链滴

设计模式 1

作者：yhm
原文链接：https://ld246.com/article/1535010302422
来源网站：链滴

<pre><code class="highlight-chroma">—可复用面向对象软件的基础

</code></pre>

设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经的总结。使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性。 毫无疑问，计模式于己于他人于系统都是多赢的，设计模式使代码编制真正工程化，设计模式是软件工程的基石如同大厦的一块块砖石一样。项目中合理的运用设计模式可以完美的解决很多问题，每种模式在现在都有相应的原理来与之对应，每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的心解决方案，这也是它能被广泛应用的原因。本章系 Java 之美[从菜鸟到高手演变]系列之设计模式，我们会以理论与实践相结合的方式来进行本章的学习，希望广大程序爱好者，学设计模式，做一个优秀的软件工程师！
企业级项目实战(带源码)地址**：**http://zz563143188.iteye.com/blog/1825168
23 种模式 java 实现源码下载地址 http://pan.baidu.com/share/link?shareid=37 668&uk=4076915866#dir/path=%2F%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B
一、设计模式的分类
总体来说设计模式分为三大类：
创建型模式，共五种：工厂方法模式、抽象工厂模式、单例模式、建造者模式、原型模式。
结构型模式，共七种：适配器模式、装饰器模式、代理模式、外观模式、桥接模式、组合模式、元模式。
行为型模式，共十一种：策略模式、模板方法模式、观察者模式、迭代子模式、责任链模式、命模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。
其实还有两类：并发型模式和线程池模式。用一个图片来整体描述一下：
二、设计模式的六大原则
1、开闭原则（Open Close Principle）
开闭原则就是说对扩展开放，对修改关闭。在程序需要进行拓展的时候，能去修改原有的代码，实现一个热插拔的效果。所以一句话概括就是：为了使程序的扩展性好，易于护和升级。想要达到这样的效果，我们需要使用接口和抽象类，后面的具体设计中我们会提到这点。
2、里氏代换原则（Liskov Substitution Principle）
里氏代换原则(Liskov Substitution Principle LSP)面向对象设计的基本原则之一。 里氏代换原中说，任何基类可以出现的地方，子类一定可以出现。 LSP 是继承复用的基石，只有当衍生类可以替掉基类，软件单位的功能不受到影响时，基类才能真正被复用，而衍生类也能够在基类的基础上增加的行为。里氏代换原则是对“开-闭”原则的补充。实现“开-闭”原则的关键步骤就是抽象化。而基与子类的继承关系就是抽象化的具体实现，所以里氏代换原则是对实现抽象化的具体步骤的规范。— From Baidu 百科
3、依赖倒转原则（Dependence Inversion Principle）
这个是开闭原则的基础，具体内容：真对接口编程，依赖于抽象而不依赖于具体。
4、接口隔离原则（Interface Segregation Principle）
这个原则的意思是：使用多个隔离的接口，比使用单个接口要好。还是一个降低类之间的耦合度意思，从这儿我们看出，其实设计模式就是一个软件的设计思想，从大型软件架构出发，为了升级和护方便。所以上文中多次出现：降低依赖，降低耦合。
5、迪米特法则（最少知道原则）（Demeter Principle）
为什么叫最少知道原则，就是说：一个实体应当尽量少的与其他实体之间发生相互作用，使得系功能模块相对独立。

6、合成复用原则（Composite Reuse Principle）
原则是尽量使用合成/聚合的方式，而不是使用继承。
三、Java 的 23 中设计模式
从这一块开始，我们详细介绍 Java 中 23 种设计模式的概念，应用场景等情况，并结合他们的点及设计模式的原则进行分析。
1、工厂方法模式（Factory Method）
工厂方法模式分为三种：
11、普通工厂模式，就是建立一个工厂类，对实现了同一接的一些类进行实例的创建。首先看下关系图：
举例如下：（我们举一个发送邮件和短信的例子）
首先，创建二者的共同接口：
[java] view plaincopy

public interface Sender {
public void Send();
}

其次，创建实现类：
[java] view plaincopy

public class MailSender implements Sender {
@Override
public void Send() {
System.out.println(“this is mailsender!”);
}
}

[java] view plaincopy

public class SmsSender implements Sender {

@Override

public void Send() {

System.out.println(“this is sms sender!”);

}

}

最后，建工厂类：
[java] view plaincopy

public class SendFactory {

public Sender produce(String type) {

if (“mail”.equals(type)) {

return new MailSender();

} else if (“sms”.equals(type)) {

return new SmsSender();

} else {

System.out.println(“请输入正确的类型!”);

return null;

}

}

}

我们来测试下：

public class FactoryTest {

public static void main(String[] args) {

SendFactory factory = new SendFactory();

Sender sender = factory.produce(“sms”);

sender.Send();

}

}

输出：this is sms sender!
22、多个工厂方法模式，是对普通工厂方法模式的改进，在通工厂方法模式中，如果传递的字符串出错，则不能正确创建对象，而多个工厂方法模式是提供多个厂方法，分别创建对象。关系图：
将上面的代码做下修改，改动下 SendFactory 类就行，如下：
[java] view plaincopypublic class SendFactory {
public Sender produceMail(){

return new MailSender();

}

public Sender produceSms(){

return new SmsSender();

}

}

测试类如下：
[java] <a href="https://ld246.com/forward?goto=http%3A%2F%2Fblog.csdn.net%2Fzhangerqing%2Farticle%2Fdetails%2F8194653" title="view plain" target="_bla

k" rel="nofollow ugc">view plaincopy

public class FactoryTest {

public static void main(String[] args) {

SendFactory factory = new SendFactory();

Sender sender = factory.produceMail();

sender.Send();

}

}

输出：this is mailsender!
33、静态工厂方法模式，将上面的多个工厂方法模式里的方置为静态的，不需要创建实例，直接调用即可。
[java] view plaincopy

public class SendFactory {

public static Sender produceMail(){

return new MailSender();

}

public static Sender produceSms(){

return new SmsSender();

}

}

[java] view plaincopy

public class FactoryTest {

public static void main(String[] args) {

Sender sender = SendFactory.produceMail();

sender.Send();

}

}

输出：this is mailsender!
总体来说，工厂模式适合：凡是出现了大量的产品需要创建，并且具有共同的接口时，可以通过工厂方法模式进行创建。在以上的三种模式中，第一种如果传入的字符串有误，不能正确创建对象，第种相对于第二种，不需要实例化工厂类，所以，大多数情况下，我们会选用第三种——静态工厂方法式。
2、抽象工厂模式（Abstract Factory）
工厂方法模式有一个问题就是，类的创建依赖工厂类，也就是说，如果想要拓展程序，必须对工类进行修改，这违背了闭包原则，所以，从设计角度考虑，有一定的问题，如何解决？就用到抽象工模式，创建多个工厂类，这样一旦需要增加新的功能，直接增加新的工厂类就可以了，不需要修改之的代码。因为抽象工厂不太好理解，我们先看看图，然后就和代码，就比较容易理解。
请看例子：
[java] view plaincopy

public interface Sender {
public void Send();
}

两个实现类：
[java] <a href="https://ld246.com/forward?goto=http%3A%2F%2Fbl

g.csdn.net%2Fzhangerqing%2Farticle%2Fdetails%2F8194653" title="view plain" target="_bla
k" rel="nofollow ugc">view plain<a href="https://ld246.com/forward?goto=http%3A%2
%2Fblog.csdn.net%2Fzhangerqing%2Farticle%2Fdetails%2F8194653" title="copy" target="_bl
nk" rel="nofollow ugc">copy

public class MailSender implements Sender {
@Override
public void Send() {
System.out.println(“this is mailsender!”);
}
}

[java] <a href="https://ld246.com/forward?goto=http%3A%2F%2Fbl
g.csdn.net%2Fzhangerqing%2Farticle%2Fdetails%2F8194653" title="view plain" target="_bla
k" rel="nofollow ugc">view plain<a href="https://ld246.com/forward?goto=http%3A%2
%2Fblog.csdn.net%2Fzhangerqing%2Farticle%2Fdetails%2F8194653" title="copy" target="_bl
nk" rel="nofollow ugc">copy

public class SmsSender implements Sender {

@Override

public void Send() {

System.out.println(“this is sms sender!”);

}

}

两个工厂类：
[java] <a href="https://ld246.com/forward?goto=http%3A%2F%2Fbl
g.csdn.net%2Fzhangerqing%2Farticle%2Fdetails%2F8194653" title="view plain" target="_bla
k" rel="nofollow ugc">view plain<a href="https://ld246.com/forward?goto=http%3A%2
%2Fblog.csdn.net%2Fzhangerqing%2Farticle%2Fdetails%2F8194653" title="copy" target="_bl
nk" rel="nofollow ugc">copy

public class SendMailFactory implements Provider {

@Override

public Sender produce(){

return new MailSender();

}

}

[java] view plaincopy

public class SendSmsFactory implements Provider{

@Override

public Sender produce() {

return new SmsSender();

}

}

在提供一个接口：
[java] view plaincopy

public interface Provider {
public Sender produce();
}

测试类：
[java] view plaincopy

public class Test {

public static void main(String[] args) {

Provider provider = new SendMailFactory();

Sender sender = provider.produce();

sender.Send();

}

}

其实这个模式的好处就是，如果你现在想增加一个功能：发及时信息，则只需做一个实现类，实 Sender 接口，同时做一个工厂类，实现 Provider 接口，就 OK 了，无需去改动现成的代码。这样，拓展性较好！
3、单例模式（Singleton）
单例对象（Singleton）是一种常用的设计模式。在 Java 应用中，单例对象能保证在一个 JVM，该对象只有一个实例存在。这样的模式有几个好处：
1、某些类创建比较频繁，对于一些大型的对象，这是一笔很大的系统开销。
2、省去了 new 操作符，降低了系统内存的使用频率，减轻 GC 压力。
3、有些类如交易所的核心交易引擎，控制着交易流程，如果该类可以创建多个的话，系统完全了。（比如一个军队出现了多个司令员同时指挥，肯定会乱成一团），所以只有使用单例模式，才能证核心交易服务器独立控制整个流程。
首先我们写一个简单的单例类：
[java] view plaincopy

public class Singleton {

/* 持有私有静态实例，防止被引用，此处赋值为 null，目的是实现延迟加载 */

private static Singleton instance = null;

/* 私有构造方法，防止被实例化 */

private Singleton() {

}

/* 静态工程方法，创建实例 */

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton();

}

return instance;

}

/* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */

public Object readResolve() {

return instance;

}

}

这个类可以满足基本要求，但是，像这样毫无线程安全保护的类，如果我们把它放入多线程的环下，肯定就会出现问题了，如何解决？我们首先会想到对 getInstance 方法加 synchronized 关键字如下：
[java] view plaincopy

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

但是，synchronized 关键字锁住的是这个对象，这样的用法，在性能上会有所下降，因为每次用 getInstance()，都要对对象上锁，事实上，只有在第一次创建对象的时候需要加锁，之后就不需了，所以，这个地方需要改进。我们改成下面这个：
[java] view plaincopy

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

似乎解决了之前提到的问题，将 synchronized 关键字加在了内部，也就是说当调用的时候是不要加锁的，只有在 instance 为 null，并创建对象的时候才需要加锁，性能有一定的提升。但是，这的情况，还是有可能有问题的，看下面的情况：在 Java 指令中创建对象和赋值操作是分开进行的，就是说 instance = new Singleton();语句是分两步执行的。但是 JVM 并不保证这两个操作的先后顺，也就是说有可能 JVM 会为新的 Singleton 实例分配空间，然后直接赋值给 instance 成员，然后再初始化这个 Singleton 实例。这样就可能出错了，我们以 A、B 两个线程为例：
a>A、B 线程同时进入了第一个 if 判断
b>A 首先进入 synchronized 块，由于 instance 为 null，所以它执行 instance = new Singlton();
c> 由于 JVM 内部的优化机制，JVM 先画出了一些分配给 Singleton 实例的空白内存，并赋给 instance 成员（注意此时 JVM 没有开始初始化这个实例），然后 A 离开了 synchronized 块。</>
d>B 进入 synchronized 块，由于 instance 此时不是 null，因此它马上离开了 synchronized 块并将结果返回给调用该方法的程序。
e> 此时 B 线程打算使用 Singleton 实例，却发现它没有被初始化，于是错误发生了。
所以程序还是有可能发生错误，其实程序在运行过程是很复杂的，从这点我们就可以看出，尤其在写多线程环境下的程序更有难度，有挑战性。我们对该程序做进一步优化：
[java] view plaincopy

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

实际情况是，单例模式使用内部类来维护单例的实现，JVM 内部的机制能够保证当一个类被加的时候，这个类的加载过程是线程互斥的。这样当我们第一次调用 getInstance 的时候，JVM 能够我们保证 instance 只被创建一次，并且会保证把赋值给 instance 的内存初始化完毕，这样我们就不

担心上面的问题。同时该方法也只会在第一次调用的时候使用互斥机制，这样就解决了低性能问题。样我们暂时总结一个完美的单例模式：
[java] view plaincopy

public class Singleton {

/* 私有构造方法，防止被实例化 */

private Singleton() {

}

/* 此处使用一个内部类来维护单例 */

private static class SingletonFactory {

private static Singleton instance = new Singleton();

}

/* 获取实例 */

public static Singleton getInstance() {

return SingletonFactory.instance;

}

/* 如果该对象被用于序列化，可以保证对象在序列化前后保持一致 */

public Object readResolve() {

return getInstance();

}

}

其实说它完美，也不一定，如果在构造函数中抛出异常，实例将永远得不到创建，也会出错。所说，十分完美的东西是没有的，我们只能根据实际情况，选择最适合自己应用场景的实现方法。也有这样实现：因为我们只需要在创建类的时候进行同步，所以只要将创建和 getInstance()分开，单独创建加 synchronized 关键字，也是可以的：
[java] view plaincopy

public class SingletonTest {

private static SingletonTest instance = null;

private SingletonTest() {

}

private static synchronized void syncInit() {

if (instance == null) {

instance = new SingletonTest();

}

}

public static SingletonTest getInstance() {

if (instance == null) {

syncInit();

}

return instance;

}

}

考虑性能的话，整个程序只需创建一次实例，所以性能也不会有什么影响。