

SpringBoot

1. SpringBoot应用回顾

1.1 约定优于配置

概念：约定优于配置(Convention over Configuration)，又称按约定编程，是一种软件设计规范。

本质上是对系统、类库或框架中一些东西假定一个大众化合理的默认值(缺省值)。

例如在模型中存在一个名为User的类，那么对应到数据库会存在一个名为user的表，此时无需做额外的配置，只有在偏离这个约定时才需要做相关的配置（例如你想将表名命名为t_user等非user时才需要写关于这个名字的配置）。

如果所用工具的约定与你的期待相符，便可省去配置；反之，你可以配置来达到你所期待的方式。

简单来说就是假如你所期待的配置与约定的配置一致，那么就可以不做任何配置，约定不符合期待时才需要对约定进行替换配置。

好处：大大减少了配置项

1.2 SpringBoot概念

1.2.1 什么是SpringBoot

spring官方的网站：<https://spring.io/>



翻译：通过Spring Boot，可以轻松地创建独立的，基于生产级别的基于Spring的应用程序，并且可以“运行”它们

其实Spring Boot 的设计是为了让你尽可能快的跑起来 Spring 应用程序并且尽可能减少你的配置文件。

以下内容来自百度百科

SpringBoot是由Pivotal团队在2013年开始研发、2014年4月发布第一个版本的全新开源的轻量级框架。它基于Spring4.0设计，不仅继承了Spring框架原有的优秀特性，而且还通过简化配置来进一步简化了Spring应用的整个搭建和开发过程。另外SpringBoot通过集成大量的框架使得依赖包的版本冲突，以及引用的不稳定性等问题得到了很好的解决。

1.2.2 SpringBoot主要特性

- 1、SpringBoot Starter: 他将常用的依赖分组进行了整合，将其合并到一个依赖中，这样就可以一次性添加到项目的Maven或Gradle构建中；
- 2、使编码变得简单，SpringBoot采用JavaConfig的方式对Spring进行配置，并且提供了大量的注解，极大的提高了工作效率。
- 3、自动配置：SpringBoot的自动配置特性利用了Spring对条件化配置的支持，合理地推测应用所需的bean并自动化配置他们；
- 4、使部署变得简单，SpringBoot内置了三种Servlet容器，Tomcat, Jetty,undertow.我们只需要一个Java的运行环境就可以跑SpringBoot的项目了，SpringBoot的项目可以打成一个jar包。

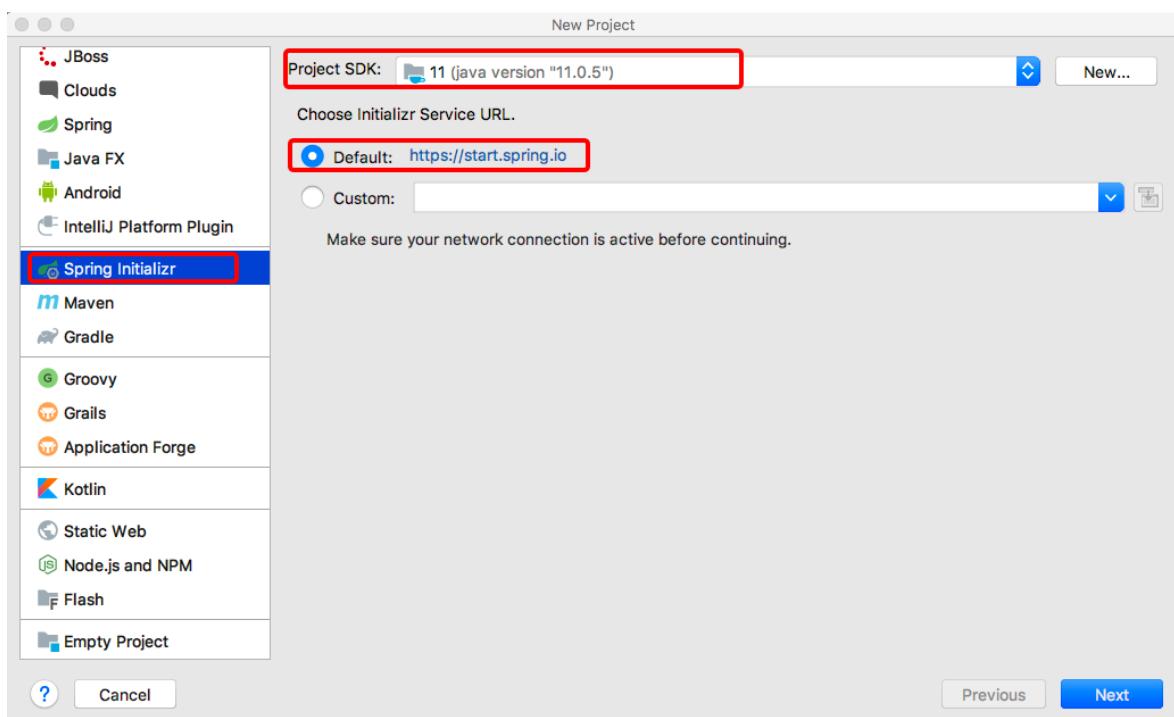
1.3 SpringBoot 案例实现

1.3.1 案例实现

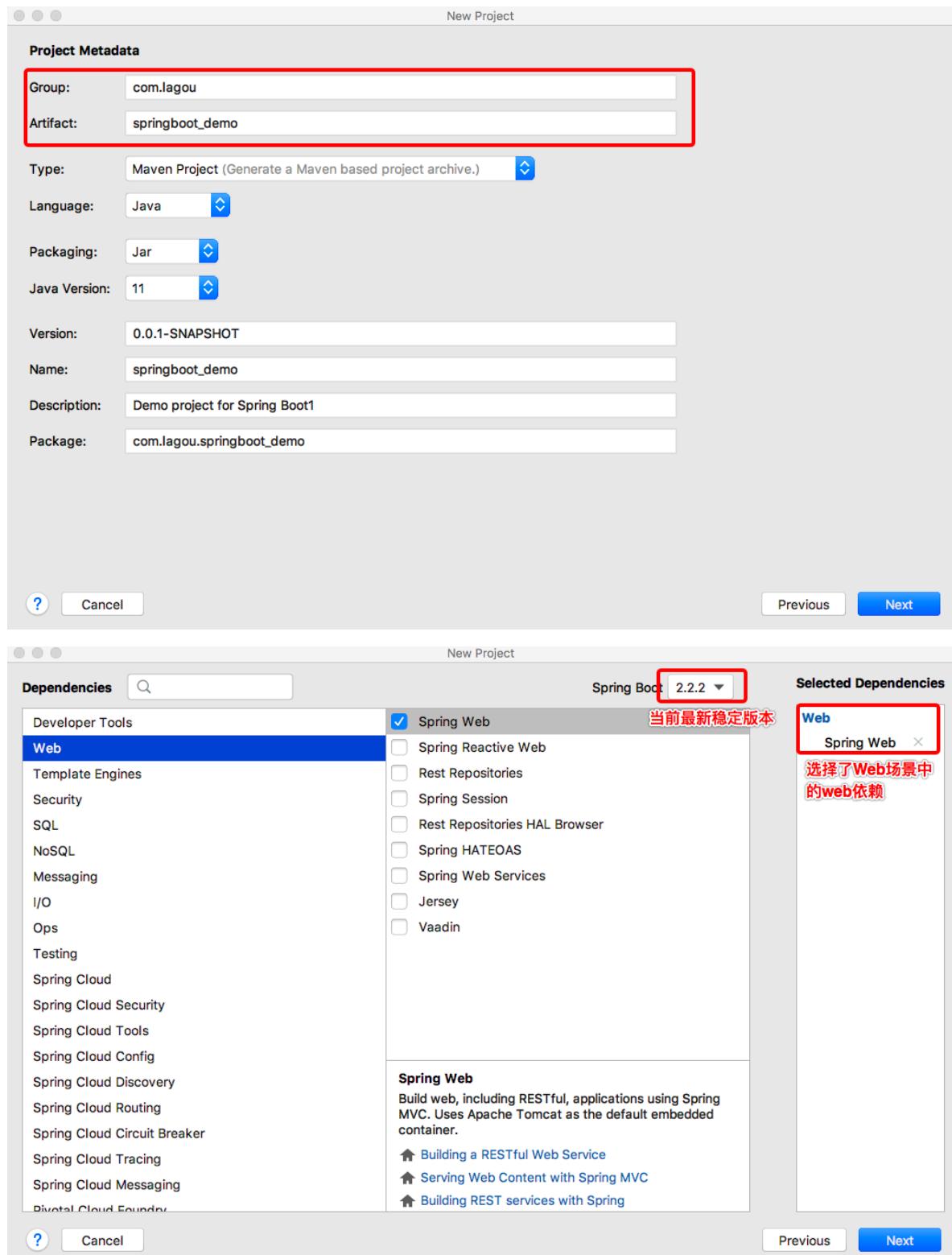
案例需求：使用Spring Initializr方式构建Spring Boot项目，并请求Controller中的目标方法，将返回值响应到页面

(1) 使用Spring Initializr方式构建Spring Boot项目

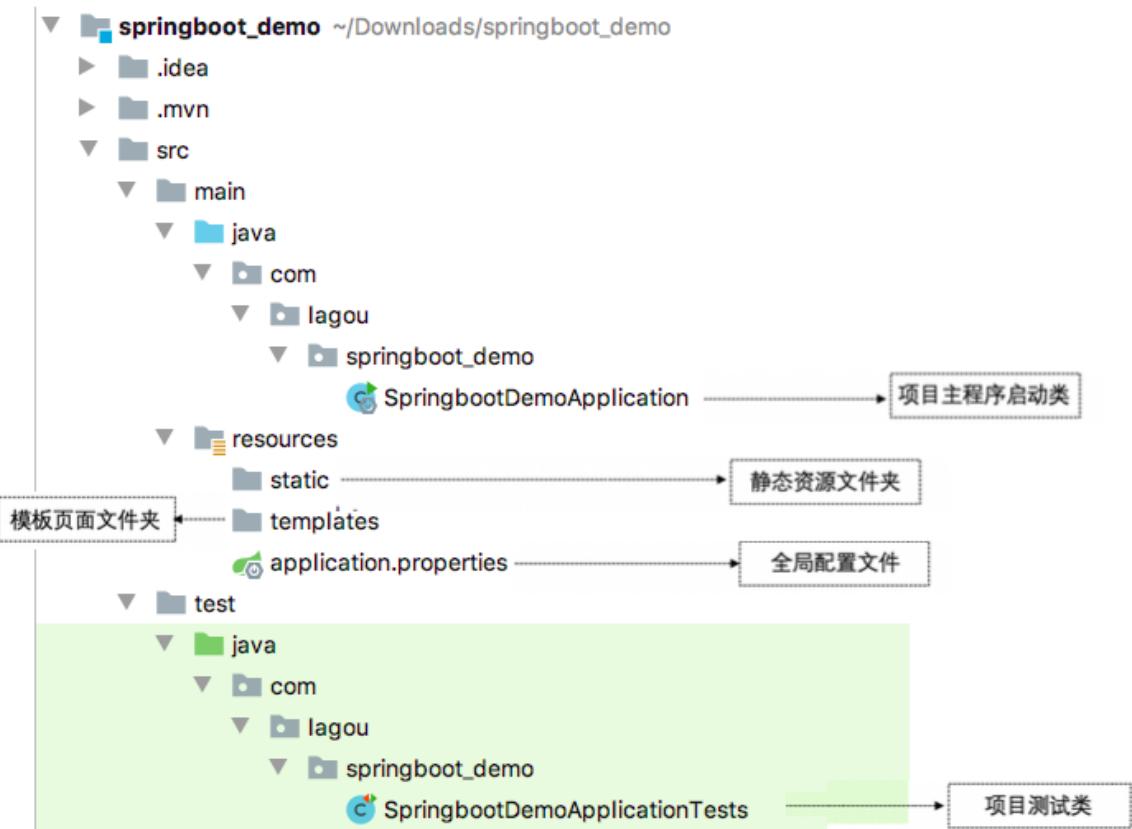
本质上说，Spring Initializr是一个Web应用，它提供了一个基本的项目结构，能够帮助我们快速构建一个基础的Spring Boot项目



Project SDK"用于设置创建项目使用的JDK版本,这里,使用之前初始化设置好的JDK版本即可;在"Choose Initializr Service URL (选择初始化服务地址)"下使用默认的初始化服务地址"<https://start.spring.io>"进行Spring Boot项目创建(注意使用快速方式创建Spring Boot项目时,所在主机须在联网状态下)



Spring Boot项目就创建好了。创建好的Spring Boot项目结构如图：



使用Spring Initializr方式构建的Spring Boot项目会默认生成项目启动类、存放前端静态资源和页面的文件夹、编写项目配置的配置文件以及进行项目单元测试的测试类

(2) 创建Controller

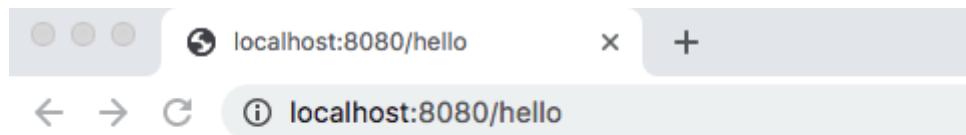
com.lagou包下创建名称为controller的包，在该包下创建一个请求处理控制类HelloController，并编写一个请求处理方法 (注意：将项目启动类SpringBootDemoApplication移动到com.lagou包下)

```
@RestController // 该注解为组合注解，等同于Spring中@Controller+@ResponseBody注解
public class DemoController {

    @RequestMapping("/demo")
    public String demo(){
        return "hello springBoot";
    }
}
```

(3) 运行项目

运行主程序启动类SpringbootDemoApplicat
ion，项目启动成功后，在控制台上会发现Spring Boot项目默认启动的端口号为8080，此时，可以在浏览器上访问“<http://localhost:8080/hello>”



页面输出的内容是“hello Spring Boot”，至此，构建Spring Boot项目就完成了

附：解决中文乱码：

解决方法一：

```
@RequestMapping(produces = "application/json; charset=utf-8")
```

解决方法二：

```
#设置响应为utf-8  
spring.http.encoding.force-response=true
```

1.3.2 疑问

- 1. starter是什么？我们如何去使用这些starter？
- 2. 为什么包扫描只会扫描核心启动类所在的包及其子包
- 3. 在springBoot启动的过程中，是如何完成自动装配的？
- 4. 内嵌Tomcat是如何被创建及启动的？
- 5. 使用了web场景对应的starter，springmvc是如何自动装配？

(保留疑问：在源码剖析部分，主要将以上疑问进行解答)

1.4 热部署

在开发项目过程中，当修改了某些代码后需要本地验证时，需要重启本地服务进行验证，启动这个项目，如果项目庞大的话还是需要较长时间的，spring开发团队为我们带来了一个插件：spring-boot-devtools，很好的解决了本地验证缓慢的问题。

1.4.1 热部署实现演示

1. 添加spring-boot-devtools热部署依赖启动器

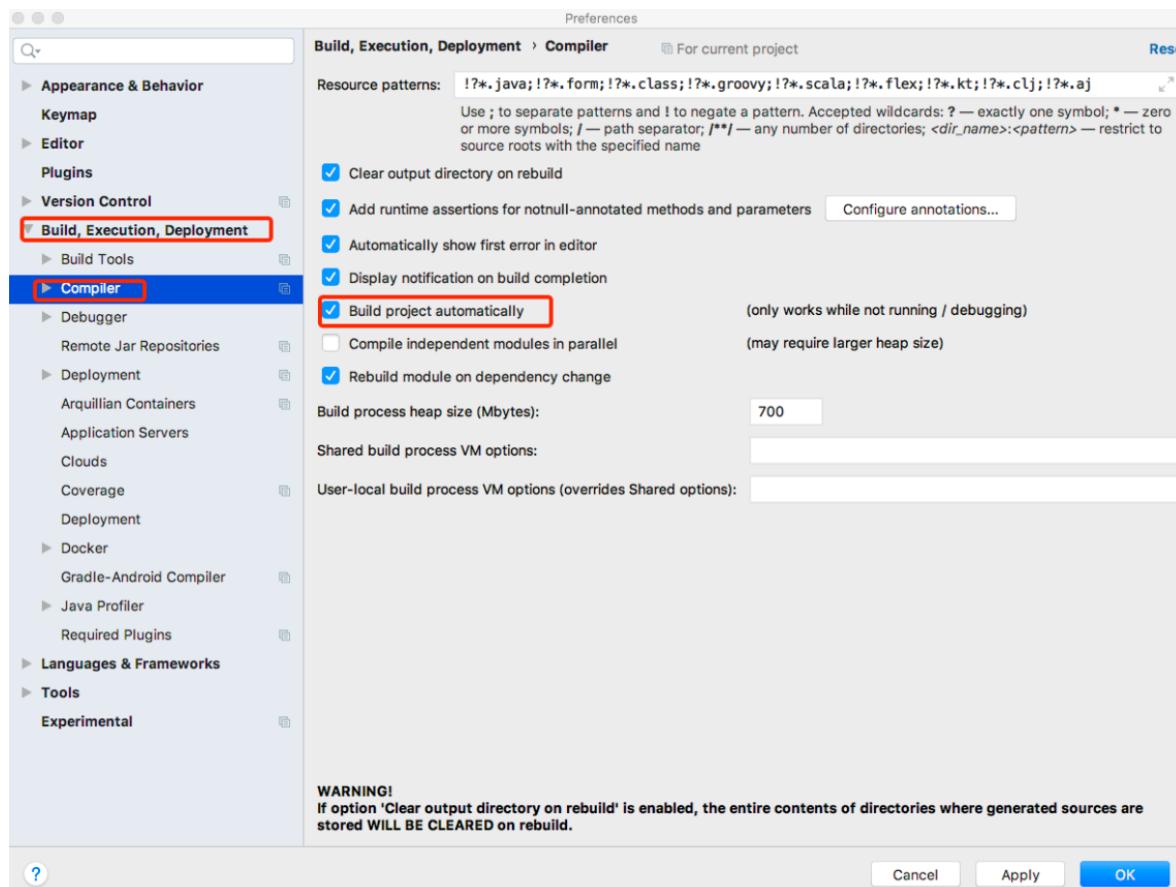
在Spring Boot项目进行热部署测试之前，需要先在项目的pom.xml文件中添加spring-boot-devtools热部署依赖启动器：

```
<!-- 引入热部署依赖 -->  
<dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-devtools</artifactId>  
</dependency>
```

由于使用的是IDEA开发工具，添加热部署依赖后可能没有任何效果，接下来还需要针对IDEA开发工具进行热部署相关的功能设置

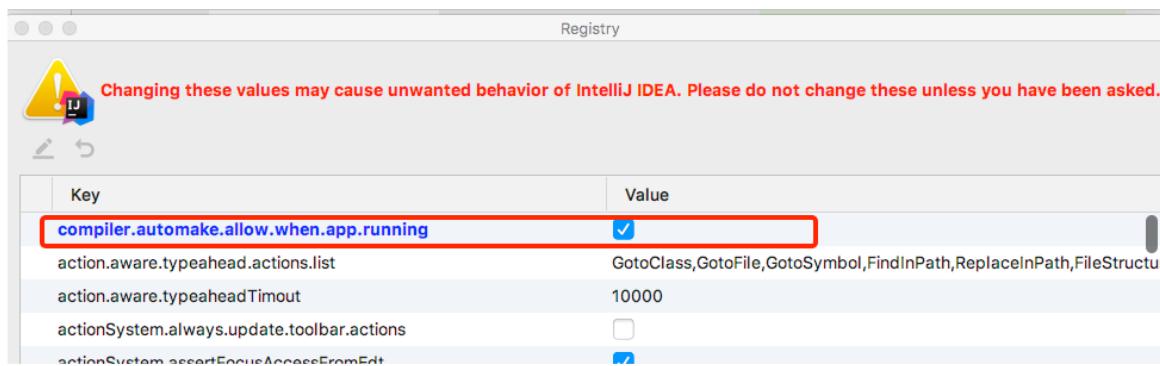
2. IDEA工具热部署设置

选择IDEA工具界面的【File】->【Settings】选项，打开Compiler面板设置页面



选择Build下的Compiler选项，在右侧勾选“Build project automatically”选项将项目设置为自动编译，单击【Apply】→【OK】按钮保存设置

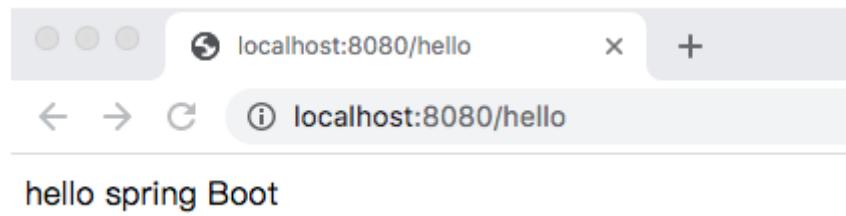
在项目任意页面中使用组合快捷键“Ctrl+Shift+Alt+/”打开Maintenance选项框，选中并打开Registry页面，具体如图1-17所示



列表中找到“compiler.automake.allow.when.app.running”，将该选项后的Value值勾选，用于指定IDEA工具在程序运行过程中自动编译，最后单击【Close】按钮完成设置

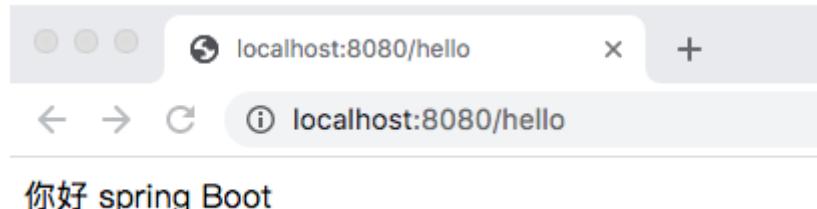
3. 热部署效果测试

启动chapter01 <http://localhost:8080/hello>



页面原始输出的内容是“hello Spring Boot”。

为了测试配置的热部署是否有效，接下来，在不关闭当前项目的情况下，将DemoController类中的请求处理方法hello()的返回值修改为“你好，Spring Boot”并保存，查看控制台信息会发现项目能够自动构建和编译，说明项目热部署生效



可以看出，浏览器输出了“你好，Spring Boot”，说明项目热部署配置成功

1.4.2 热部署原理分析

该原理其实很好说明，就是在编辑器上启动项目，然后改动相关的代码，然后编辑器自动触发编译替换掉历史的.class文件后，项目检测到有文件变更后会重启Spring-Boot项目。

可以看看官网的触发描述：

Triggering a restart

As DevTools monitors classpath resources, the only way to trigger a restart is to update the classpath. The way in which you cause the classpath to be updated depends on the IDE that you are using. In Eclipse, saving a modified file causes the classpath to be updated and triggers a restart. In IntelliJ IDEA, building the project (`Build → Build Project`) has the same effect.

可以看到，我们引入了插件后，插件会监控我们classpath的资源变化，当classpath有变化后，会触发重启

Restart vs Reload

The restart technology provided by Spring Boot works by using two classloaders. Classes that do not change (for example, those from third-party jars) are loaded into a *base* classloader. Classes that you are actively developing are loaded into a *restart* classloader. When the application is restarted, the *restart* classloader is thrown away and a new one is created. This approach means that application restarts are typically much faster than “cold starts”, since the *base* classloader is already available and populated.

If you find that restarts are not quick enough for your applications or you encounter classloading issues, you could consider reloading technologies such as [JRebel](#) from ZeroTurnaround. These work by rewriting classes as they are loaded to make them more amenable to reloading.



base-classloader

restartClassLoader

这里提到了，该插件重启快速的原因：这里对类加载采用了两种类加载器，对于第三方jar包采用base-classloader来加载，对于开发人员自己开发的代码则使用restartClassLoader来进行加载，这使得比停掉服务重启要快的多，因为使用插件只是重启开发人员编写的代码部分。

这边做个简单的验证：

```
@Component
public class Devtools implements InitializingBean {

    private static final Logger log = LoggerFactory.getLogger(Devtools.class);

    @Override
    public void afterPropertiesSet() throws Exception {
        log.info("guava-jar classLoader: " +
DispatcherServlet.class.getClassLoader().toString());
        log.info("Devtools ClassLoader: " +
this.getClass().getClassLoader().toString());
    }

}
```

这边先去除spring-boot-devtools插件，跑下工程：

```
[main] com.lagou.config.Devtools      : guava-jar classLoader: jdk.internal.loader.ClassLoaders$AppClassLoader@2437c6dc
[main] com.lagou.config.Devtools      : Devtools ClassLoader: jdk.internal.loader.ClassLoaders$AppClassLoader@2437c6dc
```

可以看到，DispatcherServlet (第三方jar包) 和Devtools(自己编写的类)使用的都是AppClassLoader 加载的。

我们现在加上插件，然后执行下代码：

```
restartedMain] com.lagou.config.Devtools      : guava-jar classLoader: jdk.internal.loader.ClassLoaders$AppClassLoader@2437c6dc
restartedMain] com.lagou.config.Devtools      : Devtools ClassLoader: org.springframework.boot.devtools.restart.classloader.RestartClassLoader@5aa68d7
```

发现第三方的jar包的类加载器确实是使用的系统的类加载器，而我们自己写的代码的类加载器为RestartClassLoader，并且每次重启，类加载器的实例都会改变。

上图为代码修改前后类文件的变更。

1.4.3 排除资源

某些资源在更改后不一定需要触发重新启动。例如，Thymeleaf模板可以就地编辑。默认情况下，改变资源 /META-INF/maven, /META-INF/resources, /resources, /static, /public, 或 /templates 不触发重新启动，但确会触发现场重装。如果要自定义这些排除项，则可以使用该 `spring.devtools.restart.exclude` 属性。例如，仅排除 /static, /public 您将设置以下属性：

```
spring.devtools.restart.exclude=static/**,public/**
```

1.5 全局配置文件

1.5.1 全局配置文件概述及优先级

全局配置文件能够对一些默认配置值进行修改及自定义配置。

Spring Boot 使用一个 `application.properties` 或者 `application.yaml` 的文件作为全局配置文件

`SpringApplication` loads properties from `application.properties` files in the following locations and adds them to the Spring `Environment`:

- 1. A `/config` subdirectory of the current directory → 根目录下的 `/config` 目录下的【配置文件】
- 2. The current directory → 根目录下的【配置文件】
- 3. A classpath `/config` package → 类路径下的 `/config` 目录下的【配置文件】
- 4. The classpath root → 类路径下的【配置文件】

The list is ordered by precedence (properties defined in locations higher in the list override those defined in lower locations) <https://blog.csdn.net/f6>

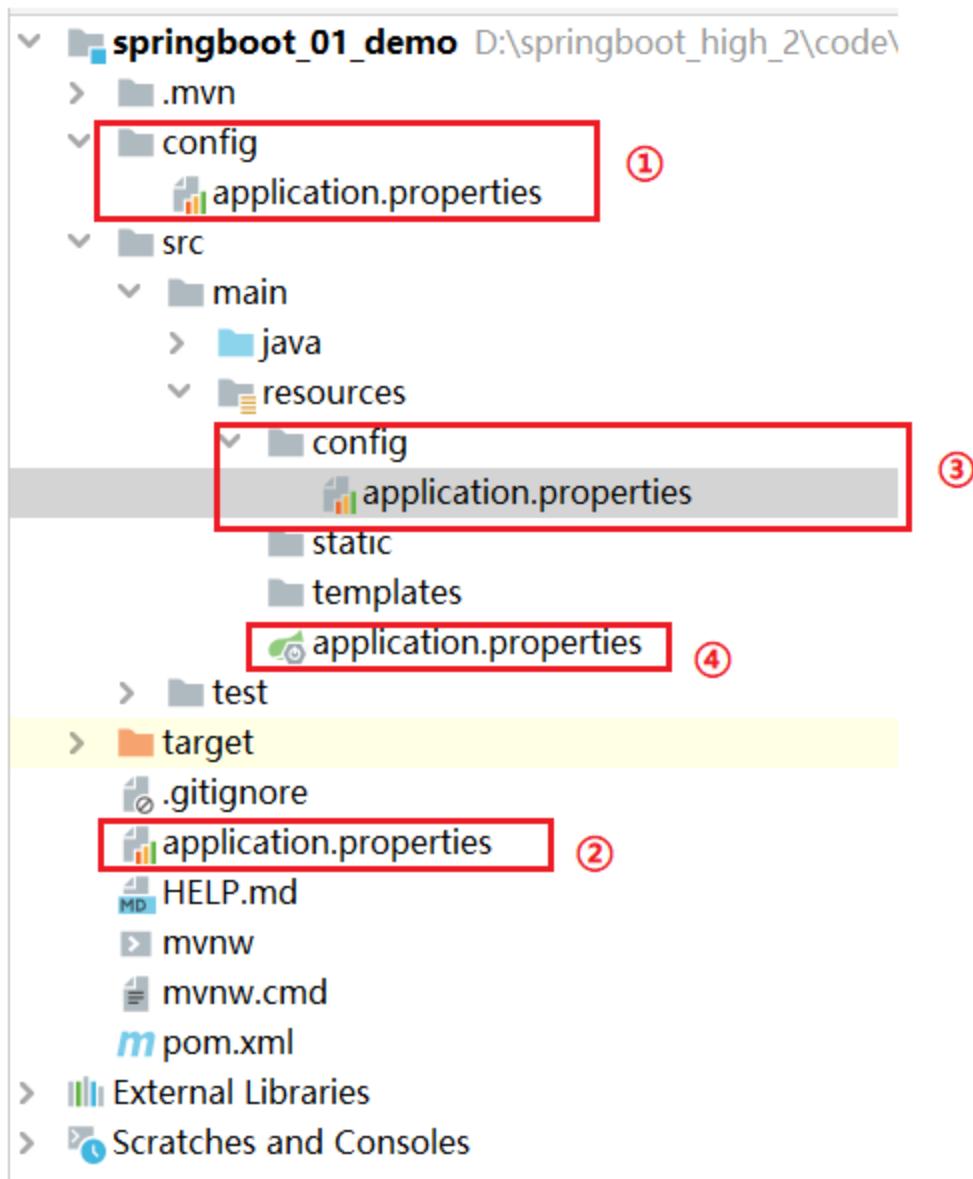
也可以从 `ConfigFileApplicationListener` 这类便可看出，其中 `DEFAULT_SEARCH_LOCATIONS` 属性设置了加载的目录：

翻译成文件系统：

```
-file:./config/  
-file:./  
-classpath:/config/  
-classpath:/
```

翻译成语言如下（按照优先级从高到低的顺序）：

1. 先去项目根目录找 config 文件夹下找配置文件
2. 再去根目录下找配置文件
3. 去 resources 下找 config 文件夹下找配置文件
4. 去 resources 下找配置文件



整个设计非常巧妙。SpringBoot会从这四个位置全部加载主配置文件，如果高优先级中配置文件属性与低优先级配置文件不冲突的属性，则会共同存在—互补配置。

SpringBoot会加载全部主配置文件；互补配置；

备注：

这里说的配置文件，都还是项目里面。最终都会被打进jar包里面的，需要注意。

1、如果同一个目录下，有application.yml也有application.properties，默认先读取application.properties。

2、如果同一个配置属性，在多个配置文件都配置了，默认使用第1个读取到的，后面读取的不覆盖前面读取到的。

3、创建SpringBoot项目时，一般的配置文件放置在“项目的resources目录下”

如果我们的配置文件名字不叫application.properties或者application.yml，可以通过以下参数来指定配置文件的名字，myproject是配置文件名

```
$ java -jar myproject.jar --spring.config.name=myproject
```

我们同时也可以指定其他位置的配置文件来生效

指定配置文件和默认加载的这些配置文件共同起作用形成互补配置。

```
java -jar run-0.0.1-SNAPSHOT.jar --  
spring.config.location=D:/application.properties
```

接下来，将针对这两种全局配置文件application.properties及application.yml进行讲解：

知识点补充！

Spring Boot 2.4 改进了处理 application.properties 和 application.yml 配置文件的方式，

如果是2.4.0之前版本，优先级properties>yaml

但是如果是2.4.0的版本，优先级yaml>properties

如果想继续使用 Spring Boot 2.3 的配置逻辑，也可以通过在 application.properties 或者 application.yml 配置文件中添加以下参数：

```
spring.config.use-legacy-processing = true
```

1.5.2 application.properties配置文件

使用Spring Initializr方式构建Spring Boot项目时，会在resource目录下自动生成一个空的 application.properties文件，Spring Boot项目启动时会自动加载application.properties文件。

我们可以在application.properties文件中定义Spring Boot项目的相关属性，当然，这些相关属性可以是系统属性、环境变量、命令参数等信息，也可以是自定义配置文件名称和位置

```
server.port=8081  
spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
spring.config.name=application
```

接下来，通过一个案例对Spring Boot项目中application.properties配置文件的具体使用进行讲解

演示：

预先准备了两个实体类文件，后续会演示将application.properties配置文件中的自定义配置属性注入到Person实体类的对应属性中

(1) 先在项目的com.lagou包下创建一个pojo包，并在该包下创建两个实体类Pet和Person

```
public class Pet {  
  
    private String type;  
    private String name;  
    // 省略属性getxx()和setxx()方法  
    // 省略toString()方法  
  
}
```

```
@Component //用于将Person类作为Bean注入到Spring容器中
```

```

@ConfigurationProperties(prefix = "person") //将配置文件中以person开头的属性注入到该类
中
public class Person {

    private int id;           //id
    private String name;      //名称
    private List hobby;        //爱好
    private String[] family;   //家庭成员
    private Map map;
    private Pet pet;          //宠物
    // 省略属性getXX()和setXX()方法
    // 省略toString()方法

}

```

@ConfigurationProperties(prefix = "person")注解的作用是将配置文件中以person开头的属性值通过setXX()方法注入到实体类对应属性中

@Component注解的作用是将当前注入属性值的Person类对象作为Bean组件放到Spring容器中，只有这样才能被@ConfigurationProperties注解进行赋值

(2) 打开项目的resources目录下的application.properties配置文件，在该配置文件中编写需要对Person类设置的配置属性



编写application.properties配置文件时，由于要配置的Person对象属性是我们自定义的，Spring Boot无法自动识别，所以不会有任何书写提示。在实际开发中，为了出现代码提示的效果来方便配置，在使用@ConfigurationProperties注解进行配置文件属性值注入时，可以在pom.xml文件中添加一个Spring Boot提供的配置处理器依赖：

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>

```

在pom.xml中添加上述配置依赖后，还需要重新运行项目启动类或者使用“Ctrl+F9”快捷键（即Build Project）重构当前Spring Boot项目方可生效

(3) 查看application.properties配置文件是否正确，同时查看属性配置效果，打开通过IDEA工具创建的项目测试类，在该测试类中引入Person实体类Bean，并进行输出测试

```

@RunWith(SpringRunner.class) // 测试启动器，并加载Spring Boot测试注解
@SpringBootTest // 标记为Spring Boot单元测试类，并加载项目的ApplicationContext上下文环境
class SpringbootDemoApplicationTests {

    // 配置测试
    @Autowired
    private Person person;

    @Test
    void configurationTest() {
        System.out.println(person);
    }
}

```

打印结果：

```
Person{id=1, name='tom', hobby=[吃饭, 睡觉, 打豆豆], family=[father, mother], map={k1=v1, k2=v2}, pet=Pet{type='dag', name='旺财'}}
```

可以看出，测试方法configurationTest()运行成功，同时正确打印出了Person实体类对象。至此，说明application.properties配置文件属性配置正确，并通过相关注解自动完成了属性注入。

1.5.3 application.yaml配置文件

YAML文件格式是Spring Boot支持的一种JSON超集文件格式，以数据为中心，比properties、xml等更适合做配置文件

- yaml和xml相比，少了一些结构化的代码，使数据更直接，一目了然
- 相比properties文件更简洁

-
- YAML文件的扩展名可以使用.yml或者.yaml。
 - application.yml文件使用“key: (空格) value”格式配置属性，使用缩进控制层级关系。

这里，针对不同数据类型的属性值，介绍一下YAML

(1) value值为普通数据类型（例如数字、字符串、布尔等）

当YAML配置文件中配置的属性值为普通数据类型时，可以直接配置对应的属性值，同时对于字符串类型的属性值，不需要额外添加引号，示例代码如下

```

server:
  port: 8080
  servlet:
    context-path: /hello

```

(2) value值为数组和单列集合

当YAML配置文件中配置的属性值为数组或单列集合类型时，主要有两种书写方式：缩进式写法和行内式写法。

其中，缩进式写法还有两种表示形式，示例代码如下

```
person:  
  hobby:  
    - play  
    - read  
    - sleep
```

或者使用如下示例形式

```
person:  
  hobby:  
    play,  
    read,  
    sleep
```

上述代码中，在YAML配置文件中通过两种缩进式写法对person对象的单列集合（或数组）类型的爱好hobby赋值为play、read和sleep。其中一种形式为“-（空格）属性值”，另一种形式为多个属性值之前加英文逗号分隔（注意，最后一个属性值后不要加逗号）。

```
person:  
  hobby: [play, read, sleep]
```

通过上述示例对比发现，YAML配置文件的行内式写法更加简明、方便。另外，包含属性值的中括号“[]”还可以进一步省略，在进行属性赋值时，程序会自动匹配和校对

(3) value值为Map集合和对象

当YAML配置文件中配置的属性值为Map集合或对象类型时，YAML配置文件格式同样可以分为两种书写方式：缩进式写法和行内式写法。

其中，缩进式写法的示例代码如下

```
person:  
  map:  
    k1: v1  
    k2: v2
```

对应的行内式写法示例代码如下

```
person:  
  map: {k1: v1, k2: v2}
```

在YAML配置文件中，配置的属性值为Map集合或对象类型时，缩进式写法的形式按照YAML文件格式编写即可，而行内式写法的属性值要用大括号“{}”包含。

接下来，在Properties配置文件演示案例基础上，通过配置application.yaml配置文件对Person对象进行赋值，具体使用如下

(1) 在项目的resources目录下，新建一个application.yaml配置文件，在该配置文件中编写为Person类设置的配置属性

```
#对实体类对象Person进行属性配置
person:
  id: 1
  name: lucy
  hobby: [吃饭, 睡觉, 打豆豆]
  family: [father, mother]
  map: {k1: v1, k2: v2}
  pet: {type: dog, name: 旺财}
```

(2) 再次执行测试

```
Person{id=1, name='lucy', hobby=[吃饭, 数据, 打豆豆], family=[father, mother], map={k1=v1, k2=v2}, pet=Pet{type='dog', name='旺财'}}
```

可以看出，测试方法configurationTest()同样运行成功，并正确打印出了Person实体类对象。

1.6 属性注入

使用Spring Boot全局配置文件设置属性时：

如果配置属性是Spring Boot已有属性，例如服务端口server.port，那么Spring Boot内部会自动扫描并读取这些配置文件中的属性值并覆盖默认属性。

如果配置的属性是用户自定义属性，例如刚刚自定义的Person实体类属性，还必须在程序中注入这些配置属性方可生效。

1.6.1 属性注入常用注解

@Configuration：声明一个类作为配置类

@Bean：声明在方法上，将方法的返回值加入Bean容器

@Value：属性注入

@ConfigurationProperties(prefix = "jdbc")：批量属性注入

@PropertySource("classpath:/jdbc.properties")指定外部属性文件。在类上添加

1.6.2 @Value属性值注入

1.引入数据源连接依赖

```
<dependency>
  <groupId>com.github.drtrang</groupId>
  <artifactId>druid-spring-boot2-starter</artifactId>
  <version>1.1.10</version>
</dependency>
```

2.application.properties添加信息

```
jdbc.driverClassName=com.mysql.jdbc.Driver  
jdbc.url=jdbc:mysql://127.0.0.1:3306/springboot_h  
jdbc.username=root  
jdbc.password=123
```

3.配置数据源

创建JdbcConfiguration类： 使用spring中的value注解对每个属性进行注入,用bean注解将返回值添加到容器中

```
@Configuration  
public class JdbcConfiguration {  
  
    @Value("${jdbc.url}")  
    String url;  
  
    @Value("${jdbc.driverClassName}")  
    String driverClassName;  
  
    @Value("${jdbc.username}")  
    String username;  
  
    @Value("${jdbc.password}")  
    String password;  
  
    @Bean  
    public DataSource dataSource() {  
        DruidDataSource dataSource = new DruidDataSource();  
        dataSource.setUrl(url);  
        dataSource.setDriverClassName(driverClassName);  
        dataSource.setUsername(username);  
        dataSource.setPassword(password);  
        return dataSource;  
    }  
}
```

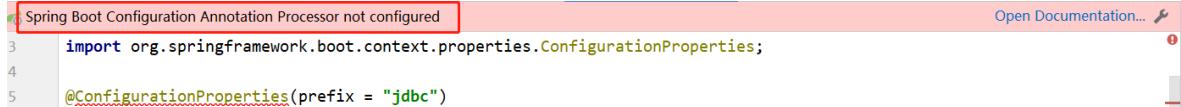
1.6.3 @ConfigurationProperties批量注入

新建 JdbcProperties，用来进行属性注入：

```
@ConfigurationProperties(prefix = "jdbc") //这里需要定义出在application文件中定义属性值得前缀信息  
public class JdbcProperties {  
  
    private String url;  
  
    private String driverClassName;  
  
    private String username;  
  
    private String password;  
  
    // 注：要生成属性的set方法
```

```
}
```

注：添加@ConfigurationProperties注解后有警告：springboot 配置注释处理器未配置（编写配置文件此时无提示）



```
Spring Boot Configuration Annotation Processor not configured
import org.springframework.boot.context.properties.ConfigurationProperties;
@ConfigurationProperties(prefix = "jdbc")
```

添加spring-boot-configuration-processor后出现提示，加完依赖后通过Ctrl+F9来使之生效

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

接着发现，仍然有红色警告

```
@ConfigurationProperties(prefix = "jdbc")
public class JdbcPro Not registered via @EnableConfigurationProperties, marked as Spring component, or scanned
via @ConfigurationPropertiesScan
private String uri;
```

@EnableConfigurationProperties 是 Spring Boot 提供的一个注解，使用该注解用于启用应用对另外一个注解 @ConfigurationProperties 的支持，，用于设置一组使用了注解 @ConfigurationProperties 的类，用作为 bean 定义注册到容器中。

```
@Configuration
@EnableConfigurationProperties(JdbcProperties.class)
@ConfigurationProperties(prefix = "jdbc")
public class JdbcProperties {
```

2.application.properties添加信息

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://127.0.0.1:3306/springboot_h
jdbc.username=root
jdbc.password=123
```

注意：将配置信息添加到这里，通过前缀进行区分，进行引用

3.查看效果

```
@RunWith(SpringRunner.class)
@SpringBootTest
class Springbootlesson1ApplicationTests {

    @Autowired
    private JdbcProperties jdbcProperties;

    @Test
    public void jdbcPropertiesTest(){
        System.out.println(jdbcProperties);
    }
}
```

打印结果: JdbcProperties{url='jdbc:mysql://127.0.0.1:3306/springboot_h',
driverClassName='com.mysql.jdbc.Driver', username='root', password='root'}

1.6.4 第三方配置

除了 `@ConfigurationProperties` 用于注释类之外，您还可以在公共 `@Bean` 方法上使用它。当要将属性绑定到控件之外的第三方组件时，这样做特别有用。

效果演示:

创建一个其他组件类

```
@Data
public class AnotherComponent {

    private boolean enabled;

    private InetAddress remoteAddress;
}
```

创建MyService

```
@Configuration
public class Myservice {

    @ConfigurationProperties("another")
    @Bean
    public AnotherComponent anotherComponent(){
        return new AnotherComponent();
    }
}
```

配置文件

```
another.enabled=true
another.remoteAddress=192.168.10.11
```

测试类

```

@Autowired
private AnotherComponent anotherComponent;

@Test
public void myServiceTest(){
    System.out.println(anotherComponent);
}

```

我们通过测试可以获得AnotherComponent组件的实例对象。

1.6.5 松散绑定

Spring Boot使用一些宽松的规则将环境属性绑定到@ConfigurationProperties bean，因此环境属性名和bean属性名之间不需要完全匹配

例如属性类：

```

@Data
@Component
@ConfigurationProperties("acme.my-person.person")
public class OwnerProperties {

    private String firstName;

}

```

```

acme:
  my-person:
    person:
      first-name: 泰森

```

属性文件中配置	说明
acme.my-project.person.first-name	羊肉串模式case, 推荐使用
acme.myProject.person.firstName	标准驼峰模式
acme.my_project.person.first_name	下划线模式
ACME_MYPROJECT_PERSON_FIRSTNAME	大写下划线, 如果使用系统环境时候推荐使用

1.6.6 @ConfigurationProperties vs @Value

特征	@ConfigurationProperties	@value
宽松的绑定	yes	Limited (详见下方官网截图)
元数据支持	yes	no
SpEL 表达式	no	yes
应用场景	批量属性绑定	单个属性绑定



If you do want to use `@Value`, we recommend that you refer to property names using their canonical form (kebab-case using only lowercase letters). This will allow Spring Boot to use the same logic as it does when relaxed binding `@ConfigurationProperties`. For example, `@Value("{demo.item-price}")` will pick up `demo.item-price` and `demo.itemPrice` forms from the `application.properties` file, as well as `DEMO_ITEMPRICE` from the system environment. If you used `@Value("{demo.itemPrice}")` instead, `demo.item-price` and `DEMO_ITEMPRICE` would not be considered.

1.7 SpringBoot日志框架

1.7.1 日志框架介绍

在项目的开发中，日志是必不可少的一个记录事件的组件，不管是记录运行情况还是追踪线上问题，都离不开对日志的分析，所以也会相应的在项目中实现和构建我们所需要的日志框架。

而市面上常见的日志框架有很多，比如：JCL、SLF4J、JBoss-logging、jUL、log4j、log4j2、logback等等，我们该如何选择呢？

通常情况下，日志是由一个抽象层+实现层的组合来搭建的。

日志-抽象层	日志-实现层
JCL (Jakarta Commons Logging) 、 SLF4J (Simple Logging Facade for Java) 、 jboss-logging	jul (java.util.logging) 、 log4j、 logback、 log4j2

Spring 框架选择使用了 JCL 作为默认日志输出。而 Spring Boot 默认选择了 SLF4J 结合 LogBack

1.7.2 SLF4J 的使用

在开发的时候不应该直接使用日志实现类，应该使用日志的抽象层。具体参考 [SLF4J 官方](#)。

SLF4J 官方给出了简单示例。

首先要为系统导入 SLF4J 的 jar .

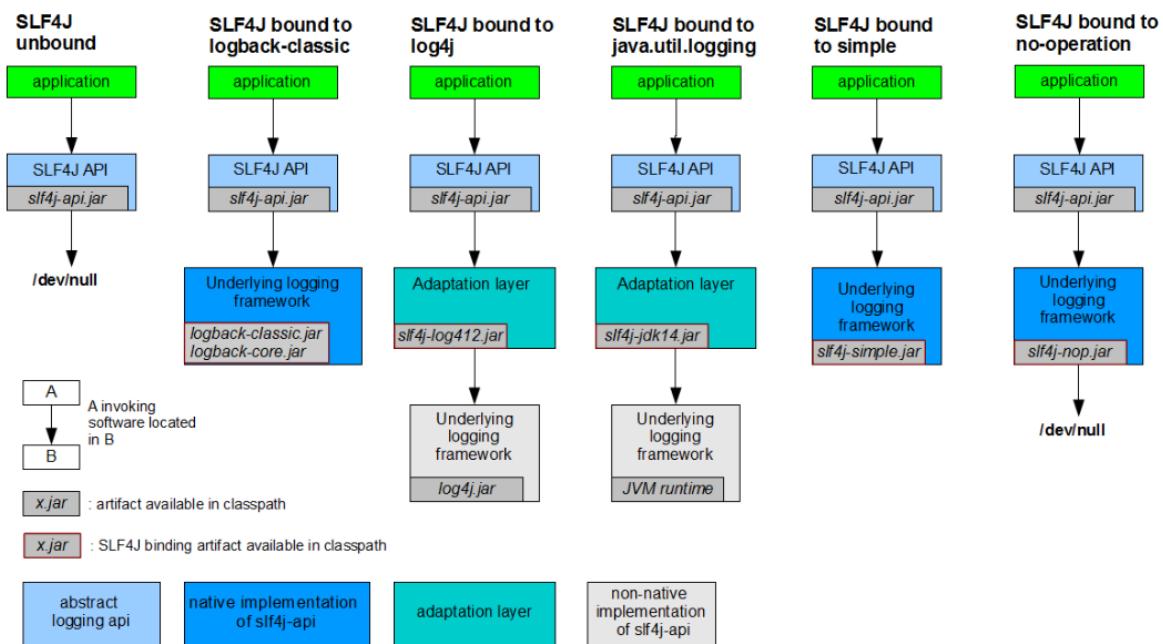
```

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class HelloWorld {
    public static void main(String[] args) {
        HelloWorld.class);
        logger.info("Hello World");
    }
}

```

下图是 SLF4J 结合各种日志框架的官方示例，从图中可以清晰的看出 SLF4J API 永远作为日志的门面，直接应用与应用程序中。



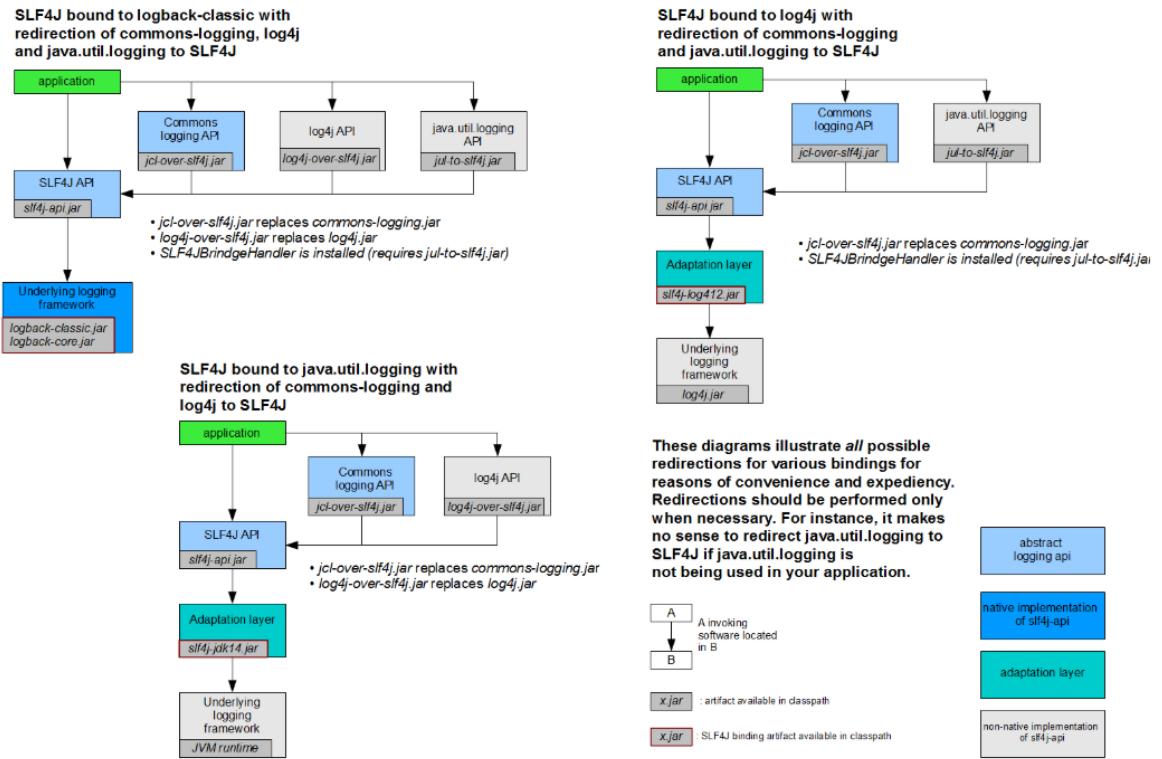
注意:由于每一个日志的实现框架都有自己的配置文件，所以在使用 SLF4j 之后，配置文件还是要使用实现日志框架的配置文件。

1.7.3. 统一日志框架的使用

遗留问题：A项目（slf4j + logback）：Spring（commons logging）、Hibernate（jboss-logging）、mybatis....

一般情况下，在项目中存在着各种不同的第三方 jar，且它们的日志选择也可能不尽相同，显然这样是不利于我们使用的，那么如果我们想为项目设置统一的日志框架该怎么办呢？

在 [SLF4J 官方](#)，也给了我们参考的例子



从图中我们得到一种统一日志框架使用的方式，可以使用一种和要替换的日志框架类完全一样的 jar 进行替换，这样不至于原来的第三方 jar 报错，而这个替换的 jar 其实使用了 SLF4J API. 这样项目中的日志就都可以通过 SLF4J API 结合自己选择的框架进行日志输出。

统一日志框架使用步骤归纳如下：

1. 排除系统中的其他日志框架。
2. 使用中间包替换要替换的日志框架。
3. 导入我们选择的 SLF4J 实现。

1.7.4. Spring Boot 的日志关系

① 排除其他日志框架

根据上面总结的要统一日志框架的使用，第一步要排除其他的日志框架，在 Spring Boot 的 Maven 依赖里可以清楚的看到 Spring Boot 排除了其他日志框架。

```
<dependency>
    <groupId>org.springframework.ws</groupId>
    <artifactId>spring-ws-security</artifactId>
    <version>${spring-ws.version}</version>
    <exclusions>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework.ws</groupId>
    <artifactId>spring-ws-support</artifactId>
    <version>${spring-ws.version}</version>
    <exclusions>
        <exclusion>
            <artifactId>commons-logging</artifactId>
            <groupId>commons-logging</groupId>
        </exclusion>
    </exclusions>
</dependency>
```

我们自行排除依赖时也只需要按照图中的方式就好了。

② 统一框架引入替换包

Spring Boot 是使用了 SLF4J+logback 的日志框架组合，查看 Spring Boot 项目的 Maven 依赖关系可以看到 Spring Boot 的核心启动器 spring-boot-starter 引入了 spring-boot-starter-logging.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-logging</artifactId>
    <version>2.4.0.RELEASE</version>
</dependency>
```

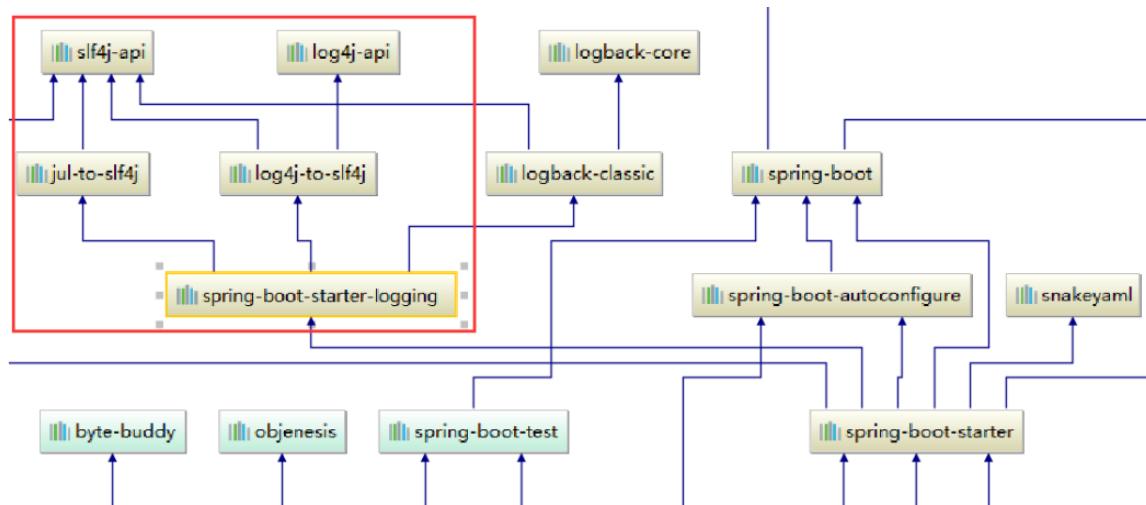
而 spring-boot-starter-logging 的 Maven 依赖主要引入了 logback-classic (包含了日志框架 Logback 的实现)，log4j-to-slf4j (在 log4j 日志框架作者开发此框架的时候还没有想到使用日志抽象层进行开发，因此出现了 log4j 向 slf4j 转换的工具)，jul-to-slf4j (Java 自带的日志框架转换为 slf4j).

```
<dependencies>
    <dependency>
        <groupId>ch.qos.logback</groupId>
        <artifactId>logback-classic</artifactId>
        <version>1.2.3</version>
        <scope>compile</scope>
    </dependency>
    <dependency>
        <groupId>org.apache.logging.log4j</groupId>
        <artifactId>log4j-to-slf4j</artifactId>
        <version>2.13.3</version>
        <scope>compile</scope>
    </dependency>
```

```
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>jul-to-slf4j</artifactId>
    <version>1.7.30</version>
    <scope>compile</scope>
</dependency>
</dependencies>
```

从上面的分析，Spring Boot 对日志框架的使用已经是清晰明了了

，我们使用 IDEA 工具查看 Maven 依赖关系，可以清晰的看到日志框架的引用



由此可见，Spring Boot 可以自动的适配日志框架，而且底层使用 **SLF4j + LogBack** 记录日志，如果我们自行引入其他框架，需要排除其日志框架。

1.7.5. Spring Boot 的日志使用

日志级别和格式

从上面的分析，发现 Spring Boot 默认已经使用了 **SLF4J + LogBack**。所以我们在不进行任何额外操作的情况下就可以使用 **SLF4J + Logback** 进行日志输出。

编写 Java 测试类进行测试。

```
import org.junit.Test;
import org.junit.runner.RunWith;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

/**
 * 测试日志输出,
 * SLF4J 日志级别从小到大trace,debug,info,warn,error
 *
 */
@RunWith(SpringRunner.class)
@SpringBootTest
public class LogbackTest {
```

```

Logger logger = LoggerFactory.getLogger(getClass());

@Test
public void testLog() {
    logger.trace("Trace 日志...");
    logger.debug("Debug 日志...");
    logger.info("Info 日志...");
    logger.warn("Warn 日志...");
    logger.error("Error 日志...");
}
}

```

已知日志级别从小到大为 trace < debug < info < warn < error . 运行得到输出如下。由此可见 **Spring Boot 默认日志级别为INFO.**

```

2020-11-16 19:58:43.094 INFO 39940 --- [           main]
com.lagou.Springboot01DemoApplicationTests : Info 日志...
2020-11-16 19:58:43.094 WARN 39940 --- [           main]
com.lagou.Springboot01DemoApplicationTests : Warn 日志...
2020-11-16 19:58:43.094 ERROR 39940 --- [          main]
com.lagou.Springboot01DemoApplicationTests : Error 日志...

```

从上面的日志结合 Logback 日志格式可以知道 Spring Boot 默认日志格式是

```

%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50} - %msg%n
# %d{yyyy-MM-dd HH:mm:ss.SSS} 时间
# %thread 线程名称
# %-5level 日志级别从左显示5个字符宽度
# %logger{50} 类名
# %msg%n 日志信息加换行

```

至于为什么 Spring Boot 的默认日志输出格式是这样?

The screenshot shows an IDE interface with the file 'defaults.xml' open. The file contains XML configuration for Logback. A specific line of code is highlighted with a red box: '<property name="CONSOLE_LOG_PATTERN" value="\${CONSOLE_LOG_PATTERN:-%clr(%d\${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd} %L)}'.

```

<?xml version="1.0" encoding="UTF-8"?>

<!--
Default logback configuration provided for import
-->

<included>
<conversionRule conversionWord="clr" converterClass="org.springframework.boot.logging.logback.ColorConverter" />
<conversionRule conversionWord="wex" converterClass="org.springframework.boot.logging.logback.WhitespaceThrowableProxyConverter" />
<conversionRule conversionWord="wEx" converterClass="org.springframework.boot.logging.logback.ExtendedWhitespaceThrowableProxyConverter" />
<property name="CONSOLE_LOG_PATTERN" value="${CONSOLE_LOG_PATTERN:-%clr(%d${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd} %L)}"/>
<property name="FILE_LOG_PATTERN" value="${FILE_LOG_PATTERN:-%d${LOG_DATEFORMAT_PATTERN:-yyyy-MM-dd} %L}"/>
<logger name="org.apache.catalina.startup.DigesterFactory" level="ERROR"/>
<logger name="org.apache.catalina.util.LifecycleBase" level="ERROR"/>
<logger name="org.apache.coyote.http11.Http11NioProtocol" level="WARN"/>
<logger name="org.apache.sshd.common.util.SecurityUtils" level="WARN"/>
<logger name="org.apache.tomcat.util.net.NioSelectorPool" level="WARN"/>
<logger name="org.eclipse.jetty.util.component.AbstractLifeCycle" level="ERROR"/>
<logger name="org.hibernate.validator.internal.util.Version" level="WARN"/>
<logger name="org.springframework.boot.actuate.endpoint.jmx" level="WARN"/>
</included>

```

我们可以在 Spring Boot 的源码里找到答案。

1.7.6 自定义日志输出

可以直接在配置文件编写日志相关配置

```
# 日志配置
# 指定具体包的日志级别
logging.level.com.lagou=debug
# 控制台和日志文件输出格式
logging.pattern.console=%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level
%logger{50} - %msg%
logging.pattern.file=%d{yyyy-MM-dd HH:mm:ss.SSS} [%thread] %-5level %logger{50}
- %msg%
# 日志输出路径，默认文件spring.log
logging.file.path=spring.log
#logging.file.name=log.log
```

关于日志的输出路径，可以使用 logging.file 或者 logging.path 进行定义，两者存在关系如下表。

Logging.file	Logging.path	例子	描述
(没有)	(没有)		仅控制台记录。
具体文件	(没有)	my.log	写入指定的日志文件，名称可以是精确位置或相对于当前目录。
(没有)	具体目录	/var/log	写入 spring.log 指定的目录，名称可以是精确位置或相对于当前目录。

1.7.7 替换日志框架

因为 Log4j 日志框架已经年久失修，原作者都觉得写的不好，所以下面演示替换日志框架为 Log4j2 的方式。根据[官网](#)我们 Log4j2 与 logging 需要二选一，因此修改 pom 如下

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <artifactId>spring-boot-starter-logging</artifactId>
            <groupId>org.springframework.boot</groupId>
        </exclusion>
    </exclusions>
</dependency>

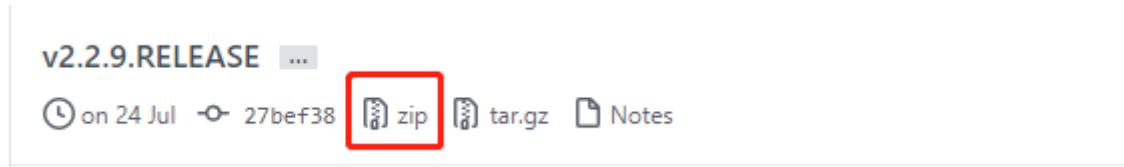
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

2. SpringBoot源码剖析

2.1 SpringBoot源码环境构建

2.1.1 下载源码

- <https://github.com/spring-projects/spring-boot/releases>
- 下载对应版本的源码（课程中采用spring-boot-2.2.9.RELEASE）



2.1.2 环境准备

- 1、JDK1.8+
- 2、Maven3.5+

```
C:\> C:\WINDOWS\system32\cmd.exe
Microsoft Windows [版本 10.0.18362.1139]
(c) 2019 Microsoft Corporation。保留所有权利。

C:\Users\0713>java -version
java version "11.0.2" 2019-01-15 LTS
Java(TM) SE Runtime Environment 18.9 (build 11.0.2+9-LTS)
Java HotSpot(TM) 64-Bit Server VM 18.9 (build 11.0.2+9-LTS, mixed mode)

C:\Users\0713>mvn -version
Apache Maven 3.5.2 (138edd61fd100ec658bfa2d307c43b76940a5d7d; 2017-10-18T15:58:13+08:00)
Maven home: D:\maven_lagou\apache-maven-3.5.2\bin\..
Java version: 11.0.2, vendor: Oracle Corporation
Java home: D:\jdk11
Default locale: zh_CN, platform encoding: GBK
OS name: "windows 10", version: "10.0", arch: "amd64", family: "windows"
```

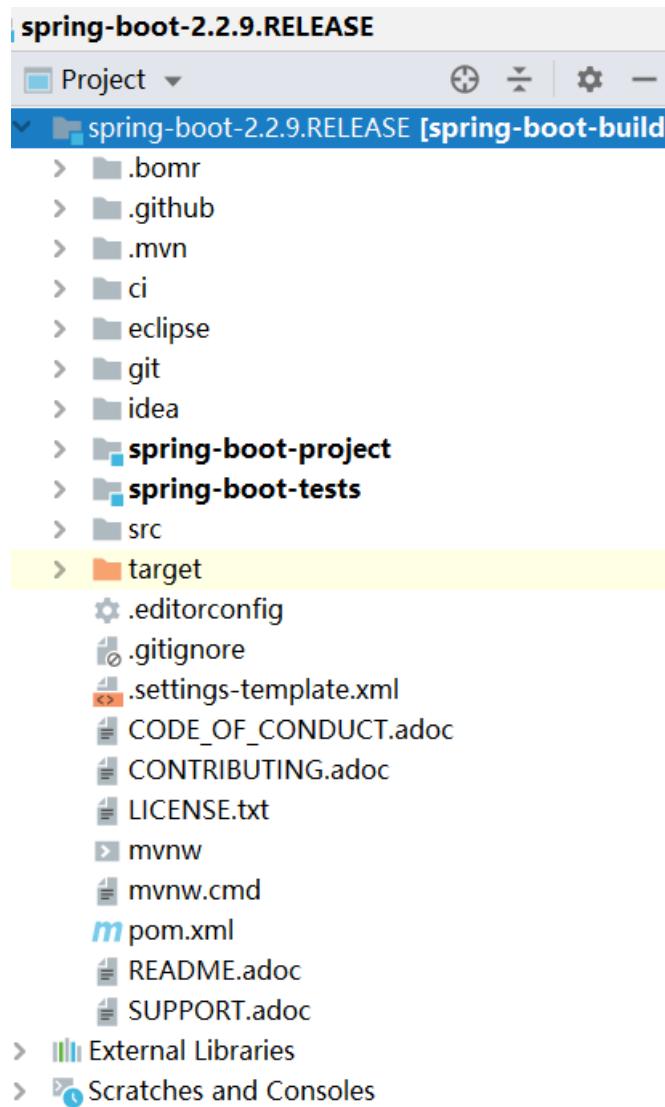
2.1.2 编译源码

- 进入spring-boot源码根目录
- 执行mvn命令: **mvn clean install -DskipTests -Pfast** // 跳过测试用例，会下载大量jar包（时间会长一些）

```
C:\Windows\System32\cmd.exe
[INFO] Spring Boot Thymeleaf Starter ..... SUCCESS [ 0.381 s]
[INFO] Spring Boot Undertow Starter ..... SUCCESS [ 0.497 s]
[INFO] Spring Boot WebFlux Starter ..... SUCCESS [ 0.672 s]
[INFO] Spring Boot WebSocket Starter ..... SUCCESS [ 0.539 s]
[INFO] Spring Boot Web Services Starter ..... SUCCESS [ 0.651 s]
[INFO] Spring Boot CLI ..... SUCCESS [ 7.781 s]
[INFO] Spring Boot Docs ..... SUCCESS [ 8.783 s]
[INFO] Spring Boot Project ..... SUCCESS [ 0.200 s]
[INFO] Spring Boot Samples Invoker ..... SUCCESS [ 0.490 s]
[INFO] Spring Boot Tests ..... SUCCESS [ 0.061 s]
[INFO] Spring Boot Integration Tests ..... SUCCESS [ 0.060 s]
[INFO] Spring Boot Configuration Processor Tests ..... SUCCESS [ 0.578 s]
[INFO] Spring Boot DevTools Tests ..... SUCCESS [ 0.904 s]
[INFO] Spring Boot Hibernate 5.2 tests ..... SUCCESS [ 26.790 s]
[INFO] Spring Boot Server Tests ..... SUCCESS [ 0.974 s]
[INFO] Spring Boot Launch Script Integration Tests ..... SUCCESS [ 1.011 s]
-----
[INFO] BUILD SUCCESS
-----
[INFO] Total time: 16:59 min
[INFO] Finished at: 2020-11-17T12:06:48+08:00
```

2.1.3 导入IDEA

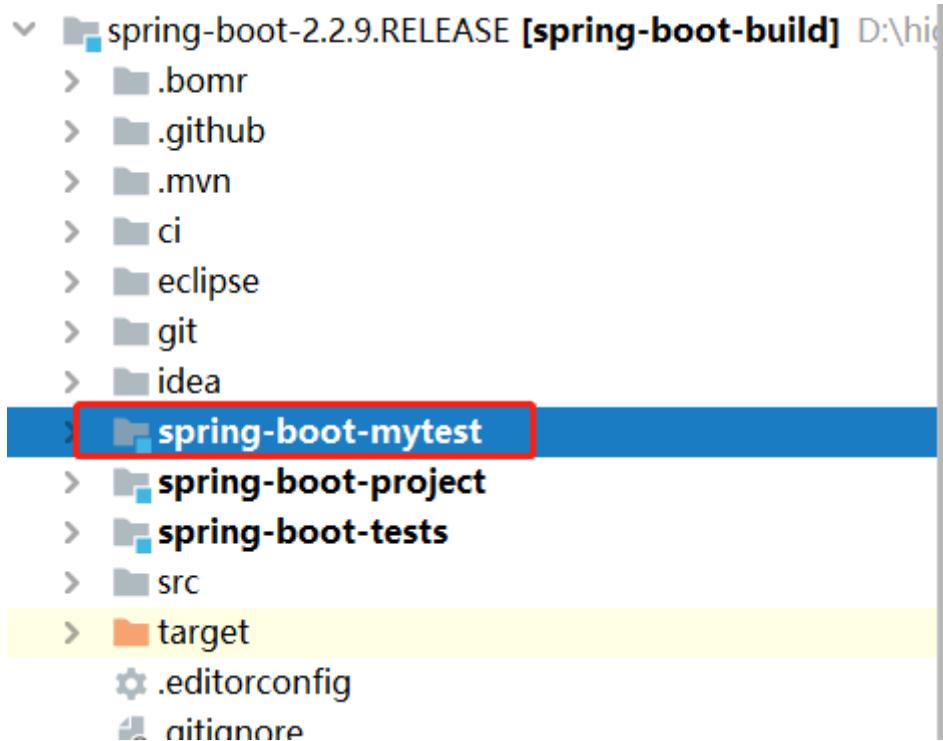
将编译后的项目导入IDEA中



打开pom.xml关闭maven代码检查

```
<properties>
    <revision>2.2.9.RELEASE</revision>
    <main.basedir>${basedir}</main.basedir>
    <disable.checks>true</disable.checks>
</properties>
```

2.1.4 新建一个module



2.1.5 新建一个Controller

```
@RestController
public class TestController {

    @RequestMapping("/test")
    public String test(){
        System.out.println("源码环境搭建完成");
        return "源码环境搭建完成";
    }
}
```

启动测试

```
Console Endpoints
2020-11-17 16:14:28.746 INFO 11228 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http)
2020-11-17 16:14:28.764 INFO 11228 --- [           main] com.lagou.SpringBootMytestApplication : Started SpringBootMytestApplication in
2020-11-17 16:14:51.408 INFO 11228 --- [nio-8080-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet
2020-11-17 16:14:51.408 INFO 11228 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2020-11-17 16:14:51.409 INFO 11228 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 1 ms
源码环境搭建完成
```

2.2 源码剖析-依赖管理

问题：（1）为什么导入dependency时不需要指定版本？

在Spring Boot入门程序中，项目pom.xml文件有两个核心依赖，分别是spring-boot-starter-parent和spring-boot-starter-web，关于这两个依赖的相关介绍具体如下

spring-boot-starter-parent

在chapter01项目中的pom.xml文件中找到spring-boot-starter-parent依赖，示例代码如下：

```
<!-- Spring Boot父项目依赖管理 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.9.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

上述代码中，将spring-boot-starter-parent依赖作为Spring Boot项目的统一父项目依赖管理，并将项目版本号统一为2.2.9.RELEASE，该版本号根据实际开发需求是可以修改的

使用“Ctrl+鼠标左键”进入并查看spring-boot-starter-parent底层源文件，先看spring-boot-starter-parent做了哪些事

首先看spring-boot-starter-parent的properties节点

```
<properties>
    <main.basedir>${basedir}/../../../../../</main.basedir>
    <java.version>1.8</java.version>
    <resource.delimiter>@</resource.delimiter> <!-- delimiter that doesn't
clash with Spring ${} placeholders -->
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
    <maven.compiler.source>${java.version}</maven.compiler.source>
    <maven.compiler.target>${java.version}</maven.compiler.target>
</properties>
```

在这里spring-boot-starter-parent定义了：

1. 工程的Java版本为1.8。
2. 工程代码的编译源文件编码格式为UTF-8
3. 工程编译后的文件编码格式为UTF-8
4. Maven打包编译的版本

再来看spring-boot-starter-parent的「build」节点

接下来看POM的build节点，分别定义了resources资源和pluginManagement

```
<resources>
    <resource>
        <filtering>true</filtering>
        <directory>${basedir}/src/main/resources</directory>
        <includes>
            <include>**/application*.yml</include>
            <include>**/application*.yaml</include>
            <include>**/application*.properties</include>
        </includes>
```

```

    </resource>
<resources>
    <directory>${basedir}/src/main/resources</directory>
    <excludes>
        <exclude>**/application*.yml</exclude>
        <exclude>**/application*.yaml</exclude>
        <exclude>**/application*.properties</exclude>
    </excludes>
</resources>

```

我们详细看一下 resources 节点，里面定义了资源过滤，针对 application 的 yml、 properties 格式进行了过滤，可以支持支持不同环境的配置，比如 application-dev.yml、 application-test.yml、 application-dev.properties、 application-dev.properties 等等。

pluginManagement 则是引入了相应的插件和对应的版本依赖

最后来看spring-boot-starter-parent的父依赖 spring-boot-dependencies

spring-boot-dependencies的properties节点

我们看定义POM，这个才是SpringBoot项目的真正管理依赖的项目，里面定义了SpringBoot相关的版本

```

<properties>
    <main.basedir>${basedir}/...</main.basedir>
    <!-- Dependency versions -->
    <activemq.version>5.15.13</activemq.version>
    <antlr2.version>2.7.7</antlr2.version>
    <appengine-sdk.version>1.9.81</appengine-sdk.version>
    <artemis.version>2.10.1</artemis.version>
    <aspectj.version>1.9.6</aspectj.version>
    <assertj.version>3.13.2</assertj.version>
    <atomikos.version>4.0.6</atomikos.version>
    <awaitility.version>4.0.3</awaitility.version>
    <bitronix.version>2.1.4</bitronix.version>
    <byte-buddy.version>1.10.13</byte-buddy.version>
    <caffeine.version>2.8.5</caffeine.version>
    <cassandra-driver.version>3.7.2</cassandra-driver.version>
    <classmate.version>1.5.1</classmate.version>
    .....
</properties>

```

spring-boot-dependencies的dependencyManagement节点

在这里， dependencies 定义了SpringBoot版本的依赖的组件以及相应版本。

```

<dependencyManagement>
    <dependencies>
        <!-- Spring Boot -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot</artifactId>
            <version>${revision}</version>
        </dependency>
        <dependency>

```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-test</artifactId>
<version>${revision}</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-test-autoconfigure</artifactId>
    <version>${revision}</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator</artifactId>
    <version>${revision}</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-actuator-autoconfigure</artifactId>
    <version>${revision}</version>
</dependency>
.....
</dependencyManagement>
```

`spring-boot-starter-parent` 通过继承 `spring-boot-dependencies` 从而实现了 SpringBoot 的版本依赖管理, 所以我们的 SpringBoot 工程继承 `spring-boot-starter-parent` 后已经具备版本锁定等配置了, 这也就是在 Spring Boot 项目中部分依赖不需要写版本号的原因

(2) 问题2: `spring-boot-starter-parent` 父依赖启动器的主要作用是进行版本统一管理, 那么项目运行依赖的 JAR 包是从何而来的?

spring-boot-starter-web

查看 `spring-boot-starter-web` 依赖文件源码, 核心代码具体如下

```

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-tomcat</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-el</artifactId>
        </exclusion>
    </exclusions>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
</dependency>
```

```
</dependencies>
```

从上述代码可以发现，spring-boot-starter-web依赖启动器的主要作用是打包了Web开发场景所需的底层所有依赖（基于依赖传递，当前项目也存在对应的依赖jar包）

正是如此，在pom.xml中引入spring-boot-starter-web依赖启动器时，就可以实现Web场景开发，而不需要额外导入Tomcat服务器以及其他Web依赖文件等。

当然，这些引入的依赖文件的版本号还是由spring-boot-starter-parent父依赖进行的统一管理。

Spring Boot除了提供有上述介绍的Web依赖启动器外，还提供了其他许多开发场景的相关依赖，我们可以打开Spring Boot官方文档，搜索“Starters”关键字查询场景依赖启动器

Spring Boot在该 org.springframework.boot 组下提供了以下应用程序启动器：

表1. Spring Boot应用程序启动器

名称	描述
spring-boot-starter	核心入门工具，包括自动配置支持，日志记录和YAML
spring-boot-starter-activemq	使用Apache ActiveMQ的JMS消息传递入门
spring-boot-starter-amqp	使用Spring AMQP和Rabbit MQ的入门
spring-boot-starter-aop	使用Spring AOP和AspectJ进行面向方面编程的入门
spring-boot-starter-artemis	使用Apache Artemis的JMS消息传递入门
spring-boot-starter-batch	使用Spring Batch的入门
spring-boot-starter-cache	开始使用Spring Framework的缓存支持
spring-boot-starter-data-cassandra	使用Cassandra分布式数据库和Spring Data Cassandra的入门
spring-boot-starter-data-cassandra-reactive	使用Cassandra分布式数据库和Spring Data Cassandra Reactive的入门
spring-boot-starter-data-couchbase	使用Couchbase面向文档的数据库和Spring Data Couchbase的入门
spring-boot-starter-data-couchbase-reactive	使用Couchbase面向文档的数据库和Spring Data Couchbase Reactive的入门
spring-boot-starter-data-elasticsearch	使用Elasticsearch搜索和分析引擎以及Spring Data Elasticsearch的入门者

列出了Spring Boot官方提供的部分场景依赖启动器，这些依赖启动器适用于不同的场景开发，使用时只需要在pom.xml文件中导入对应的依赖启动器即可。

需要说明的是，Spring Boot官方并不是针对所有场景开发的技术框架都提供了场景启动器，例如阿里巴巴的Druid数据源等，Spring Boot官方就没有提供对应的依赖启动器。为了充分利用Spring Boot框架的优势，在Spring Boot官方没有整合这些技术框架的情况下，Druid等技术框架所在的开发团队主动与Spring Boot框架进行了整合，实现了各自的依赖启动器，例如druid-spring-boot-starter等。我们在pom.xml文件中引入这些第三方的依赖启动器时，切记要配置对应的版本号

2.3 源码剖析-自动配置

自动配置：根据我们添加的jar包依赖，会自动将一些配置类的bean注册进ioc容器，我们可以需要的地方使用@autowired或者@Resource等注解来使用它。

问题：Spring Boot到底是如何进行自动配置的，都把哪些组件进行了自动配置？

Spring Boot应用的启动入口是@SpringBootApplication注解标注类中的main()方法，

@SpringBootApplication：`springBoot` 应用标注在某个类上说明这个类是 `SpringBoot` 的主配置类，`SpringBoot` 就应该运行这个类的 `main()` 方法启动 `SpringBoot` 应用。

@SpringBootApplication

下面，查看@SpringBootApplication内部源码进行分析，核心代码具体如下

```
@Target({ElementType.TYPE}) //注解的适用范围,Type表示注解可以描述在类、接口、注解或枚举中
@Retention(RetentionPolicy.RUNTIME) //表示注解的生命周期, Runtime运行时
@Documented //表示注解可以记录在javadoc中
@Inherited //表示可以被子类继承该注解
@SpringBootConfiguration      // 标明该类为配置类
@EnableAutoConfiguration      // 启动自动配置功能
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes =
TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes =
AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {

    // 根据class来排除特定的类，使其不能加入spring容器，传入参数value类型是class类型。
    @AliasFor(annotation = EnableAutoConfiguration.class)
    Class<?>[] exclude() default {};

    // 根据classname 来排除特定的类，使其不能加入spring容器，传入参数value类型是class的全
    // 名字符串数组。
    @AliasFor(annotation = EnableAutoConfiguration.class)
    String[] excludeName() default {};

    // 指定扫描包，参数是包名的字符串数组。
    @AliasFor(annotation = ComponentScan.class, attribute = "basePackages")
    String[] scanBasePackages() default {};

    // 扫描特定的包，参数类似是Class类型数组。
    @AliasFor(annotation = ComponentScan.class, attribute =
"basePackageClasses")
    Class<?>[] scanBasePackageClasses() default {};

}
```

从上述源码可以看出，`@SpringBootApplication`注解是一个组合注解，前面4个是注解的元数据信息，我们主要看后面3个注解：`@SpringBootConfiguration`、`@EnableAutoConfiguration`、`@ComponentScan`三个核心注解，关于这三个核心注解的相关说明具体如下

@SpringBootConfiguration

@SpringBootConfiguration：`SpringBoot` 的配置类，标注在某个类上，表示这是一个 `SpringBoot` 的配置类。

查看@SpringBootConfiguration注解源码，核心代码具体如下。

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented

@Configuration // 配置类的作用等同于配置文件，配置类也是容器中的一个对象
public @interface SpringBootConfiguration {
}

```

从上述源码可以看出，`@SpringBootConfiguration`注解内部有一个核心注解`@Configuration`，该注解是Spring框架提供的，表示当前类为一个配置类（XML配置文件的注解表现形式），并可以被组件扫描器扫描。由此可见，`@SpringBootConfiguration`注解的作用与`@Configuration`注解相同，都是标识一个可以被组件扫描器扫描的配置类，只不过`@SpringBootConfiguration`是被Spring Boot进行了重新封装命名而已

@EnableAutoConfiguration

```

package org.springframework.boot.autoconfigure;

// 自动配置包
@AutoConfigurationPackage

// Spring的底层注解@Import，给容器中导入一个组件;
// 导入的组件是AutoConfigurationPackages.Registrar.class
@Import(AutoConfigurationImportSelector.class)

// 告诉SpringBoot开启自动配置功能，这样自动配置才能生效。
public @interface EnableAutoConfiguration {

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    // 返回不会被导入到 Spring 容器中的类
    Class<?>[] exclude() default {};

    // 返回不会被导入到 Spring 容器中的类名
    String[] excludeName() default {};

}

```

`Spring` 中有很多以 `Enable` 开头的注解，其作用就是借助 `@Import` 来收集并注册特定场景相关的 `Bean`，并加载到 `IoC` 容器。

`@EnableAutoConfiguration` 就是借助 `@Import` 来收集所有符合自动配置条件的 `bean` 定义，并加载到 `IoC` 容器。

@AutoConfigurationPackage

```

package org.springframework.boot.autoconfigure;

@Import(AutoConfigurationPackages.Registrar.class) // 导入Registrar中注册的组件
public @interface AutoConfigurationPackage {

}

```

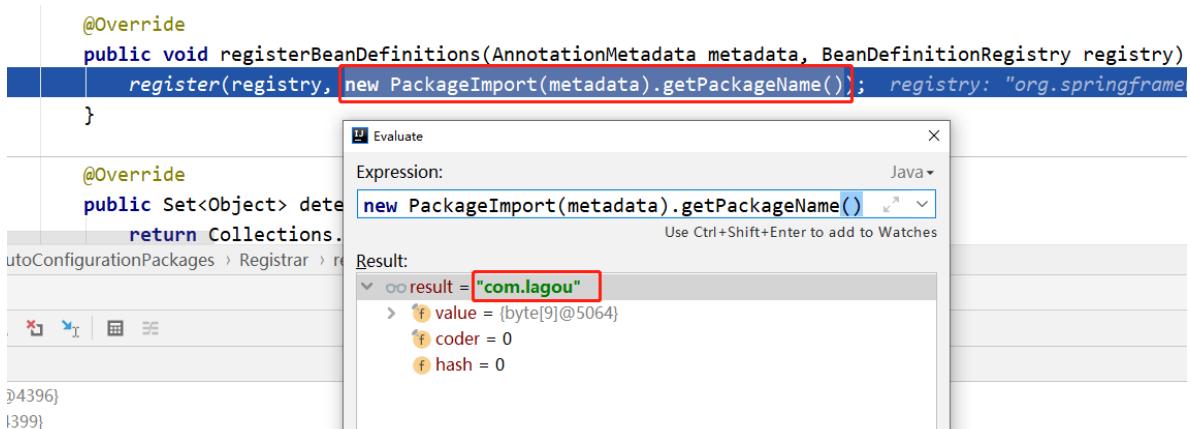
`@AutoConfigurationPackage`：自动配置包，它也是一个组合注解，其中最重要的注解是`@Import(AutoConfigurationPackages.Registrar.class)`，它是 Spring 框架的底层注解，它的作用就是给容器中导入某个组件类，例如`@Import(AutoConfigurationPackages.Registrar.class)`，它就是将 `Registrar` 这个组件类导入到容器中，可查看 `Registrar` 类中 `registerBeanDefinitions` 方法：

```

@Override
public void registerBeanDefinitions(AnnotationMetadata metadata,
BeanDefinitionRegistry registry) {
    // 将注解标注的元信息传入，获取到相应的包名
    register(registry, new PackageImport(metadata).getPackageName());
}

```

我们对 `new PackageImport(metadata).getPackageName()` 进行检索，看看其结果是什么？



再看 `register` 方法

```

public static void register(BeanDefinitionRegistry registry, String...
packageNames) {
    // 这里参数 packageNames 缺省情况下就是一个字符串，是使用了注解
    // @SpringBootApplication 的 Spring Boot 应用程序入口类所在的包

    if (registry.containsBeanDefinition(BEAN)) {
        // 如果该bean已经注册，则将要注册包名称添加进去
        BeanDefinition beanDefinition = registry.getBeanDefinition(BEAN);
        ConstructorArgumentValues constructorArguments = beanDefinition
            .getConstructorArgumentValues();
        constructorArguments.addIndexedArgumentValue(0,
            addBasePackages(constructorArguments, packageNames));
    }
    else {
        //如果该bean尚未注册，则注册该bean，参数中提供的包名称会被设置到bean定义中去
        GenericBeanDefinition beanDefinition = new GenericBeanDefinition();
        beanDefinition.setBeanClass(BasePackages.class);

        beanDefinition.getConstructorArgumentValues().addIndexedArgumentValue(0,

```

```

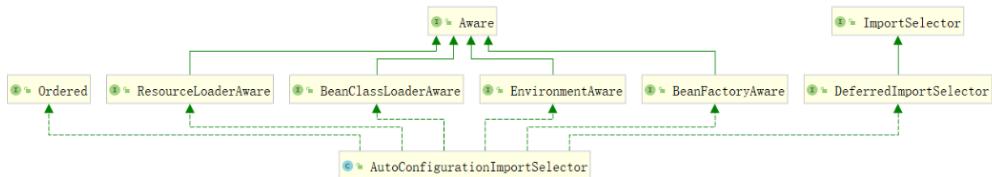
        packageNames);
        beanDefinition.setRole(BeanDefinition.ROLE_INFRASTRUCTURE);
        registry.registerBeanDefinition(BEAN, beanDefinition);
    }
}

```

AutoConfigurationPackages.Registrar这个类就干一个事，注册一个 Bean，这个 Bean 就是 org.springframework.boot.autoconfigure.AutoConfigurationPackages.BasePackages，它有一个参数，这个参数是使用了 @AutoConfigurationPackage 这个注解的类所在的包路径，保存自动配置类以供之后的使用，比如给 JPA entity 扫描器用来扫描开发人员通过注解 @Entity 定义的 entity 类。

@Import(AutoConfigurationImportSelector.class)

@Import({AutoConfigurationImportSelector.class})：将 AutoConfigurationImportSelector 这个类导入到 Spring 容器中，AutoConfigurationImportSelector 可以帮助 springboot 应用将所有符合条件的 @Configuration 配置都加载到当前 SpringBoot 创建并使用的 IOC 容器(ApplicationContext)中。



可以看到 AutoConfigurationImportSelector 重点是实现了 DeferredImportSelector 接口和各种 Aware 接口，然后 DeferredImportSelector 接口又继承了 ImportSelector 接口。

其不光实现了 ImportSelector 接口，还实现了很多其它的 Aware 接口，分别表示在某个时机会被回调。

确定自动配置实现逻辑的入口方法：

跟自动配置逻辑相关的入口方法在 DeferredImportSelectorGrouping 类的 getImports 方法处，因此我们就从 DeferredImportSelectorGrouping 类的 getImports 方法来开始分析SpringBoot的自动配置源码好了。

先看一下 getImports 方法代码：

```

// ConfigurationClassParser.java

public Iterable<Group.Entry> getImports() {
    // 遍历DeferredImportSelectorHolder对象集合deferredImports, deferredImports集合
    // 装了各种ImportSelector, 当然这里装的是AutoConfigurationImportSelector
    for (DeferredImportSelectorHolder deferredImport : this.deferredImports) {
        // 【1】，利用AutoConfigurationGroup的process方法来处理自动配置的相关逻辑，决定
        // 导入哪些配置类（这个是我们分析的重点，自动配置的逻辑全在这了）
        this.group.process(deferredImport.getConfigurationClass().getMetadata(),
                           deferredImport.getImportSelector());
    }
    // 【2】，经过上面的处理后，然后再进行选择导入哪些配置类
    return this.group.selectImports();
}

```

标【1】处的代码是我们分析的重中之重，自动配置的相关绝大部分逻辑全在这里了。那么 `this.group.process(deferredImport.getConfigurationClass().getMetadata(), deferredImport.getImportSelector());` 主要做的事情就是在 `this.group` 即 `AutoConfigurationGroup` 对象的 `process` 方法中，传入的 `AutoConfigurationImportSelector` 对象来选择一些符合条件的自动配置类，过滤掉一些不符合条件的自动配置类，就是这么个事情。

注：

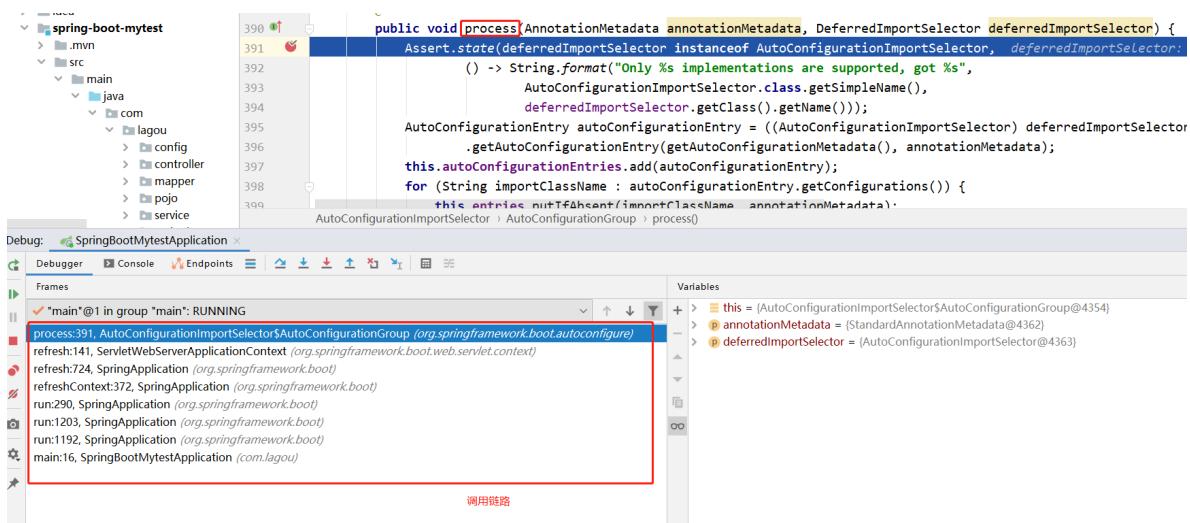
`AutoConfigurationGroup`: 是 `AutoConfigurationImportSelector` 的内部类，主要用来处理自动配置相关的逻辑，拥有 `process` 和 `selectImports` 方法，然后拥有 `entries` 和 `autoConfigurationEntries` 集合属性，这两个集合分别存储被处理后的符合条件的自动配置类，我们知道这些就足够了；

`AutoConfigurationImportSelector`: 承担自动配置的绝大部分逻辑，负责选择一些符合条件的自动配置类；

`metadata`: 标注在 `SpringBoot` 启动类上的 `@SpringBootApplication` 注解元数据

标【2】的 `this.group.selectImports` 的方法主要是针对前面的 `process` 方法处理后的自动配置类再进一步有选择的选择导入

再进入到 `AutoConfigurationImportSelector$AutoConfigurationGroup` 的 `process` 方法：



通过图中我们可以看到，跟自动配置逻辑相关的入口方法在 `process` 方法中

分析自动配置的主要逻辑

```
// AutoConfigurationImportSelector$AutoConfigurationGroup.java

// 这里用来处理自动配置类，比如过滤掉不符合匹配条件的自动配置类
public void process(AnnotationMetadata annotationMetadata,
                    DeferredImportSelector deferredImportSelector) {
    Assert.state(
        deferredImportSelector instanceof AutoConfigurationImportSelector,
        () -> String.format("Only %s implementations are supported, got %s",
            AutoConfigurationImportSelector.class.getSimpleName(),
            deferredImportSelector.getClass().getName()));

    // 【1】，调用getAutoConfigurationEntry方法得到自动配置类放入
    // autoConfigurationEntry对象中
    AutoConfigurationEntry autoConfigurationEntry =
        ((AutoConfigurationImportSelector) deferredImportSelector)
            .getAutoConfigurationEntry(annotationMetadata);
}
```

```

    // 【2】，又将封装了自动配置类的autoConfigurationEntry对象装进
    autoConfigurationEntries集合
    this.autoConfigurationEntries.add(autoConfigurationEntry);
    // 【3】，遍历刚获取的自动配置类
    for (String importClassName : autoConfigurationEntry.getConfigurations()) {
        // 这里符合条件的自动配置类作为key, annotationMetadata作为值放进entries集合
        this.entries.putIfAbsent(importClassName, annotationMetadata);
    }
}

```

上面代码中我们再来看标【1】的方法 `getAutoConfigurationEntry`，这个方法主要是用来获取自动配置类有关，承担了自动配置的主要逻辑。直接上代码：

```

// AutoConfigurationImportSelector.java

// 获取符合条件的自动配置类，避免加载不必要的自动配置类从而造成内存浪费
protected AutoConfigurationEntry getAutoConfigurationEntry(
    AutoConfigurationMetadata autoConfigurationMetadata,
    AnnotationMetadata annotationMetadata) {
    // 获取是否有配置spring.boot.enableautoconfiguration属性，默认返回true
    if (!isEnabled(annotationMetadata)) {
        return EMPTY_ENTRY;
    }
    // 获得@Configuration标注的Configuration类即被审视introspectedClass的注解数据，
    // 比如：@SpringBootApplication(exclude = FreeMarkerAutoConfiguration.class)
    // 将会获取到exclude = FreeMarkerAutoConfiguration.class和excludeName="" 的注解
    // 数据
    AnnotationAttributes attributes = getAttributes(annotationMetadata);
    // 【1】得到spring.factories文件配置的所有自动配置类
    List<String> configurations = getCandidateConfigurations(annotationMetadata,
        attributes);
    // 利用LinkedHashSet移除重复的配置类
    configurations = removeDuplicates(configurations);
    // 得到要排除的自动配置类，比如注解属性exclude的配置类
    // 比如：@SpringBootApplication(exclude = FreeMarkerAutoConfiguration.class)
    // 将会获取到exclude = FreeMarkerAutoConfiguration.class的注解数据
    Set<String> exclusions = getExclusions(annotationMetadata, attributes);
    // 检查要被排除的配置类，因为有些不是自动配置类，故要抛出异常
    checkExcludedClasses(configurations, exclusions);
    // 【2】将要排除的配置类移除
    configurations.removeAll(exclusions);
    // 【3】因为从spring.factories文件获取的自动配置类太多，如果有些不必要的自动配置类都加载
    // 进内存，会造成内存浪费，因此这里需要进行过滤
    // 注意这里会调用AutoConfigurationImportFilter的match方法来判断是否符合
    // @ConditionalOnBean, @ConditionalOnClass 或 @ConditionalOnWebApplication，后面会重点分
    // 析一下
    configurations = filter(configurations, autoConfigurationMetadata);
    // 【4】获取了符合条件的自动配置类后，此时触发AutoConfigurationImportEvent事件，
    // 目的是告诉ConditionEvaluationReport条件评估报告器对象来记录符合条件的自动配置类
    // 该事件什么时候会被触发？--> 在刷新容器时调用invokeBeanFactoryPostProcessors后置处
    // 理器时触发
    fireAutoConfigurationImportEvents(configurations, exclusions);
    // 【5】将符合条件和要排除的自动配置类封装进AutoConfigurationEntry对象，并返回
    return new AutoConfigurationEntry(configurations, exclusions);
}

```

深入 `getCandidateConfigurations` 方法

这个方法中有一个重要方法 `loadFactoryNames`，这个方法是让 `SpringFactoryLoader` 去加载一些组件的名字。

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    // 这个方法需要传入两个参数getSpringFactoriesLoaderFactoryClass()和
    getBeanClassLoader()
    // getSpringFactoriesLoaderFactoryClass()这个方法返回的是
    EnableAutoConfiguration.class
    // getBeanClassLoader()这个方法返回的是beanClassLoader（类加载器）
    List<String> configurations =
    SpringFactoriesLoader.loadFactoryNames(getSpringFactoriesLoaderFactoryClass(),
        getBeanClassLoader());
    Assert.notEmpty(configurations, "No auto configuration classes found in
META-INF/spring.factories. If you "
        + "are using a custom packaging, make sure that file is
correct.");
    return configurations;
```

继续点开 `loadFactory` 方法

```
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable
ClassLoader classLoader) {

    //获取出入的键
    String factoryClassName = factoryClass.getName();
    return
    (List)loadSpringFactories(classLoader).getOrDefault(factoryClassName,
    Collections.emptyList());
}

private static Map<String, List<String>> loadSpringFactories(@Nullable
ClassLoader classLoader) {
    MultiValueMap<String, String> result =
    (MultiValueMap)cache.get(classLoader);
    if (result != null) {
        return result;
    } else {
        try {

            //如果类加载器不为null，则加载类路径下spring.factories文件，将其中设置的
            //配置类的全路径信息封装为Enumeration类对象
            Enumeration<URL> urls = classLoader != null ?
            classLoader.getResources("META-INF/spring.factories") :
            classLoader.getSystemResources("META-INF/spring.factories");
            LinkedMultiValueMap result = new LinkedMultiValueMap();

            //循环Enumeration类对象，根据相应的节点信息生成Properties对象，通过传入
            //的键获取值，在将值切割为一个个小的字符串转化为Array，方法result集合中
            while(urls.hasMoreElements()) {
                URL url = (URL)urls.nextElement();
                UrlResource resource = new UrlResource(url);
                Properties properties =
                PropertiesLoaderUtils.loadProperties(resource);
                Iterator var6 = properties.entrySet().iterator();
```

```

        while(var6.hasNext()) {
            Entry<?, ?> entry = (Entry)var6.next();
            String factoryClassName =
((String)entry.getKey()).trim();
            String[] var9 =
StringUtil.commaDelimitedListToStringArray((String)entry.getValue());
            int var10 = var9.length;

            for(int var11 = 0; var11 < var10; ++var11) {
                String factoryName = var9[var11];
                result.add(factoryClassName, factoryName.trim());
            }
        }
    }

    cache.put(classLoader, result);
    return result;
}
}
}

```

从代码中我们可以知道，在这个方法中会遍历整个ClassLoader中所有jar包下的spring.factories文件。spring.factories里面保存着springboot的默认提供的自动配置类。

META-INF/spring.factories

```

# 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.cloud.CloudServiceConnectorsAutoConfiguration,
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration,
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,

```

AutoConfigurationEntry方法主要做的事情就是获取符合条件的自动配置类，避免加载不必要的自动配置类从而造成内存浪费。我们下面总结下AutoConfigurationEntry方法主要做的事情：

- 【1】从spring.factories配置文件中加载EnableAutoConfiguration自动配置类），获取的自动配置类如图所示。
- 【2】若@EnableAutoConfiguration等注解标有要exclude的自动配置类，那么再将这个自动配置类排除掉；
- 【3】排除掉要exclude的自动配置类后，然后再调用filter方法进行进一步的过滤，再次排除一些不符合条件的自动配置类；
- 【4】经过重重过滤后，此时再触发AutoConfigurationImportEvent事件，告诉ConditionEvaluationReport条件评估报告器对象来记录符合条件的自动配置类；
- 【5】最后再将符合条件的自动配置类返回。

总结了 AutoConfigurationEntry 方法主要的逻辑后，我们再来细看一下

AutoConfigurationImportSelector 的 filter 方法：

```
// AutoConfigurationImportSelector.java

private List<String> filter(List<String> configurations,
                           AutoConfigurationMetadata autoConfigurationMetadata) {
    long startTime = System.nanoTime();
    // 将从spring.factories中获取的自动配置类转出字符串数组
    String[] candidates = StringUtils.toStringArray(configurations);
    // 定义skip数组，是否需要跳过。注意skip数组与candidates数组顺序一一对应
    boolean[] skip = new boolean[candidates.length];
    boolean skipped = false;
    // getAutoConfigurationImportFilters方法：拿到OnBeanCondition,
    OnClassCondition和OnWebApplicationCondition
    // 然后遍历这三个条件类去过滤从spring.factories加载的大量配置类
    for (AutoConfigurationImportFilter filter :
        getAutoConfigurationImportFilters()) {
        // 调用各种aware方法，将beanClassLoader,beanFactory等注入到filter对象中，
        // 这里的filter对象即OnBeanCondition, OnClassCondition或
        OnWebApplicationCondition
        invokeAwareMethods(filter);
        // 判断各种filter来判断每个candidate（这里实质要通过candidate（自动配置类）拿到其标注的
        // @ConditionalOnClass,@ConditionalOnBean和@ConditionalOnWebApplication里面的注解值）是否匹配，
        // 注意candidates数组与match数组一一对应
        //*****【主线，重点关注】*****
        boolean[] match = filter.match(candidates, autoConfigurationMetadata);
        // 遍历match数组，注意match顺序跟candidates的自动配置类一一对应
        for (int i = 0; i < match.length; i++) {
            // 若有不匹配的话
            if (!match[i]) {
                // 不匹配的将记录在skip数组，标志skip[i]为true，也与candidates数组一一
                // 对应
                skip[i] = true;
                // 因为不匹配，将相应的自动配置类置空
                candidates[i] = null;
                // 标注skipped为true
                skipped = true;
            }
        }
    }
    // 这里表示若所有自动配置类经过OnBeanCondition, OnClassCondition和
    OnWebApplicationCondition过滤后，全部都匹配的话，则全部原样返回
    if (!skipped) {
        return configurations;
    }
    // 建立result集合来装匹配的自动配置类
    List<String> result = new ArrayList<>(candidates.length);
    for (int i = 0; i < candidates.length; i++) {
        // 若skip[i]为false，则说明是符合条件的自动配置类，此时添加到result集合中
        if (!skip[i]) {
            result.add(candidates[i]);
        }
    }
    // 打印日志
}
```

```

    if (logger.isTraceEnabled()) {
        int numberFiltered = configurations.size() - result.size();
        logger.trace("Filtered " + numberFiltered + " auto configuration class
in "
                    + TimeUnit.NANOSECONDS.toMillis(System.nanoTime() - startTime)
                    + " ms");
    }
    // 最后返回符合条件的自动配置类
    return new ArrayList<>(result);
}

```

`AutoConfigurationImportSelector` 的 `filter` 方法主要做的事情就是调用 `AutoConfigurationImportFilter` 接口的 `match` 方法来判断每一个自动配置类上的条件注解（若有的话）`@ConditionalOnClass`,`@ConditionalOnBean` 或 `@ConditionalOnWebApplication` 是否满足条件，若满足，则返回`true`，说明匹配，若不满足，则返回`false`说明不匹配。

我们现在知道 `AutoConfigurationImportSelector` 的 `filter` 方法主要做了什么事情就行了，现在先不用研究的过深

关于条件注解的讲解

`@Conditional` 是 Spring 4 新提供的注解，它的作用是按照一定的条件进行判断，满足条件给容器注册 bean。

- `@ConditionalOnBean`: 仅仅在当前上下文中存在某个对象时，才会实例化一个 Bean。
- `@ConditionalOnClass`: 某个 class 位于类路径上，才会实例化一个 Bean。
- `@ConditionalOnExpression`: 当表达式为 true 的时候，才会实例化一个 Bean。基于 SpEL 表达式的条件判断。
- `@ConditionalOnMissingBean`: 仅仅在当前上下文中不存在某个对象时，才会实例化一个 Bean。
- `@ConditionalOnMissingClass`: 某个 class 类路径上不存在的时候，才会实例化一个 Bean。
- `@ConditionalOnNotWebApplication`: 不是 web 应用，才会实例化一个 Bean。
- `@ConditionalOnWebApplication`: 当项目是一个 Web 项目时进行实例化。
- `@ConditionalOnNotWebApplication`: 当项目不是一个 Web 项目时进行实例化。
- `@ConditionalOnProperty`: 当指定的属性有指定的值时进行实例化。
- `@ConditionalOnJava`: 当 JVM 版本为指定的版本范围时触发实例化。
- `@ConditionalOnResource`: 当类路径下有指定的资源时触发实例化。
- `@ConditionalOnJndi`: 在 JNDI 存在的条件下触发实例化。
- `@ConditionalOnSingleCandidate`: 当指定的 Bean 在容器中只有一个，或者有多个但是指定了首选的 Bean 时触发实例化。

有选择的导入自动配置类

`this.group.selectImports` 方法是如何进一步有选择的导入自动配置类的。直接看代码：

```

// AutoConfigurationImportSelector$AutoConfigurationGroup.java

public Iterable<Entry> selectImports() {
    if (this.autoConfigurationEntries.isEmpty()) {
        return Collections.emptyList();
    }
    // 这里得到所有要排除的自动配置类的 set 集合
    Set<String> allExclusions = this.autoConfigurationEntries.stream()
        .map(AutoConfigurationEntry::getExclusions)

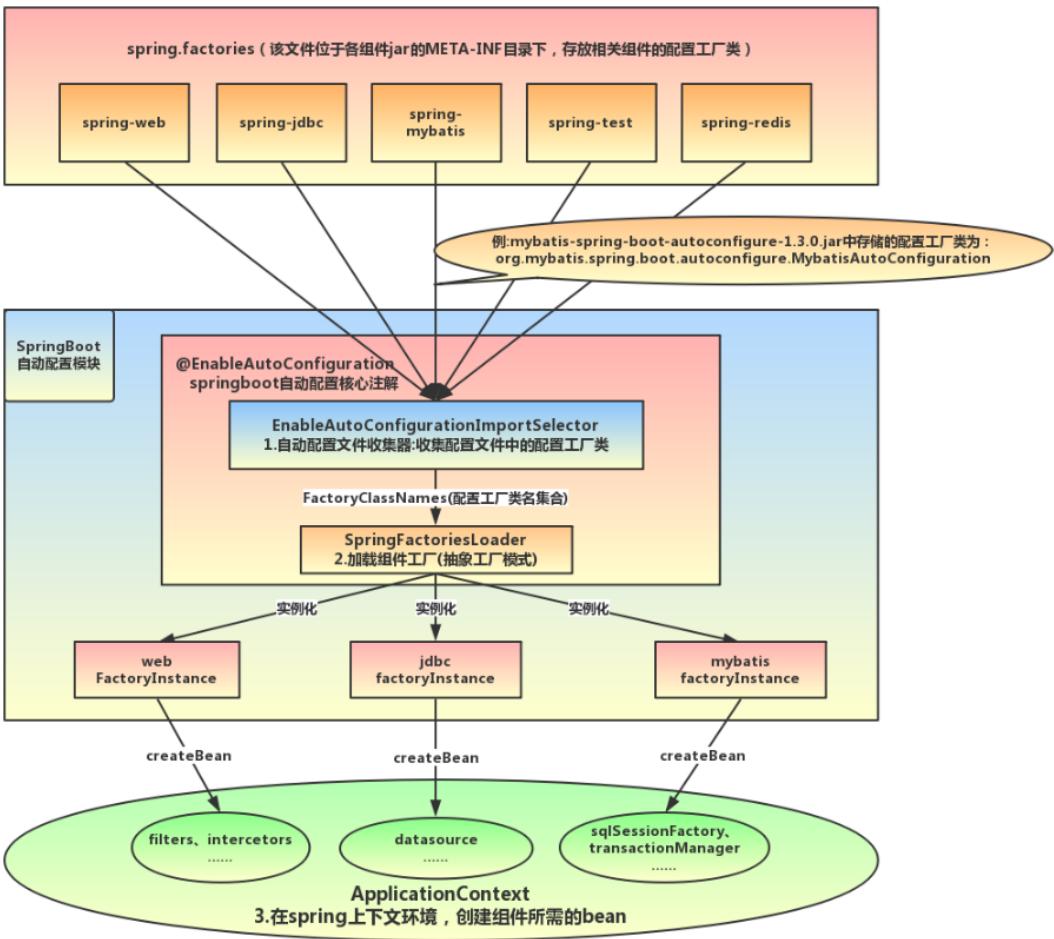
```

```
.flatMap(Collection::stream).collect(Collectors.toSet());
// 这里得到经过滤后所有符合条件的自动配置类的set集合
Set<String> processedConfigurations = this.autoConfigurationEntries.stream()
    .map(AutoConfigurationEntry::getConfigurations)
    .flatMap(Collection::stream)
    .collect(Collectors.toCollection(LinkedHashSet::new));
// 移除掉要排除的自动配置类
processedConfigurations.removeAll(allExclusions);
// 对标注有@Order注解的自动配置类进行排序,
return sortAutoConfigurations(processedConfigurations,
    getAutoConfigurationMetadata())
    .stream()
    .map((importClassName) -> new Entry(
        this.entries.get(importClassName), importClassName))
    .collect(Collectors.toList());
}
```

可以看到，`selectImports`方法主要是针对经过排除掉`exclude`的和被`AutoConfigurationImportFilter`接口过滤后的满足条件的自动配置类再进一步排除`exclude`的自动配置类，然后再排序

最后，我们再总结下SpringBoot自动配置的原理，主要做了以下事情：

1. 从`spring.factories`配置文件中加载自动配置类；
2. 加载的自动配置类中排除掉`@EnableAutoConfiguration`注解的`exclude`属性指定的自动配置类；
3. 然后再用`AutoConfigurationImportFilter`接口去过滤自动配置类是否符合其标注注解（若有标注的话）`@ConditionalOnClass`,`@ConditionalOnBean`和`@ConditionalOnWebApplication`的条件，若都符合的话则返回匹配结果；
4. 然后触发`AutoConfigurationImportEvent`事件，告诉`ConditionEvaluationReport`条件评估报告器对象来分别记录符合条件和`exclude`的自动配置类。
5. 最后Spring再将最后筛选后的自动配置类导入IOC容器中



以 `HttpEncodingAutoConfiguration` (Http 编码自动配置) 为例解释自动配置原理

```
// 表示这是一个配置类，和以前编写的配置文件一样，也可以给容器中添加组件
@Configuration

// 启动指定类的ConfigurationProperties功能；将配置文件中对应的值和
// HttpEncodingProperties绑定起来；
@EnableConfigurationProperties({HttpEncodingProperties.class})

// Spring底层@Conditional注解，根据不同的条件，如果满足指定的条件，整个配置类里面的配置就会
// 生效。
// 判断当前应用是否是web应用，如果是，当前配置类生效。并把HttpEncodingProperties加入到 ioc
// 容器中
@ConditionalOnWebApplication

// 判断当前项目有没有这个CharacterEncodingFilter：SpringMVC中进行乱码解决的过滤器
@ConditionalOnClass({CharacterEncodingFilter.class})

// 判断配置文件中是否存在某个配置 spring.http.encoding.enabled 如果不存在，判断也是成立的
// matchIfMissing = true 表示即使我们配置文件中不配置
// spring.http.encoding.enabled=true，也是默认生效的
@ConditionalOnProperty(
    prefix = "spring.http.encoding",
    value = {"enabled"},
    matchIfMissing = true
)
```

```

public class HttpEncodingAutoConfiguration {

    // 它已经和SpringBoot配置文件中的值进行映射了
    private final HttpEncodingProperties properties;

    // 只有一个有参构造器的情况下，参数的值就会从容器中拿
    public HttpEncodingAutoConfiguration(HttpEncodingProperties properties) {
        this.properties = properties;
    }

    @Bean    //给容器中添加一个组件，这个组件中的某些值需要从properties中获取
    @ConditionalOnMissingBean({CharacterEncodingFilter.class}) //判断容器中没有这个组件
    public CharacterEncodingFilter characterEncodingFilter() {
        CharacterEncodingFilter filter = new OrderedCharacterEncodingFilter();
        filter.setEncoding(this.properties.getCharset().name());

        filter.setForceRequestEncoding(this.properties.shouldForce(Type.REQUEST));
        filter.setForceResponseEncoding(this.properties.shouldForce(Type.RESPONSE));
        return filter;
    }
}

```

根据当前不同的条件判断，决定这个配置类是否生效。

一旦这个配置类生效，这个配置类就会给容器中添加各种组件；这些组件的属性是从对应的 properties 类中获取的，这些类里面的每一个属性又是和配置文件绑定的。

```

# 我们能配置的属性都是来源于这个功能的properties类
spring.http.encoding.enabled=true
spring.http.encoding.charset=utf-8
spring.http.encoding.force=true

```

所有在配置文件中能配置的属性都是在 `xxxProperties` 类中封装着，配置文件能配置什么就可以参照某个功能对应的这个属性类。

```

// 从配置文件中获取指定的值和bean的属性进行绑定
@ConfigurationProperties(prefix = "spring.http.encoding")
public class HttpEncodingProperties {
    public static final Charset DEFAULT_CHARSET = Charset.forName("UTF-8");
}

```

精髓

1. SpringBoot 启动会加载大量的自动配置类
2. 我们看我们需要实现的功能有没有 SpringBoot 默认写好的自动配置类
3. 我们再来看这个自动配置类中到底配置了哪些组件；（只要我们有我们要用的组件，我们就不需要再来配置了）
4. 给容器中自动配置类添加组件的时候，会从 `properties` 类中获取某些属性，我们就可以在配置文件中指定这些属性的值。

`xxxAutoConfiguration`：自动配置类，用于给容器中添加组件从而代替之前我们手动完成大量繁琐的配置。

`xxxProperties`：封装了对应自动配置类的默认属性值，如果我们需要自定义属性值，只需要根据 `xxxProperties` 寻找相关属性在配置文件设值即可。

@ComponentScan注解

@ComponentScan使用

主要是从定义的扫描路径中，找出标识了需要装配的类自动装配到spring 的bean容器中。

常用属性如下：

- basePackages、value：指定扫描路径，如果为空则以@ComponentScan注解的类所在的包为基本的扫描路径
- basePackageClasses：指定具体扫描的类
- includeFilters：指定满足Filter条件的类
- excludeFilters：指定排除Filter条件的类

includeFilters和excludeFilters 的FilterType可选：ANNOTATION=注解类型 默认、ASSIGNABLE_TYPE(指定固定类)、ASPECTJ(ASPECTJ类型)、REGEX(正则表达式)、CUSTOM(自定义类型)，自定义的Filter需要实现TypeFilter接口

@ComponentScan的配置如下：

```
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
                                  @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
```

借助excludeFilters将TypeExcludeFilter及FilterType这两个类进行排除

当前@ComponentScan注解没有标注basePackages及value，所以扫描路径默认为@ComponentScan注解的类所在的包为基本的扫描路径（也就是标注了@SpringBootApplication注解的项目启动类所在的路径）

抛出疑问：@EnableAutoConfiguration注解是通过@Import注解加载了自动配置固定的bean

@ComponentScan注解自动进行注解扫描

那么真正根据包扫描，把组件类生成实例对象存到IOC容器中，又是怎么来完成的？

2.4 源码剖析-Run方法执行流程

SpringBoot项目的main函数

```
@SpringBootApplication //来标注一个主程序类，说明这是一个Spring Boot应用
public class MyTestMVCAplication {

    public static void main(String[] args) {
        SpringApplication.run(MyTestMVCAplication.class, args);
    }
}
```

点进run方法

```

public static ConfigurableApplicationContext run(Class<?> primarySource,
String... args) {
    // 调用重载方法
    return run(new Class<?>[] { primarySource }, args);
}

public static ConfigurableApplicationContext run(Class<?>[] primarySources,
String[] args) {
    // 两件事：1. 初始化SpringApplication 2. 执行run方法
    return new SpringApplication(primarySources).run(args);
}

```

SpringApplication() 构造方法

继续查看源码，`SpringApplication` 实例化过程，首先是进入带参数的构造方法，最终回到两个参数的构造方法。

```

public SpringApplication(Class<?>... primarySources) {
    this(null, primarySources);
}

@SuppressWarnings({"unchecked", "rawtypes"})
public SpringApplication(ResourceLoader resourceLoader, Class<?>...
primarySources) {
    // 设置资源加载器为null
    this.resourceLoader = resourceLoader;

    // 断言加载资源类不能为null
    Assert.notNull(primarySources, "PrimarySources must not be null");

    // 将primarySources数组转换为List，最后放到LinkedHashSet集合中
    this.primarySources = new LinkedHashSet<>
((Arrays.asList(primarySources)));

    // 【1.1 推断应用类型，后面会根据类型初始化对应的环境。常用的一般都是servlet环境】
    this.webApplicationType = webApplicationType.deduceFromClasspath();

    // 【1.2 初始化classpath下 META-INF/spring.factories 中已配置的
    ApplicationContextInitializer】
    setInitializers((Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));

    // 【1.3 初始化classpath下所有已配置的 ApplicationListener】
    setListeners((Collection)
getSpringFactoriesInstances(ApplicationListener.class));

    // 【1.4 根据调用栈，推断出 main 方法的类名】
    this.mainApplicationClass = deduceMainApplicationClass();
}

```

deduceWebApplicationType();

```
// 常量值
```

```

    private static final String[] WEB_ENVIRONMENT_CLASSES =
    {"javax.servlet.Servlet",
     "org.springframework.web.context.ConfigurableWebApplicationContext"};

    private static final String REACTIVE_WEB_ENVIRONMENT_CLASS =
    "org.springframework."
        + "web.reactive.DispatcherHandler";

    private static final String MVC_WEB_ENVIRONMENT_CLASS =
    "org.springframework."
        + "web.servlet.DispatcherServlet";

    private static final String JERSEY_WEB_ENVIRONMENT_CLASS =
    "org.glassfish.jersey.server.ResourceConfig";

    /**
     * 判断 应用的类型
     * NONE: 应用程序不是web应用，也不应该用web服务器去启动
     * SERVLET: 应用程序应作为基于servlet的web应用程序运行，并应启动嵌入式servlet
     * web (tomcat) 服务器。
     * REACTIVE: 应用程序应作为 reactive web应用程序运行，并应启动嵌入式 reactive web服
     * 务器。
     * @return
     */
    private WebApplicationType deduceWebApplicationType() {
        //classpath下必须存在org.springframework.web.reactive.DispatcherHandler
        if (ClassUtils.isPresent(REACTIVE_WEB_ENVIRONMENT_CLASS, null)
            && !ClassUtils.isPresent(MVC_WEB_ENVIRONMENT_CLASS, null)
            && !ClassUtils.isPresent(JERSEY_WEB_ENVIRONMENT_CLASS, null))
        {
            return WebApplicationType.REACTIVE;
        }
        for (String className : WEB_ENVIRONMENT_CLASSES) {
            if (!ClassUtils.isPresent(className, null)) {
                return WebApplicationType.NONE;
            }
        }
        //classpath环境下存在javax.servlet.Servlet或者
        org.springframework.web.context.ConfigurableWebApplicationContext
        return WebApplicationType.SERVLET;
    }
}

```

返回类型是WebApplicationType的枚举类型， WebApplicationType 有三个枚举， 三个枚举的解释如其中注释

具体的判断逻辑如下：

- WebApplicationType.REACTIVE classpath下存在
org.springframework.web.reactive.DispatcherHandler
- WebApplicationType.SERVLET classpath下存在javax.servlet.Servlet或者
org.springframework.web.context.ConfigurableWebApplicationContext
- WebApplicationType.NONE 不满足以上条件。

```
setInitializers(Collection)
getSpringFactoriesInstances(ApplicationContextInitializer.class));
```

初始化classpath下 META-INF/spring.factories中已配置的ApplicationContextInitializer

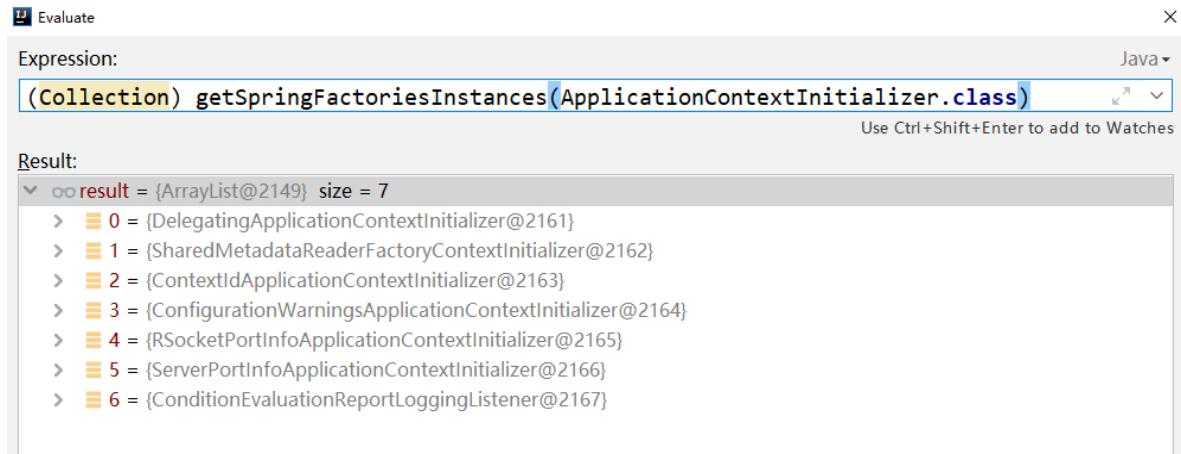
```
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type) {
    return getSpringFactoriesInstances(type, new Class<?>[] {});
}

/**
 * 通过指定的classloader 从META-INF/spring.factories获取指定的Spring的工厂实例
 * @param type
 * @param parameterTypes
 * @param args
 * @param <T>
 * @return
 */
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
                                                       Class<?>[] parameterTypes,
                                                       Object... args) {
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    // Use names and ensure unique to protect against duplicates
    //通过指定的classLoader从 META-INF/spring.factories 的资源文件中,
    //读取 key 为 type.getName() 的 value
    Set<String> names = new LinkedHashSet<>
(SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    //创建Spring工厂实例
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
                                                       classLoader, args, names);
    //对Spring工厂实例排序 (org.springframework.core.annotation.Order注解指定的顺序)
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

看看 getSpringFactoriesInstances 都干了什么，看源码，有一个方法很重要 loadFactoryNames() 这个方法很重要，这个方法是spring-core中提供的从META-INF/spring.factories中获取指定的类 (key) 的同一入口方法。

在这里，获取的是key为 org.springframework.context.ApplicationContextInitializer 的类。

debug看看都获取到了哪些



上面说了，是从classpath下 META-INF/spring.factories中获取，我们验证一下：

```

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

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

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

# Run Listeners
org.springframework.boot.SpringApplicationRunListener=
org.springframework.boot.context.event.EventPublishingRunListener

# Error Reporters
org.springframework.boot.SpringApplicationExceptionReporter=
org.springframework.boot.diagnostics.FailureAnalyzers

# Application Context Initializers
org.springframework.context.ApplicationContextInitializer=
org.springframework.boot.context.ConfigurationWarningsApplicationContextInitializer,\
org.springframework.boot.context.ContextIdApplicationContextInitializer,\
org.springframework.boot.context.config.DelegatingApplicationContextInitializer,\
org.springframework.boot.rsocket.context.RSocketPortInfoApplicationContextInitializer,\
org.springframework.boot.web.context.ServerPortInfoApplicationContextInitializer

# Application Listeners

```

发现在上图所示的两个工程中找到了debug中看到的结果。

`ApplicationContextInitializer` 是Spring框架的类，这个类的主要目的就是在`ConfigurableApplicationContext` 调用`refresh()`方法之前，回调这个类的`initialize`方法。

通过`ConfigurableApplicationContext` 的实例获取容器的环境`Environment`，从而实现对配置文件的修改完善等工作。

setListeners(Collection)

`getSpringFactoriesInstances(ApplicationListener.class);`

初始化classpath下 META-INF/spring.factories中已配置的 ApplicationListener。

ApplicationListener 的加载过程和上面的 ApplicationContextInitializer 类的加载过程是一样的。不多说了，至于 ApplicationListener 是spring的事件监听器，典型的观察者模式，通过 ApplicationEvent 类和 ApplicationListener 接口，可以实现对spring容器全生命周期的监听，当然也可以自定义监听事件

总结

关于 SpringApplication 类的构造过程，到这里我们就梳理完了。纵观 SpringApplication 类的实例化过程，我们可以看到，合理的利用该类，我们能在spring容器创建之前做一些预备工作，和定制化的需求。

比如，自定义SpringBoot的Banner，比如自定义事件监听器，再比如在容器refresh之前通过自定义 ApplicationContextInitializer 修改配置一些配置或者获取指定的bean都是可以的

run(args)

上一小节我们查看了SpringApplication类的实例化过程，这一小节总结SpringBoot启动流程最重要的部分run方法。通过run方法梳理出SpringBoot启动的流程，

经过深入分析后，大家会发现SpringBoot也就是给Spring包了一层皮，事先替我们准备好Spring所需要的环境及一些基础

```
/**  
 * Run the Spring application, creating and refreshing a new  
 * {@link ApplicationContext}.  
 *  
 * @param args the application arguments (usually passed from a Java main  
 * method)  
 * @return a running {@link ApplicationContext}  
 *  
 * 运行spring应用，并刷新一个新的 ApplicationContext (Spring的上下文)  
 * ConfigurableApplicationContext 是 ApplicationContext 接口的子接口。在  
 ApplicationContext  
 * 基础上增加了配置上下文的工具。 ConfigurableApplicationContext是容器的高级接口  
 */  
public ConfigurableApplicationContext run(String... args) {  
    //记录程序运行时间  
    Stopwatch stopwatch = new Stopwatch();  
    stopwatch.start();  
    // ConfigurableApplicationContext Spring 的上下文  
    ConfigurableApplicationContext context = null;  
    Collection<SpringBootExceptionReporter> exceptionReporters = new ArrayList<>()  
    ;  
    configureHeadlessProperty();  
    //从META-INF/spring.factories中获取监听器  
    //1、获取并启动监听器  
    SpringApplicationRunListeners listeners = getRunListeners(args);  
    listeners.starting();  
    try {  
        ApplicationArguments applicationArguments = new  
DefaultApplicationArguments(  
            args);  
        //2、构造应用上下文环境  
        ConfigurableEnvironment environment = prepareEnvironment(listeners,  
applicationArguments);  
        //处理需要忽略的Bean  
        configureIgnoreBeanInfo(environment);  
        //打印banner  
        Banner printedBanner = printBanner(environment);  
        //3、初始化应用上下文  
        context = createApplicationContext();  
        //实例化SpringBootExceptionReporter.class, 用来支持报告关于启动的错误  
        exceptionReporters = getSpringFactoriesInstances(  
            SpringBootExceptionReporter.class,  
            new Class[]{ConfigurableApplicationContext.class}, context);  
        //4、刷新应用上下文前的准备阶段  
        prepareContext(context, environment, listeners, applicationArguments,  
printedBanner);  
        //5、刷新应用上下文  
        refreshContext(context);  
        //刷新应用上下文后的扩展接口  
        afterRefresh(context, applicationArguments);  
        //时间记录停止
```

```

        stopWatch.stop();
        if (this.logStartupInfo) {
            new StartupInfoLogger(this.mainApplicationClass)
                .logStarted(getApplicationLog(), stopWatch);
        }
        //发布容器启动完成事件
        listeners.started(context);
        callRunners(context, applicationArguments);
    } catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, listeners);
        throw new IllegalStateException(ex);
    }

    try {
        listeners.running(context);
    } catch (Throwable ex) {
        handleRunFailure(context, ex, exceptionReporters, null);
        throw new IllegalStateException(ex);
    }
    return context;
}

```

在以上的代码中，启动过程中的重要步骤共分为六步

- 第一步：获取并启动监听器
- 第二步：构造应用上下文环境
- 第三步：初始化应用上下文
- 第四步：刷新应用上下文前的准备阶段
- 第五步：刷新应用上下文
- 第六步：刷新应用上下文后的扩展接口

OK，下面SpringBoot的启动流程分析，我们就根据这6大步骤进行详细解读。最总要的是第四，五步。我们会着重的分析。

第一步：获取并启动监听器

事件机制在Spring是很重要的一部分内容，通过事件机制我们可以监听Spring容器中正在发生的一些事件，同样也可以自定义监听事件。Spring的事件为Bean和Bean之间的消息传递提供支持。当一个对象处理完某种任务后，通知另外的对象进行某些处理，常用的场景有进行某些操作后发送通知，消息、邮件等情况。

```

private SpringApplicationRunListeners getRunListeners(String[] args) {
    Class<?>[] types = new Class<?>[]{SpringApplication.class, String[].class};
    return new SpringApplicationRunListeners(logger,
getSpringFactoriesInstances(
            SpringApplicationRunListener.class, types, this, args));
}

```

在这里面是不是看到一个熟悉的方法：getSpringFactoriesInstances()，可以看下面的注释，前面的小节我们已经详细介绍过该方法是怎么一步步的获取到META-INF/spring.factories中的指定的key的value，获取到以后怎么实例化类的。

```

/**
 * 通过指定的classloader 从META-INF/spring.factories获取指定的Spring的工厂实例
 * @param type

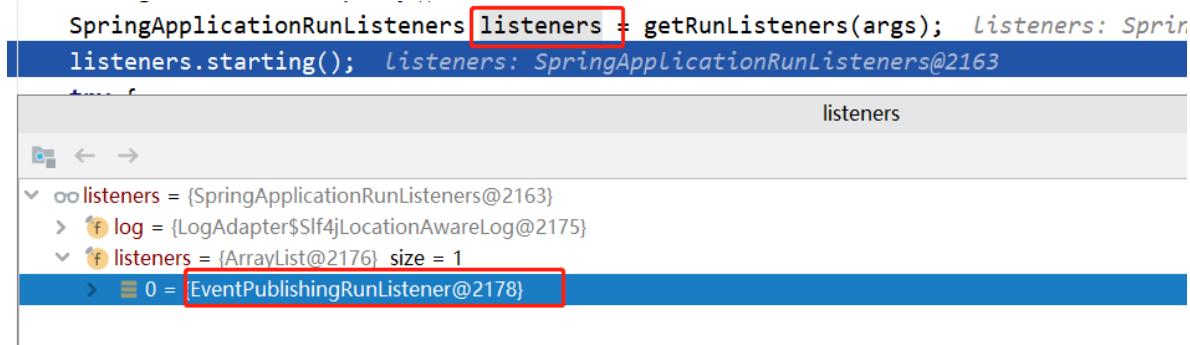
```

```

* @param parameterTypes
* @param args
* @param <T>
* @return
*/
private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
                                                       Class<?>[] parameterTypes,
                                                       Object... args) {
    ClassLoader classLoader = Thread.currentThread().getContextClassLoader();
    // Use names and ensure unique to protect against duplicates
    // 通过指定的classLoader从 META-INF/spring.factories 的资源文件中,
    // 读取 key 为 type.getName() 的 value
    Set<String> names = new LinkedHashSet<>
(SpringFactoriesLoader.loadFactoryNames(type, classLoader));
    // 创建Spring工厂实例
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
                                                       classLoader, args, names);
    // 对Spring工厂实例排序 (org.springframework.core.annotation.Order注解指定的顺序)
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}

```

回到run方法，debug这个代码 SpringApplicationRunListeners listeners = getRunListeners(args); 看一下获取的是哪个监听器：



EventPublishingRunListener监听器是Spring容器的启动监听器。

listeners.starting(); 开启了监听事件。

第二步：构造应用上下文环境

应用上下文环境包括什么呢？包括计算机的环境，Java环境，Spring的运行环境，Spring项目的配置（在SpringBoot中就是那个熟悉的application.properties/yml）等等。

首先看一下prepareEnvironment()方法。

```

private ConfigurableEnvironment prepareEnvironment(
    SpringApplicationRunListeners listeners,
    ApplicationArguments applicationArguments) {
    // Create and configure the environment
    // 创建并配置相应的环境
    ConfigurableEnvironment environment = getOrCreateEnvironment();
    // 根据用户配置，配置 environment 系统环境
    configureEnvironment(environment, applicationArguments.getSourceArgs());
    // 启动相应的监听器，其中一个重要的监听器 ConfigFileApplicationListener 就是加载项目
    // 配置文件的监听器。
}

```

```
listeners.environmentPrepared(environment);
bindToSpringApplication(environment);
if (this.webApplicationType == WebApplicationType.NONE) {
    environment = new EnvironmentConverter(getClassLoader())
        .convertToStandardEnvironmentIfNecessary(environment);
}
ConfigurationPropertySources.attach(environment);
return environment;
}
```

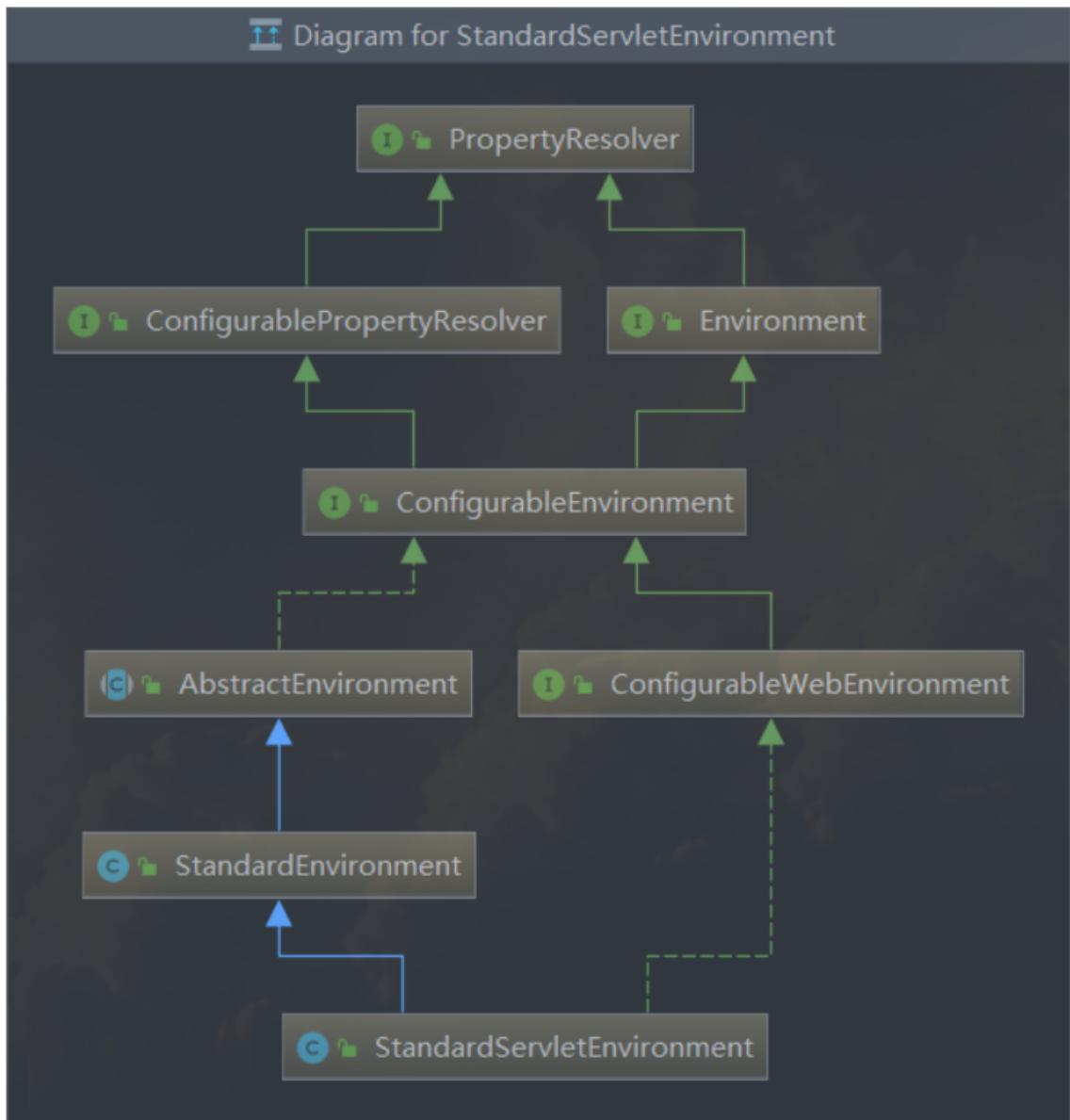
看上面的注释，方法中主要完成的工作，首先是创建并按照相应的应用类型配置相应的环境，然后根据用户的配置，配置系统环境，然后启动监听器，并加载系统配置文件。

ConfigurableEnvironment environment = getOrCreateEnvironment();

看看getOrCreateEnvironment()干了些什么。

```
private ConfigurableEnvironment getOrCreateEnvironment() {
    if (this.environment != null) {
        return this.environment;
    }
    //如果应用类型是 SERVLET 则实例化 StandardServletEnvironment
    if (this.webApplicationType == WebApplicationType.SERVLET) {
        return new StandardServletEnvironment();
    }
    return new StandardEnvironment();
}
```

通过代码可以看到根据不同的应用类型初始化不同的系统环境实例。前面咱们已经说过应用类型是怎么判断的了，这里就不在赘述了



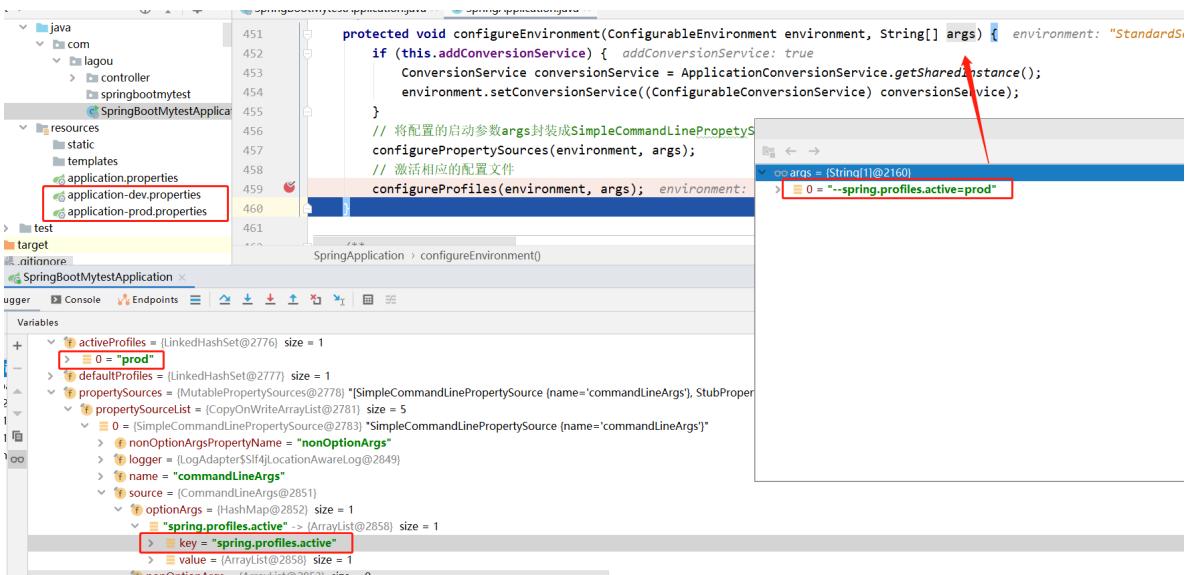
从上面的继承关系可以看出，StandardServletEnvironment是StandardEnvironment的子类。这两个对象也没什么好讲的，当是web项目的时候，环境上会多一些关于web环境的配置。

`configureEnvironment(environment, applicationArguments.getSourceArgs());`

```

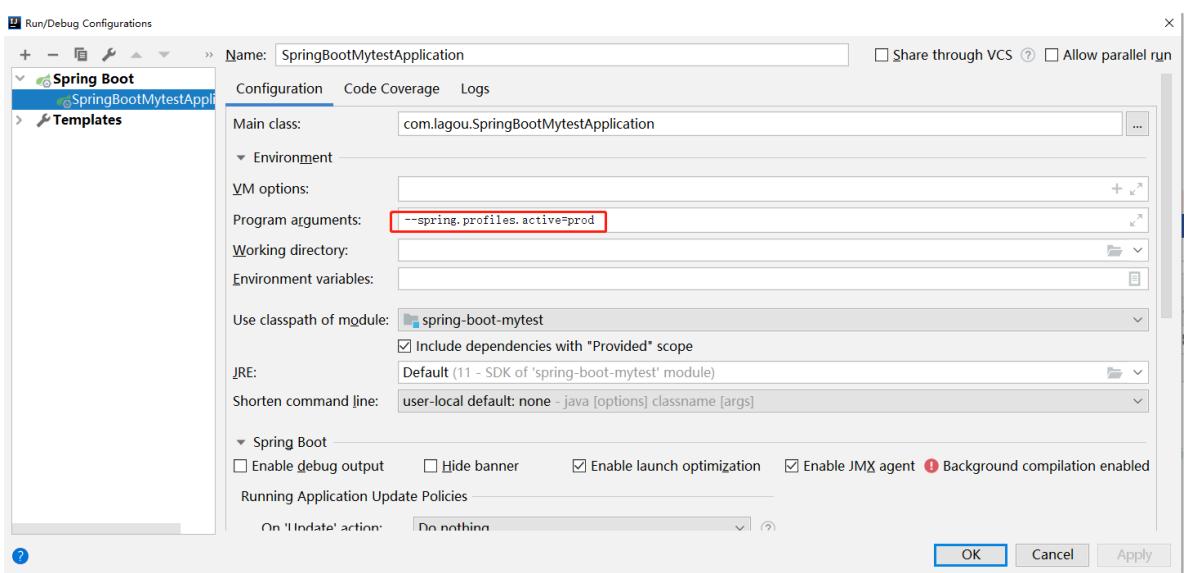
protected void configureEnvironment(ConfigurableEnvironment environment,
                                    String[] args) {
    // 将main 函数的args封装成 SimpleCommandLinePropertySource 加入环境中。
    configurePropertySources(environment, args);
    // 激活相应的配置文件
    configureProfiles(environment, args);
}
  
```

在执行完方法中的两行代码后，debug的截图如下



如下图所示，我在spring的启动参数中指定了参数：--spring.profiles.active=prod

(就是启动多个实例用的)



在configurePropertySources(environment, args);中将args封装成了SimpleCommandLinePropertySource并加入到了environment中。

configureProfiles(environment, args);根据启动参数激活了相应的配置文件。

listeners.environmentPrepared(environment);

进入到方法一路跟下去就到了SimpleApplicationEventMulticaster类的multicastEvent()方法。

```
--> SimpleApplicationEventMulticaster

public void multicastEvent(ApplicationEvent event) {
    multicastEvent(event, resolveDefaultEventType(event));
}
```

```

@Override
public void multicastEvent(final ApplicationEvent event, @Nullable ResolvableType eventType) { event: "org.springfram
    ResolvableType type = (eventType != null ? eventType : resolveDefaultEventType(event)); type: "org.spring;
    Executor executor = getTaskExecutor(); executor: null
    for (ApplicationListener<?> listener : getApplicationListeners(event, type)) { event: "org.springframew
        if (executor != null)
            executor.evaluate(listener);
        else {
            invokeListener(listener);
        }
    }
}

private ResolvableTy
simpleApplicationEventMulticaste

```

Expression: `getApplicationListeners(event, type)`

Result:

- 0 = {ConfigFileApplicationListener@3004}
- 1 = {AnsiOutputApplicationListener@3005}
- 2 = {LoggingApplicationListener@3006}
- 3 = {ClasspathLoggingApplicationListener@3007}
- 4 = {BackgroundPreinitializer@3008}
- 5 = {DelegatingApplicationListener@3009}
- 6 = {FileEncodingApplicationListener@3010}

查看`getApplicationListeners(event, type)`执行结果，发现一个重要的监听器
ConfigFileApplicationListener。

先看看这个类的注释

```

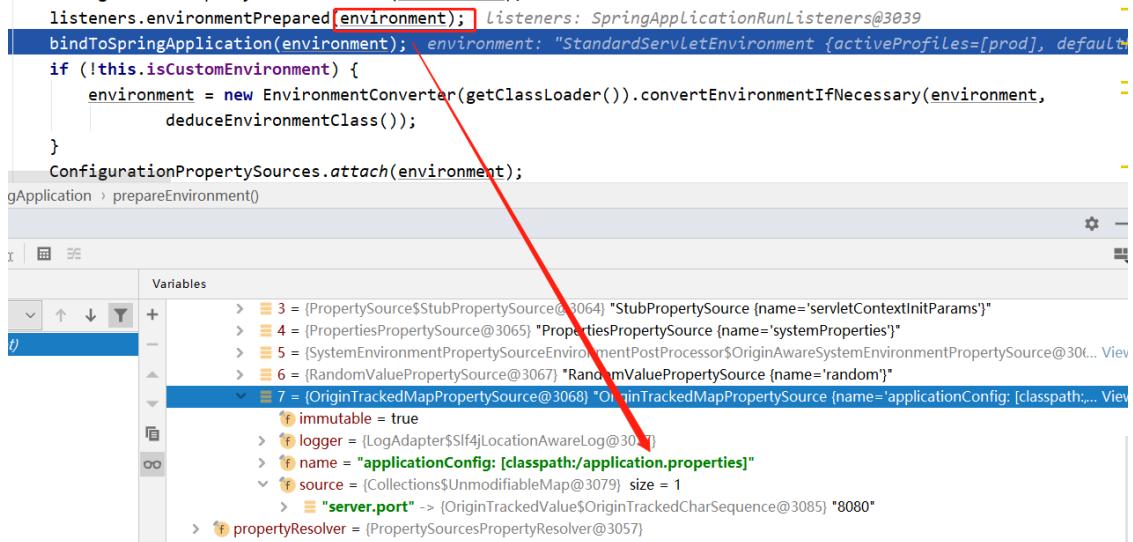
* {@link EnvironmentPostProcessor} that configures the context environment by
loading
* properties from well known file locations. By default properties will be
loaded from
* 'application.properties' and/or 'application.yml' files in the following
locations:
* <ul>
* <li>file:./config</li>
* <li>file:.</li>
* <li>classpath:config</li>
* <li>classpath:</li>
* </ul>
* The list is ordered by precedence (properties defined in locations higher in
the list
* override those defined in lower locations).
* <p>
* Alternative search locations and names can be specified using
* {@link #setSearchLocations(String)} and {@link #setSearchNames(String)}.
* <p>
* Additional files will also be loaded based on active profiles. For example if
a 'web'
* profile is active 'application-web.properties' and 'application-web.yml' will
be
* considered.
* <p>
* The 'spring.config.name' property can be used to specify an alternative name
to load
* and the 'spring.config.location' property can be used to specify alternative
search
* locations or specific files.
* <p>
* 从默认的位置加载配置文件，将其加入上下文的environment变量中

```

这个监听器默认的从注释中

标签所示的几个位置加载配置文件，并将其加入上下文的 environment 变量中。当然也可以通过配置指定。

debug 跳过 listeners.environmentPrepared(environment); 这一行，查看 environment 属性，果真如上面所说的，配置文件的配置信息已经添加上来了。



第三步：初始化应用上下文

在 SpringBoot 工程中，应用类型分为三种，如下代码所示。

```
public enum WebApplicationType {  
    /**  
     * 应用程序不是 web 应用，也不应该用 web 服务器去启动  
     */  
    NONE,  
    /**  
     * 应用程序应作为基于 servlet 的 web 应用程序运行，并应启动嵌入式 servlet  
     * web (tomcat) 服务器。  
     */  
    SERVLET,  
    /**  
     * 应用程序应作为 reactive web 应用程序运行，并应启动嵌入式 reactive web 服务器。  
     */  
    REACTIVE  
}
```

对应三种应用类型，SpringBoot 项目有三种对应的应用上下文，我们以 web 工程为例，即其上下文为 AnnotationConfigServletWebServerApplicationContext。

```
public static final String DEFAULT_WEB_CONTEXT_CLASS =  
    "org.springframework.boot."  
    +  
    "web.servlet.context.AnnotationConfigServletWebServerApplicationContext";  
public static final String DEFAULT_REACTIVE_WEB_CONTEXT_CLASS =  
    "org.springframework."  
    +  
    "boot.web.reactive.context.AnnotationConfigReactiveWebServerApplicationConte  
xt";
```

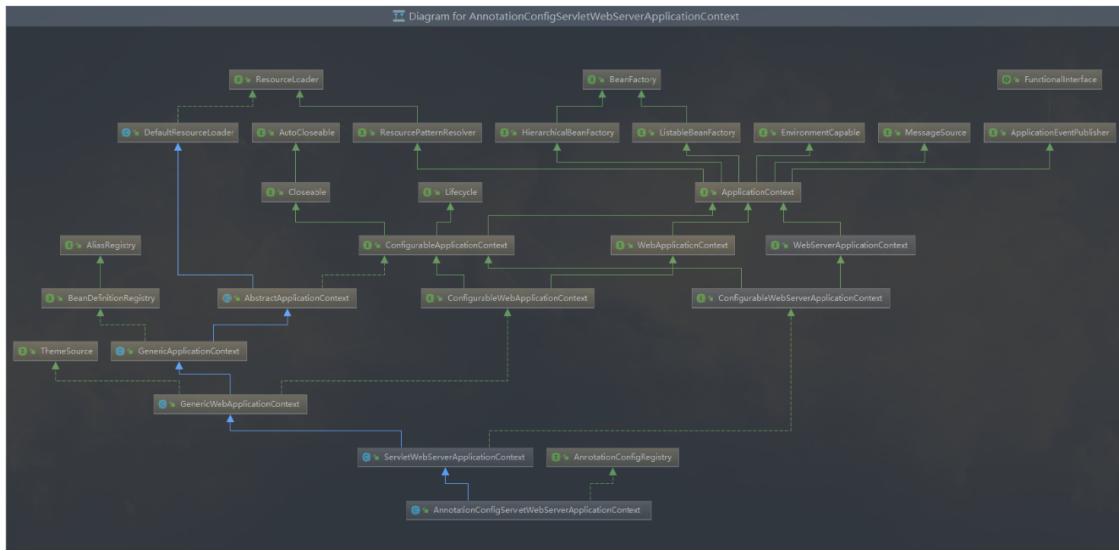
```

public static final String DEFAULT_CONTEXT_CLASS =
"org.springframework.context."
    + "annotation.AnnotationConfigApplicationContext";

protected ConfigurableApplicationContext createApplicationContext() {
    Class<?> contextClass = this.applicationContextClass;
    if (contextClass == null) {
        try {
            switch (this.webApplicationType) {
                case SERVLET:
                    contextClass = Class.forName(DEFAULT_WEB_CONTEXT_CLASS);
                    break;
                case REACTIVE:
                    contextClass =
Class.forName(DEFAULT_REACTIVE_WEB_CONTEXT_CLASS);
                    break;
                default:
                    contextClass = Class.forName(DEFAULT_CONTEXT_CLASS);
            }
        } catch (ClassNotFoundException ex) {
            throw new IllegalStateException(
                "Unable create a default ApplicationContext, "
                + "please specify an ApplicationContextClass",
                ex);
        }
    }
    return (ConfigurableApplicationContext)
BeanUtils.instantiateClass(contextClass);
}

```

我们先看一下AnnotationConfigServletWebServerApplicationContext的设计



应用上下文可以理解成IoC容器的高级表现形式，应用上下文确实在IoC容器的基础上丰富了一些高级功能。

应用上下文对IoC容器是持有的关系。他的一个属性beanFactory就是IoC容器（DefaultListableBeanFactory）。所以他们之间是持有，和扩展的关系。

接下来看GenericApplicationContext类

```

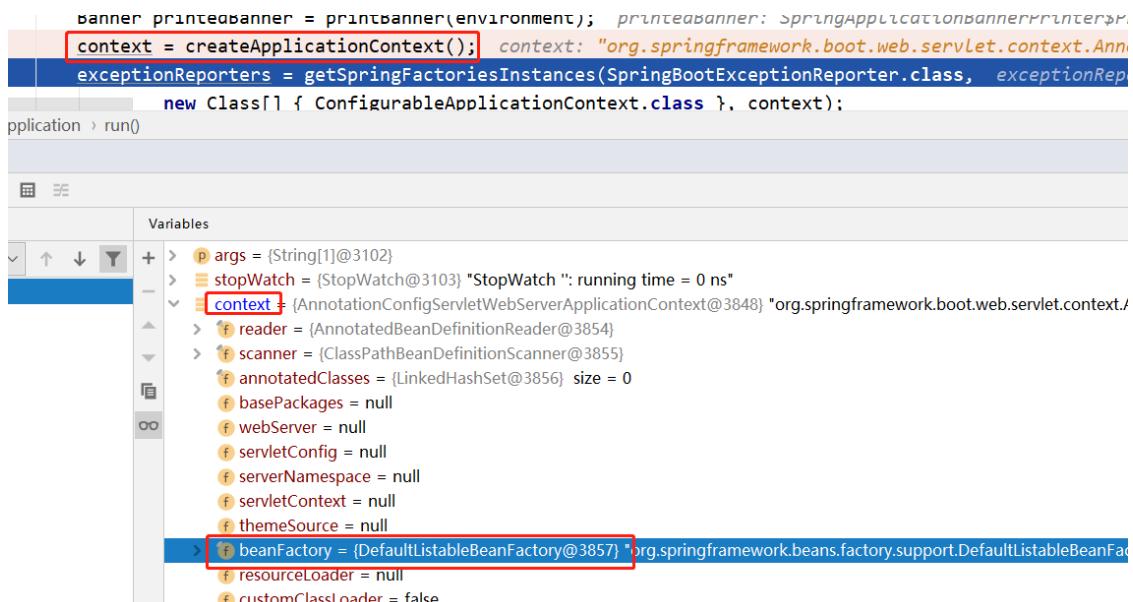
public class GenericApplicationContext extends AbstractApplicationContext
implements BeanDefinitionRegistry {
    private final DefaultListableBeanFactory beanFactory;
    ...
    public GenericApplicationContext() {
        this.beanFactory = new DefaultListableBeanFactory();
    }
    ...
}

```

beanFactory正是在AnnotationConfigServletWebServerApplicationContext实现的接口GenericApplicationContext中定义的。在上面createApplicationContext()方法中的，BeanUtils.instantiateClass(contextClass)这个方法中，不但初始化了AnnotationConfigServletWebServerApplicationContext类，也就是我们的上下文context，同样也触发了GenericApplicationContext类的构造函数，从而IoC容器也创建了。

仔细看他的构造函数，有没有发现一个很熟悉的类DefaultListableBeanFactory，没错，DefaultListableBeanFactory就是IoC容器真实面目了。在后面的refresh()方法分析中，DefaultListableBeanFactory是无处不在的存在感。

debug跳过createApplicationContext()方法。



如上图所示，context就是我们熟悉的上下文（也有人称之为容器，都可以，看个人爱好和理解），beanFactory就是我们所说的IoC容器的真实面孔了。细细感受下上下文和容器的联系和区别，对于我们理解源码有很大的帮助。在我们学习过程中，我们也是将上下文和容器严格区分开来的。

第四步：刷新应用上下文前的准备阶段

prepareContext()方法

前面我们介绍了SpringBoot启动流程run()方法的前三步，接下来再来介绍：

第四步：刷新应用上下文前的准备阶段。也就是prepareContext()方法。

首先看prepareContext()方法。

```

private void prepareContext(ConfigurableApplicationContext context,

```

```

        ConfigurableEnvironment environment,
        SpringApplicationRunListeners listeners,
        ApplicationArguments applicationArguments,
        Banner printedBanner) {
    //设置容器环境
    context.setEnvironment(environment);
    //执行容器后置处理
    postProcessApplicationContext(context);
    //执行容器中的 ApplicationContextInitializer 包括spring.factories和通过三种
    //方式自定义的
    applyInitializers(context);
    //向各个监听器发送容器已经准备好的事件
    listeners.contextPrepared(context);
    if (this.logStartupInfo) {
        logStartupInfo(context.getParent() == null);
        logStartupProfileInfo(context);
    }

    // Add boot specific singleton beans
    //将main函数中的args参数封装成单例Bean，注册进容器
    context.getBeanFactory().registerSingleton("springApplicationArguments",
        applicationArguments);
    //将 printedBanner 也封装成单例，注册进容器
    if (printedBanner != null) {
        context.getBeanFactory().registerSingleton("springBootBanner",
        printedBanner);
    }

    // Load the sources
    Set<Object> sources = getAllSources();
    Assert.notEmpty(sources, "Sources must not be empty");
    //加载我们的启动类，将启动类注入容器
    load(context, sources.toArray(new Object[0]));
    //发布容器已加载事件
    listeners.contextLoaded(context);
}

```

首先看这行 Set sources = getAllSources(); 在getAllSources()中拿到了我们的启动类。

我们重点讲解这行 load(context, sources.toArray(new Object[0]));，其他的方法请参阅注释。

跟进load()方法，看源码

```

protected void load(ApplicationContext context, Object[] sources) {
    if (logger.isDebugEnabled()) {
        logger.debug(
            "Loading source " +
        StringUtils.arrayToCommaDelimitedString(sources));
    }
    //创建 BeanDefinitionLoader
    BeanDefinitionLoader loader = createBeanDefinitionLoader(
        getBeanDefinitionRegistry(context), sources);
    if (this.beanNameGenerator != null) {
        loader.setBeanNameGenerator(this.beanNameGenerator);
    }
    if (this.resourceLoader != null) {

```

```

        loader.setResourceLoader(this.resourceLoader);
    }
    if (this.environment != null) {
        loader.setEnvironment(this.environment);
    }
    loader.load();
}

```

getBeanDefinitionRegistry()

继续看getBeanDefinitionRegistry()方法的源码

```

private BeanDefinitionRegistry getBeanDefinitionRegistry(ApplicationContext
context) {
    if (context instanceof BeanDefinitionRegistry) {
        return (BeanDefinitionRegistry) context;
    }
    ...
}

```

这里将我们前文创建的上下文强转为BeanDefinitionRegistry，他们之间是有继承关系的。BeanDefinitionRegistry定义了很重要的方法registerBeanDefinition()，该方法将BeanDefinition注册进DefaultListableBeanFactory容器的beanDefinitionMap中。

createBeanDefinitionLoader()

继续看createBeanDefinitionLoader()方法，最终进入了BeanDefinitionLoader类的构造方法，如下

```

BeanDefinitionLoader(BeanDefinitionRegistry registry, Object... sources) {
    Assert.notNull(registry, "Registry must not be null");
    Assert.notEmpty(sources, "Sources must not be empty");
    this.sources = sources;
    //注解形式的Bean定义读取器 比如: @Configuration @Bean @Component @Controller
    // @Service等等
    this.annotatedReader = new AnnotatedBeanDefinitionReader(registry);
    //XML形式的Bean定义读取器
    this.xmlReader = new XmlBeanDefinitionReader(registry);
    if (isGroovyPresent()) {
        this.groovyReader = new GroovyBeanDefinitionReader(registry);
    }
    //类路径扫描器
    this.scanner = new ClassPathBeanDefinitionScanner(registry);
    //扫描器添加排除过滤器
    this.scanner.addExcludeFilter(new ClassExcludeFilter(sources));
}

```

先记住上面的三个属性，上面三个属性在 BeanDefinition的Resource定位，和BeanDefinition的注册中起到了很重要的作用。

loader.load();

跟进load()方法

```
private int load(Object source) {
    Assert.notNull(source, "Source must not be null");
    // 从Class加载
    if (source instanceof Class<?>) {
        return load((Class<?>) source);
    }
    // 从Resource加载
    if (source instanceof Resource) {
        return load((Resource) source);
    }
    // 从Package加载
    if (source instanceof Package) {
        return load((Package) source);
    }
    // 从CharSequence 加载 ? ?
    if (source instanceof CharSequence) {
        return load((CharSequence) source);
    }
    throw new IllegalArgumentException("Invalid source type " +
        source.getClass());
}
```

当前我们的主类会按Class加载。

继续跟进load()方法。

```
private int load(Class<?> source) {
    if (isGroovyPresent()
        && GroovyBeanDefinitionSource.class.isAssignableFrom(source)) {
        // Any GroovyLoaders added in beans{} DSL can contribute beans here
        GroovyBeanDefinitionSource loader =
        BeanUtils.instantiateClass(source,
            GroovyBeanDefinitionSource.class);
        load(loader);
    }
    if (isComponent(source)) {
        // 将 启动类的 BeanDefinition 注册进 beanDefinitionMap
        this.annotatedReader.register(source);
        return 1;
    }
    return 0;
}
```

isComponent(source)判断主类是不是存在@Component注解，主类@SpringBootApplication是一个组合注解，包含@Component。

this.annotatedReader.register(source);跟进register()方法，最终进到AnnotatedBeanDefinitionReader类的doRegisterBean()方法。

```
<T> void doRegisterBean(Class<T> annotatedClass, @Nullable Supplier<T>
instanceSupplier, @Nullable String name,
@Nullable Class<? extends Annotation>[] qualifiers,
BeanDefinitionCustomizer... definitionCustomizers) {
```

```

//将指定的类 封装为AnnotatedGenericBeanDefinition
AnnotatedGenericBeanDefinition abd = new
AnnotatedGenericBeanDefinition(annotatedClass);
if (this.conditionEvaluator.shouldSkip(abd.getMetadata())) {
    return;
}

abd.setInstanceSupplier(instanceSupplier);
// 获取该类的 scope 属性
ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata(abd);
abd.setScope(scopeMetadata.getScopeName());
String beanName = (name != null ? name :
this.beanNameGenerator.generateBeanName(abd, this.registry));

AnnotationConfigUtils.processCommonDefinitionAnnotations(abd);
if (qualifiers != null) {
    for (Class<? extends Annotation> qualifier : qualifiers) {
        if (Primary.class == qualifier) {
            abd.setPrimary(true);
        }
        else if (Lazy.class == qualifier) {
            abd.setLazyInit(true);
        }
        else {
            abd.addQualifier(new AutowireCandidateQualifier(qualifier));
        }
    }
}
for (BeanDefinitionCustomizer customizer : definitionCustomizers) {
    customizer.customize(abd);
}

BeanDefinitionHolder definitionHolder = new BeanDefinitionHolder(abd,
beanName);
definitionHolder =
AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder,
this.registry);
// 将该BeanDefinition注册到IoC容器的beanDefinitionMap中
BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder,
this.registry);
}

```

在该方法中将主类封装成AnnotatedGenericBeanDefinition

BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder, this.registry);方法将 BeanDefinition注册进beanDefinitionMap

```

public static void registerBeanDefinition(
    BeanDefinitionHolder definitionHolder, BeanDefinitionRegistry
registry)
    throws BeanDefinitionStoreException {
// Register bean definition under primary name.
// primary name 其实就是id吧
String beanName = definitionHolder.getBeanName();
registry.registerBeanDefinition(beanName,
definitionHolder.getBeanDefinition());

```

```

    // Register aliases for bean name, if any.
    // 然后就是注册别名
    String[] aliases = definitionHolder.getAliases();
    if (aliases != null) {
        for (String alias : aliases) {
            registry.registerAlias(beanName, alias);
        }
    }
}

```

继续跟进registerBeanDefinition()方法。

```

@Override
public void registerBeanDefinition(String beanName, BeanDefinition
beanDefinition)
    throws BeanDefinitionStoreException {

    Assert.hasText(beanName, "Bean name must not be empty");
    Assert.notNull(beanDefinition, "BeanDefinition must not be null");

    if (beanDefinition instanceof AbstractBeanDefinition) {
        try {
            // 最后一次校验了
            // 对bean的Overrides进行校验，还不知道会在哪处理这些overrides
            ((AbstractBeanDefinition) beanDefinition).validate();
        } catch (BeanDefinitionValidationException ex) {
            throw new
            BeanDefinitionStoreException(beanDefinition.getResourceDescription(),
            beanName,
                    "Validation of bean definition failed", ex);
        }
    }
    // 判断是否存在重复名字的bean，之后看允不允许override
    // 以前使用synchronized实现互斥访问，现在采用ConcurrentHashMap
    BeanDefinition existingDefinition =
this.beanDefinitionMap.get(beanName);
    if (existingDefinition != null) {
        //如果该类不允许 overriding 直接抛出异常
        if (!isAllowBeanDefinitionOverriding()) {
            throw new
            BeanDefinitionStoreException(beanDefinition.getResourceDescription(),
            beanName,
                    "Cannot register bean definition [" + beanDefinition +
"] for bean '" + beanName +
"' : There is already [" + existingDefinition +
"] bound.");
        } else if (existingDefinition.getRole() < beanDefinition.getRole())
{
            // e.g. was ROLE_APPLICATION, now overriding with ROLE_SUPPORT
            or ROLE_INFRASTRUCTURE
            if (logger.isWarnEnabled()) {
                logger.warn("Overriding user-defined bean definition for
bean '" + beanName +
"' with a framework-generated bean definition:
replacing [" +
existingDefinition + "] with [" + beanDefinition +
"]");
            }
        }
    }
}

```

```

        }
    } else if (!beanDefinition.equals(existingDefinition)) {
        if (logger.isInfoEnabled()) {
            logger.info("Overriding bean definition for bean '" +
beanName +
                "' with a different definition: replacing [" +
existingDefinition +
                    "] with [" + beanDefinition + "]");
        }
    } else {
        if (logger.isDebugEnabled()) {
            logger.debug("Overriding bean definition for bean '" +
beanName +
                "' with an equivalent definition: replacing [" +
existingDefinition +
                    "] with [" + beanDefinition + "]");
        }
    }
    //注册进beanDefinitionMap
    this.beanDefinitionMap.put(beanName, beanDefinition);
} else {
    if (hasBeanCreationStarted()) {
        // Cannot modify startup-time collection elements anymore (for
stable iteration)
        synchronized (this.beanDefinitionMap) {
            this.beanDefinitionMap.put(beanName, beanDefinition);
            List<String> updatedDefinitions = new ArrayList<>(
this.beanDefinitionNames.size() + 1);
            updatedDefinitions.addAll(this.beanDefinitionNames);
            updatedDefinitions.add(beanName);
            this.beanDefinitionNames = updatedDefinitions;
            if (this.manualSingletonNames.contains(beanName)) {
                Set<String> updatedSingletons = new LinkedHashSet<>(
this.manualSingletonNames);
                updatedSingletons.remove(beanName);
                this.manualSingletonNames = updatedSingletons;
            }
        }
    } else {
        // Still in startup registration phase
        //如果仍处于启动注册阶段，注册进beanDefinitionMap
        this.beanDefinitionMap.put(beanName, beanDefinition);
        this.beanDefinitionNames.add(beanName);
        this.manualSingletonNames.remove(beanName);
    }
    this.frozenBeanDefinitionNames = null;
}

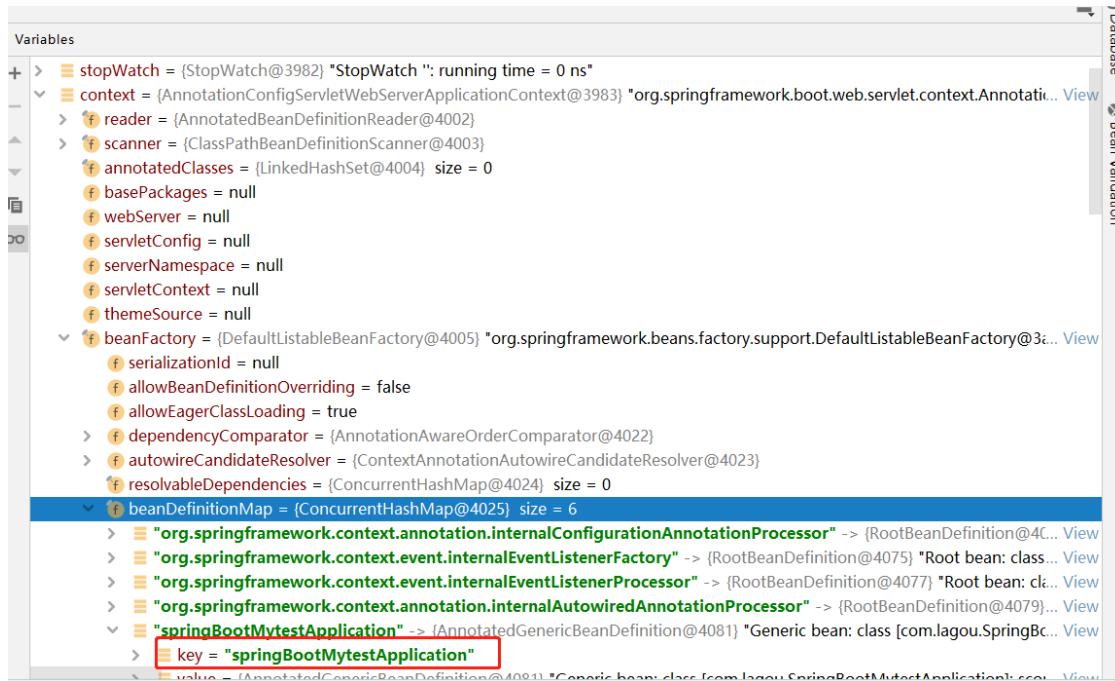
if (existingDefinition != null || containssingleton(beanName)) {
    resetBeanDefinition(beanName);
}
}

```

最终来到DefaultListableBeanFactory类的registerBeanDefinition()方法，
DefaultListableBeanFactory类还熟悉吗？相信大家一定非常熟悉这个类了。
DefaultListableBeanFactory是IoC容器的具体产品。

仔细看这个方法registerBeanDefinition(), 首先会检查是否已经存在, 如果存在并且不允许被覆盖则直接抛出异常。不存在的话就直接注册进beanDefinitionMap中。

debug跳过prepareContext()方法, 可以看到, 启动类的BeanDefinition已经注册进来了。



OK, 到这里启动流程的第五步就算讲完了, 其实在这没必要讲这么细, 因为启动类 BeanDefinition的注册流程和后面我们自定义的BeanDefinition的注册流程是一样的。这先介绍一遍这个流程, 后面熟悉了这个流程就好理解了。后面马上就到最最最重要的refresh()方法了。

第五步：刷新应用上下文（IOC容器的初始化过程）

首先我们要知道到IoC容器的初始化过程, 主要分下面三步:

- 1 BeanDefinition的Resource定位
- 2 BeanDefinition的载入
- 3 向IoC容器注册BeanDefinition

在上一小节介绍了prepareContext()方法, 在准备刷新阶段做了什么工作。

接下来我们主要从refresh()方法中总结IoC容器的初始化过程。

从run方法的, refreshContext()方法一路跟下去, 最终来到AbstractApplicationContext类的refresh()方法。

```
@Override  
public void refresh() throws BeansException, IllegalStateException {  
    synchronized (this.startupShutdownMonitor) {  
        // Prepare this context for refreshing.  
        //刷新上下文环境  
        prepareRefresh();  
        // Tell the subclass to refresh the internal bean factory.  
        //这里是在子类中启动 refreshBeanFactory() 的地方  
        ConfigurableListableBeanFactory beanFactory =  
obtainFreshBeanFactory();  
        // Prepare the bean factory for use in this context.  
        //准备bean工厂, 以便在此上下文中使用  
        prepareBeanFactory(beanFactory);  
    }  
}
```

```

try {
    // Allows post-processing of the bean factory in context
    subclasses.
        //设置 beanFactory 的后置处理
        postProcessBeanFactory(beanFactory);
        // Invoke factory processors registered as beans in the context.
        //调用 BeanFactory 的后处理器，这些处理器是在Bean 定义中向容器注册的
        invokeBeanFactoryPostProcessors(beanFactory);
        // Register bean processors that intercept bean creation.
        //注册Bean的后处理器，在Bean创建过程中调用
        registerBeanPostProcessors(beanFactory);
        // Initialize message source for this context.
        //对上下文中的消息源进行初始化
        initMessageSource();
        // Initialize event multicaster for this context.
        //初始化上下文中的事件机制
        initApplicationEventMulticaster();
        // Initialize other special beans in specific context
    subclasses.
        //初始化其他特殊的Bean
        onRefresh();
        // Check for listener beans and register them.
        //检查监听Bean并且将这些监听Bean向容器注册
        registerListeners();
        // Instantiate all remaining (non-lazy-init) singletons.
        //实例化所有的 (non-lazy-init) 单件
        finishBeanFactoryInitialization(beanFactory);
        // Last step: publish corresponding event.
        //发布容器事件，结束Refresh过程
        finishRefresh();
    } catch (BeansException ex) {
        if (logger.isWarnEnabled()) {
            logger.warn("Exception encountered during context
initialization - " +
                    "cancelling refresh attempt: " + ex);
        }
        // Destroy already created singletons to avoid dangling
        resources.
        destroyBeans();
        // Reset 'active' flag.
        cancelRefresh(ex);
        // Propagate exception to caller.
        throw ex;
    } finally {
        // Reset common introspection caches in Spring's core, since we
        // might not ever need metadata for singleton beans anymore...
        resetCommonCaches();
    }
}
}

```

从以上代码中我们可以看到，refresh()方法中所作的工作也挺多，我们没办法面面俱到，主要根据IoC容器的初始化步骤进行分析，所以我们主要介绍重要的方法，其他的请看注释。

obtainFreshBeanFactory();

在启动流程的第三步：初始化应用上下文。中我们创建了应用的上下文，并触发了 GenericApplicationContext类的构造方法如下所示，创建了beanFactory，也就是创建了 DefaultListableBeanFactory类。

```
public GenericApplicationContext() {  
    this.beanFactory = new DefaultListableBeanFactory();  
}
```

关于obtainFreshBeanFactory()方法，其实就是拿到我们之前创建的beanFactory。

```
protected ConfigurableListableBeanFactory obtainFreshBeanFactory() {  
    //刷新beanFactory  
    refreshBeanFactory();  
    //获取beanFactory  
    ConfigurableListableBeanFactory beanFactory = getBeanFactory();  
    if (logger.isDebugEnabled()) {  
        logger.debug("Bean factory for " + getDisplayName() + ": " +  
beanFactory);  
    }  
    return beanFactory;  
}
```

从上面代码可知，在该方法中主要做了三个工作，刷新beanFactory，获取beanFactory，返回beanFactory。

首先看一下refreshBeanFactory()方法，跟下去来到GenericApplicationContext类的refreshBeanFactory()发现也没做什么。

```
@Override  
protected final void refreshBeanFactory() throws IllegalStateException {  
    if (!this.refreshed.compareAndSet(false, true)) {  
        throw new IllegalStateException(  
            "GenericApplicationContext does not support multiple refresh  
attempts: just call 'refresh' once");  
    }  
    this.beanFactory.setSerializationId(getId());  
}
```

TIPS:

1, AbstractApplicationContext类有两个子类实现了refreshBeanFactory()，但是在前面第三步初始化上下文的时候，

实例化了GenericApplicationContext类，所以没有进入

AbstractRefreshableApplicationContext中的refreshBeanFactory()方法。

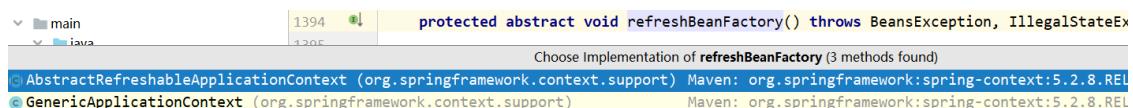
2, this.refreshed.compareAndSet(false, true)

这行代码在这里表示：GenericApplicationContext只允许刷新一次

这行代码，很重要，不是在Spring中很重要，而是这行代码本身。首先看一下this.refreshed属性：

```
private final AtomicBoolean refreshed = new AtomicBoolean();
```

java J.U.C并发包中很重要的一个原子类AtomicBoolean。通过该类的compareAndSet()方法可以实现一段代码绝对只实现一次的功能。



```
prepareBeanFactory(beanFactory);
```

从字面意思上可以看出准备BeanFactory。

看代码，具体看看做了哪些准备工作。这个方法不是重点，看注释吧。

```
protected void prepareBeanFactory(ConfigurableListableBeanFactory beanFactory) {
    // Tell the internal bean factory to use the context's class loader etc.
    // 配置类加载器：默认使用当前上下文的类加载器
    beanFactory.setBeanClassLoader(getClassLoader());
    // 配置EL表达式：在Bean初始化完成，填充属性的时候会用到
    beanFactory.setBeanExpressionResolver(new StandardBeanExpressionResolver(beanFactory.getBeanClassLoader()));
    // 添加属性编辑器 PropertyEditor
    beanFactory.addPropertyEditorRegistrar(new ResourceEditorRegistrar(this, getEnvironment()));

    // Configure the bean factory with context callbacks.
    // 添加Bean的后置处理器
    beanFactory.addBeanPostProcessor(new ApplicationContextAwareProcessor(this));
    // 忽略装配以下指定的类
    beanFactory.ignoreDependencyInterface(EnvironmentAware.class);
    beanFactory.ignoreDependencyInterface(EmbeddedValueResolverAware.class);
    beanFactory.ignoreDependencyInterface(ResourceLoaderAware.class);

    beanFactory.ignoreDependencyInterface(ApplicationEventPublisherAware.class);
    ;
    beanFactory.ignoreDependencyInterface(MessageSourceAware.class);
    beanFactory.ignoreDependencyInterface(ApplicationContextAware.class);

    // BeanFactory interface not registered as resolvable type in a plain factory.
    // MessageSource registered (and found for autowiring) as a bean.
    // 将以下类注册到 beanFactory (DefaultListableBeanFactory) 的
    // resolvableDependencies 属性中
    beanFactory.registerResolvableDependency(BeanFactory.class, beanFactory);
    beanFactory.registerResolvableDependency(ResourceLoader.class, this);

    beanFactory.registerResolvableDependency(ApplicationEventPublisher.class, this);
    beanFactory.registerResolvableDependency(ApplicationContext.class, this);

    // Register early post-processor for detecting inner beans as ApplicationListeners.
    // 将早期后处理器注册为application监听器，用于检测内部bean
    beanFactory.addBeanPostProcessor(new ApplicationListenerDetector(this));

    // Detect a LoadTimeWeaver and prepare for weaving, if found.
    // 如果当前BeanFactory包含LoadTimeWeaver Bean，说明存在类加载期织入AspectJ，
    // 则把当前BeanFactory交给类加载期BeanPostProcessor实现类
    LoadTimeWeaverAwareProcessor来处理，
    // 从而实现类加载期织入AspectJ的目的。
```

```

    if (beanFactory.containsBean(LOAD_TIME_WEAVER_BEAN_NAME)) {
        beanFactory.addBeanPostProcessor(new
LoadTimeWeaverAwareProcessor(beanFactory));
        // Set a temporary ClassLoader for type matching.
        beanFactory.setTempClassLoader(new
ContextTypeMatchClassLoader(beanFactory.getBeanClassLoader()));
    }

    // Register default environment beans.
    // 将当前环境变量 (environment) 注册为单例bean
    if (!beanFactory.containsLocalBean(ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(ENVIRONMENT_BEAN_NAME,
getEnvironment());
    }
    // 将当前系统配置 (systemProperties) 注册为单例Bean
    if (!beanFactory.containsLocalBean(SYSTEM_PROPERTIES_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_PROPERTIES_BEAN_NAME,
getEnvironment().getSystemProperties());
    }
    // 将当前系统环境 (systemEnvironment) 注册为单例Bean
    if (!beanFactory.containsLocalBean(SYSTEM_ENVIRONMENT_BEAN_NAME)) {
        beanFactory.registerSingleton(SYSTEM_ENVIRONMENT_BEAN_NAME,
getEnvironment().getSystemEnvironment());
    }
}

```

四、postProcessBeanFactory(beanFactory);

postProcessBeanFactory()方法向上下文中添加了一系列的Bean的后置处理器。

后置处理器工作的时机是在所有的beanDefinition加载完成之后，bean实例化之前执行。简单来说Bean的后置处理器可以修改BeanDefinition的属性信息。

五、invokeBeanFactoryPostProcessors(beanFactory); (重点)

IoC容器的初始化过程包括三个步骤，在invokeBeanFactoryPostProcessors()方法中完成了IoC容器初始化过程的三个步骤。

1. 第一步：Resource定位

在SpringBoot中，我们都知道他的包扫描是从主类所在的包开始扫描的，prepareContext()方法中，会先将主类解析成BeanDefinition，然后在refresh()方法的invokeBeanFactoryPostProcessors()方法中解析主类的BeanDefinition获取basePackage的路径。这样就完成了定位的过程。其次SpringBoot的各种starter是通过SPI扩展机制实现的自动装配，SpringBoot的自动装配同样也是在invokeBeanFactoryPostProcessors()方法中实现的。还有一种情况，在SpringBoot中有很多的@EnableXXX注解，细心点进去看的应该就知道其底层是@Import注解，在invokeBeanFactoryPostProcessors()方法中也实现了对该注解指定的配置类的定位加载。

常规的在SpringBoot中有三种实现定位，第一个是主类所在包的，第二个是SPI扩展机制实现的自动装配（比如各种starter），第三种就是@Import注解指定的类。（对于非常规的不说了）

2. 第二步：BeanDefinition的载入

在第一步中说了三种Resource的定位情况，定位后紧接着就是BeanDefinition的分别载入。所谓的载入就是通过上面的定位得到的basePackage，SpringBoot会将该路径拼接成：classpath:com/lagou/**/.class这样的形式，然后一个叫做xPathMatchingResourcePatternResolver的类会将该路径下所有的.class文件都加载进来，然后遍历判断是不是有@Component注解，如果有的话，就是我们要装载的BeanDefinition。大致过程就是这样的了。

TIPS:

@Configuration, @Controller, @Service等注解底层都是@Component注解，只不过包装了一层罢了。

3、第三个过程：注册BeanDefinition

这个过程通过调用上文提到的BeanDefinitionRegister接口的实现来完成。这个注册过程把载入过程中解析得到的BeanDefinition向IoC容器进行注册。通过上文的分析，我们可以看到，在IoC容器中将BeanDefinition注入到一个ConcurrentHashMap中，IoC容器就是通过这个HashMap来持有这些BeanDefinition数据的。比如DefaultListableBeanFactory中的beanDefinitionMap属性。

OK，总结完了，接下来我们通过代码看看具体是怎么实现的。

```
protected void
invokeBeanFactoryPostProcessors(ConfigurableListableBeanFactory beanFactory)
{
    PostProcessorRegistrationDelegate.invokeBeanFactoryPostProcessors(beanFactory,
        getBeanFactoryPostProcessors());
    ...
}

// PostProcessorRegistrationDelegate类
public static void invokeBeanFactoryPostProcessors(
    ConfigurableListableBeanFactory beanFactory,
    List<BeanFactoryPostProcessor> beanFactoryPostProcessors) {
    ...
    invokeBeanDefinitionRegistryPostProcessors(currentRegistryProcessors,
        registry);
    ...
}

// PostProcessorRegistrationDelegate类
private static void invokeBeanDefinitionRegistryPostProcessors(
    Collection<? extends BeanDefinitionRegistryPostProcessor>
    postProcessors, BeanDefinitionRegistry registry) {

    for (BeanDefinitionRegistryPostProcessor postProcessor : postProcessors)
    {
        postProcessor.postProcessBeanDefinitionRegistry(registry);
    }
}

// ConfigurationClassPostProcessor类
@Override
public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
    registry) {
    ...
    processConfigBeanDefinitions(registry);
}

// ConfigurationClassPostProcessor类
public void processConfigBeanDefinitions(BeanDefinitionRegistry registry) {
```

```

    ...
    do {
        parser.parse(candidates);
        parser.validate();
        ...
    }
    ...
}

```

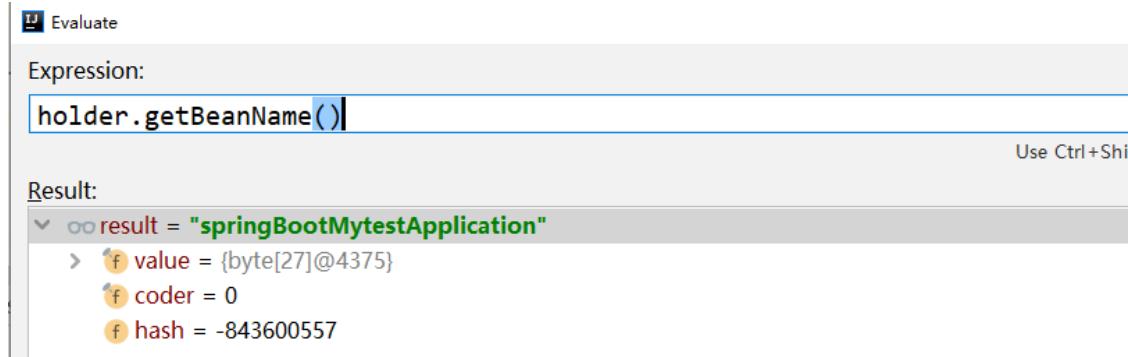
一路跟踪调用栈，来到ConfigurationClassParser类的parse()方法。

```

// ConfigurationClassParser类
public void parse(Set<BeanDefinitionHolder> configCandidates) {
    this.deferredImportSelectors = new LinkedList<>();
    for (BeanDefinitionHolder holder : configCandidates) {
        BeanDefinition bd = holder.getBeanDefinition();
        try {
            // 如果是SpringBoot项目进来的，bd其实就是前面主类封装成的
            AnnotatedGenericBeanDefinition (AnnotatedBeanDefinition接口的实现类)
            if (bd instanceof AnnotatedBeanDefinition) {
                parse(((AnnotatedBeanDefinition) bd).getMetadata(),
holder.getBeanName());
            } else if (bd instanceof AbstractBeanDefinition &&
((AbstractBeanDefinition) bd).hasBeanClass()) {
                parse(((AbstractBeanDefinition) bd).getBeanClass(),
holder.getBeanName());
            } else {
                parse(bd.getBeanClassName(), holder.getBeanName());
            }
        } catch (BeanDefinitionStoreException ex) {
            throw ex;
        } catch (Throwable ex) {
            throw new BeanDefinitionStoreException(
                "Failed to parse configuration class [" +
bd.getBeanClassName() + "]", ex);
        }
    }
    // 加载默认的配置---》（对springboot项目来说这里就是自动装配的入口了）
    processDeferredImportSelectors();
}

```

看上面的注释，在前面的prepareContext()方法中，我们详细介绍了我们的主类是如何一步步的封装成AnnotatedGenericBeanDefinition，并注册进IoC容器的beanDefinitionMap中的。



继续沿着parse(((AnnotatedBeanDefinition) bd).getMetadata(), holder.getBeanName());方法跟下去

看doProcessConfigurationClass()方法。 (SpringBoot的包扫描的入口方法，重点)

```
// ConfigurationClassParser类
protected final void parse(AnnotationMetadata metadata, String beanName)
throws IOException {
    processConfigurationClass(new ConfigurationClass(metadata, beanName));
}

// ConfigurationClassParser类
protected void processConfigurationClass(ConfigurationClass configClass)
throws IOException {

    ...
    // Recursively process the configuration class and its superclass
    // hierarchy.
    // 递归地处理配置类及其父类层次结构。
    SourceClass sourceClass = asSourceClass(configClass);
    do {
        // 递归处理Bean，如果有父类，递归处理，直到顶层父类
        sourceClass = doProcessConfigurationClass(configClass, sourceClass);
    }
    while (sourceClass != null);

    this.configurationClasses.put(configClass, configClass);
}

// ConfigurationClassParser类
protected final SourceClass doProcessConfigurationClass(ConfigurationClass configClass,
    SourceClass sourceClass)
throws IOException {

    // Recursively process any member (nested) classes first
    // 首先递归处理内部类，（SpringBoot项目的主类一般没有内部类）
    processMemberClasses(configClass, sourceClass);

    // Process any @PropertySource annotations
    // 针对 @PropertySource 注解的属性配置处理
    for (AnnotationAttributes propertySource :
        AnnotationConfigUtils.attributesForRepeatable(
            sourceClass.getMetadata(), PropertySources.class,
            org.springframework.context.annotation.PropertySource.class)) {
        if (this.environment instanceof ConfigurableEnvironment) {
            processPropertySource(propertySource);
        } else {
            logger.warn("Ignoring @PropertySource annotation on [" +
                sourceClass.getMetadata().getClassName() +
                "]. Reason: Environment must implement
ConfigurableEnvironment");
        }
    }

    // Process any @ComponentScan annotations
    // 根据 @ComponentScan 注解，扫描项目中的Bean (SpringBoot 启动类上有该注解)
    Set<AnnotationAttributes> componentScans =
        AnnotationConfigUtils.attributesForRepeatable(
            sourceClass.getMetadata(), ComponentScans.class,
            ComponentScan.class);
}
```

```

    if (!componentScans.isEmpty() &&
        !this.conditionEvaluator.shouldSkip(sourceClass.getMetadata(),
ConfigurationPhase.REGISTER_BEAN)) {
        for (AnnotationAttributes componentScan : componentScans) {
            // The config class is annotated with @ComponentScan -> perform
            // the scan immediately
            // 立即执行扫描, (SpringBoot项目为什么是从主类所在的包扫描, 这就是关键
            // 了)
            Set<BeanDefinitionHolder> scannedBeanDefinitions =
                this.componentScanParser.parse(componentScan,
sourceClass.getMetadata().getClassName());
            // Check the set of scanned definitions for any further config
            // classes and parse recursively if needed
            for (BeanDefinitionHolder holder : scannedBeanDefinitions) {
                BeanDefinition bdCand =
holder.getBeanDefinition().getOriginatingBeanDefinition();
                if (bdCand == null) {
                    bdCand = holder.getBeanDefinition();
                }
                // 检查是否是@ConfigurationClass (是否有configuration/component
                // 两个注解), 如果是, 递归查找该类相关联的配置类。
                // 所谓相关的配置类, 比如@Configuration中的@Bean定义的bean。或者在
                // 有@Component注解的类上继续存在@Import注解。
                if
(ConfigurationClassUtils.checkConfigurationClassCandidate(bdCand,
this.metadataReaderFactory)) {
                    parse(bdCand.getBeanClassName(), holder.getBeanName());
                }
            }
        }
    }

    // Process any @Import annotations
    // 递归处理 @Import 注解 (SpringBoot项目中经常用的各种@Enable*** 注解基本都是封装
    // 的@Import)
    processImports(configClass, sourceClass, getImports(sourceClass), true);

    // Process any @ImportResource annotations
    AnnotationAttributes importResource =
        AnnotationConfigUtils.attributesFor(sourceClass.getMetadata(),
ImportResource.class);
    if (importResource != null) {
        String[] resources = importResource.getStringArray("locations");
        Class<? extends BeanDefinitionReader> readerClass =
importResource.getClass("reader");
        for (String resource : resources) {
            String resolvedResource =
this.environment.resolveRequiredPlaceholders(resource);
            configClass.addImportedResource(resolvedResource, readerClass);
        }
    }

    // Process individual @Bean methods
    Set<MethodMetadata> beanMethods =
retrieveBeanMethodMetadata(sourceClass);
    for (MethodMetadata methodMetadata : beanMethods) {
        configClass.addBeanMethod(new BeanMethod(methodMetadata,
configClass));
    }
}

```

```

    }

    // Process default methods on interfaces
    processInterfaces(configClass, sourceClass);

    // Process superclass, if any
    if (sourceClass.getMetadata().hasSuperClass()) {
        String superclass = sourceClass.getMetadata().getSuperClassName();
        if (superclass != null && !superclass.startsWith("java") &&
            !this.knownSuperclasses.containsKey(superclass)) {
            this.knownSuperclasses.put(superclass, configClass);
            // Superclass found, return its annotation metadata and recurse
            return sourceClass.getSuperClass();
        }
    }

    // No superclass -> processing is complete
    return null;
}

```

我们大致说一下这个方法里面都干了什么

TIPS:

在以上代码的`parse(bdCand.getBeanClassName(), holder.getBeanName())`;会进行递归调用，
因为当Spring扫描到需要加载的类会进一步判断每一个类是否满足是@Component/@Configuration注解的类，
如果满足会递归调用`parse()`方法，查找其相关的类。
同样的`processImports(configClass, sourceClass, getImports(sourceClass), true);`
通过@Import注解查找到的类同样也会递归查找其相关的类。
两个递归在debug的时候会很乱，用文字叙述起来更让人难以理解，所以，我们只关注对主类的解析，及其类的扫描过程。

上面代码中 for (AnnotationAttributes propertySource :

`AnnotationConfigUtils.attributesForRepeatable(... 获得主类上的@PropertySource注解)`，解析该注解并将该注解指定的properties配置文件中的值存储到Spring的 Environment中，
Environment接口提供方法去读取配置文件中的值，参数是properties文件中定义的key值。

`Set componentScans = AnnotationConfigUtils.attributesForRepeatable(
sourceClass.getMetadata(), ComponentScans.class, ComponentScan.class);` 解析主类上的
@ComponentScan注解，后面的代码将会解析该注解并进行包扫描。

`processImports(configClass, sourceClass, getImports(sourceClass), true);` 解析主类上的
@Import注解，并加载该注解指定的配置类。

TIPS:

在spring中好多注解都是一层一层封装的，比如@EnableXXX，是对@Import注解的二次封装。
@SpringBootApplication注解
=@ComponentScan+@EnableAutoConfiguration+@Import+@Configuration+@Component。
@Controller, @Service等等是对@Component的二次封装。。。

继续向下看：

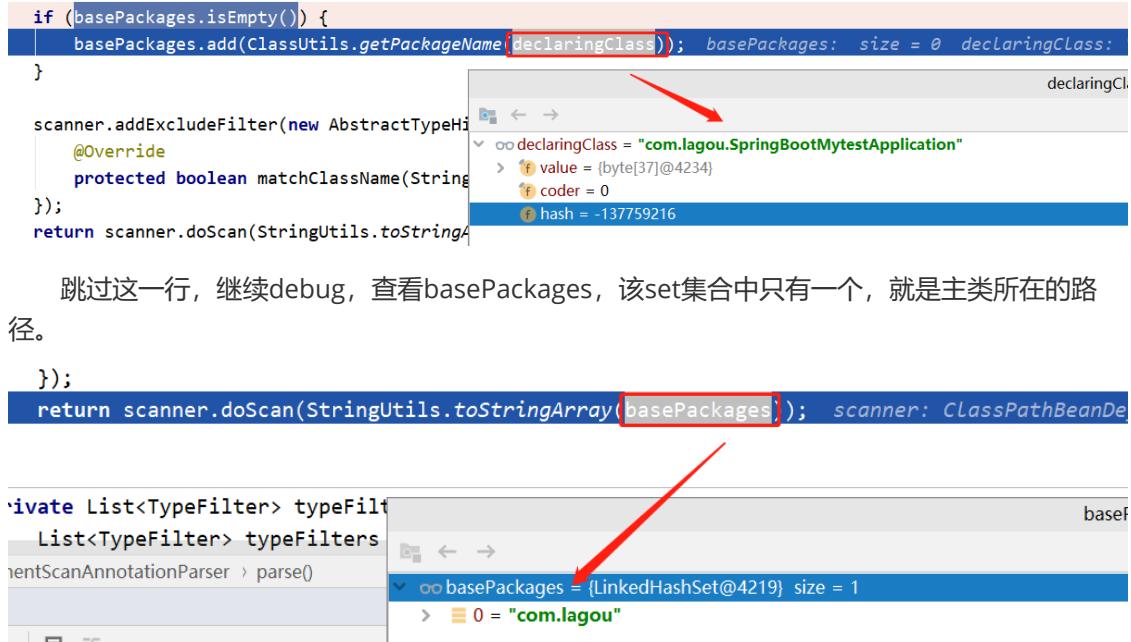
```
Set scannedBeanDefinitions = this.componentScanParser.parse(componentScan,  
sourceClass.getMetadata().getClassName());
```

进入该方法

```
// ComponentScanAnnotationParser类  
public Set<BeanDefinitionHolder> parse(AnnotationAttributes componentScan,  
final String declaringClass) {  
    ClassPathBeanDefinitionScanner scanner = new  
    ClassPathBeanDefinitionScanner(this.registry,  
        componentScan.getBoolean("useDefaultFilters"), this.environment,  
        this.resourceLoader);  
    ...  
    // 根据 declaringClass (如果是SpringBoot项目，则参数为主类的全路径名)  
    if (basePackages.isEmpty()) {  
        basePackages.add(classutils.getPackageName(declaringClass));  
    }  
    ...  
    // 根据basePackages扫描类  
    return scanner.doScan(StringUtils.toStringArray(basePackages));  
}
```

发现有两行重要的代码

为了验证代码中的注释，debug，看一下declaringClass，如下图所示确实是我们的主类的全路径名。



TIPS：
为什么只有一个还要用一个集合呢，因为我们也可以用@ComponentScan注解指定扫描路径。

到这里呢IoC容器初始化三个步骤的第一步，Resource定位就完成了，成功定位到了主类所在的包。

接着往下看 return scanner.doScan(StringUtils.toStringArray(basePackages)); Spring是如何进行类扫描的。进入doScan()方法。

```

1 // ComponentScanAnnotationParser类
2 protected Set<BeanDefinitionHolder> doscan(String... basePackages) {
3     Assert.notEmpty(basePackages, "At least one base package must be
specified");
4     Set<BeanDefinitionHolder> beanDefinitions = new LinkedHashSet<>();
5     for (String basePackage : basePackages) {
6         // 从指定的包中扫描需要装载的Bean
7         Set<BeanDefinition> candidates =
findCandidateComponents(basePackage);
8         for (BeanDefinition candidate : candidates) {
9             ScopeMetadata scopeMetadata =
this.scopeMetadataResolver.resolveScopeMetadata(candidate);
10            candidate.setScope(scopeMetadata.getScopeName());
11            String beanName =
this.beanNameGenerator.generateBeanName(candidate, this.registry);
12            if (candidate instanceof AbstractBeanDefinition) {
13                postProcessBeanDefinition((AbstractBeanDefinition)
candidate, beanName);
14            }
15            if (candidate instanceof AnnotatedBeanDefinition) {
16
AnnotationConfigUtils.processCommonDefinitionAnnotations((AnnotatedBeanDefinition)
candidate);
17        }
18        if (checkCandidate(beanName, candidate)) {
19            BeanDefinitionHolder definitionHolder = new
BeanDefinitionHolder(candidate, beanName);
20            definitionHolder =
21
AnnotationConfigUtils.applyScopedProxyMode(scopeMetadata, definitionHolder,
this.registry);
22            beanDefinitions.add(definitionHolder);
23            //将该 Bean 注册进 IOC容器 (beanDefinitionMap)
24            registerBeanDefinition(definitionHolder, this.registry);
25        }
26    }
27 }
28 return beanDefinitions;
29 }

```

这个方法中有两个比较重要的方法，第7行 Set candidates = findCandidateComponents(basePackage); 从basePackage中扫描类并解析成BeanDefinition，拿到所有符合条件的类后在第24行 registerBeanDefinition(definitionHolder, this.registry); 将该类注册进IoC容器。也就是说在这个方法中完成了IoC容器初始化过程的第二三步，BeanDefinition的载入，和BeanDefinition的注册。

findCandidateComponents(basePackage);

跟踪调用栈

复制代码

```

1 // ClassPathScanningCandidateComponentProvider类
2 public Set<BeanDefinition> findCandidateComponents(String basePackage) {
3     ...
4     else {

```

```
5         return scanCandidateComponents(basePackage);
6     }
7 }
8 // ClassPathScanningCandidateComponentProvider类
9 private Set<BeanDefinition> scanCandidateComponents(String basePackage) {
10    Set<BeanDefinition> candidates = new LinkedHashSet<>();
11    try {
12        //拼接扫描路径, 比如: classpath*:com/lagou/**/*.class
13        String packageSearchPath =
ResourcePatternResolver.CLASSPATH_ALL_URL_PREFIX +
14                resolveBasePackage(basePackage) + '/' +
this.resourcePattern;
15        //从 packageSearchPath 路径中扫描所有的类
16        Resource[] resources =
getResourcePatternResolver().getResources(packageSearchPath);
17        boolean traceEnabled = logger.isTraceEnabled();
18        boolean debugEnabled = logger.isDebugEnabled();
19        for (Resource resource : resources) {
20            if (traceEnabled) {
21                logger.trace("Scanning " + resource);
22            }
23            if (resource.isReadable()) {
24                try {
25                    MetadataReader metadataReader =
getMetadataReaderFactory().getMetadataReader(resource);
26                    // //判断该类是不是 @Component 注解标注的类, 并且不是需要排除掉的类
27                    if (isCandidateComponent(metadataReader)) {
28                        //将该类封装成
ScannedGenericBeanDefinition (BeanDefinition接口的实现类) 类
29                        ScannedGenericBeanDefinition sbd = new
ScannedGenericBeanDefinition(metadataReader);
30                        sbd.setResource(resource);
31                        sbd.setSource(resource);
32                        if (isCandidateComponent(sbd)) {
33                            if (debugEnabled) {
34                                logger.debug("Identified candidate
component class: " + resource);
35                            }
36                            candidates.add(sbd);
37                        } else {
38                            if (debugEnabled) {
39                                logger.debug("Ignored because not a
concrete top-level class: " + resource);
40                            }
41                        }
42                    } else {
43                        if (traceEnabled) {
44                            logger.trace("Ignored because not matching
any filter: " + resource);
45                        }
46                    }
47                } catch (Throwable ex) {
48                    throw new BeanDefinitionStoreException(
49                            "Failed to read candidate component class: "
+ resource, ex);
50                }
51            } else {

```

```

52             if (traceEnabled) {
53                 logger.trace("Ignored because not readable: " +
resource);
54             }
55         }
56     } catch (IOException ex) {
57         throw new BeanDefinitionStoreException("I/O failure during
classpath scanning", ex);
58     }
59 }
60 return candidates;
61 }

```

在第13行将basePackage拼接成classpath:org/springframework/boot/demo/**/.class，在第16行的getResources(packageSearchPath);方法中扫描到了该路径下的所有的类。然后遍历这些Resources，在第27行判断该类是不是 @Component 注解标注的类，并且不是需要排除掉的类。在第29行将扫描到的类，解析成ScannedGenericBeanDefinition，该类是BeanDefinition接口的实现类。OK，IoC容器的BeanDefinition载入到这里就结束了。

回到前面的doScan()方法，debug看一下结果（截图中所示的就是定位的需要交给Spring容器管理的类）。

registerBeanDefinition(definitionHolder, this.registry);

查看registerBeanDefinition()方法。是不是有点眼熟，在前面介绍prepareContext()方法时，我们详细介绍了主类的BeanDefinition是怎么一步一步的注册进DefaultListableBeanFactory的beanDefinitionMap中的。完成了BeanDefinition的注册，就完成了IoC容器的初始化过程。此时，在使用的IoC容器DefaultListableFactory中已经建立了整个Bean的配置信息，而这些BeanDefinition已经可以被容器使用了。他们都在BeanbefinitionMap里被检索和使用。容器的作用就是对这些信息进行处理和维护。这些信息是容器简历依赖反转的基础。

```

protected void registerBeanDefinition(BeanDefinitionHolder
definitionHolder, BeanDefinitionRegistry registry) {
    BeanDefinitionReaderUtils.registerBeanDefinition(definitionHolder,
registry);
}

```

OK，到这里IoC容器的初始化过程的三个步骤就梳理完了。当然这只是针对SpringBoot的包扫描的定位方式的BeanDefinition的定位，加载，和注册过程。前面我们说过，还有两种方式@Import和SPI扩展实现的starter的自动装配。

@Import注解的解析过程

现在大家也应该知道了，各种@EnableXXX注解，很大一部分都是对@Import的二次封装（其实也是为了解耦，比如当@Import导入的类发生变化时，我们的业务系统也不需要改任何代码）。

我们又要回到上文中的ConfigurationClassParser类的doProcessConfigurationClass方法的第68行processImports(configClass, sourceClass, getImports(sourceClass), true)；跳跃性比较大。上面解释过，我们只针对主类进行分析，因为这里有递归。

processImports(configClass, sourceClass, getImports(sourceClass), true);中configClass和sourceClass参数都是主类相对应的。

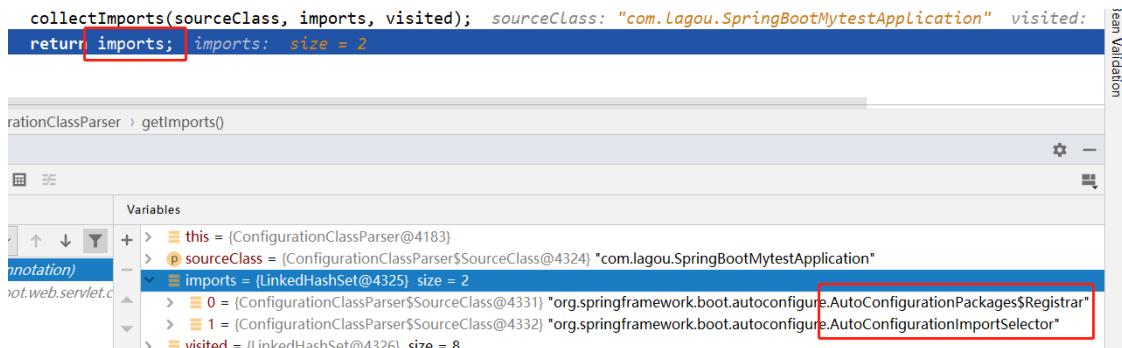
首先看getImports(sourceClass);

```

private Set<SourceClass> getImports(SourceClass sourceClass) throws
IOException {
    Set<SourceClass> imports = new LinkedHashSet<>();
    Set<SourceClass> visited = new LinkedHashSet<>();
    collectImports(sourceClass, imports, visited);
    return imports;
}

```

debug



两个呢是主类上的@SpringBootApplication中的@Import注解指定的类

接下来，是不是要进行执行了

记下来再回到ConfigurationClassParser类的parse(Set configCandidates):

```

public void parse(Set<BeanDefinitionHolder> configCandidates) {
    for (BeanDefinitionHolder holder : configCandidates) {
        BeanDefinition bd = holder.getBeanDefinition();
        try {
            if (bd instanceof AnnotatedBeanDefinition) {
                parse(((AnnotatedBeanDefinition) bd).getMetadata(),
holder.getBeanName());
            }
            else if (bd instanceof AbstractBeanDefinition &&
((AbstractBeanDefinition) bd).hasBeanClass()) {
                parse(((AbstractBeanDefinition) bd).getBeanClass(),
holder.getBeanName());
            }
            else {
                parse(bd.getBeanClassName(), holder.getBeanName());
            }
        }
        catch (BeanDefinitionStoreException ex) {
            throw ex;
        }
        catch (Throwable ex) {
            throw new BeanDefinitionStoreException(
                "Failed to parse configuration class [" +
bd.getBeanClassName() + "]", ex);
        }
    }

    // 去执行组件类
    this.deferredImportSelectorHandler.process();
}

```

点进process方法：

```
public void process() {
    List<DeferredImportSelectorHolder> deferredImports =
this.deferredImportSelectors;
    this.deferredImportSelectors = null;
    try {
        if (deferredImports != null) {
            DeferredImportSelectorGroupingHandler handler = new
DeferredImportSelectorGroupingHandler();
            deferredImports.sort(DEFERRED_IMPORT_COMPARATOR);
            deferredImports.forEach(handler::register);
            // 继续点击进去
            handler.processGroupImports();
        }
    }
    finally {
        this.deferredImportSelectors = new ArrayList<>();
    }
}
```

继续点击handler.processGroupImports();

```
public void processGroupImports() {
    for (DeferredImportSelectorGrouping grouping :
this.groupings.values()) {
        Predicate<String> exclusionFilter =
grouping.getCandidateFilter();
        // 查看调用的getImports
        grouping.getImports().forEach(entry -> {
            ConfigurationClass configurationClass =
this.configurationClasses.get(entry.getMetadata());
            try {
                processImports(configurationClass,
assourceClass(configurationClass, exclusionFilter),
collections.singleton(assourceClass(entry.getImportClassName(),
exclusionFilter)),
exclusionFilter, false);
            }
            catch (BeanDefinitionStoreException ex) {
                throw ex;
            }
            catch (Throwable ex) {
                throw new BeanDefinitionStoreException(
                    "Failed to process import candidates for
configuration class [" +
configurationClass.getMetadata().getClassName() + "]", ex);
            }
        });
    }
}
```

```

// 是不是很熟悉了
public Iterable<Group.Entry> getImports() {
    for (DeferredImportSelectorHolder deferredImport :
this.deferredImports) {
        // 调用了process方法

this.group.process(deferredImport.getConfigurationClass().getMetadata(),
                    deferredImport.getImportSelector());
    }
    return this.group.selectImports();
}

```

和之前介绍的process完美衔接

```

public void process(AnnotationMetadata annotationMetadata,
DeferredImportSelector deferredImportSelector) {
    Assert.state(deferredImportSelector instanceof
AutoConfigurationImportSelector,
() -> String.format("Only %s implementations are
supported, got %s",
AutoConfigurationImportSelector.class.getSimpleName(),
deferredImportSelector.getClass().getName()));

        // 【1】，调用getAutoConfigurationEntry方法得到自动配置类放入
autoConfigurationEntry对象中
        AutoConfigurationEntry autoConfigurationEntry =
((AutoConfigurationImportSelector) deferredImportSelector)

        .getAutoConfigurationEntry(annotationMetadata,
annotationMetadata);

        // 【2】，又将封装了自动配置类的autoConfigurationEntry对象装进
autoConfigurationEntries集合
        this.autoConfigurationEntries.add(autoConfigurationEntry);
        // 【3】，遍历刚获取的自动配置类
        for (String importClassName :
autoConfigurationEntry.getConfigurations()) {
            // 这里符合条件的自动配置类作为key, annotationMetadata作为值放进
entries集合
            this.entries.putIfAbsent(importClassName,
annotationMetadata);
        }
    }
}

```

第六步：刷新应用上下文后的扩展接口

```

protected void afterRefresh(ConfigurableApplicationContext context,
                           ApplicationArguments args) {
}

```

扩展接口，设计模式中的模板方法，默认为空实现。如果有自定义需求，可以重写该方法。比如打印一些启动结束log，或者一些其它后置处理。

2.5 源码剖析-自定义Start

SpringBoot starter机制

SpringBoot中的starter是一种非常重要的机制，能够抛弃以前繁杂的配置，将其统一集成进starter，应用者只需要在maven中引入starter依赖，SpringBoot就能自动扫描到要加载的信息并启动相应的默认配置。starter让我们摆脱了各种依赖库的处理，需要配置各种信息的困扰。SpringBoot会自动通过classpath路径下的类发现需要的Bean，并注册进IOC容器。SpringBoot提供了针对日常企业应用研发各种场景的spring-boot-starter依赖模块。所有这些依赖模块都遵循着约定成俗的默认配置，并允许我们调整这些配置，即遵循“约定大于配置”的理念。

比如我们在springboot里面要引入redis,那么我们需要在pom中引入以下内容

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

这其实就是一个starter。

简而言之，starter就是一个外部的项目，我们需要使用它的时候就可以在当前springboot项目中引入它。

为什么要自定义starter

在我们的日常开发工作中，经常会有一些独立于业务之外的配置模块，我们经常将其放到一个特定的包下，然后如果另一个工程需要复用这块功能的时候，需要将代码硬拷贝到另一个工程，重新集成一遍，麻烦至极。如果我们将这些可独立于业务代码之外的功能配置模块封装成一个个starter，复用的时候只需要将其在pom中引用依赖即可，再由SpringBoot为我们完成自动装配，就非常轻松了

自定义starter的案例

以下案例是开发中遇到的部分场景

- ▲ 动态数据源。
- ▲ 登录模块。
- ▲ 基于AOP技术实现日志切面。

.....

自定义starter的命名规则

SpringBoot提供的starter以 `spring-boot-starter-xxx` 的方式命名的。

官方建议自定义的starter使用 `xxx-spring-boot-starter` 命名规则。以区分SpringBoot生态提供的starter

自定义starter代码实现

整个过程分为两部分：

- 自定义starter
- 使用starter

(1) 自定义starter

首先，先完成自定义starter

(1) 新建maven jar工程，工程名为zdy-spring-boot-starter，导入依赖：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure</artifactId>
        <version>2.2.9.RELEASE</version>
    </dependency>
</dependencies>
```

(2) 编写javaBean

```
@EnableConfigurationProperties(SimpleBean.class)
@ConfigurationProperties(prefix = "simplebean")
public class SimpleBean {

    private int id;
    private String name;

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

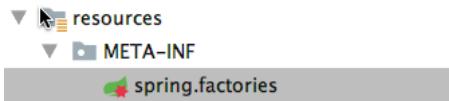
    @Override
    public String toString() {
        return "SimpleBean{" +
            "id=" + id +
            ", name='" + name + '\'' +
            '}';
    }
}
```

(3) 编写配置类MyAutoConfiguration

```
@Configuration  
public class MyAutoConfiguration {  
  
    static {  
        System.out.println("MyAutoConfiguration init....");  
    }  
  
    @Bean  
    public SimpleBean simpleBean(){  
        return new SimpleBean();  
    }  
}
```

(4) resources下创建/META-INF/spring.factories

注意：META-INF是自己手动创建的目录，spring.factories也是手动创建的文件，在该文件中配置自己的自动配置类



```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\  
com.lagou.config.MyAutoConfiguration
```

上面这句话的意思就是SpringBoot启动的时候会去加载我们的simpleBean到IOC容器中。这其实是一种变形的SPI机制

(2) 使用自定义starter

(1) 导入自定义starter的依赖

```
<dependency>  
    <groupId>com.lagou</groupId>  
    <artifactId>zdy-spring-boot-starter</artifactId>  
    <version>1.0-SNAPSHOT</version>  
</dependency>
```

(2) 在全局配置文件中配置属性值

```
simplebean.id=1  
simplebean.name=自定义starter
```

(3) 编写测试方法

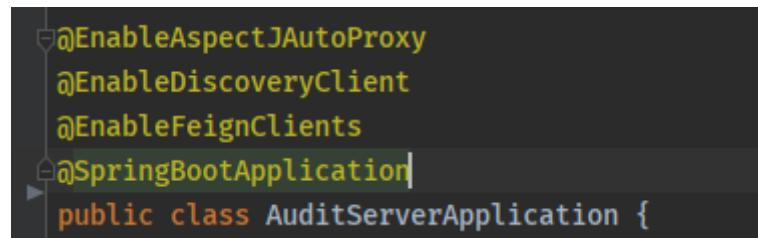
```
//测试自定义starter
@Autowired
private SimpleBean simpleBean;

@Test
public void zdyStarterTest(){
    System.out.println(simpleBean);
}
```

但此处还有一个问题，如果有一天我们不想要启动工程的时候自动装配SimpleBean呢？可能有的同学会想，那简单啊，我们去pom中把依赖注释掉，的确，这是一种方案，但为免有点Low。

热插拔技术

还记得我们经常会在启动类Application上面加@EnableXXX注解吗？



其实这个@Enablexxx注解就是一种热拔插技术，加了这个注解就可以启动对应的starter，当不需要对应的starter的时候只需要把这个注解注释掉就行，是不是很优雅呢？那么这是如何实现的呢？

改造zdy工程新增热插拔支持类

新增标记类ConfigMarker

```
public class ConfigMarker {  
}
```

新增EnableRegisterServer注解

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Import({ConfigMarker.class})
public @interface EnableRegisterServer {  
}
```

改造 MyAutoConfiguration 新增条件注解 @ConditionalOnBean(ConfigMarker.class) ,
@ConditionalOnBean 这个是条件注解，前面的意思代表只有当期上下文中包含 ConfigMarker 对象，被标注的类才会被实例化。

```
@Configuration
@ConditionalOnBean(ConfigMarker.class)
public class MyAutoConfiguration {  
  
    static {
```

```

        System.out.println("MyAutoConfiguration init....");
    }

    @Bean
    public SimpleBean simpleBean(){
        return new SimpleBean();
    }

}

```

改造service工程

在启动类上新增@EnableImRegisterServer注解

```

@SpringBootApplication
@EnableImRegisterServer
public class SpringbootUseApplication {

    public static void main(String[] args) { SpringApplication.run(SpringbootUseApplication.class, args); }
}

```

到此热插拔就实现好了，当你加了@EnableImRegisterserver 的时候启动zy工程就会自动装配SimpleBean，反之则不装配。

让的原理也很简单，当加了@EnableImRegisterserver 注解的时候，由于这个注解使用了@Import({ConfigMarker.class})，所以会导致Spring去加载 ConfigMarker 到上下文中，而又因为条件注解 @ConditionalOnBean(ConfigMarker.class) 的存在，所以 MyAutoConfiguration 类就会被实例化。

关于条件注解的讲解

- @ConditionalOnBean: 仅仅在当前上下文中存在某个对象时，才会实例化一个Bean。
- @ConditionalOnClass: 某个class位于类路径上，才会实例化一个Bean。
- @ConditionalOnExpression: 当表达式为true的时候，才会实例化一个Bean。基于SpEL表达式的条件判断。
- @ConditionalOnMissingBean: 仅仅在当前上下文中不存在某个对象时，才会实例化一个Bean。
- @ConditionalOnMissingClass: 某个class类路径上不存在的时候，才会实例化一个Bean。
- @ConditionalOnNotWebApplication: 不是web应用，才会实例化一个Bean。
- @ConditionalOnWebApplication: 当项目是一个Web项目时进行实例化。
- @ConditionalOnNotWebApplication: 当项目不是一个Web项目时进行实例化。
- @ConditionalOnProperty: 当指定的属性有指定的值时进行实例化。
- @ConditionalOnJava: 当JVM版本为指定的版本范围时触发实例化。
- @ConditionalOnResource: 当类路径下有指定的资源时触发实例化。
- @ConditionalOnJndi: 在JNDI存在的条件下触发实例化。
- @ConditionalOnSingleCandidate: 当指定的Bean在容器中只有一个，或者有多个但是指定了首选的Bean时触发实例化。

2.6 源码剖析-内嵌Tomcat

Spring Boot默认支持Tomcat，Jetty，和Undertow作为底层容器。

而Spring Boot默认使用Tomcat，一旦引入spring-boot-starter-web模块，就默认使用Tomcat容器。

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

Servlet容器的使用

默认servlet容器

我们看看spring-boot-starter-web这个starter中有什么



核心就是引入了tomcat和SpringMvc

切换servlet容器

那如果我么想切换其他Servlet容器呢，只需如下两步：

- 将tomcat依赖移除掉
- 引入其他Servlet容器依赖

引入jetty：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <!--移除spring-boot-starter-web中的tomcat-->
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
    <groupId>org.springframework.boot</groupId>
```

```

        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <!--引入jetty-->
    <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>

```

内嵌Tomcat自动配置原理

在启动springboot的时候可谓是相当简单，只需要执行以下代码

```

public static void main(String[] args) {
    SpringApplication.run(SpringBootMytestApplication.class, args);
}

```

那些看似简单的事物，其实并不简单。我们之所以觉得他简单，是因为复杂性都被隐藏了。通过上诉代码，大概率可以提出以下几个疑问

- SpringBoot是如何启动内置tomcat的
- SpringBoot为什么可以响应请求，他是如何配置的SpringMvc

SpringBoot启动内置tomcat流程

1、进入SpringBoot启动类，点进@SpringBootApplication源码，如下图



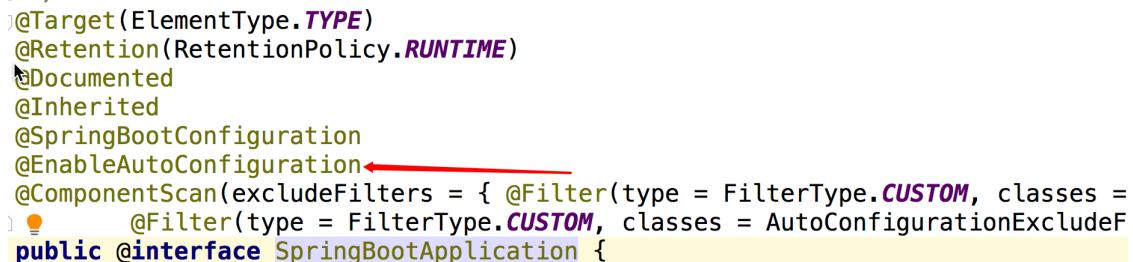
```

@SpringBootApplication //能够扫描Spring组件并自动配置Spring Boot
public class SpringbootDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootDemoApplication.class, args);
    }

}

```



```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = { @Filter(type = FilterType.CUSTOM, classes = { @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeF
public @interface SpringBootApplication {

```

2、继续点进@EnableAutoConfiguration,进入该注解，如下图

```

-----  

@Inherited  

@AutoConfigurationPackage  

@Import(AutoConfigurationImportSelector.class)  

public @interface EnableAutoConfiguration {

```

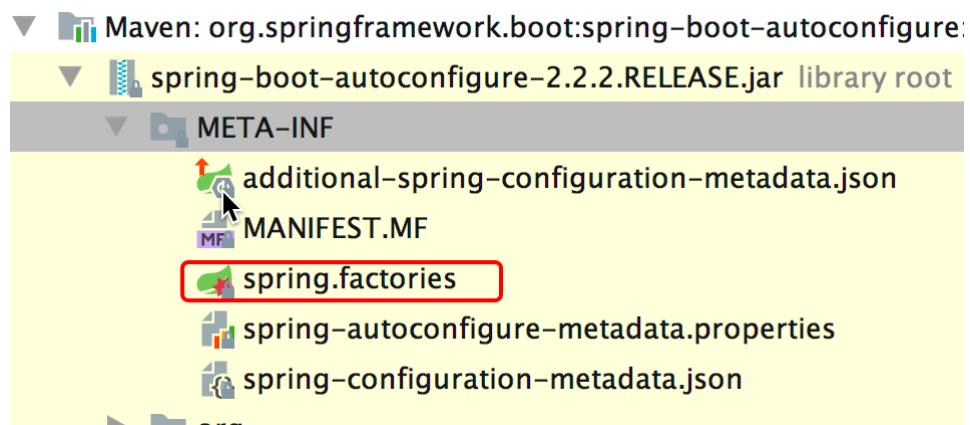
3、上图中使用@Import注解对AutoConfigurationImportSelector类进行了引入，该类做了什么事情呢？进入源码，首先调用selectImport()方法，在该方法中调用了getAutoConfigurationEntry()方法，在之中又调用了getCandidateConfigurations()方法，getCandidateConfigurations()方法就去META-INF/spring.factory配置文件中加载相关配置类

```

protected List<String> getCandidateConfigurations(AnnotationMetadata metadata, AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(metadata);
    getBeanClassLoader());
    Assert.notEmpty(configurations, message: "No auto configuration classes found in META-INF/spring.factory file. " + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}

```

这个spring.factories配置文件是加载的spring-boot-autoconfigure的配置文件



继续打开spring.factories配置文件，找到tomcat所在的类，tomcat加载在ServletWebServerFactoryAutoConfiguration配置类中

```

network.boot.autoconfigure.web.servlet.DispatcherServletAutoConfiguration,
network.boot.autoconfigure.web.servlet.ServletWebServerFactoryAutoConfiguration, //
network.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfiguration,
network.boot.autoconfigure.web.servlet.HttpEncodingAutoConfiguration

```

进入该类，里面也通过@Import注解将EmbeddedTomcat、EmbeddedJetty、EmbeddedUndertow等嵌入式容器类加载进来了，springboot默认是启动嵌入式tomcat容器，如果要改变启动jetty或者undertow容器，需在pom文件中去设置。如下图：

```

@Configuration(proxyBeanMethods = false)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@ConditionalOnClass(ServletRequest.class)
@ConditionalOnWebApplication(type = Type.SERVLET)
@EnableConfigurationProperties(ServerProperties.class)
@Import({ ServletWebServerFactoryAutoConfiguration.BeanPostProcessorsRegistrar.class,
        // ServletWebServerFactoryConfiguration.EmbeddedTomcat.class,
        // ServletWebServerFactoryConfiguration.EmbeddedJetty.class,
        // ServletWebServerFactoryConfiguration.EmbeddedUndertow.class })
public class ServletWebServerFactoryAutoConfiguration {

```

继续进入EmbeddedTomcat类中，见下图：

```

public static class EmbeddedTomcat {
    @Bean
    public TomcatServletWebServerFactory tomcatServletWebServerFactory(
        ObjectProvider<TomcatConnectorCustomizer> connectorCustomizers,
        ObjectProvider<TomcatContextCustomizer> contextCustomizers,
        ObjectProvider<TomcatProtocolHandlerCustomizer<?>> protocolHandlerCustomizers) {
        TomcatServletWebServerFactory factory = new TomcatServletWebServerFactory();
        factory.getTomcatConnectorCustomizers()
            .addAll(connectorCustomizers.orderedStream().collect(Collectors.toList()));
        factory.getTomcatContextCustomizers()
            .addAll(contextCustomizers.orderedStream().collect(Collectors.toList()));
        factory.getTomcatProtocolHandlerCustomizers()
            .addAll(protocolHandlerCustomizers.orderedStream().collect(Collectors.toList()));
        return factory;
    }
}

```

进入TomcatServletWebServerFactory类，里面的getWebServer()是关键方法，如图：

```

@Override
public WebServer getWebServer(ServletContextInitializer... initializers) {
    if (this.disableMBeanRegistry) {
        Registry.disableRegistry();
    }
    Tomcat tomcat = new Tomcat(); 实例化一个tomcat
    File baseDir = (this.baseDirectory != null) ? this.baseDirectory : createTempDir();
    tomcat.setBaseDir(baseDir.getAbsolutePath());
    Connector connector = new Connector(this.protocol); 设置tomcat相关dir,protocol等信息
    connector.setThrowOnFailure(true);
    tomcat.getService().addConnector(connector);
    customizeConnector(connector);
    tomcat.setConnector(connector);
    tomcat.getHost().setAutoDeploy(false);
    configureEngine(tomcat.getEngine());
    for (Connector additionalConnector : this.additionalTomcatConnectors) {
        tomcat.getService().addConnector(additionalConnector);
    }
    prepareContext(tomcat.getHost(), initializers);
    return getTomcatWebServer(tomcat); 传统tomcat实例到下一个方法
}

```

继续进入getTomcatWebServer()等方法，一直往下跟到tomcat初始化方法，调用tomcat.start()方法，tomcat就正式开启运行，见图

```

private void initialize() throws WebServerException {
    logger.info("Tomcat initialized with port(s): " + getPortsDescription());
    synchronized (this.monitor) {
        try {
            addInstanceIdToEngineName();

            Context context = findContext();
            context.addLifecycleListener((event) -> {
                if (context.equals(event.getSource()) && Lifecycle.START_EVENT.equals(event.getName())) {
                    // Remove service connectors so that protocol binding doesn't
                    // happen when the service is started.
                    removeServiceConnectors();
                }
            });
            // Start the server to trigger initialization listeners
            this.tomcat.start();
        }
    }
}

```

走到这里tomcat在springboot中的配置以及最终启动的流程就走完了，相信大家肯定有一个疑问，上上图中的getWebServer()方法是在哪里调用的呢？上面的代码流程并没有发现getWebServer()被调用的地方。因为getWebServer()方法的调用根本就不在上面的代码流程中，它是在另外一个流程中被调用的

getWebServer()的调用分析

首先进入SpringBoot启动类的run方法：

```
springApplication.run(HppaApplication.class, args);
```

这个会最终调用到一个同名方法run(String... args)

```
public ConfigurableApplicationContext run(String... args) {
    //Stopwatch主要是用来统计每项任务执行时长，例如Spring Boot启动占用总时长。
    Stopwatch stopwatch = new Stopwatch();
    stopwatch.start();
    ConfigurableApplicationContext context = null;
    Collection<SpringBootExceptionReporter> exceptionReporters = new
    ArrayList<>();
    configureHeadlessProperty();
    //第一步：获取并启动监听器 通过加载META-INF/spring.factories 完成了
    SpringApplicationRunListener实例化工作
    SpringApplicationRunListeners listeners = getRunListeners(args);
    //实际上是调用了EventPublishingRunListener类的starting()方法
    listeners.starting();
    try {
        ApplicationArguments applicationArguments = new
        DefaultApplicationArguments(args);
        //第二步：构造容器环境，简而言之就是加载系统变量，环境变量，配置文件
        ConfigurableEnvironment environment = prepareEnvironment(listeners,
        applicationArguments);
        //设置需要忽略的bean
        configureIgnoreBeanInfo(environment);
        //打印banner
        Banner printedBanner = printBanner(environment);
        //第三步：创建容器
        context = createApplicationContext();
        //第四步：实例化SpringBootExceptionReporter.class，用来支持报告关于启动的
        错误
        exceptionReporters =
        getSpringFactoriesInstances(SpringBootExceptionReporter.class,
            new Class[] { ConfigurableApplicationContext.class },
        context);
        //第五步：准备容器 这一步主要是在容器刷新之前的准备动作。包含一个非常关键的操作：
        将启动类注入容器，为后续开启自动化配置奠定基础。
        prepareContext(context, environment, listeners,
        applicationArguments, printedBanner);
        //第六步：刷新容器 springBoot相关的处理工作已经结束，接下的工作就交给了
        spring。内部会调用spring的refresh方法，
        // refresh方法在spring整个源码体系中举足轻重，是实现 ioc 和 aop的关键。
        refreshContext(context);
        //第七步：刷新容器后的扩展接口 设计模式中的模板方法，默认为空实现。如果有自定义需
        求，可以重写该方法。比如打印一些启动结束log，或者一些其它后置处理。
        afterRefresh(context, applicationArguments);
        stopwatch.stop();
        if (this.logStartupInfo) {
            new
            StartupInfoLogger(this.mainApplicationClass).logStarted(getApplicationLog(),
            stopwatch);
        }
        //发布应用已经启动的事件
        listeners.started(context);
        /*
```

```

        * 遍历所有注册的ApplicationRunner和CommandLineRunner，并执行其run()方法。
        * 我们可以实现自己的ApplicationRunner或者CommandLineRunner，来对
SpringBoot的启动过程进行扩展。
    */
    callRunners(context, applicationArguments);
}
catch (Throwable ex) {
    handleRunFailure(context, ex, exceptionReporters, listeners);
    throw new IllegalStateException(ex);
}

try {
    //应用已经启动完成的监听事件
    listeners.running(context);
}
catch (Throwable ex) {
    handleRunFailure(context, ex, exceptionReporters, null);
    throw new IllegalStateException(ex);
}
return context;
}

```

这个方法大概做了以下几件事

1. 获取并启动监听器 通过加载META-INF/spring.factories 完成了 SpringApplicationRunListener实例化工作
2. 构造容器环境，简而言之就是加载系统变量，环境变量，配置文件
3. 创建容器
4. 实例化SpringBootExceptionReporter.class，用来支持报告关于启动的错误
5. 准备容器
6. 刷新容器
7. 刷新容器后的扩展接口

那么内置tomcat启动源码，就是隐藏在上诉第六步：refreshContext方法里面，该方法最终会调用到AbstractApplicationContext类的refresh()方法

进入refreshContext()方法，如图：

```

public void refresh() throws BeansException, IllegalStateException {
    synchronized (this.startupShutdownMonitor) {
        // Prepare this context for refreshing.
        prepareRefresh();

        // Tell the subclass to refresh the internal bean factory.
        ConfigurableListableBeanFactory beanFactory =
obtainFreshBeanFactory();

        // Prepare the bean factory for use in this context.
        prepareBeanFactory(beanFactory);

        try {
            // Allows post-processing of the bean factory in context
subclasses.
            postProcessBeanFactory(beanFactory);

            // Invoke factory processors registered as beans in the context.
        }
    }
}

```

```

        invokeBeanFactoryPostProcessors(beanFactory);

        // Register bean processors that intercept bean creation.
        registerBeanPostProcessors(beanFactory);

        // Initialize message source for this context.
        initMessageSource();

        // Initialize event multicaster for this context.
        initApplicationEventMulticaster();

        // Initialize other special beans in specific context
subclasses.
        onRefresh();

        // Check for listener beans and register them.
        registerListeners();

        // Instantiate all remaining (non-lazy-init) singletons.
        finishBeanFactoryInitialization(beanFactory);

        // Last step: publish corresponding event.
        finishRefresh();
    }

    catch (BeansException ex) {
        if (logger.isWarnEnabled()) {
            logger.warn("Exception encountered during context
initialization - " +
                    "cancelling refresh attempt: " + ex);
        }
    }

    // Destroy already created singletons to avoid dangling
resources.
    destroyBeans();

    // Reset 'active' flag.
    cancelRefresh(ex);

    // Propagate exception to caller.
    throw ex;
}

finally {
    // Reset common introspection caches in Spring's core, since we
    // might not ever need metadata for singleton beans anymore...
    resetCommonCaches();
}
}
}
}

```

一直点击refresh()方法，如图：

onRefresh()会调用到ServletWebServerApplicationContext中的createWebServer()



```
private void createWebServer() {
    WebServer webServer = this.webServer;
    ServletContext servletContext = getServletContext();
    if (webServer == null && servletContext == null) {
        ServletWebServerFactory factory = getWebServerFactory();
        this.webServer = factory.getWebServer(getSelfInitializer());
    }
    else if (servletContext != null) {
        try {
            getSelfInitializer().onStartup(servletContext);
        }
        catch (ServletException ex) {
            throw new ApplicationContextException("Cannot initialize servlet context", ex);
        }
    }
    initPropertySources();
}
```

createWebServer()就是启动web服务，但是还没有真正启动Tomcat，既然webServer是通过ServletWebServerFactory来获取的，那就来看看这个工厂的真面目。



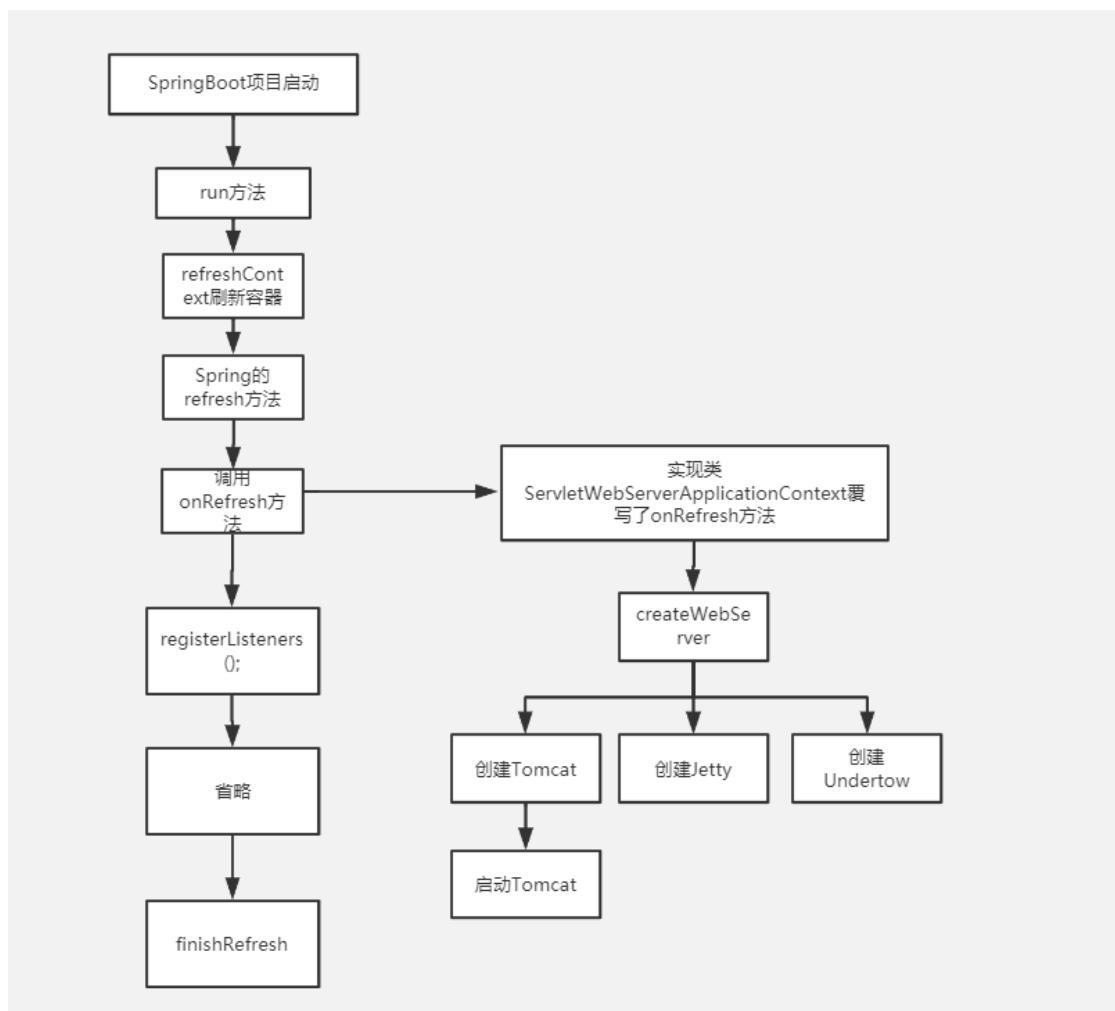
可以看到，tomcat,Jetty都实现了这个getWebServer方法，我们看TomcatServletWebServerFactory中的getWebServer(ServletContextInitializer... initializers);

```

@Override
public WebServer getWebServer(ServletContextInitializer... initializers) {
    if (this.disableMBeanRegistry) {
        Registry.disableRegistry();
    }
    Tomcat tomcat = new Tomcat();
    File baseDir = (this.baseDirectory != null) ? this.baseDirectory : createTemp...
    tomcat.setBaseDir(baseDir.getAbsolutePath());
    Connector connector = new Connector(this.protocol);
    connector.setThrowOnFailure(true);
    tomcat.getService().addConnector(connector);
    customizeConnector(connector);
    tomcat.setConnector(connector);
    tomcat.getHost().setAutoDeploy(false);
    configureEngine(tomcat.getEngine());
    for (Connector additionalConnector : this.additionalTomcatConnectors) {
        tomcat.getService().addConnector(additionalConnector);
    }
    prepareContext(tomcat.getHost(), initializers);
    return getTomcatWebServer(tomcat);
}

```

最终就调用了TomcatServletWebServerFactory类的getWebServer()方法。



小结

springboot的内部通过 new Tomcat() 的方式启动了一个内置Tomcat。但是这里还有一个问题，这里只是启动了tomcat，但是我们的springmvc是如何加载的呢？下一章我们讲接收，springboot是如何自动装配springmvc的

2.7 源码剖析-自动配置SpringMVC

在上一小节，我们介绍了SpringBoot是如何启动一个内置tomcat的。我们知道我们在SpringBoot项目里面是可以直接使用诸如 @RequestMapping 这类的SpringMVC的注解，那么同学们会不会奇怪，这是为什么？我明明没有配置SpringMVC为什么就可以使用呢？

其实仅仅引入starter是不够的，回忆一下，在一个普通的WEB项目中如何去使用SpringMVC，我们首先就是要在web.xml中配置如下配置

```
<servlet>
    <description>spring mvc servlet</description>
    <servlet-name>springMvc</servlet-name>
    <servlet-
class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>springMvc</servlet-name>
    <url-pattern>*.do</url-pattern>
</servlet-mapping>
```

但是在SpringBoot中，我们没有了web.xml文件，我们如何去配置一个 DispatcherServlet 呢？其实Servlet3.0规范中规定，要添加一个Servlet，除了采用xml配置的方式，还有一种通过代码的方式，伪代码如下

```
servletContext.addServlet(name, this.servlet);
```

那么也就是说，如果我们能动态往web容器中添加一个我们构造好的 DispatcherServlet 对象，是不是就实现自动装配SpringMVC了

自动配置（一）自动配置DispatcherServlet和DispatcherServletRegistry

springboot的自动配置基于SPI机制，实现自动配置的核心要点就是添加一个自动配置的类，SpringBoot MVC的自动配置自然也是相同原理。

所以，先找到springmvc对应的自动配置类。

```
org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoCon
figuration
```

DispatcherServletAutoConfiguration自动配置类

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration(proxyBeanMethods = false)
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(ServletWebServerFactoryAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {
    ...
}
```

1、首先注意到，@Configuration表名这是一个配置类，将会被spring给解析。

2、@ConditionalOnWebApplication意味着当时一个web项目，且是Servlet项目的时候才会被解析。

3、@ConditionalOnClass指明DispatcherServlet这个核心类必须存在才解析该类。

4、@AutoConfigureAfter指明在ServletWebServerFactoryAutoConfiguration这个类之后再解析，设定了一个顺序。

总的来说，这些注解表明了该自动配置类的会解析的前置条件需要满足。

其次，DispatcherServletAutoConfiguration类主要包含了两个内部类，分别是

1、DispatcherServletConfiguration

2、DispatcherServletRegistrationConfiguration

顾名思义，前者是配置DispatcherServlet，后者是配置DispatcherServlet的注册类。什么是注册类？我们知道Servlet实例是要被添加（注册）到如tomcat这样的ServletContext里的，这样才能够提供请求服务。所以，DispatcherServletRegistrationConfiguration将生成一个Bean，负责将DispatcherServlet给注册到ServletContext中。

配置DispatcherServletConfiguration

我们先看看DispatcherServletConfiguration这个配置类

```
@Configuration(proxyBeanMethods = false)
@Conditional(DefaultDispatcherServletCondition.class)
@ConditionalOnClass(ServletRegistration.class)
@EnableConfigurationProperties({ HttpProperties.class,
    WebMvcProperties.class })
protected static class DispatcherServletConfiguration {

    //...
}
```

@Conditional指明了一个前置条件判断，由DefaultDispatcherServletCondition实现。主要是判断了是否已经存在DispatcherServlet，如果没有才会触发解析。

@ConditionalOnClass指明了当ServletRegistration这个类存在的时候才会触发解析，生成的DispatcherServlet才能注册到ServletContext中。

最后，@EnableConfigurationProperties将会从application.properties这样的配置文件中读取spring.http和spring.mvc前缀的属性生成配置对象HttpProperties和WebMvcProperties。

再看DispatcherServletConfiguration这个内部类的内部代码

```
@Bean(name = DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
public DispatcherServlet dispatcherServlet(HttpProperties httpProperties,
    WebMvcProperties webMvcProperties) {
    DispatcherServlet dispatcherServlet = new DispatcherServlet();

    dispatcherServlet.setDispatchOptionsRequest(webMvcProperties.isDispatchOptionsRequest());
```

```

        dispatcherServlet.setDispatchTraceRequest(webMvcProperties.isDispatchTraceRequest());

        dispatcherServlet.setThrowExceptionIfNoHandlerFound(webMvcProperties.isThrowExceptionIfNoHandlerFound());

        dispatcherServlet.setPublishEvents(webMvcProperties.isPublishRequestHandledEvents());

        dispatcherServlet.setEnableLoggingRequestDetails(httpProperties.isLogRequestDetails());
    }

    @Bean
    @ConditionalOnBean(MultipartResolver.class)
    @ConditionalOnMissingBean(name =
DispatcherServlet.MULTIPART_RESOLVER_BEAN_NAME)
    public MultipartResolver multipartResolver(MultipartResolver resolver) {
        // Detect if the user has created a MultipartResolver but named it
        // incorrectly
        return resolver;
    }
}

```

这两个方法我们比较熟悉了，就是生成了Bean。

dispatcherServlet方法将生成一个DispatcherServlet的Bean对象。比较简单，就是获取一个实例，然后添加一些属性设置。

multipartResolver方法主要是把你配置的MultipartResolver的Bean给重命名一下，防止你不是用multipartResolver这个名字作为Bean的名字。

配置DispatcherServletRegistrationConfiguration

再看注册类的Bean配置

```

@Configuration(proxyBeanMethods = false)
@Conditional(DispatcherServletRegistrationCondition.class)
@ConditionalOnClass(ServletRegistration.class)
@EnableConfigurationProperties(WebMvcProperties.class)
@Import(DispatcherServletConfiguration.class)
protected static class DispatcherServletRegistrationConfiguration {
    /**
}

```

同样的，@Conditional有一个前置判断，DispatcherServletRegistrationCondition主要判断了该注册类的Bean是否存在。

@ConditionOnClass也判断了ServletRegistration是否存在

@EnableConfigurationProperties生成了WebMvcProperties的属性对象

@Import导入了DispatcherServletConfiguration，也就是我们上面的配置对象。

再看DispatcherServletRegistrationConfiguration的内部实现

```
@Bean(name = DEFAULT_DISPATCHER_SERVLET_REGISTRATION_BEAN_NAME)
@ConditionalOnBean(value = DispatcherServlet.class, name =
DEFAULT_DISPATCHER_SERVLET_BEAN_NAME)
public DispatcherServletRegistrationBean
dispatcherServletRegistration(DispatcherServlet dispatcherServlet,
                           WebMvcProperties webMvcProperties,
                           ObjectProvider<MultipartConfigElement> multipartConfig) {
    DispatcherServletRegistrationBean registration = new
DispatcherServletRegistrationBean(dispatcherServlet,
                               webMvcProperties.getServlet().getPath());
    registration.setName(DEFAULT_DISPATCHER_SERVLET_BEAN_NAME);

    registration.setLoadOnStartup(webMvcProperties.getServlet().getLoadOnStartup());
    multipartConfig.ifAvailable(registration::setMultipartConfig);
    return registration;
}
```

内部只有一个方法，生成了DispatcherServletRegistrationBean。核心逻辑就是实例化了一个Bean，设置了一些参数，如dispatcherServlet、loadOnStartup等

总结

springboot mvc的自动配置类是DispatcherServletAutoConfiguration，主要做了两件事：

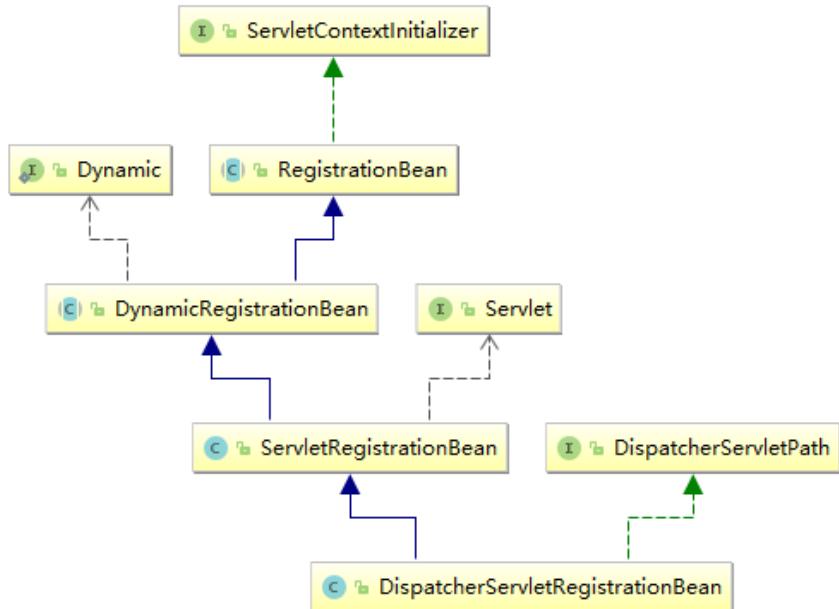
- 1) 配置DispatcherServlet
- 2) 配置DispatcherServlet的注册Bean(DispatcherServletRegistrationBean)

自动配置（二）注册DispatcherServlet到ServletContext

在上一小节的源码翻阅中，我们看到了DispatcherServlet和DispatcherServletRegistrationBean这两个Bean的自动配置。DispatcherServlet我们很熟悉，DispatcherServletRegistrationBean负责将DispatcherServlet注册到ServletContext当中

DispatcherServletRegistrationBean的类图

既然该类的职责是负责注册DispatcherServlet，那么我们得知道什么时候触发注册操作。为此，我们先看看DispatcherServletRegistrationBean这个类的类图



注册DispatcherServlet流程

ServletContextInitializer

我们看到，最上面是一个ServletContextInitializer接口。我们可以知道，实现该接口意味着是用来初始化ServletContext的。我们看看该接口

```
public interface ServletContextInitializer {
    void onStartup(ServletContext servletContext) throws ServletException;
}
```

RegistrationBean

看看RegistrationBean是怎么实现onStartup方法的

```
@Override
public final void onStartup(ServletContext servletContext) throws
ServletException {
    String description = getDescription();
    if (!isEnabled()) {
        logger.info(StringUtils.capitalize(description) + " was not
registered (disabled)");
        return;
    }

    register(description, servletContext);
}
```

调用了内部register方法，跟进它

```
protected abstract void register(String description, ServletContext
servletContext);
```

这是一个抽象方法

DynamicRegistrationBean

再看DynamicRegistrationBean是怎么实现register方法的

```
@Override  
protected final void register(String description, ServletContext servletContext) {  
    D registration = addRegistration(description, servletContext);  
    if (registration == null) {  
        logger.info(StringUtils.capitalize(description) + " was not registered (possibly already registered?)");  
        return;  
    }  
    configure(registration);  
}
```

跟进addRegistration方法

```
protected abstract D addRegistration(String description, ServletContext servletContext);
```

一样是一个抽象方法

ServletRegistrationBean

再看ServletRegistrationBean是怎么实现addRegistration方法的

```
@Override  
protected ServletRegistration.Dynamic addRegistration(String description, ServletContext servletContext) {  
    String name = getServletName();  
    return servletContext.addServlet(name, this.servlet);  
}
```

我们看到，这里直接将DispatcherServlet给add到了servletContext当中。

SpringBoot启动流程中具体体现

```
getSelfInitializer().onStartup(servletContext);
```

这段代码其实就是去加载SpringMVC，那么他是如何做到的呢？`getSelfInitializer()`最终会去调用到`ServletWebServerApplicationContext`的`selfInitialize`方法，该方法代码如下

```
private void createWebServer() {  
    WebServer webServer = this.webServer;  
    ServletContext servletContext = getServletContext();  
    if (webServer == null && servletContext == null) {  
        ServletWebServerFactory factory = getWebServerFactory();  
        this.webServer = factory.getWebServer(getSelfInitializer());  
    }  
}
```

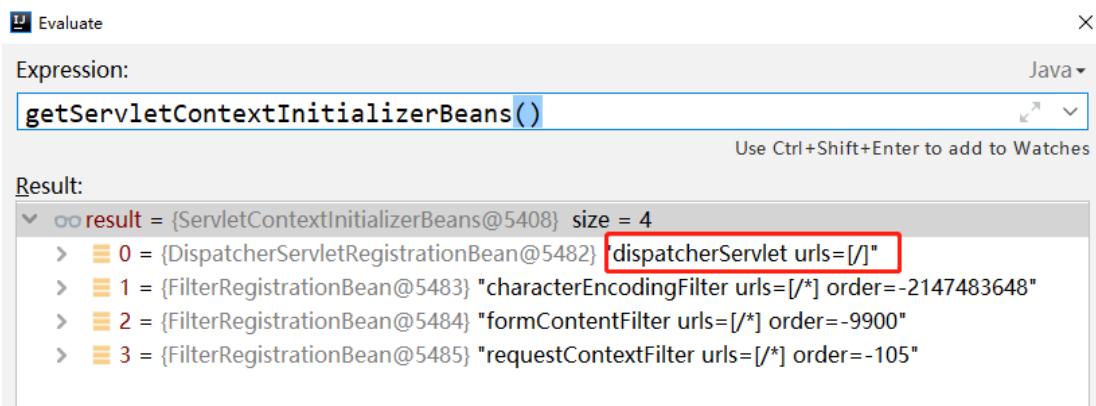
```
private void selfInitialize(ServletContext servletContext) throws  
ServletException {  
    prepareWebApplicationContext(servletContext);  
}
```

```

    ConfigurableListableBeanFactory beanFactory = getBeanFactory();
    ExistingWebApplicationScopes existingScopes = new
ExistingWebApplicationScopes(
        beanFactory);
    webApplicationContextUtils.registerWebApplicationScopes(beanFactory,
        getServletContext());
    existingScopes.restore();
    webApplicationContextUtils.registerEnvironmentBeans(beanFactory,
        getServletContext());
    for (ServletContextInitializer beans :
getServletContextInitializerBeans()) {
        beans.onStartup(servletContext);
    }
}

```

我们通过调试，知道 `getServletContextInitializerBeans()` 返回的是一个 `ServletContextInitializer` 集合，集合中有以下几个对象



然后依次去调用对象的 `onStartup` 方法，那么对于上图标红的对象来说，就是会调用到 `DispatcherServletRegistrationBean` 的 `onStartup` 方法，这个类并没有这个方法，所以最终会调用到父类 `RegistrationBean` 的 `onStartup` 方法，该方法代码如下

```

public final void onStartup(ServletContext servletContext) throws
ServletException {
    //获取当前环境到底是一个filter 还是一个servlet 还是一个listener
    String description = getDescription();
    if (!isEnabled()) {
        logger.info(StringUtils.capitalize(description) + " was not
registered (disabled)");
        return;
    }
    register(description, servletContext);
}

```

这边 `register(description, servletContext);` 会调用到 `DynamicRegistrationBean` 的 `register` 方法，代码如下

```
protected final void register(String description, ServletContext servletContext) {
    D registration = addRegistration(description, servletContext);
    if (registration == null) {
        logger.info(StringUtils.capitalize(description) + " was not registered (possibly already registered?)");
        return;
    }
    configure(registration);
}
```

addRegistration(description, servletContext) 又会调用到 `ServletRegistrationBean` 中的 `addRegistration` 方法，代码如下

```
protected ServletRegistration.Dynamic addRegistration(String description,
    ServletContext servletContext) {
    String name = getServletName();
    return servletContext.addServlet(name, this.servlet);
}
```

看到了关键的 `servletContext.addServlet` 代码了，我们通过调试，即可知到 `this.servlet` 就是 `DispatcherServlet`

```
protected ServletRegistration.Dynamic addRegistration(String description, Servlet
    String name = getServletName();  name: "dispatcherServlet"
    return servletContext.addServlet(name, this.servlet);  servletContext: Application
}
+ {DispatcherServletRegistrationBean@5481} "dispatcherServlet urls=[/]"
```

总结

SpringBoot自动装配SpringMvc其实就是在往ServletContext中加入了一个 `DispatcherServlet`。Servlet3.0规范中有这个说明，除了可以动态加Servlet,还可以动态加Listener, Filter

- o addServlet
- o addListener
- o addFilter

3. SpringBoot数据访问

3.1 数据源自动配置源码剖析

数据源配置方式

1、选择数据库驱动的库文件

在maven中配置数据库驱动

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

2、配置数据库连接

在application.properties中配置数据库连接

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://springboot_h?
useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
# spring.datasource.type=com.alibaba.druid.pool.DruidDataSource
```

3、配置spring-boot-starter-jdbc

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

4、编写测试类

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = SpringBootMytestApplication.class)
class SpringBootMytestApplicationTests {

    @Autowired
    DataSource dataSource;

    @Test
    public void contextLoads() throws SQLException {
        Connection connection = dataSource.getConnection();
    }
}
```

连接池配置方式

1、选择数据库连接池的库文件

SpringBoot提供了三种数据库连接池：

- HikariCP
- Commons DBCP2

- Tomcat JDBC Connection Pool

其中spring boot2.x版本默认使用HikariCP， maven中配置如下：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

如果不使用HikariCP， 而改用Commons DBCP2，则配置如下：

```
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-dbcp2</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.zaxxer</groupId>
            <artifactId>HikariCP</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

如果不使用HikariCP， 而改用Tomcat JDBC Connection Pool，则配置如下：

```
<dependency>
    <groupId>org.apache.tomcat</groupId>
    <artifactId>tomcat-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
    <exclusions>
        <exclusion>
            <groupId>com.zaxxer</groupId>
            <artifactId>HikariCP</artifactId>
        </exclusion>
    </exclusions>
</dependency>
```

思考：为什么说springboot默认使用的连接池类型是HikariCP，在那指定的？

数据源自动配置

spring.factories中找到数据源的配置类：

```
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,
```

```
@Configuration(proxyBeanMethods = false)
```

```

@ConditionalOnClass({ DataSource.class, EmbeddedDatabaseType.class })
@EnableConfigurationProperties(DataSourceProperties.class)
@Import({ DataSourcePoolMetadataProvidersConfiguration.class,
DataSourceInitializationConfiguration.class })
public class DataSourceAutoConfiguration {

    @Configuration(proxyBeanMethods = false)
    @Conditional(EmbeddedDatabaseCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import(EmbeddedDataSourceConfiguration.class)
    protected static class EmbeddedDatabaseConfiguration {

    }

    @Configuration(proxyBeanMethods = false)
    @Conditional(PooledDataSourceCondition.class)
    @ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
    @Import({ DataSourceConfiguration.Hikari.class,
DataSourceConfiguration.Tomcat.class,
        DataSourceConfiguration.Dbcp2.class,
DataSourceConfiguration.Generic.class,
        DataSourceJmxConfiguration.class })
    protected static class PooledDataSourceConfiguration {

    }
    ...
}

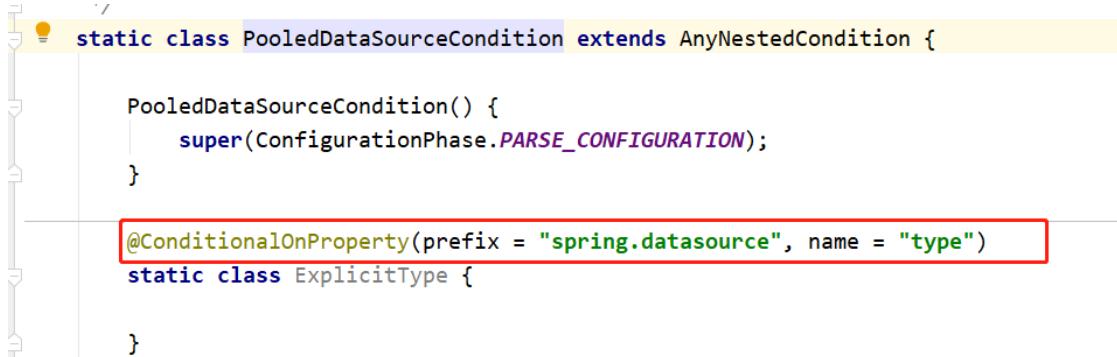
```

@Conditional(PooledDataSourceCondition.class) 根据判断条件，实例化这个类，指定了配置文件中，必须有type这个属性

```

@Configuration(proxyBeanMethods = false)
@Conditional(PooledDataSourceCondition.class)
@ConditionalOnMissingBean({ DataSource.class, XADataSource.class })
@Import({ DataSourceConfiguration.Hikari.class, DataSourceConfiguration.Tomcat.class,
        DataSourceConfiguration.Dbcp2.class, DataSourceConfiguration.Generic.class,
        DataSourceJmxConfiguration.class })
protected static class PooledDataSourceConfiguration {

```



```

static class PooledDataSourceCondition extends AnyNestedCondition {

    PooledDataSourceCondition() {
        super(ConfigurationPhase.PARSE_CONFIGURATION);
    }

    @ConditionalOnProperty(prefix = "spring.datasource", name = "type")
    static class ExplicitType {
    }
}

```

另外springboot 默认支持 type 类型设置的数据源；

```
@Import({ DataSourceConfiguration.Hikari.class,
DataSourceConfiguration.Tomcat.class,
DataSourceConfiguration.Dbcp2.class,
DataSourceConfiguration.Generic.class,
DataSourceJmxConfiguration.class })
```

```
abstract class DataSourceConfiguration {
    DataSourceConfiguration() {
    }

    protected static <T> T createDataSource(DataSourceProperties properties,
Class<? extends DataSource> type) {
        return properties.initializeDataSourceBuilder().type(type).build();
    }

    //自定义连接池 接口 spring.datasource.type 配置
    @ConditionalOnMissingBean({DataSource.class})
    @ConditionalOnProperty(
        name = {"spring.datasource.type"})
    static class Generic {
        Generic() {
        }

        @Bean
        public DataSource dataSource(DataSourceProperties properties) {
            //创建数据源 initializeDataSourceBuilder DataSourceBuilder
            return properties.initializeDataSourceBuilder().build();
        }
    }
    //Dbcp2 连接池
    @ConditionalOnClass({BasicDataSource.class})
    @ConditionalOnMissingBean({DataSource.class})
    @ConditionalOnProperty(
        name = {"spring.datasource.type"},
        havingValue = "org.apache.commons.dbcp2.BasicDataSource",
        matchIfMissing = true
    )
    static class Dbcp2 {
        Dbcp2() {
        }

        @Bean
        @ConfigurationProperties(
            prefix = "spring.datasource.dbcp2"
        )
        public BasicDataSource dataSource(DataSourceProperties properties) {
            return
(BasicDataSource)DataSourceConfiguration.createDataSource(properties,
BasicDataSource.class);
        }
    }

    //2.0 之后默认使用 hikari 连接池

    @ConditionalOnClass({HikariDataSource.class})
```

```

    @ConditionalOnMissingBean({DataSource.class})
    @ConditionalOnProperty(
        name = {"spring.datasource.type"},
        havingValue = "com.zaxxer.hikari.HikariDataSource",
        matchIfMissing = true
    )
    static class Hikari {
        Hikari() {
        }

        @Bean
        @ConfigurationProperties(
            prefix = "spring.datasource.hikari"
        )
        public HikariDataSource dataSource(DataSourceProperties properties)
    {
        HikariDataSource dataSource =
(HikariDataSource)DataSourceConfiguration.createDataSource(properties,
HikariDataSource.class);
        if (StringUtils.hasText(properties.getName())) {
            dataSource.setPoolName(properties.getName());
        }

        return dataSource;
    }
}

//2.0 之后默认不是使用 tomcat 连接池,或者使用tomcat 容器
//如果导入tomcat jdbc连接池 则使用此连接池, 在使用tomcat容器时候 或者导入此包时候
@ConditionalOnClass({org.apache.tomcat.jdbc.pool.DataSource.class})
@ConditionalOnMissingBean({DataSource.class})
//并且配置的配置是 org.apache.tomcat.jdbc.pool.DataSource 会采用tomcat 连接
池
@ConditionalOnProperty(
    name = {"spring.datasource.type"}, //name用来从application.properties
中读取某个属性值
    havingValue = "org.apache.tomcat.jdbc.pool.DataSource",
    //缺少该property时是否可以加载。如果为true, 没有该property也会正常加载; 反之
报错
    // 不管你配不配置 都以 tomcat 连接池作为连接池
    matchIfMissing = true //默认是false
)
static class Tomcat {
    Tomcat() {
    }

    //给容器中加数据源
    @Bean
    @ConfigurationProperties(
        prefix = "spring.datasource.tomcat"
    )
    public org.apache.tomcat.jdbc.pool.DataSource
dataSource(DataSourceProperties properties) {
        org.apache.tomcat.jdbc.pool.DataSource dataSource =
(org.apache.tomcat.jdbc.pool.DataSource)DataSourceConfiguration.createDataSo
urce(properties, org.apache.tomcat.jdbc.pool.DataSource.class);
        DatabaseDriver databaseDriver =
DatabaseDriver.fromJdbcUrl(properties.determineUrl());
    }
}

```

```

        String validationQuery = databaseDriver.getValidationQuery();
        if (validationQuery != null) {
            dataSource.setTestOnBorrow(true);
            dataSource.setValidationQuery(validationQuery);
        }

        return dataSource;
    }
}
}
}

```

如果在类路径没有找到 jar包 则会跑出异常

Field dataSource in com.example.springsession.demo.jpa.StudentController required a bean of type 'javax.sql.DataSource' that could not be found.

- Bean method 'dataSource' not loaded because @ConditionalOnClass did not find required class 'org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType'

- o Bean method 'dataSource' not loaded because @ConditionalOnClass did not find required classes 'javax.transaction.TransactionManager', 'org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType'

DataSourceConfiguration

配置文件中没有指定数据源时候 会根据注解判断然后选择相应的实例化数据源对象！

则 type 为空。

```

@ConditionalOnClass({HikariDataSource.class})
@ConditionalOnMissingBean({DataSource.class}) //注解判断是否执行初始化代码，即如果用户已经创建了bean，则相关的初始化代码不再执行
@ConditionalOnProperty(
    name = {"spring.datasource.type"}, //拿配置文件中的type 如果为空返回false
    havingValue = "com.zaxxer.hikari.HikariDataSource", //type 不为空则去havingValue 对比，相同则ture 否则为false
    matchIfMissing = true // 不管上面文件中是否配置，默认都进行加载 ，
    matchIfMissing的默认值为false
)
static class Hikari {
    Hikari() {
    }

    @Bean
    @ConfigurationProperties(
        prefix = "spring.datasource.hikari"
    )
    public HikariDataSource dataSource(DataSourceProperties properties) {
        //创建数据源
        HikariDataSource dataSource =
(HikariDataSource)DataSourceConfiguration.createDataSource(properties,
        //创建数据源
        HikariDataSource.class);
        if (StringUtils.hasText(properties.getName())) {
            dataSource.setPoolName(properties.getName());
        }

        return dataSource;
    }
}

```

```
}
```

createDataSource 方法

```
protected static <T> T createDataSource(DataSourceProperties properties,
class<? extends DataSource> type) {
// 使用DataSourceBuilder 建造数据源，利用反射创建type数据源，然后绑定相关属性
    return properties.initializeDataSourceBuilder().type(type).build();
}
```

DataSourceBuilder 类

设置type

```
public <D extends DataSource> DataSourceBuilder<D> type(class<D> type) {
    this.type = type;
    return this;
}
```

根据设置type的选择类型

```
private Class<? extends DataSource> getType() {
    //如果没有配置type 则为空 默认选择 findType
    Class<? extends DataSource> type = this.type != null ? this.type :
findType(this.classLoader);
    if (type != null) {
        return type;
    } else {
        throw new IllegalStateException("No supported DataSource type
found");
    }
}
```

```
public static Class<? extends DataSource> findType(ClassLoader classLoader)
{
    String[] var1 = DATA_SOURCE_TYPE_NAMES;
    int var2 = var1.length;
    int var3 = 0;

    while(var3 < var2) {
        String name = var1[var3];

        try {
            return ClassUtils.forName(name, classLoader);
        } catch (Exception var6) {
            ++var3;
        }
    }

    return null;
}
```

//数组

```
private static final String[] DATA_SOURCE_TYPE_NAMES = new String[]
{"com.zaxxer.hikari.HikariDataSource",
"org.apache.tomcat.jdbc.pool.DataSource",
"org.apache.commons.dbcp2.BasicDataSource"};
```

取出来的第一个值就是com.zaxxer.hikari.HikariDataSource，那么证实在没有指定Type的情况下，默认类型为com.zaxxer.hikari.HikariDataSource

3.2 Druid连接池的配置

整合效果实现

(1) 在pom.xml中引入druid数据源

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.10</version>
</dependency>
```

(2) 在application.yml中引入druid的相关配置

```
spring:
  datasource:
    username: root
    password: root
    url: jdbc:mysql:///springboot_h?useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC
    driver-class-name: com.mysql.cj.jdbc.Driver
    initialization-mode: always
    # 使用druid数据源
    type: com.alibaba.druid.pool.DruidDataSource
    # 数据源其他配置
    initialSize: 5
    minIdle: 5
    maxActive: 20
    maxWait: 60000
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 300000
    validationQuery: SELECT 1 FROM DUAL
    testWhileIdle: true
    testOnBorrow: false
    testOnReturn: false
    poolPreparedStatements: true
    # 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙
    filters: stat,wall,log4j
    maxPoolPreparedStatementPerConnectionSize: 20
    useGlobalDataSourceStat: true
    connectionProperties:
      druid.stat.mergeSql=true;druid.stat.showSqlMillis=500
```

进行测试：

```
com.alibaba.druid.pool.DruidDataSource : {d
```

但是Dubug查看DataSource的值，会发现有些属性是没有生效的

```
> f password = "root"
> f jdbcUrl = "jdbc:mysql://springboot_h?useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC"
> f driverClass = "com.mysql.jdbc.Driver"
f driverClassLoader = null
f connectProperties = {Properties@6539} size = 0
f passwordCallback = null
f userCallback = null
f initialSize = 0
f maxActive = 8
f minIdle = 0
```

这是因为：如果单纯在yml文件中编写如上的配置，SpringBoot肯定是读取不到druid的相关配置的。因为它并不像我们原生的jdbc，系统默认就使用DataSourceProperties与其属性进行了绑定。所以我们应该编写一个类与其属性进行绑定

(3) 编写整合druid的配置类DruidConfig

```
public class DruidConfig {

    @ConfigurationProperties(prefix = "spring.datasource")
    @Bean
    public DataSource druid(){
        return new DruidDataSource();
    }
}
```

测试的时候，突然发现控制台报错了。经过查找发现是yml文件里的

```
filters: stat,wall,log4j
```

因为我们springBoot2.0以后使用的日志框架已经不再使用log4j了。此时应该引入相应的适配器。我们可以在pom.xml文件上加入

```
<!--引入适配器-->
<dependency>
    <groupId>org.slf4j</groupId>
    <artifactId>slf4j-log4j12</artifactId>
</dependency>
```

这时进行相关测试就可以了

3.3 SpringBoot整合Mybatis

MyBatis 是一款优秀的持久层框架，Spring Boot官方虽然没有对MyBatis进行整合，但是MyBatis团队自行适配了对应的启动器，进一步简化了使用MyBatis进行数据的操作

因为Spring Boot框架开发的便利性，所以实现Spring Boot与数据访问层框架（例如MyBatis）的整合非常简单，主要是引入对应的依赖启动器，并进行数据库相关参数设置即可

整合效果实现

1、新建springboot项目，并导入mybatis的pom配置

```
<!-- 配置数据库驱动和mybatis dependency -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
</dependency>

<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <optional>true</optional>
</dependency>
```

application.yml配置

```
spring:
  datasource:
    username: root
    password: root
    url: jdbc:mysql:///springboot_h?useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC
    driver-class-name: com.mysql.jdbc.Driver
    # 使用druid数据源
    type: com.alibaba.druid.pool.DruidDataSource
```

2、基础类（使用lombok自动生成get/set方法）

```
package com.lagou.demo.domain;

import lombok.Data;

@Data
public class User {
    private Integer id;
    private String username;
    private Integer age;
}
```

3、测试dao（mybatis使用注解开发）

```

package com.lagou.dao;

import com.lagou.pojo.User;
import org.apache.ibatis.annotations.Mapper;
import org.apache.ibatis.annotations.Select;

@Mapper
public interface UserDao {

    @Select("SELECT * FROM USER")
    List<User> getUser();
}

```

4、测试service

```

package com.example.demo.service;

import com.example.demo.dao.UserDao;
import com.example.demo.domain.User;
import lombok.extern.slf4j.Slf4j;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserService {
    Logger logger = LoggerFactory.getLogger(UserService.class);

    @Autowired
    private UserDao userDao;

    public List<User> getUser(){
        List<User> userList = userDao.getUser();
        logger.info("查询出来的用户信息, {}", userList.toString());
        return userList;
    }
}

```

5、service对应的test类 (该测试类继承主测试类 (主测试类直接在启动文件上goto test即可自动生成))

```

package com.example.demo.service;

import com.example.demo.DemoApplicationTests;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;

import static org.junit.Assert.*;

```

```

public class UserServiceTest extends DemoApplicationTests {

    @Autowired
    private UserService userService;

    @Test
    public void getUser() {
        userService.getUser();
    }

}

```

```

package com.example.demo;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;

@RunWith(SpringRunner.class)
@SpringBootTest
public class DemoApplicationTests {

    @Test
    public void contextLoads() {
    }

}

```

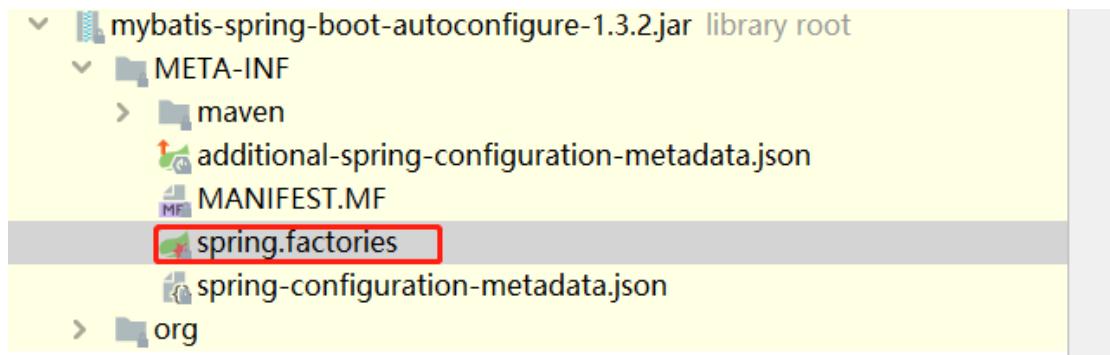
6、运行测试类输出结果

查询出来的用户信息，[User(id=1, username=test, age=11)]

3.4 Mybatis自动配置源码分析

1、springboot项目最核心的就是自动加载配置，该功能则依赖的是一个注解
@SpringBootApplication中的@EnableAutoConfiguration

2、EnableAutoConfiguration主要是通过AutoConfigurationImportSelector类来加载
以mybatis为例，*selector通过反射加载spring.factories中指定的java类，也就是加载
MybatisAutoConfiguration类（该类有Configuration注解，属于配置类）



```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.mybatis.spring.boot.autoconfigure.MybatisAutoConfiguration
```

```
/*
 * {@link EnableAutoConfiguration Auto-Configuration} for Mybatis.
 * Contributes a 重点: SqlSessionFactory 和 SqlSessionTemplate 两个类
 * {@link SqlSessionFactory} and a {@link SqlSessionTemplate}.
 *
 * If {@link org.mybatis.spring.annotation.MapperScan} is used, or a
 * configuration file is specified as a property, those will be considered,
 * otherwise this auto-configuration will attempt to register mappers based
 * on
 * the interface definitions in or under the root auto-configuration
 * package.
 *
 * @author Eddú Meléndez
 * @author Josh Long
 * @author Kazuki Shimizu
 * @author Eduardo Macarrón
 */
@org.springframework.context.annotation.Configuration
@ConditionalOnClass({ SqlSessionFactory.class, SqlSessionFactoryBean.class
})
@ConditionalOnBean(DataSource.class)
@EnableConfigurationProperties(MybatisProperties.class)
@AutoConfigureAfter(DataSourceAutoConfiguration.class)
public class MybatisAutoConfiguration {

    private static final Logger logger =
LoggerFactory.getLogger(MybatisAutoConfiguration.class);

    // //与mybatis配置文件对应
    private final MybatisProperties properties;

    private final Interceptor[] interceptors;

    private final ResourceLoader resourceLoader;

    private final DatabaseIdProvider databaseIdProvider;

    private final List<ConfigurationCustomizer> configurationCustomizers;

    public MybatisAutoConfiguration(MybatisProperties properties,
                                    ObjectProvider<Interceptor[]>
interceptorsProvider,
                                    ResourceLoader resourceLoader,
                                    ObjectProvider<DatabaseIdProvider>
databaseIdProvider,
                                    ObjectProvider<List<ConfigurationCustomizer>>
configurationCustomizersProvider) {
        this.properties = properties;
        this.interceptors = interceptorsProvider.getIfAvailable();
    }
}
```

```

        this.resourceLoader = resourceLoader;
        this.databaseIdProvider = databaseIdProvider.getIfAvailable();
        this.configurationCustomizers =
configurationCustomizersProvider.getIfAvailable();
    }

        //postConstruct作用是在创建类的时候先调用， 校验配置文件是否存在
    @PostConstruct
    public void checkConfigFileExists() {
        if (this.properties.isCheckConfigLocation() &&
Stringutils.hasText(this.properties.getConfigLocation())) {
            Resource resource =
this.resourceLoader.getResource(this.properties.getConfigLocation());
            Assert.state(resource.exists(), "Cannot find config location: " +
resource
                + " (please add config file or check your Mybatis
configuration)");
        }
    }

        //conditionalOnMissingBean作用：在没有类的时候调用， 创建sqlSessionFactory
sqlsessionfactory最主要的是创建并保存了Configuration类
    @Bean
    @ConditionalOnMissingBean
    public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws
Exception {
        SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
        factory.setDataSource(dataSource);
        factory.setVfs(SpringBootVFS.class);
        if (Stringutils.hasText(this.properties.getConfigLocation())) {

            factory.setConfigLocation(this.resourceLoader.getResource(this.properties.g
etConfigLocation()));
        }
        Configuration configuration = this.properties.getConfiguration();
        if (configuration == null &&
!Stringutils.hasText(this.properties.getConfigLocation())) {
            configuration = new Configuration();
        }
        if (configuration != null &&
!Collectionutils.isEmpty(this.configurationCustomizers)) {
            for (ConfigurationCustomizer customizer :
this.configurationCustomizers) {
                customizer.customize(configuration);
            }
        }
        factory.setConfiguration(configuration);
        if (this.properties.getConfigurationProperties() != null) {

            factory.setConfigurationProperties(this.properties.getConfigurationProperti
es());
        }
        if (!Objectutils.isEmpty(this.interceptors)) {
            factory.setPlugins(this.interceptors);
        }
        if (this.databaseIdProvider != null) {
            factory.setDatabaseIdProvider(this.databaseIdProvider);
        }
    }
}

```

```

    if (StringUtils.hasLength(this.properties.getTypeAliasesPackage())) {
        factory.setTypeAliasesPackage(this.properties.getTypeAliasesPackage());
    }
    if (StringUtils.hasLength(this.properties.getTypeHandlersPackage())) {
        factory.setTypeHandlersPackage(this.properties.getTypeHandlersPackage());
    }
    if (!ObjectUtils.isEmpty(this.properties.resolveMapperLocations())) {
        factory.setMapperLocations(this.properties.resolveMapperLocations());
    }

    // //获取SqlSessionFactory的getObject()中的对象注入Spring容器，也就是
    // SqlSessionFactory对象
    return factory.getObject();
}

@Bean
@ConditionalOnMissingBean
// 往Spring容器中注入SqlSessionTemplate对象
public SqlSessionTemplate sqlSessionTemplate(SqlSessionFactory
sqlSessionFactory) {
    ExecutorType executorType = this.properties.getExecutorType();
    if (executorType != null) {
        return new SqlSessionTemplate(sqlSessionFactory, executorType);
    } else {
        return new SqlSessionTemplate(sqlSessionFactory);
    }
}

```

3、MybatisAutoConfiguration:

- ①类中有个MybatisProperties类，该类对应的是mybatis的配置文件
- ②类中有个sqlSessionFactory方法，作用是创建SqlSessionFactory类、Configuration类（mybatis最主要的类，保存着与mybatis相关的东西）
- ③SqlSessionTemplate，作用是与mapperProxy代理类有关

sqlSessionFactory主要是通过创建了一个SqlSessionFactoryBean，这个类实现了FactoryBean接口，所以在Spring容器就会注入这个类中定义的getObject方法返回的对象。

看一下getObject()方法做了什么？

```

@Override
public SqlSessionFactory getObject() throws Exception {
    if (this.sqlSessionFactory == null) {
        afterPropertiesSet();
    }

    return this.sqlSessionFactory;
}
@Override
public void afterPropertiesSet() throws Exception {
    notNull(dataSource, "Property 'dataSource' is required");
    notNull(sqlSessionFactoryBuilder, "Property 'sqlSessionFactoryBuilder' is required");
}

```

```

        state((configuration == null && configLocation == null) || !
(configuration != null && configLocation != null),
        "Property 'configuration' and 'configLocation' can not specified
with together");

        this.sqlSessionFactory = buildSqlSessionFactory();
    }
}

```

```

protected SqlSessionFactory buildSqlSessionFactory() throws Exception {

    final Configuration targetConfiguration;

    XMLConfigBuilder xmlConfigBuilder = null;
    if (this.configuration != null) {
        targetConfiguration = this.configuration;
        if (targetConfiguration.getVariables() == null) {
            targetConfiguration.setVariables(this.configurationProperties);
        } else if (this.configurationProperties != null) {

            targetConfiguration.getVariables().putAll(this.configurationProperties);
        }
    } else if (this.configLocation != null) {
        xmlConfigBuilder = new
XMLConfigBuilder(this.configLocation.getInputStream(), null,
this.configurationProperties);
        targetConfiguration = xmlConfigBuilder.getConfiguration();
    } else {
        LOGGER.debug(
            () -> "Property 'configuration' or 'configLocation' not specified,
using default MyBatis Configuration");
        targetConfiguration = new Configuration();

        Optional.ofNullable(this.configurationProperties).ifPresent(targetConfiguration::setVariables);
    }

    Optional.ofNullable(this.objectFactory).ifPresent(targetConfiguration::setObjectFactory);

    Optional.ofNullable(this.objectWrapperFactory).ifPresent(targetConfiguration::setObjectWrapperFactory);

    Optional.ofNullable(this.vfs).ifPresent(targetConfiguration::setVfsImpl);

    if (hasLength(this.typeAliasesPackage)) {
        scanClasses(this.typeAliasesPackage,
this.typeAliasesSuperType).stream()
            .filter(clazz -> !clazz.isAnonymousClass()).filter(clazz ->
!clazz.isInterface())
            .filter(clazz ->
!clazz.isMemberClass()).forEach(targetConfiguration.getTypeAliasRegistry()::registerAlias);
    }

    if (!isEmpty(this.typeAliases)) {
        Stream.of(this.typeAliases).forEach(typeAlias -> {

```

```

        targetConfiguration.getTypeAliasRegistry().registerAlias(typeAlias);
        LOGGER.debug(() -> "Registered type alias: '" + typeAlias + "'");
    });
}

if (!isEmpty(this.plugins)) {
    Stream.of(this.plugins).forEach(plugin -> {
        targetConfiguration.addInterceptor(plugin);
        LOGGER.debug(() -> "Registered plugin: '" + plugin + "'");
    });
}

if (hasLength(this.typeHandlersPackage)) {
    scanClasses(this.typeHandlersPackage,
    TypeHandler.class).stream().filter(clazz -> !clazz.isAnonymousClass())
        .filter(clazz -> !clazz.isInterface()).filter(clazz ->
!Modifier.isAbstract(clazz.getModifiers()))
        .forEach(targetConfiguration.getTypeHandlerRegistry()::register);
}

if (!isEmpty(this.typeHandlers)) {
    Stream.of(this.typeHandlers).forEach(typeHandler -> {
        targetConfiguration.getTypeHandlerRegistry().register(typeHandler);
        LOGGER.debug(() -> "Registered type handler: '" + typeHandler +
"']");
    });
}

targetConfiguration.setDefaultEnumTypeHandler(defaultEnumTypeHandler);

if (!isEmpty(this.scriptingLanguageDrivers)) {
    Stream.of(this.scriptingLanguageDrivers).forEach(languageDriver -> {
        targetConfiguration.getLanguageRegistry().register(languageDriver);
        LOGGER.debug(() -> "Registered scripting language driver: '" +
languageDriver + "'");
    });
}
Optional.ofNullable(this.defaultScriptingLanguageDriver)
    .ifPresent(targetConfiguration::setDefaultScriptingLanguage);

if (this.databaseIdProvider != null) {// fix #64 set databaseId before
parse mapper xmls
try {

    targetConfiguration.setDatabaseId(this.databaseIdProvider.getDatabaseId(thi
s.dataSource));
} catch (SQLException e) {
    throw new NestedIOException("Failed getting a databaseId", e);
}
}

Optional.ofNullable(this.cache).ifPresent(targetConfiguration::addCache);

if (xmlConfigBuilder != null) {
    try {
        xmlConfigBuilder.parse();
    }
}

```

```

        LOGGER.debug(() -> "Parsed configuration file: '" +
this.configLocation + "'");
    } catch (Exception ex) {
        throw new NestedIOException("Failed to parse config resource: " +
this.configLocation, ex);
    } finally {
        ErrorContext.instance().reset();
    }
}

targetConfiguration.setEnvironment(new Environment(this.environment,
    this.transactionFactory == null ? new
SpringManagedTransactionFactory() : this.transactionFactory,
    this.dataSource));

if (this.mapperLocations != null) {
    if (this.mapperLocations.length == 0) {
        LOGGER.warn(() -> "Property 'mapperLocations' was specified but
matching resources are not found.");
    } else {
        for (Resource mapperLocation : this.mapperLocations) {
            if (mapperLocation == null) {
                continue;
            }
            try {
                XMLMapperBuilder xmlMapperBuilder = new
XMLMapperBuilder(mapperLocation.getInputStream(),
                    targetConfiguration, mapperLocation.toString(),
targetConfiguration.getSqlFragments());
                //这个方法已经是mybatis的源码，初始化流程
                xmlMapperBuilder.parse();
            } catch (Exception e) {
                throw new NestedIOException("Failed to parse mapping resource:
'" + mapperLocation + "'", e);
            } finally {
                ErrorContext.instance().reset();
            }
            LOGGER.debug(() -> "Parsed mapper file: '" + mapperLocation +
"']");
        }
    }
} else {
    LOGGER.debug(() -> "Property 'mapperLocations' was not specified.");
}
//这个方法已经是mybatis的源码，初始化流程
return this.sqlSessionFactoryBuilder.build(targetConfiguration);
}

```

这个已经很明显了，实际上就是调用了MyBatis的初始化流程

现在已经得到了SqlSessionFactory了，接下来就是如何扫描到相关的Mapper接口了。

这个需要看这个注解

@MapperScan(basePackages = "com.mybatis.mapper")

```

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Import(MapperScannerRegistrar.class)
@Repeatable(MapperScans.class)
public @interface MapperScan

```

通过@Import的方式会扫描到MapperScannerRegistrar类。

MapperScannerRegistrar实现了ImportBeanDefinitionRegistrar接口，那么在spring实例化之前就会调用到registerBeanDefinitions方法

```

public class MapperScannerRegistrar implements
ImportBeanDefinitionRegistrar, ResourceLoaderAware

```

```

@Override
public void registerBeanDefinitions(AnnotationMetadata
importingClassMetadata, BeanDefinitionRegistry registry) {
    //拿到MapperScan注解，并解析注解中定义的属性封装成AnnotationAttributes对象
    AnnotationAttributes mapperScanAttrs = AnnotationAttributes
        .fromMap(importingClassMetadata.getAnnotationAttributes(MapperScan.class.getName()));

    if (mapperScanAttrs != null) {
        registerBeanDefinitions(importingClassMetadata, mapperScanAttrs,
registry,
            generateBaseBeanName(importingClassMetadata, 0));
    }
}

void registerBeanDefinitions(AnnotationMetadata annoMeta,
AnnotationAttributes annoAttrs,
BeanDefinitionRegistry registry, String beanName) {

    BeanDefinitionBuilder builder =
BeanDefinitionBuilder.genericBeanDefinition(MapperScannerConfigurer.class);
    builder.addPropertyValue("processPropertyPlaceHolders", true);

    Class<? extends Annotation> annotationClass =
annoAttrs.getClass("annotationClass");
    if (!Annotation.class.equals(annotationClass)) {
        builder.addPropertyValue("annotationClass", annotationClass);
    }

    Class<?> markerInterface = annoAttrs.getClass("markerInterface");
    if (!Class.class.equals(markerInterface)) {
        builder.addPropertyValue("markerInterface", markerInterface);
    }

    Class<? extends BeanNameGenerator> generatorClass =
annoAttrs.getClass("nameGenerator");
    if (!BeanNameGenerator.class.equals(generatorClass)) {
        builder.addPropertyValue("nameGenerator",
BeanUtils.instantiateClass(generatorClass));
    }
}

```

```

        Class<? extends MapperFactoryBean> mapperFactoryBeanClass =
annoAttrs.getClass("factoryBean");
        if (!MapperFactoryBean.class.equals(mapperFactoryBeanClass)) {
            builder.addPropertyValue("mapperFactoryBeanClass",
mapperFactoryBeanClass);
        }

        String sqlSessionTemplateRef =
annoAttrs.getString("sqlSessionTemplateRef");
        if (StringUtils.hasText(sqlSessionTemplateRef)) {
            builder.addPropertyValue("sqlSessionTemplateBeanName",
annoAttrs.getString("sqlSessionTemplateRef"));
        }

        String sqlSessionFactoryRef =
annoAttrs.getString("sqlSessionFactoryRef");
        if (StringUtils.hasText(sqlSessionFactoryRef)) {
            builder.addPropertyValue("sqlSessionFactoryBeanName",
annoAttrs.getString("sqlSessionFactoryRef"));
        }

        List<String> basePackages = new ArrayList<>();
basePackages.addAll(
    Arrays.stream(annoAttrs.getStringArray("value")).filter(StringUtils::hasText)
.collect(Collectors.toList()));

        basePackages.addAll(Arrays.stream(annoAttrs.getStringArray("basePackages"))
.filter(StringUtils::hasText)
.collect(Collectors.toList()));

        basePackages.addAll(Arrays.stream(annoAttrs.getClassArray("basePackageClasses"))
.map(ClassUtils::getPackageName)
.collect(Collectors.toList()));

        if (basePackages.isEmpty()) {
            basePackages.add(getDefaultBasePackage(annoMeta));
        }

        String lazyInitialization = annoAttrs.getString("lazyInitialization");
        if (StringUtils.hasText(lazyInitialization)) {
            builder.addPropertyValue("lazyInitialization", lazyInitialization);
        }

        builder.addPropertyValue("basePackage",
StringUtils.collectionToCommaDelimitedString(basePackages));

        //把类型为MapperScannerConfigurer的注册到spring容器中
        registry.registerBeanDefinition(beanName, builder.getBeanDefinition());
    }
}

```

MapperScannerConfigurer实现了BeanDefinitionRegistryPostProcessor接口，所以接着又会扫描并调用到postProcessBeanDefinitionRegistry方法。

```

public class MapperScannerConfigurer
    implements BeanDefinitionRegistryPostProcessor, InitializingBean,
ApplicationContextAware, BeanNameAware

@Override
    public void postProcessBeanDefinitionRegistry(BeanDefinitionRegistry
registry) {
    if (this.processPropertyPlaceHolders) {
        processPropertyPlaceHolders();
    }

    ClassPathMapperScanner scanner = new ClassPathMapperScanner(registry);
    scanner.setAddToConfig(this.addToConfig);
    scanner.setAnnotationClass(this.annotationClass);
    scanner.setMarkerInterface(this.markerInterface);
    scanner.setSqlSessionFactory(this.sqlSessionFactory);
    scanner.setSqlSessionTemplate(this.sqlSessionTemplate);
    scanner.setSqlSessionFactoryBeanName(this.sqlSessionFactoryBeanName);
    scanner.setSqlSessionTemplateBeanName(this.sqlSessionTemplateBeanName);
    scanner.setResourceLoader(this.applicationContext);
    scanner.setBeanNameGenerator(this.nameGenerator);
    scanner.setMapperFactoryBeanClass(this.mapperFactoryBeanClass);
    if (StringUtils.hasText(lazyInitialization)) {
        scanner.setLazyInitialization(Boolean.valueOf(lazyInitialization));
    }
    scanner.registerFilters();
    scanner.scan(
        StringUtils.tokenizeToStringArray(this.basePackage,
ConfigurableApplicationContext.CONFIG_LOCATION_DELIMITERS));
}

public int scan(String... basePackages) {
    int beanCountAtScanStart = this.registry.getBeanDefinitionCount();

    doScan(basePackages);

    // Register annotation config processors, if necessary.
    if (this.includeAnnotationConfig) {

        AnnotationConfigUtils.registerAnnotationConfigProcessors(this.registry);
    }

    return (this.registry.getBeanDefinitionCount() -
beanCountAtScanStart);
}

@Override
    public Set<BeanDefinitionHolder> doScan(String... basePackages) {
        //这个方法主要就注册扫描basePackages路径下的mapper接口，然后封装成一个
        BeanDefinition后加入到spring容器中
        Set<BeanDefinitionHolder> beanDefinitions = super.doScan(basePackages);

        if (beanDefinitions.isEmpty()) {
            LOGGER.warn(() -> "No MyBatis mapper was found in '" +
Arrays.toString(basePackages))
        }
    }
}

```

```

        + "' package. Please check your configuration.");
    } else {
        //这个方法主要会把原BeanDefinition的beanClass类型，修改为MapperFactoryBean
        processBeanDefinitions(beanDefinitions);
    }

    return beanDefinitions;
}

```

修改了mapper的beanClass类型为MapperFactoryBean

```

private void processBeanDefinitions(Set<BeanDefinitionHolder>
beanDefinitions) {
    GenericBeanDefinition definition;
    for (BeanDefinitionHolder holder : beanDefinitions) {
        definition = (GenericBeanDefinition) holder.getBeanDefinition();
        String beanClassName = definition.getBeanClassName();
        LOGGER.debug(() -> "Creating MapperFactoryBean with name '" +
holder.getBeanName() + "' and '" + beanClassName
                + "' mapperInterface");

        // the mapper interface is the original class of the bean
        // but, the actual class of the bean is MapperFactoryBean

        definition.getConstructorArgumentValues().addGenericArgumentValue(beanClassName);
        // issue #59
        //修改beanClass类型
        definition.setBeanClass(this.mapperFactoryBeanClass);

        definition.getPropertyValues().add("addToConfig", this.addToConfig);

        boolean explicitFactoryUsed = false;
        if (StringUtils.hasText(this.sqlSessionFactoryBeanName)) {
            definition.getPropertyValues().add("sqlSessionFactory",
                    new RuntimeBeanReference(this.sqlSessionFactoryBeanName));
            explicitFactoryUsed = true;
        } else if (this.sqlSessionFactory != null) {
            definition.getPropertyValues().add("sqlSessionFactory",
                    this.sqlSessionFactory);
            explicitFactoryUsed = true;
        }

        if (StringUtils.hasText(this.sqlSessionTemplateBeanName)) {
            if (explicitFactoryUsed) {
                LOGGER.warn(
                    () -> "Cannot use both: sqlSessionTemplate and
sqlSessionFactory together. sqlSessionFactory is ignored.");
            }
            definition.getPropertyValues().add("sqlSessionTemplate",
                    new RuntimeBeanReference(this.sqlSessionTemplateBeanName));
            explicitFactoryUsed = true;
        } else if (this.sqlSessionTemplate != null) {
            if (explicitFactoryUsed) {
                LOGGER.warn(
                    () -> "Cannot use both: sqlSessionTemplate and
sqlSessionFactory together. sqlSessionFactory is ignored.");
            }
        }
    }
}

```

```

        definition.getPropertyValues().add("sqlSessionTemplate",
this.sqlSessionTemplate);
        explicitFactoryUsed = true;
    }

    if (!explicitFactoryUsed) {
        LOGGER.debug(() -> "Enabling autowire by type for MapperFactoryBean
with name '" + holder.getBeanName() + "'.");
        definition.setAutowireMode(AbstractBeanDefinition.AUTOWIRE_BY_TYPE);
    }
    definition.setLazyInit(lazyInitialization);
}
}

```

上述几步主要是完成通过

`@MapperScan(basePackages = "com.mybatis.mapper")`这个定义，扫描指定包下的 mapper 接口，然后设置每个 mapper 接口的 beanClass 属性为 MapperFactoryBean 类型并加入到 spring 的 bean 容器中。

MapperFactoryBean 实现了 FactoryBean 接口，所以当 spring 从待实例化的 bean 容器中遍历到这个 bean 并开始执行实例化时返回的对象实际上是 getObject 方法中返回的对象。

```

public class MapperFactoryBean<T> extends SqlSessionDaoSupport implements
FactoryBean<T>

```

最后看一下 MapperFactoryBean 的 getObject 方法，实际上返回的就是 mybatis 中通过 getMapper 拿到的对象，熟悉 mybatis 源码的就应该清楚，这个就是 mybatis 通过动态代理生成的 mapper 接口实现类'

```

@Override
public T getObject() throws Exception {
    return getSqlSession().getMapper(this.mapperInterface);
}

```

到此， mapper 接口现在也通过动态代理生成了实现类，并且注入到 spring 的 bean 容器中了，之后使用者就可以通过 @Autowired 或者 getBean 等方式，从 spring 容器中获取到了

3.5 SpringBoot + Mybatis 实现动态数据源切换

动态数据源介绍

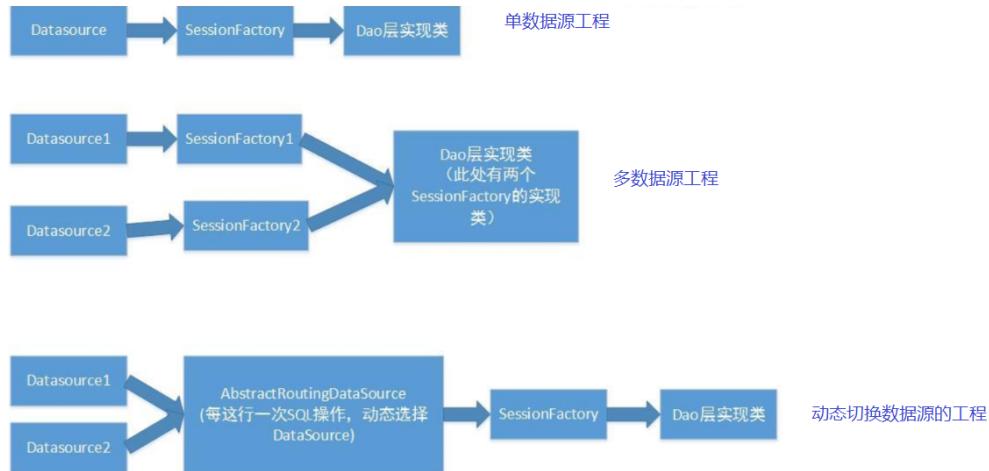
业务背景

电商订单项目分正向和逆向两个部分：其中正向数据库记录了订单的基本信息，包括订单基本信息、订单商品信息、优惠卷信息、发票信息、账期信息、结算信息、订单备注信息、收货人信息等；逆向数据库主要包含了商品的退货信息和维修信息。数据量超过 500 万行就要考虑分库分表和读写分离，那么我们在正向操作和逆向操作的时候，就需要动态的切换到相应的数据库，进行相关操作。

解决思路

现在项目的结构设计基本上是基于MVC的，那么数据库的操作集中在dao层完成，主要业务逻辑在service层处理，controller层处理请求。假设在执行dao层代码之前能够将数据源（DataSource）换成我们想要执行操作的数据源，那么这个问题就解决了

原理图



Spring内置了一个AbstractRoutingDataSource，它可以把多个数据源配置成一个Map，然后，根据不同的key返回不同的数据源。因为AbstractRoutingDataSource也是一个DataSource接口，因此，应用程序可以先设置好key，访问数据库的代码就可以从AbstractRoutingDataSource拿到对应的一个真实的数据源，从而访问指定的数据库

查看AbstractRoutingDataSource类：

```
/**  
 * Abstract {@link javax.sql.DataSource} implementation that routes {@link #getConnection()}  
 * calls to one of various target DataSources based on a lookup key. The  
 * latter is usually  
 * (but not necessarily) determined through some thread-bound transaction  
 * context.  
 *  
 * @author Juergen Hoeller  
 * @since 2.0.1  
 * @see #setTargetDataSources  
 * @see #setDefaultTargetDataSource  
 * @see #determineCurrentLookupKey()  
 */  
//翻译结果如下  
/**  
 * 抽象 {@link javax.sql.DataSource} 路由 {@link #getConnection ()} 的实现  
 * 根据查找键调用不同的目标数据之一。后者通常是  
 * (但不一定) 通过某些线程绑定事务上下文来确定。  
 *  
 * @author  
 * @since 2.0.1  
 * @see #setTargetDataSources  
 * @see #setDefaultTargetDataSource  
 * @see #determineCurrentLookupKey ()  
 */
```

```

public abstract class AbstractRoutingDataSource extends AbstractDataSource
implements InitializingBean {

    .....

    /**
     * Specify the map of target DataSources, with the lookup key as key.
     * The mapped value can either be a corresponding {@link
javax.sql.DataSource}
     * instance or a data source name String (to be resolved via a
     * {@link #setDataSourceLookup DataSourceLookup}).
     * <p>The key can be of arbitrary type; this class implements the
     * generic lookup process only. The concrete key representation will
     * be handled by {@link #resolvespecifiedLookupKey(Object)} and
     * {@link #determineCurrentLookupKey()}.
     */
    //翻译如下
    /**
     * 指定目标数据源的映射，查找键为键。
     * 映射的值可以是相应的{@link javax.sql.DataSource}
     * 实例或数据源名称字符串（要通过
     * {@link #setDataSourceLookup DataSourceLookup}）。
     * 键可以是任意类型的；这个类实现了
     * 通用查找过程只。 具体的关键表示将
     * 由{@link #resolveSpecifiedLookupKey (Object) }和
     * {@link #determineCurrentLookupKey () }。
     */
    public void setTargetDataSources(Map<Object, Object> targetDataSources)
    {
        this.targetDataSources = targetDataSources;
    }

    .....

    /**
     * Determine the current lookup key. This will typically be
     * implemented to check a thread-bound transaction context.
     * <p>Allows for arbitrary keys. The returned key needs
     * to match the stored lookup key type, as resolved by the
     * {@link #resolveSpecifiedLookupKey} method.
     */
    //翻译如下
    /**
     * 确定当前的查找键。这通常会
     * 实现以检查线程绑定的事务上下文。
     * <p> 允许任意键。返回的密钥需要
     * 与存储的查找密钥类型匹配，如
     * {@link #resolveSpecifiedLookupKey} 方法。
     */
    protected abstract Object determineCurrentLookupKey();
}

```

上面源码中还有另外一个核心的方法 `setTargetDataSources(Map<Object, Object> targetDataSources)`，它需要一个Map，在方法注释中我们可以得知，这个Map存储的就是我们配置的多个数据源的键值对。我们整理一下这个类切换数据源的运作方式，这个类在连接数据库之前会执行`determineCurrentLookupKey()`方法，这个方法返回的数据将作为key去

targetDataSources中查找相应的值，如果查找到相对应的DataSource，那么就使用此DataSource获取数据库连接

它是一个abstract类，所以我们使用的话，推荐的方式是创建一个类来继承它并且实现它的determineCurrentLookupKey()方法，这个方法介绍上面也进行了说明，就是通过这个方法进行数据源的切换

环境准备：

1.实体类

```
public class Product {

    private Integer id;
    private String name;
    private Double price;

    public Integer getId() {
        return id;
    }

    public void setId(Integer id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public Double getPrice() {
        return price;
    }

    public void setPrice(Double price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "Product{" +
            "id=" + id +
            ", name='" + name + '\'' +
            ", price=" + price +
            '}';
    }
}
```

2.ProductMapper

```

public interface ProductMapper {

    @Select("select * from product")
    public List<Product> findAllProductM();

    @Select("select * from product")
    public List<Product> findAllProducts();

}

```

3.ProductService

```

@Service
public class ProductService {

    @Autowired
    private ProductMapper productMapper;

    public void findAllProductM(){
        // 查询Master
        List<Product> allProductM = productMapper.findAllProductM();
        System.out.println(allProductM);
    }

    public void findAllProducts(){
        // 查询Slave
        List<Product> allProducts = productMapper.findAllProducts();
        System.out.println(allProducts);
    }

}

```

4.ProductController

```

@RestController
public class ProductController {

    @Autowired
    private ProductService productService;

    @GetMapping("/findAllProductM")
    public String findAllProductM() {

        productService.findAllProductM();
        return "master";
    }

    @GetMapping("/findAllProducts")
    public String findAllProducts() {
        productService.findAllProducts();
        return "slave";
    }
}

```

具体实现

第一步：配置多数据源

首先，我们在application.properties中配置两个数据源

```
spring.druid.datasource.master.password=root
spring.druid.datasource.master.username=root
spring.druid.datasource.master.jdbc-
url=jdbc:mysql://localhost:3306/product_master?
useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC
spring.druid.datasource.master.driver-class-name=com.mysql.cj.jdbc.Driver

spring.druid.datasource.slave.password=root
spring.druid.datasource.slave.username=root
spring.druid.datasource.slave.jdbc-
url=jdbc:mysql://localhost:3306/product_slave?
useUnicode=true&characterEncoding=utf-8&useSSL=true&serverTimezone=UTC
spring.druid.datasource.slave.driver-class-name=com.mysql.cj.jdbc.Driver
```

在SpringBoot的配置代码中，我们初始化两个数据源：

```
@Configuration
public class MyDataSourceConfiguratioin {

    Logger logger =
LoggerFactory.getLogger(MyDataSourceConfiguratioin.class);
    /**
     * Master data source.
     */
    @Bean("masterDataSource")
@ConfigurationProperties(prefix = "spring.druid.datasource.master")
DataSource masterDataSource() {

    logger.info("create master datasource...");
    return DataSourceBuilder.create().build();
}

    /**
     * Slave data source.
     */
    @Bean("slaveDataSource")
@ConfigurationProperties(prefix = "spring.druid.datasource.slave")
DataSource slaveDataSource() {
    logger.info("create slave datasource...");
    return DataSourceBuilder.create().build();
}

}
```

第二步：编写RoutingDataSource

然后，我们用Spring内置的RoutingDataSource，把两个真实的数据源代理为一个动态数据源。

```

public class RoutingDataSource extends AbstractRoutingDataSource {

    @Override
    protected Object determineCurrentLookupKey() {
        return "masterDataSource";
    }
}

```

,对这个 `RoutingDataSource` , 需要在SpringBoot中配置好并设置为主数据源:

```

@Bean
@Primary
DataSource primaryDataSource(
    @Autowired @Qualifier("masterDataSource") DataSource masterDataSource,
    @Autowired @Qualifier("slaveDataSource") DataSource slaveDataSource
) {
    logger.info("create routing datasource...");
    Map<Object, Object> map = new HashMap<>();
    map.put("masterDataSource", masterDataSource);
    map.put("slaveDataSource", slaveDataSource);
    RoutingDataSource routing = new RoutingDataSource();
    routing.setTargetDataSources(map);
    routing.setDefaultTargetDataSource(masterDataSource);
    return routing;
}

```

现在, `RoutingDataSource` 配置好了, 但是, 路由的选择是写死的, 即永远返回"masterDataSource",

- 现在问题来了: 如何存储动态选择的key以及在哪设置key?

在Servlet的线程模型中, 使用`ThreadLocal`存储key最合适, 因此, 我们编写一个 `RoutingDataSourceContext` , 来设置并动态存储key:

```

public class RoutingDataSourceContext {

    // holds data source key in thread local:
    static final ThreadLocal<String> threadLocalDataSourceKey = new ThreadLocal<>();

    public static String getDataSourceRoutingKey() {
        String key = threadLocalDataSourceKey.get();
        return key == null ? "masterDataSource" : key;
    }

    public RoutingDataSourceContext(String key) {
        threadLocalDataSourceKey.set(key);
    }

    public void close() {
        threadLocalDataSourceKey.remove();
    }
}

```

```
}
```

然后，修改 `RoutingDataSource`，获取key的代码如下：

```
public class RoutingDataSource extends AbstractRoutingDataSource {  
    protected Object determineCurrentLookupKey() {  
        return RoutingDataSourceContext.getDataSourceRoutingKey();  
    }  
}
```

这样，在某个地方，例如一个Controller的方法内部，就可以动态设置DataSource的Key：

```
@GetMapping("/findAllProductM")  
public String findAllProductM() {  
    String key = "masterDataSource";  
    RoutingDataSourceContext routingDataSourceContext = new  
    RoutingDataSourceContext(key);  
    productService.findAllProductM();  
  
    return "master";  
}  
  
@GetMapping("/findAllProducts")  
public String findAllProducts() {  
    String key = "slaveDataSource";  
    RoutingDataSourceContext routingDataSourceContext = new  
    RoutingDataSourceContext(key);  
    productService.findAllProducts();  
    return "slave";  
}
```

到此为止，我们已经成功实现了数据库的动态路由访问。

优化：

以上代码是可行的，但是，需要读数据库的地方，就需要加上一大段 `RoutingDataSourceContext ctx = ...` 代码，使用起来十分不便。有没有方法可以简化呢？

有！

我们仔细想想，Spring提供的声明式事务管理，就只需要一个 `@Transactional` 注解，放在某个Java方法上，这个方法就自动具有了事务。

我们也可以编写一个类似的 `@Routingwith("slaveDataSource")` 注解，放到某个Controller的方法上，这个方法内部就自动选择了对应的数据源。

```
@Target(ElementType.METHOD)  
@Retention(RetentionPolicy.RUNTIME)  
public @interface Routingwith {  
  
    String value() default "master";  
}
```

编译前需要添加一个Maven依赖：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-aop</artifactId>
</dependency>
```

切面类：

```
@Aspect
@Component
public class RoutingAspect {

    @Around("@annotation(routingwith)")
    public Object routingWithDataSource(ProceedingJoinPoint joinPoint,
Routingwith routingwith) throws Throwable {
        String key = routingwith.value();
        RoutingDataSourceContext ctx = new
RoutingDataSourceContext(key);
        return joinPoint.proceed();

    }
}
```

注意方法的第二个参数RoutingWith是Spring传入的注解实例，我们根据注解的value()获取配置的key。

改造方法：

```
@Routingwith("masterDataSource")
@GetMapping("/findAllProductM")
public String findAllProductM() {
    /* String key = "masterDataSource";
    RoutingDataSourceContext routingDataSourceContext = new
RoutingDataSourceContext(key);*/
    productService.findAllProductM();

    return "lagou";
}

@Routingwith("slaveDataSource")
@GetMapping("/findAllProducts")
public String findAllProducts() {
    /*String key = "slaveDataSource";
    RoutingDataSourceContext routingDataSourceContext = new
RoutingDataSourceContext(key);*/
    productService.findAllProducts();
    return "lagou";
}
```

到此为止，我们就实现了用注解动态选择数据源的功能

4. SpringBoot缓存深入

4.1 JSR107

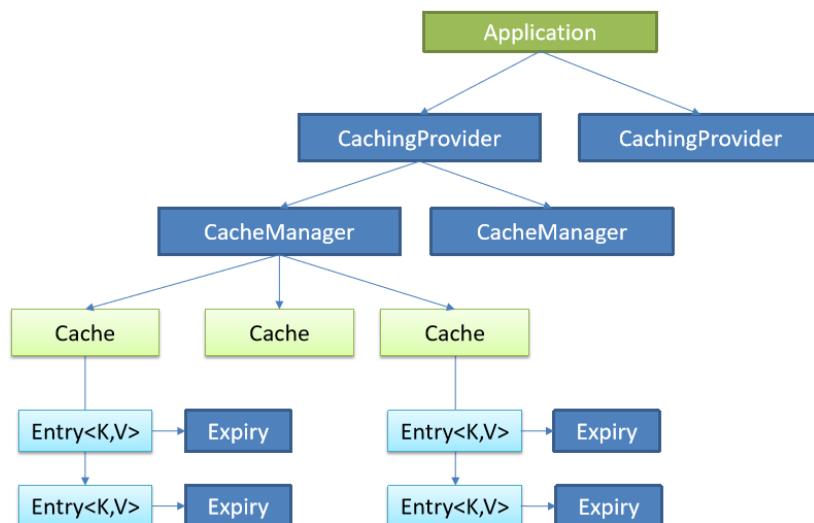
JSR是Java Specification Requests 的缩写，Java规范请求，故名思议提交Java规范，JSR-107呢就是关于如何使用缓存的规范，是java提供的一个接口规范，类似于JDBC规范，没有具体的实现，具体的实现就是reids等这些缓存。

4.1.1 JSR107核心接口

Java Caching (JSR-107) 定义了5个核心接口，分别是CachingProvider、CacheManager、Cache、Entry和Expiry。

- CachingProvider (缓存提供者)：创建、配置、获取、管理和控制多个CacheManager
- CacheManager (缓存管理器)：创建、配置、获取、管理和控制多个唯一命名的Cache，Cache存在于CacheManager的上下文中。一个CacheManager仅对应一个CachingProvider
- Cache (缓存)：是由CacheManager管理的，CacheManager管理Cache的生命周期，Cache存在于CacheManager的上下文中，是一个类似map的数据结构，并临时存储以key为索引的值。一个Cache仅被一个CacheManager所拥有
- Entry (缓存键值对)：是一个存储在Cache中的key-value对
- Expiry (缓存时效)：每一个存储在Cache中的条目都有一个定义的有效期。一旦超过这个时间，条目就自动过期，过期后，条目将不可以访问、更新和删除操作。缓存有效期可以通过ExpiryPolicy设置

4.1.2 JSR107图示



一个应用里面可以有多个缓存提供者(CachingProvider)，一个缓存提供者可以获取到多个缓存管理器(CacheManager)，一个缓存管理器管理着不同的缓存(Cache)，缓存中是一个个的缓存键值对(Entry)，每个entry都有一个有效期(Expiry)。缓存管理器和缓存之间的关系有点类似于数据库中连接池和连接的关系。

使用JSR-107需导入的依赖

```
<dependency>
    <groupId>javax.cache</groupId>
    <artifactId>cache-api</artifactId>
</dependency>
```

4.2 Spring的缓存抽象

4.2.1 缓存抽象定义

Spring从3.1开始定义了org.springframework.cache.Cache和org.springframework.cache.CacheManager接口来统一不同的缓存技术；并支持使用Java Caching (JSR-107) 注解简化我们进行缓存开发。

Spring Cache 只负责维护抽象层，具体的实现由自己的技术选型来决定。将缓存处理和缓存技术解除耦合。

每次调用需要缓存功能的方法时，Spring会检查指定参数的指定的目标方法是否已经被调用过，如果有就直接从缓存中获取方法调用后的结果，如果没有就调用方法并缓存结果后返回给用户。下次调用直接从缓存中获取。

使用Spring缓存抽象时我们需要关注以下两点：

- ① 确定那些方法需要被缓存
- ② 缓存策略

4.2.2 重要接口

- Cache：缓存抽象的规范接口，缓存实现有：RedisCache、EhCache、ConcurrentMapCache等
- CacheManager：缓存管理器，管理Cache的生命周期

4.3 Spring缓存使用

4.3.1 重要概念&缓存注解

案例实践之前，先介绍下Spring提供的重要缓存注解及几个重要概念

几个重要概念&缓存注解：

概念/注解	作用
Cache	缓存接口，定义缓存操作。实现有：RedisCache、EhCacheCache、ConcurrentMapCache等
CacheManager	缓存管理器，管理各种缓存(Cache)组件
@Cacheable	主要针对方法配置，能够根据方法的请求参数对其结果进行缓存
@CacheEvict	清空缓存
@CachePut	保证方法被调用，又希望结果被缓存。
@EnableCaching	开启基于注解的缓存
keyGenerator	缓存数据时key生成策略
serialize	缓存数据时value序列化策略

说明：

① @Cacheable标注在方法上，表示该方法的结果需要被缓存起来，缓存的键由keyGenerator的策略决定，缓存的值的形式则由serialize序列化策略决定(序列化还是json格式)；标注上该注解之后，在缓存时效内再次调用该方法时将不会调用方法本身而是直接从缓存获取结果

② @CachePut也标注在方法上，和@Cacheable相似也会将方法的返回值缓存起来，不同的是标注@CachePut的方法每次都会被调用，而且每次都会将结果缓存起来，适用于对象的更新

4.3.2 环境搭建

(1) 创建SpringBoot应用：选中Mysql、Mybatis、Web模块

(2) 创建数据库表

```
SET FOREIGN_KEY_CHECKS=0;

DROP TABLE IF EXISTS `department`;
CREATE TABLE `department` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `departmentName` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

DROP TABLE IF EXISTS `employee`;
CREATE TABLE `employee` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `lastName` varchar(255) DEFAULT NULL,
  `email` varchar(255) DEFAULT NULL,
  `gender` int(2) DEFAULT NULL,
  `d_id` int(11) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

(3) 创建表对应的实体Bean

```
@Data  
public class Employee {  
    private Integer id;  
    private String lastName;  
    private String email;  
    private Integer gender; //性别 1男 0女  
    private Integer dId;  
}  
  
@Data  
public class Department {  
    private Integer id;  
    private String departmentName;  
}
```

(4) 整合mybatis操作数据库

数据源配置：驱动可以不写，SpringBoot会根据连接自动判断

```
spring.datasource.url=jdbc:mysql://localhost:3306/springboot_h  
spring.datasource.username=root  
spring.datasource.password=root  
#spring.datasource.driver-class-name=com.mysql.jdbc.Driver  
  
#开启驼峰命名  
mybatis.configuration.map-underscore-to-camel-case=true
```

使用注解版Mybatis：使用@MapperScan指定mapper接口所在的包

```
@SpringBootApplication  
@MapperScan(basePackages = "com.lagou.cache.mappers")  
public class SpringbootCacheApplication {  
  
    public static void main(String[] args) {  
        SpringApplication.run(SpringbootCacheApplication.class, args);  
    }  
}
```

创建对应的mapper接口

```
public interface EmployeeMapper {  
  
    @Select("SELECT * FROM employee WHERE id = #{id}")  
    public Employee getEmpById(Integer id);  
  
    @Insert("INSERT INTO employee(lastName,email,gender,d_id) VALUES(#{lastName},#{email},#{gender},#{dId})")  
    public void insertEmp(Employee employee);
```

```

    @Update("UPDATE employee SET lastName = #{lastName},email = #
{email},gender = #{gender},d_id = #{dId} WHERE id = #{id}")
    public void updateEmp(Employee employee);

    @Delete("DELETE FROM employee WHERE id = #{id}")
    public void deleteEmpById(Integer id);
}

```

编写Service:

```

@Service
public class EmployeeService {

    @Autowired
    EmployeeMapper employeeMapper;

    public Employee getEmpById(Integer id){
        Employee emp = employeeMapper.getEmpById(id);
        return emp;
    }
}

```

编写Controller:

```

@RestController
public class EmployeeController {

    @Autowired
    EmployeeService employeeService;

    @GetMapping("/emp/{id}")
    public Employee getEmp(@PathVariable("id") Integer id){
        return employeeService.getEmpById(id);
    }
}

```

测试:

测试之前可以先配置一下Logger日志，让控制台将Sql打印出来:

```
logging.level.com.lagou.cache.mappers=debug
```

```

' DEBUG 28736 --- [nio-8080-exec-1] c.l.mapper.EmployeeMapper.getEmpById : ==> Preparing: SELECT * FROM employee WHERE id = ?
' DEBUG 28736 --- [nio-8080-exec-1] c.l.mapper.EmployeeMapper.getEmpById : ==> Parameters: 1(Integer)
' DEBUG 28736 --- [nio-8080-exec-1] c.l.mapper.EmployeeMapper.getEmpById : <== Total: 1
' DEBUG 28736 --- [nio-8080-exec-4] c.l.mapper.EmployeeMapper.getEmpById : ==> Preparing: SELECT * FROM employee WHERE id = ?
' DEBUG 28736 --- [nio-8080-exec-4] c.l.mapper.EmployeeMapper.getEmpById : ==> Parameters: 1(Integer)
' DEBUG 28736 --- [nio-8080-exec-4] c.l.mapper.EmployeeMapper.getEmpById : <== Total: 1

```

结论：当前还没有看到缓存效果，因为还没有进行缓存的相关设置

4.3.3 @Cacheable初体验

① 开启基于注解的缓存功能：主启动类标注@EnableCaching

```

@SpringBootApplication
@MapperScan(basePackages = "com.lagou.cache.mappers")
@EnableCaching //开启基于注解的缓存
public class SpringbootCacheApplication {

    public static void main(String[] args) {
        SpringApplication.run(SpringbootCacheApplication.class, args);
    }
}

```

② 标注缓存相关注解：@Cacheable、CacheEvict、CachePut

@Cacheable：将方法运行的结果进行缓存，以后再获取相同的数据时，直接从缓存中获取，不再调用方法

```

@Cacheable(cacheNames = {"emp"})
public Employee getEmpById(Integer id){
    Employee emp = employeeMapper.getEmpById(id);
    return emp;
}

```

@Cacheable注解的属性：

属性名	描述
cacheNames/value	指定缓存的名字，缓存使用CacheManager管理多个缓存组件Cache，这些Cache组件就是根据这个名字进行区分的。对缓存的真正CRUD操作在Cache中定义，每个缓存组件Cache都有自己唯一的名字，通过cacheNames或者value属性指定，相当于将缓存的键值对进行分组，缓存的名字是一个数组，也就是说可以将一个缓存键值对分到多个组里面
key	缓存数据时的key的值，默认是使用方法参数的值，可以使用SpEL表达式计算key的值
keyGenerator	缓存的生成策略，和key二选一，都是生成键的，keyGenerator可自定义
cacheManager	指定缓存管理器(如ConcurrentHashMap、Redis等)
cacheResolver	和cacheManager功能一样，和cacheManager二选一
condition	指定缓存的条件(满足什么条件时才缓存)，可用SpEL表达式(如#id>0，表示当入参id大于0时才缓存)
unless	否定缓存，即满足unless指定的条件时，方法的结果不进行缓存，使用unless时可以在调用的方法获取到结果之后再进行判断(如#result==null，表示如果结果为null时不缓存)
sync	是否使用异步模式进行缓存

注：

①既满足condition又满足unless条件的也不进行缓存

②使用异步模式进行缓存时(sync=true): unless条件将不被支持

可用的SpEL表达式见下表:

名字	位置	描述	示例
methodName	root object	当前被调用的方法名	#root.methodName
method	root object	当前被调用的方法	#root.method.name
target	root object	当前被调用的目标对象	#root.target
targetClass	root object	当前被调用的目标对象类	root.targetClass
args	root object	当前被调用的方法的参数列表	#root.args[0]
caches	root object	当前方法调用使用的缓存列表 (如@Cacheable(value={"cache1", "cache2"})), 则有两个cache	#root.caches[0].name
argument name	evaluation context	方法参数的名字, 可以直接 #参数名, 也可以使用#p0或#a0的形式, 0代表参数的索引	#iban、#a0、#p0
result	evaluation context	方法执行后的返回值(仅当方法执行之后的判断有效, 如"unless", "cache put"的表达式, "cache evict"的表达式 beforeInvocation=false)	#result

4.4 缓存自动配置原理源码剖析

在springBoot中所有的自动配置都是...AutoConfiguration 所以我们去搜 CacheAutoConfiguration 这个类

在这个类中有一个静态内部类 CacheConfigurationImportSelector 他有一个 selectImport 方法是用来给容器中添加一些缓存要用的组件;

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfigu\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
```

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(CacheManager.class)
@ConditionalOnBean(CacheAspectSupport.class)
@ConditionalOnMissingBean(value = CacheManager.class, name = "cacheResolver")
@EnableConfigurationProperties(CacheProperties.class)
@AutoConfigureAfter({ CouchbaseAutoConfiguration.class, HazelcastAutoConfiguration.class,
    HibernateJpaAutoConfiguration.class, RedisAutoConfiguration.class })
@Import({ CacheConfigurationImportSelector.class, CacheManagerEntityManagerFactoryDependsOnPostProcessor.class })
public class CacheAutoConfiguration {

```

```

static class CacheConfigurationImportSelector implements ImportSelector {

    @Override
    public String[] selectImports(AnnotationMetadata importingClassMetadata) {
        CacheType[] types = CacheType.values();
        String[] imports = new String[types.length];
        for (int i = 0; i < types.length; i++) {
            imports[i] = CacheConfigurations.getConfigurationClass(types[i]);
        }
        return imports;
    }
}

```

我们在这里打上断点,debug调试一下看看 imports 中有哪些缓存组件

```

return imports; imports: {"org.springframework.boot.autoconfigure.cache.GenericCacheConfiguration", "org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration", "org.springframework.boot.autoconfigure.cache.EhCacheCacheConfiguration", "org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration", "org.springframework.boot.autoconfigure.cache.InfinispanCacheConfiguration", "org.springframework.boot.autoconfigure.cache.CouchbaseCacheConfiguration", "org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration", "org.springframework.boot.autoconfigure.cache.CaffeineCacheConfiguration", "org.springframework.boot.autoconfigure.cache.SimpleCacheConfiguration", "org.springframework.boot.autoconfigure.cache.NoOpCacheConfiguration"}

```

```

oo imports = {String[10]@4526}
>   0 = "org.springframework.boot.autoconfigure.cache.GenericCacheConfiguration"
>   1 = "org.springframework.boot.autoconfigure.cache.JCacheCacheConfiguration"
>   2 = "org.springframework.boot.autoconfigure.cache.EhCacheCacheConfiguration"
>   3 = "org.springframework.boot.autoconfigure.cache.HazelcastCacheConfiguration"
>   4 = "org.springframework.boot.autoconfigure.cache.InfinispanCacheConfiguration"
>   5 = "org.springframework.boot.autoconfigure.cache.CouchbaseCacheConfiguration"
>   6 = "org.springframework.boot.autoconfigure.cache.RedisCacheConfiguration"
>   7 = "org.springframework.boot.autoconfigure.cache.CaffeineCacheConfiguration"
>   8 = "org.springframework.boot.autoconfigure.cache.SimpleCacheConfiguration"
>   9 = "org.springframework.boot.autoconfigure.cache.NoOpCacheConfiguration"

```

我们可以看到这里总共有十个缓存组件;我们随便去看一个会发现在他的注解上表明了什么时候使用这个组件;

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnClass(RedisConnectionFactory.class) // classpath下要存在对应的
                                               class文件才会进行配置
@AutoConfigureAfter(RedisAutoConfiguration.class)
@ConditionalOnBean(RedisConnectionFactory.class)
@ConditionalOnMissingBean(CacheManager.class)
@Conditional(CacheCondition.class)
class RedisCacheConfiguration {

```

十个缓存组件: 最终会发现只有 simpleCacheConfiguration 是被使用的,所以也就说明默认情况下使用 simpleCacheConfiguration;

然后我们进入到 simpleCacheConfiguration 中:

```

@Configuration(proxyBeanMethods = false)
@ConditionalOnMissingBean(CacheManager.class)
@Conditional(CacheCondition.class)
class SimpleCacheConfiguration {

```

```

@Bean
ConcurrentMapCacheManager cacheManager(CacheProperties cacheProperties,
    CacheManagerCustomizers cacheManagerCustomizers) {
    ConcurrentMapCacheManager cacheManager = new
ConcurrentMapCacheManager();
    List<String> cacheNames = cacheProperties.getCacheNames();
    if (!cacheNames.isEmpty()) {
        cacheManager.setCacheNames(cacheNames);
    }
    return cacheManagerCustomizers.customize(cacheManager);
}

}

```

我们会发现他给springBoot容器添加了一个bean，是一个`CacheManager`；

`ConcurrentMapCacheManager`实现了`CacheManager`接口

再来看`ConcurrentMapCacheManager`的`getCache`方法

```

@Override
@Nullable
public Cache getCache(String name) {
    Cache cache = this.cacheMap.get(name);
    if (cache == null && this.dynamic) {
        synchronized (this.cacheMap) {
            cache = this.cacheMap.get(name);
            if (cache == null) {
                cache = createConcurrentMapCache(name);
                this.cacheMap.put(name, cache);
            }
        }
    }
    return cache;
}

```

`getCache`方法使用了**双重锁校验**（这种验证机制一般是用在单例模式中）

我们可以看到如果没有`Cache`会调用

`cache = this.createConcurrentMapCache(name);`

```

protected Cache createConcurrentMapCache(String name) {
    SerializationDelegate actualSerialization = (isStoreByValue() ?
this.serialization : null);
    return new ConcurrentMapCache(name, new ConcurrentHashMap<>(256),
isAllowNullValues(), actualSerialization);
}

```

这个方法会创建一个`ConcurrentMapCache`这个就是我们说的`Cache`；

```
public class ConcurrentMapCache extends AbstractValueAdaptingCache {  
  
    private final String name;  
  
    private final ConcurrentHashMap<Object, Object> store;  
  
    @Nullable  
    private final serializationDelegate serialization;
```

在这个类里面有这样三个属性;

```
private final ConcurrentHashMap<Object, Object> store;
```

这个就是前文中的 Entry 用来存放键值对;

在 ConcurrentMapCache 中我们会看到一些操作 Cache 的方法,选几个重要的

```
@Override  
@Nullable  
protected Object lookup(Object key) {  
    return this.store.get(key);  
}
```

lookup 方法是根据key来找value的;

```
@Override  
public void put(Object key, @Nullable Object value) {  
    this.store.put(key, toStoreValue(value));  
}
```

put 方法顾名思义是用来添加键值对的;

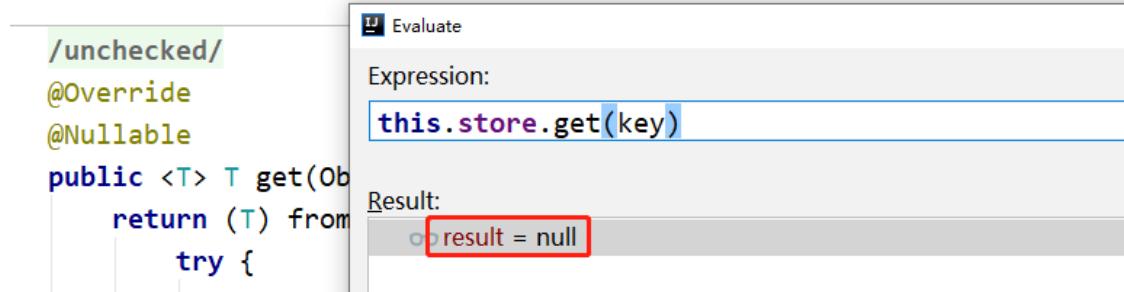
到这里基本上就结束了,接下来我们来详细分析一下 @Cacheable 注解

4.5 @Cacheable源码分析

我们在上述的两个方法上打上断点;debug运行springBoot;

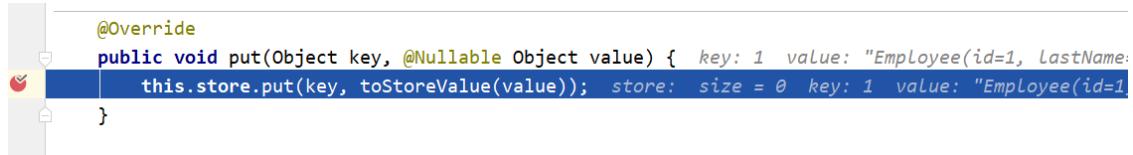
访问getEmp接口;

```
protected Object lookup(Object key) { key: 1  
    return this.store.get(key); store: size = 0 key: 1  
}
```

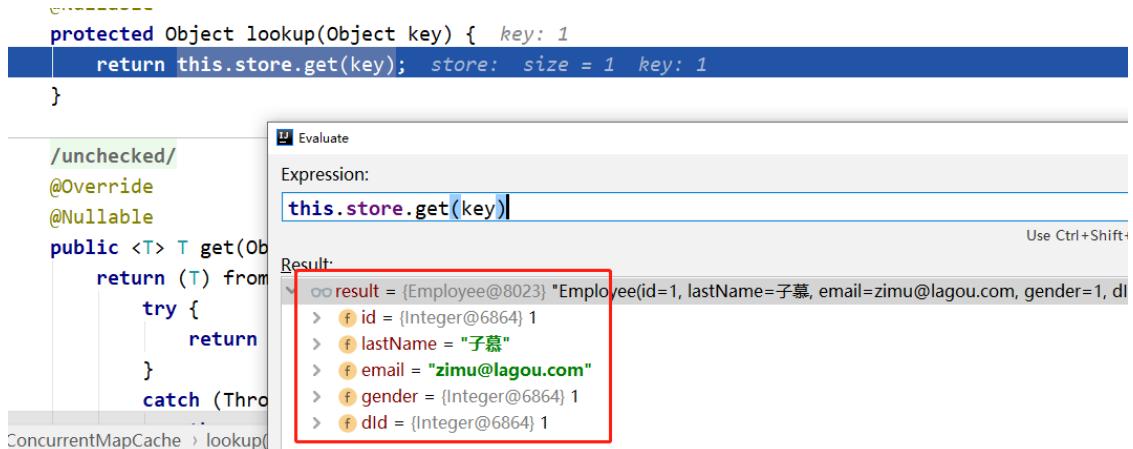


我们会发现他来到了 `lookup` 方法这里,说明注解的执行在被注解的方法前,然后这里我们会返回 `null`;

我们放行到下一个注解会发现调用了 `put` 方法



添加了 Cache; 然后我们第二次对 `getEmp` 接口发起请求我们会发现这一次缓存内容不再为 `null`



`@Cacheable` 运行流程:

①方法运行之前, 先去查询 Cache(缓存组件), 按照 `cacheNames` 指定的名字获取(CacheManager 先获取相应的缓存, 第一次获取缓存如果没有 Cache 组件会自动创建)

②去 Cache 中查找缓存的内容, 使用的 key 默认就是方法的参数:

key 默认是使用 `keyGenerator` 生成的, 默认使用的是 `SimpleKeyGenerator`

`SimpleKeyGenerator` 生成 key 的默认策略:

如果没有参数: `key = new SimpleKey();`

如果有 1 个参数: `key = 参数的值`

如果有多个参数: `key = new SimpleKey(params);`

③没有查到缓存就调用目标方法

④将目标方法返回的结果放进缓存中

总结: `@Cacheable` 标注的方法在执行之前会先检查缓存中有没有这个数据, 默认按照参数的值为 key 查询缓存, 如果没有就运行方法并将结果放入缓存, 以后再来调用时直接使用缓存中的数据。

核心：

1、使用CacheManager(ConcurrentMapCacheManager)按照名字得到Cache(ConcurrentMapCache)组件

2、key使用keyGenerator生成， 默认使用SimpleKeyGenerator

4.6 @CachePut&@CacheEvict&@CacheConfig

@CachePut

1、说明：既调用方法，又更新缓存数据，一般用于更新操作，在更新缓存时一定要和想更新的缓存有相同的缓存名称和相同的key(可类比同一张表的同一条数据)

2、运行时机：

①先调用目标方法

②将目标方法的结果缓存起来

3、示例：

```
@CachePut(value = "emp", key = "#employee.id")
public Employee updateEmp(Employee employee){
    employeeMapper.updateEmp(employee);
    return employee;
}
```

总结：@CachePut标注的方法总会被调用，且调用之后才将结果放入缓存，因此可以使用#result获取到方法的返回值。

@CacheEvict

1、说明：缓存清除，清除缓存时要指明缓存的名字和key，相当于告诉数据库要删除哪个表中的哪条数据，key默认为参数的值

2、属性：

value/cacheNames：缓存的名字

key：缓存的键

allEntries：是否清除指定缓存中的所有键值对，默认为false，设置为true时会清除缓存中的所有键值对，与key属性二选一使用

beforeInvocation：在@CacheEvict注解的方法调用之前清除指定缓存，默認為false，即在方法调用之后清除缓存，設置為true時則會在方法調用之前清除緩存(在方法調用之前還是之後清除緩存的區別在於方法調用時是否會出現異常，若不出現異常，這兩種設置沒有區別，若出現異常，設置為在方法調用之後清除緩存將不起作用，因為方法調用失敗了)

3、示例：

```
@CacheEvict(value = "emp", key = "#id", beforeInvocation = true)
public void delEmp(Integer id){
    employeeMapper.deleteEmpById(id);
}
```

@CacheConfig

1、作用：标注在类上，抽取缓存相关注解的公共配置，可抽取的公共配置有缓存名字、主键生成器等(如注解中的属性所示)

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface CacheConfig {
    String[] cacheNames() default {};
    String keyGenerator() default "";
    String cacheManager() default "";
    String cacheResolver() default "";
}
```

2、示例：通过@CacheConfig的cacheNames 属性指定缓存的名字之后，该类中的其他缓存注解就不必再写value或者cacheName了，会使用该名字作为value或cacheName的值，当然也遵循就近原则

```
@Service
@CacheConfig(cacheNames = "emp")
public class EmployeeService {

    @Autowired
    EmployeeMapper employeeMapper;

    @Cacheable
    public Employee getEmpById(Integer id) {
        Employee emp = employeeMapper.getEmpById(id);
        return emp;
    }

    @CachePut(key = "#employee.id")
    public Employee updateEmp(Employee employee) {
        employeeMapper.updateEmp(employee);
        return employee;
    }
}
```

```

    @CacheEvict(key = "#id", beforeInvocation = true)
    public void delEmp(Integer id) {
        employeeMapper.deleteEmpById(id);
    }
}

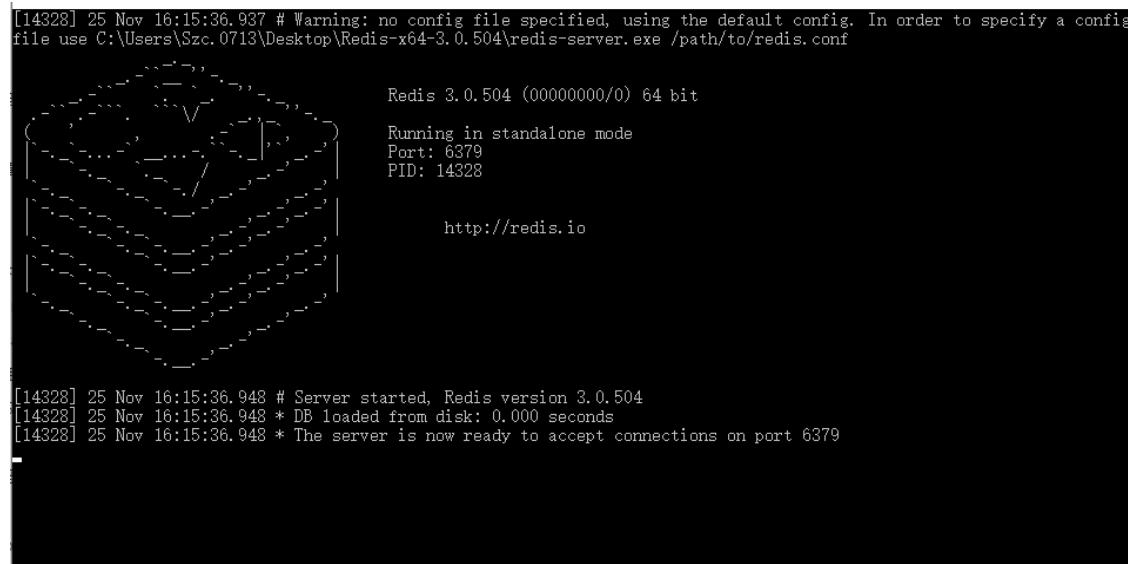
```

4.7 基于Redis的缓存实现

SpringBoot默认开启的缓存管理器是ConcurrentMapCacheManager，创建缓存组件是ConcurrentMapCache，将缓存数据保存在一个个的ConcurrentHashMap<Object, Object>中。

开发时我们可以使用缓存中间件：redis、memcache、ehcache等，这些缓存中间件的启用很简单——只要向容器中加入相关的bean就会启用，可以启用多个缓存中间件

4.7.1 安装启动Redis



4.7.2 整合Redis

①引入Redis的starter

```

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

```

引入redis的starter之后，会在容器中加入redis相关的一些bean，其中有两个跟操作redis相关的：

RedisTemplate和StringRedisTemplate(用来操作字符串：key和value都是字符串)，template中封装了操作各种数据类型的操作(stringRedisTemplate.opsForValue()、stringRedisTemplate.opsForList()等)

```

@Configuration

```

```

@ConditionalOnClass({JedisConnection.class, RedisOperations.class,
Jedis.class})
@EnableConfigurationProperties({RedisProperties.class})
public class RedisAutoConfiguration {
    public RedisAutoConfiguration() {
    }

    @Configuration
    protected static class RedisConfiguration {
        protected RedisConfiguration() {
        }

        @Bean
        @ConditionalOnMissingBean(
            name = {"redisTemplate"})
        public RedisTemplate<Object, Object>
redisTemplate(RedisConnectionFactory redisConnectionFactory) throws
UnknownHostException {
            RedisTemplate<Object, Object> template = new RedisTemplate();
            template.setConnectionFactory(redisConnectionFactory);
            return template;
        }

        @Bean
        @ConditionalOnMissingBean({StringRedisTemplate.class})
        public StringRedisTemplate
stringRedisTemplate(RedisConnectionFactory redisConnectionFactory) throws
UnknownHostException {
            StringRedisTemplate template = new StringRedisTemplate();
            template.setConnectionFactory(redisConnectionFactory);
            return template;
        }
    }
//...

```

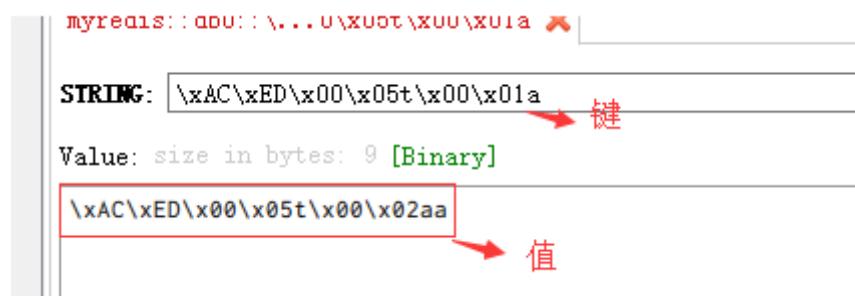
②配置redis：只需要配置redis的主机地址(端口默认即为6379，因此可以不指定)

```
spring.redis.host=127.0.0.1
```

③测试：

访问<http://localhost:8080/emp/1>

使用redis存储对象时，该对象必须可序列化(实现Serializable接口)，否则会报错，此时存储的结果在redis的管理工具中查看如下：由于序列化的原因值和键都变为了另外一种形式



SpringBoot默认采用的是JDK的对象序列化方式，我们可以切换为使用JSON格式进行对象的序列化操作，这时需要我们自定义序列化规则(当然我们也可以使用Json工具先将对象转化为Json格式之后再保存至redis，这样就无需自定义序列化)

4.8 自定义RedisCacheManager

4.8.1 Redis注解默认序列化机制

打开Spring Boot整合Redis组件提供的缓存自动配置类 RedisCacheConfiguration (org.springframework.boot.autoconfigure.cache包下的)，查看该类的源码信息，其核心代码如下

```
@Configuration
class RedisCacheConfiguration {
    @Bean
    public RedisCacheManager cacheManager(RedisConnectionFactory
                                          redisConnectionFactory, ResourceLoader
                                          resourceLoader) {

        RedisCacheManagerBuilder builder =
            RedisCacheManager.builder(redisConnectionFactory)

        .cacheDefaults(this.determineConfiguration(resourceLoader.getClassLoader()))
        ;

        List<String> cacheNames = this.cacheProperties.getCacheNames();
        if(!cacheNames.isEmpty()) {
            builder.initialCacheNames(new LinkedHashSet(cacheNames));
        }
        return
    (RedisCacheManager)this.customizerInvoker.customize(builder.build());
    }

    private org.springframework.data.redis.cache.RedisCacheConfiguration
    determineConfiguration(ClassLoader classLoader){
        if(this.redisCacheConfiguration != null) {
            return this.redisCacheConfiguration;
        } else {
            Redis redisProperties = this.cacheProperties.getRedis();
            org.springframework.data.redis.cache.RedisCacheConfiguration
            config =
                org.springframework.data.redis.cache.RedisCacheConfiguration.defaultCacheCo
                nfig();
            config =
            config.serializeValuesWith(SerializationPair.fromSerializer(
                new
                JdkSerializationRedisSerializer(classLoader)));
            ...
            return config;
        }
    }
}
```

从上述核心源码中可以看出，RedisCacheConfiguration内部同样通过Redis连接工厂RedisConnectionFactory定义了一个缓存管理器RedisCacheManager；同时定制RedisCacheManager时，也默认使用了JdkSerializationRedisSerializer序列化方式。

如果想要使用自定义序列化方式的RedisCacheManager进行数据缓存操作，可以参考上述核心代码创建一个名为cacheManager的Bean组件，并在该组件中设置对应的序列化方式即可

4.8.2 自定义RedisCacheManager

在项目的Redis配置类RedisConfig中，按照上一步分析的定制方法自定义名为cacheManager的Bean组件

```
@Bean
public RedisCacheManager cacheManager(RedisConnectionFactory
redisConnectionFactory) {
    // 分别创建String和JSON格式序列化对象，对缓存数据key和value进行转换
    RedisSerializer<String> strSerializer = new StringRedisSerializer();
    Jackson2JsonRedisSerializer jacksonSeial =
    new Jackson2JsonRedisSerializer(Object.class);
    // 解决查询缓存转换异常的问题
    ObjectMapper om = new ObjectMapper();
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
    om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
    jacksonSeial.setObjectMapper(om);
    // 定制缓存数据序列化方式及时效
    RedisCacheConfiguration config =
RedisCacheConfiguration.defaultCacheConfig()
        .entryTtl(Duration.ofDays(1))
        .serializeKeysWith(RedisSerializationContext.SerializationPair
            .fromSerializer(strSerializer))

        .serializeValuesWith(RedisSerializationContext.SerializationPair
            .fromSerializer(jacksonSeial))
        .disableCachingNullValues();
    RedisCacheManager cacheManager = RedisCacheManager
        .builder(redisConnectionFactory).cacheDefaults(config).build();
    return cacheManager;
}
```

上述代码中，在RedisConfig配置类中使用@Bean注解注入了一个默认名称为方法名的cacheManager组件。在定义的Bean组件中，通过RedisCacheConfiguration对缓存数据的key和value分别进行了序列化方式的定制，其中缓存数据的key定制为StringRedisSerializer（即String格式），而value定制为了Jackson2JsonRedisSerializer（即JSON格式），同时还使用entryTtl(Duration.ofDays(1))方法将缓存数据有效期设置为1天

完成基于注解的Redis缓存管理器RedisCacheManager定制后，可以对该缓存管理器的效果进行测试（使用自定义序列化机制的RedisCacheManager测试时，实体类可以不用实现序列化接口）

```
[{"com.lagou.pojo.Employee": {"dId": 1, "did": 1, "email": "zimu@lagou.com", "gender": 1, "id": 1, "lastName": "子慕"}}]
```

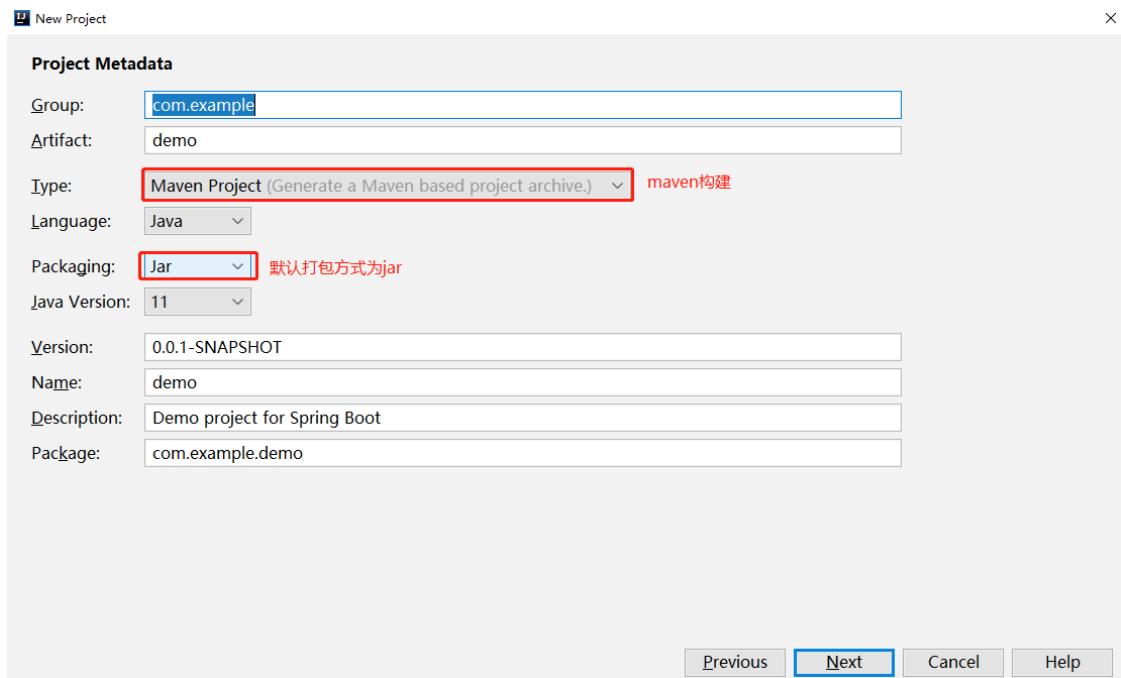
5. SpringBoot部署与监控

5.1 SpringBoot 项目部署

目前，前后端分离的架构已成主流，而使用SpringBoot构建Web应用是非常快速的，项目发布到服务器上的时候，只需要打成一个jar包，然后通过命令：java -jar jar包名称即可启动服务了。

5.1.1 jar包(官方推荐)

SpringBoot项目默认打包成jar包



jar包方式启动，也就是使用SpringBoot内置的tomcat运行。服务器上面只要你配置了jdk1.8及以上就ok，不需要外置tomcat。

1、SpringBoot将项目打包成jar包

a.首先在pom.xml文件中导入Springboot的maven依赖

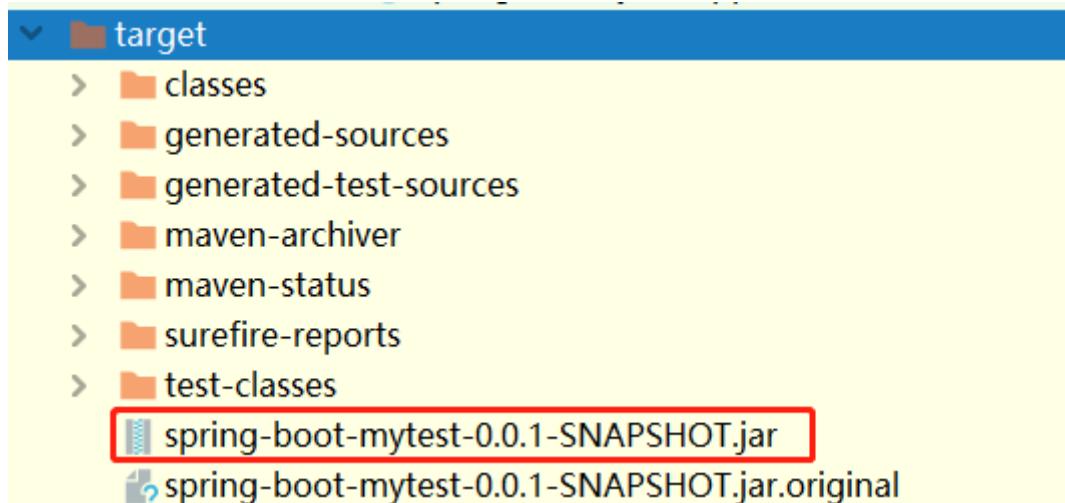
```
<!--将应用打包成一个可以执行的jar包-->
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
```

b.执行package



c.package完成以后,target中会生成一个.jar包;



d.可以将jar包上传到Linux服务器上，以jar运行（此处本地验证打包成功）

```
java -jar spring-boot-mytest-0.0.1-SNAPSHOT.jar
```

```

D:\high_springboot2.0\spring_code\spring-boot-2.2.9.RELEASE\spring-boot-mytest\target>java -jar spring-boot-mytest-0.0.1-SNAPSHOT.jar
SLF4J: Class path contains multiple SLF4J bindings.
SLF4J: Found binding in [jar:file:/D:/high_springboot2.0/spring_code/spring-boot-2.2.9.RELEASE/spring-boot-mytest/target/spring-boot-mytest-0.0.1-SNAPSHOT.jar!/BOOT-INF/lib/logback-classic-1.2.3.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: Found binding in [jar:file:/D:/high_springboot2.0/spring_code/spring-boot-2.2.9.RELEASE/spring-boot-mytest/target/spring-boot-mytest-0.0.1-SNAPSHOT.jar!/BOOT-INF/lib/slf4j-log4j12-1.7.30.jar!/org/slf4j/impl/StaticLoggerBinder.class]
SLF4J: See http://www.slf4j.org/codes.html#multiple_bindings for an explanation.
SLF4J: Actual binding is of type [ch.qos.logback.classic.util.ContextSelectorStaticBinder]


:: Spring Boot ::   (v2.2.9.RELEASE)

2020-11-25 19:31:41.217  INFO 6376 --- [           main] com.lagou.SpringBootMytestApplication : Starting SpringBootMytestApplication v0.0.1-SNAPSHOT on LAPTOP-TTLBKCRP with PID 6376 (D:\high_springboot2.0\spring_code\spring-boot-2.2.9.RELEASE\spring-boot-mytest\target\spring-boot-mytest-0.0.1-SNAPSHOT.jar started by Szc.0713 in D:\high_springboot2.0\spring_code\spring-boot-2.2.9.RELEASE\spring-boot-mytest\target)
2020-11-25 19:31:41.222  INFO 6376 --- [           main] com.lagou.SpringBootMytestApplication : No active profile set, falling back to default profiles: default
2020-11-25 19:31:42.944  INFO 6376 --- [           main] .s.d.r.c.RepositoryConfigurationDelegate : Multiple Spring Data modules found, entering strict repository configuration mode!

```

5.1.2 war包

传统的部署方式：将项目打成war包，放入tomcat 的webapps目录下面，启动tomcat，即可访问。

SpringBoot项目改造打包成war的流程

1、pom.xml配置修改

```

<packaging>jar</packaging>
//修改为
<packaging>war</packaging>

```

2、pom文件添加如些依赖

```

<!--添加servlet-api的依赖，用来打war包-->
<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <scope>provided</scope>
</dependency>

```

3、排除springboot内置的tomcat干扰

```

<!--最终打成war包，排除内置的tomcat-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </exclusion>
    </exclusions>
</dependency>

```

4、改造启动类

如果是war包发布，需要增加SpringBootServletInitializer子类，并重写其configure方法，或者将main函数所在的类继承SpringBootServletInitializer，并重写configure方法

当时打包为war时上传到tomcat服务器中访问项目始终报404错就是忽略了这个步骤！！！

改造之前：

```
@SpringBootApplication
public class MainApp {

    public static void main(String[] args) {
        SpringApplication.run(MainApp.class, args);
    }
}
```

改造之后：

```
@SpringBootApplication
public class SpringBootMytestApplication extends
SpringBootServletInitializer {

    public static void main(String[] args) {
        SpringApplication.run(SpringBootMytestApplication.class, args);
    }

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder
builder) {

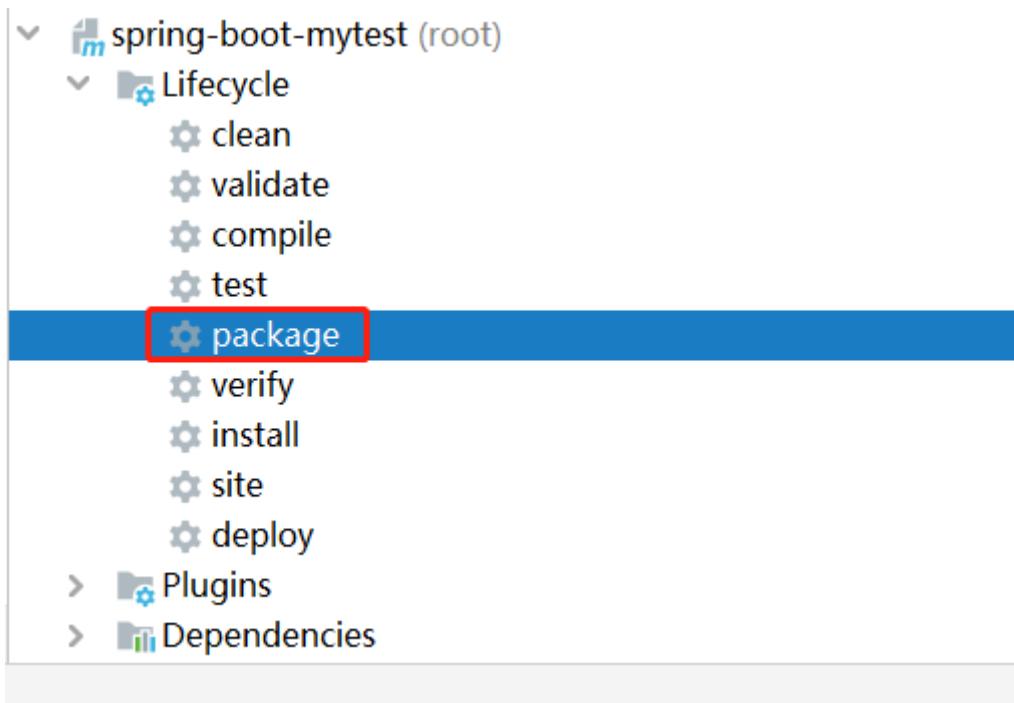
        // 注意这里要指向原先用main方法执行的Application启动类
        return builder.sources(SpringBootMytestApplication.class);
    }
}
```

这种改造方式也是官方比较推荐的方法

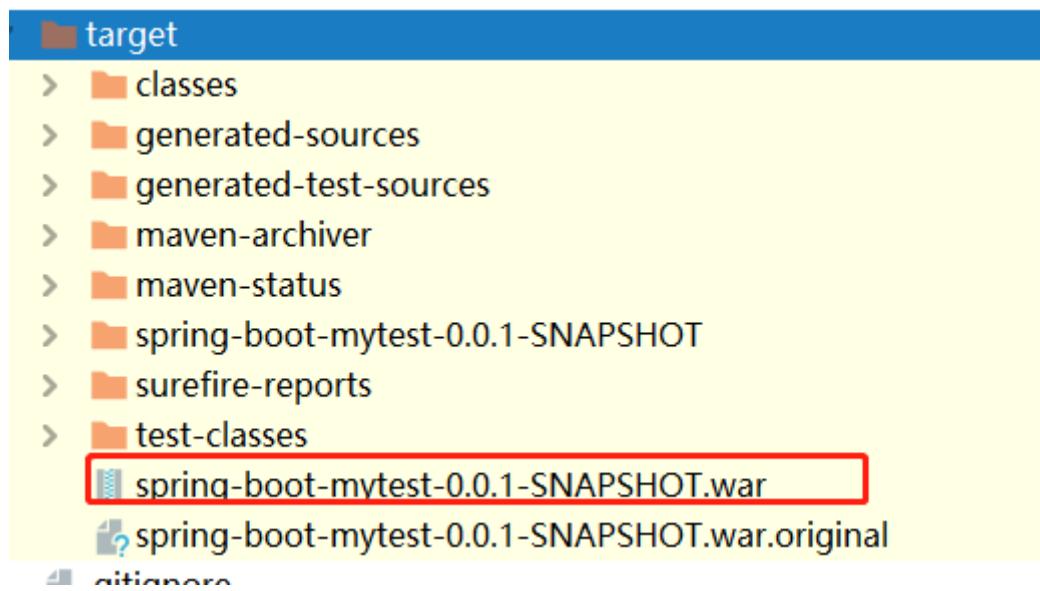
5、pom文件中不要忘了maven编译插件

```
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

6、在IDEA中使用mvn clean命令清除旧的包，并使用mvn package生成新的war包



执行完毕后，可以看到war包已经生成了，默认是在target目录下，位置可以在pom文件中进行配置：



7、使用外部Tomcat运行该 war 文件（把 war 文件直接丢到 tomcat的webapps目录，启动tomcat）

注意事项：

将项目打成war包，部署到外部的tomcat中，这个时候，不能直接访问spring boot 项目中配置文件配置的端口。application.yml中配置的server.port配置的是spring boot内置的tomcat的端口号，打成war包部署在独立的tomcat上之后，配置的server.port是不起作用的。一定要注意这一点！！

5.1.3 jar包和war包方式对比

1.SpringBoot项目打包时能打成 jar 与 war包，对比两种打包方式：

jar更加简单方便，使用 java -jar xx.jar 就可以启动。所以打成 jar 包的最多。

而 war包可以部署到tomcat的 webapps 中，随Tomcat的启动而启动。具体使用哪种方式，应视应用场景而定。

2、打jar包时不会把src/main/webapp 下的内容打到jar包里(你认为的打到jar包里面，路径是不行的会报404)

打war包时会把src/main/webapp 下的内容打到war包里

3.打成什么文件包进行部署与项目业务有关，就像提供 rest 服务的项目需要打包成 jar文件，用命令运行很方便。。。而有大量css、js、html，且需要经常改动的项目，打成 war 包去运行比较方便，因为改动静态资源可以直接覆盖，很快看到改动后的效果，这是 jar 包不能比的

(举个栗'子：项目打成 jar 包运行，一段时间后，前端要对其中某几个页面样式进行改动，使其更美观，那么改动几个css、html后，需要重新打成一个新的 jar 包，上传服务器并运行，这种改动频繁时很不友好，文件大时上传服务器很耗时，那么 war包就能免去这种烦恼，只要覆盖几个 css与html即可)

5.1.4 多环境部署

在项目运行中，包括多种环境，例如线上环境prod(product)、开发环境dev(development)、测试环境test、提测环境qa、单元测试unittest等等。不同的环境需要进行不同的配置，从而在不同的场景中跑我们的程序。例如prod环境和dev环境通常需要连接不同的数据库、需要配置不同的日志输出配置。还有一些类和方法，在不同的环境下有不同的实现方式。

Spring Boot 对此提供了支持，一方面是注解@Profile，另一方面还有多资源配置文件。

@Profile

`@profile` 注解的作用是指定类或方法在特定的 Profile 环境生效，任何 `@Component` 或 `@Configuration` 注解的类都可以使用 `@Profile` 注解。在使用DI来依赖注入的时候，能够根据 `@profile` 标明的环境，将注入符合当前运行环境的相应的bean。

使用要求：

- `@Component` 或 `@Configuration` 注解的类可以使用 `@profile`
- `@Profile` 中需要指定一个字符串，约定生效的环境

1、`@Profile` 的使用位置

(1) `@Profile` 修饰类

```
@Configuration  
@Profile("prod")  
public class JndiDataConfig {  
  
    @Bean(destroyMethod="")  
    public DataSource dataSource() throws Exception {  
        Context ctx = new InitialContext();  
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");  
    }  
}
```

(2) `@Profile` 修饰方法

```

@Configuration
public class AppConfig {

    @Bean("dataSource")
    @Profile("dev")
    public DataSource standaloneDataSource() {
        return new EmbeddedDatabaseBuilder()
            .setType(EmbeddedDatabaseType.HSQL)
            .addScript("classpath:com/bank/config/sql/schema.sql")
            .addScript("classpath:com/bank/config/sql/test-data.sql")
            .build();
    }

    @Bean("dataSource")
    @Profile("prod")
    public DataSource jndiDataSource() throws Exception {
        Context ctx = new InitialContext();
        return (DataSource) ctx.lookup("java:comp/env/jdbc/datasource");
    }
}

```

(3) @Profile 修饰注解

@Profile 注解支持定义在其他注解之上，以创建自定义场景注解。这样就创建了一个 @Dev 注解，该注解可以标识bean使用于 @Dev 这个场景。后续就不再需要使用 @Profile("dev") 的方式，这样即可以简化代码。

```

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Profile("prod")
public @interface Production {
}

```

2、profile激活

实际使用中，注解中标示了 prod、test、qa 等多个环境，运行时使用哪个 profile 由 spring.profiles.active 控制，以下说明 2 种方式：配置文件方式、命令行方式。

(1) 配置文件方式激活profile

确定当前使用的是哪个环境，这边环境的值与 application-prod.properties 中 - 后面的值对应，这是 SpringBoot 约定好的。

在 resources/application.properties 中添加下面的配置。需要注意的是，spring.profiles.active 的取值应该与 @Profile 注解中的标示保持一致。

```
spring.profiles.active=dev
```

除此之外，同理还可以在 resources/application.yml 中配置，效果是一样的：

```

spring:
  profiles:
    active: dev

```

(2) 命令行方式激活profile

在打包后运行的时候，添加参数：

```
java -jar spring-boot-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev;
```

多Profile的资源文件

除了@profile注解的可以标明某些方法和类具体在哪个环境下注入。springboot的环境隔离还可以使用多资源文件的方式，进行一些参数的配置。

①资源配置文件

Springboot的资源配置文件除了application.properties之外，还可以有对应的资源文件application-{profile}.properties。

假设，一个应用的工作环境有：dev、test、prod

那么，我们可以添加 4 个配置文件：

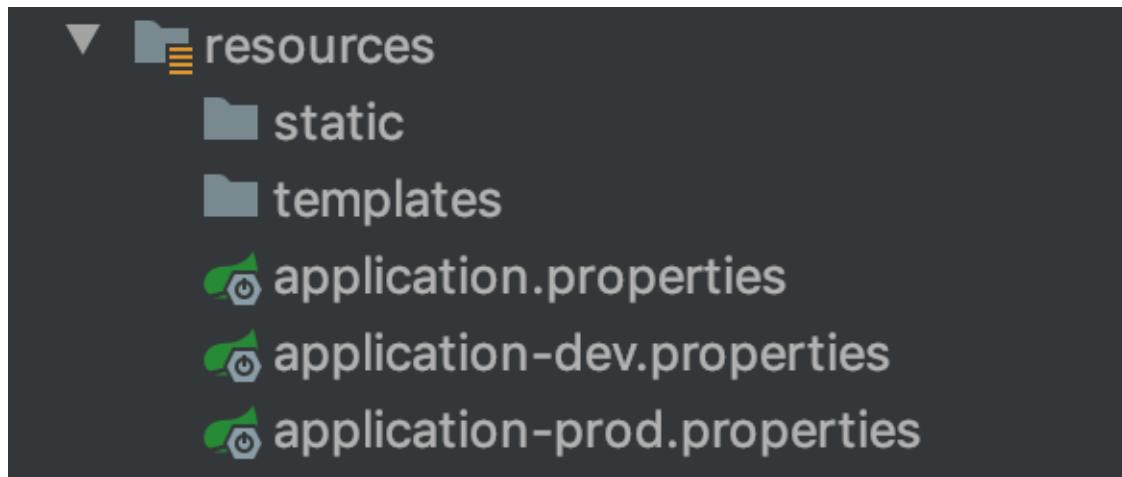
- application.properties - 公共配置
- application-dev.properties - 开发环境配置
- application-test.properties - 测试环境配置
- application-prod.properties - 生产环境配置

不同的properties配置文件也可以是在 application.properties 文件中来激活 profile：

```
spring.profiles.active = test
```

②效果演示

以下是一个多个资源配置文件的例子，主要区分了开发环境dev和线上环境prod。



在controller层中的Sound.java中新建一个接口，返回配置文件中的信息：name和location。

```
@Controller
@RequestMapping("/sound")
public class Sound {

    @Value("${com.name}")
    private String name;

    @Value("${com.location}")
    private String location;

    @RequestMapping("hello")
    @ResponseBody
```

```
public String sound1(){
    System.out.println(name + "hello Spring Boot, " +location);
    return name + ", hello Spring Boot! " +location;
}
```

application.properties文件内容如下：

```
## 多环境配置文件激活属性
spring.profiles.active=prod
```

application-dev.properties文件内容如下：

```
# 服务端口
server.port=1111

#可以定义一些自己使用的属性，然后通过@value("${属性名}")注解来加载对应的配置属性
com.name=DEV
com.location=Beijing
```

application-prod.properties文件内容如下：

```
server.port=2222

#可以定义一些自己使用的属性，然后通过@value("${属性名}")注解来加载对应的配置属性
com.name=Prod
com.location=Hubei
```

启动Springboot后，访问<http://localhost:2222/sound/hello>，则会有如下结果。如果此时访问<http://localhost:1111/sound/hello>则会无法访问，因为此时 `spring.profiles.active=prod` 激活的是prod环境，使用的是application-prod.properties中的配置。

Prod, hello Spring Boot! Hubei

更改application-dev.properties文件，`spring.profiles.active=dev`激活dev环境。重启Springboot则可以访问<http://localhost:1111/sound/hello>。

DEV, hello Spring Boot! Beijing

5.2 SpringBoot 监控

微服务的特点决定了功能模块的部署是分布式的，大部分功能模块都是运行在不同的机器上，彼此通过服务调用进行交互，前后台的业务流会经过很多个微服务的处理和传递，出现了异常如何快速定位是哪个环节出现了问题？

在这种情况下，微服务的监控显得尤为重要。springboot作为微服务框架，除了它强大的快速开发功能外，还有就是它提供了actuator模块，引入该模块能够自动为springboot应用提供一系列用于监控的端点

5.2.1 Actuator

什么是Actuator

Actuator是spring boot的一个附加功能,可帮助你在应用程序生产环境时监视和管理应用程序。可以使用HTTP的各种请求来监管,审计,收集应用的运行情况。Spring Boot Actuator提供了对单个Spring Boot的监控,信息包含: 应用状态、内存、线程、堆栈等等,比较全面的监控了Spring Boot应用的整个生命周期。特别对于微服务管理十分有意义。

Actuator 的 REST 接口

Actuator 监控分成两类: 原生端点和用户自定义端点; 自定义端点主要是指扩展性, 用户可以根据自己的实际应用, 定义一些比较关心的指标, 在运行期进行监控。

原生端点是在应用程序里提供众多 Web 接口, 通过它们了解应用程序运行时的内部状况。原生端点又可以分成三类:

- 应用配置类: 可以查看应用在运行期的静态信息: 例如自动配置信息、加载的 springbean 信息、yml 文件配置信息、环境信息、请求映射信息;
- 度量指标类: 主要是运行期的动态信息, 例如堆栈、请求链、一些健康指标、metrics 信息等;
- 操作控制类: 主要是指 shutdown, 用户可以发送一个请求将应用的监控功能关闭。

Actuator 提供了 13 个接口, 具体如下表所示。

HTTP方法	路径	描述
GET	/auditevents	显示应用暴露的审计事件(比如认证进入、订单失败)
GET	/beans	描述应用程序上下文里全部的 Bean, 以及它们的关系
GET	/conditions	就是 1.0 的 /autoconfig , 提供一份自动配置生效的条件情况, 记录哪些自动配置条件通过了, 哪些没通过
GET	/configprops	描述配置属性(包含默认值)如何注入Bean
GET	/env	获取全部环境属性
GET	/env/{name}	根据名称获取特定的环境属性值
GET	/flyway	提供一份 Flyway 数据库迁移信息
GET	/liquibase	显示Liquibase 数据库迁移的纤细信息
GET	/health	报告应用程序的健康指标, 这些值由 HealthIndicator 的实现类提供
GET	/heapdump	dump 一份应用的 JVM 堆信息
GET	/httptrace	显示HTTP足迹, 最近100个HTTP request/response
GET	/info	获取应用程序的定制信息, 这些信息由info打头的属性提供
GET	/logfile	返回log file中的内容(如果 logging.file 或者 logging.path 被设置)
GET	/loggers	显示和修改配置的loggers
GET	/metrics	报告各种应用程序度量信息, 比如内存用量和HTTP请求计数
GET	/metrics/{name}	报告指定名称的应用程序度量值
GET	/scheduledtasks	展示应用中的定时任务信息
GET	/sessions	如果我们使用了 Spring Session 展示应用中的 HTTP sessions 信息
POST	/shutdown	关闭应用程序, 要求endpoints.shutdown.enabled设置为 true
GET	/mappings	描述全部的 URI路径, 以及它们和控制器(包含Actuator端点)的映射关系
GET	/threaddump	获取线程活动的快照

体验Actuator

使用Actuator功能与springBoot使用其他功能一样简单, 只需要在pom.xml中添加如下依赖:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

为了保证 actuator 暴露的监控接口的安全性，需要添加安全控制的依赖 `spring-boot-start-security` 依赖，访问应用监控端点时，都需要输入验证信息。Security 依赖，可以选择不加，不进行安全管理。

配置文件

```
info.app.name=spring-boot-actuator
info.app.version= 1.0.0
info.app.test=test

management.endpoints.web.exposure.include=*

#展示细节，除了always之外还有when-authorized、never，默认值是never
management.endpoint.health.show-details=always
#management.endpoints.web.base-path=/monitor

management.endpoint.shutdown.enabled=true
```

- `management.endpoints.web.base-path=/monitor` 代表启用单独的url地址来监控 Spring Boot 应用，为了安全一般都启用独立的端口来访问后端的监控信息
- `management.endpoint.shutdown.enabled=true` 启用接口关闭 Spring Boot

配置完成之后，启动项目就可以继续验证各个监控功能了。

属性详解

在 Spring Boot 2.x 中为了安全期间，Actuator 只开放了两个端点 `/actuator/health` 和 `/actuator/info`。可以在配置文件中设置打开。

可以打开所有的监控点

```
management.endpoints.web.exposure.include=*
```

也可以选择打开部分

```
management.endpoints.web.exposure.include=beans,trace
```

Actuator 默认所有的监控点路径都在 `/actuator/*`，当然如果有需要这个路径也支持定制。

```
management.endpoints.web.base-path=/manage
```

设置完重启后，再次访问地址就会变成 `/manage/*`

Actuator 几乎监控了应用涉及的方方面面，我们重点讲述一些经常在项目中常用的属性。

health

health 主要用来检查应用的运行状态，这是我们使用最高频的一个监控点。通常使用此接口提醒我们应用实例的运行状态，以及应用不“健康”的原因，比如数据库连接、磁盘空间不够等。

默认情况下 health 的状态是开放的，添加依赖后启动项目，访问：

`http://localhost:8080/actuator/health` 即可看到应用的状态。

用的状态。

```
{  
    "status" : "UP"  
}
```

health 通过合并几个健康指数检查应用的健康情况。Spring Boot Actuator 有几个预定义的健康指标比如 `DataSourceHealthIndicator`, `DiskSpaceHealthIndicator`, `MongoHealthIndicator`, `RedisHealthIndicator` 等，它使用这些健康指标作为健康检查的一部分。

举个例子，如果你的应用使用 Redis，`RedisHealthIndicator` 将被当作检查的一部分；如果使用 MongoDB，那么 `MongoHealthIndicator` 将被当作检查的一部分。

可以在配置文件中关闭特定的健康检查指标，比如关闭 redis 的健康检查：

```
management.health.redis.enabled=false
```

默认，所有的这些健康指标被当作健康检查的一部分。

info

info 就是我们自己配置在配置文件中以 info 开头的配置信息，比如我们在示例项目中的配置是：

```
info.app.name=spring-boot-actuator  
info.app.version= 1.0.0  
info.app.test= test
```

启动示例项目，访问：`http://localhost:8080/actuator/info` 返回部分信息如下：

```
{  
    "app": {  
        "name": "spring-boot-actuator",  
        "version": "1.0.0",  
        "test": "test"  
    }  
}
```

beans

根据示例就可以看出，展示了 bean 的别名、类型、是否单例、类的地址、依赖等信息。

启动示例项目，访问：`http://localhost:8080/actuator/beans` 返回部分信息如下：

```
[  
{  
    "context": "application:8080:management",  
    "parent": "application:8080",  
    "beans": [  
        {  
            "name": "management",  
            "type": "org.springframework.boot.actuate.endpoint.web.EndpointHandler",  
            "scope": "singleton",  
            "target": "management",  
            "factory": null,  
            "factoryMethod": null,  
            "qualifiers": null,  
            "aliases": null,  
            "dependencies": null  
        },  
        {  
            "name": "managementConfig",  
            "type": "org.springframework.boot.actuate.endpoint.web.EndpointHandler",  
            "scope": "singleton",  
            "target": "managementConfig",  
            "factory": null,  
            "factoryMethod": null,  
            "qualifiers": null,  
            "aliases": null,  
            "dependencies": null  
        },  
        {  
            "name": "managementCondition",  
            "type": "org.springframework.boot.actuate.endpoint.web.EndpointHandler",  
            "scope": "singleton",  
            "target": "managementCondition",  
            "factory": null,  
            "factoryMethod": null,  
            "qualifiers": null,  
            "aliases": null,  
            "dependencies": null  
        },  
        {  
            "name": "managementInfo",  
            "type": "org.springframework.boot.actuate.endpoint.web.EndpointHandler",  
            "scope": "singleton",  
            "target": "managementInfo",  
            "factory": null,  
            "factoryMethod": null,  
            "qualifiers": null,  
            "aliases": null,  
            "dependencies": null  
        },  
        {  
            "name": "managementLog",  
            "type": "org.springframework.boot.actuate.endpoint.web.EndpointHandler",  
            "scope": "singleton",  
            "target": "managementLog",  
            "factory": null,  
            "factoryMethod": null,  
            "qualifiers": null,  
            "aliases": null,  
            "dependencies": null  
        },  
        {  
            "name": "managementMetrics",  
            "type": "org.springframework.boot.actuate.endpoint.web.EndpointHandler",  
            "scope": "singleton",  
            "target": "managementMetrics",  
            "factory": null,  
            "factoryMethod": null,  
            "qualifiers": null,  
            "aliases": null,  
            "dependencies": null  
        },  
        {  
            "name": "managementTracing",  
            "type": "org.springframework.boot.actuate.endpoint.web.EndpointHandler",  
            "scope": "singleton",  
            "target": "managementTracing",  
            "factory": null,  
            "factoryMethod": null,  
            "qualifiers": null,  
            "aliases": null,  
            "dependencies": null  
        }  
    ]  
}
```

```

        "bean": "embeddedServletContainerFactory",
        "aliases": [
            ],
            "scope": "singleton",
            "type":
"org.springframework.boot.context.embedded.tomcat.TomcatEmbeddedServletConta
inerFactory",
            "resource": "null",
            "dependencies": [
                ]
            },
            {
                "bean": "endpointWebMvcChildContextConfiguration",
                "aliases": [
                    ],
                    "scope": "singleton",
                    "type":
"org.springframework.boot.actuate.autoconfigure.EndpointWebMvcChildContextCo
nfiguration$$EnhancerBySpringCGLIB$$a4a10f9d",
                    "resource": "null",
                    "dependencies": [
                        ]
                    }
            }
        ]
    ]
]

```

conditions

Spring Boot 的自动配置功能非常便利，但有时候也意味着出问题比较难找出具体的原因。使用 conditions 可以在应用运行时查看代码了某个配置在什么条件下生效，或者某个自动配置为什么没有生效。

启动示例项目，访问：<http://localhost:8080/actuator/conditions> 返回部分信息如下：

```

{
    "positiveMatches": {
        "DevToolsDataSourceAutoConfiguration": {
            "notMatched": [
                {
                    "condition": "DevToolsDataSourceAutoConfiguration.DevToolsDataSourceCondition",
                    "message": "DevTools DataSource Condition did not find a single DataSource bean"
                }
            ],
            "matched": [ ]
        },
        "RemoteDevToolsAutoConfiguration": {
            "notMatched": [
                {
                    "condition": "OnPropertyCondition",
                    "message": "@ConditionalOnProperty (spring.devtools.remote.secret) did not find property 'secret'"
                }
            ]
        }
    }
}

```

```

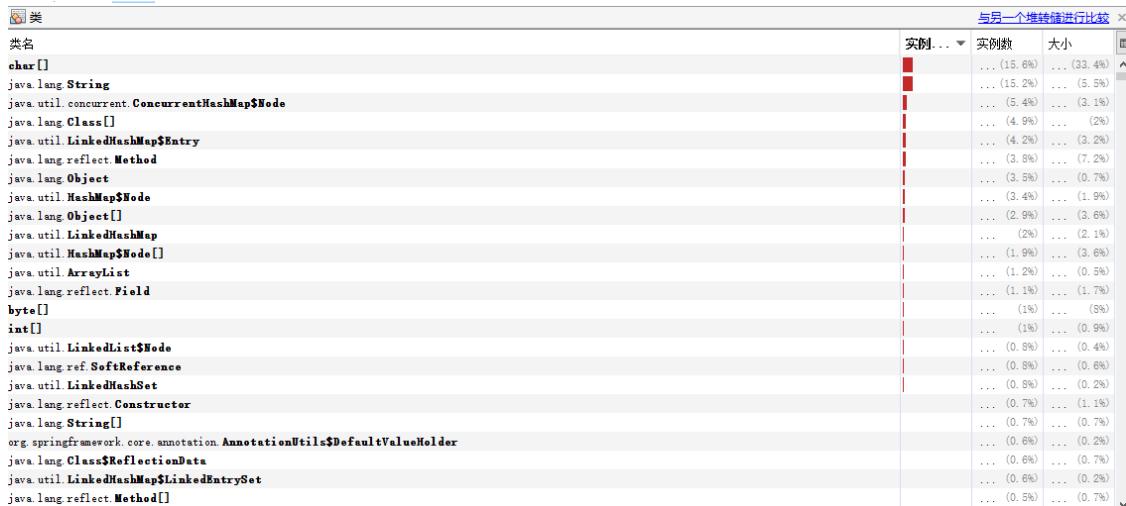
        }
    ],
    "matched": [
        {
            "condition": "OnClassCondition",
            "message": "@ConditionalOnClass found required classes
'javax.servlet.Filter', 'org.springframework.http.server.ServerHttpRequest';
@ConditionalOnMissingClass did not find unwanted class"
        }
    ]
}
}

```

heapdump

返回一个 GZip 压缩的 JVM 堆 dump

启动示例项目，访问：<http://localhost:8080/actuator/heapdump> 会自动生成一个 Jvm 的堆文件 heapdump，我们可以使用 JDK 自带的 Jvm 监控工具 VisualVM 打开此文件查看内存快照。类似如下图：



mappings

描述全部的 URI 路径，以及它们和控制器的映射关系

启动示例项目，访问：<http://localhost:8080/actuator/mappings> 返回部分信息如下：

```
{
    "/**/favicon.ico": {
        "bean": "faviconHandlerMapping"
    },
    "{[/hello]}": {
        "bean": "requestMappingHandlerMapping",
        "method": "public java.lang.String com.neo.controller.HelloController.index()"
    },
    "{[/error]}": {
        "bean": "requestMappingHandlerMapping",
        "method": "public org.springframework.http.ResponseEntity<java.util.Map<java.lang.String, java.lang.Object>> org.springframework.boot.autoconfigure.web.BasicErrorController.error(javax.servlet.http.HttpServletRequest)"
    }
}
```

threaddump

/threaddump 接口会生成当前线程活动的快照。这个功能非常好，方便我们在日常定位问题的时候查看线程的情况。主要展示了线程名、线程ID、线程的状态、是否等待锁资源等信息。

启动示例项目，访问：<http://localhost:8080/actuator/threaddump> 返回部分信息如下：

```
[
{
    "threadName": "http-nio-8088-exec-6",
    "threadId": 49,
    "blockedTime": -1,
    "blockedCount": 0,
    "waitedTime": -1,
    "waitedCount": 2,
    "lockName": "java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject@1630a501",
    "lockOwnerId": -1,
    "lockOwnerName": null,
    "inNative": false,
    "suspended": false,
    "threadstate": "WAITING",
    "stackTrace": [
        {
            "methodName": "park",
            "fileName": "unsafe.java",
            "lineNumber": -2,
            "className": "sun.misc.Unsafe",
            "nativeMethod": true
        },
        ...
        {
            "methodName": "run",
            "fileName": "TaskThread.java",
            "lineNumber": 61,
            "className": "org.apache.tomcat.util.threads.TaskThread$WrappingRunnable",
            "nativeMethod": false
        }
    ]
}
```

```
        "nativeMethod": false
    }
    ...
],
"lockInfo": {
    "className":
"java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject",
    "identityHashCode": 372286721
}
}
...
]
```

生产出现问题的时候，可以通过应用的线程快照来检测应用正在执行的任务。

shutdown

开启接口优雅关闭 Spring Boot 应用，要使用这个功能首先需要在配置文件中开启：

```
management.endpoint.shutdown.enabled=true
```

配置完成之后，启动示例项目，使用 curl 模拟 post 请求访问 shutdown 接口。

shutdown 接口默认只支持 post 请求。

```
curl -X POST "http://localhost:8080/actuator/shutdown"
{
    "message": "Shutting down, bye..."
}
```

此时你会发现应用已经被关闭。

5.2.2 Spring Boot Admin

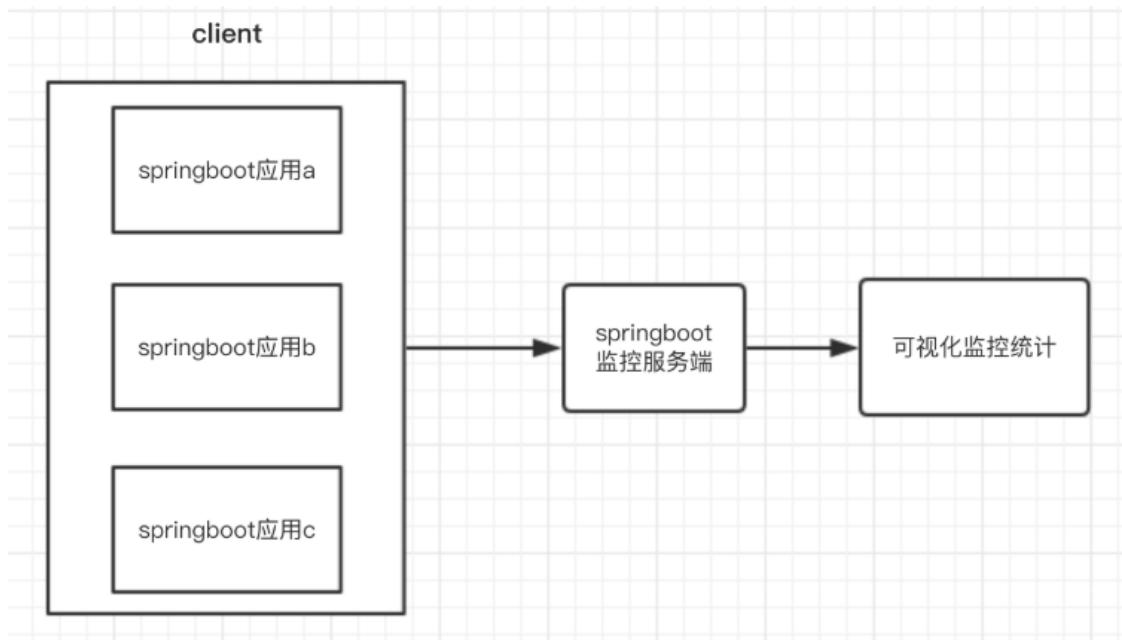
什么是Spring Boot Admin

对于spring actuator而言，最大的缺点在于是以json形式来进行展示，为了更好的进行监控显示，我们来介绍一个更加方便的工具：spring boot admin。

Spring Boot Admin：可视化后台管理系统

Spring Boot Admin 是一个针对spring-boot的actuator接口进行UI美化封装的监控工具。他可以返回在列表中浏览所有被监控spring-boot项目的基本信息比如：Spring容器管理的所有的bean、详细的Health信息、内存信息、JVM信息、垃圾回收信息、各种配置信息（比如数据源、缓存列表和命中率）等，Threads 线程管理，Environment 管理等。

利用springbootadmin进行监控的架构图如下：



springbootadmin监控

通俗点，就是我们如果有n个springboot业务系统需要监控的话，那么需要一个额外的springbootadmin应用来进行监控这些client，client和server之间需要做一点配置即可。

搭建Server端

pom.xml

```

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
</dependency>

```

application.yml

```

server:
  port: 8081

```

@EnableAdminServer

```

@EnableAdminServer
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }
}

```

启动服务端：

The screenshot shows the Spring Boot Admin interface. At the top, there's a navigation bar with links for Wallboard, Applications, Journal, About, and Logout. Below the navigation, there are three sections: APPLICATIONS (0), INSTANCES (0), and STATUS (all up). A message below the APPLICATIONS section says "No applications registered.".

目前client监控信息为空

搭建client端

pom.xml

```

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

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>

<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.1.0</version>
</dependency>

```

application.yml

```

server:
  port: 8080
#自定义配置信息用于"/actuator/info"读取
info:
  name: 老王
  age: 100
  phone: 110

#通过下面的配置启用所有的监控端点，默认情况下，这些端点是禁用的;
management:
  endpoints:

```

```

web:
  exposure:
    include: "*"
endpoint:
  health:
    show-details: always

## 将Client作为服务注册到Server，通过Server来监听项目的运行情况
spring:
  boot:
    admin:
      client:
        url: http://localhost:8081
##application实例名
application:
  name : spring-boot-admin-client

```

app.java

```

@RestController
@SpringBootApplication
public class App {

    public static void main(String[] args) {
        SpringApplication.run(App.class, args);
    }

    @RequestMapping("/index")
    public String index() {
        return "这是 index";
    }

    @RequestMapping("/home")
    public String home() {
        return "这是 home";
    }
}

```

启动 client.....

几秒后刷新，可以看到 client 端已注册到 server。

The screenshot shows the Spring Boot Admin dashboard. At the top, there's a navigation bar with links for Wallboard, Applications, Journal, About, and Logout. Below the navigation, there are three summary sections: 'APPLICATIONS' (1), 'INSTANCES' (1), and 'STATUS' (all up). Under the 'APPLICATIONS' section, there's a table with one row. The table has columns for 'UP' (with a green checkmark icon), 'spring-boot-admin-client', '33s', and 'http://192.168.1.103:8080/'. The 'UP' column contains a green checkmark icon.

查看 client 详细信息：

localhost:8081/instances/e68c97ccce01/details

Spring Boot Admin

spring-boot-admin-client
Id: e68c97ccce01

应用端 应用 日志报表 关于我们 zh-CN

Insights

性能 环境 类 配置属性 计划任务 日志配置 JVM 映射 缓存

细节

信息

app	name: spring-boot-actuator version: 1.0.0 test: test
name	老王
age	100
phone	110

元数据

startup	2020-12-24T18:04:27.2665501+08:00
---------	-----------------------------------

健康

Instance

db	UP
database	MySQL
result	1
validationQuery	/* ping */ SELECT 1

diskSpace

total	382 GB
free	97.7 GB
threshold	10.5 MB

ping

ping	UP
------	----

<http://192.168.42.1:8080/> <http://192.168.42.1:8080/actuator> <http://192.168.42.1:8080/actuator/health>