[**Java开发中的23种设计模式详解(转)**](http://www.cnblogs.com/maowang1991/archive/2013/04/15/3023236.html)

**设计模式（Design Patterns）**

                                  ——可复用面向对象软件的基础

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、代码设计经验的总结。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，设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，每种模式在现在中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，以及该问题的核心解决方案，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，学好设计模式，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=372668&uk=4076915866#dir/path=%2F%E5%AD%A6%E4%B9%A0%E6%96%87%E4%BB%B6>

**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、享元模式。

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、命令模式、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，不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，易于维护和升级。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。 里氏代换原则中说，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换掉基类，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新的行为。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。而基类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的意思，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，为了升级和维护方便。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，使得系统功能模块相对独立。

**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，并结合他们的特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，对实现了同一接口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](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public interface Sender {
2. public void Send();
3. }

其次，创建实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class MailSender implements Sender {
2. @Override
3. public void Send() {
4. System.out.println("this is mailsender!");
5. }
6. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class SmsSender implements Sender {
3. @Override
4. public void Send() {
5. System.out.println("this is sms sender!");
6. }
7. }

最后，建工厂类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

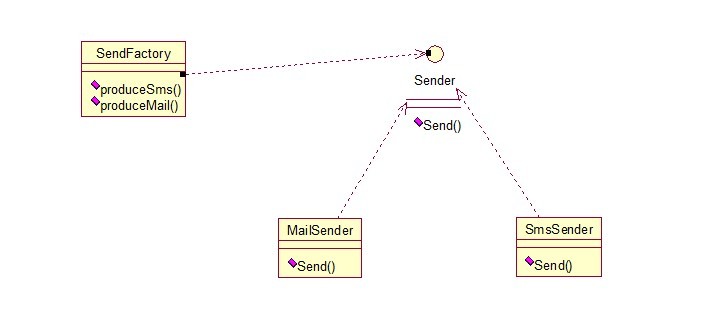
1. public class SendFactory {
3. public Sender produce(String type) {
4. if ("mail".equals(type)) {
5. return new MailSender();
6. } else if ("sms".equals(type)) {
7. return new SmsSender();
8. } else {
9. System.out.println("请输入正确的类型!");
10. return null;
11. }
12. }
13. }

我们来测试下：

1. public class FactoryTest {
3. public static void main(String[] args) {
4. SendFactory factory = new SendFactory();
5. Sender sender = factory.produce("sms");
6. sender.Send();
7. }
8. }

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，在普通工厂方法模式中，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工厂方法，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 plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)public class SendFactory {

   public Sender produceMail(){

1. return new MailSender();
2. }
4. public Sender produceSms(){
5. return new SmsSender();
6. }
7. }

测试类如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class FactoryTest {
3. public static void main(String[] args) {
4. SendFactory factory = new SendFactory();
5. Sender sender = factory.produceMail();
6. sender.Send();
7. }
8. }

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法置为静态的，不需要创建实例，直接调用即可。

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class SendFactory {
3. public static Sender produceMail(){
4. return new MailSender();
5. }
7. public static Sender produceSms(){
8. return new SmsSender();
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

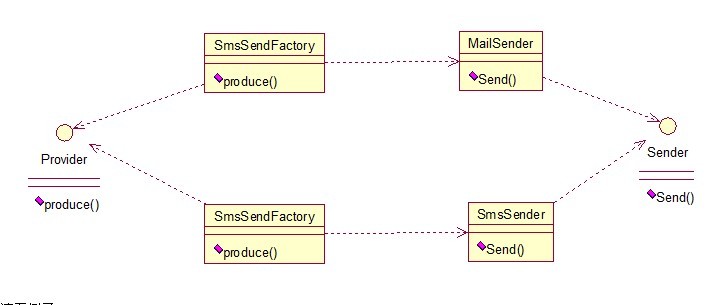
1. public class FactoryTest {
3. public static void main(String[] args) {
4. Sender sender = SendFactory.produceMail();
5. sender.Send();
6. }
7. }

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，第三种相对于第二种，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——静态工厂方法模式。

**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，必须对工厂类进行修改，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？就用到抽象工厂模式，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，不需要修改之前的代码。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](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public interface Sender {
2. public void Send();
3. }

两个实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class MailSender implements Sender {
2. @Override
3. public void Send() {
4. System.out.println("this is mailsender!");
5. }
6. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class SmsSender implements Sender {
3. @Override
4. public void Send() {
5. System.out.println("this is sms sender!");
6. }
7. }

两个工厂类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class SendMailFactory implements Provider {
3. @Override
4. public Sender produce(){
5. return new MailSender();
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class SendSmsFactory implements Provider{
3. @Override
4. public Sender produce() {
5. return new SmsSender();
6. }
7. }

在提供一个接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public interface Provider {
2. public Sender produce();
3. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class Test {
3. public static void main(String[] args) {
4. Provider provider = new SendMailFactory();
5. Sender sender = provider.produce();
6. sender.Send();
7. }
8. }

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，实现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。这样做，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中，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，系统完全乱了。（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，才能保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](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class Singleton {
3. /\* 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 \*/
4. private static Singleton instance = null;
6. /\* 私有构造方法，防止被实例化 \*/
7. private Singleton() {
8. }
10. /\* 静态工程方法，创建实例 \*/
11. public static Singleton getInstance() {
12. if (instance == null) {
13. instance = new Singleton();
14. }
15. return instance;
16. }
18. /\* 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 \*/
19. public Object readResolve() {
20. return instance;
21. }
22. }

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境下，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](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public static synchronized Singleton getInstance() {
2. if (instance == null) {
3. instance = new Singleton();
4. }
5. return instance;
6. }

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，因为每次调用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，之后就不需要了，所以，这个地方需要改进。我们改成下面这个：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public static Singleton getInstance() {
2. if (instance == null) {
3. synchronized (instance) {
4. if (instance == null) {
5. instance = new Singleton();
6. }
7. }
8. }
9. return instance;
10. }

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需要加锁的，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，这样的情况，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，也就是说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序，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，然后再去初始化这个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 Singleton();

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，并赋值给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，尤其是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](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. private static class SingletonFactory{
2. private static Singleton instance = new Singleton();
3. }
4. public static Singleton getInstance(){
5. return SingletonFactory.instance;
6. }

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载的时候，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能够帮我们保证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，这样我们就不用担心上面的问题。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。这样我们暂时总结一个完美的单例模式：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class Singleton {
3. /\* 私有构造方法，防止被实例化 \*/
4. private Singleton() {
5. }
7. /\* 此处使用一个内部类来维护单例 \*/
8. private static class SingletonFactory {
9. private static Singleton instance = new Singleton();
10. }
12. /\* 获取实例 \*/
13. public static Singleton getInstance() {
14. return SingletonFactory.instance;
15. }
17. /\* 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 \*/
18. public Object readResolve() {
19. return getInstance();
20. }
21. }

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。所以说，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。也有人这样实现：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，单独为创建加synchronized关键字，也是可以的：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class SingletonTest {
3. private static SingletonTest instance = null;
5. private SingletonTest() {
6. }
8. private static synchronized void syncInit() {
9. if (instance == null) {
10. instance = new SingletonTest();
11. }
12. }
14. public static SingletonTest getInstance() {
15. if (instance == null) {
16. syncInit();
17. }
18. return instance;
19. }
20. }

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。

**补充：采用"影子实例"的办法为单例对象的属性同步更新**

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class SingletonTest {
3. private static SingletonTest instance = null;
4. private Vector properties = null;
6. public Vector getProperties() {
7. return properties;
8. }
10. private SingletonTest() {
11. }
13. private static synchronized void syncInit() {
14. if (instance == null) {
15. instance = new SingletonTest();
16. }
17. }
19. public static SingletonTest getInstance() {
20. if (instance == null) {
21. syncInit();
22. }
23. return instance;
24. }
26. public void updateProperties() {
27. SingletonTest shadow = new SingletonTest();
28. properties = shadow.getProperties();
29. }
30. }

通过单例模式的学习告诉我们：

1、单例模式理解起来简单，但是具体实现起来还是有一定的难度。

2、synchronized关键字锁定的是对象，在用的时候，一定要在恰当的地方使用（注意需要使用锁的对象和过程，可能有的时候并不是整个对象及整个过程都需要锁）。

到这儿，单例模式基本已经讲完了，结尾处，笔者突然想到另一个问题，就是采用类的静态方法，实现单例模式的效果，也是可行的，此处二者有什么不同？

首先，静态类不能实现接口。（从类的角度说是可以的，但是那样就破坏了静态了。因为接口中不允许有static修饰的方法，所以即使实现了也是非静态的）

其次，单例可以被延迟初始化，静态类一般在第一次加载是初始化。之所以延迟加载，是因为有些类比较庞大，所以延迟加载有助于提升性能。

再次，单例类可以被继承，他的方法可以被覆写。但是静态类内部方法都是static，无法被覆写。

最后一点，单例类比较灵活，毕竟从实现上只是一个普通的Java类，只要满足单例的基本需求，你可以在里面随心所欲的实现一些其它功能，但是静态类不行。从上面这些概括中，基本可以看出二者的区别，但是，从另一方面讲，我们上面最后实现的那个单例模式，内部就是用一个静态类来实现的，所以，二者有很大的关联，只是我们考虑问题的层面不同罢了。两种思想的结合，才能造就出完美的解决方案，就像HashMap采用数组+链表来实现一样，其实生活中很多事情都是这样，单用不同的方法来处理问题，总是有优点也有缺点，最完美的方法是，结合各个方法的优点，才能最好的解决问题！

**4、建造者模式（Builder）**

工厂类模式提供的是创建单个类的模式，而建造者模式则是将各种产品集中起来进行管理，用来创建复合对象，所谓复合对象就是指某个类具有不同的属性，其实建造者模式就是前面抽象工厂模式和最后的Test结合起来得到的。我们看一下代码：

还和前面一样，一个Sender接口，两个实现类MailSender和SmsSender。最后，建造者类如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class Builder {
3. private List<Sender> list = new ArrayList<Sender>();
5. public void produceMailSender(int count){
6. for(int i=0; i<count; i++){
7. list.add(new MailSender());
8. }
9. }
11. public void produceSmsSender(int count){
12. for(int i=0; i<count; i++){
13. list.add(new SmsSender());
14. }
15. }
16. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class Test {
3. public static void main(String[] args) {
4. Builder builder = new Builder();
5. builder.produceMailSender(10);
6. }
7. }

从这点看出，建造者模式将很多功能集成到一个类里，这个类可以创造出比较复杂的东西。所以与工程模式的区别就是：工厂模式关注的是创建单个产品，而建造者模式则关注创建符合对象，多个部分。因此，是选择工厂模式还是建造者模式，依实际情况而定。

**5、原型模式（Prototype）**

原型模式虽然是创建型的模式，但是与工程模式没有关系，从名字即可看出，该模式的思想就是将一个对象作为原型，对其进行复制、克隆，产生一个和原对象类似的新对象。本小结会通过对象的复制，进行讲解。在Java中，复制对象是通过clone()实现的，先创建一个原型类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

1. public class Prototype implements Cloneable {
3. public Object clone() throws CloneNotSupportedException {
4. Prototype proto = (Prototype) super.clone();
5. return proto;
6. }
7. }

很简单，一个原型类，只需要实现Cloneable接口，覆写clone方法，此处clone方法可以改成任意的名称，因为Cloneable接口是个空接口，你可以任意定义实现类的方法名，如cloneA或者cloneB，因为此处的重点是super.clone()这句话，super.clone()调用的是Object的clone()方法，而在Object类中，clone()是native的，具体怎么实现，我会在另一篇文章中，关于解读Java中本地方法的调用，此处不再深究。在这儿，我将结合对象的浅复制和深复制来说一下，首先需要了解对象深、浅复制的概念：

浅复制：将一个对象复制后，基本数据类型的变量都会重新创建，而引用类型，指向的还是原对象所指向的。

深复制：将一个对象复制后，不论是基本数据类型还有引用类型，都是重新创建的。简单来说，就是深复制进行了完全彻底的复制，而浅复制不彻底。

此处，写一个深浅复制的例子：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

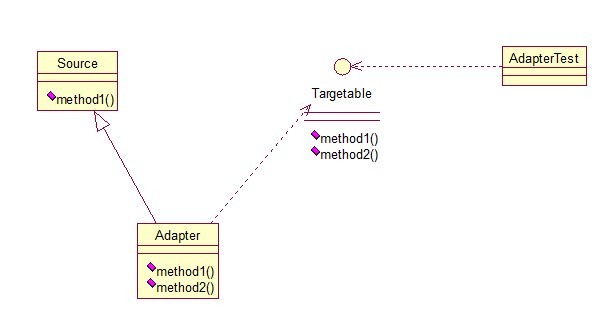
1. public class Prototype implements Cloneable, Serializable {
3. private static final long serialVersionUID = 1L;
4. private String string;
6. private SerializableObject obj;
8. /\* 浅复制 \*/
9. public Object clone() throws CloneNotSupportedException {
10. Prototype proto = (Prototype) super.clone();
11. return proto;
12. }
14. /\* 深复制 \*/
15. public Object deepClone() throws IOException, ClassNotFoundException {
17. /\* 写入当前对象的二进制流 \*/
18. ByteArrayOutputStream bos = new ByteArrayOutputStream();
19. ObjectOutputStream oos = new ObjectOutputStream(bos);
20. oos.writeObject(this);
22. /\* 读出二进制流产生的新对象 \*/
23. ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArray());
24. ObjectInputStream ois = new ObjectInputStream(bis);
25. return ois.readObject();
26. }
28. public String getString() {
29. return string;
30. }
32. public void setString(String string) {
33. this.string = string;
34. }
36. public SerializableObject getObj() {
37. return obj;
38. }
40. public void setObj(SerializableObject obj) {
41. this.obj = obj;
42. }
44. }
46. class SerializableObject implements Serializable {
47. private static final long serialVersionUID = 1L;
48. }

要实现深复制，需要采用流的形式读入当前对象的二进制输入，再写出二进制数据对应的对象。

我们接着讨论设计模式，上篇文章我讲完了5种创建型模式，这章开始，我将讲下7种结构型模式：适配器模式、装饰模式、代理模式、外观模式、桥接模式、组合模式、享元模式。其中对象的适配器模式是各种模式的起源，我们看下面的图：



 适配器模式将某个类的接口转换成客户端期望的另一个接口表示，目的是消除由于接口不匹配所造成的类的兼容性问题。主要分为三类：类的适配器模式、对象的适配器模式、接口的适配器模式。首先，我们来看看**类的适配器模式**，先看类图：



核心思想就是：有一个Source类，拥有一个方法，待适配，目标接口时Targetable，通过Adapter类，将Source的功能扩展到Targetable里，看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Source {
3. public void method1() {
4. System.out.println("this is original method!");
5. }
6. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public interface Targetable {
3. /\* 与原类中的方法相同 \*/
4. public void method1();
6. /\* 新类的方法 \*/
7. public void method2();
8. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Adapter extends Source implements Targetable {
3. @Override
4. public void method2() {
5. System.out.println("this is the targetable method!");
6. }
7. }

Adapter类继承Source类，实现Targetable接口，下面是测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class AdapterTest {
3. public static void main(String[] args) {
4. Targetable target = new Adapter();
5. target.method1();
6. target.method2();
7. }
8. }

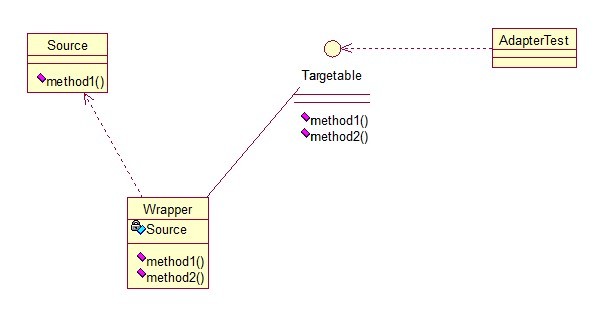
输出：

this is original method!  
this is the targetable method!

这样Targetable接口的实现类就具有了Source类的功能。

**对象的适配器模式**

基本思路和类的适配器模式相同，只是将Adapter类作修改，这次不继承Source类，而是持有Source类的实例，以达到解决兼容性的问题。看图：



只需要修改Adapter类的源码即可：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Wrapper implements Targetable {
3. private Source source;
5. public Wrapper(Source source){
6. super();
7. this.source = source;
8. }
9. @Override
10. public void method2() {
11. System.out.println("this is the targetable method!");
12. }
14. @Override
15. public void method1() {
16. source.method1();
17. }
18. }

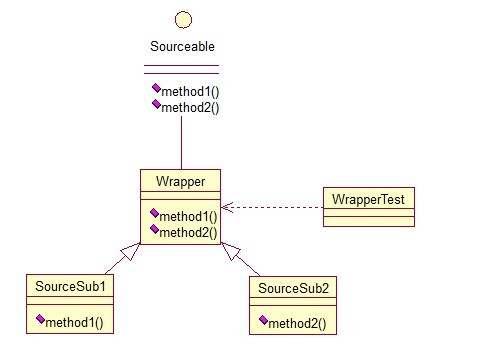
测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class AdapterTest {
3. public static void main(String[] args) {
4. Source source = new Source();
5. Targetable target = new Wrapper(source);
6. target.method1();
7. target.method2();
8. }
9. }

输出与第一种一样，只是适配的方法不同而已。

第三种适配器模式是**接口的适配器模式**，接口的适配器是这样的：有时我们写的一个接口中有多个抽象方法，当我们写该接口的实现类时，必须实现该接口的所有方法，这明显有时比较浪费，因为并不是所有的方法都是我们需要的，有时只需要某一些，此处为了解决这个问题，我们引入了接口的适配器模式，借助于一个抽象类，该抽象类实现了该接口，实现了所有的方法，而我们不和原始的接口打交道，只和该抽象类取得联系，所以我们写一个类，继承该抽象类，重写我们需要的方法就行。看一下类图：



这个很好理解，在实际开发中，我们也常会遇到这种接口中定义了太多的方法，以致于有时我们在一些实现类中并不是都需要。看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public interface Sourceable {
3. public void method1();
4. public void method2();
5. }

抽象类Wrapper2：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public abstract class Wrapper2 implements Sourceable{
3. public void method1(){}
4. public void method2(){}
5. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class SourceSub1 extends Wrapper2 {
2. public void method1(){
3. System.out.println("the sourceable interface's first Sub1!");
4. }
5. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class SourceSub2 extends Wrapper2 {
2. public void method2(){
3. System.out.println("the sourceable interface's second Sub2!");
4. }
5. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class WrapperTest {
3. public static void main(String[] args) {
4. Sourceable source1 = new SourceSub1();
5. Sourceable source2 = new SourceSub2();
7. source1.method1();
8. source1.method2();
9. source2.method1();
10. source2.method2();
11. }
12. }

测试输出：

the sourceable interface's first Sub1!  
the sourceable interface's second Sub2!

达到了我们的效果！

 讲了这么多，总结一下三种适配器模式的应用场景：

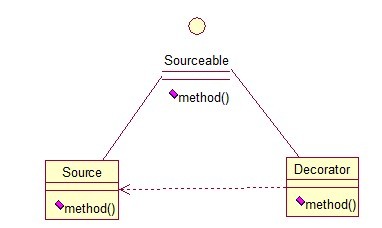
类的适配器模式：当希望将**一个类**转换成满足**另一个新接口**的类时，可以使用类的适配器模式，创建一个新类，继承原有的类，实现新的接口即可。

对象的适配器模式：当希望将一个对象转换成满足另一个新接口的对象时，可以创建一个Wrapper类，持有原类的一个实例，在Wrapper类的方法中，调用实例的方法就行。

接口的适配器模式：当不希望实现一个接口中所有的方法时，可以创建一个抽象类Wrapper，实现所有方法，我们写别的类的时候，继承抽象类即可。

**7、装饰模式（Decorator）**

顾名思义，装饰模式就是给一个对象增加一些新的功能，而且是动态的，要求装饰对象和被装饰对象实现同一个接口，装饰对象持有被装饰对象的实例，关系图如下：



Source类是被装饰类，Decorator类是一个装饰类，可以为Source类动态的添加一些功能，代码如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public interface Sourceable {
2. public void method();
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Source implements Sourceable {
3. @Override
4. public void method() {
5. System.out.println("the original method!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Decorator implements Sourceable {
3. private Sourceable source;
5. public Decorator(Sourceable source){
6. super();
7. this.source = source;
8. }
9. @Override
10. public void method() {
11. System.out.println("before decorator!");
12. source.method();
13. System.out.println("after decorator!");
14. }
15. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class DecoratorTest {
3. public static void main(String[] args) {
4. Sourceable source = new Source();
5. Sourceable obj = new Decorator(source);
6. obj.method();
7. }
8. }

输出：

before decorator!  
the original method!  
after decorator!

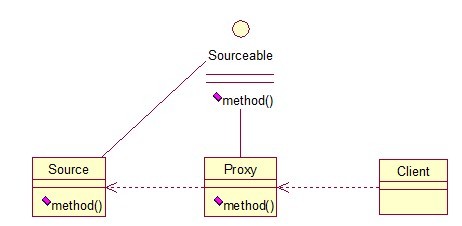
装饰器模式的应用场景：

1、需要扩展一个类的功能。

2、动态的为一个对象增加功能，而且还能动态撤销。（继承不能做到这一点，继承的功能是静态的，不能动态增删。）

缺点：产生过多相似的对象，不易排错！

**8、代理模式（Proxy）**

其实每个模式名称就表明了该模式的作用，代理模式就是多一个代理类出来，替原对象进行一些操作，比如我们在租房子的时候回去找中介，为什么呢？因为你对该地区房屋的信息掌握的不够全面，希望找一个更熟悉的人去帮你做，此处的代理就是这个意思。再如我们有的时候打官司，我们需要请律师，因为律师在法律方面有专长，可以替我们进行操作，表达我们的想法。先来看看关系图：

根据上文的阐述，代理模式就比较容易的理解了，我们看下代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public interface Sourceable {
2. public void method();
3. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Source implements Sourceable {
3. @Override
4. public void method() {
5. System.out.println("the original method!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Proxy implements Sourceable {
3. private Source source;
4. public Proxy(){
5. super();
6. this.source = new Source();
7. }
8. @Override
9. public void method() {
10. before();
11. source.method();
12. atfer();
13. }
14. private void atfer() {
15. System.out.println("after proxy!");
16. }
17. private void before() {
18. System.out.println("before proxy!");
19. }
20. }

测试类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class ProxyTest {
3. public static void main(String[] args) {
4. Sourceable source = new Proxy();
5. source.method();
6. }
8. }

输出：

before proxy!  
the original method!  
after proxy!

代理模式的应用场景：

如果已有的方法在使用的时候需要对原有的方法进行改进，此时有两种办法：

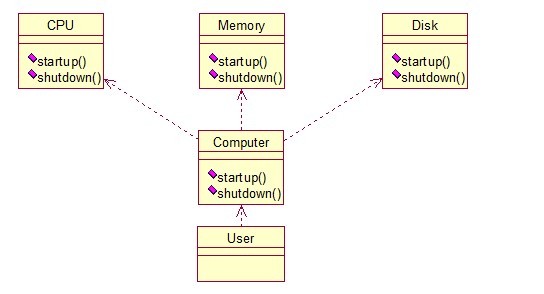
1、修改原有的方法来适应。这样违反了“对扩展开放，对修改关闭”的原则。

2、就是采用一个代理类调用原有的方法，且对产生的结果进行控制。这种方法就是代理模式。

使用代理模式，可以将功能划分的更加清晰，有助于后期维护！

**9、外观模式（Facade）**

外观模式是为了解决类与类之家的依赖关系的，像spring一样，可以将类和类之间的关系配置到配置文件中，而外观模式就是将他们的关系放在一个Facade类中，降低了类类之间的耦合度，该模式中没有涉及到接口，看下类图：（我们以一个计算机的启动过程为例）



我们先看下实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class CPU {
3. public void startup(){
4. System.out.println("cpu startup!");
5. }
7. public void shutdown(){
8. System.out.println("cpu shutdown!");
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Memory {
3. public void startup(){
4. System.out.println("memory startup!");
5. }
7. public void shutdown(){
8. System.out.println("memory shutdown!");
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Disk {
3. public void startup(){
4. System.out.println("disk startup!");
5. }
7. public void shutdown(){
8. System.out.println("disk shutdown!");
9. }
10. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Computer {
2. private CPU cpu;
3. private Memory memory;
4. private Disk disk;
6. public Computer(){
7. cpu = new CPU();
8. memory = new Memory();
9. disk = new Disk();
10. }
12. public void startup(){
13. System.out.println("start the computer!");
14. cpu.startup();
15. memory.startup();
16. disk.startup();
17. System.out.println("start computer finished!");
18. }
20. public void shutdown(){
21. System.out.println("begin to close the computer!");
22. cpu.shutdown();
23. memory.shutdown();
24. disk.shutdown();
25. System.out.println("computer closed!");
26. }
27. }

User类如下：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class User {
3. public static void main(String[] args) {
4. Computer computer = new Computer();
5. computer.startup();
6. computer.shutdown();
7. }
8. }

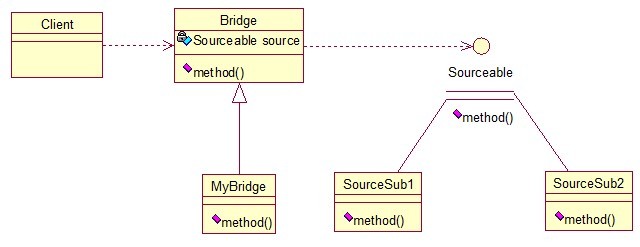
输出：

start the computer!  
cpu startup!  
memory startup!  
disk startup!  
start computer finished!  
begin to close the computer!  
cpu shutdown!  
memory shutdown!  
disk shutdown!  
computer closed!

如果我们没有Computer类，那么，CPU、Memory、Disk他们之间将会相互持有实例，产生关系，这样会造成严重的依赖，修改一个类，可能会带来其他类的修改，这不是我们想要看到的，有了Computer类，他们之间的关系被放在了Computer类里，这样就起到了解耦的作用，这，就是外观模式！

**10、桥接模式（Bridge）**

桥接模式就是把事物和其具体实现分开，使他们可以各自独立的变化。桥接的用意是：**将抽象化与实现化解耦，使得二者可以独立变化**，像我们常用的JDBC桥DriverManager一样，JDBC进行连接数据库的时候，在各个数据库之间进行切换，基本不需要动太多的代码，甚至丝毫不用动，原因就是JDBC提供统一接口，每个数据库提供各自的实现，用一个叫做数据库驱动的程序来桥接就行了。我们来看看关系图：



实现代码：

先定义接口：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public interface Sourceable {
2. public void method();
3. }

分别定义两个实现类：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class SourceSub1 implements Sourceable {
3. @Override
4. public void method() {
5. System.out.println("this is the first sub!");
6. }
7. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class SourceSub2 implements Sourceable {
3. @Override
4. public void method() {
5. System.out.println("this is the second sub!");
6. }
7. }

定义一个桥，持有Sourceable的一个实例：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public abstract class Bridge {
2. private Sourceable source;
4. public void method(){
5. source.method();
6. }
8. public Sourceable getSource() {
9. return source;
10. }
12. public void setSource(Sourceable source) {
13. this.source = source;
14. }
15. }

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class MyBridge extends Bridge {
2. public void method(){
3. getSource().method();
4. }
5. }

测试类：

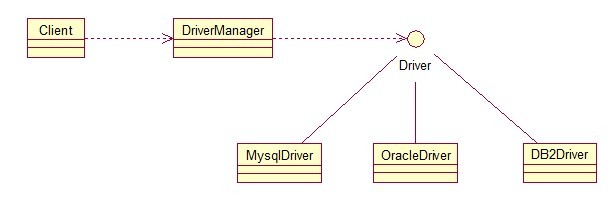
**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class BridgeTest {
3. public static void main(String[] args) {
5. Bridge bridge = new MyBridge();
7. /\*调用第一个对象\*/
8. Sourceable source1 = new SourceSub1();
9. bridge.setSource(source1);
10. bridge.method();
12. /\*调用第二个对象\*/
13. Sourceable source2 = new SourceSub2();
14. bridge.setSource(source2);
15. bridge.method();
16. }
17. }

output：

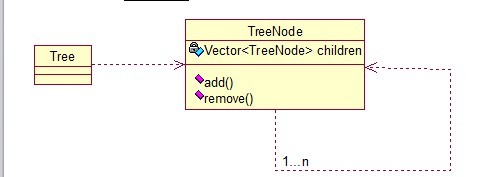
this is the first sub!  
this is the second sub!

这样，就通过对Bridge类的调用，实现了对接口Sourceable的实现类SourceSub1和SourceSub2的调用。接下来我再画个图，大家就应该明白了，因为这个图是我们JDBC连接的原理，有数据库学习基础的，一结合就都懂了。



**11、组合模式（Composite）**

组合模式有时又叫**部分-整体**模式在处理类似树形结构的问题时比较方便，看看关系图：



直接来看代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class TreeNode {
3. private String name;
4. private TreeNode parent;
5. private Vector<TreeNode> children = new Vector<TreeNode>();
7. public TreeNode(String name){
8. this.name = name;
9. }
11. public String getName() {
12. return name;
13. }
15. public void setName(String name) {
16. this.name = name;
17. }
19. public TreeNode getParent() {
20. return parent;
21. }
23. public void setParent(TreeNode parent) {
24. this.parent = parent;
25. }
27. //添加孩子节点
28. public void add(TreeNode node){
29. children.add(node);
30. }
32. //删除孩子节点
33. public void remove(TreeNode node){
34. children.remove(node);
35. }
37. //取得孩子节点
38. public Enumeration<TreeNode> getChildren(){
39. return children.elements();
40. }
41. }

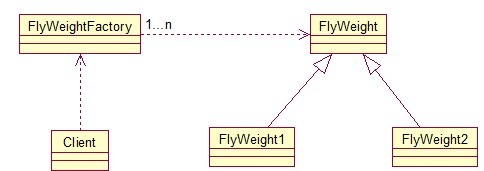
**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class Tree {
3. TreeNode root = null;
5. public Tree(String name) {
6. root = new TreeNode(name);
7. }
9. public static void main(String[] args) {
10. Tree tree = new Tree("A");
11. TreeNode nodeB = new TreeNode("B");
12. TreeNode nodeC = new TreeNode("C");
14. nodeB.add(nodeC);
15. tree.root.add(nodeB);
16. System.out.println("build the tree finished!");
17. }
18. }

使用场景：将多个对象组合在一起进行操作，常用于表示树形结构中，例如二叉树，数等。

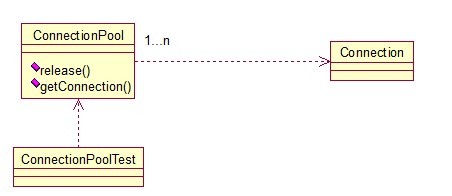
**12、享元模式（Flyweight）**

享元模式的主要目的是实现对象的共享，即共享池，当系统中对象多的时候可以减少内存的开销，通常与工厂模式一起使用。



FlyWeightFactory负责创建和管理享元单元，当一个客户端请求时，工厂需要检查当前对象池中是否有符合条件的对象，如果有，就返回已经存在的对象，如果没有，则创建一个新对象，FlyWeight是超类。一提到共享池，我们很容易联想到Java里面的JDBC连接池，想想每个连接的特点，我们不难总结出：适用于作共享的一些个对象，他们有一些共有的属性，就拿数据库连接池来说，url、driverClassName、username、password及dbname，这些属性对于每个连接来说都是一样的，所以就适合用享元模式来处理，建一个工厂类，将上述类似属性作为内部数据，其它的作为外部数据，在方法调用时，当做参数传进来，这样就节省了空间，减少了实例的数量。

看个例子：



看下数据库连接池的代码：

**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8239539)

1. public class ConnectionPool {
3. private Vector<Connection> pool;
5. /\*公有属性\*/
6. private String url = "jdbc:mysql://localhost:3306/test";
7. private String username = "root";
8. private String password = "root";
9. private String driverClassName = "com.mysql.jdbc.Driver";
11. private int poolSize = 100;
12. private static ConnectionPool instance = null;
13. Connection conn = null;
15. /\*构造方法，做一些初始化工作\*/
16. private ConnectionPool() {
17. pool = new Vector<Connection>(poolSize);
19. for (int i = 0; i < poolSize; i++) {
20. try {
21. Class.forName(driverClassName);
22. conn = DriverManager.getConnection(url, username, password);
23. pool.add(conn);
24. } catch (ClassNotFoundException e) {
25. e.printStackTrace();
26. } catch (SQLException e) {
27. e.printStackTrace();
28. }
29. }
30. }
32. /\* 返回连接到连接池 \*/
33. public synchronized void release() {
34. pool.add(conn);
35. }
37. /\* 返回连接池中的一个数据库连接 \*/
38. public synchronized Connection getConnection() {
39. if (pool.size() > 0) {
40. Connection conn = pool.get(0);
41. pool.remove(conn);
42. return conn;
43. } else {
44. return null;
45. }
46. }
47. }

通过连接池的管理，实现了数据库连接的共享，不需要每一次都重新创建连接，节省了数据库重新创建的开销，提升了系统的性能！本章讲解了7种结构型模式，因为篇幅的问题，剩下的11种行为型模式，

本章是关于设计模式的最后一讲，会讲到第三种设计模式——行为型模式，共11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、命令模式、备忘录模式、状态模式、访问者模式、中介者模式、解释器模式。这段时间一直在写关于设计模式的东西，终于写到一半了，写博文是个很费时间的东西，因为我得为读者负责，不论是图还是代码还是表述，都希望能尽量写清楚，以便读者理解，我想不论是我还是读者，都希望看到高质量的博文出来，从我本人出发，我会一直坚持下去，不断更新，源源动力来自于读者朋友们的不断支持，我会尽自己的努力，写好每一篇文章！希望大家能不断给出意见和建议，共同打造完美的博文！

先来张图，看看这11中模式的关系：

第一类：通过父类与子类的关系进行实现。第二类：两个类之间。第三类：类的状态。第四类：通过中间类

