《设计模式学习笔记》

朱春旭

2021.04.07

# 一．面向对象7大设计原则

## 1.开闭原则

开闭原则全称开放-封闭原则（Open Closed Principle， OCP）由勃兰特·梅耶（Bertrand Meyer）提出，他在 1988 年的著作《面向对象软件构造》（Object Oriented Software Construction）中提出：软件实体应当对扩展开放，对修改关闭。

开闭原则的含义是：当应用的需求改变时，在不修改软件实体的源代码或者二进制代码的前提下，可以扩展模块的功能，使其满足新的需求。

生活实例理解

在我们日常编写代码时，有没有出现过这种情况？当一个已经做完的功能被产品经理要求追加一个新的需求时，我们往往要修改掉一部分之前写好的代码，但是这一小小的改动往往会造成“牵一发而动全身”的后果，比如与被修改的那部分代码有关的功能全部都要跟着修改，否则就报错；亦或是被修改的那段代码是一个已离职员工编写的，你对这段代码并不了解，改动后你甚至不知道会在哪个地方报错。因此，为了尽量避免进入上述的困境，我们在设计一段代码时应该尽量遵循开闭原则，即在编写出一段代码后，如果后面需要追加新的功能，可以直接扩展新的代码，不用去更改以前的代码。

说白了就是要“解耦”，如果类之间耦合度过高，就会出现一处修改，处处报错的情况。因此，如何设计能降低类之间的耦合度是重中之重，这也是很多设计原则和设计模式最终的目的。

想做到这点是十分困难的，这要求编写代码的人具备扎实的功底和丰富的工作经验，所以这是一条漫长的路。不过相信你在学习完所有面向对象设计原则和设计模式以后，你会对开闭原则有一些自己的领悟！

## 里氏替换原则

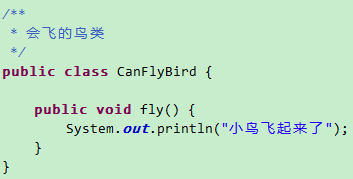
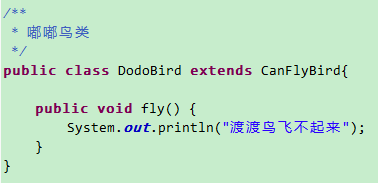
里氏替换原则（Liskov Substitution Principle，LSP）由麻省理工学院计算机科学实验室的里斯科夫（Liskov）女士在 1987 年的“面向对象技术的高峰会议”（OOPSLA）上发表的一篇文章《数据抽象和层次》（Data Abstraction and Hierarchy）里提出来的，她提出：继承必须确保超类所拥有的性质在子类中仍然成立。

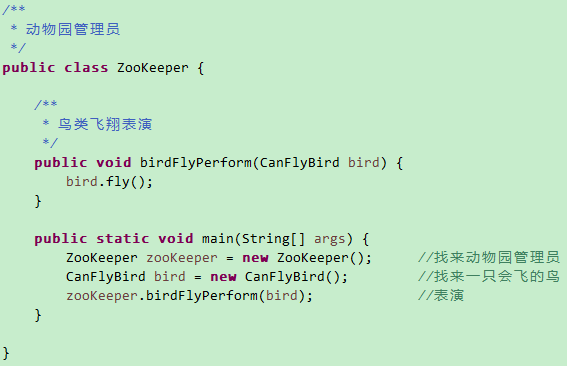
里氏替换原则阐述了有关继承的一些原则，它规定良好的继承关系应该具备这样的性质：

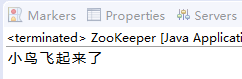
如果两个类是继承关系，那么在任意使用父类的地方，将父类替换为子类都不应该改变程序的结果。

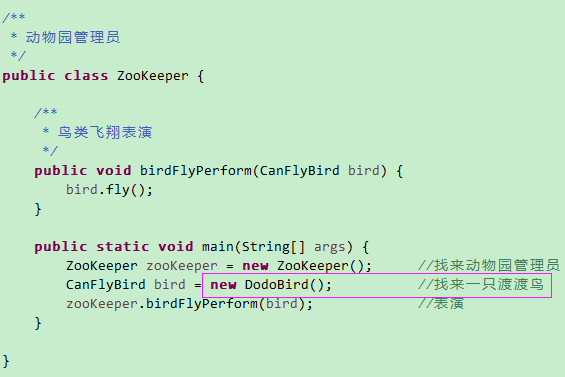
满足上述条件的继承关系可以称为遵循了里氏替换原则，目的是提高代码复用性，程序健壮性，系统可维护性；不遵循里氏替换原则的继承关系在运用**多态**时出错的可能性很大。

上面的说明可能不太好理解，举个例子：不是所有鸟类都会飞，例如渡渡鸟，它虽然是鸟类，但是它不会飞。那么可以说渡渡鸟是鸟类，但不能说渡渡鸟是“飞鸟类”。









看上述代码案例，将飞鸟类**替换**成嘟嘟鸟时，这个表演可以说是凉了~。观众本来应该是拍手叫好的，但是换成嘟嘟鸟上场表演，都是目瞪口呆、一片唏嘘，即这样替换后改变了程序原本的结果。

从程序的角度出发，将上面的例子反过来看，它反映了如果你要为两个类设计为继承关系，那么子类就不能重写父类的普通方法，否则会在运用**多态**时有大概率出错（这里的出错可能是报错，也可能是运行结果与预期不符）。

什么是多态？

上例中，birdFlyPerform方法中的参数要求传入的是CanFlyBird类型的对象，实际上呢，真正传过来的对象是new DodoBird()得到的实例，只不过它使用“向上转型”由父类引用；从代码上看，bird这个对象好像是CanFlyBird类型的，但是在程序运行时，bird这个对象可以是任意子类的实例；当然，执行的也是子类继承或重写的方法。这种现象就是多态。Java中允许我们在运行时动态识别对象和类的类型信息的方式一共用两种：一种是多态，另一种是反射。

总的来说，要想设计出来的继承关系遵循里氏替换原则应该参照下面的规范：

1.子类可以在父类的基础上扩展一些属于自己的功能

2.子类应该实现父类的抽象方法，但尽量不要覆盖父类的非抽象方法

## 依赖倒置原则

依赖倒置原则（Dependence Inversion Principle，DIP）是 Object Mentor 公司总裁罗伯特·马丁（Robert C.Martin）于 1996 年在 C++ Report 上发表的文章。依赖倒置原则的原始定义为：高层模块不应该依赖低层模块，两者都应该依赖其抽象；抽象不应该依赖细节，细节应该依赖抽象。其核心思想是：要面向接口编程，不要面向实现编程。

首先，这里讲的依赖倒置和Spring的IOC（控制反转）不是一回事，虽然它们的目标都是降低类之间的耦合性，但实现的方式不同（需要自行去了解IOC的工作原理，这里不介绍）。

回顾“里氏替换原则”里举的例子，在“鸟类飞翔表演”方法的参数中，我们设置的是CanFlyBird这个父类，并没有设置为具体的子类，这其实就是遵循了依赖倒置原则；试想，如果这里我写的是具体的子类，那么有多个鸟类要进行表演时怎么传参？按一般的想法是不是要复制好几个方法，改一改参数（即重载），最后再把这几个方法里公共的部分抽出来。也就是说如果我们在写一个方法时总是传入一个子类，只注重去实现一个功能，那么往往在有新需求到来时，会让你不得不得去修改这个方法，或是扩展类似的方法，最终使代码的结构越来越乱，越来越难维护。如果你还是难以理解这段话，不妨现在先直接跳到“简单工厂模式”和“工厂方法模式”去学习一下，深刻理解一下工厂方法模式为什么比简单工厂模式的扩展性更强，再回来理解这段话。

依赖倒置原则是实现开闭原则的重要手段，为什么？

其实你在理解了上面一大段话后应该知道，真正实现了开闭原则的是什么？是**多态**。因为多态的存在，使得传递参数时可以将参数抽象化，只在运行时赋予真正的意义，正是这种一对多的形式允许子类可以无限扩展，但不管怎么扩展，方法中的参数始终保持不变。引入一个抽象类替代引入一堆实现类，不就实现解耦了吗。而实现多态的前提就是，你必须定义出基于继承关系的结构（继承和实现接口都算），这正是依赖倒置原则要求我们做的事。

依赖倒置原则与里氏替换原则的关系是：

依赖倒置原则强调我们应该尽量去设计基于继承关系的结构，而里氏替换原则强调我们在设计这种继承关系的结构时应该遵循什么规则。

## 单一职责原则

单一职责原则（Single Responsibility Principle，SRP）又称单一功能原则，由罗伯特·C.马丁（Robert C. Martin）于《敏捷软件开发：原则、模式和实践》一书中提出的。这里的职责是指类变化的原因，单一职责原则规定一个类应该有且仅有一个引起它变化的原因，否则类应该被拆分。

这个原则应该很好理解了，我们从最开始学习编程时就应当被告知一句话：不要把所有代码都写在main函数里，要把功能模块化，封装在不同的函数里。

这句话强调了面向对象三大特性之一的“封装”。我的理解是封装应当包含对“类”的封装和对“方法”的封装这两个方面，而单一职责原则强调的是在类的层面做的封装，另一个是对方法层面的封装，就是后面要介绍的接口隔离原则。（方法是我通俗的叫法，准确的应该叫接口）

单一职责原则强调一个类应该只具备一个职责，听起来简单，但要真正做到是很困难的。划分的依据是什么？粒度多大比较合适？这需要开发人员具备一定的分析能力和经验。

我们能做的就是尽量按职能去划分不同的类，让不相关的两个类不要产生过多的交集。多积攒实际开发经验，在遇到问题时积累设计上的不足，体会其中的原理。

## 接口隔离原则

接口隔离原则（Interface Segregation Principle，ISP）要求程序员尽量将臃肿庞大的接口拆分成更小的和更具体的接口。2002 年罗伯特·C.马丁给“接口隔离原则”的定义是：客户端不应该被迫依赖于它不使用的方法；该原则还有另外一个定义：一个类对另一个类的依赖应该建立在最小的接口上。

事实上，我们在写代码时已经在遵循接口隔离原则了。例如一个需求是用户A在商城B中购买了商品C；虽然只有这一句话，但是我们要做的事情有很多，首先要取得用户A的账户信息，判断余额是否充足，再取得商品C的库存量是否充足；然后生成一笔交易订单，扣除用户A的账户余额和商品C的库存量，增加商城B的金额，写日志明细等等。这里面描述的各功能项都应该设计为独立的接口，在这一个业务功能上将这些接口组合在一起，而不是一股脑得将所有代码全写在一块。因为这些独立的接口是能够被其他业务功能复用的。

当然，如何更优地封装这些接口的职能需要我们积累更多的开发经验。

接口隔离原则和单一职责原则的关系是：

它们都强调了封装思想，目的是实现“高内聚，低耦合”。不同的是，单一职责原则做在类层面，接口隔离原则做在接口层面。

## 6.迪米特法则

迪米特法则（Law of Demeter，LoD）又叫作最少知识原则（Least Knowledge Principle，LKP)，产生于 1987 年美国东北大学（Northeastern University）的一个名为迪米特（Demeter）的研究项目，由伊恩·荷兰（Ian Holland）提出，被 UML 创始者之一的布奇（Booch）普及，后来又因为在经典著作《程序员修炼之道》（The Pragmatic Programmer）提及而广为人知。

迪米特法则的定义是：只与你的直接朋友交谈，不跟“陌生人”说话。其含义是：如果两个软件实体无须直接通信，那么就不应当发生直接的相互调用，可以通过第三方转发该调用。其目的是降低类之间的耦合度，提高模块的相对独立性。

举个例子，你正在使用的软件出了bug，需要将这个bug提交给软件公司来修复它。此时你应该留存售后服务中心的联系方式，而不是某个技术员的联系方式。用代码体现的话就是：

new [AfterService](http://www.baidu.com/link?url=kwzAb58Xz9ZKlT2yYoCwWaTeLJao36cfIvuNTpf-HH1JCO__9dpPtm0Wooa9rQRhUYwy2b-UMnQmtQoMVf9lzRnv2O0kbt6f-jaXT09HoEYPqR7BZP9wct5khe94TkGo" \t "https://www.baidu.com/_blank)().fix(); //联系售后服务中心执行修复（由内部选举技术员，调用者无需关心）

new T[echnician](http://www.baidu.com/link?url=gDCoMlMZ-J9dgBPtUpA_ece5gHn5ZD7pzNAA6q08CqtZLBWFwSSAPBfb6VKVrcbeL1YRHpW_Y9l4V1smUoTO9wjoIUzwKidiGZJy2SDApju" \t "https://www.baidu.com/_blank)A().fix(); //打电话给技术员A，修复

看似都能为你解决问题，但是如果技术员A一旦离职了，换成了技术员B，那么你再打电话给技术员A可就不管用了，此时你必须修改你和技术员A之间的依赖关系；但是如果你之前存的是售后服务电话，那么还会由售后服务中心还会为你指派技术员B，此时不需要修改依赖关系；这就是强耦合和低耦合的一个体现。理解了上述的关系，你就不难理解网关的重要性了。

技术体现

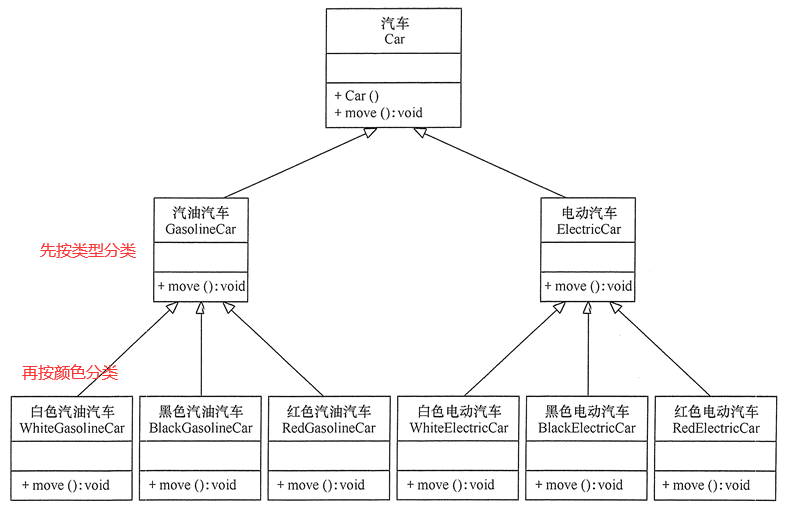
网关、负载均衡、各种任务调度中心

在服务器集群下，服务的消费者只需要向网关或者负载均衡的扮演者（以下统一称作网关）发起请求，由网关来调配用于提供服务的服务器，而无需直接对他们身后的服务器集群中的各服务器发起请求。这样做的好处是：如果集群中的某台服务器宕机或者又新增了几台能提供服务的服务器时，只需要通知到网关，无需通知到所有可能调用它们的服务消费者。此外，还可以在网关处统一设置路由拦截、记录日志等行为。

## 7.合成复用原则

合成复用原则（Composite Reuse Principle，CRP）又叫组合/聚合复用原则（Composition/Aggregate Reuse Principle，CARP）。它要求在软件复用时，要尽量先使用组合或者聚合等关联关系来实现，其次才考虑使用继承关系来实现。

合成复用原则是对依赖倒置原则与里氏替换原则的一个补充，它解决了直接继承的一些弊端。我们看下面的一张图，图中所示的继承关系我们通常叫做“继承复用”：

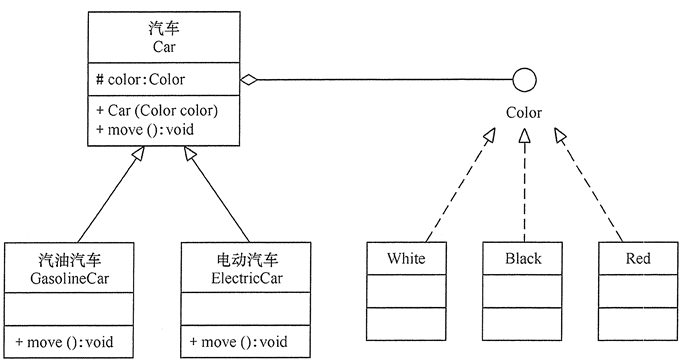


上图中的大概意思就是：我们对汽车这个类先按类型划分，可以划分为汽油汽车和电动汽车两个子类，再按照颜色分类就变成了2\*3=6个子类。

弊端一，这才2种类型和3种颜色就已经有6个子类了，假设现有m种类型和n种颜色，那么再增加一种类型就得增加n个子类，增加一种颜色就得增加m个子类，照这种方式设计，那类型和颜色一旦多起来还得了？

弊端二，如果上述的汽油汽车一旦需要发生修改，那么下面一连串带颜色的汽油汽车都得跟着修改，这就违背了开闭原则。

那怎么办呢？细想一下，每次增多的这些子类好像都有共同的特点；就像你在写代码时发现两段代码中有相同的部分，说明这段代码可以进行封装（抽象）啊。合成复用原则告诉了我们应该怎样做，看下面一张图：



这里最重要的改变就是将颜色抽象出来，并且作为汽车类的一个属性，这种方式通常叫做“聚合复用”，其次才使用了对类型的继承复用。

此后，我们再需要添加一种类型或者颜色就不需要再增加很多子类了，对汽油汽车进行修改时，也不用对一大堆带颜色的汽油汽车进行修改了，实现了开闭原则。

## 总结

开闭原则是我们设计一个具备可复用性、可扩展性和可维护性系统的核心原则。它主要强调了我们设计的系统以及下属的子功能模块应该是“高内聚、低耦合”的。其余的六大设计原则是实现开闭原则的手段。

里氏替换原则和依赖倒置原则要求我们尽量设计基于“继承”关系的程序结构，利用“多态”的性质增加可扩展性，体现了“低耦合”，增强可扩展性。

单一职责原则和接口隔离原则要求我们尽量去细化职能，“封装”成接口，多复用，体现了“高内聚、低耦合”，增强可复用性。

迪米特法则和合成复用原则要求我们降低不相关类之间的依赖关系，体现了“低耦合”，增强可维护性。

# 二．GOF的23种设计模式

艾瑞克·伽马（ErichGamma）、理査德·海尔姆（Richard Helm）、拉尔夫·约翰森（Ralph Johnson）、约翰·威利斯迪斯（John Vlissides）等 4 位作者合作出版了《设计模式：可复用面向对象软件的基础》（Design Patterns: Elements of Reusable Object-Oriented Software）一书，在本教程中收录了 23 个设计模式，这是设计模式领域里程碑的事件，导致了软件设计模式的突破。这 4 位作者在软件开发领域里也以他们的“四人组”（Gang of Four，**GoF**）匿名著称。

## 创建型模式

### 单例模式 Singleton

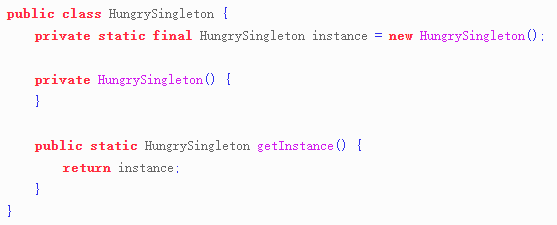
保证一个类仅有一个实例，且该实例必须自行创建，并对外提供一个全局访问点。如果一个类具备“一旦创建出来实例是可以被多个地方复用的”的特性，就可以用单例模式去设计它。目的是通过资源共享的方式节省内存开销。

单例模式分为懒汉式和饿汉式

懒汉式是指初始状态下只声明一个静态对象，但不立即创建它，只有当它被使用时才去创建它。



饿汉式是指初始状态下声明一个静态对象，且立即创建它。



懒汉式和饿汉式最重要的区别有两个：

1. 懒汉式的好处是如果这个类并不被常用，那么在启动项目时就不用为它立即创建实例，可以节省内存资源。饿汉式的好处是在启动项目时就为它立即创建了实例，当一个业务需要使用该类时有现成的可以用，省去了实时创建的过程，加快响应速度。两种方式各有优点，应该结合实际业务需要来决定设计为哪一种。
2. 在多线程的环境下，饿汉式是线程安全的，而常规的懒汉式是线程不安全的（理解这句话需要先了解Java内存模型）。因此如果要在多线程环境下使用懒汉式，对象前的volatile关键字和get方法前的synchronized关键字是必不可少的。

### 原型模式 Prototype

当需要另一个一模一样的对象时，不使用常规的new来创建对象，而是通过拷贝原型创建新的对象；如果是需要一个相似的对象，也可以通过拷贝一个过去做简单修改。这样做的好处是省去了new的繁琐过程，并且基于二进制流的复制通常比new一个对象来的更快。

生活案例理解

制作泥土人偶时，需要先做一个身体，然后再为它们加上不同的表情，颜色等细节。那么它们的身体都是一样的，却每次都要重新捏一个身体出来，如果有复制的能力就好了，直接复制一个过来改造细节能省很多事，可惜没有如果。现实中通常实现不了，可是程序中能实现！

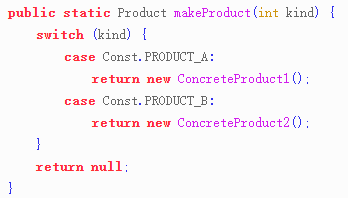
技术体现

当一个页面中多处需要加载同一个对象中的数据，但是它们各自修改时数据又需要保持独立（即不能简单的使用同一个对象，因为会指向同一个地址，当其中一个修改了值，其他都会跟着变），因此需要通过这个原型，拷贝几个相同的对象出来。常见的做法就是通过某些语言会自带的clone方法或自己重写的clone方法或者通过序列化和反序列化的手段来实现。

深拷贝和潜拷贝都可以被用在原型模式中，具体使用哪一种需要根据实际情况决定。

### 简单工厂模式 Simple Factory

简单工厂模式也叫静态工厂方法模式，目的是在不同条件下创建不同实例。对于使用者（调用者）来说，只需要知道一个参数（类似商品的名称），将这个参数传入工厂的制造方法中（将商品名称告知工厂），即可获得一个对应的对象实例，不需要知道具体制造的细节，职责分工明确。

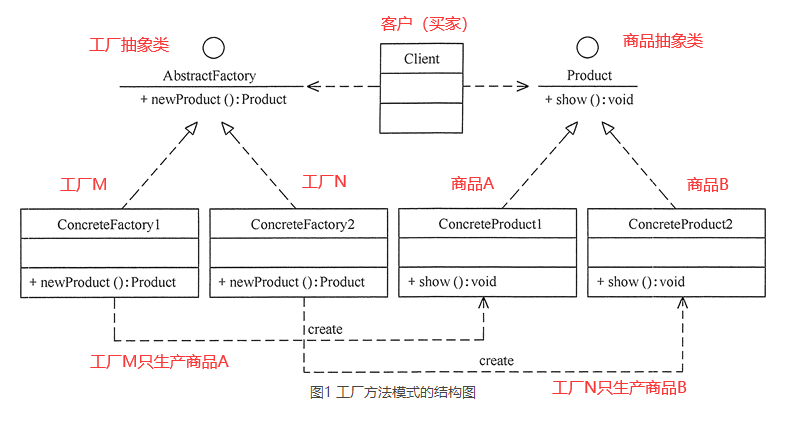


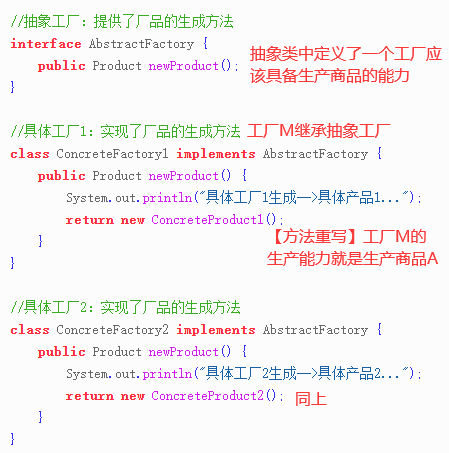
简单工厂的缺点是

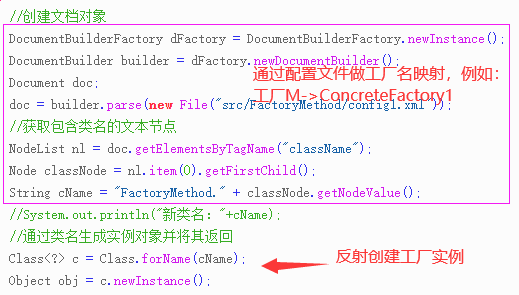
1. 职责过重，所有商品的创建都需要在工厂的制造方法中完成，即商品过多时将造成制造方法中代码臃肿。因此简单工厂模式只适用于商品较少的场景。
2. 每增加或减少一个商品都需要对制造方法中的代码做修改，违背了开闭原则。
3. 简单工厂使用了静态工厂方法，使得子类无法继承，不能形成基于继承的等级结构。

### 工厂方法模式 Factory Method

工厂方法模式在简单工厂模式的基础上做出改进，主要解决了简单工厂违背开闭原则的问题。它的核心思想是由原来的一个工厂生产多个商品，改为开放多个工厂，每个工厂只生产一种商品。例如：原来的工厂M可以生产A，B两种商品，现在扩建一个新的工厂N，让工厂M只生成A，让工厂N只生成B，有几个商品就需要扩建出几个工厂。





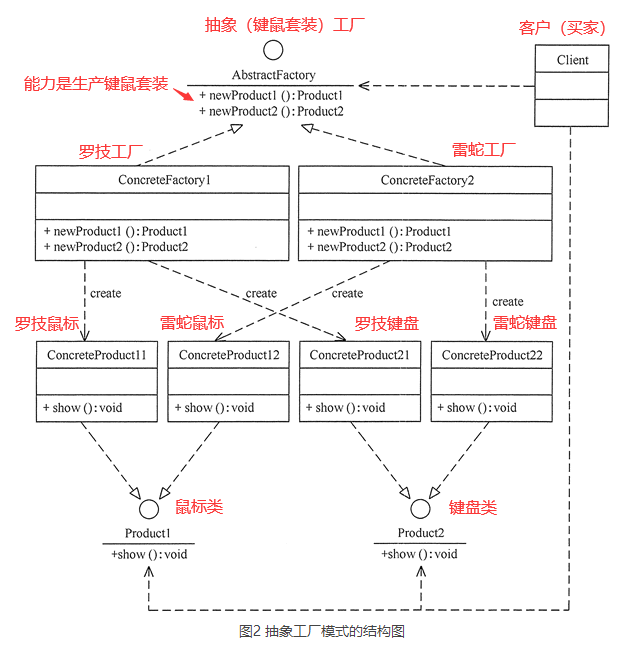


按照工厂方法模式设计后，如果增加新的商品，那么只需要新增一个生产这种商品的工厂类并继承自抽象工厂，重写父类的生产商品方法即可。再结合“反射”来创建类的实例，这样可以做到新增和减少商品完全不需要修改之前的代码，遵循了开闭原则。

工厂方法模式虽然解决了简单工厂模式违背开闭原则的问题，但也带来另一个缺点，很显然，每个工厂只生成一个商品的话，如果商品越来越多，那么工厂类就会越来越多。

### 抽象工厂模式 Abstract Factory

抽象工厂模式是对工厂方法模式在特定情况下的优化。例如在工厂方法模式下，要生产“罗技鼠标”和“雷蛇鼠标”这两种商品，那么就需要建造“罗技鼠标工厂”和“雷蛇鼠标工厂”，它们共同继承自“鼠标工厂”，能力是生产鼠标；但是在现实中，当一个品牌做大时，往往不只生产一种商品，还会生产一些配套的商品来一起使用，像鼠标搭配键盘，枪械搭配子弹。那么如果罗技和雷蛇又开始生产键盘时，按照之前的工厂方法模式就需要再扩建两个工厂“罗技键盘工厂”和“雷蛇键盘工厂”，此时已经有4个工厂类和2个抽象工厂类了。但是我们发现，它们之中有一些共性是可以**抽象**出来的，比如罗技和雷蛇这两个品牌都是既生产鼠标又生产键盘，那么抽象后的工厂应该是“罗技工厂”和“雷蛇工厂”，能力是生产鼠标和键盘，此时只需要2个工厂类和1个抽象工厂类。



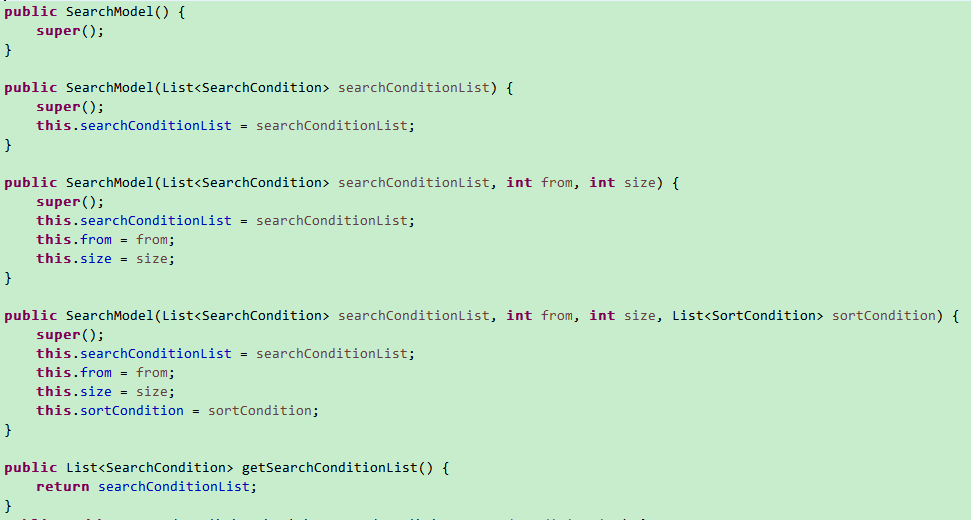
抽象工厂模式并不能完全解决工厂方法模式的弊端，但可以起到一定缓解作用，前提是符合上述的特定情况（即方法工厂中是能找出可抽象部分的）

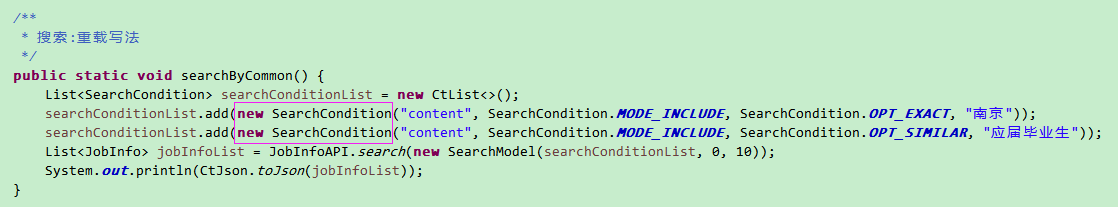
### 建造者模式 Builder

建造者模式一般用来创建复杂的复合对象，开放个性配置步骤是它的特点。

举个例子，我们在创建一个类的实例时可以使用在其内部定义的构造器，而我们常常会因为在不同的情况下需要实例中附带的参数也不同，这种情况下通常可以使用“重载”来解决。

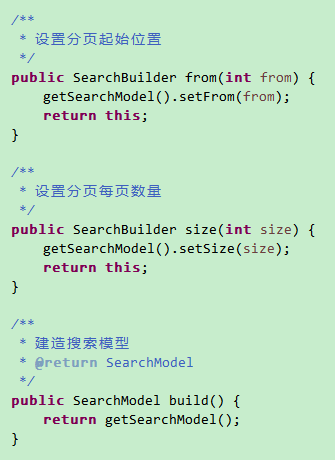
使用重载

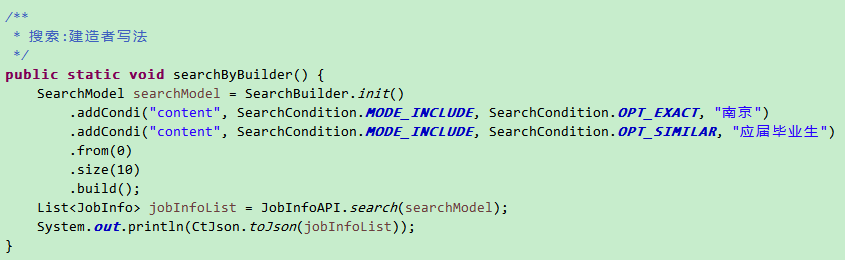




在上图中展示了对构造器进行重载解决上述问题，但是使用重载的方式设计会有弊端，即当类中属性过多，需要传入参数的“组合”情况过多时，编写和辨别重载方法是件很麻烦的事情。

使用建造者模式





建造者模式的设计规则（写法）

在“建造”实例的过程中，始终返回对象本身，目的是让其能够一直调用自身中的其他建造方法，即可以像上图中所示那样一直使用“.xxx()”的方式进行建造；在建造的最终阶段，应该返回本次建造的最终结果，即上图中的build()方法。

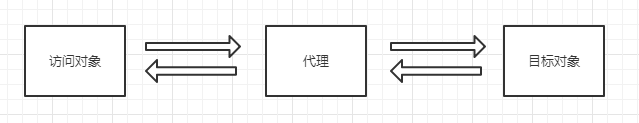
对比重载的方式可以看出建造者模式具备以下两个优点

1. 要建造一个对象，不必定义复杂的重载方法，可能用到的属性只需要定义一遍，在使用时就可以对类中属性进行灵活“组装”，需要什么就装什么，不需要也可以通过在定义时赋上一个默认值，不用担心不组装它，就没有值的情况。
2. 回上面去看重载写法的图，会发现有一句代码被粉色线条框起来了，SearchCondition这个类是我们所需要的建造的SearchModel中的一个属性，它在业务上是被需要使用的，但是再看建造者写法中，除了在引用常量时使用到了该类，并没有创建该类的实例（即在建造者模式中这个类的出现是非必须的），没有出现这个类的原因是它被封装在建造者类中了。所以建造者模式的第2个优点是对调用者来说，可以屏蔽掉一些非必要出现的类，尽量地简化代码，提高“可读性”。

## 结构型模式

### 代理模式 Proxy

代理模式的定义：由于某些原因需要给某对象提供一个代理以控制对该对象的访问。这时，访问对象不适合或者不能直接引用目标对象，代理对象作为访问对象和目标对象之间的中介。其形式如下：

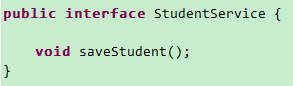


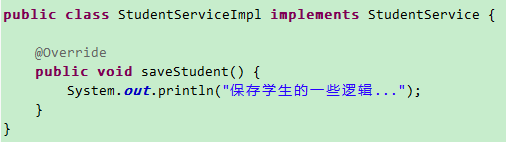
代理模式的目标主要有三个：

1. 对目标对象隐藏了访问对象的存在，也可以对访问对象隐藏目标的存在（目标对象会以为是代理直接发起的请求，并不知道访问对象的存在；同理，代理也可以对访问对象隐藏目标的存在）；
2. 实现解耦（访问对象和目标对象都跟代理有关系，但访问对象与目标对象之间没有依赖关系）；
3. 代理可以在请求或响应的数据上做修改或扩展。（例如：小明想向小红表白，但他很害羞，不擅长言语表达，于是他找来“情圣”小刚帮帮他。他对小刚说，麻烦你帮我转告小红就说“我很喜欢你”，但是小刚在向小红转告时可以说“自从看见你的第一天起就为你心动，我很喜欢你，你愿意陪我一起浪迹天涯，共度余生吗？”。这就是代理在原请求上的扩展，（专业术语叫“代码增强”）

下面用代码模拟我们日常开发中使用的Service层事务管理从静态代理到动态代理的演变。

静态代理





上面两张图是我们在日常开发中最常见的写法，如果没有做Spring事务相关配置的话，saveStudent方法是不受事务管理的。要想让它接受事务管理，我们需要在执行它之前，先开启事务（即调用ORM框架提供的类似begintransaction的方法），在任务完成后提交事务并关闭事务（即commit方法和close方法），如果在任务执行过程中出现异常，可以通过捕获异常并做事务回滚（即rollback）。同时我们还可以在保存的前后写一些日志信息。

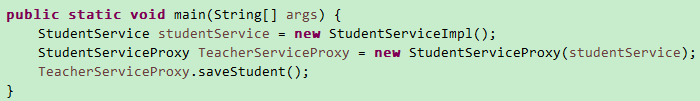
上述的方案似乎可以通过我们最开始提到的代理模式来解决（找一个代理来替我完成这个任务并且在这基础上扩展一些功能）。

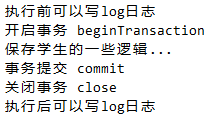
到这里你可能会有一些疑问，为什么一定要找一个代理替我完成，我自己也可以扩展自己的逻辑代码实现事务管理。这个的确是可以的，从代码的角度看是没问题的，但是从面向对象的角度看，是不恰当的，因为这不应该是saveStudent本身的职能。就像前面说的例子，小明因为不擅长语言表达所以找了小刚帮他向小红表白，那如果小明让自己变得擅长语言表达不就不需要小刚帮忙了吗，但这在现实中不大符合常理。因此，根据上述方案，我们定义一个代理类，如下图。



代理类因为需要代替访问对象执行相同的事情（即必须具备相同的职能），因此需要实现和访问对象相同的接口。

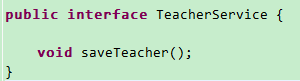
编写测试类和输出结果如下：

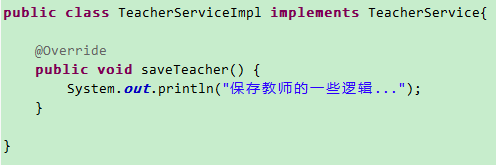




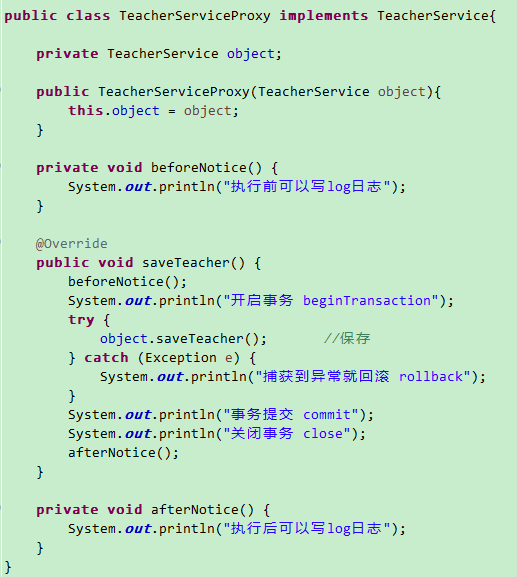
上面是静态代理的最简形式，如果我们现在不止有一个StudentService了，要再加一个TeacherService且也需要让它受到事务管理呢？

首先，我们需要扩展出TeacherService和TeacherServiceImpl的逻辑代码，这是必不可少的。

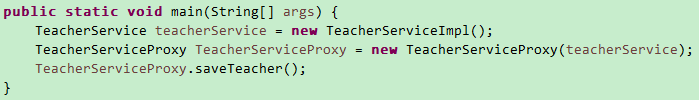


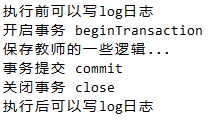


其次，要扩展出TeacherService对应的代理类



编写测试类和输出结果如下：





以此类推，如果有n个service，我们需要写对应的n个代理类，这代码量就相当恐怖了，这是静态代理的主要弊端。现在，我们再去对比一下两个代理类之间的区别，发现只有对访问对象的依赖关系发生了修改，其他地方都是一样的，看到这种现象我们应该意识到这段代码其实是可以抽象（封装）的。

既然只有访问对象不同，我们能否定义出一个通用的代理类，允许其代理Object类型对象（即任意对象）呢。但是如果一个对象是Object类型的，我们该如何调用里面的方法呢，因为我们不能再使用obj.xxx()的方式调用其中的方法了。确实有一种方式可以支持我们这样做，并且我们在前面讲到工厂方法模式时就提到了，那就是反射。

动态代理

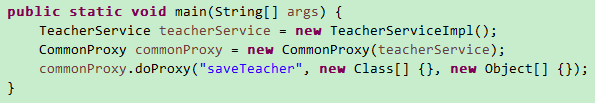
动态代理的出现解决了静态代理的弊端，动态代理在静态代理基础上封装了代理类，并使用反射技术让通用的代理类能够执行和扩展任意访问对象中的任意方法。

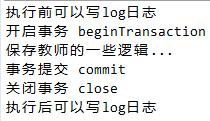
在Java中动态代理一般可分为两种，一种是JDK的动态代理，另一种是CGLib的动态代理。这两种代理也会在下面分别介绍。

我们现在不妨先接着上述案例，根据理解，先尝试用反射自己编写出动态代理的代码。定义一个通用的代理类：



编写测试类和输出结果：





可以看到输出结果与静态代理时输出的结果一致，并且对代理类封装了，再增加Service时也不用相应增加代理类了，可以说动态代理的效果已经实现了。

不过！

效果虽然已经实现了，但是味道变了。回顾上文，我们为了封装代理类，而引出了一个CommonProxy类，它使用了反射技术来达到通用代理的目的。细想一下，既然反射技术可以调用任意对象中的任意方法，那我全都用反射去调不就好了，在一个类里写一堆方法，再到某个地方调用足以，何必定义复杂的类之间的关系？

这就是典型的面向过程的思想了。一个好的程序，不应该只注重实现需求，还应该兼顾到可复用性、可维护性等。我们最开始就介绍了面向对象的7大原则，其目的都是基于面向对象的思想去设计程序，提倡高内聚、低耦合；如果到处都在使用像上图那样的反射的话（方法只需写一个字符串），那么如果一旦目标方法的名字更改了，或者参数构造更改了，那么到处都得跟着改，违背了开闭原则。

所以，上述的写法虽然实现了通用代理的效果，但它并不是一个好的设计。

再从形式上看，我们在使用spring的“依赖注入”时，按上述写法应该是

@AutoWired

protect CommonProxy commonProxy;

在使用时，应该如下：

commonProxy.doProxy(“saveTeacher”, ...(省略) );

但我们在实际开发中，并不是如此，我们会像这样写

@Resource

protect TeacherService teacherService;

在使用时代码如下：

teacherService.saveTeacher( ...(省略) );

这就很奇怪了，我们在使用时好像并没有看到与代理相关的代码，很正常地调用我们自己写的方法，但是在运行时它的确又被事务接管了（即使用了代理的“代码增强”特性），我们能知道的就是我们可能在配置中设定了aop对service层的包做了扫描。那么，Spring到底是怎么做到的？

下面，最精彩（最难理解）的部分来了。

首先，我们来攻破第一个盲点。

@Resource

protect TeacherService teacherService;

（这段话有点绕，如果不能理解可以先接着往下看）在这段代码中，其实teacherService对象拿到的并不是我们定义的TeacherServiceImpl的实例，**实际上是代理类的实例**。Spring参照teacherService接口生成了一个代理类，而后创建它的实例，并将我们定义的TeacherServiceImpl的实例创建出来扔进去用来处理我们自己的逻辑，而代理类的实例则负责对我们的逻辑做“代码增强”（这里就是指事务管理等），最后将这个代理类的实例注入到teacherService对象中。

Spring 的AOP总是优先使用JDK动态代理的，如果访问对象本身没有实现任何接口，则使用CGLib动态代理；但是SpringBoot 2.x的AOP是默认使用CGLib动态代理的（因为JDK的只能通过接口代理容易报错）。下面我们分别介绍这两种动态代理方式以及它们的区别：

JDK动态代理

JDK动态代理的核心源码在java.lang.reflect包下，在1.8版本后结合新特性做了一些调整，例如引进了WeakCache（弱引用缓存，在后面会介绍什么是弱引用，为什么要用弱引用）。

我们依然接着上面的案例，使用JDK动态代理方式来简单模拟：

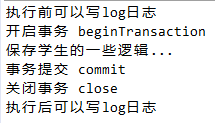
首先需要定义一个动态代理的处理类，它必须实现InvocationHandler接口（该接口在java.lang.reflect包下，它要求实现类需要重写一个invoke方法）。

内部代码与我们上面写的CommonProxy类似，构造中都需要传入一个Object对象（被代理对象，即访问者对象），我们需要在invoke方法中使用反射调用这个object对象的方法来实现我们自己的业务逻辑。相同的，在反射的上面和下面需要增加代码增强和环绕通知（这是我们创建代理的目的）。如下图：



接着，我们写测试代码，创建这个代理类并运行输出结果：





上面的测试代码是不是很像我们平常写的代码？我们最终拿到的都是studentService，调用的也是我们自己定义在接口里的方法，只是在使用spring依赖注入时，对我们隐藏了上面

new StudentServiceImpl() 的过程以及创建代理类实例的过程。

下面，我们了解一下这个JDK动态代理的原理是什么（下面分析JDK1.8版本的源代码）。

我们先从下面这行代码开始看



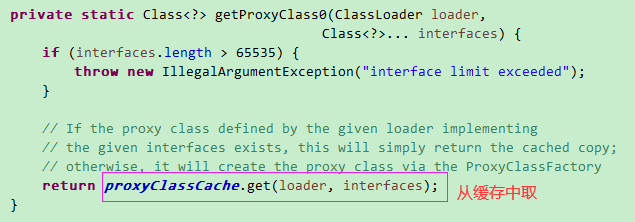
Proxy是java.lang.reflect包下提供的一个类，newProxyInstance是其中的静态方法，找到对应源码，如下图：



我们只关注标记起来的几处重点，其他不用管。首先，这个方法要求传入三个参数：

1. 被代理类的类装载器（后面会说到用处）
2. 需要代理的接口列表
3. 代理处理类

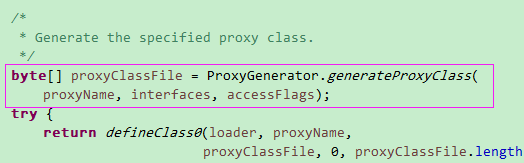
然后来到getProxyClass0方法，这个方法是用来获得代理类的，准确的说是缓存中有就从缓存中获取，缓存中没有就立即创建，这里的缓存是指WeakCache（1.8新特性的衍生类），源码如下：



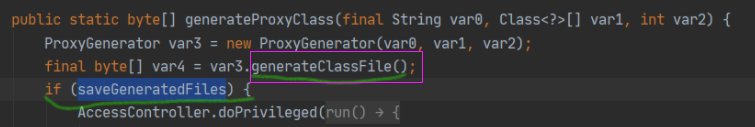
接着，在get方法里没有找到相关的逻辑，又去找proxyClassCache的定义



我们发现，这个WeakCache在定义时就为其初始化了。最后，顺藤摸瓜，终于在new ProxyClassFactory() 方法中找到了这样一段代码：



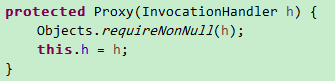
ProxyGenerator是sun.misc包下提供的一个类，看字面意思是代理生成器。找到generateProxyClass的源码：



再找到generateClassFile的源码

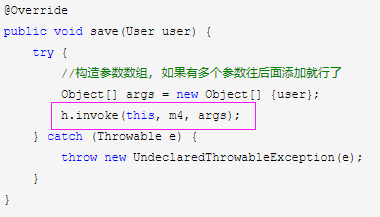


代码有点长，下面部分就不贴了，感兴趣自己去网上找找。这个方法的代码大概意思就是要用字节码的形式创建出一个代理类的class文件，并且这个代理类中要生成需要代理的接口中的方法，还要为代理类加上toString, hashCode, equals等等东西，并且这个代理类要继承自Proxy类，而Proxy类是有一个构造的。



所以代理类作为子类也会有这个构造，需要传入一个处理类（Handler），那么代理类内部也会保存这个handler。

最后，让代理类的所有方法全部调用handler的invoke方法。比如需要代理的接口中的方法（如userService中定义的save方法），如果是调用了handler中的invoke，就相当于在代理类中调用了传入的handler实现类里定义的invoke方法（即业余代码和代码增强、环绕通知）。这个跟使用接口实现回调函数是一样的原理。代码大致如下：

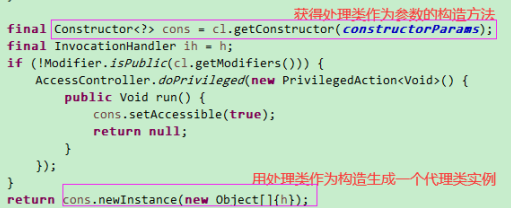


我们上面提到过，传入的参数除了需要代理的接口列表和Handler外还有一个类加载器，这个类加载器就是用来将生成的代理类class文件加载到内存中的。

到这里，我们就把getProxyClass0这个方法讲完了，cl拿到了代理类的类型。



还剩下的两句代码就很好理解了：



我们刚刚也说到了，代理类作为子类它也会有Handler这个参数的构造，因此cons就拿到了这个构造，最后将我们传入的自己定义好的Handler实现类作为参数去构造一个代理类实例。

这样就形成了一个“套娃”的结构，我们最后总结一下整个流程：

Spring为我们注入的是代理类实例，这个代理类实例中包含了一个Handler实例，这个Handler实例里呢又包含了一个Object实例（套娃），这个Object就是Impl（我们的service实现类，里面写着我们要做的业务代码），然后通过在invoke方法中使用反射来调用我们的业务代码，并且在上下文做代码增强（事务开启、提交、回滚、关闭等），以及环绕通知。

CGLib动态代理

CGLib的全称是Code Generation Library，是一个开源的高性能Code生成类库，底层使用了ASM（一个短小精悍的字节码操作框架）来操作字节码生成新的类。

下面用CGLib动态代理的方式来模拟上面的案例：

首先引入cglib的依赖



或

<dependency>

<groupId>cglib</groupId>

<artifactId>cglib</artifactId>

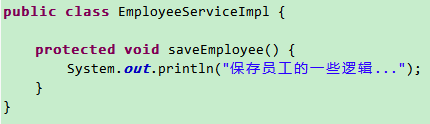
<version>3.1</version>

</dependency>

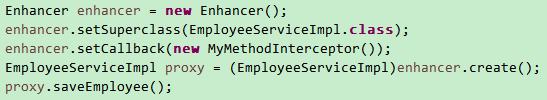
定义一个“方法拦截器”，类似JDK动态代理中的Handler

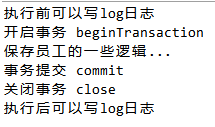


定义一个业务类，注意，这个业务类并没有实现任何Service



编写测试代码和输出结果

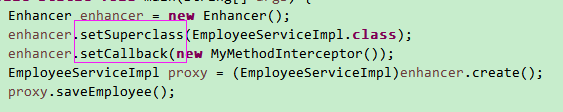




下面来解释CGLib动态代理的原理

CGLib动态代理的源码比JDK动态代理的源码复杂多了，也更难理解。因为不同版本的代码会不一样，下面分析CGLib 3.1版本的源代码为例。

先从测试代码看起

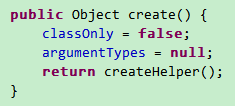


Enhancer是CGLib提供的一个增强器，用来生成代理类。往里面set父类和回调以后，会保存到这个增强器对象内部的两个变量里面，分别是

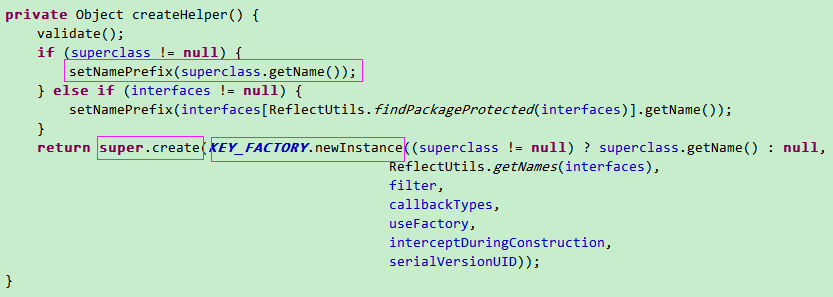




然后我们进入create方法，看它是怎么生成代理类的

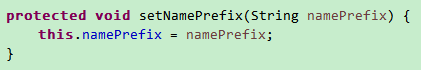


这一步看不出来，我们继续找createHelper方法



这一段代码里有3个重点

1. setNamePrefix方法，这个没什么好说的，将接口的名字设置到一个叫namePrefix的变量里，在后面给新生成的代理类命名有用。



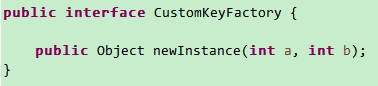
1. KEY\_FACTORY.newInstance，我们看一下KEY\_FACTORY的定义



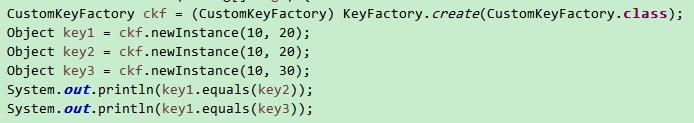
这个东西咋一看是看不懂的。这段代码里的KeyFactory，其实是CGLib提供的一个工具类，它是用来将一组数据放一起生成为一个key的，有点类似数据库的聚簇索引，一般用来作为map和set的key值。

它要求我们自己定义一个接口，这个接口可以是一个或多个入参（一组数据），然后返回一个Object。KeyFactory会根据我们自己定义的这个接口，给我们创建出一个拥有上述能力的实例（就是返回值Object），然我们可以用这个实例生成key。下面写一个例子来演示一下：

首先，定义一个这样的接口，里面有个newInstance方法，它的目的是要根据两个整数生成一个对应的key



然后编写测试类并输出结果





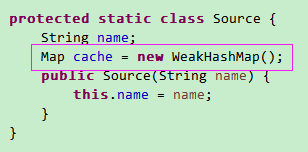
从结果中可以看出，如果一组数据相同，则生成出来的key值也相同（但它们并不是同一个实例，即地址是不同的）

现在再回顾上面的源代码，EnhanceKey就是我们定义的CustomKeyFactory，在KeyFactory.create后得到EnhanceKey的实例，再使用newInstance将一组乱七八糟的东西放一块去生成一个key值，然后作为super.create方法的入参。

1. super.create方法



上图是create方法的上半部分，线条标记区域内的代码是用来控制缓存机制运作的。我们找到缓存的定义：

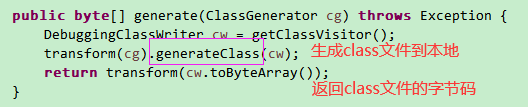


可以看出它是用一个弱引用HashMap模拟的，并且从上图的实际使用来看，它在根据ClassLoader作为key获取对应的值后，又强制转换为Map，这表明这个缓存是一个二级缓存，其存储结构应该是 Map<ClassLoader, Map<Object, WeakReference> >，其中Object就是传入的key，WeakReference是生成的代理类（的类型）并转变成弱引用。

如果是初次使用，缓存中没有时，它就会生成一个代理类的类型出来，并扔进缓存，如下图：



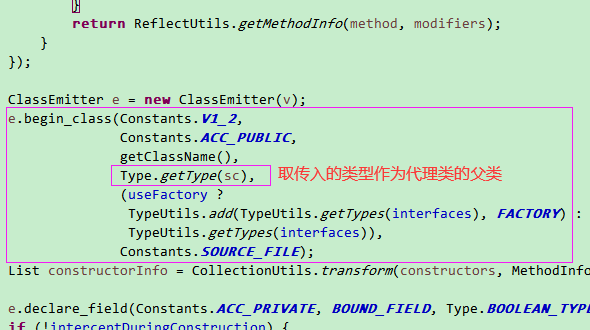
strategy是生成策略，我们继续找到strategy.generate方法



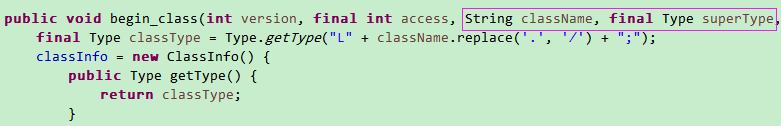
最后，进入generateClass方法，这个方法由Enhancer自己（作为AbstractClassGenerator的子类）来实现。

generateClass方法种代码很多，其中有一段代码如下图：





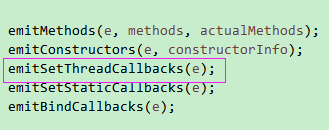
ClassEmitter可能是用来构造一个类的底层工具，其中的begin\_class方法如下图

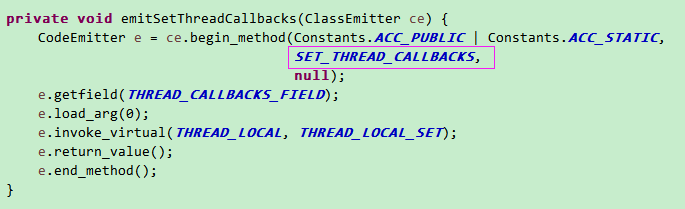


注意第4个参数规定生成的代理类的父类是谁，对应传入的是我们设定的EmployeeServiceImpl，即生成的代理类继承我们自己的业务类。第3个参数是规定了生成的代理类的类名是什么，对应传入的是getClassName方法，这里要提到命名策略NamingPolicy。因为生成出来的代理类不能和我们自己的业务类同名，因此它会在类的后面增加一些乱七八糟的东西，例如EmployeeServiceImpl$$EnhancerByCGLIB$$e1a36bab.class（以我们自己的业务类名字作为前缀，也就是前面说到的，将名字放进了变量namePrefix中，具体的命名规则这里不再赘述，感兴趣自己去看吧）。

现在我们还需要弄明白一个东西，就是方法拦截器到底是怎么工作的？

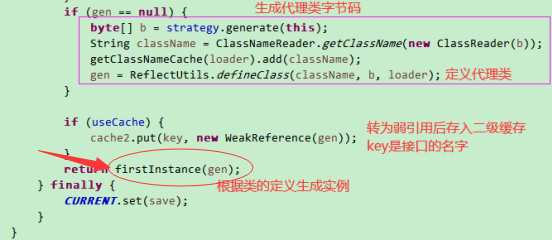
接着刚才的代码往下找，找到



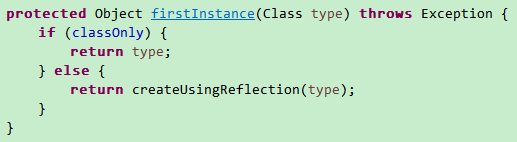


这段代码是往代理类里加一个set方法，这个set方法就是传入一个callback（方法拦截器），并将它赋给这个代理类。

到这里，generateClass讲完，其余代码感兴趣可以自己去研究，我们回到之前的create方法，最下面还有一行代码（firstInstance方法）没有讲。

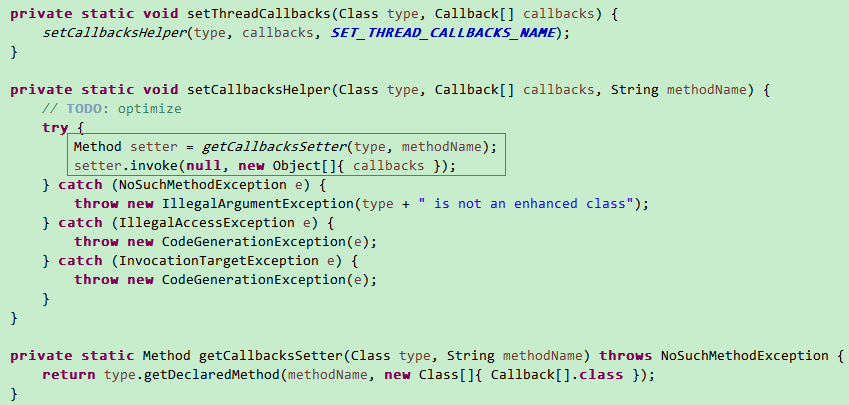


找到firstInstance方法的源码



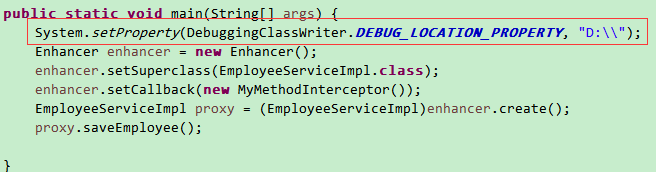


反射生成实例就不说了，我们找到setThreadCallbacks以及相关的几个方法

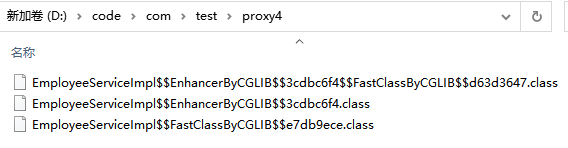


可以看出，最终是从类的所有方法声明中找到前面加在类里用来赋予代理类callback的setter方法，并且invoke了它，将传入的callbacks也就是我们最开始说的，从测试代码中传入的方法拦截器，写入进了代理类中。

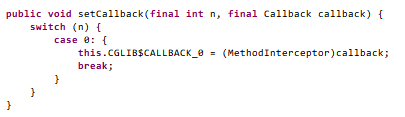
到这里，原理就讲完了，我们来看看生成出来的代理类长啥样。在测试类中增加一行代码：



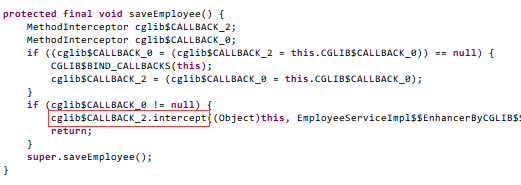
加上这行代码并运行后，就会在我设置的D盘里写出生成的代理类。



对中间那个进行反编译（工具luyten），可以在里面找到这样一段代码



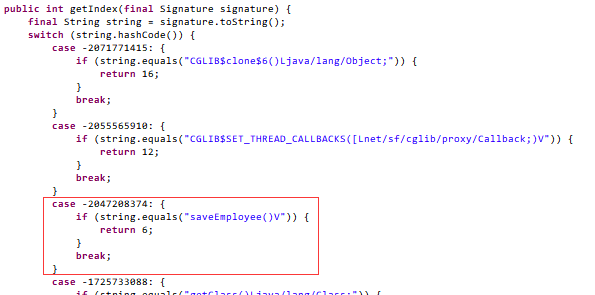
证明我们上面说的通过setter方法传入callback是正确的。再找到我们定义在业务类里的方法：

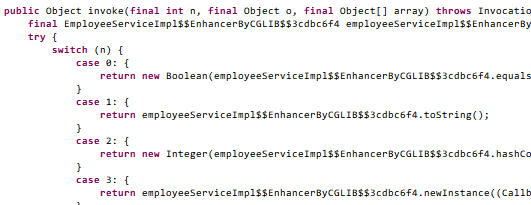


可以看出，当我们使用代理类调用saveEmployee方法时，会调用方法拦截器中的intercept方法，而我们在intercept中做了代码增强和环绕通知，以此来实现代理的效果。

这里，可以看出JDK动态代理和CGLib动态代理有一点区别。同样是生成出来的代理类，JDK的代理类中是使用反射去调用handler中的invoke方法；而CGLib觉得，反射比较耗费时间，所以搞了一个叫FastClass的机制，这个机制是在生成代理类的同时还会生成原业务类和代理类对应的FastClass类（就是为什么上面D盘写出来是3个文件）。

我们来看一下代理类对应的FastClass文件反编译后的内容，主要包含两个方法，一个是getIndex方法，另一个是invoke方法。





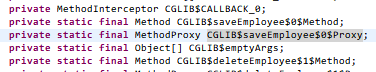
因为内容过多，这里截图只截取这两个方法中的部分代码，可以看出getIndex方法用来匹配一个方法，并返回一个数字；invoke方法传入一个数字并根据数字来调用对应方法，刚好能与getindex中的数字对上，即这个数字充当了“索引”。

上面已经说到了，在我们调用自己的业务方法时，代理类的callback会调用我们自己定义的方法拦截器中的intercept方法，回顾一下那个方法，里面有一句关键代码：

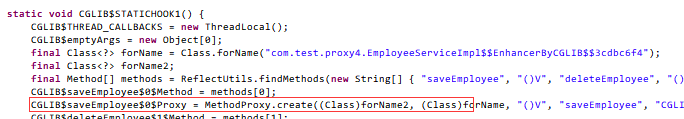


这个methodProxy是什么？invokeSuper又是怎么工作的？

对代理类的反编译内容中可以找到,在代理类的属性中会为每一个方法都分配一个MethodProxy：



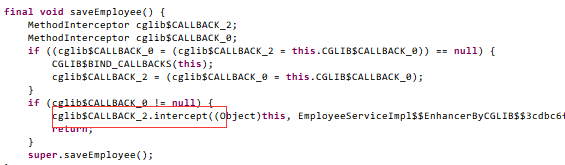
在一个叫“hook”的静态方法中，定义了创建MethodProxy的方法



有一个静态代码块调用了这个hook，这就表示代理类在创建时，所有方法对应的ProxyMethod也跟着创建出来了



然后在调用具体方法，进而调用到intercept方法的最后，将这个methodProxy传过去了



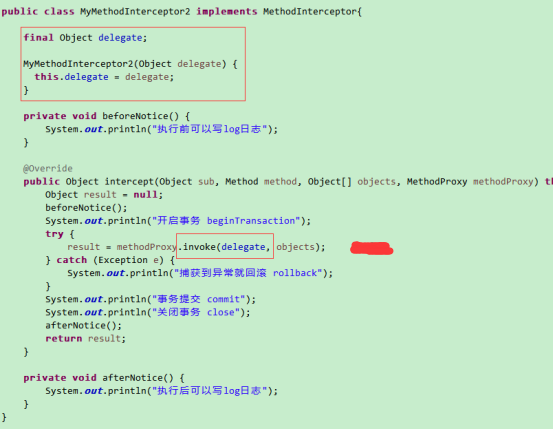


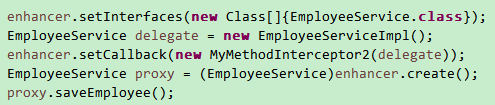
当在intercept中proxyMethod调用了invokeSuper时，会去找FastClass里的getIndex方法，匹配到对应的方法名后会得到一个数字索引，再用这个数字索引去invoke方法里调起真实的方法。

这种用索引的方式直接调用方法当然会比反射来的快一点，因此有CGLib动态代理比JDK动态代理更快的说法。但是，并不绝对！

从理论上来说，谁更快主要与接口或类中方法的数量有关，因为使用switch...case做索引也是需要一个一个进行值的比较，假设有n个方法，则索引其中一个方法的平均次数是(n+1)/2，每一次索引又需要花费时间，如果n的数值足够大，致使平均索引花费时间超过了反射的平均时间时，CGLib动态代理就不如JDK动态代理快了。从这里也能看出，如果我们使用CGLib做动态代理，常用的方法应该尽量写在上面，应该能让索引更快一点；要是从实际角度说不准，因为不同的版本优化程度不一样，在实际运行环境中也会有一些差异。

刚刚比较的是JDK动态代理和CGLib动态代理的性能，但是要知道CGLib动态代理并不是只能通过继承的方式来代理；我们的案例中是使用继承的方式，但是它也可以通过接口的方式进行代理的。





通过接口方式代理的写法与JDK的handler非常相似，这里必须将EmployeeServiceImpl的实例传进去，否则会进入死循环。

那么同样是CGLib做动态代理，通过继承方式和通过接口方式哪个性能更好？答案是接口，为什么？因为继承的类本身也是继承自Object类的，Object类里自带了一些方法，这些方法也会被写到FastClass的索引项里去；相反，接口里的就很干净了，所以接口的方式一般会稍微快于继承的方式。

两种动态代理的比较

1. 两种动态代理都是采用字节码方式生成代理类class；不同的是JDK使用sun的ProxyGenerator，而CGLib使用ASM。
2. JDK只能通过接口的方式代理，CGLib既可以通过继承，也可以通过接口的方式代理。
3. 不管哪种动态代理，我们使用的都是被Spring注入的代理类中的方法，但是执行逻辑不一样。JDK先使用反射进入handler中的invoke方法，再使用反射调用业务方法；CGLib则是通过set进来的callback直接调用Intercept中的invoke方法，再通过FastClass的索引快速找到要调用的业务方法，使用事先new出来的业务类实例调用。（即在代理过程中，JDK中使用了两次反射，CGLib没有用到）
4. JDK和CGLib内部都使用了弱引用缓存，JDK在1.8后使用新特性引入了WeakCache类。

弱引用 WeakReference

不管是JDK动态代理还是CGLib动态代理的源码中都引入了WeakReference这个类（WeakCache里面也使用了WeakReference），这个类是指弱引用。在Java中引用分为四种类型，分别是强引用，软引用，弱引用，虚引用（感兴趣自己百度学习），它们与GC（垃圾回收机制）有关。我们在日常开发中使用的基本都是强引用，它不管在任何时候都不会被GC自行回收；而弱引用会在被GC扫描到时无条件回收；这里使用弱引用的原因很简单，不管是JDK动态代理还是CGLib动态代理，它们都用字节码方式真实创建了一个代理类出来并装载到了内存里；虽然已经做了缓存，不会在使用代理时无限制地创建代理类了，但是如果只是临时用了一下以后就不会再用了呢，它仍然是会存在于内存中，造成内存泄露。因此，要把它做成弱引用，让GC每次扫描时清掉它。

代理模式的技术体现

IP代理、Spring AOP等

### 适配器模式 Adapter

还记得小时候是怎么将手机内置的小存储卡插入到电脑上的吗？一般的手机存储卡都是小卡（如下图左边），而电脑只能插入类似相机存储卡那种的大卡（如下图右边）。

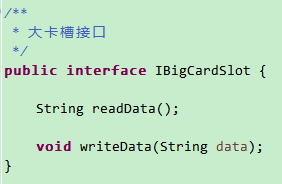
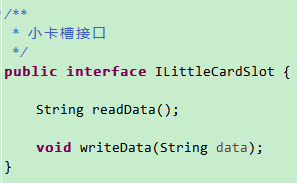
手机存储卡 相机存储卡 

所以，我们会用到一个东西叫SD卡套（如下图）将小卡插入卡套中，再将卡套插入电脑插口，这个SD卡套就充当了适配器的角色，它的正规名字就叫SD卡适配器。

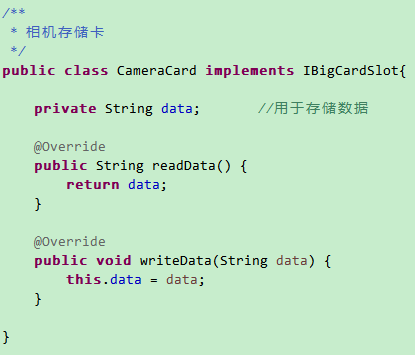
SD卡套 

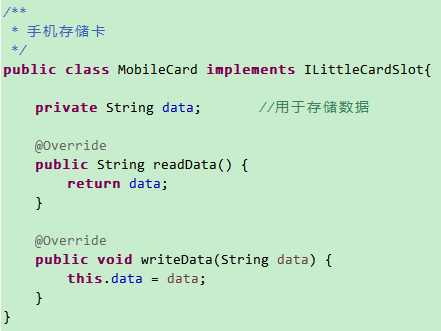
下面用代码来模拟这个过程：

首先，定义两种卡槽接口，大卡槽接口和小卡槽接口，并且这两种卡槽接口的能力都是读和写数据。

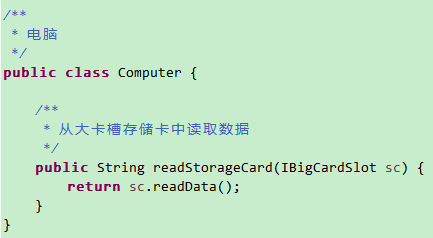
 

已知相机存储卡是实现了大卡槽接口的，手机存储卡是实现了小卡槽接口的。

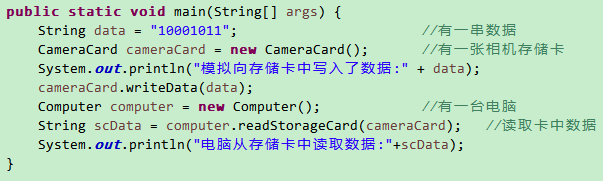




最后再定义一个电脑

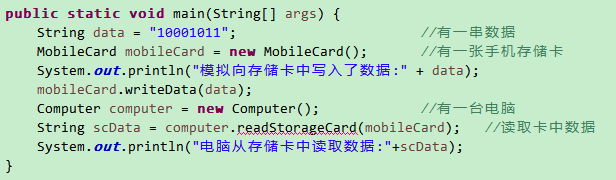


编写测试代码





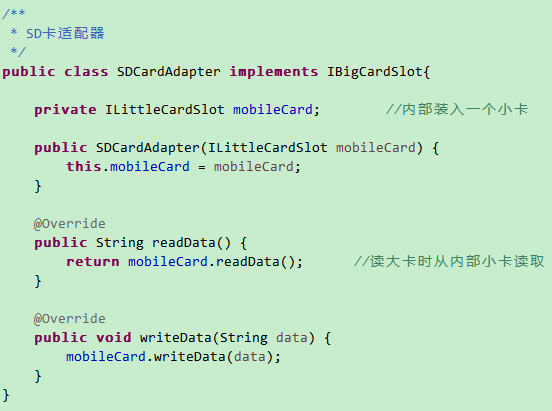
从结果看出成功用电脑读取出存储卡中的数据了，现在我们再试试用电脑直接读取小卡。

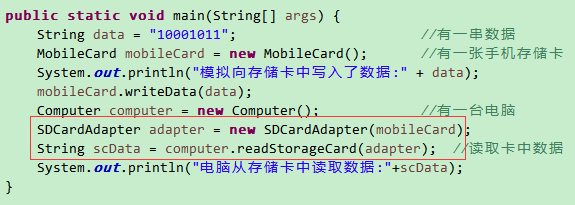


我们把相机存储卡替换为手机存储卡后看到，在电脑读取卡中数据那行代码处报错了，因为电脑的读取方法中规定了它只接受大卡，而手机存储卡是小卡，根本插不进去。

重点来了，我们需要引入一个适配器，这个适配器前面实现大卡槽，后面实现小卡槽，让小卡能够插入进去。

定义SD卡适配器







可以看到，引入适配器后，就可以使用电脑从手机存储卡中正常读取数据了。

最后总结一下适配器模式的应用场景

在一个系统中，如果出现既定的两个接口无法对接，又都需要用到它们时，可以引入适配器来连接这两个接口，并将一个接口的输入输出规范转换到另一个接口所需要的规范。

再举个例子，让电脑开机需要220V供电，电脑主机有标准的电源接口，它可能只接受12V的电压；但插座有标准的三插口的接口，它只提供220V的电压，那这两个接口肯定不能直接对接，所以引入电源适配器，将这两个接口连接起来，并且通过内置的变压器，将220V电压转为12V电压。

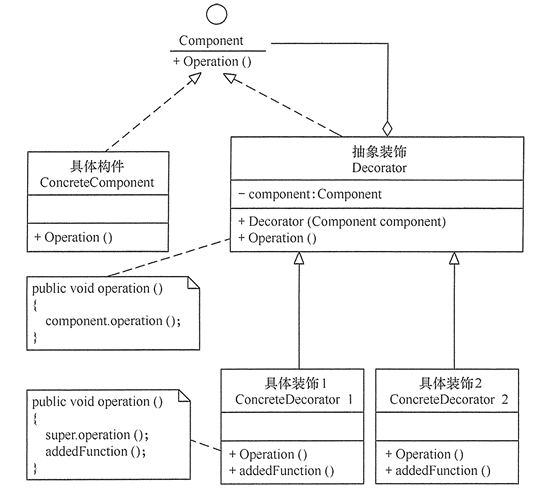
### 桥接模式 Bridge

还记得在第一部分介绍的面向对象7大设计原则中的合成复用原则吗？如果说合成复用是一个原则，是一个规范，那么按照这个规范写出来的代码就是桥接模式。

### 装饰器模式 Decorator

装饰器模式是基于一个已存在的类并在保留其原本方法的基础上扩展新的方法的一种设计模式。

先来看一下标准的装饰器模式对应的UML图



我的第一反应是：这么复杂？如果只是为了实现保留类原本的方法并扩展新的方法，那么写一个子类去继承它，并且在子类中写新的方法不就可以实现了吗？

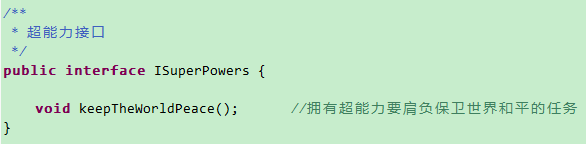
事实上，我在实际开发中也确实是这样使用的，例如我封装的CtDate继承自Date，CtList继承自ArrayList，它们确实能够实现我的需求，可为何大名鼎鼎的装饰器模式同样也是实现我这个需求却设计得更加复杂呢？

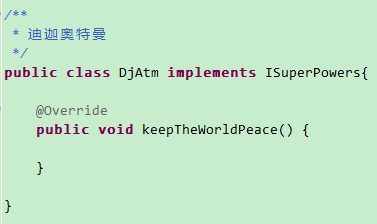
这个问题困扰了我近两个小时，直到我去思考一个问题：难道是基于直接继承的方式去扩展功能存在某种弊端？想到这里就一下子顿悟了，这不就是合成复用原则的意义吗？

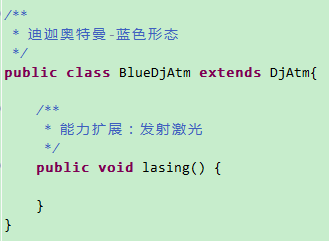
下面举个例子并用代码来讲解装饰器模式应该如何编写，以及上面所述的合成复用原则到底是怎么体现的。

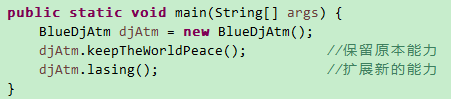
我们小时候应该都看过《迪迦奥特曼》，如果我没记错的话，迪迦奥特曼有三种形态，蓝色，红色以及红蓝复合形态，每种形态都有各自的特点。

假设，迪迦奥特曼是一个既有的类，那么三种形态的变换可以视为对迪迦奥特曼的增强（扩展），至此，我们可以写出如下代码。



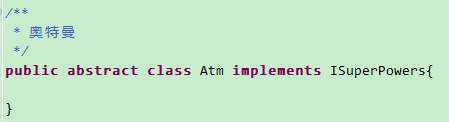


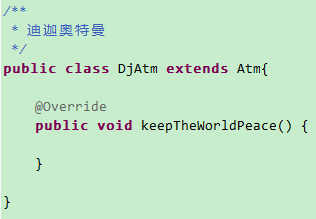




其实通过以上代码，我们已经实现了保留原本能力，并扩展新能力的需求，这种方式就是基于继承关系，子类对父类的能力进行扩展。但是它会带来一个什么后果呢？

事实上，我们这段代码只针对了迪迦奥特曼，而迪迦奥特曼又是奥特曼的子类，关系应该如下图所示。

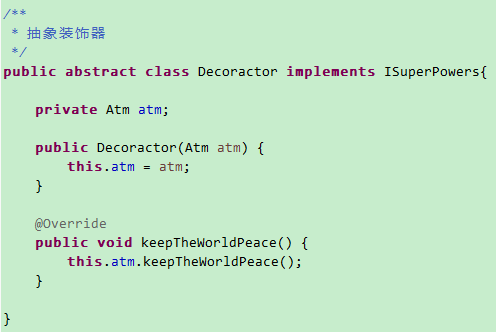




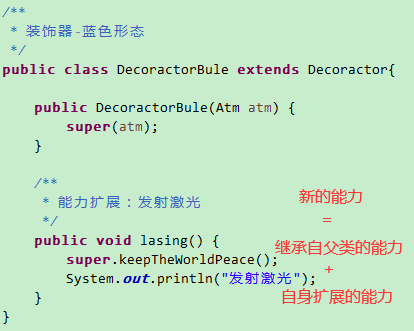
设想，如果有n个奥特曼，每个奥特曼都有3种形态，则基于继承关系的设计将会有3\*n个类，也正是为了解决这个问题，我们在本笔记开始的时候就引出了合成复用原则。

合成复用原则告诉我们，解决这个问题的办法就是抽取它们中共同的部分作为类的属性。下面开始介绍装饰器模式是如何编写的。

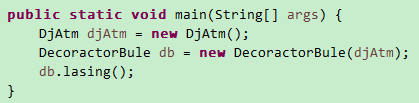
首先定义一个抽象装饰器，与奥特曼一起实现超能力接口，并将奥特曼作为它的属性。



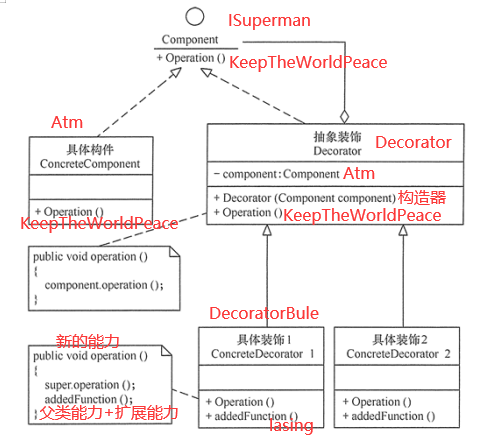
接着定义具体装饰器



修改测试类（迪迦奥特曼被蓝色装饰器装饰，可以使用扩展能力）



最后，回过头来对比一下刚开始的UML图



到这里，你或许有一个感觉，装饰器模式好像和适配器模式很像，尤其是他们都是将别的类作为自己的属性。

没错，实际上装饰器模式和适配器模式都属于包装模式（Wrapper），它们的结构非常相似，不同的是它们的设计目的和实际意义。适配器模式旨在通过简单的转换复用现有的功能，最终仍然只有一个功能。装饰器模式旨在不修改原本类的基础上扩展新的功能，实际上出现了两个功能。

### 外观模式 Facade

外观模式也叫门面模式，是迪米特法则的应用。按照迪米特法则的规范写出来的代码就是外观模式。

### 享元模式 Flyweight

所谓享元模式，享指的是共享，元指的是对象，即基于共享对象的一种设计模式，它注重对象的创建，维护以及如何共享使用。

说到对象的共享使用，不得不联想到单例模式。它们的区别我们在最后介绍，先来看一下什么是享元模式。

从代码结构上，我将其概括为通过工厂模式创建一批对象，利用某种数据结构来维护，以达到共享使用的目的。光是这样说还是难以理解，我来举个生活中最常见的例子，你需要从中体会我上面所说的话。

例如，我们在生活中用笔去写字，笔从何来？买一个或造一个，这都无所谓，从代码的角度不过是new Pen();而已。再细致一点，笔又分为黑笔BlackPen，红笔RedPen，它们都是Pen的子类，这时候我们为了更方便地创建对象，不妨引入工厂模式PenFactory。要搞来一支黑笔就是BlackPen bp = PenFactory.get(“blackPen”);（简单工厂模式）。

现在，我要搞来一支黑笔在纸上画一幅画。.... OK，假设我已经画完了，这时，我的同桌也想用黑笔去画一幅画，他也去搞来一支黑笔，那么目前就已经从工厂中弄出两支黑笔来了（即两个对象），假设全班的同学都需要用黑笔画一幅画，那么就会创建出几十个黑笔对象来。

基于上面描述的画面我们先考虑一个问题，如果全班同学要同时画画，那么每人都应该有一只黑笔，这是必不可少的。但如果并不要求全班同学同一时间去画画，或者说允许某些人先画完，某些人后画完呢。假设全班有40个同学，只有1支黑笔，则40个同学需要挨个排队画画，假设有2支黑笔，则这40个同学就可以两两排队画画，以此类推，我们可以知道，在这种情况下，需要几只黑笔完全由实际需求而定，在这种情况下，这几支黑笔对象完成了“共享”，而共享带来的好处就是，在不改变结果的前提下，实际创建的对象数量在减少，即内存开销得到减少。

为了让共享使用的过程更加有秩序，防止有人把黑笔用完以后乱丢或者乱传递给其他同学，制定了一个规则，就是在教室的前台放一个盒子，用完的人要放回盒子，需要用的人从盒子里拿。

现在，我们将上面的例子用专业名词来翻译一遍。黑笔对象实现了共享，因此黑笔即“享元”，而黑笔画出来的画却不一定是相同的，因此我们把黑笔画出来的画叫做“非享元信息”，盒子就是用来维护对象的数据结构（容器）。简而言之就是创建一批对象放在容器中，使用者使用这批相同的对象去做不同的事情，并且这些对象在被前一个使用者用完以后会放回容器，供其他使用者取用。

这...这不就是所谓的池化技术，线程池、连接池的部分原理吗？

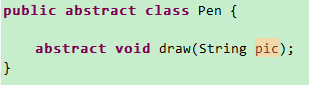
在线程池中初始化一些线程，需要的时候取出一个线程（享元），去做不同的业务逻辑（非享元信息），用完放回池中，需要时再从池中取。

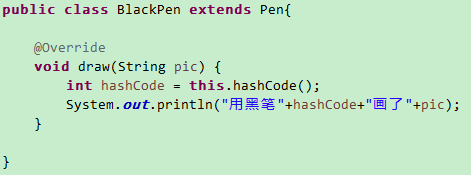
讲到这，你应该已经对享元模式的大致结构有一个了解了，这里再补充一点关于享元模式中的容器，到底采取什么数据结构，其实完全由你的实际需求决定。

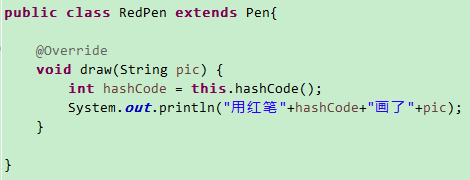
例如线程池，创建出来的线程都是一样的，大可以使用Set容器，但有些时候需要对你创建出来的这批对象进行分组来使用，像我上面举的例子，黑笔和红笔，可以使用Map作为容器，其中颜色作为key，对象作为value，需要使用黑笔时，先去这个Map中找找有没有key为blackPen对应的value，如果没有，用工厂去new一个出来并且放入Map中，下次再取的时候就有了吧，从这点看，容器其实就是充当缓存的作用啦。

俗话说好记性不如烂笔头，不如我们就把上面讲的红笔黑笔的例子用代码来模拟一下。

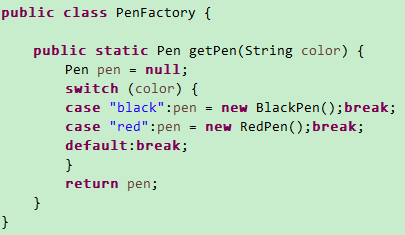
定义笔的相关类



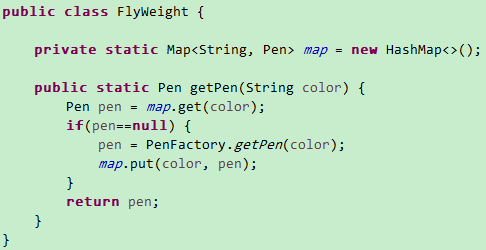




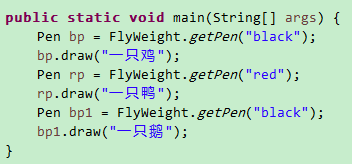
定义笔的工厂类

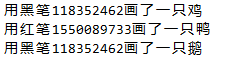


定义享元角色



定义测试类





可以从输出结果看到，第一个输出和第三个输出是同一个对象，因为是从享元角色类的map中取出来的，实现了对象的共享但做了不同的事情。

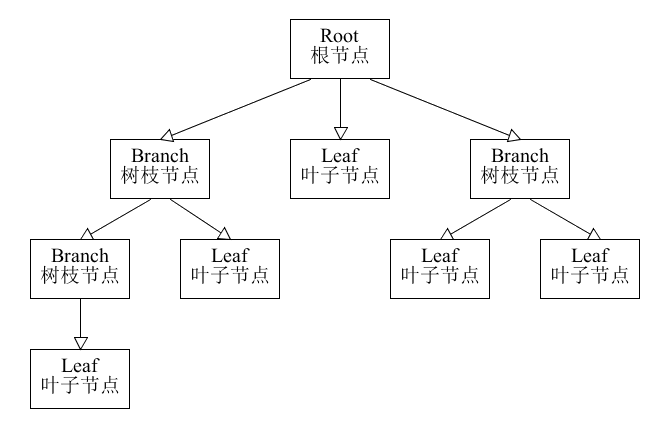
最后说一下开头提到的，享元模式和单例模式的区别是什么？

首先，相同点是它们的目的都是使用了共享对象的方式为了节省系统资源开销，其次结构和形式上有相似之处，不同的是单例模式限制了全局只能有一个对象，而享元模式可以根据实际情况创建更多对象。

### 组合模式 Composite

他是一种以树型结构来表示对象整体与部分关系的设计模式，其目的是获得操作上的一致性。

其应用典型的就是我们熟知的N叉树。



不管是根节点还是树枝节点还是叶子节点，如果能概括出它们的相似部分，将它们都定义成相同或相似的数据结构，就可以让操作性得到一致。例如：

定义二叉树节点的部分代码

Class Node {

Node leftChild;

Node rightChild;

getLeftChild(){

return this.leftChild;

}

get RightChild(){

return this.rightChild;

}

}

操作一致性的好处就是用一个循环或者递归就可以遍历其复杂的结构。当然，缺点就是所有对象都长的差不多，过于抽象，不利于具体业务上的理解。

## 行为型模式

### 模板方法模式 Template Method

模板方法模式是基于继承关系的一种设计模式，其本质是从一批具有相似结构或业务的类中抽出共同的，不变的部分放于父类中，让这些类成为子类，被抽出的功能可以被所有子类调用，并且一般不需要做修改。

例如我们在写Controller或Service时，经常会先创建出BaseController或BaseService，以BaseService为例，其中常用操作有save、update、list、get、delete即简单的增删改查。我们的任何具有特定业务的Service都可以继承自BaseService，它们共用父类的通用方法，并且可以在子类中扩展自己业务的方法。这其实就是模板方法模式的典型应用。

模板方法模式封装了不变的部分，扩展可变的部分，符合开闭原则，大大增强了代码复用性。其缺点主要是子类越来越多，系统又来又庞大。

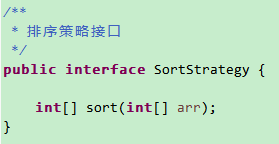
### 策略模式 Strategy

策略模式指定义一系列算法供客户端自由选择，各算法都有自己的适用场景，因此客户端可以通过自己的需要选择合适的策略。

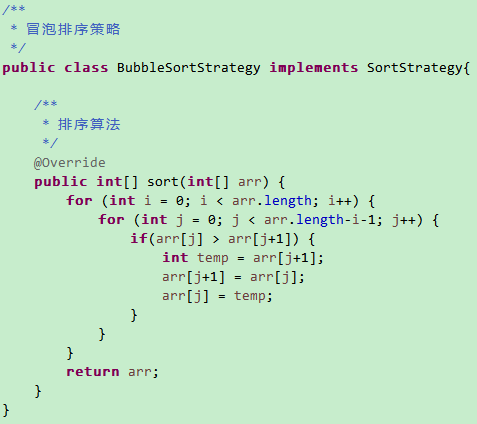
例如spring的aop在做动态代理时，是使用jdk的动态代理还是cglib动态代理完全是可配的，这其实就是策略模式的应用。

下面我们以排序算法为例，写一个使用策略模式的示例：

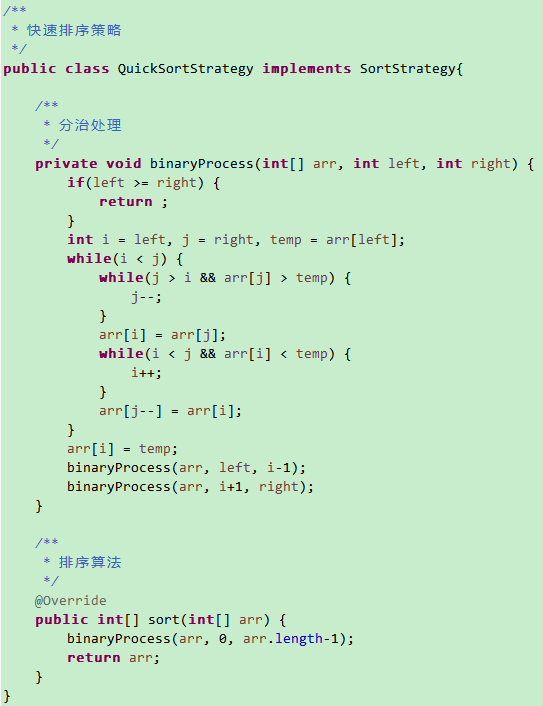
首先定义一个排序策略接口，其中要求实现一个排序算法。



其次定义一个冒泡排序策略，它实现了排序策略接口，则必须重写其中的排序算法。我们将经典的冒泡排序算法写进去。

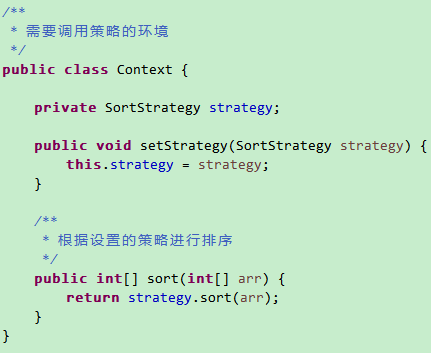


有了冒泡排序策略，我们还需要有其他的策略用于客户选择，这里选择用快速排序算法做一个快排策略。



当然你可以定义更多的排序策略用于替换，这里仅用于演示，就不写其他的排序算法了。

下面定义一个调用策略的环境，将策略对象装载在内部，并定义通用排序方法，在其中根据不同的策略对象调用到不同的排序算法。



最后定义测试类



输出结果：



因为1万个数字太长了，这里就没输出，只输出了算法的耗时情况用于对比。

最后对比一下与它相似的设计模式：

策略模式与工厂模式的区别

工厂模式只负责创建对象，不关心你使用对象中的什么功能（方法），策略模式注重实现某个功能的过程，因此工厂模式属于创建型模式，而策略模式属于行为型模式。事实上，策略模式经常与工厂模式结合使用，例如在上例中测试类new出具体策略类可以使用工厂模式代替。

策略模式和装饰器模式的区别

从结构上看，它们有相似之处，例如都是将对象装在内部，但是装饰器模式旨在扩展原本类的功能，它一般会先执行原本类的方法再执行扩展方法；而策略模式旨在提供一系列算法策略，让客户端根据实际情况自由选择合适的策略执行。

以上可以看出策略模式的优点是灵活，各场景适应能力强；缺点是对客户端要求较高，需要客户端理解各种策略的优劣和适用场景。

### 命令模式 Command

命令模式的目的是让某个操作的发起者和执行者实现解耦。

我们还以在介绍迪米特法则时提到的例子来讲解命令模式的运用。假设我新买了一个电脑，用了没几天出现了故障，现在我想返厂检修。从迪米特法则的介绍中可知让我直接去联系技术员是不合理的，站在客户的角度来看，你厂家只要把这东西修好就行，具体是谁修，怎么修我根本不用关心。因此我只需要联系售后中心即可，而后由售后中心内部去选举技术员来修理。

但在实际生活中操作还是会有一些细节方面的问题。例如我需要将电脑送到厂家，备注好故障描述、寄回收件人等信息；同时，对于厂家来说，会出现技术员太忙了需要排队，然后国庆节了技术员集体休假，休假结束后再继续做，要记录下这次维修方便日后备查和统计，如果修不好要能还原成寄来的样子等等问题。把这些问题总结一下就是：

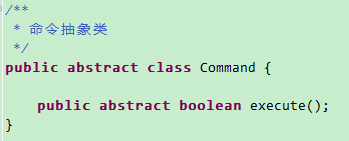
1. 需要支持将某些信息捆绑在一起（备注故障描述、寄回收件人等信息）
2. 需要支持排队（技术员太忙了）
3. 需要支持数据可存储（电脑修到一半，遇到国庆节集体休假，休假结束后再继续修）
4. 需要支持写日志备查（记录下这次维修方便日后备查）【扩展】
5. 需要支持还原（尝试了一下结果发现水平不够，要还原到送来的状态）【扩展】

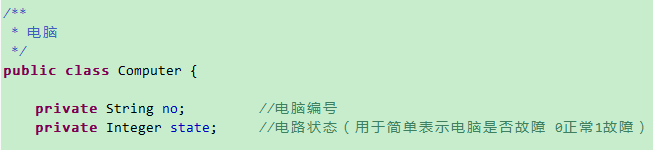
实际上，命令模式的核心就是引入一个对象的概念来解决这些问题（一个命令即一个Command对象）。

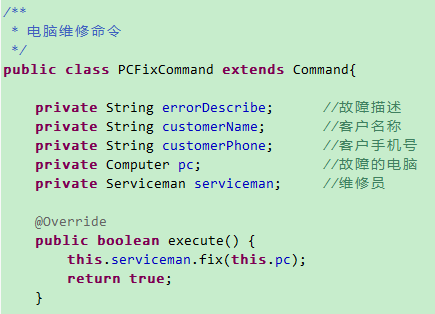
对象中可以保存需要的各种信息，可以引入队列来实现排队，可以通过序列化来存储，可以根据对象中信息写维修日志，可以通过备份初始状态的对象来实现还原（即后面会讲到的备忘录模式）

下面用代码来模拟上述场景

先建立命令抽象类和具体实现类，命令类中应该包含捆绑的相关信息

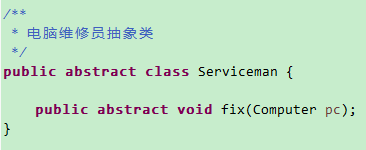


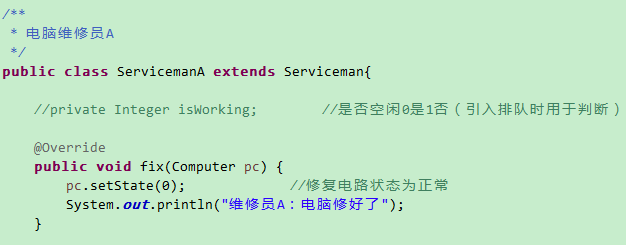




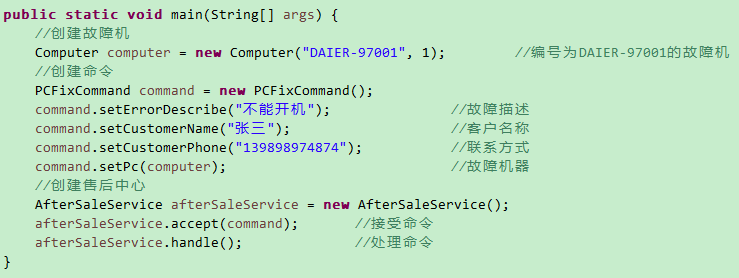
其次需要有售后中心、维修员

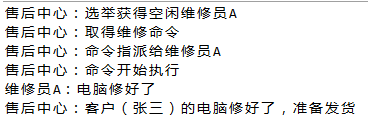






最后编写测试类





### 责任链模式 Chain of Responsibility

责任链模式是基于“数据链表”结构（链表）实现的一种设计模式，适用于需要按一定顺序且有职责划分的事件，其目的是使事件的发起者和处理者解耦（即请求发起后，事件被提交到链条上，发起人无需再关心该事件如何在各个节点上流转，每个节点的处理人是谁，以及处理方式是什么，只需关心最终的结果即可）。

下面举一个生活中的实例

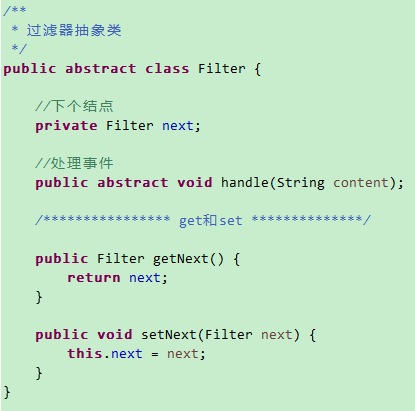
某学校请假机制如下：请假天数小于3天，可以直接由班主任审批通过；超过3天小于7天必须由年级主任审批通过；超过7天必须由校长审批通过。假如小明要请假，对于小明来说他不需要知道学校的请假机制，因为他只需要向他的班主任提出请假请求，再由班主任决定自己是否有权审批，如果没有权限，则链式地向上级传递，直到该请求被某级别处理。

通过上述描述不难联想到：Struts或Spring Security 的 过滤器链（Filter Chain），工作流引擎的设计中都包含了责任链模式。

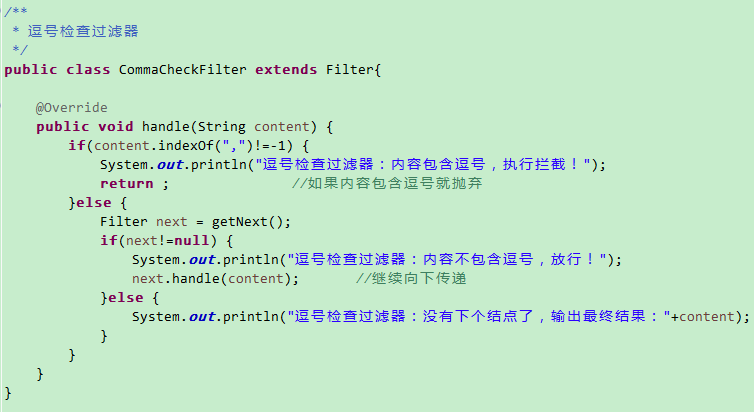
责任链模式的特点就是将各处理节点组合起来变成链式的结构，每个节点只保存后一个节点的信息，最重要的是每个节点可以根据自己的情况来决定自己是否处理事件以及是否继续向下一节点流转。

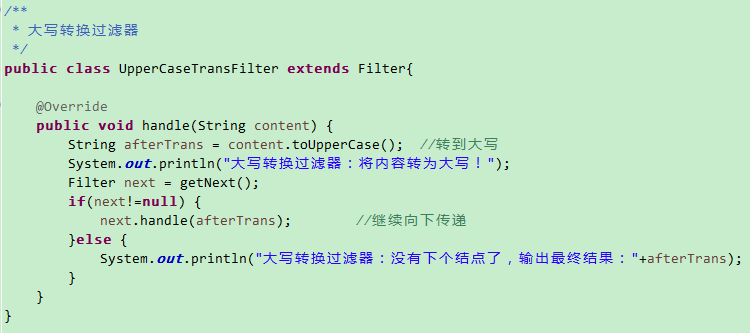
最后自己编写一个过滤器链的简单示例

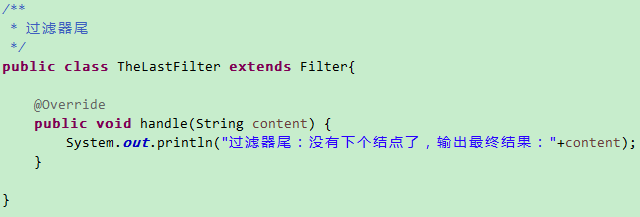
首先编写过滤器抽象类，其中应该包含指向下一个节点的引用和处理事件的方法



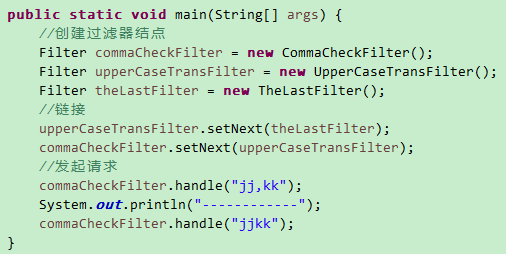
然后模拟三个有具体功能的过滤器

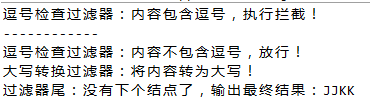






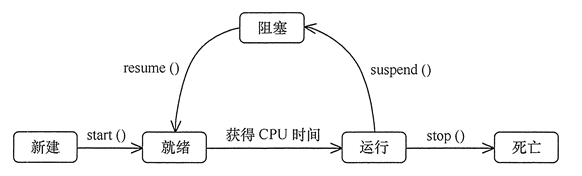
最后编写测试类





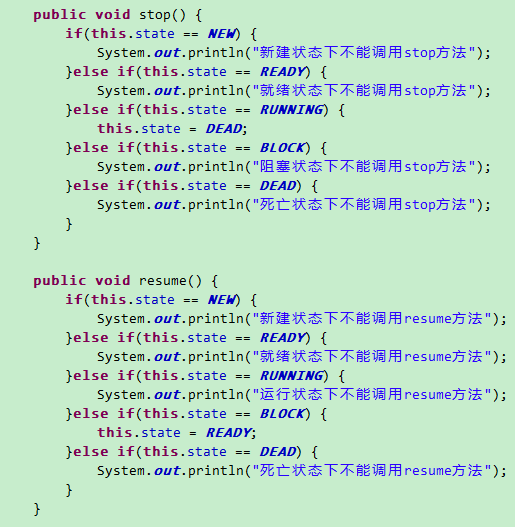
### 状态模式 State

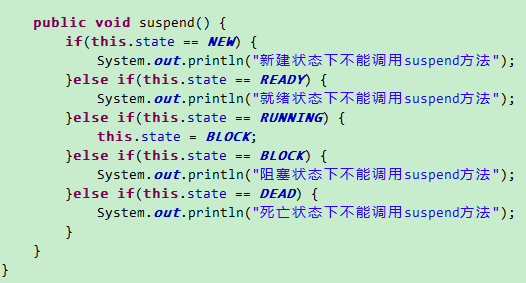
先来看一个例子



上图为线程状态转换图，可以从图中知道，某些方法只有在特定的状态下才能调用，如果让你编写代码来表示出图中的逻辑你会怎么写？你是否也会写出类似下图的代码？





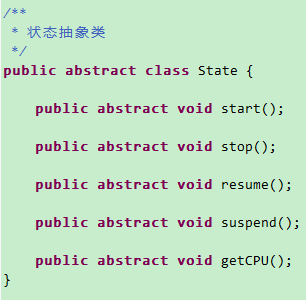
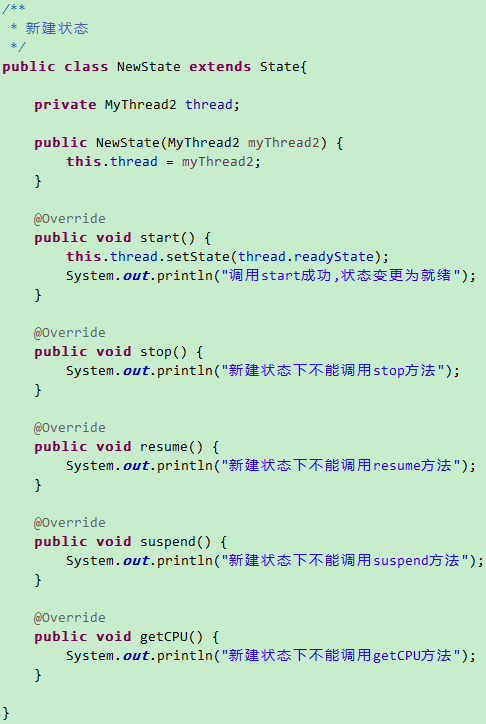


你可能会说，这些方法中其实只要一个if和else就可以了，即在else中输出非xx状态下不能调用xx方法，根本不需要写那么多else if语句。这样其实也没错，但是为了让这个例子更通用，我需要让它表现出在每种状态下都去做一件事情。

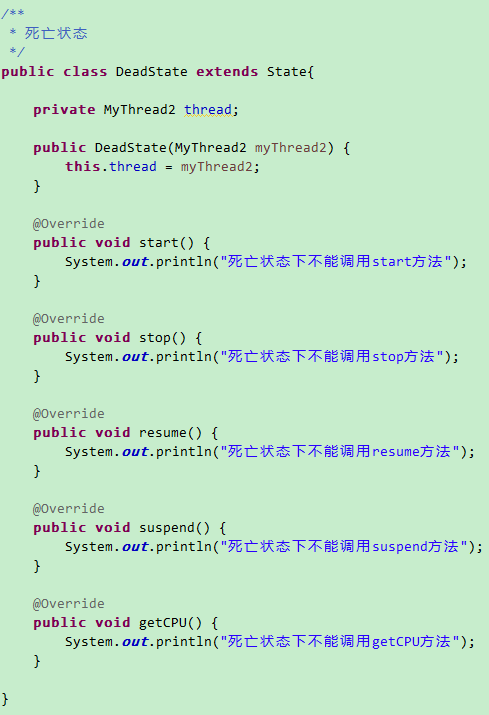
在此基础上可以看出，if else结构的判断的数量=方法总数\*状态总数，即每多一个方法就要增加状态总数个判断，每多一个状态就要增加方法总数个判断，并且在这么密集的判断语句中，如果以后需要变更状态之间的转换规则，想要理清原来的逻辑都很困难。

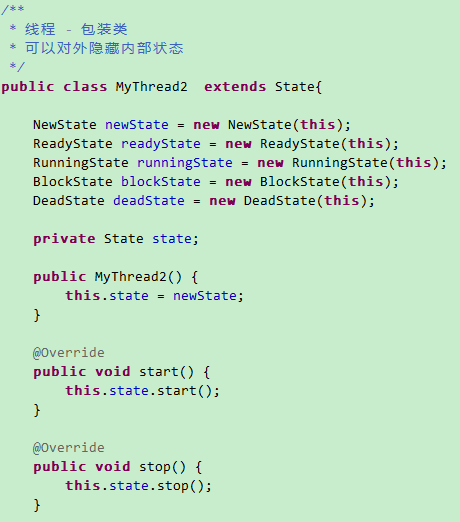
为什么困难，因为我们在以上帝视角去管理所有的状态和方法，其实就是在面向过程编程。这就像是让你指挥交通，如果是面向过程编程，那么你要做很多事情，例如要计算红绿灯的倒计时间，又要在规定时间内控制某些车辆通过，这是很累的一件事。比面向过程编程更合理的是需要我们面向对象编程，当我们将所有车辆都视为对象，它们都有自己的灵魂和思想时，我们就只需要制定一套规则就好，让每个对象自己去判断在面对红灯的状态下和在面对绿灯的状态下该做什么事情，这样就变得简单多了。简而言之，就是让“状态”变为对象，且每个状态的对象都只需要关心自己该干什么。

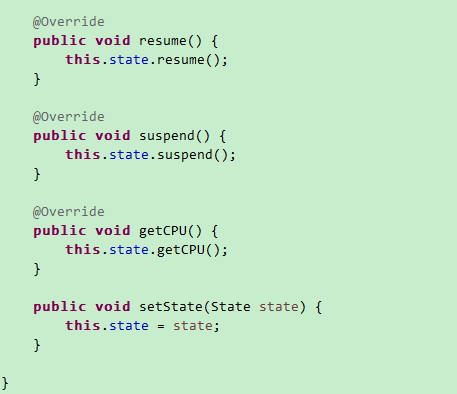
下面来看一下使用“状态模式”设计的代码（为了排版把图片缩小了，请放大查看）

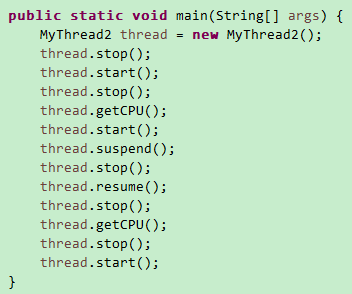
 

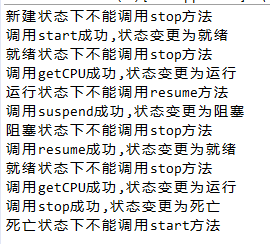
 





最后编写测试类





在使用状态模式后，如果要增加新的状态，只需增加一个新的状态类，修改与它有关联的状态的代码即可。

总结一下，如果业务中出现“行为随状态改变而改变”的场景时，可以考虑使用状态模式，其优点是增强程序可扩展性和代码可读性，缺点是增加了类的数量，这也是使用所有设计模式必须付出的代价。

### 观察者模式 Observer

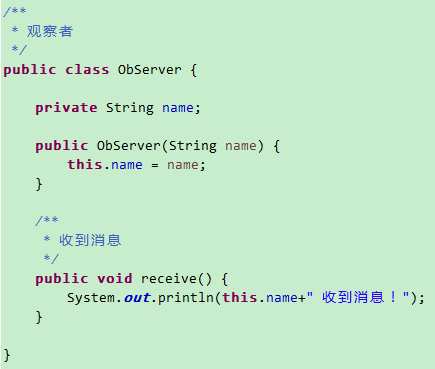
观察者模式也叫发布/订阅模式。

先来举一个例子，有十个家庭被困在了一座孤岛上，他们分居在岛的不同地方，如果能有船到达这个小岛，他们就可以坐着船离开。于是他们每个家庭都派出一个人到小岛的岸边，十个人就这样每天都在这里干等着，一旦有船只靠岸了，它们就各自回家带着家人准备离开小岛。

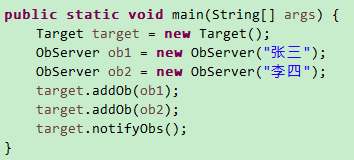
还是上面的例子，如果使用观察者模式去安排，就变成了这样：每天守在岸边的十个人中只留下一个人，其余人都回家劳作，留下的人记录下所有家庭的地址，如果有船只靠岸，他可以在按顺序通知到所有家庭（假设船只靠岸时间足够长），那么在同样的结果下，他们的工作效率提升了。

观察者模式的结构非常简单，其核心是在目标中创建观察者List，用于保存所有观察者的引用，以便统一通知。下面列举一种简单的写法：





编写测试类





消息队列中如果使用了JMS的发布/订阅模型，其中就包含了观察者模式的应用。

### 中介者模式 Mediator

中介者模式是“迪米特法则”的典型应用。

### 迭代器模式

### 访问者模式

### 备忘录模式

### 解释器模式

# 三．衍生的设计模式

### 反应器模式Reactor

由 观察者模式 衍生

技术体现

NIO模型