

0. 学习目标

- 说出Spring Boot的作用
- 应用Spring Boot Yaml配置文件
- 了解Spring Boot自动配置原理
- 使用Spring Boot整合SpringMVC
- 使用Spring Boot整合连接池
- 使用Spring Boot整合Mybatis
- 使用Spring Boot整合Redis
- 部署Spring Boot项目

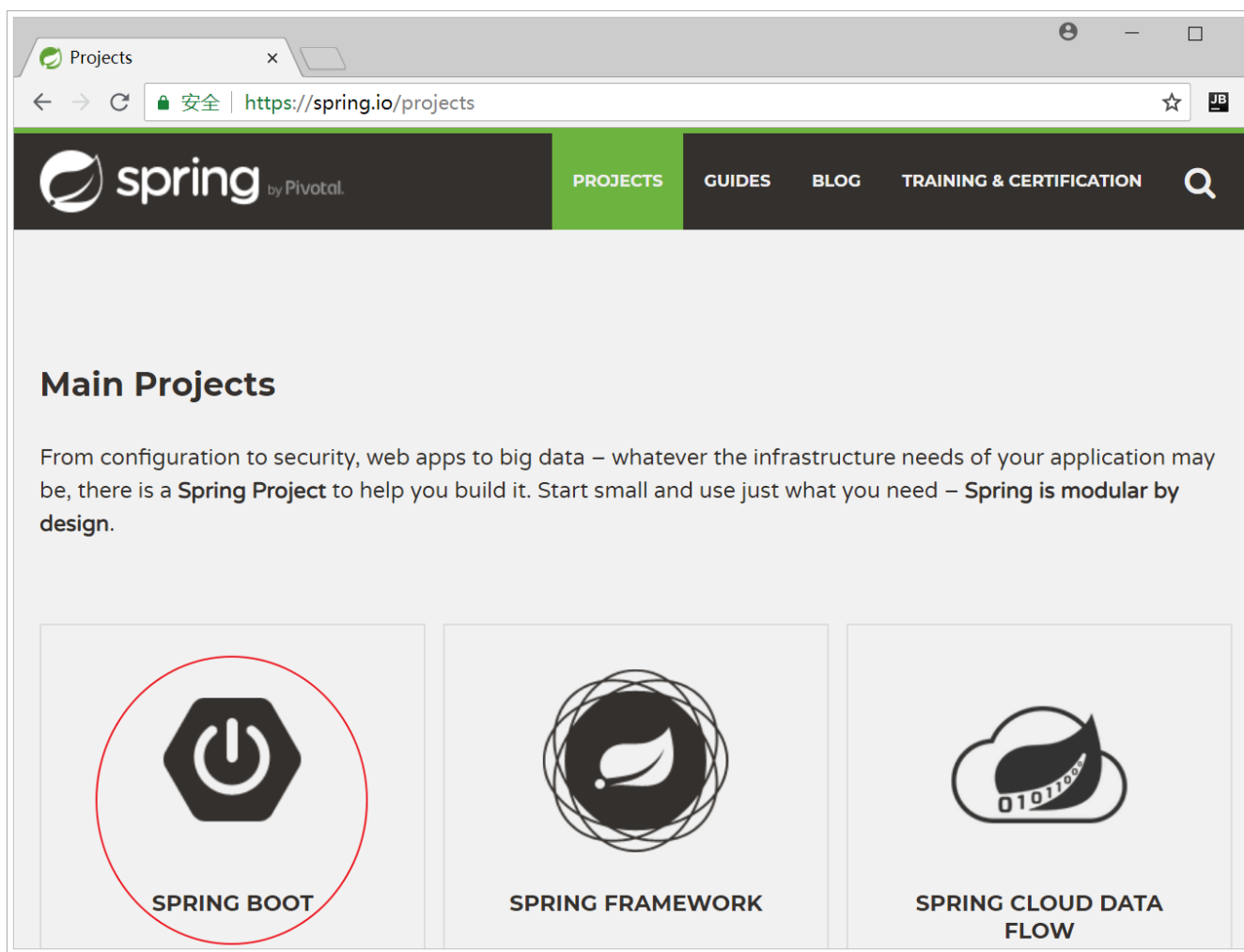
1. Spring Boot概述

在这一部分，我们主要了解以下3个问题：

- 什么是Spring Boot
- 为什么要学习Spring Boot
- Spring Boot的特点

1.1. 什么是Spring Boot

Spring Boot是Spring项目中的一个子工程，与我们所熟知的Spring-framework 同属于spring的产品：



首页Spring Boot简介可以看到下面的一段介绍：

Spring Boot is designed to get you up and running as quickly as possible, with minimal upfront configuration of Spring. Spring Boot takes an opinionated view of building production-ready applications.

翻译一下：

Spring Boot的设计目的是让您尽可能快地启动和运行，而无需预先配置Spring。Spring Boot以一种固定的方式来构建可用于生产级别的应用程序。

一般把Spring Boot称为搭建程序的 **脚手架** 或者说是**便捷搭建 基于Spring的工程 脚手架**。其最主要作用就是帮助开发人员快速的构建庞大的spring项目，并且尽可能的减少一切xml配置，做到开箱即用，迅速上手，让开发人员关注业务而非配置。

1.2. 为什么要学习Spring Boot

java一直被人诟病的一点就是臃肿、麻烦。当我们还在辛苦的搭建项目时，可能Python程序员已经把功能写好了，究其原因注意是两点：

- 复杂的配置

项目各种配置其实是开发时的损耗，因为在思考 Spring 特性配置和解决业务问题之间需要进行思维切换，所以写配置挤占了写应用程序逻辑的时间。

- 一个是混乱的依赖管理

项目的依赖管理也是件吃力不讨好的事情。决定项目里要用哪些库就已经够让人头痛的了，你还要知道这些库的哪个版本和其他库不会有冲突，这难题实在太棘手。并且，依赖管理也是一种损耗，添加依赖不是写应用程序代码。一旦选错了依赖的版本，随之而来的不兼容问题毫无疑问会是生产力杀手。

而Spring Boot让这一切成为过去！

Spring Boot 简化了基于Spring的应用开发，只需要“run”就能创建一个独立的、生产级别的Spring应用。Spring Boot为Spring平台及第三方库提供开箱即用的设置（提供默认设置，存放默认配置的包就是启动器starter），这样我们就可以简单的开始。多数Spring Boot应用只需要很少的Spring配置。

我们可以使用Spring Boot创建java应用，并使用java -jar 启动它，就能得到一个生产级别的web工程。

1.3. Spring Boot的特点

Spring Boot 主要特点是：

- 创建独立的Spring应用，为所有 Spring 的开发者提供一个非常快速的、广泛接受的入门体验
- 直接嵌入应用服务器，如tomcat、jetty、undertow等；不需要去部署war包
- 提供固定的启动器依赖去简化组件配置；实现开箱即用（启动器starter-其实就是Spring Boot提供的一个jar包），通过自己设置参数（.properties或.yml的配置文件），即可快速使用。
- 自动地配置Spring和其它有需要的第三方依赖
- 提供了一些大型项目中常见的非功能性特性，如内嵌服务器、安全、指标、健康检测、外部化配置等
- 绝对没有代码生成，也无需 XML 配置。

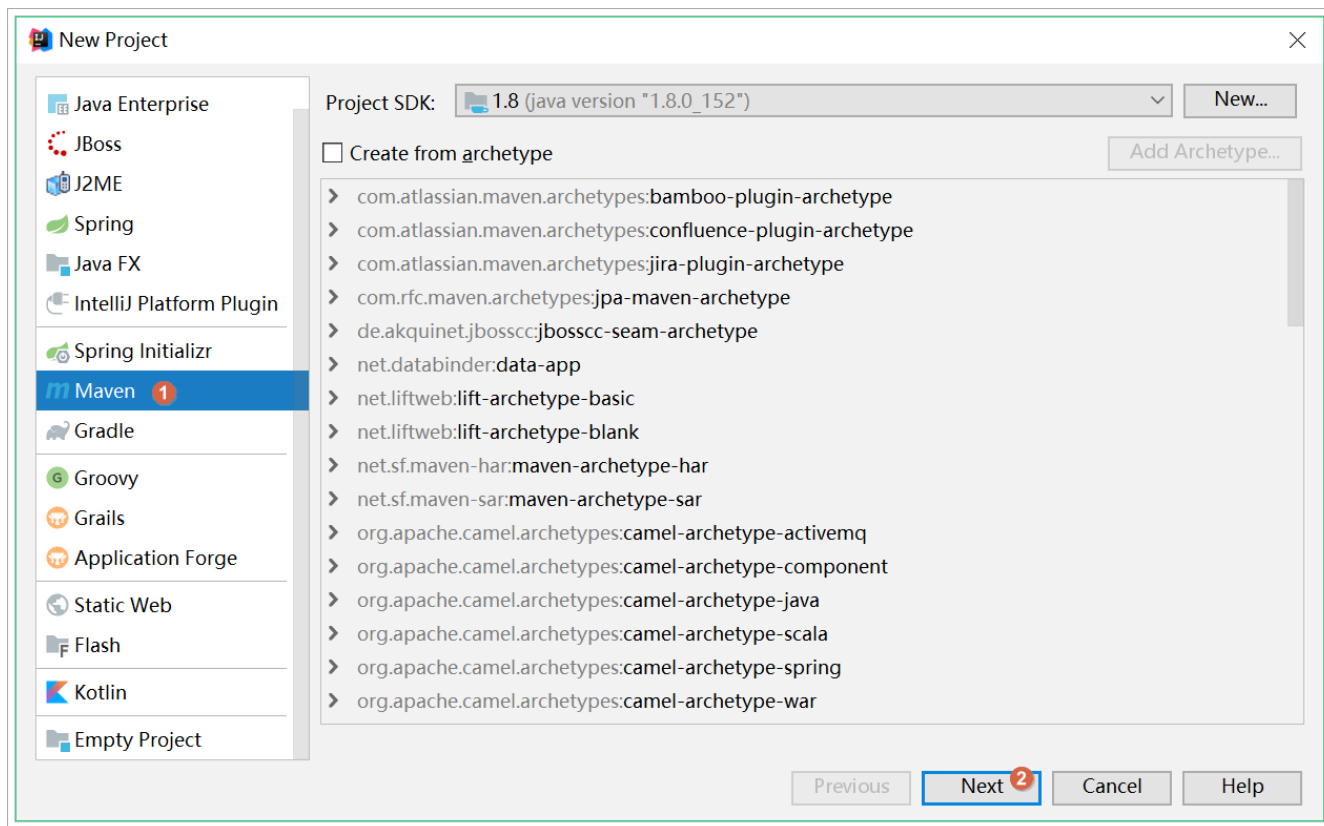
更多细节，大家可以到[官网](#)查看。

2. 快速入门

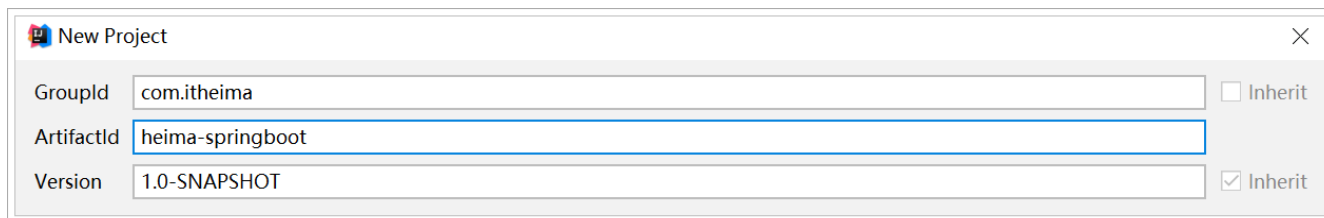
接下来，我们就来利用Spring Boot搭建一个web工程，体会一下Spring Boot的魅力所在！

2.1. 创建工程

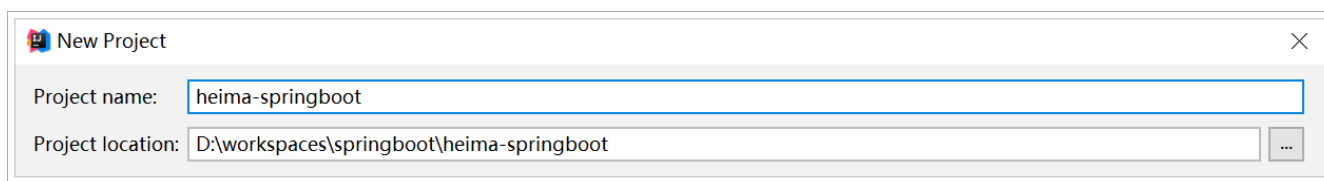
新建一个maven jar工程：



项目信息：



项目路径：



2.2.添加依赖

看到这里很多同学会有疑惑，前面说传统开发的问题之一就是依赖管理混乱，怎么这里我们还需要管理依赖呢？难道Spring Boot不帮我们管理吗？

别着急，现在我们的项目与Spring Boot还没有什么关联。Spring Boot提供了一个名为spring-boot-starter-parent的工程，里面已经对各种常用依赖（并非全部）的版本进行了管理，我们的项目需要以这个项目为父工程，这样我们就不用操心依赖的版本问题了，需要什么依赖，直接引入坐标即可！

2.2.1. 添加父工程坐标

```

1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-parent</artifactId>
4   <version>2.1.5.RELEASE</version>
5 </parent>

```

2.2.2. 添加web启动器

为了让Spring Boot帮我们完成各种自动配置，我们必须引入Spring Boot提供的自动配置依赖，我们称为启动器。因为我们是web项目，这里我们引入web启动器，在 `pom.xml` 文件中加入如下依赖：

```

1 <dependencies>
2   <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-web</artifactId>
5   </dependency>
6 </dependencies>

```

需要注意的是，我们并没有在这里指定版本信息。因为Spring Boot的父工程已经对版本进行了管理了。

这个时候，我们会发现项目中多出了大量的依赖。

那些依赖都是Spring Boot根据 `spring-boot-starter-web` 这个依赖自动引入的，而且所有的版本都已经管理好，不会出现冲突。

2.2.3. 管理jdk版本

如果我们想要修改Spring Boot项目的jdk版本，只需要简单的添加以下属性即可，如果没有需求，则不添加。同样的在 `pom.xml` 文件中添加如下：

```

1 <properties>
2   <java.version>1.8</java.version>
3 </properties>

```

2.2.4. 完整pom文件

`heima-springboot\pom.xml` 文件内容如下：

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5     http://maven.apache.org/xsd/maven-4.0.0.xsd">
6   <modelVersion>4.0.0</modelVersion>
7   <groupId>com.leyou.demo</groupId>
8   <artifactId>springboot-demo</artifactId>
9   <version>1.0-SNAPSHOT</version>
10
11   <properties>

```

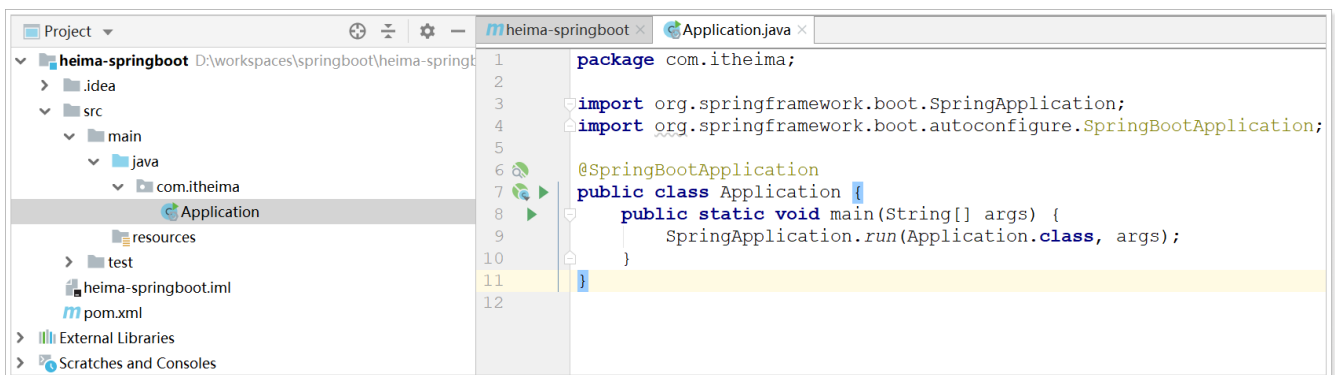
```

12     <java.version>1.8</java.version>
13 </properties>
14
15 <parent>
16     <groupId>org.springframework.boot</groupId>
17     <artifactId>spring-boot-starter-parent</artifactId>
18     <version>2.1.5.RELEASE</version>
19 </parent>
20
21 <dependencies>
22     <dependency>
23         <groupId>org.springframework.boot</groupId>
24         <artifactId>spring-boot-starter-web</artifactId>
25     </dependency>
26 </dependencies>
27 </project>

```

2.3. 启动类

Spring Boot项目通过main函数即可启动，我们需要创建一个启动类：



编写 `heima-springboot\src\main\java\com\itheima\Application.java` 如下：

```

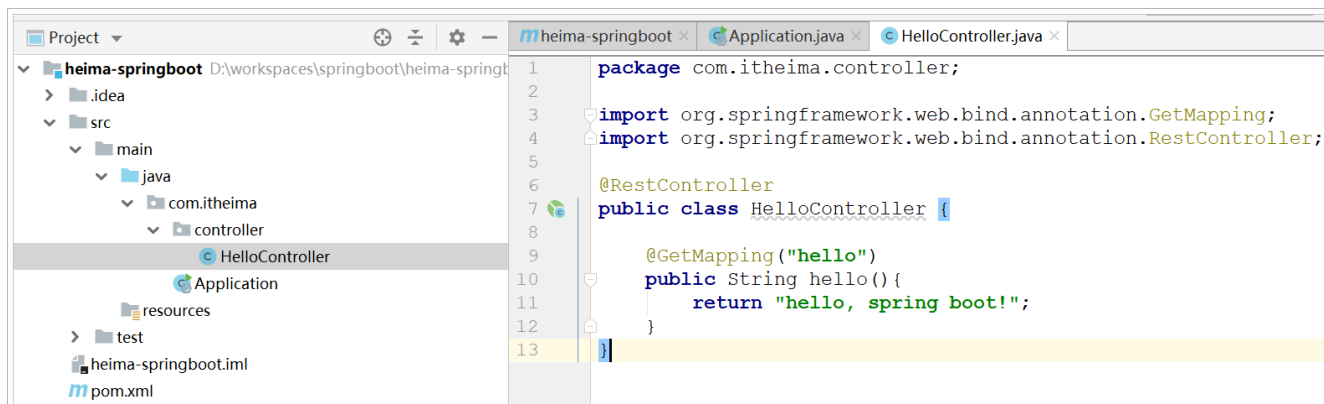
1 @SpringBootApplication
2 public class Application {
3     public static void main(String[] args) {
4         SpringApplication.run(Application.class, args);
5     }
6 }

```

2.4. 编写controller

接下来，就可以像以前那样开发SpringMVC的项目了！

编写 `heima-springboot\src\main\java\com\itheima\controller\HelloController.java`

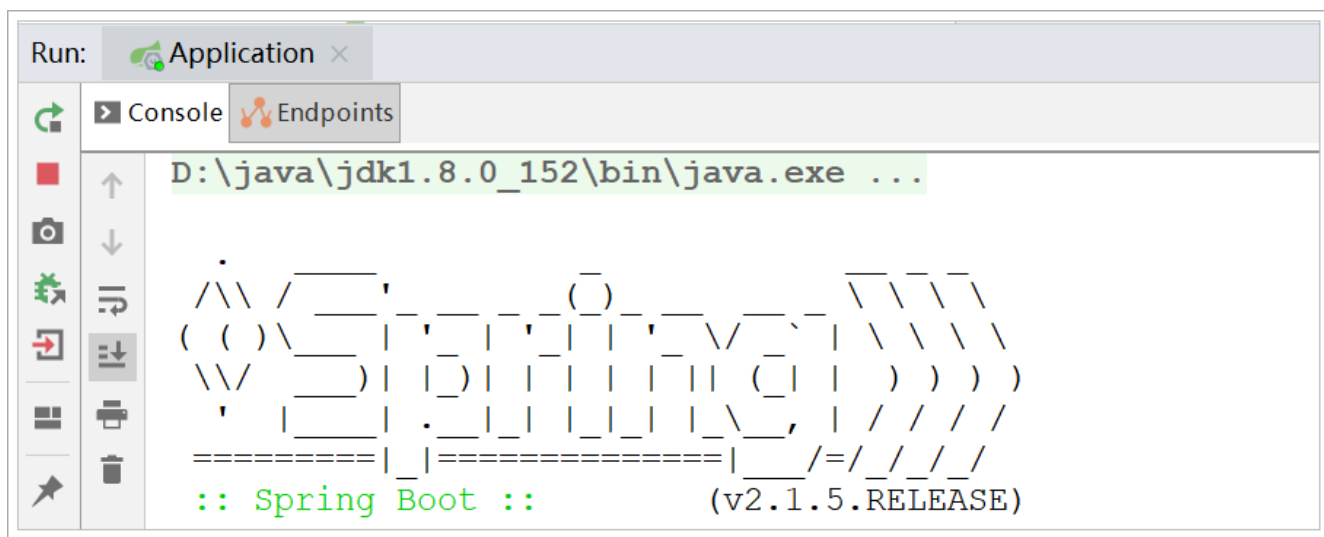


代码如下：

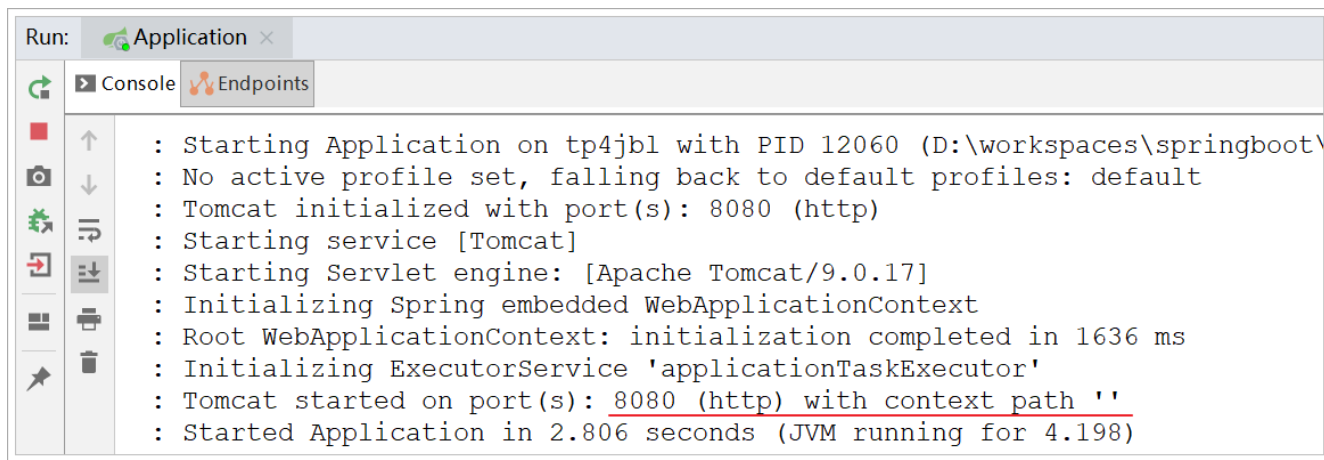
```
1 @RestController
2 public class HelloController {
3
4     @GetMapping("hello")
5     public String hello(){
6         return "hello, spring boot!";
7     }
8 }
9
```

2.5. 启动测试

接下来，运行main函数，查看控制台：

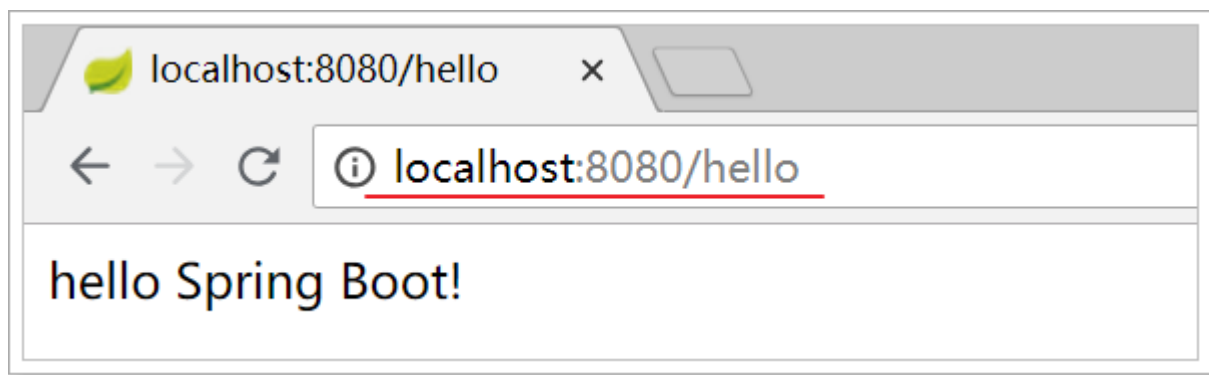


并且可以看到监听的端口信息：



- 1) 监听的端口是8080
- 2) SpringMVC的项目路径是：空
- 3) /hello 路径已经映射到了 `HelloController` 中的 `hello()` 方法

打开页面访问: <http://localhost:8080/hello>



测试成功了!

3. java配置应用

在入门案例中，我们没有任何的配置，就可以实现一个SpringMVC的项目了，快速、高效!

但是有同学会有疑问，如果没有任何的xml，那么我们如果要配置一个Bean该怎么办？比如我们要配置一个数据库连接池，以前会这么配置：

```

1 <!-- 配置连接池 -->
2 <bean id="dataSource" class="com.alibaba.druid.pool.DruidDataSource"
3     init-method="init" destroy-method="close">
4     <property name="url" value="${jdbc.url}" />
5     <property name="username" value="${jdbc.username}" />
6     <property name="password" value="${jdbc.password}" />
7 </bean>

```

现在该怎么做呢？

3.1. Spring配置历史

事实上，在Spring3.0开始，Spring官方就已经开始推荐使用java配置来代替传统的xml配置了，我们不妨来回顾一下Spring的历史：

- Spring1.0时代

在此时因为jdk1.5刚刚出来，注解开发并未盛行，因此一切Spring配置都是xml格式，想象一下所有的bean都用xml配置，细思极恐啊，心疼那个时候的程序员2秒

- Spring2.0时代

Spring引入了注解开发，但是因为并不完善，因此并未完全替代xml，此时的程序员往往是把xml与注解进行结合，貌似我们之前都是这种方式。

- Spring3.0及以后

3.0以后Spring的注解已经非常完善了，因此Spring推荐大家使用完全的java配置来代替以前的xml，不过似乎在国内并未推广盛行。然后当Spring Boot来临，人们才慢慢认识到java配置的优雅。

有句古话说的好：拥抱变化，拥抱未来。所以我们也应该顺应时代潮流，做时尚的弄潮儿，一起来学习下java配置的玩法。

3.2. 尝试java配置

java配置主要靠java类和一些注解，比较常用的注解有：

- `@Configuration`：声明一个类作为配置类，代替xml文件
- `@Bean`：声明在方法上，将方法的返回值加入Bean容器，代替 `<bean>` 标签
- `@Value`：属性注入
- `@PropertySource`：指定外部属性文件，

我们接下来用java配置来尝试实现连接池配置：

1. 在 `pom.xml` 文件中添加Druid连接池依赖如下

```
1 <dependency>
2   <groupId>com.alibaba</groupId>
3   <artifactId>druid</artifactId>
4   <version>1.1.6</version>
5 </dependency>
```

2. 使用数据操作工具创建数据库 `springboot_test`

3. 然后在项目中创建 `heima-springboot\src\main\resources\jdbc.properties` 文件，内容如下

```
1 jdbc.driverClassName=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://127.0.0.1:3306/springboot_test
3 jdbc.username=root
4 jdbc.password=root
```

4. 编写 `heima-springboot\src\main\java\com\itheima\config\JdbcConfig.java` 如下

```
1 package com.itheima.config;
```

```

2
3 import com.alibaba.druid.pool.DruidDataSource;
4 import org.springframework.beans.factory.annotation.Value;
5 import org.springframework.context.annotation.Bean;
6 import org.springframework.context.annotation.Configuration;
7 import org.springframework.context.annotation.PropertySource;
8
9 import javax.sql.DataSource;
10
11 @Configuration
12 @PropertySource("classpath:jdbc.properties")
13 public class JdbcConfig {
14     @Value("${jdbc.url}")
15     String url;
16     @Value("${jdbc.driverClassName}")
17     String driverClassName;
18     @Value("${jdbc.username}")
19     String username;
20     @Value("${jdbc.password}")
21     String password;
22
23     @Bean
24     public DataSource dataSource(){
25         DruidDataSource dataSource = new DruidDataSource();
26         dataSource.setDriverClassName(driverClassName);
27         dataSource.setUrl(url);
28         dataSource.setUsername(username);
29         dataSource.setPassword(password);
30         return dataSource;
31     }
32 }
33

```

解读:

- `@Configuration`: 声明我们 `JdbcConfig` 是一个配置类
- `@PropertySource`: 指定属性文件的路径是: `classpath:jdbc.properties`
- 通过 `@Value` 为属性注入值
- 通过 `@Bean` 将 `dataSource()` 方法声明为一个注册Bean的方法, Spring会自动调用该方法, 将方法的返回值加入Spring容器中。

然后我们就可以在任意位置通过 `@Autowired` 注入DataSource了!

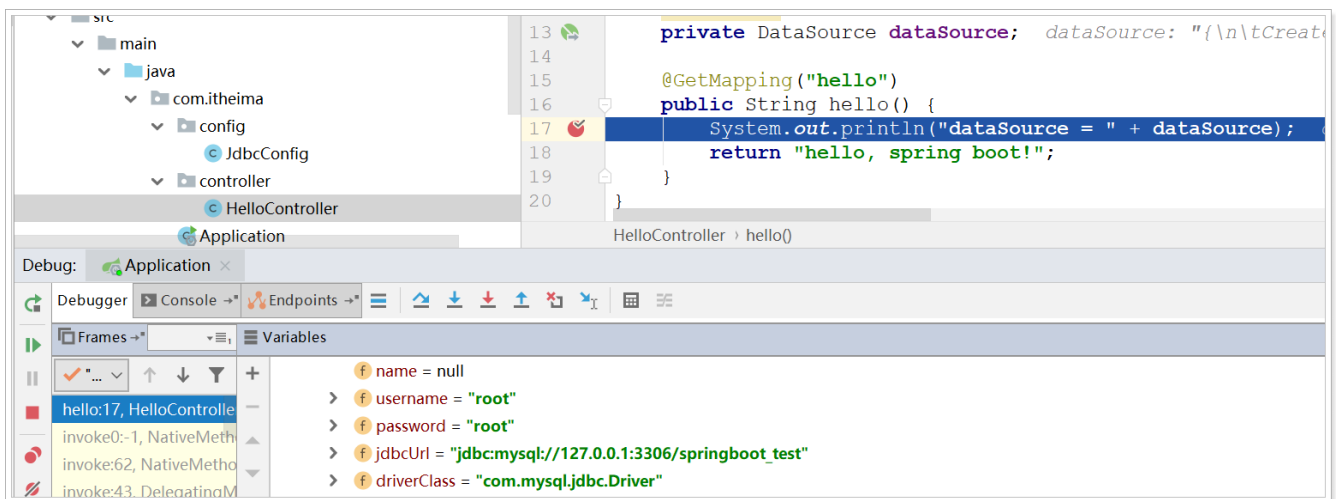
5. 在 `HelloController` 中注入DataSource进行测试, 改造代码如下:

```

1  @RestController
2  public class HelloController {
3
4      @Autowired
5      private DataSource dataSource;
6
7      @GetMapping("hello")
8      public String hello() {
9          System.out.println("dataSource = " + dataSource);
10         return "hello, spring boot!";
11     }
12 }

```

然后打断点，Debug运行并查看：



属性注入成功了！

3.3. Spring Boot的属性注入

属性文件的名称有变化，默认的文件名必须是：application.properties或application.yml

在上面的案例中，我们实验了java配置方式。不过属性注入使用的是@Value注解。这种方式虽然可行，但是不够强大，因为它只能注入基本类型值。

在Spring Boot中，提供了一种新的属性注入方式，支持各种java基本数据类型及复杂类型的注入。

1) 新建 heima-springboot\src\main\java\com\itheima\config\JdbcProperties.java，用于进行属性注入：

```

1  package com.itheima.config;
2
3  import org.springframework.boot.context.properties.ConfigurationProperties;
4
5  @ConfigurationProperties(prefix = "jdbc")
6  public class JdbcProperties {

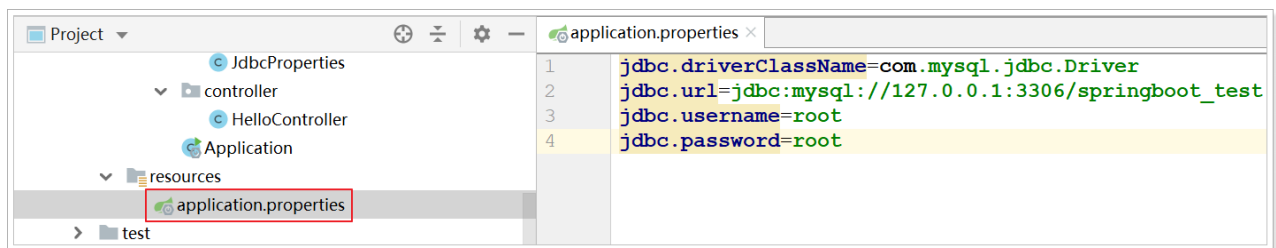
```

```

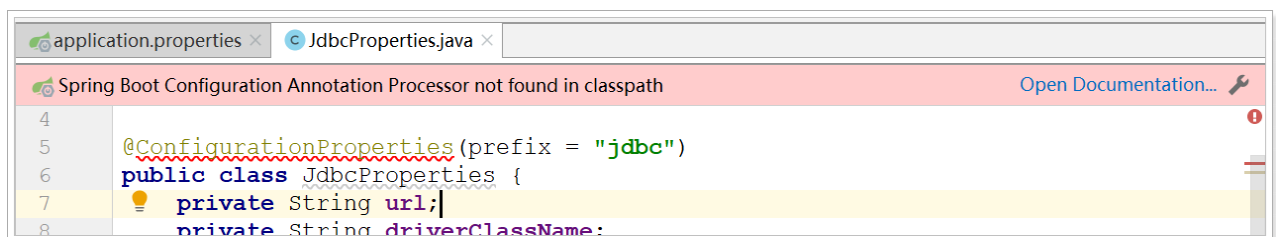
7     private String url;
8     private String driverClassName;
9     private String username;
10    private String password;
11    // ... 略
12    // getters 和 setters
13 }
14

```

- 在类上通过@ConfigurationProperties注解声明当前类为属性读取类
- prefix="jdbc" 读取属性文件中，前缀为jdbc的值。
- 在类上定义各个属性，名称必须与属性文件中 jdbc. 后面部分一致
- 需要注意的是，这里我们并没有指定属性文件的地址，所以我们需要把jdbc.properties名称改为application.properties，这是Spring Boot默认读取的属性文件名：



【注意】如果出现如下提示，项目也可以运行；



如果要去掉上述的提示，则可以在 pom.xml 文件中添加如下依赖：

```

1     <dependency>
2         <groupId> org.springframework.boot </groupId>
3         <artifactId>spring-boot-configuration-processor</artifactId>
4         <!--不传递依赖-->
5         <optional>true</optional>
6     </dependency>

```

2) 将JdbcConfig类原来全部注释掉或删除，修改为如下内容：

```

1 @Configuration
2 @EnableConfigurationProperties(JdbcProperties.class)
3 public class JdbcConfig {
4
5     @Bean

```

```

6     public DataSource dataSource(JdbcProperties jdbc) {
7         DruidDataSource dataSource = new DruidDataSource();
8         dataSource.setUrl(jdbc.getUrl());
9         dataSource.setDriverClassName(jdbc.getDriverClassName());
10        dataSource.setUsername(jdbc.getUsername());
11        dataSource.setPassword(jdbc.getPassword());
12        return dataSource;
13    }
14 }

```

- 通过 `@EnableConfigurationProperties(JdbcProperties.class)` 来声明要使用 `JdbcProperties` 这个类的对象
- 然后要使用配置的话；可以通过以下方式注入 `JdbcProperties`：
 - `@Autowired`注入

```

1 @Autowired
2 private JdbcProperties prop;

```

- 构造函数注入

```

1 private JdbcProperties prop;
2 public JdbcConfig(Jdbcproperties prop){
3     this.prop = prop;
4 }

```

- 声明有 `@Bean` 的方法参数注入

```

1 @Bean
2 public DataSource dataSource(JdbcProperties prop){
3     // ...
4 }

```

本例中，我们采用第三种方式。

3) 测试结果；与前面的测试一样的。

大家会觉得这种方式似乎更麻烦了，事实上这种方式有更强大的功能，也是Spring Boot推荐的注入方式。与 `@Value` 对比关系：

24.7.5 @ConfigurationProperties vs. @Value

The `@Value` annotation is a core container feature, and it does not provide the same features as type-safe configuration properties. The following table summarizes the features that are supported by `@ConfigurationProperties` and `@Value`:

Feature	@ConfigurationProperties	@Value
Relaxed binding	Yes	No
Meta-data support	Yes	No
SpEL evaluation	No	Yes

If you define a set of configuration keys for your own components, we recommend you group them in a POJO annotated with `@ConfigurationProperties`. You should also be aware that, since `@Value` does not support relaxed binding, it is not a good candidate if you need to provide the value by using environment variables.

优势:

- Relaxed binding: 松散绑定
 - 不严格要求属性文件中的属性名与成员变量名一致。支持驼峰，中划线，下划线等等转换，甚至支持对象引导。比如：user.friend.name：代表的是user对象中的friend属性中的name属性，显然friend也是对象。@value注解就难以完成这样的注入方式。
 - meta-data support: 元数据支持，帮助IDE生成属性提示（写开源框架会用到）。

3.4. 更优雅的注入

事实上，如果一段属性只有一个Bean需要使用，我们无需将其注入到一个类（JdbcProperties，将该类上的所有注解去掉）中。而是直接在需要的地方声明即可；再次修改 JdbcConfig 类为如下代码：

```
1  @Configuration
2  public class JdbcConfig {
3
4      @Bean
5      // 声明要注入的属性前缀，Spring Boot会自动把相关属性通过set方法注入到DataSource中
6      @ConfigurationProperties(prefix = "jdbc")
7      public DataSource dataSource() {
8          return new DruidDataSource();
9      }
10 }
```

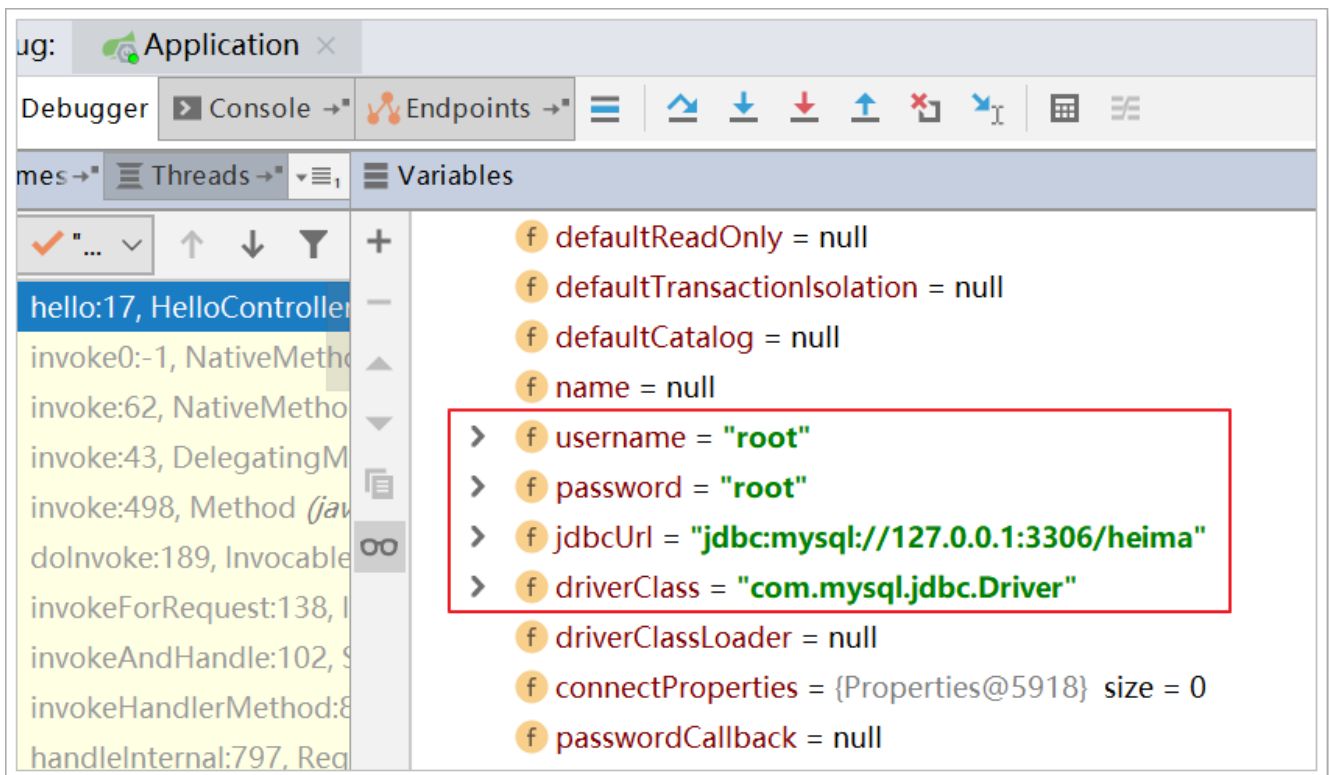
我们直接把 `@ConfigurationProperties(prefix = "jdbc")` 声明在需要使用的 `@Bean` 的方法上，然后Spring Boot就会自动调用这个Bean（此处是DataSource）的set方法，然后完成注入。使用的前提是：**该类必须有对应属性的set方法！**

```

10
11  @Configuration
12  public class JdbcConfig {
13
14      @Bean
15      @ConfigurationProperties(prefix = "jdbc")
16      public DataSource dataSource() {
17          return new DruidDataSource();
18      }
19  }

```

我们将 jdbc.properties 文件中的 jdbc.url 中的数据库名改成: /heima, 再次测试:



The screenshot shows the Spring IDE's 'Variables' tab for the 'Application' context. The 'jdbc' bean is expanded, showing its properties. The 'jdbc.url' property is highlighted with a red box, indicating the database URL configuration.

Property	Value
defaultReadOnly	null
defaultTransactionIsolation	null
defaultCatalog	null
name	null
username	"root"
password	"root"
jdbc.url	"jdbc:mysql://127.0.0.1:3306/heima"
driverClass	"com.mysql.jdbc.Driver"
driverClassLoader	null
connectProperties	{Properties@5918} size = 0
passwordCallback	null

3.5. Yaml配置文件

配置文件除了可以使用 application.properties 类型, 还可以使用后缀名为: .yaml 或者 .yml 的类型, 也就是: application.yaml 或者 application.yml

正如YAML所表示的YAML Ain't Markup Language, YAML 是一种简洁的非标记语言。YAML以数据为中心, 使用空白, 缩进, 分行组织数据, 从而使得表示更加简洁易读。

基本格式:

```
1 jdbc:
2   driverClassName: com.mysql.jdbc.Driver
3   url: jdbc:mysql://127.0.0.1:3306/springboot_test
4   username: root
5   password: root
```

把application.properties修改为application.yml进行测试。

如果两个配置文件都有，会把两个文件的配置合并，如果有重复属性，以properties中的为准。

如果是配置数组、list、set等结构的内容，那么在yml文件中格式为：

```
key:
- value1
- value2
```

3.6. 多个Yaml配置文件

当一个项目中有多个yml配置文件的时候，可以以application-*.yml命名；在application.yml中配置项目使用激活这些配置文件即可。

创建 application-abc.yml 文件如下：

```
1 itcast:
2   url: http://www.itcast.cn
```

创建 application-def.yml 文件如下：

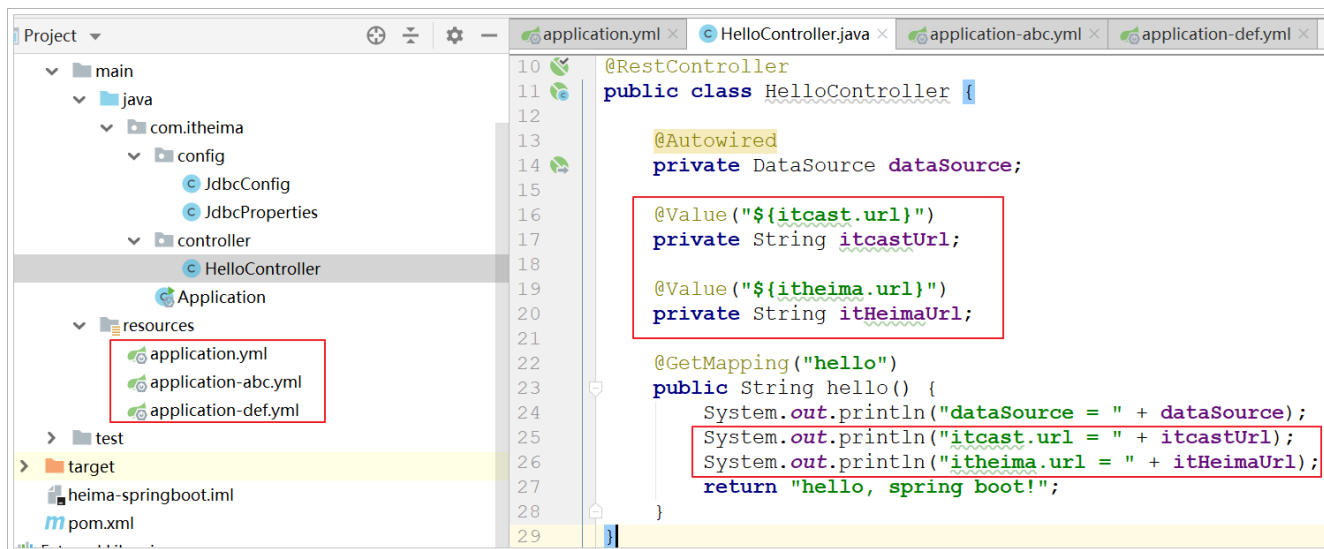
```
1 itheima:
2   url: http://www.itheima.com
```

在 application.yml 文件中添加如下配置：

```
1 #加载其它配置文件
2 spring:
3   profiles:
4     active: abc,def
```

多个文件名只需要写application-之后的名称，在多个文件之间使用,隔开。

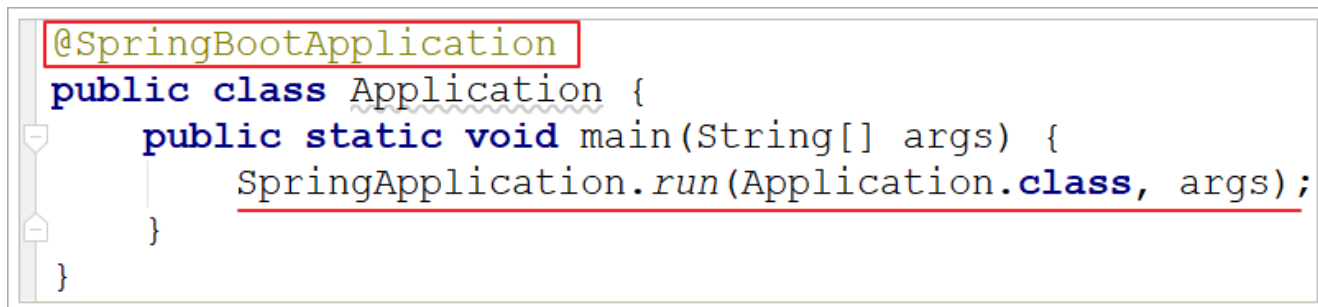
修改代码测试：



4. 自动配置原理

使用Spring Boot之后，一个整合了SpringMVC的WEB工程开发，变的无比简单，那些繁杂的配置都消失不见了，这是如何做到的？

一切魔力的开始，都是从我们的main函数来的，所以我们再次来看下启动类：



我们发现特别的地方有两个：

- 注解：@SpringBootApplication
- run方法：SpringApplication.run()

我们分别来研究这两个部分。

4.1. 了解@SpringBootApplication

点击进入，查看源码：

```

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

```

这里重点的注解有3个：

- @SpringBootConfiguration
- @EnableAutoConfiguration
- @ComponentScan

4.1.1. @SpringBootConfiguration

我们继续点击查看源码：

```

/**
 * Indicates that a class provides Spring Boot application
 * {@link Configuration @Configuration}. Can be used as an alternative to the Spring's
 * standard {@code @Configuration} annotation so that configuration can be found
 * automatically (for example in tests).
 * <p>
 * Application should only ever include <em>one</em> {@code @SpringBootConfiguration} .
 * most idiomatic Spring Boot applications will inherit it from
 * {@code @SpringBootApplication}.
 *
 * @author Phillip Webb
 * @since 1.4.0
 */
@Target (ElementType.TYPE)
@Retention (RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {
}

```

通过这段我们可以看出，在这个注解上面，又有一个@Configuration注解。通过上面的注释阅读我们知道：这个注解的作用就是声明当前类是一个配置类，然后Spring会自动扫描到添加了@Configuration的类，并且读取其中的配置信息。而@SpringBootConfiguration是来声明当前类是SpringBoot应用的配置类，项目中只能有一个。所以一般我们无需自己添加。

4.1.2. @EnableAutoConfiguration

关于这个注解，官网上有一段说明：

The second class-level annotation is `@EnableAutoConfiguration`. This annotation tells Spring Boot to “guess” how you want to configure Spring, based on the jar dependencies that you have added. Since `spring-boot-starter-web` added Tomcat and Spring MVC, the auto-configuration assumes that you are developing a web application and sets up Spring accordingly.

简单翻译以下：

第二级的注解 `@EnableAutoConfiguration`，告诉Spring Boot基于你所添加的依赖，去“猜测”你想要如何配置Spring。比如我们引入了 `spring-boot-starter-web`，而这个启动器中帮我们添加了 `tomcat`、`SpringMVC` 的依赖。此时自动配置就知道你是要开发一个web应用，所以就帮你完成了web及SpringMVC的默认配置了！

总结，Spring Boot内部对大量的第三方库或Spring内部库进行了默认配置，这些配置是否生效，取决于我们是否引入了对应库所需的依赖，如果有那么默认配置就会生效。

所以，我们使用SpringBoot构建一个项目，只需要引入所需框架的依赖，配置就可以交给SpringBoot处理了。除非你不希望使用SpringBoot的默认配置，它也提供了自定义配置的入口。

4.1.3. @ComponentScan

我们跟进源码：

```
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
@Documented
@Repeatable(ComponentScans.class)
public @interface ComponentScan {
```

并没有看到什么特殊的地方。我们查看注释：

```
/**
 * Configures component scanning directives for use with {@link Configuration} classes.
 * Provides support parallel with Spring XML's {@code <context:component-scan>} element.
 *
 * <p>Either {@link #basePackageClasses} or {@link #basePackages} (or its alias
 * {@link #value}) may be specified to define specific packages to scan. If specific
 * packages are not defined, scanning will occur from the package of the
 * class that declares this annotation.
```

大概的意思：

配置组件扫描的指令。提供了类似与 `<context:component-scan>` 标签的作用

通过 `basePackageClasses` 或者 `basePackages` 属性来指定要扫描的包。如果没有指定这些属性，那么将从声明这个注解的类所在的包开始，扫描包及子包

而我们的 `@SpringBootApplication` 注解声明的类就是 `main` 函数所在的启动类，因此扫描的包是该类所在包及其子包。因此，一般启动类会放在一个比较前的包目录中。

4.2. 默认配置原理

4.2.1. spring.factories

在SpringApplication类构建的时候，有这样一段初始化代码：

```
dbcConfig.java × Application.java × SpringApplication.java × @SpringBootApplication.java × @ComponentScan.java × @SpringBo

//unchecked, rawtypes/
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {
    this.resourceLoader = resourceLoader;
    Assert.notNull(primarySources, message: "PrimarySources must not be null");
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));
    this.webApplicationType = WebApplicationType.deduceFromClasspath();
    setInitializers((Collection) getSpringFactoriesInstances(
        ApplicationContextInitializer.class));
    setListeners((Collection) getSpringFactoriesInstances(ApplicationListener.class));
    this.mainApplicationClass = deduceMainApplicationClass();
}
```

跟进去：

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

private <T> Collection<T> getSpringFactoriesInstances(Class<T> type,
    Class<?>[] parameterTypes, Object... args) {
    ClassLoader classLoader = getClassLoader();
    // Use names and ensure unique to protect against duplicates
    Set<String> names = new LinkedHashSet<>{
        SpringFactoriesLoader.loadFactoryNames(type, classLoader)};
    List<T> instances = createSpringFactoriesInstances(type, parameterTypes,
        classLoader, args, names);
    AnnotationAwareOrderComparator.sort(instances);
    return instances;
}
```

这里发现会通过loadFactoryNames尝试加载一些FactoryName，然后利用createSpringFactoriesInstances将这些加载到的类名进行实例化。

继续跟进loadFactoryNames方法：

```
on.java × SpringFactoriesLoader.java × @SpringBootApplication.java × @ComponentScan.java × @SpringBootConfiguration.java × Hell

*/
public static List<String> loadFactoryNames(Class<?> factoryClass, @Nullable ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    return loadSpringFactories(classLoader).getOrDefault(factoryClassName, Collections.emptyList());
}

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

    try {
        Enumeration<URL> urls = (classLoader != null ? 通过类加载器加载文件，读取内容
            : classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        while (urls.hasMoreElements()) {
            for (String fileName : urls.nextElement().getFileList()) {
                if (!fileName.endsWith(".properties")) continue;
                result.add(factoryClassName, loadFactoryNames(factoryClass, classLoader, fileName));
            }
        }
    } catch (IOException ex) {
        logger.warn("Unable to load factory names: " + ex.getMessage());
    }
    return result;
}
```

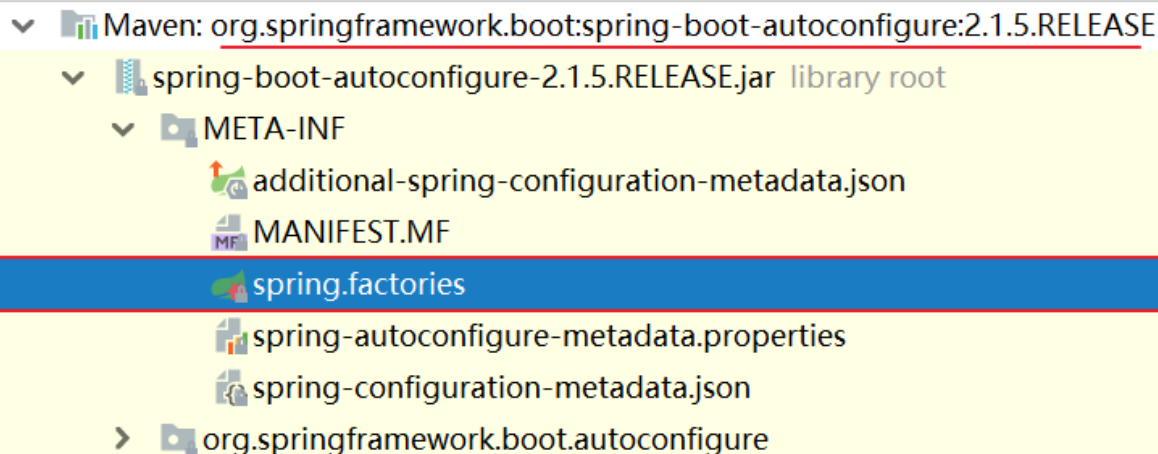
发现此处会利用类加载器加载某个文件：FACTORIES_RESOURCE_LOCATION，然后解析其内容。我们找到这个变量的声明：

```
public final class SpringFactoriesLoader {

    /**
     * The location to look for factories.
     * <p>Can be present in multiple JAR files.
     */
    public static final String FACTORIES_RESOURCE_LOCATION = "META-INF/spring.factories";
}
```

可以发现，其地址是：META-INF/spring.factories，我们知道，ClassLoader默认是从classpath下读取文件，因此，SpringBoot会在初始化的时候，加载所有classpath:META-INF/spring.factories文件，包括jar包当中的。

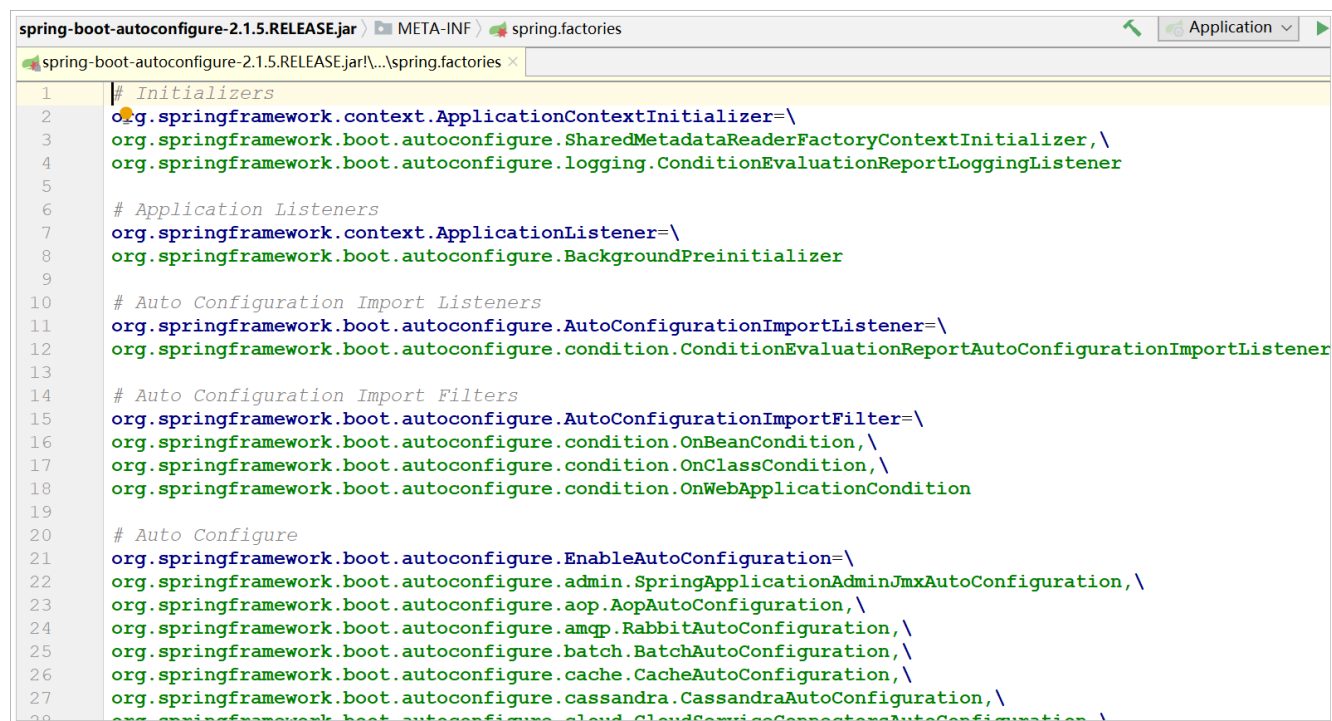
而在Spring的一个依赖包：spring-boot-autoconfigure中，就有这样的文件：



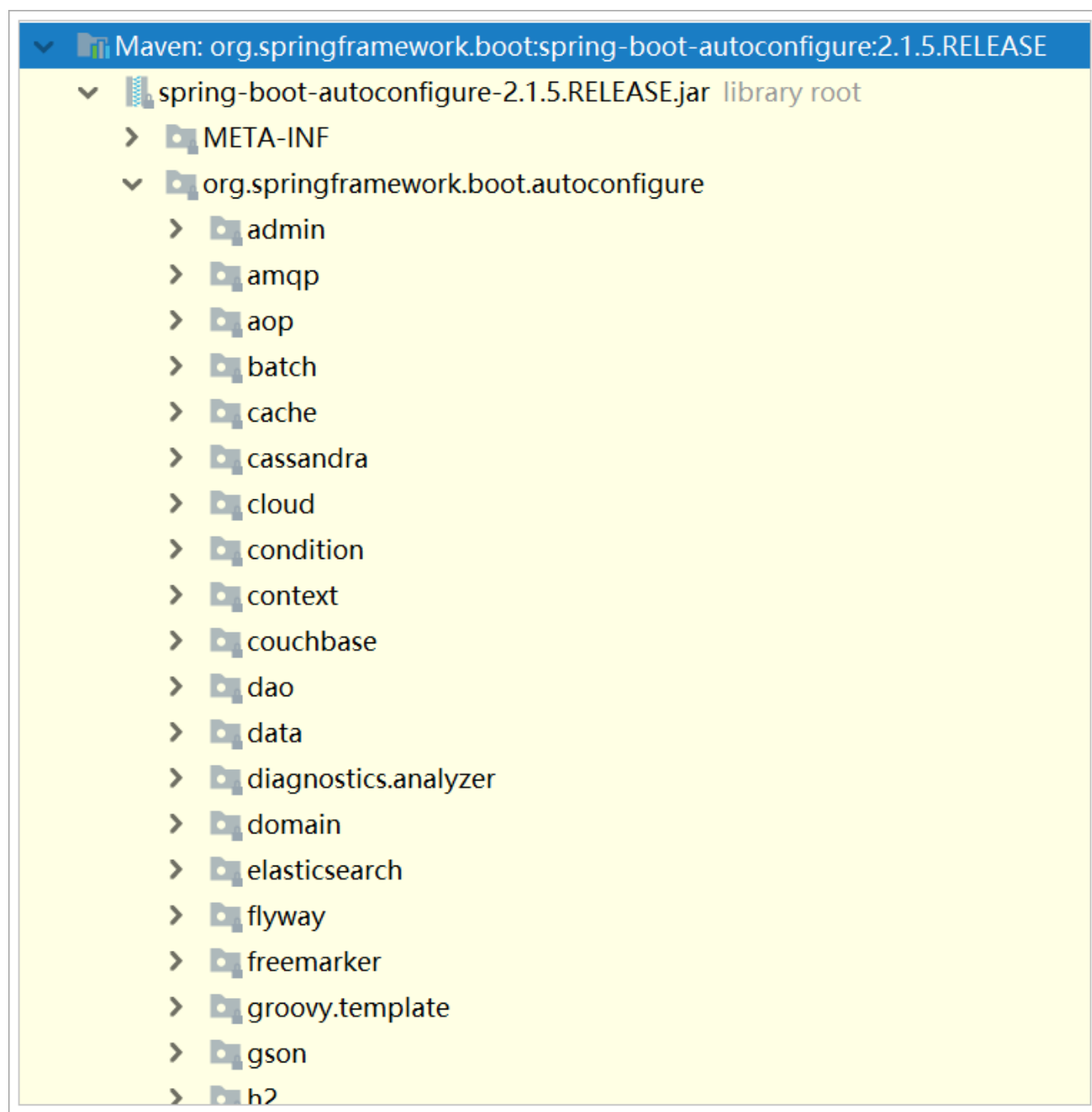
以后我们引入的任何第三方启动器，只要实现自动配置，也都会有类似文件。

4.2.1. 默认配置类

我们打开刚才的spring.factories文件：



可以发现以EnableAutoConfiguration接口为key的一系列配置，key所对应的值，就是所有的自动配置类，可以在当前的jar包中找到这些自动配置类：

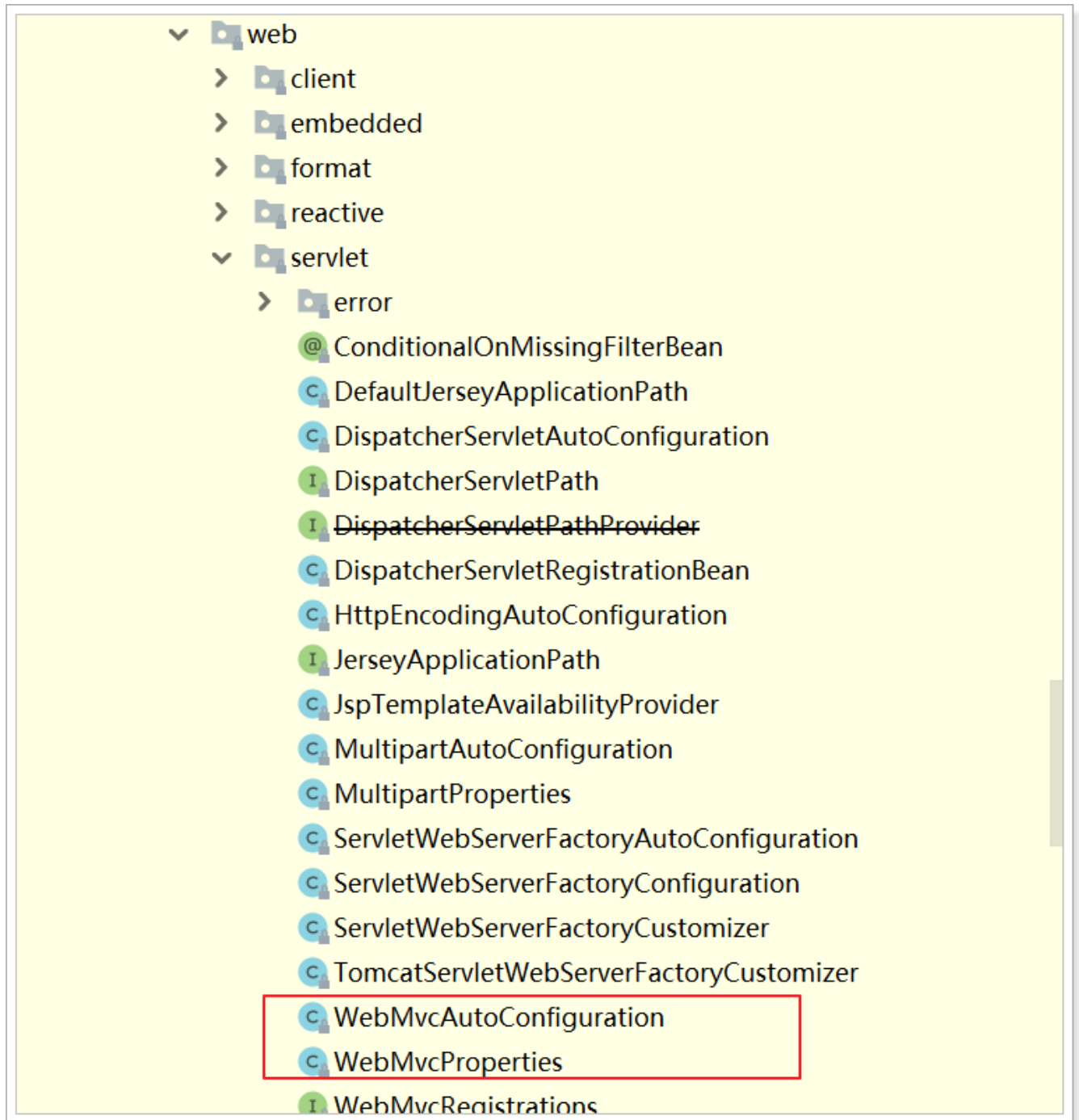


非常多，几乎涵盖了现在主流的开源框架，例如：

- redis
- jms
- amqp
- jdbc
- jackson
- mongodb
- jpa
- solr
- elasticsearch

... 等等

我们来看一个我们熟悉的，例如SpringMVC，查看mvc 的自动配置类：



打开WebMvcAutoConfiguration：


```
gFactoriesLoader.java × spring.factories × WebMvcAutoConfiguration.java × @SpringBootApplication.java × @ComponentSc
* @author Bruce Brouwer
* @author Artsiom Yudovin
*/
@Configuration
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,
    TaskExecutionAutoConfiguration.class, ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

    public static final String DEFAULT_PREFIX = "";

    public static final String DEFAULT_SUFFIX = "";

    private static final String[] SERVLET_LOCATIONS = { "/" };
}
```

我们看到这个类上的4个注解：

- `@Configuration`：声明这个类是一个配置类
- `@ConditionalOnWebApplication(type = Type.SERVLET)`
ConditionalOn，翻译就是在某个条件下，此处就是满足项目的类是Type.SERVLET类型，也就是一个普通web工程，显然我们就是
- `@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, webMvcConfigurer.class })`
这里的条件是OnClass，也就是满足以下类存在：Servlet、DispatcherServlet、WebMvcConfigurer，其中Servlet只要引入了tomcat依赖自然会有，后两个需要引入SpringMVC才会有。这里就是判断你是否引入了相关依赖，引入依赖后该条件成立，当前类的配置才会生效！
- `@ConditionalOnMissingBean(webMvcConfigurationSupport.class)`
这个条件与上面不同，OnMissingBean，是说环境中没有指定的Bean这个才生效。其实这就是自定义配置的入口，也就是说，如果我们自己配置了一个WebMVCConfigurationSupport的类，那么这个默认配置就会失效！

接着，我们查看该类中定义了什么：

视图解析器：


```
oader.java × spring.factories × WebMvcAutoConfiguration.java × @SpringBootApplication.java × @ComponentSc

@Bean
@ConditionalOnMissingBean
public InternalResourceViewResolver defaultViewResolver() {
    InternalResourceViewResolver resolver = new InternalResourceViewResolver();
    resolver.setPrefix(this.mvcProperties.getView().getPrefix());
    resolver.setSuffix(this.mvcProperties.getView().getSuffix());
    return resolver;
}

@Bean
@ConditionalOnBean(View.class)
@ConditionalOnMissingBean
public BeanNameViewResolver beanNameViewResolver() {
    BeanNameViewResolver resolver = new BeanNameViewResolver();
    resolver.setOrder(Ordered.LOWEST_PRECEDENCE - 10);
    return resolver;
}

@Bean
```

处理器适配器 (HandlerAdapter) :

```
riesLoader.java × spring.factories × WebMvcAutoConfiguration.java × @SpringBootApplication.java × @ComponentS

}

@Bean
@Override
public RequestMappingHandlerAdapter requestMappingHandlerAdapter() {
    RequestMappingHandlerAdapter adapter = super.requestMappingHandlerAdapter();
    adapter.setIgnoreDefaultModelOnRedirect(this.mvcProperties == null
        || this.mvcProperties.isIgnoreDefaultModelOnRedirect());
    return adapter;
}

@Override
protected RequestMappingHandlerAdapter createRequestMappingHandlerAdapter() {
    if (this.mvcRegistrations != null
        && this.mvcRegistrations.getRequestMappingHandlerAdapter() != null) {
        return this.mvcRegistrations.getRequestMappingHandlerAdapter();
    }
    return super.createRequestMappingHandlerAdapter();
}

@Bean
```

还有很多，这里就不一一截图了。

4.2.2. 默认配置属性

另外，这些默认配置的属性来自哪里呢？

```
gFactoriesLoader.java × spring.factories × WebMvcAutoConfiguration.java × @SpringBootApplication.java × @Component
ConditionalProperty(prefix = "spring.mvc.formcontentfilter.enabled", name = "enabled",
    matchIfMissing = true)
public OrderedFormContentFilter formContentFilter() { return new OrderedFormContentFi

// Defined as a nested config to ensure WebMvcConfigurer is not read when not
// on the classpath
@Configuration
@Import(EnableWebMvcConfiguration.class)
@EnableConfigurationProperties({ WebMvcProperties.class, ResourceProperties.class })
@Order(0)
public static class WebMvcAutoConfigurationAdapter
    implements WebMvcConfigurer, ResourceLoaderAware {

    private static final Log logger = LoggerFactory.getLog(WebMvcConfigurer.class);
```

我们看到，这里通过@EnableAutoConfiguration注解引入了两个属性：WebMvcProperties和ResourceProperties。这不正是SpringBoot的属性注入玩法嘛。

我们查看这两个属性类：

```
SpringFactoriesLoader.java × spring.factories × WebMvcAutoConfiguration.java × WebMvcProperties.java ×
}

public static class View {

    /**
     * Spring MVC view prefix.
     */
    private String prefix;

    /**
     * Spring MVC view suffix.
     */
    private String suffix;
```

找到了内部资源视图解析器的prefix和suffix属性。

ResourceProperties中主要定义了静态资源（.js,.html,.css等）的路径：

```
SpringFactoriesLoader.java × spring.factories × WebMvcAutoConfiguration.java × ResourceProperties.java × W
* @author Kristine Jetzke
* @since 1.1.0
*/
@ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false)
public class ResourceProperties {

    private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {
        "classpath:/META-INF/resources/", "classpath:/resources/",
        "classpath:/static/", "classpath:/public/" };

    /**
     * Locations of static resources. Defaults to classpath:[/META-INF/resources/,
     * /resources/, /static/, /public/].
     */
    private String[] staticLocations = CLASSPATH_RESOURCE_LOCATIONS;
```

如果我们要覆盖这些默认属性，只需要在application.properties中定义与其前缀prefix和字段名一致的属性即可。

4.3. 总结

SpringBoot为我们提供了默认配置，而默认配置生效的步骤：

- @EnableAutoConfiguration注解会去寻找 META-INF/spring.factories 文件，读取其中以 EnableAutoConfiguration 为key的所有类的名称，这些类就是提前写好的自动配置类
- 这些类都声明了 @Configuration 注解，并且通过 @Bean 注解提前配置了我们所需要的一切实例
- 但是，这些配置不一定生效，因为有 @ConditionalOn 注解，满足一定条件才会生效。比如条件之一：是一些相关的类要存在
- 类要存在，我们只需要引入了相关依赖（启动器），依赖有了条件成立，自动配置生效。
- 如果我们自己配置了相关Bean，那么会覆盖默认的自动配置的Bean
- 我们还可以通过配置application.yml文件，来覆盖自动配置中的属性

1) 启动器

所以，我们如果不想配置，只需要引入依赖即可，而依赖版本我们也不用操心，因为只要引入了SpringBoot提供的starter（启动器），就会自动管理依赖及版本了。

因此，玩SpringBoot的第一件事情，就是找启动器，SpringBoot提供了大量的默认启动器

2) 全局配置

另外，SpringBoot的默认配置，都会读取默认属性，而这些属性可以通过自定义 application.properties 文件来进行覆盖。这样虽然使用的还是默认配置，但是配置中的值改成了我们自定义的。

因此，玩SpringBoot的第二件事情，就是通过 application.properties 来覆盖默认属性值，形成自定义配置。我们需要知道SpringBoot的默认属性key，非常多，可以再idea中自动提示

属性文件支持两种格式，application.properties和application.yml

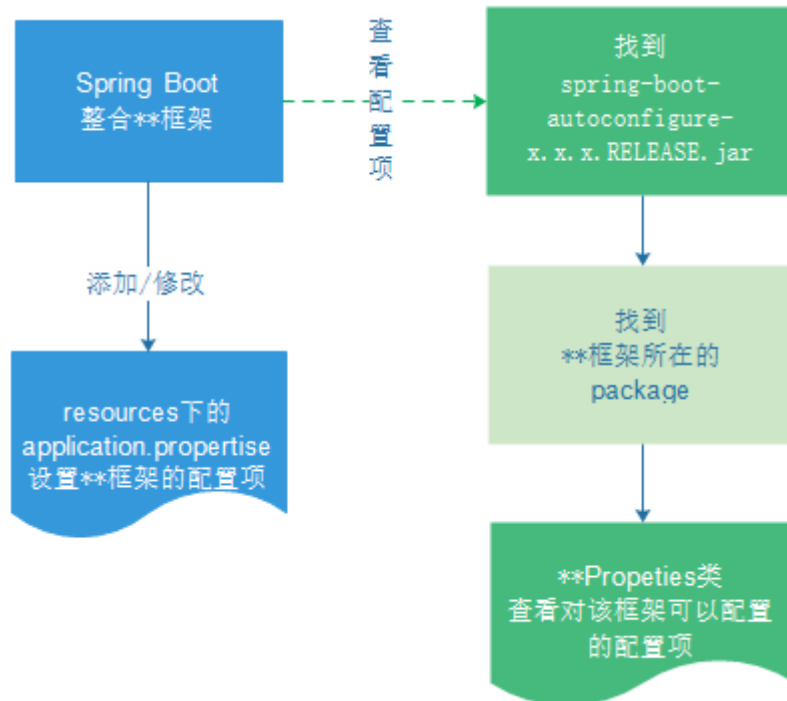
yml的语法实例：

```

1 jdbc:
2   driverClassName: com.mysql.jdbc.Driver
3   url: jdbc:mysql://127.0.0.1:3306/springboot_test
4   username: root
5   password: root
6
7 server:
8   port: 80

```

如果properties和yml文件都存在，如果有重叠属性，默认以Properties优先。遇到需要修改的组件的配置项流程为：



5. Spring Boot实践

接下来，我们来看看如何用SpringBoot来整合SSM，在数据库中引入一张用户表tb_user和实体类User。

tb_user表：详见 资料\tb_user.sql 文件，将该文件导入数据库中。

创建 heima-springboot\src\main\java\com\itheima\pojo\User.java 如下：

```

1 package com.itheima.pojo;
2
3 public class User{
4     // id
5     private Long id;
6
7     // 用户名
8     private String userName;

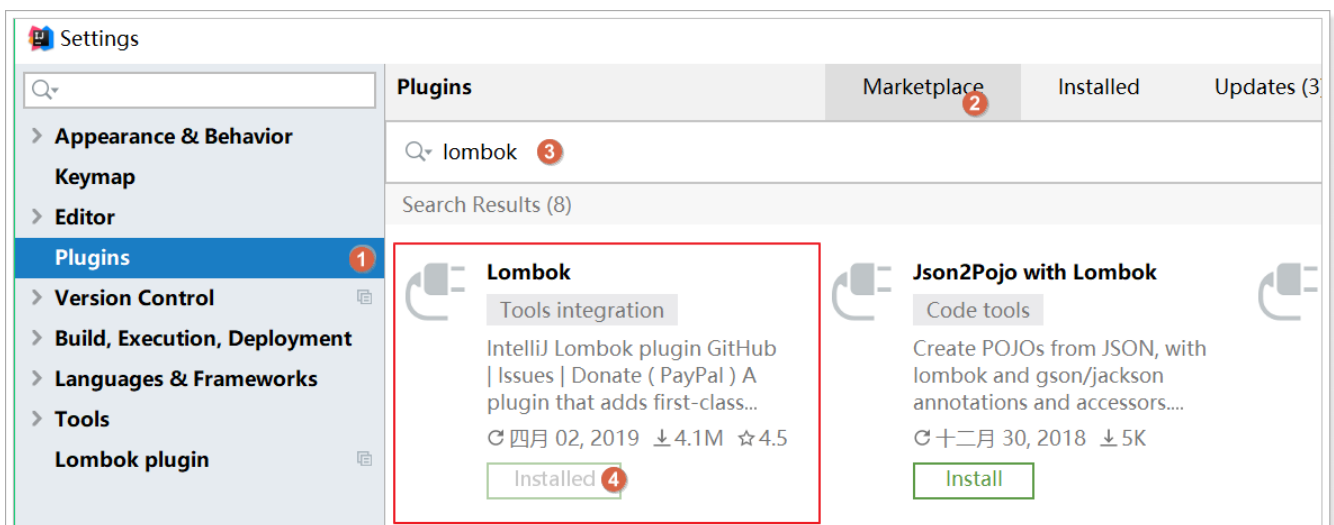
```

```
9
10 // 密码
11 private String password;
12
13 // 姓名
14 private String name;
15
16 // 年龄
17 private Integer age;
18
19 // 性别, 1男性, 2女性
20 private Integer sex;
21
22 // 出生日期
23 private Date birthday;
24
25 // 创建时间
26 private Date created;
27
28 // 更新时间
29 private Date updated;
30
31 // 备注
32 private String note;
33
34 // getter和setter省略
35 }
```

5.1. Lombok

我们编写pojo时，经常需要编写构造函数和getter、setter方法，属性多的时候，就非常浪费时间，使用lombok插件可以解决这个问题：

在IDEA中安装lombok插件；不安装插件在IDEA中使用lombok的注解虽然编译能通过，但是源码会报错。所以为了让IDEA更好的辨别lombok注解则才安装插件。



需要在maven工程中的 `pom.xml` 文件引入依赖：

```
1 <dependency>
2   <groupId>org.projectlombok</groupId>
3   <artifactId>lombok</artifactId>
4 </dependency>
```

然后可以在Bean上使用：

@Data：自动提供getter和setter、hashCode、equals、toString等方法

@Getter：自动提供getter方法

@Setter：自动提供setter方法

@Slf4j：自动在bean中提供log变量，其实用的是slf4j的日志功能。

例如：在javabean上加@Data，那么就可以省去getter和setter等方法的编写，lombok插件会自动生成。

```
application.yml × HelloController.java × User.java × ml
1 package com.itheima.pojo;
2
3 import lombok.Data;
4
5 import java.util.Date;
6
7 @Data
8 public class User{
9     // id
10    private Long id;
11
12    // 用户名
13    private String userName;
14
15    // 密码
16    private String password;
17
18    // 姓名
19    private String name;
20
```

5.2. 整合SpringMVC

虽然默认配置已经可以使用SpringMVC了，不过我们有时候需要进行自定义配置。

可以在 `application.yml` 文件中配置日志级别控制：

```
1 logging:
2   level:
3     com.itheima: debug
4     org.springframework: info
```

5.2.1. 修改端口

查看SpringBoot的全局属性可知，端口通过以下方式配置：

修改 `application.yml` 配置文件，添加如下配置：

```
1 # 映射端口
2 server:
3   port: 80
```

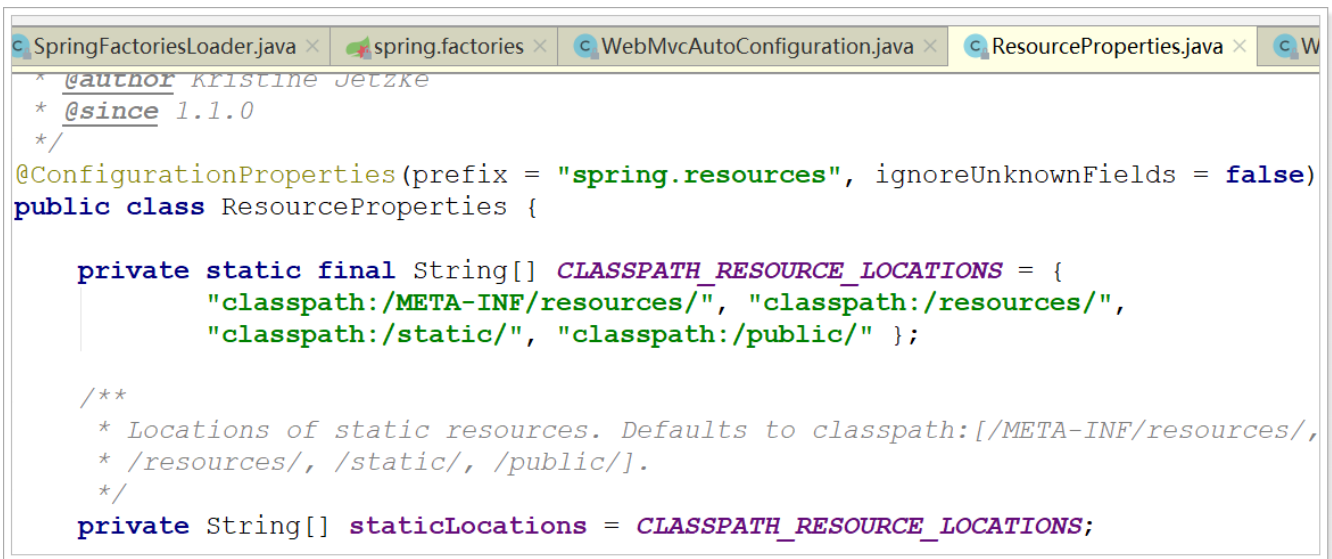
重启服务后测试：



5.2.2. 访问静态资源

现在，我们的项目是一个jar工程，那么就没有webapp，我们的静态资源该放哪里呢？

回顾我们在上面看的源码，有一个叫做ResourceProperties的类，里面就定义了静态资源的默认查找路径：

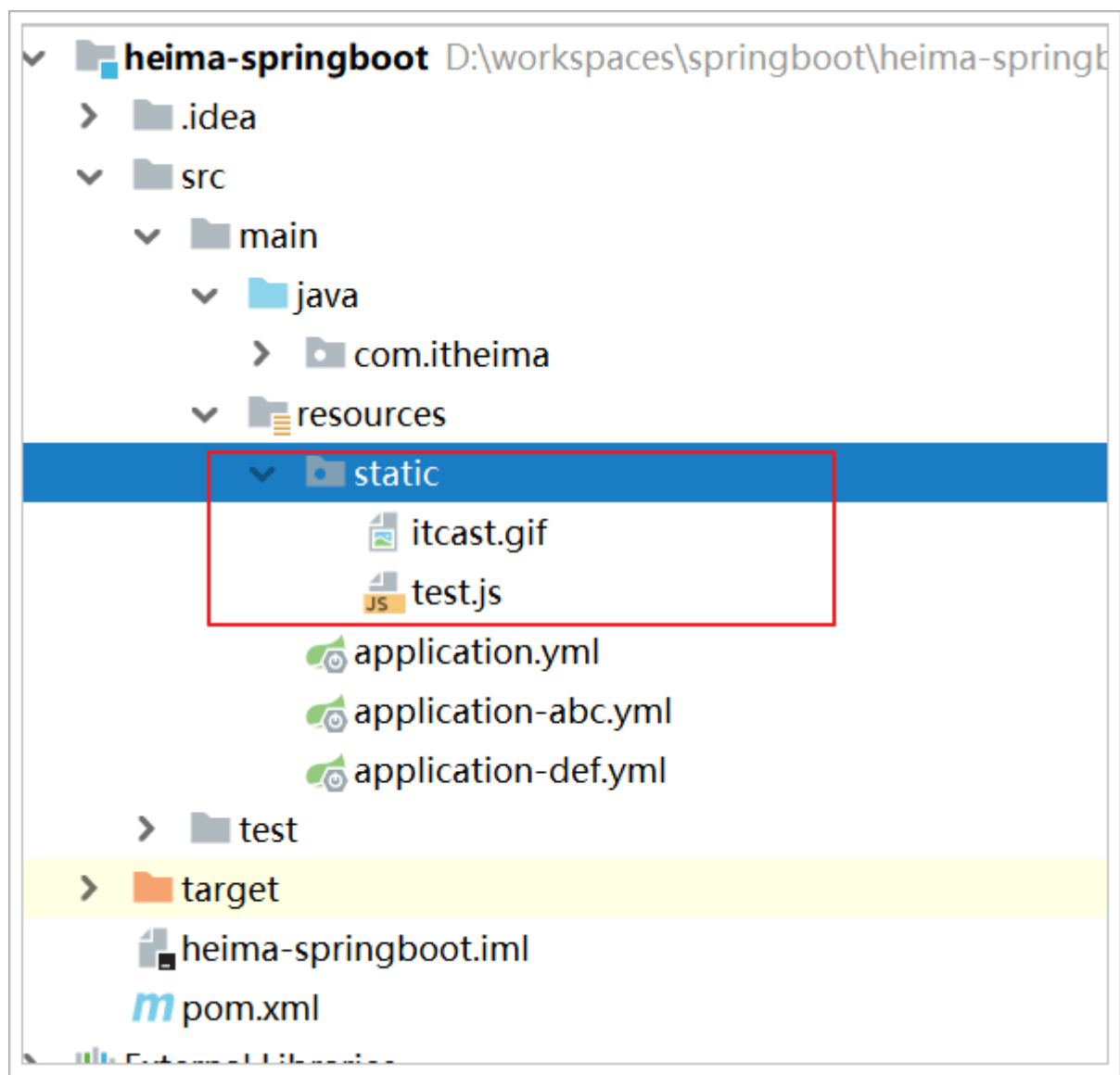


默认的静态资源路径为：

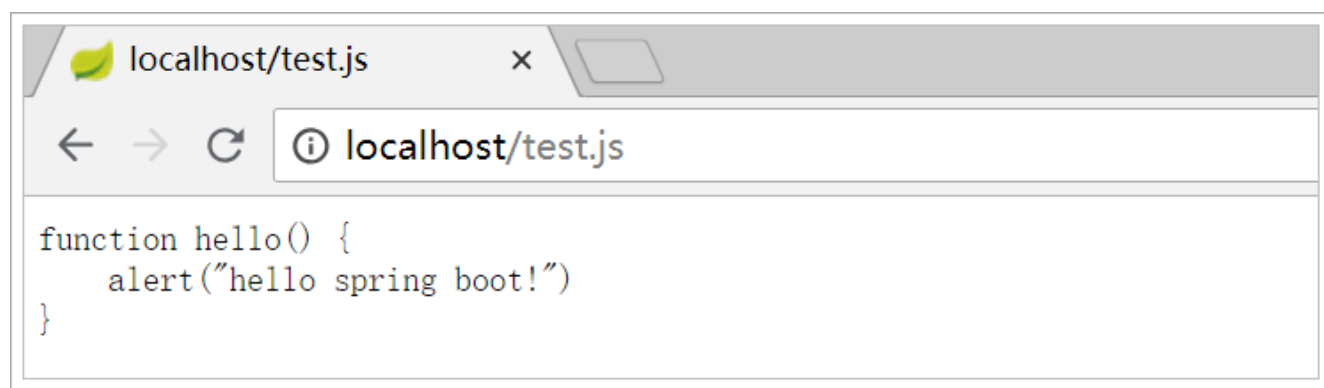
- `classpath:/META-INF/resources/`
- `classpath:/resources/`
- `classpath:/static/`
- `classpath:/public`

只要静态资源放在这些目录中任何一个，SpringMVC都会帮我们处理。

我们习惯会把静态资源放在 `classpath:/static/` 目录下。我们创建目录 `static`，并且从 `资料` 文件夹中复制 `itcast.gif` 和 `test.js` 如下：



重启项目后测试：



注意：如果访问图片时候没有显示；可以先将项目先clean再启动，或者创建 public、resources 文件夹，然后图片放置到public或resources中。

5.2.3. 添加拦截器

拦截器也是我们经常需要使用的，在SpringBoot中该如何配置呢？

拦截器不是一个普通属性，而是一个类，所以就要用到java配置方式了。在SpringBoot官方文档中有这么一段说明：

If you want to keep Spring Boot MVC features and you want to add additional [MVC configuration](#) (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`. If you wish to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, you can declare a `WebMvcRegistrationsAdapter` instance to provide such components.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`.

翻译：

如果你想要保持Spring Boot 的一些默认MVC特征，同时又想自定义一些MVC配置（包括：拦截器，格式化器，视图控制器、消息转换器 等等），你应该让一个类实现 `WebMvcConfigurer`，并且添加 `@Configuration` 注解，但是**千万不要加** `@EnableWebMvc` 注解。如果你想要自定义 `HandlerMapping`、`HandlerAdapter`、`ExceptionHandler` 等组件，你可以创建一个 `WebMvcRegistrationsAdapter` 实例 来提供以上组件。

如果你想要完全自定义SpringMVC，不保留SpringBoot提供的一切特征，你可以自己定义类并且添加 `@Configuration` 注解和 `@EnableWebMvc` 注解

总结：通过实现 `WebMvcConfigurer` 并添加 `@Configuration` 注解来实现自定义部分SpringMvc配置。

1. 创建 `heima-springboot\src\main\java\com\itheima\interceptor\MyInterceptor.java` 拦截器，内容如下：

```
1 package com.itheima.interceptor;
2
3 import lombok.extern.slf4j.Slf4j;
4 import org.springframework.web.servlet.HandlerInterceptor;
5 import org.springframework.web.servlet.ModelAndView;
6
7 import javax.servlet.http.HttpServletRequest;
8 import javax.servlet.http.HttpServletResponse;
9
10 @Slf4j
11 public class MyInterceptor implements HandlerInterceptor {
12
13     @Override
14     public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
15     Object handler) throws Exception {
16         log.debug("这是MyInterceptor拦截器的preHandle方法");
17         return true;
18     }
19
20     @Override
21     public void postHandle(HttpServletRequest request, HttpServletResponse response,
22     Object handler, ModelAndView modelAndView) throws Exception {
23         log.debug("这是MyInterceptor拦截器的postHandle方法");
24     }
25 }
```

```

24     @Override
25     public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
26         log.debug("这是MyInterceptor拦截器的afterCompletion方法");
27     }
28
29 }
30

```

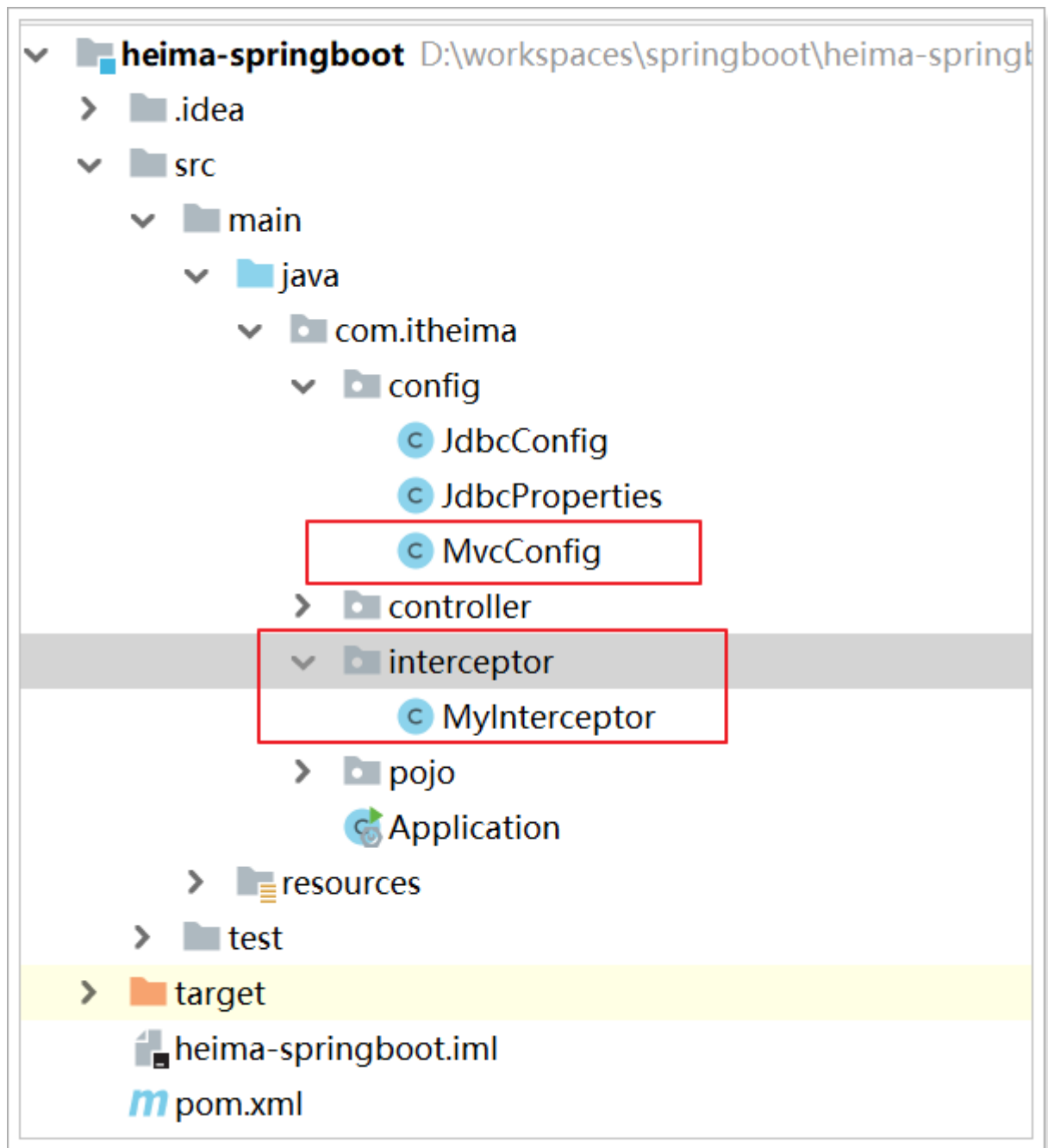
2. 定义配置类 `heima-springboot\src\main\java\com\itheima\config\MvcConfig.java`，用于注册拦截器，内容如下：

```

1  package com.itheima.config;
2
3  import com.itheima.interceptor.MyInterceptor;
4  import org.springframework.context.annotation.Bean;
5  import org.springframework.context.annotation.Configuration;
6  import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
7  import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
8
9  @Configuration
10 public class MvcConfig implements WebMvcConfigurer {
11
12     /**
13      * 将拦截器注册到spring ioc容器
14      * @return myInterceptor
15      */
16     @Bean
17     public MyInterceptor myInterceptor(){
18         return new MyInterceptor();
19     }
20
21     /**
22      * 重写该方法；往拦截器链添加自定义拦截器
23      * @param registry 拦截器链
24      */
25     @Override
26     public void addInterceptors(InterceptorRegistry registry) {
27         //通过registry添加myInterceptor拦截器，并设置拦截器路径为 /*
28         registry.addInterceptor(myInterceptor()).addPathPatterns("/*");
29     }
30 }
31

```

结构如下：



接下来访问<http://localhost/hello> 并查看日志：

```
INFO 10400 --- [main] t.m.m.autoconfigure.MapperCacheDisabler : Clear tk.mybatis.mapper.version.VersionUtil
INFO 10400 --- [main] t.m.m.autoconfigure.MapperCacheDisabler : Clear EntityHelper entityTableMap cache.
INFO 10400 --- [main] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 80 (http) with
INFO 10400 --- [main] com.itheima.Application : Started Application in 3.378 seconds (JVM
INFO 10400 --- [p-nio-80-exec-1] o.a.c.c.C.[Tomcat].[localhost].[/] : Initializing Spring DispatcherServlet 'dis
INFO 10400 --- [p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
INFO 10400 --- [p-nio-80-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 7 ms
DEBUG 10400 --- [p-nio-80-exec-1] com.itheima.interceptor.MyInterceptor : 这是MyInterceptor拦截器的preHandle方法
DEBUG 10400 --- [p-nio-80-exec-1] com.itheima.interceptor.MyInterceptor : 这是MyInterceptor拦截器的postHandle方法
DEBUG 10400 --- [p-nio-80-exec-1] com.itheima.interceptor.MyInterceptor : 这是MyInterceptor拦截器的afterCompletion方法
```

5.3. 整合jdbc和事务

spring中的jdbc连接和事务是配置中的重要一环，在SpringBoot中该如何处理呢？

答案是不需要处理，我们只要找到SpringBoot提供的启动器即可，在 `pom.xml` 文件中添加如下依赖：

```
1 <dependency>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-starter-jdbc</artifactId>
4 </dependency>
```

当然，不要忘了数据库驱动，SpringBoot并不知道我们用的什么数据库，这里我们选择MySQL；同样的在 `pom.xml` 文件中添加如下依赖：

```
1 <dependency>
2   <groupId>mysql</groupId>
3   <artifactId>mysql-connector-java</artifactId>
4   <version>5.1.46</version>
5 </dependency>
```

至于事务，SpringBoot中通过注解来控制。就是我们熟知的 `@Transactional` 使用的时候设置在对应的类或方法上即可。

创建 `heima-springboot\src\main\java\com\itheima\service\UserService.java` 业务类如下：

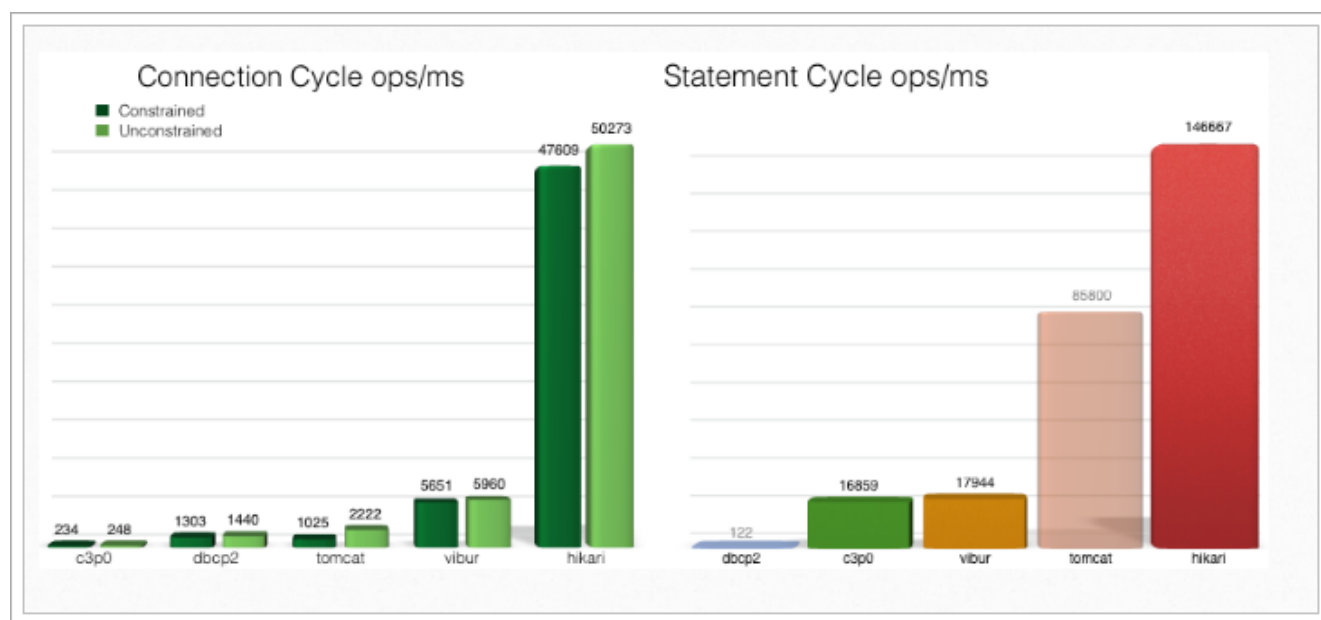
```
1 package com.itheima.service;
2
3 import com.itheima.pojo.User;
4 import org.springframework.stereotype.Service;
5 import org.springframework.transaction.annotation.Transactional;
6
7 @Service
8 public class UserService {
9
10     public User queryById(Long id){
11         //根据id查询
12         return new User();
13     }
14
15     @Transactional
16     public void saveUser(User user){
17         System.out.println("新增用户...");
18     }
19 }
20
```

5.4. 整合连接池

其实，在刚才引入jdbc启动器的时候，SpringBoot已经自动帮我们引入了一个连接池：

```
Dependencies
├── org.springframework.boot:spring-boot-starter-web:2.1.5.RELEASE
│   ├── com.alibaba:druid:1.1.6
│   ├── org.springframework.boot:spring-boot-configuration-processor:2.1.5.RELEASE
│   ├── org.projectlombok:lombok:1.18.8
│   └── org.springframework.boot:spring-boot-starter-jdbc:2.1.5.RELEASE
│       ├── org.springframework.boot:spring-boot-starter:2.1.5.RELEASE (omitted)
│       └── com.zaxxer:HikariCP:3.2.0
│           ├── org.springframework:spring-jdbc:5.1.7.RELEASE
│           ├── mysql:mysql-connector-java:5.1.46
│           ├── org.mybatis.spring.boot:mybatis-spring-boot-starter:2.0.1
│           ├── tk.mybatis:mapper-spring-boot-starter:2.1.5
│           ├── org.springframework.boot:spring-boot-starter-test:2.1.5.RELEASE
│           └── org.springframework.boot:spring-boot-starter-data-redis:2.1.5.RELEASE
```

HikariCP应该是目前速度最快的连接池了，我们看看它与c3p0的对比：



因此，我们只需要指定连接池参数即可；打开 application.yml 添加修改配置如下：

```
1 spring:
2   datasource:
3     driver-class-name: com.mysql.jdbc.Driver
4     url: jdbc:mysql://localhost:3306/springboot_test
5     username: root
6     password: root
```

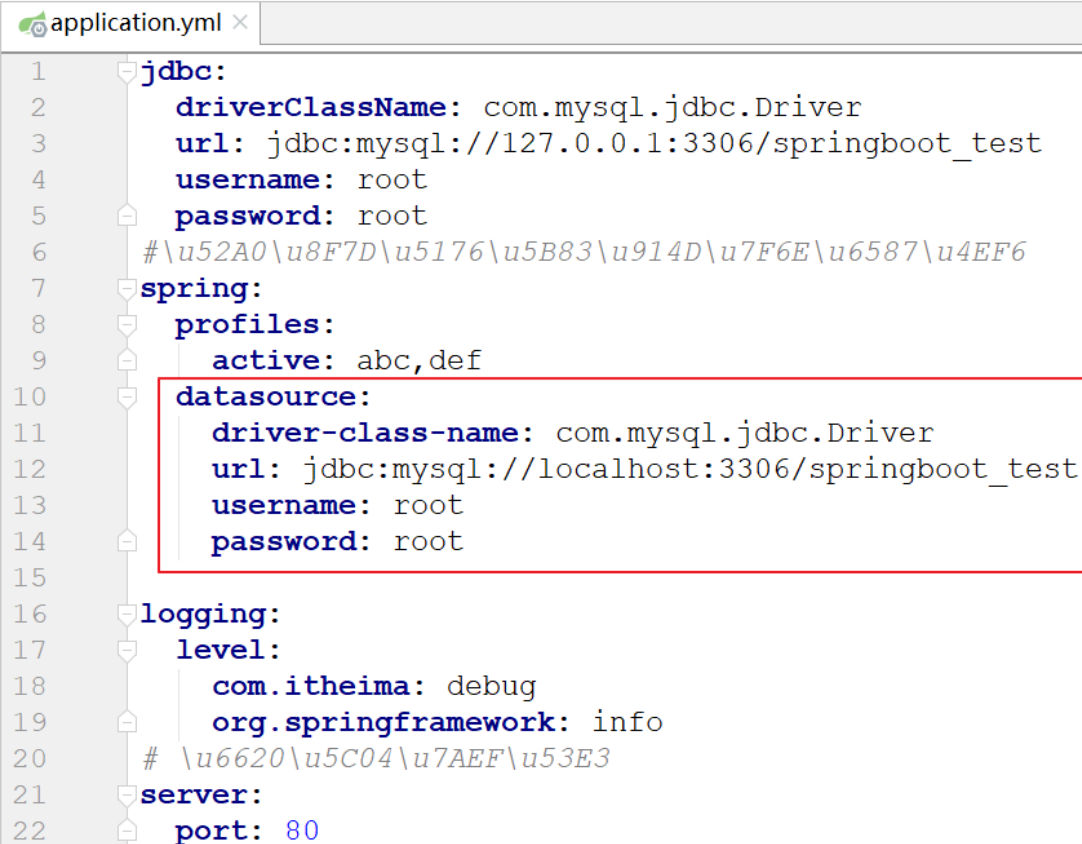
【注意】

把 JdbcConfig 类中的druid的配置删除或注释;

```
//@Configuration
public class JdbcConfig {

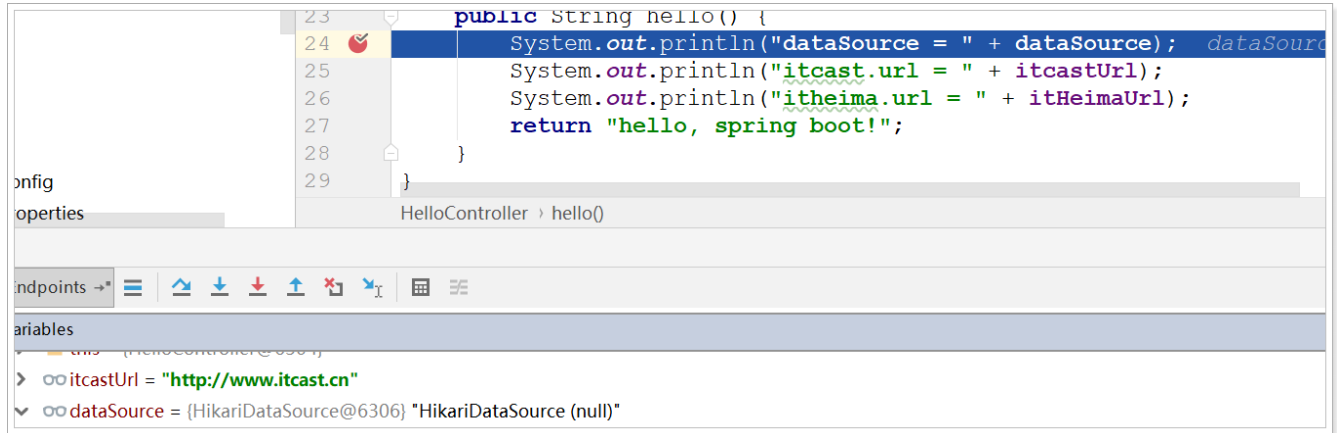
    /*@Bean
    @ConfigurationProperties(prefix = "jdbc")
    public DataSource dataSource() {
        return new DruidDataSource();
    }*/
}
```

在配置完hikari数据库连接池后的 application.yml 文件如下:



```
application.yml x
1 jdbc:
2   driverClassName: com.mysql.jdbc.Driver
3   url: jdbc:mysql://127.0.0.1:3306/springboot_test
4   username: root
5   password: root
6   #\u52A0\u8F7D\u5176\u5B83\u914D\u7F6E\u6587\u4EF6
7   spring:
8     profiles:
9       active: abc,def
10    datasource:
11      driver-class-name: com.mysql.jdbc.Driver
12      url: jdbc:mysql://localhost:3306/springboot_test
13      username: root
14      password: root
15
16  logging:
17    level:
18      com.itheima: debug
19      org.springframework: info
20    # \u6620\u5C04\u7AEF\u53E3
21  server:
22    port: 80
```

启动项目，访问 <http://localhost/hello>；查看后台输出，一样可以在HelloController中获取到datasource。



5.5. 整合mybatis

5.5.1. mybatis

1. SpringBoot官方并没有提供Mybatis的启动器，不过Mybatis[官网](#)自己实现了。在项目的 `pom.xml` 文件中加入如下依赖：

```
1 <!--mybatis -->
2 <dependency>
3     <groupId>org.mybatis.spring.boot</groupId>
4     <artifactId>mybatis-spring-boot-starter</artifactId>
5     <version>2.0.1</version>
6 </dependency>
```

2. 配置 `application.yml`，常用配置如下：

```
1 # mybatis配置
2 mybatis:
3     # 实体类别名包路径
4     type-aliases-package: com.itheima.pojo
5     # 映射文件路径
6     # mapper-locations: classpath:mappers/*.xml
7     configuration:
8         # 控制台输出执行sql
9         log-impl: org.apache.ibatis.logging.stdout.StdOutImpl
10
```

3. 配置Mapper扫描

需要注意，这里没有配置mapper接口扫描包，因此我们需要给每一个Mapper接口添加 `@Mapper` 注解，才能被识别。


```

1 @Mapper
2 public interface UserMapper {
3 }

```

或者，我们也可以不加注解，而是在启动类上添加扫描包注解(推荐)：

```

1 @SpringBootApplication
2 @MapperScan("com.itheima.mapper")
3 public class Application {
4     public static void main(String[] args) {
5         // 启动代码
6         SpringApplication.run(Application.class, args);
7     }
8 }

```

以下代码示例中，我们将采用@MapperScan扫描方式进行。

5.5.2. 通用mapper

1. 通用Mapper的作者也为自己的插件编写了启动器，我们直接引入即可。在项目的 `pom.xml` 文件中加入如下依赖：

```

1 <!-- 通用mapper -->
2 <dependency>
3     <groupId>tk.mybatis</groupId>
4     <artifactId>mapper-spring-boot-starter</artifactId>
5     <version>2.1.5</version>
6 </dependency>

```

注意：一旦引入了通用Mapper的启动器，会覆盖Mybatis官方启动器的功能，因此**需要移除对官方Mybatis启动器的依赖**。

2. 编写UserMapper

无需任何配置就可以使用了。如果有特殊需要，可以到通用mapper官网查看：<https://github.com/abel533/Mapper/wiki/3.config>

编写 `heima-springboot\src\main\java\com\itheima\mapper\UserMapper.java` 如下：

```

1 package com.itheima.mapper;
2
3 import com.itheima.pojo.User;
4 import tk.mybatis.mapper.common.Mapper;
5
6 public interface UserMapper extends Mapper<User> {
7 }
8

```

3. 把启动类上的@MapperScan注解修改为**通用mapper中自带的**：

```
Application.java x
1 package com.itheima;
2
3 import org.springframework.boot.SpringApplication;
4 import org.springframework.boot.autoconfigure.SpringBootApplication;
5 import tk.mybatis.spring.annotation.MapperScan; 导入的包是 tk 开头的
6
7 @SpringBootApplication
8 @MapperScan("com.itheima.mapper")
9 public class Application {
10     public static void main(String[] args) { SpringApplication.run(Application.class, args); }
13 }
```

4. 在User实体类上加JPA注解

修改 heima-springboot\src\main\java\com\itheima\pojo\User.java 如下:

```
1 @Data
2 @Table(name = "tb_user")
3 public class User{
4     // id
5     @Id
6     //开启主键自动回填
7     @KeySql(useGeneratedKeys = true)
8     private Long id;
9
10    // 用户名
11    private String userName;
12
13    // 密码
14    private String password;
15
16    // 姓名
17    private String name;
18
19    // 年龄
20    private Integer age;
21
22    // 性别, 1男性, 2女性
23    private Integer sex;
24
25    // 出生日期
26    private Date birthday;
27
28    // 创建时间
29    private Date created;
30
31    // 更新时间
32    private Date updated;
33
34    // 备注
35    private String note;
36 }
```

5. 对 `UserService` 的代码进行简单改造

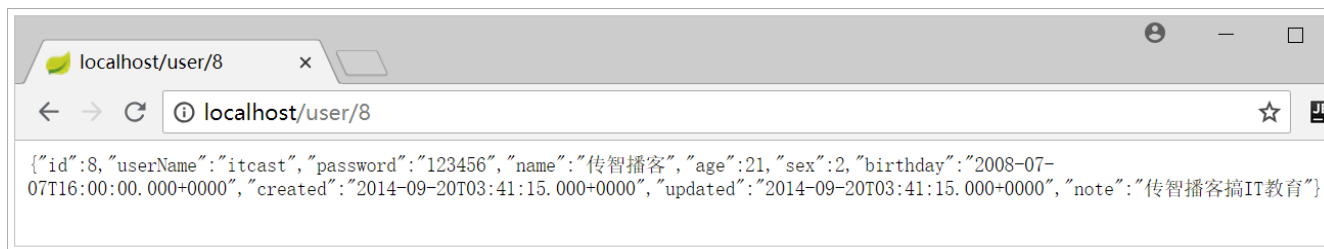
```
1  @Service
2  public class UserService {
3
4      @Autowired
5      private UserMapper userMapper;
6
7      public User queryById(Long id){
8          //根据id查询
9          return userMapper.selectByPrimaryKey(id);
10     }
11
12     @Transactional
13     public void saveUser(User user){
14         System.out.println("新增用户...");
15         userMapper.insertSelective(user);
16     }
17 }
18
```

5.6. 启动测试

将 `HelloController` 进行简单改造：

```
1  @RestController
2  public class HelloController {
3
4      @Autowired
5      private UserService userService;
6
7      /**
8       * 根据id获取用户
9       * @param id 用户id
10      * @return 用户
11      */
12      @GetMapping("/user/{id}")
13      public User queryById(@PathVariable Long id){
14          return userService.queryById(id);
15      }
16
17  }
18
```

我们启动项目，查看：



5.7. Junit测试

1. 在springboot项目中如果要使用Junit进行单元测试，则需要添加如下的依赖：

```
1      <dependency>
2          <groupId>org.springframework.boot</groupId>
3          <artifactId>spring-boot-starter-test</artifactId>
4      </dependency>
5
```

2. 在测试包下编写测试类

在测试类上面必须要添加 `@SpringBootTest` 注解。

编写测试类 heima-springboot\src\test\java\com\itheima\service\UserServiceTest.java

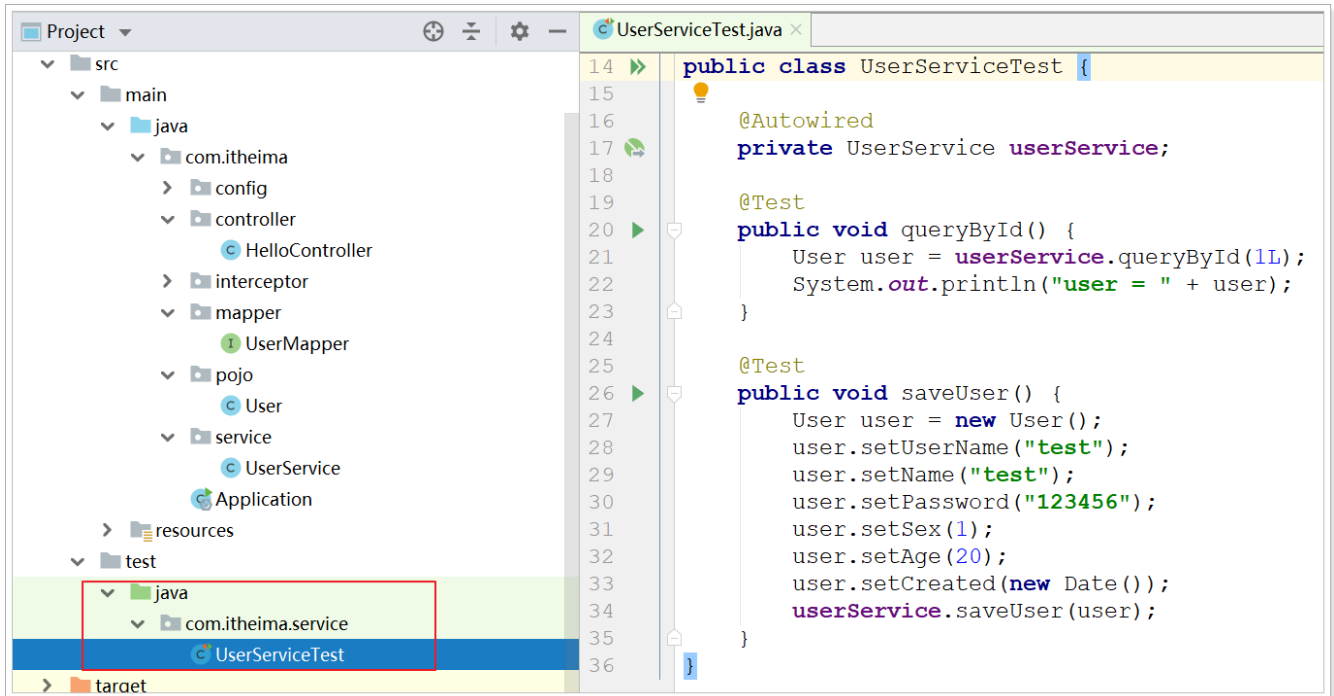
```
1  package com.itheima.service;
2
3  import com.itheima.pojo.User;
4  import org.junit.Test;
5  import org.junit.runner.RunWith;
6  import org.springframework.beans.factory.annotation.Autowired;
7  import org.springframework.boot.test.context.SpringBootTest;
8  import org.springframework.test.context.junit4.SpringRunner;
9
10 import java.util.Date;
11
12 @RunWith(SpringRunner.class)
13 @SpringBootTest
14 public class UserServiceTest {
15
16     @Autowired
17     private UserService userService;
18
19     @Test
20     public void queryById() {
21         User user = userService.queryById(1L);
22         System.out.println("user = " + user);
23     }
24
25     @Test
26     public void saveUser() {
27         User user = new User();
```

```

28     user.setUserName("test");
29     user.setName("test");
30     user.setPassword("123456");
31     user.setSex(1);
32     user.setAge(20);
33     user.setCreated(new Date());
34     userService.saveUser(user);
35 }
36 }

```

测试代码结构如：



5.8. 整合Redis

- 在 pom.xml 文件中添加如下依赖；

```

1     <dependency>
2         <groupId>org.springframework.boot</groupId>
3         <artifactId>spring-boot-starter-data-redis</artifactId>
4     </dependency>

```

- 配置 application.yml 文件；

```

1 spring:
2     redis:
3         host: localhost
4         port: 6379

```

```
application.yml x
1 jdbc:
2   driverClassName: com.mysql.jdbc.Driver
3   url: jdbc:mysql://127.0.0.1:3306/springboot_test
4   username: root
5   password: root
6   #\u52A0\u8F7D\u5176\u5B83\u914D\u7F6E\u6587\u4EF6
7   spring:
8     profiles:
9       active: abc,def
10    datasource:
11      driver-class-name: com.mysql.jdbc.Driver
12      url: jdbc:mysql://localhost:3306/springboot_test
13      username: root
14      password: root
15      redis:
16        host: localhost
17        port: 6379
```

- 编写 `src\test\java\com\itheima\redis\RedisTest.java` 测试代码;

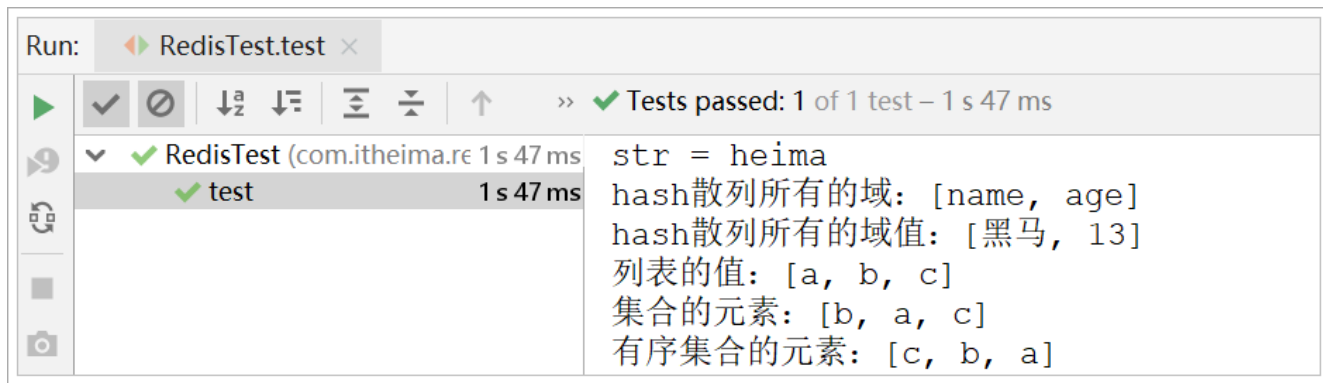
```
1 package com.itheima.redis;
2
3 import org.junit.Test;
4 import org.junit.runner.RunWith;
5 import org.springframework.beans.factory.annotation.Autowired;
6 import org.springframework.boot.test.context.SpringBootTest;
7 import org.springframework.data.redis.core.RedisTemplate;
8 import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
9 import org.springframework.test.context.junit4.SpringRunner;
10
11 import java.util.List;
12 import java.util.Set;
13
14 @RunWith(SpringRunner.class)
15 @SpringBootTest
16 public class RedisTest {
17
18     @Autowired
19     private RedisTemplate redisTemplate;
20
21     @Test
22     public void test(){
23         //string字符串
24         redisTemplate.opsForValue().set("str", "heima");
25         redisTemplate.boundValueOps("str").set("heima");
26         System.out.println("str = " + redisTemplate.opsForValue().get("str"));
27         //hash散列
28         redisTemplate.boundHashOps("h_key").put("name", "黑马");
29         redisTemplate.boundHashOps("h_key").put("age", 13);
```

```

30 //获取所有域对应的值
31 Set set = redisTemplate.boundHashOps("h_key").keys();
32 System.out.println("hash散列所有的域: " + set);
33 List list = redisTemplate.boundHashOps("h_key").values();
34 System.out.println("hash散列所有的域值: " + list);
35
36 //list列表
37 redisTemplate.boundListOps("l_key").leftPush("c");
38 redisTemplate.boundListOps("l_key").leftPush("b");
39 redisTemplate.boundListOps("l_key").leftPush("a");
40 list = redisTemplate.boundListOps("l_key").range(0, -1);
41 System.out.println("列表的值: " + list);
42
43 //set集合
44 redisTemplate.boundSetOps("set_key").add("a", "b", "c");
45 set = redisTemplate.boundSetOps("set_key").members();
46 System.out.println("集合的元素: " + set);
47
48 //sorted set有序集合
49 redisTemplate.boundZSetOps("z_key").add("a", 30);
50 redisTemplate.boundZSetOps("z_key").add("b", 20);
51 redisTemplate.boundZSetOps("z_key").add("c", 10);
52 set = redisTemplate.boundZSetOps("z_key").range(0, -1);
53 System.out.println("有序集合的元素: " + set);
54 }
55 }

```

- 运行上述代码测试



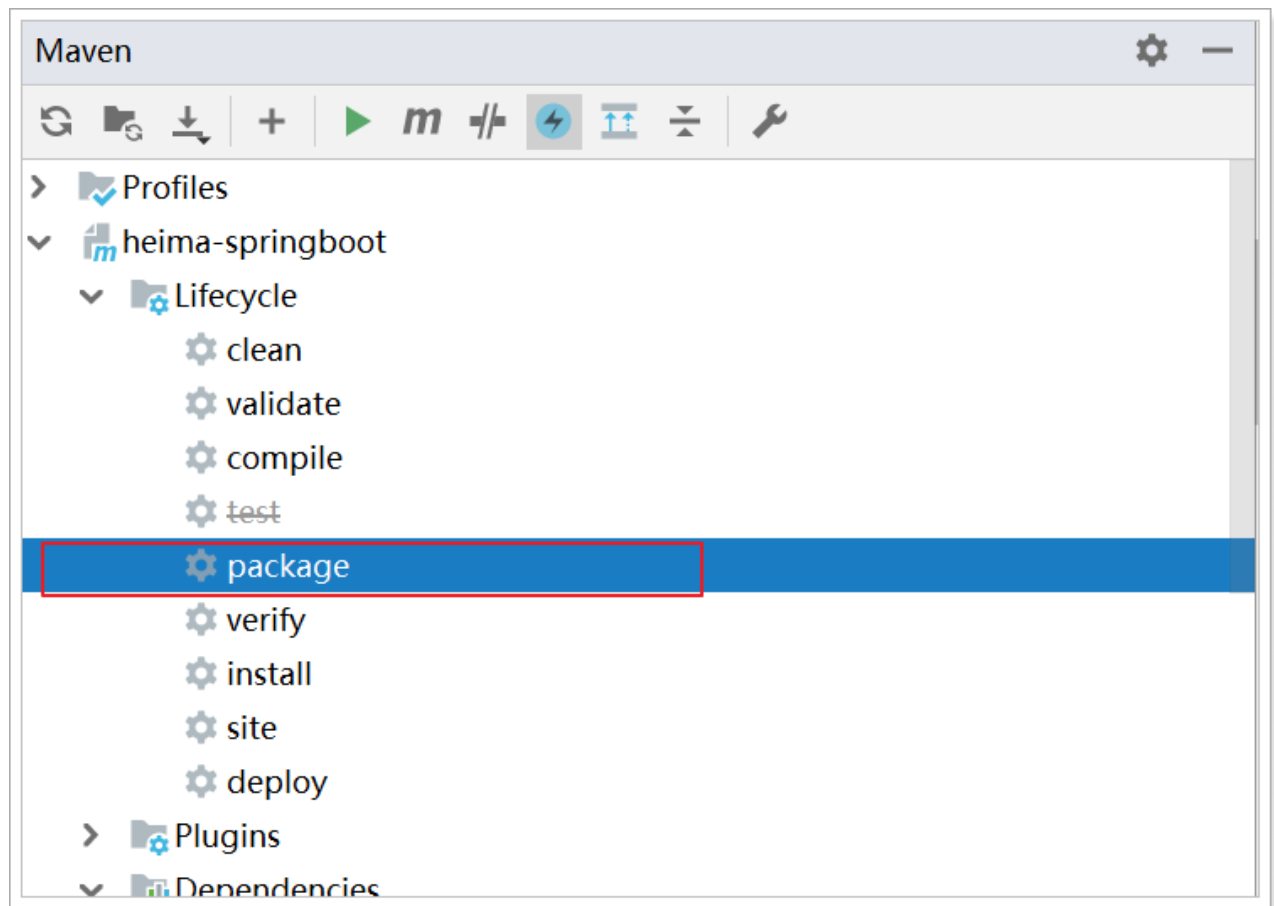
6. Spring Boot项目部署

6.1. 项目打包

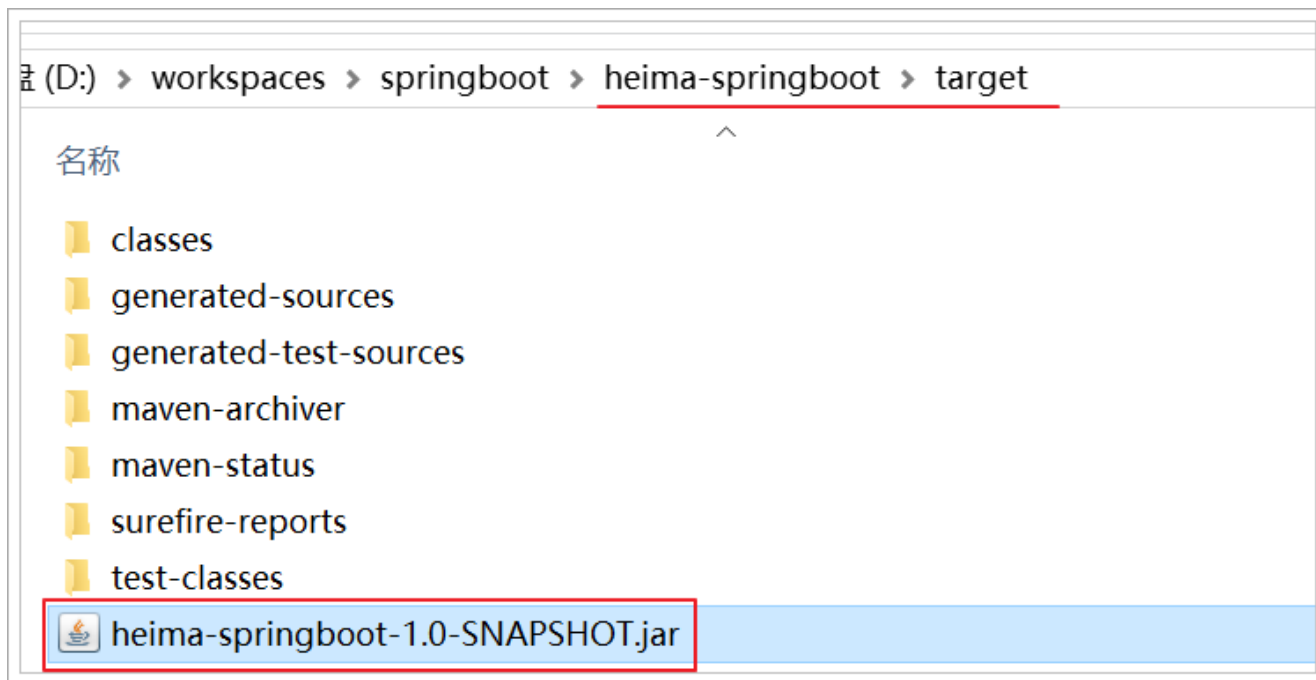
1. 添加项目的pom.xml插件；在pom.xml要显式的加入插件spring-boot-maven-plugin，否则无法产生jar 清单文件，导致打出来的jar 无法使用命令运行；

```
1      <build>
2          <plugins>
3              <!-- 打jar包时如果不配置该插件，打出来的jar包没有清单文件 -->
4              <plugin>
5                  <groupId>org.springframework.boot</groupId>
6                  <artifactId>spring-boot-maven-plugin</artifactId>
7              </plugin>
8          </plugins>
9      </build>
```

2. 使用maven的命令package打包;



之后在项目下的 `target` 目录中将有如下jar包:



【注意】在查看打出的 jar 的时候，将发现 jar 包里面包含 jar 包；这样的包称为 fatjar（肥包）


6.2. 运行

运行打出来的包；使用命令：`java -jar 包全名` 或者写一个 bat 文件，里面包含 `java -jar 包全名`；这样就可以双击启动应用。

如执行上述打出来的jar的命令为：

```
1 | java -jar heima-springboot-1.0-SNAPSHOT.jar
```

```
D:\workspaces\springboot\heima-springboot\target>java -jar heima-springboot-1.0-SNAPSHOT.jar
```



Spring Boot (v2.1.5.RELEASE)

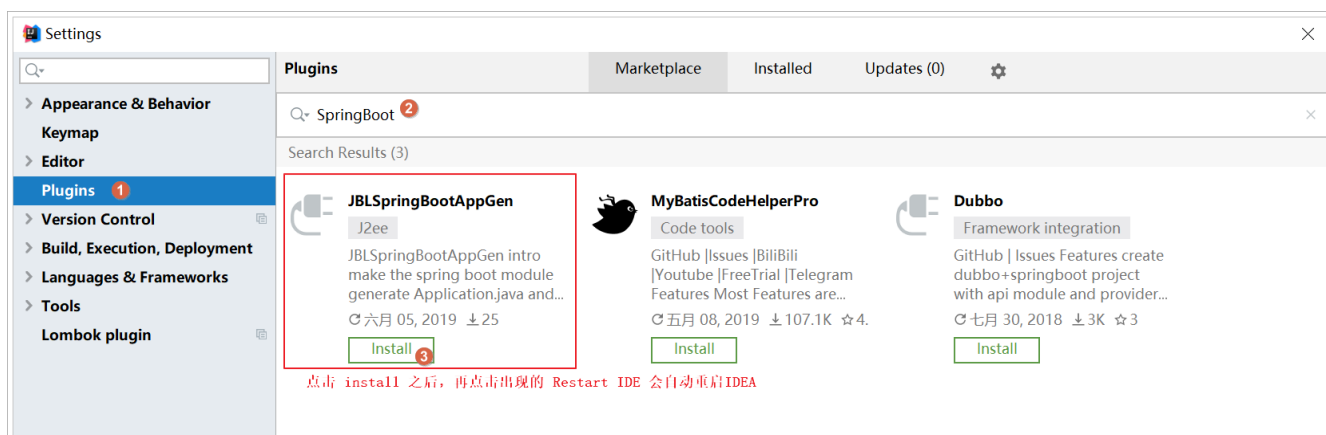
测试则可使用浏览器访问：<http://localhost/user/8>

7. 附录一插件安装

在应用spring boot工程的时候；一般情况下都需要创建启动引导类Application.java和application.yml配置文件，而且内容都是一样的；为了便捷可以安装一个IDEA的插件 JBLSpringBootApplicationGen 在项目上右击之后可以自动生成启动引导类Application.java和application.yml配置文件。

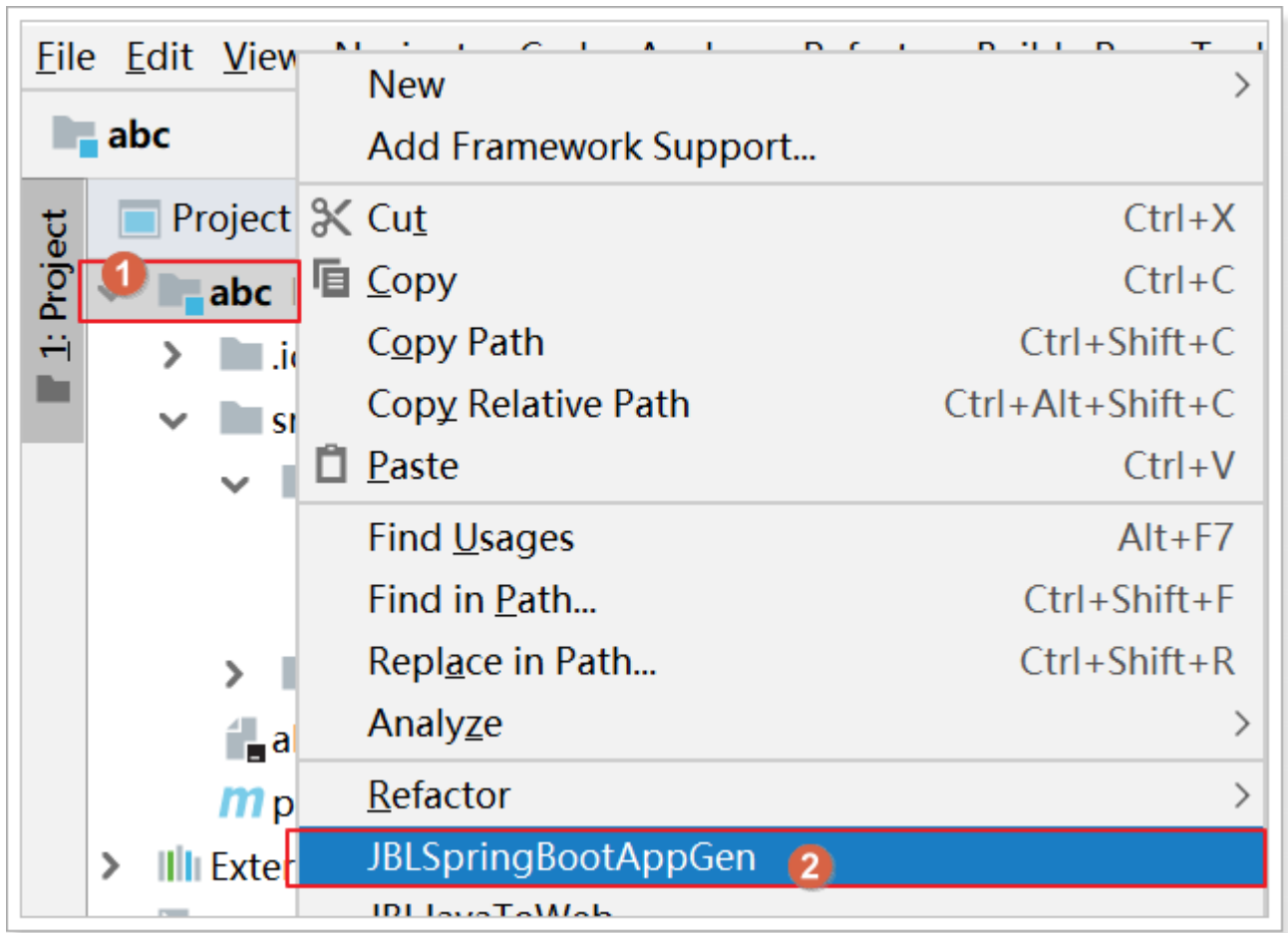
7.1. 安装插件

打开IDEA的设置界面（按 `ctrl+alt+S`）之后在插件选项中搜索SpringBoot，安装 JBLSpringBootApplicationGen

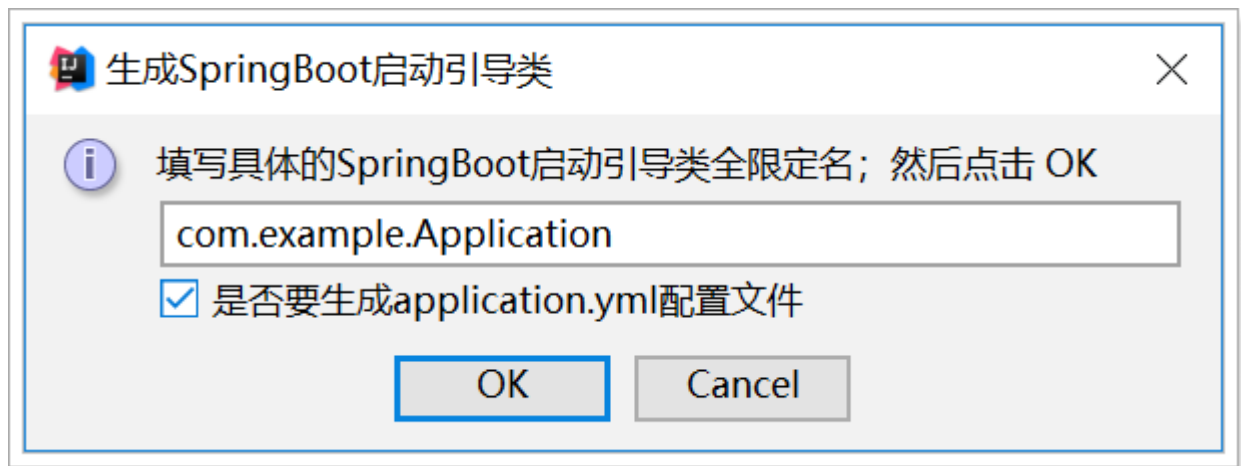


7.2. 应用插件

在IDEA中任意一个maven项目或src目录上 右击，选择 JBLSpringBootApplicationGen 即可。



在如下的界面中输入 启动引导类的名称并根据需要勾选是否要生成application.yml配置文件。



点击 **OK** 之后，在项目中将发现如下内容：

