入到容器中的对象，而是在转型失败时会抛出一个 RuntimeException()，即在运行时而非编译时发现

2、被称为泛型的通用语言特性的目的在于可表达性，而非创建类型安全的容器

- 作者认为“狗在猫列表中”的事情不会经常发生，即便发生也不会造成很严重的后果

3、在一种语言已经被广泛应用之后，在其较新的版本中引入任何种类的泛型机制，都会是一件非常棘手的任务

## Chapter 16. 数组

### 16.1. 数组为什么特殊

- 1、数组与其他种类的容器之间的区别有三方面：**效率**、**类型**和**保存基本类型**的能力
  - 在 Java 中，**数组是一种效率最高的存储和随机访问对象引用序列的方式**
  - 数组就是一个简单的线性序列，这使得元素访问非常快速，为这种速度所付出的代价就是数组对象的大小被固定，并且在其声明周期不可改变
  - 通常首选是 `ArrayList` 而不是数组，但这种弹性(自动扩张和收缩)需要开销，因此 **ArrayList 的效率比数组低很多**
- 2、数组和容器都可以保证你不能滥用它们，无论使用数组还是容器，如果越界都会得到一个表示程序员错误的 `RuntimeException` 异常
- 3、在泛型之前，其他的容器类在处理对象时，都将它们视作没有任何具体类型，也就是说，它们将这些对象都当做 Java 中所有类的根类 `Object` 处理，**数组之所以优于泛型之前的容器，就是可以创建一个数组去持有某种具体类型**，这意味着你可以通过编译期检查，来防止插入错误类型或抽取不当类型
  - 无论在编译时还是运行时，Java 都会阻止你向对象发送不恰当的消息，只是如果编译时就能指出错误，会显得更加优雅，也减少了程序使用者被异常吓着的可能性
- 4、数组可以持有基本类型，而泛型之前的容器不能，但是有了泛型，容器就可以指定并检查它们所持有对象的类型，并且有了自动包装机制
- 5、数组与容器唯一明显的差异就是数组使用`[]`来访问元素，而 `List` 使用 `add()` 和 `get()` 这样的方法
- 6、随着自动包装机制的出现，容器已经可以与数组几乎一样方便地用于基本类型了，数组硕果仅存的优点就是效率，如果要解决一般化的问题，容器还是首选

### 16.2. 数组是第一级对象

- 1、无论使用哪种类型的数组，数组标识符其实就是一个引用，指向在堆中创建的一个真实对象，这个对象用以保存指向其他对象的引用
- 2、只读成员 `length` 是数组对象的一部分，这是唯一一个可访问的字段或方法，`[]` 语法是访问数组对象的唯一方式
- 3、对象数组和基本类型数组在使用上几乎是相同的，唯一的区别就是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

### 16.3. 返回一个数组

- 1、返回一个数组对于 C 和 C++ 这样的语言来说有点困难，因为它们不能返回一个数组，而只能返回指向数组的指针，这会造成一些问题，因为它使得控制数组的声明周期变得很困难，并且容易造成内存泄露
- 2、在 Java 中，你可以返回一个数组(`new` 在堆上创建)，而且无需担心要为数组负责(垃圾收集器)---只要你需要它，它就会一直存在，当你使用完后，垃圾收集器在适当时候会清理掉它

## 16.4. 多维数组

- 1、创建多维数组很方便，对于基本类型的多维数组，可以通过使用花括号将每个向量分开
- 2、**Arrays.deepToString()**方法：可以将多维数组转换为多个 String
- 3、数组中构成矩阵的每个向量都可以具有任意长度，这被称为**粗糙数组**

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

- 很好理解，因为多维数组的第一维存放的是数组的引用，并没有限制这些引用之间的长度约束关系
- 如果用 new 创建数组时指定长度，也可以为第一维的元素赋值为非指定大小的数组的引用

```
int[][] ary=new int[1][2];
ary[0]=new int[]{1,2,3,4,5};
```

- 4、多维数组创建指定的大小，只有第一维是有效的，**在非花括号形式的初始化，只有第一维的大小是必须的**

```
int[][] ary=new int[1][];
```

## 16.5. 数组与泛型

- 1、**通常泛型与数组不能很好地结合，不能实例化具有参数化类型的数组**
- 2、**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**
- 3、编译器会继续创建具有参数化类型的引用，即：List<String>[] ls;
  - 但是无法实例化具有参数化的类型，即 ls 只能这样初始化  
`ls=new List[10]; //书上的例子是"ls=(List<String>) new List[10];"`，但是这两者都会提供类型检查，那强制类型转换的意义是什么??
  - 一旦拥有了对 List<String>[] 的引用，你就会看到你将得到某些编译期检查，即无法插入持有 String 之外类型的 List，即 `ls[0]=new ArrayList<Integer>();` 是非法的
  - 但是数组是协变类型的，因此 List<String>[] 也是一个 Object[]  
`Object[] objects=ls;`  
`objects[1]=new ArrayList<Integer>(); //居然就可以插入一个持有 Integer 的 List`
- 4、一般而言，泛型在类或方法的**边界处(进入或离开)**很有效，而在类或方法的内部，擦除通常使泛型变得不适用

## 16.6. 创建测试数据

### 16.6.1. Arrays.fill()

- 1、Java 标准库 Arrays 有一个作用十分有限的 fill() 方法：只能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(指向同一个实例)**

### 16.6.2. 数据生成器

- 1、<未完成>：在扯作者自己的 Generator 了

## 16.7. Arrays 实用功能

1、在 `java.util` 类库中可以找到 `Arrays` 类，它有一套用于数组的 `static` 实用方法，其中有六个基本方法：

- `Arrays.equals()` 用于比较两个数组是否相等 (`deepEquals()` 用于比较多维数组)
- `Arrays.fill()` 用于填充数组
- `Arrays.sort()` 用于对数组排序
- `Arrays.binarySearch()` 用于在已经排序的数组中查找元素 (未排序的话结果未知哦)
- `Arrays.toString()` 用于产生数组的 `String` 表示
- `Arrays.hashCode()` 用于产生数组的散列码
- `Arrays.asList()` 接受任意的序列或数组作为其参数，将其转变为 `List` 容器，只是这个容器是一个内部类，无法改变大小哦，一般是将 `Arrays.asList()` 的结果作为一个容器构造器的输入参数

### 16.7.1. 复制数组

1、**Java 标准类库提供有 `static` 方法 `System.arraycopy()`**，用它复制数组比用 `for` 循环复制要快得多，`System.arraycopy()` 针对所有基本类型与 `Object` 做了重载

2、`System.arraycopy()` 的参数

- 源数组
- 源数组起始偏移量(就是下标)
- 目标数组
- 目标数组起始偏移量
- 要复制的元素个数
- `System.arraycopy(i,0,j,0,i.length);`

3、**如果复制对象数组，那么只是复制了对象的引用---而不是对象本身的拷贝，这被称为浅拷贝**

4、**`System.arraycopy()` 不会执行自动包装和自动拆包，两个数组必须具有相同的确切类型**

### 16.7.2. 数组的比较

1、`Arrays` 类提供了重载后的 `equals()` 方法，用来比较整个数组，此方法同样对所有基本类型与 `Object` 都做了重载

2、相等的条件

- 元素个数必须相等，并且对应位置的元素也相等
- 元素是否相等：通过每一个元素使用 `equals()` 作比较来判断，对于基本类型，会采用其包装类型的 `equals` 来比较

### 16.7.3. 数组元素的比较

1、排序必须根据对象的实际类型执行比较操作，一种自然的解决方案是为每种不同的类型编写一个不同的排序方法，但是这样的代码难以被新的类型复用

2、程序设计的基本目标是“将保持不变的事物与会发生改变的事物相分离”，不变的是通用的排序算法，变化的是各种对象相互比较的方式 15.18

- 因此，不是将进行比较的代码编写成不同的子程序，而是使用策略设计模式(函数对象，或策略对象)

- 通过使用策略, 可以将发生变化的代码封装在单独的类中(策略对象)
  - 你可以将策略对象传递给总是相同的代码, 这些代码将使用策略来完成其算法
- 3、Java 有两种方式来提供比较功能
- 第一种是实现 `java.lang.Comparable` 接口, 使你的类天生具有比较能力, 此接口很简单, 只有 `CompareTo()`一个方法, 此方法接受另一个 T 类型的参数:
    - 如果当前对象小于参数则返回负值
    - 如果相等返回零
    - 如果当前对象大于参数返回正值
    - 即要保持当前的顺序, 利用可用的资源返回负数, 要调转顺序, 返回正数
  - 若没有实现 `java.lang.Comparable` 接口, 或者实现了该接口, 但你并不喜欢它的方式(比较的含义与你想要的含义不同)
    - 可以创建一个实现了 `Comparator` 接口的单独的类
    - 这个类有 `compare()`和 `equals()`两个方法, 其中 `equals()`方法并不强制要求实现, 因为创建一个类, 都会继承自 `Object`, 而 `Object` 带有 `equals()`方法
    - 同样, 要保持当前的顺序(第一个参数在前, 第二个参数在后), 就利用可用资源返回负数即可, 若要调转顺序, 则返回正数

#### 16.7.4. 数组排序

- 1、使用内置的排序方法, 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`
- 2、Java 标准库中的排序算法针对正排序的特殊类型进行了优化---**针对基本类型设计的"快速排序", 以及针对对象设计的"稳定归并排序"**, 所以无需担心性能

#### 16.7.5. 在已排序数组中查找

- 1、如果数组已经排好序, 就可以使用 `Arrays.binarySearch()`执行快速查找
- 2、如果对未排序的数组使用 `binarySearch()`将产生不可预料的结果

### 16.8. 总结

- 1、Java 对尺寸固定的低级数组提供了适度的支持, 这种数组强调的是性能而不是灵活性, 并且与 C 和 C++ 的数组模型类似
- 2、在 Java 初始版本中, 尺寸固定的低级数组绝对是必须的, 不仅因为 Java 设计者选择在 Java 中要包含基本类型, 而且那个版本中对容器的支持非常少
- 3、其后的 Java 版本对容器的支持都得到了明显的改进, 并且现在的容器在除了性能之外的各个方面都使得数组相形见绌
- 4、有了额外的自动包装机制和泛型, 在容器中持有基本类型就变得易如反掌了, 这也促使你使用容器来替换数组, 因为泛型可以产生类型安全的容器, 因此数组面对这一变化, 已经变得毫无优势(早期, 数组对于容器的优势在于类型安全)
- 5、当你在使用最近的 Java 版本编程时, 应该"优选容器而不是数组"

## Chapter 17. 容器深入研究

### 17.1. 完整的容器分类法

1、Java SE5 添加了：

- Queue 接口
- ConcurrentHashMap 接口及其实现 ConcurrentHashMap
- CopyOnWriteArrayList 和 CopyOnWriteHashSet
- EnumSet 和 EnumMap

### 17.2. 填充容器

1、与 Arrays 一样，相应的 Collections 类也有一些实用的 static 方法

- Collections.fill(): 复制同一个对象来填充整个容器(所有引用指向同一个对象)，而且只对 List 对象有用

#### 17.2.1. 一种 Generator 解决方案

1、<未完成>

#### 17.2.2. Map 生成器

1、<未完成>

#### 17.2.3. 使用 Abstract 类

1、设计模式：享元

- 你可以在普通的解决方案需要过多的对象，或者产生普通对象太占用空间时使用享元
- 享元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，我们可以在更加高效的外部表中查找对象的一部分或整体

2、<未完成>：不懂

### 17.3. Collection 的功能方法

1、基本方法

- boolean add(T): 确保容器持有具有泛型 T 的参数(**可选方法**)
- boolean addAll(Collection<? extends T>): 添加参数中的所有元素，只要添加了任意元素就返回 true(**可选方法**)
- void clear(): 移除容器中所有元素(**可选方法**)
- boolean contains(T): 如果容器已经持有具有泛型类型 T 的参数，返回 true
- Boolean containsAll(Collection<?>): 如果容器持有此参数中的所有元素，返回 true
- boolean isEmpty(): 容器中没有元素，返回 true
- Iterator<T> iterator(): 返回一个 Iterator<T>，用于遍历容器
- Boolean remove(Object): 如果参数在容器中，移除此元素的一个实例，如果做了移除动作，则返回 true(**可选方法**)
- boolean removeAll(Collection<?>): 移除参数中的所有元素，只要有移除动作发生，

就返回 true(可选方法)

- Boolean retainAll(Collection<?>): 只保存参数中的元素(交集), 只要 Collection 发生了改变, 就返回 true(可选方法)
- int size(): 返回容器中元素数目
- Object[] toArray(): 返回一个数组, 该数组包含容器中的所有元素
- <T> T[] toArray(T[] a): 返回一个数组, 该数组包含容器中的所有元素, 返回结果的运行时类型与参数数组 a 的类型相同, 而不是单纯的 Object[]

2、注意, Collection 不包括 get()方法, 因为 Collection 包括 Set, 而 Set 是自己维护内部顺序的, 这使得随机访问没有意义, 如果想检查 Collection 中的元素, 就必须使用迭代器

## 17.4. 可选操作

1、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

2、这是一种很不寻常的接口定义方式

- 接口是面向对象设计中的契约: 他声明"无论你选择如何实现该接口, 我保证你可以向该接口发送这些消息", 但是可选操作违反这个非常基本的原则
- 调用某些方法不会执行有意义的行为, 相反它们会抛出异常
- 将 Collection 当做参数接受的大部分方法只会从该 Collection 中读取, 而 Collection 的 **读取方法都不是可选的**

3、为什么要将方法定义为可选?

- 可以防止在设计中出现接口爆炸的情况
- 这种方式可以实现 Java 容器类库的重要目标: 容器应该易学易用

4、UnsupportedOperationException 必须是一种罕见事件, 即对于大多数类来说, 所有操作都应该可以工作, 只有在特例中才会有未获支持的操作

- 未获支持的操作只有在运行时才能探测到, 因为它们表示动态类型检查

### 17.4.1. 未获支持的操作

1、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

- 当你使用 Arrays.asList()将数组转换为 List, 就会得到这样的容器
- 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, 对它而言是有道理的, 对该 List 调用任何会引起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, 以表示对未获支持操作的调用(一个编程错误)
- 应该 Arrays.asList()的结果作为构造器的参数传递给任何 Collection(或者使用 addAll()方法, 或 Collections.addAll()静态方法), 这样可以生成允许使用所有的方法的普通容器
- **Collections.unmodifiableList()**产生不可修改的列表, 连 Arrays.asList()可以支持的 set()方法也不支持, **Collections.unmodifiableList()在任何情况下都是不可修改的**

## 17.5. List 的功能方法

1、最基本的方法

- List.add()

- List.get()
- List.set()

## 17.6. Set 和存储顺序

1、不同的 Set 实现不仅具有不同的行为，而且它们对于可以在特定的 Set 中放置的元素的类型也有不同的要求

- **Set(interface)**: 存入 Set 中的每个元素必须是唯一的，因为 Set 不保存重复元素。加入 Set 的元素必须定义 equals()方法以确保对象的唯一性，Set 和 Collection 拥有完全一样的接口，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**: 保持次序的 Set，底层为树结构，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(插入顺序)，于是在使用迭代器遍历 Set 时，结果会按元素插入的次序显示，元素也必须定义 hashCode()
- HashSet 打星号表示，如果没有特殊要求，这就是你的默认选择，因为它对速度进行了优化

### 17.6.1. SortedSet

1、SortedSet 中的元素可以保证处于排序状态，这使得它可以通过在 **SrotedSet 接口**中的下列方法提供附加的功能

- Comparator comparator()返回当前 Set 使用的 Comparator，或者返回 null
- Object first()返回容器中的第一个元素
- Object last()返回容器中的最后一个元素
- SortedSet subSet(fromElement,toElement)生成此 Set 的子集，范围从 fromElement(包含)到 toElement(不包含)
- SortedSet headSet(toElement)生成 Set 的子集，由小于 toElement 的元素组成
- SortedSet tailSet(fromElement)生成此 Set 的子集，由大于或等于 fromElement 的元素组成

## 17.7. 队列

1、除了并发应用，Queue 在 Java SE5 中仅有的两个实现是 LinkedList 和 PriorityQueue，它们的差异在于排序行为而不是性能

### 17.7.1. 优先级队列

1、优先队列中的元素必须实现 Comparable 接口(定义 compareTo 方法)或者提供一个 Comparator 函数对象

### 17.7.2. 双向队列

1、双向队列(双端队列)就像是一个队列，但你可以在任何一端添加或移除元素  
2、在 LinkedList 中包含支持双向队列的方法，但是在标准库中没有任何显式的用于双向队列的接口

3、双向队列并不像 Queue 那样常用，因为你不太可能在两端都放入元素并抽取它们

## 17.8. 理解 Map

1、映射表(关联数组)的基本思想是它维护的是键-值对，因此你可以使用键来查找值

2、标准的 Java 类库中包含了 Map 的几种基本实现，包括

- **HashMap**
- **TreeMap**
- **LinkedHashMap**
- **WeakHashMap**
- **ConcurrentHashMap**
- **IdentityHashMap**
- 它们都有相同的基本接口，但是行为特征各不相同，这表现在效率、键值对的保存及呈现次序、对象的保存周期、映射表如何在多线程程序中工作和判定“键”等价的策略等方面

### 17.8.1. 性能

1、性能是映射表中的一个重要问题，当在 `get()` 中使用线性搜索时，执行速度会相当地慢，而这正是 **HashMap** 提高速度的地方，**HashMap** 使用了特殊的值，称作散列码，来取代键的缓慢搜索

2、散列码是相对唯一的，用以代表对象的 `int` 值，它是通过将该对象的某些信息进行转换为生成的，`hashCode()` 是根类 `Object` 中的方法，因此所有 Java 对象都能产生散列码

3、**HashMap** 就是使用对象的 `hashCode()` 进行快速查询的，此方法能显著提高性能

4、Map 的基本实现

- **HashMap\***: 取代了 `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`，用迭代器遍历时，取得“键值对”的顺序是其插入的顺序，或者是最近最少使用(LRU)的次序
- **TreeMap**: 基于红黑树的实现，查看“键”和“键值对”时，它们会被排序(次序由 `Comparable` 或 `Comparator` 决定)，**TreeMap** 是唯一一个带有 `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**: 线程安全的 `Map`，它不涉及同步加锁
- **IdentityHashMap**: 使用 `==` 替代 `equals()` 对“键”进行比较的散列映射，专门解决特殊问题而设计的
- \*代表没有特殊要求时的默认选择

### 17.8.2. SortedMap

1、使用 `SortedMap(interface)`，可以确保键处于排序状态，这使得它具有额外的功能，这些功能由 `SortedMap` 接口中的下列方法提供

- `Comparator comparator()`: 返回当前 `Map` 使用的 `Comparator`；或者返回 `null`，表示以自然方式排序

- T `firstKey()`返回 Map 中第一个键
- T `lastKey()`返回 Map 中最末一个键
- `SortedMap subMap(fromKey, toKey)`生成此 Map 的子集，范围由 `fromKey(包含)`到 `to key(不包含)`的键确定
- `SortedMap headMap(toKey)`生成此 Map 的子集，由键小于 `toKey` 的所有键值对组成
- `SortedMap tailMap(fromKey)`生成此 Map 的子集，由键大于或等于 `fromKey` 的所有键值对组成

### 17.8.3. LinkedHashMap

- 1、为了提高速度，`LinkedHashMap` 散列化所有的元素，但是在遍历键值对时，却又以元素的插入顺序返回键值对
- 2、此外，可以在构造器中设定 `LinkedHashMap`，使之采用基于访问的最近最少使用(LRU)算法，于是没有被访问过的元素就会出现在队列的前面

## 17.9. 散列与散列码

1、**HashMap 使用 `equals()`判断当前的键是否与表中存在的键相同，而 `Object.equals()`只是比较对象的地址**

2、如果要使用自己的类作为 `HashMap` 的键，必须同时重载 `hashCode()` 和 `equals()`

3、正确的 `equals` 必须满足 5 个条件

- 自反性：对任意 `x`, `x.equals(x)`一定返回 `true`
- 对称性：对任意 `x` 和 `y`, 如果 `y.equals(x)` 返回 `true`, 那么 `x.equals(y)` 一定是 `true`
- 传递性：对任意 `x`、`y`、`z`, 如果有 `x.equals(y)` 返回 `true`, `y.equals(z)` 返回 `true`, 则 `x.equals(z)` 一定返回 `true`
- 一致性：对任意 `x` 和 `y`, 如果对象中用于等价比较的信息没有改变, 那么无论调用 `x.equals(y)` 多少次, 返回的结果应该保持一致, 要么一直是 `true`, 要么一直是 `false`
- 对任何不是 `null` 的 `x`, `x.equals(null)` 一定返回 `false`

### 17.9.1. 理解 `hashCode()`

1、散列的目的：想要使用一个对象来查找另一个对象

2、`Map.entrySet()`方法必须产生一个 `Map.Entry` 对象集，但 `Map.Entry` 是一个接口，用来描述依赖于实现的结构

### 17.9.2. 为速度而散列

1、散列的价值在于速度：散列使得查询得以快速进行

2、由于瓶颈位于键的查询速度，因此解决方案之一就是保持键的排序状态，然后使用 `Collections.binarySearch()` 进行查询

3、散列更进一步

- 将键保存在某处，以便能够很快找到。存储一组元素最快的数据结构是数组，所以使用它来表示键的信息(注意是键的信息，而非键本身)
- 因为数组不能调整容量，因此就有一个问题：我们希望在 `Map` 中保存数量不确定的值，但是如果键的数量被数组的容量限制了，该怎么办？
- 答案就是：数组并不保存键本身，而是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，由定义在 `Object` 中的，且可能由你的类覆盖的 `hashCode()` 方法返回

`de()`方法(在计算机科学的术语中称为散列函数)生成

4、为了解决数组容量被固定的问题，不同的键可以产生相同的下标，也就是说，可能会有冲突，因此，数组多大就不重要了，任何键总能在数组中找到它的位置，因此数组多大就不重要了，任何键总能在数组中找到它的位置

5、通常，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

### 17.9.3. 覆盖 `hashCode()`

1、首先，你无法控制 `bucket` 数组的下标值的产生，这个值依赖于具体的 `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，经过处理后成为桶的下标

2、设计 `hashCode()`时最重要的因素：

- 无论何时，对同一个对象调用 `hashCode()`都应该生成同样的值
- 此外，也不应该使 `hashCode()`依赖于具有唯一性的对象信息，尤其是使用 `this` 值，这只能产生很糟糕的 `hashCode()`，因为这样做无法生成一个新的键，使之与 `put()` 中原始键值对中的键相同
- `String` 有一个特点，如果程序中有多个 `String` 对象，都包含相同的字符串序列，那么这些 `String` 对象都映射到同一块内存区域，所以包含相同字符串序列的 `String` 的 `hashCode()` 是相同的(从 JVM 角度来看，`String` 对象都放在运行时常量池中，而非堆中)，所以使得 `String` 的 `hashCode()` 是基于内容的
- 对于基本类型的包装类型，其 `hashCode` 都是基于内容的(散列码一般与其值相同)

3、要想使得 `hashCode()`实用，它必须速度快，并且必须有意义，也就是说，它必须基于对象的内容生成散列码。散列码不必是独一无二的，应该关注其生成速度而非唯一性

4、另一个影响散列码性能的因素：好的 `hashCode()`应该产生分布均匀的散列码

## 17.10. 选择接口的不同实现

1、四种容器：`Map`、`List`、`Set` 和 `Queue`，每种接口都有不止一个实现版本

2、`Hashtable`、`Vector`、`Stack` 的特征是：过去遗留下来的类，目的只是为了支持老程序，最好不在新程序中使用它们

3、不同类型的 `Queue` 只在它们接受和产生数值的方式上有所差异

### 17.10.1. 性能测试框架

1、为了防止代码重复以及为了提供测试的一致性，将测试过程的基本功能放置到一个基本框架中

### 17.10.2. 对 `List` 的选择

1、`ArrayList`

- 无论列表大小如何，访问都很快速和一致
- 在中间插入时，随着列表增大，开销变得高昂(开辟新空间，然后拷贝过去)

2、`LinkedList`

- 访问时间对于较大的列表将明显增加，因此如果需要大量的随机访问，`LinkedList` 不是很好的选择，但是用 `getFirst`、`getLast` 或者 `get(0)` 或 `get(size()-1)` 也是很快的

### 17.10.3. 微基准测试的危险

1、`Math.random()`的范围是[0,1), 但似乎你不太可能验证能出现 0.0

### 17.10.4. 对 Set 的选择

- 1、`HashSet` 的性能总是比 `TreeSet` 好, 特别是在添加和查询元素时, 而这两个操作也是最重要的操作
- 2、`TreeSet` 存在的唯一原因是它可以维持元素的排序状态, 所以只要当需要一个排好序的 `Set` 时, 才应该使用 `TreeSet`
- 3、因为其内部结构支持排序, 并且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 `HashSet` 更快
- 4、对于插入操作, `LinkedHashSet` 比 `HashSet` 的代价更高, 这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

- 1、除了 `IdentityHashMap`, 所有 `Map` 实现的插入操作都会随着 `Map` 尺寸的变大而明显变慢, 但是查找的代价通常比插入要小得多
- 2、`TreeMap` 通常比 `HashMap` 要慢, 与 `TreeSet` 一样, `TreeMap` 是一种创建有序列表的方式。树的行为是: 总是保证有序, 并且不必进行特殊的排序, 一旦填充了整个 `TreeMap`, 就可以调用 `keySet()`方法来获取键的 `Set` 视图, 然后调用 `toArray()`来产生由这些键构成的数组
- 3、**使用 Map 时, 在没有特殊的需求时, 第一选择应该是 `HashMap`**
- 4、`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(以保持插入顺序), 正是由于这个链表, 使得其迭代速度更快

#### 5、`HashMap` 的性能因子

- 我们可以通过手工调整 `HashMap` 来提高其性能, 从而满足我们特定应用的需求
- **容量:** 表中桶位数(槽位)
- **初始容量:** 表在创建时所拥有的桶位数(`HashMap` 与 `HashSet` 都具有允许你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。** 空表的负载因子是 0, 半满表的负载因子是 0.5
  - 负载因子小的表产生冲突的可能性小, 因此对于插入和查找都是最理想的, 但是会减慢使用迭代器进行遍历的过程(大部分的桶位是空着的, 但是迭代时却会遍历这些空桶, 造成了时间的浪费)
  - `HashMap` 与 `HashSet` 都具有允许你指定负载因子的构造器, 表示当负载达到该负载因子水平时, 容器将自动增加其容量(桶位), 实现方式是使容量大致加倍, 并重新将现有对象分到新的桶位集中(称为再散列)
  - `HashMap` 使用的默认负载因子是 0.75, 这个因子在时间和空间代价之间达到了平衡, 更高的负载因子可以降低表所需的空间, 但是会增加查找代价(冲突多了, 相当于某些桶位的链表可能会很长, 而这些链表是现行查找的, 因此会增加查找的代价), 这很重要, 因为查找是我们在大多数时间里所作的操作

## 17.11. 实用方法

- 1、Java 中有大量用于容器的卓越的使用方法, 它们被表示为 `java.util.Collections` 类内部的静态方法

2、产生 Collection 或者 Collection 具体子类型的动态类型安全试图

- checkedCollection(Collection<T>, Class<T> type)
- checkedList(List<T>, Class<T> type)
- checkedMap(Map<K,V>, Class<K> keyType, Class<V> valueType)
- checkedSet(Set<T>, Class<T> type)
- checkedSortedMap(SortedMap<K,V>, Class<K> keyType, Class<V> valueType)
- checkedSortedSet(SortedSet<T>, Class<T> type)

3、返回参数 Collection 中最大或最小的元素--采用 Collection 内置的自然比较法(Comparable 接口)

- max(Collection)
- min(Collection)

4、返回参数 Collection 中最大或最小的元素--采用 Comparator 进行比较

- max(Collection, Comparator)
- min(Collection, Comparator)

5、返回 target 在 source 中第一次/最后一次出现的位置，找不到返回-1

- indexOfSubList(List source, List target)
- lastIndexOfSubList(List source, List target)

6、使用 newVal 替换所有的 oldVal

- replaceAll(List<T>, T oldValue, T newValue)

7、逆转所有元素的顺序

- reverse(List)

8、返回一个 Comparator，他可以逆转实现了 Comparator<T>的对象集合的自然顺序，第二个版本可以逆转所提供的 Comparator 的顺序

- reverseOrder()
- reverseOrder(Comparator<T>)

9、所有元素向后移动 distance 个位置，将末尾的元素循环到前面来

- rotate(List, int distance)

10、随机改变指定列表的顺序，第一种形式提供了自己的随机机制，你可以通过第二种形式提供自己的随机机制

- shuffle(List)
- shuffle(List, Random)

11、使用 List<T>中的自然顺序排序，第二种形式允许提供用于排序的 Comparator

- sort(List<T>)
- sort(List<T>, Comparator<? Super T> c)

12、将 src 中的元素复制到 dest(**dest 必须有足够的空间来容纳 src 的元素**)

- copy(List<? super T> dest, List<? extends T> src)

13、交换 List 中位置 i 与位置 j 的元素，通常比自己写要快

- swap(List, int i, int j)

14、用对象 x 替换 list 中的所有元素(**同一个对象的引用**)

- fill(List<? super T>, T x)

15、返回大小为 n 的 List<T>，此 List 不可改变，其中的引用都指向 x

- nCopies(int n, T x)

16、当两个集合没有任何元素相同时，返回 true

- disjoint(Collection, Collection)

17、返回 Collection 中等于 x 的元素个数

- frequency(Collection, Object x)

18、返回不可变的空 List、Map 或 Set，这些方法都是泛型，因此所产生的结果将被参数化为所希望的类型

- emptyList()
- emptyMap()
- emptySet()

19、产生不可变的 Set<T>、List<T>或 Map<K,V>，它们都只包含基于所给定参数的内容而形成的单一项

- singleton(T x)
- singletonList(T x)
- singletonMap(K key, V value)

### 17.11.1. List 的排序和查询

#### 17.11.2. 设定 Collection 或 Map 为不可修改

1、产生只读的 List<T>、Set<T>或 Map<K,V>，其实返回一个特定的容器，可变方法中都会抛出异常

- unmodifiableCollection(Collection<? extends T>)
- unmodifiableList(List<? extends T>)
- unmodifiableSet(Set<? extends T>)
- unmodifiableSortedSet(SortedSet<? extends T>)
- unmodifiableMap(Map<? extends K, ?extends V>)
- unmodifiableSortedMap(SortedMap<? extends K, ?extends V>)

2、对特定类型的"不可修改"方法的调用并不会产生编译时的检查，但是转换完成后，任何会改变容器的操作都会引起 UnsupportedOperationException 异常

3、无论哪种情况，在将容器设为只读之前，必须填入有意义的数据，装载数据后，就应该使用"不可修改"的方法返回引用去替换掉原本的引用，另外一方面，此方法允许你保留一份可修改的容器，作为类的 private 成员，然后通过某个方法调用返回对该容器的"只读"引用，这样一来，只有你可以修改容器的内容，而别人只能读取

### 17.11.3. Collection 或 Map 的同步控制

1、Collection 类有办法能够自动同步整个容器，其语法与"不可修改的"方法类似

- synchronizedCollection(Collection<? extends T>)
- synchronizedList(List< T>)
- synchronizedSet(Set< T>)
- synchronizedSortedSet(SortedSet< T>)
- synchronizedMap(Map<K,V>)
- synchronizedSortedMap(SortedMap<K,V>)

2、与"不可修改"类似，返回一个特定的对应容器，用 synchronized 同步块来实现每个方法，因此返回的容器自然就是同步的

3、快速报错

- Java 容器有一种保护机制，能够防止多个进程同时修改同一个容器的内容
- Java 容器类库采用快速报错(fail-fast)机制，它会探查容器上任何除了你的进程所进

行的操作以外的所有变化，一旦它发现其他进程修改了容器，就会立刻抛出 `ConcurrentModificationException` 异常

## 17.12. 持有引用

1、`java.lang.ref` 类库包含了一组类，这些类为垃圾回收提供了更大的灵活性，当存在可能会耗尽内存的大对象时，这些类显得特别有用

2、三个继承自抽象类 `Reference` 的类：`SoftReference`、`WeakReference`、`PhantomReference`

3、当垃圾回收器正在考察的对象只能通过某个 `Reference` 对象才"可获得"时，上述这些不同的派生类为垃圾回收器提供了不同级别的间接性提示

4、对象是可获得的(`reachable`)

- 是指对象可在程序中的某处找到，这意味着你在栈中有一个普通的引用，而他正指向此对象
- 也可能是你的引用指向某个对象，而那个对象含有另一个引用指向正在讨论的对象
- 也可能有更多的中间链接
- 如果一个对象是可获得的，那么垃圾回收器就不能释放它，如果一个对象是不可获得的，那么你的程序将无法使用到它，所以将其回收是安全的

5、如果想继续持有某个对象的引用，希望以后还能够访问到该对象，但是也希望能够允许垃圾回收器释放它，这时就应该使用 `Reference` 对象

6、`SoftReference`、`WeakReference`、`PhantomReference` 由强到弱排列，对应不同级别的"可获得性"

- `SoftReference`: 用以实现内存敏感的高速缓存
- `WeakReference`: 是为"规范映射"而设计的，它不妨碍垃圾回收器回收映射的"键"(或"值")，规范映射中对象的实例可以在程序的多处被同时使用，以节省存储空间
- `PhantomReference` 用以调度回收前的清理工作，它比 Java 终止机制更灵活
- 使用 `SoftReference` 和 `WeakReference` 可以选择是否将它们放入 `ReferenceQueue`(用作"回收前清理工作"的工具)。而 `PhantomReference` 只能依赖于 `ReferenceQueue`

### 17.12.1. WeakHashMap

1、容器类有一种特殊的 Map：`WeakHashMap` 用以保存 `WeakHashReference`。它使得规范映射更易于使用

2、在这种映射中，每个值只保存一份实例以节省存储空间，当程序需要那个"值"的时候，便在映射中查询现有的对象，然后使用它(而不是重新再创建)???

3、映射可将值作为其初始化中的一部分，不过通常是在需要的时候才生成"值"???

## 17.13. Java 1.0/1.1 的容器

1、在写新的程序时，绝不应该使用旧的容器，但你应该了解它们

### 17.13.1. Vector 和 Enumeration

1、Java 1.0/1.1 中，`Vector` 是唯一可以自我扩展的序列，所以它被大量使用，但缺点多到难以描述

2、Java 1.0/1.1 版的迭代器发明了一个新名字--枚举，取代了为人熟知的术语(迭代器)。`Enumeration` 接口比 `Iterator` 小，只有两个名字很长的方法

- 一个为 boolean hasMoreElements()
- 另一个为 Object nextElement()

### 17.13.2. **Hashtable**

1、基本的 **Hashtable** 与 **HashMap** 很相似，甚至方法名也相似，在新程序中，没有理由再使用 **Hashtable** 而不用 **HashMap**

### 17.13.3. **Stack**

1、Java 1.0/1.1 的栈很奇怪，继承自 **Vector**，他拥有 **Vector** 所有的特点和行为，加上一些额外的 **Stack** 的行为，永远都不应该用它，**而应该使用 LinkedList**

### 17.13.4. **BitSet**

1、想要高效地存储大量"开关"信息，**BitSet** 是很好的选择，不过它的效率仅是对空间而言，如果需要高效的访问时间，**BitSet** 比本地数组慢一点

2、**BitSet** 的最小容量是 **long**: 64 位，如果存储的内容比较小，例如 8 位，那么 **BitSet** 就浪费了一些空间

## Chapter 18. Java I/O 系统

1、对程序语言的设计者来说，创建一个好的输入/输出(I/O)系统是一项艰难的任务

- 不仅存在各种 I/O 源端和想要与之通信的接收端(文件、控制台、网络链接等)
- 还需要以许多种不同的方式与它们进行通信(顺序、随机存取、缓冲、二进制、按字符、按行、按字节等)

### 18.1. File 类

1、File(文件)这个名字有一定的误导性，我们可能认为它指代的是文件

- 它既能代表一个特定文件的名称，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，我们就可以对此集合调用 list()方法，这个方法会返回一个字符数组

#### 18.1.1. 目录列表器

1、查看一个目录列表

- 可以用两种方式来使用 File 对象
- 如果调用不带参数的 list()方法，便可以获得此 File 对象包含的全部列表
- 如果想要获得一个受限的列表，例如想要获得所有扩展名为.java 的文件，那么就需要用到“目录过滤器”

2、FilenameFilter 接口

```
public interface FilenameFilter{
 boolean accept(File dir, String name);
}
```

- 通过将 **FilenameFilter** 接口的实现(函数对象???)作为 **File.list()** 的参数，提供给 **list()** 使用，使得 **list()** 可以回调 **accept()**，进而决定哪些文件包含在列表中，这种结构常常称为回调
- 更具体地说，这是一个策略模式的例子，因为 **list()** 实现了基本的功能，而且按照 **FilenamFilter** 的形式提供了这个策略，以便完善 **list()** 在提供服务时所需的算法
- **list()** 方法为为此目录下的每个文件名调用 **accept()**，来判断该文件是否包含在内，判断结果由 **accept()** 返回的布尔值表示
- 惯用写法

```
class MyFilter implements FilenameFilter{
 private Pattern pattern;
 public MyFilter(String regex){
 pattern=Pattern.compile(regex);
 }
 public boolean accept(File dir, String name){
 return pattern.matcher(name).matches();
 }
}
```

- 或者直接用匿名内部类 **file.list(new FilternameFilter()/\* \*/);**

### 18.1.2. 目录实用工具

1、<未完成>

### 18.1.3. 目录的检查及创建

1、File 类不仅仅只代表存在的文件或目录，也可以用 File 对象来创建新的目录或尚不存在的整个目录路径

2、File 的方法

- File.canWrite(): 返回文件是否可写。
- File.canRead(): 返回文件是否可读。
- File.getName(): 返回文件名称
- File.getParent(): 返回父目录路径
- File.getPath(): 返回文件路径
- File.length(): 返回文件长度？？
- File.list(): 返回文件或目录清单(String[])//不会继续访问子文件夹
- File.listFiles(): 返回文件或目录清单(File[])//不会继续访问子文件夹
- File.exists(): 返回文件是否存在(boolean)
- File.isDirectory(): 返回该路径指示的是不是文件夹
- File.isFile(): 判断该路径指示的是不是文件(\*.\*)
- File.renameTo(): 文件更名
- File.mkdirs(): 生成指定的目录(当文件不存在时)

## 18.2. 输入和输出

1、编程语言的 I/O 类库常使用流这个抽象概念，它代表任何有能力产出数据的数据源对象或者是有能力接受数据的接收端对象，流屏蔽了实际的 I/O 设备中处理数据的细节

2、Java 类库中 I/O 类分成输入和输出两部分

- 任何 InputStream 或 Reader 派生而来的类都含有名为 read()的基本方法，用于读取单个字节或者字节数组
- 任何 OutputStream 或 Write 派生而来的类都含有名为 write()的基本方法，用于写单个字节或者字节数组
- 但是通常，我们不会用到这些方法，它们之所以存在是因为别的类可以使用它们，以便提供更有用的接口
- **因此我们很少使用单一的类来创建流对象，而是通过叠合多个对象来提供所期望的功能(装饰器设计模式)**
- Java 中"流"类库让人迷惑的主要原因就在于：创建单一的结果流，却需要创建多个对象
- 在 Java 1.0 中，类库设计者首先限定与输入有关的所有类都应该从 InputStream 继承，而与输出有关的所有类都应该从 OutputStream 继承

### 18.2.1. InputStream 类型

1、InputStream 的作用是用来表示那些从不同数据源产生输入的类，包括

- 字节数组
- String 对象
- 文件

- "管道", 工作方式与实际管道相似, 即从一端输入, 另一端输出
- 一个由其他种类的流组成的序列, 以便我们可以将它们收集合并到一个流内
- 其他数据源, 如 Internet 连接等

2、每一种数据源都有相应的 `InputStream` 子类, 另外 `FilterInputStream` 也属于一种 `InputStream` 为"装饰器"(decorator)类提供基类, "装饰器"类可以把属性或有用的接口与输入流连接在一起

表格 18-1 `InputStream` 类型

| 类                                    | 功能                                                               | 构造器参数/如何使用                                                                                                                                                        |
|--------------------------------------|------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>ByteArrayInputStream</code>    | 允许将内存缓冲区当做 <code>InputStream</code>                              | 构造器参数: 缓冲区, 字节将从中取出<br>使用: 作为一种数据源, 将其与 <code>FilterInputStream</code> 对象相连已提供有用接口                                                                                |
| <code>StringBufferInputStream</code> | 将 <code>String</code> 转化成 <code>InputStream</code>               | 构造器参数: 字符串(底层实现实际使用 <code>StringBuffer</code> )<br>使用: 作为一种数据源, 将其与 <code>FilterInputStream</code> 对象相连已提供有用接口                                                    |
| <code>FileInputStream</code>         | 用于从文件中读取信息                                                       | 构造器参数: 字符串, 表示文件名、文件或 <code>FileDescriptor</code> 对象<br>使用: 作为一种数据源, 将其与 <code>FilterInputStream</code> 对象相连已提供有用接口                                               |
| <code>PipedInputStream</code>        | 产生用于写入相关 <code>PipedOutputStream</code> 的数据, 实现管道化概念             | 构造器参数: <code>PipedOutputStream</code><br>使用: 作为多线程中数据源, 将其与 <code>FilterInputStream</code> 对象相连已提供有用接口                                                            |
| <code>SequenceInputStream</code>     | 将两个或多个 <code>InputStream</code> 对象转换成单一 <code>InputStream</code> | 构造器参数: 两个 <code>InputStream</code> 对象或一个容纳 <code>InputStream</code> 对象的容器 <code>Enumeration</code><br>使用: 作为一种数据源, 将其与 <code>FilterInputStream</code> 对象相连已提供有用接口 |
| <code>FilterInputStream</code>       | 抽象类, 作为"装饰器"的接口, 装饰器为其他 <code>InputStream</code> 提供有用接口          | 见表格 18-3<br>见表格 18-3                                                                                                                                              |

### 18.2.2. `OutputStream` 类型

- 1、`OutputStream` 决定了输出所要去往的目标: 字节数组、文件或管道
- 2、另外, `FilterOutputStream` 为"装饰器"类提供了一个基类, "装饰器"类把属性或者有用的接口与输出流连接了起来

表格 18-2 OutputStream 类型

| 类                     | 功能                                             | 构造器参数/如何使用                                                                                    |
|-----------------------|------------------------------------------------|-----------------------------------------------------------------------------------------------|
| ByteArrayOutputStream | 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                  | 构造器参数: 缓冲区初始化尺寸(可选)<br>使用: 用于指定数据的目的地, 将其与 FilterOutputStream 对象相连已提供有用接口                     |
| FileOutputStream      | 用于将信息写至文件                                      | 构造器参数: 字符串, 表示文件名、文件或 FileDescriptor 对象<br>使用: 用于指定数据的目的地, 将其与 FilterOutputStream 对象相连已提供有用接口 |
| PipedOutputStream     | 任何写入其中的信息都会自动作为相关 PipedInputStream 的输出。实现管道化概念 | PipedInputStream<br>使用: 指定用于多线程的数据的目的地, 将其与 FilterOutputStream 对象相连已提供有用接口                    |
| FilterOutputStream    | 抽象类, 作为"装饰器"的接口。其中"装饰器"为其他 OutputStream 提供有用功能 | 见表格 18-4<br>见表格 18-4                                                                          |

## 18.3. 添加属性和有用的接口

1、Java I/O 类库需要多种不同功能的组合, 这正是用装饰器模式的理由所在。这也是 Java I/O 类库里存在 filter(过滤器)类的原因所在, 抽象类 filter 是所有装饰器类的基类。装饰器类必须具有和它所装饰的对象相同的接口, 但它也可以扩展接口

2、装饰器的缺点: 虽然它给我们提供了相当多的灵活性(因为我们可以很容易地混合和匹配属性), 但是它同时也增加了代码的复杂性

- Java I/O 类库操作不便的原因在于: 我们必须创建许多类---"核心" I/O 类加上所有的装饰器, 才能得到我们所希望的单个 I/O 对象

### 18.3.1. 通过 FilterInputStream 从 InputStream 读取数据

1、FilterInputStream 类能够完成两件完全不同的事情

- DataInputStream 允许我们读取不同的基本类型数据以及 String 对象, 所有方法都以 read 开头, 例如 readByte(), readFloat() 等等, 搭配相应的 DataOutputStream, 我们就可以通过数据流将基本类型的数据从一个地方迁移到另一个地方
- 其他 FilterInputStream 类则在内部修改 InputStream 的行为方式: 是否缓冲, 是否保留它所读过的行(允许我们查询行数或设置行数), 以及是否把单一字符推回输入流等等
- **我们几乎每次都要对输入进行缓冲---不管我们正在连接的是什么 I/O 设备**, 所以 I/O 类库把无缓冲输入作为特殊情况就显得更加合理了

表格 18-3 FilterInputStream

| 类                   | 功能                                                                                                                                      | 构造器参数/如何使用                                                   |
|---------------------|-----------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------|
| DataInputStream     | 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，其对应的方法就是 DataInputStream.readInt(...)/.readBoolean(...)/readLong(...)等等 | 构造器参数: InputStream<br>使用: 包含用于读取基本类型数据的全部接口                  |
| BufferedInputStream | 使用它可以防止每次读取时都得实行实际写操作。代表使用缓冲区                                                                                                           | 构造器参数: InputStream<br>使用: 本质上不同恭接口，只不过是向进程中添加缓冲区所必须的。与接口对象搭配 |

### 18.3.2. 通过 FilterOutputStream 向 OutputStream 写入

1、与 DataInputStream 对应的是 DataOutputStream，他可以将各种基本数据类型以及 String 对象格式化输出到流中，这样一来，任何机器上的任何 DataInputStream 都能够读取它们，所有方法都以 write 开头，例如 writeByte(), writeFloat()

2、PrintStream 最初的目的便是为了可视化格式打印所有的基本数据类型以及 String 流

- PrintStream 内有两个重要的方法: print()和 println(), 后者会添加一个换行符
- PrintStream 可能会有问题，因为它捕捉了所有的 IOExceptions，因此我们必须使用 checkError()自行测试错误状态，如果出现错误它返回 true
- 另外 PrintStream 也未完全国际化，不能以平台无关的方式处理换行动作，这些问题在 PrintWrite 中得到了解决

3、BufferedOutputStream 是一个修改过的 OutputStream，它对数据流使用缓冲技术，因此当每次向流写入时，不必每次都进行实际的物理写动作

表格 18-4 FilterOutputStream 类型

| 类                | 功能                                                                                                                                         | 构造器参数/如何使用                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------|
| DataOutputStream | 与 DataInputStream 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，其对应的方法就是 DataOutputStream.writeInt(...)/.writeBoolean(...)/writeLong(...)等等 | 构造器参数: OutputStream<br>使用: 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 处理数据的显示                                                                                  | 构造器参数: OutputStream<br>, 可以使用 boolean(选用)指示是否在每次换行时清空缓冲区<br>使用: 对 OutputStream 对象的 final 封装 |

|                      |                                                      |                                                                   |
|----------------------|------------------------------------------------------|-------------------------------------------------------------------|
| BufferedOutputStream | 使用它以避免每次发送数据时都要进行实际的写操作。<br>代表使用缓冲区，可调用 flush()清空缓冲区 | 构造器参数: OutputStream，可以指定缓冲区大小(可选)<br>使用: 本质上并不提供接口，只不过向进程添加缓冲区所必须 |
|----------------------|------------------------------------------------------|-------------------------------------------------------------------|

## 18.4. Reader 和 Writer

- 1、Reader 和 Writer 提供兼容 Unicode 与面向字符的 I/O 功能
- 2、InputStream 与 OutputStream 仍有极为重要的作用，而非被 Reader 和 Writer 取代
- 3、有时我们必须把来自于"字节"层次结构中的类和"字符"层次结构中的类结合起来使用，为了实现这个目的，要用到"适配器(adapter)"类:
  - InputStreamReader: 将 InputStream 转为 Reader
  - OutputStreamWriter: 将 OutputStream 转为 Writer
- 4、设计 Reader 和 Writer 继承层次结构主要是为了国际化，老的 I/O 流继承层次结构仅支持 8 位字节流，并不能很好地处理 16 位的 Unicode 字符，由于 Unicode 用于字符国际化(Java 本身的 char 也是 16 位的 Unicode)，另外新类库的设计使得它的操作比旧类库更快

### 18.4.1. 数据的来源和去处

- 1、几乎所有原始的 Java I/O 流类都有相应的 Reader 和 Writer 类来提供天然的 Unicode 操作，然而在某些场合，面向字节的 InputStream 和 OutputStream 才是正确的解决方案，特别是 java.util.zip 类库就是面向字节的而不是面向字符的
- 2、建议：尽量尝试 Reader 和 Writer

表格 18-5 来源与去处表

| 来源                           | 去处                              |
|------------------------------|---------------------------------|
| InputStream                  | Reader(适配器: InputStreamReader)  |
| OutputStream                 | Writer(适配器: OutputStreamWriter) |
| FileInputStream              | FileReader                      |
| FileOutputStream             | FileWriter                      |
| StringBufferInputStream(已弃用) | StringReader                    |
| \                            | StringWriter                    |
| ByteArrayInputStream         | CharArrayReader                 |
| ByteArrayOutputStream        | CharArrayWriter                 |
| PipedInputStream             | PipedReader                     |
| PipedOutputStream            | PipedWriter                     |

### 18.4.2. 更改流的行为

- 1、对于 InputStream 和 OutputStream 来说，我们会使用 FilterInputStream 和 FilterOutputStream 的装饰器子类来修改"流"以满足特殊需要。Reader 和 Writer 的类继承层次结构继续沿用相同的思想---但是并不完全相同
- 2、尽管 BufferedOutputStream 是 FilterOutputStream 的子类，但是 BufferedWriter 并不是 FilterWriter 的子类(**尽管 FilterWriter 是抽象类，但其没有任何子类，把它放在那里也只是把它作为一个占位符，或者仅仅让我们不会对它所在的地方产生疑惑**)

表格 18-6 Filter 对应关系表

| 来源                         | 去处                                                         |
|----------------------------|------------------------------------------------------------|
| FilterInputStream          | FilterReader                                               |
| FilterOutputStream         | FilterWriter(抽象类, 没有子类)                                    |
| BufferedInputStream        | BufferedReader(也有 readLine())                              |
| BufferedOutputStream       | BufferedWriter                                             |
| DataInputStream            | DataInputStream(当需要使用 readLine()时,<br>应该使用 BufferedReader) |
| PrintStream                | PrintWriter                                                |
| LineNumberInputStream(已弃用) | LineNumberReader                                           |
| StreamTokenizer            | StreamTokenizer                                            |
| PushbackInputStream        | PushbackReader                                             |

3、无论我们何时使用 `readLine()`都不应该使用 `DataInputStream`(这会遭到编译器的强烈反对), 而应该使用 `BufferedReader`, 除了这一点, `DataInputStream` 仍是 I/O 类库的首选成员

4、为了更容易地过渡到使用 `PrintWriter`, 它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

#### 18.4.3. 未发生变化的类

1、以下类在 Java 1.0 和 Java1.1 之间则未做改变

- `DataOutputStream`
- `File`
- `RandomAccessFile`
- `SequenceInputStream`

### 18.5. 自我独立的类: `RandomAccessFile`

1、`RandomAccessFile` 适用于由大小已知的记录组成的文件, 所以我们可以使用 `seek()` 将记录从一处转移到另一处, 然后读取或者修改记录

2、`RandomAccessFile` 实现了 `DataInput` 与 `DataOutput` 接口(`DataInputStream` 和 `DataOutputStream` 也实现了这两个接口)之外, 它和这两个继承层次结构没有任何关联

3、从本质上来说, `RandomAccessFile` 的工作方式类似于把 `DataInputStream` 和 `DataOutputStream` 组合起来使用, 还添加了一些方法

- `getFilePointer()`: 返回当前所处的文件位置
- `seek()`: 用于在文件内移至新位置
- `length()`: 返回文件的大小(字节数)

4、其构造器还需要第二个参数(类似于 C 中的 `fopen`), 用于指示"读写"功能

5、只有 `RandomAccessFile` 支持搜寻方法, 并且只适用于文件

### 18.6. I/O 流的典型使用方式

#### 18.6.1. 缓冲输入文件

1、如果想要打开一个文件用于字符输入, 可以使用 `String` 或 `File` 对象作为文件名的 `FileInputStream`, 为了提高速度, 我们希望对那个文件进行缓冲, 那么我们将所产生的引用传给一个 `BufferedReader` 构造器

## 2、惯用法：

```
BufferedReader in=new BufferedReader(new FileReader(<filename>));
String s;
StringBuilder sb=new StringBuilder();
while((s=in.readLine())!=null)
 sb.append(s+"\n");
in.close();
...
➤ 注意，readLine()方法会读取一行，返回一个舍弃最后一个换行符的 String
➤ 当 readLine()放回 null 时，表明到达了文件尾
```

## 18.6.2. 从内存输入

### 1、惯用法

```
StringReader in=new StringReader(<String>);
int c
while((c=in.read())!=-1)
 System.out.println((char)c);
➤ read()是以 int 形式返回下一字节，因此必须类型转换为 char 才能正确打印
```

## 18.6.3. 格式化的内存输入

### 1、要读取格式化数据，可以使用 DataInputStream，它是一个面向字节的 I/O 类

### 2、惯用法

```
try{
 DataInputStream in=new DataInputStream(new ByteArrayInputStream(<byte ary>));
 while(true)
 System.out.print((char)in.readByte());
}catch(EOFException e){
 System.err.println("End of stream");
}
➤ 由于用 readByte()一次一个字节地读取字符(其返回类型是 byte 与 read()方法返回 int 类型不同)，那么任何字节都是合法的结果，因此返回值不能用来检测输入是否结束(read()可以用返回值是否是-1 来判断结尾，但 readByte()不可以)。另外，我们可以用 available()方法查看还有多少可供存取的字节
```

```
DataInputStream in=new DataInputStream(new ByteArrayInputStream(<byte ary>));
while(in.available()!=0)
 System.out.print((char)in.readByte());
➤ 注意，available()的工作方式会随着所读取的媒介类型的不同而有所不同。其字面意识就是：在没有阻塞的情况下所能读取的字节数，对于文件这意味着整个文件，但是对于不同类型的流，可能就不是这样，因此要谨慎使用
➤ 我们也可以通过捕获异常来检测输入的末尾，但是使用异常进行流控制被认为是对异常特性的错误使用
```

## 18.6.4. 基本的文件输出

### 1、FileWriter 对象可以向文件写入数据，首先，创建一个与指定文件连接的 FileWriter，实际

上，我们通常会用 `BufferedWriter` 将其包装起来用以缓冲输出(缓冲往往能明显地增加 I/O 操作的性能)，若需要提供格式化机制，再将其包装成 `PrintWriter`

## 2、惯用法

```
PrintWriter out=new PrintWriter(new BufferedWriter(new FileWriter(<file>)));
int lineCount=1;
String s;
while((s=in.readLine()!=null)
 out.println(lineCount++ +": "+s);
out.close();
```

- 我们并未使用 `LineNumberInputStream`，而是自己实现行号的功能，这个类没有多大帮助，因此没有必要用它
- 如果我们不为所有输出文件调用 `close()`，那么缓冲区可能不会被刷新清空

## 3、文本文件输出的快捷方式

- Java SE5 在 `PrintWriter` 中添加了一个辅助构造器，使得你不必在每次希望创建文本并向其写入时，都去执行所有的装饰工作

```
PrintWriter out=new PrintWriter(<file>);
➤ 以下是该构造器的源代码
public PrintWriter(String fileName) throws FileNotFoundException {
 this(new BufferedWriter(
 new OutputStreamWriter(
 new FileOutputStream(fileName))), false);
}
```

- 可以看出仍然会使用缓存，只是不必自己去装饰了

### 18.6.5. 存储和恢复数据

1、`PrintWriter` 可以对数据进行格式化，以便人们的阅读，但是为了输出可供另一个"流"恢复的数据，我们需要使用 `DataOutputStream` 写入数据，并用 `DataInputStream` 恢复数据

2、如果我们用 `DataOutputStream` 写入数据，Java 保证我们可以使用 `DataInputStream` 准确地读取数据---无论读和写数据的平台多么不同，只要两个平台都有 Java，那么这种问题就不会再发生

3、当我们使用 `DataOutputStream` 时，写字符串并且让 `DataInputStream` 能够恢复它的唯一可靠做法就是使用 `UTF-8` 编码

- `UTF-8` 是一种多字节格式，其编码长度根据实际使用的字符集会有所变化
- 将 ASCII 编为单一字节形式，而非 ASCII 字符则编码成两到三个字节的形式
- 另外，字符串的长度存储在 `UTF-8` 字符串的前两个字节中
- **writeUTF()和 readUTF()使用的是适用于 Java 的 UTF-8 变体**，因此用一个非 Java 程序读取用 `writeUTF()` 所写的字符串时，必须编写一些特殊的代码才能正确读取字符串

4、我们可以用 `writeDouble`、`writeInt` 等方法将数字存储到流中，并用相应的 `readDouble`、`readInt` 恢复它

- 为了保证所有的读方法都能正常工作，我们必须知道流中数据项所在的确切位置，因为我们在不知道其存储状况时，很有可能将保存的 `double` 数据作为一个简单的字节序列，`char` 或其他类型读入
- **因此，我们必须：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本中，以便能够对其进行解析以确定数据存放的位置**

- 对象序列化和 XML 可能使更容易的存储和读取复杂数据结构的方式

### 18.6.6. 读写随机访问文件

- 1、使用 RandomAccessFile，类似于组合使用了 DataInputStream 和 DataOutputStream，因为它们是实现了相同的接口
- 2、在使用 RandomAccessFile 时，你必须知道文件排版，这样才能正确地操作它，RandomAccessFile 拥有读取基本类型和 UTF-8 字符串的各种具体方法
- 3、RandomAccessFile 除了实现 DataInput 和 DataOutput 接口之外，有效地与 I/O 继承层次结构的其他部分实现了分离，因为它不支持装饰，所以不能将 InputStream 以及 OutputStream 子类的任何部分组合起来，我们必须假定 RandomAccessFile 已经被正确缓冲，因为我们不能为他添加这样的功能
- 4、可能会考虑使用"内存映射文件"代替 RandomAccessFile

### 18.6.7. 管道流

- 1、PipedInputStream
- 2、PipedOutputStream
- 3、PipedReader
- 4、PipedWriter

## 18.7. 文件读写的实用工具

- 1、Java I/O 类库的问题之一就是：它需要编写相当多的代码去执行这些常用操作---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
- 2、Java SE5 在 PrintWriter 中添加了方便的构造器，因此你可以很方便地打开一个文本文件进行写入操作

### 18.7.1. 读取二进制文件

## 18.8. 标准 I/O

- 1、标准 I/O 这个属于参考的是 Unix 中"程序所使用的单一信息流"这个概念
- 2、程序的所有输入都可以来自于标准输入，它的所有输出也都可以发送到标准输出，以及所有错误信息都可以发送到标准错误
- 3、标准 I/O 的意义在于：我们可以很容易地把程序串联起来，一个程序的标准输出可以称为另一程序的标准输入

### 18.8.1. 从标准输入中读取

- 1、按照 Java I/O 模型，Java 提供了 System.in、System.out、System.err
  - System.out 已经事先被包装成了 PrintStream 对象
  - System.err 同样也是 PrintStream
  - **System.in 是一个没有被包装过的未经加工的 InputStream**
- 2、我们可以直接使用 System.out 与 System.err，但在读取 System.in 之前必须对其进行包装

### 18.8.2. 将 System.out 转换成 PrintWriter

1、System.out 是一个 PrintStream，而 PrintStream 是一个 OutputStream，而 PrintWriter 有一个可以接受 OutputStream 作为参数的构造器，因此，只要需要，就可以使用那个构造器把 System.out 转换成 PrintWriter

### 18.8.3. 标准 I/O 重定向

1、Java 的 System 类提供了一些简单的静态方法调用，以允许我们对标准输入、输出和错误 I/O 流进行重定向

- **System.setIn(InputStream)**
- **System.setOut(PrintStream)**
- **System.setErr(PrintStream)**

2、如果我们突然开始在显示器上创建大量输出，而这些输出滚动的太快以至于无法阅读时，重定向输出就显得极为有用

3、I/O 重定向操纵的是字节流，而不是字符流

## 18.9. 进程控制

1、你经常会需要在 Java 内部执行其他操作系统的程序，并且要控制这些程序的输入和输出，Java 类库提供了执行这些操作的类

2、一项常见的任务是运行程序，并将产生的输出发送到控制台

3、<未完成>：例子没懂

## 18.10. 新 I/O

1、JDK 1.4 的 java.nio\* 包中引入了新的 Java I/O 类库，目的在于提高速度，实际上旧的 I/O 包已经使用 nio 重新实现过，以便充分利用这种速度提高

2、速度的提高来自于所使用的结构更接近于操作系统执行 I/O 的方式：**通道**和**缓冲器**

3、**我们并有直接和通道交互，我们只是和缓冲器交互**，并把缓冲器派送到通道，通道要么从缓冲器获得数据，要么向缓冲器发送数据

4、**唯一直接与通道交互的缓冲器是 ByteBuffer**---可以存储未加工字节的缓冲器

- java.nio.ByteBuffer 是一个相当基础的类：通过告知分配多少存储空间来创建一个 ByteBuffer 对象，并且还有一个方法选择集，用于以原始的字节形式或基本数据类型输出和读取
- 没有办法输出或读取对象，即使是字符串对象也不行，这种处理很低级，但却正好，因为这是大多数操作系统中更有效的映射方式
- **当需要从缓冲器中读取数据时，必须调用 ByteBuffer.flip() 让缓冲器做好让别人读取字节的准备**
- **当需要向缓冲器中写入数据时，必须调用 ByteBuffer.clear() 让缓冲器做好让别人写字节的准备**

5、旧的 I/O 类库中有三个类被修改了，用以产生 FileChannel

- 这三个被修改的类是 FileInputStream、FileOutputStream 以及用于读写的 RandomAccessFile
- 这些事字节操纵流，与底层的 nio 性质一致
- Reader 和 Writer 这种字符模式类不能用于产生通道，但是 java.nio.channels.Channel

s 类提供了实用方法，用以在通道中产生 Reader 和 Writer

6、通过 getChannel() 将会产生一个 FileChannel，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的访问

7、将字节放于 ByteBuffer 的方法

- 使用一种"put"方法直接对它进行填充，填入一个或多个字节，或基本数据类型的值
- 也可以使用 wrap()方法将已存在的字节数组"包装"到 ByteBuffer 中，**一旦如此，就不复复制底层的数组，而是把它作为产生的 ByteBuffer 的存储器，称之为数组支持的 ByteBuffer**
- 对于只读访问(从管道中读入 ByteBuffer，即对于 ByteBuffer 是写入)，必须显式地用静态的 allocate()方法来分配 ByteBuffer。**nio 的目标就是快速移动大量数据，因此 ByteBuffer 的大小就显得尤为重要，必须通过实际运行程序来找到最佳尺寸(甚至达到更高的速度也有可能，方法就是使用 allocateDirect()而不是 allocate())**，以产生一个与操作系统有更高耦合性的"直接缓冲器"，但是这种分配的开支会更大，并且具体实现也随操作系统的不同而不同，因此必须再次实际运行应用程序来查看直接缓冲是否可以使我们获得速度上的优势)

7、**FileChannel.read(ByteBuffer)返回-1 时表示已经到达了输入的末尾**

8、特殊方法 FileChannel.transferTo()和 FileChannel.transferFrom()允许我们将一个通道和另一个通道直接连接

- ```
public abstract long transferTo(long position, long count,
                                  WritableByteChannel target)
```
- ```
public abstract long transferFrom(ReadableByteChannel src,
 long position, long count)
```

9、ByteBuffer 的方法

- `ByteBuffer.rewind()` //返回到数据的开始部分
- `ByteBuffer.flip()` //让缓冲器做好让别人读取字节的准备
- `ByteBuffer.clear()` //让缓冲器做好让别人写字节的准备
- `ByteBuffer.allocate(int size)` //创建指定大小的 ByteBuffer 对象
- `ByteBuffer.wrap(byte[])` //返回一个包装了该字节数组的 ByteBuffer

10、惯用法

```
FileChannel fc=new FileOutputStream(<filename>).getChannel();
fc.write(ByteBuffer.wrap(<String1>.getBytes()));
fc.close();
```

```
fc=new RandomAccessFile(<filename>).getChannel();
fc.position(fc.size()); //注意，只有通过 RandomAccessFile 打开的管道才是会保留已有内容的，也就是此时 fc.size() 是文件原本的大小，若是用 FileOutputStream 打开，那么文件的大小将会变为 0
```

```
fc.write(ByteBuffer.wrap(<String2>.getBytes()));
fc.close();
```

```
fc=new FileInputStream(<filename>).getChannel();
ByteBuffer buff=ByteBuffer.allocate(1024);
fc.read(buff); //这里最多从管道读取 1024byte 存入 ByteBuffer 内
buff.flip(); //做好让人从 ByteBuffer 中读取的准备
```

```
while(buff.hasRemaining()){
 System.out.print((char)buff.get());
```

### 18.10.1. 转换数据

1、缓冲器容纳的是普通的字节，为了把它们转换成字符([上一小节用强制类型转换](#))，我们要么在输入它们的时候对其进行编码(这样，它们输出时才有意义)，要么在将其从缓冲器输出时对它们进行解码

2、案例一

```
//普通方式写入文件
FileChannel fc=new FileOutputStream(<filename>).getChannel();
fc.write(ByteBuffer.wrap("Some text".getBytes()));
fc.close();

//从管道中读取数据后放入 ByteBuffer
fc=new FileInputStream("<filename>").getChannel();
ByteBuffer buff=ByteBuffer.allocate(BSIZE);
fc.read(buff);
buff.flip();
System.out.println(buff.asCharBuffer());//Doesn't work!, 输出将是乱码

//使用平台的默认字符集对数据进行 decode，那么作为结果的 CharBuffer 可以很好地
//输出打印到控制台，使用 System.getProperty("file.encoding")发现默认字符集，它会产
//生代表字符集名称的字符串，把该字符串传给 Charset.forName()用以产生 Charset 对
//象，用它可以对字符串进行解码
buff.rewind();
String encoding=System.getProperty("file.encoding");
System.out.println(Charset.forName(encoding).decode(buff));
```

3、案例二

//在读文件时，使用能够产生可打印的输出字符集进行 encode()，这里 UTF-16BE 可以
//把文本写到文件中，当读取时，我们只需要把它转换成 CharBuffer，就会产生期望的
//文本

```
FileChannel fc=new FileOutputStream(<filename>).getChannel();
fc.write(ByteBuffer.wrap("Some text".getBytes("UTF-16BE")));
fc.close();

fc=new FileInputStream("<filename>").getChannel();
buff.clear();
fc.read(buff);
buff.flip();
System.out.println(buff.asCharBuffer());
```

4、案例三

```
//通过 CharBuffer 向 ByteBuffer 写入
FileChannel fc=new FileOutputStream(<filename>).getChannel();
buff=ByteBuffer.allocate(24);
buff.asCharBuffer().put("Some text");
fc.write(buff);
```

```
fc.close();

fc=new FileInputStream(<filename>).getChannel();
buff.clear();
fc.read(buff);
buff.flip();
System.out.println(buff.asCharBuffer());
```

5、注意"Some text".getBytes()并不等于 new byte[]{'S', 'o', 'm', 'e', ' ', 't', 'e', 'x', 't'}, 这就是为什么会产生乱码的原因(一个 char 是 16 位, 而一个 byte 只有 8 位)

### 18.10.2. 获取基本类型

1、尽管 ByteBuffer 只能保存字节类型的数据, 但是它具有可以从其所容纳的字节冲产生出各种不同基本类型值的方法

2、惯用法

```
ByteBuffer bb=ByteBuffer.allocate(1024);
bb.rewind();
bb.asCharBuffer().put("Howdy!");
char c;
while((c=bb.getChar())!=0)
 printnb(c+" ");
print();
bb.rewind();
bb.asShortBuffer().put((short)471142);
print(bb.getShort());
bb.rewind();
bb.asIntBuffer().put(99471142);
print(bb.getInt());
bb.rewind();
bb.asLongBuffer().put(99471142);
print(bb.getLong());
bb.rewind();
bb.asFloatBuffer().put(99471142);
print(bb.getFloat());
bb.rewind();
bb.asDoubleBuffer().put(99471142);
print(bb.getDouble());
bb.rewind();
```

3、注意, 使用 ShrotBuffer 的 put()方法时, 需要进行类型转换, 其他所有的试图缓冲器在使用 put()方法时, 不需要进行类型转换

### 18.10.3. 视图缓冲器

1、视图缓冲器(view buffer)可以让我们通过某个特定的基本数据类型的试图查看其底层的 ByteBuffer, ByteBuffer 依然是实际存储数据的地方

2、对视图的任何修改会映射称为对 ByteBuffer 中数据的修改

3、视图允许我们从 ByteBuffer 一次一个地(与 ByteBuffer 所支持的方式相同)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(0); //放入一个值
ib.put(new int[]{11,42,47,99,143,811,1016}); //放入一组值
ib.put(3,1811); //修改对应位置的值, 如果在可修改的范围之外, 则什么也不做?
```

4、字节存放顺序

- 不同的机器可能会使用不同的字节排序方法来存储数据
- "big endian"(高位优先)将最重要的字节放在地址最低的存储器单元
- "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 是以高位优先的形式存储数据的

#### 18.10.4. 用缓冲器操纵数据

- 1、如果一个字节数组写到文件中去, 那就应该使用 ByteBuffer.wrap()方法把字节数组包装起来, 然后用 getChannel()方法在 FileOutputStream 上打开一个通道, 接着将来自于 ByteBuffer 的数据写到 FileChannel 中
- 2、ByteBuffer 是将数据移进移出通道的唯一方式, 并且我们只能创建一个独立的基本类型缓冲器, 或者用"as..."方法从 ByteBuffer 中获得, 也就是说, 我们不能把基本类型的缓冲器转换成 ByteBuffer

#### 18.10.5. 缓冲器的细节

1、Buffer 由数据和可以高效地访问及操纵这些数据的四个索引组成

- mark
- position
- limit
- capacity

|                   |                                            |
|-------------------|--------------------------------------------|
| capacity()        | 返回缓冲区容量                                    |
| clear()           | position=0;limit=capacity;                 |
| flip()            | limit=position;position=0                  |
| limit()           | 返回 limit 值                                 |
| limit(int limit)  | 设置 limit 值                                 |
| mark()            | mark=position                              |
| reset()           | position=mark                              |
| position()        | 返回 position 值                              |
| position(int pos) | 设置 position 值                              |
| remainning()      | 返回 limit-position                          |
| hasRemaining()    | 若有介于 position 和 limit 的元素, 返回 true         |
| get()             | 返回 position 指向的字节值, 并将 position+1          |
| put()             | 向 position 指向的字节写入值, 并将 position+1         |
| get(int pos)      | 返回指定 pos 指向的字节的值, position 与 pos 无关, 且不做变化 |
| put(int pos)      | 向指定 pos 指向的字节写入值, position 与 pos 无关, 且不做变化 |

### 18.10.6. 内存映射文件

- 1、**内存映射文件允许我们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，我们就可以假定整个文件都放在内存中，而且可以完全把它当做非常大的数组来访问
- 2、为了既能读也能写，我们用 RandomAccessFile 获取文件上的通道，然后调用 map() 产生 MappedByteBuffer，这是一种特殊类型的直接缓冲器

- 我们必须制定映射文件的初始位置和映射区域的长度，这意味着我们可以映射某个大文件的较小部分
- MappedByteBuffer 由 ByteBuffer 继承而来，所以具有 ByteBuffer 的所有方法

#### 3、惯用法

```
int length=0x8FFFFFFF
MappedByteBuffer out=
 new RandomAccessFile(<filename>).getChannel()
 .map(FileChannel.MapMode.READ_WRITE, 0, length);
```

#### 4、类似于测试容器性能的测试框架 P564

### 18.10.7. 文件加锁

- 1、JDK 1.4 引入了文件加锁机制，它允许我们同步访问某个作为共享资源的文件

- 不过竞争同一文件的两个线程可能在不同的 Java 虚拟机上
- 一个是 Java 线程，另一个是操作系统中其他的某个本地线程
- **文件锁对其他的操作系统进程是可见的，因为 Java 的文件加锁直接映射到了本地操作系统的加锁工具**

- 2、通过对 FileChannel 调用 tryLock() 或 lock()，就可以获得整个文件的 FileLock

- **tryLock()** 是非阻塞式的，它设法获取锁，但是如果不能获得(其他进程已经持有相同的锁，并且不能共享时)，它将直接从方法调用中返回
- **lock()** 是阻塞式的，它要阻塞进程直至锁可以获得，或调用 lock() 的线程中断，或调用 lock() 的通道关闭，使用 FileLock.release() 可以释放锁

#### 3、可以对文件的一部分上锁

- tryLock(long position, long size, boolean shared)
- lock(long position, long size, boolean shared)

## 18.11. 压缩

- 1、Java I/O 类库中的类支持读写压缩格式的数据流

- 2、这些类属于 InputStream 和 OutputStream 继承层次结构的一部分，因为压缩类库是按字节方式而不是字符方式处理的

| 压缩类                  | 功能                                         |
|----------------------|--------------------------------------------|
| CheckedInputStream   | GetChecksum() 为任何 InputStream 产生校验和        |
| CheckedOutputStream  | GetChecksum() 为任何 OutputStream 产生校验和       |
| DeflaterOutputStream | 压缩类的基类                                     |
| ZipOutputStream      | 一个 DeflaterOutputStream，用于将数据压缩成 Zip 文件格式  |
| GZIPOutputStream     | 一个 DeflaterOutputStream，用于将数据压缩成 GZIP 文件格式 |
| InflaterInputStream  | 解压缩的基类                                     |
| ZipInputStream       | 一个 InflaterInputStream，用于解压缩 Zip 文件格式的数据   |
| GZIPInputStream      | 一个 InflaterInputStream，用于解压缩 GZIP 文件格式的数据  |

3、尽管存在许多种压缩算法，但是 Zip 和 GZIP 可能是最常用的

### 18.11.1. 用 GZIP 进行简单压缩

1、压缩类的使用非常直观--直接将输出流封装成 `GZIPOutputStream` 或 `ZIPOutputStream`，并将这个输入流封装成 `GZIPInputStream` 或 `ZipInputStream` 即可，其他操作就是通常的 I/O 读写

### 18.11.2. 用 Zip 进行多文件保存

1、支持 Zip 格式的 Java 库更加全面，利用该库可以方便地保存多个文件，它甚至有一个独立的类，使得读取 Zip 文件更加方便

2、<未完成>：感觉没兴趣

### 18.11.3. Java 档案文件

1、Zip 格式也被应用于 JAR(Java ARchive，Java 档案文件)文件格式中

- 这种文件格式就像 Zip 一样，可以将一组文件压缩到单个压缩文件中
- 同 Java 中任何东西一样，JAR 文件也是跨平台的

2、JAR 文件非常有用，尤其是在涉及因特网应用的时候

- 如果不采取 JAR 文件，Web 浏览器在下载构成一个应用的所有文件时必须重复多次请求 Web 服务器，而且这些文件都是未经压缩的
- 如果将所有文件合并到一个 JAR 文件中，只需向远程服务器发出一次请求即可
- 同时由于采用了压缩技术，可以使传输时间更短
- 另外，出于安全考虑，JAR 文件中的每个条目都可以加上数字化签名

3、一个 JAR 文件由一组压缩文件构成，同时还有一张描述所有这些文件的“文件清单”

4、Sun 的 JDK 自带的 jar 程序命令参考 P571

## 18.12. 对象序列化

1、如果对象能够在程序不运行的情况下仍能存在并保存信息，那将非常有用

- 这样，在下次运行程序时，该对象将被重建并且拥有的信息与在程序上次运行时它所拥有的信息相同
- 你可以通过将信息写入文件或数据库来达到相同的目的，但是在使万物都成为对象的精神中，如果能够将一个对象声明为持久性的，并为我们处理掉所有细节，那将会显得十分方便

2、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，这个过程甚至能通过网络进行

- 这意味着序列化机制能自动弥补不同操作系统之间的差异
- 我们不必担心数据在不同机器上的表示会不同，不必关心字节的顺序或者其他任何细节

3、对象的序列化是非常有趣的，因为利用它可以实现轻量级持久性(`lightweight persistence`)

- **持久性**意味着一个对象的生命周期并不取决于程序是否正在执行，它可以生存于程序的调用之间
- 之所以称其为**轻量级**，是因为不能用某种“`persistent`”关键字来简单地定义一个对象，并让系统自动维护其细节问题，相反，对象必须在程序中显式序列化和反序列化还原

#### 4、对象序列化概念加入到 Java 语言中的目的

- Java 的远程方法调用(Remote Method Invocation, RMI), 它使存活于其他计算机上的对象使用起来就像是存活于本机上一样, 当向远程对象发送消息时, 需要通过对对象序列化来传输参数和返回值
- 对于 Java Beans 来说, 对象序列化也是必须的???

#### 5、只要对象实现了 Serializable 接口(该接口仅仅是一个标记接口, 不包括任何方法)

- 要序列化一个对象, 首先要创建某些 OutputStream 对象, 然后将其封装在一个 ObjectOutputStream 对象内, 这时, 只要调用 writeObject()即可将对象序列化, 并发送给 OutputStream(对象序列化是基于字节的, 因此要使用 InputStream 和 OutputStream 继承层次结构)
- 要反向进行该过程, 即将一个序列还原为一个对象, 需要将一个 InputStream 封装在 ObjectInputStream 内, 然后调用 readObject(), **获取的是一个引用, 指向一个向上转型的 Object**, 所以必须向下转型才能直接设置它们

6、对象序列化不仅保存了对象的"全景图", 而且能够追踪对象内所包含的所有引用, 并保存那些对象, 接着又能对对象内包含的每个这样的引用进行追踪, 以此类推

### 18.12.1. 寻找类

1、将一个对象从它的序列化状态恢复出来, 有哪些工作是必须的呢?

- 必须保证 Java 虚拟机能找到相关的.class 文件(该对象网中所有类型的 class 文件)

### 18.12.2. 对象序列化的控制

1、也许, 出于安全的考虑, 你不希望对象中的一部分被序列化, 或者一个对象被还原后, 某子对象需要重新被创建, 因此不必将该子对象序列化

- 在这些特殊情况下, 可以通过实现 Externalizable 接口---代替实现 Serializable 接口---来对序列化过程进行控制

2、Externalizable 接口继承了 Serializable 接口, 同时添加了两个方法

- writeExternal()
- readExternal()
- 上述两个方法会在序列化和反序列化过程中被自动调用, 以便执行一些特殊操作

3、Externalizable 与 Serializable 的微小区别:

- 对于 Serializable 对象, 对象完全以它存储的二进制位为基础来构造, 而不用调用构造器
- 对于 Externalizable 对象, 所有的普通构造器都会被调用(包括在字段定义时的初始化), 然后调用 readExternal()

4、如果从一个 Externalizable 对象继承, 通常需要调用基类版本的 writeExternal()和 readExternal()来为基类组件提供恰当的存储和恢复功能

5、为了正常运行, **我们不仅需要在 writeExternal()方法(没有任何默认行为来为 Externalizable 对象写入任何成员对象)中将来自对象的重要信息写入, 还必须在 readExternal()方法中恢复数据**

- 因为 Externalizable 对象的默认构造行为使其看起来似乎像某种自动发生的存储与恢复操作
- 即 Externalizable 默认不保存任何字段

6、transient(瞬时)关键字

- 若子对象表示的是我们不希望将其序列化的敏感信息(如密码), 通常就会发生这种情况

况，即使对象中的这些信息是 `private` 的，一经序列化处理，人们就可以通过读取文件或者拦截网络传输的方式来访问到它

- 一种解决方法是：将类实现为 `Externalizable`，在 `writeExternal()` 内部只对所需部分进行显式序列化
- 另一种解决方案：利用 `transient` 关键字，关闭序列化
- 由于 `Externalizable` 对象默认不保存它们的任何字段，所以 `transient` 关键字只能和 `Serializable` 对象一起使用

## 7、`Externalizable` 的替代方法

- 可以实现 `Serializable` 接口，并添加(而非覆盖或实现)，名为 `writeObject()` 和 `readObject()` 的方法，一旦对象被序列化或者被反序列化还原，就会自动分别调用这两个方法

➤ 也就是说，只要我们提供了这两个方法，就会使用它们而不是默认的序列化机制

- 这两个方法必须拥有准确的方法特征签名

```
private void writeObject(ObjectOutputStream stream) throws IOException
private void readObject(ObjectInputStream stream) throws IOException, ClassNotFoundException
Exception
```

- 我们并没有从这个类的其他方法中调用上述两个方法，而是 **ObjectOutputStream** 和 **ObjectInputStream** 对象的 `writeObject()` 和 `readObject()` 方法调用**你对象的 writeObject() 和 readObject() 方法(由于是 private 的，因此只能是通过反射来实现)**

➤ 在接口中定义的东西都是自动 `public` 的，如果 `writeObject()` 和 `readObject()` 必须是 `private` 的，那么它们不会是接口的一部分，因此，我们必须要完全遵循其方法特征签名，所以其效果就和实现了接口一样

➤ 在调用 `ObjectOutputStream.writeObject()` 时，会检查所传递的 `Serializable` 对象，看看是否实现了自己的 `writeObject()`，如果是这样，那就跳过正常的序列化过程并调用它的 `writeObject()`

➤ 还有另外一个技巧，在 `writeObject()` 内部，可以调用 `defaultWriteObject()` 来进行选择默认的 `writeObject()`，类似的，在 `readObject()` 内部，可以调用 `defaultReadObject()`

- 如果打算用默认机制写入非 `transient` 部分，那么必须调用 `defaultWriteObject()` 作为 `writeObject()` 中的第一个操作，并让 `defaultReadObject()` 作为 `readObject()` 中的第一个操作

➤ 例子说明

```
o.writeObject(sc);
```

- `writeObject()` 方法必须检查 `sc`，判断它是否拥有自己的 `writeObject()` 方法，不是检查接口，因为这里根本没有接口，也不是检查类型，而是利用反射来真正地搜索方法)，如果有就会使用它
- `readObject()` 也是采用了这个方法
- 或许这是解决这个问题唯一可行的方法，但它确实有点古怪

## 18.12.3. 使用“持久性”

1、一个比较诱人的使用序列化技术的想法是：存储程序的一些状态，以便我们随后可以很容易地将程序恢复到当前状态

2、我们可以对任何可 `Serializable` 对象“深度复制”----深度复制意味着我们复制的是整个对象网，而不仅仅是基本对象及其引用

3、只要将任何对象序列化到单一流中，就可以恢复出于我们写入时一样的对象网，并且没有任何意外重复复制出的对象

- 我们可以在写入第一个对象和写入最后一个对象期间改变这些对象的状态，无论对象在被序列化时是什么状态，它们都可以被写出

4、Class 是 Serializable 的，因此只需直接对 Class 对象序列化，就可以很容易地保存 static 字段

#### 18.12.4. XML

1、对象序列化的一个重要限制是它只是 Java 的解决方案：只有 Java 程序才能反序列化这种对象

2、一种更具互操作性的解决方案是将数据转换为 XML 格式，这可以使其被各种各样的平台和语言使用

#### 18.12.5. Preferences

1、Preferences API 与对象序列化相比，前者与对象持久性更密切，因为它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，并且每个字符串的存储长度不能超过 8K

2、Preferences API 用于存储和读取用户偏好以及程序配置项的设置

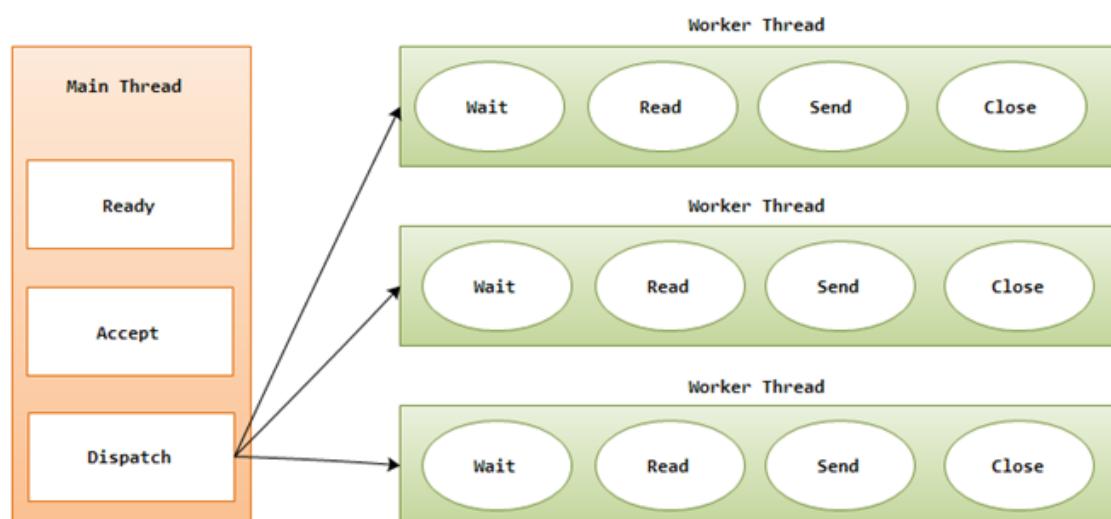
### 18.13. 阻塞/非阻塞 I/O

#### 18.13.1. 阻塞 I/O

1、通常在进行同步 I/O 操作时，如果读取数据，代码会阻塞直至有可供读取的数据。同样，写入调用将会阻塞直至数据能够写入

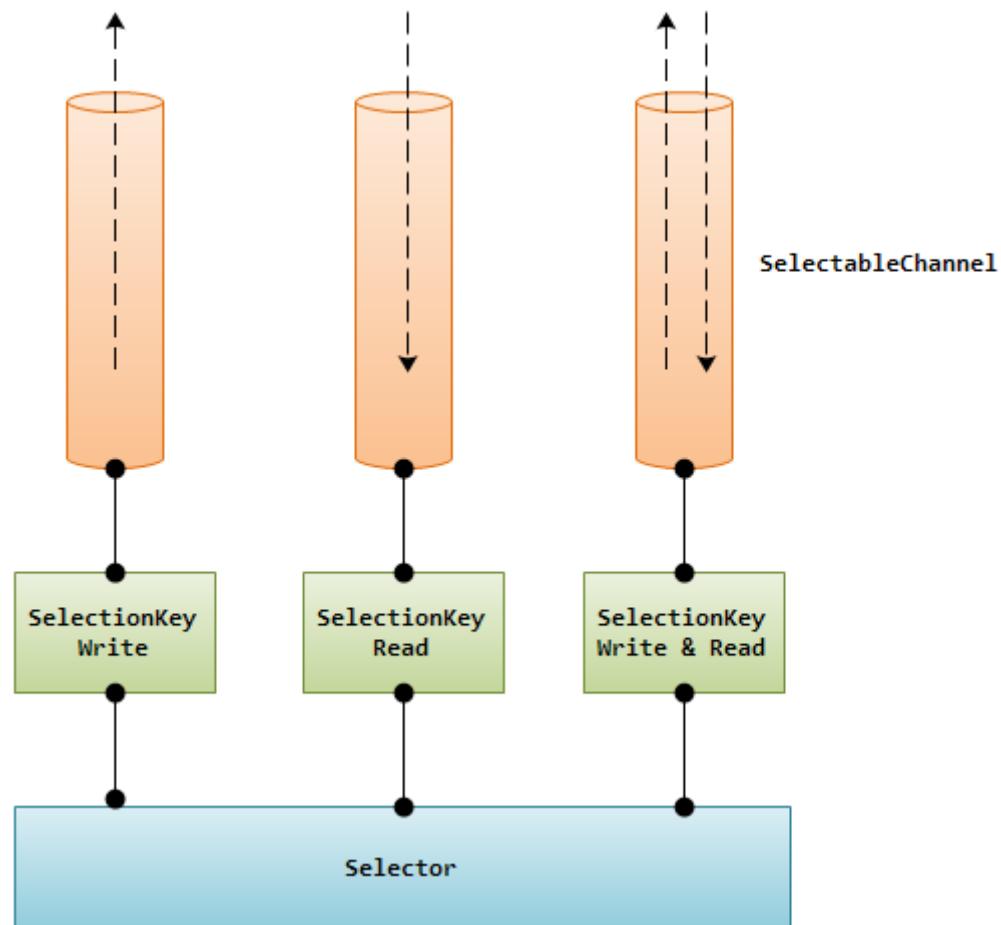
2、传统的 Server/Client 模式会基于 TPR(Thread per Request)，服务器会为每个客户端请求建立一个线程，由该线程单独负责处理一个客户请求。这种模式带来的一个问题就是线程数量的剧增，大量的线程会增大服务器的开销。大多数的实现为了避免这个问题，都采用了线程池模型，并设置线程池线程的最大数量，这由带来了新的问题，如果线程池中有 200 个线程，而有 200 个用户都在进行大文件下载，会导致第 201 个用户的请求无法及时处理，即便第 201 个用户只想请求一个几 KB 大小的页面

3、传统的 Server/Client 模式如下图所示



### 18.13.2. 非阻塞 I/O

1、NIO 中非阻塞 I/O 采用了基于 **Reactor 模式**的工作方式，I/O 调用不会被阻塞，相反是注册感兴趣的特定 I/O 事件，如可读数据到达，新的套接字连接等等，在发生特定事件时，系统再通知我们。NIO 中实现非阻塞 I/O 的核心对象就是 **Selector**，**Selector** 就是注册各种 I/O 事件地方，而且当那些事件发生时，就是这个对象告诉我们所发生的事件，如下图所示



2、从图中可以看出，当有读或写等任何注册的事件发生时，可以从 **Selector** 中获得相应的 **SelectionKey**，同时从 **SelectionKey** 中可以找到发生的事件和该事件所发生的具体的 **SelectableChannel**，以获得客户端发送过来的数据

3、使用 NIO 中非阻塞 I/O 编写服务器处理程序，大体上可以分为下面三个步骤：

- 1) 向 **Selector** 对象注册感兴趣的事件
- 2) 从 **Selector** 中获取感兴趣的事件
- 3) 根据不同的事件进行相应的处理

4、例子

```
/*
 * 注册事件
 */
protected Selector getSelector() throws IOException {
 // 创建 Selector 对象
 Selector sel = Selector.open();

 // 创建可选择通道，并配置为非阻塞模式
}
```

```

 ServerSocketChannel server = ServerSocketChannel.open();
 server.configureBlocking(false);

 // 绑定通道到指定端口
 ServerSocket socket = server.socket();
 InetSocketAddress address = new InetSocketAddress(port);
 socket.bind(address);

 // 向 Selector 中注册感兴趣的事件
 server.register(sel, SelectionKey.OP_ACCEPT);
 return sel;
}

➤ 创建了 ServerSocketChannel 对象，并调用 configureBlocking()方法，配置为非阻塞模式，接下来的三行代码把该通道绑定到指定端口，最后向 Selector 中注册事件，此处指定的是参数是 OP_ACCEPT，即指定我们想要监听 accept 事件，也就是新的连接发生时所产生的事件，对于 ServerSocketChannel 通道来说，我们唯一可以指定的参数就是 OP_ACCEPT

```

```

/*
 * 开始监听
 */
public void listen() {
 System.out.println("listen on " + port);
 try {
 while(true) {
 // 该调用会阻塞，直到至少有一个事件发生
 selector.select();
 Set<SelectionKey> keys = selector.selectedKeys();
 Iterator<SelectionKey> iter = keys.iterator();
 while (iter.hasNext()) {
 SelectionKey key = (SelectionKey) iter.next();
 iter.remove();
 process(key);
 }
 }
 } catch (IOException e) {
 e.printStackTrace();
 }
}

```

➤ 在非阻塞 I/O 中，内部循环模式基本都是遵循这种方式。首先调用 select()方法，该方法会阻塞，直到至少有一个事件发生，然后再使用 selectedKeys()方法获取发生事件的 SelectionKey，再使用迭代器进行循环

```
/*
```

```
* 根据不同的事件做处理
*/
protected void process(SelectionKey key) throws IOException{
 // 接收请求
 if (key.isAcceptable()) {
 ServerSocketChannel server = (ServerSocketChannel) key.channel();
 SocketChannel channel = server.accept();
 channel.configureBlocking(false);
 channel.register(selector, SelectionKey.OP_READ);
 }
 // 读信息
 else if (key.isReadable()) {
 SocketChannel channel = (SocketChannel) key.channel();
 int count = channel.read(buffer);
 if (count > 0) {
 buffer.flip();
 CharBuffer charBuffer = decoder.decode(buffer);
 name = charBuffer.toString();
 SelectionKey sKey = channel.register(selector, SelectionKey.OP_WRITE);
 sKey.attach(name);
 } else {
 channel.close();
 }
 buffer.clear();
 }
 // 写事件
 else if (key.isWritable()) {
 SocketChannel channel = (SocketChannel) key.channel();
 String name = (String) key.attachment();
 ByteBuffer block = encoder.encode(CharBuffer.wrap("Hello " + name));
 if(block != null) {
 channel.write(block);
 }
 else {
 channel.close();
 }
 }
}
```

## Chapter 19. 枚举类型

1、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，这是一种非常有用的功能

### 19.1. 基本 enum 特性

1、基本方法

- values()方法：可以遍历 enum 实例，values()方法返回 enum 实例的数组，而且该数组中的元素严格保持其在 enum 中声明时的顺序
- ordinal()方法：返回一个 int 值，这是每个 enum 实例在声明时的次序，从 0 开始
- getDeclaringClass()方法：获知其所属的 enum 类
- name()方法：返回 enum 实例声明时的名字，这与使用 toString()方法效果相同
- valueOf()方法：根据给定的名字返回 enum 实例，如果不存在给定名字的实例，将会抛出异常

2、**创建 enum 时，编译器会为你生成一个相关的类，这个类继承自 java.lang.Enum**

- Enum 类实现了 Comparable 接口，所以它具有 compareTo()方法，同时还实现了 Serializable 接口

3、enum 中的实例：相当与一个类型中包含了它自身类型的若干个对象

#### 19.1.1. 将静态导入用于 enum

1、使用 static import 能够将 enum 实例标识符代入当前的命名空间，所以无需再用 enum 类型来修饰 enum 实例

- 注意，在定义 enum 的同一个文件中，该技巧无法使用
- 如果在默认包中定义 enum，该技巧也无法使用

### 19.2. 向 enum 中添加方法

1、除了不能继承自一个 enum 之外，我们基本上可以将 enum 看作一个常规的类

- 我们可以向 enum 中添加方法
- 甚至可以含有 main 方法

2、一般来说，我们希望每个枚举实例能够返回对自身的描述，而不仅仅只是默认的 toString()实现，这只能返回枚举实例的名字

3、**如果打算定义自己的方法，那么必须在 enum 实例序列的最后添加一个分号，同时 Java 要求你必须先定义 enum 实例**，否则在编译期会得到错误信息

4、一旦 enum 的定义结束，编译器就不允许我们再使用其构造器来创建任何实例了

- enum 就是一个奇怪的类，因为该类中一般包含了该类的对象(实例)(类似于作为静态字段??)，而且不允许在类外创建该类的实例，即便构造器是 public 的

### 19.3. switch 语句中的 enum

1、在 switch 中使用 enum，是 enum 提供的一项非常便利的功能

- 一般来说 switch 中只能使用整数值，而枚举实例天生具备整数值的次序，并且可以通过 ordinal()方法取得其次序(编译器帮我们做了类似的工作)，因此我们可以在 switch 中使用 enum

ch 语句中使用 enum

- 一般情况下我们必须使用 enum 类型来修饰一个 enum 实例，但是在 case 语句中却不必如此

## 19.4. values() 的神秘之处

- 1、编译器为你创建的 enum 类都继承自 Enum 类，但是 Enum 类并没有 values()方法
  - values()是由编译器添加的 static 方法
  - 另外，编译器还添加了 valueOf()方法
- 2、将 enum 实例向上转型为 Enum，那么 values()方法就不可用了，但是 Class 中有一个 getEnumConstants()方法，所以即便 Enum 接口中没有 values()方法，我们仍然可以通过 Class 对象取得所有 enum 实例
  - 由于 getEnumConstants()是 Class 类的方法，我们甚至可以对不是枚举的类调用此方法，只不过返回的是 null，因此当你试图使用返回结果时会发生异常

## 19.5. 实现而非继承

- 1、由于 enum 继承自 java.lang.Enum 类，由于 Java 不支持多重继承，因此 enum 不能再继承其他类

## 19.6. 随机选取

- 1、利用了古怪的循环(CRG)

```
public static <T extends Enum<T>> T random(Class<T> ec){}
```

- 古怪的语法<T extends Enum<T>>表示 T 是一个 enum 实例
- Class<T>作为参数就可以利用 Class 对象得到 enum 实例的数组了

## 19.7. 使用接口组织枚举

- 1、无法从 enum 继承子类令人沮丧，这种需求有时源自我们希望扩展 enum 中的元素，有时因为我们希望使用子类将一个 enum 中的元素进行分组
- 2、在一个接口的内部，创建实现该接口的枚举，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
- 3、对 enum 而言，实现接口是使其子类化的唯一办法
  - 子类化：将一组 enum 进行分类，分成不同的子类
  - 将这些 enum 都置于同一个 interface 下，即 enum 都实现了该 interface(即便没有方法)，然后这个 interface 便成了这些 enum 的父类，这些 enum 便成了该 interface 的子类

```
interface Food{
 enum Dessert implements Food{A,B,C}
 enum Coffee implements Food{D,E,F}
}
```

## 19.8. 使用 EnumSet 代替标志

- 1、Set 是一种集合，只能向其中添加不重复的对象，当然 enum 也要求其成员都是唯一的，所以 enum 看起来也具有集合的行为
- 2、不过，不能从 enum 中添加或删除元素，所以它只能算是不太有用的集合
- 3、为了通过 enum 创建一种替代品，以替代传统的基于 int 的"标志位"，这种标志可以用来表示某种"开关"信息，不过使用这种标志，我们最终操作的是一些 bit，而不是这些 bit 想要表达的概念，因此很容易写出令人难以理解的代码
- 4、EnumSet 的设计充分考虑到了速度因素，因为它必须与非常高效的 bit 标志相竞争(其操作与 HashSet 比，非常的快)。就其内部而言，它可能就是将一个 long 值作为 bit 向量，所以 EnumSet 非常快速高效
- 5、使用 EnumSet 的优点是：它在说明一个二进制位是否存在时，具有更好的表达能力，并且无需担心性能
- 6、EnumSet 的基础是 long，一个 long 值 64 位，而一个 enum 实例只需一位 bit 表示其是否存在，Enum 会在必要的时候增加一个 long

## 19.9. 使用 EnumMap

- 1、EnumMap 是一种特殊的 Map，它要求其中的键(key)必须来自一个 enum，由于 enum 本身的限制，所以 EnumMap 在内部可由数组实现，因此 EnumMap 速度很快
- 2、与 EnumSet 一样，enum 实例定义时的次序决定了其在 EnumMap 中的顺序
- 3、**enum 的每个实例作为一个键总是存在的，但是如果为这个键调用 put 方法来存入相应的值，其对应的值就是 null，说白了就是个数组**

## 19.10. 常量相关方法

- 1、Java 的 enum 有一个非常有趣的特性，即它允许程序员为 enum 实例编写方法，从而为每个 enum 实例赋予各自不同的行为
- 2、要实现常量方法，你需要为 enum 定义一个或多个 abstract 方法(不是 abstract 也可以，即可以覆盖常量相关的方法)，然后为每个 enum 实例实现该抽象方法
- 3、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

### 19.10.1. 使用 enum 的职责链

- 1、在职责链设计模式中，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，它遍历这个链，直到链中某个解决方案能够处理该请求

### 19.10.2. 使用 enum 的状态机

- 1、枚举类型非常适合用来创建状态机
  - 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，一旦任务执行结束，状态机就会立刻离开瞬时状态
  - 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状态转移到不同的新状态

## 19.11. 多路分发

- 1、当你要处理多种交互类型时，程序可能会变得相当混乱
  - 例如一个系统要分析和执行数学表达式，我们可能会声明 `Number.plus(Number)` 等，其中 `Number` 是各种数字对象的超类，当你声明 `a.plus(b)` 时，你并不知道 `a` 或 `b` 的确切类型，那么如何让它们正确地交互呢
- 2、Java 只支持单路分发
  - 也就是说，如果要执行的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类型
  - **因此，我们必须自己来判定其他类型，从而实现自己的动态绑定**
  - 例如

```
a.compete(b);
public Outcome compete(Item it){return it.eval(this);} //通过 a.compete(b)反过来调用 b.eval(this) 即 b.eval(a)，让动态调用机制先作用于 a 然后作用于 b
```

    - 其中 `a` 和 `b` 的静态类型都是一种超类，但是 `a` 与 `b` 的实际类型均不知道
    - 要判定 `a` 的类型，分发机制会在 `a` 的实际类型的 `compet()` 内部起到分发作用
    - `compet()` 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(this 的静态类型是实际类型吗???)**

### 19.11.1. 使用 enum 分发

- 1、第一次分发是实际的方法调用，第二次方法使用 `switch`，不过这样做是安全的，因为 `enum` 限制了 `switch` 语句的选择分支

### 19.11.2. 使用常量相关的方法

- 1、常量相关方法允许我们为每个 `enum` 实例提供方法的不同实现，这使得常量相关的方法似乎是实现多路分发的完美解决方案，**不过 enum 实例虽然有不同的行为，但他们仍然不是类型**
- 2、最好的办法是将 `enum` 用在 `switch` 语句中，来作为第二次分发，`enum` 会限制 `switch` 语句的选择分支

### 19.11.3. 使用 EnumMap 分发

- 1、使用 `EnumMap` 能够实现真正的两路分发
- 2、感觉就是把所有可能的结果都存到了 `Map` 中，2 路并发就是一个 2 维的 `Map`

### 19.11.4. 使用二维数组

- 1、可以进一步简化两路分发的解决方案，我们可以使用二维数组，将竞争者映射到竞争结果，采用这种方式能够获得最简洁最直接的解决方案

## Chapter 20. 注解

- 1、注解(也被称为元数据)为我们在代码中添加信息提供了一种形式化的方法，使我们可以在稍后的某个时刻非常方便地使用这些数据
- 2、注解在一定程度上是在把元数据与源代码文件结合在一起，而不是保存在外部文档中这一大的趋势之下所催生的，同时，注解也是来自像 C# 之类的其他语言对 Java 造成语言特性压力所作出的一种回应
- 3、注解使得我们能够以将由编译器来测试和验证的格式，存储有关程序的额外信息
- 4、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
- 5、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之中的方法
  - `@Deprecated`，如果程序员使用了注解为它的元素，那么编译器会发出警告信息
  - `@SuppressWarnings`，关闭不当的编译器警告信息
- 6、注解是真正语言级的概念，一旦构造出来，就享有编译期的类型检查保护，注解是在实际的源代码级别保存所有的信息，而不是某种注释性的文字

### 20.1. 基本语法

- 1、被注解的方法与其他方法没有区别，注解可以与任何修饰符共同作用于方法，从语法角度来看，注解的使用方式几乎与修饰符的使用一模一样

#### 20.1.1. 定义注解

- 1、注解的定义看起来很像接口的定义，事实上，与其他任何 Java 接口一样，注解也将会编译成 `class` 文件
- 2、定义注解时，会需要一些元注解，如 `@Target` 和 `@Retention`
  - `@Target` 用来定义你的注解将应用于什么地方(例如是一个方法或者域)
  - `@Retention` 用来定义该注解在哪一个级别可用，在源代码中(SOURCE)、类文件中(CLASS)或者运行时(RUNTIME)
- 3、在注解中，一般都会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。
  - 注解的元素看起来就像接口的方法，唯一的区别是你可以为其指定默认值
  - 没有元素的注解称为标记注解(marker annotation)
- 4、注解元素在使用时表现为名-值对的形式

#### 20.1.2. 元注解

- 1、Java 目前只内置了三种标准注解，以及四种元注解

- 元注解专职负责注解其他的注解
- `@Target`: 表示该注解可以用于什么地方
  - `CONSTRUCTOR`: 构造器的声明
  - `FIELD`: 域声明(包括 `enum` 实例)
  - `LOCAL_VARIABLE`: 局部变量声明
  - `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存该注解信息
- SOURCE: 注解将被编译器丢弃
- CLASS: 注解在 class 文件中可用，在 VM 中被丢弃
- 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

2、**大多数时候，程序员主要定义自己的注解，并编写自己的处理器来处理它们**

## 20.2. 编写注解处理器

- 1、在使用注解的过程中，很重要的一个部分就是创建与使用注解处理器
- 2、Java SE5 扩展了反射机制的 API，以帮助程序员构造这类工具，同时它还提供一个外部工具 apt 帮助程序员解析带有注解的 Java 源代码
- 3、以下两个反射方法都属于 AnnotatedElement 接口(Class、Method、Field 等类都实现了该接口)
  - getDeclaredMethods()
  - getAnnotation(): 返回指定类型的**注解对象**，如果被注解的方法上没有该类型的注解，则返回 null

### 20.2.1. 注解元素

- 1、注解元素可用如下类型表示
  - 所有基本类型
  - String
  - Class
  - enum
  - Annotation
  - 以上类型的数组
  - **另外注解也可以作为元素的类型，也就是说注解可以被嵌套**
  - 使用其他类型，编译器就会报错，也不允许使用任何包装类型

### 20.2.2. 默认值限制

- 1、元素不能有不确定的值，也就是说，元素必须要么具有默认值，要么在使用注解时提供元素的值
- 2、对于非基本类型的元素，无论在源代码中声明时，或是在注解接口中定义默认值时，都不能以 null 作为其值
- 3、这个约束使得处理器很难表现一个元素的存在或缺失的状态，因为每个注解的声明中，所有的元素都存在，并且都具有相应的值，为了绕开这个约束，只能定义一些特殊的值，例如空字符串或者负数

### 20.2.3. 生成外部文件

- 1、程序员使用外部的描述文件时，它就拥有了同一个类的两个单独的信息员，这经常导致代码同步的问题，同时它也要求项目工作的程序员，必须同时知道如何编写 Java 程序，以及如何编辑描述文件

2、<未完成>：理解吧

#### 20.2.4. 注解不支持继承

1、不能使用关键字 `extends` 来继承某个`@interface`

#### 20.2.5. 实现注解处理器

1、<未完成>

### 20.3. 使用 apt 处理注解

1、注解处理工具 `apt`，这是 Sun 为了帮助注解的处理过程而提供的工具

2、与 `javac` 一样，`apt` 被设计为操作 Java 源文件，而不是编译后的类

- 默认情况下，`apt` 会在处理完源文件后编译它们
- 如果在系统构建的过程中会自动创建一些新的源文件，那么这个特性非常有用
- 3、当注解处理器生成一个新的源文件时，该文件会在新一轮的注解处理中接受检查，直到不再有新的源文件生成位置，然后他再编译所有的源文件
- 4、程序员自定义的每一个注解都需要自己的处理器，而 `apt` 工具能够很容易地将多个注解处理器组合在一起
  - 程序员可以指定多个要处理的类，这比程序员自己遍历所有类文件简单
  - 还可以添加监听器，并在一轮注解处理过程结束的时候收到通知信息

5、**使用 `apt` 生成注解处理器时，我们无法利用 Java 的反射机制，因为我们操作的是源码，而不是编译后的类，使用 mirror API 能够解决这个问题**

### 20.4. 将观察者模式用于 apt

1、当有更多的注解和更多的处理器时，为了防止这种复杂性迅速攀升，`mirror API` 提供了对访问者设计模式的支持

2、一个访问者会遍历某个数据结构或一个对象的集合，对其中的每一个对象执行一个操作，该数据结构无需有序，而你对每个对象执行的操作，都是**特定于此对象的类型**，这就**将操作与对象解耦**，也就是说，你可以添加新的操作，而无需向类的定义中添加方法

- Java 类可以看作是一系列对象的集合，例如 `TypeDeclaration` 对象，`FieldDeclaration` 对象以及 `MethodDeclaration` 对象等
- 当你配合访问者模式使用 `apt` 工具时，需要提供一个 `Visitor` 类，它具有一个能够处理你要访问的各种声明的方法，然后你就可以为方法、类以及域上的注解实现相应的处理行为

### 20.5. 基于注解的单元测试

1、单元测试是对类中的每个方法提供一个或多个测试的一种时间，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，最著名的单元测试工具就是 `JUnit`
- 使用注解之前的 `JUnit`，程序员必须创建一个独立的类来保存其单元测试，有了注解，我们可以直接在要验证的类里面编写测试，这将大大减少单元测试所需的时间和麻烦之处

2、这个基于注解的测试框架叫做@Unit，其最基本的测试形式，可能也是最常用的一个注解是@Test

- 我们用@Test 来标记测试方法，测试方法不带参数，并返回 boolean 结果来说明测试成功或失败
- 程序员可以任意命名他的测试方法
- 同时@Unit 测试方法可以是任意你喜欢的访问修饰符方式，包括 private
- 例子

**import net.mindview.atunit.\*;//提供了@Test 注解**

```
import net.mindview.util.*;//提供了 OSExecute 类库，不过这不是必须的
public class AtUnitExample1 {
 public String methodOne(){
 return "This is methodOne";
 }
 public int methodTwo(){
 System.out.println("This is methodTwo");
 return 2;
 }
 @Test boolean methodOneTest(){
 return methodOne().equals("This is methodOne");
 }
 @Test boolean m2(){ return methodTwo()==2;}
 @Test private boolean m3(){return true;}
 @Test boolean failureTest() { return false;}
 @Test boolean anotherDisappointment() { return false;}
 public static void main(String[] args) throws Exception{
 OSExecute.command("java net/mindview/atunit/AtUnit chapter20/AtUnitExam
ple1");
 }
}
```

- 直接把测试代码(@Test 注解标记的这些)添加到当前类中
- 如何执行?

- 目前在 eclipse 下无法执行，因为 eclipse 平台下，当前路径是工程路径而不是 class 文件的根节点，因此直接在 eclipse 平台下是无法跑通的，即便 eclipse 可以找到所有的 class 文件，但是该命令(绿色部分)却是无法找到指定的 class 文件的，因此是不可行的
- 打开终端，进入该工程，再进入 bin 目录(cd bin)
- 然后执行 java chapter20/AtUnitExample1
- 或者执行 java net/mindview/atunit/AtUnit chapter20/AtUnitExample1

3、我们并非必须将测试方法嵌入到原本的类中，因为有时候根本做不到，要生成一个非嵌入式的测试，最简单的办法就是继承

```
public class AtUnitExternalTest extends AtUnitExample1{
 @Test boolean _methodOne(){
 return methodOne().equals("This is methodOne");
 }
}
```

```

 @Test boolean _methodTwo(){ return methodTwo()==2;}
 public static void main(String[] args){
 OSExecute.command("java net.mindview.atunit.AtUnit
 chapter20/AtUnitExternalTest");
 }
}

```

#### 4、或者可以采用组合的形式创建非嵌入式的测试

```

public class AtUnitComposition {
 AtUnitExample1 testObject=new AtUnitExample1();
 @Test boolean _methodOne(){
 return testObject.methodOne().equals("This is methodOne");
 }
 @Test boolean _methodTwo(){
 return testObject.methodTwo()==2;
 }
 public static void main(String[] args) throws Exception{
 OSExecute.command("java net.mindview.atunit.AtUnit
 chapter20/AtUnitComposition");
 }
}

```

#### 5、`@Test` 方法允许程序返回 `void`，这是`@Test` 方法的第二种形式

- 在这种情况下，要想表示测试成功，可以使用 Java 的 `assert` 语句
- Java 的断言机制一般要求程序员在 `java` 命令行中加上`-ea` 标志，不过`@Unit` 已经自动打开了该功能
- 表示失败的话，甚至可以使用异常
- 一个失败的 `assert` 或从测试方法中抛出异常，都将被看做是一个失败的测试，但是 `@Unit` 并不会就在这个失败的测试上打住，它会继续运行，直到所有的测试都运行完毕
- <assert>介绍

##### **assert <boolean 表达式>**

- 如果<boolean 表达式>为 `true`，则程序继续执行
- 如果为 `false`，则程序抛出 `AssertionError`，并终止执行

##### **assert <boolean 表达式> : <错误信息表达式>**

- 如果<boolean 表达式>为 `true`，则程序继续执行
- 如果为 `false`，则程序抛出 `java.lang.AssertionError`，并输入<错误信息表达式>

```

public class AtUnitExample2 {
 public String methodOne(){
 return "This is methodOne";
 }
 public int methodTwo(){
 System.out.println("This is methodTwo");
 return 2;
 }
 @Test void assertExample(){

```

```

 assert methodOne().equals("This is methodOne");
 }
 @Test void assertFailureExample(){
 assert 1==2;"What a surprise!";
 }
 @Test void exceptionExample() throws IOException{
 new FileInputStream("nofile.txt");
 }
 @Test boolean assertAndReturn(){
 assert methodTwo()==2:"methodTwo must equal 2";
 return methodOne().equals("This is methodOne");
 }
 public static void main(String[] args) throws Exception{
 OSExecute.command("java net.mindview.atunit.AtUnit chapter20/AtUnitExampl
e2");
 }
}

```

6、对于每一个单元测试而言，`@Unit` 都会用默认的构造器，为该测试所属的类创建一个新的实例，并在此新创建的对象上运行测试，然后丢弃该对象，以避免对其他测试产生副作用

- 如果没有默认构造器，或者新对象需要复杂的构造过程，那么可以创建一个 `static` 方法专门负责构造对象，然后用`@TestObjectCreate` 注解将其标记出来
- 加入了`@TestObjectCreate` 注解的方法必须声明为 `static`，并且返回一个你正在测试的类型的对象，这一切都由`@Unit` 负责确保成立

```

public class AtUnitExample3 {
 private int n;
 public AtUnitExample3(int n){this.n=n;}
 public int getN(){ return n;}
 public String methodOne(){
 return "This is methodOne";
 }
 public int methodTwo(){
 System.out.println("This is methodTwo");
 return 2;
 }
 @TestObjectCreate static AtUnitExample3 create(){
 return new AtUnitExample3(47);
 }
 @Test boolean initialization(){return n==47;}
 @Test boolean methodOneTest(){
 return methodOne().equals("This is methodOne");
 }
 @Test boolean m2(){return methodTwo()==2;}
 public static void main(String[] args) throws Exception{

```

```
 OSExecute.command("java net.mindview.atunit.AtUnit
chapter20/AtUnitExample3");
}
}
```

7、有时候，我们需要向单元测试中添加一些额外的域

- 这时可以用 @TestProperty 注解，由它注解的域表示只在单元测试中使用(因此，在将产品发布给客户之前，它们应该被删除掉)

8、如果你的测试对象要执行某些初始化工作，并且使用完毕后还需要进行某些清理工作，那么可以选择使用 static @TestObjectCleanup 方法，当测试对象使用结束后，该方法会为你执行清理工作

### 20.5.1. 将@Unit 用于泛型

1、泛型为 @Unit 出了一个难题，因为必须针对某个特定类型的参数或参数集才能进行测试

- 解决办法：让测试类继承自泛型的一个特定版本即可

### 20.5.2. 不需要任何"套件"

1、与 JUnit 相比，@Unit 有一个比较大的优点，就是 @Unit 不需要任何套件(suites)

- 在 JUnit 中，程序员必须告诉测试工具你打算测试什么，这就要用套件来组织测试，以便 JUnit 能够找到它们，并运行其中包含的测试
- @Unit 只是简单地搜索类文件，检查其是否具有恰当的注解，然后运行 @Test 方法

## 20.6. 深入理解 Java：注解(Annotation)基本概念

### 20.6.1. 什么是注解

1、Annotation(注解)就是 Java 提供了一种元程序中的元素关联任何信息和着任何元数据(metadata)的途径和方法。Annotation(注解)是一个接口，程序可以通过反射来获取指定程序元素的 Annotation 对象，然后通过 Annotation 对象来获取注解里面的元数据

2、Annotation(注解)是 JDK5.0 及以后版本引入的。它可以用于创建文档，跟踪代码中的依赖性，甚至执行基本编译时检查。从某些方面看，annotation 就像修饰符一样被使用，并应用于包、类型、构造方法、方法、成员变量、参数、本地变量的声明中。这些信息被存储在 Annotation 的 "name=value" 结构对中

3、Annotation 的成员在 Annotation 类型中以 无参数的方法的形式 被声明。其方法名和返回值定义了该成员的名字和类型。在此有一个特定的默认语法：允许声明任何 Annotation 成员的默认值：一个 Annotation 可以将 name=value 对作为没有定义默认值的 Annotation 成员的值，当然也可以使用 name=value 对来覆盖其它成员默认值。这一点有些近似类的继承特性，父类的构造函数可以作为子类的默认构造函数，但是也可以被子类覆盖

4、Annotation 能被用来为某个程序元素(类、方法、成员变量等)关联任何的信息。需要注意的是，这里存在着一个基本的规则：Annotation 不能影响程序代码的执行，无论增加、删除 Annotation，代码都始终如一的执行。另外，尽管一些 annotation 通过 java 的反射 api 方法在运行时被访问，而 java 语言解释器在工作时忽略了这些 annotation。正是由于 java 虚拟机忽略了 Annotation，导致了 annotation 类型在代码中是“不起作用”的；只有通过某种配套的工具才会对 annotation 类型中的信息进行访问和处理。本文中将涵盖标准的 Annotation 和 meta-annotation 类型，陪伴这些 annotation 类型的工具是 java 编译器(当然要以某种特殊的

方式处理它们)

### 20.6.2. 什么是 metadata(元数据)

- 1、元数据从 metadata 一词译来，就是"关于数据的数据"的意思。
- 2、元数据的功能作用有很多，比如：你可能用过 Javadoc 的注释自动生成文档。这就是元数据功能的一种。总的来说，元数据可以用来创建文档，跟踪代码的依赖性，执行编译时格式检查，代替已有的配置文件。如果要对于元数据的作用进行分类，目前还没有明确的定义，不过我们可以根据它所起的作用，大致可分为三类：
  - 1) 编写文档：通过代码里标识的元数据生成文档
  - 2) 代码分析：通过代码里标识的元数据对代码进行分析
  - 3) 编译检查：通过代码里标识的元数据让编译器能实现基本的编译检查
- 3、在 Java 中元数据以标签的形式存在于 Java 代码中，元数据标签的存在并不影响程序代码的编译和执行，它只是被用来生成其它的文件或在运行时知道被运行代码的描述信息。
- 4、综上所述
  - 1) 元数据以标签的形式存在于 Java 代码中
  - 2) 元数据描述的信息是类型安全的，即元数据内部的字段都是有明确类型的
  - 3) 元数据需要编译器之外的工具额外的处理用来生成其它的程序部件
  - 4) 元数据可以只存在于 Java 源代码级别，也可以存在于编译之后的 Class 文件内部。

### 20.6.3. Annotation 和 Annotation 类型

- 1、Annotation：Annotation 使用了在 java5.0 所带来的新语法，它的行为十分类似 public、final 这样的修饰符。每个 Annotation 具有一个名字和成员个数 $\geq 0$ 。每个 Annotation 的成员具有被称为 name=value 对的名字和值(就像 javabean 一样)，name=value 装载了 Annotation 的信息
- 2、Annotation 类型：Annotation 类型定义了 Annotation 的名字、类型、成员默认值。一个 Annotation 类型可以说是一个特殊的 java 接口，它的成员变量是受限制的，而声明 Annotation 类型时需要使用新语法。当我们通过 java 反射 api 访问 Annotation 时，返回值将是一个实现了该 annotation 类型接口的对象，通过访问这个对象我们能方便的访问到其 Annotation 成员

### 20.6.4. 注解的分类

- 1、根据注解参数的个数，我们可以将注解分为三类
  - 1) 标记注解：一个没有成员定义的 Annotation 类型被称为标记注解。这种 Annotation 类型仅使用自身的存在与否来为我们提供信息。比如后面的系统注解@Override
  - 2) 单值注解
  - 3) 完整注解
- 2、根据注解使用方法和用途，我们可以将 Annotation 分为三类
  - 1) JDK 内置系统注解
  - 2) 元注解
  - 3) 自定义注解

### 20.6.5. 系统内置标准注解

- 1、注解的语法比较简单，除了@符号的使用外，他基本与 Java 固有的语法一致，JavaSE 中内置三个标准注解，定义在 java.lang 中：

- 1) **@Override**: 用于修饰此方法覆盖了父类的方法
- 2) **@Deprecated**: 用于修饰已经过时的方法
- 3) **@SuppressWarnnings**: 用于通知 java 编译器禁止特定的编译警告

#### 20.6.5.1. **Override**, 限定重写父类方法

1、**@Override** 是一个标记注解类型，它被用作标注方法。它说明了被标注的方法重载了父类的方法，起到了断言的作用。如果我们使用了这种 **Annotation** 在一个没有覆盖父类方法的方法时，java 编译器将以一个编译错误来警示。这个 **annotation** 常常在我们试图覆盖父类方法而确又写错了方法名时发挥威力。使用方法极其简单：在使用此 **annotation** 时只要在被修饰的方法前面加上**@Override** 即可

#### 20.6.5.2. **@Deprecated**, 标记已过时

- 1、同样 **Deprecated** 也是一个标记注解。当一个类型或者类型成员使用**@Deprecated** 修饰的话，编译器将不鼓励使用这个被标注的程序元素。而且这种修饰具有一定的“延续性”：如果我们在代码中通过继承或者覆盖的方式使用了这个过时的类型或者成员，虽然继承或者覆盖后的类型或者成员并不是被声明为**@Deprecated**，但编译器仍然要报警。
- 2、值得注意，**@Deprecated** 这个 annotation 类型和 javadoc 中的**@deprecated** 这个 tag 是有区别的：前者是 java 编译器识别的，而后者是被 javadoc 工具所识别用来生成文档(包含程序成员为什么已经过时、它应当如何被禁止或者替代的描述)
- 3、在 java5.0，java 编译器仍然象其从前版本那样寻找**@deprecated** 这个 javadoc tag，并使用它们产生警告信息。但是这种状况将在后续版本中改变，我们现在就开始使用**@Deprecated** 来修饰过时的方法而不是**@deprecated** javadoc tag。

#### 20.6.5.3. **SuppressWarnings**, 抑制编译器警告

- 1、**@SuppressWarnings** 被用于有选择的关闭编译器对类、方法、成员变量、变量初始化的警告。在 java5.0，sun 提供的 javac 编译器为我们提供了-Xlint 选项来使编译器对合法的程序代码提出警告，此种警告从某种程度上代表了程序错误。例如当我们使用一个 generic collection 类而又没有提供它的类型时，编译器将提示出“unchecked warning”的警告。通常当这种情况发生时，我们就需要查找引起警告的代码。如果它真的表示错误，我们就需要纠正它。例如如果警告信息表明我们代码中的 switch 语句没有覆盖所有可能的 case，那么我们就应增加一个默认的 case 来避免这种警告。
- 2、有时我们无法避免这种警告，例如，我们使用必须和非 generic 的旧代码交互的 generic collection 类时，我们不能避免这个 unchecked warning。此时**@SuppressWarnings** 就要派上用场了，在调用的方法前增加**@SuppressWarnings** 修饰，告诉编译器停止对此方法的警告。
- 3、**SuppressWarnings** 不是一个标记注解。它有一个类型为 String[] 的成员，这个成员的值为被禁止的警告名。对于 javac 编译器来讲，被-Xlint 选项有效的警告名也同样对**@SuppressWarnings** 有效，同时编译器忽略掉无法识别的警告名。
- 4、**SuppressWarnings** 注解的常见参数值的简单说明：

- 1) **deprecation**: 使用了不赞成使用的类或方法时的警告
- 2) **unchecked**: 执行了未检查的转换时的警告，例如当使用集合时没有用泛型 (Generics) 来指定集合保存的类型
- 3) **fallthrough**: 当 Switch 程序块直接通往下一种情况而没有 Break 时的警告
- 4) **path**: 在类路径、源文件路径等中有不存在的路径时的警告

- 5) `serial`: 当在可序列化的类上缺少 `serialVersionUID` 定义时的警告
- 6) `finally`: 任何 `finally` 子句不能正常完成时的警告
- 7) `all`: 关于以上所有情况的警告

## 20.7. 深入理解 Java: 注解(Annotation) 自定义注解入门

### 20.7.1. 元注解

1、元注解的作用就是负责注解其他注解。Java5.0 定义了 4 个标准的 meta-annotation 类型，它们被用来提供对其它 annotation 类型作说明。Java5.0 定义的元注解：

- 1) `@Target`
- 2) `@Retention`
- 3) `@Documented`
- 4) `@Inherited`

#### 2、`@Target`

- `@Target` 说明了 Annotation 所修饰的对象范围：Annotation 可被用于 packages、types(类、接口、枚举、Annotation 类型)、类型成员(方法、构造方法、成员变量、枚举值)、方法参数和本地变量(如循环变量、catch 参数)。在 Annotation 类型的声明中使用了 target 可更加明晰其修饰的目标
- 作用：用于描述注解的使用范围(即：被描述的注解可以用在什么地方)
- 取值(ElementType)有
  - 1) `CONSTRUCTOR`: 用于描述构造器
  - 2) `FIELD`: 用于描述域
  - 3) `LOCAL_VARIABLE`: 用于描述局部变量
  - 4) `METHOD`: 用于描述方法
  - 5) `PACKAGE`: 用于描述包
  - 6) `PARAMETER`: 用于描述参数
  - 7) `TYPE`: 用于描述类、接口(包括注解类型)或 enum 声明

#### 3、`@Retention`

- `@Retention` 定义了该 Annotation 被保留的时间长短：某些 Annotation 仅出现在源代码中，而被编译器丢弃；而另一些却被编译在 class 文件中；编译在 class 文件中的 Annotation 可能会被虚拟机忽略，而另一些在 class 被装载时将被读取(请注意并不影响 class 的执行，因为 Annotation 与 class 在使用上是被分离的)。使用这个 meta-Annotation 可以对 Annotation 的“生命周期”限制
- 作用：表示需要在什么级别保存该注释信息，用于描述注解的生命周期(即：被描述的注解在什么范围内有效)
- 取值(RetentionPolicy)有
  - 1) `SOURCE`: 在源文件中有效(即源文件保留)
  - 2) `CLASS`: 在 class 文件中有效(即 class 保留)
  - 3) `RUNTIME`: 在运行时有效(即运行时保留)
- Retention meta-annotation 类型有唯一的 value 作为成员，它的取值来自 `java.lang.annotation.RetentionPolicy` 的枚举类型值

#### 4、`@Documented`

- `@Documented` 用于描述其它类型的 annotation 应该被作为被标注的程序成员的公共 API，因此可以被例如 javadoc 此类的工具文档化。`Documented` 是一个标记注解，没有成员

#### 5、`@Inherited`

- `@Inherited` 元注解是一个标记注解，`@Inherited` 阐述了某个被标注的类型是被继承的。如果一个使用了`@Inherited` 修饰的 annotation 类型被用于一个 class，则这个 annotation 将被用于该 class 的子类
- 注意：`@Inherited annotation` 类型是被标注过的 class 的子类所继承。类并不从它所实现的接口继承 annotation，方法并不从它所重载的方法继承 annotation
- 当`@Inherited annotation` 类型标注的 annotation 的 Retention 是 `RetentionPolicy.RUNTIME`，则反射 API 增强了这种继承性。如果我们使用 `java.lang.reflect` 去查询一个`@Inherited annotation` 类型的 annotation 时，反射代码检查将展开工作：检查 class 和其父类，直到发现指定的 annotation 类型被发现，或者到达类继承结构的顶层

### 20.7.2. 自定义注解

1、使用`@interface` 自定义注解时，自动继承了 `java.lang.annotation.Annotation` 接口，由编译程序自动完成其他细节。在定义注解时，不能继承其他的注解或接口。`@interface` 用来声明一个注解，其中的每一个方法实际上是声明了一个配置参数。**方法的名称就是参数的名称，返回值类型就是参数的类型**(返回值类型只能是基本类型、Class、String、enum)。可以通过 `default` 来声明参数的默认值

#### 2、定义注解格式

```
public @interface 注解名 {定义体}
```

#### 3、注解参数的可支持数据类型

- 1) 所有基本数据类型(int,float,boolean,byte,double,char,long,short)
- 2) String 类型
- 3) Class 类型
- 4) enum 类型
- 5) Annotation 类型
- 6) 以上所有类型的数组

#### 4、Annotation 类型里面的参数设定规则

- 1) 只能用 `public` 或默认(`default`)这两个访问权修饰。例如：`String value();`这里把方法设为 `default` 默认类型
- 2) 参数成员只能用基本类型 byte,short,char,int,long,float,double,boolean 八种基本数据类型和 String,Enum,Class,annotations 等数据类型，以及这些类型的数组。例如：`String value();`这里的参数成员就为 String
- 3) 如果只有一个参数成员，最好把参数名称设为"value"，后加小括号.例:下面的例子 `FruitName` 注解就只有一个参数成员

#### 5、注解元素的默认值

- 注解元素必须有确定的值，要么在定义注解的默认值中指定，要么在使用注解时指定，非基本类型的注解元素的值不可为 `null`
- 因此，使用空字符串或 0 作为默认值是一种常用的做法。这个约束使得处理器很难表现一个元素的存在或缺失的状态，因为每个注解的声明中，所有元素都存在，并且都具有相应的值，为了绕开这个约束，我们只能定义一些特殊的值，例如空字符串或者负数，一次表示某个元素不存在，在定义注解时，这已经成为一个习惯用法

## 20.8. 深入理解 Java：注解(Annotation)——注解处理器

1、如果没有用来读取注解的方法和工作，那么注解也就不会比注释更有用处了。使用注解的过程中，很重要的一部分就是创建于使用注解处理器。Java SE5 扩展了反射机制的 API，以帮助程序员快速的构造自定义注解处理器

### 20.8.1. 注解处理器类库(java.lang.reflect.AnnotatedElement)

1、Java 使用 Annotation 接口来代表程序元素前面的注解，该接口是所有 Annotation 类型的父接口。除此之外，Java 在 java.lang.reflect 包下新增了 AnnotatedElement 接口，该接口代表程序中可以接受注解的程序元素，该接口主要有如下几个实现类

- 1) Class: 类定义
- 2) Constructor: 构造器定义
- 3) Field: 累的成员变量定义
- 4) Method: 类的方法定义
- 5) Package: 类的包定义

2、java.lang.reflect 包下主要包含一些实现反射功能的工具类，实际上，java.lang.reflect 包所有提供的反射 API 扩充了读取运行时 Annotation 信息的能力。**当一个 Annotation 类型被定义为运行时的 Annotation 后，该注解才能是运行时可见**，当 class 文件被装载时被保存在 class 文件中的 Annotation 才会被虚拟机读取

3、AnnotatedElement 接口是所有程序元素(Class、Method 和 Constructor)的父接口，所以程序通过反射获取了某个类的 AnnotatedElement 对象之后，程序就可以调用该对象的如下四个方法来访问 Annotation 信息：

- 1) <T extends Annotation> T getAnnotation(Class<T> annotationClass): 返回改程序元素上存在的、指定类型的注解，如果该类型注解不存在，则返回 null
- 2) Annotation[] getAnnotations(): 返回该程序元素上存在的所有注解
- 3) boolean isAnnotationPresent(Class<?extends Annotation> annotationClass): 判断该程序元素上是否包含指定类型的注解，存在则返回 true，否则返回 false
- 4) Annotation[] getDeclaredAnnotations(): 返回直接存在于此元素上的所有注释。与此接口中的其他方法不同，该方法将忽略继承的注释。(如果没有注释直接存在于此元素上，则返回长度为零的一个数组)该方法的调用者可以随意修改返回的数组；这不会对其他调用者返回的数组产生任何影响

## 20.9. 深入理解 Java：注解(Annotation)示例

### 20.9.1. 注解定义

1、MyAnnotation1.java

```
package hcf;

import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited
```

```

public @interface MyAnnotation1{
 public String name() default "";
 public int age() default -1;
 public boolean sex() default true;
}

2、MyAnnotation2.java
package hcf;

import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
@Inherited

public @interface MyAnnotation2{
 public MyAnnotation1 value() default
 @MyAnnotation1(name="love",age=100000,sex=true);
}

```

## 20.9.2. 注解处理器定义

```

1、UseAnnotation.java
package hcf;

import java.lang.reflect.*;

public class UseAnnotation{
 @MyAnnotation1(name="hcf",age=15,sex=true)
 public int i;

 @MyAnnotation2(@MyAnnotation1(name="lh",age=25,sex=false))
 public int j;

 public static void compile(Class<?> c){
 Field[] fields=c.getDeclaredFields();
 for(Field field:fields){
 if(field.isAnnotationPresent(MyAnnotation1.class)){
 MyAnnotation1 annotation=field.getAnnotation(MyAnnotation1.class);
 System.out.print("name: "+annotation.name());
 System.out.print(" age: "+annotation.age());
 System.out.println(" sex: "+annotation.sex());
 }
 if(field.isAnnotationPresent(MyAnnotation2.class)){
 MyAnnotation2 annotation=field.getAnnotation(MyAnnotation2.class);
 }
 }
 }
}

```

```
 System.out.print("name: "+annotation.value().name());
 System.out.print(" age: "+annotation.value().age());
 System.out.println(" sex: "+annotation.value().sex());
 }
}
}

public static void main(String[] args){
 compile(UseAnnotation.class);
}
}
```

## Chapter 21. 并发

1、Java 是一种多线程语言，并且提出了并发问题

### 21.1. 并发的多面性

- 1、并发编程令人困惑的主要原因：使用并发时需要解决的问题有多个，而实现并发的方式也有多重，并且在这两者之间没有明显的映射关系。
- 2、用并发解决的问题大体上可以分为“速度”和“设计可管理性”两种

#### 21.1.1. 更快的执行

1、**并发通常是提高运行在单处理器上的程序的性能**

- 这听起来违背直觉，在单处理器上运行的并发程序开销确实比程序的所有部分顺序执行的开销大，因为增加了上下文切换的代价
- 阻塞：如果程序中某个任务因为该程序控制范围之外的某些条件(通常是 I/O)而导致不能继续执行，那么我们就说这个任务或线程阻塞了
- 如果使用并发来编写程序，那么当一个任务阻塞时，其他任务还能继续执行
- **从性能的角度看，如果没有任务会阻塞，那么在单处理器上使用并发就没有任何意义**

2、在单处理器系统中的性能提高的常见事例是**事件驱动的编程**

- 使并发最吸引人的一个原因就是要产生具有可响应的用户界面(事件驱动，不用周期性地判断)
- 通过创建单独的执行线程来响应用户的输入，即使这个线程在大多数时间里都是阻塞的，但是程序可以保证具有一定度的可响应性

3、**实现并发最直接的方式就是在操作系统级别使用进程。**

- 进程是运行在它自己的地址空间内的自包容的程序
- 多任务操作系统可以通过周期性地将 CPU 从一个**进程**切换到另一个**进程**，来实现同时运行多个程序(**进程**)，这使得每个进程“看起来”在其执行过程中停停歇歇，操作系统会将进程相互隔开，**进程彼此不会产生干扰**
- 但与此相反，Java 所使用的并发系统会共享诸如内存和 I/O 这样的资源，因此编写**多线程(线程会共享资源)**程序最基本的困难在于协调不同进程驱动的任务之间对这些资源的使用，**以使得这些资源不会同时被多个任务访问**
- 对于**进程**来说：他们之间没有任何通信的必要，他们都是完全独立的

4、**Java 采取了更加传统的方式，在顺序型语言的基础上提供对线程的支持**

- 线程机制是在由执行程序表示的单一进程中创建任务
- 这种方式产生的一个好处就是操作系统的透明性，对 Java 而言，这是一个重要的目标(编写一次，到处执行)

#### 21.1.2. 改进代码设计

1、在单 CPU 机器上使用多任务的程序在任意时刻仍旧只在执行一项工作，从理论上讲，肯定可以不用任何任务而编写出相同的程序。但是并发可以使程序设计极大地简化。

2、**Java 的线程机制是抢占式的，这表示调度机制会周期性的中断线程，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。**

3、协作式系统中，每个任务都会自动的放弃控制，这要求程序员要有意识地在每个任务中插入某种类型的让步语句，否则该线程就一直会获得执行权

- 协作式系统的优点：上下文切换的开销通常比抢占式系统要低廉许多，并且对可以同时执行的线程数量没有任何限制

## 21.2. 基本的线程机制

1、**并发编程使我们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)中的每一个都将有执行线程来驱动**

2、**一个线程就是在进程中的一个单一的顺序控制流。**

- 单个进程可以拥有多个并发执行的任务，但是你的程序使得每个任务都好像有其自己的 CPU 一样
- 其底层机制是切分 CPU 时间，但通常你不需要考虑它

### 21.2.1. 定义任务

1、线程可以驱动任务，由实现 Runnable 接口来提供(编写 run 方法)

2、通常 run()被写成无限循环的形式，这意味着，除非有某个条件使得 run()终止，否则它将永远运行下去

3、Thread.yield()表示：我已经执行完生命周期中最重要的部分了，此刻正式切换给其他任务执行一段时间的大好时机

- 其调用是对线程调度器的一种建议

4、当从 Runnable 导出一个类时，它必须具有 run 方法，但是这个方法并无特殊之处---它不会产生任何内在的线程能力，要实现线程行为，你必须显式地将一个任务附着到线程上

5、**main 函数也是一个线程**

### 21.2.2. Thread 类

1、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须的初始化操作
- 然后调用 Runnable 的 run()方法，以便在这个新线程中启动该任务

2、**任何线程都可以启动另一个线程**

3、线程调度机制是非确定性的，每次运行顺序可能不完全一样

4、Thread 在创建时都"注册"了自己，有一个对它的引用，而且在它的任务退出其 run()并死亡之前，垃圾回收期无法清除它。(创建 Thread 时不捕获其引用，也不必担心被垃圾回收器回收)

5、Thread 方法介绍

- Thread.start(): start()用来启动一个线程，当调用 start 方法后，系统才会开启一个新的线程来执行用户定义的子任务，在这个过程中，会为相应的线程分配需要的资源
- Thread.run(): run()方法是不需要用户来调用的，当通过 start 方法启动一个线程之后，当线程获得了 CPU 执行时间，便进入 run 方法体去执行具体的任务。注意，继承 Thread 类必须重写 run 方法，在 run 方法中定义具体要执行的任务
- Thread.sleep():
  - Thread.sleep(long millis)
  - Thread.sleep(long millis, int nanos)
  - sleep 相当于让线程睡眠，交出 CPU，让 CPU 去执行其他的任务

- 注意, `sleep` 方法不会释放锁, 也就是说如果当前线程持有对某个对象的锁, 则即使调用 `sleep` 方法, 其他线程也无法访问这个对象
  - 注意, 如果调用了 `sleep` 方法, 必须捕获 `InterruptedException` 异常或者将该异常向上层抛出。当线程睡眠时间满后, 不一定会立即得到执行, 因为此时可能 CPU 正在执行其他的任务。所以说调用 `sleep` 方法相当于让线程进入阻塞状态
- `Thread.yield()`: 调用 `yield` 方法会让当前线程交出 CPU 权限, 让 CPU 去执行其他的线程。它跟 `sleep` 方法类似, **同样不会释放锁**
- 但是 `yield` 不能控制具体的交出 CPU 的时间, 另外, `yield` 方法只能让拥有相同优先级的线程有获取 CPU 执行时间的机会
  - 注意, 调用 `yield` 方法并不会让线程进入阻塞状态, 而是让线程重回就绪状态, 它只需要等待重新获取 CPU 执行时间, 这一点是和 `sleep` 方法不一样的
- `Thread.join()`:
- `Thread.join()`
  - `Thread.join(long millis)`
  - `Thread.join(long millis, int nanos)`
  - 假如在 `main` 线程中, 调用 `thread.join` 方法, 则 `main` 方法会等待 `thread` 线程执行完毕或者等待一定的时间
    - 如果调用的是无参 `join` 方法, 则等待 `thread` 执行完毕
    - 如果调用的是指定了时间参数的 `join` 方法, 则等待一定的时间
  - `join` 方法实现是通过 `wait`: 当在 `t1` 线程中调用 `t2.join()` 时, `t1` 线程会获取线程对象 `t2` 的锁(因此 `t1` 线程必须能够获取 `t2` 的锁才行), 调用该对象的 `wait`, 直到该对象唤醒 `main`(可能 `t2` 执行完毕后会主动执行以下 `notify?`)
- `Thread.interrupt()`: 顾名思义, 即中断的意思。单独调用 `interrupt` 方法可以使得处于阻塞状态的线程抛出一个异常, 也就是说, 它可以用来中断一个正处于阻塞状态的线程; 另外, 通过 `interrupt` 方法和 `isInterrupted()` 方法来停止正在运行的线程
- 通过 `interrupt` 方法可以中断**处于阻塞状态**的线程
  - 直接调用 `interrupt` 方法**不能中断正在运行中的**线程
  - 但是如果配合 `isInterrupted()` 能够中断正在运行的线程, 因为调用 `interrupt` 方法相当于将中断标志位置为 `true`, 那么可以通过调用 `isInterrupted()` 判断中断标志是否被置位来中断线程的执行
- `Thread.stop()` 和 `Thread.destroy()` 是被废弃的方法
- 以下是关系到线程属性的几个方法
- `Thread.getId()`: 获取线程 ID
  - `Thread.getName()/Thread.setName()`: 获取/设置线程名称
  - `Thread.getPriority()/Thread.setPriority()`: 获取/设置线程优先级
  - `Thread.setDaemon()/Thread.Daemon()`: 用来设置线程是否成为守护线程和判断线程是否是守护线程
  - 守护线程和用户线程的区别在于: 守护线程依赖于创建它的线程, 而用户线程则不依赖。举个简单的例子: 如果在 `main` 线程中创建了一个守护线程, 当 `main` 方法运行完毕之后, 守护线程也会随着消亡。而用户线程则不会, 用户线程会一直运行直到其运行完毕。在 JVM 中, 像垃圾收集器线程就是守护线程
- `Thread.currentThread()`: 获取当前线程引用

### 21.2.3. Executor

1、Java SE5 的 `java.util.concurrent` 包中的执行器(Executor)将为你管理 `Thread` 对象，从而简化并发编程(Java API)

- 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，这个中介对象执行任务
- Executor 允许你管理异步任务的执行，而无须显式地管理线程的生命周期
- Executor 在 Java SE5 之后是启动任务的优选方法

2、`ExecutorService`: 具有服务生命周期的 Executor，(`ExecutorService` 继承了 `Executor`，两者都是接口)，知道如何构建恰当的上下文来执行 `Runnable` 对象

3、`ExecutorService exec=`

- `Executors.newCachedThreadPool();`
- `Executors.newFixedThreadPool(int); //限制线程的数量`
- `Executors.newSingleThreadExecutor();`
- `Executor.execute(Runnable)`: 为任务创建一个线程
- `Executor.shutdown()`: 防止新任务被提交给这个 Executor，当前线程将继续运行在 shutdown() 被调用前提交的所有任务。(当提交的所有线程运行完毕后，程序可以正常结束。如果不执行 `shutdown()` 的话，只要向该 Executor 提交过任务，那么程序就不会正常结束，就卡着了，也许是 Executor 内部维护了一个线程)
- `Executor.shutdownNow()`: 发送一个 `interrupt()` 调用给它启动的所有线程。

4、线程池

- 在任何线程池中，现有线程在可能的情况下都会被自动复用
- `CachedThreadPool`: //为每个任务都创建一个线程。在程序执行过程中，通常会创建于所需数量相同的线程，然后在它回收旧线程时停止创建新线程，因此它是合理的 Executor 的首选
- `FixedThreadPool`: //限制线程的数量
- `SingleThreadExecutor`: //线程数量为 1，如果向 SingleThread 提交多个任务，那么这些任务将会排队，每个任务会在下一个任务开始之前运行结束。因此 SingleThreadExecute 会序列化所提交的任务，并维护它自己的悬挂任务队列。

#### 21.2.3.1. ThreadPoolExecutor

1、`public ThreadPoolExecutor(int corePoolSize,  
                                  int maximumPoolSize,  
                                  long keepAliveTime,  
                                  TimeUnit unit,  
                                  BlockingQueue<Runnable> workQueue,  
                                  ThreadFactory threadFactory,  
                                  RejectedExecutionHandler handler)`

- `corePoolSize`: 线程池维护线程的最少量
- `maximumPoolSize`: 线程池维护线程的最大数量
- `keepAliveTime`: 线程池维护线程所允许的空闲时间
- `unit`: 线程池维护线程所允许的空闲时间的单位
- `workQueue`: 线程池所使用的缓冲队列
- `threadFactory`: 线程工厂
- `handler`: 线程池对拒绝任务的处理策略

## 2、运转流程

- 1) 当池子大小小于 corePoolSize 就新建线程，并处理请求
- 2) 当池子大小等于 corePoolSize，把请求放入 workQueue 中，池子里的空闲线程就去从 workQueue 中取任务并处理
- 3) 当 workQueue 放不下新入的任务时，新建线程入池，并处理请求，如果池子大小撑到了 maximumPoolSize 就用 RejectedExecutionHandler 来做拒绝处理
- 4) 另外，当池子的线程数大于 corePoolSize 的时候，**多余的线程**会等待 keepAliveTime 长的时间，如果无请求可处理就自行销毁

## 3、几个包装接口的初始化源码

```
public static ExecutorService newFixedThreadPool(int nThreads) {
 return new ThreadPoolExecutor(nThreads, nThreads,
 0L, TimeUnit.MILLISECONDS,
 new LinkedBlockingQueue<Runnable>());
}

public static ExecutorService newCachedThreadPool() {
 return new ThreadPoolExecutor(0, Integer.MAX_VALUE,
 60L, TimeUnit.SECONDS,
 new SynchronousQueue<Runnable>());
}
```

### 21.2.4. 从任务中产生返回值

- 1、Runnable 是执行工作的独立任务，但它不返回任何值。
- 2、**若希望任务完成时返回值，可以实现 Callable<T> 接口而不是 Runnable 接口**
- 3、Callable<T>是一种具有类型参数的泛型，类型参数 T 表示的是从方法 call()中返回的值，并且必须使用 ExecutorService.submit(Callable)方法调用它
- 4、**submit(...)方法会返回 Future<T> 对象，他用 Callable 返回结果的特定类型进行了参数化。可以用 isDone() 方法来查询 Future<T> 是否已经完成，当任务完成时，它具有一个结果，可以用 Future.get() 方法来获取该结果**

#### 5、<Future>

- boolean cancel(boolean mayInterruptIfRunning): 试图取消对此任务的执行
- T get(): 如有必要，等待计算完成，然后检索其结果
- T get(long timeout, TimeUnit unit)
- boolean isCancelled(): 如果在任务正常完成前将其取消，则返回 true
- boolean isDone(): 如果任务已经完成，则返回 true

### 21.2.5. 休眠

- 1、影响任务行为的简单方法就是调用 sleep()，这将使任务中止执行给定的时间。
- 2、sleep 的调用会抛出 InterruptedException 异常，**异常不能跨线程传播，必须本地处理所有在任务内产生的异常**
- 3、语法
  - TimeUnit.MILLISECONDS.sleep(100); // Java SE5/6-style
  - Thread.sleep(100); // 之前的风格
- 4、任务进行睡眠(即阻塞)，使得线程调度器可以切换到另一个线程，进而驱动另一个任务

## 21.2.6. 优先级

- 1、线程的优先级将线程的重要性传递给了调度器，调度器倾向于让优先权高的线程先执行。
- 2、并不意味着优先权较低的线程将得不到执行(不会导致死锁)，**优先级较低的线程仅仅是执行的频率较低**

- 3、在任何时候都能用 `setPriority()` 来修改优先级

```
Thread.currentThread().setPriority(Thread.MIN_PRIORITY);
```

- 4、JDK 有 10 个优先级

- 但它与多数操作系统都不能映射得很好
- 比如 Windows 有 7 个优先级且不是固定的
- 唯一可移植的方法是当调整优先级的时候，只使用 `MAX_PRIORITY`、`NORM_PRIORITY` 和 `MIN_PRIORITY`

## 21.2.7. 让步

- 1、**如果知道已经完成了在 `run()` 方法的循环一次迭代过程中所需的工作，就可以给线程调度机制一个暗示：你的工作已经做得差不多了，可以让别的线程使用 CPU 了，这个暗示通过调用 `Thread.yield()` 方法来做出(这只是暗示，并不保证一定被采纳，并且只是建议相同优先级的其他线程可以运行)**

## 21.2.8. 后台线程

- 1、**后台线程是指程序运行的时候在后台提供一种通用服务的线程，并且这种线程并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，程序也就终止了，同时会杀死进程中所有的后台线程。反过来说，只要有任何非后台线程还在运行，程序就不会终止，比如 `main` 就是一个非后台线程**

- 2、`Thread.setDaemon(true);` 必须在启动线程之前调用该方法。

- 3、通过编写定制的 `ThreadFactory`(接口，含有方法 `public Thread newThread(Runnable r)`) 可以定制由 `Executor` 创建的线程的属性

- 4、`Executors.newCachedThreadPool(ThreadFactory);` 该方法接受 `ThreadFactory` 对象作为参数用于创建 `Thread` 对象。

- 5、可以通过 `Thread.isDaemon()` 方法来确定线程是否为后台线程

- 6、**后台线程创建的任何线程将被自动设置为后台线程**

- 7、**后台线程不执行 `finnlay` 子句的情况下就会终止其 `run()` 方法。因为当最后一个非后台线程终止时，后台线程会突然终止，不会有任何人希望出现的确认形式，因此不能以优雅的方式来关闭后台线程。**

## 21.2.9. 编码的变体

- 1、可以直接从 `Thread` 继承，代替继承 `Runnable`

- 2、利用适当的 `Thread` 构造器为 `Thread` 对象赋予具体的名称，这个名称可以通过 `getName()` 获得。

## 21.2.10. 术语

- 1、执行的任务和驱动它的线程之间有一个差异。我们对 `Thread` 类实际没有任何控制权(执行器将替你处理线程的创建和管理)你创建任务，通过某种方式将一个线程附着到任务上，以使得这个线程可以驱动任务

2、`Thread` 类本身不执行任何操作，他只是驱动赋予它的任务

3、从概念上讲

- 我们希望创建独立于其他任务运行的任务，因此我们应该能够定义任务，然后说"开始"(start)，并且不用操心细节
- 在物理上，创建线程可能会代价高昂，因此你必须保存并管理它们
- 从实现的角度，将任务从线程中分离出来是很有意义的

4、**我们尝试在描述要执行的工作时使用术语"任务"，只有在引用到驱动任务的具体机制时才会使用术语"线程"**

### 21.2.11. 加入一个线程

1、一个线程可以在其他线程之上调用 `join()` 方法，其效果是等待一段时间直到第二个线程结束才继续执行

- 如果某个线程 `a` 在另一个线程 `t` 上调用 `t.join()`，此线程 `a` 将被挂起，直到目标线程 `t` 结束才恢复
- 也可以在调用 `join()` 时带上一个超时参数(单位可以是毫秒，或者纳秒)，这样如果目标线程在这段时间到期时还没有结束的话，`join()` 方法总能返回。

2、对 `join()` 方法的调用可以被中断，做法是在调用线程上调用 `interrupt()` 方法，这时需要用到 `try-catch` 子句

### 21.2.12. 创建有响应的用户界面

1、使用线程的动机之一就是建立有相应的用户界面。

### 21.2.13. 线程组(忽略它即可)

1、最好把线程组看成试一次不成功的尝试，你只要忽略它就好了

### 21.2.14. 捕获异常??

1、由于线程的本质特性，你不能捕获从线程中逃逸的异常

2、为了解决这个问题，Java SE5 中引入了新接口，`Thread.UncaughtExceptionHandler`，它允许在每个 `Thread` 对象上都附着一个异常处理器，`Thread.UncaughtExceptionHandler.uncaughtException()`会在线程因未捕获异常而临近死亡时被调用

## 21.3. 共享受限资源

1、可以把单线程程序当做在问题域求解的单一实体，每次只能做一件事，永远不用担心诸如"两个实体试图同时使用同一个资源"的问题

2、并发可以同时做许多事情了，同时也出现了两个或多个线程彼此互相干涉的问题

3、`volatile` 关键字：确保本条指令不会因编译器的优化而省略，且要求每次直接读值

### 21.3.1. 不正确地访问资源

1、递增程序自身也需要多个步骤，递增不是原子性的操作，如果不保护任务，即使单一的递增也是不安全的

### 21.3.2. 解决共享资源竞争

1、使用线程时的一个基本问题：你坐在桌边手拿着叉子，正要去叉盘子中的最后一片实物，

当你的叉子就要够着它时，这片实物突然消失了，因为你的线程被挂起了，而另一个餐者进入并吃掉了它。

2、**防止冲突的方法就是当资源被一个任务使用时，在其上加锁，第一个访问某项资源的任务必须锁定这项资源，是其他任务在其被解锁前无法访问。**

- 基本上所有并发模式在解决线程冲突问题的时候，都是采用序列化访问共享资源的方案，这意味着在给定时刻，只允许一个任务访问共享资源
- 因为锁语句产生了一种互相排斥的效果，这种机制常常称为互斥量

3、**当资源被锁住时，其他线程簇拥在门口，当当前线程打开锁准备离开时，“最接近”锁的线程将得到访问资源的权限。可以通过 `yield()` 以及 `setPriority()` 来给线程调度器提供建议，但建议未必有太多效果，这取决于具体平台和 JVM 实现**

4、**Java 以提供关键字 `synchronized` 的形式，为防止资源冲突提供了内置支持。**

- 当任务要执行被 `synchronized` 关键字保护的代码片段的时候，它将检查锁是否可用，然后获取锁，执行代码，解放锁
- `synchronized` 不属于方法特征签名的组成部分，在覆盖方法的时候可以添加
- 声明 `synchronized` 方法的方式：  
`synchronized void f(){}/*...*/`

5、共享资源一般是以对象形式存在的内存片段，但也可以是文件、输入/输出端口，或是打印机。

- 要控制对共享资源的访问，得先把它包装进一个对象。然后把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` 的方法的调用中，那么在这个线程从该方法返回之前，其他所有要调用该类中任何标记为 `synchronized` 方法的线程都会被阻塞。

6、**所有对象都自动含有单一的锁(也称为监视器)，当在对象上调用任意 `synchronized` 方法的时候，此对象都被加锁，这是该对象上其他 `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。换言之，对于某个特定的对象来说，其所有的 `synchronized` 方法共享一个锁**

7、**使用并发时，将域设置为 `private` 是非常重要的，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**

8、一个任务可以多次获得对象的锁：如果一个方法在同一个对象上调用第二个方法，后者又调用了同一对象上的另一个方法，就会发生这样的情况。

- **JVM 负责跟踪对象被加锁的次数，若一个对象被解锁(锁被完全释放)，其计数变为 0。**
- 在任务第一次给对象加锁的时候，计数变为 1，每当这个相同的任务在这个对象上获得锁，计数都会递增。
- 显然，只有首先获得了锁的任务才能继续获得多个锁。
- 每当任务离开一个 `synchronized` 方法，计数递减。
- 当计数为 0 的时候，锁被完全释放，此时别的任务就能使用该资源

9、**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。**

10、**如果在类中有超过一个方法在处理临界数据，那么必须同步所有相关的方法。如果只同步一个方法，那么其他方法将会随意的忽略这个对象锁，并可以在无任何惩罚的情况下被调用。每个访问临界共享资源的方法必须被同步，否则它们就不会正确地工作。**

## 15、显示使用 Lock 对象：

- Lock 对象必须被显式得创建、锁定和释放
- private Lock lock=new ReentrantLock();*//作为私有字段，惯用法内部化*
- 与内建的锁形式相比，缺乏优雅，但更具灵活性
- 如果在使用 synchronized 关键字时，某些事物失败了，那就会抛出一个异常，但是没有机会做任何清理工作，以维护系统使其处于良好状态；有了显式的 Lock 对象，就可以用 finally 子句将系统维护在正确的状态。

```
lock.lock()
try{
 //do something
}finally{
 lock.unlock();
}
```

- 解决特殊问题才考虑使用显式的 Lock 对象
  - 例如：synchronized 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，然后放弃他。
  - boolean captured=lock.tryLock();*//可以失败，立即返回是否获取成功*
  - boolean captured=lock.tryLock(2, TimeUnit.SECONDS);*//该尝试可以在 2s 后失败：在 2 秒内若获取失败，继续获取直至成功，或者 2s 后返回获取失败*

### 21.3.3. 原子性与易变性

1、(不正确的?)有关 Java 线程的讨论中，一个不正确的知识是"原子操作不需要进行同步控制"

2、(不正确的?)原子操作是不能被线程调度机制中断的操作；一旦操作开始，那么它一定可以在可能发生的"上下文切换"之前(切换到其他线程执行)执行完毕。

3、依赖于原子性是很棘手且很危险的事情

4、原子性可以应用于除 long 和 double 之外的所有基本类型之上的"简单操作"。对于读取和写入除 long 和 double 之外的基本类型变量这样的操作，可以保证它们会被当做不可分(原子)的操作来操作内存。但是 JVM 可以将(64 位 long 和 double 变量)的读取和写入当做两个分离的 32 位操作来执行，这样就产生了一个在读取和写入操作中间发生的上下文切换，从而导致不同的任务可以看到不正确结果的可能性(实际上 JVM 实现的时候是会避免这种情况的，了解即可)

5、当定义 long 或 double 变量时，如果使用 volatile 关键字，就会获得(简单的赋值与返回操作的)原子性(实际上，JVM 的实现，即便不加 volatile 也能获得简单赋值与返回的原子性)

6、一个任务做出修改，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使用的(例如，修改值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的状态有不同的视图

7、volatile 关键字确保了应用中的可见性。如果将一个域声明为 volatile，那么只要对这个域产生了些操作，那么所有的读操作就都可以看到这个修改。volatile 域会立即被写入到主存中，而读取操作就发生在主存中

8、在非 volatile 域上的原子操作不必刷新到主存中去，因此其他读取该域的任务也不必看到这个新值，如果多个任务在同时访问某个域，那么这个域就应该是 volatile 的，否则这个域就只能经由同步来访问，同步也会导致向主存中刷新。因此如果一个域完全由 synchronized 方法或语句块防护(synchronized 可以确保可见性)，那就不必将其设置为 volatile

9、一个任务所作的任何写入操作对这个任务来说都是可视的，因此如果它只需要在这个任务内部可视，就不需要将其设置为 volatile。

10、**使用 volatile 而不是 synchronized 的唯一安全情况就是：类中只有一个可变的域。除此之外第一选择就应该使用 synchronized 关键字**

- 当一个域的值依赖于它之前的值， volatile 就无法工作了，如果更改这个域的操作不是原子的，那么即便 volatile 直接修改主存，也有可能在非原子的操作中插入另一个线程的其他操作
- 从主存读取该值时保证同步，将该值写入主存保证同步，但是中间那些操作，如果不是原子性的，就有可能插入其他线程的操作，导致在更新的时候，实际上并非是最新的，详见 JVMPoint

11、在 Java 中对域中的值做赋值(一个写操作)和返回(一个读操作)操作通常都是原子性的，很遗憾，在 Java 中，递增操作不是原子性的(涉及一个读操作和写操作)

12、基本上如果一个域可能会被多个任务同时访问，或者这些任务中至少有一个是写入任务，那么你就应该将这个域设置为 volatile，即告诉编译器：不要执行任何移出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。

13、volatile：可视性与有序性

#### 21.3.4. 原子类

1、Java SE5 引入了 AtomicInteger、AtomicLong、AtomicReference 等特殊的原子性变量，他们提供下面形式的原子性条件更新操作：

```
boolean compareAndSet(expectedValue,updateValue)
```

2、这些类被调整为可以使用在某些现代处理器上的可获得的，并且是机器级别上的原子性。

3、**对于常规编程来说，很少会派出用场。**

#### 21.3.5. 临界区

1、有时，你只是希望防止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”，它也是用 synchronized 关键字建立。这里 synchronized 被用来指定某个对象，此对象的锁被用来对花括号内的代码进行同步控制。

```
synchronized(syncObject){
 //这里的代码在一个时刻只能被一个任务访问，即带有锁
}
```

2、这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，如果其他线程已经得到这个锁，那么就得等到锁被释放后才能进入临界区。

3、适用场景：某人交给你一个非线程安全的类(Pair)，需要在线程环境中使用它，通过创建该类的 Manager 类(PairManager)来实现，该 Manager 类持有该类的对象，并控制对它的一切访问。采用同步控制块(传入的参数就是 Manager 类的 this 本身)

4、采用同步控制块进行同步，对象不加锁的时间更长。使得其他线程能够更多地访问。

5、还可以显式使用 Lock 对象来创建临界区。

#### 21.3.6. 在其他对象上同步

1、由于 synchronized 块必给定一个在其上进行同步的对象，最合理的方式是使用正在被调用的当前对象:synchronized(this)，在这种方式中，如果获得了 synchronized 块上的锁，那么该对象的其他 synchronized 方法和临界区就不能被调用了，因此如果在 this 上同步，临界

区的效果就会直接缩小在同步的范围内。

2、有时必须在另一个对象上同步，但是如果你这么做，就必须确保所有相关的任务都是在同一个对象上同步的。

```
class DualSynch{
 private Object syncObject=new Object();
 public synchronized void f(){ //该方法在 this 同步
 for(int i=0;i<5;i++){
 print("f()");
 Thread.yield();
 }
 }
 public void g(){
 synchronized(syncObject){// g()在 syncObject 上同步
 for(int i=0;i<5;i++){
 print("g()");
 Thread.yield();
 }
 }
 }
}
```

- 如果两个线程分别调用 g 和 f，不会导致阻塞；如果 g 方法中传入的是 this，会导致两个方法都在 this 上同步，因此当一个线程调用 f()或 g()后，另一个线程调用的剩下的方法将会被阻塞

### 21.3.7. 线程本地存储

- 1、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。
- 2、线程本地存储是一种自动化机制，可以为使用相同变量的每个不同线程都创建不同的存储。因此，如果你有 5 个线程都要使用变量 x 所表示的对象，那么线程本地存储就会生成 5 个用于 x 的不同存储块，主要是，它们使得你可以将状态与线程关联起来
- 3、java.lang.ThreadLocal 类创建和管理线程本地存储。ThreadLocal 对象通常当做静态域存储。

## 21.4. 终结任务

- 1、在之前的某些事例中，cancel()和 isCancelled()方法被防放置到了一个所有任务都可以看到的类中，通过检查 isCancelled()来确定何时终止它们自己(while 的判断条件，等等方式)
- 2、但在某些情况下，任务必须更加突然地终止

### 21.4.1. 在阻塞时终结

#### 1、线程状态

- 1) 新建(new): 当线程被创建时，它只会短暂地处于这种状态。此时它已经分配了必要的系统资源，并执行了初始化，此刻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): 在这种状态下，只要调度器把时间片分配给线程，线程就可以运行。

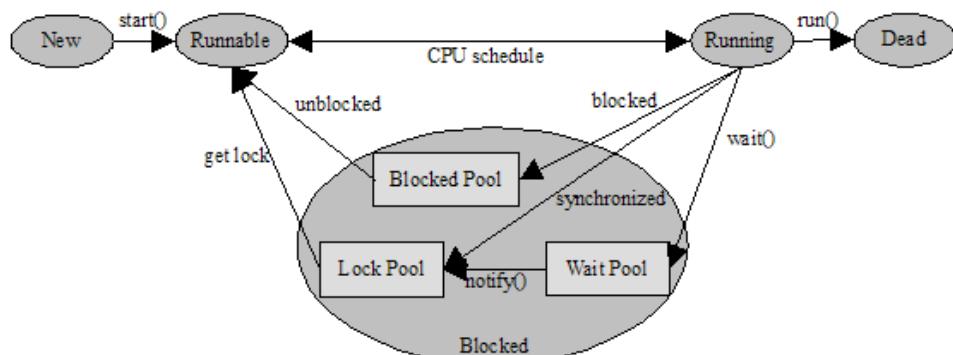
也就是说，在任意时刻，线程可以运行也可以不运行。只要调度器能分配时间片给线程，他就可以运行，这不同于死亡和阻塞状态

- 3) **运行(Running):** Java 运行系统通过调度选中一个处于就绪状态的线程，使其占有 CPU 并转为运行状态。此时，系统真正执行 run()方法
- 4) **阻塞(Blocked):** 线程能够运行，但有某个条件阻止它运行。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，它才有可能执行的操作
- 5) **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

## 2、一个任务可以进入阻塞状态，原因有以下几种：

- 通过调用 sleep 使任务进入休眠状态，在这种情况下，任务在指定的时间内不会运行
- 通过调用 wait()使线程挂起，直到线程得到了 notify()或 notifyAll()消息(或者在 Java SE5 的 java.util.concurrent 类库中等价的 signal()或 signalAll()消息)，线程才会进入就绪状态
- 任务在等待某个输入/输出完成
- 任务试图在某个对象上调用其同步控制方法，但是对象锁不可用，因为另一个任务已经获取了这个锁。

## 3、我们需要查看的问题：有时希望终止处于阻塞状态的任务，如果处于阻塞状态的任务，你不能等待其到达代码中能检查其状态值的某一点，因而决定让它主动地终止，那么你就必须强制这个任务跳出阻塞状态



### 21.4.2. 中断

1、在 Runnable.run()方法的中间打断它，与等待该方法到达对 cancel 标志的测试，或者到达程序员准备好离开该方法的其他一些地方相比，要棘手得多。

2、**当你打断被阻塞的任务时，可能需要清理资源，正因为这一点，在 run()方法中间打断，更像是抛出异常，因此在 Java 线程中的这种类型的异常中断中用到了异常**

3、**Thread 类包含 interrupt()方法，因此你可以终止被阻塞的任务，这个方法将设置线程的中断状态。**

4、**如果一个线程已经被阻塞，或者试图执行一个阻塞操作，那么设置这个线程的中断状态(调用 Thread.interrupt()或者 Future.cancel(true))将抛出 InterruptedException**

5、为了调用 interrupt，你必须持有 Thread 对象，但是新的 concurrent 类库避免对 Thread 对象的直接操作，转而尽量通过 Executor 来执行所有操作。**如果在 Executor 上调用 shutdownNow()**，那么它将发送一个 interrupt()调用给它启动的所有线程。

6、若希望只中断某个单一任务，可以通过使用 Executor.submit()来启动任务，就可以持有该任务的上下文，submit 将返回一个泛型 Future<?>，其中有一个未修饰的参数，因为你永远都不会在其上调用 get()。持有这个 Future，我们可以在其上调用 cancel()。并因此可以使⽤它来中断某个特定任务。

```
static void test(Runnable r) throws InterruptedException{
 Future<?> f=exec.submit(r);
 f.cancel(true);
}
```

7、I/O 和在 synchronized 块上的等待是不可中断的(Thread.interrupt()或者向 Executor.submit()返回的 Future 对象调用 cancel()不会抛出 InterruptedException)但是可以通过关闭任务在其发生阻塞的底层资源来等效中断(一旦底层资源关闭，任务将解除阻塞)

8、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

9、如果尝试着在一个对象上调用其 synchronized 方法，而这个对象的锁已经被其他任务获得，那么调用任务将被挂起(阻塞)，直至这个锁可获得

10、同一任务能够调用在同一个对象中的其他 synchronized 方法，因为这个任务已经持有锁了。

11、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。因此 Java SE5 并发类库添加了一个特性，即在 ReentrantLock 上阻塞的任务具备可以被中断的，这与在 synchronized 方法或临界区上阻塞的任务完全不同。

#### 21.4.3. 检查中断

1、当在线程上调用 interrupt()时，中断发生的唯一时刻就是任务要进入到阻塞操作中，或者已经在阻塞操作内部时。

2、如果只能通过在阻塞调用上抛出异常来退出，那么久无法总是可以离开 run()循环。因此，如果调用 interrupt()以停止某个任务，那么在 run()循环碰巧没有产生任何阻塞调用的情况下，你的任务将需要第二种方式来退出

3、Java 中断模型也是这么简单，每个线程对象里都有一个 boolean 类型的标识(不一定就是 Thread 类的字段，实际上也的确不是，这几个方法最终都是通过 native 方法来完成的)，代表着是否有中断请求(该请求可以来自所有线程，包括被中断的线程本身)。

3、这种机会是由中断状态来表示的，其状态可以通过 interrupt()来设置。

4、可以通过调用 interrupted()来检查中断状态，这不仅可以告诉你 interrupt()是否被调用过(返回 true 则说明处于中断状态，即之前调用了 interrupt())，而且还可以清除中断状态，调用后将状态标记为非中断状态。

5、清除中断状态可以确保并发结构不会就某个人物被中断这个问题通知你两次，你可以经由单一的 InterruptedException 或单一的成功的 Thread.interrupted()测试来得到这种通知。

6、如果想要再次检查以了解是否被中断，则可以在调用 Thread.interrupted()时将结果存储起来

7、惯用法

```
public void run(){
 try{
 while(!Thread.interrupted()){
 NeedsCleanup n1=new NeedsCleanup(1);
 try{
 print("Sleeping");
 }
```

```

 TimeUnit.SECONDS.sleep(1);
 NeedsCleanup n2=new NeedsCleanup(2);
 try{
 print("Calculating");
 for(int i=0;i<2500000;i++)
 d=d+(Math.PI+Math.E)/d;
 print("Finished time-consuming operation");
 }finally{
 n2.cleanup();//记得清理资源
 }
 }finally{
 n1.cleanup();//记得清理资源
 }
}
print("Exiting via while() test");
}catch(InterruptedException e){
 print("Exiting via InterruptedException");
}
}

```

#### 21.4.4. interrupt、isInterrupt、interrupted

1、`interrupt`方法用于中断线程。调用该方法的线程的状态为将被置为"中断"状态。

- 注意：线程中断仅仅是置线程的中断状态位，不会停止线程。需要用户自己去监视线程的状态并做处理
- 支持线程中断的方法(也就是线程中断后会抛出 `InterruptedException` 的方法)就是在监视线程的中断状态，一旦线程的中断状态被置为"中断状态"，就会抛出中断异常。

2、`interrupted` 和 `isInterrupted`

```

public static boolean interrupted () {
 return currentThread().isInterrupted(true);
}

```

```

public boolean isInterrupted () {
 return isInterrupted(false);
}

```

- 这两个方法有两个主要区别

- 1) `interrupted` 是作用于当前线程，`isInterrupted` 是作用于调用该方法的线程对象所对应的线程(线程对象对应的线程不一定是当前运行的线程。例如我们可以在 A 线程中去调用 B 线程对象的 `isInterrupted` 方法)
- 2) 这两个方法最终都会调用同一个方法，只不过参数一个是 `true`，一个是 `false`

- 被调用的方法如下(本地方法)

```

private native boolean isInterrupted(boolean ClearInterrupted);

```

- 如果这个参数为 `true`，说明返回线程的状态位后，要清掉原来的状态位(恢复成原来情况)。这个参数为 `false`，就是直接返回线程的状态位。
- 这两个方法很好区分，只有当前线程才能清除自己的中断位(对应 `interrupted()` 方

法)

#### 21.4.5. sleep()、wait()、join()与 interrupt()

##### 1、sleep()&interrupt()

- 线程 A 正在使用 sleep()暂停着: Thread.sleep(100000);
- 如果要取消他的等待状态, 可以在正在执行的线程里(比如这里是 B)调用 a.interrupt()令线程 A 放弃睡眠操作, 这里 a 是线程 A 对应到的 Thread 实例
- 执行 interrupt()时, 并不需要获取 Thread 实例的锁定。任何线程在任何时刻, 都可以调用其他线程 interrupt()。当 sleep 中的线程被调用 interrupt()时, 就会放弃暂停的状态。并抛出 InterruptedException。丢出异常的, 是 A 线程。

##### 2、wait()&interrupt()

- 线程 A 调用了 wait()进入了等待状态, 也可以用 interrupt()取消
- 不过这时候要小心锁定的问题。线程在进入等待区, 会把锁定解除, **当对等待中的线程调用 interrupt()时(注意是等待的线程调用其自己的 interrupt()), 会先重新获取锁定, 再抛出异常。在获取锁定之前, 是无法抛出异常的**
- 假设线程 a 正在 wait, 等待的对象是 obj, 而此时线程 b 正持有 obj 的锁, 那么调用 a.interrupt 是没有用的, 因为 a 无法获取锁

##### 3、join()&interrupt()

- 当线程以 join()等待其他线程结束时, 一样可以使用 interrupt()取消之。**因为调用 join()不需要获取锁定, 故与 sleep()时一样, 会马上跳到 catch 块里**
- 注意调用 interrupt()方法, 一定是**通过阻塞的线程来调用其自己的 interrupt 方法(在其他线程中获取阻塞线程的引用, 然后调用 interrupt 方法)**。如在线程 a 中调用 t.join()。则 a 会等 t 执行完后在执行 t.join 后的代码, 当在线程 b 中调用 a.interrupt()方法, 则会抛出 InterruptedException

##### 4、interrupt()只是改变中断状态而已

- interrupt()不会中断一个正在运行的线程。这一方法实际上完成的是, 在线程受到阻塞时抛出一个中断信号, 这样线程就得以退出阻塞的状态
- 更确切的说, 如果线程被 Object.wait, Thread.join 和 Thread.sleep 三种方法之一阻塞, 那么, 它将接收到一个中断异常(InterruptedException), 从而提早地终结被阻塞状态
- **如果线程没有被阻塞, 这时调用 interrupt()将不起作用**
- 线程 A 在执行 sleep, wait, join 时, 线程 B 调用 A 的 interrupt 方法。的确这个时候 A 会有 InterruptedException 异常抛出来。但这其实是在 sleep, wait, join 这些方法内部会不断检查中断状态的值, 而自己抛出的 InterruptedException
- 如果线程 A 正在执行一些指定的操作时如赋值、for、while、if、调用方法等, 都不会去检查中断状态, 所以线程 A 不会抛出 InterruptedException, 而会一直执行着自己的操作。当线程 A 终于执行到 wait(), sleep(), join()时。才马上会抛出 InterruptedException
- 若没有调用 sleep()、wait()、join()这些方法, 或是没有在线程里自己检查中断状态自己抛出 InterruptedException 的话, 那 InterruptedException 是不会被抛出来的

### 21.5. 线程之间的协作

#### 1、当任务协作时, 关键的问题是这些任务之间的握手, 为了实现这种握手, 我们使用了相

同的基础特性：互斥。在这种情况下，互斥能够保证只有一个任务可以响应某个信号，这样就可以根除任何可能的竞争条件。

2、在互斥之上，我们为任务添加了一种新途径，可以将其自身挂起，直至某些外部条件发生变化。

### 21.5.1. `wait()`与`notifyAll()`

1、当线程在等待某个条件的时候

- 1) 我们当然可以让线程不断的进行测试，当条件达到之后就执行任务，但是这种方法显然 CPU 利用率太低了
- 2) `wait()`提供了一种更好的途径，当我们等待的条件还没有达到时，我们可以选择将线程挂起。如果当前条件满足了，那么就可以调用 `notifyAll()`来将这个线程唤醒继续执行或等待
- 3) 这就是利用 `wait()`和 `notifyAll()`来进行线程通信的基本逻辑，看起来似乎很简单。但是其实还是有一些地方是值得注意的

2、我们说调用 `wait()`时线程会被挂起，但是同时还值得注意的是当前线程持有对象的锁是会被释放的；也是说，其它的线程可以重新获得该对象的锁，即该对象其它的 `synchronized` 方法在 `wait()`期间是可以被执行的，这一点是很重要的，因为只有执行其它的 `synchronized` 方法才可能使得当前线程等待的条件达到。而与 `wait()`类似的 `sleep()`/`yield()`方法是不会释放锁的

3、**`wait()`和`notify()`/`notifyAll()`这几个方法都需要放到同步控制方法或则是同步代码控制块中使用；这一点也是因为他们需要操作对象的锁。而`sleep()`/`yield()`就没有这个要求了，可以在任何地方进行调用**

4、有两种形式的 `wait()`

- 1) 第一种版本接受毫秒数作为参数，与 `sleep` 接受参数的意思指“在此期间暂停”，但是与 `sleep` 不同，对于 `wait` 而言：
  - 在 `wait()`期间对象锁是释放的
  - 可以通过 `notify()`、`notifyAll()`、或者命令时间到期，从 `wait()`中恢复执行。
- 2) 第二种版本的不接受任何参数，这种 `wait()`会一直等待下去，直到线程接收到 `notify` ()或者 `notifyAll()`消息。

7、**`wait()`、`notify()`以及`notifyAll()`的特殊方面：**

- 这些方法是基于 `Object` 的一部分，而不是属于 `Thread` 的一部分。(为什么作为通用基类 `Object` 的实现？因为这些方法操纵的锁也是对象的一部分，`wait` 和 `notify` 就是依赖该 `Object` 来进行通信的)
- 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()`，如果在非同步控制方法里调用这些方法，程序能通过编译，但是运行时会得到 `IllegalMonitorStateException` 异常。并伴随着一些含糊的消息(比如当前线程不是拥有者：意思是，调用 `wait()`、`notify()`和 `notifyAll()`的任务在调用这些方法前必须拥有(获取)对象的锁)

8、**当调用`notify()`或`notifyAll()`，将唤醒在对`wait()`调用中被挂起的任务，为了使该任务从`wait()`中唤醒，它必须首先重新获得它进入`wait()`时释放的锁，在这个锁变得可用之前，这个任务是不会被唤醒的。**

9、**必须用一个检查感兴趣的条件的`while`循环包围`wait()`，这很重要：**

- 可能有多个任务出于相同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(使得又该让这个任务进入 `wait`)，如果属于这种情况，那么这个任务应该被再次

### 挂起

- 在这个任务从 `wait()` 被唤醒的时刻，可能会有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，或者执行器操作已显得无关紧要。此时，应该通过再次调用 `wait()` 将其挂起。
- 可能某些任务出于不同的原因在等待你对象上的锁(在这种情况下，必须使用 `notifyAll()`)，在这种情况下，你需要检查是否已经由正确的原因为之唤醒，如果不是，就再次调用 `wait()` 将其挂起。

### 10、错失的信号：如果不用检查感兴趣的条件包围 `wait()` 会出现信号丢失的情况

T1:

```
synchronized(sharedMonitor){
 <setup condition for T2>
 sharedMonitor.notify();
}
```

T2:

```
while(someCondition){
 //Point 1
 synchronized(sharedMonitor){
 sharedMonitor.wait();
 }
}
```

- T1 是通知 T2 的线程，T1 将条件作出某些改变，以通知 T2，但上述写法会出现信号的错失：如果此时 T1 在执行，且 T2 处于 Point 1 位置等待同步锁的释放，不久后，T1 更改了状态，并通知了 T2 且释放了锁，T2 获取锁后，直接进入了 `wait()`。问题出在 T2 并没有判断该状态是否是“感兴趣的信号”，因为 `while` 的条件在之前已经判断过了，T1 对条件的改变已经为时已晚。
  - 可以将 T2 改为如下形式
- ```
synchronized(sharedMonitor){  
    while(someCondition)  
        sharedMonitor.wait();  
}
```

21.5.2. `notify()` 与 `notifyAll()`

1、在技术上，可能会有多个任务在单个对象上处于 `wait()` 状态，因此调用 `notifyAll()` 比调用 `notify()` 更安全。

2、使用 `notify()`:

- 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会有一个被唤醒，如果你希望使用 `notify()`，就必须保证被唤醒的是恰当的任务。
- 另外，为了使用 `notify()`，所有任务必须等待相同条件，因为如果有多个任务在等待不同条件，那么你就不会知道是否唤醒了恰当的任务。
- 如果使用 `notify()`，当条件发生变化时，必须只有一个任务能够从中受益。
- 最后，这些限制对所有可能存在的子类都必须总是起作用(???)

3、对于 `notifyAll()` 的描述：`notifyAll()` 将唤醒所有正在等待的任务

- 事实上，`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

4、当某种条件达到时，Java 提供了两种方法来唤醒等待的线程： notify() 和 notifyAll()

- 可以简单的认为它们之间的不同是 `notify()` 只会随机唤醒一个线程，而 `notifyAll()` 会唤醒所有的线程
- 但是实际上没有那么简单，首先 `notify()` 会唤醒一个线程，那么到底什么样的线程才可能被它唤醒，是所有的线程都有机会被它唤醒么？当然不是这样，我们要明确调用 `notify()` 或 `notifyAll()` 的线程肯定是持有某个对象的锁，而我们唤醒的线程也就是在等待这些锁挂起的线程(即前面在持有这个对象锁而调用 `wait()` 将自己挂起的线程)
- 所以 `notify()` 只会随机唤醒一个在等待当前对象锁挂起的线程，而 `notifyAll()` 会唤醒所有在等待当前对象锁挂起的线程而不是所有挂起的线程，当然这种唤醒是依次进行的而不是同时唤醒的
- 正是因为 `notifyAll()` 会唤醒所有等待当前对象锁的挂起线程，可能某些线程虽然被唤醒但是条件还没达到需要继续挂起

21.5.3. `wait()`, `notify()`, `notifyAll()` 必须在同步方法/代码块中调用

1、在 Java 中，所有对象都能够被作为“**监视器 monitor**”——指一个拥有一个独占锁，一个入口队列和一个等待队列的实体 **entity**。所有对象的非同步方法都能够在任意时刻被任意线程调用，此时不需要考虑加锁的问题。而对于对象的同步方法来说，在任意时刻有且仅有一个拥有该对象独占锁的线程能够调用它们。例如，一个同步方法是独占的。如果在线程调用某一对象的同步方法时，对象的独占锁被其他线程拥有，那么当前线程将处于阻塞状态，并添加到对象的入口队列中。

2、只有在调用线程拥有某个对象的独占锁时，才能够调用该对象的 `wait()`, `notify()` 和 `notifyAll()` 方法。这一点通常不会被程序员注意，因为程序验证通常是在对象的同步方法或同步代码块中调用它们的。如果尝试在未获取对象锁时调用这三个方法，那么你将得到一个“`java.lang.IllegalMonitorStateException:current thread not owner`”

3、**当一个线程正在某一个对象的同步方法中运行时调用了这个对象的 `wait()` 方法，那么这个线程将释放该对象的独占锁并被放入这个对象的等待队列**。注意，`wait()` 方法强制当前线程释放对象锁。这意味着在调用某对象的 `wait()` 方法之前，当前线程必须已经获得该对象的锁。因此，线程必须在某个对象的同步方法或同步代码块中才能调用该对象的 `wait()` 方法。当某线程调用某对象的 `notify()` 或 `notifyAll()` 方法时，任意一个(`notify()`)或者所有(`notifyAll()`)在该对象的**等待队列中的线程，将被转移到该对象的入口队列**。接着这些队列(译者注:可能只有一个)将竞争该对象的锁，最终获得锁的线程继续执行。如果没有线程在该对象的等待队列中等待获得锁，那么 `notify()` 和 `notifyAll()` 将不起任何作用。在调用对象的 `notify()` 和 `notifyAll()` 方法之前，调用线程必须已经得到该对象的锁。因此，必须在某个对象的同步方法或同步代码块中才能调用该对象的 `notify()` 或 `notifyAll()` 方法。

4、对于处于某对象的等待队列中的线程，只有当其他线程调用此对象的 `notify()` 或 `notifyAll()` 方法时才有机会继续执行

5、调用 `wait()` 方法的原因通常是，调用线程希望某个特殊的状态(或变量)被设置之后再继续执行。调用 `notify()` 或 `notifyAll()` 方法的原因通常是，调用线程希望告诉其他等待中的线程：“特殊状态已经被设置”。这个状态作为线程间通信的通道，它必须是一个可变的共享状态(或变量)。

- 例如，生产者线程向缓冲区中写入数据，消费者线程从缓冲区中读取数据。消费者线程需要等待直到生产者线程完成一次写入操作。生产者线程需要等待消费者线程完成一次读取操作。假设 `wait()`、`notify()`、`notifyAll()` 方法不需要加锁就能够被调用。

此时消费者线程调用 `wait()` 正在进入状态变量的等待队列(译者注:可能还未进入)。在同一时刻, 生产者线程调用 `notify()` 方法打算向消费者线程通知状态改变。那么此时消费者线程将错过这个通知并一直阻塞。因此, 对象的 `wait()`、`notify()`、`notifyAll()` 方法必须在该对象的同步方法或同步代码块中被互斥地调用。

21.5.4. 生产者与消费者

- 1、在典型的生产者-消费者实现中, 应使用先进先出队列来存储被生产和消费的对象。
- 2、显式使用 `Lock` 和 `Condition` 对象
 - 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 `wait()` 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时, 你可以通过调用 `signal()` 来通知这个任务, 从而唤醒一个任务, 或者调用 `signalAll()` 来唤醒所有在这个 `Condition` 上被其自身挂起的任务。(与 `notifyAll()` 相比, `signalAll()` 是更安全的方式)
 - 惯用法

```
private Lock lock=new ReentrantLock();
private Condition condition=lock.newCondition();
```
 - `Condition` 对象用于管理任务间的通信。
 - `Condition` 对象本身不包含任何有关处理状态的信息, 因此需要定义额外的表示处理状态的信息。

21.5.5. 生产者-消费者队列

- 1、`wait()` 和 `notifyAll()` 方法以一种非常低级的方式解决了任务互操作问题, 即每次交互时都握手。
- 2、可以使用同步队列来解决任务协作的问题。同步队列在任何时刻都只允许一个任务插入或移除元素
- 3、`java.util.concurrent.BlockingQueue` 接口中提供了这个队列, 这个接口有大量的标准实现:
 - 1) **ArrayBlockingQueue**: 规定大小的 `BlockingQueue`, 其构造函数必须带一个 `int` 参数来指明其大小.其所含的对象是以 FIFO(先入先出)顺序排序的
 - 2) **LinkedBlockingQueue**: 大小不定的 `BlockingQueue`, 若其构造函数带一个规定大小的参数, 生成的 `BlockingQueue` 有大小限制, 若不带大小参数, 所生成的 `BlockingQueue` 的大小由 `Integer.MAX_VALUE` 来决定.其所含的对象是以 FIFO(先入先出)顺序排序的
 - 3) **PriorityBlockingQueue**: 类似于 `LinkedBlockingQueue`, 但其所含对象的排序不是 `FIFO`, 而是依据对象的自然排序顺序或者是构造函数的 `Comparator` 决定的顺序
 - 4) **SynchronousQueue**: 特殊的 `BlockingQueue`, 对其的操作必须是放和取交替完成的
- 4、阻塞队列可以解决非常大量的问题, 而其方式与 `wait()` 和 `notifyAll()` 相比, 则简单并可靠得多。
- 5、如果 `BlockingQueue` 是空的, 从 `BlockingQueue` 取出东西操作将会被阻断进入等待状态, 直到 `BlockingQueue` 进了东西才会被唤醒。如果 `BlockingQueue` 是满的, 任何试图往里存东西的操作也会被阻断进入等待状态, 直到 `BlockingQueue` 里有空间才会被唤醒继续操作。
- 6、方法:
 - 1) `BlockingQueue.add(anObject)`: 将 `anObject` 加到 `BlockingQueue` 里, 如果 `BlockingQueue` 可以容纳, 则返回 `true`, 否则返回 `false`
 - 2) `BlockingQueue.offer(anObject)`: 表示如果可能的话, 把 `anObject` 加到 `BlockingQueue` 里, 如果 `BlockingQueue` 可容纳, 返回 `true`, 否则返回 `false`

- 3) **BlockingQueue.put(anObject):** 把 anObject 加到 BlockingQueue 里, 如果 BlockingQueue 没有空间, 则调用此方法的线程将被阻断直到 BlockingQueue 里面有空间再继续
- 4) **BlockingQueue.poll(time):** 取走 BlockingQueue 排在首位的对象, 若不能立即取出, 则可以等待 time 参数规定的时间, 取不到时返回 null
- 5) **BlockingQueue.take(anObject):** 取走 BlockingQueue 里排在首位的对象, 若 BlockingQueue 为空, 调用此方法的线程将被阻断进入等待状态, 直到 BlockingQueue 有新对象加入

7、看 P715-P716 的例子, 非常的赞

21.5.6. 任务间使用管道进行输入/输出

- 1、通过在输入/输出在线程间通信非常有用。提供线程功能的类库以"管道"的形式对线程的输入/输出提供了支持。它们在 Java 输入/输出类库中的对应物就是 PipedWriter 类(允许任务向管道写)和 PipedReader(允许任务向管道读取)。
- 2、这个模型可以看做是"生产者-消费者"的变体, 管道基本上是一个阻塞队列, 存在于多个引入 BlockingQueue 之前的 Java 版本中。
- 3、调用 PipedReader.read()时, 如果没有更多数据, 管道将自动阻塞
- 4、BlockingQueue 用起来更加健壮和简易

21.5.7. 线程之间通信方式

21.5.7.1. 同步

- 1、这种方式, 本质上就是"共享内存"式的通信。多个线程需要访问同一个共享变量, 谁拿到了锁(获得了访问权限), 谁就可以执行

21.5.7.2. while 轮询方式

21.5.7.3. wait/notify 机制

21.5.7.4. 管道通信

21.6. 死锁

- 1、一个对象可以有 synchronized 方法或其他形式的加锁机制来防止别的任务在互斥还没有释放的时候就访问这个对象。
- 2、任务可以变成阻塞状态, 所有可能出现以下情况: 某个任务在等待另一个任务, 而后者又在等待别的任务, 直到这条链上的任务又在等待第一个任务, 这样得到了一个任务间相互等待的死循环, 没有哪个线程能继续, 这被称为死锁。
- 3、死锁例子, 哲学家进餐:
 - n 个哲学家围城一圈坐好
 - 每个哲学家都有一支筷子
 - 每个哲学家都会去获取自己的这只以及右边哲学家的这支筷子作为其进食所用
 - 当每个哲学家首先获取了右边哲学家的筷子时(在所有哲学家中没有一个获取了自

身的筷子之前), 此时将发生死锁, 因为, 哲学家只有进食完毕才会释放筷子, 但是所有的筷子均被不同的哲学家所占有。

4、死锁条件: 四个同时满足

- 互斥条件: 任务使用的资源中至少有一个是不能被共享的。
- 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
- 资源不能被任务抢占, 任务必须把资源释放当做普通的事情
- 必须有循环等待(该条件最容易不满足)

21.7. 新类库中的构件

1、Java SE5 的 `java.util.concurrent` 引入了大量涉及用来解决并发问题的新类

21.7.1. CountDownLatch

- 1、它被用来同步一个或多个任务, 强制他们等待由其它任务执行的一组操作完成
- 2、你可以向 `CountDownLatch` 对象设置一个初始计数器, 任何在这个对象上调用 `wait()` 的方法都将阻塞, 直到这个计数值到达 0。其他任务在结束工作时, 可以在该对象上调用 `countDown()` 来减少这个数值。`CountDownLatch` 被设计为支出法一次, 计数值不能被重置, 如果需要能够重置计数值的版本, 可以使用 `CyclicBarrier`
- 3、调用 `countDown()` 的任务在产生这个调用时并没有阻塞, 只有对 `await()` 的调用会被阻塞, 直到计数值到达 0
- 4、`CountDownLatch` 的典型用法是将一个程序分为 n 个相互独立的可解决问题, 并创建值为 0 的 `CountDownLatch`, 每当任务完成时, 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()`, 将它们自己拦住, 直至锁存器技术结束

5、惯用的形式: 见例子 P723

- 创建一个 `CountDownLatch` 对象 c
- 在所有相关的任务中定义一个私有的 `CountDownLatch` 的引用, 构造器接受一个 `CountDownLatch` 对象, 并将其绑定到私有引用上
- 这样所有任务在执行完各自的任务后调用 `countDown()`, 并且调用 `CountDownLatch.wait()` 进行挂起, 直达计数为 0, 被唤醒
- 这样与问题相关的并行任务就被同步到了这一个共享的 `CountDownLatch` 对象上

21.7.2. CyclicBarrier

1、适用情况:

- 你希望创建一组任务, 它们并行地执行工作, 然后在进行下一步之前, 直至所有任务都完成。
- 它使得所有的并行任务都将在栅栏处列队, 因此可以一致地向前移动, 这非常像 `CountDownLatch`, 只是 `CyclicBarrier` 可以多次重用

2、特征:

- `CyclicBarrier` 的 `await()` 方法使当前线程进入等待状态, 计数器自动-1(而 `CountDownLatch` 需要手动调用 `countDown()` 方法), 当计数器为 0 时, 当前线程被唤醒
- 可以向 `CyclicBarrier` 提供一个“栅栏”动作, 它是一个 `Runnable`, 当计数值到达 0 时自动执行, 这是 `CyclicBarrier` 和 `CountDownLatch` 之间的另一个区别。
- 与 `CountDownLatch` 相同, 所有并行的任务必须共享同一个 `CyclicBarrier`, 并且利用

该 CyclicBarrier 的对象调用 wait() 才能使得计数以及栅栏动作正确运转。即将所有并行的任务同步到该唯一的 CyclicBarrier 对象上

21.7.3. DelayQueue

1、介绍：

- 这是一个无界的 BlockingQueue，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
- 这种队列是有序的，即队头的对象延迟到期的时间最短/长(取决于 compareTo 的具体实现)，如果没有任何延迟到期，那么就不会有头元素，调用 poll() 会返回 null，调用 take() 会自动调用 wait() 直到对象满足超时条件
- Delayed 接口中有一个 getDelay 的方法，用来告知延迟到期的时间有多长，一般写法：

```
private final long trigger;  
trigger=System.nanoTime()+NANOSECONDS.convert(delay,MILLISECONDS);  
public long getDelay(TimeUnit unit){  
    return unit.convert(trigger-System.nanoTime(),NANOSECONDS);  
}
```

- Delayed 接口继承了 Comparable 接口，因此必须实现 compareTo()，用于产生预期的排序
- 具体的方法参考 21.5.5

2、惯用的实现：

- 定义任务实现 Runnable 与 Delayed
- 将任务的实例添加进 DelayQueue 的对象中
- 在一个线程中，在 while(!Thread.interrupted()) 中调用 DelayQueue.take().run() 即可，(DelayQueue 会自动调用 getDelay() 以检验是否超时，来选择执行或者 wait())

21.7.4. PriorityBlockingQueue

1、用于线程之间的通信

2、具体的方法参考 21.5.5

21.7.5. ScheduledExecutor

1、ScheduledThreadPoolExecutor 提供了 schedule() 以及 scheduleAtFixedRate() 方法：

2、public ScheduledFuture<?> schedule(Runnable command, long delay, TimeUnit unit)

- command：要执行的任务
- delay：时延，执行任务的时延
- unit：提供时间单位以及转换方法的工具类

3、public ScheduledFuture<?> scheduleAtFixedRate(Runnable command, long initialDelay, long period, TimeUnit unit)

- initialDley：第一次执行任务的起始时延
- period：执行任务的周期

21.7.6. Semaphore

1、正常的锁(来自 concurrent.locks 或内建的 synchronized 锁)，在任何时候都只允许一个任务访问一项资源，而计数信号量则允许 n 个任务同时访问这个资源，

2、可以将信号量看作是在向外分发使用资源的"许可证"，尽管事实上没有使用任何许可证对象

3、惯用的实现：

```
Semaphore available=new Semaphore(size,true); //放置 size 个许可证, true 什么意思？
```

- 注意，这个 **available** 和被共享的资源没有任何的关联！
- 在访问资源开始的时候调用 **available.acquire();** //当调用这个函数的时候，许可证已经发完，那么调用这个方法的线程将被阻塞。
- 在访问资源结束的时候调用 **available.release();** //多余的签入会被忽略

21.7.7. Exchanger

不懂

21.8. 概念性问题

21.8.1. 用户线程和守护线程

1、守护线程：

- 1) 守护线程在没有用户线程可服务时自动离开，在 Java 中比较特殊的线程是被称为守护(Daemon)线程的低级别线程
- 2) 这个线程具有最低的优先级，用于为系统中的其它对象和线程提供服务
- 3) 将一个用户线程设置为守护线程的方式是在线程对象创建之前调用线程对象的 **setDaemon** 方法
- 4) 典型的守护线程例子是 JVM 中的系统资源自动回收线程，我们所熟悉的 Java 垃圾回收线程就是一个典型的守护线程，当我们的程序中不再有任何运行中的 Thread，程序就不会再产生垃圾，垃圾回收器也就无事可做，所以当垃圾回收线程是 Java 虚拟机上仅剩的线程时，Java 虚拟机会自动离开。它始终在低级别的状态中运行，用于实时监控和管理系统中的可回收资源
- 5) 守护进程(Daemon)是运行在后台的一种特殊进程。它独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。也就是说守护线程不依赖于终端，但是依赖于系统，与系统"同生共死"。那 Java 的守护线程是什么样子的呢。当 JVM 中所有的线程都是守护线程的时候，JVM 就可以退出了。如果还有一个或以上的非守护线程则 JVM 不会退出。

2、Java 有两种 Thread： "守护线程 Daemon"与"用户线程 User"

- 1) 我们之前看到的例子都是用户，守护线程是一种"在后台提供通用性支持"的线程，它并不属于程序本体
- 2) 从字面上我们很容易将守护线程理解成是由虚拟机(virtual machine)在内部创建的，而用户线程则是自己所创建的。事实并不是这样，任何线程都可以是"守护线程 Daemon"或"用户线程 User"。他们在几乎各个方面都是相同的，唯一的区别是判断虚拟机何时离开：
 - 用户线程：Java 虚拟机在所有非守护线程已经离开后自动离开
 - 守护线程：守护线程则是用来服务用户线程的，如果没有其他用户线程在运行，那么就没有可服务对象，也就没有理由继续下去
- 3) **setDaemon(boolean on)**方法可以方便的设置线程的 Daemon 模式，true 为 Daemon 模式，false 为 User 模式。**setDaemon(boolean on)**方法必须在线程启动之前调用，当线程正在运行时调用会产生异常。**isDaemon** 方法将测试该线程是否为守护线程。值

得一提的是，当你在一个守护线程中产生了其他线程，那么这些新产生的线程不用设置 Daemon 属性，都将是守护线程，用户线程同样。

3、小结

- 1) setDaemon 需要在 start 方法调用之前使用
- 2) 线程划分为用户线程和后台(daemon)进程，setDaemon 将线程设置为后台进程
- 3) 如果 jvm 中都是后台进程，当前 jvm 将 exit。(随之而来的，所有的一切烟消云散，包括后台线程啦)
- 4) 主线程结束后，
 - 用户线程将会继续运行
 - 如果没有用户线程，都是后台进程的话，那么 jvm 结束

21.8.2. 生产者消费者示例

```
package org.hcf.thread;

import java.util.concurrent.*;
import java.util.*;

class Food{}

class Productor implements Runnable{
    private List<Food> list;
    private int boundary;
    private int cnt;

    Productor(List<Food> list,int boundary){
        this.list=list;
        this.boundary=boundary;
        cnt=0;
    }

    @Override
    public void run(){
        synchronized(list){
            while(true){
                if(list.size()>=boundary){
                    try{
                        TimeUnit.SECONDS.sleep(1);
                        list.wait();//挂起线程
                    }
                    catch(Exception e){
                        e.printStackTrace(System.out);
                    }
                }
            }
        }
    }
}
```

```

        for(int i=0;i<boundary;i++){
            System.out.println("生产第"+(cnt++)+"个产品中.....");
            list.add(new Food());
        }
        list.notifyAll();//唤醒等待 list 的线程，即将 wait 的线程从阻塞状态恢复到就绪状态，即该线程可以尝试获取锁
    }
}
}

class Consumer implements Runnable{
    private List<Food> list;
    private int cnt;

    Consumer(List<Food> list){
        this.list=list;
    }

    @Override
    public void run(){
        synchronized(list){
            while(true){
                if(list.size()==0){
                    try{
                        TimeUnit.SECONDS.sleep(1);
                        list.wait();//挂起线程
                    }
                    catch(Exception e){
                        e.printStackTrace(System.out);
                    }
                }
                else{
                    for(int i=list.size()-1;i>=0;i--){
                        System.out.println("消费第"+(cnt++)+"个产品中.....");
                    }
                    list.clear();
                    list.notifyAll();//唤醒等待 list 的线程，即将 wait 的线程从阻塞状态恢复到就绪状态，即该线程可以尝试获取锁
                }
            }
        }
    }
}

```

```

    }

public class ProductorConsumerModel{
    public static void main(String[] args){
        ExecutorService exec=Executors.newCachedThreadPool();
        List<Food> list=new ArrayList<Food>();
        Productor productor=new Productor(list,10);
        Consumer consumer=new Consumer(list);
        exec.execute(productor);
        exec.execute(consumer);
        try{
            TimeUnit.SECONDS.sleep(5);
        }
        catch(Exception e){
            e.printStackTrace(System.out);
        }
        System.out.println("Main Thread Stop");
    }
}

```

21.8.3. 生产消费者队列示例

```

package org.hcf.thread;

import java.util.concurrent.*;

class Productor2 implements Runnable{
    BlockingQueue<Food> bq;
    int cnt;

    Productor2(BlockingQueue<Food> bq){
        this.bq=bq;
        cnt=0;
    }
    @Override
    public void run(){
        while(true){
            try{
                bq.put(new Food());
                TimeUnit.MILLISECONDS.sleep(100);
            }
            catch(Exception e){
                e.printStackTrace(System.out);
            }
            System.out.println("生产第"+(cnt++)+"个产品中.....");
        }
    }
}

```

```
        }
    }
}

class Consumer2 implements Runnable{
    BlockingQueue<Food> bq;
    int cnt;

    Consumer2(BlockingQueue<Food> bq){
        this.bq=bq;
        cnt=0;
    }
    @Override
    public void run(){
        while(true){
            try{
                Food food=bq.take();
                TimeUnit.MILLISECONDS.sleep(100);
            }
            catch(Exception e){
                e.printStackTrace(System.out);
            }
            System.out.println("消费第"+(cnt++)+"个产品中.....");
        }
    }
}

public class BlockingQueueCommunication{
    public static void main(String[] args){
        ExecutorService exec=Executors.newCachedThreadPool();
        BlockingQueue<Food> bq=new ArrayBlockingQueue<Food>(10);
        Productor2 productor=new Productor2(bq);
        Consumer2 consumer=new Consumer2(bq);
        exec.execute(productor);
        exec.execute(consumer);
    }
}
```

Chapter 22. 设计模式

1、设计模式(Design pattern)是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。毫无疑问，设计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石，如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在中都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的核心解决方案，这也是它能被广泛应用的原因

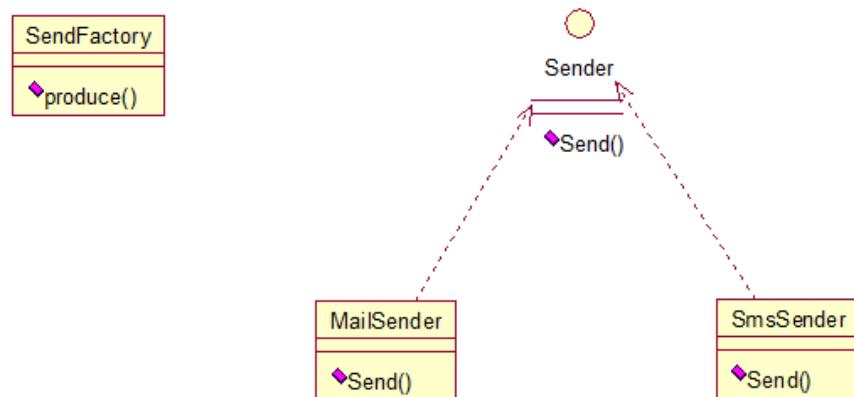
2、设计模式的分类

- 创建型模式：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式
- 结构型模式：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、享元模式
- 行为型模式：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式

22.1. 工厂方法模式 (Factory Method)

22.1.1. 普通工厂模式

1、就是建立一个工厂类，对实现了同一接口的一些类进行实例的创建



//首先，创建二者的共同接口：

```
public interface Sender {  
    public void Send();  
}
```

//其次，创建实现类：

```
public class MailSender implements Sender {  
    @Override  
    public void Send() {
```

```

        System.out.println("this is mailsender!");
    }
}

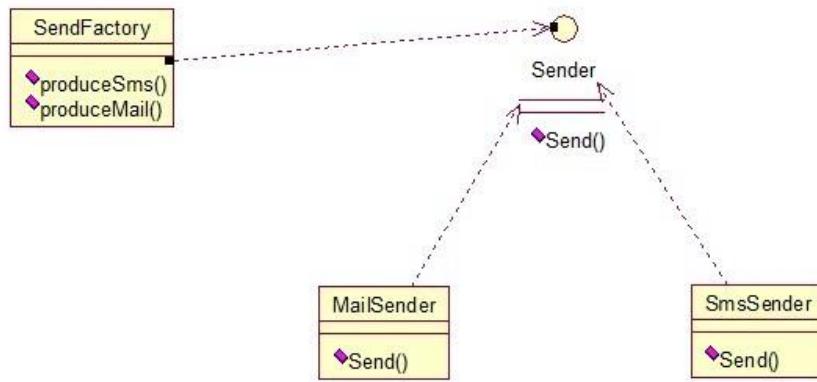
public class SmsSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}

//最后，建工厂类：
public class SendFactory {
    public Sender produce(String type) {
        if ("mail".equals(type)) {
            return new MailSender();
        }
        elseif ("sms".equals(type)) {
            return new SmsSender();
        }
        else {
            System.out.println("请输入正确的类型!");
            return null;
        }
    }
}

```

22.1.2. 多个工厂方法模式

1、是对普通工厂方法模式的改进，在普通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个工厂方法，分别创建对象



```

public class SendFactory {
    public Sender produceMail(){
        return new MailSender();
    }
}

```

```

public Sender produceSms(){
    return new SmsSender();
}
}

```

22.1.3. 静态工厂方法模式

1、将上面的多个工厂方法模式里的方法置为静态的，不需要创建实例，直接调用即可

```

public class SendFactory {
    public static Sender produceMail(){
        return new MailSender();
    }

    public static Sender produceSms(){
        return new SmsSender();
    }
}

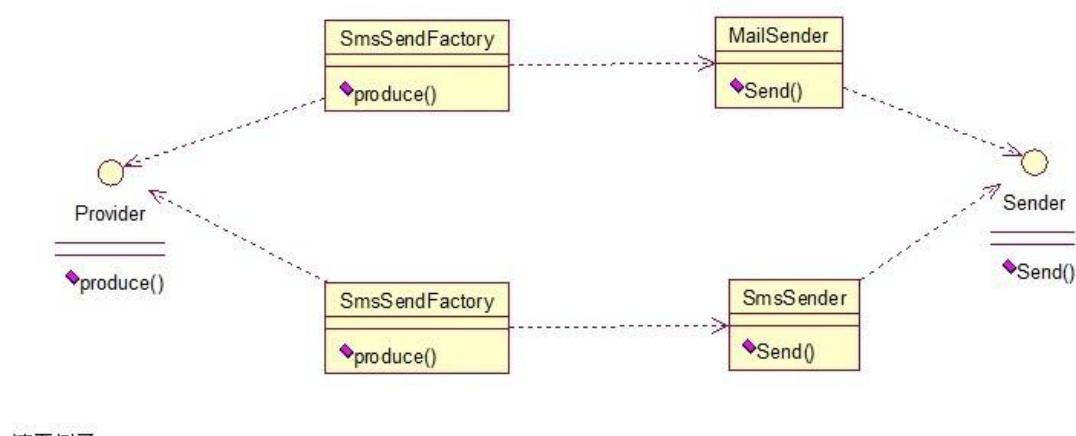
```

22.1.4. 总结

1、总体来说，工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第三种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法模式

22.2. 抽象工厂模式 (Abstract Factory)

1、工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工厂类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工厂模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之前的代码



//首先，创建二者的共同接口：

```
public interface Sender {
```

```

        public void Send();
    }

//其次，创建实现类：
public class MailSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is mailsender!");
    }
}

public class SmsSender implements Sender {
    @Override
    public void Send() {
        System.out.println("this is sms sender!");
    }
}

//工厂类接口：
public interface Provider {
    public Sender produce();
}

//两个工厂类：
public class SendMailFactory implements Provider {
    @Override
    public Sender produce(){
        return new MailSender();
    }
}

[java] view plaincopy
public class SendSmsFactory implements Provider{
    @Override
    public Sender produce() {
        return new SmsSender();
    }
}

//测试
public class Test {
    public static void main(String[] args) {
        Provider provider = new SendMailFactory();
        Sender sender = provider.produce();
        sender.Send();
    }
}

```

2、其实这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实现 Sender 接口，同时做一个工厂类，实现 Provider 接口，就 OK 了，无需去改动现

成的代码(**???main** 里面不是有 `new SendMailFactory()`，要是实现另一个工厂类，那不是这一句需要改变么`???`)。这样做，拓展性较好

22.3. 单例模式(Singleton)

1、单例对象(Singleton)是一种常用的设计模式。在 Java 应用中，单例对象能保证在一个 JVM 中，该对象只有一个实例存在。这样的模式有几个好处：

- 1) 某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销
- 2) 省去了 `new` 操作符，降低了系统内存的使用频率，减轻 GC 压力
- 3) 有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全乱了。(比如一个军队出现了多个司令员同时指挥，肯定会乱成一团)，所以只有使用单例模式，才能保证核心交易服务器独立控制整个流程

2、首先我们写一个简单的单例类

```
public class Singleton {  
    /* 持有私有静态实例，防止被引用，此处赋值为 null，目的是实现延迟加载 */  
    private static Singleton instance = null;  
  
    /* 私有构造方法，防止被实例化 */  
    private Singleton() {}  
  
    /* 静态工程方法，创建实例 */  
    public static Singleton getInstance() {  
        if (instance == null) {  
            instance = new Singleton();  
        }  
        return instance;  
    }  
  
    /* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */  
    public Object readResolve() {  
        return instance;  
    }  
}
```

3、这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环境下，肯定就会出现问题了，如何解决？我们首先会想到对 `getInstance` 方法加 `synchronized` 关键字

```
public static synchronized Singleton getInstance() {  
    if (instance == null) {  
        instance = new Singleton();  
    }  
    return instance;  
}
```

4、但是，`synchronized` 关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次调用 `getInstance()`，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，

之后就不需要了，所以，这个地方需要改进。我们改成下面这个

```
public static Singleton getInstance() {  
    if (instance == null) {  
        synchronized (instance) {  
            if (instance == null) {  
                instance = new Singleton();  
            }  
        }  
    }  
    return instance;  
}
```

5、似乎解决了之前提到的问题，将 `synchronized` 关键字加在了内部，也就是说当调用的时候是不需要加锁的，只有在 `instance` 为 `null`，并创建对象的时候才需要加锁，性能有一定的提升。但是，这样的情况，还是有可能有问题的，看下面的情况：[在 Java 指令中创建对象和赋值操作是分开进行的，也就是说 `instance = new Singleton\(\);` 语句是分两步执行的](#)。但是 JVM 并不保证这两个操作的先后顺序，也就是说有可能 JVM 会为新的 `Singleton` 实例分配空间，然后直接赋值给 `instance` 成员，然后再去初始化这个 `Singleton` 实例。这样就可能出错了，我们以 A、B 两个线程为例

- 1) A、B 线程同时进入了第一个 `if` 判断
- 2) A 首先进入 `synchronized` 块，由于 `instance` 为 `null`，所以它执行 `instance = new Singleton();`
- 3) 由于 JVM 内部的优化机制，JVM 先划出了一些分配给 `Singleton` 实例的空白内存，并赋值给 `instance` 成员(注意此时 JVM 没有开始初始化这个实例)，然后 A 离开了 `synchronized` 块
- 4) B 进入 `synchronized` 块，由于 `instance` 此时不是 `null`，因此它马上离开了 `synchronized` 块并将结果返回给调用该方法的程序
- 5) 此时 B 线程打算使用 `Singleton` 实例，却发现它没有被初始化，于是错误发生了
- 6) **我的问题：当 `instance` 为 `null` 的时候，如何使 AB 互斥??? 它连锁都还没有呢**

6、所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其是在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化

```
private static class SingletonFactory{  
    private static Singleton instance = new Singleton();  
}  
public static Singleton getInstance(){  
    return SingletonFactory.instance;  
}
```

7、实际情况是，单例模式使用内部类来维护单例的实现，[JVM 内部的机制能够保证当一个类被加载的时候，这个类的加载过程是线程互斥的](#)。这样当我们第一次调用 `getInstance` 的时候，JVM 能够帮我们保证 `instance` 只被创建一次，并且会保证把赋值给 `instance` 的内存初始化完毕，这样我们就不用担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。这样我们暂时总结一个完美的单例模式

8、其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所以说，十分完美的东西是没有的，我们只能根据实际情况，选择最适合自己应用场景的实现方法。也有人这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创

建和 getInstance() 分开，单独为创建加 synchronized 关键字，也是可以的

```
private static synchronized void syncInit() {  
    if (instance == null) {  
        instance = new SingletonTest();  
    }  
}  
public static SingletonTest getInstance() {  
    if (instance == null) {  
        syncInit();  
    }  
    return instance;  
}
```

9、总结

- 1) 单例模式理解起来简单，但是具体实现起来还是有一定的难度
- 2) synchronized 关键字锁定的是对象，在用的时候，一定要在恰当的地方使用(注意需要使用锁的对象和过程，可能有的时候并不是整个对象及整个过程都需要锁)

22.4. 建造者模式(Builder)

1、工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的 Test 结合起来得到的

2、还和前面一样，一个 Sender 接口，两个实现类 MailSender 和 SmsSender。最后，建造者类如下

```
public class Builder {  
    private List<Sender> list = new ArrayList<Sender>();  
    public void produceMailSender(int count){  
        for(int i=0; i<count; i++){  
            list.add(new MailSender());  
        }  
    }  
    public void produceSmsSender(int count){  
        for(int i=0; i<count; i++){  
            list.add(new SmsSender());  
        }  
    }  
}  
  
//测试类:  
public class Test {  
    public static void main(String[] args) {  
        Builder builder = new Builder();  
        builder.produceMailSender(10);  
    }  
}
```

3、建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工程模式的区别就是：工厂模式关注的是创建单个产品，而建造者模式则关注创建符合对象，多个部分。因此，是选择工厂模式还是建造者模式，依实际情况而定

22.5. 原型模式(Prototype)

1、原型模式虽然是创建型的模式，但是与工程模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象

2、在 Java 中，复制对象是通过 `clone()` 实现的，先创建一个原型类

```
public class Prototype implements Cloneable {  
    public Object clone() throws CloneNotSupportedException {  
        Prototype proto = (Prototype) super.clone();  
        return proto;  
    }  
}
```

3、很简单，一个原型类，只需要实现 `Cloneable` 接口，覆写 `clone` 方法，**此处 `clone` 方法可以改成任意的名称，因为 `Cloneable` 接口是个空接口，你可以任意定义实现类的方法名，如 `cloneA` 或者 `cloneB`，因为此处的重点是 `super.clone()` 这句话，`super.clone()` 调用的是 `Object` 的 `clone()` 方法，而在 `Object` 类中，`clone()` 是 native 的**

4、结合对象的浅复制和深复制来说一下，首先需要了解对象深、浅复制的概念：

- 浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的
- 深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。
 简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

5、下面给出深浅复制的例子

```
public class Prototype implements Cloneable, Serializable {  
  
    private static final long serialVersionUID = 1L;  
    private String string;  
    private SerializableObject obj;  
  
    /* 浅复制 */  
    public Object clone() throws CloneNotSupportedException {  
        Prototype proto = (Prototype) super.clone();  
        return proto;  
    }  
  
    /* 深复制 */  
    public Object deepClone() throws IOException, ClassNotFoundException {  
        /* 写入当前对象的二进制流 */  
        ByteArrayOutputStream bos = new ByteArrayOutputStream();  
        ObjectOutputStream oos = new ObjectOutputStream(bos);  
        oos.writeObject(this); // 将对象网络写入 ByteArray 流中  
        /* 读出二进制流产生的新对象 */  
    }  
}
```

```

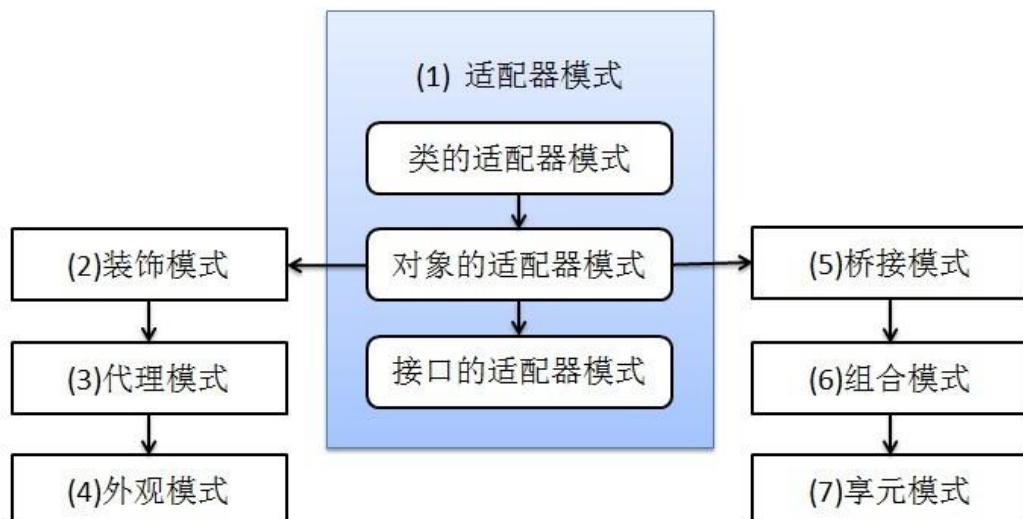
        ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
        ObjectInputStream ois = new ObjectInputStream(bis);
        return ois.readObject();
    }
    public String getString() {
        return string;
    }
    public void setString(String string) {
        this.string = string;
    }
    public SerializableObject getObj() {
        return obj;
    }
    public void setObj(SerializableObject obj) {
        this.obj = obj;
    }
}
class SerializableObject implements Serializable {
    private static final long serialVersionUID = 1L;
}

```

- 要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象

22.6. 适配器模式 (Adapter)

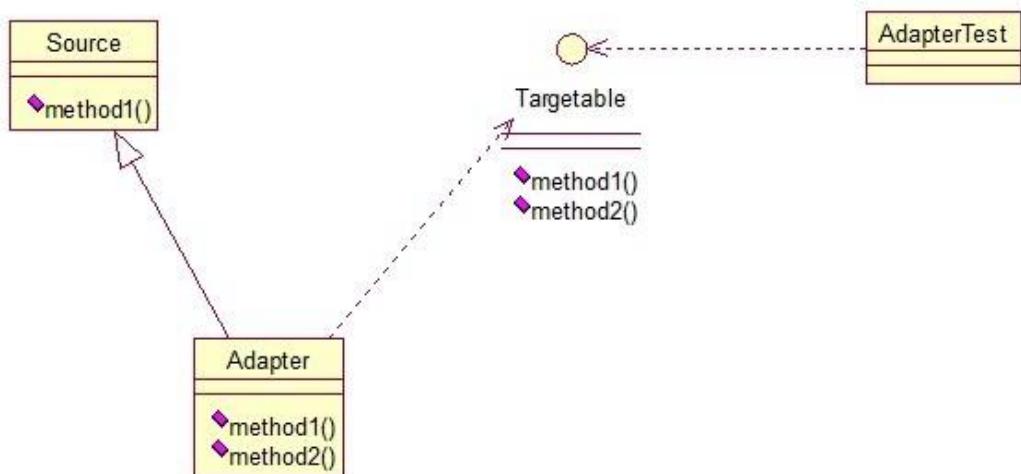
1、其中对象的适配器模式是各种模式的起源



6、适配器模式 (Adapter)

2、适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。**主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。**首先，我们来看看类的适配器模式

22.6.1. 类的适配器模式

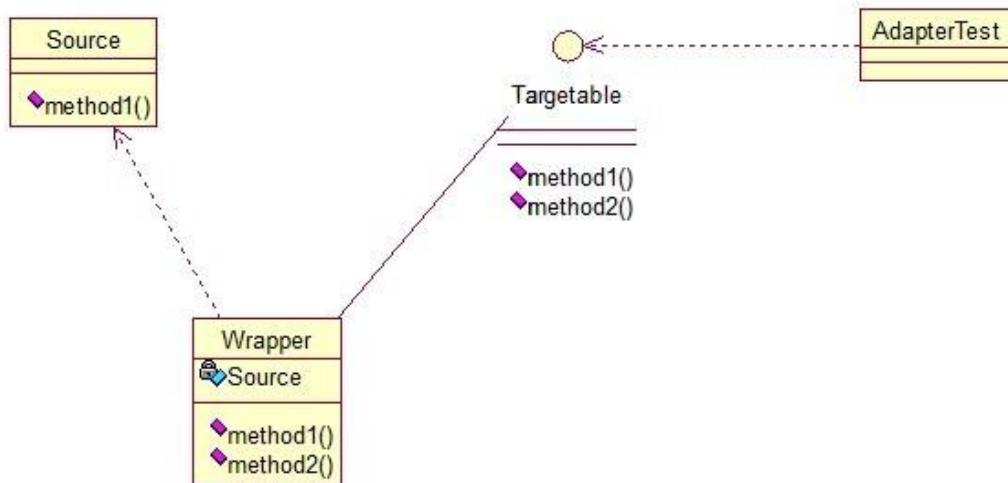


1、核心思想就是：有一个 `Source` 类，拥有一个方法，待适配，目标接口是 `Targetable`，通过 `Adapter` 类，将 `Source` 的功能扩展到 `Targetable` 里

```
public class Source {  
    public void method1() {  
        System.out.println("this is original method!");  
    }  
}  
public interface Targetable {  
    /* 与原类中的方法相同 */  
    public void method1();  
    /* 新类的方法 */  
    public void method2();  
}  
//Adapter 类继承 Source 类，实现 Targetable 接口  
public class Adapter extends Source implements Targetable {  
    @Override  
    public void method2() {  
        System.out.println("this is the targetable method!");  
    }  
}  
//测试类  
public class AdapterTest {  
    public static void main(String[] args) {  
        Targetable target = new Adapter();  
        target.method1();  
        target.method2();  
    }  
}
```

22.6.2. 对象的适配器模式

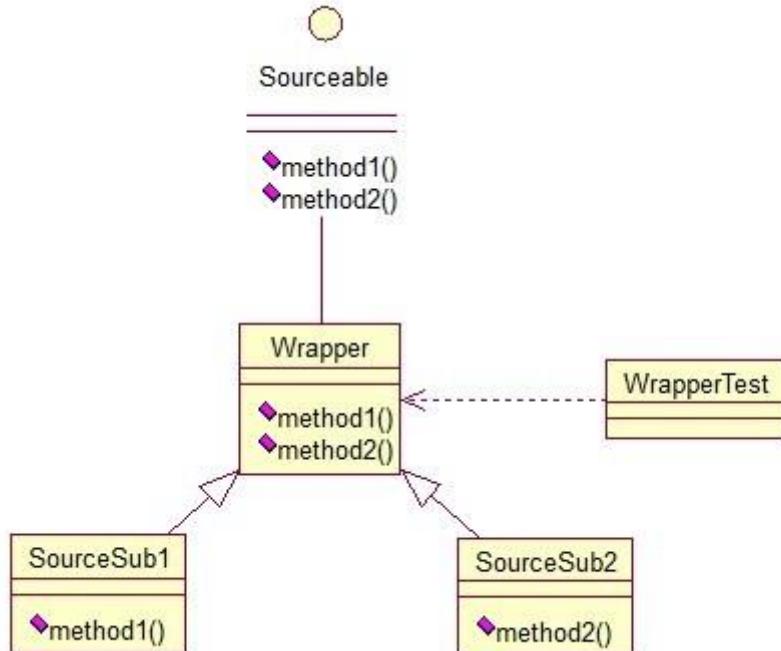
1、基本思路和类的适配器模式相同，只是将 Adapter 类作修改，这次不继承 Source 类，而是持有 Source 类的实例，以达到解决兼容性的问题



```
public class Wrapper implements Targetable {
    private Source source;
    public Wrapper(Source source){
        this.source = source;
    }
    @Override
    public void method2() {
        System.out.println("this is the targetable method!");
    }
    @Override
    public void method1() {
        source.method1();
    }
}
//测试类:
public class AdapterTest {
    public static void main(String[] args) {
        Source source = new Source();
        Targetable target = new Wrapper(source);
        target.method1();
        target.method2();
    }
}
```

22.6.3. 接口的适配器模式

1、接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，**我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法**，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图



```
public interface Sourceable {
    public void method1();
    public void method2();
}

//抽象类 Wrapper2:
public abstract class Wrapper2 implements Sourceable{
    public void method1(){}
    public void method2(){}
}

public class SourceSub1 extends Wrapper2 {
    public void method1(){
        System.out.println("the sourceable interface's first Sub1!");
    }
}

public class SourceSub2 extends Wrapper2 {
    public void method2(){
        System.out.println("the sourceable interface's second Sub2!");
    }
}

//测试类
```

```

public class WrapperTest {
    public static void main(String[] args) {
        Sourceable source1 = new SourceSub1();
        Sourceable source2 = new SourceSub2();
        source1.method1();
        source1.method2();
        source2.method1();
        source2.method2();
    }
}

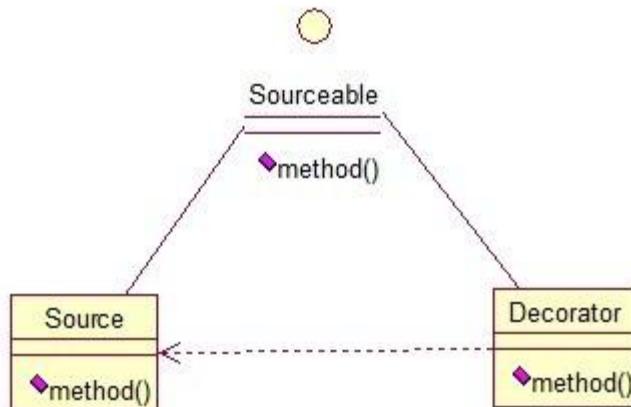
```

22.6.4. 总结

- 1、类的适配器模式：当希望将一个类转换成满足另一个新接口的类时，可以使用类的适配器模式，**创建一个新类，继承原有的类，实现新的接口即可**
- 2、对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个 **Wrapper** 类，持有原类的一个实例，在 **Wrapper** 类的方法中，调用实例的方法就行。
- 3、接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类 **Wrapper**，实现所有方法，我们写别的类的时候，继承抽象类即可

22.7. 装饰模式 (Decorator)

- 1、顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例



//Source 类是被装饰类，Decorator 类是一个装饰类，可以为 Source 类动态的添加一些功能

```

public interface Sourceable {
    public void method();
}

public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}

```

```

    }
}

public class Decorator implements Sourceable {
    private Sourceable source;
    public Decorator(Sourceable source){
        super();
        this.source = source;
    }
    @Override
    public void method() {
        System.out.println("before decorator!");
        source.method();
        System.out.println("after decorator!");
    }
}

//测试类:
public class DecoratorTest {
    public static void main(String[] args) {
        Sourceable source = new Source();
        Sourceable obj = new Decorator(source);
        obj.method();
    }
}

```

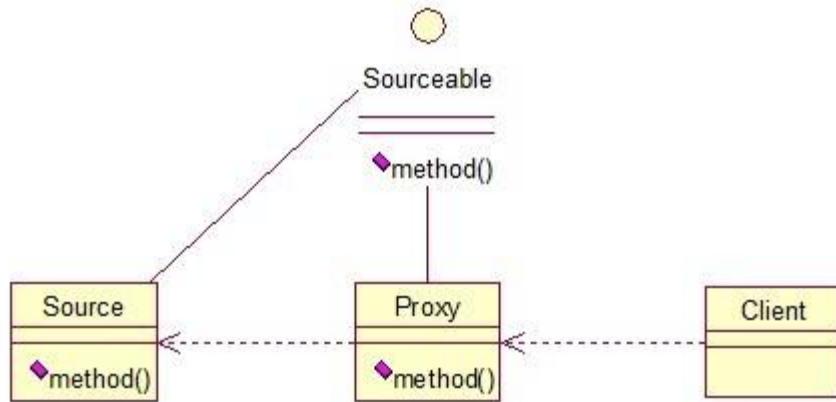
2、装饰器模式的应用场景:

- 1) 需要扩展一个类的功能
- 2) 动态的为一个对象增加功能，而且还能动态撤销(继承不能做到这一点，继承的功能是静态的，不能动态增删)

3、缺点: 产生过多相似的对象，不易排错

22.8. 代理模式(Proxy)

1、其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图



```

public interface Sourceable {
    public void method();
}

public class Source implements Sourceable {
    @Override
    public void method() {
        System.out.println("the original method!");
    }
}

public class Proxy implements Sourceable {
    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
//测试类:
public class ProxyTest {
    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }
}
  
```

```
    }  
}
```

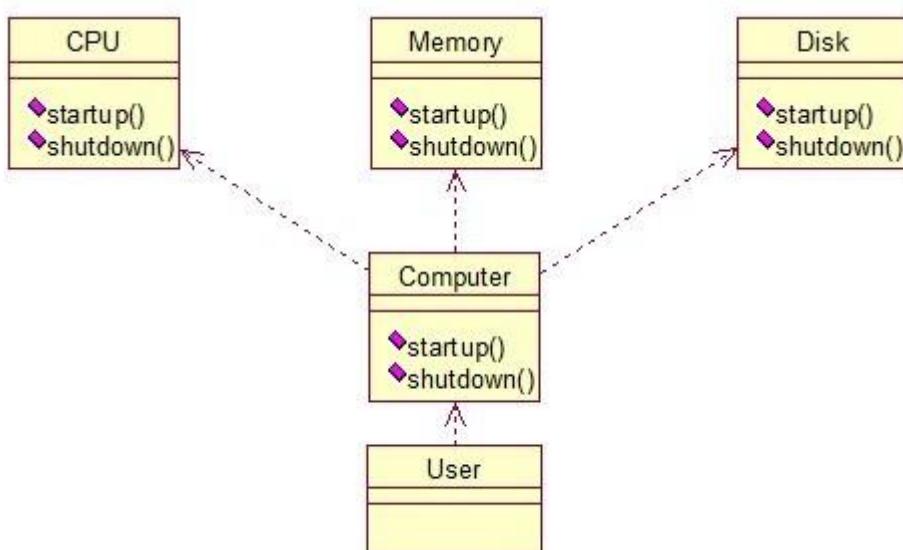
2、如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

- 1) 修改原有的方法来适应。这样违反了"对扩展开放，对修改关闭"的原则
- 2) 就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式

3、使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

22.9. 外观模式(Facade)/门面模式

1、外观模式是为了解决类与类之家的依赖关系的，像 spring 一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个 **Facade** 类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：(我们以一个计算机的启动过程为例)



```
public class CPU {  
    public void startup(){  
        System.out.println("cpu startup!");  
    }  
    public void shutdown(){  
        System.out.println("cpu shutdown!");  
    }  
}  
public class Memory {  
    public void startup(){  
        System.out.println("memory startup!");  
    }  
    public void shutdown(){  
        System.out.println("memory shutdown!");  
    }  
}
```

```

public class Disk {
    public void startup(){
        System.out.println("disk startup!");
    }
    public void shutdown(){
        System.out.println("disk shutdown!");
    }
}

public class Computer {
    private CPU cpu;
    private Memory memory;
    private Disk disk;

    public Computer(){
        cpu = new CPU();
        memory = new Memory();
        disk = new Disk();
    }
    public void startup(){
        System.out.println("start the computer!");
        cpu.startup();
        memory.startup();
        disk.startup();
        System.out.println("start computer finished!");
    }
    public void shutdown(){
        System.out.println("begin to close the computer!");
        cpu.shutdown();
        memory.shutdown();
        disk.shutdown();
        System.out.println("computer closed!");
    }
}

//User 类如下:
public class User {
    public static void main(String[] args) {
        Computer computer = new Computer();
        computer.startup();
        computer.shutdown();
    }
}

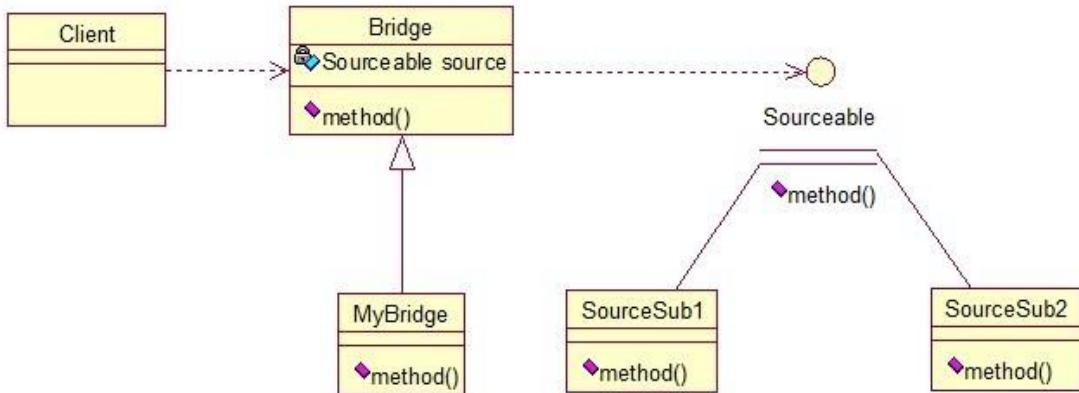
```

2、如果我们没有 Computer 类，那么，CPU、Memory、Disk 他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是我们想要看到的，有了 Computer 类，他们之间的关系被放在了 Computer 类里，这样就起到了解耦

的作用，这，就是外观模式

22.10. 桥接模式 (Bridge)

1、桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：
将抽象化与实现化解耦，使得二者可以独立变化，像我们常用的 JDBC 桥 DriverManager 一样，JDBC 进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是 JDBC 提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。我们来看看关系图



```
//先定义接口:
public interface Sourceable {
    public void method();
}

//分别定义两个实现类:
public class SourceSub1 implements Sourceable {
    @Override
    public void method() {
        System.out.println("this is the first sub!");
    }
}

public class SourceSub2 implements Sourceable {
    @Override
    public void method() {
        System.out.println("this is the second sub!");
    }
}

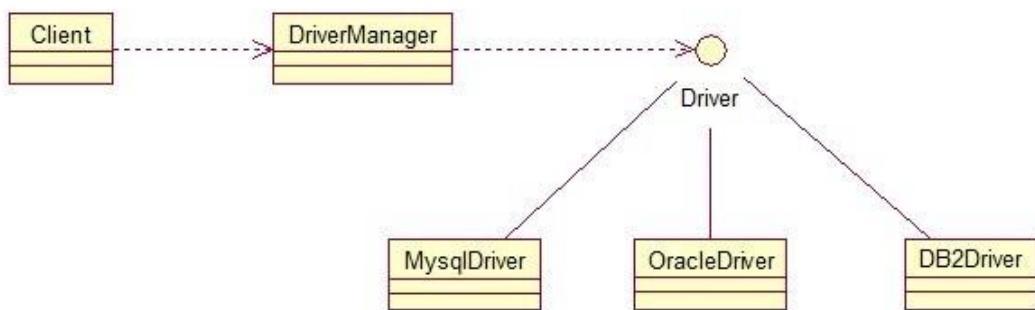
//定义一个桥，持有 Sourceable 的一个实例:
public abstract class Bridge {
    private Sourceable source;
    public void method(){
        source.method();
    }
    public Sourceable getSource() {
```

```

        return source;
    }
    public void setSource(Sourceable source) {
        this.source = source;
    }
}
public class MyBridge extends Bridge {
    public void method(){
        getSource().method();
    }
}
//测试类:
public class BridgeTest {
    public static void main(String[] args) {
        Bridge bridge = new MyBridge();
        /*调用第一个对象*/
        Sourceable source1 = new SourceSub1();
        bridge.setSource(source1);
        bridge.method();
        /*调用第二个对象*/
        Sourceable source2 = new SourceSub2();
        bridge.setSource(source2);
        bridge.method();
    }
}

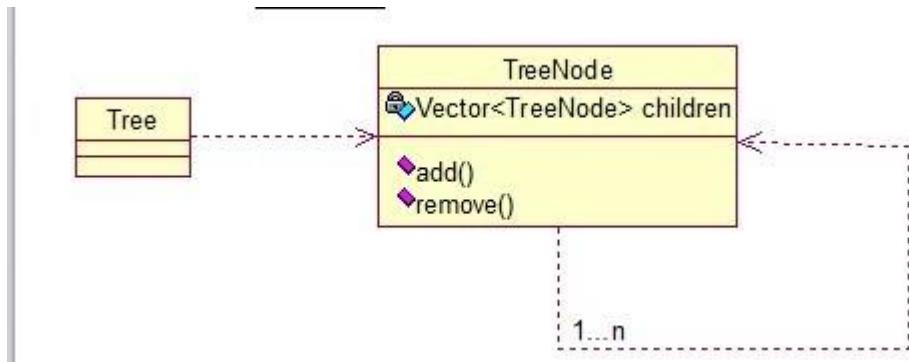
```

2、这样，就通过对 Bridge 类的调用，实现了对接口 Sourceable 的实现类 SourceSub1 和 SourceSub2 的调用。接下来我再画个图，大家就应该明白了，因为这个图是我们 JDBC 连接的原理，有数据库学习基础的，一结合就都懂了



22.11. 组合模式 (Composite)

1、组合模式有时又叫部分-整体模式在处理类似树形结构的问题时比较方便，看看关系图



```

public class TreeNode {
    private String name;
    private TreeNode parent;
    private Vector<TreeNode> children = new Vector<TreeNode>();
    public TreeNode(String name){
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public TreeNode getParent() {
        return parent;
    }
    public void setParent(TreeNode parent) {
        this.parent = parent;
    }
    //添加孩子节点
    public void add(TreeNode node){
        children.add(node);
    }
    //删除孩子节点
    public void remove(TreeNode node){
        children.remove(node);
    }
    //取得孩子节点
    public Enumeration<TreeNode> getChildren(){
        return children.elements();
    }
}
public class Tree {
    TreeNode root = null;
    public Tree(String name) {

```

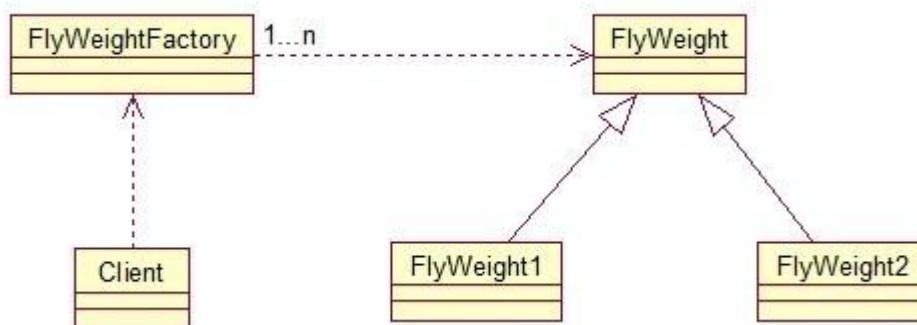
```

        root = new TreeNode(name);
    }
    public static void main(String[] args) {
        Tree tree = new Tree("A");
        TreeNode nodeB = new TreeNode("B");
        TreeNode nodeC = new TreeNode("C");
        nodeB.add(nodeC);
        tree.root.add(nodeB);
        System.out.println("build the tree finished!");
    }
}

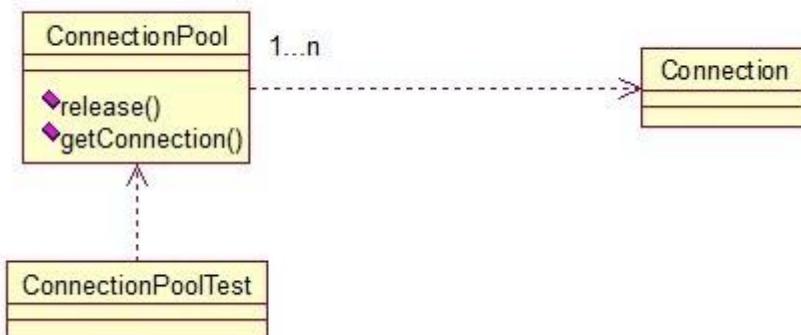
```

22.12. 享元模式 (Flyweight)

1、享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用



2、**FlyWeightFactory** 负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，**FlyWeight** 是超类。一提到共享池，我们很容易联想到 Java 里面的 JDBC 连接池，想想每个连接的特点，我们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password 及 dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量



```

public class ConnectionPool {

```

```

private Vector<Connection> pool;
/*公有属性*/
private String url = "jdbc:mysql://localhost:3306/test";
private String username = "root";
private String password = "root";
private String driverClassName = "com.mysql.jdbc.Driver";

private int poolSize = 100;
private static ConnectionPool instance = null;
Connection conn = null;

/*构造方法，做一些初始化工作*/
private ConnectionPool() {
    pool = new Vector<Connection>(poolSize);
    for (int i = 0; i < poolSize; i++) {
        try {
            Class.forName(driverClassName);
            conn = DriverManager.getConnection(url, username, password);
            pool.add(conn);
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}

/* 返回连接到连接池 */
public synchronized void release() {
    pool.add(conn);
}

/* 返回连接池中的一个数据库连接 */
public synchronized Connection getConnection() {
    if (pool.size() > 0) {
        Connection conn = pool.get(0);
        pool.remove(conn);
        return conn;
    } else {
        return null;
    }
}
}

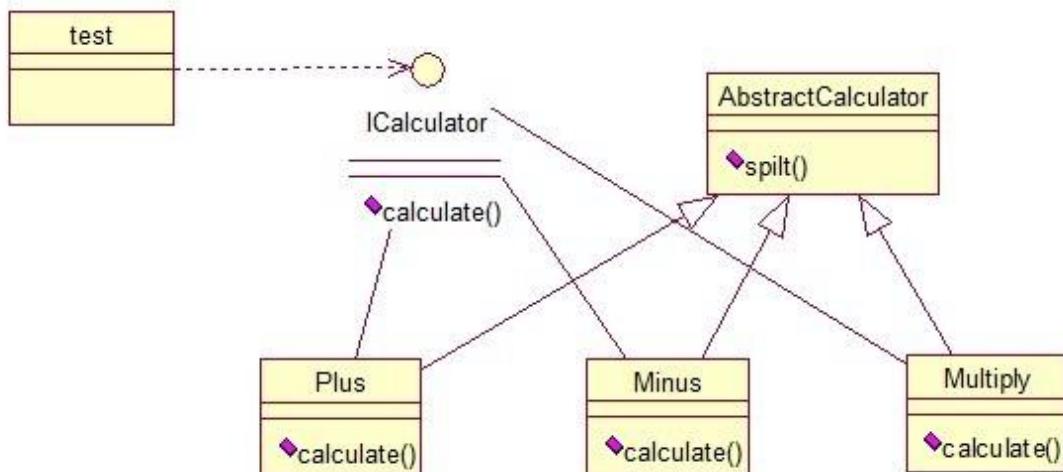
```

2、通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能

22.13. 策略模式 (strategy)



1、策略模式定义了一系列算法，并将每个算法封装起来，使他们可以相互替换，且算法的变化不会影响到使用算法的客户。需要设计一个接口，为一系列实现类提供统一的方法，多个实现类实现该接口，设计一个抽象类（可有可无，属于辅助类），提供辅助函数，关系图如下：



图中 ICalculator 提供同意的方法，
AbstractCalculator 是辅助类，提供辅助方法，接下来，依次实现下每个类：

```
//首先统一接口：
public interface ICalculator {
    public int calculate(String exp);
}

//辅助类：
public abstract class AbstractCalculator {
    public int[] split(String exp, String opt){
        String array[] = exp.split(opt);
        int arrayInt[] = new int[2];
        ...
    }
}
```

```

        arrayInt[0] = Integer.parseInt(array[0]);
        arrayInt[1] = Integer.parseInt(array[1]);
        return arrayInt;
    }
}

//三个实现类:
public class Plus extends AbstractCalculator implements ICalculator {
    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp,"\\+");
        return arrayInt[0]+arrayInt[1];
    }
}

public class Minus extends AbstractCalculator implements ICalculator {
    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp,"-");
        return arrayInt[0]-arrayInt[1];
    }
}

public class Multiply extends AbstractCalculator implements ICalculator {
    @Override
    public int calculate(String exp) {
        int arrayInt[] = split(exp,"\\*");
        return arrayInt[0]*arrayInt[1];
    }
}

//简单的测试类:
public class StrategyTest {
    public static void main(String[] args) {
        String exp = "2+8";
        ICalculator cal = new Plus();
        int result = cal.calculate(exp);
        System.out.println(result);
    }
}

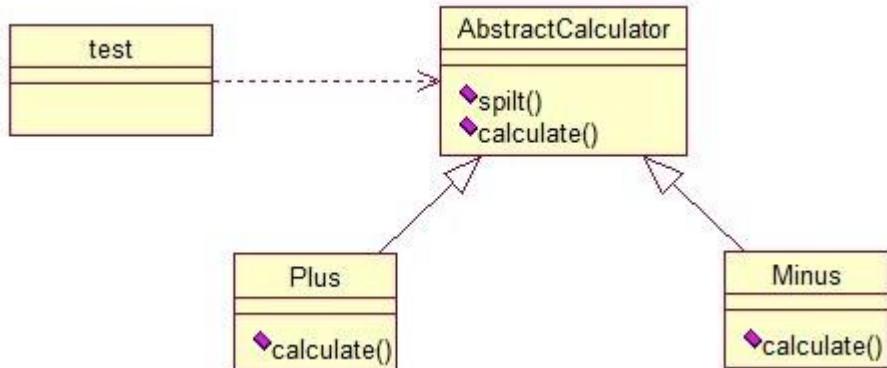
```

2、策略模式的决定权在用户，系统本身提供不同算法的实现，新增或者删除算法，对各种算法做封装。因此，策略模式多用在算法决策系统中，外部用户只需要决定用哪个算法即可。

22. 14. 模板方法模式 (Template Method)

1、解释一下模板方法模式，就是指：一个抽象类中，有一个主方法，再定义 $1...n$ 个方法，可以是抽象的，也可以是实际的方法，定义一个类，继承该抽象类，重写抽象方法，通过调

用抽象类，实现对子类的调用，先看个关系图



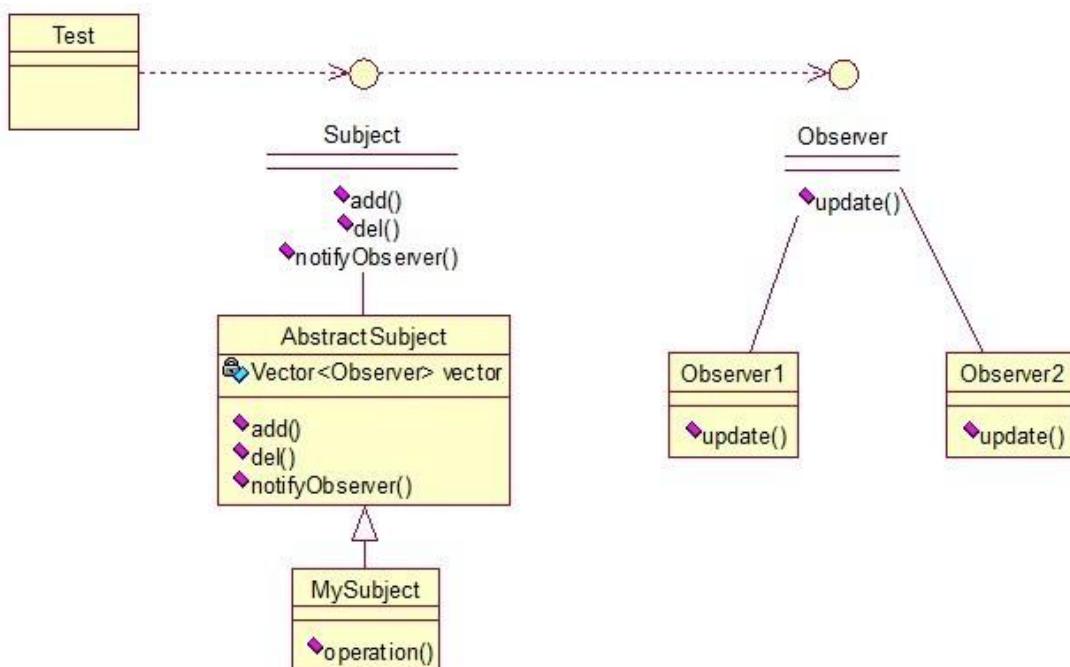
2、就是在 **AbstractCalculator** 类中定义一个主方法 **calculate**，**calculate()** 调用 **split()** 等，**Plus** 和 **Minus** 分别继承 **AbstractCalculator** 类，通过对 **AbstractCalculator** 的调用实现对子类的调用，看下面的例子：

```
public abstract class AbstractCalculator {  
    /*主方法，实现对本类其它方法的调用*/  
    public final int calculate(String exp, String opt){  
        int array[] = split(exp,opt);  
        return calculate(array[0],array[1]);  
    }  
    /*被子类重写的方法*/  
    abstract public int calculate(int num1,int num2);  
    public int[] split(String exp, String opt){  
        String array[] = exp.split(opt);  
        int arrayInt[] = new int[2];  
        arrayInt[0] = Integer.parseInt(array[0]);  
        arrayInt[1] = Integer.parseInt(array[1]);  
        return arrayInt;  
    }  
}  
public class Plus extends AbstractCalculator {  
    @Override  
    public int calculate(int num1,int num2) {  
        return num1 + num2;  
    }  
}  
//测试类：  
public class StrategyTest {  
    public static void main(String[] args) {  
        String exp = "8+8";  
        AbstractCalculator cal = new Plus();  
        int result = cal.calculate(exp, "\\+");  
        System.out.println(result);  
    }  
}
```

}

22.15. 观察者模式 (Observer)

1、包括这个模式在内的接下来的四个模式，都是类和类之间的关系，不涉及到继承，学的时候应该记得归纳，记得本文最开始的那个图。观察者模式很好理解，类似于邮件订阅和 RSS 订阅，当我们浏览一些博客或 wiki 时，经常会看到 RSS 图标，就这的意思是，当你订阅了该文章，如果后续有更新，会及时通知你。其实，简单来讲就一句话：当一个对象变化时，其它依赖该对象的对象都会收到通知，并且随着变化！对象之间是一种一对多的关系。先来看看关系图：



2、我解释下这些类的作用：MySubject 类就是我们的主对象，Observer1 和 Observer2 是依赖于 MySubject 的对象，当 MySubject 变化时，Observer1 和 Observer2 必然变化。AbstractSubject 类中定义着需要监控的对象列表，可以对其进行修改：增加或删除被监控对象，且当 MySubject 变化时，负责通知在列表内存在的对象。我们看实现代码

```
//一个 Observer 接口:  
public interface Observer {  
    public void update();  
}  
  
//两个实现类:  
public class Observer1 implements Observer {  
    @Override  
    public void update() {  
        System.out.println("observer1 has received!");  
    }  
}  
  
public class Observer2 implements Observer {  
    @Override
```

```
    public void update() {
        System.out.println("observer2 has received!");
    }
}

//Subject 接口及实现类:
public interface Subject {
    /*增加观察者*/
    public void add(Observer observer);
    /*删除观察者*/
    public void del(Observer observer);
    /*通知所有的观察者*/
    public void notifyObservers();
    /*自身的操作*/
    public void operation();
}

public abstract class AbstractSubject implements Subject {
    private Vector<Observer> vector = new Vector<Observer>();
    @Override
    public void add(Observer observer) {
        vector.add(observer);
    }
    @Override
    public void del(Observer observer) {
        vector.remove(observer);
    }
    @Override
    public void notifyObservers() {
        Enumeration<Observer> enumo = vector.elements();
        while(enumo.hasMoreElements()){
            enumo.nextElement().update();
        }
    }
}

public class MySubject extends AbstractSubject {
    @Override
    public void operation() {
        System.out.println("update self!");
        notifyObservers();
    }
}

//测试类:
public class ObserverTest {
    public static void main(String[] args) {
```

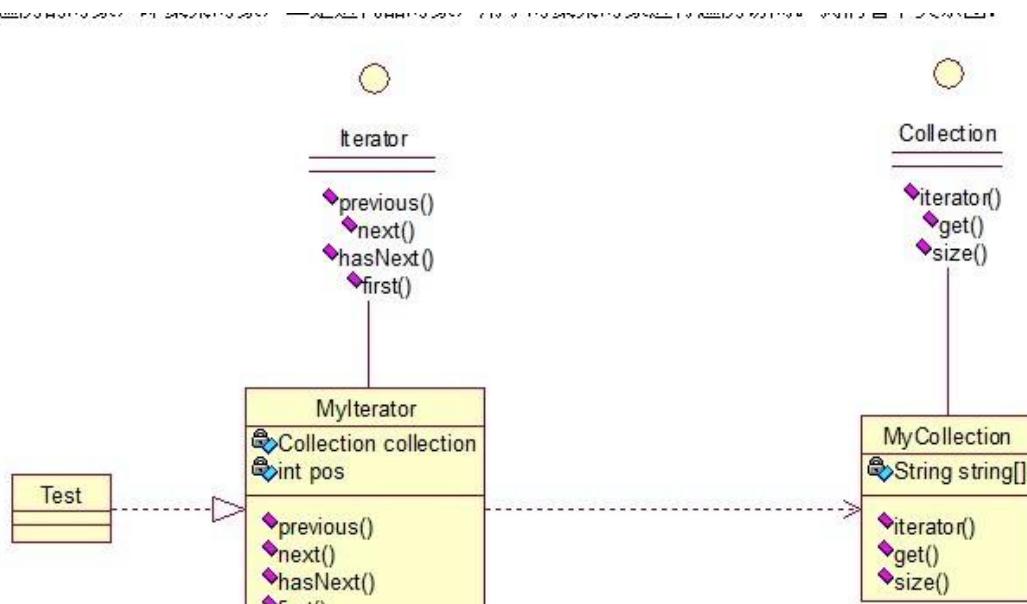
```

        Subject sub = new MySubject();
        sub.add(new Observer1());
        sub.add(new Observer2());
        sub.operation();
    }
}

```

22.16. 迭代子模式(Iterator)

1、顾名思义，迭代器模式就是顺序访问聚集中的对象，一般来说，集合中非常常见，如果对集合类比较熟悉的话，理解本模式会十分轻松。这句话包含两层意思：一是需要遍历的对象，即聚集对象，二是迭代器对象，用于对聚集对象进行遍历访问。我们看下关系图



```

//两个接口:
public interface Collection {
    public Iterator iterator();
    /*取得集合元素*/
    public Object get(int i);
    /*取得集合大小*/
    public int size();
}

public interface Iterator {
    //前移
    public Object previous();
    //后移
    public Object next();
    public boolean hasNext();
    //取得第一个元素
    public Object first();
}

```

```
//两个实现:
public class MyCollection implements Collection {
    public String string[] = {"A","B","C","D","E"};
    @Override
    public Iterator iterator() {
        return new MyIterator(this);
    }
    @Override
    public Object get(int i) {
        return string[i];
    }
    @Override
    public int size() {
        return string.length;
    }
}
public class MyIterator implements Iterator {
    private Collection collection;
    private int pos = -1;
    public MyIterator(Collection collection){
        this.collection = collection;
    }
    @Override
    public Object previous() {
        if(pos > 0){
            pos--;
        }
        return collection.get(pos);
    }
    @Override
    public Object next() {
        if(pos<collection.size()-1){
            pos++;
        }
        return collection.get(pos);
    }
    @Override
    public boolean hasNext() {
        if(pos<collection.size()-1){
            return true;
        }else{
            return false;
        }
    }
}
```

```

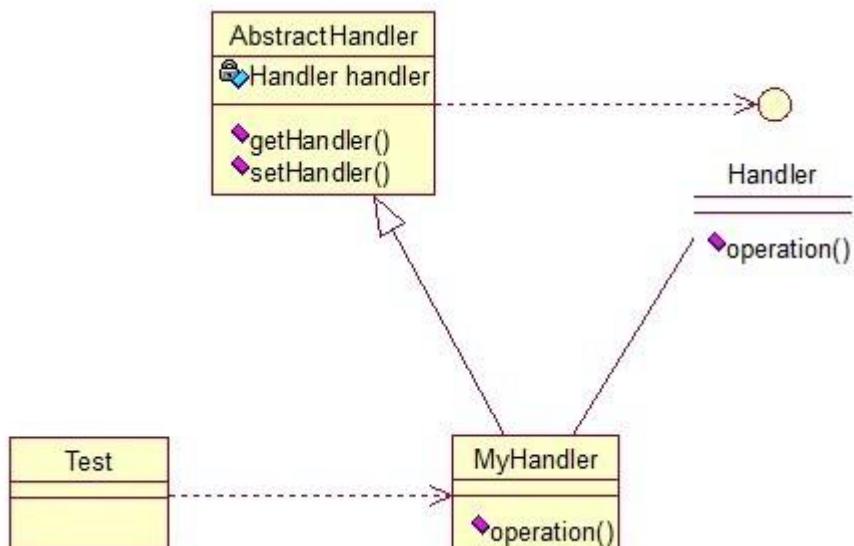
@Override
public Object first() {
    pos = 0;
    return collection.get(pos);
}
}

//测试类:
public class Test {
    public static void main(String[] args) {
        Collection collection = new MyCollection();
        Iterator it = collection.iterator();
        while(it.hasNext()){
            System.out.println(it.next());
        }
    }
}

```

22.17. 责任链模式 (Chain of Responsibility)

1、有多个对象，每个对象持有对下一个对象的引用，这样就会形成一条链，请求在这条链上传递，直到某一对象决定处理该请求。但是发出者并不清楚到底最终那个对象会处理该请求，所以，责任链模式可以实现，在隐瞒客户端的情况下，对系统进行动态的调整。先看看关系图：



2、**AbstractHandler** 类提供了 **get** 和 **set** 方法，方便 **MyHandle** 类设置和修改引用对象，**MyHandle** 类是核心，实例化后生成一系列相互持有的对象，构成一条链

```

public interface Handler {
    public void operator();
}

public abstract class AbstractHandler {
    private Handler handler;
}

```

```

public Handler getHandler() {
    return handler;
}
public void setHandler(Handler handler) {
    this.handler = handler;
}
}
public class MyHandler extends AbstractHandler implements Handler {
    private String name;
    public MyHandler(String name) {
        this.name = name;
    }
    @Override
    public void operator() {
        System.out.println(name+"deal!");
        if(getHandler()!=null){
            getHandler().operator();
        }
    }
}
public class Test {
    public static void main(String[] args) {
        MyHandler h1 = new MyHandler("h1");
        MyHandler h2 = new MyHandler("h2");
        MyHandler h3 = new MyHandler("h3");

        h1.setHandler(h2);
        h2.setHandler(h3);

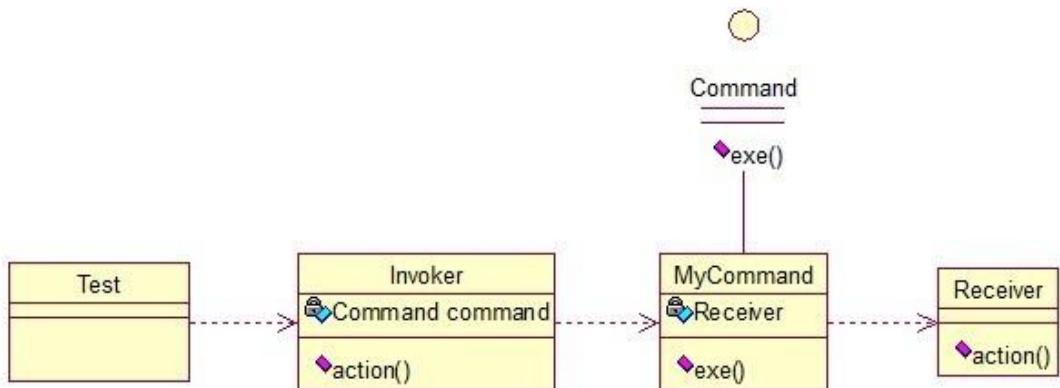
        h1.operator();
    }
}

```

2、此处强调一点就是，链接上的请求可以是一条链，可以是一个树，还可以是一个环，模式本身不约束这个，需要我们自己去实现，同时，在一个时刻，命令只允许由一个对象传给另一个对象，而不允许传给多个对象

22. 18. 命令模式 (Command)

1、命令模式很好理解，举个例子，司令员下令让士兵去干件事情，从整个事情的角度来考虑，司令员的作用是，发出口令，口令经过传递，传到了士兵耳朵里，士兵去执行。这个过程好在，三者相互解耦，任何一方都不用去依赖其他人，只需要做好自己的事儿就行，司令员要的是结果，不会去关注到底士兵是怎么实现的。我们看看关系图：



2、Invoker 是调用者(司令员), Receiver 是被调用者(士兵), MyCommand 是命令, 实现了 Command 接口, 持有接收对象, 看实现代码:

```

public interface Command {
    public void exe();
}

public class MyCommand implements Command {
    private Receiver receiver;
    public MyCommand(Receiver receiver) {
        this.receiver = receiver;
    }
    @Override
    public void exe() {
        receiver.action();
    }
}

public class Receiver {
    public void action(){
        System.out.println("command received!");
    }
}

public class Invoker {
    private Command command;
    public Invoker(Command command) {
        this.command = command;
    }
    public void action(){
        command.exe();
    }
}

public class Test {
    public static void main(String[] args) {
        Receiver receiver = new Receiver();
        Command cmd = new MyCommand(receiver);
    }
}
  
```

```

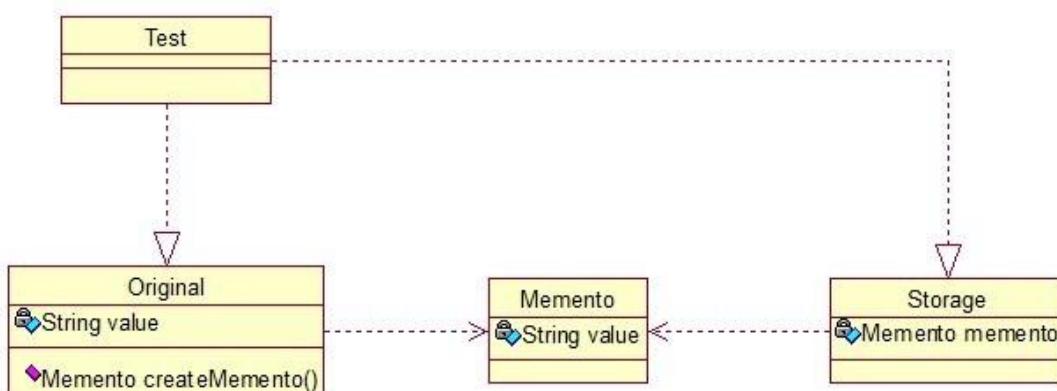
        Invoker invoker = new Invoker(cmd);
        invoker.action();
    }
}

```

2、命令模式的目的就是达到命令的发出者和执行者之间解耦，实现请求和执行分开，熟悉 Struts 的同学应该知道，Struts 其实就是一种将请求和呈现分离的技术，其中必然涉及命令模式的思想

22.19. 备忘录模式 (Memento)

1、主要目的是保存一个对象的某个状态，以便在适当的时候恢复对象，个人觉得叫备份模式更形象些，通俗的讲下：假设有原始类 A，A 中有各种属性，A 可以决定需要备份的属性，备忘录类 B 是用来存储 A 的一些内部状态，类 C 呢，就是一个用来存储备忘录的，且只能存储，不能修改等操作。做个图来分析一下



2、Original 类是原始类，里面有需要保存的属性 value 及创建一个备忘录类，用来保存 value 值。Memento 类是备忘录类，Storage 类是存储备忘录的类，持有 Memento 类的实例，该模式很好理解。直接看源码

```

public class Original {
    private String value;
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
    public Original(String value) {
        this.value = value;
    }
    public Memento createMemento(){
        return new Memento(value);
    }
    public void restoreMemento(Memento memento){
        this.value = memento.getValue();
    }
}

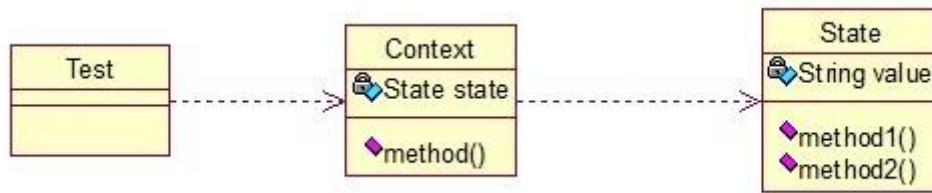
```

```
    }
}

public class Memento {
    private String value;
    public Memento(String value) {
        this.value = value;
    }
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
}
public class Storage {
    private Memento memento;
    public Storage(Memento memento) {
        this.memento = memento;
    }
    public Memento getMemento() {
        return memento;
    }
    public void setMemento(Memento memento) {
        this.memento = memento;
    }
}
//测试类:
public class Test {
    public static void main(String[] args) {
        // 创建原始类
        Original origi = new Original("egg");
        // 创建备忘录
        Storage storage = new Storage(origi.createMemento());
        // 修改原始类的状态
        System.out.println("初始化状态为: " + origi.getValue());
        origi.setValue("niu");
        System.out.println("修改后的状态为: " + origi.getValue());
        // 回复原始类的状态
        origi.restoreMemento(storage.getMemento());
        System.out.println("恢复后的状态为: " + origi.getValue());
    }
}
```

22.20. 状态模式(State)

1、核心思想就是：当对象的状态改变时，同时改变其行为，很好理解！就拿QQ来说，有几种状态，在线、隐身、忙碌等，每个状态对应不同的操作，而且你的好友也能看到你的状态，所以，状态模式就两点：1、可以通过改变状态来获得不同的行为。2、你的好友能同时看到你的变化。看图



```
public class State {
    private String value;
    public String getValue() {
        return value;
    }
    public void setValue(String value) {
        this.value = value;
    }
    public void method1(){
        System.out.println("execute the first opt!");
    }
    public void method2(){
        System.out.println("execute the second opt!");
    }
}
public class Context {
    private State state;
    public Context(State state) {
        this.state = state;
    }
    public State getState() {
        return state;
    }
    public void setState(State state) {
        this.state = state;
    }
    public void method() {
        if (state.getValue().equals("state1")) {
            state.method1();
        } else if (state.getValue().equals("state2")) {
            state.method2();
        }
    }
}
```

```

        }
    }

//测试类:
public class Test {
    public static void main(String[] args) {
        State state = new State();
        Context context = new Context(state);
        //设置第一种状态
        state.setValue("state1");
        context.method();
        //设置第二种状态
        state.setValue("state2");
        context.method();
    }
}

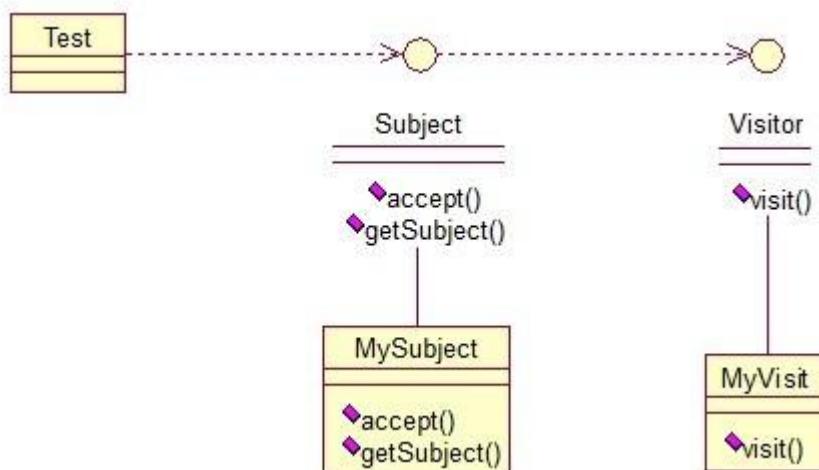
```

2、根据这个特性，状态模式在日常开发中用的挺多的，尤其是做网站的时候，我们有时希望根据对象的某一属性，区别开他们的一些功能，比如说简单的权限控制等

22.21. 访问者模式(Visitor)

1、访问者模式把数据结构和作用于结构上的操作解耦合，使得操作集合可相对自由地演化。访问者模式适用于数据结构相对稳定算法又易变化的系统。因为访问者模式使得算法操作增加变得容易。若系统数据结构对象易于变化，经常有新的数据对象增加进来，则不适合使用访问者模式。访问者模式的优点是增加操作很容易，因为增加操作意味着增加新的访问者。访问者模式将有关行为集中到一个访问者对象中，其改变不影响系统数据结构。其缺点就是增加新的数据结构很困难。

2、简单来说，访问者模式就是一种分离对象数据结构与行为的方法，通过这种分离，可达到为一个被访问者动态添加新的操作而无需做其它的修改的效果。简单关系图



```

//一个 Visitor 类(访问者), 存放要访问的对象,
public interface Visitor {
    public void visit(Subject sub);
}

```

```

}

public class MyVisitor implements Visitor {
    @Override
    public void visit(Subject sub) {
        System.out.println("visit the subject: "+sub.getSubject());
    }
}

//Subject 类, accept 方法, 接受将要访问它的对象, getSubject()获取将要被访问的属性
public interface Subject {
    public void accept(Visitor visitor);
    public String getSubject();
}

public class MySubject implements Subject {
    @Override
    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
    @Override
    public String getSubject() {
        return "love";
    }
}

//测试:
public class Test {
    public static void main(String[] args) {
        Visitor visitor = new MyVisitor();
        Subject sub = new MySubject();
        sub.accept(visitor);
    }
}

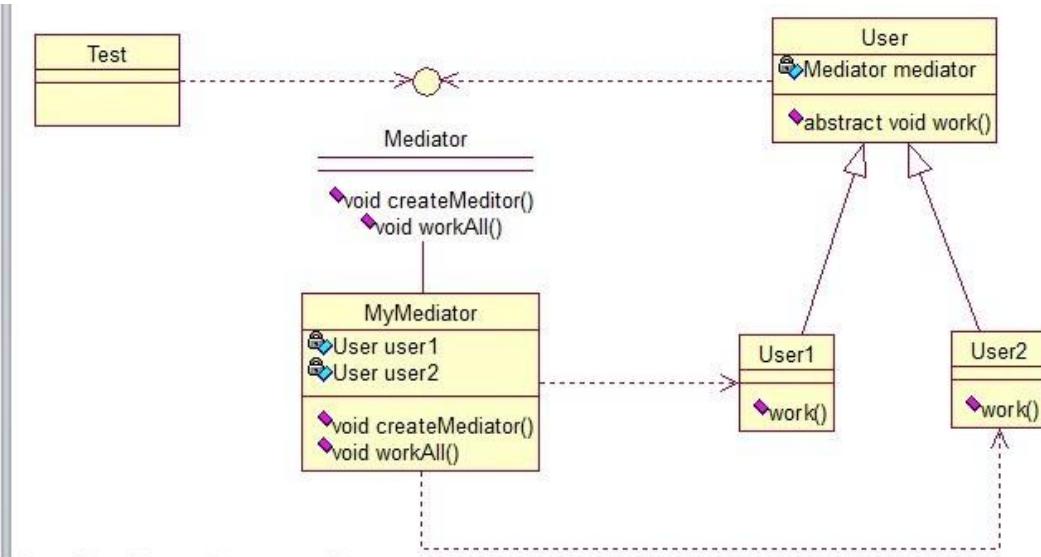
```

2、该模式适用场景：如果我们想为一个现有的类增加新功能，不得不考虑几个事情：

- 1) 新功能会不会与现有功能出现兼容性问题
 - 2) 以后会不会再需要添加
 - 3) 如果类不允许修改代码怎么办
- 面对这些问题，最好的解决方法就是使用访问者模式，访问者模式适用于数据结构相对稳定的系统，把数据结构和算法解耦

22. 22. 中介者模式 (Mediator)

1、中介者模式也是用来降低类类之间的耦合的，因为如果类类之间有依赖关系的话，不利于功能的拓展和维护，因为只要修改一个对象，其它关联的对象都得进行修改。如果使用中介者模式，只需关心和 Mediator 类的关系，具体类类之间的关系及调度交给 Mediator 就行，这有点像 spring 容器的作用。先看看图：



2、User 类统一接口，User1 和 User2 分别是不同的对象，二者之间有关联，如果不采用中介者模式，则需要二者相互持有引用，这样二者的耦合度很高，为了解耦，引入了 Mediator 类，提供统一接口，MyMediator 为其实现类，里面持有 User1 和 User2 的实例，用来实现对 User1 和 User2 的控制。这样 User1 和 User2 两个对象相互独立，他们只需要保持好和 Mediator 之间的关系就行，剩下的全由 MyMediator 类来维护！

```

public interface Mediator {
    public void createMediator();
    public void workAll();
}

public class MyMediator implements Mediator {
    private User user1;
    private User user2;
    public User getUser1() {
        return user1;
    }
    public User getUser2() {
        return user2;
    }
    @Override
    public void createMediator() {
        user1 = new User1(this);
        user2 = new User2(this);
    }
    @Override
    public void workAll() {
        user1.work();
        user2.work();
    }
}
public abstract class User {

```

```

private Mediator mediator;
public Mediator getMediator(){
    return mediator;
}
public User(Mediator mediator) {
    this.mediator = mediator;
}
public abstract void work();
}

public class User1 extends User {
    public User1(Mediator mediator){
        super(mediator);
    }
    @Override
    public void work() {
        System.out.println("user1 exe!");
    }
}

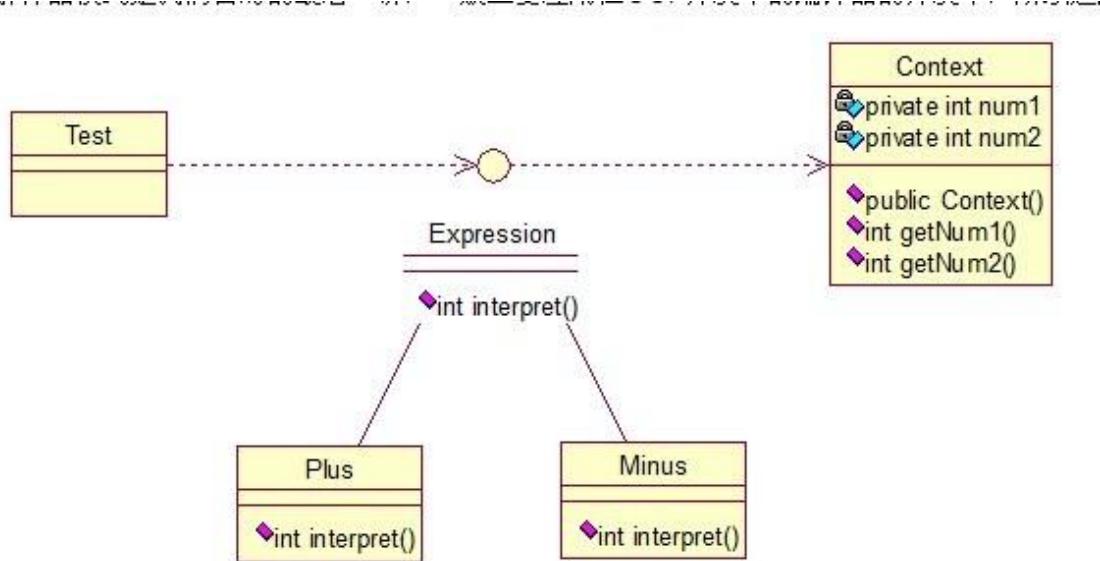
public class User2 extends User {
    public User2(Mediator mediator){
        super(mediator);
    }
    @Override
    public void work() {
        System.out.println("user2 exe!");
    }
}

//测试类:
public class Test {
    public static void main(String[] args) {
        Mediator mediator = new MyMediator();
        mediator.createMediator();
        mediator.workAll();
    }
}

```

22. 23. 解释器模式(Interpreter)

1、解释器模式一般主要应用在 OOP 开发中的编译器的开发中，所以适用面比较窄



2、Context 类是一个上下文环境类，Plus 和 Minus 分别是用来计算的实现

```

public interface Expression {
    public int interpret(Context context);
}

public class Plus implements Expression {
    @Override
    public int interpret(Context context) {
        return context.getNum1()+context.getNum2();
    }
}

public class Minus implements Expression {
    @Override
    public int interpret(Context context) {
        return context.getNum1()-context.getNum2();
    }
}

public class Context {
    private int num1;
    private int num2;
    public Context(int num1, int num2) {
        this.num1 = num1;
        this.num2 = num2;
    }
    public int getNum1() {
        return num1;
    }
    public void setNum1(int num1) {
        this.num1 = num1;
    }
    public int getNum2() {
        return num2;
    }
    public void setNum2(int num2) {
        this.num2 = num2;
    }
}

```

```
        return num2;
    }
    public void setNum2(int num2) {
        this.num2 = num2;
    }
}
public class Test {
    public static void main(String[] args) {
        // 计算 9+2-8 的值
        int result = new Minus().interpret((new Context(new Plus()
            .interpret(new Context(9, 2)), 8)));
        System.out.println(result);
    }
}
```

系统数据

System.getProperty(<String>);

Java.version	Java 运行时环境版本
java.vendor	Java 运行时环境供应商
java.vendor.url	Java 供应商的 URL
java.home	Java 安装目录
java.vm.specification.version	Java 虚拟机规范版本
java.vm.specification.vendor	Java 虚拟机规范供应商
java.vm.specification.name	Java 虚拟机规范名称
java.vm.version	Java 虚拟机实现版本
java.vm.vendor	Java 虚拟机实现供应商
java.vm.name	Java 虚拟机实现名称
java.specification.version	Java 运行时环境规范版本
java.specification.vendor	Java 运行时环境规范供应商
java.specification.name	Java 运行时环境规范名称
java.class.version	Java 类格式版本号
java.class.path	Java 类路径
java.library.path	加载库时搜索的路径列表
java.io.tmpdir	默认的临时文件路径
java.compiler	要使用的 JIT 编译器的名称
java.ext.dirs	一个或多个扩展目录的路径
os.name	操作系统的名称
os.arch	操作系统的架构
os.version	操作系统的版本
file.separator	文件分隔符(在 UNIX 系统中是"/")
path.separator	路径分隔符(在 UNIX 系统中是":")
line.separator	行分隔符(在 UNIX 系统中是"\n")
user.name	用户的账户名称
user.home	用户的主目录
user.dir	用户的当前工作目录
file.encoding	默认字符集

各种模式

装饰模式：以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案；

代理模式：给一个对象提供一个代理对象，并由代理对象来控制对原有对象的引用；

装饰模式应该为所装饰的对象增强功能；代理模式对代理的对象施加控制，并不提供对象本身的增强功能

二者的实现机制确实是一样的，可以看到他们的实例代码重复是很多的。但就语义上说，这两者的功能是相反的，模式的一个重要作用是简化其他程序员对你程序的理解

我们常常在一个代理类中创建一个对象的实例。并且，当我们使用装饰器模式的时候，我们通常的做法是将原始对象作为一个参数传给装饰者的构造器。

我们可以用另外一句话来总结这些差别：使用代理模式，代理和真实对象之间的关系通常在编译时就已经确定了，而装饰者能够在运行时递归地被构造。

策略设计模式，9.6

命令设计模式 P602

- 一般来说，命令模式首先需要一个只有一个单一方法的接口，然后从该接口实现具有各自不同行为的多个子类

职责链设计模式 P606

- 程序员以许多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，它遍历这个链，直到链中的某个方案能够处理该请求

一些问题

1、为什么说 Stack 的设计有缺陷

vector 底层用数组实现的，push、pop 性能大大降低，Stack 最好使用链表实现

当数组默认的容量发生改变时，pop、push 的性能会有较大降低；第二个：这个类是 java.util.Vector 的子类，由于是继承，stack 就会将父类的方法继承过来

如：add(int index,E element)等(具体见 Vector 的 API 说明)。这个破坏了 stack 约定的规则(只能从栈顶进，栈顶出)。所以 SUN 公司自己也给出的解释就是不要轻易用这个类。

标记：
<未完成>
<没看懂>