

修改记录

修改摘要	修改记录戳	修改人
新增《spring 源码》	2019 年 11 月 11 日	Jack

1、spring 程序入口和 xml 解析

Spring 重新定义了 java

Spring 的历史

2004 年 3 月，Spring1.0 发布

2003 年 6 月，Spring Framework 第一次以 Apache 2.0 许可证下发布 0.9 版本，2004 年 3 月，Spring1.0 正式发布

对于 Spring1.0，其源码只有一个包，在该包中包含了 aop、beans、context、core、jdbc、orm 等。对于此时的版本，Spring1.0 仅支持 XML 配置的方式。

2006 年 10 月，Spring2.0 发布

对于 2.0，Spring 主要增加了对注解的支持，实现了基于注解的配置。

在 2007 年 11 月，发布 Spring2.5，该版本具备的特性有：

添加可扩展的 XML 配置功能，用于简化 XML 配置

支持 Java5

添加额外的 IOC 容器扩展点，支持动态语言（如 groovy，aop 增强功能和新的 bean 范围）

2009 年 12 月，Spring3.0 发布

Spring3.0 主要具有的特性有：

模块重组系统

支持 Spring 表达式语言（Spring Expression）

基于 Java 的 Bean 配置 (JavaConfig)

支持嵌入式数据库： HSQL、 H2 等

支持 REST

支持 Java6

2013 年 12 月，发布 Spring4.0

对于 Spring4.0 是 Spring 版本历史上的一重大升级。其特性为：

全面支持 Java8

支持 Lambda 表达式

支持 Java8 的时间和日期 API

支持重复注解

支持 Java8 的 Optional

核心容器增强

增加泛型依赖注入

增加 Map 依赖注入

增加 List 依赖注入

支持 lazy 注解配置懒加载

支持 Condition 条件注解

CGLIB 动态代理增强

支持基于 GroovyDSL 定义 Bean

Web 增强

增强 SpringMVC，基于 Servlet3.0 开发

提供 RestController 注解

提供 AsyncRestTemplate 支持客户端的异步无阻塞请求

增加对 WebSocket 的支持

2017 年 9 月， Spring5.0 发布

Spring5.0 特性如下：

升级到 Java8、 JavaEE7

废弃低版本，将 Java8、 JavaEE 7 作为最低版本要求

兼容 Java9

兼容 JavaEE8

反应式编程模型，增加 WebFlux 模块

升级 SpringMVC，增加对最新的 API (Jackson 等) 的支持

增加函数式编程模式

重构源码，部分功能使用 Lambda 表达式实现

Spring 子项目

Spring IO Platform : Spring IO 是可集成的、构建现代化应用的版本平台。Spring IO 是模块化的、企业级的分布式系统，包括一系列依赖，是的开发者仅能对自己所需的部分进行完全的部署控制。

Spring Boot: Spring 应用快速开发工具，用来简化 Spring 应用开发过程。

Spring XD: Spring XD(eXtreme Date, 极限数据)是 Pivotal 的大数据产品。它结合了 Spring Boot 和 Grails，组成 Spring IO 平台的执行部分。

Spring Data: Spring Data 是为了简化构建基于 Spring 框架应用的数据访问实现，包括非关系数据库、Map-Reduce 框架、云数据服务等；另外，也包含对关系数据库的访问支持。

Spring Integration: Spring Integration 为企业数据集成提供了各种适配器，可以通过这些适配器来转换各种消息格式，并帮助 Spring 应用完成与企业应用系统的集成。

Spring Batch: Spring Batch 是一个轻量级的完整批处理框架，旨在帮助应用开发者构建一个健壮、高效的企业级批处理应用（这些应用的特点是不需要与用户交互，重复的操作量大，对于大容量的批量数据处理而言，这些操作往往要求较高的可靠性）

Spring Security: Spring Security 是一个能够为基于 Spring 的企业应用系统提供声明式的安全访问控制解决方案的安全框架。它提供了一组可以在 Spring 应用上下文配置的 bean，充分利用 IoC 和 AOP 功能，为应用系统提供声明式的安全访问控制功能。

Spring Hateoas: Spring Hateoas 是一个用户支持实现超文本驱动的 REST Web 服务的开发库，是 Hateoas 的实现。Hateoas(Hypermedia as the engine of application state)是 REST 架构风格中最复杂的约束，也是构建成熟 REST 服务的核心。它的重要性在于打破了客户端和服务器之间严格的契约，是的客户端可以更加智能和自适应。

Spring Social: Spring Social 是 Spring 框架的扩展，用来方便开发 Web 社交应用程序，可通过该项目来创建与各种社交网站的交互，如 Facebook, LinkedIn、Twitter 等。

Spring AMQP: Spring AMQP 是基于 Spring 框架的 AMQP 消息解决方案，提供模块化的发送和接收消息的抽象层，提供基于消息驱动的 POJO。这个项目支持 Java 和 .NET 连个版本。Spring Source 旗下的 Rabbit MQ 就是一个开源的基于 AMQP 的消息服务器。

Spring for Android: Spring for Android 为 Android 终端开发应用提供 Spring 的支持，它提供了一个在 Android 应用环境中工作、基于 Java 的 REST 客户端。

Spring Mobile: Spring Mobile 是基于 Spring MVC 构建的，为移动端的服务器应用开发提供支持。

Spring Web Flow: Spring Web Flow (SWF) 一个建立在 Spring MVC 基础上的 Web 页面流引擎。

Spring Web Service: Spring Web Service 是基于 Spring 框架的 Web 服务框架，主要侧重于基于文档驱动的 Web 服务，提供 SOAP 服务开发，允许通过多种方式创建 Web 服务。

Spring LDAP: Spring LDAP 是一个用户操作 LDAP 的 Java 框架，类似 Spring JDBC 提供了 JdbcTemplate 方式来操作数据库。这个框架提供了一个 LdapTemplate 操作模版，可帮助开发人员简化 looking up、closing contexts、encoding/decoding、filters 等操作。

Spring Session: Spring Session 致力于提供一个公共基础设施会话，支持从任意环境中访问一个会话，在 Web 环境下支持独立于容器的集群会话，支持可插拔策略来确定 Session ID，WebSocket 活跃的时候可以简单地保持 HttpSession。

Spring Shell: Spring Shell 提供交互式的 Shell，用户可以简单的基于 Spring 的编程模型来开发命令。

Spring 源码下载

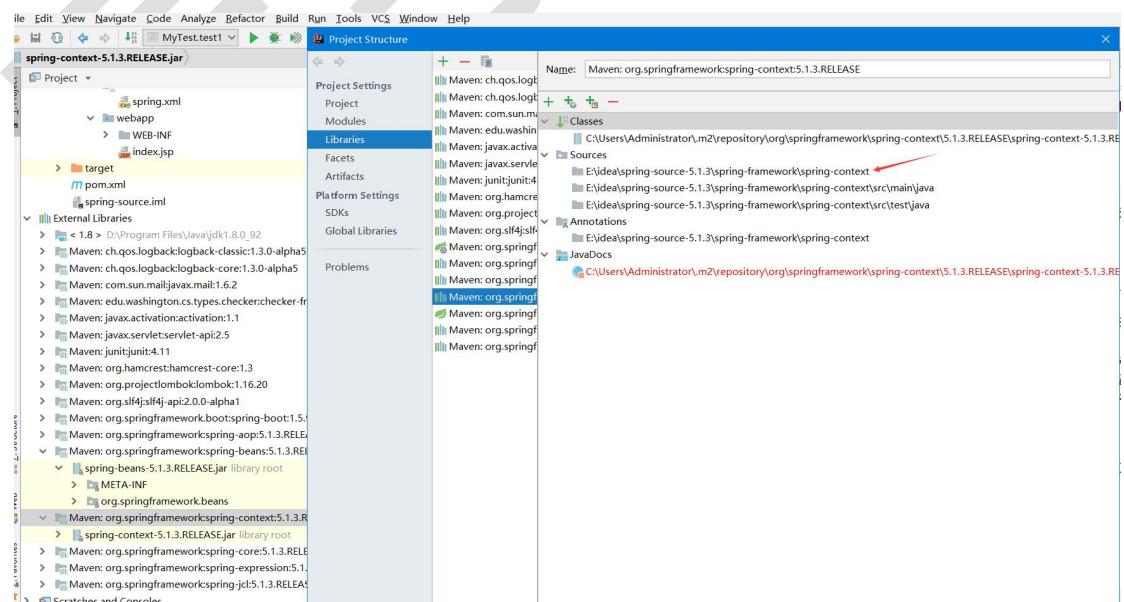
1、git clone --branch v5.1.3.RELEASE
<https://gitee.com/Z201/spring-framework.git>

2、gradle 下载，gradle 要 JDK8 的版本
3、到下载的 spring 源码路径执行 gradle 命令，
gradlew :spring-oxm:compileTestJava

```
E:\idea\spring-source-5.1.3\spring-framework>gradlew :spring-oxm:compileTestJava
```

4、用 idea 打开 spring 源码工程，在 idea 中安装插件 kotlin，重启 idea

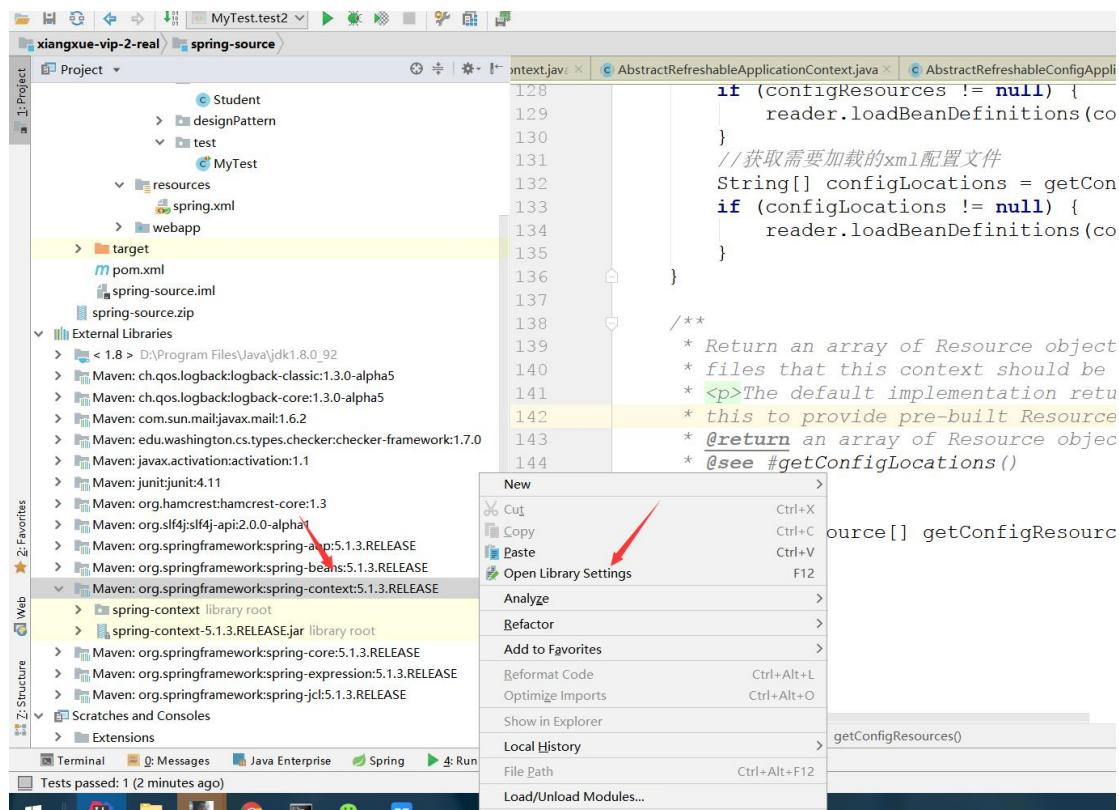
5、把编译好的源码导入到工程中



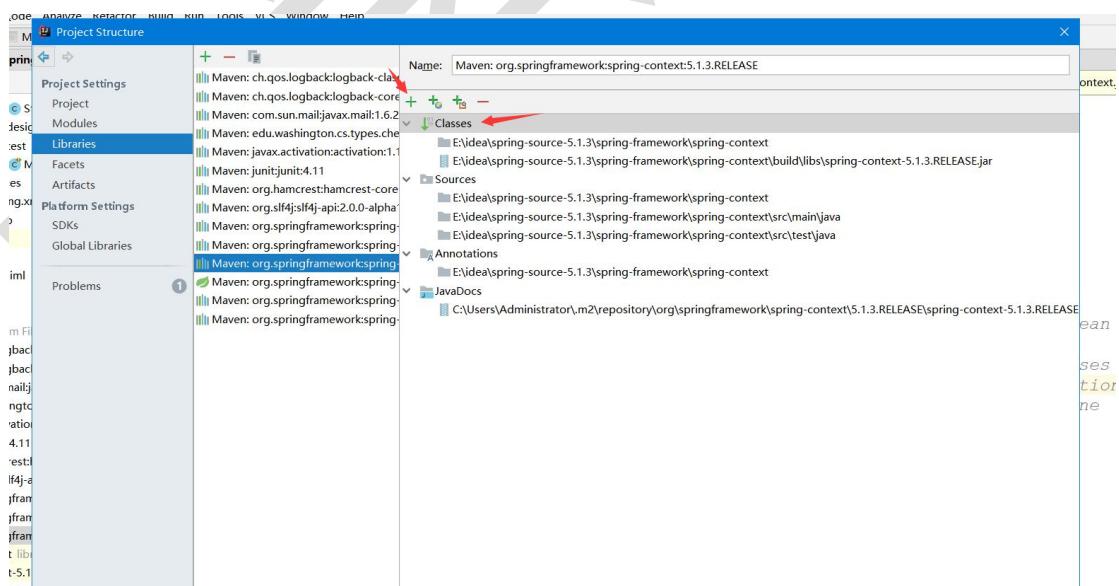
这样我们就可以在源码中写注释并且断点调试源码了。

把源码导入到工程

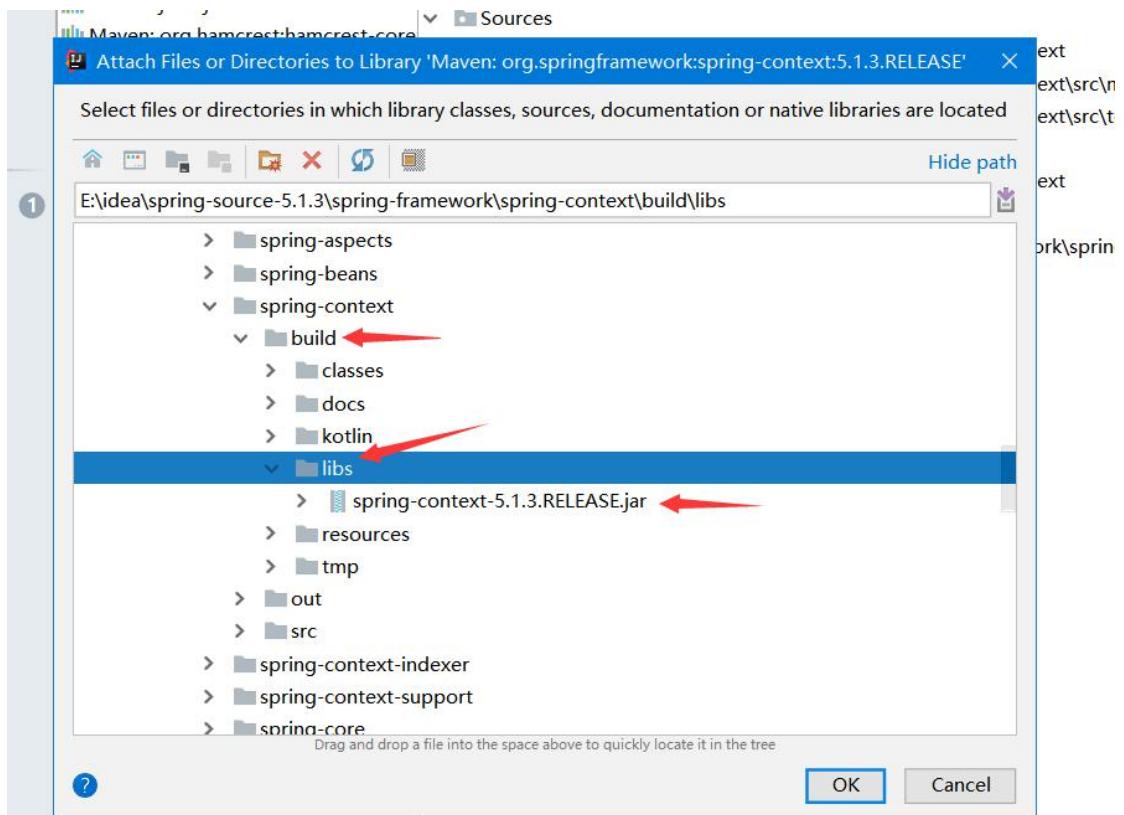
第一步



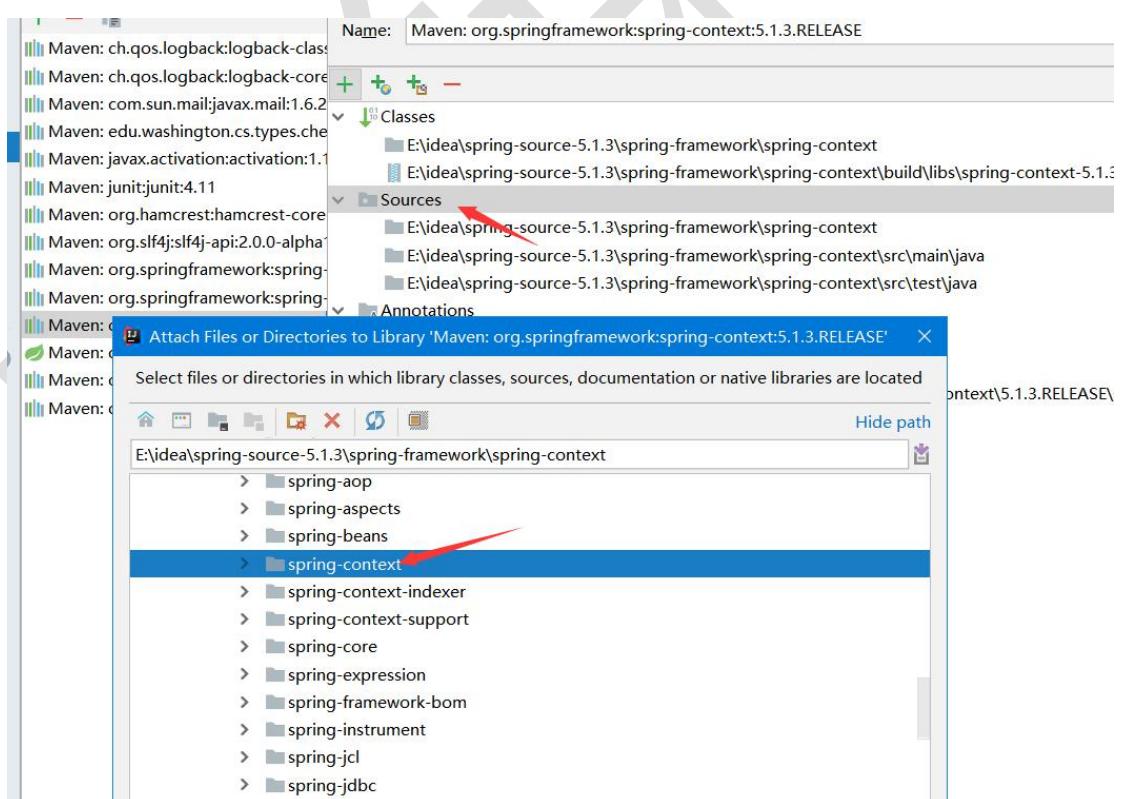
第二步



第三步



第四步



导入 jar 依赖

spring 中最核心的 4 个 jar

```
spring-beans  
spring-core  
spring-context  
spring-expression
```

一个最最简单的 spring 工程，理论上就只需要一个 jar 就够了

```
<dependency>  
    <groupId>org.springframework</groupId>  
    <artifactId>spring-context</artifactId>  
    <version>${spring.version}</version>  
</dependency>
```

spring-context 包本身就依赖了，spring-aop,spring-beans,spring-core 包

```
    <dependency>  
        <groupId>org.springframework</groupId>  
        <artifactId>spring-context</artifactId>  
        <version>5.1.3.RELEASE</version>  
    </dependency>  
    > <dependency>  
    >     <groupId>org.springframework</groupId>  
    >     <artifactId>spring-aop</artifactId>  
    >     <version>5.1.3.RELEASE</version>  
    > </dependency>  
    > <dependency>  
    >     <groupId>org.springframework</groupId>  
    >     <artifactId>spring-beans</artifactId>  
    >     <version>5.1.3.RELEASE</version>  
    > </dependency>  
    > <dependency>  
    >     <groupId>org.springframework</groupId>  
    >     <artifactId>spring-core</artifactId>  
    >     <version>5.1.3.RELEASE</version>  
    > </dependency>
```

一个空的 spring 工程是不能打印日志的，要导入 spring 依赖的日志 jar

```
<dependency>  
    <groupId>ch.qos.logback</groupId>  
    <artifactId>logback-classic</artifactId>  
    <version>LATEST</version>  
</dependency>
```

spring 中 XSD 引入

XSD 是编写 xml 文件的一种规范，有了这个规范才能校验 xml 是否写错，在 spring 中同样有 XSD 规范。例如要引入自定义标签

```
<context:component-scan base-package="com.xx.jack"/>
```

就必须引入这个标签对应的 XSD 文件

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context.xsd">

    <!--      <bean id="student" class="com.xx.jack.bean.Student">
        <property name="password" value="123"/>
    </bean>-->
    <context:component-scan base-package="com.xiangxue.jack"/>
</beans>
```

spring 容器加载方式

1、类路径获取配置文件

```
ApplicationContext applicationContext = new
ClassPathXmlApplicationContext("spring.xml");
```

2、文件系统路径获取配置文件【绝对路径】

```
ApplicationContext applicationContext = new
FileSystemXmlApplicationContext("E:\\idea\\public\\springdemo\\src\\main\\resources\\spring.xml");
```

3、无配置文件加载容器

```
ApplicationContext applicationContext = new
AnnotationConfigApplicationContext("com.xx.jack");
```

4、springboot 加载容器

```
ApplicationContext applicationContext = new
EmbeddedWebApplicationContext();
```

spring 容器加载核心方法

AbstractApplicationContext.refresh()方法

refresh()方法是 spring 容器启动过程中的核心方法，spring 容器要加载必须执行该方法。

下面我们来主要精读一下这个方法的源码。首先掌握一个模板设计模式。

设计模式 1-模板设计模式

在 spring 中大量的使用了模板设计模式，可以说是用得最多的设计模式。

```

public abstract class 爸爸 {
    public void 生活() {
        学习();
        工作();
        爱情();
    }

    public void 学习() { System.out.println("====要认认真真学习===="); }

    public void 工作() { System.out.println("====主动承担责任===="); }

    /**
     * 该方法就是一个钩子方法，通过子类的实现干预父类的方法的业务流程
     * 钩子方法挂载到父类方法中执行
     */
    //这里爸爸不强迫孩子的爱情，孩子自己实现自己的爱情
    public abstract void 爱情();
}

public class 张三 extends 爸爸 {
    @Override
    public void 爱情() {
        System.out.println("=====找肤白貌美大长腿=====");
    }
}

public class 张四 extends 爸爸 {
    @Override
    public void 爱情() {
        System.out.println("=====找有钱的=====");
    }
}

```

同学们一定要注意对模板设计模式中钩子方法的理解，其实就是通过子类实现钩子方法来干预父类的业务执行流程。

xml 解析

```
ConfigurableListableBeanFactory beanFactory = obtainFreshBeanFactory();
```

该方法主要进行 xml 解析工作，流程如下：

1、创建 XmlBeanDefinitionReader 对象

```
// Create a new XmlBeanDefinitionReader for the given BeanFactory.
XmlBeanDefinitionReader beanDefinitionReader = new XmlBeanDefinitionReader(beanFactory);
```

2、通过 Reader 对象加载配置文件

```

protected void loadBeanDefinitions(XmlBeanDefinitionReader reader) throws BeansException, IOException {
    Resource[] configResources = getConfigResources();
    if (configResources != null) {
        reader.loadBeanDefinitions(configResources);
    }
    String[] configLocations = getConfigLocations();
    if (configLocations != null) {
        reader.loadBeanDefinitions(configLocations);
    }
}

```

3、根据加载的配置文件把配置文件封装成 document 对象

```
protected int doLoadBeanDefinitions(InputSource inputSource, Resource resource)
throws BeanDefinitionStoreException {

try {
    Document doc = doLoadDocument(inputSource, resource);
    int count = registerBeanDefinitions(doc, resource);
    if (logger.isDebugEnabled()) {
        logger.debug("Loaded " + count + " bean definitions from " + resource);
    }
    return count;
}
catch (BeanDefinitionStoreException ex) {
```

4、创建 BeanDefinitionDocumentReader 对象， DocumentReader 负责对 document 对象解析

```
/*
public int registerBeanDefinitions(Document doc, Resource resource) throws BeanDefinitionStoreException {
    BeanDefinitionDocumentReader documentReader = createBeanDefinitionDocumentReader();
    int countBefore = getRegistry().getBeanDefinitionCount();
    documentReader.registerBeanDefinitions(doc, createReaderContext(resource));
    return getRegistry().getBeanDefinitionCount() - countBefore;
}
```

5、parseDefaultElement(ele, delegate);负责常规标签解析

6、delegate.parseCustomElement(ele);负责自定义标签解析

```
protected void parseBeanDefinitions(Element root, BeanDefinitionParserDelegate delegate) {
    if (delegate.isDefaultNamespace(root)) {
        NodeList nl = root.getChildNodes();
        for (int i = 0; i < nl.getLength(); i++) {
            Node node = nl.item(i);
            if (node instanceof Element) {
                Element ele = (Element) node;
                if (delegate.isDefaultNamespace(ele)) {
                    parseDefaultElement(ele, delegate);
                }
                else {
                    delegate.parseCustomElement(ele);
                }
            }
        }
    }
    else {
        delegate.parseCustomElement(root);
    }
}
```

7、最终解析的标签封装成 BeanDefinition 并缓存到容器中

8、Xml 流程分析



自定义标签解析

1、获取自定义标签的 namespace 命令空间，例如：

<http://www.springframework.org/schema/context>

```
String namespaceUri = getNamespaceURI(ele);
```

2、根据命令空间获取 NamespaceHandler 对象。

NamespaceUri 和 NamespaceHandler 之间会建立一个映射, spring 会从所有的 spring jar 包中扫描 spring.handlers 文件, 建立映射关系。

```
NamespaceHandler handler =  
this.readerContext.getNamespaceHandlerResolver().resolve(  
namespaceUri);  
  
Map<String, Object> handlerMappings =  
getHandlerMappings();  
Object handlerOrClassName =  
handlerMappings.get(namespaceUri);
```

3、反射获取 NamespaceHandler 实例

```
NamespaceHandler namespaceHandler = (NamespaceHandler)  
BeanUtils.instantiateClass(handlerClass);
```

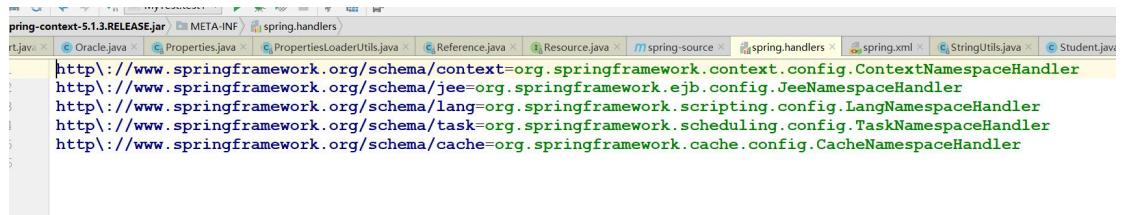
4、调用 init 方法

```
namespaceHandler.init();
```

5、调用 parse 方法

```
handler.parse(ele, new ParserContext(this.readerContext,  
this, containingBd))
```

spring.handler 文件



其实就是 namespaceUri 和类的完整限定名的映射

设计模式 2-委托模式

有两个对象参与处理同一个请求, 接受请求的对象将请求委托给另一个对象来处理。在 spring 中用得比较多

```
public interface Company {  
  
    void product();  
}
```

```
public class Sun implements Company {  
    @Override  
    public void product() {  
        System.out.println("sun product");  
    }  
}
```

```
public class Boss implements Company {  
  
    Sun sun = new Sun();  
  
    @Override  
    public void product() {  
        sun.product();  
    }  
}
```

设计模式 3-装饰模式

装饰模式有几个元素很重要，1、被装饰者。2、抽象装饰者。3、装饰者对象

```
public interface House {  
  
    //人  
    public void people();  
  
    //物品  
    public void goods();  
}
```

```
/*
 * JackHouse2001是被装饰对象，后续装饰者对该对象进行修改装饰
 */
@Data
public class JackHouse2001 implements House{

    private List<String> peoples = new ArrayList<>();

    private List<String> goods = new ArrayList<>();

    @Override
    public void people() {
        peoples.add("老爸");
        peoples.add("老妈");
        peoples.add("Jack");
        System.out.println("=====2001年Jack家里只有老爸，老妈和Jack=====");
    }

    @Override
    public void goods() {
        goods.add("手电筒");
        System.out.println("=====2001年Jack家里只有一个手电筒=====");
    }
}
```

```
/*
 * 装饰者类，是对房子的装饰，所以需要实现house接口
 *
 * 装饰者规定了装饰流程，就是规定了接口调用过程，具体子类实例方法如何调用
 * 子类去实现
 */
public abstract class Decorator implements House {

    public House house;

    public Decorator(House house) { this.house = house; }

    @Override
    public void people() { house.people(); }

    @Override
    public void goods() { house.goods(); }
}
```

具体装饰者对象

```
public class Decorator2016 extends Decorator {  
    public Decorator2016(House house) { super(house); }  
  
    public void findWife() {...}  
  
    @Override  
    public void people() {  
        //2001年的人都还在, Jack, 老爸, 老妈  
        super.people();  
  
        findWife();  
    }  
  
    public void addGoods() {...}  
  
    @Override  
    public void goods() {  
        //2001年的手电筒还在  
        super.goods();  
  
        addGoods();  
    }  
}
```

BeanDefinition 简介

BeanDefinition 在 spring 中贯穿始终, spring 要根据 BeanDefinition 对象来实例化 bean, 只要把解析的标签, 扫描的注解类封装成 BeanDefinition 对象, spring 才能实例化 bean

beanDefinition 实现类

ChildBeanDefinition, GenericBeanDefinition, RootBeanDefinition

ChildBeanDefinition

ChildBeanDefinition 是一种 bean definition, 它可以继承它父类的设置, 即 ChildBeanDefinition 对 RootBeanDefinition 有一定的依赖关系。

ChildBeanDefinition 从父类继承构造参数值, 属性值并可以重写父类的方法, 同时也可以增加新的属性或者方法。(类同于 java 类的继承关系)。若指定初始化方法, 销毁方法或者静态工厂方法, ChildBeanDefinition 将重写相应父类的设置。
depends on, autowire mode, dependency check, singleton, lazy init 一般由子类自行设定。

GenericBeanDefinition

注意: 从 spring 2.5 开始, 提供了一个更好的注册 bean definition 类 GenericBeanDefinition, 它支持动态定义父依赖, 方法是 GenericBeanDefinition.setParentName(java.lang.String), GenericBeanDefinition 可以有效的替代 ChildBeanDefinition 的绝大部分使用场合。

GenericBeanDefinition 是一站式的标准 bean definition，除了具有指定类、可选的构造参数值和属性参数这些其它 bean definition 一样的特性外，它还具有通过 parentName 属性来灵活设置 parent bean definition。

通常，GenericBeanDefinition 用来注册用户可见的 bean definition(可见的 bean definition 意味着可以在该类 bean definition 上定义 post-processor 来对 bean 进行操作，甚至为配置 parent name 做扩展准备)。RootBeanDefinition / ChildBeanDefinition 用来预定义具有 parent/child 关系的 bean definition。

RootBeanDefinition

一个 RootBeanDefinition 定义表明它是一个可合并的 bean definition：即在 spring beanFactory 运行期间，可以返回一个特定的 bean。RootBeanDefinition 可以作为一个重要的通用的 bean definition 视图。

RootBeanDefinition 用来在配置阶段进行注册 bean definition。然后，从 spring 2.5 后，编写注册 bean definition 有了更好的的方法：GenericBeanDefinition。GenericBeanDefinition 支持动态定义父类依赖，而非硬编码作为 root bean definition。

GenericBeanDefinition 创建实例

```
@Component
public class BeanDefinitionTest implements InitializingBean,BeanDefinitionRegistryPostProcessor {
    @Override
    public void afterPropertiesSet() throws Exception {
    }

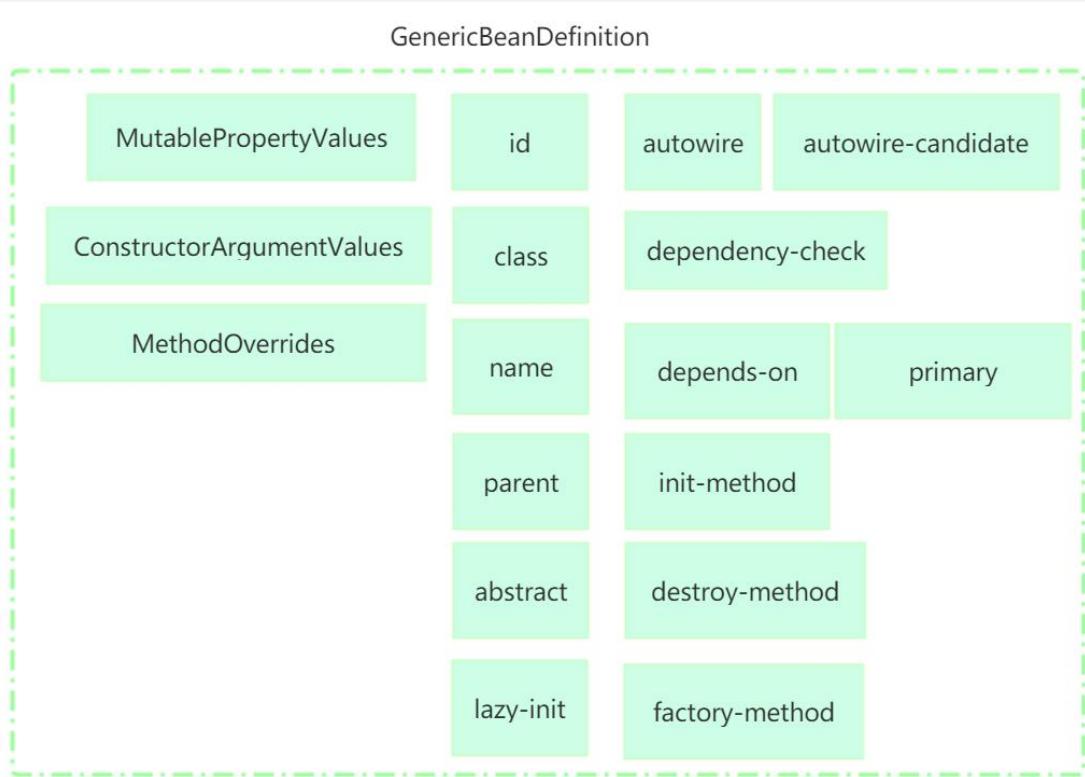
    @Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {
        GenericBeanDefinition genericBeanDefinition = new GenericBeanDefinition();
        genericBeanDefinition.setBeanClass(BeanClass.class);

        MutablePropertyValues propertyValues = genericBeanDefinition.getPropertyValues();
        propertyValues.addPropertyValue(propertyName: "username", propertyValue: "peter");

        registry.registerBeanDefinition(beanName: "beanClass", genericBeanDefinition);
    }

    @Override
    public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
    }
}
```

BeanDefinition 中的属性



- (1)、id:Bean 的唯一标识名。它必须是合法的 XMLID , 在整个 XML 文档中唯一。
- (2)、name:用来为 id 创建一个或多个别名。它可以是任意的字母符合。多个别名之间用逗号或空格分隔。
- (3)、class:用来定义类的全限定名 (包名 + 类名) 。只有子类 Bean 不用定义该属性。
- (4)、parent:子类 Bean 定义它所引用它的父类 Bean。这时前面的 class 属性失效。子类 Bean 会继承父类 Bean 的所有属性，子类 Bean 也可以覆盖父类 Bean 的属性。注意：子类 Bean 和父类 Bean 是同一个 Java 类。
- (5)、abstract (默认为“ false ”) : 用来定义 Bean 是否为抽象 Bean。它表示这个 Bean 将不会被实例化，一般用于父类 Bean ，因为父类 Bean 主要是供子类 Bean 继承使用。
- (7)、lazy-init (默认为 “ default ”) : 用来定义这个 Bean 是否实现懒初始化。如果为 “ true ” , 它将在 BeanFactory 启动时初始化所有的 SingletonBean 。反之，如果为 “ false ” , 它只在 Bean 请求时才开始创建 SingletonBean 。
- (8)、autowire (自动装配，默认为 “ default ”) : 它定义了 Bean 的自动装载方式。

1、“no” :不使用自动装配功能。

-
- 2、 “byName” :通过 Bean 的属性名实现自动装配。
 - 3、 “byType” :通过 Bean 的类型实现自动装配。
 - 4、 “constructor” :类似于 byType , 但它是用于构造函数的参数的自动组装。
 - 5、 “autodetect” :通过 Bean 类的反省机制 (introspection) 决定是使用 “constructor” 还是使用 “byType” 。
 - (10) 、 depends-on (依赖对象) :这个 Bean 在初始化时依赖的对象 , 这个对象会在这个 Bean 初始化之前创建。
 - (11) 、 init-method:用来定义 Bean 的初始化方法 , 它会在 Bean 组装之后调用。它必须是一个无参数的方法。
 - (12) 、 destroy-method :用来定义 Bean 的销毁方法 , 它在 BeanFactory 关闭时调用。同样 , 它也必须是一个无参数的方法。它只能应用于 singletonBean 。
 - (13) 、 factory-method :定义创建该 Bean 对象的工厂方法。它用于下面的 “factory-bean” , 表示这个 Bean 是通过工厂方法创建。此时 , “class” 属性失效。
 - (14) 、 factory-bean:定义创建该 Bean 对象的工厂类。如果使用了 “factory-bean” 则 “class” 属性失效。
 - (15) 、 autowire-candidate :采用 xml 格式配置 bean 时 , 将<bean/>元素的 autowire-candidate 属性设置为 false , 这样容器在查找自动装配对象时 , 将不考虑该 bean , 即它不会被考虑作为其它 bean 自动装配的候选者 , 但是该 bean 本身还是可以使用自动装配来注入其它 bean 的。
 - (16) 、 MutablePropertyValues :用于封装<property>标签的信息 , 其实类里面就是一个 list , list 里面是 PropertyValue 对象 , PropertyValue 就是一个 name 和 value 属性 , 用于封装<property>标签的名称和值信息
 - (17) 、 ConstructorArgumentValues :用于封装<constructor-arg>标签的信息 , 其实类里面就是一个 map , map 中用构造函数的参数顺序作为 key , 值作为 value 存储到 map 中
 - (18) 、 MethodOverrides :用于封装 lookup-method 和 replaced-method 标签的信息 , 同样的类里面有一个 Set 对象添加 LookupOverride 对象和 ReplaceOverride 对象

2、Bean 的实例化过程

BeanDefinitionRegistryPostProcessor 接口

Refresh().invokeBeanFactoryPostProcessors 这个方法里面。

BeanDefinitionRegistryPostProcessor 这个接口的调用分为三部：

- 1、调用实现了 PriorityOrdered 排序接口
- 2、调用实现了 Ordered 排序接口
- 3、没有实现接口的调用

这个接口的理解：获取 BeanDefinitionRegistry 对象，获取到这个对象就可以获取这个对象中注册的所有 BeanDefinition 对象，所以我们就知道了，我们拥有这个对象就可以完成里面所有 BeanDefinition 对象的修改我新增操作

```
10
11  @Component
12  public class BeanDefinitionTest implements BeanDefinitionRegistryPostProcessor {
13
14      @Override
15      public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry registry) throws BeansException {
16          GenericBeanDefinition genericBeanDefinition = new GenericBeanDefinition();
17          genericBeanDefinition.setBeanClass(BeanClass.class);
18
19          // ...
20
21          MutablePropertyValues propertyValues = genericBeanDefinition.getPropertyValues();
22          propertyValues.addProperty(propertyName: "username", propertyValue: "peter");
23
24          registry.registerBeanDefinition(beanName: "beanClass", genericBeanDefinition);
25
26          /*ClassPathBeanDefinitionScanner scanner = new ClassPathBeanDefinitionScanner(registry);
27          scanner.addIncludeFilter(new AnnotationTypeFilter(MyService.class));
28          scanner.scan("com.xiangxue.jack");*/
29      }
30
31      @Override
32      public void postProcessBeanFactory(ConfigurableListableBeanFactory beanFactory) throws BeansException {
33
34  }
```

BeanPostProcessor 的注册

Refresh().registerBeanPostProcessors 这个方法里面。

```
//拿到工程里面所有实现了BeanPostProcessor接口的类，获取到BeanDefinition的名称
String[] postProcessorNames = beanFactory.getBeanNamesForType(BeanPostProcessor.class, includeNonSingletons: true)
```

拿到 BeanFactory 中所有注册的 BeanDefinition 对象的名称 beanName。

```

//提前实例化BeanPostProcessor类型的bean，然后bean进行排序
for (String ppName : postProcessorNames) {
    if (beanFactory.isTypeMatch(ppName, PriorityOrdered.class)) {
        //getBean是实例化方法，后面我们在讲bean实例化过程是会着重讲到
        BeanPostProcessor pp = beanFactory.getBean(ppName, BeanPostProcessor.class);
        priorityOrderedPostProcessors.add(pp);

        //判断类型是否是MergedBeanDefinitionPostProcessor，如果是则代码是内部使用的
        if (pp instanceof MergedBeanDefinitionPostProcessor) {
            internalPostProcessors.add(pp);
        }
    } else if (beanFactory.isTypeMatch(ppName, Ordered.class)) {
        orderedPostProcessorNames.add(ppName);
    } else {
        nonOrderedPostProcessorNames.add(ppName);
    }
}

```

然后判断是否实现了 PriorityOrdered 排序接口， Ordered 排序接口， getBean 是将该 ppName 对应的 BeanDefinition 对象实例化。

```
//注册到BeanFactory中
registerBeanPostProcessors(beanFactory, priorityOrderedPostProcessors);
```

把对应的 BeanPostProcessor 对象注册到 BeanFactory 中， BeanFactory 中有一个 list 容器接收。

getSingleton 方法

代码位置：

AbstractBeanFactory.doGetBean 方法中

```

//着重看，大部分是单例的情况
// Create bean instance.
if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, () -> {
        try {
            return createBean(beanName, mbd, args);
        }
        catch (BeansException ex) {
            // Explicitly remove instance from singleton cache: It might have been
            // eagerly by the creation process, to allow for circular reference
            // Also remove any beans that received a temporary reference to the
            destroySingleton(beanName);
            throw ex;
        }
    });
    //改方法是FactoryBean接口的调用入口
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);
}

```

方法里面核心要点：

```
//把beanName添加到singletonsCurrentlyInCreation Set容器中，在这个集合里面的bean都是正在实例化的bean
beforeSingletonCreation(beanName);
```

```
protected void beforeSingletonCreation(String beanName) {
    //把beanName添加到singletonsCurrentlyInCreation Set容器中，在这个集合里面的bean都是正在实例化的bean
    if (!this.inCreationCheckExclusions.contains(beanName) && !this.singletonsCurrentlyInCreation.add(beanName))
        throw new BeanCurrentlyInCreationException(beanName);
}
```

把 beanName 添加到 singletonsCurrentlyInCreation Set 容器中，在这个集合里面的 bean 都是正在实例化的 bean，就是实例化还没做完的 BeanName

```
try {
    singletonObject = singletonFactory.getObject();
    newSingleton = true;
}
```

调到 getObject 方法，完成 bean 的实例化。接下来

```
finally {
    if (recordSuppressedExceptions) {
        this.suppressedExceptions = null;
    }
    //bean创建完成后singletonsCurrentlyInCreation要删除该bean
    afterSingletonCreation(beanName);
}
if (newSingleton) {
    //创建对象成功时，把对象缓存到singletonObjects缓存中，bean创建完成时放入一级缓存
    addSingleton(beanName, singletonObject);
}
```

getObject 调用完后，就代表着 Bean 实例化已经完成了，这时候就需要

1、singletonsCurrentlyInCreation 把 beanName 从这个集合中删除

2、addSingleton，把 bean 缓存到一级缓存中

```
protected void addSingleton(String beanName, Object singletonObject) {
    synchronized (this.singletonObjects) {
        //一级缓存
        this.singletonObjects.put(beanName, singletonObject);
        //三级缓存
        this.singletonFactories.remove(beanName);
        //二级缓存
        this.earlySingletonObjects.remove(beanName);
        this.registeredSingletons.add(beanName);
    }
}
```

createBean 方法

代码位置：

AbstractBeanFactory.doGetBean 方法中

```

//着重看，大部分是单例的情况
// Create bean instance.
if (mbd.isSingleton()) {
    sharedInstance = getSingleton(beanName, () -> {
        try {
            return createBean(beanName, mbd, args);
        }
        catch (BeansException ex) {
            // Explicitly remove instance from singleton cache: It might have .
            // eagerly by the creation process, to allow for circular referenc
            // Also remove any beans that received a temporary reference to th
            destroySingleton(beanName);
            throw ex;
        }
    });
    //改方法是FactoryBean接口的调用入口
    bean = getObjectTypeForBeanInstance(sharedInstance, name, beanName, mbd);
}

```

这个方法是 bean 实例化核心方法

```

if (instanceWrapper == null) {
    //创建实例,,重点看, 重要程度: 5
    instanceWrapper = createBeanInstance(beanName, mbd, args);
}

```

实例化方法，把 bean 实例化，并且包装成 BeanWrapper

1、点进这个方法里面。

```

//如果有FactoryMethodName属性
if (mbd.getFactoryMethodName() != null) {
    return instantiateUsingFactoryMethod(beanName, mbd, args);
}

```

这个方法是反射调用类中的 factoryMethod 方法。这要知道 @Bean 方法的原理，实际上 spring 会扫描有 @bean 注解的方法，然后把方法名称设置到 BeanDefinition 的 factoryMethod 属性中，接下来就会调到上面截图中的方法实现 @Bean 方法的调用。该方法里面的参数解析过程不需要了解。

2、有参构造函数的时候

```

// Candidate constructors for autowiring?
//寻找当前正在实例化的bean中有@Autowired注解的构造函数
Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
if (ctors != null || mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR ||
    mbd.hasConstructorArgumentValues() || !ObjectUtils.isEmpty(args)) {
    //如果ctors不为空，就说明构造函数上有@Autowired注解
    return autowireConstructor(beanName, mbd, ctors, args);
}

```

determineConstructorsFromBeanPostProcessors

这个方法是 BeanPostProcessor 接口类的首次应用，最终会掉到 AutowiredAnnotationBeanPostProcessor 类的方法，在方法中会扫描有注解的构造函数然后完成装配过程。然后把有有 @Autowired 注解的构造函数返回。

3、无参构造函数的实例化

```
//无参构造函数的实例化，大部分的实例是采用的无参构造函数的方式实例化  
// No special handling: simply use no-arg constructor.  
return instantiateBean(beanName, mbd);
```

这就是简单的反射实例化。大部分类的实例化都会走这个逻辑

4、类中注解的收集

实例化完成后接下来就需要对类中的属性进行依赖注入操作，但是类里面属性和方法的依赖注入往往用 @Autowired 或者 @Resource 注解，那么这些注解的依赖注入是如何完成的呢？

注解的收集：

```
// Allow post-processors to modify the merged bean definition.  
synchronized (mbd.postProcessingLock) {  
    if (!mbd.postProcessed) {  
        try {  
            //CommonAnnotationBeanPostProcessor 支持了@PostConstruct, @PreDestroy, @Resource注解  
            //AutowiredAnnotationBeanPostProcessor 支持 @Autowired, @Value注解  
            //对类中注解的装配过程  
            //重要程度5，必须看  
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);  
        } catch (Throwable ex) {  
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,  
                "Post-processing of merged bean definition failed", ex);  
        }  
        mbd.postProcessed = true;  
    }  
}
```

也是通过 BeanPostProcessor 接口类型实例来挨个处理的。

A、首先是

CommonAnnotationBeanPostProcessor 类，这个类完成了 @Resource 注解的属性或者方法的收集

这个类还对 @PostConstruct 和 @PreDestroy 支持

```
public CommonAnnotationBeanPostProcessor() {  
    setOrder(Ordered.LOWEST_PRECEDENCE - 3);  
    setInitAnnotationType(PostConstruct.class);  
    setDestroyAnnotationType(PreDestroy.class);  
    ignoreResourceType("javax.xml.ws.WebServiceContext");  
}
```

收集过程

- 1、看缓存里面有没 InjectionMetadata 对象
- 2、从类中获取所有 Field 对象，循环 field 对象，判断 field 有没有 @Resource 注解，如果有注解封装成 ResourceElement 对象
- 3、从类中获取所有 Method 对象，循环 Method 对象，判断 Method 有没有 @Resource 注解，如果有注解封装成 ResourceElement 对象
- 4、最终把两个 field 和 Method 封装的对象集合封装到 InjectionMetadata 对象中

B、然后是

AutowiredAnnotationBeanPostProcessor 类，对 @Autowired 注解的属性和方法的收集。收集过程基本上跟 @Resource 注解的收集差不多，这里就不赘述了。

5、IOC\DI 依赖注入

对应的方法：

```
//ioc di, 依赖注入的核心方法, 该方法必须看, 重要程度: 5
populateBean(beanName, mbd, instanceWrapper);
```

看到这个 if 代码块

```
//重点看这个if代码块, 重要程度 5
if (hasInstAwareBpps) {
    if (pvs == null) {
        pvs = mbd.getPropertyValues();
    }
    for (BeanPostProcessor bp : getBeanPostProcessors()) {
        if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
            //依赖注入过程, @Autowired 的支持
            PropertyValues pvsToUse = ibp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName);
            if (pvsToUse == null) {
                if (filteredPds == null) {
                    filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
                }
                //老版本用这个完成依赖注入过程, @Autowired 的支持
                pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);
                if (pvsToUse == null) {
                    return;
                }
            }
            pvs = pvsToUse;
        }
    }
}
```

这里又是一个 BeanPostProcessor 类型接口的运用，前面我们讲到了 @Resource @Autowired 注解的收集，那么这个方法就是根据收集到的注解进行反射调用。

```
public void inject(Object target, @Nullable String beanName, @Nullable PropertyValues pvs) throws Throwable {
    Collection<InjectedElement> checkedElements = this.checkedElements;
    Collection<InjectedElement> elementsToIterate =
        (checkedElements != null ? checkedElements : this.injectedElements);
    if (!elementsToIterate.isEmpty()) {
        for (InjectedElement element : elementsToIterate) {
            if (logger.isTraceEnabled()) {
                logger.trace(o: "Processing injected element of bean '" + beanName + "' : " + element);
            }
            element.inject(target, beanName, pvs);
        }
    }
}
```

循环收集到的 metaData 中的 list 对象，然后挨个调用里面的 InjectedElement 的 inject 方法完成依赖注入。

```
if (value != null) {
    ReflectionUtils.makeAccessible(field);
    field.set(bean, value);
}
```

其中 value 值的获取，如果依赖的属性是一个引用类型必定会触发该属性的 BeanFactory.getBean 操作，从而从 spring 容器中获取到对应的实例。方法的依赖注入类似这里就不再赘述。

上述是对注解@Resource 和@Autowired 的依赖注入的实现逻辑，xml 配置的依赖注入比如在 bean 标签中配置了：

```
<property name="username" value="Jack"/>
```

标签的依赖注入是这个逻辑：

```
//这个方法很鸡肋了，建议不看，是老版本用<property name="username" value="Jack"/>
//标签做依赖注入的代码实现，复杂且无用
if (pvs != null) {
    applyPropertyValues(beanName, mbd, bw, pvs);
}
```

这块逻辑是专门做 xml 配置依赖注入的，基本上现在基于 xml 配置的依赖很少使用，这里就不讲这块逻辑，没多大用。

6、bean 实例化后的操作

代码走到这里：

```
//bean 实例化+ioc 依赖注入完以后的调用，非常重要，重要程度：5
exposedObject = initializeBean(beanName, exposedObject, mbd);
```

A、首先是对某些 Aware 接口的调用

```
private void invokeAwareMethods(final String beanName, final Object bean) {
    if (bean instanceof Aware) {
        if (bean instanceof BeanNameAware) {
            ((BeanNameAware) bean).setBeanName(beanName);
        }
        if (bean instanceof BeanClassLoaderAware) {
            ClassLoader bcl = getBeanClassLoader();
            if (bcl != null) {
                ((BeanClassLoaderAware) bean).setBeanClassLoader(bcl);
            }
        }
        if (bean instanceof BeanFactoryAware) {
            ((BeanFactoryAware) bean).setBeanFactory(AbstractAutowireCapableBeanFactory.this);
        }
    }
}
```

B、然后@PostConstruct 注解方法的调用

```
Object wrappedBean = bean;
if (mbd == null || !mbd.isSynthetic()) {
    //对类中某些特殊方法的调用，比如@PostConstruct，Aware接口，非常重要 重要程度：5
    wrappedBean = applyBeanPostProcessorsBeforeInitialization(wrappedBean, beanName);
}
```

这里又是一个 BeanPostProcessor 接口的运用，

前面讲过，有@PostConstruct 注解的方法会收集到一个 metaData 对象中，现在就是通过 BeanPostProcessor 接口调到 CommonAnnotationBeanPostProcessor 类，然后在类中拿到 metaData 对象，根据对象里面的容器来反射调用有注解的方法。代码如下：

```

@Override
public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException {
    LifecycleMetadata metadata = findLifecycleMetadata(bean.getClass());
    try {
        //调用@PostConstruct注解的方法
        metadata.invokeInitMethods(bean, beanName);
    } catch (InvocationTargetException ex) {
        throw new BeanCreationException(beanName, "Invocation of init method failed", ex.getTargetException());
    } catch (Throwable ex) {
        throw new BeanCreationException(beanName, "Failed to invoke init method", ex);
    }
    return bean;
}

public void invokeInitMethods(Object target, String beanName) throws Throwable {
    Collection<LifecycleElement> checkedInitMethods = this.checkedInitMethods;
    Collection<LifecycleElement> initMethodsToIterate =
        (checkedInitMethods != null ? checkedInitMethods : this.initMethods);
    if (!initMethodsToIterate.isEmpty()) {
        for (LifecycleElement element : initMethodsToIterate) {
            if (logger.isTraceEnabled()) {
                logger.trace(o: "Invoking init method on bean '" + beanName + "' : " + element.getInitMethodName());
            }
            element.invoke(target);
        }
    }
}

```

有@PostConstruct注解的容器会收集到 initMethods 容器中，接下来就是方法的反射调用。

```

public void invoke(Object target) throws Throwable {
    ReflectionUtils.makeAccessible(this.method);
    this.method.invoke(target, (Object[]) null);
}

```

C、 InitializingBean 接口和 init-method 属性调用

```

//bean 实例化+ioc依赖注入完以后的调用，非常重要，重要程度：5
exposedObject = initializeBean(beanName, exposedObject, mbd);

```

实现了 InitializingBean 接口的类就必然会调用到 afterPropertiesSet

```

boolean isInitializingBean = (bean instanceof InitializingBean);
if (isInitializingBean && (mbd == null || !mbd.isExternallyManagedInitMethod("afterPropertiesSet")) {
    if (logger.isTraceEnabled()) {
        logger.trace(o: "Invoking afterPropertiesSet() on bean with name '" + beanName + "'");
    }
    if (System.getSecurityManager() != null) {
        try {
            AccessController.doPrivileged((PrivilegedExceptionAction<Object>) () -> {
                ((InitializingBean) bean).afterPropertiesSet();
                return null;
            }, getAccessControlContext());
        } catch (PrivilegedActionException pae) {
            throw pae.getException();
        }
    }
    else {
        ((InitializingBean) bean).afterPropertiesSet();
    }
}

```

Init-method 属性调用是在 afterPropertiesSet 之后

```
if (mbd != null && bean.getClass() != NullBean.class) {  
    String initMethodName = mbd.getInitMethodName();  
    if (StringUtils.hasLength(initMethodName) &&  
        !(isInitializingBean && "afterPropertiesSet".equals(initMethodName)) &&  
        !mbd.isExternallyManagedInitMethod(initMethodName)) {  
        invokeCustomInitMethod(beanName, bean, mbd);  
    }  
}
```



afterPropertiesSet 和 Init-method 和有 @PostConstruct 注解的方法其实核心功能都是一样的，只是调用时序不一样而已，都是在该类实例化和 IOC 做完后调用的，我们可以在这些方法中做一些在 spring 或者 servlet 容器启动的时候的初始化工作。比如缓存预热，比如缓存数据加载到内存，比如配置解析，等等初始化工作。

在这个方法里面还有一个重要的逻辑

```
if (mbd == null || !mbd.isSynthetic()) {  
    wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, beanName);  
}
```

也是一个 BeanPostProcessor 接口的运用，在这里会返回 bean 的代理实例，这个就是 AOP 的入口。

D、FactoryBean 接口

带入如下：

```
//着重看，大部分是单例的情况  
// Create bean instance.  
if (mbd.isSingleton()) {  
    sharedInstance = getSingleton(beanName, () -> {  
        try {  
            return createBean(beanName, mbd, args);  
        }  
        catch (BeansException ex) {  
            // Explicitly remove instance from singleton cache: It might have been put there  
            // eagerly by the creation process, to allow for circular reference resolution.  
            // Also remove any beans that received a temporary reference to the bean.  
            destroySingleton(beanName);  
            throw ex;  
        }  
    });  
    //改方法是FactoryBean接口的调用入口  
    bean = getObjectForBeanInstance(sharedInstance, name, beanName, mbd);  
}
```

在实例化和 IOC/DI 做完后，就会调用 FactoryBean 类型的接口，如果要获取到 FactoryBean 类本身，就必须加上“&”符号，比如
beanFactory.getBean("＆beanName")。

```
|  
| @Service  
| public class FactoryBeanDemo implements FactoryBean {  
|  
|     @Override  
|     public Object getObject() throws Exception {  
|         return new FactoryB();  
|     }  
|  
|     @Override  
|     public Class<?> getObjectType() {  
|         return FactoryB.class;  
|     }  
|  
| }
```

BeanFactory.getBean(“beanName”)只能获取到 getObject()方法返回的实例。

getObject 方法返回的实例会有单独的缓存存储，跟其他实例不是同一个缓存，对应的缓存是：factoryBeanObjectCache

E、循环依赖

循环依赖请参照流程图理解

<https://www.processor.com/view/link/5df9ce52e4b0c4255ea1a84f>

循环依赖只会出现在单例实例无参构造函数实例化情况下

有参构造函数的加@Autowired 的方式循环依赖是直接报错的，多例的循环依赖也是直接报错的。

```
| @Data  
| @Service  
| public class CircularRefA {  
|  
|     @Value("CircularRefA")  
|     private String username;  
|  
|     @Autowired  
|     private CircularRefB circularRefB;  
| }
```

```
  }  
  }  
  
  @Data  
  @Service  
  public class CircularRefB {  
  
      @Value("CircularRefB")  
      private String username;  
  
      @Autowired  
      private CircularRefA circularRefA;  
  }
```

循环依赖步骤：

- 1、A类无参构造函数实例化后，设置三级缓存
- 2、A类 populateBean 进行依赖注入，这里触发了 B 类属性的 getBean 操作
- 3、B类无参构造函数实例化后，设置三级缓存
- 4、B类 populateBean 进行依赖注入，这里触发了 A类属性的 getBean 操作
- 5、A类之前正在实例化，singletonsCurrentlyInCreation 集合中有已经有这个 A类了，三级缓存里面也有了，所以这时候是从三级缓存中拿到的提前暴露的 A 实例，该实例还没有进行 B类属性的依赖注入的，B类属性为空。
- 6、B类拿到了 A 的提前暴露实例注入到 A类属性中了
- 7、B类实例化已经完成，B类的实例化是由 A类实例化中 B 属性的依赖注入触发的 getBean 操作进行的，现在 B 已经实例化，所以 A类中 B 属性就可以完成依赖注入了，这时候 A类 B 属性已经有值了
- 8、B类 A 属性指向的就是 A类实例堆空间，所以这时候 B类 A 属性也会有值了。

BeanPostProcessor 接口理解

BeanPostProcessor 接口类型实例是针对某种特定功能的埋点，在这个点会根据接口类型来过滤掉不关注这个点的其他类，只有真正关注的类才会在这个点进行相应的功能实现。

- 1、获取有@.Autowired 注解的构造函数埋点

过滤的接口类型是：SmartInstantiationAwareBeanPostProcessor

调用的方法是：determineCandidateConstructors

```

// Candidate constructors for autowiring?
// 寻找当前正在实例化的bean中有@Autowired注解的构造函数
Constructor<?>[] ctors = determineConstructorsFromBeanPostProcessors(beanClass, beanName);
if (ctors != null && mbd.getResolvedAutowireMode() == AUTOWIRE_CONSTRUCTOR &&
    mbd.hasConstructorArgumentValues() && !ObjectUtils.isEmpty(args)) {
    // 如果ctors不为空，就说明构造函数上有@Autowired注解
    return autowireConstructor(beanName, mbd, ctors, args);
}

```

2、收集@Resource@.Autowired@Value@PostConstruct, @PreDestroy 注解的方法和属性埋点

过滤的接口类型是：MergedBeanDefinitionPostProcessor

调用的方法是：postProcessMergedBeanDefinition

```

// Allow post-processors to modify the merged bean definition.
synchronized (mbd.postProcessingLock) {
    if (!mbd.postProcessed) {
        try {
            //CommonAnnotationBeanPostProcessor 支持了@PostConstruct, @PreDestroy, @Resource注解
            //AutowiredAnnotationBeanPostProcessor 支持 @Autowired, @Value注解
            //BeanPostProcessor接口的典型运用，这里要理解这个接口
            //对类中注解的装配过程
            //重要程度5，必须看
            applyMergedBeanDefinitionPostProcessors(mbd, beanType, beanName);
        }
        catch (Throwable ex) {
            throw new BeanCreationException(mbd.getResourceDescription(), beanName,
                "Post-processing of merged bean definition failed", ex);
        }
        mbd.postProcessed = true;
    }
}

```

3、循环依赖解决中 bean 的提前暴露埋点

过滤的接口类型是：SmartInstantiationAwareBeanPostProcessor

调用的方法是：getEarlyBeanReference

```

//是否 单例bean提前暴露
boolean earlySingletonExposure = (mbd.isSingleton() && this.allowCircularReferences &&
    isSingletonCurrentlyInCreation(beanName));
if (earlySingletonExposure) {
    if (logger.isTraceEnabled()) {
        logger.trace(o: "Eagerly caching bean '" + beanName +
            "' to allow for resolving potential circular references");
    }
    //这里着重理解，对理解循环依赖帮助非常大，重要程度 5 添加三级缓存
    addSingletonFactory(beanName, () -> getEarlyBeanReference(beanName, mbd, bean));
}

```

4、阻止依赖注入埋点

过滤的接口类型是：InstantiationAwareBeanPostProcessor

调用的方法是：postProcessAfterInstantiation

```

//这里很有意思，写接口可以让所有类都不能依赖注入
if (!mbd.isSynthetic() && hasInstantiationAwareBeanPostProcessors()) {
    for (BeanPostProcessor bp : getBeanPostProcessors()) {
        if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
            if (!ibp.postProcessAfterInstantiation(bw.getWrappedInstance(), beanName)) {
                //是否需要DI，依赖注入
                continueWithPropertyPopulation = false;
                break;
            }
        }
    }
}

```

5、IOC/DI 依赖注入埋点

过滤的接口类型是：InstantiationAwareBeanPostProcessor

调用的方法是：postProcessProperties

```
//重点看这个if代码块，重要程度 5
if (hasInstAwareBpps) {
    if (pvs == null) {
        pvs = mbd.getPropertyValues();
    }
    for (BeanPostProcessor bp : getBeanPostProcessors()) {
        if (bp instanceof InstantiationAwareBeanPostProcessor) {
            InstantiationAwareBeanPostProcessor ibp = (InstantiationAwareBeanPostProcessor) bp;
            //依赖注入过程, @Autowired的支持
            PropertyValues pvsToUse = ibp.postProcessProperties(pvs, bw.getWrappedInstance(), beanName);
            if (pvsToUse == null) {
                if (filteredPds == null) {
                    filteredPds = filterPropertyDescriptorsForDependencyCheck(bw, mbd.allowCaching);
                }
                //老版本用这个完成依赖注入过程, @Autowired的支持
                pvsToUse = ibp.postProcessPropertyValues(pvs, filteredPds, bw.getWrappedInstance(), beanName);
                if (pvsToUse == null) {
                    return;
                }
            }
            pvs = pvsToUse;
        }
    }
}
```

Bean 的多例作用域

Scope 如果是 Prototype 时，不管是不是同一个线程，只要是 getBean 就会得到一个新的实例，这点要注意

Request 作用域时，是把实例存储到 request 对象中

Session 作用域时，是把实例存储到 session 对象中，request 和 session 作用域只会在 web 环境才会存在

自定义作用域

第一要获取 BeanFactory 对象，必须实现 BeanFactoryPostProcessor 接口才能获取 BeanFactory 对象。

第二调用 registerScope 方法把自定义的 scope 注册进去

第三写一个类实现 scope 接口

```

public class CustomScope implements Scope {

    private ThreadLocal local = new ThreadLocal();

    /*
     * 这个方法就是自己管理bean
     */
    @Override
    public Object get(String name, ObjectFactory<?> objectFactory) {
        System.out.println("=====CustomScope=====");
        if(local.get() != null) {
            return local.get();
        } else {
            //这个方法就是掉createbean方法获得一个实例
            Object object = objectFactory.getObject();
            local.set(object);
            return object;
        }
    }

    @Override
    public Object remove(String name) { return null; }
}

```



Bean 的销毁

在 bean 创建完成后就会对这个 bean 注册一个销毁的 Adapter 对象，代码如下：

```

// Register bean as disposable.
try {
    //注册bean销毁时的类DisposableBeanAdapter
    registerDisposableBeanIfNecessary(beanName, bean, mbd);
}

// DisposableBean interface, custom destroy method.
registerDisposableBean(beanName,
    new DisposableBeanAdapter(bean, beanName, mbd, getBeanPostProcessors(), acc));

```

这个 DisposableBeanAdapter 对象就是负责 bean 销毁的类。在这个类中收集了该 bean 是否实现了 DisposableBean 接口，是否配置 destroy-method 属性，过滤了 DestructionAwareBeanPostProcessor 类型的接口。

然后 bean 是在什么时候被销毁呢，在 tomcat 关闭的时候就会调用到 servlet 中的销毁方法，

```

@Override
public void contextDestroyed(ServletContextEvent event) {
    closeWebApplicationContext(event.getServletContext());
    ContextCleanupListener.cleanupAttributes(event.getServletContext());
}

```

在这个方法中就会最终掉用到 DisposableBeanAdapter 类的，destroy() 方法，该方法就会根据前面的收集进行调用。

ConfigurationClassPostProcessor 类

这个类作用很大，支持了

@Configuration @ComponentScan @Import @ImportResource @PropertySource @Order 等注解，对大家理解 springboot 帮助很大，真正的可以做到 0 xml 配置。

这个类可以认真读一下，但是其中的功能大部分我们都已经在前面的课程讲过，比如 @ComponentScan 注解的功能，其实前面我们在讲解自定义标签解析的时候就已经讲过，只是这里是以注解的方式出现的，这里就是殊途同归了，@Import @ImportSource 这些都讲过。

3、AOP 面向切面编程

AOP 入口

通过扫描注解 @EnableAspectJAutoProxy (proxyTargetClass = true, exposeProxy = true) 注册了 AOP 入口类，具体看看注解里面的 @Import (AspectJAutoProxyRegistrar.class)

```
//注册注解AOP入口类
AopConfigUtils.registerAspectJAnnotationAutoProxyCreatorIfNecessary(registry);

/*
 * * true
 * 1、目标对象实现了接口 - 使用CGLIB代理机制
 * 2、目标对象没有接口(只有实现类) - 使用CGLIB代理机制
 *
 * false
 * 1、目标对象实现了接口 - 使用JDK动态代理机制(代理所有实现了的接口)
 * 2、目标对象没有接口(只有实现类) - 使用CGLIB代理机制
 */
AnnotationAttributes enableAspectJAutoProxy =
    AnnotationConfigUtils.attributesFor(importingClassMetadata, EnableAspectJAutoProxy.class);
if (enableAspectJAutoProxy != null) {
    if (enableAspectJAutoProxy.getBoolean(attributeName: "proxyTargetClass")){
        AopConfigUtils.forceAutoProxyCreatorToUseClassProxying(registry);
    }

    //是否需要把代理对象暴露出来，简单来说是否需要把代理对象用ThreadLocal存起来，如果是true就是需要
    if (enableAspectJAutoProxy.getBoolean(attributeName: "exposeProxy")){
        AopConfigUtils.forceAutoProxyCreatorToExposeProxy(registry);
    }
}
```

在这个类中，注册了 AOP 入口类 AnnotationAwareAspectJAutoProxyCreator
设置了两个属性：

proxyTargetClass

true

1、目标对象实现了接口 - 使用 CGLIB 代理机制

2、目标对象没有接口(只有实现类) - 使用 CGLIB 代理机制

false

1、目标对象实现了接口 - 使用 JDK 动态代理机制(代理所有实现了的接口)

2、目标对象没有接口(只有实现类) - 使用 CGLIB 代理机制

```
exposeProxy
```

该属性设置代理对象是否需要暴露，说白了就是是否需要把代理对象设置到 ThreadLocal 中。

AOP 的其他入口类的配置是基于 xml 的形式

比如：

```
<aop:aspectj-autoproxy proxy-target-class="false"  
expose-proxy="true"/>
```

比如：

```
<aop:config proxy-target-class="false">  
    <!--<aop:pointcut>在此处定义的 pointcut 是全局的 pointcut 可以供所有的  
    aspect 使用-->  
    <!--id:表示这个 pointcut 的名称，以方便使用-->  
    <aop:pointcut id="addpointcut"  
        expression="execution(public *  
com.xiangxue.jack.service..*.add(..))"/>  
    <aop:pointcut id="delpointcut"  
        expression="execution(public *  
com.xiangxue.jack.service..*.del*(..))"/>  
    <aop:pointcut id="myMethods"  
        expression="execution(public *  
com.xiangxue.jack.service..*.*(..))"/>  
    <!--advisor 必须在 aspect 之前，要不然有 xml 约束报错-->  
    <aop:advisor advice-ref="beforeAdviceBean" order="2"  
    pointcut-ref="myMethods"/>  
    <aop:aspect id="aspect1" ref="aspectXml1" order="0">  
        <!--<aop:declare-parents  
        types-matching="com.zhuguang.jack.service.MyServiceImpl"-->  
        <!--implement-interface="com.zhuguang.jack.aop.IntroductionIntf"-->  
        <!--delegate-ref="myintroduction"/>-->  
        <!--id:表示这个 pointcut 的名称，以方便使用-->  
        <aop:pointcut id="myMethod2"  
            expression="execution(public *  
com.xiangxue.jack.service..*.*(..)) and  
@annotation(org.springframework.web.bind.annotation.RequestMapping)"/  
>  
        <aop:before method="before" pointcut-ref="myMethods"/>  
        <aop:after method="after" pointcut-ref="myMethod2"/>  
        <!-- 后置通知 returning="returnVal" 定义返回值 必须与类中声明的名称一  
样-->  
        <aop:after-returning method="afterReturning"  
        returning="returnVal"  
            pointcut="execution(public *  
com.xiangxue.jack.service..*.*(..))"/>
```

```

<!--异常通知 throwing="throwable" 指定异常通知错误信息变量，必须与类中
声明的名称一样-->
<!--<aop:after-throwing method="afterthrowing" throwing="e"
pointcut-ref="myMethods"/>-->
<aop:around method="around" pointcut-ref="myMethod2"/>
</aop:aspect>
</aop:config>

```

都是自定义标签解析，解析过程我就不赘述，请参照 `component-scan` 标签解析过程。最终也是完成 AOP 入口类的注册。

是否生成代理



当一个 bean 实例化完成后，会判断该 bean 是否生成代理，AOP 的入口，如图：

```
//bean 实例化+ioc 依赖注入完以后的调用，非常重要，重要程度：5
exposedObject = initializeBean(beanName, exposedObject, mbd);
```

在这个方法里面，具体是

```
if (mbd == null || !mbd.isSynthetic()) {
    //这个地方可能生出代理实例，是aop的入口
    wrappedBean = applyBeanPostProcessorsAfterInitialization(wrappedBean, 1
}
```

这是一个 BeanPostProcessor 接口的运用，`initializeBean` 方法我们都知道是一个 bean 实例化完成后做的操作，而这个代理实例生成也是在 bean 实例化完成后做的操作，其核心代码如下：

```
//创建当前bean的代理，如果这个bean有advice的话，重点看，重要程度5
// Create proxy if we have advice.
Object[] specificInterceptors = getAdvisesAndAdvisorsForBean(bean.getClass(), beanName, customTargetSource: null);
//如果有切面，则生成该bean的代理
if (specificInterceptors != DO_NOT_PROXY) {
    this.advisedBeans.put(cacheKey, Boolean.TRUE);
    //把被代理对象bean实例封装到SingletonTargetSource对象中
    Object proxy = createProxy(
        bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(bean));
    this.proxyTypes.put(cacheKey, proxy.getClass());
    return proxy;
}
```

这个方法就是判断当前 bean 是否有切面 advisor，如果有切面就会走到 `createProxy` 方法，生成代理对象然后返回。

其实寻找当前 bean 的切面简单来说就两步：

```

protected List<Advisor> findEligibleAdvisors(Class<?> beanClass, String beanName) {
    //找到候选的切面,其实就是一个寻找有@Aspectj注解的过程,把工程中所有有这个注解的类封装成Advisor返回
    List<Advisor> candidateAdvisors = findCandidateAdvisors();
    //判断候选的切面是否作用在当前beanClass上面,就是一个匹配过程。现在就是一个匹配
    List<Advisor> eligibleAdvisors = findAdvisorsThatCanApply(candidateAdvisors, beanClass, beanName);
    extendAdvisors(eligibleAdvisors);
    if (!eligibleAdvisors.isEmpty()) {
        //对有@Order@Priority进行排序
        eligibleAdvisors = sortAdvisors(eligibleAdvisors);
    }
    return eligibleAdvisors;
}

```

- 1、从 spring 中找所有的切面
- 2、找到拦截当前 bean 的切面

从 spring 中找所有切面，

先找到所有的 beanDefinition 对象对应的 beanName，

拿到对应的 Class 对象，判断该类上面是否有 @Aspect 注解，如果有则是我们要找的，

循环该 Class 里面的除了 @PointCut 注解的方法，找到方法上面的 Around.class, Before.class, After.class, AfterReturning.class, AfterThrowing.class 注解，并且把注解里面的信息，比如表达式，argNames，注解类型等信息封装成对象 AspectJAnnotation，然后创建 pointCut 对象，把注解对象中的表达式设置到 pointCut 对象中，然后就是创建 Advice 对象，根据不同的注解类型创建出不同的 Advice 对象，对象如下： AspectJAroundAdvice, AspectJAfterAdvice, AspectJAfterThrowingAdvice, AspectJMethodBeforeAdvice, AspectJAfterReturningAdvice 最终把注解对应的 Advice 对象和 pointCut 对象封装成 Advisor 对象。

找到拦截当前 bean 的切面

从收集到的所有切面中，每一个切面都会有 pointCut 来进行模块匹配，其实这个过程就是一个匹配过程，看看 pointCut 表达式中的内容是否包含了当前 bean，如果包含了，那么这个 bean 就有切面，就会生成代理。

代理类的调用

当判断 bean 有切面时也就是

```

//创建当前bean的代理,如果这个bean有advice的话,重点看,重要程度5
// Create proxy if we have advice.
Object[] specificInterceptors = getAdvisesAndAdvisorsForBean(bean.getClass(), beanName,
    //如果有切面,则生成该bean的代理
    if (specificInterceptors != DO_NOT_PROXY) {
        this.advisedBeans.put(cacheKey, Boolean.TRUE);
        //把被代理对象bean实例封装到SingletonTargetSource对象中
        Object proxy = createProxy(
            bean.getClass(), beanName, specificInterceptors, new SingletonTargetSource(
                this.proxyTypes.put(cacheKey, proxy.getClass())));
        return proxy;
    }
}

```

这个数组不为空时，就会往下走，走到 createProxy 方法里面，这个方法就会生成 bean 的代理实例。

创建代理的过程：

- 1、创建代理工厂对象 ProxyFactory
- 2、切面对象重新包装，会把自定义的 MethodInterceptor 类型的类包装成 Advisor 切面类并加入到代理工厂中
- 3、根据 proxyTargetClass 参数和是否实现接口来判断是采用 jdk 代理还是 cglib 代理
- 4、创建代理对象，并且把代理工厂对象传递到 jdk 和 cglib 中，注意这里的代理对象和 jdk 类和 cglib 类是一一对应的。

代理实例的调用

上面我们已经创建出来了代理对象了，现在是拿到代理对象调用，以 jdk 动态代理为例，cglib 是一样的调用逻辑。

当发生代理对象调用时，肯定会调用到实现了 invocationHandler 接口的类，这个类就是：JdkDynamicAopProxy

必定会调用到该类的 invoke 方法。ok 我们看看 invoke 方法

```
if (this.advised.exposeProxy) {  
    // Make invocation available if necessary.  
    oldProxy = AopContext.setCurrentProxy(proxy);  
    setProxyContext = true;  
}
```

该参数如果为 true 会把代理对象放到 ThreadLocal 中。我们在代理里面通过 AopContext.currentProxy 方法就可以拿到当前调用的代理类。实际上没啥用。

```
// Get the interception chain for this method.  
// 从代理工厂中拿过滤器链 Object 是一个 MethodInterceptor 类型的对象，其实就是一个 advice 对象  
List<Object> chain = this.advised.getInterceptorsAndDynamicInterceptionAdvice(method, targetClass);  
// Check whether we have any advice to run and return an advisor
```

从代理工厂中拿到切面，并且跟当前被代理类和当前被调用方法匹配，如果匹配就返回切面中的 advice 对象，这就是 advice 执行链。

只是对 advice 进行了统一包装，如下代码：

```
// 获取到切面 advisor 中的 advice，并且包装成 MethodInterceptor 类型的对象  
MethodInterceptor[] interceptors = registry.getInterceptors(advisor);  
  
// 处理 AspectJMethodBeforeAdvice AspectJAAfterReturningAdvice  
for (AdvisorAdapter adapter : this.adapters) {  
    if (adapter.supportsAdvice(advice)) {  
        interceptors.add(adapter.getInterceptor(advisor));  
    }  
}  
if (interceptors.isEmpty()) {
```

最终会把箭头指向的这两种类型的 advice 包装成 MethodInterceptor 类型的 advice，方便后续统一调用。

获取到这个执行链后，判断如果执行链为空，则直接反射调用方法，说明该方法没被拦截，如果不为空就会有一个链式调用过程。

```
// We need to create a method invocation...
invocation = new ReflectiveMethodInvocation(proxy, target, method, args, targetClass, chain);
// Proceed to the joinpoint through the interceptor chain.
retval = invocation.proceed();
```

在 ReflectiveMethodInvocation 中维护了执行链数组和数组索引

```
Object interceptorOrInterceptionAdvice =
    this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
    // Evaluate dynamic method matcher here: static part will already have
    // been evaluated and found to match.
    InterceptorAndDynamicMethodMatcher dm =
        (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
    Class<?> targetClass = (this.targetClass != null ? this.targetClass : this.method.getDeclaringClass());
    if (dm.methodMatcher.matches(this.method, targetClass, this.arguments)) {
        return dm.interceptor.invoke(invocation: this);
    }
} else {
    // Dynamic matching failed.
    // Skip this interceptor and invoke the next in the chain.
    return proceed();
}
else {
    // It's an interceptor, so we just invoke it: The pointcut will have
    // been evaluated statically before this object was constructed.

    // 调用MethodInterceptor中的invoke方法
    return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(invocation: this);
}
```

通过索引获取到数组里面的 advice，然后调用，我们举例说明，如果第一个 advice 调用到了 beforeAdvice，如下：

```
/serial/
public class MethodBeforeAdviceInterceptor implements MethodInterceptor, BeforeAdvice, Serializable {
    private final MethodBeforeAdvice advice;

    /**
     * Create a new MethodBeforeAdviceInterceptor for the given advice.
     * @param advice the MethodBeforeAdvice to wrap
     */
    public MethodBeforeAdviceInterceptor(MethodBeforeAdvice advice) {
        Assert.notNull(advice, message: "Advice must not be null");
        this.advice = advice;
    }

    @Override
    public Object invoke(MethodInvocation mi) throws Throwable {
        this.advice.before(mi.getMethod(), mi.getArguments(), mi.getThis());
        return mi.proceed();
    }
}
```

BeforeAdvice 首先会调用自己 advice 对象中的 method 对象，而这个 method 对象就是有@Before 注解的方法，我们通过解析方法上面的注解类型生成 advice 对象时把对应的 method 对象封装到了 advice 对象中，所以这里就只要反射调用 method 对象即可。

掉完后，这里有一个 mi.proceed() 这个方法又会回到 ReflectiveMethodInvocation 这个类中

```

Object interceptorOrInterceptionAdvice =
    this.interceptorsAndDynamicMethodMatchers.get(++this.currentInterceptorIndex);
if (interceptorOrInterceptionAdvice instanceof InterceptorAndDynamicMethodMatcher) {
    // Evaluate dynamic method matcher here: static part will already have
    // been evaluated and found to match.
    InterceptorAndDynamicMethodMatcher dm =
        (InterceptorAndDynamicMethodMatcher) interceptorOrInterceptionAdvice;
    Class<?> targetClass = (this.targetClass != null ? this.targetClass : this.method.getDeclaringClass());
    if (dm.methodMatcher.matches(this.method, targetClass, this.arguments)) {
        return dm.interceptor.invoke(invocation: this);
    }
} else {
    // Dynamic matching failed.
    // Skip this interceptor and invoke the next in the chain.
    return proceed();
}
else {
    // It's an interceptor, so we just invoke it: The pointcut will have
    // been evaluated statically before this object was constructed.

    // 调用MethodInterceptor中的invoke方法
    return ((MethodInterceptor) interceptorOrInterceptionAdvice).invoke(invocation: this);
}
}

```

这时候索引又会+1，然后从链中后去下一个 advice，然后下一个 advice 调用完后再调用 mi.proceed() 又会接着调用，就这样形成了一个链式调用过程。直到数组链中全部调用完后会调用到具体的 joinPoint 方法，如下：

```

public Object proceed() throws Throwable {
    // We start with an index of -1 and increment early.
    // 如果执行链中的advice全部执行完，则直接调用joinPoint方法，就是被代理方法
    if (this.currentInterceptorIndex == this.interceptorsAndDynamicMethodMatchers.size() - 1) {
        return invokeJoinpoint();
    }
}

```

当判断当前索引==执行链数组大小时就会调用到被代理方法。完成 aop 增强过程

Spring 事务

Spring 中事务也是用 AOP 切面技术来实现的。

首先用注解的方式引入事务管理功能：代码如下

```

@Component
@EnableTransactionManagement(proxyTargetClass = false)
@MapperScan(basePackages = {"com.xiangxue.jack.dao"}, annotationClass = Repository.class)
public class EnableTransactionManagementBean {

    @Bean
    public SqlSessionFactoryBean sqlSessionFactoryBean(DataSource dataSource) {
        SqlSessionFactoryBean sqlSessionFactoryBean = new SqlSessionFactoryBean();
        sqlSessionFactoryBean.setDataSource(dataSource);
        return sqlSessionFactoryBean;
    }

    /**
     * 这样也可以
     */
    @Bean
    public PlatformTransactionManager annotationDrivenTransactionManager(DataSource dataSource) {
        DataSourceTransactionManager dtm = new DataSourceTransactionManager();
        dtm.setDataSource(dataSource);
        return dtm;
    }
}

```

引用这个注解就显示的添加了注解事务功能，但是我们自己还是需要定义数据源和事务管理平台的：

数据源定义：

```
@Bean
public DataSource comboPooledDataSource() {
    ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource();
    try {
        comboPooledDataSource.setDriverClass(environment.getProperty("jdbc.driverClassName"));
        comboPooledDataSource.setJdbcUrl(environment.getProperty("jdbc.url"));
        comboPooledDataSource.setUser(environment.getProperty("jdbc.username"));
        comboPooledDataSource.setPassword(environment.getProperty("jdbc.password"));
        comboPooledDataSource.setMinPoolSize(10);
        comboPooledDataSource.setMaxPoolSize(100);
        comboPooledDataSource.setMaxIdleTime(1800);
        comboPooledDataSource.setAcquireIncrement(3);
        comboPooledDataSource.setMaxStatements(1000);
        comboPooledDataSource.setInitialPoolSize(10);
        comboPooledDataSource.setIdleConnectionTestPeriod(60);
        comboPooledDataSource.setAcquireRetryAttempts(30);
        comboPooledDataSource.setBreakAfterAcquireFailure(false);
        comboPooledDataSource.setTestConnectionOnCheckout(false);
        comboPooledDataSource.setAcquireRetryDelay(100);
    } catch (PropertyVetoException e) {
        e.printStackTrace();
    }
    return comboPooledDataSource;
}
```

定义事务管理平台：

```
@Bean
public PlatformTransactionManager annotationDrivenTransactionManager(DataSource dataSource) {
    DataSourceTransactionManager dtm = new DataSourceTransactionManager();
    dtm.setDataSource(dataSource);
    return dtm;
}
```

数据源和事务管理平台的加载都是在类
ProxyTransactionManagementConfiguration 进行的。

在理解 spring 事务的时候必须要牢牢记住几个概念：

Connection 连接、事务、和用户会话

我们在看了 jdbc 代码以后可以清楚的知道，事务就是由 connection 对象控制的，所以 connection 对象是和事务是绑定的，然后用户请求过来时又需要数据库连接来执行 sql 语句，所以用户请求线程又是跟 connection 是绑定的。这点一定要弄清楚。

在 spring 中事务传播属性有如下几种：

PROPAGATION_REQUIRED

如果当前没有事务，就新建一个事务，如果已经存在一个事务中，加入到这个事务中。这是最常见的选择。

PROPAGATION_SUPPORTS

支持当前事务，如果当前没有事务，就以非事务方式执行。

PROPAGATION_MANDATORY

使用当前的事务，如果当前没有事务，就抛出异常。

PROPAGATIONQUIRES_NEW

新建事务，如果当前存在事务，把当前事务挂起。

PROPAGATION_NOT_SUPPORTED

以非事务方式执行操作，如果当前存在事务，就把当前事务挂起。

PROPAGATION_NEVER

以非事务方式执行，如果当前存在事务，则抛出异常。

PROPAGATION_NESTED

如果当前存在事务，则在嵌套事务内执行。如果当前没有事务，则执行与 PROPAGATION_REQUIRED 类似操作。

其中最常见的，用得最多就 PROPAGATION_REQUIRED、PROPAGATIONQUIRES_NEW、PROPAGATION_NESTED 这三种。事务的传播属性是 spring 特有的，是 spring 用来控制方法事务的一种手段，说直白点就是用来控制方法是否使用同一事务的一种属性，以及按照什么规则回滚的一种手段。这三种我们会配合源码一一的理清楚。

传播属性的事务控制

这个类是事务切面的 Advice，所有有关 spring 的事务控制逻辑都在这个类里面。我们重点看看这个类：

1、获取事务属性对象

```
// If the transaction attribute is null, the method is non-transactional.  
// 获取事务属性类 AnnotationTransactionAttributeSource  
TransactionAttributeSource tas = getTransactionAttributeSource();
```

2、tas.getTransactionAttribute

这个方法就会扫描 method 上面的@Transactional 注解，把注解里面配置的属性封装到对象中，对应的对象是：RuleBasedTransactionAttribute，如图：

```
protected TransactionAttribute parseTransactionAnnotation(AnnotationAttributes attributes) {  
    RuleBasedTransactionAttribute rbta = new RuleBasedTransactionAttribute();  
    Propagation propagation = attributes.getEnum(attributeName: "propagation");  
    rbta.setPropagationBehavior(propagation.value());  
    Isolation isolation = attributes.getEnum(attributeName: "isolation");  
    rbta.setIsolationLevel(isolation.value());  
    rbta.setTimeout(attributes.getNumber(attributeName: "timeout").intValue());  
    rbta.setReadOnly(attributes.getBoolean(attributeName: "readOnly"));  
    rbta.setQualifier(attributes.getString(attributeName: "value"));  
  
    List<RollbackRuleAttribute> rollbackRules = new ArrayList<>();  
    for (Class<?> rbRule : attributes.getClassArray(attributeName: "rollbackFor")) {  
        rollbackRules.add(new RollbackRuleAttribute(rbRule));  
    }  
    for (String rbRule : attributes.getStringArray(attributeName: "rollbackForClassName")) {  
        rollbackRules.add(new RollbackRuleAttribute(rbRule));  
    }  
    for (Class<?> rbRule : attributes.getClassArray(attributeName: "noRollbackFor")) {  
        rollbackRules.add(new NoRollbackRuleAttribute(rbRule));  
    }  
    for (String rbRule : attributes.getStringArray(attributeName: "noRollbackForClassName")) {  
        rollbackRules.add(new NoRollbackRuleAttribute(rbRule));  
    }  
    rbta.setRollbackRules(rollbackRules);
```

3、开启事务

```
// Standard transaction demarcation with getTransaction and commit/rollback calls.  
TransactionInfo txInfo = createTransactionIfNecessary(tm, txAttr, joinpointIdentification);
```

点进去走到这个方法：

```
//开启事务，这里重点看  
status = tm.getTransaction(txAttr);
```

接着往下走：

```
//这里重点看，.DataSourceTransactionObject拿到对象  
Object transaction = doGetTransaction();
```

这个方法创建事务对象，事务对象往往需要跟连接挂钩，所以里面肯定会有连接对象 ConnectionHolder，在这个方法里面会首先从 ThreadLocal 中获取连接对象，如下：

```
//obtainDataSource() 获取数据源对象，其实就是数据库连接块对象  
ConnectionHolder conHolder =  
    (ConnectionHolder) TransactionSynchronizationManager.getResource(obtainDataSource());
```

这个 ThreadLocal 中 value 封装了一个 map，map 是数据源对象 DataSource 和连接对象的映射关系，也就是说，如果上一个事务中建立了映射关系，下一个事务就可以通过当前线程从 ThreadLocal 中获取到这个 map，然后根据当前使用的数据源对象拿到对应的连接对象。然后设置到事务对象中。

```
//这里重点看，.DataSourceTransactionObject拿到对象  
Object transaction = doGetTransaction();
```

该方法执行完后，往下走：

```
//第一次进来connectionHolder为空的，所以不存在事务  
if (isExistingTransaction(transaction)) {  
    // Existing transaction found -> check propagation behavior to find out how to behave.  
    return handleExistingTransaction(definition, transaction, debugEnabled);  
}
```

该方法判断当前是否存在事务，如果能从事务对象中拿到连接对象，就表示当前是存在事务的。如果不存在，也就是说是第一次创建事务。则往下走：

```
//第一次进来大部分会走这里  
else if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRED ||  
        definition.getPropagationBehavior() == TransactionDefinition.PROPAGATIONQUIRES_NEW ||  
        definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {  
    //先挂起  
    SuspendedResourcesHolder suspendedResources = suspend(transaction: null);  
    if (debugEnabled) {  
        logger.debug(o: "Creating new transaction with name [" + definition.getName() + "]: " + d  
    }  
    try {  
        boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);  
        //创建事务状态对象，其实就是封装了事务对象的一些信息，记录事务状态的  
        DefaultTransactionStatus status = newTransactionStatus(  
            definition, transaction, newTransaction: true, newSynchronization, debugEnabled, su  
        //开启事务，重点看看 DataSourceTransactionObject  
        doBegin(transaction, definition);  
        //开启事务后，改变事务状态  
        prepareSynchronization(status, definition);  
        return status;  
    }
```

该代码：

第一，创建事务状态对象，事务状态就是记录事务流转过程中状态数据的，有一个数据非常重要，直接决定了提交，回滚和恢复绑定操作，就是 newTransaction 属性，这个属性要牢记。

如果为 true 就代表当前事务允许单独提交和回滚，一般是第一次创建事务或者事务传播属性为 PROPAGATION_REQUIRE_NEW 的时候。如果为 false 则当前事务不能单独提交和回滚。

第二，doBegin 该方法则是开启事务的核心方法，点进去看看：

1、从数据源连接池中获取连接

```
//如果没有数据库连接
if (!txObject.getConnectionHolder() ||
    txObject.getConnectionHolder().isSynchronizedWithTransaction()) {
    //从连接池里面获取连接
    Connection newCon = obtainDataSource().getConnection();
    if (logger.isDebugEnabled()) {
        logger.debug(o: "Acquired Connection [" + newCon + "] for JDBC transaction");
    }
    //把连接包装成ConnectionHolder, 然后设置到事务对象中
    txObject.setConnectionHolder(new ConnectionHolder(newCon), newConnectionHolder: true);
}
```

这块从数据源获取到连接对象，有一个数据源要提一下，就是动态切换数据源的一个抽象类：AbstractRoutingDataSource，如果有多个数据源的需求，可以自己定义数据源对象然后往自定义对象中设置目标数据源和默认数据源，通过切面控制动态切换逻辑：

动态数据源：

从 ThreadLocal 中获取连接字符串

```
public class DynamicDataSource extends AbstractRoutingDataSource {

    @Override
    protected Object determineCurrentLookupKey() {
        String ds = DynamicDataSourceHolder.getDs();

        System.out.println("=====选择的数据源：" + ds);
        return ds;
    }
}
```

ThreadLocal 类：

用来保存当前线程和数据源字符串的关系

```
public class DynamicDataSourceHolder {

    private static ThreadLocal<String> local = new ThreadLocal<~>();

    public static String getDs() { return local.get(); }

    public static ThreadLocal getLocal() { return local; }
}
```

切面类：

通过获取调用方法上面的注解来获取选择的数据源字符串，然后把这个字符串设置到 ThreadLocal 中跟当前线程绑定：

```

    @Component
    @Aspect
    @Order(-1)
    public class AspectDs {
        @Before(value = "@annotation(targetSource)", argNames = "joinPoint,targetSource")
        public void xx(JoinPoint joinPoint, TargetSource targetSource) {
            System.out.println("=====AspectDs.xx");
            String value = targetSource.value();

            if(value != null && !"".equals(value)) {
                DynamicDataSourceHolder.getLocal().set(value);
            } else {
                DynamicDataSourceHolder.getLocal().set("ds1");
            }
        }
    }

```

动态数据源初始化：

```

@Bean
public DataSource dynamicDataSource() {
    ComboPooledDataSource comboPooledDataSource = new ComboPooledDataSource();
    try {
        comboPooledDataSource.setDriverClass(driverClass);
        comboPooledDataSource.setJdbcUrl(jdbcUrl);
        comboPooledDataSource.setUser(user);
        comboPooledDataSource.setPassword(password);
        comboPooledDataSource.setMinPoolSize(10);
        comboPooledDataSource.setMaxPoolSize(100);
        comboPooledDataSource.setMaxIdleTime(1800);
        comboPooledDataSource.setAcquireIncrement(3);
        comboPooledDataSource.setMaxStatements(1000);
        comboPooledDataSource.setInitialPoolSize(10);
    } catch (PropertyVetoException e) {
        e.printStackTrace();
    }

    Map<Object, Object> targetDataSources = new HashMap<>();
    targetDataSources.put("ds1", comboPooledDataSource);

    DynamicDataSource dynamicDataSource = new DynamicDataSource();
    dynamicDataSource.setTargetDataSources(targetDataSources);
    dynamicDataSource.setDefaultTargetDataSource(comboPooledDataSource);
    return dynamicDataSource;
}

```

这里建立了字符串和数据源对象的映射关系，然后把映射关系保存到了动态数据源对象中，动态数据源对象通过钩子方法 determineCurrentLookupKey 就可以从 ThreadLocal 中获取到字符串，然后根据字符串从映射关系中找到选择的数据源对象。

2、把自动提交关闭

```

if (con.getAutoCommit()) {
    txObject.setMustRestoreAutoCommit(true);
    if (logger.isDebugEnabled()) {
        logger.debug(o: "Switching JDBC Connection [" + con + "] to manual commit");
    }
    //关闭连接的自动提交，其实这步就是开启了事务
    con.setAutoCommit(false);
}

```

3、建立 ThreadLocal 的绑定关系

这个绑定关系就是前面提到的，从绑定关系中可以拿到 map，map 中建立的是数据源对象和连接对象的映射

```

// Bind the connection holder to the thread.
if (txObject.isNewConnectionHolder()) {
    //如果是新创建的事务，则建立当前线程和数据库连接的关系
    TransactionSynchronizationManager.bindResource(obtainDataSource(), txObject.getConnectionHolder());
}

```

到这里，doBegin 基本结束了，事务已经开启了。

这里我们重点看一下这个代码：

```
//第一次进来connectionHolder为空的，所以不存在事务
if (isExistingTransaction(transaction)) {
    // Existing transaction found -> check propagation behavior to find out how to behave.
    return handleExistingTransaction(definition, transaction, debugEnabled);
}
```

这个代码只有存在事务传播的时候才会走进来，就是事务下面又有事务。

```
//开启了事务
@Transactional(propagation = Propagation.REQUIRED)
@Override
public void transaction(ConsultConfigArea area, ZgGoods zgGoods) {
    @Transactional
    areaService.addArea(area);
    @Transactional
    goodsService.addGoods(zgGoods);
}
//担保重头
```

就类似于这种代码，transaction 方法上面存在事务，而下面的方法也存在事务的情况下，那么这个代码块就会进来：

这个代码里面，判断了第一个事务下面的事务特性，根据传播特性来控制事务。

重点看一下这两种：PROPAGATION_REQUIRE_NEW, PROPAGATION_NESTED

PROPAGATION_REQUIRE_NEW

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_REQUIRE_NEW) {
    if (debugEnabled) {
        logger.debug("Suspending current transaction, creating new transaction with name [" +
                     definition.getName() + "]");
    }
    //挂起
    SuspendedResourcesHolder suspendedResources = suspend(transaction);
    try {
        boolean newSynchronization = (getTransactionSynchronization() != SYNCHRONIZATION_NEVER);
        DefaultTransactionStatus status = newTransactionStatus(
            definition, transaction, newTransaction: true, newSynchronization, debugEnabled, suspendedResources);
        doBegin(transaction, definition);
        prepareSynchronization(status, definition);
        return status;
    }
    catch (RuntimeException | Error beginEx) {
        resumeAfterBeginException(transaction, suspendedResources, beginEx);
        throw beginEx;
    }
}
```

这个事务传播属性意思是，如果当前没事务创建事务，如果当前有事务则把当前事务挂起并创建新的事务，反正就是要用自己创建的事务。

第一，挂起事务

```
//挂起
SuspendedResourcesHolder suspendedResources = suspend(transaction);
try {
```

挂起事务其实就是把事务对象中的连接对象设置为 null，并且解除 ThreadLocal 的绑定关系

```
@Override  
protected Object doSuspend(Object transaction) {  
    DataSourceTransactionObject txObject = (DataSourceTransactionObject) transaction;  
    txObject.setConnectionHolder(null);  
    //解除绑定关系,  
    return TransactionSynchronizationManager.unbindResource(obtainDataSource());  
}
```

然后再返回了老的上一个事务的连接对象，这个一定要注意，因为会根据这个返回的连接对象，在该事务提交的时候重新的建立绑定关系的。

第二，doBegin 开启事务，这里不赘述

第三，返回事务状态

PROPAGATION_NESTED

嵌套事务，其实就是按照回滚点来回滚事务。

```
if (definition.getPropagationBehavior() == TransactionDefinition.PROPAGATION_NESTED) {  
    if (!isNestedTransactionAllowed()) {  
        throw new NestedTransactionNotSupportedException(  
            "Transaction manager does not allow nested transactions by default - " +  
            "specify 'nestedTransactionAllowed' property with value 'true'");  
    }  
    if (debugEnabled) {  
        logger.debug(o: "Creating nested transaction with name [" + definition.getName() + "]")  
    }  
    //默认是可以嵌套事务的  
    if (useSavepointForNestedTransaction()) {  
        // Create savepoint within existing Spring-managed transaction,  
        // through the SavepointManager API implemented by TransactionStatus.  
        // Usually uses JDBC 3.0 savepoints. Never activates Spring synchronization.  
        DefaultTransactionStatus status =  
            prepareTransactionStatus(definition, transaction, newTransaction: false, newSyn  
            //创建回滚点  
            status.createAndHoldSavepoint();  
        return status;  
    }  
}
```

在这个代码里面就是用连接对象创建了回滚点，是在方法调用之前创建了回滚点，并且并没有做挂起和开启事务 doBegin 的操作，从这里就能判断当前事务的连接对象和前事务的连接对象是一致的。

前面事务开启已经做完，接下来就是火炬传递了：



```
try {
    // This is an around advice: Invoke the next intercept
    // This will normally result in a target object being i
    //火炬传递
    retVal = invocation.proceedWithInvocation();
}
catch (Throwable ex) {
    // target invocation exception
    //事务回滚
    completeTransactionAfterThrowing(txInfo, ex);
    throw ex;
}
finally {
    cleanupTransactionInfo(txInfo);
}
//事务提交
commitTransactionAfterReturning(txInfo);
return retVal;
```

如果链式调用过程中不存在 advice 了，则会调用到被代理方法，执行相应的业务操作，如果业务操作没问题则提交事务

```
commitTransactionAfterReturning(txInfo);
```

提交的时候要注意一下事务状态，newTransaction

```
if (status.hasSavepoint()) {
    if (status.isDebug()) {
        logger.debug(o: "Releasing transaction savepoint");
    }
    unexpectedRollback = status.isGlobalRollbackOnly();
    status.releaseHeldSavepoint();
}
//如果都是PROPAGATION_REQUIRED, 最外层的才会走进来统一提交, 如果是PROPAGATION_I
else if (status.isNewTransaction()) {
    if (status.isDebug()) {
        logger.debug(o: "Initiating transaction commit");
    }
    unexpectedRollback = status.isGlobalRollbackOnly();
    doCommit(status);
}
```

只有事务状态是 true 时才允许直接提交，为 new 只会发生在第一次事务进来或者传播属性为 new 的情况下。

如果是有回滚点，在提交的时候把回滚点抹掉了，这种情况只可能是第一次事务后面的嵌套事务提交才能发生的情况。

然后在事务提交完成后，还有一个点要注意，就是资源释放和重新建立绑定关系操作：

```
finally {
    cleanupAfterCompletion(status);
}
```

```
private void cleanupAfterCompletion(DefaultTransactionStatus status) {
    status.setCompleted();
    if (status.isNewSynchronization()) {
        TransactionSynchronizationManager.clear();
    }
    if (status.isNewTransaction()) {
        doCleanupAfterCompletion(status.getTransaction());
    }
    //判断当前事务有没有挂起的连接
    if (status.getSuspendedResources() != null) {
        if (status.isDebugEnabled()) {
            logger.debug(o: "Resuming suspended transaction after completion of inner transaction");
        }
        Object transaction = (status.hasTransaction() ? status.getTransaction() : null);
        //恢复挂起连接的绑定
        resume(transaction, (SuspendedResourcesHolder) status.getSuspendedResources());
    }
}
```

如果 newTransaction 为 true 才会释放连接资源，这个很好理解，这代表着所有操作都已经做完了。这块就是会把连接资源归还给连接池。然后解除 ThreadLocal 和连接的绑定关系。

如果被挂起的事务对象不为空：

```
//判断当前事务有没有挂起的连接
if (status.getSuspendedResources() != null) {
    if (status.isDebugEnabled()) {
        logger.debug(o: "Resuming suspended transaction after completion of inner transaction");
    }
    Object transaction = (status.hasTransaction() ? status.getTransaction() : null);
    //恢复挂起连接的绑定
    resume(transaction, (SuspendedResourcesHolder) status.getSuspendedResources());
}
```

则需要恢复绑定关系：

```
@Override
protected void doResume(@Nullable Object transaction, Object suspendedResources) {
    TransactionSynchronizationManager.bindResource(obtainDataSource(), suspendedResources);
}
```

又重新建立了当前线程和连接对象的绑定关系，所以你下个事务再进来的时候就可以从绑定关系中获取到连接对象了。

如果业务操作有异常就会回滚：

```
catch (Throwable ex) {
    // target invocation exception
    //事务回滚
    completeTransactionAfterThrowing(txInfo, ex);
    throw ex;
}
-----
```

```
//按照嵌套事务按照回滚点回滚
if (status.hasSavepoint()) {
    if (status.isDebugEnabled()) {
        logger.debug(o: "Rolling back transaction to savepoint");
    }
    status.rollbackToHeldSavepoint();
}
//都为PROPAGATION_REQUIRED最外层事务统一回滚
else if (status.isNewTransaction()) {
    if (status.isDebugEnabled()) {
        logger.debug(o: "Initiating transaction rollback");
    }
    doRollback(status);
}
else {
```

回滚一样要判断事务状态，如果为 true 才允许回滚，如果有回滚点则按照回滚点来回滚。
回滚一样会伴随着连接释放，逻辑跟提交是一样的，这里不再赘述

```
finally {
    cleanupAfterCompletion(status);
}
```

如果不清楚事务的不同传播属性的流转，建议可以采用伪代码的方式来分析：

```
transation 方法
createTransactionIfNecessary = 开启事务
try {
    addArea 方法
    开启事务
    Try{
        目标类 addArea 调用
        }catch(Throwable ex) {
            事务回滚?
            throw ex;
        }
    commitTransactionAfterReturning
    addArea 方法提交事务?

    addGoods 方法
    开启事务
    Try{
        目标类 addGoods 调用
        Throw new RuntimeException();
```

```
        } catch (Throwable ex) {
            事务回滚?
            throw ex;
        }

        commitTransactionAfterReturning
        addGoods 方法提交事务?

    } catch (Throwable ex) {
        completeTransactionAfterThrowing = 事务回滚
        throw ex;
    }

    transation 方法提交事务
    commitTransactionAfterReturning = 事务提交
```

注解事务和编程式事务

注解事务是方法上面加了@Transactional 的事务，这种事务控制粒度不够细，如果方法流程很长的话会产生连接占用问题，连接占用就会导致整个系统吞吐量下降。

注解事务犹豫隔离级别的不同，可能导致可重复读的问题，比如抢火车票这种，需要用到乐观锁，但是在重复抢票的时候由于是同一个连接对象，所以每次查询的数据是一样的。

```
@Transactional
@Override
public int getTicket() {
    //1、获取锁
    List<ZgTicket> zgTickets = commonMapper.queryTicketById(ticketId: "12306");
    Map lockmap = new HashMap();
    lockmap.put("ticketId", "12306");
    lockmap.put("version", zgTickets.get(0).getVersion());
    int i = commonMapper.updateLock(lockmap);

    if (i > 0) {
        //抢票
        ZgTicket zgTicket = zgTickets.get(0);
        zgTicket.setTicketCount(2);
        int i1 = commonMapper.updateTicket(zgTicket);
    } else {
        //继续抢
        ((TransactionService) AopContext.currentProxy()).getTicket();
    }

    return 0;
}
```

getTicket 方法递归调用的时候，在可重复读的隔离级别下查询的数据是一样的。所以就永远获取不到锁。

改造后代码：

```
@Override
public int getTicketModeOne() {
    Integer execute = transactionTemplate.execute(status -> {
        //1、获取锁
        List<ZgTicket> zgTickets = commonMapper.queryTicketById(ticketId: "12306");
        Map lockmap = new HashMap();
        lockmap.put("ticketId", "12306");
        lockmap.put("version", zgTickets.get(0).getVersion());
        int i = commonMapper.updateLock(lockmap);

        if (i > 0) {
            //抢票
            ZgTicket zgTicket = zgTickets.get(0);
            zgTicket.setTicketCount(2);
            int i1 = commonMapper.updateTicket(zgTicket);
        }
        return i;
    });
    if (execute == 0) {
        //继续抢
        getTicketModeOne();
    }
    return 0;
}
```

用编程式事务解决了这个问题，这个查询语句是在 `execute` 方法内，当这个方法执行完时就提交了事务，下次再进来时又是一个单独的新事务了。

编程式事务控制粒度更细，可以只关心需要事务控制的操作，如果不需要事务控制的代码可以不放在 `execute` 方法内。

甚至可以自己手动控制事务：

```
@Autowired
PlatformTransactionManager platformTransactionManager;

public void xxx() {
    DefaultTransactionDefinition defaultTransactionDefinition = new DefaultTransactionDefinition();
    defaultTransactionDefinition.setPropagationBehavior(0);
    TransactionStatus transaction = platformTransactionManager.getTransaction(defaultTransactionDefinition);

    try {
        System.out.println("业务代码");
    } catch (Exception e) {
        platformTransactionManager.rollback(transaction);
    }

    platformTransactionManager.commit(transaction);
}
```

把控制代码可以写到切面中，跟业务代码解耦。

缓存

开启缓存功能：

```

@Component
@EnableCaching
public class CacheBean {

    @Bean
    public Cache redisCache(RedisTemplate redisTemplate) {
        RedisCache cache = new RedisCache();
        cache.setName("redisCache");
        cache.setRedisTemplate(redisTemplate);
        return cache;
    }

    @Bean
    public FactoryBean<ConcurrentMapCache> mapCache() {
        ConcurrentMapCacheFactoryBean bean = new ConcurrentMapCacheFactoryBean();
        bean.setName("mapCache");
        return bean;
    }

    @Bean
    public CacheManager simpleCacheManager(@Qualifier("redisCache") Cache redisCache, @Qualifier("mapCache")
        SimpleCacheManager simpleCacheManager = new SimpleCacheManager();
        List<Cache> list = new ArrayList<>();
        list.add(redisCache);
        list.add(concurrentMapCacheFactoryBean);
        simpleCacheManager.setCaches(list);
        return simpleCacheManager;
    }
}

```

加上@EnableCaching 注解

缓存里面有的元素: cacheManager、cache

首先需要创建缓存管理器:

```

@Bean
public CacheManager simpleCacheManager(@Qualifier("redisCache") Cache redisCache, @Qualifier("mapCache") Cache
    SimpleCacheManager simpleCacheManager = new SimpleCacheManager();
    List<Cache> list = new ArrayList<>();
    list.add(redisCache);
    list.add(concurrentMapCacheFactoryBean);
    simpleCacheManager.setCaches(list);
    return simpleCacheManager;
}

```

缓存管理器中管理了缓存对象, 比如 redis 缓存, map 缓存, mongodb 缓存, 这些缓存对象都是些了 Cache 顶层接口

缓存使用就只需要在业务方法上面加注解就行了

```

@Cacheable(cacheNames = "redisCache", key = "'jack' + #id")
@Override
public String queryData(String id) {
    System.out.println("=====CacheServiceImpl.queryData");
    List<ConsultConfigArea> areas = commonMapper.queryAreaById(id);
    return JSONObject.toJSONString(areas);
}

@CachePut(cacheNames = "redisCache", key = "'jack' + #id")
@Override
public String putCache(String id) {
    System.out.println("=====CacheServiceImpl.queryData");
    List<ConsultConfigArea> areas = commonMapper.queryAreaById(id);
    return JSONObject.toJSONString(areas);
}

```

@Cacheable 是先从缓存拿, 如果有则直接返回, 如果没有则调用被代理方法拿到返回值然后存到缓存中。

@CachePut 只管存, 调用到被代理方法后把返回值存到缓存中。

在使用注解时需要指定使用哪一个缓存，因为缓存管理器中可能会有多个缓存，redis 缓存，mongodb 缓存，map 缓存等等，需要用一个 name 来建立跟这些缓存对象的映射关系。

异步

开启异步：

```
① @Component
② @EnableAsync
public class EnableAsyncBean {

    private int corePoolSize = 10;
    private int maxPoolSize = 200;
    private int queueCapacity = 10;
    private String ThreadNamePrefix = "JackExecutor-";

    ③ @Bean
    public Executor executor() {
        ThreadPoolExecutor executor = new ThreadPoolExecutor();
        executor.setCorePoolSize(corePoolSize);
        executor.setMaxPoolSize(maxPoolSize);
        executor.setQueueCapacity(queueCapacity);
        executor.setThreadNamePrefix(ThreadNamePrefix);

        // rejection-policy: 当pool已经达到max size的时候，如何处理新任务
        // CALLER_RUNS: 不在新线程中执行任务，而是有调用者所在的线程来执行
        executor.setRejectedExecutionHandler(new ThreadPoolExecutor.CallerRunsPolicy());
        executor.initialize();
        return executor;
    }
}
```

异步没什么好讲的，注解在业务方法上面加上异步注解即可

```
* */
① @Async
② @Transactional
③ @TargetSource("ds1")
④ @Override
public String queryAccount(String id) {
    System.out.println("=====AccountServiceImpl.queryAccount");
    return "=====AccountServiceImpl.queryAccount";
}
```

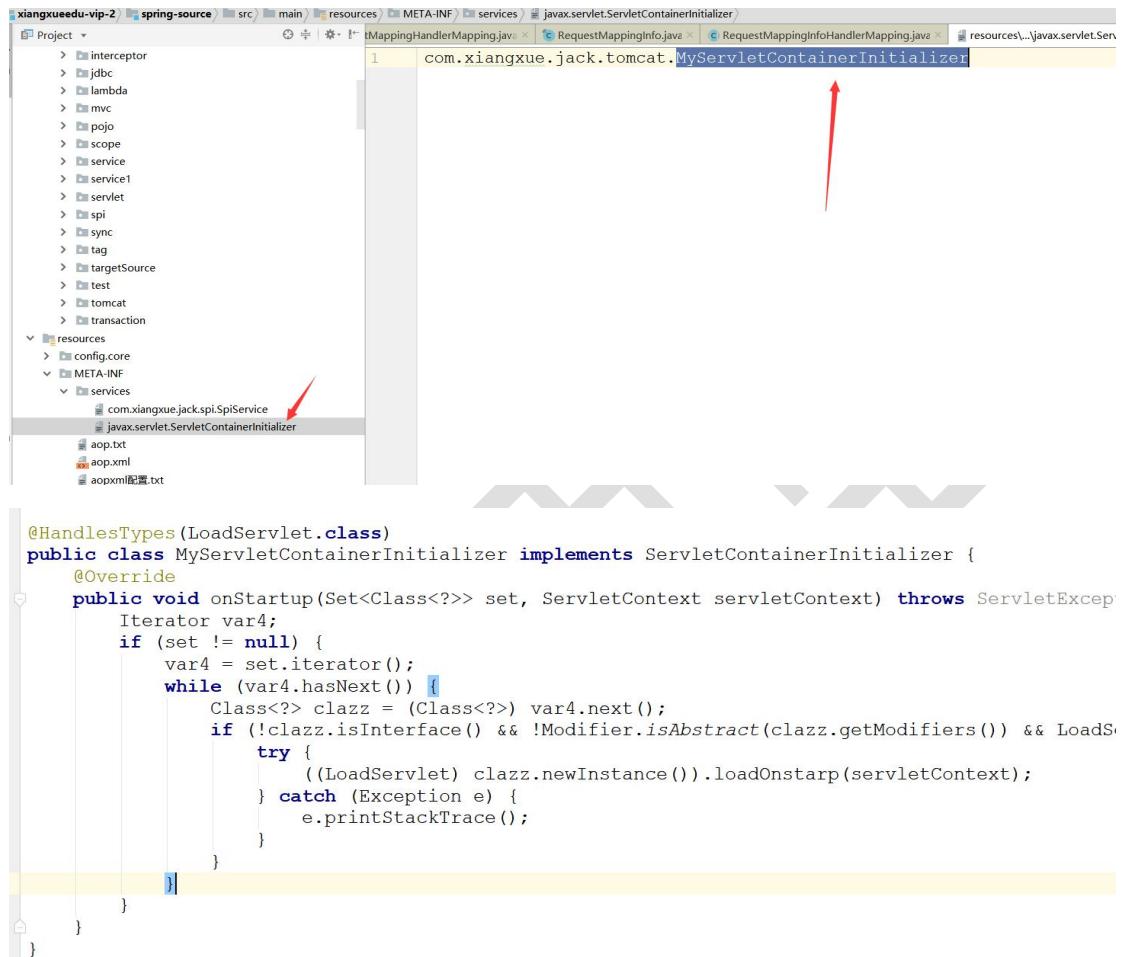
这里需要注意一下，如果采用异步，那么事务传播属性就会实现，从上一个事务中是拿不到绑定的连接对象的，也就是说是一个新的事务了。

4、Springmvc

Springmvc 是基于 servlet 规范来完成的一个请求响应模块，也是 spring 中比较大的一个模块，现在基本上都是零 xml 配置了，采用的是约定大于配置的方式，所以我们的 springmvc 也是采用这种零 xml 配置的方式。要完成这种过程，要解决两个问题 1、取代 web.xml 配置，2、取代 springmvc.xml 配置。

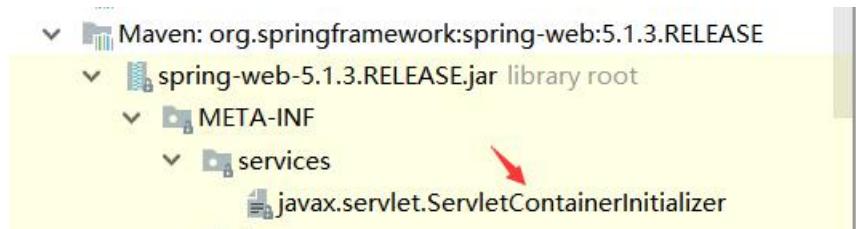
取代 web.xml 配置

在 servlet 中有一个规范，就是当 servlet 容器启动的时候会根据 spi 规范加载 META-INF/services 文件夹下面的 javax.servlet.ServletContainerInitializer 文件，该文件下面的类会实现 javax.servlet.ServletContainerInitializer 接口。如图：



该类在启动的时候会被 servlet 容器实例化，然后调用 onStartup 方法，并且 servlet 容器会收集实现了@HandlesTypes 注解里面的接口的类，并且做为入参传入到 onStartup 方法中，我们拿到 set 容器中的类就可以反射调用接口里面的方法了，这是 servlet 规范，该规范就能保证 servlet 容器在启动的时候就会完成这些操作。Springmvc 就借助这一点完成了取代 web.xml 的工作。

在 springmvc 中，spring-web jar 包下面也会有一个 javax.servlet.ServletContainerInitializer 文件，如图：



```
org.springframework.web.SpringServletContainerInitializer
```

Tomcat 就会加载这个类，调用其 `onStartup` 方法。

```
@HandlesTypes(WebApplicationInitializer.class)
public class SpringServletContainerInitializer implements ServletContainerInitializer {
    /**
     * ...
     */
    @Override
    public void onStartup(@Nullable Set<Class<?>> webAppInitializerClasses, ServletContext servletContext)
        throws ServletException {
        List<WebApplicationInitializer> initializers = new LinkedList<>();
        if (webAppInitializerClasses != null) {
            for (Class<?> waiClass : webAppInitializerClasses) {
                // Be defensive: Some servlet containers provide us with invalid classes,
                // no matter what @HandlesTypes says...
                if (!waiClass.isInterface() && !Modifier.isAbstract(waiClass.getModifiers()) &&
                    WebApplicationInitializer.class.isAssignableFrom(waiClass)) {
                    try {
                        initializers.add((WebApplicationInitializer)
                            ReflectionUtils.accessibleConstructor(waiClass).newInstance());
                    }
                }
            }
        }
    }
}
```

收集的是实现了 `WebApplicationInitializer` 接口的类，在 `springmvc` 工程中我们自己写了这么一个类，如图：

```
public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {
    //父容器
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{SpringContainer.class};
    }

    //SpringMVC配置子容器
    @Override
    protected Class<?>[] getServletConfigClasses() { return new Class<?>[]{MvcContainer.class}; }

    //获取DispatcherServlet的映射信息
    @Override
    protected String[] getServletMappings() { return new String[]{"/"}; }

    @Override
    protected Filter[] getServletFilters() {
        return super.getServletFilters();
    }
}
```

该类的父类最终会实现 `WebApplicationInitializer`，所以该类的父类必定会有一个 `onStartup` 方法。其父类截图如下：

```
/*
public abstract class AbstractDispatcherServletInitializer extends AbstractContextLoaderInitializer {
    /**
     * The default servlet name. Can be customized by overriding {@link #getServletName}.
     */
    public static final String DEFAULT_SERVLET_NAME = "dispatcher";

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        super.onStartup(servletContext);
        //注册DispatcherServlet
        registerDispatcherServlet(servletContext);
    }
}
**/
```

1、`super.onStartup` 完成了实例化 `listener` 的工作

```


    /**
     * protected void registerContextLoaderListener(ServletContext servletContext) {
     *
     *     //创建spring上下文, 注册了SpringContainer
     *     WebApplicationContext rootAppContext = createRootApplicationContext();
     *     if (rootAppContext != null) {
     *         //创建监听器
     *         /*
     *             形如这种配置
     *             <listener>
     *                 <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
     *                 <!--<listener-class>org.springframework.web.context.request.RequestContextListener</listener-class>
     *             </listener>
     *             */
     *             ContextLoaderListener listener = new ContextLoaderListener(rootAppContext);
     *             listener.setContextInitializers(getRootApplicationContextInitializers());
     *             servletContext.addListener(listener);
     *     } else {
     *         logger.debug("No ContextLoaderListener registered, as " +
     *                     "createRootApplicationContext() did not return an application context");
     *     }
     * }


```

这些代码功能就类似于在 web.xml 配置了 ContextLoaderListener，做了几个事情，1 创建了上下文对象，如图：

```


protected WebApplicationContext createRootApplicationContext() {
    Class<?>[] configClasses = getRootConfigClasses();
    if (!ObjectUtils.isEmpty(configClasses)) {
        AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
        context.register(configClasses);
        return context;
    } else {
        return null;
    }
}


```

这个上下文对象就是基于注解扫描的上下文对象，所以用这个上下文是需要注册一个类进去，这个类就是用钩子方法调用到了自己写的方法。

```


public class WebAppInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {

    //父容器
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{SpringContainer.class};
    }
}


```

在钩子方法中获取到的类 SpringContainer 就会去扫描基本包，有@ComponentScan 注解，如图：

```


//不扫描有@Controller注解的类
@ComponentScan(value = "com.xiangxue.jack", excludeFilters = {
    @ComponentScan.Filter(type = FilterType.ANNOTATION, classes = {Controller.class})
})
public class SpringContainer {
}


```

通过钩子方法获取到扫描类后，注册到了上下文对象中，然后把 spring 的上下文对象设置到了 ContextLoaderListener 监听器对象中，最后把监听器对象设置到了 servletContext 中。这里上下文对象还没有调用 refresh 方法完成 spring 的启动。

2、registerDispatcherServlet(servletContext);完成了实例化 DispatcherServlet

```

//创建springmvc的上下文，注册了MvcContainer类
WebApplicationContext servletAppContext = createServletApplicationContext();
Assert.notNull(servletAppContext, message: "createServletApplicationContext() must not return null");

//创建DispatcherServlet
FrameworkServlet dispatcherServlet = createDispatcherServlet(servletAppContext);
Assert.notNull(dispatcherServlet, message: "createDispatcherServlet(WebApplicationContext) must not return null");
dispatcherServlet.setContextInitializers(getServletApplicationContextInitializers());

ServletRegistration.Dynamic registration = servletContext.addServlet(servletName, dispatcherServlet);
if (registration == null) {
    throw new IllegalStateException("Failed to register servlet with name '" + servletName + "'.
                                         "Check if there is another servlet registered under the same name.");
}

/*
 * 如果该元素的值为负数或者没有设置，则容器会当Servlet被请求时再加载。
 * 如果值为正整数或者0时，表示容器在应用启动时就加载并初始化这个servlet,
 * 值越小，servlet的优先级越高，就越先被加载
 */
registration.setLoadOnStartup(1);
registration.addMapping(getServletMappings());
registration.setAsyncSupported(isAsyncSupported());

```

步骤跟创建监听器差不多，创建上下文对象，跟上面差不多，创建 dispatcherServlet 对象，把 servlet 对象加入到 servletContext 上下文中。把上下文对象设置到了 dispatcherServlet 对象中了，这里上下文对象还没有调用 refresh 方法，没有启动 spring 容器。

3、启动 spring 容器

监听器的启动

```

*/
@Override
public void contextInitialized(ServletContextEvent event) {
    initWebApplicationContext(event.getServletContext());
}

```

其实没什么特别的，就是会拿到上下文对象调用 refresh 方法，需要特殊记忆的就是，会把上下文对象设置到 servletContext 中。

```

    servletContext.setAttribute(WebApplicationContext.ROOT_WEB_APPLICATION_CONTEXT_ATTRIBUTE, this.context);
}

```

DispatcherServlet 的启动

因为这个是一个 servlet，servlet 要完成 spring 容器的启动，就只能在 init 方法里面做。



```

145     * properties are missing), or if subclass initialization fails.
146
147     */
148     @Override
149     public final void init() throws ServletException {
150         // Set bean properties from init parameters.
151         PropertyValues pvs = new ServletConfigPropertyValues(getServletConfig(), this.requiredProperties);
152         if (!pvs.isEmpty()) {
153             try {
154                 BeanWrapper bw = PropertyAccessorFactory.forBeanPropertyAccess(target: this);
155                 ResourceLoader resourceLoader = new ServletContextResourceLoader(getServletContext());
156                 bw.registerCustomEditor(Resource.class, new ResourceEditor(resourceLoader, getEnvironment()));
157                 initBeanWrapper(bw);
158                 bw.setPropertyValues(pvs, ignoreUnknown: true);
159             }
160             catch (BeansException ex) {
161                 if (logger.isErrorEnabled()) {
162                     logger.error(o: "Failed to set bean properties on servlet " + getServletName() + "", ex);
163                 }
164                 throw ex;
165             }
166         }
167         // Let subclasses do whatever initialization they like.
168         initServletBean();
169     }
170 }

```

```

@Override
protected final void initServletBean() throws ServletException {
    getServletContext().log(msq: "Initializing Spring " + getClass().getSimpleName());
    if (logger.isInfoEnabled()) {
        logger.info(o: "Initializing Servlet '" + getServletName() + "'");
    }
    long startTime = System.currentTimeMillis();

    try {
        this.webApplicationContext = initWebApplicationContext();
        initFrameworkServlet();
    }
    catch (ServletException | RuntimeException ex) {
        logger.error(o: "Context initialization failed", ex);
        throw ex;
    }
}

```

这里也没什么特别的，也是拿到上下文对象调用了 refresh 方法完成 spring 容器的启动。

需要注意的是这里：

```

// 这里会从 servletContext 中获取到父容器，就是通过监听器加载的容器
WebApplicationContext rootContext =
    WebApplicationContextUtils.getWebApplicationContext(getServletContext());
WebApplicationContext wac = null;

if (this.webApplicationContext != null) {
    // A context instance was injected at construction time -> use it
    wac = this.webApplicationContext;
    if (wac instanceof ConfigurableWebApplicationContext) {
        ConfigurableWebApplicationContext cwac = (ConfigurableWebApplicationContext) wac;
        if (!cwac.isActive()) {
            // The context has not yet been refreshed -> provide services such as
            // setting the parent context, setting the application context id, etc
            if (cwac.getParent() == null) {
                // The context instance was injected without an explicit parent -> set
                // the root application context (if any; may be null) as the parent
                cwac.setParent(rootContext);
            }
        }
    }
}

```

会从 servletContext 中获取父容器

```

@Nullable
public static WebApplicationContext getWebApplicationContext(ServletContext sc, String attrName) {
    Assert.notNull(sc, message: "ServletContext must not be null");
    Object attr = sc.getAttribute(attrName);
    if (attr == null) {

```

就是由 listener 负责启动的容器，然后把父容器设置到了自己的上下文对象中，所以这里监听器启动的容器是父容器，dispatcherServlet 启动的容器是子容器，两者是父子关系。

这里就用 servlet 规范完成了取代 web.xml 的工作，并启动了容器。

取代 springmvc.xml 配置

我们用一个 @EnableWebMvc 就可以完全取代 xml 配置，其实两者完成的工作是一样的，都是为了创建必要组件的实例



```
@Configuration
@EnableWebMvc
public class AppConfig extends WebMvcConfigurerAdapter {

    @Autowired
    private UserInterceptor userInterceptor;

    @Override
    public void configureViewResolvers(ViewResolverRegistry registry) { registry.jsp(prefix: "/WEB-INF/views/", suffix: ".jsp"); }

    @Override
    public void configureDefaultServletHandling(DefaultServletHandlerConfigurer configurer) { configurer.enable(); }

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(userInterceptor).addPathPatterns("/user/**").excludePathPatterns("/user/query/**");
        registry.addInterceptor(new UserInterceptor1()).addPathPatterns("/user/**").excludePathPatterns("");
        super.addInterceptors(registry);
    }

}
```

这里会导入一个核心类 WebMvcConfigurationSupport，在这个类里面会完成很多组件的实例化，比如 HandlerMapping，HandlerAdapter 等等。如图：

```
@Bean
public RequestMappingHandlerMapping requestMappingHandlerMapping() {
    RequestMappingHandlerMapping mapping = createRequestMappingHandlerMapping();
    mapping.setOrder(0);
    mapping.setInterceptors(getInterceptors());
    mapping.setContentNegotiationManager(mvcContentNegotiationManager());
    mapping.setCorsConfigurations(getCorsConfigurations());

    PathMatchConfigurer configurer = getPathMatchConfigurer();

    Boolean useSuffixPatternMatch = configurer.isUseSuffixPatternMatch();
    if (useSuffixPatternMatch != null) {
        mapping.setUseSuffixPatternMatch(useSuffixPatternMatch);
    }
    Boolean useRegisteredSuffixPatternMatch = configurer.isUseRegisteredSuffixPatternMatch();
    if (useRegisteredSuffixPatternMatch != null) {
        mapping.setUseRegisteredSuffixPatternMatch(useRegisteredSuffixPatternMatch);
    }

    @Bean
    @Nullable
    public HandlerMapping viewControllerHandlerMapping() {
        ViewControllerRegistry registry = new ViewControllerRegistry(this.applicationContext);
        addViewControllers(registry);

        AbstractHandlerMapping handlerMapping = registry.buildHandlerMapping();
        if (handlerMapping == null) {
            return null;
        }
        handlerMapping.setPathMatcher(mvcPathMatcher());
        handlerMapping.setUrlPathHelper(mvcUrlPathHelper());
        handlerMapping.setInterceptors(getInterceptors());
        handlerMapping.setCorsConfigurations(getCorsConfigurations());
        return handlerMapping;
    }
}
```

```

@Bean
public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {
    RequestMappingHandlerAdapter adapter = createRequestMappingHandlerAdapter();
    adapter.setContentNegotiationManager(mvcContentNegotiationManager());
    adapter.setMessageConverters(getMessageConverters());
    adapter.setWebBindingInitializer(getConfigurableWebBindingInitializer());
    adapter.setCustomArgumentResolvers(getArgumentResolvers());
    adapter.setCustomReturnValueHandlers(getReturnValueHandlers());

    if (jackson2Present) {
        adapter.setRequestBodyAdvice(Collections.singletonList(new JsonViewReques

```

然后在实例化过程中会涉及到很多钩子方法的调用，而这些钩子方法就是我们需要去实现的，比如获取拦截器的钩子方法，获取静态资源处理的钩子方法等等。如图：



请求之前建立映射关系

在 HandlerMapping 类实例化的时候就会完成 url 和 method 的映射关系，要根据一个请求能够唯一到找到一个类和一个方法。

```

@Bean
public RequestMappingHandlerMapping requestMappingHandlerMapping() {
    RequestMappingHandlerMapping mapping = createRequestMappingHandlerMapping();
    mapping.setOrder(0);
    mapping.setInterceptors(getInterceptors());
    mapping.setContentNegotiationManager(mvcContentNegotiationManager());
    mapping.setCorsConfigurations(getCorsConfigurations());

    PathMatchConfigurer configurer = getPathMatchConfigurer();

    Boolean useSuffixPatternMatch = configurer.isUseSuffixPatternMatch();
    if (useSuffixPatternMatch != null) {
        mapping.setUseSuffixPatternMatch(useSuffixPatternMatch);
    }
    Boolean useRegisteredSuffixPatternMatch = configurer.isUseRegisteredSuffixPat
    if (useRegisteredSuffixPatternMatch != null) {
        mapping.setUseRegisteredSuffixPatternMatch(useRegisteredSuffixPatternMatc
    }
}

```

这个是 RequestMappingHandlerMapping 的实例化。在其父类 AbstractHandlerMethodMapping 中实现了 InitializingBean 接口，所以在 RequestMappingHandlerMapping 实例化完成以后就会调用到 afterPropertiesSet 方法。在这个方法里面完成了映射关系的建立。

这里判断类上面是否有 @Controller 注解和 @RequestMapping 注解，只有这种类才需要建立映射关系

```
//如果类上面有@Controller注解或者@RequestMapping注解
if (beanType != null && isHandler(beanType)) {
    //建立uri和method的映射关系
    detectHandlerMethods(beanName);
}
```

大体思路

1、循环类里面的所有方法

2、收集方法上面的@RequestMapping注解，把注解里面的配置信息封装到类里面，该类就是RequestMappingInfo类，并且跟类上面的@RequestMapping注解封装类RequestMappingInfo合并，比如类上面是/common，方法上面是/queryUser。这两者合并后就是/common/queryUser。这样的url才是我们需要的。合并完就是这样的url

3、然后建立method对象和对应的RequestMappingInfo的映射关系，把关系存放到map中。

4、创建HandlerMethod对象，该类型封装了method、beanName、bean、方法类型等信息

5、建立RequestMappingInfo和HandlerMethod的映射关系

```
//建立uri对象和handlerMethod的映射关系
this.mappingLookup.put(mapping, handlerMethod);
```

6、建立url和RequestMappingInfo对象的映射关系

```
//建立url和RequestMappingInfo映射关系
this.urlLookup.add(url, mapping);
```

这样映射关系就已经建立好，这样根据请求url我们就可以唯一的找到一个HandlerMethod对象了，注意这个对象中还不能进行反射调用，还确实参数数组。

dispatcherServlet 处理请求

当请求过来时，首先会调用到service方法，最终会调用到dispatcherServlet中的doDispatch方法。

1、根据请求url获取HandlerExecutionChain对象

```
//这个方法很重要，重点看
// Determine handler for the current request.
mappedHandler = getHandler(processedRequest);
if (mappedHandler == null) {
```

寻找 HandlerMethod 的过程，感觉没什么好说的，前面映射关系已经建立好了，现在就是只需要从 request 对象中获取请求 url，然后从映射关系中获取 HandlerMethod 对象就可以了，如图：

```
/*
@Override
protected HandlerMethod getHandlerInternal(HttpServletRequest request) throws Exception {
    //从request对象中获取uri, /common/query2
    String lookupPath = getUrlPathHelper().getLookupPathForRequest(request);
    this.mappingRegistry.acquireReadLock();
    try {
        //根据uri从映射关系中找到对应的HandlerMethod对象
        HandlerMethod handlerMethod = lookupHandlerMethod(lookupPath, request);
        //把Controller类实例化
        return (handlerMethod != null ? handlerMethod.createWithResolvedBean() : null);
    }
    finally {
        this.mappingRegistry.releaseReadLock();
    }
}
```

获取过程是，先从

```
//建立url和RequestMappingInfo映射关系
this.urlLookup.add(url, mapping);
```

urlLookup 中获取 RequestMappingInfo 对象，然后再跟进 RequestMappingInfo 对象从

```
//建立uri对象和handlerMethod的映射关系
this.mappingLookup.put(mapping, handlerMethod);
```

获取 HandlerMethod 对象。

获取到 HandlerMethod 对象后，把 HandlerMethod 对象封装到 HandlerExecutionChain 对象中了。

```
//获取HandlerMethod和过滤器链的包装类
HandlerExecutionChain executionChain = getHandlerExecutionChain(handler, request);
```

这个对象，启动就是封装了 HandlerMethod 和一个拦截器数组而已。

2、前置过滤器

```
//前置过滤器，如果为false则直接返回
if (!mappedHandler.applyPreHandle(processedRequest, response)) {
    return;
}
```

拿到 HandlerExecutionChain 对象进行过滤器的调用，调用了 preHandle 方法，只要这个方法返回为 false，则后续请求就不会继续。

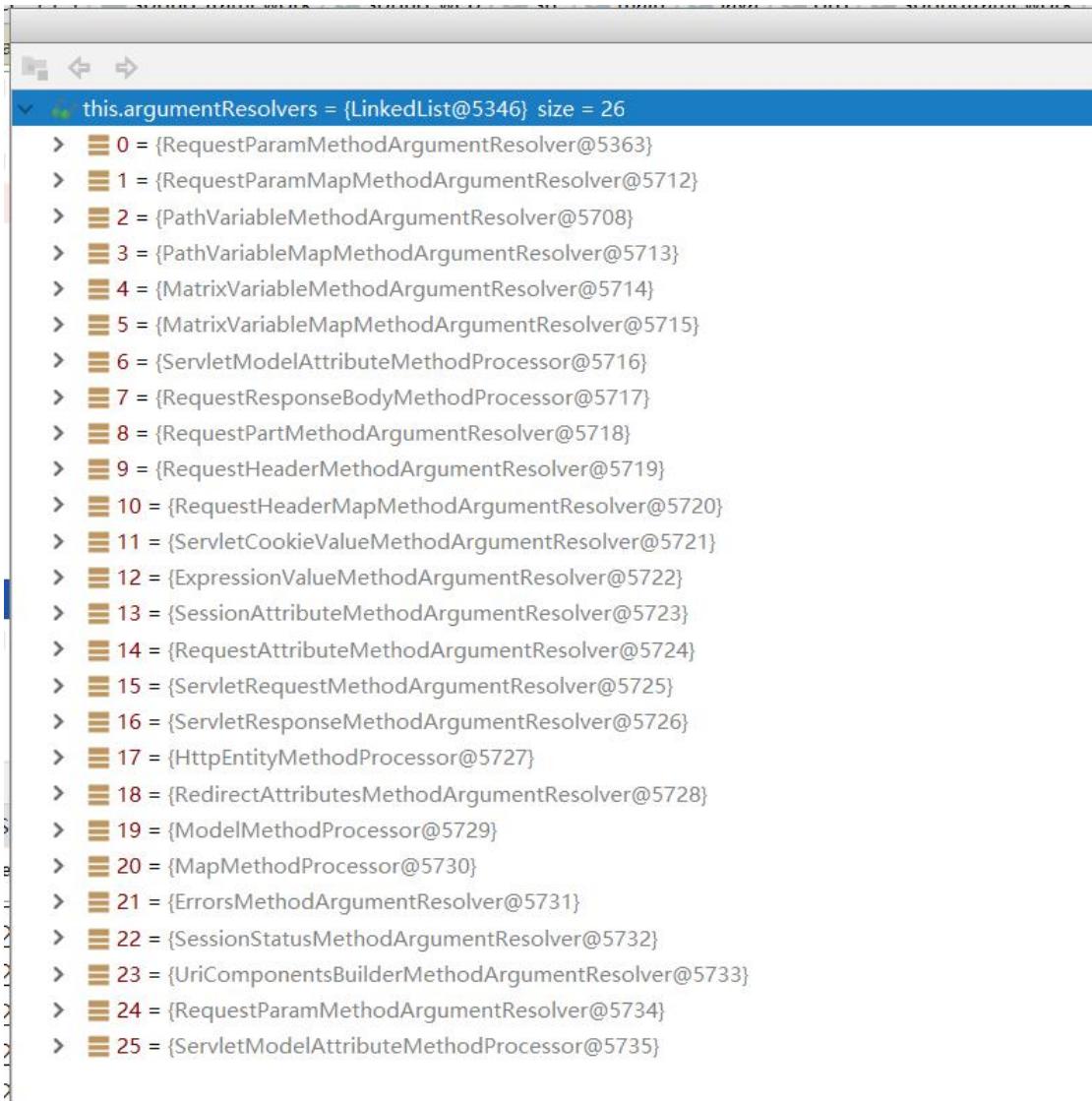
3、HandlerAdapter 调用 handle 方法，进行具体 Controller 中方法的调用

这个调用过程，关键点就在于参数的解析，其他都没什么技术含量。

```
    /*
     * @Nullable
     */
    public Object invokeForRequest(NativeWebRequest request, @Nullable ModelAndView mavContainer,
        Object... providedArgs) throws Exception {
        //获取参数数组,重点看
        Object[] args = getMethodArgumentValues(request, mavContainer, providedArgs);
        if (logger.isTraceEnabled()) {
            logger.trace(o: "Arguments: " + Arrays.toString(args));
        }
        return doInvoke(args);
    }
}
```

首先获取方法的参数列表，并且把参数封装成 MethodParameter 对象，这个对象记录了参数在参数列表中的索引，参数类型，参数上面的注解数组等等信息。

然后循环参数列表，一个个参数来处理，这里是一个典型的策略模式的运用，根据参数获取一个处理该参数的类。处理参数的解析类有 26 个，如图：



把参数一个个处理完成后，放到一个参数数组中了

```
Object[] args
```

接下来就是反射调用了，有方法 method 对象，有类对象，有参数数组就可以进行反射调用了。

```
        }
        return doInvoke(args);

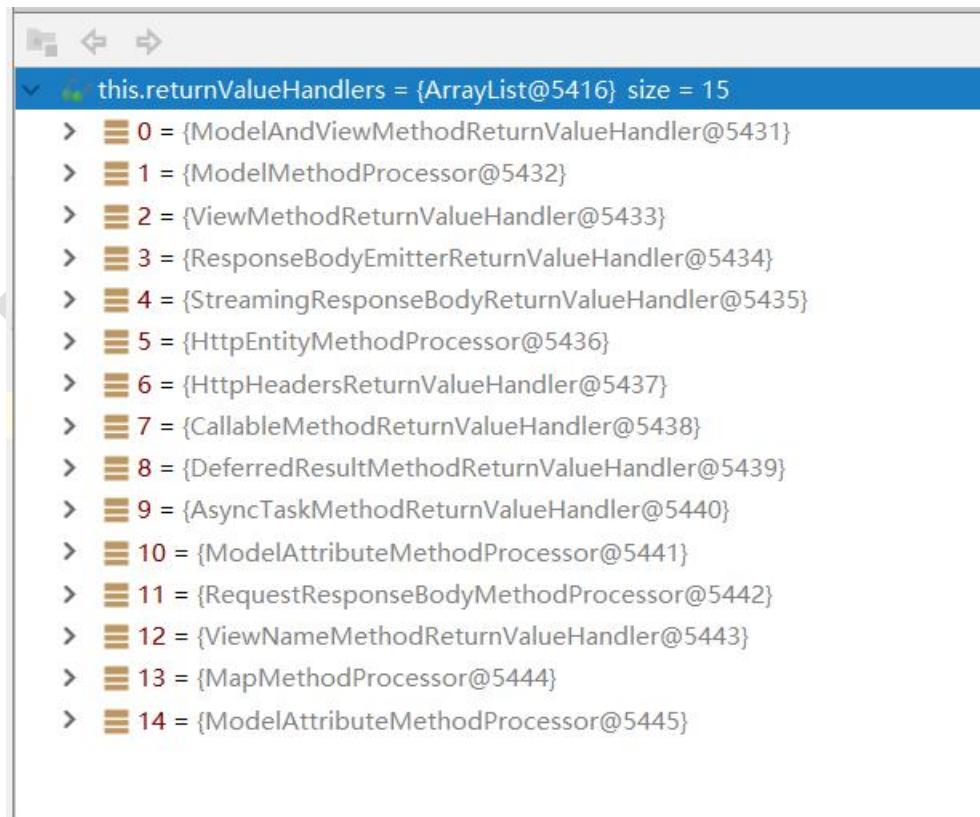
    --> return getBridgedMethod().invoke(getBean(), args);
}
```

返回值处理

当反射调用成功后，有可能方法会有返回值，而返回值处理也是一个比较重要的事情，根据什么样的方式把返回值响应回去，返回值响应时有可能是数据有可能是界面，而如果返回数据的话，要把返回值解析成对应的格式，例如如果返回值是一个 list 对象，就需要解析这个 list 对象把 list 对象解析成 json 格式。返回值解析讨论跟入参解析基本上类似。

- 1、把返回值封装成对象，对象跟 MethodParameter 对象差不多，里面包括参数名称、类型、参数注解等等信息
- 2、根据返回值类型用策略模式找到一个解析类
- 3、用这个解析类解析
- 4、这块跟两个比较典型的就差不多了，一个是带@ResponseBody 注解的，一个是直接返回字符串响应一个界面的，里面涉及到一个 ModelAndViewContainer 容器，这个容器会把视图名称设置到里面，已经 Model 数据，就是响应到界面的数据也会放到这个容器中。

返回值处理类：

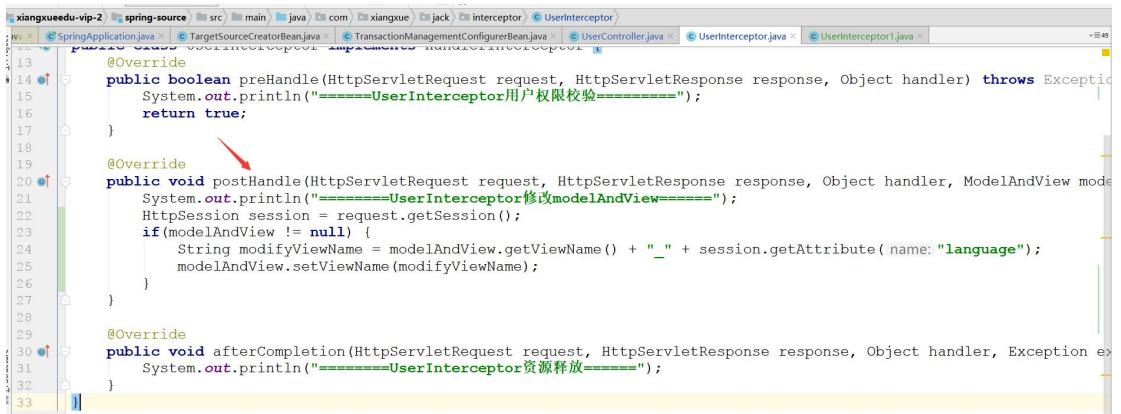


中置过滤器

中置过滤器的调用时序，是当 `ha.handle` 掉完以后，也就是 Controller 里面具体方法调用完以后才轮到中置过滤器调用。

```
//中置过滤器  
mappedHandler.applyPostHandle(processedRequest, response, mv);
```

中置过滤器的应用：



```
13  
14 @Override  
15     public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {  
16         System.out.println("=====UserInterceptor用户权限校验=====");  
17         return true;  
18     }  
19  
20 @Override  
21     public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) throws Exception {  
22         System.out.println("=====UserInterceptor修改(modelAndView)=====");  
23         HttpSession session = request.getSession();  
24         if(modelAndView != null){  
25             String modifyViewName = modelAndView.getViewName() + "_" + session.getAttribute("name:language");  
26             modelAndView.setViewName(modifyViewName);  
27         }  
28     }  
29  
30 @Override  
31     public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception exception) throws Exception {  
32         System.out.println("=====UserInterceptor资源释放=====");  
33     }
```

可以用来修改视图。

视图渲染

其实就是响应界面，如果返回值没有加`@ResponseBody` 注解时，这时候是需要响应一个界面给前端的，视图渲染借助了 servlet 中的 api，如图：

```
@Override  
protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {  
    System.out.println("=====doget====");  
    PrintWriter writer = resp.getWriter();  
    writer.print("<h1>Jack</h1>");  
  
    RequestDispatcher requestDispatcher = req.getRequestDispatcher(path: "/jsp/ok_en.jsp");  
    requestDispatcher.forward(req, resp);  
}
```

这样 servlet 就可以响应一个界面给前端。而我们 spring 也是差不多的处理方式。

视图渲染入口：

```
// Did the handler return a view to render?  
//视图渲染，响应视图  
if (mv != null && !mv.wasCleared()) {  
    render(mv, request, response);  
    if (errorView) {  
        WebUtils.clearErrorRequestAttributes(request);  
    }  
}
```

界面响应核心逻辑：

```

spring-source-5.1.3 | spring-framework | spring-webmvc | src | main | java | org | springframework | web | servlet | view | InternalResourceView.java
HandlerMethod.java | HandlerMethodArgumentResolverComposite.java | HandlerMethodReturnValueHandlerComposite.java | InitServlet.java | InitServlet1.java | InternalResourceView.java | -34

String dispatcherPath = prepareForRendering(request, response);

// Obtain a RequestDispatcher for the target resource (typically a JSP).
RequestDispatcher rd = getRequestDispatcher(request, dispatcherPath);
if (rd == null) {
    throw new ServletException("Could not get RequestDispatcher for [" + getUrl() +
        "]: Check that the corresponding file exists within your web application archive!");
}

// If already included or response already committed, perform include, else forward.
if (useInclude(request, response)) {
    response.setContentType(getContentType());
    if (logger.isDebugEnabled()) {
        logger.debug("Including [" + getUrl() + "]");
    }
    rd.include(request, response);
}

else {
    // Note: The forwarded resource is supposed to determine the content type itself.
    if (logger.isDebugEnabled()) {
        logger.debug("Forwarding to [" + getUrl() + "]");
    }
    rd.forward(request, response);
}

```

异常解析

Controller 调用过程中的异常解析使用，如图：

```

@ControllerAdvice("com.xiangxue.jack")
public class ExceptionHandlerControllerAdvice {

    @ExceptionHandler({ArrayIndexOutOfBoundsException.class})
    public @ResponseBody String handlerArrayIndexOutOfBoundsException(Exception e) {
        System.out.println("====ExceptionHandlerControllerAdvice-->" + e.getMessage());
        return "ArrayIndexOutOfBoundsException";
    }

    @ExceptionHandler({NullPointerException.class})
    public @ResponseBody String handlerNullPointerException(Exception e) {
        System.out.println("====ExceptionHandlerControllerAdvice-->" + e.getMessage());
        return "NullPointerException";
    }
}

```

类上面加上`@ControllerAdvice("com.xiangxue.jack")`注解，这个包定义就是只对这个包里面的 Controller 生效。然后类里面的方法加上

`@ExceptionHandler({ArrayIndexOutOfBoundsException.class})`

`@ExceptionHandler({NullPointerException.class})`

表示这个方法当调用过程中出现注解里面定义的异常时会被调用到，这些方法就是对异常处理的方法。

源码的核心思想差不多

- 1、收集注解包装成类
- 2、建立`@ExceptionHandler`中异常和 Method 的映射关系
- 3、根据出现的异常从映射关系中找到对应的 Method 对象
- 4、反射调用，这个调用逻辑跟 Controller 里面具体方法调用逻辑一模一样