

# 0.学习目标

- 了解SpringBoot的作用
- 掌握java配置的方式
- 了解SpringBoot自动配置原理
- 掌握SpringBoot的基本使用
- 了解Thymeleaf的基本使用

## 1. 了解SpringBoot

在这一部分，我们主要了解以下3个问题：


- 什么是SpringBoot
- 为什么要学习SpringBoot
- SpringBoot的特点

### 1.1.什么是SpringBoot

SpringBoot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：


#### Main Projects

From configuration to security, web apps to big data – whatever the infrastructure needs of your application may be, there is a **Spring Project** to help you build it. Start small and use just what you need – **Spring is modular by design.**




**SPRING BOOT**

Takes an opinionated view of building Spring applications and gets you up and running as quickly as possible.



**SPRING FRAMEWORK**

Provides core support for dependency injection, transaction management, web apps, data access, messaging and more.



**SPRING CLOUD DATA FLOW**

An orchestration service for composable data microservice applications on modern runtimes.

我们可以看到下面的一段介绍：

Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can "just run".

We take an opinionated view of the Spring platform and third-party libraries so you can get started with minimum fuss. Most Spring Boot applications need very little Spring configuration.

翻译一下：

Spring Boot你只需要“run”就可以非常轻易的构建独立的、生产级别的spring应用。

我们为spring平台和第三方依赖库提供了一种固定化的使用方式，使你能非常轻松的开始开发你的应用程序。大部分Spring Boot应用只需要很少的配置。

其实人们把Spring Boot称为搭建程序的 **脚手架**。其最主要作用就是帮我们快速的构建庞大的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，让我们关注于业务而非配置。

我们可以使用SpringBoot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。

## 1.2.为什么要学习SpringBoot

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，究其原因主要是两点：

- **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。

- **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，这也是件棘手的问题。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SpringBoot让这一切成为过去！

## 1.3.SpringBoot的特点

Spring Boot 主要特征是：

- 创建独立的spring应用程序
- **直接内嵌tomcat、jetty和undertow**（不需要打包成war包部署）
- 提供了固定化的“starter”配置，以简化构建配置
- 尽可能的自动配置spring和第三方库
- 提供产品级的功能，如：安全指标、运行状况监测和外部化配置等
- 绝对不会生成代码，并且不需要XML配置

总之，Spring Boot为所有 Spring 的开发者提供一个开箱即用的、非常快速的、广泛接受的入门体验  
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SpringBoot搭建一个web工程，体会一下SpringBoot的魅力所在！

环境要求：

### 9. System Requirements

Spring Boot 2.0.2.RELEASE requires [Java 8 or 9](#) and [Spring Framework 5.0.6.RELEASE](#) or above. Explicit build support is provided for Maven 3.2+ and Gradle 4.

#### 9.1 Servlet Containers

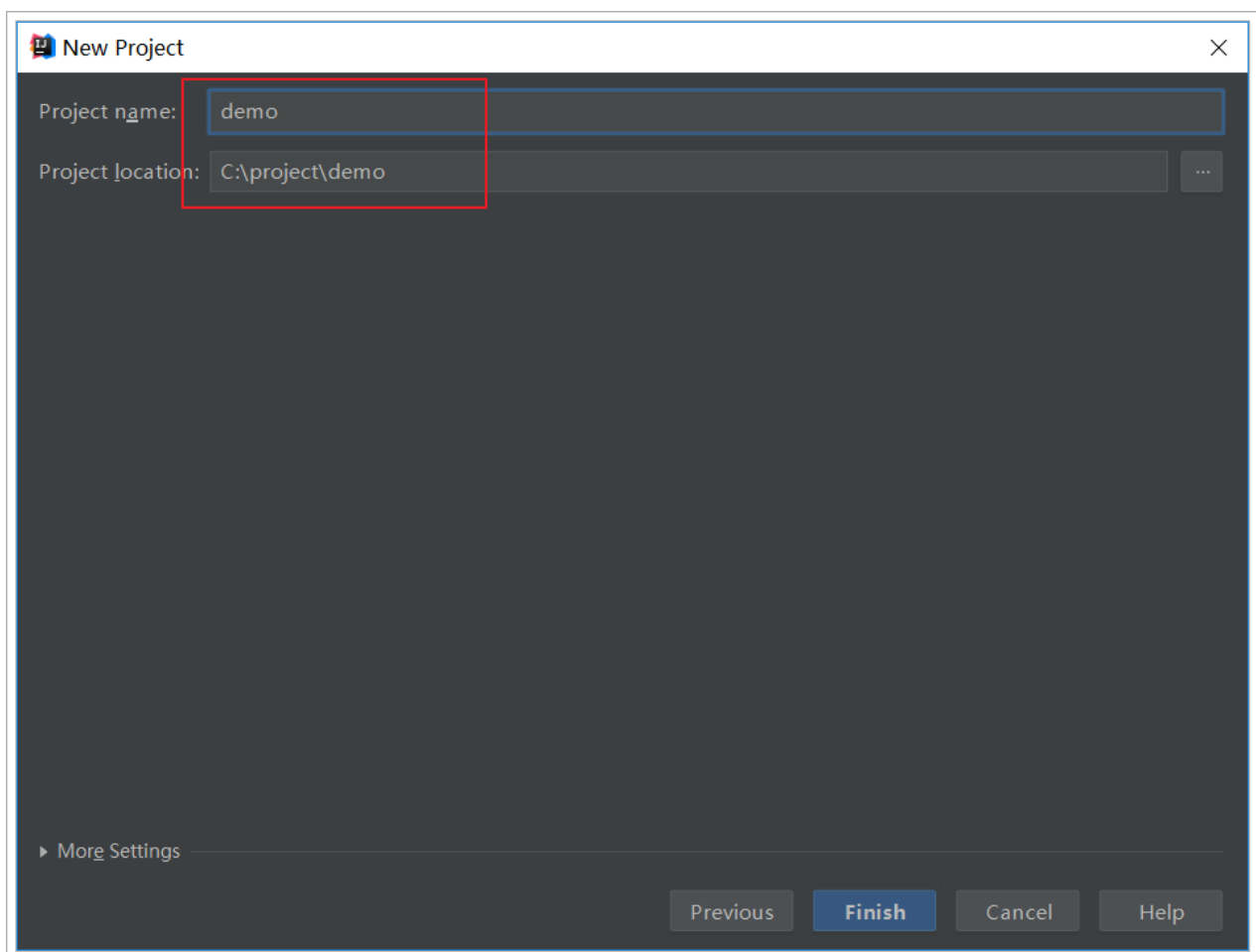
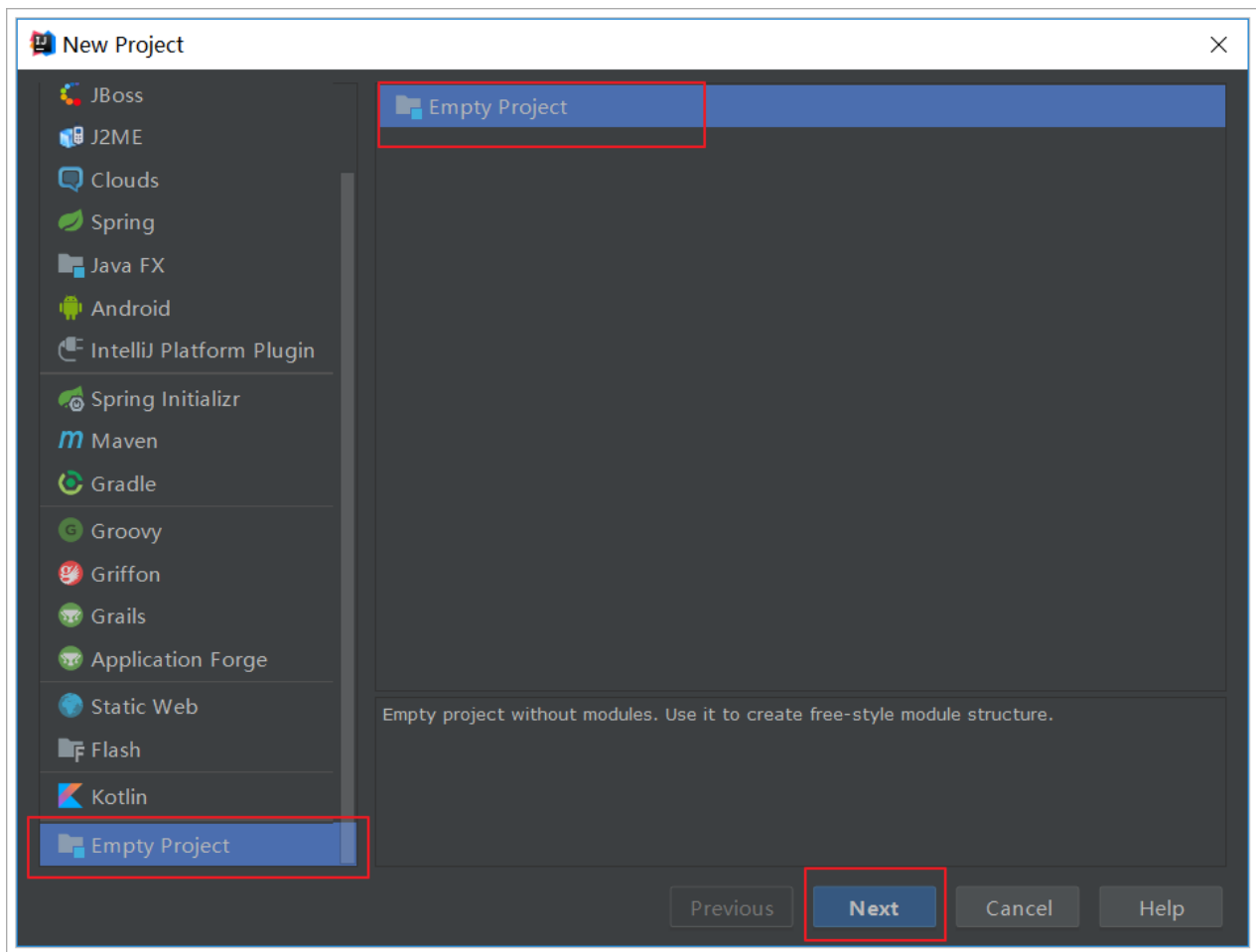
Spring Boot supports the following embedded servlet containers:

Name	Servlet Version
Tomcat 8.5	3.1
Jetty 9.4	3.1
Undertow 1.4	3.1

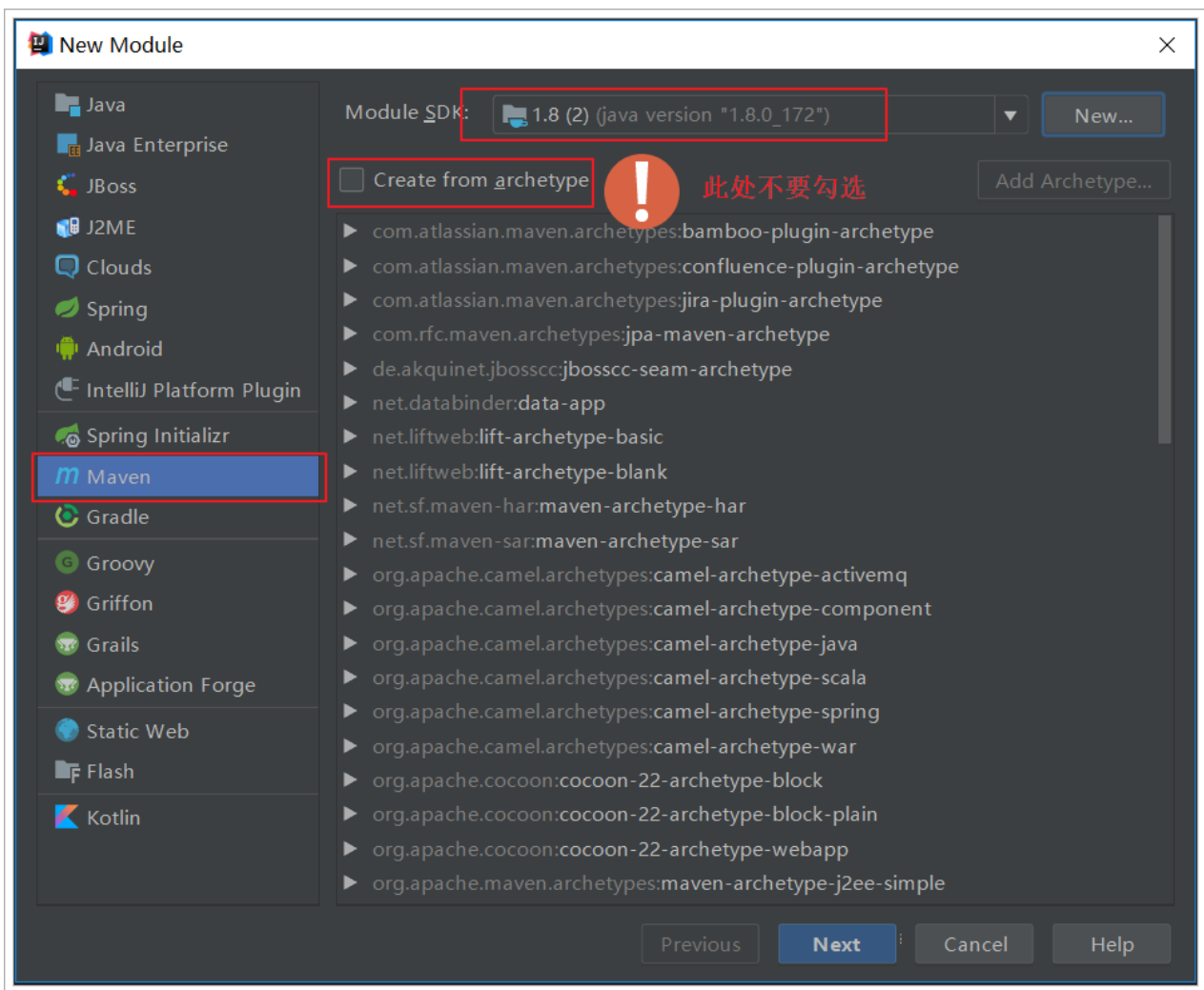
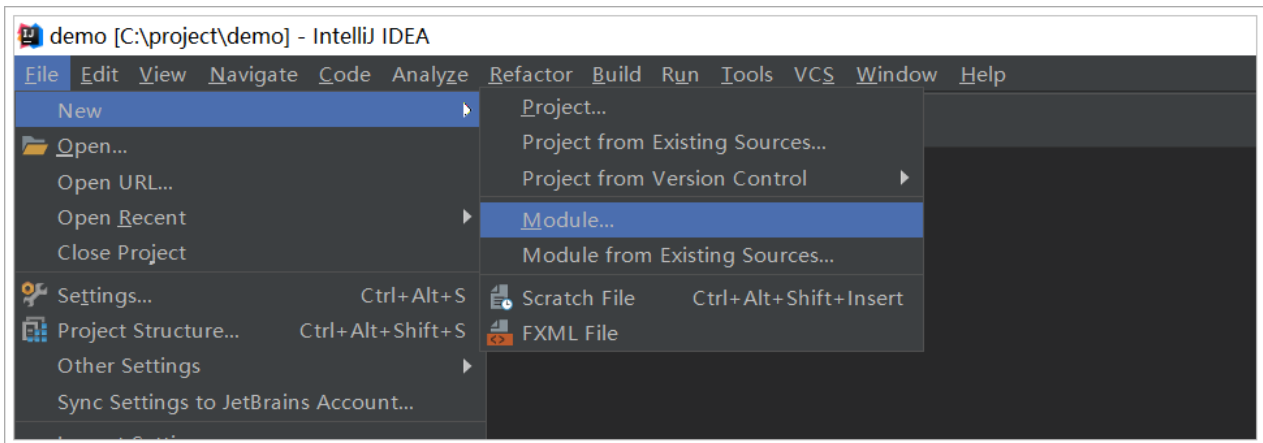
You can also deploy Spring Boot applications to any Servlet 3.1+ compatible container.

### 2.1.创建工程

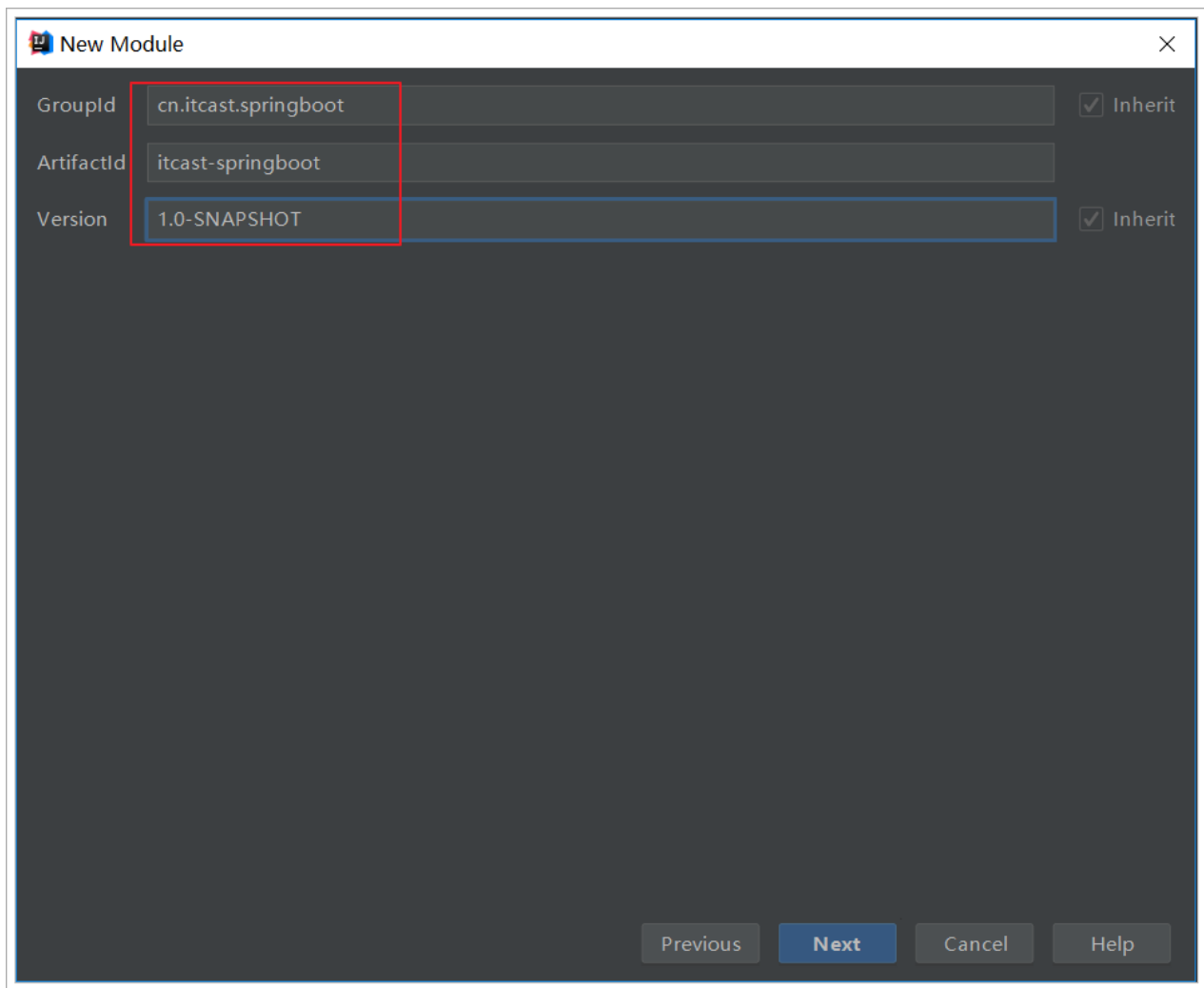
我们先新建一个空的demo工程，如下：



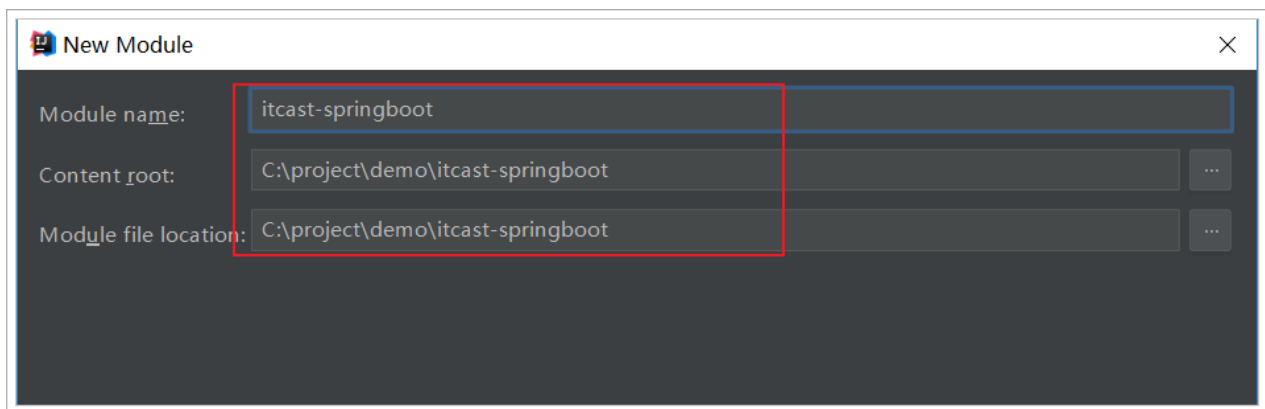
创建以modulel:



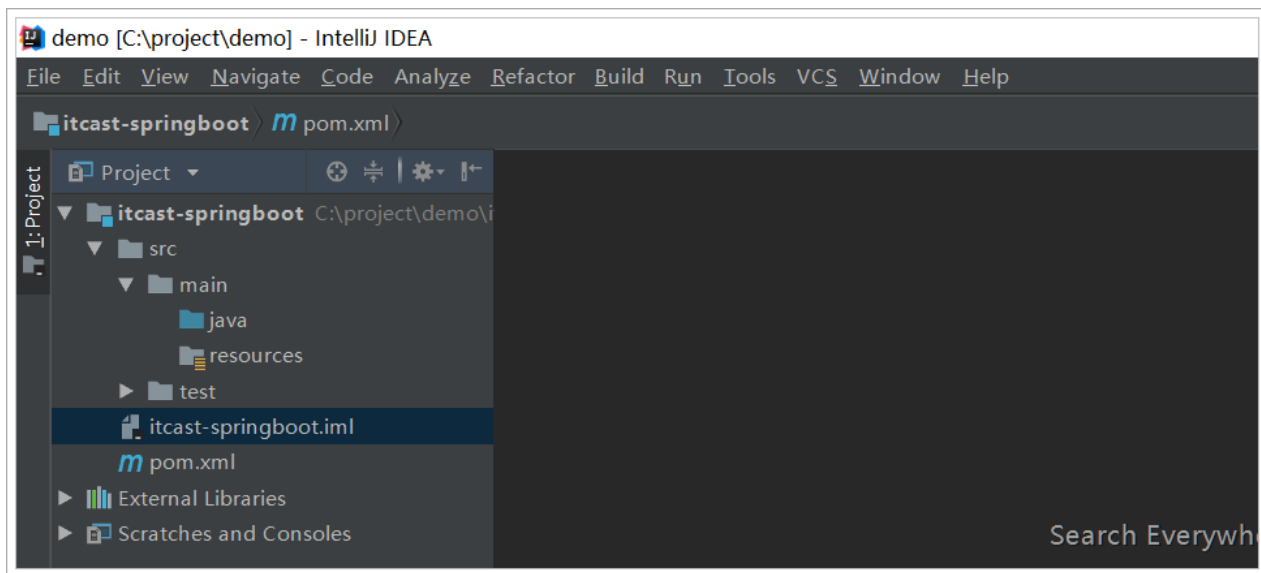
填写坐标信息:



目录结构：



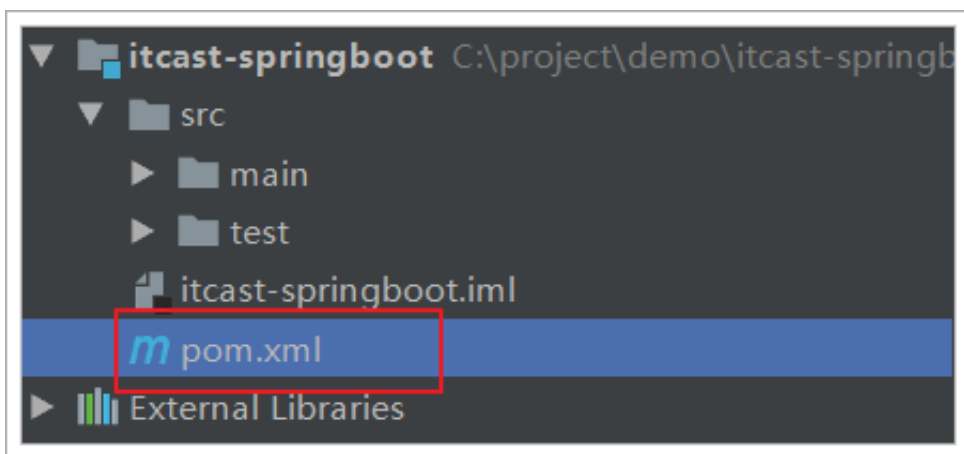
创建完成后的目录结构：



## 2.2.引入依赖

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SpringBoot不帮我们管理吗？

别着急，现在我们的项目与SpringBoot还没有什么关联。SpringBoot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！



```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
          xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

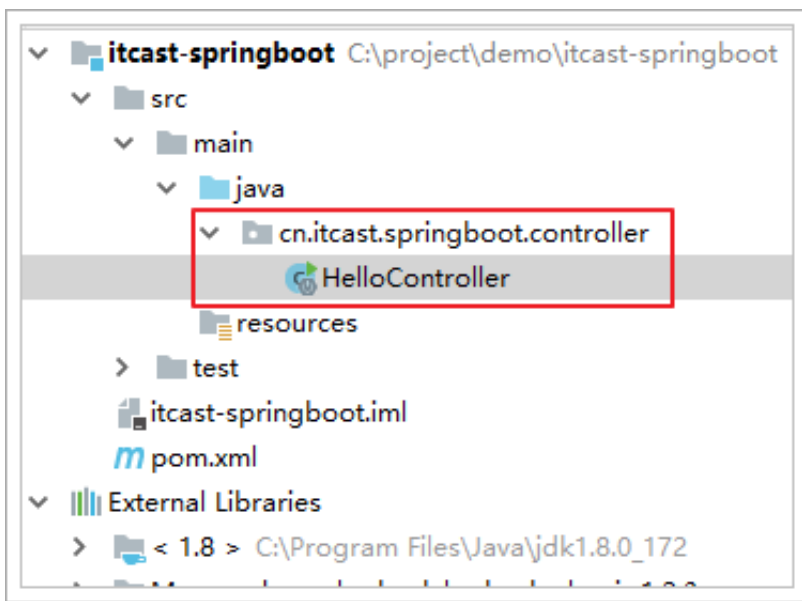
    <groupId>cn.itcast.springboot</groupId>
    <artifactId>itcast-springboot</artifactId>
    <version>1.0-SNAPSHOT</version>
```

```
<!-- 所有的springboot的工程都以spring父工程为父工程 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.6.RELEASE</version>
</parent>

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

启动器                  启动器的名字

## 2.3.编写HelloController



代码：



```

@RestController
@EnableAutoConfiguration
public class HelloController {

    @GetMapping("show")
    public String test(){
        return "hello Spring Boot!";
    }

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

```

## 2.4.启动测试

```

"C:\Program Files\Java\jdk1.8.0_172\bin\java" ...
Connected to the target VM, address: '127.0.0.1:12915', transport: 'socket'

      .  _ _ _ _ _
     /\ /  _ _ _ _ _ \
    ( ( ) \ _ _ _ _ _ \
   \V / _ _ _ _ _ \
    '  _ _ _ _ _ \
   =====|_|=====|_|/=//_/_/_/

:: Spring Boot ::      (v2.0.2.RELEASE)

2018-05-31 16:27:48.561 INFO 21752 --- [main] cn.itcast.controller.HelloController

```

```
o.s.c.t.ContextLoader      : Root WebApplicationContext: initialization completed
o.s.c.t.ServletRegistrationBean : Servlet dispatcherServlet mapped to [/]
o.s.c.t.FilterRegistrationBean  : Mapping filter: 'characterEncodingFilter' to: [/*]
o.s.c.t.FilterRegistrationBean  : Mapping filter: 'hiddenHttpMethodFilter' to: [/*]
o.s.c.t.FilterRegistrationBean  : Mapping filter: 'httpPutFormContentFilter' to: [/*]
o.s.c.t.FilterRegistrationBean  : Mapping filter: 'requestContextFilter' to: [/*]
o.s.c.r.SimpleUrlHandlerMapping : Mapped URL path [/**/favicon.ico] onto handler of type
o.s.c.rquestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot
o.s.c.rquestMappingHandlerMapping : Mapped "[/show,methods=[GET]]" onto public java.lan
o.s.c.rquestMappingHandlerMapping : Mapped "[/error]" onto public org.springframework.h
o.s.c.rquestMappingHandlerMapping : Mapped "[/error,produces=[text/html]]" onto public
o.s.c.r.SimpleUrlHandlerMapping : Mapped URL path [/webjars/**] onto handler of type [c
o.s.c.r.SimpleUrlHandlerMapping : Mapped URL path [/**] onto handler of type [class org
o.s.c.tationMBeanExporter      : Registering beans for JMX exposure on startup
o.s.c.led.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context p
o.s.c.troller.HelloController  : Started HelloController in 8.115 seconds (JVM running
```



bingo! 访问成功!

## 2.5.详解

入门工程中: pom.xml里引入了启动器的概念以@EnableAutoConfiguration注解。

### 2.5.1.启动器

为了让SpringBoot帮我们完成各种自动配置,我们必须引入SpringBoot提供的自动配置依赖,我们称为启动器。spring-boot-starter-parent工程将依赖关系声明为一个或者多个启动器,我们可以根据项目需求引入相应的启动器,因为我们是web项目,这里我们引入web启动器:

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

需要注意的是,我们并没有在这里指定版本信息。因为SpringBoot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：



这些都是SpringBoot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。

## 2.5.2.@EnableAutoConfiguration

关于这个注解，官网上有一段说明：

Enable auto-configuration of the Spring Application Context, attempting to guess and configure beans that you are likely to need. Auto-configuration classes are usually applied based on your classpath and what beans you have defined.

简单翻译以下：

开启spring应用程序的自动配置，SpringBoot基于你所添加的依赖和你自己定义的bean，试图去猜测并配置你想要的配置。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，SpringBoot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，只需要引入所需依赖，配置就可以交给SpringBoot处理了。  
根据你引入的依赖来自动配置

## 2.6.优化入门程序

现在工程中只有一个Controller，可以这么玩；那么如果有多个Controller，怎么办呢？

添加Hello2Controller：

代码：

```
@RestController
public class Hello2Controller {

    @GetMapping("show2")
    public String test(){
        return "hello Spring Boot2!";
    }

}
```

启动重新启动，访问show2测试，失败：



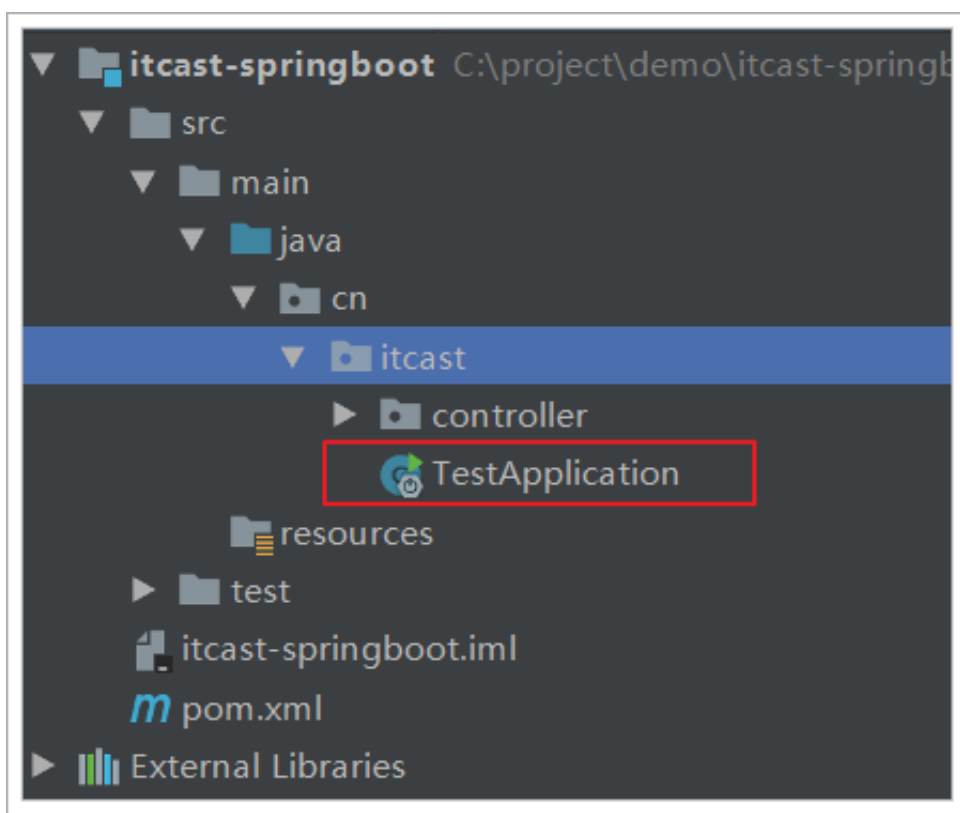
难道要在每一个Controller中都添加一个main方法和@EnableAutoConfiguration注解，这样启动一个springboot程序也太麻烦了。也无法同时启动多个Controller，因为每个main方法都监听8080端口。所以，一个springboot程序应该只有一个springboot的main方法。

所以，springboot程序引入了一个全局的引导类。

## 2.5.1.添加引导类

通常请求下，我们在一个springboot工程中都会在基包下创建一个引导类，一些springboot的全局注解（@EnableAutoConfiguration注解）以及springboot程序的入口main方法都放在该类中。

在springboot的程序的基包下（引导类和Controller包在同级目录下），创建TestApplication.class：



内容如下：

```

@EnableAutoConfiguration
public class TestApplication {

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

```

并修改HelloController，去掉main方法及@EnableAutoConfiguration：

```

@RestController
public class HelloController {

    @GetMapping("show")
    public String test(){
        return "hello Spring Boot!";
    }
}

```

启动引导类，访问show测试：



发现所有的Controller都不能访问，为什么？

回想以前程序，我们在配置文件中添加了注解扫描，它能扫描指定包下的所有Controller，而现在并没有。怎么解决——@ComponentScan注解

## 2.5.2.@ComponentScan

spring框架除了提供配置方式的注解扫描 `<context:component-scan />`，还提供了注解方式的注解扫描 `@ComponentScan`。

在TestApplication.class中，使用@ComponentScan注解：

```

@EnableAutoConfiguration
@ComponentScan
public class TestApplication {

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

}

```

重新启动，访问show或者show2：



我们跟进该注解的源码，并没有看到什么特殊的地方。我们查看注释：

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@ComponentScan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。一般启动类会放在一个比较浅的包目录中。

#### 组合注解

### 2.5.3.@SpringBootApplication

我们现在的引导类中使用了@EnableAutoConfiguration和@ComponentScan注解，有点麻烦。springboot提供了一种简便的玩法：@SpringBootApplication注解

使用@SpringBootApplication改造TestApplication：

```

@SpringBootApplication
public class TestApplication {

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

}

```

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 {

```

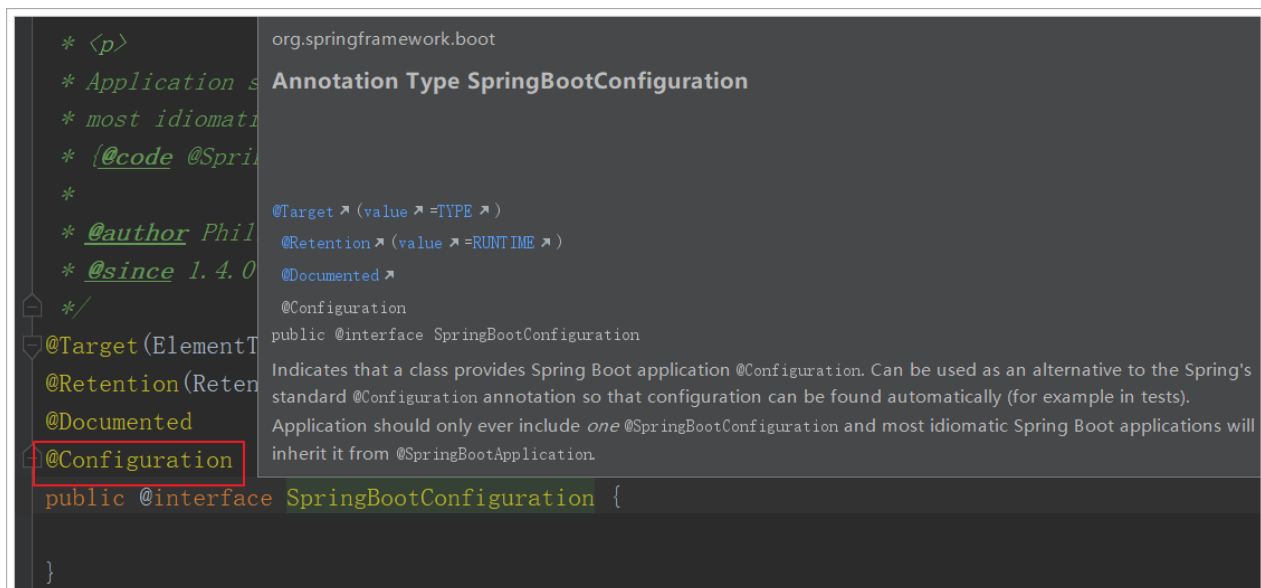
发现@SpringBootApplication其实是一个组合注解，这里重点的注解有3个：

- @SpringBootConfiguration
- @EnableAutoConfiguration：开启自动配置
- @ComponentScan：开启注解扫描

## 2.5.4. @SpringBootConfiguration

@SpringBootConfiguration注解的源码：

我们继续点击查看源码：



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 `@SpringBootApplication` 是用来声明当前类是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。

## 3.默认配置原理

springboot的默认配置方式和我们之前玩的配置方式不太一样，没有任何的xml。那么如果自己要新增配置该怎么办？比如我们要配置一个数据库连接池，以前会这么玩：

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

现在该怎么做呢？

### 3.1.回顾历史

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SpringBoot来临，人们才慢慢认识到java配置优雅。

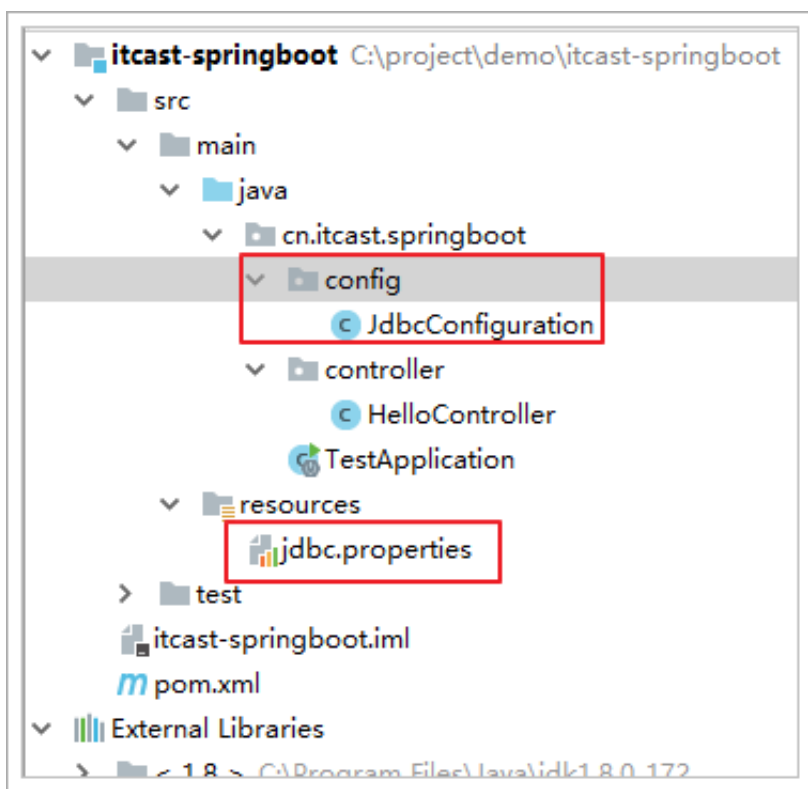
有句古话说的好：拥抱变化，拥抱未来。所以我们也应该顺应时代潮流，做时尚的弄潮儿，一起来学习下java配置玩法。

## 3.2.尝试java配置

java配置主要靠java类和一些注解来达到和xml配置一样的效果，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



### 3.2.1.引入依赖

首先在pom.xml中，引入Druid连接池依赖：

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

### 3.2.2.添加jdbc.properties

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

### 3.2.3.配置数据源

创建JdbcConfiguration类：

```
@Configuration
@PropertySource("classpath:jdbc.properties")
public class JdbcConfiguration {

    使用"$"来获取外部文件里的数据
    @Value("${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;
    }
}
```

```
}
```

解读：

- `@Configuration`：声明 `JdbcConfiguration` 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。相当于以前的bean标签

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

### 3.2.4.测试

我们在 `HelloController` 中测试：

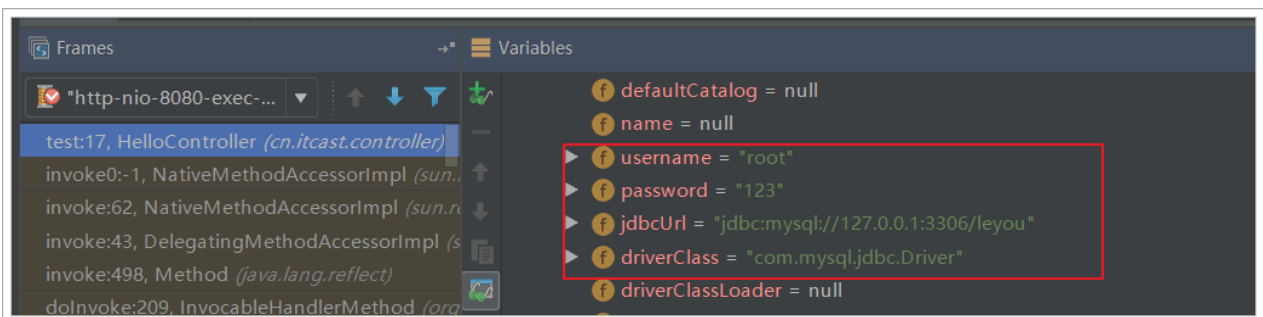
```
@RestController
public class HelloController {

    @Autowired
    private DataSource dataSource;

    @GetMapping("show")
    public String test(){
        return "hello Spring Boot!";
    }

}
```

在test方法中打一个断点，然后Debug运行并查看：



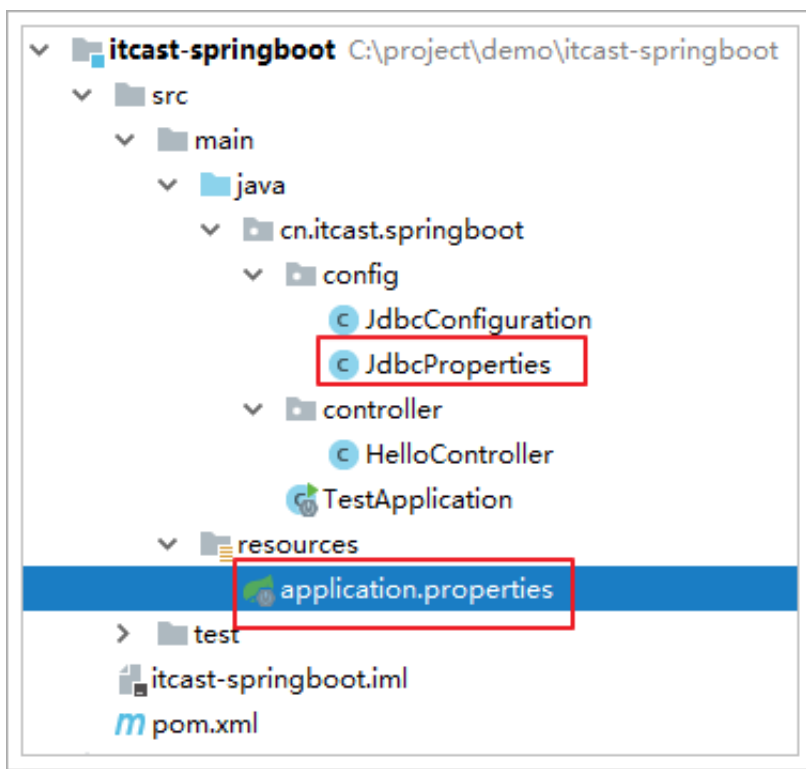
属性注入成功了！

## 3.3.SpringBoot的属性注入

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SpringBoot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 JdbcProperties，用来进行属性注入：



代码：

```
@ConfigurationProperties(prefix = "jdbc")
public class JdbcProperties {
    private String url;
    private String driverClassName;
    private String username;
    private String password;
    // ... 略
    // getters 和 setters
}
```

可以注入任意类型的数据

- 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，并且必须具有getter和setter方法
- 需要注意的是，这里我们并没有指定属性文件的地址，SpringBoot默认会读取文件名为 application.properties的资源文件，所以我们将jdbc.properties名称改为 application.properties

2) 在JdbcConfiguration中使用这个属性：

- 通过@EnableConfigurationProperties(JdbcProperties.class) 来声明要使用 JdbcProperties 这个类的对象

- 然后你可以通过以下方式在JdbcConfiguration类中注入JdbcProperties:

#### 1. @Autowired注入

```
@Configuration
@EnableConfigurationProperties(JdbcProperties.class)
public class JdbcConfiguration {

    @Autowired
    private JdbcProperties jdbcProperties;

    @Bean
    public DataSource dataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setUrl(jdbcProperties.getUrl());
        dataSource.setDriverClassName(jdbcProperties.getDriverClassName());
        dataSource.setUsername(jdbcProperties.getUsername());
        dataSource.setPassword(jdbcProperties.getPassword());
        return dataSource;
    }
}
```

#### 2. 构造函数注入

```
@Configuration
@EnableConfigurationProperties(JdbcProperties.class)
public class JdbcConfiguration {

    private JdbcProperties jdbcProperties;

    public JdbcConfiguration(JdbcProperties jdbcProperties){
        this.jdbcProperties = jdbcProperties;
    }

    @Bean
    public DataSource dataSource() {
        // 略
    }
}
```

#### 3. @Bean方法的参数注入

```

@Configuration
@EnableConfigurationProperties(JdbcProperties.class)
public class JdbcConfiguration {

    @Bean
    public DataSource dataSource(JdbcProperties jdbcProperties) {
        // ...
    }
}

```

本例中，我们采用第三种方式。

3) 测试结果：

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SpringBoot推荐的注入方式。两者对比关系：

#### 24.7.6 @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）中。而是直接在需要的地方声明即可：

```

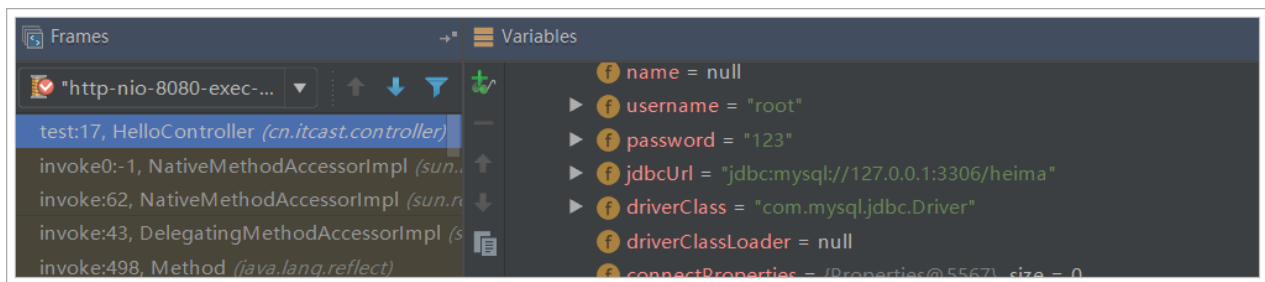
@Configuration
public class JdbcConfiguration {

    @Bean    必须保证方法的返回值有对应的set方法
    // 声明要注入的属性前缀，SpringBoot会自动把相关属性通过set方法注入到DataSource中
    @ConfigurationProperties(prefix = "jdbc")
    public DataSource dataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        return dataSource;
    }
}

```

我们直接把 `@ConfigurationProperties(prefix = "jdbc")` 声明在需要使用的 `@Bean` 的方法上，然后SpringBoot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！

我们将jdbc的url改成：/heima，再次测试：



### 3.5.SpringBoot中的默认配置

通过刚才的学习，我们知道@EnableAutoConfiguration会开启SpringBoot的自动配置，并且根据你引入的依赖来生效对应的默认配置。那么问题来了：

- 这些默认配置是怎么配置的，在哪里配置的呢？
- 为何依赖引入就会触发配置呢？
- 这些默认配置的属性来自哪里呢？

其实在我们的项目中，已经引入了一个依赖：spring-boot-autoconfigure，其中定义了大量自动配置类：

还有：

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：

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：

处理器适配器（HandlerAdapter）：

还有很多，这里就不一一截图了。

另外，这些默认配置的属性来自哪里呢？





我们看到，这里通过@EnableAutoConfiguration注解引入了两个属性：WebMvcProperties和ResourceProperties。

我们查看这两个属性类：

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：

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。

## 3.6.总结

SpringBoot为我们提供了默认配置，而默认配置生效的条件一般有两个：

- 你引入了相关依赖
- 你自己没有配置

### 1) 启动器

之所以，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，参考课前资料中提供的《SpringBoot启动器.txt》

### 2) 全局配置

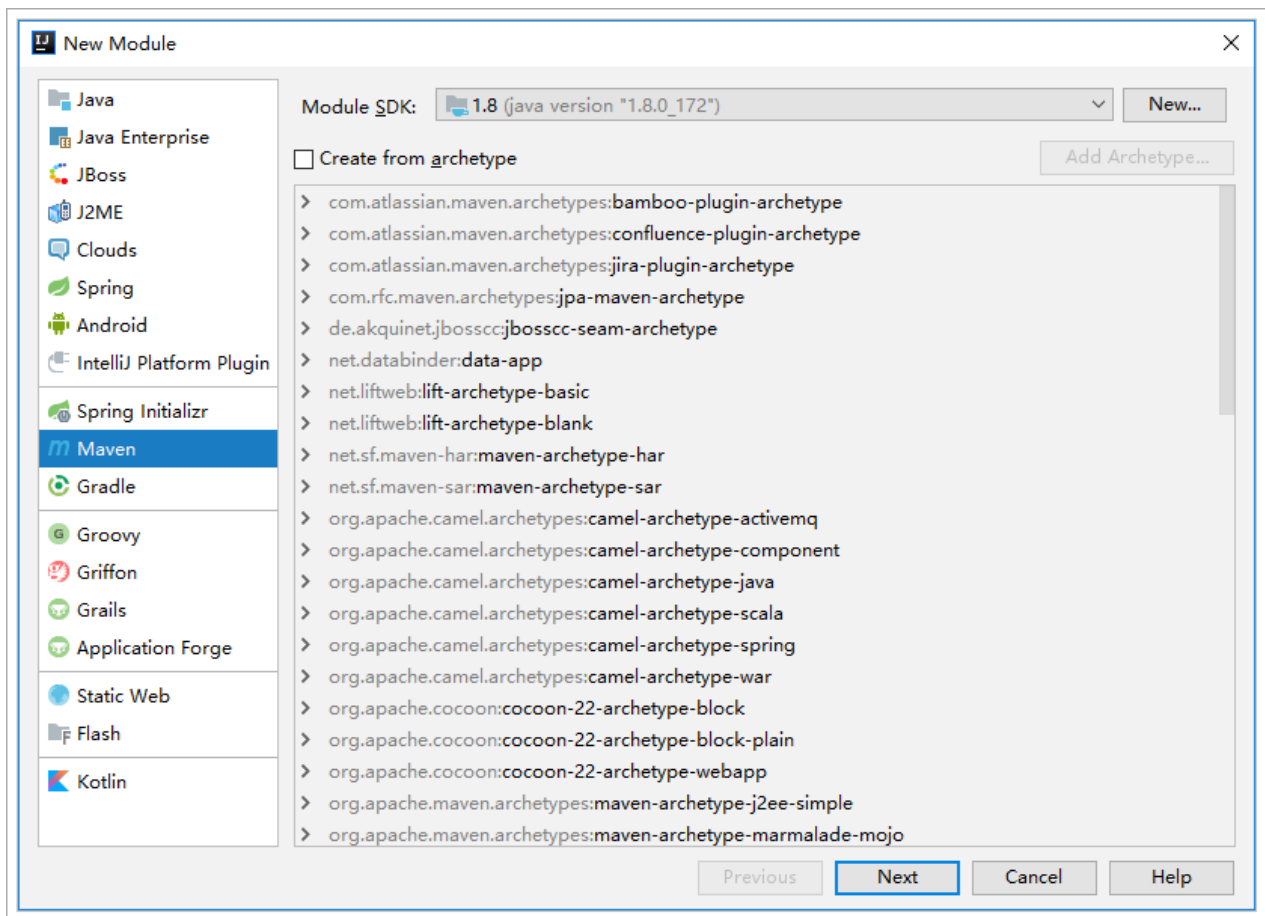
另外，SpringBoot的默认配置，都会读取默认属性，而这些属性可以通过自定义application.properties文件来进行覆盖。这样虽然使用的还是默认配置，但是配置中的值改成了我们自定义的。

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，参考课前资料提供的：《SpringBoot全局属性.md》

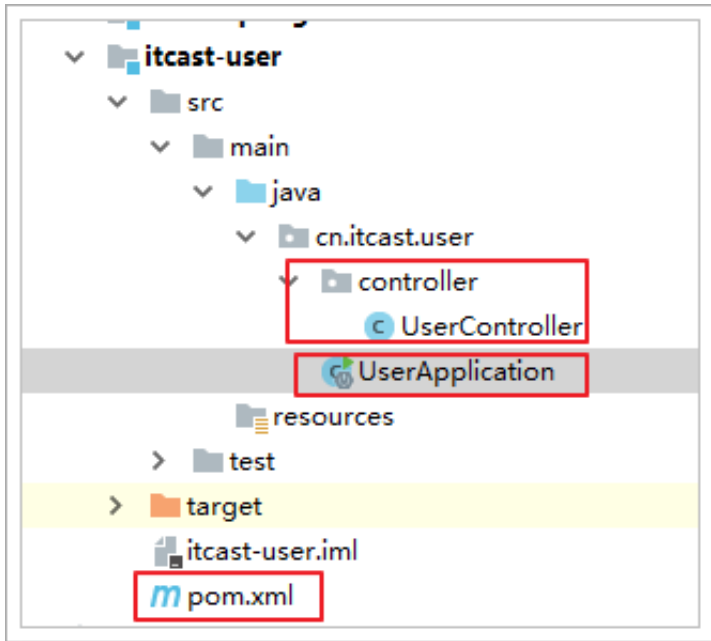
## 4.SpringBoot实战

接下来，我们来看看如何用SpringBoot来玩转以前的SSM,我们沿用之前讲解SSM用到的数据库tb\_user和实体类User

### 4.1.创建工程



## 4.2.编写基本代码



pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.itcast.user</groupId>
    <artifactId>itcast-user</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.6.RELEASE</version>
    </parent>

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

</project>
```

参照上边的项目，编写引导类：

```
@SpringBootApplication
public class UserApplication {

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

编写UserController:

```
@RestController
@RequestMapping("user")
public class UserController {

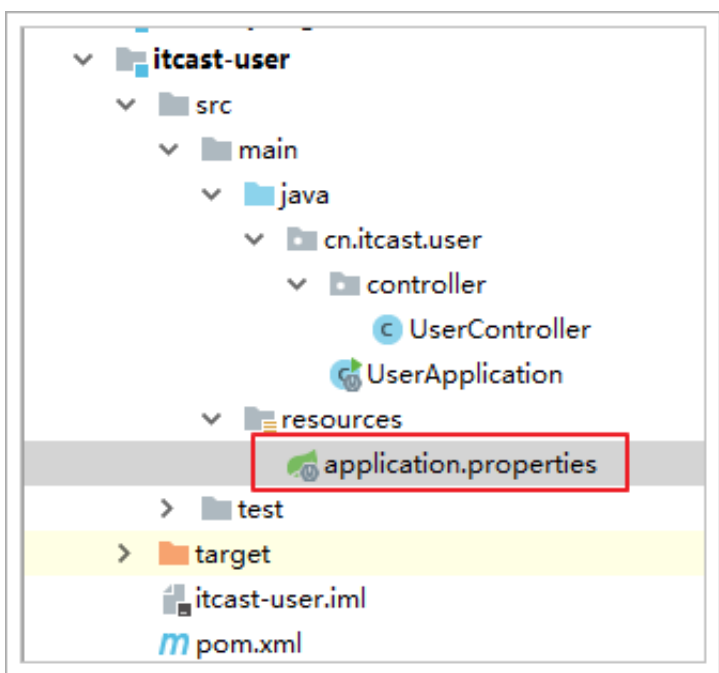
    @GetMapping("hello")
    public String test(){
        return "hello ssm";
    }
}
```

## 4.3.整合SpringMVC

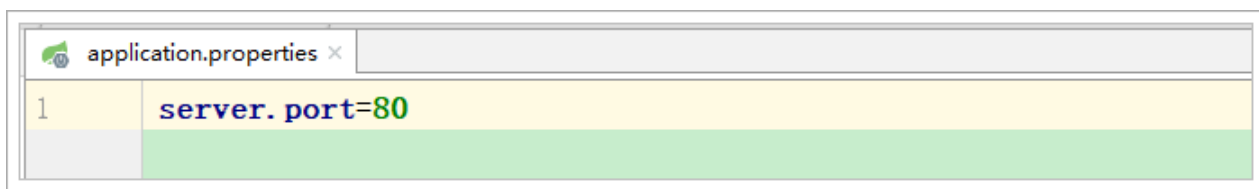
虽然默认配置已经可以使用SpringMVC了，不过我们有时候需要进行自定义配置。

### 4.3.1.修改端口

添加全局配置文件：application.properties

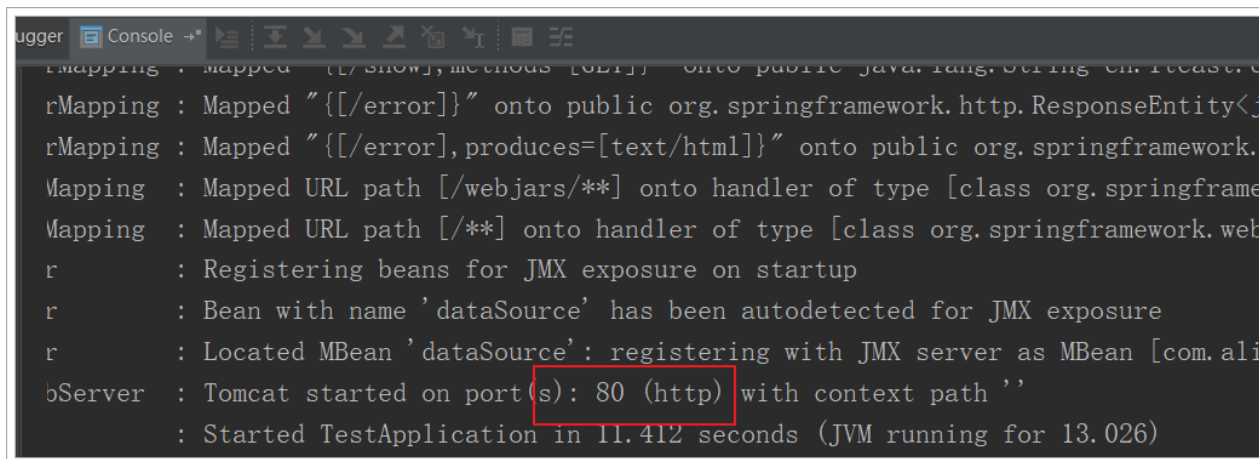


端口通过以下方式配置



# 映射端口  
server.port=80

重启服务后测试：



### 4.3.2.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？

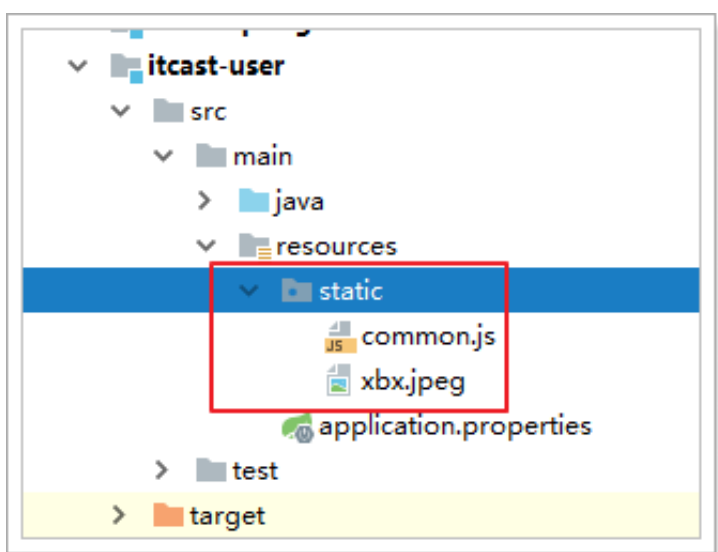
回顾我们上面看的源码，有一个叫做ResourceProperties的类，里面就定义了静态资源的默认查找路径：

默认的静态资源路径为：

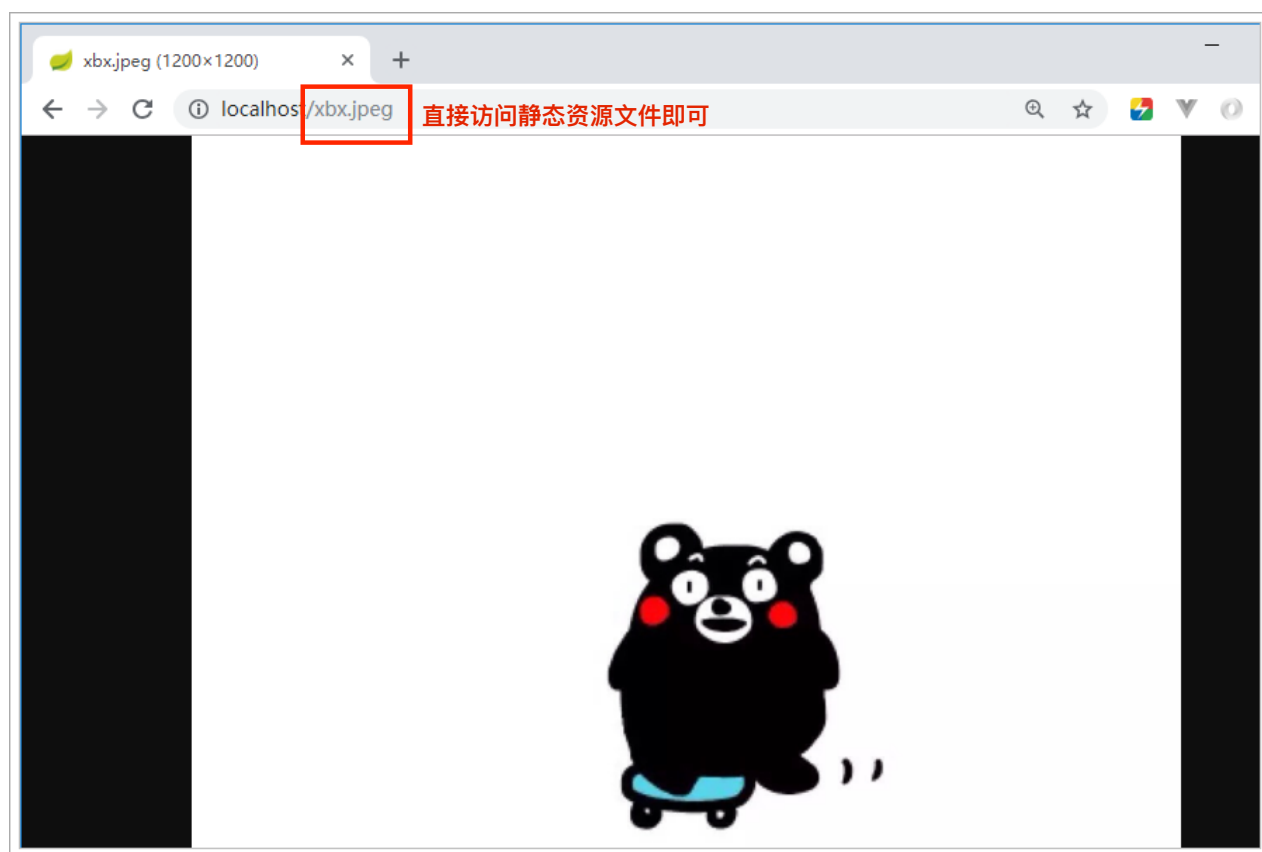
- classpath:/META-INF/resources/
- classpath:/resources/
- classpath:/static/
- classpath:/public/

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，并且添加一些静态资源：



重启项目后测试：



### 4.3.3.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 `ExceptionHandlerResolver`, 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`.

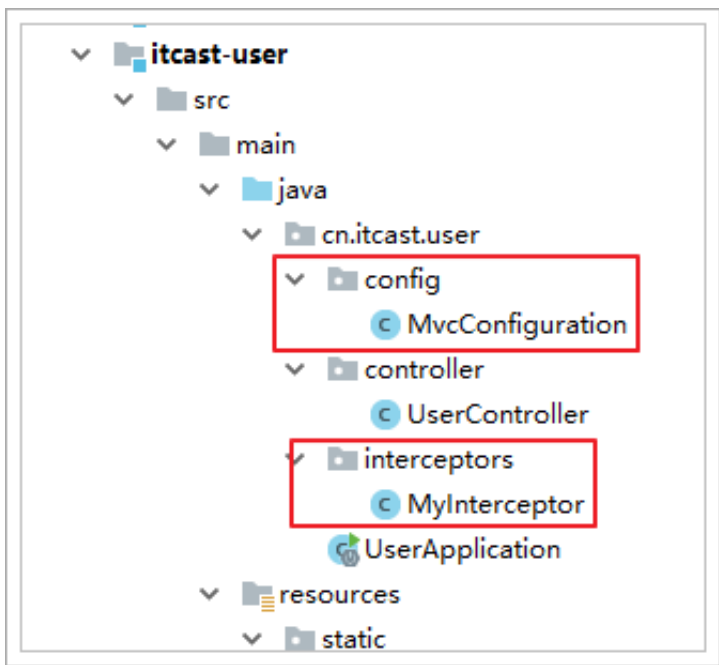
翻译：

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。

实现如下：



首先我们定义一个拦截器：

```
@Component
public class MyInterceptor implements HandlerInterceptor {
    @Override
```

```

    public boolean preHandle(HttpServletRequest request, HttpServletResponse response,
Object handler) throws Exception {
        System.out.println("preHandle method is running!");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response,
Object handler, ModelAndView modelAndView) throws Exception {
        System.out.println("postHandle method is running!");
    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse
response, Object handler, Exception ex) throws Exception {
        System.out.println("afterCompletion method is running!");
    }
}

```

然后定义配置类，注册拦截器：

```

@Configuration
public class MvcConfiguration implements WebMvcConfigurer {

    @Autowired
    private HandlerInterceptor myInterceptor;

    /**
     * 重写接口中的addInterceptors方法，添加自定义拦截器
     * @param registry
     */
    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(myInterceptor).addPathPatterns("/**");
    }
}

```

两个“\*”代表拦截所有资源，一个“\*”代表拦截一级资源

接下来运行并查看日志：

```

preHandle method is running!
postHandle method is running!
afterCompletion method is running!

```

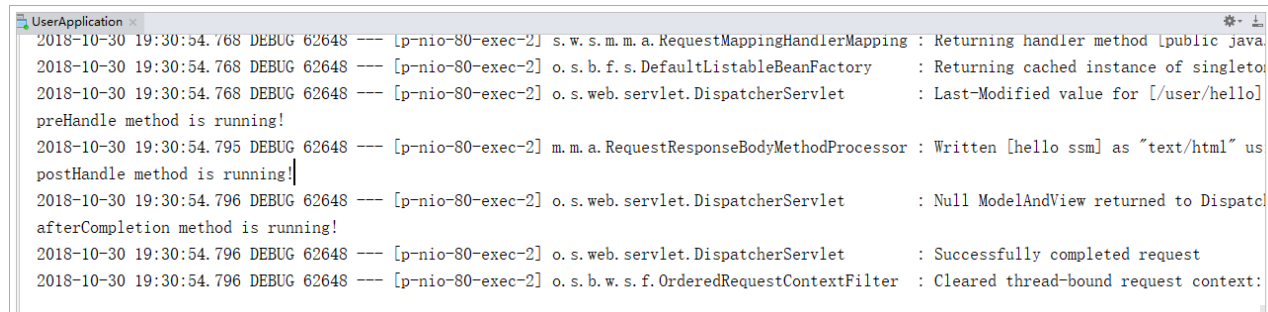
你会发现日志中只有这些打印信息，springMVC的日志信息都没有，因为springMVC记录的log级别是debug，springboot默认是显示info以上，我们需要进行配置。



SpringBoot通过 `logging.level.*=debug` 来配置日志级别，\*填写包名

```
# 设置org.springframework包的日志级别为debug
logging.level.org.springframework=debug
```

再次运行查看：



```
UserApplication
2018-10-30 19:30:54.768 DEBUG 62648 --- [p-nio-80-exec-2] s.w.s.m.m.a.RequestMappingHandlerMapping : Returning handler method [public java
2018-10-30 19:30:54.768 DEBUG 62648 --- [p-nio-80-exec-2] o.s.b.f.s.DefaultListableBeanFactory : Returning cached instance of singleton
2018-10-30 19:30:54.768 DEBUG 62648 --- [p-nio-80-exec-2] o.s.web.servlet.DispatcherServlet : Last-Modified value for [/user/hello]
preHandle method is running!
2018-10-30 19:30:54.795 DEBUG 62648 --- [p-nio-80-exec-2] m.m.a.ResponseBodyMethodProcessor : Written [hello ssm] as "text/html" us
postHandle method is running!
2018-10-30 19:30:54.796 DEBUG 62648 --- [p-nio-80-exec-2] o.s.web.servlet.DispatcherServlet : Null ModelAndView returned to Dispatch
afterCompletion method is running!
2018-10-30 19:30:54.796 DEBUG 62648 --- [p-nio-80-exec-2] o.s.web.servlet.DispatcherServlet : Successfully completed request
2018-10-30 19:30:54.796 DEBUG 62648 --- [p-nio-80-exec-2] o.s.b.w.s.f.OrderedRequestContextFilter : Cleared thread-bound request context:
```

## 4.4.整合连接池

jdbc连接池是spring配置中的重要一环，在SpringBoot中该如何处理呢？

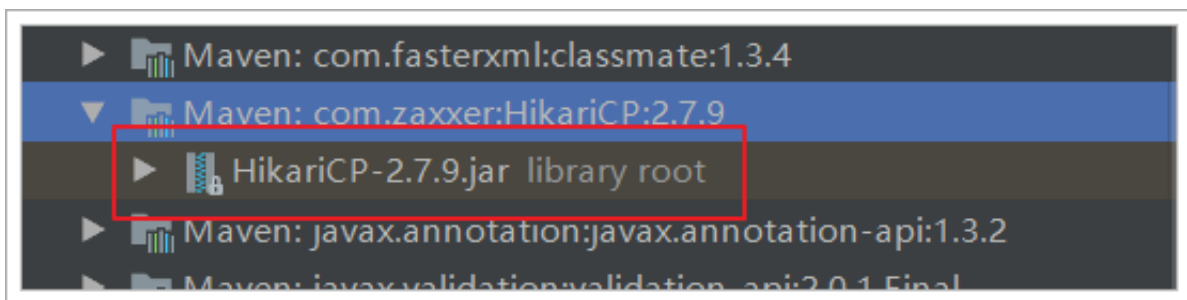
答案是不需要处理，我们只要找到SpringBoot提供的启动器即可：

spring-boot-starter-integration	Starter for using Spring Integration
spring-boot-starter-jdbc	Starter for using JDBC with the HikariCP connection pool
spring-boot-starter-jersey	Starter for building RESTful web applications using JAX-RS

在pom.xml中引入jdbc的启动器：

```
<!-- jdbc的启动器，默认使用HikariCP连接池-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<!-- 不要忘记数据库驱动，因为springboot不知道我们使用的什么数据库，这里选择mysql-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

SpringBoot已经自动帮我们引入了一个连接池：



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：

```
# 连接四大参数
spring.datasource.url=jdbc:mysql://localhost:3306/heima
spring.datasource.username=root
spring.datasource.password=root
# 可省略，SpringBoot自动推断
spring.datasource.driverClassName=com.mysql.jdbc.Driver

spring.datasource.hikari.idle-timeout=60000
spring.datasource.hikari.maximum-pool-size=30
spring.datasource.hikari.minimum-idle=10
```

当然，如果你更喜欢Druid连接池，也可以使用Druid官方提供的启动器：

```
<!-- Druid连接池 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.1.6</version>
</dependency>
```

而连接信息的配置与上面是类似的，只不过在连接池特有属性上，方式略有不同：

```
#初始化连接数
spring.datasource.druid.initial-size=1
#最小空闲连接
spring.datasource.druid.min-idle=1
#最大活动连接
spring.datasource.druid.max-active=20
#获取连接时测试是否可用
spring.datasource.druid.test-on-borrow=true
#监控页面启动
spring.datasource.druid.stat-view-servlet.allow=true
```

## 4.5.整合mybatis

## 4.5.1.mybatis

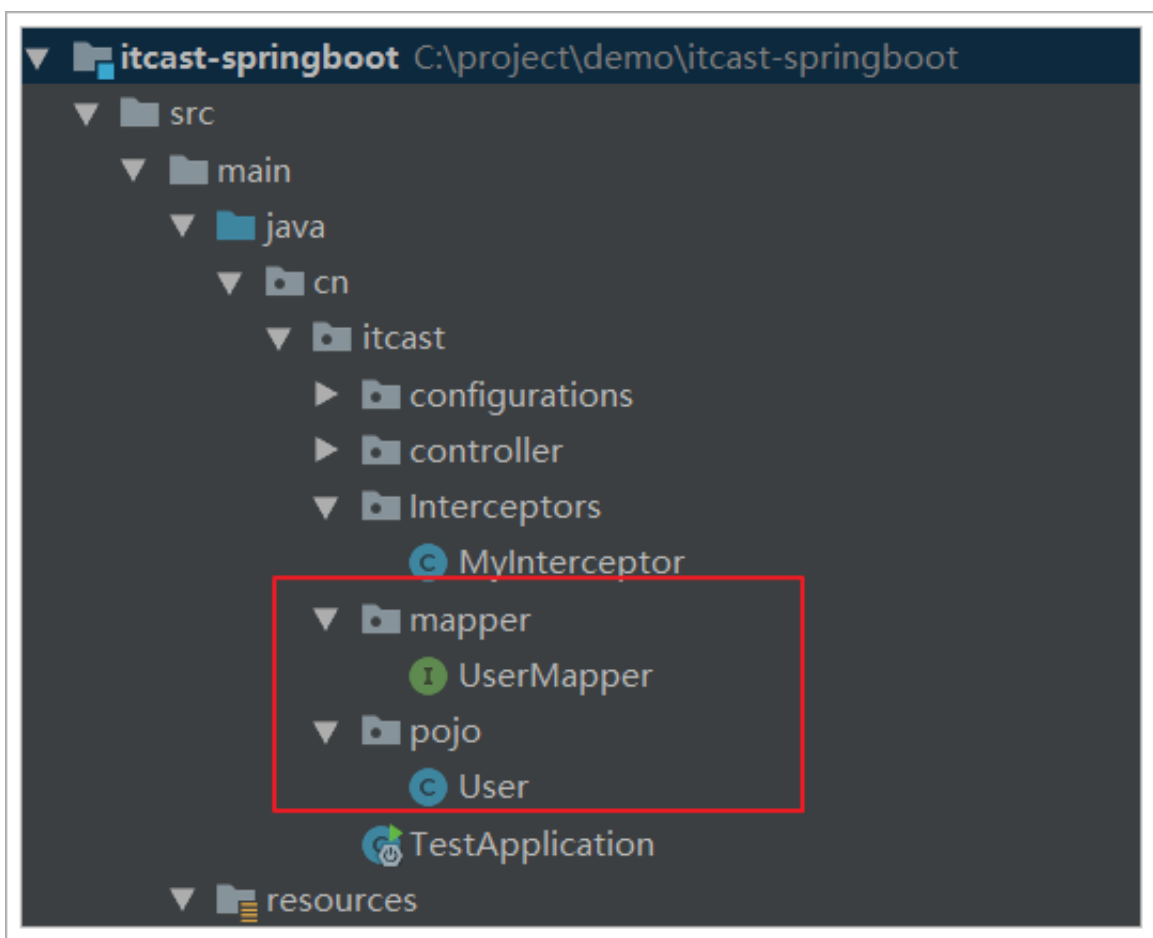
SpringBoot官方并没有提供Mybatis的启动器，不过Mybatis[官方](#)自己实现了：

```
<!--mybatis -->
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>1.3.2</version>
</dependency>
```

配置，基本没有需要配置的：

```
# mybatis 别名扫描
mybatis.type-aliases-package=cn.itcast.pojo
# mapper.xml文件位置,如果没有映射文件,请注释掉
mybatis.mapper-locations=classpath:mappers/*.xml
```

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。



```
@Mapper
public interface UserMapper {
}
```

user对象参照课前资料，需要通用mapper的注解：



名称	修改日期	类型	大小
common.js	2017/3/6 11:05	JS 文件	8 KB
SpringBoot启动器.txt	2018/6/4 23:38	TXT 文件	5 KB
SpringBoot全局属性.md	2018/5/5 14:28	Markdown File	121 KB
User.java	2018/6/5 0:34	JAVA 文件	3 KB

接下来，就去集成通用mapper。

## 4.5.2.通用mapper

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：

```
<!-- 通用mapper -->
<dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper-spring-boot-starter</artifactId>
    <version>2.0.2</version>
</dependency>
```

不需要做任何配置就可以使用了。

```
@Mapper
public interface UserMapper extends tk.mybatis.mapper.common.Mapper<User>{
}
```

## 4.6.整合事务

其实，我们引入jdbc或者web的启动器，就已经引入事务相关的依赖及默认配置了

```
▶ Maven: org.springframework:spring-core:5.0.6.RELEASE
▶ Maven: org.springframework:spring-expression:5.0.6.RELEASE
▶ Maven: org.springframework:spring-jcl:5.0.6.RELEASE
▶ Maven: org.springframework:spring-jdbc:5.0.6.RELEASE
▶ Maven: org.springframework:spring-tx:5.0.6.RELEASE
▶ Maven: org.springframework:spring-web:5.0.6.RELEASE
▶ Maven: org.springframework:spring-webmvc:5.0.6.RELEASE
▶ Maven: org.yaml:snakeyaml:1.19
▶ Maven: tk.mybatis:mapper-base:1.0.1
```

至于事务，SpringBoot中通过注解来控制。就是我们熟知的 `@Transactional`

```
@Service
public class UserService {

    @Autowired
    private UserMapper userMapper;

    public User queryById(Long id){
        return this.userMapper.selectByPrimaryKey(id);
    }

    直接加在方法时即可
    @Transactional
    public void deleteById(Long id){
        this.userMapper.deleteByPrimaryKey(id);
    }
}
```

## 4.7.启动测试

在UserController中添加测试方法，内容：

```
@RestController
@RequestMapping("user")
public class UserController {

    @Autowired
    private UserService userService;

    @GetMapping("{id}")
    public User queryUserById(@PathVariable("id")Long id){
        return this.userService.queryById(id);
    }
}
```

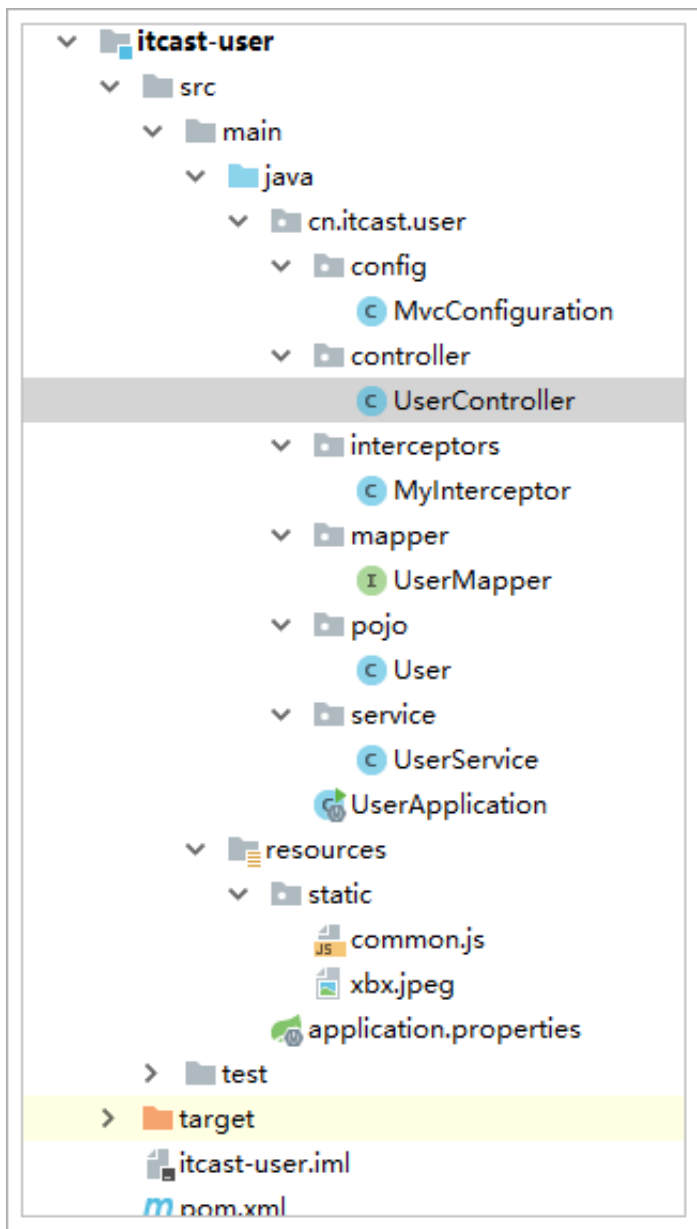
```
}

@GetMapping("hello")
public String test(){
    return "hello ssm";
}
}
```

我们启动项目，查看：



## 4.8.完整项目结构



完整的pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>cn.itcast.user</groupId>
    <artifactId>itcast-user</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.6.RELEASE</version>
```

```

</parent>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <!--jdbc的启动器，默认使用HikariCP连接池-->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
  </dependency>
  <!--不要忘记数据库驱动，因为springboot不知道我们使用的什么数据库，这里选择mysql-->
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>

  <!--mybatis -->
  <dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.2</version>
  </dependency>

  <!-- 通用mapper -->
  <dependency>
    <groupId>tk.mybatis</groupId>
    <artifactId>mapper-spring-boot-starter</artifactId>
    <version>2.0.2</version>
  </dependency>
</dependencies>

</project>

```

完整的application.properties:



```
server.port=80

logging.level.org.springframework=debug

spring.datasource.url=jdbc:mysql://localhost:3306/heima
spring.datasource.username=root
spring.datasource.password=root

# mybatis 别名扫描
mybatis.type-aliases-package=cn.itcast.pojo
# mapper.xml文件位置,如果没有映射文件,请注释掉
# mybatis.mapper-locations=classpath:mappers/*.xml
```

## 5.Thymeleaf快速入门

SpringBoot并不推荐使用jsp，但是支持一些模板引擎技术：

### 27.2.4 Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including **Thymeleaf**, FreeMarker, and Mustache.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- **Thymeleaf**
- Mustache

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

以前大家用的比较多的是Freemarker，但是我们今天的主角是Thymeleaf！

### 5.1.为什么是Thymeleaf?

简单说，Thymeleaf 是一个跟 Velocity、FreeMarker 类似的**模板引擎**，它**可以完全替代 JSP**。相较于其他的模板引擎，它有如下四个极吸引人的特点：

- **动静结合**：Thymeleaf 在有网络和无网络的环境下皆可运行，即它可以让美工在浏览器查看页面的静态效果，也可以让程序员在服务器查看带数据的动态页面效果。这是由于它支持 html 原型，然后在 html 标签里增加额外的属性来达到模板+数据的展示方式。浏览器解释 html 时会忽略未定义的标签属性，所以 thymeleaf 的模板可以静态地运行；当有数据返回到页面时，Thymeleaf 标签会动态地替换掉静态内容，使页面动态显示。
- **开箱即用**：它提供标准和spring标准两种方言，可以直接套用模板实现JSTL、OGNL表达式效果，避免每天套模板、改jstl、改标签的困扰。同时开发人员也可以扩展和创建自定义的方言。
- **多方言支持**：Thymeleaf 提供spring标准方言和一个与 SpringMVC 完美集成的可选模块，可以快

速的实现表单绑定、属性编辑器、国际化等功能。

- 与SpringBoot完美整合，SpringBoot提供了Thymeleaf的默认配置，并且为Thymeleaf设置了视图解析器，我们可以像以前操作jsp一样来操作Thymeleaf。代码几乎没有任何区别，就是在模板语法上有区别。

接下来，我们就通过入门案例来体会Thymeleaf的魅力：

## 5.2.提供数据

编写一个controller方法，返回一些用户数据，放入模型中，将来在页面渲染

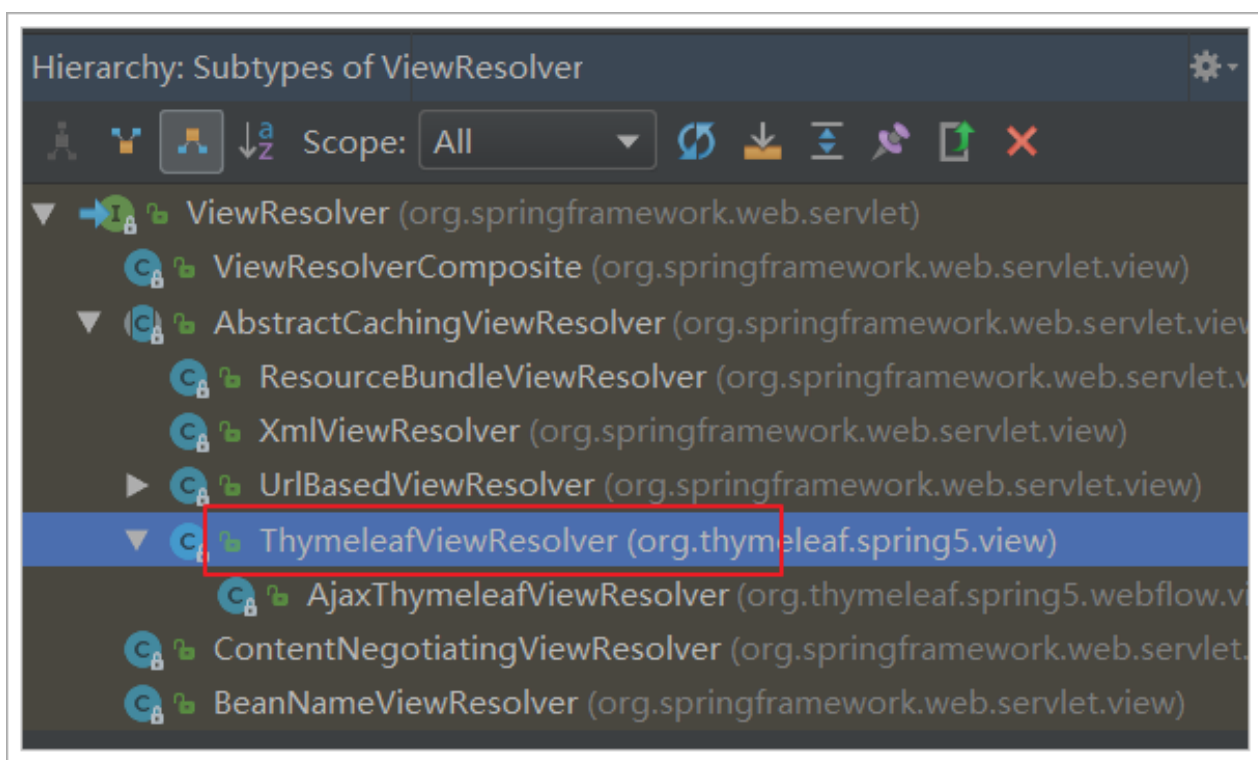
```
@GetMapping("/all")
public String all(ModelMap model) {
    // 查询用户
    List<User> users = this.userService.queryAll();
    // 放入模型
    model.addAttribute("users", users);
    // 返回模板名称（就是classpath:/templates/目录下的html文件名）
    return "users";
}
```

## 5.3.引入启动器

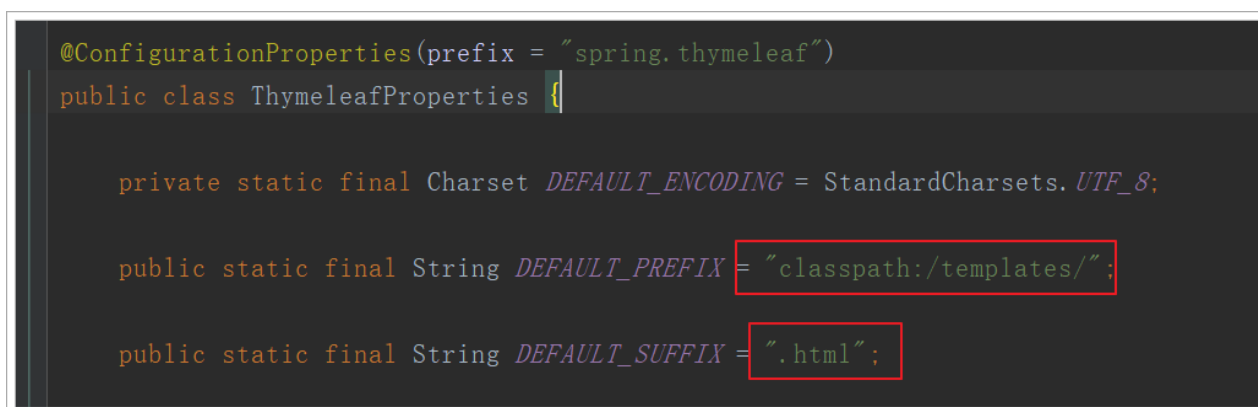
直接引入启动器：

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

SpringBoot会自动为Thymeleaf注册一个视图解析器：



与解析JSP的InternalViewResolver类似，Thymeleaf也会根据前缀和后缀来确定模板文件的位置：



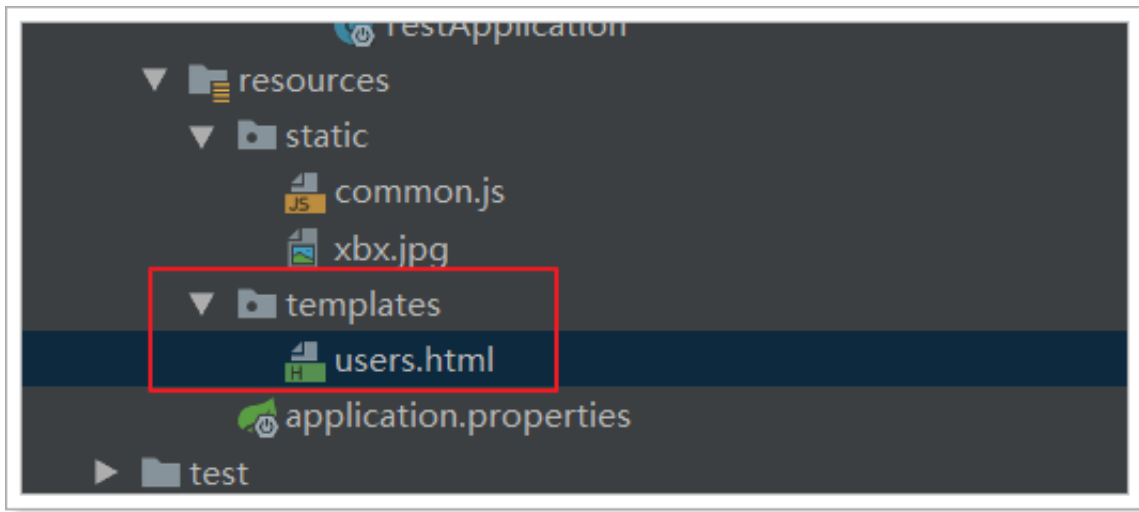
- 默认前缀： `classpath:/templates/`
- 默认后缀： `.html`

所以如果我们返回视图： `users`，会指向到 `classpath:/templates/users.html`

一般我们无需进行修改，默认即可。

## 5.4.静态页面

根据上面的文档介绍，模板默认放在classpath下的templates文件夹，我们新建一个html文件放入其中：



编写html模板，渲染模型中的数据：

注意，把html 的名称空间，改成：xmlns:th="http://www.thymeleaf.org" 会有语法提示

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
  <meta charset="UTF-8">
  <title>首页</title>
  <style type="text/css">
    table {border-collapse: collapse; font-size: 14px; width: 80%; margin: auto}
    table, th, td {border: 1px solid darkslategray;padding: 10px}
  </style>
</head>
<body>
<div style="text-align: center">
  <span style="color: darkslategray; font-size: 30px">欢迎光临! </span>
  <hr/>
  <table class="list">
    <tr>
      <th>id</th>
      <th>姓名</th>
      <th>用户名</th>
      <th>年龄</th>
      <th>性别</th>
      <th>生日</th>
    </tr>
    <tr th:each="user : ${users}">
      <td th:text="${user.id}">1</td>
      <td th:text="${user.name}">张三</td>
      <td th:text="${user.userName}">zhangsan</td>
      <td th:text="${user.age}">20</td>
      <td th:text="${user.sex}">男</td>
      <td th:text="${user.birthday}">1980-02-30</td>
    </tr>
  </table>
```


```
</div>
</body>
</html>
```

我们看到这里使用了以下语法：

- `${}`：这个类似与el表达式，但其实是ognl的语法，比el表达式更加强大
- `th-` 指令：`th-` 是利用了Html5中的自定义属性来实现的。如果不支持H5，可以用 `data-th-` 来代替
  - `th:each`：类似于 `c:foreach` 遍历集合，但是语法更加简洁
  - `th:text`：声明标签中的文本
    - 例如 `<td th-text='${user.id}'>1</td>`，如果user.id有值，会覆盖默认的1
    - 如果没有值，则会显示td中默认的1。这正是thymeleaf能够动静结合的原因，模板解析失败不影响页面的显示效果，因为会显示默认值！

## 5.5.测试

接下来，我们打开页面测试一下：



id	姓名	用户名	年龄	性别	生日
1	张三	zhangsan	30	男	1984-08-08
2	李四	lisi	21	女	1991-01-01
3	王五	wangwu	22	女	1989-01-01
4	张伟	zhangwei	20	男	1988-09-01
5	李娜	lina	28	男	1985-01-01
6	李磊	lilei	23	男	1988-08-08

## 5.6.模板缓存

Thymeleaf会在第一次对模板解析之后进行缓存，极大的提高了并发处理能力。但是这给我们开发带来了不便，修改页面后并不会立刻看到效果，我们开发阶段可以关掉缓存使用：

```
# 开发阶段关闭thymeleaf的模板缓存
spring.thymeleaf.cache=false
```

注意：

在Idea中，我们需要在修改页面后按快捷键：`Ctrl + Shift + F9` 对项目进行rebuild才可以。

eclipse中没有测试过。

我们可以修改页面，测试一下。