Spring 设计模式总结

1.简单工厂

实现方式：

BeanFactory。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。

实质：

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。

实现原理：

bean容器的启动阶段：

- 读取bean的配置,将bean元素分别转换成一个BeanDefinition对象。
- 然后通过BeanDefinitionRegistry将这些bean注册到beanFactory中，保存在它的一个ConcurrentHashMap中。
- 将BeanDefinition注册到了beanFactory之后，在这里Spring为我们提供了一个扩展的切口，允许我们通过实现接口BeanFactoryPostProcessor 在此处来插入我们定义的代码。

典型的例子就是：PropertyPlaceholderConfigurer，我们一般在配置数据库的dataSource时使用到的占位符的值，就是它注入进去的。

容器中bean的实例化阶段：

实例化阶段主要是通过反射或者CGLIB对bean进行实例化，在这个阶段Spring又给我们暴露了很多的扩展点：

- **各种的Aware接口**，比如 BeanFactoryAware，对于实现了这些Aware接口的bean，在实例化bean时Spring会帮我们注入对应的BeanFactory的实例。
- **BeanPostProcessor接口**，实现了BeanPostProcessor接口的bean，在实例化bean时Spring会帮我们调用接口中的方法。
- **InitializingBean接口**，实现了InitializingBean接口的bean，在实例化bean时Spring会帮我们调用接口中的方法。
- **DisposableBean接口**，实现了BeanPostProcessor接口的bean，在该bean死亡时Spring会帮我们调用接口中的方法。

设计意义：

松耦合。可以将原来硬编码的依赖，通过Spring这个beanFactory这个工厂来注入依赖，也就是说原来只有依赖方和被依赖方，现在我们引入了第三方——spring这个beanFactory，由它来解决bean之间的依赖问题，达到了松耦合的效果.

bean的额外处理。通过Spring接口的暴露，在实例化bean的阶段我们可以进行一些额外的处理，这些额外的处理只需要让bean实现对应的接口即可，那么spring就会在bean的生命周期调用我们实现的接口来处理该bean。[非常重要]

2.工厂方法

实现方式：

FactoryBean接口。

实现原理：

实现了FactoryBean接口的bean是一类叫做factory的bean。其特点是，spring会在使用getBean()调用获得该bean时，会自动调用该bean的getObject()方法，所以返回的不是factory这个bean，而是这个bean.getOjbect()方法的返回值。

例子：

典型的例子有spring与mybatis的结合。

代码示例：

```
<!--SqlSessionFactory-->
<bean class="org.mybatis.spring.SqlSessionFactoryBean" id="sqlSessionFactory">
    <!-- 指定spring中的数据源 -->
    <property name="dataSource" ref="dataSource"></property>
    <property name="configLocation" value="classpath:mybatis-config.xml"></property>
    <property name="mapperLocations" value="classpath:cn/tulingxueyuan/mapper/*.xml"></property>
</bean>
```

说明：

我们看上面该bean，因为实现了FactoryBean接口，所以返回的不是SqlSessionFactoryBean 的实例，而是它的SqlSessionFactoryBean.getObject() 的返回值。

扩展：设计模式是什么鬼（工厂方法）

3.单例模式

Spring依赖注入Bean实例默认是单例的。

Spring的依赖注入（包括lazy-init方式）都是发生在AbstractBeanFactory的getBean里。getBean的doGetBean方法调用getSingleton进行bean的创建。

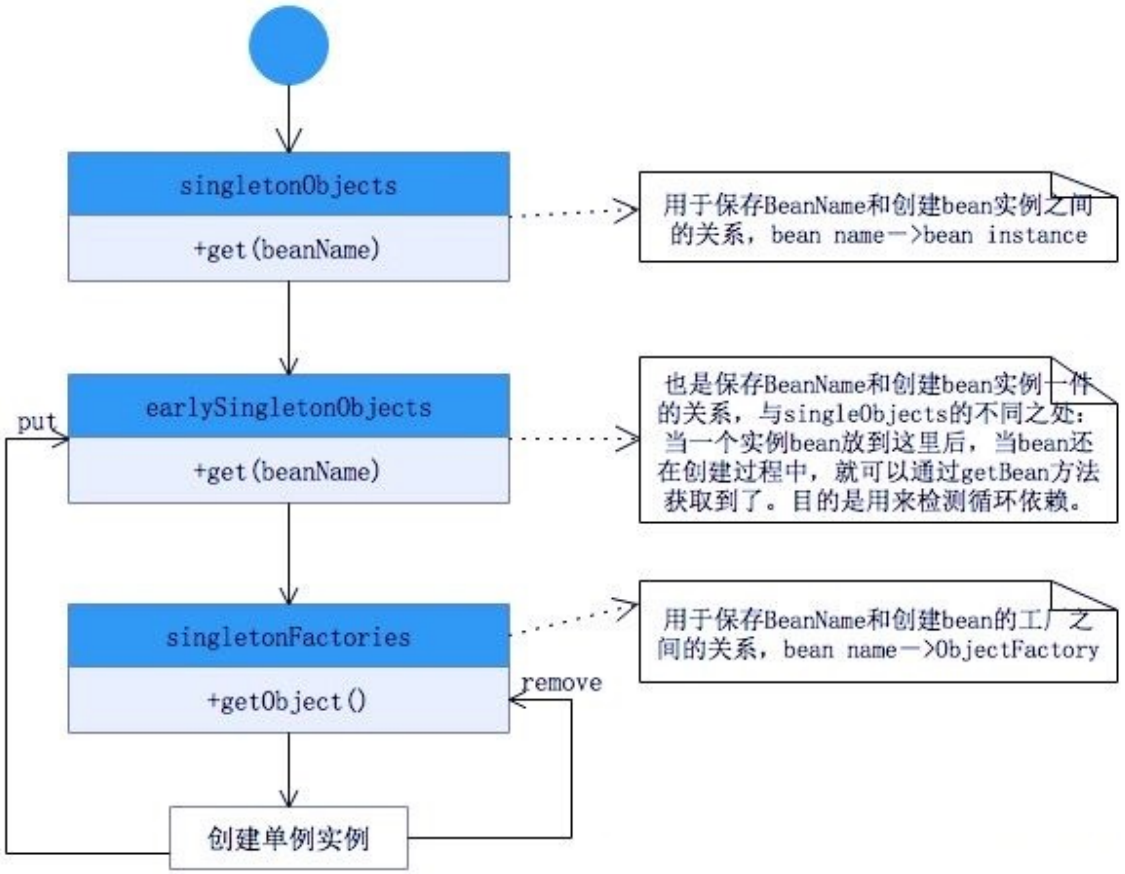
分析getSingleton()方法

```
1 public Object getSingleton(String beanName){
2 //参数true设置标识允许早期依赖
3 return getSingleton(beanName,true);
4 }
5 protected Object getSingleton(String beanName, boolean allowEarl
yReference) {
6 //检查缓存中是否存在实例
7     Object singletonObject = this.singletonObjects.get(beanName)
8 if (singletonObject == null && isSingletonCurrentlyInCreation(be
anName)) {
9 //如果为空，则锁定全局变量并进行处理。
10         synchronized (this.singletonObjects) {
11 //如果此bean正在加载，则不处理
12             singletonObject = this.earlySingletonObjects.get(be
anName);
13 if (singletonObject == null && allowEarlyReference) {
14 //当某些方法需要提前初始化的时候则会调用addSingleFactory 方法将对应
的ObjectFactory初始化策略存储在singletonFactories
15                 ObjectFactory singletonFactory = this.singleton
Factories.get(beanName);
16 if (singletonFactory != null) {
```

```
17 //调用预先设定的getObject方法
18                     singletonObject = singletonFactory.getObjec
t();
19 //记录在缓存中，earlysingletonObjects和singletonFactories互斥
20 this.earlySingletonObjects.put(beanName, singletonObject);
21 this.singletonFactories.remove(beanName);
22                 }
23             }
24         }
25     }
26 return (singletonObject != NULL_OBJECT ? singletonObject : null
27 }
```

getSingleton()过程图

ps：spring依赖注入时，使用了 双重判断加锁 的单例模式



总结

单例模式定义：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对单例的实现：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管理的是任意的java对象。

扩展：设计模式是什么鬼（单例）

4.适配器模式

实现方式：

SpringMVC中的适配器HandlerAdatper。

实现原理：

HandlerAdatper根据Handler规则执行不同的Handler。

实现过程：

DispatcherServlet根据HandlerMapping返回的handler，向HandlerAdatper发起请求，处理Handler。

HandlerAdapter根据规则找到对应的Handler并让其执行，执行完毕后Handler会向HandlerAdapter返回一个ModelAndView，最后由HandlerAdapter向DispatchServelet返回一个ModelAndView。

实现意义：

HandlerAdatper使得Handler的扩展变得容易，只需要增加一个新的Handler和一个对应的HandlerAdapter即可。

因此Spring定义了一个适配接口，使得每一种Controller有一种对应的适配器实现类，让适配器代替controller执行相应的方法。这样在扩展Controller时，只需要增加一个适配器类就完成了SpringMVC的扩展了。

扩展：设计模式是什么鬼（适配器）

5.装饰器模式

实现方式：

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。

实质：

动态地给一个对象添加一些额外的职责。

就增加功能来说，Decorator模式相比生成子类更为灵活。

扩展：设计模式是什么鬼（装饰）

6.代理模式

实现方式：

AOP底层，就是动态代理模式的实现。

动态代理：

在内存中构建的，不需要手动编写代理类

静态代理：

需要手工编写代理类，代理类引用被代理对象。

实现原理：

切面在应用运行的时刻被织入。一般情况下，在织入切面时，AOP容器会为目标对象创建动态的创建一个代理对象。SpringAOP就是以这种方式织入切面的。

织入：把切面应用到目标对象并创建新的代理对象的过程。

扩展：设计模式是什么鬼（代理）

7.观察者模式

实现方式：

spring的事件驱动模型使用的是 观察者模式 ， Spring中Observer模式常用的地方是listener的实现。

具体实现：

事件机制的实现需要三个部分,事件源,事件,事件监听器

ApplicationEvent抽象类[事件]

继承自jdk的EventObject,所有的事件都需要继承ApplicationEvent,并且通过构造器参数source得到事件源.

该类的实现类ApplicationContextEvent表示ApplicaitonContext的容器事件.

代码：

```
1 public abstract class ApplicationEvent extends EventObject{
```

```
2  private static final long serialVersionUID = 709905770818357193
7L;
3  private final long timestamp;
4  public ApplicationEvent(Object source){
5  super(source);
6  this.timestamp = System.currentTimeMillis();
7  }
8  public final longget Timestamp(){
9  returnthis.timestamp;
10  }
11 }
```

ApplicationListener接口[事件监听器]

继承自jdk的EventListener,所有的监听器都要实现这个接口。

这个接口只有一个onApplicationEvent()方法,该方法接受一个ApplicationEvent或其子类对象作为参数,在方法体中,可以通过不同对Event类的判断来进行相应的处理。

当事件触发时所有的监听器都会收到消息。

代码:

```
1 public interface ApplicationListener<EextendsApplicationEvent> e
xtends EventListener{
2  voidonApplicationEvent(E event);
3 }
```

ApplicationContext接口[事件源]

ApplicationContext是spring中的全局容器，翻译过来是”应用上下文”。

实现了ApplicationEventPublisher接口。

职责:

负责读取bean的配置文档,管理bean的加载,维护bean之间的依赖关系,可以说是负责bean的整个生命周期,再通俗一点就是我们平时所说的IOC容器。

代码:

```
1 public interface ApplicationEventPublisher {
2 voidpublishEvent(ApplicationEvent event);
3 }
4 public voi dpublishEvent(ApplicationEvent event) {
5     Assert.notNull(event, "Event must not be null");
6 if (logger.isTraceEnabled()) {
7           logger.trace("Publishing event in " + getDisplayName()
vent);
8     }
9     getApplicationEventMulticaster().multicastEvent(event);
10 if (this.parent != null) {
11 this.parent.publishEvent(event);
12     }
13 }
```

ApplicationEventMulticaster抽象类[事件源中publishEvent方法需要调用其方法getApplicationEventMulticaster]

属于事件广播器,它的作用是把Applicationcontext发布的Event广播给所有的监听器.

代码:

```
1 publicabstractclassAbstractApplicationContextextendsDefaultResou
rceLoader
2 implementsConfigurableApplicationContext, DisposableBean {
3 private ApplicationEventMulticaster applicationEventMulticaster;
4 protectedvoid registerListeners() {
5 // Register statically specified listeners first.
6 for (ApplicationListener> listener : getApplicationListeners())
7     getApplicationEventMulticaster().addApplicationListener(list
ener);
```

```
8       }
9   // Do not initialize FactoryBeans here: We need to leave all reg
 ular beans
10  // uninitialized to let post-processors apply to them!
11  String[] listenerBeanNames = getBeanNamesForType(ApplicationLis
 tener.class, true, false);
12  for (String lisName : listenerBeanNames) {
13      getApplicationEventMulticaster().addApplicationListenerBean
 isName);
14      }
15    }
16  }
```

扩展：[设计模式是什么鬼（观察者）]

8.策略模式

实现方式：

Spring框架的资源访问Resource接口。该接口提供了更强的资源访问能力，Spring 框架本身大量使用了 Resource 接口来访问底层资源。

Resource 接口介绍

source 接口是具体资源访问策略的抽象，也是所有资源访问类所实现的接口。

Resource 接口主要提供了如下几个方法:

- **getInputStream()**：定位并打开资源，返回资源对应的输入流。每次调用都返回新的输入流。调用者必须负责关闭输入流。
- **exists()**：返回 Resource 所指向的资源是否存在。
- **isOpen()**：返回资源文件是否打开，如果资源文件不能多次读取，每次读取结束应该显式关闭，以防止资源泄漏。

- **getDescription()**：返回资源的描述信息，通常用于资源处理出错时输出该信息，通常是全限定文件名或实际 URL。
- **getFile**：返回资源对应的 File 对象。
- **getURL**：返回资源对应的 URL 对象。

最后两个方法通常无须使用，仅在通过简单方式访问无法实现时，Resource 提供传统的资源访问的功能。

Resource 接口本身没有提供访问任何底层资源的实现逻辑，**针对不同的底层资源，Spring 将会提供不同的 Resource 实现类，不同的实现类负责不同的资源访问逻辑。**

Spring 为 Resource 接口提供了如下实现类：

- **UrlResource**：访问网络资源的实现类。
- **ClassPathResource**：访问类加载路径里资源的实现类。
- **FileSystemResource**：访问文件系统里资源的实现类。
- **ServletContextResource**：访问相对于 ServletContext 路径里的资源的实现类.
- **InputStreamResource**：访问输入流资源的实现类。
- **ByteArrayResource**：访问字节数组资源的实现类。

这些 Resource 实现类，针对不同的的底层资源，提供了相应的资源访问逻辑，并提供便捷的包装，以利于客户端程序的资源访问。

扩展：设计模式是什么鬼（策略）

9.模版方法模式

经典模板方法定义：

父类定义了骨架（调用哪些方法及顺序），某些特定方法由子类实现。

最大的好处：代码复用，减少重复代码。除了子类要实现的特定方法，其他方法及方法调用顺序都在父类中预先写好了。

所以父类模板方法中有两类方法：

共同的方法：所有子类都会用到的代码

不同的方法：子类要覆盖的方法，分为两种：

- 抽象方法：父类中的是抽象方法，子类必须覆盖
- 钩子方法：父类中是一个空方法，子类继承了默认也是空的

注：为什么叫钩子，子类可以通过这个钩子（方法），控制父类，因为这个钩子实际是父类的方法（空方法）！

Spring模板方法模式实质：

是模板方法模式和回调模式的结合，是Template Method不需要继承的另一种实现方式。Spring几乎所有的外接扩展都采用这种模式。

推荐：设计模式是什么鬼（模板方法）

具体实现：

JDBC的抽象和对Hibernate的集成，都采用了一种理念或者处理方式，那就是模板方法模式与相应的Callback接口相结合。

采用模板方法模式是为了以一种统一而集中的方式来处理资源的获取和释放，以JdbcTempalte为例:

```
1 public abstract class JdbcTemplate{
2 public final Object execute（String sql）{
3         Connection con=null;
4         Statement stmt=null;
5 try{
6             con=getConnection（）;
```

```
            stmt=con.createStatement（）;
Object retValue=executeWithStatement（stmt,sql）;
return retValue;
         }catch（SQLException e）{
              ...
         }finally{
             closeStatement（stmt）;
             releaseConnection（con）;
         }
     }
 protected abstract
Object executeWithStatement（Statement   stmt, String sql）;
}
```

引入回调原因：

JdbcTemplate是抽象类，不能够独立使用，我们每次进行数据访问的时候都要给出一个相应的子类实现,这样肯定不方便，所以就引入了回调。

回调代码

```
1. publicinterfaceStatementCallback{
2. Object doWithStatement（Statement stmt）;
3. }
```

利用回调方法重写JdbcTemplate方法

```
public class JdbcTemplate{
public final Object execute（StatementCallback callback）{
         Connection con=null;
         Statement stmt=null;
try{
             con=getConnection（）;
             stmt=con.createStatement（）;
Object retValue=callback.doWithStatement（stmt）;
return retValue;
          }catch（SQLException e）{
              ...
```

```
        }finally{
            closeStatement（stmt）;
            releaseConnection（con）;
        }
    }
    ...//其它方法定义
}
```

Jdbc使用方法如下：

```
JdbcTemplate jdbcTemplate=...;
finalString sql=...;
    StatementCallback callback=new StatementCallback(){
publicObject=doWithStatement(Statement stmt){
return ...;
    }
}
jdbcTemplate.execute(callback);
```

为什么JdbcTemplate没有使用继承?

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。

10.责任链模式

CglibAopProxy类第688行：

```
new CglibMethodInvocation(proxy, target, method, args, targetCla
ss, chain, methodProxy).proceed();
```

参数 chain:拦截器链，保含了目标方法的所有切面方法，从chain里面的数组元素的顺序来看，拦截器的顺序before不再after前面执行
每一个 **Interceptor有一个invoke()方法
Interceptor是一个空接口 MethodInterceptor extends Interceptor ，以下是Interceptor的继承结构：

```
public interface Advice {

}

public interface Interceptor extends Advice {

}

public interface MethodInterceptor extends Interceptor {
  Object invoke(MethodInvocation invocation) throws Throwable;
}
```

Object invoke(MethodInvocation invocation) throws Throwable；方法：
参数： MethodInvocation 类中有proceed()方法，以下是MethodInvocation的继承结构：

```
public interface Joinpoint {

 Object proceed() throws Throwable;

 Object getThis();

 AccessibleObject getStaticPart();
}

public interface Invocation extends Joinpoint {
  Object[] getArguments();
}

public interface MethodInvocation extends Invocation {
```

```
15   Method getMethod();
16 }
```

MethodInvocation extends Invocation extends JoinPoint ,proceed()方法时JoinPoint接口声明的

然后
ReflectiveMethodInvocation implements ProxyMethodInvocation ,ProxyMethodInvocation extends MethodInvocation
spring的拦截器 xxxInterceptor都实现了自己的 Object invoke(MethodInvocation invocation)方法

ReflectiveMethodInvocation类中的 proceed()方法会遍历拦截器链，调用每个拦截器的invoke方法，传入ReflectiveMethodInvocation自身作为参数，

每个拦截器的invoke方法做两件事(这两件事的执行顺序因拦截器的功能而异)：1.执行自己的业务逻辑 2.执行ReflectiveMethodInvocation的proceed()：这样就实现了链式调用

这就是责任链模式：

> 统一的业务接口：Handler接口 中的方法invoke(),即业务方法
> 责任链相当于一个负责人集合，每一个负责人都实现了自己的invoke()方法来处理传进来的数据或对象或对象的指定方法
> 如何通知下一个负责人处理业务：

方法1：设计一个责任链执行器，包含责任链集合。责任链执行器中有一个proceed(),方法内遍历执行负责人的invoke()方法，invoke方法以执行器作为参数：

invoke(执行器)，invoke(执行器)处理完业务后，执行器又调用proceed()方法，将索引移到下一个负责人位置。

这样：执行器和负责人的方法相互调用，而执行器通过移动索引通知下一个负责人处理业务。

方法2：基于链表的责任链，每一个负责人是一个责任节点Node，包含指向下一个负责人的next引用

负责人的处理业务的方法 invoke()这时不带参数，invoke()方法里面递归调用invoke()方法，并设置出口条件。

如何通知下一个负责人处理业务：invoke()方法：1.处理业务，2.next.invoke()，3.出口条件可以是next!=null

文档：Spring 设计模式总结
链接：http://note.youdao.com/noteshare?id=de39c0955afbc55e7205ba28d82e99ae&sub=wcp159755546113970