# 设计模式

**By Brother.Lee 20180116 Shanghai**

设计模式,以前也总结过一些,代码也写过,但是时间一久,大概也都忘却了.

这次也想总结一些,方便以后查阅使用.

知乎上的一些见解:

其实GoF的《设计模式》一书，一共有三个层面的内容:   
1. 指出编程开发活动中存在模式，提出总结设计模式需要关注的四要素 "名称-问题-解决方案-效果“ ，并给出描述一套模式的格式模板。  
2. 提出了面向对象开发中”针对接口编程优于针对实现编程”，”组合优于继承”的总体设计思路   
3. 选取了现实开发中基于上述设计思路所形成的23种常见设计模式作为例子详细描述

虽然第3点包括的多个具体的设计模式实例占据了最多的篇幅，但事实上第1,2点才是纲。实际上《设计模式》一书最划时代的意义，在于第1点。在此之后，出现了以设计模式的格式来组织内容的《分析模式》，《企业架构模式》，《企业集成模式》，《xUnit测试模式》，《重构》等等质量颇高的书籍.

在书中有一段我认为非常重要但很容易被忽略的话:

本书中涉及的设计模式**并不描述新的或未经证实的设计**，我们只**收录那些在不同系统中多次使用过的成功设计**。这些设计的**绝大部分以往并无文本记录**，它们或是来源于面向对象设计者圈子里的非正式交流，或是来源于某些成功的面向对象系统的某些部分，但对**设计新手**来说，这些东西是很难学得到的。尽管这些设计**不包括新的思路**，但我们**用一种新的、便于理解的方式将其展现给读者**，即：具有统一格式的、已分类编目的若干组设计模式。

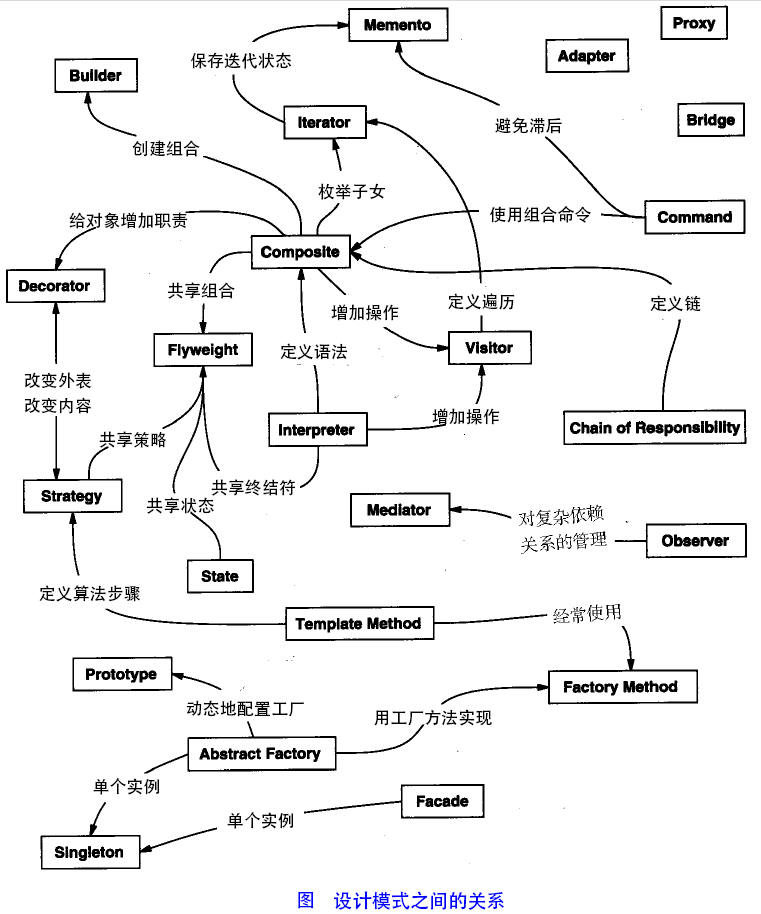
这段话的关键是：

1. 书中的模式不是作者的发明创造或独门秘籍，而是早已存在并已经广泛使用的做法，只不过没有被系统地加以记录。换而言之，只要遵循某些原则，这些所谓模式完全可能在无意识的状态下自发出现在产品代码中。

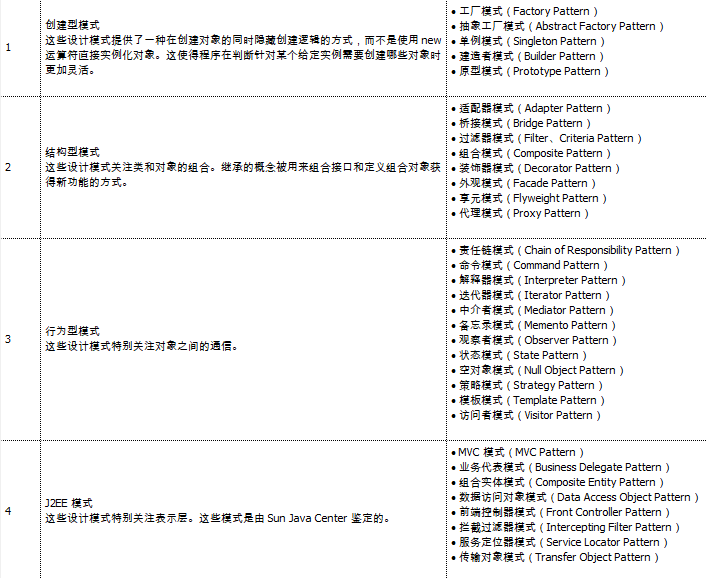
2. 这些模式在各种系统被多次使用。换而言之，你只要接触足够多的代码，必然会大量接触到这些模式的实际应用。只不过在看过《设计模式》一书之前，你可能意识不到这是一个成体系的设计手段。

3. 作者认为《设计模式》这本书的价值在于对设计模式进行了有效的组织和便于理解的描述。换而言之，这本书的写作出发点是”便于理解“，并且是面向”设计新手“的。而不少初学者却恰恰觉得这本书难以理解，这说明，作者已经在保证准确性的前提下，选用了他们所认为最便于理解的描述。比本书描述更为显浅的描述，很可能会牺牲准确性（不准确的描述对于新手来说是显然是害处大于好处）。当然某些人认为是作者表达能力有限，这种事情无法求证，但我倾向于前者。

## 设计模式之间的关系:



## 设计模式的分类:



## \*面向对象设计原则概述

对于面向对象软件系统的设计而言，在支持可维护性的同时，提高系统的可复用性是一个至关重要的问题，如何同时提高一个软件系统的可维护性和可复用性是面向对象设计需要解决的核心问题之一。在面向对象设计中，可维护性的复用是以设计原则为基础的。每一个原则都蕴含一些面向对象设计的思想，可以从不同的角度提升一个软件结构的设计水平。

面向对象设计原则为支持可维护性复用而诞生，这些原则蕴含在很多设计模式中，它们是从许多设计方案中总结出的指导性原则。面向对象设计原则也是我们用于评价一个设计模式的使用效果的重要指标之一，在设计模式的学习中，大家经常会看到诸如“XXX模式符合XXX原则”、“XXX模式违反了XXX原则”这样的语句。

|  |  |  |
| --- | --- | --- |
| 设计原则名称 | **定**  **义** | 使用频率 |
| 单一职责原则  (Single Responsibility Principle, SRP) | 一个类只负责一个功能领域中的相应职责 | ★★★★☆ |
| 开闭原则  (Open-Closed Principle, OCP) | 软件实体应对扩展开放，而对修改关闭 | ★★★★★ |
| 里氏代换原则  (Liskov Substitution Principle, LSP) | 所有引用基类对象的地方能够透明地使用其子类的对象 | ★★★★★ |
| 依赖倒转原则  (Dependence  Inversion Principle, DIP) | 抽象不应该依赖于细节，细节应该依赖于抽象 | ★★★★★ |
| 接口隔离原则  (Interface Segregation Principle, ISP) | 使用多个专门的接口，而不使用单一的总接口 | ★★☆☆☆ |
| 合成复用原则  (Composite Reuse Principle, CRP) | 尽量使用对象组合，而不是继承来达到复用的目的 | ★★★★☆ |
| 迪米特法则  (Law of Demeter, LoD) | 一个软件实体应当尽可能少地与其他实体发生相互作用 | ★★★☆☆ |

### \*.1.面向对象设计原则:单一职责原则

单一职责原则是最简单的面向对象设计原则，它用于控制类的粒度大小。单一职责原则定义如下：

**单一职责原则(Single Responsibility Principle, SRP)：一个类只负责一个功能领域中的相应职责，或者可以定义为：就一个类而言，应该只有一个引起它变化的原因。**

单一职责原则告诉我们：一个类不能太“累”！在软件系统中，一个类（大到模块，小到方法）承担的职责越多，它被复用的可能性就越小，而且一个类承担的职责过多，就相当于将这些职责耦合在一起，当其中一个职责变化时，可能会影响其他职责的运作，因此要将这些职责进行分离，将不同的职责封装在不同的类中，即将不同的变化原因封装在不同的类中，如果多个职责总是同时发生改变则可将它们封装在同一类中。

    单一职责原则是实现**高内聚、低耦合**的指导方针，它是最简单但又最难运用的原则，需要设计人员发现类的不同职责并将其分离，而发现类的多重职责需要设计人员具有较强的分析设计能力和相关实践经验。

### \*.2.面向对象设计原则:开闭原则

开闭原则是面向对象的可复用设计的第一块基石，它是最重要的面向对象设计原则。开闭原则由**Bertrand  Meyer**于1988年提出，其定义如下：

**开闭原则(Open-Closed Principle, OCP)：一个软件实体应当对扩展开放，对修改关闭。即软件实体应尽量在不修改原有代码的情况下进行扩展。**

  在开闭原则的定义中，软件实体可以指一个软件模块、一个由多个类组成的局部结构或一个独立的类。

      任何软件都需要面临一个很重要的问题，即它们的需求会随时间的推移而发生变化。当软件系统需要面对新的需求时，我们应该尽量保证系统的设计框架是稳定的。如果一个软件设计符合开闭原则，那么可以非常方便地对系统进行扩展，而且在扩展时无须修改现有代码，使得软件系统在拥有适应性和灵活性的同时具备较好的稳定性和延续性。随着软件规模越来越大，软件寿命越来越长，软件维护成本越来越高，设计满足开闭原则的软件系统也变得越来越重要。

      为了满足开闭原则，需要对系统进行抽象化设计，抽象化是开闭原则的关键。在Java、C#等编程语言中，可以为系统定义一个相对稳定的抽象层，而将不同的实现行为移至具体的实现层中完成。在很多面向对象编程语言中都提供了接口、抽象类等机制，可以通过它们定义系统的抽象层，再通过具体类来进行扩展。如果需要修改系统的行为，无须对抽象层进行任何改动，只需要增加新的具体类来实现新的业务功能即可，实现在不修改已有代码的基础上扩展系统的功能，达到开闭原则的要求。

### \*.3.面向对象设计原则:里氏代换原则

      里氏代换原则由2008年图灵奖得主、美国第一位计算机科学女博士**Barbara Liskov**教授和卡内基·梅隆大学Jeannette Wing教授于1994年提出。其严格表述如下：如果对每一个类型为S的对象o1，都有类型为T的对象o2，使得以T定义的所有程序P在所有的对象o1代换o2时，程序P的行为没有变化，那么类型S是类型T的子类型。这个定义比较拗口且难以理解，因此我们一般使用它的另一个通俗版定义：

**里氏代换原则(Liskov Substitution Principle, LSP)：所有引用基类（父类）的地方必须能透明地使用其子类的对象。**

里氏代换原则告诉我们，在软件中将一个基类对象替换成它的子类对象，程序将不会产生任何错误和异常，反过来则不成立，如果一个软件实体使用的是一个子类对象的话，那么它不一定能够使用基类对象。例如：我喜欢动物，那我一定喜欢狗，因为狗是动物的子类；但是我喜欢狗，不能据此断定我喜欢动物，因为我并不喜欢老鼠，虽然它也是动物。

      例如有两个类，一个类为BaseClass，另一个是SubClass类，并且SubClass类是BaseClass类的子类，那么一个方法如果可以接受一个BaseClass类型的基类对象base的话，如：method1(base)，那么它必然可以接受一个BaseClass类型的子类对象sub，method1(sub)能够正常运行。反过来的代换不成立，如一个方法method2接受BaseClass类型的子类对象sub为参数：method2(sub)，那么一般而言不可以有method2(base)，除非是重载方法。

      里氏代换原则是实现开闭原则的重要方式之一，由于使用基类对象的地方都可以使用子类对象，因此在程序中尽量使用基类类型来对对象进行定义，而在运行时再确定其子类类型，用子类对象来替换父类对象。

在使用里氏代换原则时需要注意如下几个问题：

      (1)子类的所有方法必须在父类中声明，或子类必须实现父类中声明的所有方法。根据里氏代换原则，为了保证系统的扩展性，在程序中通常使用父类来进行定义，如果一个方法只存在子类中，在父类中不提供相应的声明，则无法在以父类定义的对象中使用该方法。

      (2)  我们在运用里氏代换原则时，尽量把父类设计为抽象类或者接口，让子类继承父类或实现父接口，并实现在父类中声明的方法，运行时，子类实例替换父类实例，我们可以很方便地扩展系统的功能，同时无须修改原有子类的代码，增加新的功能可以通过增加一个新的子类来实现。里氏代换原则是开闭原则的具体实现手段之一。

      (3) Java语言中，在编译阶段，Java编译器会检查一个程序是否符合里氏代换原则，这是一个与实现无关的、纯语法意义上的检查，但Java编译器的检查是有局限的。

### \*.4.面向对象设计原则:依赖倒转原则

       如果说开闭原则是面向对象设计的目标的话，那么依赖倒转原则就是面向对象设计的主要实现机制之一，它是系统抽象化的具体实现。依赖倒转原则是**Robert C. Martin**在1996年为“C++Reporter”所写的专栏Engineering Notebook的第三篇，后来加入到他在2002年出版的经典著作“**Agile Software Development, Principles, Patterns, and Practices**”一书中。依赖倒转原则定义如下：

**依赖倒转原则(Dependency Inversion  Principle, DIP)：抽象不应该依赖于细节，细节应当依赖于抽象。换言之，要针对接口编程，而不是针对实现编程。**

      依赖倒转原则要求我们在程序代码中传递参数时或在关联关系中，尽量引用层次高的抽象层类，即使用接口和抽象类进行变量类型声明、参数类型声明、方法返回类型声明，以及数据类型的转换等，而不要用具体类来做这些事情。为了确保该原则的应用，一个具体类应当只实现接口或抽象类中声明过的方法，而不要给出多余的方法，否则将无法调用到在子类中增加的新方法。

      在引入抽象层后，系统将具有很好的灵活性，在程序中尽量使用抽象层进行编程，而将具体类写在配置文件中，这样一来，如果系统行为发生变化，只需要对抽象层进行扩展，并修改配置文件，而无须修改原有系统的源代码，在不修改的情况下来扩展系统的功能，满足开闭原则的要求。

      在实现依赖倒转原则时，我们需要针对抽象层编程，而将具体类的对象通过**依赖注入**(DependencyInjection, DI)的方式注入到其他对象中，依赖注入是指当一个对象要与其他对象发生依赖关系时，通过抽象来注入所依赖的对象。常用的注入方式有三种，分别是：**构造注入，设值注入（**Setter**注入）和接口注入**。构造注入是指通过构造函数来传入具体类的对象，设值注入是指通过Setter方法来传入具体类的对象，而接口注入是指通过在接口中声明的业务方法来传入具体类的对象。这些方法在定义时使用的是抽象类型，在运行时再传入具体类型的对象，由子类对象来覆盖父类对象。

### \*.5.面向对象设计原则:接口隔离原则

  接口隔离原则定义如下：

**接口隔离原则(Interface  Segregation Principle, ISP)：使用多个专门的接口，而不使用单一的总接口，即客户端不应该依赖那些它不需要的接口。**

      根据接口隔离原则，当一个接口太大时，我们需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。每一个接口应该承担一种相对独立的角色，不干不该干的事，该干的事都要干。这里的“接口”往往有两种不同的含义：一种是指一个类型所具有的方法特征的集合，仅仅是一种逻辑上的抽象；另外一种是指某种语言具体的“接口”定义，有严格的定义和结构，比如Java语言中的interface。对于这两种不同的含义，ISP的表达方式以及含义都有所不同：

      (1) 当把“接口”理解成一个类型所提供的所有方法特征的集合的时候，这就是一种逻辑上的概念，接口的划分将直接带来类型的划分。可以把接口理解成角色，一个接口只能代表一个角色，每个角色都有它特定的一个接口，此时，这个原则可以叫做“**角色隔离原则**”。

      (2) 如果把“接口”理解成狭义的特定语言的接口，那么ISP表达的意思是指**接口仅仅提供客户端需要的行为，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口**。在面向对象编程语言中，实现一个接口就需要实现该接口中定义的所有方法，因此大的总接口使用起来不一定很方便，为了使接口的职责单一，需要将大接口中的方法根据其职责不同分别放在不同的小接口中，以确保每个接口使用起来都较为方便，并都承担某一单一角色。接口应该尽量细化，同时接口中的方法应该尽量少，每个接口中只包含一个客户端（如子模块或业务逻辑类）所需的方法即可，这种机制也称为“**定制服务**”，即为不同的客户端提供宽窄不同的接口。

### \*.6.面向对象设计原则:合成复用原则

      合成复用原则又称为组合/聚合复用原则(Composition/Aggregate Reuse Principle, CARP)，其定义如下：

**合成复用原则(Composite Reuse Principle, CRP)：尽量使用对象组合，而不是继承来达到复用的目的。**

      合成复用原则就是在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用功能的目的。简言之：**复用时要尽量使用组合**/**聚合关系（关联关系），少用继承**。

      在面向对象设计中，可以通过两种方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承，但首先应该考虑使用组合/聚合，组合/聚合可以使系统更加灵活，降低类与类之间的耦合度，一个类的变化对其他类造成的影响相对较少；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

      通过继承来进行复用的主要问题在于继承复用会破坏系统的封装性，因为继承会将基类的实现细节暴露给子类，由于基类的内部细节通常对子类来说是可见的，所以这种复用又称“白箱”复用，如果基类发生改变，那么子类的实现也不得不发生改变；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；而且继承只能在有限的环境中使用（如类没有声明为不能被继承）。

### \*.7.面向对象设计原则:迪米特法则

      迪米特法则来自于1987年美国东北大学(Northeastern University)一个名为“Demeter”的研究项目。迪米特法则又称为最少知识原则(LeastKnowledge Principle, LKP)，其定义如下：

**迪米特法则(Law of  Demeter, LoD)：一个软件实体应当尽可能少地与其他实体发生相互作用。**

      如果一个系统符合迪米特法则，那么当其中某一个模块发生修改时，就会尽量少地影响其他模块，扩展会相对容易，这是对软件实体之间通信的限制，迪米特法则要求限制软件实体之间通信的宽度和深度。迪米特法则可降低系统的耦合度，使类与类之间保持松散的耦合关系。

      迪米特法则还有几种定义形式，包括**：不要和“陌生人”说话**、**只与你的直接朋友通信**等，在迪米特法则中，对于一个对象，其朋友包括以下几类：

(1) 当前对象本身(this)；

(2) 以参数形式传入到当前对象方法中的对象；

(3) 当前对象的成员对象；

(4) 如果当前对象的成员对象是一个集合，那么集合中的元素也都是朋友；

(5) 当前对象所创建的对象。

      任何一个对象，如果满足上面的条件之一，就是当前对象的“朋友”，否则就是“陌生人”。在应用迪米特法则时，一个对象只能与直接朋友发生交互，不要与“陌生人”发生直接交互，这样做可以降低系统的耦合度，一个对象的改变不会给太多其他对象带来影响。

      迪米特法则要求我们在设计系统时，应该尽量减少对象之间的交互，如果两个对象之间不必彼此直接通信，那么这两个对象就不应当发生任何直接的相互作用，如果其中的一个对象需要调用另一个对象的某一个方法的话，可以通过第三者转发这个调用。简言之，就是**通过引入一个合理的第三者来降低现有对象之间的耦合度**。

      在将迪米特法则运用到系统设计中时，要注意下面的几点：**在类的划分上，应当尽量创建松耦合的类，类之间的耦合度越低，就越有利于复用，一个处在松耦合中的类一旦被修改，不会对关联的类造成太大波及**；**在类的结构设计上，每一个类都应当尽量降低其成员变量和成员函数的访问权限**；**在类的设计上，只要有可能，一个类型应当设计成不变类**；**在对其他类的引用上，一个对象对其他对象的引用应当降到最低**。

这里讲几个常用的吧

|  |
| --- |
| 1 factory工厂模式  1 singleton单例模式  1 Prototype原型模式  2 Delegate委派模式  2 proxy代理模式    3 Template模板模式  3 strategy策略模式 |

# 创建型模式

**工厂模式,简单工厂模式,抽象工厂模式,单例模式,建造者模式,原型模式**

## 1. ****Factory****工厂方法模式+简单工厂模式+抽象工厂模式

### 1.1.简单工厂模式

工厂模式和抽象工厂模式主要解决接口选择问题。

工厂模式是最常用的一类创建型设计模式，通常我们所说的工厂模式是指**工厂方法模式**，它也是使用频率最高的工厂模式。

本章将要学习的**简单工厂模式**是工厂方法模式的“小弟”，它不属于GoF 23种设计模式，但在软件开发中应用也较为频繁，通常将它作为学习其他工厂模式的入门。

此外，工厂方法模式还有一位“大哥”——**抽象工厂模式**。这三种工厂模式各具特色，难度也逐个加大，在软件开发中它们都得到了广泛的应用，成为面向对象软件中常用的创建对象的工具。

   面对一个如此巨大、职责如此重，且与客户端代码耦合度非常高的类，我们应该怎么办？本章将要介绍的简单工厂模式**将在一定程度上解决上述问题**。

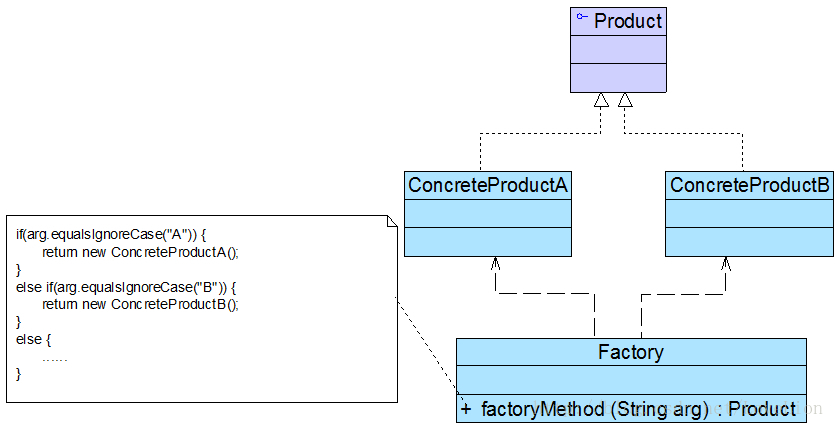
**为什么要引入工厂类，大家可参见：**[创建对象与使用对象——谈谈工厂的作用](http://blog.csdn.net/lovelion/article/details/7523392)**。**

#### 简单工厂模式概述

简单工厂模式定义如下：

**简单工厂模式(Simple Factory Pattern)：定义一个工厂类，它可以根据参数的不同返回不同类的实例，被创建的实例通常都具有共同的父类。因为在简单工厂模式中用于创建实例的方法是静态(static)方法，因此简单工厂模式又被称为静态工厂方法(Static Factory Method)模式，它属于类创建型模式。**

简单工厂模式的要点在于：**当你需要什么，只需要传入一个正确的参数，就可以获取你所需要的对象，而无须知道其创建细节。**简单工厂模式结构比较简单，其核心是工厂类的设计，其结构如图1所示：



       ● Factory（工厂角色）：工厂角色即工厂类，它是简单工厂模式的核心，负责实现创建所有产品实例的内部逻辑；工厂类可以被外界直接调用，创建所需的产品对象；在工厂类中提供了静态的工厂方法factoryMethod()，它的返回类型为抽象产品类型Product。

       ● Product（抽象产品角色）：它是工厂类所创建的所有对象的父类，封装了各种产品对象的公有方法，它的引入将提高系统的灵活性，使得在工厂类中只需定义一个通用的工厂方法，因为所有创建的具体产品对象都是其子类对象。

       ● ConcreteProduct（具体产品角色）：它是简单工厂模式的创建目标，所有被创建的对象都充当这个角色的某个具体类的实例。每一个具体产品角色都继承了抽象产品角色，需要实现在抽象产品中声明的抽象方法。

在简单工厂模式中，客户端通过工厂类来创建一个产品类的实例，而无须直接使用new关键字来创建对象，它是工厂模式家族中最简单的一员。

**public abstract class Product** {  
 **public void** methodSame() {  
 **System**.***out***.println("公共方法的实现");  
 }  
 **public abstract void** methodDiff();  
}

**public class ConcreteProductA extends** Product {  
 **public void** methodDiff() {  
 **System**.***out***.println("产品A");  
 }  
}

**public class ConcreteProductB extends** Product {  
 **public void** methodDiff() {  
 **System**.***out***.println("产品B");  
 }  
}

**public class Factory** {  
 **public static** Product getProduct(**String** arg){  
 Product product = **null**;  
 **if**(arg.equalsIgnoreCase("A")){  
 product = **new** ConcreteProductA();  
 }**else if**(arg.equalsIgnoreCase("B")){  
 product = **new** ConcreteProductB();  
 }  
 **return** product;  
 }  
}

**public class Client** {  
 **public static void** main(**String**[] args) {  
 Product product ;  
 product = Factory.getProduct("A");  
 product.methodSame();  
 product.methodDiff();  
 }  
}

#### 简单工厂模式总结

简单工厂模式提供了专门的工厂类用于创建对象，将对象的创建和对象的使用分离开，它作为一种最简单的工厂模式在软件开发中得到了较为广泛的应用。

1. 主要优点

       简单工厂模式的主要优点如下：

       (1) 工厂类包含必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户端可以免除直接创建产品对象的职责，而仅仅“消费”产品，简单工厂模式实现了对象创建和使用的分离。

       (2) 客户端无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以在一定程度减少使用者的记忆量。

       (3) 通过引入配置文件，可以在不修改任何客户端代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性。

2. 主要缺点

       简单工厂模式的主要缺点如下：

       (1) 由于工厂类集中了所有产品的创建逻辑，职责过重，一旦不能正常工作，整个系统都要受到影响。

       (2) 使用简单工厂模式势必会增加系统中类的个数（引入了新的工厂类），增加了系统的复杂度和理解难度。

       (3) 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护。

       (4) 简单工厂模式由于使用了静态工厂方法，造成工厂角色无法形成基于继承的等级结构。

3. 适用场景

       在以下情况下可以考虑使用简单工厂模式：

       (1) 工厂类负责创建的对象比较少，由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂。

       (2) 客户端只知道传入工厂类的参数，对于如何创建对象并不关心。

### 1.2.工厂方法模式

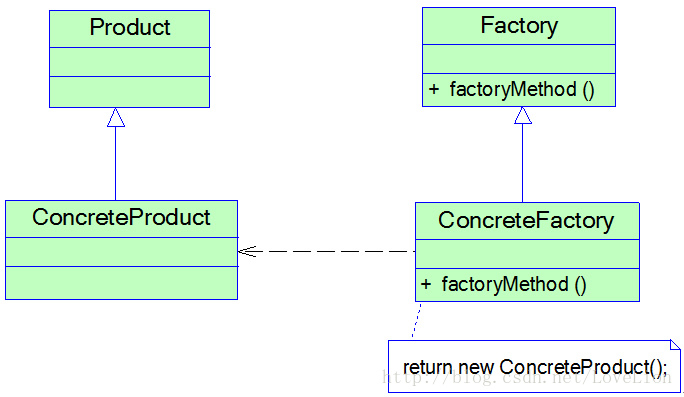
简单工厂模式虽然简单，但存在一个很严重的问题。**当系统中需要引入新产品时，由于静态工厂方法通过所传入参数的不同来创建不同的产品，这必定要修改工厂类的源代码，将违背“开闭原则”，如何实现增加新产品而不影响已有代码？**工厂方法模式应运而生，本文将介绍第二种工厂模式——工厂方法模式。

#### ****工厂方法模式概述****

在工厂方法模式中，我们**不再提供一个统一的工厂类来创建所有的产品对象，而是针对不同的产品提供不同的工厂，系统提供一个与产品等级结构对应的工厂等级结构**。工厂方法模式定义如下：

**工厂方法模式(Factory Method Pattern)：定义一个用于创建对象的接口，让子类决定将哪一个类实例化。工厂方法模式让一个类的实例化延迟到其子类。工厂方法模式又简称为工厂模式(Factory Pattern)，又可称作虚拟构造器模式(Virtual Constructor Pattern)或多态工厂模式(Polymorphic Factory Pattern)。工厂方法模式是一种类创建型模式。**

工厂方法模式提供一个抽象工厂接口来声明抽象工厂方法，而由其子类来具体实现工厂方法，创建具体的产品对象。工厂方法模式结构如图2所示：



 ● Product（抽象产品）：它是定义产品的接口，是工厂方法模式所创建对象的超类型，也就是产品对象的公共父类。

**●** ConcreteProduct**（具体产品）：**它实现了抽象产品接口，某种类型的具体产品由专门的具体工厂创建，具体工厂和具体产品之间一一对应。

**●** Factory**（抽象工厂）：**在抽象工厂类中，声明了工厂方法(Factory Method)，用于返回一个产品。抽象工厂是工厂方法模式的核心，所有创建对象的工厂类都必须实现该接口。

**●** ConcreteFactory**（具体工厂）：**它是抽象工厂类的子类，实现了抽象工厂中定义的工厂方法，并可由客户端调用，返回一个具体产品类的实例。

**public interface Logger** {  
 **public void** writeLog();  
}

**public class DatabaseLogger implements** Logger{  
 **public void** writeLog() {  
 **System**.***out***.println("数据库日志！");  
 }  
}

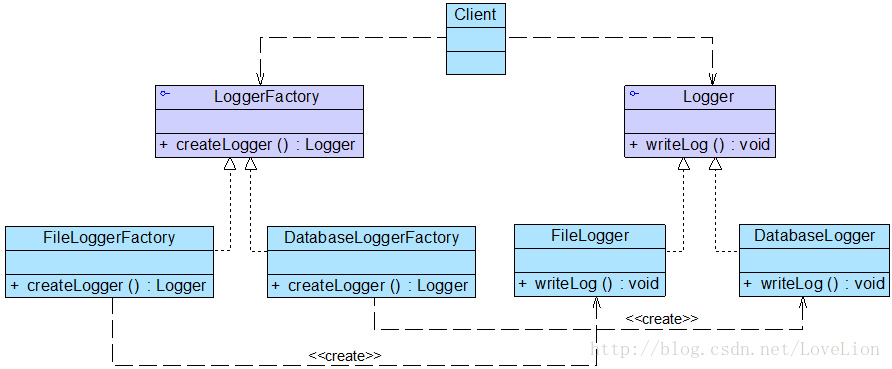
**public class FileLogger implements** Logger{  
 **public void** writeLog() {  
 **System**.***out***.println("文件日志！");  
 }  
}

**public interface LoggerFactory** {  
 **public** Logger createLogger();  
}

**public class DatabaseLoggerFactory implements** LoggerFactory {  
 **public** Logger createLogger() {  
 Logger logger = **new** DatabaseLogger();  
 **return** logger;  
 }  
}

**public class FileLoggerFactory implements** LoggerFactory {  
 **public** Logger createLogger() {  
 Logger logger = **new** FileLogger();  
 **return** logger;  
 }  
}

**public class Client** {  
 **public static void** main(**String**[] args) {  
 LoggerFactory factory;  
 Logger logger;  
   
 factory = **new** FileLoggerFactory();  
 logger = factory.createLogger();  
 logger.writeLog();  
 }  
}



#### 工厂方法模式总结

      工厂方法模式是简单工厂模式的延伸，它继承了简单工厂模式的优点，同时还弥补了简单工厂模式的不足。工厂方法模式是使用频率最高的设计模式之一，是很多开源框架和API类库的核心模式。

1. 主要优点

    工厂方法模式的主要优点如下：

       (1) 在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节，用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名。

       (2) 基于工厂角色和产品角色的多态性设计是工厂方法模式的关键。它能够让工厂可以自主确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部。工厂方法模式之所以又被称为多态工厂模式，就正是因为所有的具体工厂类都具有同一抽象父类。

       (3) 使用工厂方法模式的另一个优点是在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了，这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”。

2. 主要缺点

     工厂方法模式的主要缺点如下：

      (1) 在添加新产品时，需要编写新的具体产品类，而且还要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度，有更多的类需要编译和运行，会给系统带来一些额外的开销。

      (2) 由于考虑到系统的可扩展性，需要引入抽象层，在客户端代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度，且在实现时可能需要用到DOM、反射等技术，增加了系统的实现难度。

3. 适用场景

       在以下情况下可以考虑使用工厂方法模式：

       (1) 客户端不知道它所需要的对象的类。在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体工厂类创建，可将具体工厂类的类名存储在配置文件或数据库中。

       (2) 抽象工厂类通过其子类来指定创建哪个对象。在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类来确定具体要创建的对象，利用面向对象的多态性和里氏代换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

### 1.3.抽象工厂模式

工厂方法模式通过引入工厂等级结构，解决了简单工厂模式中工厂类职责太重的问题，但由于工厂方法模式中的每个工厂只生产一类产品，可能会导致系统中存在大量的工厂类，势必会增加系统的开销。此时，我们可以考虑**将一些相关的产品组成一个“产品族”，由同一个工厂来统一生产**，这就是我们本文将要学习的抽象工厂模式的基本思想。

#### 抽象工厂模式概述

抽象工厂模式为创建一组对象提供了一种解决方案。与工厂方法模式相比，抽象工厂模式中的具体工厂不只是创建一种产品，它负责创建一族产品。抽象工厂模式定义如下：

**抽象工厂模式(Abstract Factory Pattern)：提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类。抽象工厂模式又称为Kit模式，它是一种对象创建型模式。**

**public interface Button** {  
 **public void** display();  
}

**public class SpringButton implements** Button {  
 **public void** display() {  
 **System**.***out***.println("显示spring色按钮！");  
 }  
}

**public class SummerButton implements** Button {  
 **public void** display() {  
 **System**.***out***.println("显示summer色按钮！");  
 }  
}

**public interface ComboBox** {  
 **public void** display();  
}

**public class SpringComboBox implements** ComboBox {  
 @Override  
 **public void** display() {  
 **System**.***out***.println("显示spring色组合框！");  
 }  
}

**public class SummerComboBox implements** ComboBox {  
 @Override  
 **public void** display() {  
 **System**.***out***.println("显示summer色组合框！");  
 }  
}

**public interface SkinFactory** {  
 **public** Button createButton();  
  
 **public** TextField createTextField();  
  
 **public** ComboBox createComboBox();  
}

**public interface TextField** {  
 **public void** display();  
}

**public class SpringTextField implements** TextField {  
  
 **public void** display() {  
 **System**.***out***.println("显示spring色文本框！");  
 }  
}

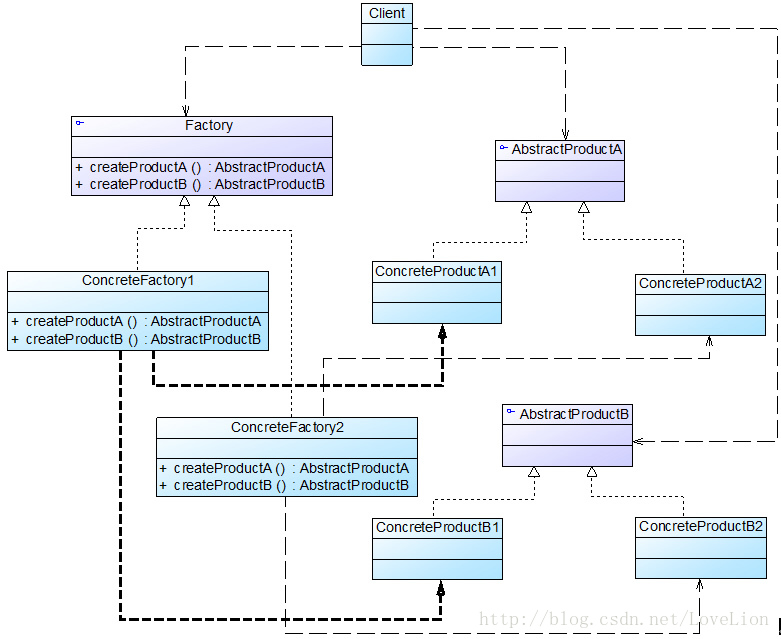
**public class SummerTextField implements** TextField {  
  
 @Override  
 **public void** display() {  
 **System**.***out***.println("显示summber色文本框！");  
 }  
  
}

**public class SpringSkinFactory implements** SkinFactory {  
 @Override  
 **public** Button createButton() {  
 **return new** SpringButton();  
 }

@Override  
 **public** TextField createTextField() {  
 **return new** SpringTextField();  
 }  
  
 @Override  
 **public** ComboBox createComboBox() {  
 **return new** SpringComboBox();  
 }  
}

**public class SummerSkinFactory implements** SkinFactory {  
 @Override  
 **public** Button createButton() {  
 **return new** SummerButton();  
 }  
  
 @Override  
 **public** TextField createTextField() {  
 **return new** SummerTextField();  
 }  
  
 @Override  
 **public** ComboBox createComboBox() {  
 **return new** SummerComboBox();  
 }  
}

**public class Client** {  
 **public static void** main(**String**[] args) {  
 SkinFactory factory;  
 Button bt;  
 TextField tf;  
 ComboBox cb;  
  
 factory = **new** SpringSkinFactory();  
   
 bt = factory.createButton();  
 tf = factory.createTextField();  
 cb = factory.createComboBox();  
   
 bt.display();  
 tf.display();  
 cb.display();  
 }  
}



       ● AbstractFactory（抽象工厂）：它声明了一组用于创建一族产品的方法，每一个方法对应一种产品。

       ● ConcreteFactory（具体工厂）：它实现了在抽象工厂中声明的创建产品的方法，生成一组具体产品，这些产品构成了一个产品族，每一个产品都位于某个产品等级结构中。

       ● AbstractProduct（抽象产品）：它为每种产品声明接口，在抽象产品中声明了产品所具有的业务方法。

       ● ConcreteProduct（具体产品）：它定义具体工厂生产的具体产品对象，实现抽象产品接口中声明的业务方法。

#### “开闭原则”的倾斜性

       Sunny公司使用抽象工厂模式设计了界面皮肤库，该皮肤库可以较为方便地增加新的皮肤，但是现在遇到一个非常严重的问题：由于设计时考虑不全面，忘记为单选按钮(RadioButton)提供不同皮肤的风格化显示，导致无论选择哪种皮肤，单选按钮都显得那么“格格不入”。Sunny公司的设计人员决定向系统中增加单选按钮，但是发现原有系统居然不能够在符合“开闭原则”的前提下增加新的组件，原因是抽象工厂SkinFactory中根本没有提供创建单选按钮的方法，如果需要增加单选按钮，首先需要修改抽象工厂接口SkinFactory，在其中新增声明创建单选按钮的方法，然后逐个修改具体工厂类，增加相应方法以实现在不同的皮肤中创建单选按钮，此外还需要修改客户端，否则单选按钮无法应用于现有系统。

       怎么办？答案是抽象工厂模式无法解决该问题，这也是抽象工厂模式最大的缺点。**在抽象工厂模式中，增加新的产品族很方便，但是增加新的产品等级结构很麻烦**，抽象工厂模式的这种性质称为**“开闭原则”的倾斜性**。“开闭原则”要求系统对扩展开放，对修改封闭，通过扩展达到增强其功能的目的，对于涉及到多个产品族与多个产品等级结构的系统，其功能增强包括两方面：

**(1) 增加产品族**：对于增加新的产品族，抽象工厂模式很好地支持了“开闭原则”，只需要增加具体产品并对应增加一个新的具体工厂，对已有代码无须做任何修改。

**(2) 增加新的产品等级结构**：对于增加新的产品等级结构，需要修改所有的工厂角色，包括抽象工厂类，在所有的工厂类中都需要增加生产新产品的方法，违背了“开闭原则”。

       正因为抽象工厂模式存在“开闭原则”的倾斜性，它以一种倾斜的方式来满足“开闭原则”，为增加新产品族提供方便，但不能为增加新产品结构提供这样的方便，因此要求设计人员在设计之初就能够全面考虑，不会在设计完成之后向系统中增加新的产品等级结构，也不会删除已有的产品等级结构，否则将会导致系统出现较大的修改，为后续维护工作带来诸多麻烦。

#### 抽象工厂模式总结

       抽象工厂模式是工厂方法模式的进一步延伸，由于它提供了功能更为强大的工厂类并且具备较好的可扩展性，在软件开发中得以广泛应用，尤其是在一些框架和API类库的设计中，例如在Java语言的AWT（抽象窗口工具包）中就使用了抽象工厂模式，它使用抽象工厂模式来实现在不同的操作系统中应用程序呈现与所在操作系统一致的外观界面。抽象工厂模式也是在软件开发中最常用的设计模式之一。

1. 主要优点

       抽象工厂模式的主要优点如下：

       (1) 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建。由于这种隔离，更换一个具体工厂就变得相对容易，所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变整个软件系统的行为。

       (2) 当一个产品族中的多个对象被设计成一起工作时，它能够保证客户端始终只使用同一个产品族中的对象。

       (3) 增加新的产品族很方便，无须修改已有系统，符合“开闭原则”。

2. 主要缺点

       抽象工厂模式的主要缺点如下：

       增加新的产品等级结构麻烦，需要对原有系统进行较大的修改，甚至需要修改抽象层代码，这显然会带来较大的不便，违背了“开闭原则”。

3. 适用场景

       在以下情况下可以考虑使用抽象工厂模式：

       (1) 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是很重要的，用户无须关心对象的创建过程，将对象的创建和使用解耦。

       (2) 系统中有多于一个的产品族，而每次只使用其中某一产品族。可以通过配置文件等方式来使得用户可以动态改变产品族，也可以很方便地增加新的产品族。

       (3) 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来。同一个产品族中的产品可以是没有任何关系的对象，但是它们都具有一些共同的约束，如同一操作系统下的按钮和文本框，按钮与文本框之间没有直接关系，但它们都是属于某一操作系统的，此时具有一个共同的约束条件：操作系统的类型。

       (4) 产品等级结构稳定，设计完成之后，不会向系统中增加新的产品等级结构或者删除已有的产品等级结构。

## 2. ****Singleton****单例模式

**工厂模式,简单工厂模式,抽象工厂模式,单例模式,建造者模式,原型模式**

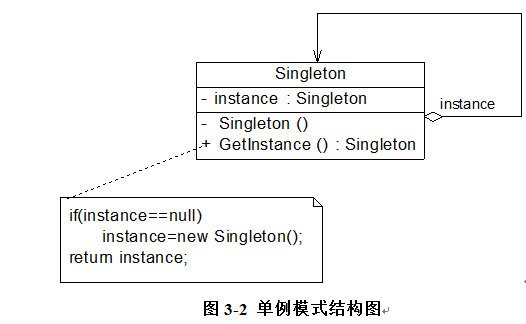
### 2.1.单例模式的动机

对于一个软件系统的某些类而言，我们无须创建多个实例。

回到实际开发中，我们也经常遇到类似的情况，为了节约系统资源，有时需要确保系统中某个类只有唯一一个实例，当这个唯一实例创建成功之后，我们无法再创建一个同类型的其他对象，所有的操作都只能基于这个唯一实例。为了确保对象的唯一性，我们可以通过单例模式来实现，这就是单例模式的动机所在。

### 2.2 单例模式概述

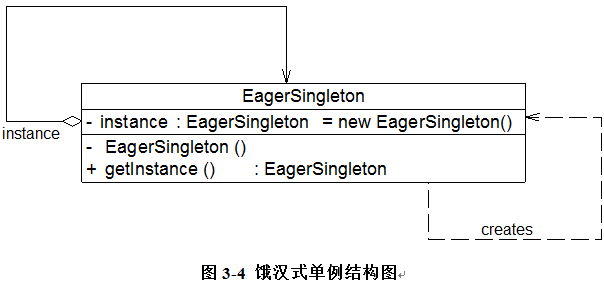
**单例模式(Singleton Pattern)：确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例，这个类称为单例类，它提供全局访问的方法。单例模式是一种对象创建型模式。**



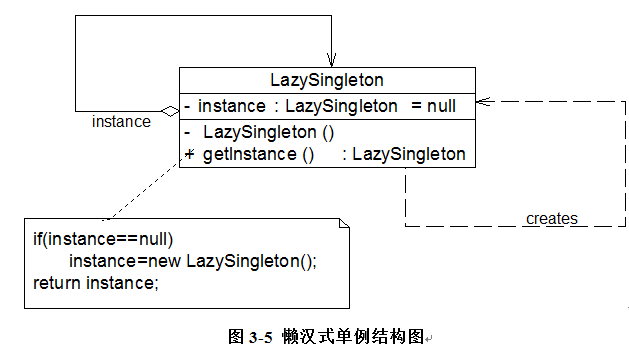
 ● Singleton（单例）：在单例类的内部实现只生成一个实例，同时它提供一个静态的getInstance()工厂方法，让客户可以访问它的唯一实例；为了防止在外部对其实例化，将其构造函数设计为私有；在单例类内部定义了一个Singleton类型的静态对象，作为外部共享的唯一实例。

### ****2.3.饿汉式单例类****

饿汉式单例类是实现起来最简单的单例类，饿汉式单例类结构图如图3-4所示：



### ****2.4.懒汉式单例类****



### 2.5.饿汉式单例类与懒汉式单例类比较

      饿汉式单例类在类被加载时就将自己实例化，它的优点在于无须考虑多线程访问问题，可以确保实例的唯一性；从调用速度和反应时间角度来讲，由于单例对象一开始就得以创建，因此要优于懒汉式单例。但是无论系统在运行时是否需要使用该单例对象，由于在类加载时该对象就需要创建，因此从资源利用效率角度来讲，饿汉式单例不及懒汉式单例，而且在系统加载时由于需要创建饿汉式单例对象，加载时间可能会比较长。

      懒汉式单例类在第一次使用时创建，无须一直占用系统资源，实现了延迟加载，但是必须处理好多个线程同时访问的问题，特别是当单例类作为资源控制器，在实例化时必然涉及资源初始化，而资源初始化很有可能耗费大量时间，这意味着出现多线程同时首次引用此类的机率变得较大，需要通过双重检查锁定等机制进行控制，这将导致系统性能受到一定影响。

2.6.单例模式总结

    单例模式作为一种目标明确、结构简单、理解容易的设计模式，在软件开发中使用频率相当高，在很多应用软件和框架中都得以广泛应用。

**1.主要优点**

       单例模式的主要优点如下：

       (1) 单例模式提供了对唯一实例的受控访问。因为单例类封装了它的唯一实例，所以它可以严格控制客户怎样以及何时访问它。

       (2) 由于在系统内存中只存在一个对象，因此可以节约系统资源，对于一些需要频繁创建和销毁的对象单例模式无疑可以提高系统的性能。

       (3) 允许可变数目的实例。基于单例模式我们可以进行扩展，使用与单例控制相似的方法来获得指定个数的对象实例，既节省系统资源，又解决了单例单例对象共享过多有损性能的问题。

**2.主要缺点**

       单例模式的主要缺点如下：

       (1) 由于单例模式中没有抽象层，因此单例类的扩展有很大的困难。

       (2) 单例类的职责过重，在一定程度上违背了“单一职责原则”。因为单例类既充当了工厂角色，提供了工厂方法，同时又充当了产品角色，包含一些业务方法，将产品的创建和产品的本身的功能融合到一起。

       (3) 现在很多面向对象语言(如Java、C#)的运行环境都提供了自动垃圾回收的技术，因此，如果实例化的共享对象长时间不被利用，系统会认为它是垃圾，会自动销毁并回收资源，下次利用时又将重新实例化，这将导致共享的单例对象状态的丢失。

**3.适用场景**

       在以下情况下可以考虑使用单例模式：

       (1) 系统只需要一个实例对象，如系统要求提供一个唯一的序列号生成器或资源管理器，或者需要考虑资源消耗太大而只允许创建一个对象。

       (2) 客户调用类的单个实例只允许使用一个公共访问点，除了该公共访问点，不能通过其他途径访问该实例。

## \*深入浅出单实例SINGLETON设计模式

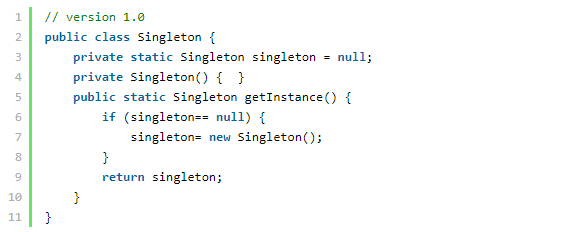
单实例Singleton设计模式可能是被讨论和使用的最广泛的一个设计模式了，这可能也是面试中问得最多的一个设计模式了。

这个设计模式主要目的是想在整个系统中只能出现一个类的实例。这样做当然是有必然的，比如你的软件的全局配置信息，或者是一个Factory，或是一个主控类，等等。

你希望这个类在整个系统中只能出现一个实例。当然，作为一个技术负责人的你，你当然有权利通过使用非技术的手段来达到你的目的。比如：你在团队内部明文规定，“XX类只能有一个全局实例，如果某人使用两次以上，那么该人将被处于2000元的罚款！”（呵呵），你当然有权这么做。但是如果你的设计的是东西是一个类库，或是一个需要提供给用户使用的API，恐怕你的这项规定将会失效。因为，你无权要求别人会那么做。所以，这就是为什么，我们希望通过使用技术的手段来达成这样一个目的的原因。  
 本文会带着你深入整个Singleton的世界，当然，我会放弃使用C++语言而改用Java语言，因为使用Java这个语言可能更容易让我说明一些事情。

### \*.1.Singleton的教学版本

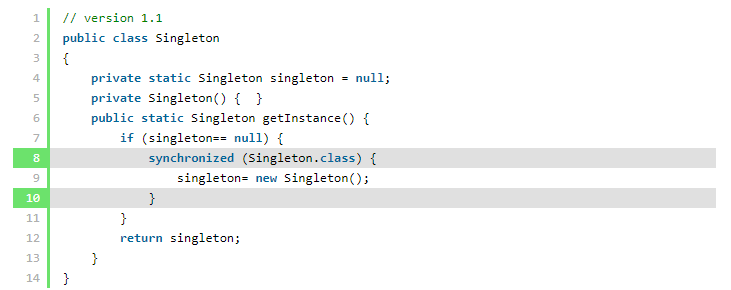
这里，我将直接给出一个Singleton的简单实现，因为我相信你已经有这方面的一些基础了。我们姑且把这个版本叫做1.0版



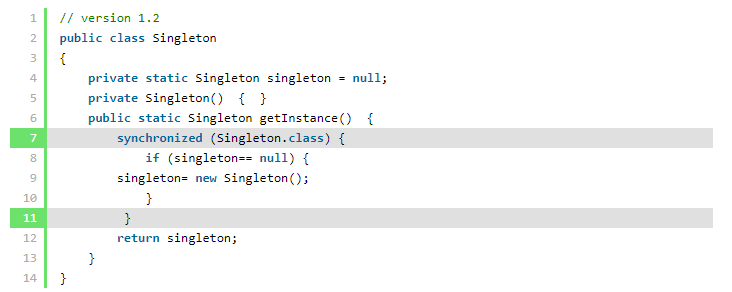
在上面的实例中，我想说明下面几个Singleton的特点：（下面这些东西可能是尽人皆知的，没有什么新鲜的）  
1. 私有（private）的构造函数，表明这个类是不可能形成实例了。这主要是怕这个类会有多个实例。  
2. 即然这个类是不可能形成实例，那么，我们需要一个静态的方式让其形成实例：getInstance()。注意这个方法是在new自己，因为其可以访问私有的构造函数，所以他是可以保证实例被创建出来的。  
3. 在getInstance()中，先做判断是否已形成实例，如果已形成则直接返回，否则创建实例。  
4. 所形成的实例保存在自己类中的私有成员中。  
5. 我们取实例时，只需要使用Singleton.getInstance()就行了。  
当然，如果你觉得知道了上面这些事情后就学成了，那得给你当头棒喝一下了，事情远远没有那么简单。

### \*.2.Singleton的实际版本

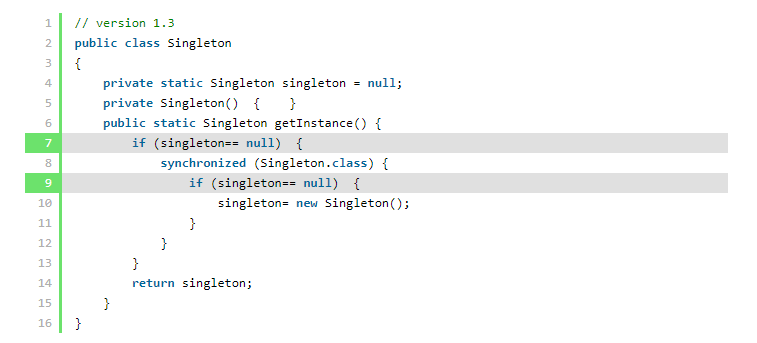
上面的这个程序存在比较严重的问题，因为是全局性的实例，所以，在多线程情况下，所有的全局共享的东西都会变得非常的危险，这个也一样，在多线程情况下，如果多个线程同时调用getInstance()的话，那么，可能会有多个进程同时通过 (singleton== null)的条件检查，于是，多个实例就创建出来，并且很可能造成内存泄露问题。嗯，熟悉多线程的你一定会说——“我们需要线程互斥或同步”，没错，我们需要这个事情，于是我们的Singleton升级成1.1版，如下所示：



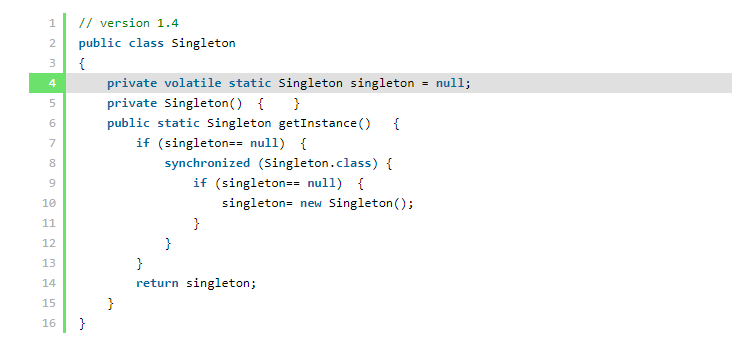
嗯，使用了Java的synchronized方法，看起来不错哦。应该没有问题了吧？！错！这还是有问题！为什么呢？前面已经说过，如果有多个线程同时通过(singleton== null)的条件检查（因为他们并行运行），虽然我们的synchronized方法会帮助我们同步所有的线程，让我们并行线程变成串行的一个一个去new，那不还是一样的吗？同样会出现很多实例。嗯，确实如此！看来，还得把那个判断(singleton== null)条件也同步起来。于是，我们的Singleton再次升级成1.2版本，如下所示：



不错不错，看似很不错了。在多线程下应该没有什么问题了，不是吗？的确是这样的，1.2版的Singleton在多线程下的确没有问题了，因为我们同步了所有的线程。只不过嘛……，什么？！还不行？！是的，还是有点小问题，我们本来只是想让new这个操作并行就可以了，现在，只要是进入getInstance()的线程都得同步啊，注意，创建对象的动作只有一次，后面的动作全是读取那个成员变量，这些读取的动作不需要线程同步啊。这样的作法感觉非常极端啊，为了一个初始化的创建动作，居然让我们达上了所有的读操作，严重影响后续的性能啊！  
还得改！嗯，看来，在线程同步前还得加一个(singleton== null)的条件判断，如果对象已经创建了，那么就不需要线程的同步了。OK，下面是1.3版的Singleton。



感觉代码开始变得有点罗嗦和复杂了，不过，这可能是最不错的一个版本了，这个版本又叫“双重检查”Double-Check。下面是说明：  
1. 第一个条件是说，如果实例创建了，那就不需要同步了，直接返回就好了。  
2. 不然，我们就开始同步线程。  
3. 第二个条件是说，如果被同步的线程中，有一个线程创建了对象，那么别的线程就不用再创建了。  
相当不错啊，干得非常漂亮！请大家为我们的1.3版起立鼓掌！  
但是，如果你认为这个版本大攻告成，你就错了。  
主要在于singleton = new Singleton()这句，这并非是一个原子操作，事实上在 JVM 中这句话大概做了下面 3 件事情。  
1. 给 singleton 分配内存  
2. 调用 Singleton 的构造函数来初始化成员变量，形成实例  
3. 将singleton对象指向分配的内存空间（执行完这步 singleton才是非 null 了）  
但是在 JVM 的即时编译器中存在指令重排序的优化。也就是说上面的第二步和第三步的顺序是不能保证的，最终的执行顺序可能是 1-2-3 也可能是 1-3-2。如果是后者，则在 3 执行完毕、2 未执行之前，被线程二抢占了，这时 instance 已经是非 null 了（但却没有初始化），所以线程二会直接返回 instance，然后使用，然后顺理成章地报错。  
对此，我们只需要把singleton声明成 volatile 就可以了。下面是1.4版：

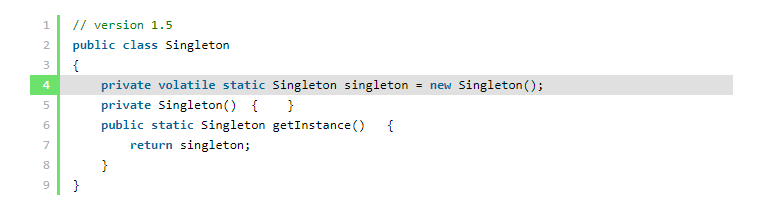


### \*.3.使用 volatile 有两个功用：

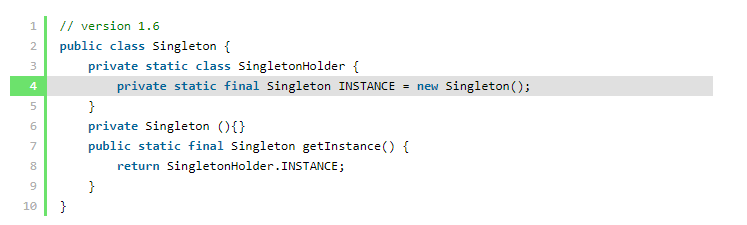
1）这个变量不会在多个线程中存在复本，直接从内存读取。  
2）这个关键字会禁止指令重排序优化。也就是说，在 volatile 变量的赋值操作后面会有一个内存屏障（生成的汇编代码上），读操作不会被重排序到内存屏障之前。  
但是，这个事情仅在Java 1.5版后有用，1.5版之前用这个变量也有问题，因为老版本的Java的内存模型是有缺陷的。

### \*.4.Singleton 的简化版本

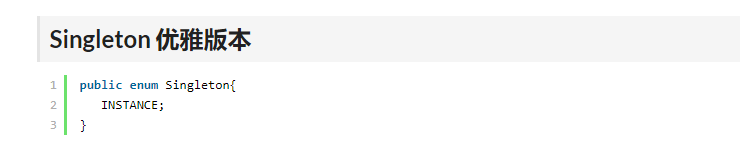
上面的玩法实在是太复杂了，一点也不优雅，下面是一种更为优雅的方式：  
这种方法非常简单，因为单例的实例被声明成 static 和 final 变量了，在第一次加载类到内存中时就会初始化，所以创建实例本身是线程安全的。



但是，这种玩法的最大问题是——当这个类被加载的时候，new Singleton() 这句话就会被执行，就算是getInstance()没有被调用，类也被初始化了。  
于是，这个可能会与我们想要的行为不一样，比如，我的类的构造函数中，有一些事可能需要依赖于别的类干的一些事（比如某个配置文件，或是某个被其它类创建的资源），我们希望他能在我第一次getInstance()时才被真正的创建。这样，我们可以控制真正的类创建的时刻，而不是把类的创建委托给了类装载器。  
好吧，我们还得绕一下：  
下面的这个1.6版是老版《Effective Java》中推荐的方式。



上面这种方式，仍然使用JVM本身机制保证了线程安全问题；由于 SingletonHolder 是私有的，除了 getInstance() 之外没有办法访问它，因此它只有在getInstance()被调用时才会真正创建；同时读取实例的时候不会进行同步，没有性能缺陷；也不依赖 JDK 版本。



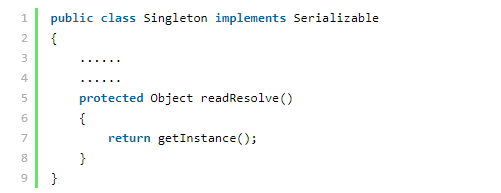
居然用枚举！！看上去好牛逼，通过EasySingleton.INSTANCE来访问，这比调用getInstance()方法简单多了。  
默认枚举实例的创建是线程安全的，所以不需要担心线程安全的问题。但是在枚举中的其他任何方法的线程安全由程序员自己负责。还有防止上面的通过反射机制调用私用构造器。  
这个版本基本上消除了绝大多数的问题。代码也非常简单，实在无法不用。这也是新版的《Effective Java》中推荐的模式。

### \*.5.Singleton的其它问题

怎么？还有问题？！当然还有，请记住下面这条规则——“无论你的代码写得有多好，其只能在特定的范围内工作，超出这个范围就要出Bug了”，这是“陈式第一定理”，呵呵。你能想一想还有什么情况会让这个我们上面的代码出问题吗？  
在C++下，我不是很好举例，但是在Java的环境下，嘿嘿，还是让我们来看看下面的一些反例和一些别的事情的讨论（当然，有些反例可能属于钻牛角尖，可能有点学院派，不过也不排除其实际可能性，就算是提个醒吧）：

其一、Class Loader。不知道你对Java的Class Loader熟悉吗？“类装载器”？！C++可没有这个东西啊。这是Java动态性的核心。顾名思义，类装载器是用来把类(class)装载进JVM的。JVM规范定义了两种类型的类装载器：启动内装载器(bootstrap)和用户自定义装载器(user-defined class loader)。 在一个JVM中可能存在多个ClassLoader，每个ClassLoader拥有自己的NameSpace。一个ClassLoader只能拥有一个class对象类型的实例，但是不同的ClassLoader可能拥有相同的class对象实例，这时可能产生致命的问题。如ClassLoaderA，装载了类A的类型实例A1，而ClassLoaderB，也装载了类A的对象实例A2。逻辑上讲A1=A2，但是由于A1和A2来自于不同的ClassLoader，它们实际上是完全不同的，如果A中定义了一个静态变量c，则c在不同的ClassLoader中的值是不同的。  
于是，如果咱们的Singleton 1.3版本如果面对着多个Class Loader会怎么样？呵呵，多个实例同样会被多个Class Loader创建出来，当然，这个有点牵强，不过他确实存在。难道我们还要整出个1.4版吗？可是，我们怎么可能在我的Singleton类中操作Class Loader啊？是的，你根本不可能。在这种情况下，你能做的只有是——“保证多个Class Loader不会装载同一个Singleton”。

其二、序例化。如果我们的这个Singleton类是一个关于我们程序配置信息的类。我们需要它有序列化的功能，那么，当反序列化的时候，我们将无法控制别人不多次反序列化。不过，我们可以利用一下Serializable接口的readResolve()方法，比如：

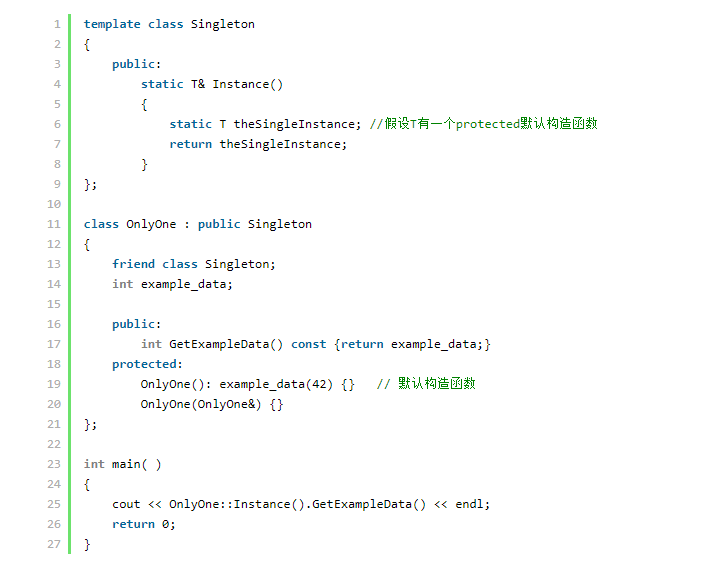


其三、多个Java虚拟机。如果我们的程序运行在多个Java的虚拟机中。什么？多个虚拟机？这是一种什么样的情况啊。嗯，这种情况是有点极端，不过还是可能出现，比如EJB或RMI之流的东西。要在这种环境下避免多实例，看来只能通过良好的设计或非技术来解决了。

其四，volatile变量。关于volatile这个关键字所声明的变量可以被看作是一种 “程度较轻的同步synchronized”；与 synchronized 块相比，volatile 变量所需的编码较少，并且运行时开销也较少，但是它所能实现的功能也仅是synchronized的一部分。当然，如前面所述，我们需要的Singleton只是在创建的时候线程同步，而后面的读取则不需要同步。所以，volatile变量并不能帮助我们即能解决问题，又有好的性能。而且，这种变量只能在JDK 1.5+版后才能使用。

其五、关于继承。是的，继承于Singleton后的子类也有可能造成多实例的问题。不过，因为我们早把Singleton的构造函数声明成了私有的，所以也就杜绝了继承这种事情。

其六，关于代码重用。也话我们的系统中有很多个类需要用到这个模式，如果我们在每一个类都中有这样的代码，那么就显得有点傻了。那么，我们是否可以使用一种方法，把这具模式抽象出去？在C++下这是很容易的，因为有模板和友元，还支持栈上分配内存，所以比较容易一些（程序如下所示），Java下可能比较复杂一些，聪明的你知道怎么做吗？



## 3. ****Builder****建造者模式

**工厂模式,简单工厂模式,抽象工厂模式,单例模式,建造者模式,原型模式**

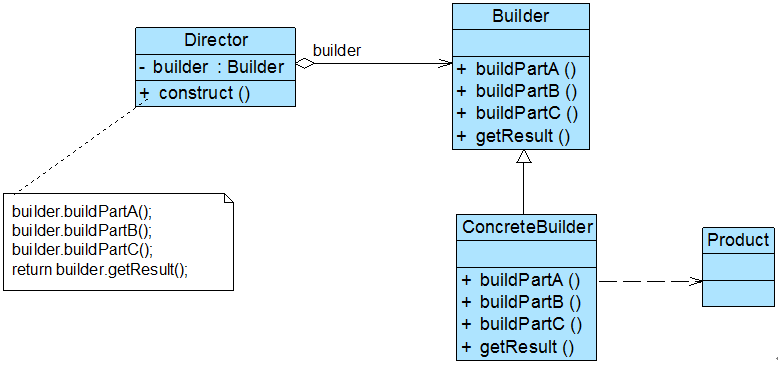
没有人买车会只买一个轮胎或者方向盘，大家买的都是一辆包含轮胎、方向盘和发动机等多个部件的完整汽车。如何将这些部件组装成一辆完整的汽车并返回给用户，这是建造者模式需要解决的问题。建造者模式又称为生成器模式，它是一种较为复杂、使用频率也相对较低的创建型模式。建造者模式为客户端返回的不是一个简单的产品，而是一个由多个部件组成的复杂产品。

### 3.1.建造者模式概述

建造者模式是较为复杂的创建型模式，它将客户端与包含多个组成部分（或部件）的复杂对象的创建过程分离，客户端无须知道复杂对象的内部组成部分与装配方式，只需要知道所需建造者的类型即可。它关注如何一步一步创建一个的复杂对象，不同的具体建造者定义了不同的创建过程，且具体建造者相互独立，增加新的建造者非常方便，无须修改已有代码，系统具有较好的扩展性。

建造者模式定义如下：

**建造者模式(Builder Pattern)：将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示。建造者模式是一种对象创建型模式。**



● Builder（抽象建造者）：它为创建一个产品Product对象的各个部件指定抽象接口，在该接口中一般声明两类方法，一类方法是buildPartX()，它们用于创建复杂对象的各个部件；另一类方法是getResult()，它们用于返回复杂对象。Builder既可以是抽象类，也可以是接口。

●ConcreteBuilder（具体建造者）：它实现了Builder接口，实现各个部件的具体构造和装配方法，定义并明确它所创建的复杂对象，也可以提供一个方法返回创建好的复杂产品对象。

●Product（产品角色）：它是被构建的复杂对象，包含多个组成部件，具体建造者创建该产品的内部表示并定义它的装配过程。

● Director（指挥者）：指挥者又称为导演类，它负责安排复杂对象的建造次序，指挥者与抽象建造者之间存在关联关系，可以在其construct()建造方法中调用建造者对象的部件构造与装配方法，完成复杂对象的建造。客户端一般只需要与指挥者进行交互，在客户端确定具体建造者的类型，并实例化具体建造者对象（也可以通过配置文件和反射机制），然后通过指挥者类的构造函数或者Setter方法将该对象传入指挥者类中。

### 3.2.code:

**public class Actor** {  
 **private String** *type*;  
 **private String** *sex*;  
 **private String** *face*;  
 **private String** *costume*;  
 **private String** *hairstyle*;  
 **public String** getType() {  
 **return** *type*;  
 }  
 **public void** setType(**String** type) {  
 **this**.*type* = type;  
 }  
 **public String** getSex() {  
 **return** *sex*;  
 }  
 **public void** setSex(**String** sex) {  
 **this**.*sex* = sex;  
 }  
 **public String** getFace() {  
 **return** *face*;  
 }  
 **public void** setFace(**String** face) {  
 **this**.*face* = face;  
 }  
 **public String** getCostume() {  
 **return** *costume*;  
 }  
 **public void** setCostume(**String** costume) {  
 **this**.*costume* = costume;  
 }  
 **public String** getHairstyle() {  
 **return** *hairstyle*;  
 }  
 **public void** setHairstyle(**String** hairstyle) {  
 **this**.*hairstyle* = hairstyle;  
 }  
}

**public abstract class ActorBuilder** {  
 **protected** Actor *actor* = **new** Actor();  
  
 **public abstract void** buildType();  
  
 **public abstract void** buildSex();  
  
 **public abstract void** buildFace();  
  
 **public abstract void** buildConstume();  
  
 **public abstract void** buildHairstyle();  
  
 **public** Actor createActor() {  
 **return** *actor*;  
 }  
}

**public class AngelBuilder extends** ActorBuilder {  
 **public void** buildType() {  
 actor.setType("天使");  
 }  
  
 **public void** buildSex() {  
 actor.setSex("女");  
 }  
  
 **public void** buildFace() {  
 actor.setFace("漂亮");  
 }  
  
 **public void** buildConstume() {  
 actor.setCostume("白裙");  
 }  
  
 **public void** buildHairstyle() {  
 actor.setHairstyle("披肩长发");  
 }  
  
}

**public class DevilBuilder extends** ActorBuilder {  
  
 **public void** buildType() {  
 actor.setType("恶魔");  
 }  
  
 **public void** buildSex() {  
 actor.setSex("妖");  
 }  
  
 **public void** buildFace() {  
 actor.setFace("丑陋");  
 }  
  
 **public void** buildConstume() {  
 actor.setCostume("黑衣");  
 }  
  
 **public void** buildHairstyle() {  
 actor.setHairstyle("光头");  
 }  
}

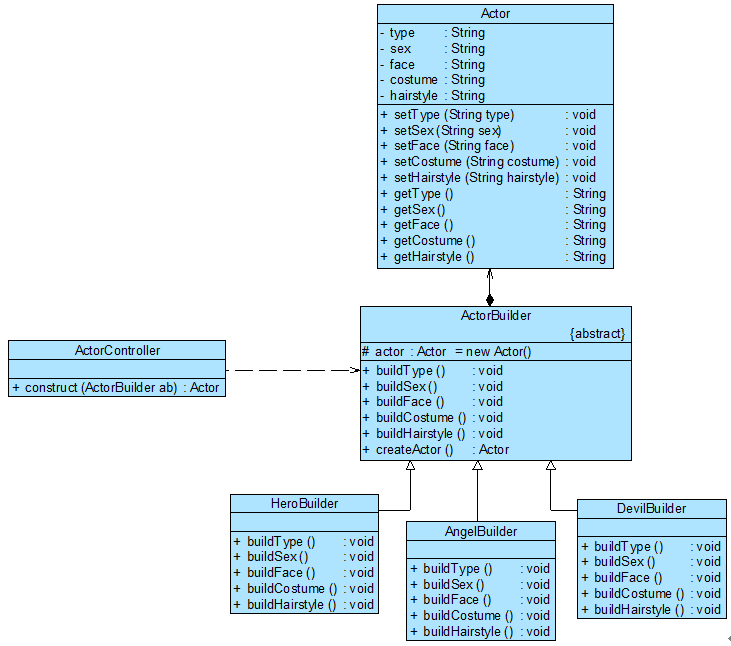
**public class HeroBuilder extends** ActorBuilder {  
  
 **public void** buildType() {  
 actor.setType("英雄");  
 }  
  
 **public void** buildSex() {  
 actor.setSex("男");  
 }  
  
 **public void** buildFace() {  
 actor.setFace("英俊");  
 }  
  
 **public void** buildConstume() {  
 actor.setCostume("盔甲");  
 }  
  
 **public void** buildHairstyle() {  
 actor.setHairstyle("飘逸");  
 }   
}

**public class ActorController** {  
 **public** Actor construct(ActorBuilder ab) {  
 Actor actor;  
  
 ab.buildType();  
 ab.buildSex();  
 ab.buildFace();  
 ab.buildConstume();  
 ab.buildHairstyle();  
  
 actor = ab.createActor();  
 **return** actor;  
 }  
}

**public class Client** {  
 **public static void** main(**String** args[]) {  
 ActorBuilder ab; // 针对抽象建造者编程  
 ab = **new** HeroBuilder(); // 反射生成具体建造者对象  
  
 ActorController ac = **new** ActorController();  
 Actor actor;  
 actor = ac.construct(ab); // 通过指挥者创建完整的建造者对象  
  
 **String** type = actor.getType();  
 **System**.***out***.println(type + "的外观：");  
 **System**.***out***.println("性别：" + actor.getSex());  
 **System**.***out***.println("面容：" + actor.getFace());  
 **System**.***out***.println("服装：" + actor.getCostume());  
 **System**.***out***.println("发型：" + actor.getHairstyle());  
 }  
}

天使的外观：  
性别：女  
面容：漂亮  
服装：白裙  
发型：披肩长发

### 3.3.完整解决方案



### 3.4.建造者模式总结

建造者模式的核心在于如何一步步构建一个包含多个组成部件的完整对象，使用相同的构建过程构建不同的产品，在软件开发中，如果我们需要创建复杂对象并希望系统具备很好的灵活性和可扩展性可以考虑使用建造者模式。

#### 1.主要优点

      建造者模式的主要优点如下：

(1) 在建造者模式中，客户端不必知道产品内部组成的细节，将产品本身与产品的创建过程解耦，使得相同的创建过程可以创建不同的产品对象。

(2) 每一个具体建造者都相对独立，而与其他的具体建造者无关，因此可以很方便地替换具体建造者或增加新的具体建造者，用户使用不同的具体建造者即可得到不同的产品对象。由于指挥者类针对抽象建造者编程，增加新的具体建造者无须修改原有类库的代码，系统扩展方便，符合“开闭原则”

(3) 可以更加精细地控制产品的创建过程。将复杂产品的创建步骤分解在不同的方法中，使得创建过程更加清晰，也更方便使用程序来控制创建过程。

#### 2.主要缺点

      建造者模式的主要缺点如下：

(1) 建造者模式所创建的产品一般具有较多的共同点，其组成部分相似，如果产品之间的差异性很大，例如很多组成部分都不相同，不适合使用建造者模式，因此其使用范围受到一定的限制。

(2) 如果产品的内部变化复杂，可能会导致需要定义很多具体建造者类来实现这种变化，导致系统变得很庞大，增加系统的理解难度和运行成本。

#### 3.适用场景

      在以下情况下可以考虑使用建造者模式：

(1) 需要生成的产品对象有复杂的内部结构，这些产品对象通常包含多个成员属性。

(2) 需要生成的产品对象的属性相互依赖，需要指定其生成顺序。

(3) 对象的创建过程独立于创建该对象的类。在建造者模式中通过引入了指挥者类，将创建过程封装在指挥者类中，而不在建造者类和客户类中。

(4) 隔离复杂对象的创建和使用，并使得相同的创建过程可以创建不同的产品。

## 4. Prototype 原型模式

**工厂模式,简单工厂模式,抽象工厂模式,单例模式,建造者模式,原型模式**

      张纪中版《西游记》以出乎意料的造型和雷人的台词遭到广大观众朋友的热议，我们在此对该话题不作过多讨论。但无论是哪个版本的《西游记》，孙悟空都是其中的一号雄性主角，关于他（或它）拔毛变小猴的故事几乎人人皆知，孙悟空可以用猴毛根据自己的形象，复制（又称“克隆”或“拷贝”）出很多跟自己长得一模一样的“身外身”来。在设计模式中也存在一个类似的模式，可以通过一个原型对象克隆出多个一模一样的对象，该模式称之为原型模式。

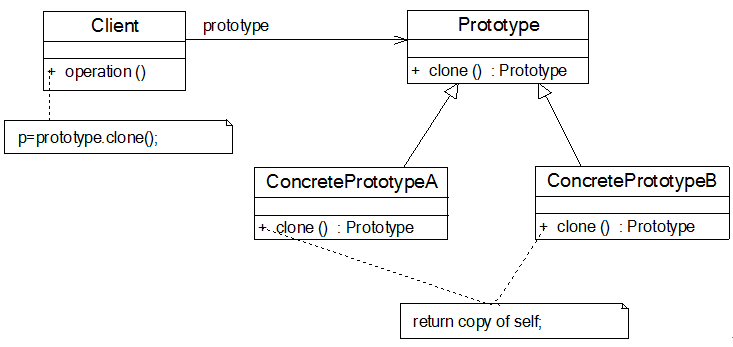
### 4.1.原型模式概述

      在使用原型模式时，我们需要首先创建一个原型对象，再通过复制这个原型对象来创建更多同类型的对象。试想，如果连孙悟空的模样都不知道，怎么拔毛变小猴子呢？原型模式的定义如下：

原型模式(Prototype  Pattern)：使用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。原型模式是一种对象创建型模式。

      原型模式的工作原理很简单：将一个原型对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝自己来实现创建过程。由于在软件系统中我们经常会遇到需要创建多个相同或者相似对象的情况，因此原型模式在真实开发中的使用频率还是非常高的。原型模式是一种“另类”的创建型模式，创建克隆对象的工厂就是原型类自身，工厂方法由克隆方法来实现。

      需要注意的是通过克隆方法所创建的对象是全新的对象，它们在内存中拥有新的地址，通常对克隆所产生的对象进行修改对原型对象不会造成任何影响，每一个克隆对象都是相互独立的。通过不同的方式修改可以得到一系列相似但不完全相同的对象。



●Prototype（抽象原型类）：它是声明克隆方法的接口，是所有具体原型类的公共父类，可以是抽象类也可以是接口，甚至还可以是具体实现类。

● ConcretePrototype（具体原型类）：它实现在抽象原型类中声明的克隆方法，在克隆方法中返回自己的一个克隆对象。

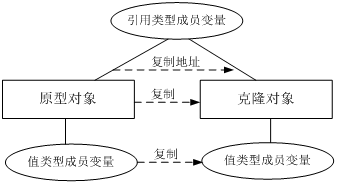
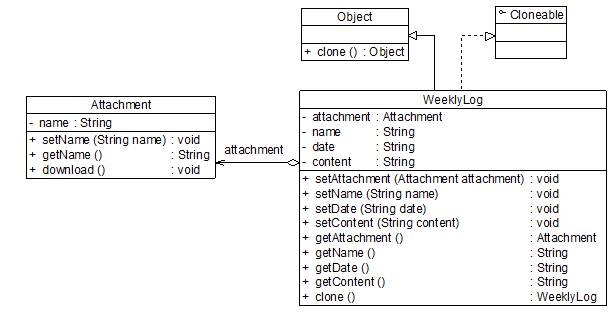
● Client（客户类）：让一个原型对象克隆自身从而创建一个新的对象，在客户类中只需要直接实例化或通过工厂方法等方式创建一个原型对象，再通过调用该对象的克隆方法即可得到多个相同的对象。由于客户类针对抽象原型类Prototype编程，因此用户可以根据需要选择具体原型类，系统具有较好的可扩展性，增加或更换具体原型类都很方便。

      原型模式的核心在于如何实现克隆方法，下面将介绍两种在Java语言中常用的克隆实现方法：

在Java语言中，数据类型分为值类型（基本数据类型）和引用类型，值类型包括int、double、byte、boolean、char等简单数据类型，引用类型包括类、接口、数组等复杂类型。浅克隆和深克隆的主要区别在于是否支持引用类型的成员变量的复制，下面将对两者进行详细介绍。

#### 1.浅克隆

      在浅克隆中，如果原型对象的成员变量是值类型，将复制一份给克隆对象；如果原型对象的成员变量是引用类型，则将引用对象的地址复制一份给克隆对象，也就是说原型对象和克隆对象的成员变量指向相同的内存地址。简单来说，在浅克隆中，当对象被复制时只复制它本身和其中包含的值类型的成员变量，而引用类型的成员对象并没有复制，如图7-4所示：

**图7-4 浅克隆示意图 图7-5 带附件的周报结构图（浅克隆）**

在Java语言中，通过覆盖Object类的clone()方法可以实现浅克隆。为了让大家更好地理解浅克隆和深克隆的区别，我们首先使用浅克隆来实现工作周报和附件类的复制，其结构如图7-5所示：

**public class Attchment** {  
 **private String** *name*;  
  
 **public String** getName() {  
 **return** *name*;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
  
 **public void** download() {  
 **System**.***out***.println("下载文件,文件名为:" + *name*);  
 }  
}

修改工作周报类WeeklyLog，修改后的代码如下：

**public class WeeklyLog implements Cloneable** {  
 **private Attchment** *attchment*;  
 **private String** *name*;  
 **private String** *date*;  
 **private String** *content*;  
  
 **public Attchment** getAttchment() {  
 **return** *attchment*;  
 }  
  
 **public void** setAttchment(**Attchment** attchment) {  
 **this**.*attchment* = attchment;  
 }  
  
 **public String** getName() {  
 **return** *name*;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public String** getDate() {  
 **return** *date*;  
 }  
  
 **public void** setDate(**String** date) {  
 **this**.*date* = date;  
 }  
  
 **public String** getContent() {  
 **return** *content*;  
 }  
  
 **public void** setContent(**String** content) {  
 **this**.*content* = content;  
 }  
  
 **public WeeklyLog** clone() {  
 **Object** obj = **null**;  
 **try** {  
 obj = **super**.clone();  
 **return** (**WeeklyLog**) obj;  
 } **catch** (**CloneNotSupportedException** e) {  
 **System**.***out***.println("不支持复制!");  
 **return null**;  
 }  
 }  
}

**public class Client** {  
 **public static void** main(**String**[] args) {  
 **WeeklyLog** log\_previous,log\_new;  
 log\_previous = **new** WeeklyLog();//创建原型对象  
 **Attchment** attchment = **new** Attchment();//创建附件对象  
 log\_previous.setAttchment(attchment);//将附件添加到周报中  
  
 log\_new = log\_previous.clone();//调用克隆方法创建克隆对象  
  
 //比较周报  
 **System**.***out***.println("周报是否相同?"+(log\_previous == log\_new));  
 //比较附件  
 **System**.***out***.println("附件是否相同?"+(log\_previous.getAttchment() == log\_new.getAttchment()));  
 }  
}

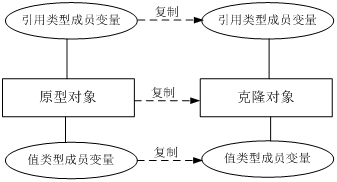
编译并运行程序，输出结果如下：

|  |
| --- |
| 周报是否相同？  false  附件是否相同？ true |

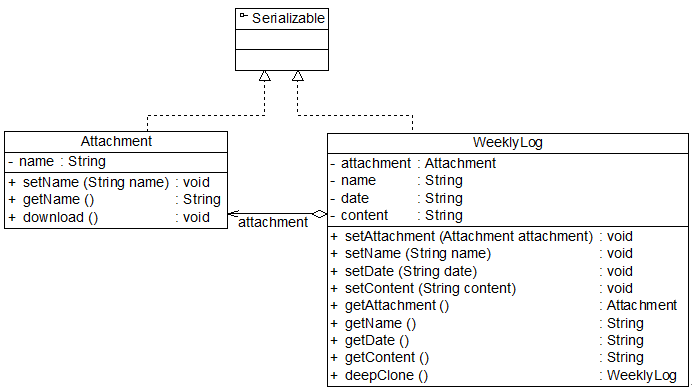
由于使用的是浅克隆技术，因此工作周报对象复制成功，通过“==”比较原型对象和克隆对象的内存地址时输出false；但是比较附件对象的内存地址时输出true，说明它们在内存中是同一个对象。

#### 2.深克隆

      在深克隆中，无论原型对象的成员变量是值类型还是引用类型，都将复制一份给克隆对象，深克隆将原型对象的所有引用对象也复制一份给克隆对象。简单来说，在深克隆中，除了对象本身被复制外，对象所包含的所有成员变量也将复制，如图7-6所示：



      在Java语言中，如果需要实现深克隆，可以通过序列化(Serialization)等方式来实现。序列化就是将对象写到流的过程，写到流中的对象是原有对象的一个拷贝，而原对象仍然存在于内存中。通过序列化实现的拷贝不仅可以复制对象本身，而且可以复制其引用的成员对象，因此通过序列化将对象写到一个流中，再从流里将其读出来，可以实现深克隆。需要注意的是能够实现序列化的对象其类必须实现Serializable接口，否则无法实现序列化操作。下面我们使用深克隆技术来实现工作周报和附件对象的复制，由于要将附件对象和工作周报对象都写入流中，因此两个类均需要实现Serializable接口，其结构如图7-7所示：



//附件类  
**class Attachment implements Serializable**{  
 **private String** *name*; //附件名  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public String** getName() {  
 **return this**.*name*;  
  
 }  
  
 **public void** download() {  
 **System**.***out***.println("下载附件，文件名为" + *name*);  
  
 }  
}

//工作周报类  
**class WeeklyLog implements Serializable** {  
  
 **private Attachment** *attachment*;  
 **private String** *name*;  
 **private String** *date*;  
 **private String** *content*;  
  
 **public void** setAttachment(**Attachment** attachment) {  
 **this**.*attachment* = attachment;  
 }  
  
 **public void** setName(**String** name) {  
 **this**.*name* = name;  
 }  
  
 **public void** setDate(**String** date) {  
 **this**.*date* = date;  
 }  
  
 **public void** setContent(**String** content) {  
 **this**.*content* = content;  
 }  
  
 **public Attachment** getAttachment() {  
 **return** (**this**.*attachment*);  
 }  
  
 **public String** getName() {  
 **return** (**this**.*name*);  
 }  
  
 **public String** getDate() {  
 **return** (**this**.*date*);  
 }  
  
 **public String** getContent() {  
 **return** (**this**.*content*);  
 }  
  
 //使用序列化技术实现深克隆  
  
 **public WeeklyLog** deepClone() **throws IOException**, **ClassNotFoundException**, **OptionalDataException** {  
  
 //将对象写入流中  
 **ByteArrayOutputStream** bao = **new** ByteArrayOutputStream();  
 **ObjectOutputStream** oos = **new** ObjectOutputStream(bao);  
 oos.writeObject(**this**);  
  
 //将对象从流中取出  
 **ByteArrayInputStream** bis = **new** ByteArrayInputStream(bao.toByteArray());  
 **ObjectInputStream** ois = **new** ObjectInputStream(bis);  
 **return** (**WeeklyLog**) ois.readObject();  
 }  
}

**class Client** {  
  
 **public static void** main(**String** args[]) {  
 **WeeklyLog** log\_previous, log\_new = **null**;  
 log\_previous = **new** WeeklyLog(); //创建原型对象  
 **Attachment** attachment = **new** Attachment(); //创建附件对象  
 log\_previous.setAttachment(attachment); //将附件添加到周报中  
  
 **try** {  
 log\_new = log\_previous.deepClone(); //调用深克隆方法创建克隆对象  
 } **catch** (**Exception** e) {  
 **System**.***err***.println("克隆失败！");  
 }  
  
 //比较周报  
 **System**.***out***.println("周报是否相同？ " + (log\_previous == log\_new));  
 //比较附件  
 **System**.***out***.println("附件是否相同？ " + (log\_previous.getAttachment() == log\_new.getAttachment()));  
 }  
}

编译并运行程序，输出结果如下：

|  |
| --- |
| 周报是否相同？  false  附件是否相同？  false |

       从输出结果可以看出，由于使用了深克隆技术，附件对象也得以复制，因此用“==”比较原型对象的附件和克隆对象的附件时输出结果均为false。深克隆技术实现了原型对象和克隆对象的完全独立，对任意克隆对象的修改都不会给其他对象产生影响，是一种更为理想的克隆实现方式。

### 4.2.原型模式总结

      原型模式作为一种快速创建大量相同或相似对象的方式，在软件开发中应用较为广泛，很多软件提供的复制(Ctrl + C)和粘贴(Ctrl + V)操作就是原型模式的典型应用，下面对该模式的使用效果和适用情况进行简单的总结。

#### ****1.**主要优点**

      原型模式的主要优点如下：

(1) 当创建新的对象实例较为复杂时，使用原型模式可以简化对象的创建过程，通过复制一个已有实例可以提高新实例的创建效率。

(2) 扩展性较好，由于在原型模式中提供了抽象原型类，在客户端可以针对抽象原型类进行编程，而将具体原型类写在配置文件中，增加或减少产品类对原有系统都没有任何影响。

(3) 原型模式提供了简化的创建结构，工厂方法模式常常需要有一个与产品类等级结构相同的工厂等级结构，而原型模式就不需要这样，原型模式中产品的复制是通过封装在原型类中的克隆方法实现的，无须专门的工厂类来创建产品。

(4) 可以使用深克隆的方式保存对象的状态，使用原型模式将对象复制一份并将其状态保存起来，以便在需要的时候使用（如恢复到某一历史状态），可辅助实现撤销操作。

#### ****2.**主要缺点**

      原型模式的主要缺点如下：

(1) 需要为每一个类配备一个克隆方法，而且该克隆方法位于一个类的内部，当对已有的类进行改造时，需要修改源代码，违背了“开闭原则”。

(2) 在实现深克隆时需要编写较为复杂的代码，而且当对象之间存在多重的嵌套引用时，为了实现深克隆，每一层对象对应的类都必须支持深克隆，实现起来可能会比较麻烦。

#### ****3.**适用场景**

        在以下情况下可以考虑使用原型模式：

(1) 创建新对象成本较大（如初始化需要占用较长的时间，占用太多的CPU资源或网络资源），新的对象可以通过原型模式对已有对象进行复制来获得，如果是相似对象，则可以对其成员变量稍作修改。

(2) 如果系统要保存对象的状态，而对象的状态变化很小，或者对象本身占用内存较少时，可以使用原型模式配合备忘录模式来实现。

(3) 需要避免使用分层次的工厂类来创建分层次的对象，并且类的实例对象只有一个或很少的几个组合状态，通过复制原型对象得到新实例可能比使用构造函数创建一个新实例更加方便。

# 二.结构型模式

## 1.Delegate委派模式

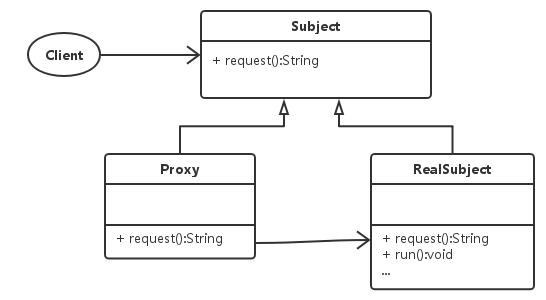
委派模式（Delegate）是面向对象设计模式中常用的一种模式。这种模式的原理为类B和类A是两个互相没有任何关系的类，B具有和A一模一样的方法和属性；并且调用B中的方法，属性就是调用A中同名的方法和属性。B好像就是一个受A授权委托的中介。第三方的代码不需要知道A的存在，也不需要和A发生直接的联系，通过B就可以直接使用A的功能，这样既能够使用到A的各种公能，又能够很好的将A保护起来了。一举两得，岂不很好！下面用一个很简单的例子来解释下：

**public class A** {  
 **void** method1(){}  
 **void** method2(){}  
}

**public class B** {  
 **A** *a* = **new** A();  
 **void** method1(){*a*.method1();}  
 **void** method2(){*a*.method2();}  
}

**public class Test** {  
 **public static void** main(**String**[] args) {  
 **B** b = **new** B();  
 b.method1();//invoke method1 of class A in fact  
 b.method2();//invoke method2 of class A in fact  
 }  
}

## 2.Proxy代理模式



### 2.1.什么是代理模式？

       代理模式的定义：代理模式给某一个对象提供一个代理对象，并由代理对象控制对原对象的引用。举例说明，就是一个人或者一个机构代表另一个人或者另一个机构采取行动。在一些情况下，一个客户不想或者不能够直接引用一个对象，而代理对象可以在客户端和目标对象之前起到中介的作用。

应用场景举例：

       通过上面的代理模式描述我们可以知道，其目的就是为了控制对象引用，生活场景中我们以买车为例，如果我们要买一辆轿车必须通过汽车4S店，汽车4s店就是充当代理角色，其目的就是控制买车客户的买车行为，必须通过汽车4S店才能从汽车厂商买一辆车。

1.）首先新建一个买车的接口

**public interface IBuyCar** {  
 //买车  
 **void** buyCar();  
}

2.）声明一个要买车的客户，实现买车接口

**public class Customer implements IBuyCar** {  
 **private int** *cash*;//购车款  
  
 **public int** getCash() {  
 **return** *cash*;  
 }  
  
 **public void** setCash(**int** cash) {  
 **this**.*cash* = cash;  
 }  
  
 @Override  
 **public void** buyCar() {  
 **System**.***out***.println("买车花费了-->" + *cash* + "元.");  
 }  
}

3.）声明一个买车代理汽车4S店，同样也实现买车接口，必须接受客户下单

**public class CarShopProxy implements IBuyCar**{  
 **private Customer** *customer*;//接收买车客户  
  
 **public CarShopProxy**(**Customer** customer) {  
 **this**.*customer* = customer;  
 }  
  
 @Override  
 **public void** buyCar() {  
 *customer*.buyCar();  
 }  
}

4.)创建一个客户端，模拟一次买车

**public class Client** {  
 **public static void** main(**String**[] args) {  
 **Customer** customer = **new** Customer();  
 customer.setCash(**120000**);  
 **CarShopProxy** carShopProxy = **new** CarShopProxy(customer);  
 carShopProxy.buyCar();  
 }  
}

### 2.2.JDK代理

一、现实生活中的代理？  
 在现实生活中，我们常见的有服务器代理商、联想PC代理商、百事可乐、火车票、机票等代理商，为什么会有这些个代理商呢？设想以买火车票为场景，如果我要买一张从广州去长沙的火车票，就必须去火车站排队购票，如果排队的人比较多的话，非常的耽误时间。但有了火车票代理商之后，我就可以直接去找个离我最近的代理商买票，因为这样的代理商不止一个二个，遍布全市各地。 所以代理商的出现不但减轻了火车站售票员的工作压力，同时也为市民购票提供了许多方便。只是代理商会收5块的手续费。从这个示例中可以发现和买票有关的一些名词：火车站、售票、代理商？解释这些名词在程序中代理的含义：火车站：称为目标，售票：目标的最终行为，代理商：和火车站具有同样售票行为的代理商，不过代理商在售票前和售票后会做一些操作，比如查询余票、售票后收取手续费等操作。

二、JDK中的代理

1、特征：   
 1）、jdk中代理的一个很重要的特征：代理类和目标类都拥有相同的接口，所以它们都拥有相同的行为。代理类的对象本身并不真正提供服务，而是调用目标类对象的相关方法，来提供特定的服务。

如：火车票代理商自己并不提供火车票销售的服务，而是调用火车站的售票服务。为顾客提供查询余票信息、火车的运营时间、火车票的销售等服务，这些服务都是来自火车站（目标类）。代理类主要负责为目标类预处理消息（如：身份证信息是否正确）、过滤消息（如：该顾客是否为通缉犯，是否公安监视的人），并把这些消息转发给目标类，以及事后处理消息（收取顾客的手续费）等。   
 2）、必须实现一个或多个接口

#### 2.2.1.JDK静态代理

a、定义一个火车站售票的接口

**public interface Ticket** {  
  
 **public void** ticket();  
}

b、售票接口的实现类

**public class TicketImpl implements Ticket** {  
  
 @Override  
 **public void** ticket() {  
 **System**.***out***.println("成功售出一张火车票！");  
 }  
}

c、火车票销售的代理类，与目标类实现了相同的接口

/\*\*  
 \* 火车票销售代理类  
 \*/  
**public class TicketImplProxy implements Ticket** {  
 **private TicketImpl** *ticketImpl*; //目标类  
 **public TicketImplProxy**(**TicketImpl** ticketImpl) {  
 **this**.*ticketImpl* = ticketImpl;  
 }  
  
 @Override  
 **public void** ticket() {  
 **System**.***out***.println("售票前验证顾客的身份信息…………");  
 *ticketImpl*.ticket();  
 **System**.***out***.println("售票后收取顾客的手续费…………");  
 }  
}

d、测试类：

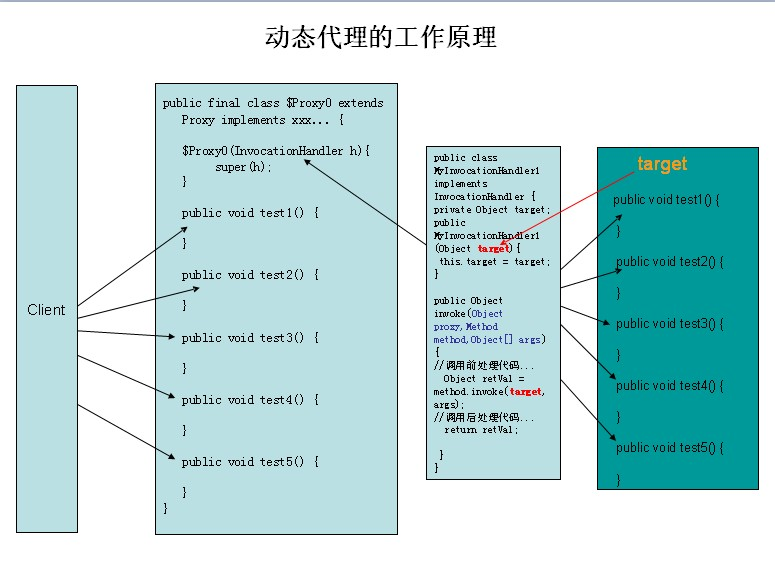
**public class StaticProxyTest** {  
 **public static void** main(**String**[] args) **throws Exception** {  
 **TicketImpl** ticketImpl = **new** TicketImpl(); //要代理的目标  
 **TicketImplProxy** staticProxy = **new** TicketImplProxy(ticketImpl);  
 staticProxy.ticket();  
 }  
}

运行结果：



静态代理的缺陷： 一个代理类只能代理一个目标类，如果在一个系统中想代理多个目标类的话，就得写多个代理类，这将是一件很繁琐的工作。所以在JDK1.3之后，引出了动态代理的概念，动态代理可以在程序运行的时候，通过反射机制动态的生成一个类的字节码文件，并实现目标类相同的接口。

#### **2.2.2.JDK动态代理**

1）、介绍： jdk中的动态代理，主要用到了java.lang.reflect包中的两个类：Proxy类和InvocationHandler接口，Proxy类用于生成代理类，InvocationHandler接口用于调用目标类的方法之前或之后，做一些处理。比如：记录日志、事务处理、效率测试等，均在该接口的invoke方法中实现。传说中AOP思想的原理就是从这里开始扩展的。  
  
  
2）、动态代理机制深入分析（以ArrayList为代理目标类为例）：  
 1、使用Proxy类的getProxyClass方法，获得ArrayList代理类的字节码文件

**Class** clazzProxy = Proxy.getProxyClass(ArrayList.**class**.getClassLoader(), ArrayList.**class**.getInterfaces());

2、打印代理类的结构信息获得这个Class对象的字节码文件之后，我们打印出来看看这个类的名字叫什么？

**System**.***out***.println("代理类名称：" + clazzProxy.getName());

代理类名称：$Proxy0

通过JDK反射机制，获代理类的结构，比如：

它的父类是谁？实现了哪些接口？有哪些个构造方法？有哪些个成员方法？

等。。。下面将通反射机制，获取这个代理类的组织结构信息：

**System**.***out***.println("代理类" + clazzProxy.getName() + "的父类：" + clazzProxy.getSuperclass().getName());

代理类$Proxy0的父类：java.lang.reflect.Proxy

**Class[] clazzInterfaces = clazzProxy.getInterfaces();**  
**StringBuilder** sbInterfaces = **new** StringBuilder();  
**for** (**Class** clazzInterface : clazzInterfaces) {  
 sbInterfaces.append(clazzInterface.getName()).append(",");  
}  
sbInterfaces.deleteCharAt(sbInterfaces.length() - **1**);  
**System**.***out***.println("代理类" + clazzProxy.getName() + "所实现的接口：" + sbInterfaces);

代理类$Proxy0所实现的接口：**java.util.List,java.util.RandomAccess,java.lang.Cloneable,java.io.Serializable**

**System**.***out***.println("代理类" + clazzProxy.getName() + "的访问修饰符：" + clazzProxy.getModifiers());

代理类$Proxy0的访问修饰符：17  
  
由访问修饰符的值可得知代理类的修饰符为：代理类的访问修饰符为：public final，由clazzProxy.getModifiers();

得知为17，其中public代表1，final代表16，相加得17。参考java.lang.reflect.Modifier

**System**.***out***.println("**\n**-------------打印代理类" + clazzProxy.getName() + "的构造方法列表-------------");  
**Constructor[] constructors = clazzProxy.getConstructors();**  
**for** (Constructor constructor : constructors) {  
 **System**.***out***.println("访问修饰符：" + constructor.getModifiers());  
 **String** name = constructor.getName();  
 **int** modifiers = constructor.getModifiers();  
 **StringBuilder** sb = **new** StringBuilder(name);  
 sb.append('(');  
 **Class**[] clazzParameters = constructor.getParameterTypes();  
 **for** (**Class** clazzParameter : clazzParameters) {  
 sb.append(clazzParameter.getName()).append(",");  
 }  
 **if** (clazzParameters != **null** && clazzParameters.*length* != **0**) {  
 sb.deleteCharAt(sb.length()-**1**);  
 }  
 sb.append(')');  
 **System**.***out***.println(sb.toString());  
}

-------------打印代理类$Proxy0的构造方法列表-------------访问修饰符：**1  
$Proxy0(java.lang.reflect.InvocationHandler)**

**System**.***out***.println("**\n**-------------打印代理类" + clazzProxy.getName() + "的方法列表-------------");  
**Method[] methods = clazzProxy.getMethods();**  
**for** (Method method : methods) {  
 **String** name = method.getName();  
 **StringBuilder** sb = **new** StringBuilder(name);  
 sb.append('(');  
 **Class**[] clazzParameters = method.getParameterTypes();  
 **for** (**Class** clazzParameter : clazzParameters) {  
 sb.append(clazzParameter.getName()).append(",");  
 }  
 **if** (clazzParameters != **null** && clazzParameters.*length* != **0**) {  
 sb.deleteCharAt(sb.length()-**1**);  
 }  
 sb.append(')');  
 **System**.***out***.println(sb.toString());  
}

打印结果：

-------------打印代理类$Proxy0的方法列表-------------

add(**java.lang.Object**)  
add(**int**,**java.lang.Object**)  
get(**int**)  
equals(**java.lang.Object**)  
toString()  
hashCode()  
indexOf(**java.lang.Object**)  
clear()  
contains(**java.lang.Object**)  
isEmpty()  
lastIndexOf(**java.lang.Object**)  
addAll(**java.util.Collection**)  
addAll(**int**,**java.util.Collection**)  
iterator()  
size()  
toArray()  
toArray([Ljava.lang.Object;)  
remove(**java.lang.Object**)  
remove(**int**)  
set(**int**,**java.lang.Object**)  
containsAll(**java.util.Collection**)  
removeAll(**java.util.Collection**)  
retainAll(**java.util.Collection**)  
subList(**int**,**int**)  
listIterator()  
listIterator(**int**)  
isProxyClass(**java.lang.Class**)  
getProxyClass(**java.lang.ClassLoader**,[Ljava.lang.Class;)  
getInvocationHandler(**java.lang.Object**)  
newProxyInstance(**java.lang.ClassLoader**,[Ljava.lang.Class;,**java.lang.reflect.InvocationHandler**)  
wait(**long**)  
wait()  
wait(**long**,**int**)  
getClass()  
notify()  
notifyAll()

这些方法分别来自List接口、Proxy和Object类，因为代理类会自动实现目标类相同的接口。  
//由上述信息可推断出代理类的文件结构（程序中只实现打印代理类的声明部份）

**StringBuilder** sbProxyClassStruct = **new** StringBuilder("public final class ");  
sbProxyClassStruct.append(clazzProxy.getName())  
 .append(" extends ").append(clazzProxy.getSuperclass().getName())  
 .append(" implements ").append(sbInterfaces).append(" { } ");  
**System**.***out***.println("代理类的结构：" + sbProxyClassStruct);

程序中只打印了代理类的声明部份，结果：  
代理类声明部份的结构：

**public final class $Proxy0 extends java.lang.reflect.Proxy implements java.util.List**,**java.util.RandomAccess**,**java.lang.Cloneable**,**java.io.Serializable** { }

由上述信息可推断出代理类的文件结构：

**public final class $Proxy0 extends java.lang.reflect.Proxy implements java.util.List**, **java.util.RandomAccess**, **java.lang.Cloneable**, **java.io.Serializable** {  
 **public $Proxy0**(InvocationHandler h) {  
 **super**(h);  
// *TODO Auto-generated constructor stub* }  
  
 @Override  
 **public boolean** add(**Object** e) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public void** add(**int** index, **Object** element) {  
// *TODO Auto-generated method stub* }  
  
 @Override  
 **public boolean** addAll(Collection c) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public boolean** addAll(**int** index, Collection c) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public void** clear() {  
// *TODO Auto-generated method stub* }  
  
 @Override  
 **public boolean** contains(**Object** o) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public boolean** containsAll(Collection c) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public Object** get(**int** index) {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public int** indexOf(**Object** o) {  
// *TODO Auto-generated method stub* **return 0**;  
 }  
  
 @Override  
 **public boolean** isEmpty() {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public** Iterator iterator() {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public int** lastIndexOf(**Object** o) {  
// *TODO Auto-generated method stub* **return 0**;  
 }  
  
 @Override  
 **public** ListIterator listIterator() {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public** ListIterator listIterator(**int** index) {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public boolean** remove(**Object** o) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public Object** remove(**int** index) {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public boolean** removeAll(Collection c) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public boolean** retainAll(Collection c) {  
// *TODO Auto-generated method stub* **return false**;  
 }  
  
 @Override  
 **public Object** set(**int** index, **Object** element) {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public int** size() {  
// *TODO Auto-generated method stub* **return 0**;  
 }  
  
 @Override  
 **public** List subList(**int** fromIndex, **int** toIndex) {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public Object**[] toArray() {  
// *TODO Auto-generated method stub* **return null**;  
 }  
  
 @Override  
 **public Object**[] toArray(**Object**[] a) {  
// *TODO Auto-generated method stub* **return null**;  
 }  
}

由上述信息得知该代理类只有一个带InvocationHandler参数的构造方法，现在我们来生成这个代理类的实例对象：  
//方式1，创建一个内部类，并实现InvocationHandler接口

Constructor proxy1 = clazzProxy.getConstructor(InvocationHandler.**class**);  
**System**.***out***.println("------------创建代理类实例，方式1--------------------");  
**class MyInvocationHandler implements** InvocationHandler {  
 ArrayList *target* = **new** ArrayList();  
 @Override  
 **public Object** invoke(**Object** proxy, Method method, **Object**[] args) **throws Throwable** {  
 **String** name = method.getName();  
 **System**.***out***.println(name + "方法调用前...");  
 **Object** retVal = method.invoke(*target*, args);  
 **System**.***out***.println(name + "方法调用后的返回结果为：" + retVal);  
 **System**.***out***.println(name + "方法调用后...**\n**");  
 **return** retVal;  
 }  
}  
List list1 = (List) proxy1.newInstance(**new** MyInvocationHandler());  
list1.add("zhangsan");  
list1.add("lisi");  
**System**.***out***.println(list1.size());

调用List接口的add方法和size方法后的输出结果：  
---------------------创建代理类$Proxy0的实例对象----------------------

------------创建代理类实例，方式1--------------------  
 add方法调用前...  
 add方法调用后的返回结果为：**true** add方法调用后...  
  
 add方法调用前...  
 add方法调用后的返回结果为：**true** add方法调用后...  
  
 size方法调用前...  
 size方法调用后的返回结果为：**2** size方法调用后...  
 **2**

//方式2

Constructor prox2 = clazzProxy.getConstructor(InvocationHandler.**class**);  
List list2 = (List)prox2.newInstance(**new** InvocationHandler() {  
 List *target* = **new** ArrayList();  
 @Override  
 **public Object** invoke(**Object** proxy, Method method, **Object**[] args) **throws Throwable** {  
 **Object** retVal = method.invoke(*target*, args);  
 **return** retVal;  
 }}  
);

//方式3(匿名类)，将创建代理类和创建代理类实例对象的步聚合二为一

**System**.***out***.println("------------创建代理类实例，方式3--------------------");  
List list3 = (List) Proxy.newProxyInstance(  
 ArrayList.**class**.getClassLoader(),  
 ArrayList.**class**.getInterfaces(),  
 **new** InvocationHandler() {  
 ArrayList *target* = **new** ArrayList();  
  
 @Override  
 **public Object** invoke(**Object** proxy, Method method,  
 **Object**[] args) **throws Throwable** {  
 //测试方法的运行效率  
 **long** beginTime = **System**.currentTimeMillis();  
 **Object** retVal = method.invoke(*target*, args);  
 **long** endTime = **System**.currentTimeMillis();  
 **System**.***out***.println(method.getName() + " 方法运行时长为：" + (endTime - beginTime));  
 **return** retVal;  
 }  
 }  
);  
list3.add("hy");  
list3.add("yangxin");  
**System**.***out***.println(list3.size());

输出结果：------------创建代理类实例，方式3--------------------

add 方法运行时长为：**0** add 方法运行时长为：**0** size 方法运行时长为：**0  
 2**

\*通过以上代码的分析，可将生成代理类的代码，抽取成一个生成代理的通用方法，并将功能代码用一个对象封装起来（如：测试方法的运行时间，方法运行前或运行后需要做的事情）

/\*\*  
 \* 获得代理对象  
 \*  
 \* @param target 目标（被代理的对象）  
 \* @param advice 目标对象中的方法被调用前要执行的功能  
 \* @return 目标的代理对象  
 \*/  
**private static Object** getProxy(**final Object** target, **final** Advice advice) {  
 **return** Proxy.newProxyInstance(  
 target.getClass().getClassLoader(),  
 target.getClass().getInterfaces(),  
 **new** InvocationHandler() {  
 @Override  
 **public Object** invoke(**Object** proxy, Method method,  
 **Object**[] args) **throws Throwable** {  
 **Object** retVal = **null**;  
 **try** {  
 **advice**.doBefore(**target**, method, args);//方法执行前  
 retVal = method.invoke(**target**, args);  
 **advice**.doAfter(**target**, method, args, retVal);//方法执行后  
 } **catch** (**Exception** e) {  
 **advice**.doThrow(**target**, method, args, e);//方法抛出异常  
 } **finally** {  
 **advice**.doFinally(**target**, method, args);//方法最终执行代码(用于释放数据资源、关闭IO流等操作)  
 }  
 **return** retVal;  
 }  
 }  
 );  
}

封装功能的对象：

/\*\*  
 \* aop接口，提供方法运行前、方法运行后、方法运行中产生Exception、方法最终运行代码  
 \*  
 \*/  
**public interface Advice** {  
  
 /\*\*  
 \* 方法运行前  
 \* @param target 被代理的目标对象  
 \* @param method 被调用的方法  
 \* @param args 方法的参数  
 \*/  
 **public void** doBefore(**Object** target, Method method, **Object**[] args);  
  
 /\*\*  
 \* 方法运行后  
 \* @param target 被代理的目标对象  
 \* @param method 被调用的方法对象  
 \* @param args 方法的参数  
 \* @param retVal 方法的返回值  
 \*/  
 **public void** doAfter(**Object** target, Method method, **Object**[] args, **Object** retVal);  
  
 /\*\*  
 \* 方法运行时产生的异常  
 \* @param target 被代理的目标对象  
 \* @param method 被调用的方法  
 \* @param args 方法参数  
 \* @param e 运行时的异常对象  
 \*/  
 **public void** doThrow(**Object** target, Method method, **Object**[] args, **Exception** e);  
  
 /\*\*  
 \* 最终要执行的功能(如释放数据库连接的资源、关闭IO流等)  
 \* @param target 被代理的目标对象  
 \* @param method 被调用的方法  
 \* @param args 方法参数  
 \*/  
 **public void** doFinally(**Object** target, Method method, **Object**[] args);  
}

再建一个日志功能的实现类LogAdvice，用于测试：

/\*\*  
 \* 日志功能切入类  
 \*/  
**public class LogAdvice implements** Advice {  
 **long** *beginTime* = **System**.currentTimeMillis();  
  
 @Override  
 **public void** doBefore(**Object** target, Method method, **Object**[] args) {  
 **System**.***out***.println(target.getClass().getSimpleName() + "." + method.getName() + "方法被调用，参数值：" + Arrays.toString(args));  
 }  
  
 @Override  
 **public void** doAfter(**Object** target, Method method, **Object**[] args, **Object** retVal) {  
 **long** endTime = **System**.currentTimeMillis();  
 **System**.***out***.println(target.getClass().getSimpleName() +  
 "." + method.getName() + "方法运行结束，返回值：" + retVal + "，耗时" + (endTime - *beginTime*) + "毫秒。");  
 }  
  
 @Override  
 **public void** doThrow(**Object** target, Method method, **Object**[] args, **Exception** e) {  
 **System**.***out***.println("调用" + target.getClass().getSimpleName() + "." + method.getName() + "方法发生异常，异常消息：");  
 e.printStackTrace();  
 }  
  
 @Override  
 **public void** doFinally(**Object** target, Method method, **Object**[] args) {  
 **System**.***out***.println("doFinally...");  
 }  
}

测试生成代理的通用方法：

ArrayList target = **new** ArrayList();  
List list4 = (List)getProxy(target,**new** LogAdvice());  
list4.add("张三");  
list4.add("李四");  
list4.add("王五");  
System.out.println(list4.size());  
list4.get(**3**); //演示异常advice

输出结果：

ArrayList.add方法被调用，参数值：[张三]  
ArrayList.add方法运行结束，返回值：**true**，耗时0毫秒。  
doFinally...  
ArrayList.add方法被调用，参数值：[李四]  
ArrayList.add方法运行结束，返回值：**true**，耗时0毫秒。  
doFinally...  
ArrayList.add方法被调用，参数值：[王五]  
ArrayList.add方法运行结束，返回值：**true**，耗时0毫秒。  
doFinally...  
ArrayList.size方法被调用，参数值：**null**ArrayList.size方法运行结束，返回值：**3**，耗时0毫秒。  
doFinally...  
**3**ArrayList.get方法被调用，参数值：[**3**]

调用ArrayList.get方法发生异常，异常消息：

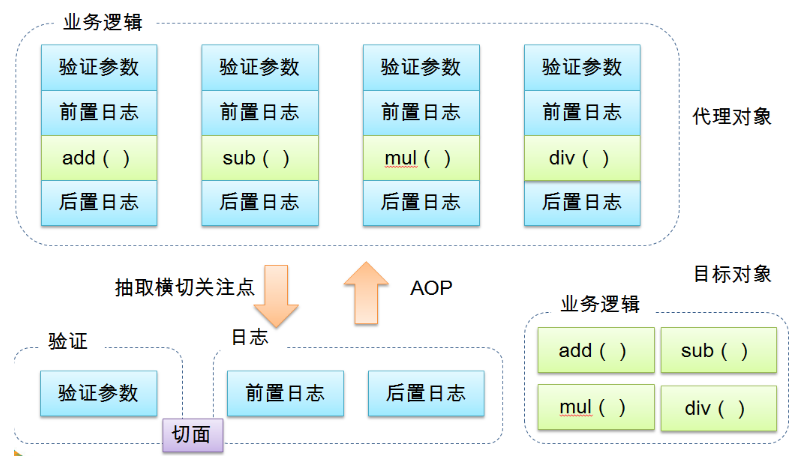
java.lang.reflect.InvocationTargetException  
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)  
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:**39**)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:**25**)  
at java.lang.reflect.Method.invoke(Method.java:**597**)  
at proxy.DynamicProxyTest$3.invoke(DynamicProxyTest.java:**163**)  
at $Proxy0.get(Unknown Source)  
at proxy.DynamicProxyTest.main(DynamicProxyTest.java:**143**)  
Caused by: java.lang.IndexOutOfBoundsException: Index: **3**, Size: **3**at java.util.ArrayList.RangeCheck(ArrayList.java:**547**)  
at java.util.ArrayList.get(ArrayList.java:**322**)  
... **7** more  
doFinally...

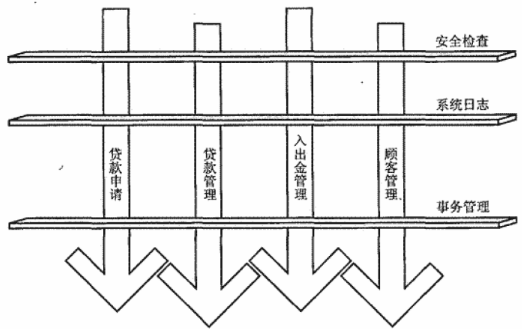
完毕，通过动态代理机制，实现了传说中的AOP思想。

### 2.3.spring中的aop,用到的代理

一、为什么需要代理模式二、静态代理三、动态代理，使用JDK内置的Proxy实现四、动态代理，使用cglib实现五、使用Spring实现AOP六、使用IOC配置的方式实现AOP七、使用XML配置Spring AOP切面八、示例下载九、视频十、作业

AOP（Aspect Oriented Programming）意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是OOP的延续，是软件开发中的一个热点，也是Spring框架中的一个重要内容，是函数式编程的一种衍生范型。利用AOP可以对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。





#### 一、为什么需要代理模式

假设需实现一个计算的类Math、完成加、减、乘、除功能，如下所示：

**public class Math** {  
 //加  
 **public int** add(**int** n1, **int** n2) {  
 **int** result = n1 + n2;  
 **System**.***out***.println(n1 + "+" + n2 + "=" + result);  
 **return** result;  
 }  
  
  
 //减  
 **public int** sub(**int** n1, **int** n2) {  
 **int** result = n1 - n2;  
 **System**.***out***.println(n1 + "-" + n2 + "=" + result);  
 **return** result;  
 }  
  
 //乘  
 **public int** mut(**int** n1, **int** n2) {  
 **int** result = n1 \* n2;  
 **System**.***out***.println(n1 + "X" + n2 + "=" + result);  
 **return** result;  
 }  
  
 //除  
 **public int** div(**int** n1, **int** n2) {  
 **int** result = n1 / n2;  
 **System**.***out***.println(n1 + "/" + n2 + "=" + result);  
 **return** result;  
 }  
}

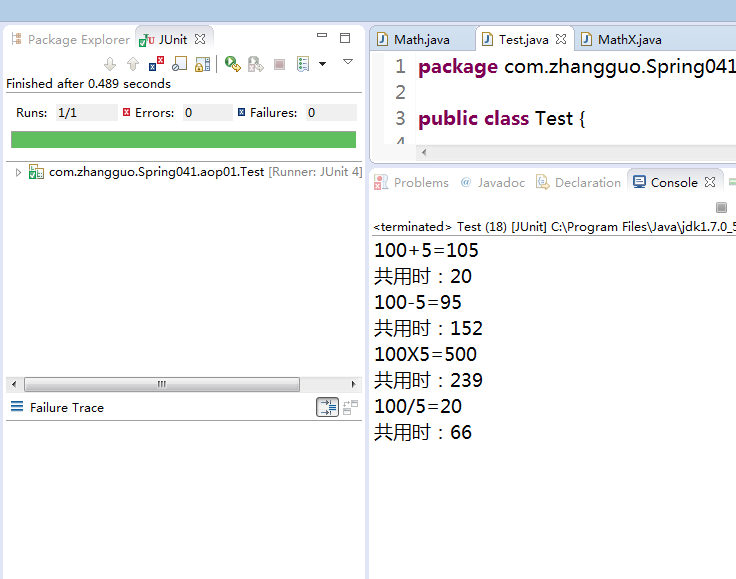
现在需求发生了变化，要求项目中所有的类在执行方法时输出执行耗时。最直接的办法是修改源代码，如下所示：

**public class Math** {  
 //加  
 **public int** add(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = n1 + n2;  
 **System**.***out***.println(n1 + "+" + n2 + "=" + result);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //减  
 **public int** sub(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = n1 - n2;  
 **System**.***out***.println(n1 + "-" + n2 + "=" + result);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //乘  
 **public int** mut(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = n1 \* n2;  
 **System**.***out***.println(n1 + "X" + n2 + "=" + result);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //除  
 **public int** div(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = n1 / n2;  
 **System**.***out***.println(n1 + "/" + n2 + "=" + result);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //模拟延时  
 **public void** lazy() {  
 **try** {  
 **int** n = (**int**) **new** Random().nextInt(**500**);  
 **Thread**.sleep(n);  
 } **catch** (**InterruptedException** e) {  
 e.printStackTrace();  
 }  
 }  
}

测试运行：

**public class Test** {  
 @**org.**junit.Test  
 **public void** test01() {  
 **Math** math = **new** Math();  
 **int** n1 = **100**, n2 = **5**;  
 math.add(n1, n2);  
 math.sub(n1, n2);  
 math.mut(n1, n2);  
 math.div(n1, n2);  
 }  
}

运行结果：



缺点：

1、工作量特别大，如果项目中有多个类，多个方法，则要修改多次。

2、违背了设计原则：开闭原则（OCP），对扩展开放，对修改关闭，而为了增加功能把每个方法都修改了，也不便于维护。

3、违背了设计原则：单一职责（SRP），每个方法除了要完成自己本身的功能，还要计算耗时、延时；每一个方法引起它变化的原因就有多种。

4、违背了设计原则：依赖倒转（DIP），抽象不应该依赖细节，两者都应该依赖抽象。而在Test类中，Test与Math都是细节。

使用静态代理可以解决部分问题。

#### 二、静态代理

 1、定义抽象主题接口。

/\*\*  
 \* 接口  
 \* 抽象主题  
 \*/  
**public interface IMath** {  
 //加  
 **int** add(**int** n1, **int** n2);  
  
 //减  
 **int** sub(**int** n1, **int** n2);  
  
 //乘  
 **int** mut(**int** n1, **int** n2);  
  
 //除  
 **int** div(**int** n1, **int** n2);  
}

2、主题类，算术类，实现抽象接口。

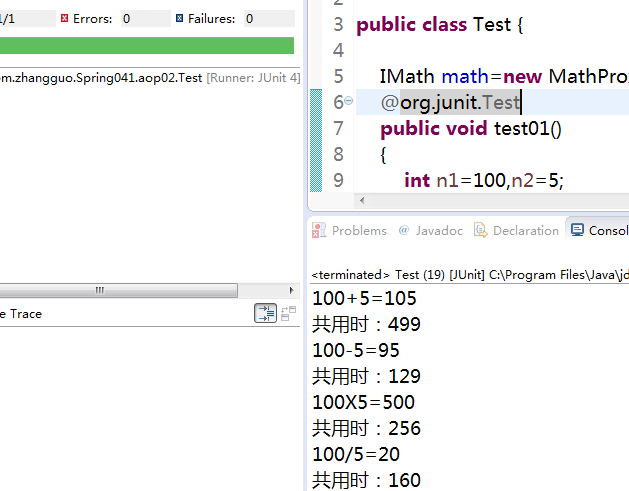
/\*\*  
 \* 被代理的目标对象  
 \* 真实主题  
 \*/  
**public class Math implements IMath** {  
 //加  
 **public int** add(**int** n1, **int** n2) {  
 **int** result = n1 + n2;  
 **System**.***out***.println(n1 + "+" + n2 + "=" + result);  
 **return** result;  
 }  
  
 //减  
 **public int** sub(**int** n1, **int** n2) {  
 **int** result = n1 - n2;  
 **System**.***out***.println(n1 + "-" + n2 + "=" + result);  
 **return** result;  
 }  
  
 //乘  
 **public int** mut(**int** n1, **int** n2) {  
 **int** result = n1 \* n2;  
 **System**.***out***.println(n1 + "X" + n2 + "=" + result);  
 **return** result;  
 }  
  
 //除  
 **public int** div(**int** n1, **int** n2) {  
 **int** result = n1 / n2;  
 **System**.***out***.println(n1 + "/" + n2 + "=" + result);  
 **return** result;  
 }  
}

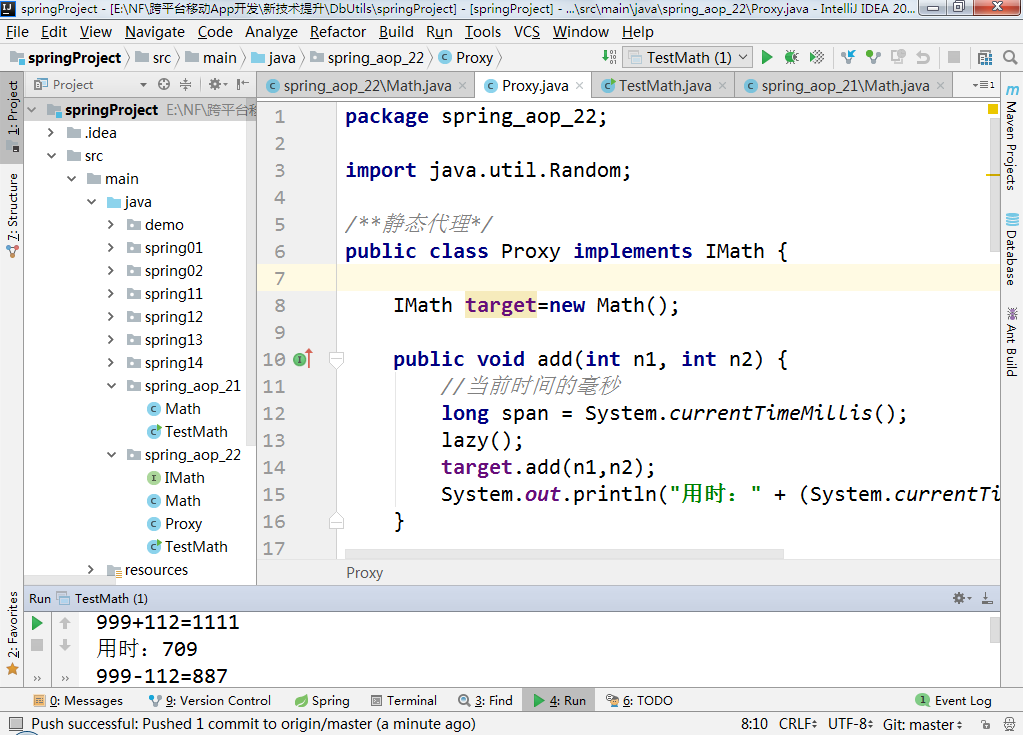
3、代理类

/\*\*  
 \* 静态代理类  
 \*/  
**public class MathProxy implements IMath** {  
  
 //被代理的对象  
 **IMath** *math* = **new** Math();  
  
 //加  
 **public int** add(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = *math*.add(n1, n2);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //减法  
 **public int** sub(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = *math*.sub(n1, n2);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //乘  
 **public int** mut(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = *math*.mut(n1, n2);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //除  
 **public int** div(**int** n1, **int** n2) {  
 //开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
 **int** result = *math*.div(n1, n2);  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
 **return** result;  
 }  
  
 //模拟延时  
 **public void** lazy() {  
 **try** {  
 **int** n = (**int**) **new** Random().nextInt(**500**);  
 **Thread**.sleep(n);  
 } **catch** (**InterruptedException** e) {  
 e.printStackTrace();  
 }  
 }  
}

4、测试运行

**public class Test** {  
 **IMath** *math*=**new** MathProxy();  
 @**org.**junit.Test  
 **public void** test01()  
 {  
 **int** n1=**100**,n2=**5**;  
 *math*.add(n1, n2);  
 *math*.sub(n1, n2);  
 *math*.mut(n1, n2);  
 *math*.div(n1, n2);  
 }  
}





5、小结

通过静态代理，是否完全解决了上述的4个问题：

已解决：

5.1、解决了“开闭原则（OCP）”的问题，因为并没有修改Math类，而扩展出了MathProxy类。

5.2、解决了“依赖倒转（DIP）”的问题，通过引入接口。

5.3、解决了“单一职责（SRP）”的问题，Math类不再需要去计算耗时与延时操作，但从某些方面讲MathProxy还是存在该问题。

未解决：

5.4、如果项目中有多个类，则需要编写多个代理类，工作量大，不好修改，不好维护，不能应对变化。

如果要解决上面的问题，可以使用动态代理。

#### 三、动态代理，使用JDK内置的Proxy实现

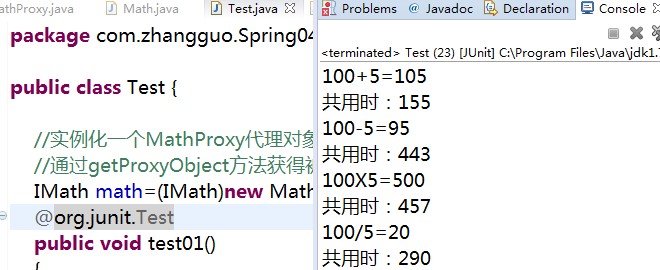
只需要一个代理类，而不是针对每个类编写代理类。

在上一个示例中修改代理类MathProxy如下：

/\*\*  
 \* 动态代理类  
 \*/  
**public class DynamicProxy implements InvocationHandler** {  
 //被代理的对象  
 **Object** *targetObject*;  
  
 /\*\*  
 \* 获得被代理后的对象  
 \*  
 \* @param object 被代理的对象  
 \* @return 代理后的对象  
 \*/  
 **public Object** getProxyObject(**Object** object) {  
 **this**.*targetObject* = object;  
 **return** Proxy.newProxyInstance(  
 *targetObject*.getClass().getClassLoader(), //类加载器  
 *targetObject*.getClass().getInterfaces(), //获得被代理对象的所有接口  
 **this**); //InvocationHandler对象  
 //loader:一个ClassLoader对象，定义了由哪个ClassLoader对象来生成代理对象进行加载  
 //interfaces:一个Interface对象的数组，表示的是我将要给我需要代理的对象提供一组什么接口，如果我提供了一组接口给它，那么这个代理对象就宣称实现了该接口(多态)，这样我就能调用这组接口中的方法了  
 //h:一个InvocationHandler对象，表示的是当我这个动态代理对象在调用方法的时候，会关联到哪一个InvocationHandler对象上，间接通过invoke来执行  
 }  
  
  
 /\*\*  
 \* 当用户调用对象中的每个方法时都通过下面的方法执行，方法必须在接口  
 \* proxy 被代理后的对象  
 \* method 将要被执行的方法信息（反射）  
 \* args 执行方法时需要的参数  
 \*/  
 **public Object** invoke(**Object** proxy, **Method** method, **Object**[] args) **throws Throwable** {  
 //被织入的内容，开始时间  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
  
 //使用反射在目标对象上调用方法并传入参数  
 **Object** result = method.invoke(*targetObject*, args);  
  
 //被织入的内容，结束时间  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
  
 **return** result;  
 }  
  
 //模拟延时  
 **public void** lazy() {  
 **try** {  
 **int** n = (**int**) **new** Random().nextInt(**500**);  
 **Thread**.sleep(n);  
 } **catch** (**InterruptedException** e) {  
 e.printStackTrace();  
 }  
 }  
}

测试运行：

**public class Test** {  
 //实例化一个MathProxy代理对象  
 //通过getProxyObject方法获得被代理后的对象  
 **IMath** *math* = (**IMath**) **new** DynamicProxy().getProxyObject(**new** Math());  
  
 @**org.**junit.Test  
 **public void** test01() {  
 **int** n1 = **100**, n2 = **5**;  
 *math*.add(n1, n2);  
 *math*.sub(n1, n2);  
 *math*.mut(n1, n2);  
 *math*.div(n1, n2);  
 }  
  
 IMessage *message* = (IMessage) **new** DynamicProxy().getProxyObject(**new** Message());  
  
 @**org.**junit.Test  
 **public void** test02() {  
 *message*.message();  
 }  
}



小结：

JDK内置的Proxy动态代理可以在运行时动态生成字节码，而没必要针对每个类编写代理类。中间主要使用到了一个接口InvocationHandler与Proxy.newProxyInstance静态方法，参数说明如下：

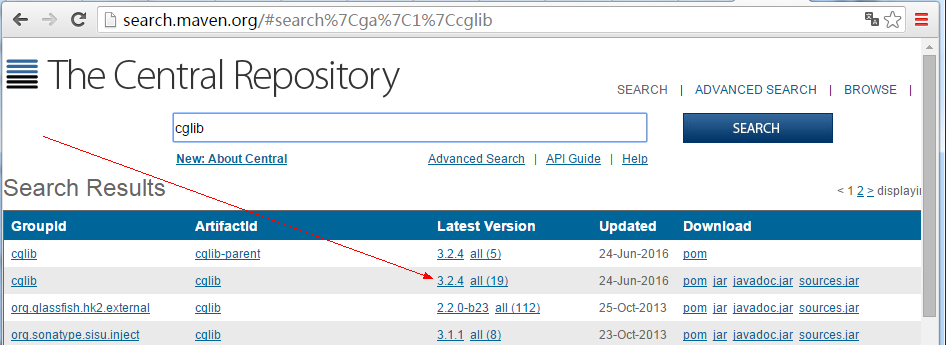
使用内置的Proxy实现动态代理有一个问题：被代理的类必须实现接口，未实现接口则没办法完成动态代理。

如果项目中有些类没有实现接口，则不应该为了实现动态代理而刻意去抽出一些没有实例意义的接口，通过cglib可以解决该问题。

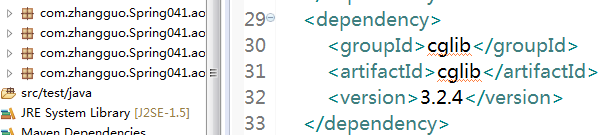
#### 四****、动态代理，使用cglib实现****

CGLIB(Code Generation Library)是一个开源项目,是一个强大的，高性能，高质量的Code生成类库，它可以在运行期扩展Java类与实现Java接口，通俗说cglib可以在运行时动态生成字节码。

4.1、引用cglib，通过maven



修改pom.xml文件，添加依赖



保存pom.xml配置文件，将自动从共享资源库下载cglib所依赖的jar包，主要有如下几个：



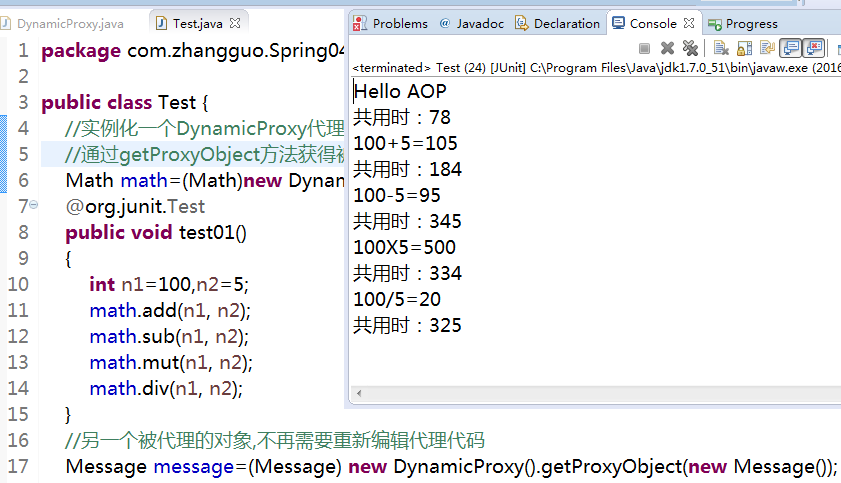
4.2、使用cglib完成动态代理，大概的原理是：cglib继承被代理的类，重写方法，织入通知，动态生成字节码并运行，因为是继承所以final类是没有办法动态代理的。具体实现如下：

*/\*  
 \* 动态代理类  
 \* 实现了一个方法拦截器接口  
 \*/***public class DynamicProxy implements** MethodInterceptor {  
  
 // 被代理对象  
 **Object** *targetObject*;  
  
 //Generate a new class if necessary and uses the specified callbacks (if any) to create a new object instance.  
 //Uses the no-arg constructor of the superclass.  
 //动态生成一个新的类，使用父类的无参构造方法创建一个指定了特定回调的代理实例  
 **public Object** getProxyObject(**Object** object) {  
 **this**.*targetObject* = object;  
 //增强器，动态代码生成器  
 Enhancer enhancer = **new** Enhancer();  
 //回调方法  
 enhancer.setCallback(**this**);  
 //设置生成类的父类类型  
 enhancer.setSuperclass(*targetObject*.getClass());  
 //动态生成字节码并返回代理对象  
 **return** enhancer.create();  
 }  
  
 // 拦截方法  
 **public Object** intercept(**Object** object, Method method, **Object**[] args, MethodProxy methodProxy) **throws Throwable** {  
 // 被织入的横切内容，开始时间 before  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
  
 // 调用方法  
 **Object** result = methodProxy.invoke(*targetObject*, args);  
  
 // 被织入的横切内容，结束时间  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
  
 **return** result;  
 }  
  
 // 模拟延时  
 **public void** lazy() {  
 **try** {  
 **int** n = (**int**) **new** Random().nextInt(**500**);  
 **Thread**.sleep(n);  
 } **catch** (**InterruptedException** e) {  
 e.printStackTrace();  
 }  
 }  
}

测试运行：

**public class Test** {  
 //实例化一个DynamicProxy代理对象  
 //通过getProxyObject方法获得被代理后的对象  
 **Math** *math*=(**Math**)**new** DynamicProxy().getProxyObject(**new** Math());  
 @**org.**junit.Test  
 **public void** test01()  
 {  
 **int** n1=**100**,n2=**5**;  
 *math*.add(n1, n2);  
 *math*.sub(n1, n2);  
 *math*.mut(n1, n2);  
 *math*.div(n1, n2);  
 }  
 //另一个被代理的对象,不再需要重新编辑代理代码  
 Message *message*=(Message) **new** DynamicProxy().getProxyObject(**new** Message());  
 @**org.**junit.Test  
 **public void** test02()  
 {  
 *message*.message();  
 }  
}

运行结果：



4.3、小结

使用cglib可以实现动态代理，即使被代理的类没有实现接口，但被代理的类必须不是final类。

#### 五、使用Spring实现AOP

横切关注点：跨越应用程序多个模块的方法或功能。（软件系统，可以看做由一组关注点即业务或功能或方法组成。其中，直接的业务关注点是直切关注点，而为直切关注点服务的，就是横切关注点。）即是，与我们业务逻辑无关的，但是我们需要关注的部分，就是横切关注点。

切面（ASPECT）：横切关注点被模块化的特殊对象。即，它是一个类。

通知（Advice）：切面必须要完成的工作。即，它是类中的一个方法。

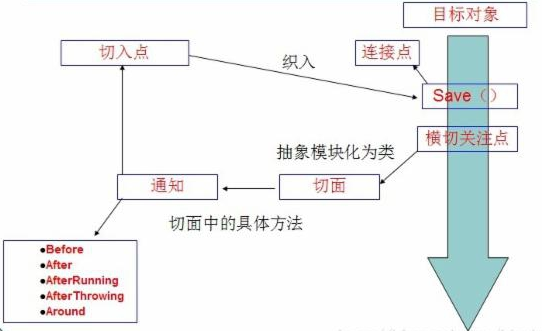
目标（Target）：被通知对象。

代理（Proxy）：向目标对象应用通知之后创建的对象。

切入点（PointCut）：切面通知执行的“地点”的定义。

连接点（JointPoint）：与切入点匹配的执行点。

下面示意图：



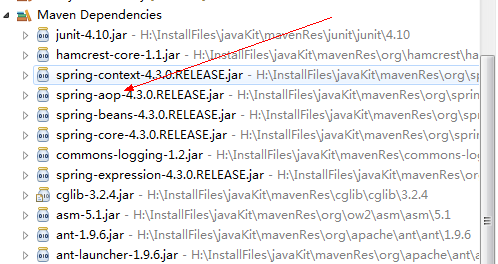
SpringAOP中，通过Advice定义横切逻辑，Spring中支持5种类型的Advice:



5.1、新建 一个Maven项目，在项目中引入Spring核心库与AOP，修改pom.xml文件，在dependencies中增加如下节点：

<**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context</**artifactId**>  
 <**version**>4.3.0.RELEASE</**version**>  
</**dependency**>

当保存pom.xml文件时会从远程共享库自动将需要引入的jar包下载到本地并引入项目中：



5.2、定义通知（Advice）

前置通知

**import java.lang.reflect.Method**;  
**import org.**springframework.aop.MethodBeforeAdvice;  
  
/\*\*  
 \* 前置通知  
 \*/  
**public class BeforeAdvice implements** MethodBeforeAdvice {  
 /\*\*  
 \* method 方法信息  
 \* args 参数  
 \* target 被代理的目标对象  
 \*/  
 **public void** before(**Method** method, **Object**[] args, **Object** target) **throws Throwable** {  
 **System**.***out***.println("-----------------前置通知-----------------");  
 }  
}

后置通知

**import java.lang.reflect.Method**;  
**import org.**springframework.aop.AfterReturningAdvice;  
  
/\*\*  
 \* 后置通知  
 \*/  
**public class AfterAdvice implements** AfterReturningAdvice {  
 */\*  
 \* returnValue 返回值  
 \* method 被调用的方法  
 \* args 方法参数  
 \* target 被代理对象  
 \*/* **public void** afterReturning(**Object** returnValue, **Method** method, **Object**[] args, **Object** target) **throws Throwable** {  
 **System**.***out***.println("-----------------后置通知-----------------");  
 }  
}

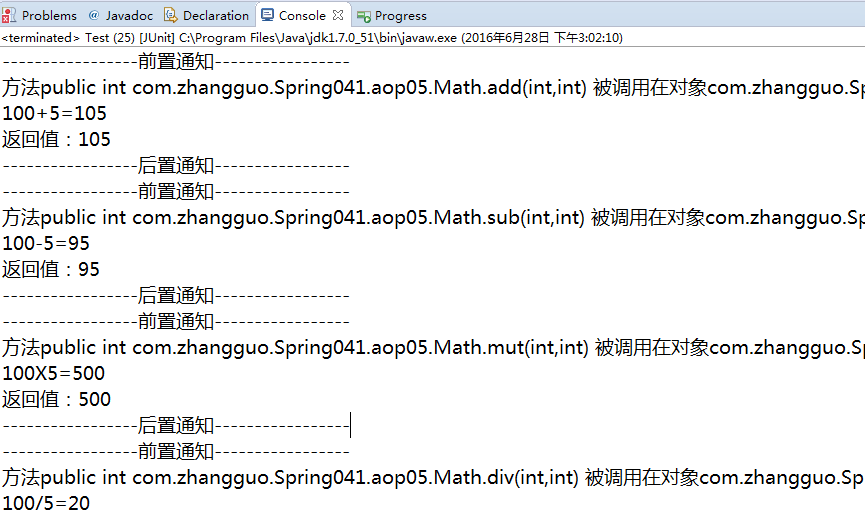
环绕通知

/\*\*  
 \* 环绕通知  
 \* 方法拦截器  
 \*/  
**public class SurroundAdvice implements** MethodInterceptor {  
  
 **public Object** invoke(MethodInvocation i) **throws Throwable** {  
 //前置横切逻辑  
 **System**.***out***.println("方法" + i.getMethod() + " 被调用在对象" + i.getThis() + "上，参数 " + i.getArguments());  
 //方法调用  
 **Object** ret = i.proceed();  
 //后置横切逻辑  
 **System**.***out***.println("返回值："+ ret);  
 **return** ret;  
 }  
}

5.3、创建代理工厂、设置被代理对象、添加通知。

**public class Test** {  
  
 @**org.**junit.Test  
 **public void** test01() {  
 //实例化Spring代理工厂  
 ProxyFactory factory = **new** ProxyFactory();  
 //设置被代理的对象  
 factory.setTarget(**new** Math());  
 //添加通知，横切逻辑  
 factory.addAdvice(**new** BeforeAdvice());  
 factory.addAdvice(**new** AfterAdvice());  
 factory.addAdvice(**new** SurroundAdvice());  
 //从代理工厂中获得代理对象  
 **IMath** math = (**IMath**) factory.getProxy();  
 **int** n1 = **100**, n2 = **5**;  
 math.add(n1, n2);  
 math.sub(n1, n2);  
 math.mut(n1, n2);  
 math.div(n1, n2);  
 }  
  
 @**org.**junit.Test  
 **public void** test02() {  
 //message.message();  
 }  
}

运行结果：



5.4、封装代理创建逻辑

 在上面的示例中如果要代理不同的对象需要反复创建ProxyFactory对象，代码会冗余。同样以实现方法耗时为示例代码如下：

5.4.1、创建一个环绕通知：

/\*\*  
 \* 用于完成计算方法执行时长的环绕通知  
 \*/  
**public class TimeSpanAdvice implements** MethodInterceptor {  
  
 **public Object** invoke(MethodInvocation invocation) **throws Throwable** {  
 // 被织入的横切内容，开始时间 before  
 **long** start = **System**.currentTimeMillis();  
 lazy();  
  
 //方法调用  
 **Object** result = invocation.proceed();  
  
 // 被织入的横切内容，结束时间  
 **Long** span = **System**.currentTimeMillis() - start;  
 **System**.***out***.println("共用时：" + span);  
  
 **return** result;  
 }  
  
 // 模拟延时  
 **public void** lazy() {  
 **try** {  
 **int** n = (**int**) **new** Random().nextInt(**500**);  
 **Thread**.sleep(n);  
 } **catch** (**InterruptedException** e) {  
 e.printStackTrace();  
 }  
 }  
}

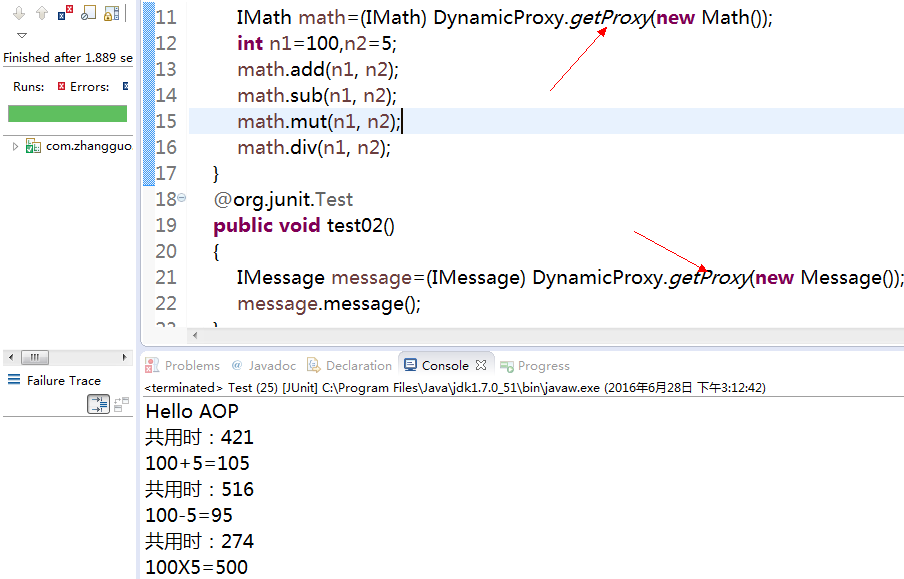
 5.4.2、封装动态代理类

/\*\*  
 \* 动态代理类  
 \*  
 \*/  
**public abstract class DynamicProxy** {  
 /\*\*  
 \* 获得代理对象  
 \* @param object 被代理的对象  
 \* @return 代理对象  
 \*/  
 **public static Object** getProxy(**Object** object){  
 //实例化Spring代理工厂  
 ProxyFactory factory=**new** ProxyFactory();  
 //设置被代理的对象  
 factory.setTarget(object);  
 //添加通知，横切逻辑  
 factory.addAdvice(**new** TimeSpanAdvice());  
 **return** factory.getProxy();  
 }  
}

5.4.3、测试运行

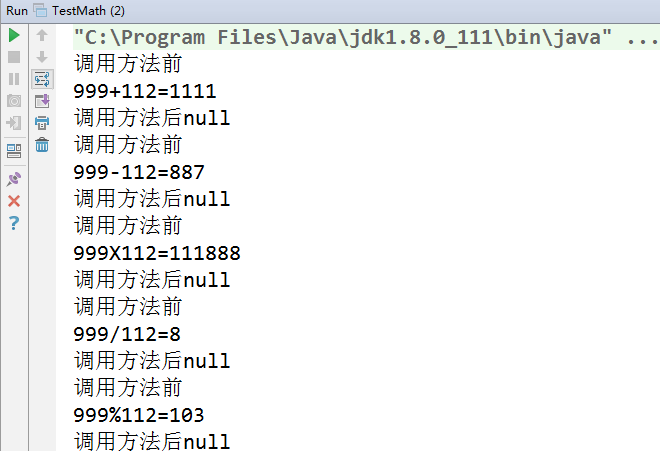
**public class Test** {  
  
 @**org.**junit.Test  
 **public void** test01()  
 {  
 //从代理工厂中获得代理对象  
 **IMath** math=(**IMath**) **DynamicProxy**.getProxy(**new** Math());  
 **int** n1=**100**,n2=**5**;  
 math.add(n1, n2);  
 math.sub(n1, n2);  
 math.mut(n1, n2);  
 math.div(n1, n2);  
 }  
 @**org.**junit.Test  
 **public void** test02()  
 {  
 IMessage message=(IMessage) **DynamicProxy**.getProxy(**new** Message());  
 message.message();  
 }  
}

运行结果：



**import org.**aopalliance.intercept.MethodInterceptor;  
**import org.**aopalliance.intercept.MethodInvocation;  
**import org.**springframework.aop.framework.ProxyFactory;  
  
**public class SpringProxy**<T> **implements** MethodInterceptor {  
  
 /\*\*获得代理后的对象\*/  
 **public** T getProxyObject(**Object** target){  
 //代理工厂  
 ProxyFactory proxy=**new** ProxyFactory();  
 //添加被代理的对象  
 proxy.setTarget(target);  
 //添加环绕通知  
 proxy.addAdvice(**this**);  
 //获得代理后的对象  
 **return** (T) proxy.getProxy();  
 }  
  
 **public Object** invoke(MethodInvocation methodInvocation) **throws Throwable** {  
 before();  
 //调用方法获得结果  
 **Object** result=methodInvocation.proceed();  
 after(result);  
 **return** result;  
 }  
  
 **public void** before(){  
 **System**.***out***.println("调用方法前");  
 }  
 **public void** after(**Object** result){  
 **System**.***out***.println("调用方法后"+result);  
 }  
}

运行结果：



#### 六、使用IOC配置的方式实现AOP

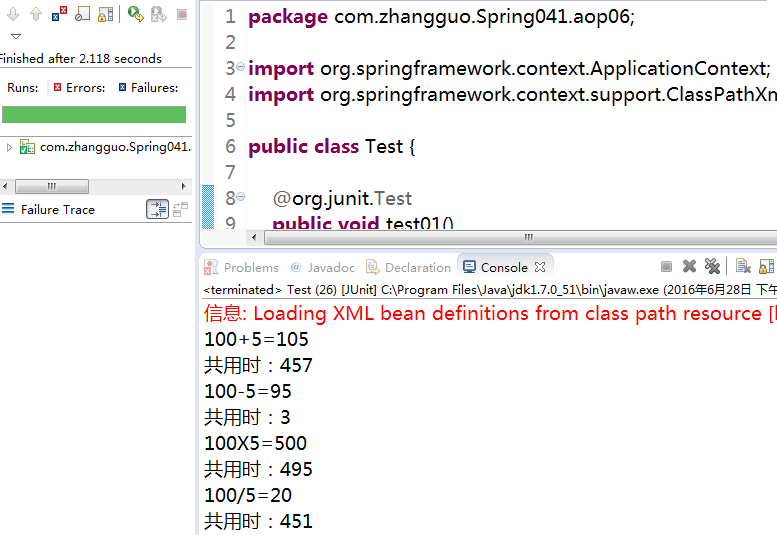
6.1、引入Spring IOC的核心jar包，方法与前面相同。

6.2、创建IOC的配置文件beans.xml，内容如下：

*<?***xml version**="1.0" **encoding**="UTF-8"*?>*<**beans xmlns**="http://www.springframework.org/schema/beans"  
 **xmlns:***xsi*="http://www.w3.org/2001/XMLSchema-instance" **xmlns:***p*="http://www.springframework.org/schema/p"  
 *xsi***:schemaLocation**="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd">  
 <!-- 被代理的目标对象 -->  
 <**bean id**="target" **class**="com.zhangguo.Spring041.aop06.Math"></**bean**>  
 <!--通知、横切逻辑-->  
 <**bean id**="advice" **class**="com.zhangguo.Spring041.aop06.AfterAdvice"></**bean**>  
 <!--代理对象 -->  
 <!--interceptorNames 通知数组 -->  
 <!--p:target-ref 被代理的对象-->  
 <!--p:proxyTargetClass 被代理对象是否为一个类，如果是则使用cglib,否则使用jdk动态代理 -->  
 <**bean id**="proxy" **class**="org.springframework.aop.framework.ProxyFactoryBean"  
 *p***:interceptorNames**="advice"  
 *p***:target-ref**="target"  
 *p***:proxyTargetClass**="true"></**bean**>  
</**beans**>

6.3、获得代理类的实例并测试运行

**public class Test** {  
 @**org.**junit.Test  
 **public void** test01()  
 {  
 //容器  
 ApplicationContext ctx=**new** ClassPathXmlApplicationContext("beans.xml");  
 //从代理工厂中获得代理对象  
 **IMath** math=(**IMath**)ctx.getBean("proxy");  
 **int** n1=**100**,n2=**5**;  
 math.add(n1, n2);  
 math.sub(n1, n2);  
 math.mut(n1, n2);  
 math.div(n1, n2);  
 }  
}



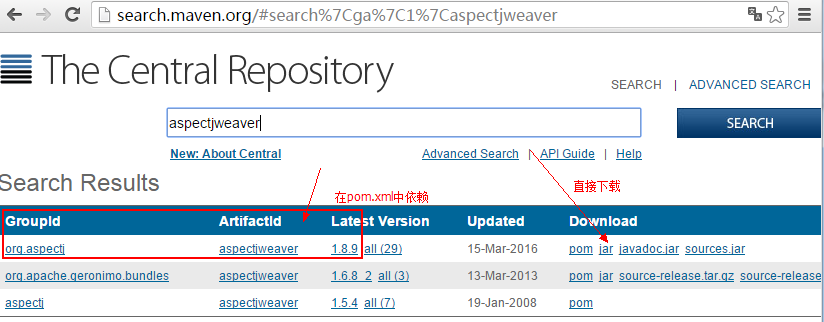
 6.4、小结

这里有个值得注意的问题：从容器中获得proxy对象时应该是org.springframework.aop.framework.ProxyFactoryBean类型的对象(如下代码所示)，但这里直接就转换成IMath类型了，这是因为：ProxyFactoryBean本质上是一个用来生产Proxy的FactoryBean。如果容器中的某个对象持有某个FactoryBean的引用它取得的不是FactoryBean本身而是 FactoryBean的getObject()方法所返回的对象。所以如果容器中某个对象依赖于ProxyFactoryBean那么它将会使用到 ProxyFactoryBean的getObject()方法所返回的代理对象这就是ProxyFactryBean得以在容器中使用的原因。

ProxyFactoryBean message = **new** ProxyFactoryBean();  
message.setTarget(**new** Message());  
message.addAdvice(**new** SurroundAdvice());  
((IMessage) message.getObject()).message();

#### 七、使用XML配置Spring AOP切面

7.1、添加引用，需要引用一个新的jar包：aspectjweaver，该包是AspectJ的组成部分。可以去http://search.maven.org搜索后下载或直接在maven项目中添加依赖。



<**project xmlns**="http://maven.apache.org/POM/4.0.0" **xmlns:***xsi*="http://www.w3.org/2001/XMLSchema-instance"  
 *xsi***:schemaLocation**="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
 <**modelVersion**>4.0.0</**modelVersion**>  
 <**groupId**>com.zhangguo</**groupId**>  
 <**artifactId**>Spring041</**artifactId**>  
 <**version**>0.0.1-SNAPSHOT</**version**>  
 <**packaging**>jar</**packaging**>  
 <**name**>Spring041</**name**>  
 <**url**>http://maven.apache.org</**url**>  
 <**properties**>  
 <**project.build.sourceEncoding**>UTF-8</**project.build.sourceEncoding**>  
 <**spring.version**>4.3.0.RELEASE</**spring.version**>  
 </**properties**>  
 <**dependencies**>  
 <**dependency**>  
 <**groupId**>junit</**groupId**>  
 <**artifactId**>junit</**artifactId**>  
 <**scope**>test</**scope**>  
 <**version**>4.10</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.springframework</**groupId**>  
 <**artifactId**>spring-context</**artifactId**>  
 <**version**>${spring.version}</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>org.aspectj</**groupId**>  
 <**artifactId**>aspectjweaver</**artifactId**>  
 <**version**>1.8.9</**version**>  
 </**dependency**>  
 <**dependency**>  
 <**groupId**>cglib</**groupId**>  
 <**artifactId**>cglib</**artifactId**>  
 <**version**>3.2.4</**version**>  
 </**dependency**>  
 </**dependencies**>  
</**project**>

7.2、定义通知

该通知不再需要实现任何接口或继承抽象类，一个普通的bean即可，方法可以带一个JoinPoint连接点参数，用于获得连接点信息，如方法名，参数，代理对象等。

/\*\*  
 \* 通知  
 \*/  
**public class Advices** {  
 //前置通知  
 **public void** before(JoinPoint jp) {  
 **System**.***out***.println("--------------------bofore--------------------");  
 **System**.***out***.println("方法名：" + jp.getSignature() + "，参数：" + jp.getArgs().length + "，代理对象：" + jp.getTarget());  
 }  
  
 //后置通知  
 **public void** after(JoinPoint jp) {  
 **System**.***out***.println("--------------------after--------------------");  
 }  
}

通知的类型有多种，有些参数会不一样，特别是环绕通知，通知类型如下：

//前置通知  
**public void** beforeMethod(JoinPoint joinPoint)  
  
//后置通知  
**public void** afterMethod(JoinPoint joinPoint)  
  
//返回值通知  
**public void** afterReturning(JoinPoint joinPoint, **Object** result)  
  
//抛出异常通知  
//在方法出现异常时会执行的代码可以访问到异常对象，可以指定在出现特定异常时在执行通知代码  
**public void** afterThrowing(JoinPoint joinPoint, **Exception** ex)  
  
//环绕通知  
//环绕通知需要携带ProceedingJoinPoint类型的参数  
//环绕通知类似于动态代理的全过程：ProceedingJoinPoint类型的参数可以决定是否执行目标方法。  
//而且环绕通知必须有返回值，返回值即为目标方法的返回值  
**public Object** aroundMethod(ProceedingJoinPoint pjd)

7.3、配置IOC容器依赖的XML文件beansOfAOP.xml

*<?***xml version**="1.0" **encoding**="UTF-8"*?>*<**beans xmlns**="http://www.springframework.org/schema/beans"  
 **xmlns:***xsi*="http://www.w3.org/2001/XMLSchema-instance"  
 **xmlns:***p*="http://www.springframework.org/schema/p"  
 **xmlns:***aop*="http://www.springframework.org/schema/aop"  
 *xsi***:schemaLocation**="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/aop  
 http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">  
  
 <!--被代理的目标对象 -->  
 <**bean id**="math" **class**="com.zhangguo.Spring041.aop08.Math"></**bean**>  
 <!-- 通知 -->  
 <**bean id**="advice" **class**="com.zhangguo.Spring041.aop08.Advices"></**bean**>  
 <!-- AOP配置 -->  
 <!-- proxy-target-class属性表示被代理的类是否为一个没有实现接口的类，Spring会依据实现了接口则使用JDK内置的动态代理，如果未实现接口则使用cblib -->  
 <*aop***:config proxy-target-class**="true">  
 <!-- 切面配置 -->  
 <!--ref表示通知对象的引用 -->  
 <*aop***:aspect ref**="advice">  
 <!-- 配置切入点(横切逻辑将注入的精确位置) -->  
 <*aop***:pointcut expression**="execution(\* com.zhangguo.Spring041.aop08.Math.\*(..))" **id**="pointcut1"/>  
 <!--声明通知，method指定通知类型，pointcut指定切点，就是该通知应该注入那些方法中 -->  
 <*aop***:before method**="before" **pointcut-ref**="pointcut1"/>  
 <*aop***:after method**="after" **pointcut-ref**="pointcut1"/>  
 </*aop***:aspect**>  
 </*aop***:config**>  
</**beans**>

加粗部分的内容是在原IOC内容中新增的，主要是为AOP服务，如果引入失败则没有智能提示。xmlns:是xml namespace的简写。xmlns:xsi：其xsd文件是xml需要遵守的规范，通过URL可以看到，是w3的统一规范，后面通过xsi:schemaLocation来定位所有的解析文件，这里只能成偶数对出现。

<bean id="advice" class="com.zhangguo.Spring041.aop08.Advices"></bean>表示通知bean，也就是横切逻辑bean。<aop:config proxy-target-class="true">用于AOP配置，proxy-target-class属性表示被代理的类是否为一个没有实现接口的类，Spring会依据实现了接口则使用JDK内置的动态代理，如果未实现接口则使用cblib；在Bean配置文件中，所有的Spring AOP配置都必须定义在<aop:config>元素内部。对于每个切面而言，都要创建一个<aop:aspect>元素来为具体的切面实现引用后端Bean实例。因此，切面Bean必须有一个标识符，供<aop:aspect>元素引用。

aop:aspect表示切面配置， ref表示通知对象的引用；aop:pointcut是配置切入点，就是横切逻辑将注入的精确位置，那些包，类，方法需要拦截注入横切逻辑。

aop:before用于声明通知，method指定通知类型，pointcut指定切点，就是该通知应该注入那些方法中。在aop Schema中，每种通知类型都对应一个特定地XML元素。通知元素需要pointcut-ref属性来引用切入点，或者用pointcut属性直接嵌入切入点表达式。method属性指定切面类中通知方法的名称。有如下几种：

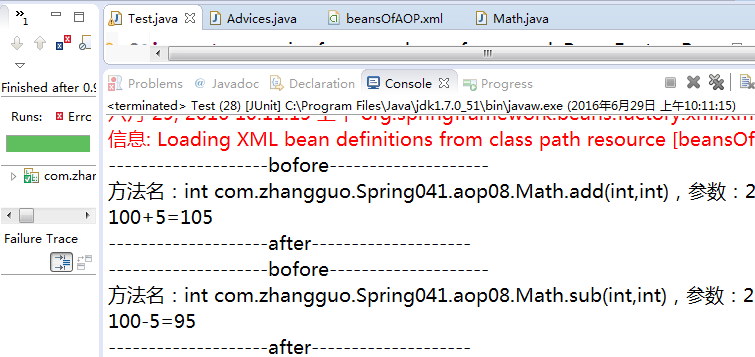
<!-- 前置通知 -->  
<*aop***:before method**="before" **pointcut-ref**="pointcut1"/>  
 <!-- 后置通知 -->  
<*aop***:after method**="after" **pointcut-ref**="pointcut1"/>  
 <!--环绕通知 -->  
<*aop***:around method**="around" **pointcut**="execution(\* com.zhangguo.Spring041.aop08.Math.s\*(..))"/>  
 <!--异常通知 -->  
<*aop***:after-throwing method**="afterThrowing" **pointcut**="execution(\* com.zhangguo.Spring041.aop08.Math.d\*(..))" **throwing**="exp"/>  
 <!-- 返回值通知 -->  
<*aop***:after-returning method**="afterReturning" **pointcut**="execution(\* com.zhangguo.Spring041.aop08.Math.m\*(..))" **returning**="result"/>

关于execution请查看另一篇文章：

7.4、获得代理对象

**public class Test** {  
 @**org.**junit.Test  
 **public void** test01() {  
 //容器  
 ApplicationContext ctx=**new** ClassPathXmlApplicationContext("beansOfAOP.xml");  
 //从代理工厂中获得代理对象  
 **IMath** math=(**IMath**)ctx.getBean("math");  
 **int** n1=**100**,n2=**5**;  
 math.add(n1, n2);  
 math.sub(n1, n2);  
 math.mut(n1, n2);  
 math.div(n1, n2);  
 }  
}

7.5、测试运行



7.6、环绕通知、异常后通知、返回结果后通知

在配置中我们发现共有5种类型的通知，前面我们试过了前置通知与后置通知，另外几种类型的通知如下代码所示：

/\*\*  
 \* 通知  
 \*/  
**public class Advices** {  
 //前置通知  
 **public void** before(JoinPoint jp) {  
 **System**.***out***.println("--------------------前置通知--------------------");  
 **System**.***out***.println("方法名：" + jp.getSignature().getName() + "，参数：" + jp.getArgs().length + "，被代理对象：" + jp.getTarget().getClass().getName());  
 }  
  
 //后置通知  
 **public void** after(JoinPoint jp) {  
 **System**.***out***.println("--------------------后置通知--------------------");  
 }  
  
 //环绕通知  
 **public Object** around(ProceedingJoinPoint pjd) **throws Throwable** {  
 **System**.***out***.println("--------------------环绕开始--------------------");  
 **Object** object = pjd.proceed();  
 **System**.***out***.println("--------------------环绕结束--------------------");  
 **return** object;  
 }  
  
 //异常后通知  
 **public void** afterThrowing(JoinPoint jp, **Exception** exp) {  
 **System**.***out***.println("--------------------异常后通知，发生了异常：" + exp.getMessage() + "--------------------");  
 }  
  
 //返回结果后通知  
 **public void** afterReturning(JoinPoint joinPoint, **Object** result) {  
 **System**.***out***.println("--------------------返回结果后通知--------------------");  
 **System**.***out***.println("结果是：" + result);  
 }  
}

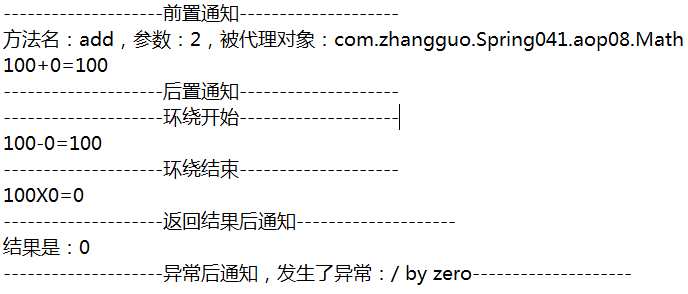
容器配置文件beansOfAOP.xml如下：

*<?***xml version**="1.0" **encoding**="UTF-8"*?>*<**beans xmlns**="http://www.springframework.org/schema/beans"  
 **xmlns:***xsi*="http://www.w3.org/2001/XMLSchema-instance"  
 **xmlns:***p*="http://www.springframework.org/schema/p"  
 **xmlns:***aop*="http://www.springframework.org/schema/aop"  
 *xsi***:schemaLocation**="http://www.springframework.org/schema/beans  
 http://www.springframework.org/schema/beans/spring-beans.xsd  
 http://www.springframework.org/schema/aop  
 http://www.springframework.org/schema/aop/spring-aop-4.3.xsd">  
  
 <!--被代理的目标对象 -->  
 <**bean id**="math" **class**="com.zhangguo.Spring041.aop08.Math"></**bean**>  
 <!-- 通知 -->  
 <**bean id**="advice" **class**="com.zhangguo.Spring041.aop08.Advices"></**bean**>  
 <!-- AOP配置 -->  
 <!-- proxy-target-class属性表示被代理的类是否为一个没有实现接口的类，Spring会依据实现了接口则使用JDK内置的动态代理，如果未实现接口则使用cblib -->  
 <*aop***:config proxy-target-class**="true">  
 <!-- 切面配置 -->  
 <!--ref表示通知对象的引用 -->  
 <*aop***:aspect ref**="advice">  
 <!-- 配置切入点(横切逻辑将注入的精确位置) -->  
 <*aop***:pointcut expression**="execution(\* com.zhangguo.Spring041.aop08.Math.a\*(..))" **id**="pointcut1"/>  
 <!--声明通知，method指定通知类型，pointcut指定切点，就是该通知应该注入那些方法中 -->  
 <*aop***:before method**="before" **pointcut-ref**="pointcut1"/>  
 <*aop***:after method**="after" **pointcut-ref**="pointcut1"/>  
 <*aop***:around method**="around" **pointcut**="execution(\* com.zhangguo.Spring041.aop08.Math.s\*(..))"/>  
 <*aop***:after-throwing method**="afterThrowing" **pointcut**="execution(\* com.zhangguo.Spring041.aop08.Math.d\*(..))"  
 **throwing**="exp"/>  
 <*aop***:after-returning method**="afterReturning"  
 **pointcut**="execution(\* com.zhangguo.Spring041.aop08.Math.m\*(..))" **returning**="result"/>  
 </*aop***:aspect**>  
 </*aop***:config**>  
</**beans**>

测试代码：

**public class Test** {  
  
 @**org.**junit.Test  
 **public void** test01() {  
 //容器  
 ApplicationContext ctx = **new** ClassPathXmlApplicationContext("beansOfAOP.xml");  
 //从代理工厂中获得代理对象  
 **IMath** math = (**IMath**) ctx.getBean("math");  
 **int** n1 = **100**, n2 = **0**;  
 math.add(n1, n2);  
 math.sub(n1, n2);  
 math.mut(n1, n2);  
 math.div(n1, n2);  
 }  
}

运行结果：



小结：不同类型的通知参数可能不相同；aop:after-throwing需要指定通知中参数的名称throwing="exp"，则方法中定义应该是这样：afterThrowing(JoinPoint jp,Exception exp)；aop:after-returning同样需要设置returning指定方法参数的名称。通过配置切面的方法使AOP变得更加灵活。

#### 八、示例下载

<https://git.coding.net/zhangguo5/Spring.git>

[点击下载](http://files.cnblogs.com/files/best/Spring041.rar)

#### 九、视频

<https://www.bilibili.com/video/av16071354/>

# 三.行为型模式

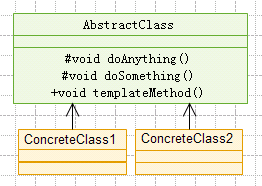
## 1.Template模板模式

模板方法模式：定义一个算法中的操作框架，而将一些步骤延迟到子类中。使得子类可以不改变算法的结构即可重定义该算法的某些特定步骤。

（Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure）

模板方法模式的通用类图非常简单，仅仅使用了Java的继承机制，但它是一个非常广泛的模式。其类图如下，其中AbstractClass叫做抽象模板，它的方法分为两类：

* 基本方法：是由子类实现的方法，并且在模板方法被调用。（一般都加上final关键字，防止被覆写）
* 模板方法：可以有一个或几个，一般是一个具体方法，也就是一个框架，实现对基本方法的调用，完成固定的逻辑。（抽象模板中的基本方法尽量设计为protected类型，符合迪米特法则，不需要暴露的属性或方法尽量不要设置为protected类型。实现类若非必要，尽量不要扩大父类中的访问权限）



**public abstract class AbstractClass** {  
 **protected abstract void** doAnything();  
 **protected abstract void** doSomething();  
 **public final void** templateMethod(){  
 */\*  
 \* 调用基本方法，完成相关的逻辑  
 \*/* **this**.doAnything();  
 **this**.doSomething();  
 }  
}  
**public class ConcreteClass1 extends AbstractClass** {  
 @Override  
 **protected void** doAnything() {  
 // *TODO Auto-generated method stub* //子类实现具体  
 }  
  
 @Override  
 **protected void** doSomething() {  
 // *TODO Auto-generated method stub* }  
}  
**public class ConcreteClass2 extends AbstractClass** {  
 @Override  
 **protected void** doAnything() {  
 // *TODO Auto-generated method stub* //子类实现具体  
 }  
  
 @Override  
 **protected void** doSomething() {  
 // *TODO Auto-generated method stub* }  
}

### 模板方法模式的优点

封装不变部分，扩展可变部分。把认为不变部分的算法封装到父类中实现，而可变部分的则可以通过继承来继续扩展。

提取公共部分代码，便于维护。

行为由父类控制，子类实现。

### 模板方法模式的缺点

按照设计习惯，抽象类负责声明最抽象、最一般的事物属性和方法，实现类负责完成具体的事务属性和方法，但是模板方式正好相反，子类执行的结果影响了父类的结果，会增加代码阅读的难度。

### 模板方法模式的使用场景

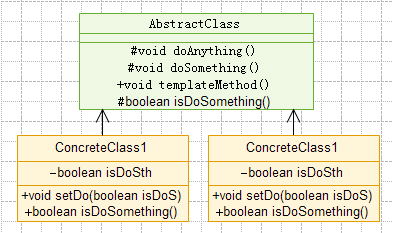
多个子类有共有的方法，并且逻辑基本相同

重要、复杂的算法，可以把核心算法设计为模板方法，周边的相关细节功能则由各个子类实现

重构时，模板方法是一个经常使用的方法，把相同的代码抽取到父类中，然后通过构造函数约束其行为。

模板方法模式的扩展

如上通用类图中，若在ConcreteClass2中不想执行doSomething()方法,那么需要对模板方法做一些修改，其通用类图如下：



**public abstract class AbstractClass** {  
 **protected abstract void** doAnything();  
 **protected abstract void** doSomething();  
 **protected boolean** isDoSomething(){ //父类方法返回真  
 **return true**;  
 }  
 **public final void** templateMethod(){  
 */\*  
 \* 调用基本方法，完成相关的逻辑  
 \*/* **this**.doAnything();  
 **if**(**this**.isDoSomething())  
 **this**.doSomething();  
 }  
}

**public class ConcreteClass1 extends AbstractClass** {  
 **private boolean** *isDoSth*;  
 @Override  
 **protected void** doAnything() {  
 // *TODO Auto-generated method stub* //子类实现具体  
 }  
  
 @Override  
 **protected void** doSomething() {  
 // *TODO Auto-generated method stub* }  
 **protected void** setDo(**boolean** isDo){  
 **this**.*isDoSth* = isDo;  
 }  
 **protected boolean** isDoSomething(){  
 **return** *isDoSth*;  
 }  
}

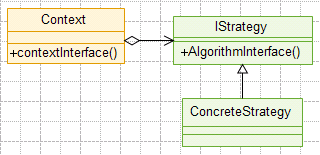
### 最佳实践

模板方法模式是通过父类建立框架，子类在重写了父类部分方法之后，在调用从父类继承的方法，产生不同的效果，通过修改子类，影响父类行为的结果，模板方法在一些开源框架中应用非常多，它提供了一个抽象类，然后开源框架写了一堆子类，如果需要扩展功能，可以继承此抽象类，然后覆写protected基本方法，然后在调用一个类似TemplateMethod()的模板方法，完成扩展开发。

## 2.Strategy策略模式

策略模式：也叫作政策模式，定义一组算法，将每个算法都封装起来，并且使他们之间可以互换

策略模式的使用就是面向对象的继承和多态机制，其通用类图如下：



Context封装角色，也叫作上下文角色，屏蔽高层模块对策略、算法的直接访问，封装可能的变化。

Istrategy抽象策略角色，策略算法家族的抽象，通常为接口定义每个算法必须具有的方法和属性。

ConcreteStrategy具体策略角色，实现抽象策略中的操作该类含有具体的算法。

通用类图的源码如下：

**public interface IStrategy** {  
 //定义具体策略必须具有的方法  
 **public void** doSomething();  
}

**public class ConcreteStrategy1 implements IStrategy**{  
 **public void** doSomething(){  
 **System**.***out***.println("this is a concreteStrategy");  
 }  
}

**public class ConcreteStrategy2 implements IStrategy**{  
 @Override  
 **public void** doSomething() {  
 // *TODO Auto-generated method stub* **System**.***out***.println("This is concreteStrategy2");  
 }  
}

**public class Context** {  
 **private IStrategy** *strategy* = **null**;  
 **public Context**(**IStrategy** strategy){  
 **this**.*strategy* = strategy;  
 }  
 **public void** doSomething(){  
 **this**.*strategy*.doSomething();  
 }  
}

**public class Client** {  
 **public static void** main(**String**[] args) {  
 // *TODO Auto-generated method stub* **Context** text = **new** Context(**new** ConcreteStrategy2());  
 text.doSomething();  
 text = **new** Context(**new** ConcreteStrategy1());  
 text.doSomething();  
 }  
}

### 策略模式的优点

算法可以自由的切换，通过实现抽象策略，通过封装角色对其封装，保证对外提供“可自由切换”的策略。

避免使用多重条件判断，如果有多重策略，那么每个策略只需实现自己的方法，至于采用何种策略，可以通过其他模块决定。

扩展性良好，可以在现有的系统中任意的加入新的策略，只需继承IStrategy接口，符合OCP原则。

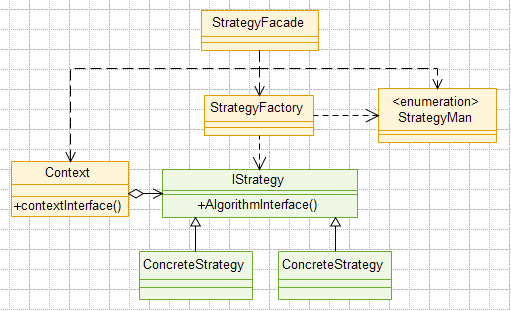
### 策略模式的缺点

策略类数量增多，每个策略都是一个类，复用的可能性很小，类数量增多

所有的策略都需要对外暴露，上层模块必须知道有哪些策略，然后才能知道采用哪种策略，可以通过使用工厂方法模式、代理模式和享元模式修正。

### 策略模式的扩展（工厂方法模式+策略模式）

    首先策略枚举StrategyMan，负责对具体策略的映射，然后建立一个简单的工厂，根据策略管理类的枚举项创建一个策略对象，简单而实用，策略模式的缺陷得到了弥补。其通用类图如下：



**public enum StrategyMan** {  
 ***Strategy1***("com.strategy.ConcreteStrategy1"),  
 ***Strategy2***("com.strategy.ConcreteStrategy2");  
 **String** *value* = "";  
 **private StrategyMan**(**String** value){  
 **this**.*value* = value;  
 }  
 **public String** getValue(){  
 **return this**.*value*;  
 }  
}  
  
**public class StrategyFactory** {  
 **public static** IStrategy getStrategy(**StrategyMan** strategyman){  
 IStrategy strategy = **null**;  
 **try** {  
 strategy = (IStrategy) **Class**.forName(strategyman.getValue()).newInstance();  
 } **catch** (**InstantiationException** | **IllegalAccessException** | **ClassNotFoundException** e) {  
 // *TODO Auto-generated catch block* e.printStackTrace();  
 }  
 **return** strategy;  
 }  
}  
  
**public class StrategyFacade** {  
 **public void** doSomething(**String** str){  
 **StrategyMan** sMan = **null**;  
 **if**(str.equalsIgnoreCase("Strategy1")){  
 sMan = **StrategyMan**.***Strategy1***;  
 }**else if**(str.equalsIgnoreCase("Strategy2")){  
 sMan = **StrategyMan**.***Strategy2***;  
 }  
 IStrategy strategy = **StrategyFactory**.getStrategy(sMan);  
 Context context = **new** Context(strategy);  
 context.doSomething();  
 }  
}  
  
**public class Client** {  
  
 /\*\*  
 \* @param args  
 \*/  
 **public static void** main(**String**[] args) {  
 // *TODO Auto-generated method stub* **StrategyFacade** sFacade = **new** StrategyFacade();  
 sFacade.doSomething("Strategy1");  
 }  
}