# 第1章面向对象的六大原则

## 1.1 单一职责原则

简单来说，就是一个类中应该是一组相关性很高的函数、数据的封装。比如设计一个ImageLoader类，把下载和缓存都写这个类里了，要修改缓存的类也会修改到下载类的代码，所以应该把缓存相关的代码封装一个类。

## 1.2 开闭原则

开闭原则定义：软件中的对象对于扩展是开放的，但是对于修改是封闭的。 也就是说，一旦程序开发完成，程序中一个类的实现只应该因错误而被修改，新的或者改变的特性应该通过新建不同的类实现，新建的类可以通过继承的方式来重用原来的代码。

比如ImageLoader中的缓存类，缓存策略可以有多个，所以应该设计一个ImageCache接口，然后不同的缓存策略都实现这个接口，再给ImageLoader提供一个setImageCache方法，这样，当我们需使用新的缓存策略时，只需要再写一个类实现ImageCache，然后把这个新对象通过setImageChache设置给ImageLoader，这样我们实现了新的缓存，而且并没有修改到ImageLoader的代码。

开闭原则指导我们，当软件需要变化时，应该尽量通过扩展的方式来实现变化，而不是通过修改已有的代码来实现。这里的“应该尽量”4个字说明开闭原则并不是说绝对不可以修改原始类的。当我们嗅到原来的代码“腐化气味”时，应该尽早地重构，以便使代码恢复到正常的“进化”过程，而不是通过继承等方式添加新的实现，这会导致类型的膨胀以及历史遗留代码的冗余。我们的开发过程中也没有那么理想化的状况，完全地不用修改原来的代码的情况比较少，因此，在开发过程中需要自己结合具体情况进行考量，是通过修改旧代码还是通过继承使得软件系统更稳定、更灵活，在保证去除“代码腐化”的同时，也保证原有模块的正确性。

## 1.3 里氏替换原则

就是所有引用基类的地方，都可以替换为任意的子类。 比如ImageCache有MemoryCache、DiskCache、DoubleCache3个子类，ImageLoade的setImageCache(ImageCache cache)方法可以接受任意的子类，而且不会导致任何错误或异常。这就是里氏替换原则。 而反过来则不行，出现子类引用的地方不能替换为父类。

开闭原则和里氏替换原则往往是生死相依、不离不弃的，通过里氏替换来达到对扩展开放，对修改关闭的效果。

## 1.4 依赖代倒置原则

依赖倒置原则指定了一种特定的解耦形式，使得高层次的模块不依赖于低层次模块的实现细节，而是依赖抽象，依赖模块被颠倒了。

抽象是指抽象类或接口，细节是指实现类。

依赖倒置一句话理解就是“面向接口编程或者面向抽象编程”，这里的抽象指的是抽象类或接口类。

打个比方理解依赖：

class ImageLoader {

private MemoryCache cache;

}

如上代码，ImageLoader依赖了MemoryCache，这就是依赖细节，如果把MemoryCache换成接口ImageCache，则就是依赖抽象。

依赖注入：

class ImageLoader {

private ImageCache cache;

public void setImageCache(ImageCache cache) {

this.cache = cache;

}

}

上面的setImageCache方法就是把ImageCache注入进来了，这就是传说中的依赖注入。

通过以上几节中我们发现，要想让系统更为灵活，抽象似乎成了我们唯一的手段。

## 1.5 接口隔离原则

定义：类间的依赖关系应该建立在最小的接口上。接口隔离原则的目的是让系统解耦，从而容易重构、更改和重新部署。

接口隔离原则将非常庞大、臃肿的接口拆分成更小的和更具体的接口，使客户端依赖的接口尽可能的小，这样客户端将只需要知道他们感兴趣的方法。

比如有很多的IO流，关流的时候需要判断是否为空，然后try/catch再关。这些代码很臃肿，所以可以设计一个方法来关流closeIo(Closeable c)，这里的Closeable只有一个close方法，这就是最小化依赖原则，这就是接口隔离原则。 再比如ImageCache接口，只提供了最重要的put和get方法。

## 1.6 迪米原则

也称为最少知识原则，含意为：一个对象应该对其它对象有最少的了解。通俗地讲就是，一个类应该对自己需要调用的类知道得最少，只需要知道需要调用的方法即可。就比如对于ImageLoader，它依赖了缓存类，对于缓存只需要知道有put和get方法即可，所以我们写了一个接口ImageCache。

总结：

在应用的开发过程，最难的不是完成应用的开发工作，而是在后续的升级、维护过程中让应用系统能够拥抱变化。拥抱变化 也就意味在满足且不破坏系统稳定的前提下保持高扩展性、高内聚、低耦合，在经历了各版本的变更之后依然保持清晰、灵活、稳定的系统架构。当然这是一个比较理想的情况，我们必须朝这个方向去努力。遵循六大原则就是走向灵活软件之路的第一步。

# 第二章 单例模式

单例模式的使用场景：

确保某个类只有一个对象实例，避免生产多个消耗过多的资源。或者某种类型的对象只应该有一个。

单例模式实现方式：

## 饿汉式：

**public class** Singleton {  
 **public static final** Singleton ***instance*** = **new** Singleton();  
 **private** Singleton(){}  
 **public static** Singleton getInstance() { **return *instance***; }  
}

这种方式的缺点是类一加载就创建实例。

## 懒汉式：

**public class** Singleton {  
 **public static** Singleton *instance*;  
 **private** Singleton(){}  
 **public static synchronized** Singleton getInstance() {  
 **if** (*instance* == **null**) {  
 *instance* = **new** Singleton();  
 }  
 **return** *instance*;  
 }  
}

这种方式的优点是在需要的时候才创建实例，缺点是实例创建后再调用getInstance方法时还是要进行同步，造成了不必要的同步开销。

## 双检查单例：

**public class** Singleton {  
 **public static** Singleton *instance*;  
 **private** Singleton(){}  
 **public static** Singleton getInstance() {  
 **if** (*instance* == **null**) {  
 **synchronized** (Singleton.**class**) {  
 **if** (*instance* == **null**) {  
 *instance* = **new** Singleton();  
 }  
 }  
 }  
 **return** *instance*;  
 }  
}

## 静态内部类单例：

虽然双检查单例在一定程度上解决了资源消耗、多余的同步、线程安全等问题，但是，它还是在某些情况下出现失效的问题，这个问题被称为双重检查锁定失效，在《Java 并发编程实践》一书的最后谈到了这个问题，并指出这种“优化”是丑陋的，不赞成使用。而建议使用下面的代码代替：

**public class** Singleton {  
 **private** Singleton(){}  
 **public** Singleton getInstance() {  
 **return** SingletonHolder.***instance***;  
 }  
  
 **private static class** SingletonHolder {  
 **private static final** Singleton ***instance*** = **new** Singleton();  
 }  
}

当加载Singleton类的时候，SingletonHolder类并不会跟着立马加载，当调用getInstance方法的时候才会加载SingletonHolder并实例化instance。

这种方式即确保了线程安全，同时也延迟了单例的实例化，所以这是推荐使用的单例模式实现方式。

## 枚举单例：

**public enum** SingletonEnum {  
 ***INSTANCE***;  
}

写法简单是枚举的最大优点，枚举和普通的Java类一样可以拥有变量、方法，最重要的是枚举实例的创建是线程安全，而且在任何情况下它都是一个单例，而上面的其它单例并不能保证，通过序列化可以将一个单例的实例对象写到磁盘，然后再读回来，从而有效地获得一个新实例。

## 使用容器实现单例模式

**public class** SingletonManager {  
 **private static** Map<String, Object> *map* = **new** HashMap();  
 **private** SingletonManager() {}  
  
 **public static void** registerService(String key, Object instance) {  
 **if** (!*map*.containsKey(key)) {  
 *map*.put(key, instance);  
 }  
 }  
  
 **public static** Object getService(String key) {  
 **return** *map*.get(key);  
 }  
}

# 第三章 Builder模式

对象属性多，如果在构造方法接收参数，则需要的重载构造方法过多，可以使用Builder模式解决。

**public class** Car {  
 **private** String **a**;  
 **private** String **b**;  
 **private** String **c**;  
  
 **public static class** Builder {  
 **private** String **a**;  
 **private** String **b**;  
 **private** String **c**;  
  
 **public** Builder setA(String a) {  
 **this**.**a** = a;  
 **return this**;  
 }  
  
 **public** Builder setB(String b) {  
 **this**.**b** = b;  
 **return this**;  
 }  
  
 **public** Builder setC(String c) {  
 **this**.**c** = c;  
 **return this**;  
 }  
  
 **public** Car build() {  
 Car car = **new** Car();  
 **if** (**a** != **null**) {  
 car.**a** = **a**;  
 }  
 **if** (**b** != **null**) {  
 car.**b** = **b**;  
 }  
 **if** (**c** != **null**) {  
 car.**c** = **c**;  
 }  
 **return** car;  
 }  
 }  
  
}

没有理解这样写的好处，可以把那些set方法声明在Car上明呀，这样就不需要多余的Build类了。

# 第四章 原型模式

就是在使用一个对象时，不是new，而是使用一个已经存在的对象的拷贝。最常见的就是clone方法，克隆有浅克隆和深克隆，浅克隆只是克隆表面。如有对象A，通过A克隆出对象B，这时A对象中的一个属性和B对象中的一个属性指向的是同一个对象，这就叫浅克隆。深克隆不允许有指向同一个对象的情况存在，如ArrayList的clone方法就是深克隆，如下：

**public** Object clone() {  
 **try** {  
 ArrayList<?> v = (ArrayList<?>) **super**.clone();  
 v.elementData = Arrays.*copyOf*(**elementData**, **size**);  
 v.**modCount** = 0;  
 **return** v;  
 } **catch** (CloneNotSupportedException e) {  
 *// this shouldn't happen, since we are Cloneable* **throw new** InternalError(e);  
 }  
}

# 第五章 工厂方法模式