# 单例设计模式

## 定义与使用场景

1、在内存中一个类有且仅有一个对象实例

2、由于单例设计模式只有一个对象实例，避免产生多个对象造成的资源浪费问题。

3、如果一个对象需要消耗较多的资源或初始化较慢的，可以通过单例设计模式产生一个常驻内存的方式来实现。(IO操作、数据库操作、配置读取操作等)

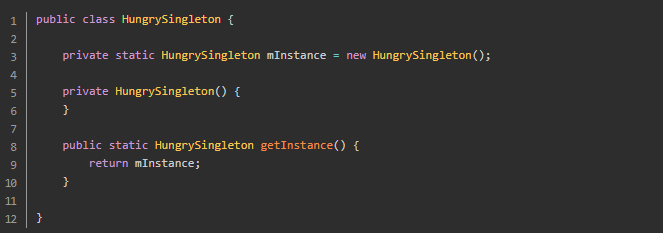
## 实现方式

1.私有构造方法

2.提供static属性

3.提供一个获取实例的方法 getInstance()

## 饿汉式



1、在类初始化时就立即加载了对象，不管之后是不是需要使用到它，没有懒加载的优势。

2、由于在类加载时，是一个天然的线程安全的模式，所以避免了多线程并发创建对象的问题

3、由于没有使用同步锁，所以并发调用效率较高

为什么是线程安全的?

首先，singleton 作为类成员变量的实例化发生在类Singleton 类加载的初始化阶段，初始化阶段是执行类构造器<clinit>() 方法的过程。

<clinit>() 方法是由编译器自动收集类中的所有类变量（static）的赋值动作和静态语句块（static{}）块中的语句合并产生的。因此，private static Singleton singleton = new Singleton();也会被放入到这个方法中。

虚拟机会保证一个类的<clinit>()方法在多线程环境中被正确的加锁、同步，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的<clinit>()方法，其他线程都需要阻塞等待，直到活动线程执行<clinit>()方法完毕。需要注意的是，其他线程虽然会被阻塞，但如果执行<clinit>()方法的那条线程退出<clinit>()方法后，其他线程唤醒后不会再次进入<clinit>()方法。同一个类加载器下，一个类型只会初始化一次。

## 懒汉式

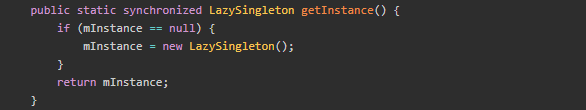
|  |
| --- |
| **public class** LazySingleton {   **private static** LazySingleton *lazySingleton* ;  **private** LazySingleton () {  }  **public static** LazySingleton getInstance(){  **if**(!Optional.*ofNullable*(*lazySingleton*).isPresent()){  **return** *lazySingleton* = **new** LazySingleton();  }  **return** *lazySingleton* ;  } } |

1、在类初始化时不初始化对象，在真正需要使用的时候再创建(懒加载)

2、如果在单线程时 synchronized 关键字可以不加，但是在多线程并发的情况下就必须通过 synchronized 关键字来保证生成有且一个实例

3、由于使用synchronized 关键字导致并发的时候效率不高

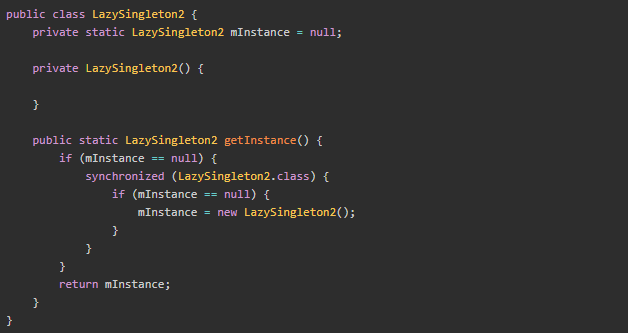
解决方法1、：添加synchronized关键字



添加synchronized关键字，但是效率比较低，解决懒加载问题

解决方法二 double check

解决问题2：使用double check，解决效率问题，实现单例。但是可能会造成空指针问题。对象在初始化工程中，可能值还没有初始化完成，就切换得到另一个线程，判断部位null，但是会造成空



首先我们分析一下为什么需要两次判空

第一层判空主要是为了避免不必要的同步

第二层判空主要是为了确保在mInstance = null 情况下进行实例化。举例: 假设A、B线程同时走到 synchronized 进行同步，将B线程挂起后，A线程通过 if (mInstance == null) 判断，进行对象的实例化操作。A线程完成后，B线程就开始进判断 if (mInstance == null) 这时 mInstance 已经被实例化了，所以跳出了同步代码块。

线程安全问题：

由于 mInstance = new LazySingleton2() 这句代码看起来虽然只有一句，但是它并不是原子操作

1.为mInstance分配内存

2.使用LazySingleton()的构造方法初始化成员变量

3.将mInstance对象指向分配的内存空间（执行完mInstance就为不等于null）

编译器和处理器会对指令进行重排序。正常的顺应该为1->2->3，但是重排序之后，可能会将顺序变为1->3->2。我们将注意力移到第一层判空 if (mInstance == null) ，在判空的时候如果指令顺序以1->3->2的顺序执行，并且执行了3还未执行2时，就出现了 mInstance 不等于 null 的情况，就直接将mInstance返回并拿去使用，但这时mInstance并未完成初始化的操作，使用的一定会存在问题。总得来说就是避免其它线程可能会看到一个未完全初始化的对象实例。

### 使用volatile解决

volatile关键字

1.可以提供线程共享变量的可见性（体现为并发编程的可见性）

2.禁止指令重排序（体现为并发编程的有序性）

|  |
| --- |
| **public class** SingletonObject5 {  **private static volatile** SingletonObject5 *instance*;  **private** SingletonObject5(){  }   **public static** SingletonObject5 getInstance(){  **if**(**null**==*instance*){  **synchronized** (SingletonObject5.**class**){  **if**(**null**==*instance*){  *instance* = **new** SingletonObject5();  }  }  }  **return** *instance*;  } } |

## .静态内部类实现单例

|  |
| --- |
| **public class** SingletonObject6 {  **private** SingletonObject6(){  }  **private static class** InstacenHolder{  **private final static** SingletonObject6 ***instance*** = **new** SingletonObject6();  }  **public static** SingletonObject6 getInstance(){  **return** InstacenHolder.***instance***;  } } |

1、只有调用了getInstance()才会加载静态内部类（懒加载）

2、类在加载的时候是天然线程安全的，保证内存只有一个对象实例

3、不存在同步锁，并发调用效率高

## 使用枚举方式

|  |
| --- |
| **public class** SingletonObject7 {   **private** SingletonObject7(){  }   **private enum** Singleton{  ***INSTACE***;   **private final** SingletonObject7 **instace**;   Singleton(){  **instace** = **new** SingletonObject7();  }  **public** SingletonObject7 getInstance(){  **return instace**;  }   }  **private static class** InstacenHolder{  **private final static** SingletonObject7 ***instance*** = **new** SingletonObject7();  }   **public static** SingletonObject7 getInstance(){  **return** InstacenHolder.***instance***;  } } |

为什么该方式是线程安全的？

枚举反编译之后，都是static类型的，因为static类型的属性会在类被加载之后被初始，当一个Java类第一次被真正使用到的时候静态资源被初始化、Java类的加载和初始化过程都是线程安全的。所以，创建一个enum类型是线程安全的。

# 工厂模式

## 简单工厂模式

### 简单工厂模式

简单工厂模式属于**创建型模式**，又叫做**静态工厂方法模式**，不属于23种GOF设计模式之一。是由一个工厂对象决定创建出哪一种产品类的实例。实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类（这些产品类继承自一个父类或接口）的实例。

### 作用

作用：将“类实例化的操作”与“使用对象的操作”分开，让使用者不用知道具体参数就可以实例化出所需要的“产品”类，从而避免了在客户端代码中显式指定，实现了解耦。

### 角色

工厂：负责实现创建所有实例的内部逻辑，并提供一个外界调用的方法，创建所需的产品对象。

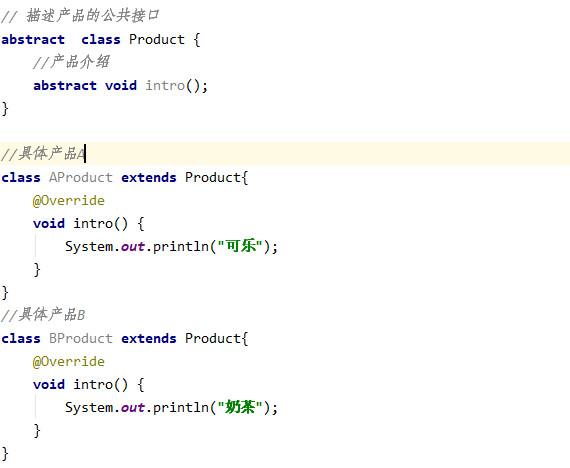
抽象产品：负责描述产品的公共接口

具体产品：描述生产的具体产品。

举个简单易懂的例子：

“假设”有一台饮料机（工厂），可以调出各种口味的饮料（抽象产品），有三个按钮（参数）对应这三种饮料（具体产品）。这时候你可以根据点击按钮来选择你喜欢的饮料。

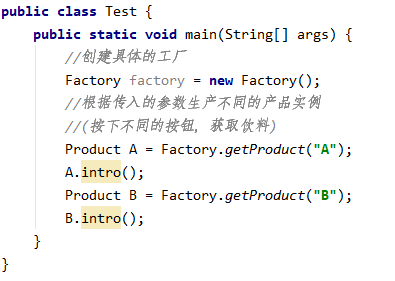
### 实现



工厂类 Factory.java



测试



## 工厂方法模式

### 工厂方法模式

又称工厂模式、多态工厂模式和虚拟构造器模式，通过定义工厂父类负责定义创建对象的公共接口，而子类则负责生成具体的对象。一种常用的对象创建型设计模式,此模式的核心精神是封装 类中不变的部分。

作用：将类的实例化（具体产品的创建）延迟到工厂类的子类（具体工厂）中完成，即由子类来决定应该实例化（创建）哪一个类。

### 角色

主要角色

抽象工厂：描述具体工厂的公共接口

具体工厂：描述具体工厂，创建产品的实例，供外界调用

抽象产品：负责描述产品的公共接口

具体产品：描述生产的具体产品

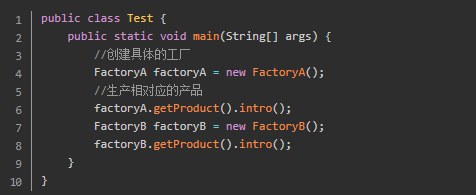
### 实现



工厂的实现



测试



### 总结

根据例子可以描述为：一个抽象产品类，可以派生出多个具体产品类。一个抽象工厂类，可以派生出多个具体工厂类。每个具体工厂类只能创建一个具体产品类的实例。

优点：

符合开-闭原则：新增一种产品时，只需要增加相应的具体产品类和相应的工厂子类即可

符合单一职责原则：每个具体工厂类只负责创建对应的产品

缺点：

加了系统的复杂度：类的个数将成对增加

加了系统的抽象性和理解难度

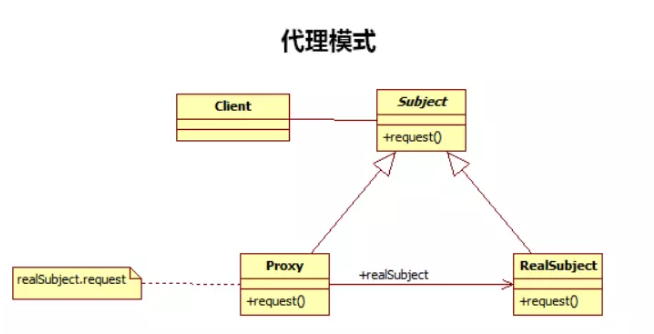
个具体工厂只能创建一种具体产品

# 代理模式

## 什么是代理模式

抽象点说是一个类代表另一个类的功能，或者说一个对象为另一个对象提供一个代理或者占位符以控制对这个对象的访问。同样我也会举例子来说明，这里我举一个买车票的例子。通常我们我们买车票需要去车站买，但是这样会很麻烦，可能要坐很久的车去车站，然后在排队买票。但是如果我们去一个卖车票的代理点买车票可能就会省去这么多的事情。这样车票售卖处就代理你购买车票。

## 代理模式结构



## 应用

下面我们就用具体的代码来实现上面这个买车票的静态代理。

首先创建一个售票服务的接口，它有售票咨询和退票的业务可以供客户选择。



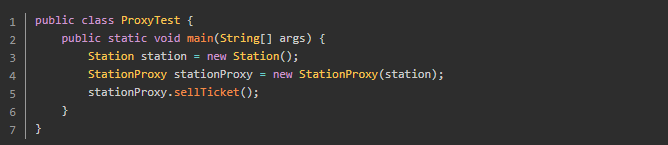
然后创建一个售票服务接口实现类，就好比是车站。

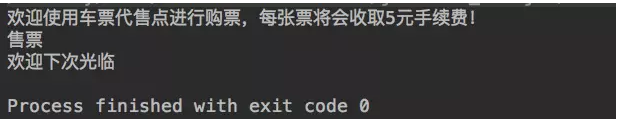


然后创建一个代理售票点



创建购买车票的角色,去代理点完成购买车票的需求





从结果看到售票代理点成功帮助客户购买到车票，节省了客户去车站排队等待的时间和精力。代理模式有点像是委派模式中的中介，前面的文章也提到过静态代理和策略模式是委派模式的一种组合。那当然除了静态代理还有动态代理和CGLIB代理，感兴趣的伙伴可以自己去研究研究。

# 模板方法模式(Template Method)

## 定义

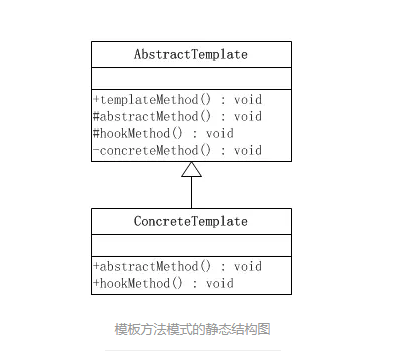
模板方法模式是类的行为模式。准备一个抽象类，将部分逻辑以具体方法以及具体构造函数的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。这就是模板方法模式的用意。

## 模板方法模式的结构

模板方法模式是所有模式中最为常见的几个模式之一，是基于继承的代码服用的基本技术。

模板方法模式需要开发抽象类和具体子类的设计师之间的协作。一个设计师负责给出一个算法的轮廓和骨架，另一些设计师则负责给出这个算法的各个逻辑步骤。代表这些具体逻辑步骤的方法称作基本方法(primitive method)； 而将这些基本方法汇总起来的方法叫做模板方法(template method)，这个设计模式的名字就是由此而来。

模板方法所代表的行为称为顶级行为，其逻辑称为顶级逻辑。模板方法模式的静态结构图如下所示：



这里涉及到两个角色：

抽象模板(Abstract Template)角色，具有如下责任：

1、定义了一个或多个抽象操作以便让子类实现。这些抽象操作叫做基本操作。他们是一个顶级逻辑的组成结构。

2、定义并实现了一个模板方法。这个模板方法一般是一个具体方法，它给出了一个顶级逻辑的骨架，而逻辑的组成步骤在相应的抽象操作中，推迟到子类实现。顶级逻辑也有可能调用一些具体方法。

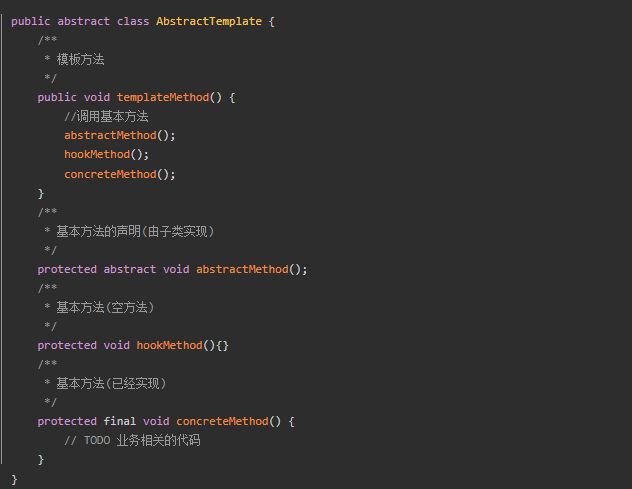
具体模板(Concrete Template)角色，具有如下责任：

1、实现父类所定义的一个或多个抽象方法，他们是一个顶级逻辑的组成步骤。

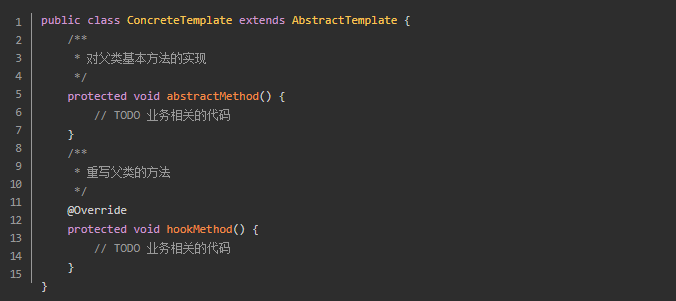
2、每一个抽象模板角色都可以有任意多个具体模板角色与之对应，而每一个具体模板角色都可以给出这些抽象方法(也就是顶级逻辑的组成步骤)的不同实现，从而使得顶级逻辑的实现各不相同。

示例代码:

抽象模板角色类，abstractMethod()、hookMethod()等基本方法是顶级逻辑的组成步骤，这个顶级逻辑是由templateMethod()方法代表。



具体模板角色类，实现了父类所声明的基本方法，abstractMethod()方法代表的就是强制子类实现的剩余逻辑，而hookMethod()方法是可以选择实现的逻辑，不是必须实现的。



模板方法模式的关键是：**子类可以置换掉父类的可变部分，但是子类却不可以改变模板方法所代表的顶级逻辑。**

每当定义一个新的子类时，不要按照控制流程的思路去想，而应当按照“责任”的思路去想。换而言之，应当考虑哪些操作是必需置换掉的，哪些操作是可以置换掉的，以及哪些操作时不可以置换掉的。使用模板模式可以使这些责任变得清晰。

## 模板方法模式中的方法

模板方法模式中的方法可以分为两大类：**模板方法**和**基本方法**

### **模板方法** 一个模板方法是定义在抽象类中的，把基本操作方法组合在一起形成一个总算法或者一个总行为的方法。

一个抽象类可以有任意多个模板方法，而不限于一个。每一个模板方法都可以调用任意多个具体方法。

### 基本方法

基本方法又可以分为三类：抽象方法(Abstract Method)、具体方法(Concrete Method)和钩子方法(Hook Method)。

抽象方法：一个抽象方法由抽象类声明，由具体子类实现。在Java语言里抽象方法以Abstract关键字标示。

具体方法：一个具体方法由抽象类声明并实现，而子类并不实现或置换。

钩子方法：一个钩子方法由抽象类声明并实现，而子类会加以拓展。通常抽象类给出的实现是一个空实现，作为方法的默认实现。

在上面的例子中，AbstractTemplate是一个抽象类，她有三个方法，其中abstractMethod()是一个抽象方法，它由抽象类声明为抽象方法，并由子类实现；hookMethod()是一个钩子方法，它由抽象类声明并提供默认实现，并且由子类置换掉；concreteMethod()是一个具体方法，它由抽象类声明并实现。

### 默认钩子方法

一个钩子方法常常由抽象类给出一个空实现作为此方法的默认实现。这种空的钩子方法叫做“Do Nothing Hook”。显然，这种默认钩子方法在缺省适配模式里面已经见过了，一个缺省适配模式讲的是一个类为一个接口提供一个默认的空实现，从而使得缺省适配类的子类不必像实现接口那样必须给出所有方法的实现，因为通常一个具体类并不需要所有的方法。

### 命名规则

命名规则是设计师之间可以沟通的管道之一，使用恰当的命名规则可以帮助不同设计师之间的沟通。

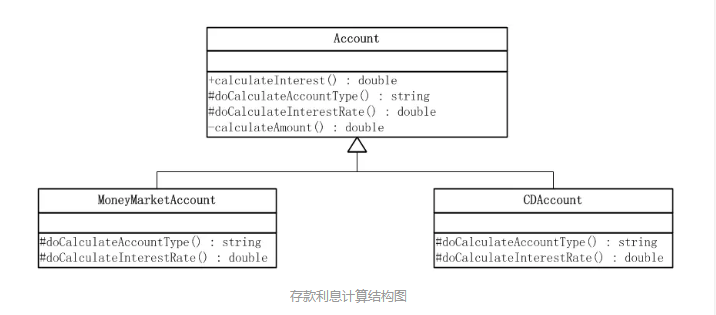
钩子方法的命名应当以do开始，这是熟悉设计模式的Java开发人员的标准做法。在上面的例子中，钩子方法hookMethod()应当以do开头；在HttpServlet类中，也遵从这一命名规则，例如doGet()、doPost()等方法。

## 使用场景

考虑一个计算存款利息的例子。假设系统需要支持两种存款账号，即货币市场(Money Market)账号和定期存款(Certificate of Deposite)账号。这两种账号的存款利息是不同的，因此，在计算 一个存户的存款利息额时，必须区分两种不同的账号类型。

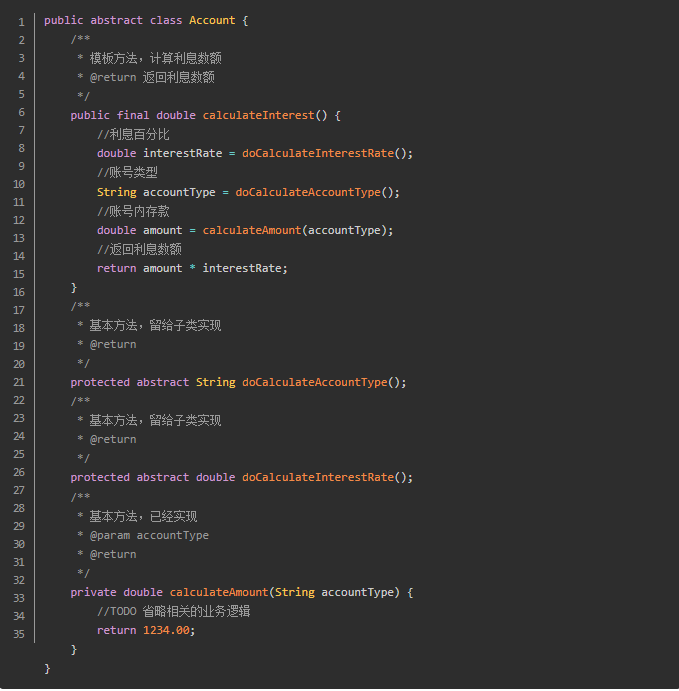
这个系统的总行为应当是计算出利息，这就决定了作为一个模板方法模式的顶级逻辑应当是利息计算。由于利息计算涉及到两个步骤： 一个基本方法给出账号种类，另一个基本方法给出利息百分比。这两个基本方法构成具体逻辑，因为账号的类型不同，所以具体逻辑会有所不同。

显然，系统需要一个抽象角色给出顶级行为的实现，而将两个作为细节步骤的基本方法留给具体子类实现。由于需要考虑的账号有两种：一种是货币市场账号，另一种是定期存款账号。系统的类结构如下图所示：

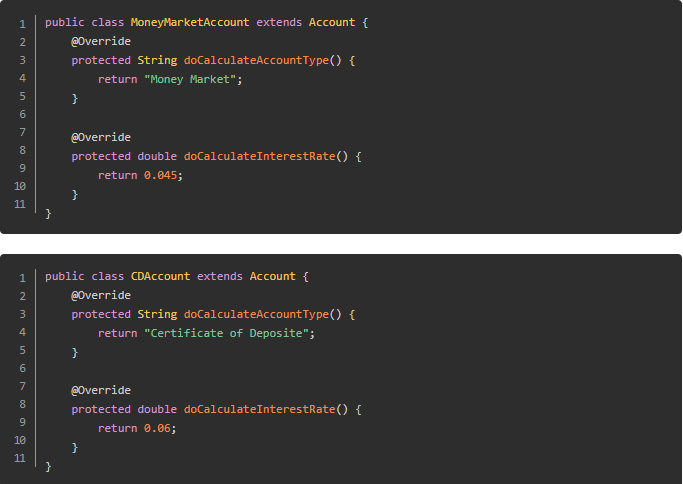


### 示例代码

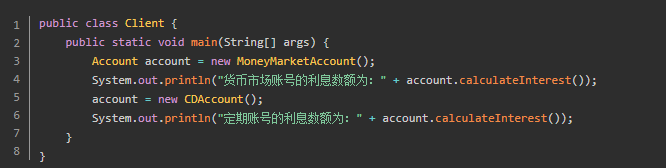
抽象模板角色类



### 具体模板角色类



客户端类



## 模板方法模式在Servlet中的应用

使用过Servlet的人都清楚，除了要在web.xml做相应的配置之外，还需要继承一个叫做HttpServlet的抽象类。HttpServlet类中，提供了一个service()方法，这个方法调用七个do方法中的一个或几个，完成对于客户端调用的响应。这些do方法需要由HttpServlet的具体子类提供，因此这是典型的模板方法模式。下面是service()方法的源代码：

# 观察者模式

## 观察者模式

观察者模式（Observer Pattern），又叫 发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式 或 从属者（Dependents）模式。

观察者模式 是对象的行为模式，定义了一种一对多的依赖关系，一个主题对象可被多个观察者对象同时监听，使得每当主题对象状态变化时，所有依赖于它的对象都会得到通知并被自动更新。

观察者模式 核心：松耦合观察者与被观察者，以类似于消息/广播发送的机制联动两者，使被观察者的变动能通知到感兴趣的观察者们，从而做出相应的响应。

### 主要解决

当系统一方行为依赖于另一方行为的变动时，可使用 观察者模式 松耦合联动双方，使得一方的变动可以通知到感兴趣的另一方对象，从而让另一方对象对此做出响应。

### 优缺点

优点

观察者和被观察者是松耦合（抽象耦合）的，符合 依赖倒置原则；

分离了表示层（观察者）和数据逻辑层（被观察者），并且建立了一套触发机制，使得数据的变化可以响应到多个表示层上；

实现了一对多的通讯机制，支持事件注册机制，支持兴趣分发机制，当被观察者触发事件时，只有感兴趣的观察者可以接收到通知；

缺点

如果观察者数量过多，则事件通知会耗时较长；

事件通知呈线性关系，如果其中一个观察者处理事件卡壳，会影响后续的观察者接收该事件；

如果观察者和被观察者之间存在循环依赖，则可能造成两者之间的循环调用，导致系统崩溃；

### 使用场景

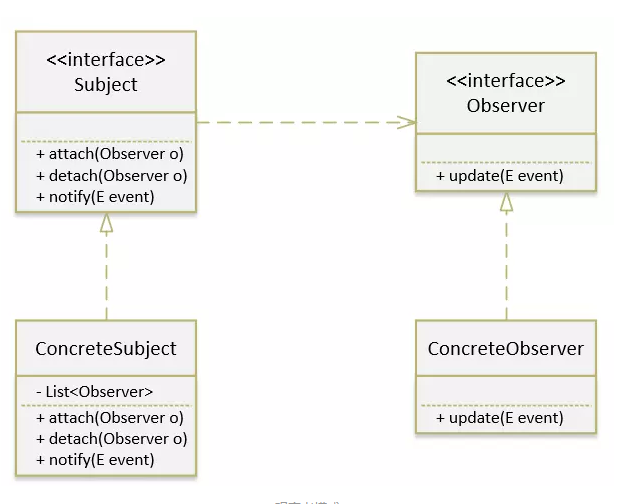
当一个抽象模型包含两个方面内容，其中一个方面依赖于另一个方面；

其他一个或多个对象的变化依赖于另一个对象的变化；

实现类似广播机制的功能，无需知道具体收听者，只需分发广播，系统中感兴趣的对象会自动接收该广播；

多层级嵌套使用，形成一种链式触发机制，使得事件具备跨域（跨越两种观察者类型）通知；

## 模式讲解



从 UML 类图中，我们可以看到，观察者模式 主要包含三种角色：

抽象主题（Subject）：指被观察的对象（Observable）。该角色是一个抽象类或接口，定义了增加、删除、通知观察者对象的方法；

具体主题（ConcreteSubject）：具体被观察者，当其内状态变化时，会通知已注册的观察者；

抽象观察者（Observer）：定义了响应通知的更新方法；

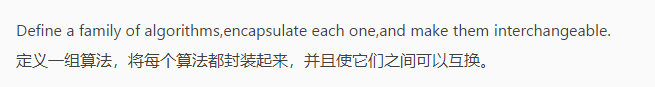
具体观察者（ConcrereObserver）：在得到状态更新时，会自动做出响应；

## Spring中使用该模式

# 策略模式

## 简介

### 策略模式



策略模式（Strategy Pattern） 也叫 政策模式（Policy Pattern）。指的是对象具备某个行为，但是在不同的场景中，该行为有不同的实现算法。比如一个人的交税比率与他的工资有关，不同的工资水平对应不同的税率。

策略模式 使用的就是面向对象的继承和多态机制，从而实现同一行为在不同场景下具备不同实现。

策略模式本质：分离算法，选择实现

### 主要解决

在有多种算法相似的情况下，使用 if...else 或 switch...case 所带来的复杂性和臃肿性。

### 优缺点

优点

算法多样性，且具备自由切换功能；

有效避免多重条件判断，增强了封装性，简化了操作，降低出错概率；

扩展性良好，策略类遵顼 里氏替换原则，可以很方便地进行策略扩展；

缺点

策略类数量增多，且所有策略类都必须对外暴露，以便客户端能进行选择；

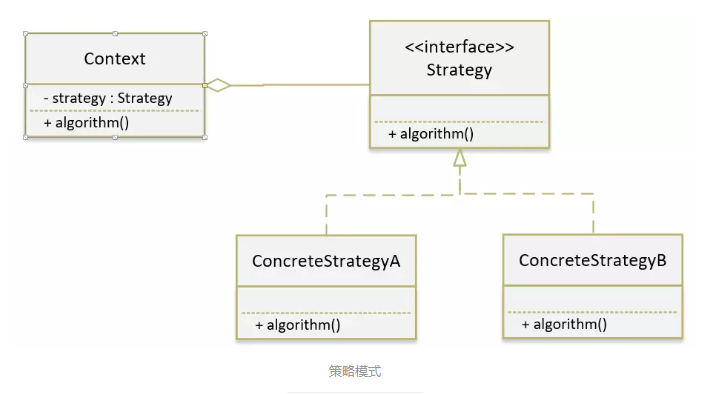
### 使用场景

针对同一类型问题，有多种处理方式，每一种都能独立解决问题；

算法需要自由切换的场景；

需要屏蔽算法规则的场景；

## 模式讲解



从 UML 类图中，我们可以看到，策略模式 主要包含三种角色：

上下文角色（Context）：用来操作策略的上下文环境，屏蔽高层模块（客户端）对策略，算法的直接访问，封装可能存在的变化；

抽象策略角色（Strategy）：规定策略或算法的行为；

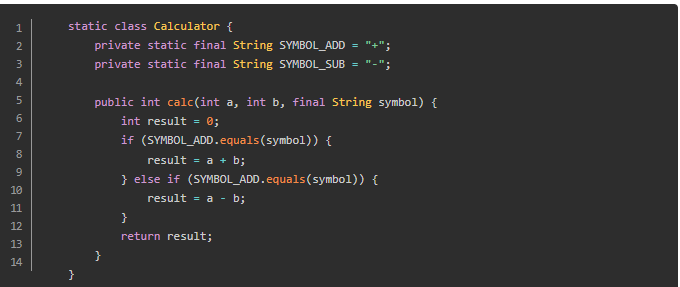
具体策略角色（ConcreteStrategy）：具体的策略或算法实现；

### [策略模式](https://baike.baidu.com/item/%E7%AD%96%E7%95%A5%E6%A8%A1%E5%BC%8F/646307?fr=aladdin) 的通用代码

|  |
| --- |
| **class** Client {  **public static void** main(String[] args) {  *//选择一个具体策略* IStrategy strategy = **new** ConcreteStrategyA();  *//来一个上下文环境* Context context = **new** Context(strategy);  *//客户端直接让上下文环境执行算法* context.algorithm();  }   *//抽象策略类 Strategy* **interface** IStrategy {  **void** algorithm();  }   *//具体策略类 ConcreteStrategy* **static class** ConcreteStrategyA **implements** IStrategy {   @Override  **public void** algorithm() {  System.***out***.println(**"Strategy A"**);  }  }   *//具体策略类 ConcreteStrategy* **static class** ConcreteStrategyB **implements** IStrategy {   @Override  **public void** algorithm() {  System.***out***.println(**"Strategy B"**);  }  }   *//上下文环境* **static class** Context {  **private** IStrategy **mStrategy**;   **public** Context(IStrategy strategy) {  **this**.**mStrategy** = strategy;  }   **public void** algorithm() {  **this**.**mStrategy**.algorithm();  }  } } |

## 例子

例子：假设现在有两个数与一个运算符，要求使用该运算符操作这两个数。  
分析：直接思路：通过判断运算符符号，对这两个数进行运算。代码如下所示：



但是这样写的话，如果我们现在要扩展乘法\*或除法/运算，那么就要在calc方法内增加对应的if...else判断，代码臃肿并且扩展性太低。  
而如果采用[策略模式](https://baike.baidu.com/item/%E7%AD%96%E7%95%A5%E6%A8%A1%E5%BC%8F/646307?fr=aladdin)，将各种运算符的计算都归并到对应具体策略，这样，就能简化代码并且带来很好的扩展性，具体代码如下：

|  |
| --- |
| **class** Client {  **public static void** main(String[] args) {  ICalculator calculator = **new** Add();  Context context = **new** Context(calculator);  **int** result = context.calc(1,2);  System.***out***.println(result);  }   **interface** ICalculator {  **int** calc(**int** a, **int** b);  }   **static class** Add **implements** ICalculator {  @Override  **public int** calc(**int** a, **int** b) {  **return** a + b;  }  }   **static class** Sub **implements** ICalculator {  @Override  **public int** calc(**int** a, **int** b) {  **return** a - b;  }  }   **static class** Multi **implements** ICalculator {  @Override  **public int** calc(**int** a, **int** b) {  **return** a \* b;  }  }   **static class** Divide **implements** ICalculator {  @Override  **public int** calc(**int** a, **int** b) {  **return** a / b;  }  }   **static class** Context {  **private** ICalculator **mCalculator**;   **public** Context(ICalculator calculator) {  **this**.**mCalculator** = calculator;  }   **public int** calc(**int** a, **int** b) {  **return this**.**mCalculator**.calc(a, b);  }  } } |

从上面代码中，我们可以看到，我们完全消除了对运算符号进行判断的哪些if...else的冗余代码，取而代之的是客户端直接决定使用哪种算法，然后交由上下文获取结果。并且上面代码中我们还扩展了乘法Multi和除法Divide运算，所需要做的就只是扩展相应的策略类而已。

**注意**：[策略模式](https://baike.baidu.com/item/%E7%AD%96%E7%95%A5%E6%A8%A1%E5%BC%8F/646307?fr=aladdin) 中的上下文环境（Context），其职责本来是隔离客户端与策略类的耦合，让客户端完全与上下文环境沟通，无需关系具体策略。但是从上面的代码中我们可以看到，客户端内部直接自己指定要哪种策略（ICalculator calculator = new Add())，客户端与具体策略类耦合了，而上下文环境在这里其的作用只是负责调度执行，获取结果，并没有完全起到隔离客户端与策略类的作用。一般可以通过简单工厂模式将具体策略的创建与客户端进行隔离，或者是通过 策略枚举 将上下文环境与具体策略类融合在一起，简化代码。当具体策略相对稳定时，推荐使用 策略枚举 简化代码，具体代码如下：

|  |
| --- |
| **class** EnumClient {  **public static void** main(String[] args) {  **int** result = Calculator.***ADD***.calc(1, 2);  System.***out***.println(result);  *// System.out.println(Calculator.ADD.getSymbol());* }   **static enum** Calculator {  *// 加法运算* ***ADD***(**"+"**) {  @Override  **public int** calc(**int** a, **int** b) {  **return** a + b;  }  },  ***SUB***(**"-"**) {  @Override  **public int** calc(**int** a, **int** b) {  **return** a - b;  }  };  **private** String **symbol**;   **private** Calculator(String symbol) {  **this**.**symbol** = symbol;  }   **public** String getSymbol() {  **return this**.**symbol**;  }   **public abstract int** calc(**int** a, **int** b);  } } |

## 策略模式的实战(spring中)

### 模式组成

抽象的策略接口:不同的实现类继承的接口，定义具体实现公有的方法。(StrategyService)

策略的具体实现:实现接口，并在接口方法中做具体的实现。(TStrategyService)

策略类: 持有一个抽象策略的引用，并提供统一调用的入口(StrategyController)。

常量定义:获取对应策略类的实现(TypeEnum)。

### 获取实现的引用

策略类中获取具体实现的引用

### StrategyController。

|  |
| --- |
| @RestController @RequestMapping(**"/strategy"**) **public class** StrategyController {  *//所有的实现该类的对象* **private static final** Map<String, StrategyService> ***StrategyServicesMap*** = **new** HashMap<>();  **private static final** String ***type*** = **"type"**;  **public** StrategyController(List<StrategyService> strategyServices) {  **for** (StrategyService eventService : strategyServices) {  ***StrategyServicesMap***.put(eventService.getType(), eventService);  }  }  @PostMapping  **public** Result create(@RequestBody Object element) {  JSONObject json = JSON.*parseObject*(element.toString());  String objectType = json.getString(***type***);  StrategyService strategyService = ***StrategyServicesMap***.get(objectType);  Class strategyClass = strategyService.getStrategyClass();  Object o = json.toJavaObject(strategyClass);  **return** Result.*create*(strategyService.create(o));  }  @PutMapping  **public int** delete(@RequestBody Object element) {  **return** 0;  }   @PutMapping  **public int** update(@RequestBody Object element) {  **return** 0;  }   @GetMapping  **public int** find(@RequestBody Object element) {  **return** 0;  } } |

### StrategyService

|  |
| --- |
| *//策略接口* **public interface** StrategyService<T> {  **int** create(T t);  **int** delete(Integer id);  **int** update(T t);  T find(Integer id);  String getType();  Class getStrategyClass(); } |

### TStrategyService

BBServiceImpl

|  |
| --- |
| *//策略B实现* @Service **public class** BServiceImpl **implements** StrategyService<B> {  @Override  **public int** create(B b) {  **return** 0;  }  @Override  **public int** delete(Integer id) {  **return** 0;  }  @Override  **public int** update(B b) {  **return** 0;  }  @Override  **public** B find(Integer id) {  **return null**;  }  @Override  **public** String getType() {  **return** TypeEnum.***B***.getType();  }  @Override  **public** Class getStrategyClass() {  **return** B.**class**;  } } |

AServiceImpl

|  |
| --- |
| *//策略A实现* @Service **public class** AServiceImpl **implements** StrategyService<A> {  @Override  **public int** create(A a) {  **return** 0;  }  @Override  **public int** delete(Integer id) {  **return** 0;  }  @Override  **public int** update(A a) {  **return** 0;  }   @Override  **public** A find(Integer id) {  **return null**;  }  @Override  **public** String getType() {  **return** TypeEnum.***A***.getType();  }  @Override  **public** Class getStrategyClass() {  Type[] genericInterfaces = **this**.getClass().getGenericInterfaces();  Class clazz = (Class) ((ParameterizedType) genericInterfaces[0]).getActualTypeArguments()[0];  **return** clazz;  } } |

### 枚举

|  |
| --- |
| *//类型枚举* **public enum** TypeEnum {  A(**"A"**),  B(**"B"**);  **private** String type;  TypeEnum(String type) {  **this**.type = type;  }  **public** String getType() {  **return** type;  } } |

# 监听者模式

## 观察者模式

### 简介

观察者模式（Observer Pattern），又叫 发布-订阅（Publish/Subscribe）模式、模型-视图（Model/View）模式、源-监听器（Source/Listener）模式 或 从属者（Dependents）模式。

观察者模式 是对象的行为模式，定义了一种一对多的依赖关系，一个主题对象可被多个观察者对象同时监听，使得每当主题对象状态变化时，所有依赖于它的对象都会得到通知并被自动更新。

观察者模式 核心：松耦合观察者与被观察者，以类似于消息/广播发送的机制联动两者，使被观察者的变动能通知到感兴趣的观察者们，从而做出相应的响应。

主要解决: 当系统一方行为依赖于另一方行为的变动时，可使用 观察者模式 松耦合联动双方，使得一方的变动可以通知到感兴趣的另一方对象，从而让另一方对象对此做出响应。

### 优缺点

优点：

观察者和被观察者是松耦合（抽象耦合）的，符合 依赖倒置原则；

分离了表示层（观察者）和数据逻辑层（被观察者），并且建立了一套触发机制，使得数据的变化可以响应到多个表示层上；

实现了一对多的通讯机制，支持事件注册机制，支持兴趣分发机制，当被观察者触发事件时，只有感兴趣的观察者可以接收到通知；

缺点

如果观察者数量过多，则事件通知会耗时较长；

事件通知呈线性关系，如果其中一个观察者处理事件卡壳，会影响后续的观察者接收该事件；

如果观察者和被观察者之间存在循环依赖，则可能造成两者之间的循环调用，导致系统崩溃

### 使用场景

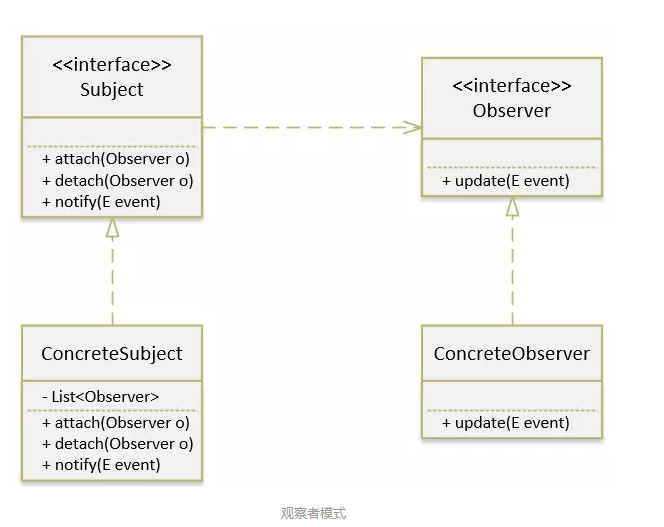
当一个抽象模型包含两个方面内容，其中一个方面依赖于另一个方面；

其他一个或多个对象的变化依赖于另一个对象的变化；

实现类似广播机制的功能，无需知道具体收听者，只需分发广播，系统中感兴趣的对象会自动接收该广播；

多层级嵌套使用，形成一种链式触发机制，使得事件具备跨域（跨越两种观察者类型）通知；

## 模式讲解



从 UML 类图中，我们可以看到，观察者模式 主要包含三种角色：

* **抽象主题（Subject）**：指被观察的对象（Observable）。该角色是一个抽象类或接口，定义了增加、删除、通知观察者对象的方法；
* **具体主题（ConcreteSubject）**：具体被观察者，当其内状态变化时，会通知已注册的观察者；
* **抽象观察者（Observer）**：定义了响应通知的更新方法；
* **具体观察者（ConcrereObserver）**：在得到状态更新时，会自动做出响应；

### 以下是 观察者模式 的通用代码

|  |
| --- |
| **class** Client {  **public static void** main(String[] args) {  *// 被观察者* ISubject<String> observable = **new** ConcreteSubject<>();  *// 观察者* IObserver<String> observer = **new** ConcreteObserver<>();  *// 注册* observable.attach(observer);  *// 通知* observable.notify(**"hello"**);  }  *//抽象观察者* **interface** IObserver<E> {  **void** update(E event);  }  *//抽象主题者* **interface** ISubject<E> {  **boolean** attach(IObserver<E> observer);   **boolean** detach(IObserver<E> observer);   **void** notify(E event);  }  *//具体观察者* **static class** ConcreteObserver<E> **implements** IObserver<E> {  @Override  **public void** update(E event) {  System.***out***.println(**"receive event: "** + event);  }  }  *//具体主题者* **static class** ConcreteSubject<E> **implements** ISubject<E> {  **private** List<IObserver<E>> **mObservers** = **new** ArrayList<>();   @Override  **public boolean** attach(IObserver<E> observer) {  **return** !**this**.**mObservers**.contains(observer) && **this**.**mObservers**.add(observer);  }   @Override  **public b oolean** detach(IObserver<E> observer) {  **return this**.**mObservers**.remove(observer);  }   @Override  **public void** notify(E event) {  **for** (IObserver<E> observer : **this**.**mObservers**) {  observer.update(event);  }  }  } } |

## 举个例子

例子：比如，我们每天都会设置闹钟，然后听到闹钟响时，就会自动起床，关闭闹钟，刷牙洗脸，开始新的一天。

分析：我们一听到闹钟响铃时，就会自动起床。按 观察者模式 进行分析，我们相当于观察者（Observer），监听闹钟响铃。闹钟相当于被观察者（Observable），时间到的时间就会通过响铃通知我们。

|  |
| --- |
| **class** Client {  **public static void** main(String[] args) {  *//来一个闹钟* IObservable<String> alarmClock = **new** AlarmClock();  *//我* IObserver<String> me = **new** Me();  *//设置闹钟时间（注册监听）* alarmClock.register(me);  *//闹钟响铃* alarmClock.notify(**"ring! ring ! ring! time is up!"**);  }   **interface** IObserver<E> {  **void** update(E event);  }   **interface** IObservable<E> {  **boolean** register(IObserver<E> observer);   **boolean** unRegister(IObserver<E> observer);   **void** notify(E event);  }   **static class** Me **implements** IObserver<String> {   @Override  **public void** update(String event) {  System.***out***.println(**"receive: "** + event);  System.***out***.println(**this**.getClass().getSimpleName()+**": time to wake up!"**);  }  }   **static class** AlarmClock **implements** IObservable<String> {  **private** List<IObserver<String>> **mObservers** = **new** ArrayList<>();   @Override  **public boolean** register(IObserver<String> observer) {  **return** !**this**.**mObservers**.contains(observer) && **this**.**mObservers**.add(observer);  }   @Override  **public boolean** unRegister(IObserver<String> observer) {  **return this**.**mObservers**.remove(observer);  }   @Override  **public void** notify(String event) {  **for** (IObserver<String> observer : **this**.**mObservers**) {  observer.update(event);  }  }  } } |

## 监听者模式多线程

主题

|  |
| --- |
| **public abstract class** ObserverableRunable **implements** Runnable {   **final protected** LifeCycleListener **listener**;   **public** ObserverableRunable(**final** LifeCycleListener listener){  **this**.**listener** = listener;  }   **protected void** notifyChange(**final** RunableEvent runableEvent){  **listener**.onEvent(runableEvent);  }   **public enum** RunableState{  ***RUNNING***,***ERROR***,***DONE***;  }   **public static class** RunableEvent{  **private final** RunableState **state**;  **private final** Thread **thread**;  **private final** Throwable **cause**;   **public** RunableEvent(RunableState state, Thread thread, Throwable cause) {  **this**.**state** = state;  **this**.**thread** = thread;  **this**.**cause** = cause;  }   **public** RunableState getState() {  **return state**;  }   **public** Thread getThread() {  **return thread**;  }   **public** Throwable getCause() {  **return cause**;  }  } } |

抽象观察者（Observer）

|  |
| --- |
| **public interface** LifeCycleListener {  **void** onEvent(ObserverableRunable.RunableEvent runableEvent); } |

具体观察者（ConcrereObserver）

|  |
| --- |
| **public class** ThreadLifeCycleObserver **implements** LifeCycleListener{   **private final** Object **Lock** = **new** Object();   **public void** concurrentQuery(List<String> ids){  **if**(ids==**null**||ids.isEmpty()){  **return**;  }  ids.stream().forEach(id->**new** Thread(**new** ObserverableRunable(**this**) {  @Override  **public void** run() {  **try** {  notifyChange(**new** RunableEvent(RunableState.***RUNNING***,Thread.*currentThread*(),**null**));  System.***out***.println(**"querry for the id"**+id);  notifyChange(**new** RunableEvent(RunableState.***DONE***,Thread.*currentThread*(),**null**));  }**catch** (Exception e){  notifyChange(**new** RunableEvent(RunableState.***DONE***,Thread.*currentThread*(),**null**));  }  }  },id).start());  }  @Override  **public void** onEvent(ObserverableRunable.RunableEvent runableEvent) {  **synchronized** (**Lock**){  System.***out***.println(**"the runalbe"** +runableEvent.getThread().getName()+runableEvent.getState());  }  } } |

## spring中使用监听者模式(构造器注入)

### 抽象主题

|  |
| --- |
| **public interface** Subject<E> {  **void** attach(Observer<E> observer);  **void** detach(Observer<E> observer);  **void** notify(E message); } |

### 2抽象观察

|  |
| --- |
| **public interface** Observer<E> {  **void** getMessage(E message); } |

### 具体主题

|  |
| --- |
| @Service **public class** ConcreteSubject **implements** Subject<String> {  //由spring的构造器注入方式注入  **private** List<Observer> **list**;  **public** ConcreteSubject(List<Observer> observerList) {  **this**.**list** = observerList;  }  **public void** doWork(){  notify(**"hello"**);  }  @Override  **public void** attach(Observer observer) {  **list**.add(observer);  }  @Override  **public void** detach(Observer observer) {  **list**.remove(observer);  }  @Override  **public void** notify(String message) {  **for** (Observer observer : **list**) {  observer.getMessage(message);  }  } } |

### 3具体的观察者A

|  |
| --- |
| @Service **public class** AObserverImpl **implements** Observer<String>{  @Override  **public void** getMessage(String message) {  System.***out***.println(**"a接受到消息"**);  } } |

### 具体的观察者B

|  |
| --- |
| @Service **public class** BObserverImpl **implements** Observer<String>{  @Override  **public void** getMessage(String message) {  System.***out***.println(**"b接受到消息"**);  } } |

## spring实现观察者模式(初始化bean注入)

### 抽象主题

|  |
| --- |
| **public interface** Subject<E> {  **void** attach(Observer<E> observer);  **void** detach(Observer<E> observer);  **void** notify(E message); } |

### 抽象观察

|  |
| --- |
| **public interface** Observer<E> {  **void** getMessage(E message); } |

### 具体主题

|  |
| --- |
| @Component **public class** ConcreteSubject **implements** Subject<String> {  *//通过InitializingBean注入* **private** List<Observer> **observers** = **new** ArrayList<>();  **public void** doWork(){  notify(**"hello"**);  }  @Override  **public void** attach(Observer observer) {  **observers**.add(observer);  }  @Override  **public void** detach(Observer observer) {  **observers**.remove(observer);  }  @Override  **public void** notify(String message) {  **for** (Observer observer : **observers**) {  observer.getMessage(message);  }  } } |

### 具体观察者A

|  |
| --- |
| @Component **public class** AObserverImpl **implements** Observer<String>, InitializingBean {  @Autowired  **private** ConcreteSubject **concreteSubject**;  @Override  **public void** getMessage(String message) {  System.***out***.println(**"a接受到消息"**);  }  @Override  **public void** afterPropertiesSet() **throws** Exception {  **concreteSubject**.attach(**this**);  } } |

### 具体观察者B

|  |
| --- |
| @Component **public class** BObserverImpl **implements** Observer<String>, InitializingBean {  @Autowired  **private** ConcreteSubject **concreteSubject**;  @Override  **public void** getMessage(String message) {  System.***out***.println(**"b接受到消息"**);  }  @Override  **public void** afterPropertiesSet() **throws** Exception {  **concreteSubject**.attach(**this**);  } } |

# 单线程执行设计模式

# 不可变对象的设计模式

## 不可变对象设计模式

不可变对象:一定是A程安全的1面的在何属性或者引用英型的属性都不能被修改

可变对象不一定是不安全的sEngBuftet

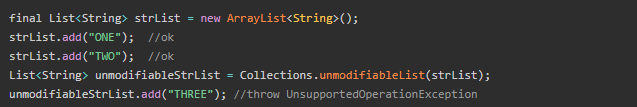
### 什么是Immutable

Immutable意为不可改变的，如果一个对象定义成了不可变的（即Immutable Object），就意味着该对象在初始化完成之后它的属性是不能够被修改的。在并发编程中我们可以将对象设计成Immutable Object从而不用加锁实现线程安全，因为不可变对象一定是线程安全的，同时由于不需要用一些锁机制等保证内存一致性问题也减少了同步开销。

谈到Immutable Object会让很多Javaer联想到Java语言中的final关键字，final关键字可以修饰属性、方法和类。

final的作用为:final 修饰的 class 代表该类不可以继承扩展，final 的变量是不可以修改的，而 final 的方法也是不可以重写的（override）。

那是不是被final修饰的对象就可以认为是Immutable Object呢？当然不是。请看下面的例子



这个例子中strList声明成final，只能说明strList变量指向的地址空间不能改变但是该地址空间指向的内容是可以修改的。

### 如何定义Immutable Object

Java 语言目前还没有原生的不可变对象的支持，但在Java™ Tutorials中给出的如何定义一个不可变对象的方法。

1、类中的属性不提供"setter"方法；

2、类中所有的属性声明成private和final类型；

3、类也声明成final的，以防止类被继承；

4、如果有属性是引用类型的，也要防止引用类型的属性被调用方修改了，如通过构造器初始化所有成员，尤其是引用对象要进行深拷贝(deep copy，符合copy-on-write 原则)；

5、如果确实需要实现 getter 方法，或者其他可能会返回内部状态的方法，也要深拷贝，创建私有的 copy。

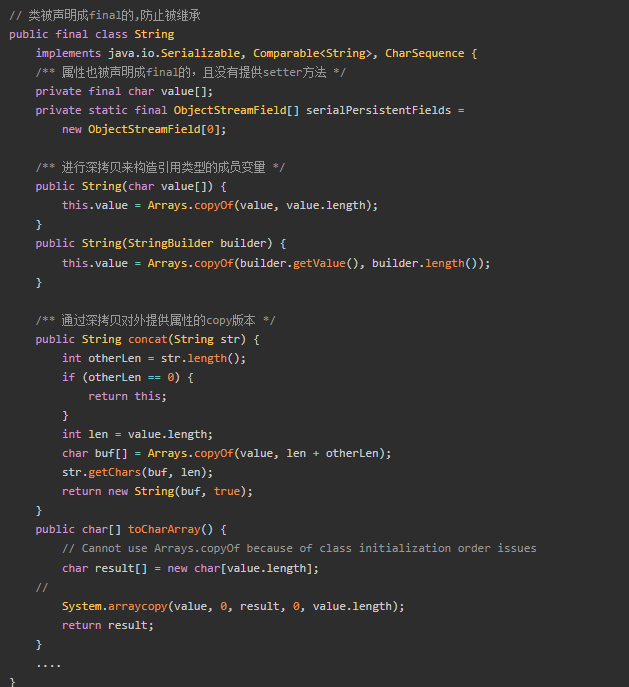
针对上面提到的第4点进行一下说明，如果构造器传入的对象直接赋值给成员变量，还是可以通过对传入对象的修改进而导致改变内部变量的值。如

## 例如

|  |
| --- |
| **public class** ImmutableTest {  **private final int age**;  **private final** String **name**;   **private final** List<String> **list**;   **public** ImmutableTest(**int** age, String name) {  **this**.**age** = age;  **this**.**name** = name;  **this**.**list** = **new** ArrayList<>();  }   *//调用该方法，获取list，修改其中的值* **public** List<String> getList() {  *//返回一个都不可修改的list* **return** Collections.*unmodifiableList*(**list**);  } } |

## String对象的不可变性

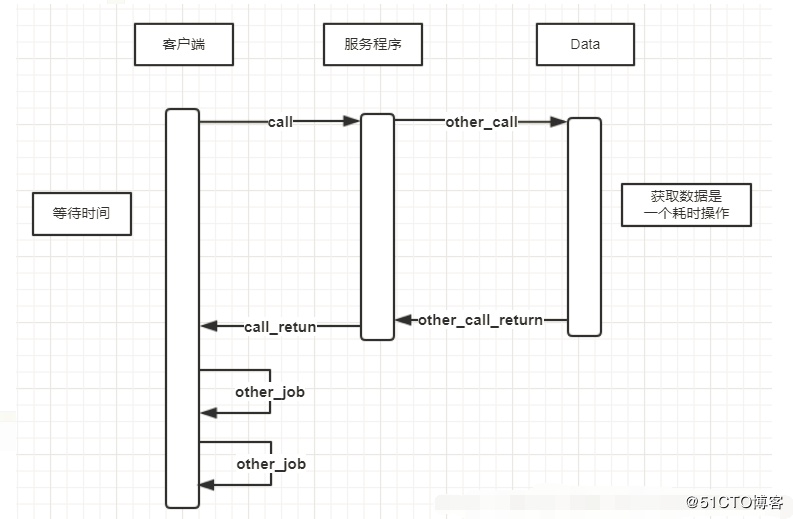
String是典型的 Immutable 类，被声明成为 final class，它的属性基本都是 final 的。由于它的不可变性，类似拼接、裁剪字符串等动作，都会产生新的 String 对象。下面是String源码的部分摘录，可以看到它是符合Immutable Object的定义规则的。



# Future设计模式

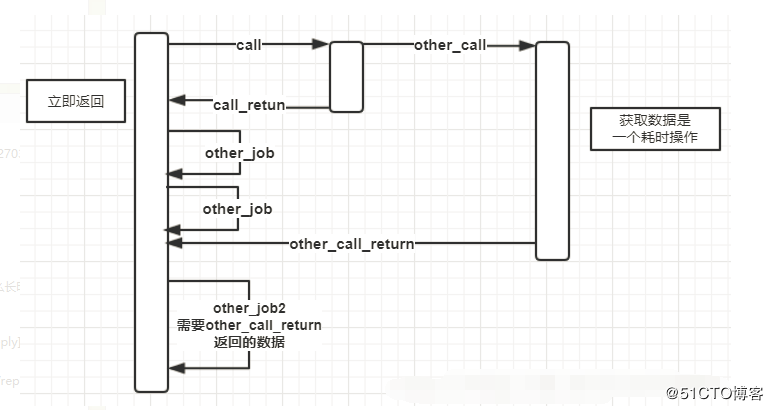
## 引入

有时候会遇到一种情况，当流程到了某个一个节点的时候，数据处理需要花费比较长的时间。只是在最后返回成功或者失败的时候体现，数据处理的结果并不影响后面流程的进展，现在介绍一种设计模式来充分利用时间片段，这个设计模式就是-Future设计模式。



上图显示的是一个串行程序调用的流程，可以看出当有一个程序执行的操作比较耗时的时候，后面的程序必须等待该模块的执行完成，才能继续往下执行。

Future的设计模式流程图



Future的设计模式可以看出，主线程直接跳过了耗时的子线程，整个流程没有长时间的等待操作，充分利用了耗时的时间片段，提高了系统的执行效率，与用户的体验。

每个角色说一下

1、Main：启动系统，调用Client发出请求；

2、Client：返回Data对象，理解返回FutureData，并开启ClientThread线程装配RealData；

3、Data：返回数据的接口；

4、FutureData：Future数据，构造很快，但是是一个虚拟的数据，需要装配RealData；

5、RealData：真实数据，构造比较慢。

## Futrue案例

### 案例角色

Futrue:未来的一个平局

FurtueTask 将自己的逻辑进行隔离

FutrueServuce 桥接Futrue和FutrueClient

# 多线程设计模式

## Guarded Suspension

## balking pattern

## produce-consumer