


# 深入挖掘Spring系列 -- 从设计模式角度看Spring

原创 Danny\_idea 于 2021-06-04 22:19:23 发布 141 收藏 1

分类专栏: spring 源码分析 java

版权

 **spring** 同时被 3 个专栏收录

1 订阅 11 篇文章 订阅专栏

## Spring的生态演进变化

Spring是一款伟大的 **框架** 产品，在发展过程中一直都是靠一家叫做Pivotal的技术公司在背后支撑。Spring真正流行的时间是在2007年11月份，发布了2.5版本的时候。

Spring Source 在3.0升级为了后续的发展所以拆分为了Spring Framework4.0 发布于2013年，随后Spring Boot发布于2014年，和传统的Spring Framework有所不同，SpringBoot是一款完全独立的产品路线，很多设计都是在为了简化对于Spring的使用而发明的。

2014年发布了一个Spring Cloud，这是业界第一个完整的微服务解决方案。早期时候以Spring Cloud Netflix为代。这款框架在2015年3月开源，2018年12.12日后进入了维护模式。

Netflix公司对外开源其实目的是：

- 想对外宣传，吸收外界的技术点，学习和完善现有的技术框架。
- 试探市场中对于这套技术解决方案的接受性。
- 挣钱。

2017年09月 Spring Framework发布了5.0

2019年08月01日 发布了Spring Cloud Alibaba 1.0

阿里巴巴和pivotal合作创建的一个框架标准

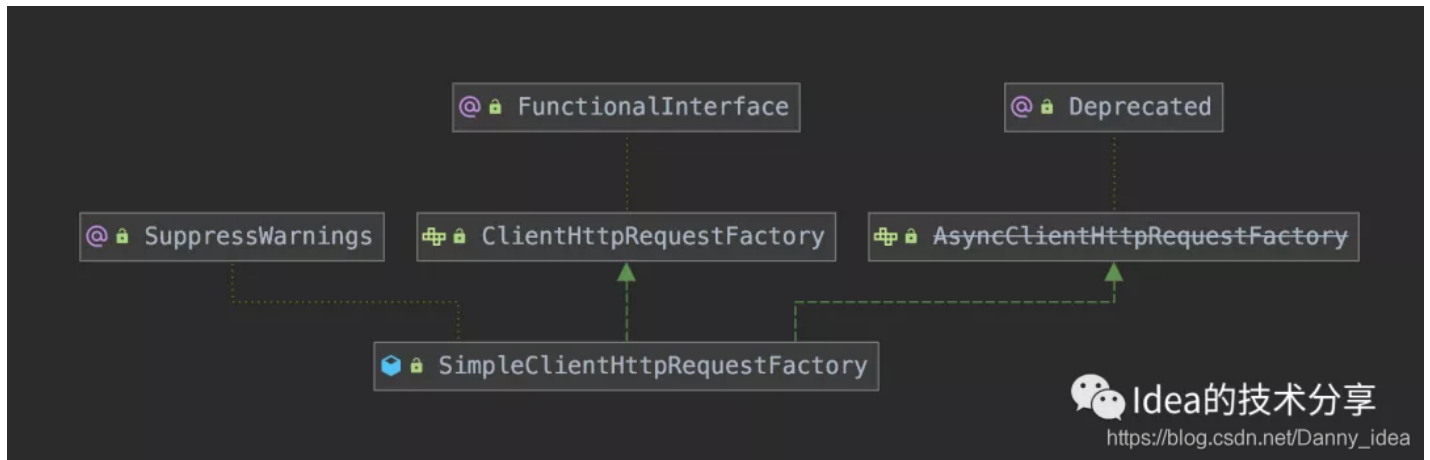
## Spring用了哪些设计模式

首先来看下边这段代码案例：

```
1 package org.idea.spring.framework.http.util;
2
3 import org.springframework.http.HttpMethod;
4 import org.springframework.http.client.ClientHttpRequest;
5 import org.springframework.http.client.ClientHttpRequestFactory;
6 import org.springframework.http.client.ClientHttpResponse;
7 import org.springframework.http.client.SimpleClientHttpRequestFactory;
8 import org.springframework.util.StreamUtils;
9
10 import java.io.IOException;
11 import java.io.InputStream;
12 import java.net.URI;
13 import java.net.URISyntaxException;
14 import java.nio.charset.Charset;
15
16 /**
17  * @Author Linhao
18  * @Date created in 9:09 下午 2021/5/22
19  */
20 public class HttpRequestFactoryDemo {
21
22     public static void main(String[] args) throws URISyntaxException, IOException {
23         ClientHttpRequestFactory chrF = new SimpleClientHttpRequestFactory();
24         ClientHttpRequest clientHttpRequest = chrF.createRequest(new URI("http://www.baidu.com"), HttpMethod.GET);
25         ClientHttpResponse clientHttpResponse = clientHttpRequest.execute();
26         InputStream inputStream = clientHttpResponse.getBody();
27         String response = StreamUtils.copyToString(inputStream, Charset.forName("UTF-8"));
28         inputStream.close();
29     }
30 }
```

```
29 |         system.out.println(response);
30 |     }
31 | }
32 |
```

一个简单的http发送，这里面主要使用的是Spring框架中的 `ClientHttpRequestFactory` 对象，这个对象也是 `RestTemplate` 中涉及到的一个重要组件，该对象在设计的时候，对http请求封装了一个工厂。例如我们的 `SimpleClientHttpRequestFactory` 就是其中一种实现。从这个角度来看，这里采用了工厂相关的 **设计模式**。



### 关于工厂模式

工厂模式包含了三种类型：

- 简单工厂模式
- 工厂方法模式（这种用得不多，这里我直接略过）
- 抽象工厂模式

### 简单工厂模式

这种设计比较好理解，我写了个简单的案例如下：

```
1 | public class SessionFactory {
2 |     public Session getSession(){
3 |         /** 省略 **/
4 |         return new Session();
5 |     }
6 | }
```

没有任何的接口定义，就是一个简单的Factory，专门负责生产指定的session。但是这种设计很明显存在扩展性的问题，假设后期需要融合更多种类的Session，就会出现以下情况：

```
1 | public class SessionFactory {
2 |
3 |     public Session getSession(){
4 |         return new Session();
5 |     }
6 |
7 |     public RedisSession getRedisSession(){
8 |         return new RedisSession();
9 |     }
10 |
11 |     public ZookeeperSession getZookeeperSession(){
12 |         return new ZookeeperSession();
13 |     }
14 |
15 |     public WebSocketSession getWebSocketSession(){
16 |         return new WebSocketSession();
17 |     }
18 |
19 | }
```

每次新增一个Session的类型都需要对SessionFactory这个父类做修改，这样的设计导致了SessionFactory这个类包含了多种业务职责，代码只会越堆越多，职责变得混乱。

### 抽象工厂模式

将原先的sessionFactory抽象成为一个接口，然后各种类的session都统一继承一个父类。大体如下：

```
1 public class Session {
2
3     String name;
4
5     public Session(String name) {
6         this.name = name;
7     }
8
9     @Override
10    public String toString() {
11        return "Session{" +
12            "name='" + name + '\'' +
13        '}'
14    }
15 }
```

```
1
2 public class ZookeeperSession extends Session {
3
4     public ZookeeperSession(String name){
5         super(name);
6     }
7 }
8 public class WebSocketSession extends Session {
9
10    public WebSocketSession(String name) {
11        super(name);
12    }
13 }
14
15 public class RedisSession extends Session {
16
17    public RedisSession(String name) {
18        super(name);
19    }
20 }
21 }
```

SessionFactory模块改成：

```
1 public interface ISessionFactory {
2     Session getSession();
3 }
```

然后各自的子类进行基础



这样能够保证代码结构的设计遵守了开放封闭原则和依赖倒置原则。

#### 好处:

从原先的单一工厂拆分为多个工厂，不同工厂的含义不一样，满足了设计原则的单一职责。

将session的生成规则进行了封装，调用分不必关心session的生产过程。

#### 不足点:

每次新增一个工厂都需要写一堆的代码。可以结合一些反射来进行优化。

### Spring中的观察者模式

Spring内部的监听器使用场景:

```
1
2 package org.idea.spring.framework.event;
3
4 import org.springframework.context.ApplicationEvent;
5 import org.springframework.context.ApplicationEventPublisher;
6 import org.springframework.context.ApplicationEventPublisherAware;
7 import org.springframework.context.ApplicationListener;
8 import org.springframework.context.annotation.AnnotationConfigApplicationContext;
9
10 /**
11  * 事件发布者
12  *
13  * @Author Linhao
14  * @Date created in 4:47 下午 2021/5/23
15  */
16 public class ApplicationEventPublisherDemo implements ApplicationEventPublisherAware {
17
18     public static void main(String[] args) {
19         AnnotationConfigApplicationContext annotationConfigApplicationContext = new AnnotationConfigApplicationContext(
20             annotationConfigApplicationContext.register(ApplicationEventPublisherDemo.class);
21         annotationConfigApplicationContext.addApplicationListener(new ApplicationListener<MyEvent>() {
22             @Override
23             public void onApplicationEvent(MyEvent event) {
24                 System.out.println("application listener 接收到消息: " + event);
25             }
26         });
27         annotationConfigApplicationContext.refresh();
28         annotationConfigApplicationContext.close();
29     }
30
31     @Override
32     public void setApplicationEventPublisher(ApplicationEventPublisher applicationEventPublisher) {
33         System.out.println("====");
34         applicationEventPublisher.publishEvent(new MyEvent("hello world") {
35         });
36         applicationEventPublisher.publishEvent("pay load event");
37     }
38 }
```

自定义一套属于自己的事件

```

1
2 package org.idea.spring.framework.event;
3
4 import org.springframework.context.ApplicationEvent;
5
6 /**
7  * @Author Linhao
8  * @Date created in 6:21 下午 2021/5/23
9  */
10 public class MyEvent extends ApplicationEvent {
11     /**
12      * Create a new {@code ApplicationEvent}.
13      *
14      * @param source the object on which the event initially occurred or with
15      *               which the event is associated (never {@code null})
16      */
17     public MyEvent(Object source) {
18         super(source);
19     }
20 }

```

启动之后你会发现有如下消息:

```

18:22:34.895 [main] DEBUG org.springframework.beans.factory.support.DefaultListableBeanFactory - Creating shared instance of single
====
application listener 接收到消息: org.idea.spring.framework.event.ApplicationEventPublisherDemo$2[source=hello world]

```

其实代码里面发送了两条消息，但是监听器由于只是监听了MyEvent事件，所以这里没有将PayLoad事件打印出来。

这里所参考到的设计模式就是经典的观察者模式。

本质上观察者模式就是声明一个事件，然后观察者们都定于到这个事件中，并且使用一个类似list都数据结构将这些观察者们存放起来，最后通过遍历去触发它们对应的回调函数。类似的这种设计在JDK8中已经有提供标准的api，代码案例如下：

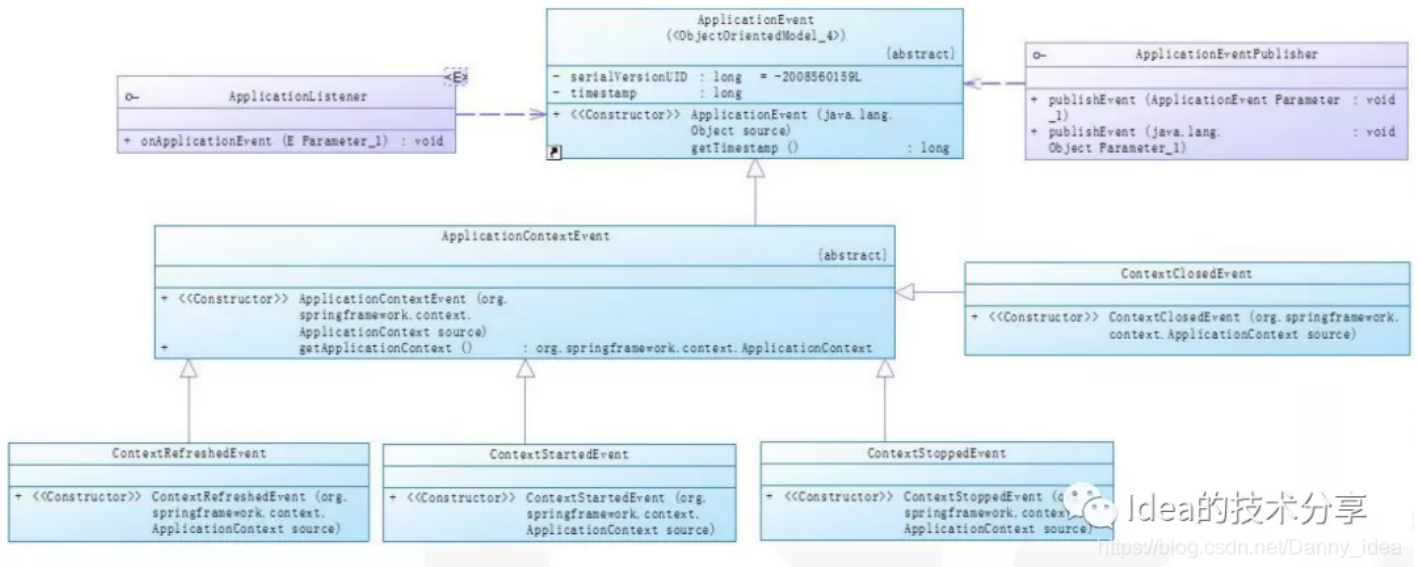
```

1 package org.idea.spring.framework.event;
2
3 import java.util.EventListener;
4 import java.util.EventObject;
5 import java.util.Observable;
6 import java.util.Observer;
7
8 /**
9  * @Author Linhao
10  * @Date created in 3:52 下午 2021/5/23
11  */
12 public class EventDemo {
13
14     public static void main(String[] arg) {
15         EventObservable observable = new EventObservable();
16         observable.addObserver(new EventObserver());
17         observable.notifyObservers("send message");
18     }
19
20     static class EventObservable extends Observable {
21         @Override
22         protected synchronized void setChanged() {
23             super.setChanged();
24         }
25
26         @Override
27         public void notifyObservers(Object data){
28             this.setChanged();
29             super.notifyObservers(new EventObject(data));
30             clearChanged();
31         }
32     }
33
34     static class EventObserver implements Observer, EventListener {
35
36         @Override

```

```
37     public void update(Observable o, Object msg) {
38         EventObject eventObject = (EventObject) msg;
39         System.out.println("接收数据: " + eventObject);
40     }
41 }
42
43 }
44
```

关于事件部分的整体设计如下图所示：

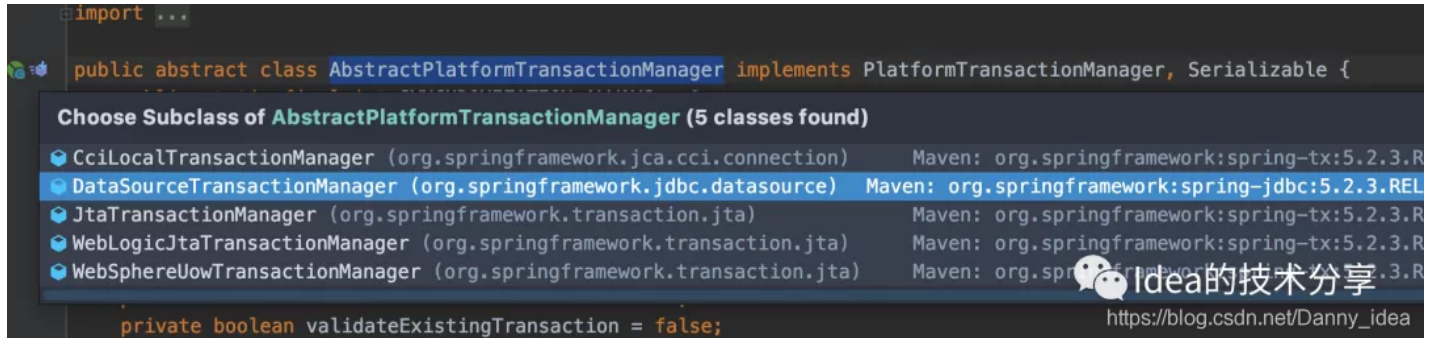


## Spring中的模版模式

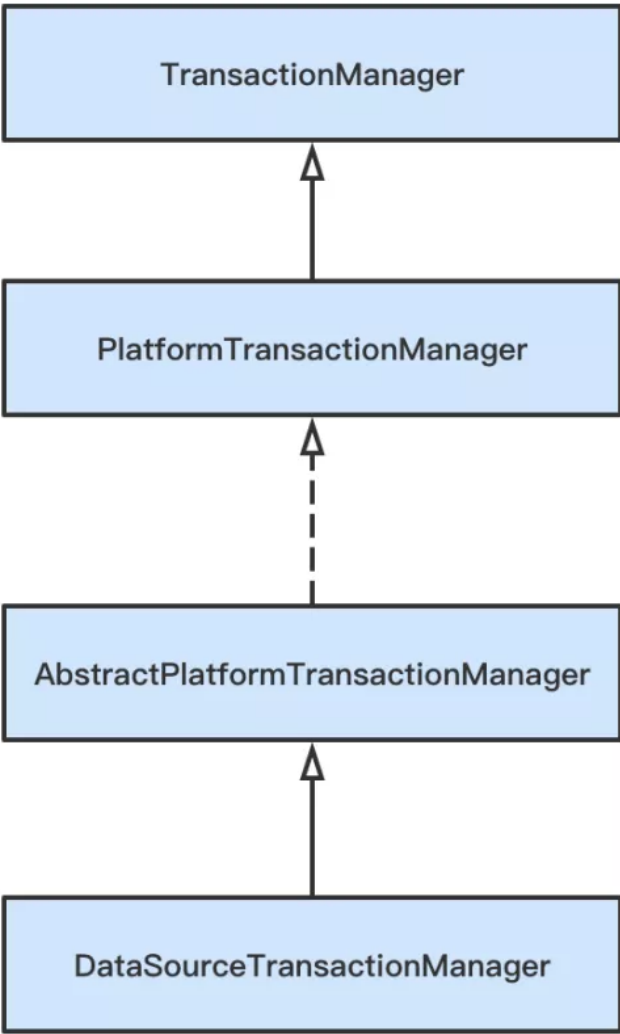
在Spring内部源代码中，模版模式也是使用非常多的一种设计，例如我们的事务管理部分：

org.springframework.transaction.PlatformTransactionManager

org.springframework.transaction.support.AbstractPlatformTransactionManager



这里我们以我们常用的DataSourceTransactionManager作为讲解案例分析：



Idea的技术分享

[https://blog.csdn.net/Danny\\_idea](https://blog.csdn.net/Danny_idea)

DataSourceTransactionManager通常会在我们的数据库访问中处理一些和事务有关的会话信息，例如提交，回滚。那么这些统一的提交回滚方法其实都是由其父类所制定的。




AbstractPlatformTransactionManager.java | TransactionManager.java | DataSourceTransactionManager.java | DataSourceTransactionManagerAutoConfiguration.class | DataSourceTransactionM

abstract

9 matches

1178 \* <p>Note that exceptions will get propagated to the commit caller  
1179 \* and cause a rollback of the transaction.  
1180 \* @param status the status representation of the transaction  
1181 \* @throws RuntimeException in case of errors; will be <b>propagated to the caller</b>  
1182 \* (note: do not throw TransactionException subclasses here!)  
1183 \*/  
1184 @protected void prepareForCommit(DefaultTransactionStatus status) {  
1185 }  
1186  
1187 /\*\*  
1188 \* Perform an actual commit of the given transaction.  
1189 \* <p>An implementation does not need to check the "new transaction" flag  
1190 \* or the rollback-only flag; this will already have been handled before.  
1191 \* Usually, a straight commit will be performed on the transaction object  
1192 \* contained in the passed-in status.  
1193 \* @param status the status representation of the transaction  
1194 \* @throws TransactionException in case of commit or system errors  
1195 \* @see DefaultTransactionStatus#getTransaction  
1196 \*/  
1197 protected abstract void doCommit(DefaultTransactionStatus status) throws TransactionException;  
1198  
1199 /\*\*  
1200 \* Perform an actual rollback of the given transaction.  
1201 \* <p>An implementation does not need to check the "new transaction" flag;  
1202 \* this will already have been handled before. Usually, a straight rollback  
1203 \* will be performed on the transaction object contained in the passed-in status.  
1204 \* @param status the status representation of the transaction  
1205 \* @throws TransactionException in case of system errors  
1206 \* @see DefaultTransactionStatus#getTransaction  
1207 \*/  
1208 protected abstract void doRollback(DefaultTransactionStatus status) throws TransactionException;  
1209  
1210 /\*\*  
1211 \* Set the given transaction rollback-only. Only called on rollback  
1212 \* if the current transaction participates in an existing one.

 Idea的技术分享  
https://blog.csdn.net/Danny\_idea

这三段源代码主要目的就是查询对应uri匹配到适配器，从而处理后台的响应逻辑部分代码。

Spring内部的策略模式

在之前我有一篇文章中介绍了关于Spring容器内部加载资源属性的一些技巧使用，其中有提到过Resource这个接口，其实Resource接口就是一层封装，对于不同的资源处理有不同的子类实现。

Spring中获取资源的方式一共有以下四种：

- 通过Resource接口获取资源
- 通过ResourceLoader接口获取资源
- 通过ApplicationContext获取资源
- 将resource注入到bean中的方式获取资源

实现类	描述
ClassPathResource	通过类路径获取资源文件
FileSystemResource	通过文件系统获取资源
UrlResource	通过URL地址获取资源
ByteArrayResource	获取字节数组封装的资源
ServletContextResource	获取ServletContext环境下的资源
InputStreamResource	获取输入流封装的资源

这部分我们可以通过一个代码案例来实践理解下：



```
1
2 package org.idea.spring.resource;
3
4 import org.springframework.core.io.*;
5
6 /**
7  * @Author Linhao
8  * @Date created in 7:38 下午 2021/5/23
9  */
10 public class ResourceLoaderDemo {
```

从这段代码的执行结果可以看出ResourceLoader会根据我们输入到字符串规则来匹配不同的资源加载规则。

```
@Override
public Resource getResource(String location) {
    Assert.notNull(location, message: "Location must not be null");

    for (ProtocolResolver protocolResolver : getProtocolResolvers()) {
        Resource resource = protocolResolver.resolve(location, resourceLoader: this);
        if (resource != null) {
            return resource;
        }
    }

    if (location.startsWith("/")) {
        return getResourceByPath(location);
    }
    else if (location.startsWith(CLASSPATH_URL_PREFIX)) {
        return new ClassPathResource(location.substring(CLASSPATH_URL_PREFIX.length()), getClassLoader());
    }
    else {
        try {
            // Try to parse the location as a URL...
            URL url = new URL(location);
            return (ResourceUtils.isFileURL(url) ? new FileUrlResource(url) : new UrlResource(url));
        }
        catch (MalformedURLException ex) {
            // No URL -> resolve as resource path.
            return getResourceByPath(location);
        }
    }
}
```

DefaultResourceLoader > getResource()

Idea的技术分享  
[https://blog.csdn.net/Danny\\_idea](https://blog.csdn.net/Danny_idea)

而实际上呢，当我们进入源代码去debug查阅到时候也会发现确实逻辑是这么实现的。

在实际应用中，我们也可以从ApplicationContext中去获取Resource对象，代码如下：

```
1 public class MyResource implements ApplicationContextAware {
2     private ApplicationContext applicationContext;
3
4     public void setApplicationContext(ApplicationContext applicationContext) throws BeansException {
5         this.applicationContext = applicationContext;
6     }
7
8     public void resource() throws IOException {
9         Resource resource = applicationContext.getResource("file:D:\\test.txt");
10        System.out.println(resource.getFilename());
11        System.out.println(resource.contentLength());
12    }
13
14
15 }
```

## Spring内部的代理模式

代理模式这块比较代表性的就是aop这部分了，解决代码复用，公共函数抽取，简化开发，业务之间的解耦；例如日志功能，因为日志功能往往横跨系统中的每个业务模块，使用 AOP 可以很好的将日志功能抽离出来。

关于Spring内部的AOP的代理实现主要有cglib和JDK两种方式，源代码核心部分如下：

```
public class DefaultAopProxyFactory implements AopProxyFactory, Serializable {

    @Override
    public AopProxy createAopProxy(AdvisedSupport config) throws AopConfigurationException {
        if (config.isOptimize() || config.isProxyTargetClass() || hasNoUserSuppliedProxyInterfaces(config)) {
            Class<?> targetClass = config.getTargetClass();
            if (targetClass == null) {
                throw new AopConfigurationException("TargetSource cannot determine target class: " +
                    "Either an interface or a target is required for proxy creation.");
            }
            if (targetClass.isInterface() || Proxy.isProxyClass(targetClass)) {
                return new JdkDynamicAopProxy(config);
            }
            return new ObjenesisCglibAopProxy(config);
        }
        else {
            return new JdkDynamicAopProxy(config);
        }
    }
}
```

Idea的技术分享  
[https://blog.csdn.net/Danny\\_idea](https://blog.csdn.net/Danny_idea)

这部分代码可能有不少读者会有疑惑：

为什么代码中那么推崇JDK代理，而不是CGLIB代理呢，不是说CGLIB代理要比JDK代理效率更高吗？

误区解释：我在初期学习CGLIB和JDK代理的时候在网上看了不少的资料都说CGLIB代理的效率要比JDK代理高很多，但是通过实战之后发现，高版本的JDK（如JDK8）已经对其自身的代理机制做了优化，性能要比CGLIB更佳。

另外在Spring的官网上也看到了这么一段话：

### 5.1.3. AOP Proxies

Spring AOP defaults to using standard JDK *dynamic proxies* for AOP proxies. This enables any interface (or set of interfaces) to be proxied.

Spring AOP can also use CGLIB proxies. This is necessary to proxy classes rather than interfaces. CGLIB is used by default if a business object does not implement an interface. As it is good practice to program to interfaces rather than classes; business classes normally will implement one or more business interfaces. It is possible to [force the use of CGLIB](#), in those (hopefully rare) cases where you need to advise a method that is not declared on an interface, or where you need to pass a proxied object to a method as a concrete type.

It is important to grasp the fact that Spring AOP is *proxy-based*. See [Understanding AOP proxies](#) for an in-depth discussion of exactly what this implementation detail actually means.

关于代理部分的总结

- 1.默认使用 JDK 动态代理，这样便可以代理所有的接口类型（interface）
- 2.Spring AOP也支持CGLIB的代理方式。如果我们被代理对象没有实现任何接口，则默认是CGLIB
- 3.我们可以强制使用CGLIB，指定proxy-target-class = "true" 或者 基于注解@EnableAspectJAutoProxy(proxyTargetClass = true)

其实关于Spring内部的设计模式使用场景还有很多，大部分都是多种设计模式的混合使用，例如BeanFactory模块的设计。希望今天的这篇文章能够对你有所帮助。