笔记7

第4章 对象与类

4.1 面向对象程序设计概论

java是完全面向对象的 面向对象的程序是由对象组成的 每个对象包含对用户公开的特定功能部分和隐藏的实现部分
从根本上来说 只要对象能够满足要求 就不必关心具体功能的实现过程 在OOP中 不必关心对象的具体实现 只要能够满足用户的需求即可
传统的结构化程序设计通过设计一系列的过程（即算法）来求解问题 这种工作方式是将操作数据放为第一位 然后再决定如何组织数据以便于数据操作 而在OOP中是调换了这个顺序 将数据放到第一位 然后再考虑操作数据的算法
面向对象更加适用于规模较大的问题 容易将问题简单化 更容易程序掌握 也容易找出bug 容易分级

4.1.1 类

类是构造对象的模板或蓝图 由类**构造**（construct）对象的过程称为创建**类的实例**（instance）
封装是与对象有关的一个重要概念 从形式上看 封装不过是将数据和行为组合在一个包中 并对对象的使用者隐藏了数据的实现方式 对象中的数据称为**实例域**（instance field）操作数据的过程称为**方法**（method） 对于每个特定状态的**类实例（对象）**都有一组特定的实例域值 这些值的集合就是这个对象的当前**状态**（status）
实现封装的关键在于**绝对不能**让类中的方法直接地访问其他类的实例域 程序仅通过对象的方法与对象数据进行交互 封装给对象赋予了“黑盒”概念
提高重用性和可靠性的关键 这意味着一个类可以全面地改变存储数据的方式 只要仍旧使用同样的方法操作操作数据，其他对象就不会知道或介意所发生的变化 oop的另一个原则会让自定义java类变得轻而易举 这就是 可以通过扩展一个类来建立另外一个新类 事实上 在java中 所有的类都源自于一个“神通广大的超类”它就是Object 通过扩展一个类来建立另外一个类的过程称为继承

4.1.2 对象

1.对象的行为（behavior）
2.对象的标识（state）
3.对象状态（identity）

每个对象都保存着描述当前特征的信息 这就是对象的状态 但这种改变不会是自发的 对象状态的改变必须通过调用方法实现（如果不经过方法调用就可以改变对象状态 只能说明封装性受到了破坏） 但是 对象的状态并不能完全描述一个对象 每个对象都有一个唯一的**身份（identity）**

对象的这些关键特性在彼此之间相互影响着

4.1.3 识别类

编写程序首先从设计类开始 然后往每个类 添加方法

4.1.4 类之间的关系

在类之间 最常见的关系是
1 依赖（“uses-a”）
2 聚合（‘has-a’）
3 继承（‘is-a’）

依赖是一种最明显 最常见的关系 如果一个类的方法操作另一个类的对象 我们就说一个类依赖另外一个类
应该尽可能的将相互依赖的类减至最少 换句话说 就是让耦合度减少

聚合（aggregation）是一种具体且易于理解的关系 列如 一个Order对象包含一些Item对象 聚合关系意味着类A的对象包含类B的对象

继承即是“is-a”关系 是一种表示特殊与一般关系的

4.2使用预定义类

4.2.1对象与对象变量

要想使用对象 就必须首先构造对象 并制定其初始状态 关于为什么要实现类来表示时间 而不是像其他语言内置一个bulit-in类型的原因有很多 但是主要是不好处理 有的地方使用的方式是 月/日/年 有的地方是 日/月/年 这些难以设计 可能陷入混乱 所以我们将这个任务交往实现类
一个对象变量并没有包含一个对象 而仅仅是引用一个对象

4.2.3更改器方法与访问器方法

get set和add方法在概念上有区别的 get方法仅仅查看并返回对象的状态 而set和add却对对象的状态进行修改 对实例域在做出修改的方法称为 更改器的方法（mutator method） 仅访问实例域而不进行修改的方法称为访问器方法(accessir method)

通常的习惯在访问器方法名前面加上前缀get 在更改器方法前面加上前缀set

4.3 用户自定义赛

首先需要定义的**javaBean**
在一个源文件中 只能有一个公有类 但可以有任意数目的非公有类

类通常包含类型属于某个类类型的实例域

4.3.4 从构造器开始

构造器与类同名 构造器总是伴随着new操作符的的执行被调用 其实在任何类的初始化过程中都需要视同到 而不能对一个已经存在的对象调用构造器来达到重新设置实例域的目的

切记 **不要在构造器不能定义与实例域重名的局部变量**

4.3.5 隐式参数与显示参数

```
publicvoidraiseSalary(doublebyPercent) {
doubleraris=salary*byPercent/100;
salary += raris;
}
```

raiseSalary方法有俩个参数 第一个参数是隐式参数 是出现在方法名钱的Employee类对象 第二个参数是位于方法名后面括号中的数值 这是一个显示参数

显示参数是明显地列在方法声明中 比如double byPercent 隐式参数没有出现在方法声明中 在每一个方法中 关键字this表示隐式参数

4.3.6 封装

保护了不受外部的影响 假设 name是只读域 就不会受到外部的改变 再比如salary不是只读域 但是它只能用raiseSalary进行修改 特别是一旦这个域值出现了错误 只要调试这个方法就好了 如果salary是public的 破坏这个值的捣乱者有可能出现在任何地方

在有些时候 需要获得或设置实例域的值 因此 应该提供下面三项内容

一个私有的数据域

一个公有的域访问器方法

一个公有的域更改器方法

这样做比提供一个简单的公有数据越复杂些 但是却又如下好处

首先 可以改变内部实现 除了该类的方法之外 不会影响到其他方法

更改器可以执行错误检查 然而直接对域进行赋值将不会进行这些处理

注意（在这本书中是这么建议的 但是实际开发中很少见到 或许是我见的少）

不要编写返回引用可变对象的访问器方法 如果对这个返回的对象进行使用 对其调用更改器方法就可以自动地改变这个对象的私有状态 如果需要返回的是一个可变数据域的拷贝 就应该使用Clone

4.3.8 私有方法

在实现一个方法的时候 由于公有数据域非常危险 所以将所有数据域都应该设为私有的 对于私有方法 如果改用其他方法实现相应的操作则不必保留原有方法 只要是私有的 类的设计者就该坚信 它不会被外部的其他类操作调用

4.3.9 final实例域

可以将实例域定义为final 构建对象时必须初始化这样的域 也就是说 必须确保在每一个构造器执行之后 这个域的值被设置 并且在后面的操作中 不能够再对它进行修改

final修饰符大都应用于基本（primitive）类型域或不可变（immutable）类的域（如果类中的每个方法都不会改变其对象 这种对象就是不可变的类 列如 String类就是一个不可变的类）对于可变的类 使用final修饰符可能会对读者造成困难

比如

private final Date hiredate 仅仅意味着存储在hiredate变量中的对象构造之后不能被改变 并不意味hiredate对象是一个常量 任何对象都能对hiredate调用setTime更改器

4.4 静态域与静态文件

4.4.1静态域

每个类只有一样这样的域 而每一个对象对于所有的实例域却都有自己的一份拷贝 在绝大多数的面向对象设计语言中 静态域被称为类域 术语 ‘static’只是沿用了C++的叫法 并无实际意义

4.4.2静态常量

静态变量用的比较少 但是静态常量的用的比较多

4.4.3静态方法

静态方法是一种不能向对象实施操作的方法 可以认为静态方法是不能访问ID实例域的，因为它不能操作对象 但是 静态方法可以访问自身类中的静态域 有下面俩种情况下使用静态方法

一个方法不需要访问对象状态 其所需参数都是通过显示参数提供

一个方法只需要访问类的静态域

Java中的静态域与静态方法在功能上与C++功能相同 书写有些不同

4.4.4 工厂方法

静态方法还有一种常见的用途 比如NumberFormat类使用工厂方法产生了不同风格的样式对象

4.4.5 mian方法

main方法不对任何对象进行操作 事实上在启动程序时还没有任何一个对象 静态的main方法将执行并创建程序所需要的对象

4.5 方法参数

按值调用（call by value）表示方法接受的是调用者提供的值 而**按引用调用（call by reference）**表示方法接受的是调用者提供的变量地址 一个方法可以修改传递引用的对应的变量值 而不能修改传递值调用所对应的变量值 （在很久以前 还有一种按名称调用 Algol程序语言采用的方式 但是现在已经成为历史）

java设计语言总是按值调用 也就是说 **方法得到的是所有参数值的一个拷贝** 特别是 方法不能修改传递给它的任何参数变量的内容

方法类型有俩种类型

基本数据类型（数字 布尔型）

对象引用

一个方法不可能修改一个基本数据类型的参数 而对象引用作为参数就不同了

改变对象参数状态的方法并不是一个难事 理由很简单 方法得到的是对象引用的拷贝 对象引用及其他的拷贝同时引用同一个对象

误解 很多程序员认为java程序设计语言对对象采用的是引用调用 实际上 对象引用进行的是**值传递**

总结一下java程序设计语言中方法参数的使用情况

一个方法不能修改一个基本数据类型的参数（即数值型与布尔型）

一个方法可以改变一个对象参数的状态

一个方法不能让对象参数引用一个新的对象

4.6 对象构造

java提供了多种编写构造器的方式

4.6.1重载

如果有多个方法 有相同的名字 不同的参数 便产生了重载 编译器必须挑选出具体执行哪个方法 它通过用各个方法给出的参数类型与特定方法调用所使用的值类型进行匹配来挑选相应的方法 如果编译器找不出匹配的参数 或者找出多个可能的匹配 就会产生编译时错误 （这个过程叫重载解析）

4.6.2 默认域初始化

如果在构造器中没有显示地给域赋予初值，那么就会自动地赋为默认值：数值为0，布尔值为false，对象引用为null 然而 这样是不好的 影响代码可读性 **注释：** 这是域与局域变量的主要不同点 必须明确地初始化方法中的局部变量

4.6.3 无参的构造器

如果在编写一个类时没有提供构造器 那么系统就会提供一个无参构造器 这个构造器将所有的实例域设置为默认值 于是实例域中的数值型数据设置为0 布尔型数据设置为false 所有对象变量设置为null

4.6.4 显示域初始化

由于类的构造器方法可以重载 所以可以采用多种形式设置类的实例域的初始状态 确保不管怎样调用构造器 每个实例域都可以被设置为一个有意义的初值 这是一种很好的设计习惯

可以在类定义中 直接将一个值赋给任何域 例如：

```
classEmployee2
{
```

```
privateString name = "";
//.....
}
```

在执行构造器之前 先执行赋值操作　当一个类的所有构造器都希望把相同的值赋予某个特定的实例越 这种方式特别有用

初始值不一定是常量 还可以通过方法对域进行初始化

```
classEmployee2
{
privateString name = "";
privatestaticintnxtId;
privateintanInt=ceshi();
publicintceshi(){
intr=nxtId;
nxtId++;
returnr;
   }
//.....
}
```

4.6.5 参数名

在编写很小的构造器时（这是十分常见的），常常在参数命名上出现错误 通常参数用单个字符命名 比如

```
publicSynchronizedTest(intc){
```

这样有一个缺陷 只有阅读代码才能够了解参数n和参数s的含义 于是有些程序员在每个参数前面加了一个前缀 “a”

```
publicSynchronizedTest(intaceshi){
```

这样 每一个读者一眼就能看到参数的含义 还有一种常用的技巧 它基于这样的事实 参数变量用同样的名字将实例域屏蔽起来 但是 可以用this.name 的形式来访问实例域 回想一下 this指示隐式参数 也就是被构造的对象 也就是下面这样

```
publicSynchronizedTest(intceshi) {
this.ceshi = ceshi;
}
```

4.6.6 调用另外一个构造器

关键字this引用方法的隐式参数 然而 这个关键字还有另外一个含义

如果构造器的第一个语句形如 this（...）这个构造器将调用同一个类的另一个构造器 比如下面这样

```
publicSynchronizedTest() {
this(1);
}
```

采用这种方式使用this关键字非常有用 这样对公有的构造器代码部分只需要编写一次即可

4.6.7初始化块

前面已经讲过俩种初始化数据源的方法

在构造器中设置值

在声明中赋值

Java还有第三种机制 称为**初始化块（initialization block）** 在一个类的声明中 可以包含多个代码块 只要构造类的对象 这些块就被执行 首先运行初始化块 然后才运行构造器的主体部分 这种机制不是必须的 也不常见 通常 直接将初始化代码放在构造器中

注释：即使在类的后面定义 仍然可以在初始化块中设置域 但是 为了避免循环定义 不要读取在后面初始化的域

由于初始化数据域有多种途径 所以列出构造过程的所有路径可能相当混乱 **下面是用构造器的具体处理步骤**

1 所有数据域都被初始化为默认值（0，false或null）

2 按照在类声明中出现的次序 **依次执行所有域初始化语句和初始化块**

3 如果构造器第一行调用了第二个构造器 则执行第二个构造器主体

4 执行这个构造器的主体

应该精心组织好初始化代码 这样有利于其他程序员的理解 可以通过提供一个初始化值 或者使用一个静态的初始化块来对静态域进行初始化

如果对类的静态域进行初始化的代码比较复杂 那么就可以使用静态的初始化块

在类第一次加载的时候 将会进行静态域的初始化 与实例域一样 除非将它们显示地设置成其他值 否则默认的初始值是0，false或null 所有的静态初始化语句以及静态初始化块都将依赖类定义的顺序执行

注释：

即使在类的后面定义 仍然可以在初始化块中设置域 但是 为了避免循环定义 不要读取在后面初始化的域

4.6.8 对象析构与finalize方法
java不支持析构器 有自动的垃圾回收器 不要人工回收内存
当然 某些对象使用了内存之外的其他资源 列如 文件或使用了系统资源的另一个对象的句柄 在这种情况下 当资源不再需要时 将其回收和再利用将显的十分重要
可以为如何一个类添加finalize方法 finalize方法将在垃圾回收器去清除对象之前调用 在实际应用中 不要依赖使用finalize方法回收任何短缺的资源
这是因为很难知道这个方法什么时候能够调用
注释：
有俩个方法 一个是System.runFinalizersOnexit(true)的方法能够确保finalizer方法在java关闭前调用 不过 这个方法并不安全 不鼓励使用 有一种代替的方法是使用方法Runtime.addShutdownHook添加“关闭锁”（shutdown hook）

如果某个资源需要在使用完毕后立刻被关闭 那么就需要人工管理对象 对象用完时 可以应用一个close方法来完成相应的清理操作

4.7 包
java允许使用包（backage）将类组织起来 借助于包可以方便地组织代码 并将自己的代码与别人提供的代码库分开管理
使用包的主要原因是确保类名的唯一性 Sun公司建议将公司的因特网域名（这显然是独一无二的）以逆序的形式

4.7.1 类的导入
一个类可以使用所属包的所有类 以及其他包中的公有类（public class）我们可以采用俩种方式访问另一个包中的公有类 第一种方式是在每个域名之前添加完整的包名
另外就是import语句
如果在重复使用同一个名字的情况下 发生命名冲突 可以采用特定的impact语句来解决或者在每个类名之前加上完整的类名
在包中定位是编译器的工作 类文件中的字节码肯定使用完整的包名来引用其他类

4.7.2 静态导入
import不仅可以导入类 还增加了导入静态方法和静态域的功能

```
importstaticjava.lang.System.*;
```

这样就可以使用System类的静态方法或静态域 而不必加类名前缀
实际上是否有更多的程序员采用System.out或System.exit的简写形式 ，似乎是一件值得怀疑的事情 这种编写形式不利于代码的清晰度

4.7.3 将类放进包中
要想将一个类放入包中 就必须将包的名字放在源文件的开头 包中定义类的代码之前
如果没有在源文件中放置package语句 这个源文件中的类就被放置到一个默认包（defaulf package）中 默认包是一个没有名字的包
将包中的文件放到与完整的包名匹配的子目录中
列如 ：com.xxx.corejava 中的文件就会放到 com/xxx/corejava （window中是com\xxx\corejava中） 编译器将类文件也放在相同的目录结构中
如果**没有目录结构的文件 用jdk自带的编译器编译的时候**
javac PackageTest.java 这样
如果不是默认包中的 而是将类分别放在不同的包中
javac com/xxx/corejava.java
需要注意 编译器对文件（带有文件分隔符和扩展名.java的文件）进行操作 而java解释器加载类(带有.分隔符)
警告：编译器在编译源文件的时候不会检查目录结构 列如 假定有一个源文件开头下有下列语句
package com.mycompany
即使这个源文件没有在子目录com/mycompany下 也可以进行编译 如果它不依赖其他包 就不会出现编译错误 但是 最终的程序将无法运行 这是因为虚拟机找不到其他类文件

4.7.4 包作用域
如果一个域没有被访问控制符修饰 这个（方法 类或变量）可以被同一个包中的所有方法访问 因此变量必须显示的标记为private 不然的话将默认为包可见 显然 这样做会破坏封装性 问题主要出于人们经常忘记关键字private

或许对此有些疑问 答案是：视具体情况而定 在默认情况下 包不是一个封闭的实体 也就是说 任何人都可以向包中添加更多的类 当然 有敌意或低水平的程序员很可能利用包的可见性添加一些具体修改变量功能的代码

从JDK1.2版开始 JDK的实现者修改了类加载器 **明确禁止加载用户自定义的，包名以“java.”开始的文件**

```
"C:\Program ...
Exception in thread "main" java.lang.SecurityException: Prohibited package name: java
```

当然 用户自定义的类无法从这种保护中受益 然而 可以通过**包密封**机制来解决将各种包混杂一起的问题 如果将一个包密封起来 就不能再向这个包添加类

4.8 类路径（这些东西由于IDE的存在很少使用到了）
类存储在文件系统的子目录中 类的路径必须与包名匹配

类文件也可以存储在JAR（java归档）文件中，这一个JAR文件中，可以包含多个压缩形式的类文件与子目录 这样既可以节省又可以改善性能 在程序中用到第三方的库文件时 通常会给出一个或多个需要包含的JAR文件
提示：JAR文件采用的ZIP格式组织文件和子目录 可以使用所有ZIP实用程序查看内部的rt.jar以及其他JAR文件
为了使类能被多个程序共享 需要如下几点
1把类放到一个目录中 列如/home/user/classdir 需要注意 这个目录是包树状结构的基目录
如果希望将com.xxx.corejava.Employee类添加到其中 这个Employee.class 类文件就必须位于子目录/home/user/classdir/com/xxx/corejava中

2 将JAR文件放在一个目录中 列如是：/home/user/archives
3设置类路径（class path）类路径是所有包含类文件的路径的集合
在UNIX环境中 类路径的不同项目之间采用冒号（：）分割
在window环境中 则以分号（；）分割

在上述俩种情况 句号(.)表示当前目录

从JAVASE6开始 可以向JAR文件目录中指定通配符
但是在UNIX中 禁止使用*防止Shell命令进一步扩展
在归档目录中所有JAR文件（但不包括.class文件）都包含在类路径中
由于运行时库文件会被自动搜索 所以不必将它们显示地列在类路径中
类路径所列出的目录和归档文件是搜索类的初始点 编译器的任务不止这样 它还要查看源文件是否比类文件新 如果是这样的话 源文件就会自动的重新编译 一个源文件只能包含一个公有类 并且文件名必须与公有类匹配 因此 编译器很容易定位公有类所在的源文件

设置类路径
最好采用-classpath（或-cp）选项来指定类路径
java -classpath /home/user/classdir:.:/home/user/archive.jar MyProg
或者
java -classpath c:\classdir;.;\archives\archive.jar MyProg
利用classpath选项设置类路径是首选的方法 也可以通过设置CLASSPATH环境比那里完成这个操作 其详细情况依赖于所使用的Shell 在Bourne Again shell（bash）中 命令格式如下
：export CLASSPATH=/home/user/classdir;.;/home/user/archives/archive.jar
在C shell中 命令格式如下：
setenv CLASSPATH /home/userclassdir；.;/home/use/archives/archive.jar
在windows shell 命令如下
set Classpath=c:\classdir;.;c:\archives\archive.jar
知道退出shell 类路径设置均有效

4.9 注释的插入
JDK包包含一个很有用的工具 叫javadoc 它可以由源文件生成一个HTML文件 由于文档注释与源代码在一个文件中 在修改源代码的同时重新运行javadoc就可以轻而易举的保持俩者的一致性

4.9.1 注释的插入
包
公有类与接口
公有和受保护的构造器和方法
公有的和受保护的域

注释以/**开始 并以*/结束 每个/**...*/文档注释在标记之后紧跟着**自由格式文本**
自由格式文本第一句应该是一个概要性的句子 javaDoc实用程序自动地将这些句子抽取出来形成概要页
在自由格式文本中 可以使用HTML语句

4.9.2类注释
类注释必须放在import语句之后 类定义之前

4.9.3 方法注释
每一个方法注释必须放在所描述方法之前 除了通用标记之外 还可以使用如下标记：
@param变量描述

@return描述

@throws类描述

4.9.4 域注释

4.9.5 通用注解

@author 姓名
这个标记将产生了一个‘author’（作者）条目
@version 文本

这个标记将产生了一个‘vsersion’（版本）条目
@since 文本
这个标记将产生了一个“since”（始于）条目
@deprecated 文本
这个标记将对类，方法或变量添加一个不再使用的注释 通过@see和@link标记 可以使用超级链接 链接到Javadoc文档的相关部分或外部文档
@see 引用
这个标记将在‘see also’部分增加一个超级链接
@see com.horstmann.corejava.Employee#raiseSalary(double) 比如这个就是链接到com.horstmann.corejava.Employee#raiseSalary(double) 方法的超链接 可以省略包名 和类名都省去 此时 链接将定位与当前包或当前类
需要注意 一定要用井号（#） 而不是使用句号（.）分割类名与方法名，或类名与变量名
Java编译器本身可以熟练地断定句点在分割包，子包，类，内部类与方法和变量时的不同含义 但是javadoc实用程序就没有这么聪明了 因此必须对它提供帮助
如果@see 标记后面有一个<字符 就需要指定一个超链接 可以超链接到任何URL 列如
@see baidu
在上述的各种情况 都可以指定一个可选的标签（label）作为链接锚（link anchor）如果省略label 用户看到的锚的名称就是，目标代码名或URL
如果@see标记后面有一个双引号（"）字符 文本就会显示在“see also”部分 比如：@see “Core java2 vlume2” 可以为一个特性添加多个@see标记 但必须将它们放在一起
如果愿意的话 还可以在注释的任何位置放置指向其他类或方法的超级链接 以及插入一个专用的标记 列如
{@link package.class#feature label} 这里的特性描述规则与@see标记规则一样

4.9.6 包与概述注释
可以直接将类，方法和变量的注释放置在java源文件中 只要用/**...*/文档注释界定就可以了 但是 要想产生包注释 就需要在每一个包目录中添加一个单独的文件
1 提供一个以package.html命名的HTML文件 在标记<body>...</body>之间的所有文本都会被抽取出来
2 提供一个以package-info.java命名的java文件 这个文件必须包含一个初始的以/**和*/界定的javadoc注释 跟随在一个包语句之后 它不应该包含更多的代码和注释
还可以为所有的源文件提供一个概述性的注释 这个注释将被放置在一个名为overview.html文件中 这个文件位于包含所有源文件的父目录中 标记为<body>...</body>之间的所有内容将被抽取出来当用户从导航栏中选择“Overview”时，就会显示出这些注释内容

4.9.7 注释的抽取
在**百度百科上有具体步骤** 同时可以对多种形式命令行选项对javadoc进行调整 例如 可以使用-author和@version标记（默认情况下 这些标记会被省略） 另一个很有用的选项是 -link 用来为标准类添加超链接
例如：javadoc -link http://docs.oracle.com/javase/7/docs/api*.java 那么 所有的标准类库都会自动地链接到Oracle网站的文档
如果使用-linksource选项 则每个源文件都被转换为HTML（不对代码着色 单包含行编号）并且每个类和方法名都将转变为指向源代码的超链接
4.10 类设计技巧
1 一定要保持数据私有
2 一定要对数据初始化
3 不要在类中使用过多的基本类型
4 不是所有的域都需要独立的域访问器和域更改器
5 将职责过多的类进行分解
6类名和方法名要能体现它们的职责 命名类名的良好习惯是采用一个名词（Order），前面有形容词（RushOrder）修饰的名词或动名词（有“-ing”后缀）修饰名词（列如 BillingAddress）对于方法来说 习惯是访问器方法用小写get开头更改器方法用小写set开头