## 软件需求分析之类型一致性和OCP原则

黄永涛 41912210

#### 一、类型一致性原则

1 类class和类型type

从抽象的角度来理解，最好将类视为类型的实现。也就是说类型包括了类的目标以及类的状态空间和行为。实际上，一个类型可以有多个类，每个类都包括自己独立的内部设计。一个典型的例子是STL链表类型，在GNU实现中的list类与windows实现的list类，二者拥有相同的外部特性，但在内部实现上却有着不同。

注意子类型与子类的区别，在下一节将给出子类型的定义。从语法上讲，我们可以将任意一个类定义为另一类的子类，但从语义上讲，它们没有任何意义，因为这些子类和其相应的父类之间没有任何关系。因此，即使s是t的子类(派生类)，并不意味着s是t的子类型。

2 类型一致原则

类型一致性原则来源于抽象数据类型原理，此原理是面向对象的基础。类型一致性原则对于创建类库的类层次结构至关重要，在某些书中也叫为liskov替换原则，此原则的表达为：如果s为t的真子类型，则s必须与t一致，即类型s的对象可以出现在类型t的对象所需要的任何环境中，并且当此对象的任何获取操作执行时，仍能保证其正确性。

在完善的面向对象的设计中，每个类的类型必须与其基类相一致，即类或派生类的继承层次结构必须遵循类型一致原则。要这个做的原因在于，为了毫不费力的利用多态性，我们必须能传递子类对象以代替基类对象。

2.1 抗变性与协变性原则

为了保证子类的类型与基类保持一致，首先要保证子类的不变式至少和基类的不变式一样强。

其次还需要满足下列三个操作限制条件：

a 每个基类的操作必须与其子类中一个操作相对应，它们具有相同的名字和函数原型。

b 抗变原则：每个子类操作的前置条件弱于或等于基类操作的前置条件

c 协变原则：每个子类操作的后置条件强于或等于基类操作的后置条件

2.2 抗变性和协变性的实例

雇员类的一个派生类是经理类，但如何才能保证经理类是雇员的真子类呢？

假设雇员的不变式是gradLevel>0，而经理的不变式为gradLevel>20，这就保证了经理类的不变式比雇员类的不变式强，从而满足了要求。

其次，两个类都需要计算红利，基类和派生类在计算时只是输入参数范围不同。如果基类的输入参数为[0,5]，输出参数为[0,10]，则派生类经理的计算操作的前置条件要更弱，即输入参数若比(0,5)更小是不合法的。而后置条件应该更强，应该在(0,10)之间。

3 闭合行为原则

在类型一致的情况下，仅在只读情况下即仅执行获取操作时，才能得到完善的设计。如果处理的情况中执行了修改操作，我们还需要遵守闭合行为原则。此原则要求，子类从基类继承的行为必须符合子类的不变式。

在基于类型/子类型层次结构的继承层次结构中，派生类C的任何操作，包括从基类继承的操作，均应该满足C的类不变式。

下面举个例子来说明对这个原则的理解。基类为多边形，派生类为四边形。多边形的操作中有一个是move，即将图形整个移动到另一个位置。这个操作对四边形而言，移动之后还是四边形，因此这个操作是闭合的。而另一个操作是add\_vertex，即为多边形增加一个顶点。如果四边形执行这个操作，则其不再是四边形了。因此add\_vertex操作并不是闭合的。

从上面的例子中我们看到，基类行为中的子类闭合并不是自动产生的，我们必须将其设计进来。作为派生类的设计者，必须深思熟虑，并明确覆盖其基类的操作，否则将违背派生类的不变式。

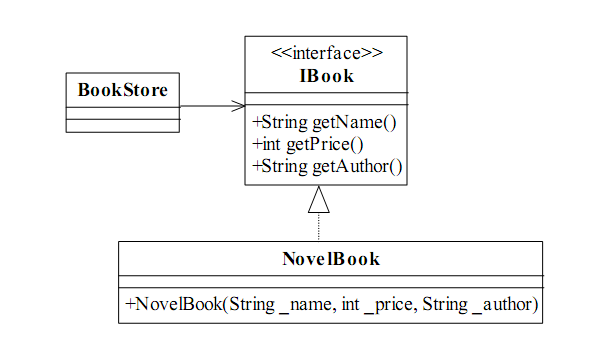
#### 开闭原则

1. 定义

software entities like classes, [module](https://so.csdn.net/so/search?q=module" \t "https://blog.csdn.net/sinat_20645961/article/details/_blank)s and functions should be open for extension but closed for modifications.  
一个软件实体应该对扩展开放，对修改关闭。

2.什么是[开闭原则](https://so.csdn.net/so/search?q=%E5%BC%80%E9%97%AD%E5%8E%9F%E5%88%99" \t "https://blog.csdn.net/sinat_20645961/article/details/_blank)

我们思考这样一个问题：一个软件产品只要在生命期内，都会发生变化，变化既然是一个既定的事实，我们就应该在设计时候尽量适应这些变化，以提高项目的稳定性和灵活性，真正实现“拥抱变化” ，开闭原则告诉我们通过尽量通过扩展软件实体的行为来实现变化，而不通过修改来已有的代码来完成变化，它是为软件实体的未来事件而制定的对现行开发设计进行约束的一个原则，我们举例什么是开闭原则，以书店销售书籍为例，类图如下：



IBook 是定义了数据的三个属性：名称、价格和作者，小说类 NovelBook 是一个具体的实现类，所有小说书籍的总称，BookStore 指的是书店，我们先来看 **IBook接口:**

public interface IBook {

//书籍有名称

public String getName();

//书籍有售价

public int getPrice();

//书籍有作者

public String getAuthor();

}

**小说书籍的源代码如下：**

public class NovelBook implements IBook {

//书籍名称

private String name;

//书籍的价格

private int price;

//书籍的作者

private String author;

//通过构造函数传递书籍数据

public NovelBook(String \_name,int \_price,String \_author){

this.name = \_name;

this.price = \_price;

this.author = \_author;

}

//获得作者是谁

public String getAuthor() {

return this.author;

}

//书籍叫什么名字

public String getName() {

return this.name;

}

//获得书籍的价格

public int getPrice() {

return this.price;

}

}

**然后我们看书店是怎么销售书籍的：**

public class BookStore {

private final static ArrayList<IBook> bookList = new ArrayList<IBook>();

//静态模块初始化，项目中一般是从持久层初始化产生

static{

bookList.add(new NovelBook("天龙八部",3200,"金庸"));

bookList.add(new NovelBook("巴黎圣母院",5600,"雨果"));

bookList.add(new NovelBook("悲惨世界",3500,"雨果"));

bookList.add(new NovelBook("金瓶梅",4300,"兰陵笑笑生"));

}

//模拟书店买书

public static void main(String[] args) {

NumberFormat formatter = NumberFormat.getCurrencyInstance();

formatter.setMaximumFractionDigits(2);

System.out.println("------------书店买出去的书籍记录如下：---------------------");

for(IBook book:bookList){

System.out.println("书籍名称：" + book.getName()+"\t书籍作者：" +

book.getAuthor()+ "\t书籍价格：" + formatter.format(book.getPrice()/100.0)+"元");

}

}

}

注意，我们在 BookStore 中声明了一个静态模块，实现了数据的初始化，这部分应该是从持久层产生的，由持久层工具进行管理。运行结果如下：

------------书店买出去的书籍记录如下：---------------------

书籍名称：天龙八部 书籍作者：金庸 书籍价格：￥32.00元

书籍名称：巴黎圣母院 书籍作者：雨果 书籍价格：￥56.00元

书籍名称：悲惨世界 书籍作者：雨果 书籍价格：￥35.00元

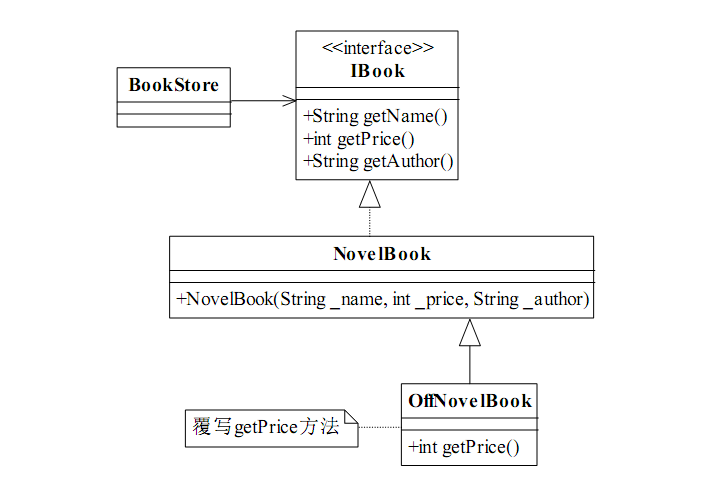
书籍名称：金瓶梅 书籍作者：兰陵笑笑生 书籍价格：￥43.00元

项目投产了，书籍正常销售出去，书店也盈利了。从 2008年开始，全球经济都开始下滑，对零售业影响还是比较大，书店为了生存开始打折销售：所有 40 元以上的书籍 9折销售，其他的 8 折销售。对已经投产的项目来说，这就是一个变化，我们来看看这样的一个需求变化，我们该怎么去应对，有三种方法可以解决这个问题：

**修改接口**。 在 IBook 上新增加一个方法 getOffPrice()， 专门进行打折处理， 所有的实现类实现该方法。但是这样修改的后果就是实现类 NovelBook 要修改,BookStore 中的main方法也修改， 同时 IBook作为接口应该是稳定且可靠的，不应该经常发生变化，否则接口做为契约的作用就失去了效能，——因此，该方案否定。

**修改实现类**。修改 NovelBook 类中的方法，直接在 getPrice()中实现打折处理，好办法，我相信大家在项目中经常使用的就是这样办法，通过 class 文件替换的方式可以完成部分业务（或是缺陷修复）变化，该方法在项目有明确的章程（团队内约束）或优良的架构设计时，是一个非常优秀的方法，但是该方法还是有缺陷的，例如采购书籍人员也是要看价格的，由于该方法已经实现了打折处理价格，因此采购人员看到的也是打折后的价格，这就产生了信息的蒙蔽效果，导致信息不对称而出现决策失误的情况。——因此，该方案也不是一个最优的方案。

**通过扩展实现变化**。增加一个子类 OffNovelBook，覆写 getPrice 方法，高层次的模块（也就是 static静态模块区）通过 OffNovelBook 类产生新的对象，完成对业务变化开发任务。——好办法，修改也少，风险也小，我们来看类图：



OffNovelBook 类继承了NovelBook，并覆写了 getPrice 方法，不修改原有的代码。我们来看**新增加的子类** **OffNovelBook**:

public class OffNovelBook extends NovelBook {

public OffNovelBook(String \_name,int \_price,String \_author){

super(\_name,\_price,\_author);

}

//覆写销售价格

@Override

public int getPrice(){

//原价

int selfPrice = super.getPrice();

int offPrice=0;

if(selfPrice>4000){ //原价大于40元，则打9折

offPrice = selfPrice \* 90 /100;

}else{

offPrice = selfPrice \* 80 /100;

}

return offPrice;

}

}

很简单，仅仅覆写了 getPrice 方法，通过扩展完成了新增加的业务。然后我们来看 BookStore 类的修改：

public class BookStore {

private final static ArrayList<IBook> bookList = new ArrayList<IBook>();

//静态模块初始化，项目中一般是从持久层初始化产生

static{

bookList.add(new OffNovelBook("天龙八部",3200,"金庸"));

bookList.add(new OffNovelBook("巴黎圣母院",5600,"雨果"));

bookList.add(new OffNovelBook("悲惨世界",3500,"雨果"));

bookList.add(new OffNovelBook("金瓶梅",4300,"兰陵笑笑生"));

}

//模拟书店买书

public static void main(String[] args) {

NumberFormat formatter = NumberFormat.getCurrencyInstance();

formatter.setMaximumFractionDigits(2);

System.out.println("------------书店买出去的书籍记录如下：---------------------");

for(IBook book:bookList){

System.out.println("书籍名称：" + book.getName()+"\t书籍作者：" +

book.getAuthor()+ "\t书籍价格：" + formatter.format(book.getPrice()/100.0)+"元");

}

}

}

我们只修改了静态模块初始化部分，其他的部分没有任何改动，看运行结果：

------------书店买出去的书籍记录如下：---------------------

书籍名称：天龙八部 书籍作者：金庸 书籍价格：￥25.60元

书籍名称：巴黎圣母院 书籍作者：雨果 书籍价格：￥50.40元

书籍名称：悲惨世界 书籍作者：雨果 书籍价格：￥28.00元

书籍名称：金瓶梅 书籍作者：兰陵笑笑生 书籍价格：￥38.70元

OK，搞定，打折销售开发完成了。看到这里，各位可能有想法了：增加了一个 OffNoveBook 类后，你的业务逻辑还是修改了，你修改了 static 静态模块区域，这部分确实修改了，该部分属于高层次的模块，是由持久层产生的，在业务规则改变的情况下高层模块必须有部分改变以适应新业务，改变时尽量的少，压制变化风险的扩散。注意：开闭原则说是对扩展开放，对修改关闭，并不意味着不做任何的修改，**我们可以把变化归纳为以下几个类型**：

**逻辑变化**。只变化一个逻辑，而不涉及到其他模块，比如原有的一个算法是 a \* b + c，现在需要修改为a \* b \*c， 可以通过修改原有的类中的方法方式来完成， 前提条件是所有依赖或关联类都按照相同的逻辑处理。

**子模块变化**。一个模块变化，会对其他的模块产生影响，特别是一个低层次的模块变化必然引起高层模块的变化，因此在通过扩展完成变化时，高层次的模块修改是必然的，刚刚的书籍打折处理就是类似的处理模块，该部分的变化甚至会引起界面的变化。

**可见视图变化**。可见视图是提供给客户使用的界面，如 jsp 程序，swing 界面等，该部分的变化一般会引起连锁反应（特别是在国内做项目，做欧美的外包项目一般不会影响太大） ，如果仅仅是界面上按钮、文字的重新排布倒是简单，最司空见惯的是业务耦合变化，什么意思呢？一个展示数据的列表，按照原有的需求是六列，突然有一天要增加一列，而且这一列要跨度 N 张表，处理 M 个逻辑才能展现出来，这样的变化是比较恐怖的，但是我们还是可以通过扩展来完成变化，这就依赖我们原有的设计是否灵活。

我们再来回顾一下书店销售书籍的程序，首先是我们有一个还算灵活的设计（不灵活是什么样子？BookStore 中所有使用到 IBook 的地方全部修改为实现类,然后再扩展一个 ComputerBook书籍， 你就知道什么是不灵活了） ，然后有一个需求变化，然后我们通过扩展一个子类拥抱了变化，然后把子类投入运行环境中， 新逻辑正式投产。 通过分析， 我们发现我们并没有修改原有的模块代码， IBook 接口没有改变， NovelBook类没有改变，这属于已有的业务代码，我们保持了历史的纯洁性。放弃修改历史的想法吧，一个项目的基本路径应该是这样的项目开发、重构、测试、投产、运维，其中的重构可以对原有的设计和代码进行修改，运维尽量减少对原有代码的修改，保持历史代码的纯洁性，提高系统得到稳定性。

**为什么要使用开闭原则**

每个事物的诞生都有它存在的必要性，存在即合理，那我们的开闭原则的存在也是合理的，为什么这么说呢?

首先，开闭原则是那么的著名，只要是做面向对象编程的，甭管是什么语言，Java 也好C++也好或者是 Smalltalk，在做开发时都会提及开闭原则，如果你是个架构师，没有听说过开闭原则，那你绝对可以闭口气把自己憋死，如果你是一个 Java 程序员，没有听说过开闭原则，那你最好喝口水把自己呛死。

其次，开闭原则是最基础的一个原则，前边五个章节介绍的原则（单一职责原则、里氏替换原则、依赖倒置原则、接口隔离原则、迪米特法则）都是开闭原则的具体形态，也就是说前五个原则就是指导设计的工具和方法，而开闭原则才是其精神领袖，换一个角度来理解，依照 Java 语言的称谓，开闭原则是抽象类，其他五大原则则是具体的实现类，开闭原则在面向对象设计领域中的地位就类似于的牛顿第一定律在力学界、勾股定律在几何学、质能方程在狭义相对论中的地位，其地位无人能及。

最后，开闭原则是非常重要的，通过以下几个方面来理解其重要性：

1. **开闭原则对测试的影响**。每个已经投产了的代码都是有意义的，并且都受系统的规则约束着，这样的代码都是经过“千锤百炼”的测试过程，不仅保证逻辑是正确的，还要保证苛刻条件（高压力、异常、错误）下的不产生“有毒代码” （Poisonous Code） ，因此有变化提出时，我们就需要考虑一下，原有健壮的代码是否可以不修改，仅仅通过扩展开实现变化呢？否则，就需要把原有的测试过程回笼一遍，需要进行单元测试、功能测试、集成测试，甚至是验收测试，现在虽然在大力提倡自动化测试工具，但是仍然代替不了人工的测试工作。

以上面我们提到的书店售书为例，IBook 接口写完了，实现类 NovelBook 也写好，我们需要写一个测试类进行测试，测试类源代码如下：

public class NovelBookTest extends TestCase {

private String name = "平凡的世界";

private int price = 6000;

private String author = "路遥";

private IBook novelBook = new NovelBook(name,price,author);

//测试getPrice方法

public void testGetPrice() {

//原价销售，判断输入和输出的值是否相等进行断言

super.assertEquals(this.price, this.novelBook.getPrice());

}

}

单元测试通过，显示绿条，当然是绿条了，这么简单的逻辑再不是绿条，那就找块豆腐把自己撞死。在单元测试中，有一句非常有名的话，叫做“Keep the bar green to keep the code clean” ，保持绿条有利于代码整洁，这是什么意思呢？绿条就是 Junit 运行的两种结果中的一种：要么是红条，单元测试失败；要么是绿条，单元测试通过。一个方法的测试方法一般不少于 3个，为什么呢？首先是正常的业务逻辑要保证测试到，其次是边界条件要测试到，然后是异常要测试到，比较重要的方法的测试方法甚至有十多个，而且单元测试是对类的测试，类内的方法耦合是允许的，在这样的条件下，如果再想着通过修改一个方法或多个方法代码来完成变化，基本上就是痴人说梦，该类的所有测试方法都要重构，想象一下你在一堆你并不熟悉的代码中进行重构时的感觉吧！

在书店售书的例子中，增加了一个打折销售的需求，如果我们直接修改 getPrice方法，来实现业务需求的变化，那我们就要修改单元测试类，想想看我举这个例子是非常简单的，如果是一个复杂的逻辑，你的测试类就要修改的面目全非，还有，在实际的项目中一般一个类只有一个测试类，其中可以有很多的测试方法，在一堆本来就很复杂的断言中进行大量修改，难免就会出现测试遗漏情况，这是一个项目经理很难容忍的事情。

所以，我们需要通过扩展来实现业务逻辑的变化，而不是修改。上面的例子中通过增加一个子类OffNovelBook 来完成了业务需求的变化，对测试有什么好处呢？我们重新生成一个测试文件OffNovelBookTest，然后对 getPrice进行测试，单元测试是孤立测试，我只要保证我提供的方法正确就成了，其他的我不管，这不是单元测试的范畴。OK，源代码如下：

public class OffNovelBookTest extends TestCase {

private IBook below40NovelBook = new OffNovelBook("平凡的世界",3000,"路遥");

private IBook above40NovelBook = new OffNovelBook("平凡的世界",6000,"路遥");

//测试低于40元的数据是否是打8折

public void testGetPriceBelow40() {

super.assertEquals(2400, this.below40NovelBook.getPrice());

}

//测试大于40的书籍是否是打9折

public void testGetPriceAbove40(){

super.assertEquals(5400, this.above40NovelBook.getPrice());

}

}

新增加的类，新增加的测试方法，只要保证新增加类就是正确的就可以了。

2. **开闭原则可以提高复用性**。在面向对象的设计中，我们所有的逻辑都是从原子逻辑组合而来的，而不是在一个类中独立实现一个业务逻辑，只要这样代码才可以复用，粒度越小，被复用的可能性就越大。那为什么要复用呢？较少代码量，避免相同的逻辑分散在多个角落，避免日后的维护人员为了修改一个微小的缺陷或新增加新功能，而要在整个项目中到处找相关的代码，然后发出对开发人员“极度失望”的感慨。那怎么才能提高复用率呢？缩小逻辑粒度，直到一个逻辑不可再拆分为止。

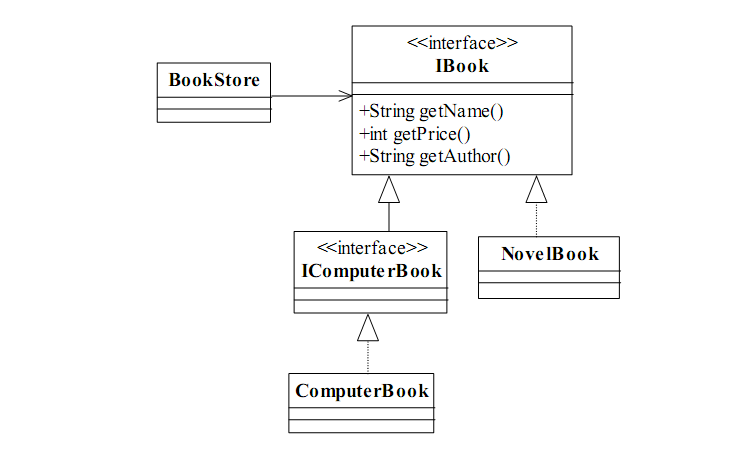
3. **开闭原则可以提高可维护性**。一个软件投产后，维护人员的工作不仅仅是对数据进行维护，还可能对程序进行扩展，那维护人员最乐意做的事情，就是扩展一个类，而不是修改一个类，甭管原有的代码写的多么优秀还是写的多么糟糕，让维护人员读懂原的代码，然后再修改是一件很痛苦的事情，不要让他在原有的代码海洋里徜徉完毕后再修改，这是对维护人员的一种折磨和摧残。

4. **面向对象开发的要求**。万物皆对象，我们需要把所有的事物都抽象成对象，然后针对对象进行操作，但是万物皆运动，有运动就有变化，有变化就要有策略去应对，怎么快速的应对？就需要在设计之初考虑到所有可能变化的因素，然后留下接口，等待可变因素转变真正的变化时轻松应对。

**怎么使用开闭原则**

开闭原则是一个非常虚的原则，前边五个原则是对开闭原则的具体解释，但是开闭原则并不局限于这么多，它“虚”的没有边界，就像“好好学习，天天向上”的口号一样，告诉我们要好好学习，但是学什么，怎么学并没有告诉我们，需要去体会去掌握，开闭原则也是一个口号，那我们怎么把这个口号应用到我们实际的工作中呢？

抽象约束。抽象是对一组事物的通用描述，没有具体的实现，也就表示它可以有非常多可能性，可以跟随需求的变化而变化，因此通过接口或抽象类可以约束一组行为，并且能够实现扩展开放，其包含三层含义：一是通过接口或抽象类约束扩展，对扩展进行边界限定，不允许出现在接口或抽象类中不存在的public 方法；二是在参数类型定义、输入输出参数尽量使用接口或者抽象类，而不是实现类，三是抽象层尽量保持稳定，一旦确定即不允许修改。还是以书店为例，目前只是销售小说书籍，单一经营毕竟是有风险的，于是书店新增加了一种书籍：计算机书籍，它不仅仅包含书籍名称、作者、价格等信息，还有一个独特的属性：面向的是什么领域，也就是它的范围(scope)，比如是讲语言的，数据库，还是硬件等等，我们先来看类图的修改：



增加了一个接口 IComputerBook 和实现类 ComputerBook，而 BookStore 不用做任何修改就可以完成书店销售计算机书籍的业务，我们来看源代码：

public interface IComputerBook extends IBook{

//计算机书籍是有一个范围

public String getScope();

}

很简单，计算机数据增加了一个方法，就是获得该书籍的范围，同时继承 IBook接口，毕竟计算机书籍也是书籍。其实现类如下：

public class ComputerBook implements IComputerBook {

private String name;

private String scope;

private String author;

private int price;

public ComputerBook(String \_name,int \_price,String \_author,String \_scope){

this.name=\_name;

this.price = \_price;

this.author = \_author;

this.scope = \_scope;

}

public String getScope() {

return this.scope;

}

public String getAuthor() {

return this.author;

}

public String getName() {

return this.name;

}

public int getPrice() {

return this.price;

}

}

public class BookStore {

private final static ArrayList<IBook> bookList = new ArrayList<IBook>();

//静态模块初始化，项目中一般是从持久层初始化产生

static{

bookList.add(new OffNovelBook("天龙八部",3200,"金庸"));

bookList.add(new OffNovelBook("巴黎圣母院",5600,"雨果"));

bookList.add(new OffNovelBook("悲惨世界",3500,"雨果"));

bookList.add(new OffNovelBook("金瓶梅",4300,"兰陵笑笑生"));

//增加计算机书籍

bookList.add(new ComputerBook("Think in Java",4300,"Bruce Eckel","编程语言"));

}

//模拟书店买书

public static void main(String[] args) {

NumberFormat formatter = NumberFormat.getCurrencyInstance();

formatter.setMaximumFractionDigits(2);

System.out.println("------------书店买出去的书籍记录如下：---------------------");

for(IBook book:bookList){

System.out.println("书籍名称：" + book.getName()+"\t书籍作者：" +

book.getAuthor()+ "\t书籍价格：" + formatter.format(book.getPrice()/100.0)+"元");

}

}

}

运行结果如下：

------------书店买出去的书籍记录如下：---------------------

书籍名称：天龙八部 书籍作者：金庸 书籍价格：￥32.00元

书籍名称：巴黎圣母院 书籍作者：雨果 书籍价格：￥56.00元

书籍名称：悲惨世界 书籍作者：雨果 书籍价格：￥35.00元

书籍名称：金瓶梅 书籍作者：兰陵笑笑生 书籍价格：￥43.00元

书籍名称：Think in Java 书籍作者：Bruce Eckel 书籍价格：￥43.00元

如果我是做维护的，我就非常乐意做这样的事情，简单而且不需要与其他的业务做耦合，我只需要做的事情就是在原有的代码上进行添砖加瓦，就可以实现业务的变化。我们来看看这段代码给我们带来了几层意思。

首先， ComputerBook 类必须实现 IBook 的三个方法，是通过 IComputerBook 接口传递进来的约束，也就是我们制定的 IBook接口对扩展类 ComputerBook 产生了约束力，这个约束力同时对我们扩展提供非常好的帮助，否则 BookStore 类就需要进行大量的修改了。

其次，如果我们原有的程序设计时采用的不是接口，而是实现类，那会出现什么问题呢？我们把 BookStore 类中的私有变量 bookList修改为：

private final static ArrayList<NovelBook> bookList = new ArrayList<NovelBook>();

把原有IBook的依赖修改为对NovelBook实现类的依赖， 想想看我们这次的扩展是否还能继续下去呢？一旦这样设计，我们就根本没有办法扩展，需要修改原有的业务逻辑（也就是 main 方法） ，这样的扩展基本上就是形同虚设。

最后，如果我们在 IBook 上增加一个方法 getScope 是否可以呢？答案是不可以，因为原有的实现类NovelBook 已经在投产运行中，它不需要改方法，而且接口是与其他模块交流的契约，修改契约就等于让其他模块修改。因此，我们的接口或抽象类一旦定义，就应该立即执行，不能有修改接口的思想，除非是彻底的大返工。

所以，要实现对扩展开放，首要的前提条件就是：抽象约束。

**参数控制模块行为**。程序员是一个很苦很累的活，那怎么才能减轻我们的压力呢？答案是尽量是参数来控制我们的程序的行为，减少重复开发。参数可以从文件中获得，也可以从数据库中获得，举个非常简单的例子，login 方法中提供了这样的逻辑：先检查 IP 地址是否在允许访问的列表中，然后再决定是否需要到数据库中验证密码（如果采用 SSH 架构，则可以通过 Struts 的拦截器来实现） ，该行为就是一个典型的参数控制模块行为的例子，其中达到极致的就是控制翻转（Inversion of Control） ，使用最多的就是Spring 容器，在 SpringContext 配置文件中，有这样一段配置：

<bean id="father" class="xxx.xxx.xxx.Father" />

<bean id="xx" class="xxx.xxx.xxx.xxx">

<property name="biz" ref="father"></property>

</bean>

然后，通过建立一个 Father 类的子类 Son，完成一个新的业务，修改一下配置文件：

<bean id="son" class="xxx.xxx.xxx.Son" />

<bean id="xx" class="xxx.xxx.xxx.xxx">

<property name="biz" ref="son"></property>

</bean>

**制定项目章程**。在一个团队中，项目章程的建立是非常重要的，因为在章程中指定了所有人员都必须遵守的约束，而对项目来说约定是优于配置。相信大家都做过项目，会发现一个项目会产生非常多的配置文件，举个简单的例子，以 SSH 项目开发为例，一个项目中的 Bean 配置文件是非常多的，管理非常麻烦，如果需要扩展就需要增加子类，并修改 SpringContext 文件，而如果你在项目中指定这样一个章程：所有的 Bean 都自动注入，使用 Annotation 进行装配，进行扩展时，甚至只用写一个子类，然后由持久层生成对象，其他的都不需要修改，这就需要项目内约束，每个项目成员都必须遵守，该方法需要一个团队有较高的自觉性，需要一个较长时间的磨合，一旦项目成员都熟悉这样的规则，比通过接口或抽象类进行约束效率更高，而且扩展性一点也没有减少。

**封装变化**。对变化的封装包含两层含义：一是对相同的变化封装到一个接口或抽象类中，二是对不同的变化封装到不同的接口或抽象类中，不应该出现两个不同的变化出现同一个接口或抽象类中。封装变化，准确的讲就是封装可能发生的变化，一旦预测到或“第六感”发觉有变化，就可以进行封装，23 个设计模式都是从各个不同的角度对变化进行封装，我们会在各个模式中逐步讲解。