01、SpringBoot源码分析

必须要掌握源码分析

- spring （重点）
- springmvc（重点）
- springboot （重点）
- mybatis（重点）
- mp
- tomcat
- dubbo
- 微服务源码分析

如何看源码

- 你要明白接口意义，继承意义？
- 为什么我们要写实现类实现该接口？
- 实现接口的目的是什么？谁来调用这个接口实现类？
- 源码主线很多，但是你一定看核心主线，或者某个点，从点到面。逐个击破。
- 设计模式，开发架构思想。

SpringBoot简化了基于Spring的Java应用开发，降低了使用难度。从这个意义上来讲SpringBoot是对Spring框架的进一步封装。这个封装是的很多使用者：“只知其然，而不知其所以然”。在使用过程中出现问题时，不知道如何排错或者不能更好地使用SpringBoot。接下来本章节从几个维度来分析SpringBoot的源码：

- SpringBoot的自动配置原理
- SpringBoot启动流程
- SpringBoot的自定义starter
- SpringBoot的内嵌Web服务器原理
- SpringBoot和Spring的关系
- SpringBoot的扫描包@CompentScan的原理
- SpringBoot的@Value的原理

02、下载SpringBoot的源码

springboot的源代码托管给了github。下载地址是：

> *https://github.com/spring-projects/spring-boot.git*

手动下载：

```
git clone https://github.com/spring-projects/spring-boot.git
```

03、源码分析核心主线

springboot是对spring的封装，最终springboot中将所初始化bean都将放入到ioc容器中去

思考问题

SpringBoot它如何去取代传统的基于xml的方式spring注入bean的方式呢

答案：利用java面向对象的方式 + 注解机制

本质问题：

使用spring框架到底解决了开发中什么问题?

- 定义需要被ioc容器管理的类（bean） -----(xml/注解注册bean) ---- 查找bean(xml/主键) --| 接口（根据接口找到所有的实现类） |-- 创建对象 -| 接口（根据接口找到所有的实现类） |-- 存储对象 (map)--- 使用对象---注入对象---注销 （springbean的生命周期）
- 给属性注入值 （依赖注入）

使用spring的最大的感受是什么？

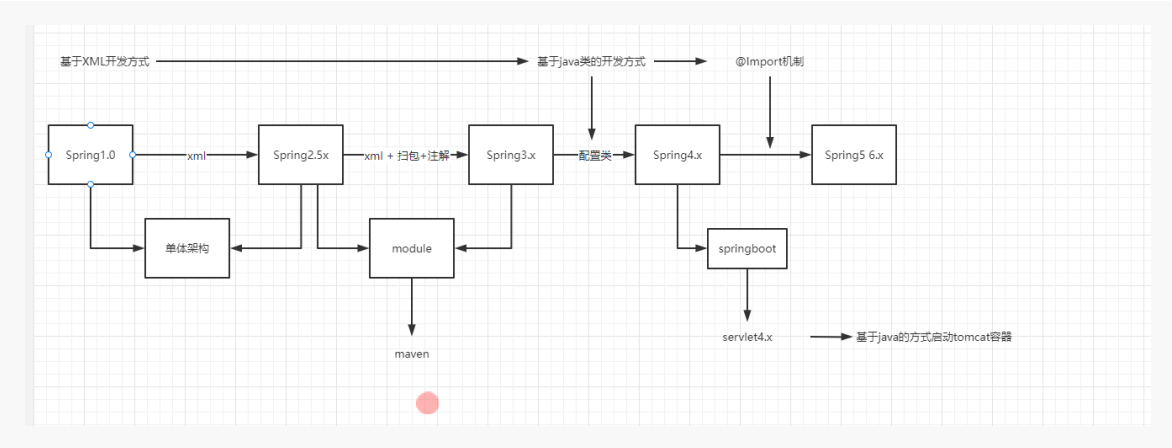
从传统的开发使用new创建对象的方式转变到容器管理对象的过程。它其实并没有脱离java面向对象。

spring是一个框架，也是一个产品。是使用java语音开发出来的产品而已。

用spring的好处

- 统一管理bean，不需要自己在去创建对象 容器管理对象的过程。
- 单列的对象 （在内存中存在一份内存空间，防止bean频繁创建造成内存消耗）
- 为后续对象的改造和增强提供了基础（AOP）
- 从传统的开发方式转变到容器管理对象的转变。

04、Spring整个发展历程



05、Spring第一阶段：基于XML的方式

Spring是如何把开发类（bean）放入它容器去管理的呢？

1：必须要由一个类(管理bean)：

```
1  package com.kuangstudy.service1;
2
3  /**
4   * @author 飞哥
5   * @Title: 学相伴出品
6   * @Description: 我们有一个学习网站：https://www.kuangstudy.com
7   * @date 2021/12/30 20:54
8   */
9  public class UserServiceImpl implements IUserService {
10
11     public void saveUser() {
```

```
        System.out.println("保存用户了...");
    }
}

```

2：使用xml方式定义bean节点

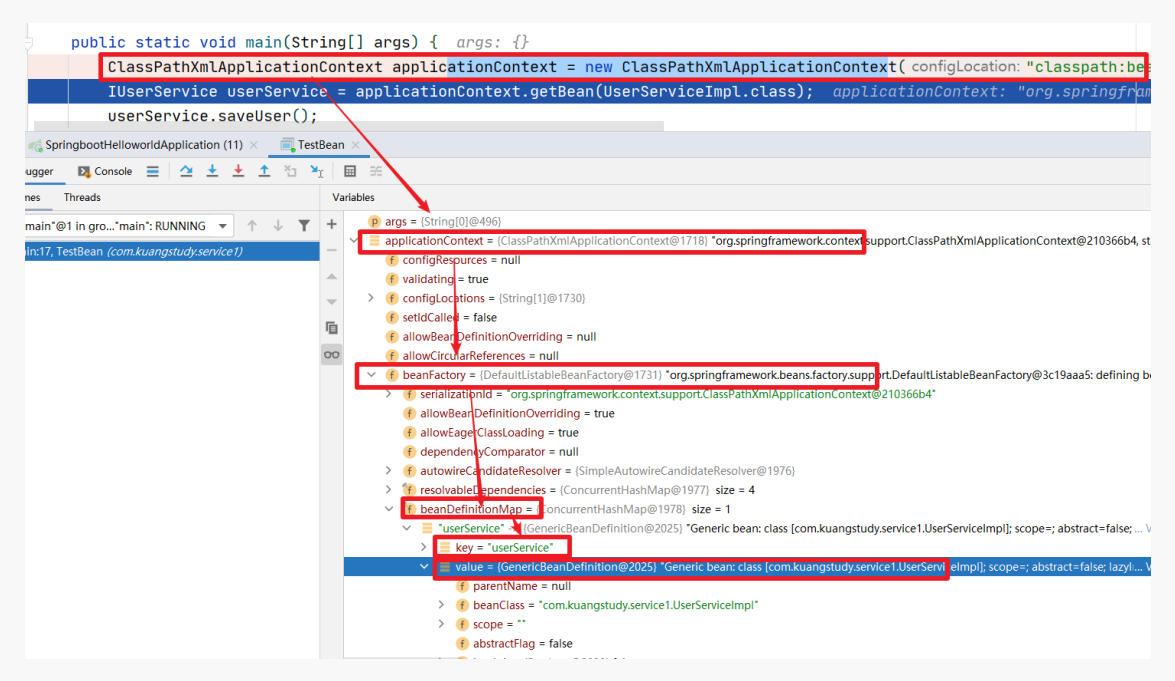
```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
xmlns:tx="http://www.springframework.org/schema/tx"

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
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx.xsd">

    <!--配置了bean节点-->
    <bean id="userService"
class="com.kuangstudy.service1.UserServiceImpl"></bean>


</beans>
```

3：必须要找到一个上下文

```
package com.kuangstudy.service1;

import
org.springframework.context.support.ClassPathXmlApplicationContext;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/30 20:56
 */
public class TestBean {

```

```
15      public static void main(String[] args) {
16          ClassPathXmlApplicationContext applicationContext = new
    ClassPathXmlApplicationContext("classpath:bean.xml");
17          IUserService userService =
    applicationContext.getBean(UserServiceImpl.class);
18          userService.saveUser();
19      }
20  }
21
```

4：上下文作用和功能：加载xml，解析xml，并且并解析的bean节点用java类存起来（就好比：jdbc查询数据库一样，把每条表的记录用一个bean去存起来是一样的道理。db----bean）（xml--bean）



5：存起来放容器中(ConcurrentHashMap)

上面截图开源看出来，ioc容器就是一个 ConcurrentHashMap

06、Spring第二阶段：基于扫包+注解的方式

Spring基于扫包+注解的方式

- 好处：从xml大量配置和依赖过程解放出来了。但是能够完全解放，不能。
- 缺陷：包名是固定，包名也不统一。

实现步骤：

1：定义一个OrderService

一定要加注解 @Service

```
package com.kuangstudy.service2;

import org.springframework.context.annotation.ClassPathBeanDefinitionScanner;
import org.springframework.stereotype.Service;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/30 21:06
 */
@Service("aorservcie")// 什么情况下去改呢？一般如果你类名出现冲突，apackage.OrderService, bpackage.OrderService
public class OrderService {


    public void makeorder() {
        System.out.println("下单了...");
    }
}

```

2：配置扫包

```
<!--扫包的方式：-->
<context:component-scan base-package="com.kuangstudy.service2">
</context:component-scan>
```

3：打印使用

```
package com.kuangstudy.service1;

```

```
import com.kuangstudy.service2.OrderService;
import
org.springframework.context.support.ClassPathXmlApplicationContext;

import java.util.concurrent.ConcurrentHashMap;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/30 20:56
 */
public class TestBean {

    public static void main(String[] args) {
        ClassPathXmlApplicationContext applicationContext = new
ClassPathXmlApplicationContext("classpath:bean.xml");
//        IUserService userService =
applicationContext.getBean(UserServiceImpl.class);
//        OrderService orderService =
(OrderService)applicationContext.getBean("orderService");
//        userService.saveUser();
//        orderService.makeorder();

        String[] beanDefinitionNames =
applicationContext.getBeanDefinitionNames();
        for (String beanDefinitionName : beanDefinitionNames) {
            System.out.println(beanDefinitionName);
        }

    }
}

```

结果

```
orderService
org.springframework.context.annotation.internalConfigurationAnnotationProcessor
org.springframework.context.annotation.internalAutowiredAnnotationProcessor
org.springframework.context.annotation.internalCommonAnnotationProcessor
org.springframework.context.event.internalEventListenerProcessor
org.springframework.context.event.internalEventListenerFactory
userService
```

07、Spring第三阶段：配置@Configuration + @Bean的方式

Spring配置@Configuration + @Bean的方式

这个中模式真正意义上的零配置的起始，如下步骤:

1：定义bean

```java
package com.kuangstudy.service3;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/30 21:27
 */
public class CourseService {

    public void saveCourse() {
        System.out.println("保存课程!!!");
    }
}

```

2：写一个配置类

```
package com.kuangstudy.service3;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import javax.swing.*;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/30 21:27
 */
@Configuration
public class CourseServiceConfiguration {

    @Bean
    public CourseService courseService() {
        return new CourseService();
    }
}

```

3：找个上下文

```
package com.kuangstudy.service3;

import
org.springframework.context.annotation.AnnotationConfigApplicationCo
ntext;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/30 21:27
 */
public class TestConfig {

    public static void main(String[] args) {
```

```
16          AnnotationConfigApplicationContext applicationContext =
17                  new
    AnnotationConfigApplicationContext(CourseServiceConfiguration.class)
    ;
18          CourseService courseService =
    applicationContext.getBean(CourseService.class);
19          courseService.saveCourse();
20      }
21  }
22
```

- 使用AnnotationConfigApplicationContext来加载配置类。通过配置类和@Bean方法最大感受就是没有XML文件了。

08、SpringBoot大融合

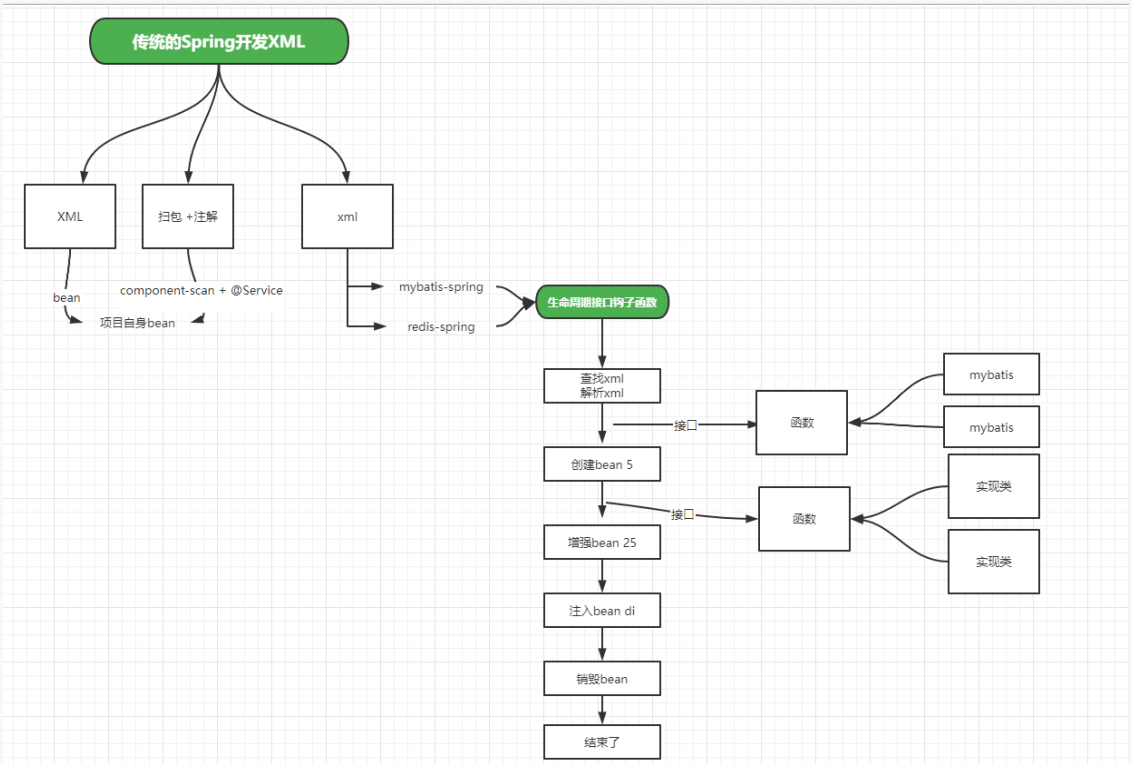
- 基于传统xml的方式：在默认包类上（启动类）用 @ImportResource("classpath:bean.xml")
- 扫包 + 注解 --- 使用启动类@ComponentScan + @Service / @Controller / @Component 取代XML的方式，默认就是从当前@ComponentScan注解的类的包作为扫包的入口。比如：启动类，一般来说在项目中放在启动类即可。没必要在其他的配置类取重复申明扫包。因为默认扫包递归的。如果在其他包下继续申明扫包就会出现冲突，引发报错

```
1  <context:component-scan base-package="com.kuangstudy.service2">
   </context:component-scan> + @Service / @Controller / @Component
```

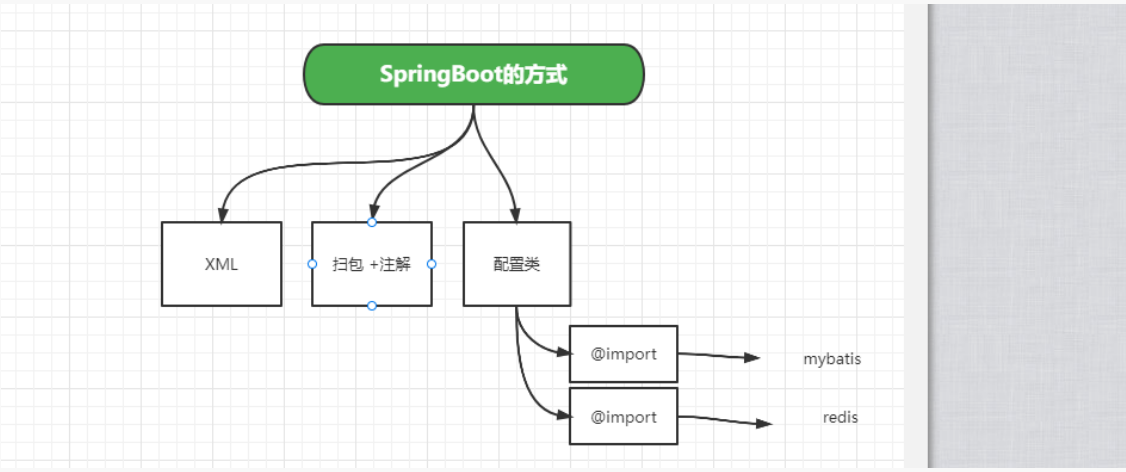
- 配置类的方式 --- @Conifugration+@Bean的方式(最大化体验)
- @Import + 自动装配 （配置类，selectImport）

SpringBoot是如何引入第三方这种机制的呢？

- 基于传统的xml的方式是通过：生命周期接口钩子函数进行扩展和融合第三方的技术。



- springboot是通过配置类+@Import机制来完成的。



但是会存在一个问题：项目存在无限的@import比如：

```
@Import({
    ProjectInfoAutoConfiguration.class,
    TaskSchedulingAutoConfiguration.class,
        MybatisAutoConfiguration.class,
        RedisAutoConfiguration.class
})
```

- 存在问题1：如果我要依赖第三方的东西就必须要知晓它们对应配置类和属性配置类。
- 存在问题2：维护起来是一个灾难。耦合度太高了。不灵活。

解决方案：

所有的加载第三方的配置类(starter)的过程全部使用自动装配的方式。

09、@SpringBootConfiguration存在的意义？

原理是：@Configuration

如果一个类被标注： @SpringBootConfiguration 说明就是配置类。你就可以做下面的事情，如下：

配置类好处

- 可以突破包的限制
- 方便扩展和配置

主要的应用

- 开发 starter (配置类 + @Bean）（1%自研starter）
- 覆盖内部的stater的配置类 （99%可能性都用做覆盖用的）
    - 官方提供starter的配置和@Bean有些时候是满足不了业务需求，是进行覆盖扩展。
- 初始化一个bean (但是在项目自身：其实都扫包注解就完成了)

我们什么时候会用到配置类，或者在项目中看到一个配置类思考是什么？

- 如果你觉得官方提供stater里面提供配置类，满足不了你需求的时候，你可以考虑定义一个配置类去覆盖内部的规则，（个性化定制）。
- 开发精神，自研starter ，因为你要明白一个道理，官方提供starter就100个多，不可能满足以后你项目中所有的需求和满足所有技术服务，这个如果满足不你就必须自己去开发starter,这个时候用配置类 + @Bean
- 如果在项目中看到有配置类，说明两点：要么覆盖内部的配置类 要么就是：公司写的注解。你可以直接拿来使用。

为什么在项目中我们不用配置类去初始化**mapper ,service ,controller**里面的**bean**呢？

因为 @ComponentScan("com.xq") 给做了，所以没必要多此一举去做这个事情。当然如果你应用去用配置类去初始化也是没有问题。只不过不推荐。

小结

从这节课程中明白一个代理，扫包其实可以解决依赖第三方依赖的问题，但是不灵活，因为你不知道有多少个包要进行管理和扫包。这个就排除了。配置类也可以达到和扫包一样的目的，可以把对应的bean放入到ioc容器，但是它好处就是可以：突破包的限制问题，这也是为什么官方提供的stater和第三方提供的starter 内部的架构方式全部都是配置类，其实就依托配置不受包限制的问题。这样就可以非常灵活的扩展和覆盖的bean。

10、@SpringBootApplication 注解简单认识

- @SpringBootConfiguration : 覆盖第三方的配置类，内部的starter全部是用配置类来完成的定义的，就是因为：配置类可以突破包的限制，
- @EnableAutoConfiguration： 用于自动装配第三方的配置类 + @Bean 的机制。比如 mybatis 官方提供的spring的starter

- @ComponentScan : 扫包+注解（@Service ， @Controller,@Component等），用于加载项目自身的service,mapper,dao,web,controller的。

一句话：

- @SpringBootConfiguration 这个是标准和扩展的规范，突破包的限制。(如果你看到官方不爽开源考虑覆盖，或者官方没有提供自研starter)
- @ComponentScan 加载自己的bean
- @EnableAutoConfiguration 加载starter官方的自定义starterbean .加载别人的bean

11、starter机制里面的原理

- 全部都配置类 @Configuration + @Bean + 属性配置类
- 条件注解@Condition

为什么要配置类：可以突破包的限制问题。

12、为什么要会提供一个@Import机制

自动装配其实就指：@EnableAutoConfiguration。@EnableAutoConfiguration它是一个注解，内部的原理是：@Import(AutoConfigurationImportSelector.class)

@Import

它spring自身提供的一种专门用来加载配置类、selectImport接口子类子类的机制。它可以在容易启动的时候会把@Import的配置类，和对应selectImport接口子类子类进行初始化和把对应的bean加载ioc容器中。说白了：就是另外一种加载bean到ioc容器的机制。类似于：xml中的

@Import的思想：模块化，组件化，可插拔的机制。

@Import 作用

- 开关机制，模块化，组件化（自定义starter）
- SpringBoot项目连接第三方技术的机制（加载）

分布式项目架构

```
-- kss-admin-parent
  -- kss-pug-common
  -- kss-pug-mapper
  -- kss-pug-web
  -- kss-pug-service
  -- kss-pug-pojo
  -- kss-coponent-katcha-config
  -- kss-coponent-katcha-starter
     --- META-INF/spring.factoires
  -- kss-pug-weixin-api
  -- kss-pug-admin-api
     -- @Import(kss-coponent-katcha-config)
pom.xml
```

13、Starter中的配置类是如何加载到项目去的？

比如:mybatis ,redis,kafka等都用@Import来完成会怎么样呢？出现和扫包和注解一样恶心的事情。你必须一个个配置类找到去配置和定义和引入。

```
<!--mybatis-plus依赖-->
<dependency>
    <groupId>com.baomidou</groupId>
    <artifactId>mybatis-plus-boot-starter</artifactId>
    <version>3.4.2</version>
</dependency>

<!--web依赖-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>

<!--redis-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
```

比如上面的web,mybatis，redis 都是starter，我们starter里面全部是有配置类构成的，那么这些个配置类是如何加载到项目中去的？

答案就是：@Import

如下：

```
@Import({
    RedisAutoConfiguration.class,
    KafkaAutoConfiguration.class,
    MybatisPlusAutoConfiguration.class
})
```

但是如果这样去做的化，会出现一个问题，一样要直到里面的配置类是上面名字，就会出现上面的问题。和扫包其实没上面差不别如下：

```
@ComponentScan(basePackage={
    "com.kuagnstudy",
    "org.spring.redis",
    "org.apache.mybatis"
})
```

所以为了避免@Import频繁的去导入配置类，springboot提供了一个自动装配机制。

另外加载bean的机制也应用而生：importSelector接口。这个接口的子类就是自动装配的原理。

14、SpringBoot自动装配原理

springboot借鉴了java9的新特性spi机制。

- 1：用一种约定成俗机制，spi机制，在你每个偶starter的项目下resources/META-INF/spring.factories
  com.kuangstudy.selectimport.MyEnableAutoConfiguration = com.kuangstudy.config.KafkaConfiguration
- 2：同类加载器把你项目jar找到，mybatis-boot-starter.jar spring-boot-kafka-starter.jar
- 3: 用io流读取这个文件，然后里面对应注解对应的com.kuangstudy.selectimport.MyEnableAutoConfiguration的配置类全部找到
- 4: 放入到一个集合中，准备就绪，
- 5: SpringApplication.run()启动项目就会触发spring的生命周期refresh();
- 6: 在spring生命周期的某个方法会去找到 ImportSelector所有的子类调用selectImports()方法。
- 7：在这个方法中，把匹配出来的配置类进行一次条件过滤。然后放入这个数组。
- 8: 放数组，ioc容器会自动把数组中加载配置类将其初始化到ioc容器中。。。

自动装配的原理 - @EnableAutoConfiguration

@EnableAutoConfiguration 它的原理是：@Import（配置类或者importselector接口的子类）

而自动装配真正的原理是：AutoConfigurationImportSelectors它实现了ImportSelector接口 。

- 实现ImportSelector接口以后，一定要覆盖selectImports方法。如下：

```
@Override
    public String[] selectImports(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return NO_IMPORTS;
        }
        AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(annotationMetadata);
        return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations());
    }
```

这个方法中，就告诉我们它会找到所有的项目的jar，并循环找到这些jar包包括META-INF/spring.factories的jar包，然后进行解析META-INF/spring.factories,把对应的配置类放入到数组中。

核心方法：getAutoConfigurationEntry(annotationMetadata);如下：

```
/**
     * Return the {@link AutoConfigurationEntry} based on the {@link AnnotationMetadata}
     * of the importing {@link Configuration @Configuration} class.
     * @param annotationMetadata the annotation metadata of the configuration class
     * @return the auto-configurations that should be imported
     */
    protected AutoConfigurationEntry getAutoConfigurationEntry(AnnotationMetadata annotationMetadata) {
        if (!isEnabled(annotationMetadata)) {
            return EMPTY_ENTRY;
        }
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        // 根据对应注解信息EnableAutoConfiguration去找到META-INF/spring.factories的配置类，放入到集合中。
```

```
13          List<String> configurations =
getCandidateConfigurations(annotationMetadata, attributes);
14          configurations = removeDuplicates(configurations);
15          Set<String> exclusions =
getExclusions(annotationMetadata, attributes);
16          checkExcludedClasses(configurations, exclusions);
17          configurations.removeAll(exclusions);
18          configurations =
getConfigurationClassFilter().filter(configurations);
19          fireAutoConfigurationImportEvents(configurations,
exclusions);
20          return new AutoConfigurationEntry(configurations,
exclusions);
21      }
```

核心方法如下：getCandidateConfigurations()

```
1  protected List<String>
   getCandidateConfigurations(AnnotationMetadata metadata,
   AnnotationAttributes attributes) {
2       // 根据对应注解信息EnableAutoConfiguration去找到META-
   INF/spring.factories的配置类，放入到集合中。
3          List<String> configurations =
   SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderF
   actoryClass(),
4                  getBeanClassLoader());
5          Assert.notEmpty(configurations, "No auto configuration
   classes found in META-INF/spring.factories. If you "
6                  + "are using a custom packaging, make sure that
   file is correct.");
7          return configurations;
8      }
```

核心方法：SpringFactoriesLoader.loadFactoryName()如下：

```
public static List<String> loadFactoryNames(Class<?>
factoryType, @Nullable ClassLoader classLoader) {
        ClassLoader classLoaderToUse = classLoader;
        if (classLoaderToUse == null) {
            classLoaderToUse =
SpringFactoriesLoader.class.getClassLoader();
        }
        String factoryTypeName = factoryType.getName();
        return
loadSpringFactories(classLoaderToUse).getOrDefault(factoryTypeN
ame, Collections.emptyList());
    }


```

核心方法如下：loadSpringFactories(classLoaderToUse)

```
private static Map<String, List<String>>
loadSpringFactories(ClassLoader classLoader) {
        //1: 创建一个容器：Map 这个一个缓存机制。好处就是：避免重复解析
       // 2: 为什么要用缓存呢？ loadSpringFactories会加载很多次，为了
提升性能解析一次就够了，后续全部从缓存中获取，从而提升效率和性能，比如：自动装
配，比如：spring启动调用，上下文监听初始化会调用等。
        Map<String, List<String>> result =
cache.get(classLoader);
        // 如果缓存存在数据之间返回
        if (result != null) {
            return result;
        }
        //  如果没有就创建一个容器，为什么是HashMap， 因为META-
INF/spring.factories文件是以key=value存在的。一般在底层用Map或者
Properties装载比较多，
        result = new HashMap<>();
        try {
            //  public static final String
FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";
            // 这里是根据类加载器classLoader会根据双亲模型把层层去把项目
中依赖jar包，jdkjar包，项目中target目录。找到以后遍历jar遍历target目录看
是否哪些jar或者目录存在META-INF/spring.factories文件，如果存在全部匹配
出来。
            Enumeration<URL> urls =
classLoader.getResources("META-INF/spring.factories");
            // 把找到所有的META-INF/spring.factories文件，开始进行循
环解析
            while (urls.hasMoreElements()) {
                // 找到META-INF/spring.factories文件
                URL url = urls.nextElement();
```

```
                // 开始进行资源定位
                UrlResource resource = new UrlResource(url);
                // 开始解析放入到 Properties
                Properties properties =
 PropertiesLoaderUtils.loadProperties(resource);
                // 在把Properties存放好的数据转换成map返回。
                for (Map.Entry<?, ?> entry :
properties.entrySet()) {
                    String factoryTypeName = ((String)
entry.getKey()).trim();
                    String[] factoryImplementationNames =

StringUtils.commaDelimitedListToStringArray((String)
entry.getValue());
                    for (String factoryImplementationName :
factoryImplementationNames) {
                        result.computeIfAbsent(factoryTypeName,
key -> new ArrayList<>())

.add(factoryImplementationName.trim());
                    }
                }
            }

            // Replace all lists with unmodifiable lists
containing unique elements
            result.replaceAll((factoryType, implementations) ->
implementations.stream().distinct()

.collect(Collectors.collectingAndThen(Collectors.toList(),
Collections::unmodifiableList)));
            cache.put(classLoader, result);
        }
        catch (IOException ex) {
            throw new IllegalArgumentException("Unable to load
factories from location [" +
                    FACTORIES_RESOURCE_LOCATION + "]", ex);
        }
        // 返回
        return result;
    }
```

loadSpringFactories 这个方法会把

- 核心文件解析：META-INF/spring.factories

  这个文件一般存在与：springboot官方的配置jar中，或者自定义的starter都会存在，比如：

官方的：spring-boot-autoconfigure-2.6.1.jar

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

# Environment Post Processors
org.springframework.boot.env.EnvironmentPostProcessor=\
org.springframework.boot.autoconfigure.integration.IntegrationPropertiesEnvironmentPostProcessor

# Auto Configuration Import Listeners
org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportAutoConfigurationImportListener

# Auto Configuration Import Filters
org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
org.springframework.boot.autoconfigure.condition.OnBeanCondition,\
org.springframework.boot.autoconfigure.condition.OnClassCondition,\
org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.context.LifecycleAutoConfiguration,\
org.springframework.boot.autoconfigure.context.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration,\
org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.elasticsearch.ReactiveElasticsearchRestClientAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.ldap.LdapRepositoriesAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jReactiveDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jReactiveRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.r2dbc.R2dbcDataAutoConfiguration,\
org.springframework.boot.autoconfigure.data.r2dbc.R2dbcRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfiguration,\
org.springframework.boot.autoconfigure.elasticsearch.ElasticsearchRestClientAutoConfiguration,\
org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfiguration,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,\
org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration,\
```

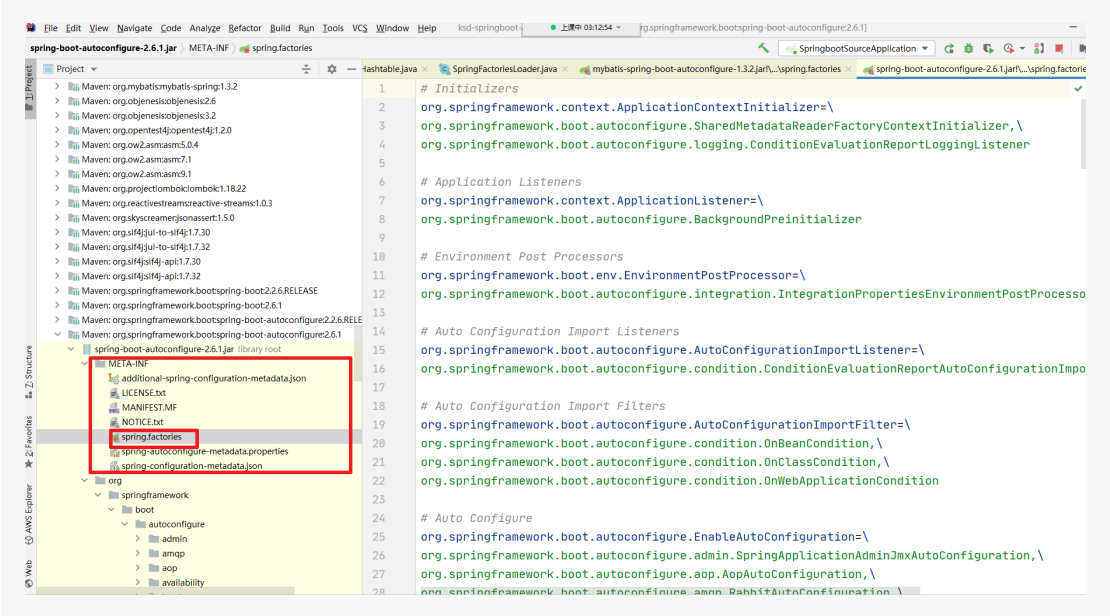
```
org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAutoConfiguration,\
org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\
org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\
org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
org.springframework.boot.autoconfigure.integration.IntegrationAutoConfiguration,\
org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration,\
org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
org.springframework.boot.autoconfigure.availability.ApplicationAvailabilityAutoConfiguration,\
org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
org.springframework.boot.autoconfigure.liquibase.LiquibaseAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
org.springframework.boot.autoconfigure.neo4j.Neo4jAutoConfiguration,\
org.springframework.boot.autoconfigure.netty.NettyAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration,\
org.springframework.boot.autoconfigure.r2dbc.R2dbcAutoConfiguration,\
org.springframework.boot.autoconfigure.r2dbc.R2dbcTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketRequesterAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketServerAutoConfiguration,\
org.springframework.boot.autoconfigure.rsocket.RSocketStrategiesAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceAutoConfiguration,\
org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetailsServiceAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.security.rsocket.RSocketSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.saml2.Saml2RelyingPartyAutoConfiguration,\
org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.client.reactive.ReactiveOAuth2ClientAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAuth2ResourceServerAutoConfiguration,\
org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.ReactiveOAuth2ResourceServerAutoConfiguration,\
org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
org.springframework.boot.autoconfigure.sql.init.SqlInitializationAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration,\
org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration,\
org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration,\
org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration,\
org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactoryCustomizerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveMultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.WebSessionIdResolverAutoConfiguration,\
```

```
org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.ClientHttpConnectorAutoConfiguration,\
org.springframework.boot.autoconfigure.web.reactive.function.client.WebClientAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactiveAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessagingAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfiguration,\
org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplateAutoConfiguration

# Failure analyzers
org.springframework.boot.diagnostics.FailureAnalyzer=\
org.springframework.boot.autoconfigure.data.redis.RedisUrlSyntaxFailureAnalyzer,\
org.springframework.boot.autoconfigure.diagnostics.analyzer.NoSuchBeanDefinitionFailureAnalyzer,\
org.springframework.boot.autoconfigure.flyway.FlywayMigrationScriptMissingFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.DataSourceBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.jdbc.HikariDriverConfigurationFailureAnalyzer,\
org.springframework.boot.autoconfigure.jooq.NoDslContextBeanFailureAnalyzer,\
org.springframework.boot.autoconfigure.r2dbc.ConnectionFactoryBeanCreationFailureAnalyzer,\
org.springframework.boot.autoconfigure.r2dbc.MissingR2dbcPoolDependencyFailureAnalyzer,\
```

```
org.springframework.boot.autoconfigure.r2dbc.MultipleConnectionPoolConfigurationsFailureAnalzyer,\
org.springframework.boot.autoconfigure.r2dbc.NoConnectionFactoryBeanFailureAnalyzer,\
org.springframework.boot.autoconfigure.session.NonUniqueSessionRepositoryFailureAnalyzer

# Template availability providers
org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvider=\
org.springframework.boot.autoconfigure.freemarker.FreeMarkerTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.mustache.MustacheTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafTemplateAvailabilityProvider,\
org.springframework.boot.autoconfigure.web.servlet.JspTemplateAvailabilityProvider

# DataSource initializer detectors
org.springframework.boot.sql.init.dependency.DatabaseInitializerDetector=\
org.springframework.boot.autoconfigure.flyway.FlywayMigrationInitializerDatabaseInitializerDetector

# Depends on database initialization detectors
org.springframework.boot.sql.init.dependency.DependsOnDatabaseInitializationDetector=\
org.springframework.boot.autoconfigure.batch.JobRepositoryDependsOnDatabaseInitializationDetector,\
org.springframework.boot.autoconfigure.quartz.SchedulerDependsOnDatabaseInitializationDetector,\
org.springframework.boot.autoconfigure.session.JdbcIndexedSessionRepositoryDependsOnDatabaseInitializationDetector

```



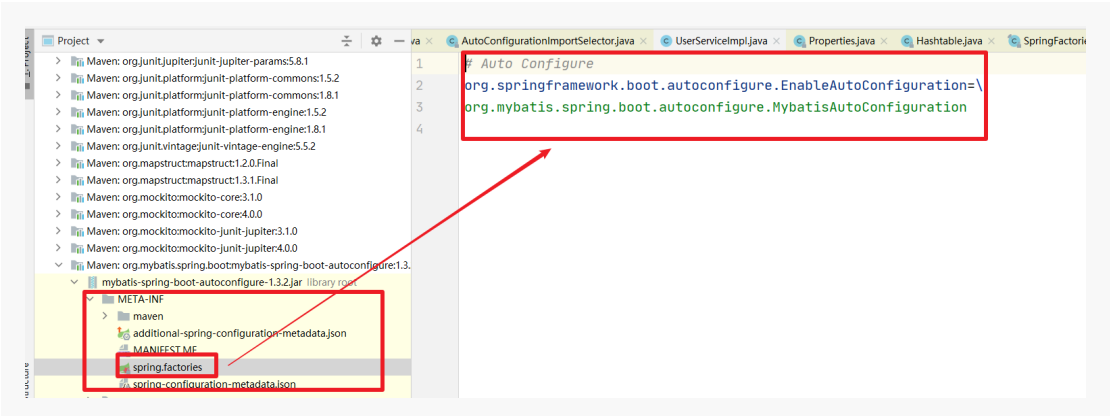
myabtis的starter如下：

mybatis-spring-boot-autoconfigure-1.3.2.jar

通过源码分析得知。META-INF/spring.factories 文件的内容很多，但是我们只要需要配置类。以及配置类中满足条件怎么匹配出来呢？都是问题，怎么解决呢？答案：条件注解

- @EnableAutoConfiguration 过滤一次
- @Condition需要的配置类用条件注解来过滤

整个流程图如下：



解析以后的Map如下：

```
map.put("org.springframework.context.ApplicationContextInitializer",
["org.springframework.boot.autoconfigure.SharedMetadataReaderFactory
ContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationRe
portLoggingListener"])
```

```
3
4  map.put("org.springframework.context.ApplicationListener",
   ["org.springframework.boot.autoconfigure.BackgroundPreinitializer"])
5
6
7
8  map.put("org.springframework.boot.autoconfigure.EnableAutoConfigurat
   ion",
   ["org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
9  org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJ
   mxAutoConfiguration,\
10 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
11 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,
   \
12 org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,
   \
13 org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,
   \
14 org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfig
   uration,\"])
```

DEBUG跟踪：

查看：06、AutoConfigurationImportSelector到底怎么初始化.md 文件

问题1：selectImports方法为什么不进？

因为自动装配为了实现过滤，弄了一个委托接口DeferredImportSelector接口。那么原来的接口的ImportSelector接口方法 selectImports()是一个历史代码，是不会在使用和不会在生效了。

问题2：谁来触发AutoConfigurationImportSelector的？selectImports方法？

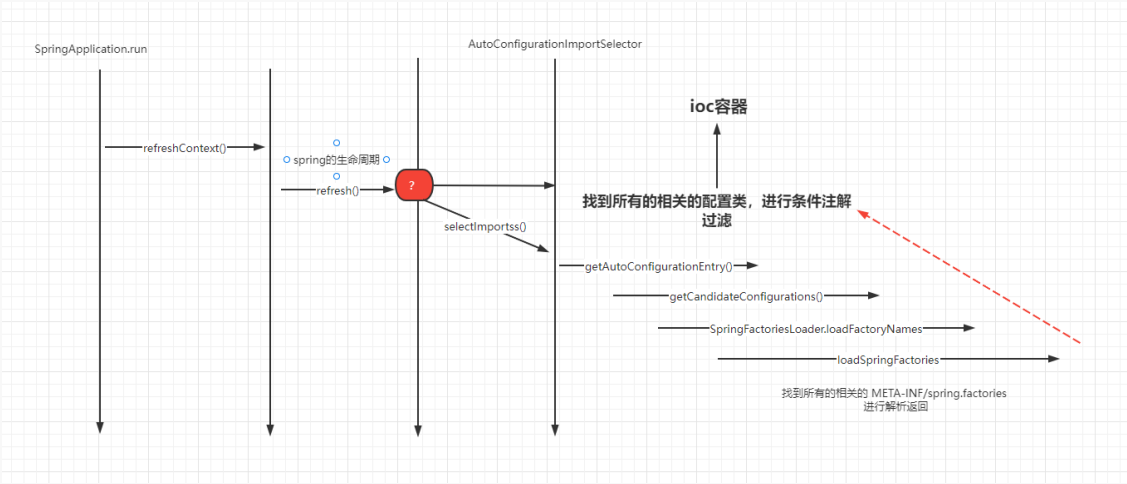
答案：spring生命周期的refresh()的invokeBeanFactoryPostProcessors()方法会来触发。

问题3：spring生命周期的refresh()又谁来掉的呢？

答案：你启动主类的时候来调用如下：

```java
@SpringBootApplication
public class SpringbootSelectImportsApplication {
    public static void main(String[] args) {
        // 这行代码会去启动ring生命周期的refresh()方法。

SpringApplication.run(SpringbootSelectImportsApplication.class,
args);
    }
}
```

整个流程图如下：



spring框架在扫包+注解这个阶段为什么这个不能完全取代XML呢？

1：那个时候配置类虽然出来来，但是还不成熟，因为第三技术比如 myabtis,日志，定期任务，事务，redis这些项目依赖组件还依赖传统spring生命周期得接口钩子得函数得方式。

2：扫包+注解：包名是固定，包名也不统一。如果要依赖第三方技术就必须每个都要扫，而且没办法完全抛弃xml。

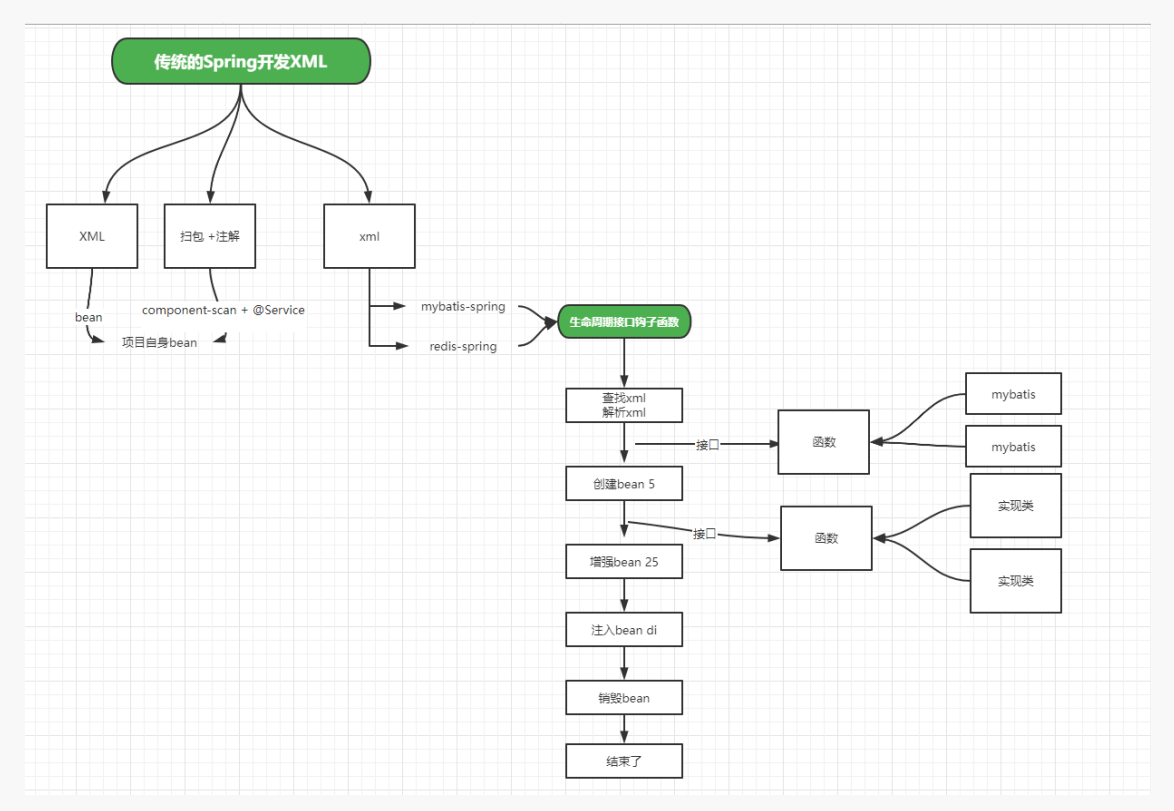
3：后续spring为了推动零xml

> 直到配置类的出现才得以全部解放，但是*spring*并没有延续这种和传统耦合开发方式结合，而且重新开了一个项目分支，*springboot*，将其配置类进行发扬光大。后续得组件*starter*提供基础。

SpringBean的生命周期接口钩子函数

是指spring框架提供了一系列的接口，让其一些第三方的技术进行融合和扩展。就是组件和扩展使用的。

SpringBean的生命周期 指的就是：一个bean从定义到存放到ioc管理，依赖注入，到销毁bean所有的过程。



@SpringBootApplication的简单认识

- @SpringBootConfiguration : 覆盖第三方的配置类，内部的starter全部是用配置类来完成的定义的，就是因为：配置类可以突破包的限制，
- @EnableAutoConfiguration： 用于自动装配第三方的配置类 + @Bean 的机制。比如 mybatis 官方提供的spring的starter

- @ComponentScan : 扫包+注解（@Service ， @Controller,@Component等），用于加载项目自身的service,mapper,dao,web,controller的。

一句话：

- @SpringBootConfiguration 这个是标准和扩展的规范，突破包的限制。(如果你看到官方不爽开源考虑覆盖，或者官方没有提供自研start)
- @ComponentScan 加载自己的bean
- @EnableAutoConfiguration 加载starter官方的自定义starterbean .加载别人的bean

15、条件注解@Condition

@Conditional注解

@Conditional从Spring4开始引入，用于条件性启动或者禁用@Configuration类或者@Bean方法。Starter配置的一些Bean可能需要修改，比如：默认数据源是HikariDataSource换成Druid数据源，那么默认的数据库HikariDataSource对应的Bean就不能在配置了，否则就会存在两个数据源，因而某些Bean是否需要注册到Spring容器是有条件的。SpringBoot使用@Conditional来完成Bean的条件注册。接下来用一些例子来说明：

需求

根据当前操作系统返回列举文件夹的命令：

- Windows -- dir
- Linux -- ls

01、新建一个maven项目 spring-boot-conditional-20

02、定义接口

```
package com.conditional.service;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:19
 */
public class LinuxListService implements ListService {

    @Override
    public String showCommand() {
        return "ls";
    }
}

```

window服务

```
package com.conditional.service;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:19
 */
public class WindowListService implements ListService {

    @Override
    public String showCommand() {
        return "dir";
    }
}

```

linux服务

```
package com.conditional.service;
```

```
2
3  /**
4   * @author 飞哥
5   * @Title: 学相伴出品
6   * @Description: 我们有一个学习网站：https://www.kuangstudy.com
7   * @date 2021/12/28 17:19
8   */
9  public class LinuxListService implements ListService {
10
11     @Override
12     public String showCommand() {
13         return "ls";
14     }
15 }
16
```

03、定义Controller

```
1  package com.conditional.controller;
2
3  import com.conditional.service.ListService;
4  import org.springframework.beans.factory.annotation.Autowired;
5  import org.springframework.web.bind.annotation.RestController;
6
7  /**
8   * @author 飞哥
9   * @Title: 学相伴出品
10  * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
11  * 记得关注和三连哦!
12  * @Description: 我们有一个学习网站：https://www.kuangstudy.com
13  * @date 2021/12/28 17:21
14  */
15 @RestController
16 public class ListController {
17
18     @Autowired
19     private ListService listService;
20 }
21
```

04、定义配置类

```java
package com.conditional.config;

import com.conditional.service.LinuxListService;
import com.conditional.service.ListService;
import com.conditional.service.WindowListService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:22
 */
@Configuration
public class ApplicationConfiguration {

    @Bean
    public ListService windowListService() {
        return new WindowListService();
    }

    @Bean
    public ListService linuxListService() {
        return new LinuxListService();
    }
}

```

05、定义启动类

```java
package com.conditional;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

import java.util.Arrays;

@SpringBootApplication
public class SpringBootConditional2OApplication {
```

```
11
12      public static void main(String[] args) {
13          ConfigurableApplicationContext applicationContext =
14
 SpringApplication.run(SpringBootConditional20Application.class,
args);
15
16          // 打印所有的bean，以便于测试
17          String[] beanDefinitionNames =
applicationContext.getBeanDefinitionNames();
18
 Arrays.stream(beanDefinitionNames).forEach(System.out::println);
19      }
20
21  }
22
```

06、测试

点击运行启动类，控制台输出，如下所示，以为你有两个实现类满足条件，Spring无法判断注入那个实现类对象给接口。

```
"C:\Program Files\Java\jdk1.8.0_221\bin\java.exe" -
XX:TieredStopAtLevel=1 -noverify -Dspring.output.ansi.enabled=always
-Dcom.sun.management.jmxremote -Dspring.jmx.enabled=true -
Dspring.liveBeansView.mbeanDomain -
Dspring.application.admin.enabled=true "-javaagent:C:\Program
Files\JetBrains\IntelliJ IDEA
2020.2.1\lib\idea_rt.jar=13809:C:\Program Files\JetBrains\IntelliJ
IDEA 2020.2.1\bin" -Dfile.encoding=UTF-8 -classpath "C:\Program
Files\Java\jdk1.8.0_221\jre\lib\charsets.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\deploy.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\access-bridge-64.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\cldrdata.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\dnsns.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\jaccess.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\jfxrt.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\localedata.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\nashorn.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunec.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunjce_provider.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunmscapi.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\sunpkcs11.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\ext\zipfs.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\javaws.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jce.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jfr.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jfxswt.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\jsse.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\management-agent.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\plugin.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\resources.jar;C:\Program
Files\Java\jdk1.8.0_221\jre\lib\rt.jar;C:\yykk\旅游项目实战开发\学相伴旅
游项目实战\07、SpringBoot入门&深入&分析和学习\13、SpringBoot的远离分析
\spring-boot-conditional-
20\target\classes;C:\yykk\respository\org\springframework\boot\sprin
g-boot-starter-web\2.6.2\spring-boot-starter-web-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter\2.6.2\spring-boot-starter-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-
boot\2.6.2\spring-boot-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
autoconfigure\2.6.2\spring-boot-autoconfigure-
2.6.2.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter-logging\2.6.2\spring-boot-starter-logging-
2.6.2.jar;C:\yykk\respository\ch\qos\logback\logback-
classic\1.2.9\logback-classic-
1.2.9.jar;C:\yykk\respository\ch\qos\logback\logback-
core\1.2.9\logback-core-
```

```
1.2.9.jar;C:\yykk\respository\org\apache\logging\log4j\log4j-to-
slf4j\2.17.0\log4j-to-slf4j-
2.17.0.jar;C:\yykk\respository\org\apache\logging\log4j\log4j-
api\2.17.0\log4j-api-2.17.0.jar;C:\yykk\respository\org\slf4j\jul-
to-slf4j\1.7.32\jul-to-slf4j-
1.7.32.jar;C:\yykk\respository\jakarta\annotation\jakarta.annotation
-api\1.3.5\jakarta.annotation-api-
1.3.5.jar;C:\yykk\respository\org\yaml\snakeyaml\1.29\snakeyaml-
1.29.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter-json\2.6.2\spring-boot-starter-json-
2.6.2.jar;C:\yykk\respository\com\fasterxml\jackson\core\jackson-
databind\2.13.1\jackson-databind-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\core\jackson-
annotations\2.13.1\jackson-annotations-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\core\jackson-
core\2.13.1\jackson-core-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\datatype\jackso
n-datatype-jdk8\2.13.1\jackson-datatype-jdk8-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\datatype\jackso
n-datatype-jsr310\2.13.1\jackson-datatype-jsr310-
2.13.1.jar;C:\yykk\respository\com\fasterxml\jackson\module\jackson-
module-parameter-names\2.13.1\jackson-module-parameter-names-
2.13.1.jar;C:\yykk\respository\org\springframework\boot\spring-boot-
starter-tomcat\2.6.2\spring-boot-starter-tomcat-
2.6.2.jar;C:\yykk\respository\org\apache\tomcat\embed\tomcat-embed-
core\9.0.56\tomcat-embed-core-
9.0.56.jar;C:\yykk\respository\org\apache\tomcat\embed\tomcat-embed-
el\9.0.56\tomcat-embed-el-
9.0.56.jar;C:\yykk\respository\org\apache\tomcat\embed\tomcat-embed-
websocket\9.0.56\tomcat-embed-websocket-
9.0.56.jar;C:\yykk\respository\org\springframework\spring-
web\5.3.14\spring-web-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
beans\5.3.14\spring-beans-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
webmvc\5.3.14\spring-webmvc-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
aop\5.3.14\spring-aop-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
context\5.3.14\spring-context-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
expression\5.3.14\spring-expression-
5.3.14.jar;C:\yykk\respository\org\slf4j\slf4j-api\1.7.32\slf4j-api-
1.7.32.jar;C:\yykk\respository\org\springframework\spring-
core\5.3.14\spring-core-
5.3.14.jar;C:\yykk\respository\org\springframework\spring-
```

```
jcl\5.3.14\spring-jcl-5.3.14.jar"
com.conditional.SpringBootConditional20Application

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::                (v2.6.2)

2021-12-28 17:24:56.917  INFO 9128 --- [           main] c.c.SpringBootConditional20Application   : Starting SpringBootConditional20Application using Java 1.8.0_221 on DESKTOP-27SNMQ8 with PID 9128 (C:\yykk\旅游项目实战开发\学相伴旅游项目实战\07、SpringBoot入门&深入&分析和学习\13、SpringBoot的远离分析\spring-boot-conditional-20\target\classes started by 86150 in C:\yykk\旅游项目实战开发\学相伴旅游项目实战\07、SpringBoot入门&深入&分析和学习\13、SpringBoot的远离分析\spring-boot-conditional-20)
2021-12-28 17:24:56.921  INFO 9128 --- [           main] c.c.SpringBootConditional20Application   : No active profile set, falling back to default profiles: default
2021-12-28 17:24:58.001  INFO 9128 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8080 (http)
2021-12-28 17:24:58.014  INFO 9128 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2021-12-28 17:24:58.014  INFO 9128 --- [           main] org.apache.catalina.core.StandardEngine  : Starting Servlet engine: [Apache Tomcat/9.0.56]
2021-12-28 17:24:58.144  INFO 9128 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/]       : Initializing Spring embedded WebApplicationContext
2021-12-28 17:24:58.145  INFO 9128 --- [           main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 1170 ms
2021-12-28 17:24:58.208  WARN 9128 --- [           main] ConfigServletWebServerApplicationContext : Exception encountered during context initialization - cancelling refresh attempt: org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'listController': Unsatisfied dependency expressed through field 'listService'; nested exception is org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type 'com.conditional.service.ListService' available: expected single matching bean but found 2: windowListService,linuxListService
```

```
2021-12-28 17:24:58.211  INFO 9128 --- [           main] o.apache.catalina.core.StandardService   : Stopping service [Tomcat]
2021-12-28 17:24:58.227  INFO 9128 --- [           main] ConditionEvaluationReportLoggingListener :

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2021-12-28 17:24:58.257 ERROR 9128 --- [           main] o.s.b.d.LoggingFailureAnalysisReporter   :

***************************
APPLICATION FAILED TO START
***************************

Description:

Field listService in com.conditional.controller.ListController required a single bean, but 2 were found:
    - windowListService: defined by method 'windowListService' in class path resource [com/conditional/config/ApplicationConfiguration.class]
    - linuxListService: defined by method 'linuxListService' in class path resource [com/conditional/config/ApplicationConfiguration.class]


Action:

Consider marking one of the beans as @Primary, updating the consumer to accept multiple beans, or using @Qualifier to identify the bean that should be consumed


Process finished with exit code 1

```

07、Window下的条件类

定义类WindowConditional实现接口Conditional，如果是在window下就返回true，否则返回false.

```java
package com.conditional.conditional;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
```

```
import org.springframework.core.type.AnnotatedTypeMetadata;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:29
 */
public class WindowsConditional implements Condition {

    @Override
    public boolean matches(ConditionContext context,
AnnotatedTypeMetadata metadata) {
        return
context.getEnvironment().getProperty("os.name").toLowerCase().contai
ns("windows");
    }
}

```

定义类LinuxConditional实现接口Conditional，如果是在linux下就返回true，否则返回false.

```
package com.conditional.conditional;

import org.springframework.context.annotation.Condition;
import org.springframework.context.annotation.ConditionContext;
import org.springframework.core.type.AnnotatedTypeMetadata;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:29
 */
public class LinuxConditional implements Condition {

    @Override
    public boolean matches(ConditionContext context,
AnnotatedTypeMetadata metadata) {
        return
context.getEnvironment().getProperty("os.name").toLowerCase().contai
ns("linux");
```

```
    }
}

```

08、修改配置类

```
package com.conditional.config;

import com.conditional.conditional.LinuxConditional;
import com.conditional.conditional.WindowsConditional;
import com.conditional.service.LinuxListService;
import com.conditional.service.ListService;
import com.conditional.service.WindowListService;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:22
 */
@Configuration
public class ApplicationConfiguration {

    @Bean
    @Conditional(WindowsConditional.class)
    public ListService windowListService() {
        return new WindowListService();
    }

    @Bean
    @Conditional(LinuxConditional.class)
    public ListService linuxListService() {
        return new LinuxListService();
    }
}

```

09测试

运行启动类，项目正常启动，由于项目是在windows系统下。所以WindowsConditional条件满足，所以会把WindowListService注册到spring的ioc容器中，自然结果就是：dir

10、测试添加在类上

修改ApplicationConfiguration，把注解条件增加在配置类上，为了验证@Conditional的方法和类哪个优先，让类返回false.

```
package com.conditional.config;

import com.conditional.conditional.LinuxConditional;
import com.conditional.conditional.WindowsConditional;
import com.conditional.service.LinuxListService;
import com.conditional.service.ListService;
import com.conditional.service.WindowListService;
import
org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Conditional;
import org.springframework.context.annotation.Configuration;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦！
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 17:22
 */
@Configuration
@Conditional(LinuxConditional.class)
public class ApplicationConfiguration {

    @Bean
    @Conditional(WindowsConditional.class)
    public ListService windowListService() {
        return new WindowListService();
    }

    @Bean
    @Conditional(LinuxConditional.class)
    public ListService linuxListService() {
        return new LinuxListService();
```

```
35      }
36  }
37
```

很清晰的看到项目启动失败了。因为你当前配置类必须在Linux环境下才会加载，而当前环境是window很明显匹配不上，因此整个配置类都不再处理。

常见的**Conditional**注解

只用一个注解就好，不要自己再来实现Condtion接口，Spring框架提供了一系列相关的注解，如下表

注解	说明
`@ConditionalOnSingleCandidate`	当给定类型的bean存在并且指定为Primary的给定类型存在时,返回true
`@ConditionalOnMissingBean`	当给定的类型、类名、注解、昵称在beanFactory中不存在时返回true.各类型间是or的关系
`@ConditionalOnBean`	与上面相反，要求bean存在
`@ConditionalOnMissingClass`	当给定的类名在类路径上不存在时返回true,各类型间是and的关系
`@ConditionalOnClass`	与上面相反，要求类存在
`@ConditionalOnCloudPlatform`	当所配置的CloudPlatform为激活时返回true
`@ConditionalOnExpression`	spel表达式执行为true
`@ConditionalOnJava`	运行时的java版本号是否包含给定的版本号.如果包含,返回匹配,否则,返回不匹配
`@ConditionalOnProperty`	要求配置属性匹配条件
`@ConditionalOnJndi`	给定的jndi的Location 必须存在一个.否则,返回不匹配
`@ConditionalOnNotWebApplication`	web环境不存在时
`@ConditionalOnWebApplication`	web环境存在时
`@ConditionalOnResource`	要求制定的资源存在

例子	说明

例子	说明
@ConditionalOnBean(javax.sql.DataSource.class)	Spring容器或者所有父容器中要存在至少一个javax.sql.DataSource类的实
@ConditionalOnClass({ Configuration.class,FreeMarkerConfigurationFactory.class })	类加载器中必须存在Configuration和FreeMarkerConfigurationFa这两个类
@ConditionalOnExpression(“’${server.host}’==’localhost’”)	server.host配置项的值需要是localhost
ConditionalOnJava(JavaVersion.EIGHT)	Java版本至少是8
@ConditionalOnMissingBean(value = ErrorController.class, search = SearchStrategy.CURRENT)	Spring当前容器中不存在ErrorController类型的bean
@ConditionalOnMissingClass(“GenericObjectPool”)	类加载器中不能存在GenericObjectPool这个类
@ConditionalOnNotWebApplication	必须在非Web应用下才会生效
@ConditionalOnProperty(prefix = “spring.aop”, name = “auto”, havingValue = “true”, matchIfMissing = true)	应用程序的环境中必须有spring.aop.auto这项配置，且的值是true或者环境中不存在spring.aop.auto配置(matchIfMissing为true)
@ConditionalOnResource(resources=”mybatis.xml”)	类加载路径中必须存在mybatis.xml文件
@ConditionalOnSingleCandidate(PlatformTransactionManager.class)	Spring当前或父容器中必须存在PlatformTransactionManager个类型的实例，且只有一个实
@ConditionalOnWebApplication	必须在Web应用下才会生效

16、SpringBoot启动流程

01、SpringApplication初始化方法

我们在SpringBoot启动类中调用SpringApplication的静态方法run。如下代码所示：

```
1 package com.conditional;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import org.springframework.context.ConfigurableApplicationContext;
```

```
6
7  import java.util.Arrays;
8
9  @SpringBootApplication
10 public class SpringBootConditional20Application {
11
12     public static void main(String[] args) {
13
     SpringApplication.run(SpringBootConditional20Application.class,
   args);
14     }
15 }
16
```

run方法如下：

```
1   public static ConfigurableApplicationContext run(Class<?>
   primarySource, String... args) {
2           return run(new Class[]{primarySource}, args);
3       }
```

它又调用了另外一个重载run方法，首先创建一个SpringApplication对象，然后调用非静态run方法

```
1  public static ConfigurableApplicationContext run(Class<?>[]
   primarySources, String[] args) {
2          return (new SpringApplication(primarySources)).run(args);
3      }
```

上面是一个run方法的重载。注意这个时候查看的时候要分为两段来分析：

- 构造函数部分
- run方法部分

构造函数部分

```
1  public SpringApplication(Class<?>... primarySources) {
2          this((ResourceLoader)null, primarySources);
3      }
```

```
1   public SpringApplication(ResourceLoader resourceLoader, Class<?>...
   primarySources) {
2           this.sources = new LinkedHashSet();
3           this.bannerMode = Mode.CONSOLE;
4           this.logStartupInfo = true;
```

```
        this.addCommandLineProperties = true;
        this.addConversionService = true;
        this.headless = true;
        this.registerShutdownHook = true;
        this.additionalProfiles = Collections.emptySet();
        this.isCustomEnvironment = false;
        this.lazyInitialization = false;
        this.applicationContextFactory = ApplicationContextFactory.DEFAULT;
        this.applicationStartup = ApplicationStartup.DEFAULT;
        this.resourceLoader = resourceLoader;
        Assert.notNull(primarySources, "PrimarySources must not be null");

        this.primarySources = new LinkedHashSet(Arrays.asList(primarySources));
        this.webApplicationType = WebApplicationType.deduceFromClasspath();
        this.bootstrapRegistryInitializers = new ArrayList(this.getSpringFactoriesInstances(BootstrapRegistryInitializer.class));

 this.setInitializers(this.getSpringFactoriesInstances(ApplicationContextInitializer.class));

 this.setListeners(this.getSpringFactoriesInstances(ApplicationListener.class));
        this.mainApplicationClass = this.deduceMainApplicationClass();
    }
```

该构造函数的作用是：

- 对primarySources初始化
- 根据jar包推断webApplicationType的类型，进而创建对应类型的ApplicationContext
- 初始化ApplicationContextInitializer列表
- 初始化ApplicationListener列表
- 推断包含main方法的主类。

对**primarySources**初始化

spring现在提倡了使用java配置来替代XML配置信息可以来自多个类，这里指定一个主配置类。也就是当前启动类

webApplicationType类型

```
//
// Source code recreated from a .class file by IntelliJ IDEA
// (powered by FernFlower decompiler)
//

package org.springframework.boot;

import org.springframework.util.ClassUtils;

public enum WebApplicationType {
    NONE,
    SERVLET,
    REACTIVE;

    private static final String[] SERVLET_INDICATOR_CLASSES = new String[]{"javax.servlet.Servlet", "org.springframework.web.context.ConfigurableWebApplicationContext"};
    private static final String WEBMVC_INDICATOR_CLASS = "org.springframework.web.servlet.DispatcherServlet";
    private static final String WEBFLUX_INDICATOR_CLASS = "org.springframework.web.reactive.DispatcherHandler";
    private static final String JERSEY_INDICATOR_CLASS = "org.glassfish.jersey.servlet.ServletContainer";
    private static final String SERVLET_APPLICATION_CONTEXT_CLASS = "org.springframework.web.context.WebApplicationContext";
    private static final String REACTIVE_APPLICATION_CONTEXT_CLASS = "org.springframework.boot.web.reactive.context.ReactiveWebApplicationContext";

    private WebApplicationType() {
    }

    static WebApplicationType deduceFromClasspath() {
        if (ClassUtils.isPresent("org.springframework.web.reactive.DispatcherHandler", (ClassLoader)null) && !ClassUtils.isPresent("org.springframework.web.servlet.DispatcherServlet", (ClassLoader)null) && !ClassUtils.isPresent("org.glassfish.jersey.servlet.ServletContainer", (ClassLoader)null)) {
            return REACTIVE;
        } else {
            String[] var0 = SERVLET_INDICATOR_CLASSES;
            int var1 = var0.length;
```

```
31
32              for(int var2 = 0; var2 < var1; ++var2) {
33                  String className = var0[var2];
34                  if (!ClassUtils.isPresent(className,
    (ClassLoader)null)) {
35                      return NONE;
36                  }
37              }
38
39              return SERVLET;
40          }
41      }
42
43      static WebApplicationType deduceFromApplicationContext(Class<?>
    applicationContextClass) {
44          if
    (isAssignable("org.springframework.web.context.WebApplicationContext
    ", applicationContextClass)) {
45              return SERVLET;
46          } else {
47              return
    isAssignable("org.springframework.boot.web.reactive.context.Reactive
    WebApplicationContext", applicationContextClass) ? REACTIVE : NONE;
48          }
49      }
50
51      private static boolean isAssignable(String target, Class<?>
    type) {
52          try {
53              return ClassUtils.resolveClassName(target,
    (ClassLoader)null).isAssignableFrom(type);
54          } catch (Throwable var3) {
55              return false;
56          }
57      }
58  }
59
```

方法deduceFromClasspath主要根据几个常量指定类是否在类路径上返回webApplicationType的类型：

- NONE, 不需要内嵌Web容器
- SERVLET,：一个基于Servlet的Web应用，应该启动内嵌的Servlet容器
- REACTIVE;：一个基于Reactive的Web应用，应该启动内嵌的Reactive容器

> 含义是：根据*jar*包推断该项目是*REACTIVE*还是*Servlet*。或者非*Web*项目的*None*。

初始化ApplicationContextInitializer和ApplicationListener

这两个初始化是读取自动配置类的原理一样，都是到jar的META-INF/spring.factories中读取它。它们分别读取的key不同如下：

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextInitializer,\
org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLoggingListener

# Application Listeners
org.springframework.context.ApplicationListener=\
org.springframework.boot.autoconfigure.BackgroundPreinitializer

```

> *从META-INF/spring.factories文件中获取ApplicationContextInitializer和实ApplicationListener例分别保存到initializers和listeners集合中。*

推断包含main方法的主类。

```
private Class<?> deduceMainApplicationClass() {
        try {
            StackTraceElement[] stackTrace = (new RuntimeException()).getStackTrace();
            StackTraceElement[] var2 = stackTrace;
            int var3 = stackTrace.length;

            for(int var4 = 0; var4 < var3; ++var4) {
                StackTraceElement stackTraceElement = var2[var4];
                if ("main".equals(stackTraceElement.getMethodName())) {
                    return Class.forName(stackTraceElement.getClassName());
                }
            }
        } catch (ClassNotFoundException var6) {
        }

```

```
        return null;
    }

```

以标准Java程序启动，从main方法开始执行，目前正在执行的方法通过调用栈可以找到main方法所在类，

> 根据运行时堆栈信息推断当前*main*方法的类名，然后保存到*mainApplicationClass*属性中。

02、run方法部分

当SpringApplication创建完毕后，就开始执行run方法了。如下所示：

```
public ConfigurableApplicationContext run(String... args) {
        // 开启一个计时器，用于记录运行的时间
        long startTime = System.nanoTime();
        // 创建一个顶级父加载器,主要用来做缓存使用
        DefaultBootstrapContext bootstrapContext = createBootstrapContext();
        // 配置应用上下文
        ConfigurableApplicationContext context = null;
        configureHeadlessProperty();
        //  获取所有的监听器
        SpringApplicationRunListeners listeners = getRunListeners(args);
        // 通知所有的SpringApplicationRunListener的子类对象启动starting()监听方法
        // 同时会调用environmentPrepared()方法去完成运行环境准备，
        listeners.starting(bootstrapContext, this.mainApplicationClass);
        try {
            ApplicationArguments applicationArguments = new DefaultApplicationArguments(args);
            ConfigurableEnvironment environment = prepareEnvironment(listeners, bootstrapContext, applicationArguments);
            configureIgnoreBeanInfo(environment);
            // 打印banner
            Banner printedBanner = printBanner(environment);
            // 根据WebapplicationType创建spring应用上下文。
            context = createApplicationContext();
            context.setApplicationStartup(this.applicationStartup);
```

```
            // 准备上下文，加载BeanDefintion对象
            prepareContext(bootstrapContext, context, environment, listeners, applicationArguments, printedBanner);
            // 刷新上下文，初始化springioc容器对象
            refreshContext(context);
            // 暂时空实现，没有任何逻辑，无需查看
            afterRefresh(context, applicationArguments);
            // 计时器结束，
            Duration timeTakenToStartup = Duration.ofNanos(System.nanoTime() - startTime);
            if (this.logStartupInfo) {
                new StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(), timeTakenToStartup);
            }
            // 启动监听器
            listeners.started(context, timeTakenToStartup);
            // 执行后置的ApplicationRunner接口和CommandLineRunner接口的子类方法
            callRunners(context, applicationArguments);
        }
        catch (Throwable ex) {
            handleRunFailure(context, ex, listeners);
            throw new IllegalStateException(ex);
        }
        try {
            Duration timeTakenToReady = Duration.ofNanos(System.nanoTime() - startTime);
            listeners.ready(context, timeTakenToReady);
        }
        catch (Throwable ex) {
            handleRunFailure(context, ex, null);
            throw new IllegalStateException(ex);
        }
        return context;
    }

```

该方法完成的工作如下：

- 启动一个秒表（StopWatch）来统计启动时间
- 通过SpringFactoriesLoader.loadFactoryName获取jar目录下的META-INF/spring.factories下配置的SpringApplicationRunListeners。该接口对SpringApplicaiton的run方法不同阶段进行监听。
- listeners.starting(bootstrapContext, this.mainApplicationClass); 调用了所有SpringApplicationRunListeners的starting()方法。

- ConfigurableEnvironment environment = this.prepareEnvironment(listeners, bootstrapContext, applicationArguments); 根据WebApplicationType类型准备对应类型的类型ConfigurableEnvironment，同时调用listeners.environmentPrepared(bootstrapContext, (ConfigurableEnvironment)environment);通知所有的SpringApplicationRunListener环境准备完毕。
- 打印Banner，如果spring.main.banner-mode=off。就不打印，如果值是console就打印banner到控制台。如果是log。就输出到日志，我们可以在resoruces目录下现金一个banner.txt来修改默认的banner.
- 根据WebApplicationType类型，创建一个类型的ApplicationContext对象。
- 准备上下文

```
1  private void prepareContext(DefaultBootstrapContext
   bootstrapContext, ConfigurableApplicationContext context,
   ConfigurableEnvironment environment, SpringApplicationRunListeners
   listeners, ApplicationArguments applicationArguments, Banner
   printedBanner) {
2          context.setEnvironment(environment);// 设置运行环境
3          this.postProcessApplicationContext(context);
   //applicationContext进行后置处理
4          this.applyInitializers(context);//调用所有的
   ApplicationContextInitializer的initialize方法
5          listeners.contextPrepared(context); // 通知所有监听器上下文准备
   完毕
6          bootstrapContext.close(context);
7          if (this.logStartupInfo) {
8              this.logStartupInfo(context.getParent() == null);
9              this.logStartupProfileInfo(context);
10         }
11
12         ConfigurableListableBeanFactory beanFactory =
   context.getBeanFactory();
13         beanFactory.registerSingleton("springApplicationArguments",
   applicationArguments);
14         if (printedBanner != null) {
15             beanFactory.registerSingleton("springBootBanner",
   printedBanner);
16         }
17
18         if (beanFactory instanceof
   AbstractAutowireCapableBeanFactory) {
19
   ((AbstractAutowireCapableBeanFactory)beanFactory).setAllowCircularRe
   ferences(this.allowCircularReferences);
20             if (beanFactory instanceof DefaultListableBeanFactory) {
```

```
21
    ((DefaultListableBeanFactory)beanFactory).setAllowBeanDefinitionOver
    riding(this.allowBeanDefinitionOverriding);
22              }
23          }
24
25          if (this.lazyInitialization) {
26              context.addBeanFactoryPostProcessor(new
    LazyInitializationBeanFactoryPostProcessor());
27          }
28
29          // 加载所有资源
30          Set<Object> sources = this.getAllSources();
31          Assert.notEmpty(sources, "Sources must not be empty");
32          // 注册所有bean到springioc容器
33          this.load(context, sources.toArray(new Object[0]));
34          // 通知监听器上下文加载完毕。
35          listeners.contextLoaded(context);
36      }
37
```

16、SpringBoot的内置web容器原理（了解）

---

springboot可以做到零配置，前面的@Configuration+@Bean 机制去掉了bean.xml。但是项目中还有一个配置文件web.xml这个去掉，这个去掉的功劳是依托于servlet本身的tomcat做javaapi的支持。也就告诉我们可以用java代码些一个tomcat能够去运行应用程序。

好处：就不在需要依托于外部的tomcat容器。

实现如下：

1：创建一个maven项目

2：导入tomcat的依赖

```xml
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-core</artifactId>
    <version>9.0.52</version>
</dependency>
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <version>9.0.52</version>
</dependency>
```

3：编写tomcat的启动类

```java
package com.kuangstudy.tomcat;

import org.apache.catalina.Context;
import org.apache.catalina.LifecycleException;
import org.apache.catalina.WebResourceRoot;
import org.apache.catalina.startup.Tomcat;
import org.apache.catalina.webresources.DirResourceSet;
import org.apache.catalina.webresources.StandardRoot;

import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.File;
import java.io.IOException;

/**
 * @author 飞哥
 * @Title: 学相伴出品
 * @Description: 飞哥B站地址：https://space.bilibili.com/490711252
 * 记得关注和三连哦!
 * @Description: 我们有一个学习网站：https://www.kuangstudy.com
 * @date 2021/12/28 18:41
 */
public class EmbedTomcatServer {

    public static void main(String[] args) throws LifecycleException
{
        // 启动Tomcat:
        Tomcat tomcat = new Tomcat();
        tomcat.setPort(Integer.getInteger("port", 9999));
        tomcat.getConnector();
        // 创建webapp:
```

```
        Context ctx = tomcat.addWebapp("", new File("src/main/webapp").getAbsolutePath());
        WebResourceRoot resources = new StandardRoot(ctx);
        resources.addPreResources(
                new DirResourceSet(resources, "/WEB-INF/classes", new File("target/classes").getAbsolutePath(), "/"));
        ctx.setResources(resources);
        tomcat.start();
        tomcat.getServer().await();
    }
}

```

4：定义一个servlet

```
package com.kuangstudy.servlet;

import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;
import java.io.PrintWriter;

@WebServlet(urlPatterns = "/hello")
public class HelloServlet extends HttpServlet {
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
        resp.setContentType("text/html");
        String name = req.getParameter("name");
        if (name == null) {
            name = "world";
        }
        PrintWriter pw = resp.getWriter();
        pw.write("<h1>Hello, " + name + "!</h1>");
        pw.flush();
    }
}
```

5：启动EmbedTomcatServer

6：访问http://localhost:9999/hello 如下

```
Hello, world!
```

jsp同理。

7：完整的结构如下：