**设计模式**

# 单例模式（进程内）

单例模式在程序设计中使用的频率非常之高，其设计的目的是为了在程序中提供唯一一个对象（保证只被构造一次），例如写入日志的log对象，windows的任务管理器实现（只能打开一个）。这里主要介绍单例模式使用Java的实现（包括饿汉式及懒汉式）。

这里使用Log类作为例子，Log对象需要在程序中只有一个对象且只初始化一次。

## 饿汉式

饿汉式的单例模式理解起来是比较容易的，就是在单例类加载的时候就初始化需要单例的对象。实现也比较容易。

public class Singleton{

private static Log logObj = new Log();

public static Log getInstance(){

return logObj;

}

}

## 懒汉式

如果logObj需要占用很大的内存，如果一开始就初始化logObj，那么会占用大量的内存。此时，有人就想，如果我在想用的时候再初始化Log类的对象，像懒汉一样，只有用到的时候再初始化，需要怎么设计呢？

### 实现一（非线程安全版本）

public class Singleton{

private static Log logObj = null;

public static Log getInstance(){

if(logObj == null){

logObj = new Log();

}

return logObj;

}

}

### 实现二（线程安全版本）

public class Singleton{

private static Log logObj = null;

public static synchronized Log getInstance(){

if(logObj == null){

logObj = new Log();

}

return logObj;

}

}

为了实现线程安全，这个版本的实现牺牲了一定的效率，如果logObj已经初始化，那么其他线程还需要同步的进入getInstance方法，会造成效率的损失。于是，有些人实现了下面的版本。

### 实现三（错误版本）

public class Singleton{

private static Log logObj = null;

public static Log getInstance(){

if(logObj == null){

synchronized(Singleton.class){

if(logObj == null){

logObj = new Log();

}

}

}

return logObj;

}

}

乍看起来上面的版本是没问题的，如果某个线程A发现logObj 还没初始化，那么就进入同步块初始化logObj，如果在这期间有其他线程B进入，那么线程B就会等待进入同步块，等待A 线程退出同步块，logObj 已经初始化了，B 线程进入同步块后发现logObj 不为null，退出同步块，不再初始化logObj。 这样既实现了线程安全，又兼顾了效率，确实是很聪明的编码方式。但是问题来了，由于指令重排序的存在，会导致Log在完全初始化之前logObj就已经不为null。这样其他线程可能会得到未完全初始化的对象。

### 解决方法

1. JDK1.5版本后扩展了volitile语义，可以保证上述代码的正确性，为此只要将logObj 声明为volitile即可（volitile之前只是保证内存的可见性而已）。
2. 使用静态内部类。

加载一个类时，其内部类不会同时被加载。一个类被加载，当且仅当其某个静态成员（静态域、构造器、静态方法等）被调用时发生。

并且jvm会保证类加载的线程安全问题，所以利用这个特性可以写出兼顾效率与保证线程安全的版本。

### 实现四（兼顾效率与线程安全的版本）

public class Singleton{

static class LogHolder{

static Log logObj = new Log();

}

public static Log getInstance(){

return LogHolder.logObj;

}

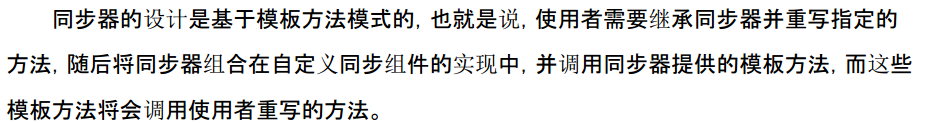
}

这样在Singleton类加载时，并不会加载LogHolder，也就不会初始化Log，如果有线程访问getInstance方法，那么jvm会首先加载LogHolder类，并保证初始化logObj，最后返回logObj。

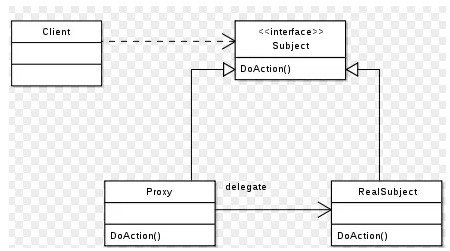
# 扩展（实现线程范围内单例）

|  |
| --- |
| /\*\*  \* 线程内共享单例  \* **@author** cjl  \*  \*/  **public** **class** ThreadScopeSingleton {  **private** **static** ThreadLocal<ThreadScopeSingleton> *map* =**new** ThreadLocal<ThreadScopeSingleton>();    **private** ThreadScopeSingleton(){}    **public** **static** ThreadScopeSingleton getThreadInstance(){  **if**(*map*.get()==**null**){  *map*.set(**new** ThreadScopeSingleton());  }  **return** *map*.get();  }  } |

# 模板方法



# 代理模式



既然有了静态代理，为什么需要动态代理呢，因为静态代理有一个最大的缺陷：**接口与代理类是1对1的，有多个接口需要代理，就需要新建多个代理类，繁琐，类爆炸**。

# 装饰模式

I/O大量使用了装饰模式

对类增强：继承

Order 对订单进行增强，

Page<T> : size , page , t : 装饰模式，

# 模板方法

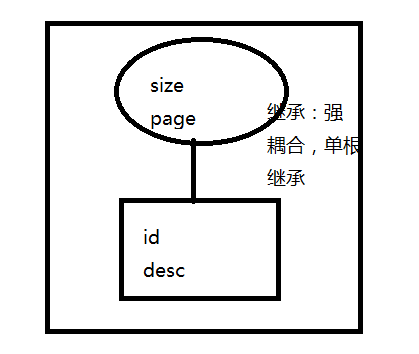
具体参考：spring 源码 中 **jdbctemplate** resttmelate, redistemplate

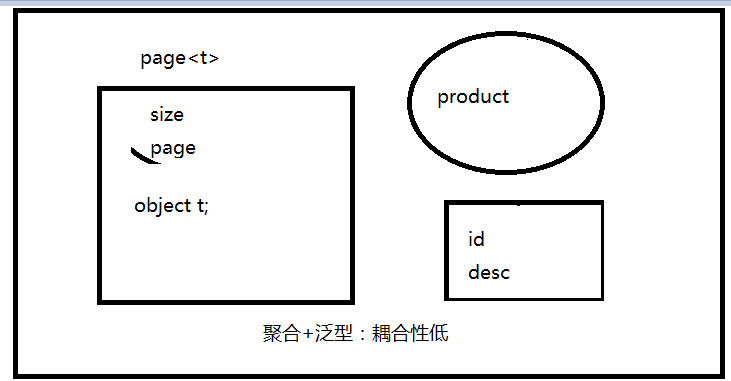
**而且spring 进行了升华： 模板方法+回调方法**

# 不同维度理解

## 增强

类增强： **继承、聚合、装饰模式**





方法增强：代理模式

方法里的代码增强：模板方法 spring : jdbctemplate

## 重复

垂直重复： 继承

横向重复： AOP

方法代码重复：模板方法

# 设计原则

重要：**23种设计模式，是设计原则的具体实现。**

Single Responsibility Principle　 : **单一职责原则**

　　Liskov Substitution Principle : **里氏替换原则**

Dependence Inversion Principle ：**依赖倒置原则**

**依赖抽象，不要依赖具体实现**

**依赖接口，不要依赖类**

　　Interface Segregation Principle : **接口隔离原则**

Law of Demeter　　　　 : **迪米特法则**

**最少知道原则。耦合性低，安全性高。**

Open Closed Principle : **开闭原则**

**对扩展开放，对修改封闭！！！**

**6大原则最重要，最核心的一个原则是哪个？？？**

**开闭原则！！！！！！！**

**为什么开闭？？ 极大的减少代码修改带来的人力和时间成本。**

**最终目标：赚钱！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！！**