**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5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7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。

其实还有两类：并发型模式和线程池模式。

**二、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(垃圾回收）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

**三、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，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：

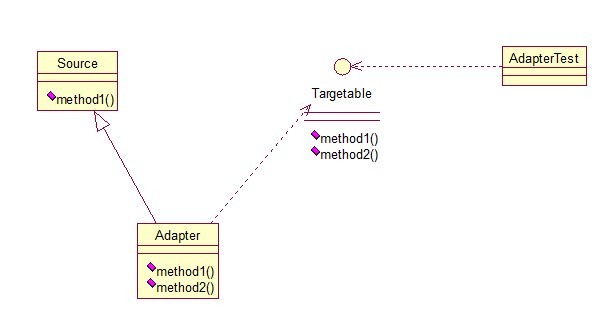
**[java]** [view plaincopy](http://blog.csdn.net/zhangerqing/article/details/8194653)

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。

**三、适配器模式(Adapter Class/Object)**

 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：



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！

 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。

**引用：**

**http://www.cnblogs.com/maowang1991/archive/2013/04/15/3023236.html**