1.谈一下你所了解的设计模式?

答：总体来说设计模式分为三类：

（1）创建型模式：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。

（2）结构型模式：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。

（3）行为模式：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。

设计模式的六大原则：

（1）开闭原则....

工厂模式：（Factory）

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

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

其次，创建实现类：

[java] view plaincopy

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

[java] view plaincopy

```java
public class SmsSender implements Sender {

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

最后，建工厂类：

```java
[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：

```java
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!

工厂模式适合：凡是出现了大量的产品需要创建，并且有共同的接口是时，可以通过工厂模式进行创建。如果传入的字符串有误，可以在工厂类前加 static 为静态工厂模式。

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。这样设计的好处：

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的压力。

```java
[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;
    }
}
```

这样这个类可以满足基本的要求，但是，像这样毫无线程安全保护的类，如果我们把它放在多线程的环境下，如何解决？我们首先会想到 getInstance 和 synchronized 如下：

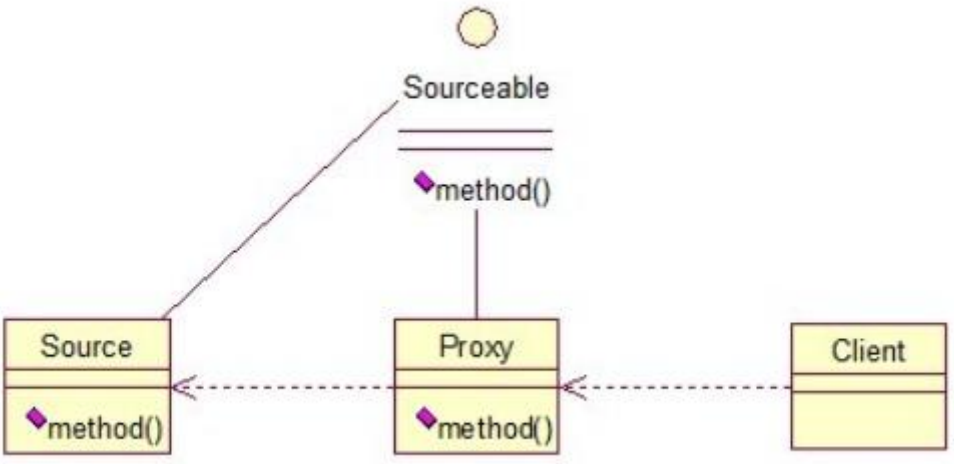
```java
[java] view plaincopy
public class SingletonTest {

    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;
    }
}
```

总结：

1.synchronized关键字锁定的是对象。

代理模式（Proxy）

其实每个模式名称就表明了该应用的作用，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会去找中介，因为你对该地区的房屋不熟悉，信息掌握的不够全面，所以我们希望找一个更熟悉的人去帮你去做。



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

```java
public interface Sourceable {
    public void method();
}
```

[java] view plaincopy

```java
public class Source implements Sourceable {

    @Override
    public void method() {
        System.out.println("the original method!");
    }
}
```

```java
public class Proxy implements Sourceable {

    private Source source;
    public Proxy(){
        super();
        this.source = new Source();
    }
    @Override
    public void method() {
        before();
        source.method();
        atfer();
    }
    private void atfer() {
        System.out.println("after proxy!");
    }
    private void before() {
        System.out.println("before proxy!");
    }
}
```

测试类：

```java
public class ProxyTest {

    public static void main(String[] args) {
        Sourceable source = new Proxy();
        source.method();
    }

}
```

输出：

before proxy!

the original method!

after proxy!

代理模式应用场景：

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：

1.修改原有的方法来适应。这样就违反了“对扩展开放，对修改关闭”的原则

2.就是采用一个代理类的调用原有的方法，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，有助于后期的维护。