Spring中使用了哪些设计模式?

1、简单工厂模式

又叫做静态工厂方法（StaticFactory Method）模式，但不属于23种GOF设计模式之一。

简单工厂模式的实质是由一个工厂类根据传入的参数，动态决定应该创建哪一个产品类。

spring中的BeanFactory就是简单工厂模式的体现，根据传入一个唯一的标识来获得bean对象，但是否是在传入参数后创建还是传入参数前创建这个要根据具体情况来定。如下配置，就是在 HelloItxxz 类中创建一个 itxxzBean。

```
<beans>
  <bean id="singletonBean" class="com.itxxz.HelloItxxz">
     <constructor-arg>
       <value>Hello! 这是singletonBean!value>
     </constructor-arg>
 </ bean>
 <bean id="itxxzBean" class="com.itxxz.HelloItxxz"  singleton="false">
   <constructor-arg>
      <value>Hello! 这是itxxzBean! value>
   </constructor-arg>
 </bean>
</beans>
```

2、工厂方法模式

通常由应用程序直接使用new创建新的对象，为了将对象的创建和使用相分离，采用工厂模式,即应用程序将对象的创建及初始化职责交给工厂对象。

一般情况下,应用程序有自己的工厂对象来创建bean.如果将应用程序自己的工厂对象交给Spring管理,那么Spring管理的就不是普通的bean,而是工厂Bean。

就以工厂方法中的静态方法为例讲解一下:

```
import java.util.Random;
public class StaticFactoryBean {
 public static Integer createRandom() {
      return new Integer(new Random().nextInt());
  }
}
```

建一个config.xm配置文件，将其纳入Spring容器来管理,需要通过factory-method指定静态方法名称:

```
<bean id="random"
class="example.chapter3.StaticFactoryBean" factory-method="createRandom"
scope="prototype"
/>
```

测试:

```
public static void main(String[] args) {
//调用getBean()时,返回随机数.如果没有指定factory-method,会返回StaticFactoryBean的实例,即返回工厂Bean的实例
XmlBeanFactory factory = new XmlBeanFactory(new
ClassPathResource("config.xml"));
System.out.println("我是IT学习者创建的实例:"+factory.getBean("random").toString());
}
```

3、单例模式

保证一个类仅有一个实例，并提供一个访问它的全局访问点。
spring中的单例模式完成了后半句话，即提供了全局的访问点BeanFactory。但没有从构造器级别去控制单例，这是因为spring管理的是是任意的java对象。

核心提示点：Spring下默认的bean均为singleton，可以通过singleton=“true|false” 或者 scope="?"来指定。

4、适配器模式

在Spring的Aop中，使用的Advice（通知）来增强被代理类的功能。Spring实现这一AOP功能的原理就使用代理模式（1、JDK动态代理。2、CGLib字节码生成技术代理。）对类进行方法级别的切面增强，即，生成被代理类的代理类， 并在代理类的方法前，设置拦截器，通过执行拦截器重的内容增强了代理方法的功能，实现的面向切面编程。

Adapter类接口：Target

```
public interface AdvisorAdapter {

boolean supportsAdvice(Advice advice);

 MethodInterceptor getInterceptor(Advisor advisor);

} MethodBeforeAdviceAdapter类，Adapter
class MethodBeforeAdviceAdapter implements AdvisorAdapter, Serializable {

 public boolean supportsAdvice(Advice advice) {
       return (advice instanceof MethodBeforeAdvice);
 }

 public MethodInterceptor getInterceptor(Advisor advisor) {
       MethodBeforeAdvice advice = (MethodBeforeAdvice) advisor.getAdvice();
 return new MethodBeforeAdviceInterceptor(advice);
 }

}
```

5、包装器模式

在我们的项目中遇到这样一个问题：我们的项目需要连接多个数据库，而且不同的客户在每次访问中根据需要会去访问不同的数据库。我们以往在spring和hibernate框架中总是配置一个数据源，因而sessionFactory的dataSource属性总是指向这个数据源并且恒定不变，所有DAO在使用sessionFactory的时候都是通过这个数据源访问数据库。

但是现在，由于项目的需要，我们的DAO在访问sessionFactory的时候都不得不在多个数据源中不断切换，问题就出现了：如何让sessionFactory在执行数据持久化的时候，根据客户的需求能够动态切换不同的数据源？我们能不能在spring的框架下通过少量修改得到解决？是否有什么设计模式可以利用呢？

首先想到在spring的applicationContext中配置所有的dataSource。这些dataSource可能是各种不同类型的，比如不同的数据库：Oracle、SQL Server、MySQL等，也可能是不同的数据源：比如apache 提供的org.apache.commons.dbcp.BasicDataSource、spring提供的org.springframework.jndi.JndiObjectFactoryBean等。然后sessionFactory根据客户的每次请求，将dataSource属性设置成不同的数据源，以到达切换数据源的目的。

spring中用到的包装器模式在类名上有两种表现：一种是类名中含有Wrapper，另一种是类名中含有Decorator。基本上都是动态地给一个对象添加一些额外的职责。

6、代理模式

为其他对象提供一种代理以控制对这个对象的访问。 从结构上来看和Decorator模式类似，但Proxy是控制，更像是一种对功能的限制，而Decorator是增加职责。
spring的Proxy模式在aop中有体现，比如JdkDynamicAopProxy和Cglib2AopProxy。

7、观察者模式

定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。
spring中Observer模式常用的地方是listener的实现。如ApplicationListener。

8、策略模式

定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化。
spring中在实例化对象的时候用到Strategy模式
在SimpleInstantiationStrategy中有如下代码说明了策略模式的使用情况：

```
// Don't override the class with CGLIB if no overrides.
if (beanDefinition.getMethodOverrides().isEmpty()) {
    Constructor constructorToUse = (Constructor) beanDefinition.resolvedConstructorOrFactoryMethod;
    if (constructorToUse == null) {
        Class clazz = beanDefinition.getBeanClass();
        if (clazz.isInterface()) {
            throw new BeanInstantiationException(clazz, "Specified class is an interface");
        }
        try {
            constructorToUse = clazz.getDeclaredConstructor((Class[]) null);
            beanDefinition.resolvedConstructorOrFactoryMethod = constructorToUse;
        }
        catch (Exception ex) {
            throw new BeanInstantiationException(clazz, "No default constructor found", ex);
        }
    }
    return BeanUtils.instantiateClass(constructorToUse, null);
}
else {
    // Must generate CGLIB subclass.
    return instantiateWithMethodInjection(beanDefinition, beanName, owner);
}
```

9、模板方法模式

定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。Template Method使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

Template Method模式一般是需要继承的。这里想要探讨另一种对Template Method的理解。spring中的JdbcTemplate，在用这个类时并不想去继承这个类，因为这个类的方法太多，但是我们还是想用到JdbcTemplate已有的稳定的、公用的数据库连接，那么我们怎么办呢？我们可以把变化的东西抽出来作为一个参数传入JdbcTemplate的方法中。但是变化的东西是一段代码，而且这段代码会用到JdbcTemplate中的变量。怎么办？那我们就用回调对象吧。

在这个回调对象中定义一个操纵JdbcTemplate中变量的方法，我们去实现这个方法，就把变化的东西集中到这里了。然后我们再传入这个回调对象到JdbcTemplate，从而完成了调用。这可能是Template Method不需要继承的另一种实现方式。

以下是一个具体的例子：
JdbcTemplate中的execute方法

```
public Object execute(ConnectionCallback action) throws DataAccessException {
    Assert.notNull(action, "Callback object must not be null");

    Connection con = DataSourceUtils.getConnection(getDataSource());
    try {
        Connection conToUse = con;
        if (this.nativeJdbcExtractor != null) {
            // Extract native JDBC Connection, castable to OracleConnection or the like.
            conToUse = this.nativeJdbcExtractor.getNativeConnection(con);
        }
        else {
            // Create close-suppressing Connection proxy, also preparing returned Statements.
            conToUse = createConnectionProxy(con);
        }
        return action.doInConnection(conToUse);
    }
    catch (SQLException ex) {
        // Release Connection early, to avoid potential connection pool deadlock
        // in the case when the exception translator hasn't been initialized yet.
        DataSourceUtils.releaseConnection(con, getDataSource());
        con = null;
        throw getExceptionTranslator().translate("ConnectionCallback", getSql(action), ex);
    }
    finally {
        DataSourceUtils.releaseConnection(con, getDataSource());
    }
}
```

JdbcTemplate执行execute方法

```
jdbcTemplate.execute(new ConnectionCallback() {
    public Object doInConnection(Connection con) throws SQLException, DataAccessException {
        // Do the insert
        PreparedStatement ps = null;
        try {
            ps = con.prepareStatement(getInsertString());
            setParameterValues(ps, values, null);
            ps.executeUpdate();
        } finally {
            JdbcUtils.closeStatement(ps);
        }
        //Get the key
        Statement keyStmt = null;
        ResultSet rs = null;
        HashMap keys = new HashMap(1);
        try {
            keyStmt = con.createStatement();
            rs = keyStmt.executeQuery(keyQuery);
            if (rs.next()) {
                long key = rs.getLong(1);
                keys.put(getGeneratedKeyNames()[0], key);
                keyHolder.getKeyList().add(keys);
            }
        } finally {
            JdbcUtils.closeResultSet(rs);
            JdbcUtils.closeStatement(keyStmt);
        }
        return null;
    }
});
```