Java之美[从菜鸟到高手演变]之设计模式

2012年11月29日 02:26:22 终点 阅读数：361598 标签： java 系统架构 设计模式

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！

在阅读过程中有任何问题，请及时联系：egg。

邮箱：xtfggef@gmail.com 微博：http://weibo.com/xtfggef

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：

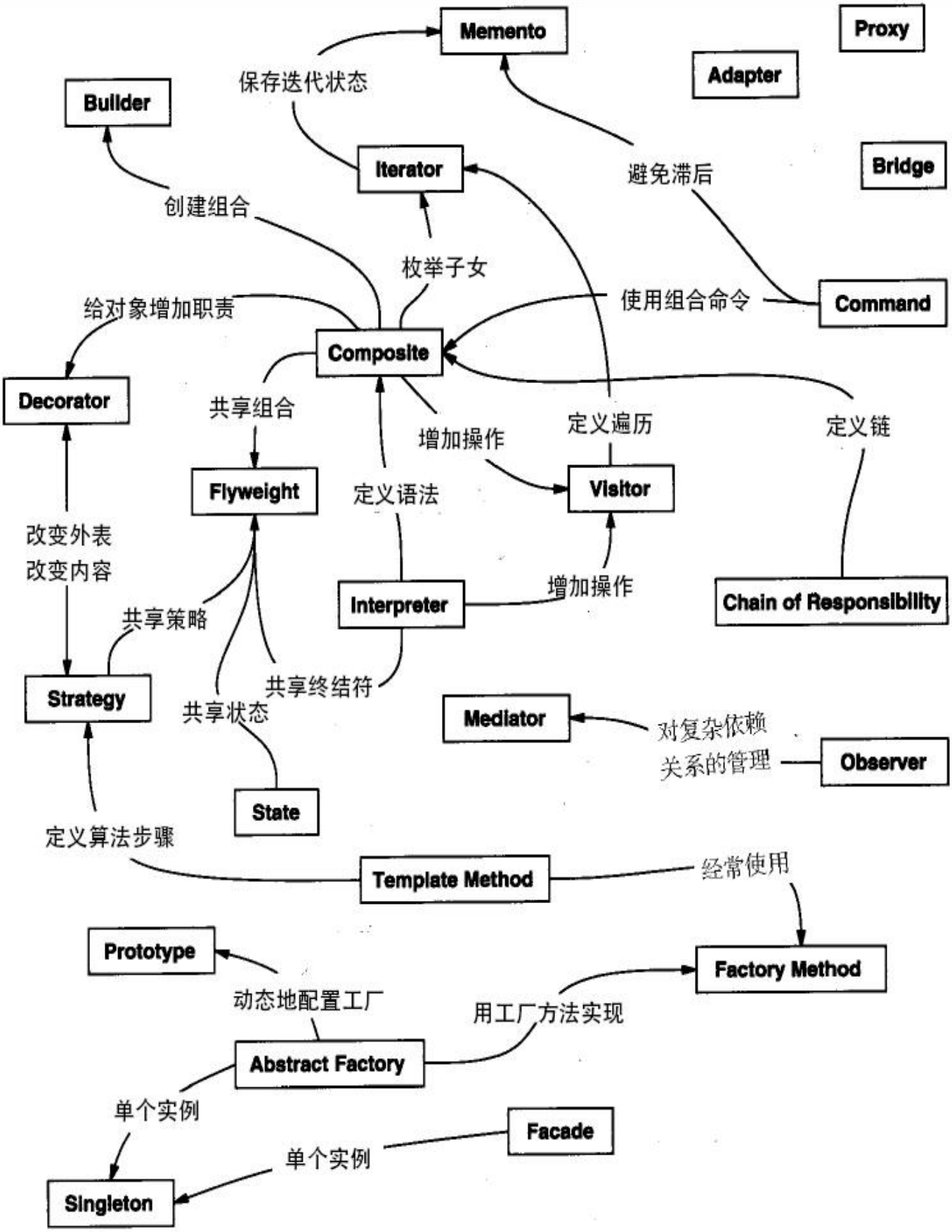


图　设计模式之间的关系

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。

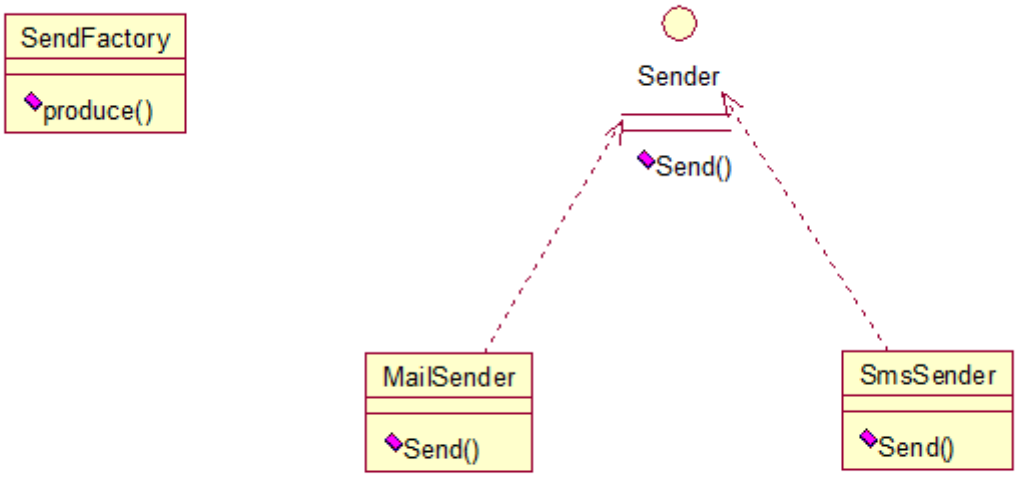
三、Java的23中设计模式

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：

```
public interface Sender {
        public void Send();
}
```

其次，创建实现类：

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

```
public class SmsSender implements Sender {

        @Override
        public void Send() {
```

```
                System.out.println("this is sms sender!");
        }
}
```

最后，建工厂类：

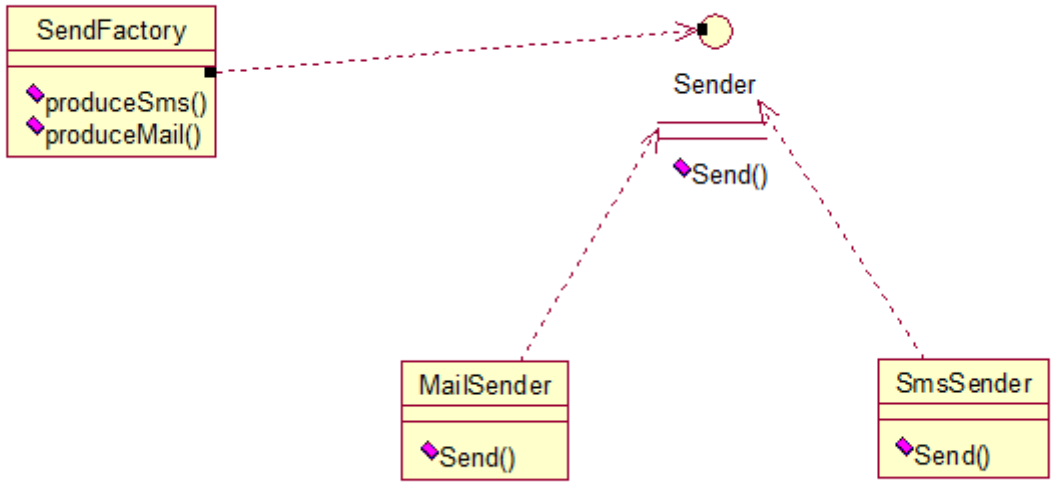
```
public class SendFactory {

        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：

```
public class SendFactory {

        public Sender produceMail(){
                return new MailSender();
        }

        public Sender produceSms(){
                return new SmsSender();
        }
}
```

测试类如下：

```
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法置为静态的，不需要创建实例，直接调用即可。

```
public class SendFactory {

        public static Sender produceMail(){
                return new MailSender();
        }

        public static Sender produceSms(){
                return new SmsSender();
        }
}
```

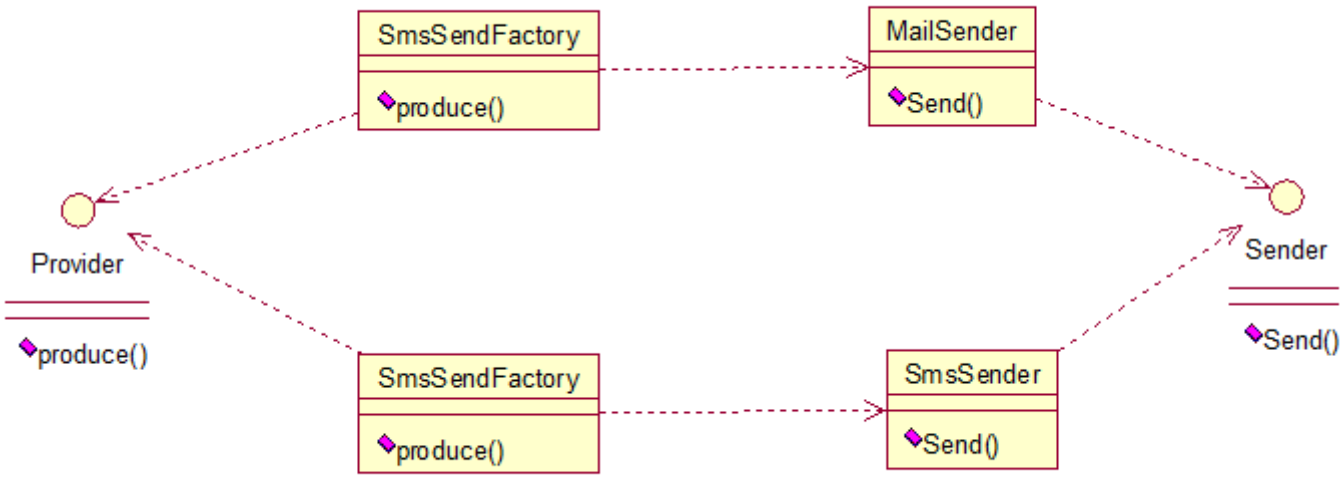
```
public class FactoryTest {

        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：

```
public interface Sender {
        public void Send();
}
```

两个实现类：

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

```
public class SmsSender implements Sender {

        @Override
        public void Send() {
                System.out.println("this is sms sender!");
        }
}
```

两个工厂类：

```
public class SendMailFactory implements Provider {

        @Override
        public Sender produce(){
                return new MailSender();
        }
}
```

```
public class SendSmsFactory implements Provider{

        @Override
        public Sender produce() {
                return new SmsSender();
        }
}
```

在提供一个接口：

```
public interface Provider {
        public Sender produce();
}
```

测试类：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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：

```
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。

补充：采用"影子实例"的办法为单例对象的属性同步更新

```
public class SingletonTest {

        private static SingletonTest instance = null;
        private Vector properties = null;

        public Vector getProperties() {
                return properties;
        }

        private SingletonTest() {
        }

        private static synchronized void syncInit() {
                if (instance == null) {
                        instance = new SingletonTest();
                }
        }
```

```
        public static SingletonTest getInstance() {

             if (instance == null) {                    syncInit();
                }
                return instance;
        }

        public void updateProperties() {
                SingletonTest shadow = new SingletonTest();
                properties = shadow.getProperties();
        }
}
```

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：

```
public class Builder {

        private List<Sender> list = new ArrayList<Sender>();

        public void produceMailSender(int count){
                for(int i=0; i<count; i++){
                        list.add(new MailSender());
                }
        }

        public void produceSmsSender(int count){
                for(int i=0; i<count; i++){
                        list.add(new SmsSender());
                }
        }
}
```

测试类：

```
public class Test {

        public static void main(String[] args) {
                Builder builder = new Builder();
                builder.produceMailSender(10);
        }
}
```

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：

```
public class Prototype implements Cloneable {

        public Object clone() throws CloneNotSupportedException {
                Prototype proto = (Prototype) super.clone();
                return proto;
        }
}
```

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：

```
public class Prototype implements Cloneable, Serializable {

        private static final long serialVersionUID = 1L;
        private String string;

        private SerializableObject obj;

        /* 浅复制 */
        public Object clone() throws CloneNotSupportedException {
                Prototype proto = (Prototype) super.clone();
                return proto;
        }

        /* 深复制 */
        public Object deepClone() throws IOException, ClassNotFoundException {

                /* 写入当前对象的二进制流 */
                ByteArrayOutputStream bos = new ByteArrayOutputStream();
                ObjectOutputStream oos = new ObjectOutputStream(bos);
```

```
                oos.writeObject(this);
                /* 读出二进制流产生的新对象 */

            ByteArrayInputStream bis = new ByteArrayInputStream(bos.toByteArra
                ObjectInputStream ois = new ObjectInputStream(bis);
            return ois.readObject();        }

        public String getString() {
                return string;
        }

        public void setString(String string) {
                this.string = string;
        }

        public SerializableObject getObj() {
                return obj;
        }

        public void setObj(SerializableObject obj) {
                this.obj = obj;
        }

}

class SerializableObject implements Serializable {
        private static final long serialVersionUID = 1L;
}
```

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。

由于文章篇幅较长，为了更好的方便读者阅读，我将接下了的其它介绍放在另一篇文章中（也许会分两篇来），感谢大家提出宝贵的意见和建议！