

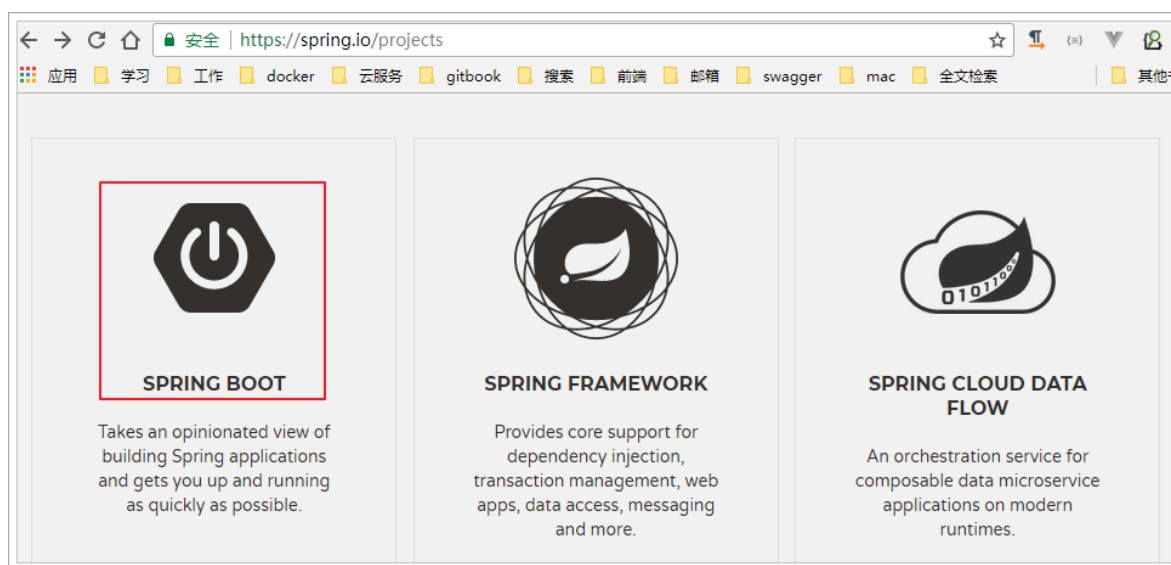
1. 了解SpringBoot

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：



我们可以看到下面的一段介绍：

Takes an opinionated view of building production-ready Spring applications. Spring Boot favors convention over configuration and is designed to get you up and running as quickly as possible.

翻译一下：

用一些固定的方式来构建生产级别的spring应用。Spring Boot 推崇约定大于配置的方式以便于你能够尽可能快速的启动并运行程序。

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，让我们关注与业务而非配置。

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SpringBoot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。

1.3.SpringBoot的特点

Spring Boot 主要目标是：

- 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
- 开箱即用（启动器starter-其实就是SpringBoot提供的一个jar包），但通过自己设置参数（.properties），即可快速摆脱这种方式。
- 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。

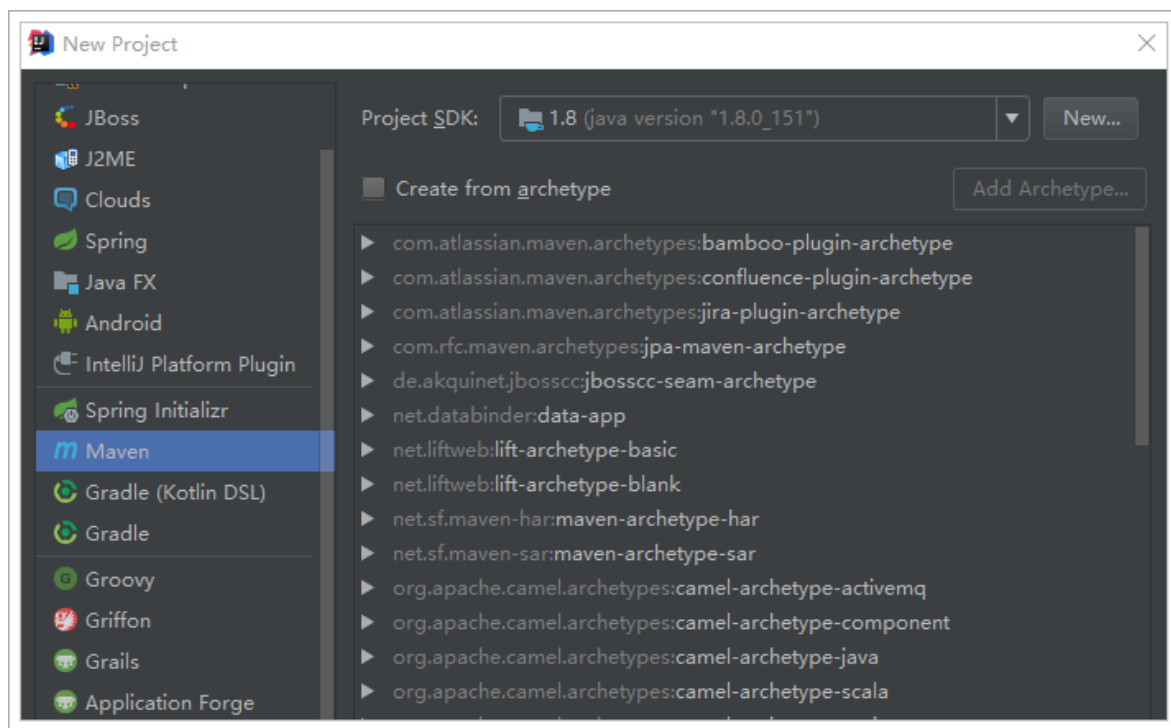
更多细节，大家可以到[官网](#)查看。

2.快速入门

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

2.1.创建工程

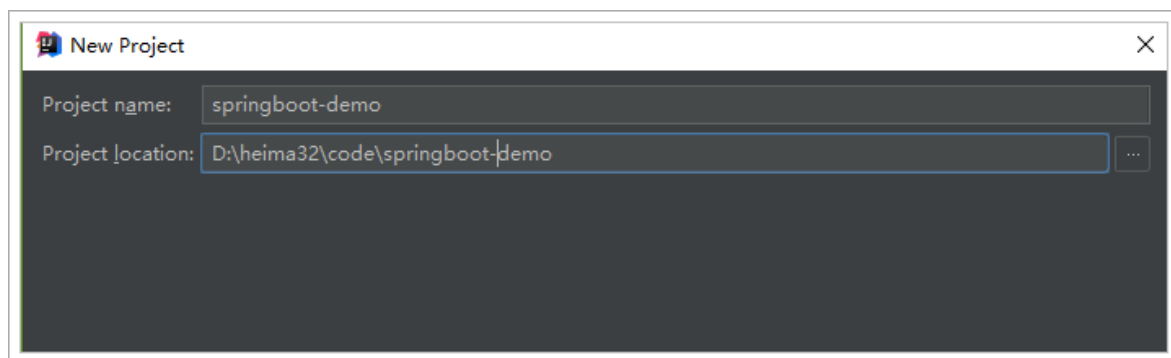
我们先新建一个空的工程：



项目信息：



地址：



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！

2.2.1.添加父工程坐标

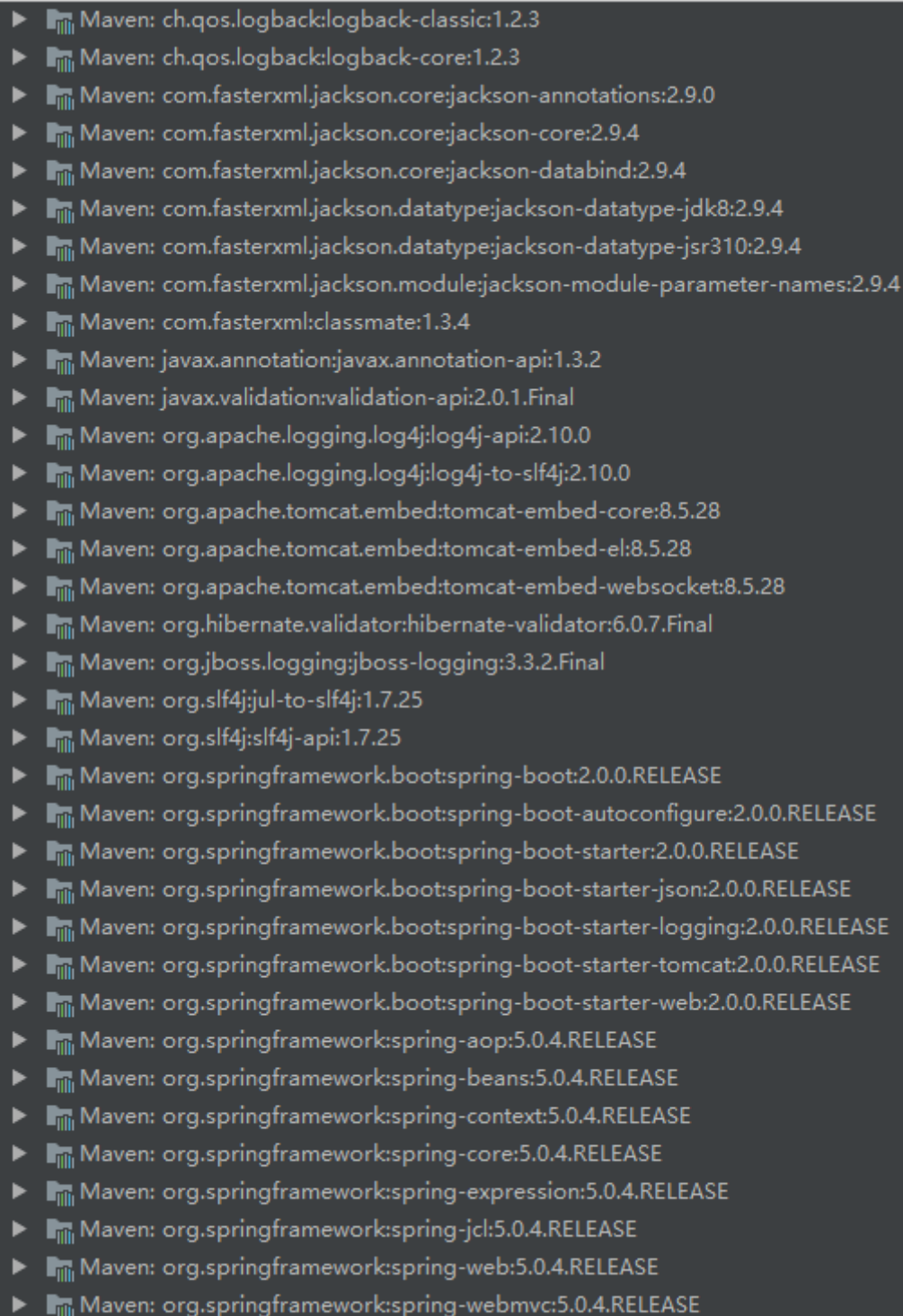
```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.3.RELEASE</version>
</parent>
```

2.2.2.添加web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**。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：



```
▶ Maven: ch.qos.logback:logback-classic:1.2.3
▶ Maven: ch.qos.logback:logback-core:1.2.3
▶ Maven: com.fasterxml.jackson.core;jackson-annotations:2.9.0
▶ Maven: com.fasterxml.jackson.core;jackson-core:2.9.4
▶ Maven: com.fasterxml.jackson.core;jackson-databind:2.9.4
▶ Maven: com.fasterxml.jackson.datatype;jackson-datatype-jdk8:2.9.4
▶ Maven: com.fasterxml.jackson.datatype;jackson-datatype-jsr310:2.9.4
▶ Maven: com.fasterxml.jackson.module;jackson-module-parameter-names:2.9.4
▶ Maven: com.fasterxml:classmate:1.3.4
▶ Maven: javax.annotation:javax.annotation-api:1.3.2
▶ Maven: javax.validation:validation-api:2.0.1.Final
▶ Maven: org.apache.logging.log4j:log4j-api:2.10.0
▶ Maven: org.apache.logging.log4j:log4j-to-slf4j:2.10.0
▶ Maven: org.apache.tomcat.embed:tomcat-embed-core:8.5.28
▶ Maven: org.apache.tomcat.embed:tomcat-embed-el:8.5.28
▶ Maven: org.apache.tomcat.embed:tomcat-embed-websocket:8.5.28
▶ Maven: org.hibernate.validator:hibernate-validator:6.0.7.Final
▶ Maven: org.jboss.logging:jboss-logging:3.3.2.Final
▶ Maven: org.slf4j:jul-to-slf4j:1.7.25
▶ Maven: org.slf4j:slf4j-api:1.7.25
▶ Maven: org.springframework.boot:spring-boot:2.0.0.RELEASE
▶ Maven: org.springframework.boot:spring-boot-autoconfigure:2.0.0.RELEASE
▶ Maven: org.springframework.boot:spring-boot-starter:2.0.0.RELEASE
▶ Maven: org.springframework.boot:spring-boot-starter-json:2.0.0.RELEASE
▶ Maven: org.springframework.boot:spring-boot-starter-logging:2.0.0.RELEASE
▶ Maven: org.springframework.boot:spring-boot-starter-tomcat:2.0.0.RELEASE
▶ Maven: org.springframework.boot:spring-boot-starter-web:2.0.0.RELEASE
▶ Maven: org.springframework:spring-aop:5.0.4.RELEASE
▶ Maven: org.springframework:spring-beans:5.0.4.RELEASE
▶ Maven: org.springframework:spring-context:5.0.4.RELEASE
▶ Maven: org.springframework:spring-core:5.0.4.RELEASE
▶ Maven: org.springframework:spring-expression:5.0.4.RELEASE
▶ Maven: org.springframework:spring-jcl:5.0.4.RELEASE
▶ Maven: org.springframework:spring-web:5.0.4.RELEASE
▶ Maven: org.springframework:spring-webmvc:5.0.4.RELEASE
```

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。

2.2.3.管理jdk版本

如果我们想要修改SpringBoot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。

```
<properties>
    <java.version>1.8</java.version>
</properties>
```

2.2.4.完整pom

```
<?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>com.leyou.demo</groupId>
  <artifactId>springboot-demo</artifactId>
  <version>1.0-SNAPSHOT</version>

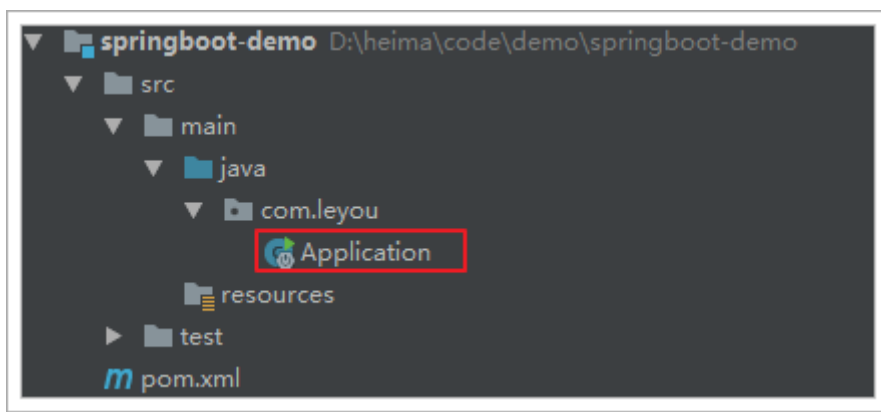
  <properties>
    <java.version>1.8</java.version>
  </properties>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
  </parent>

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



然后编写main函数：

```
package com.leyou;

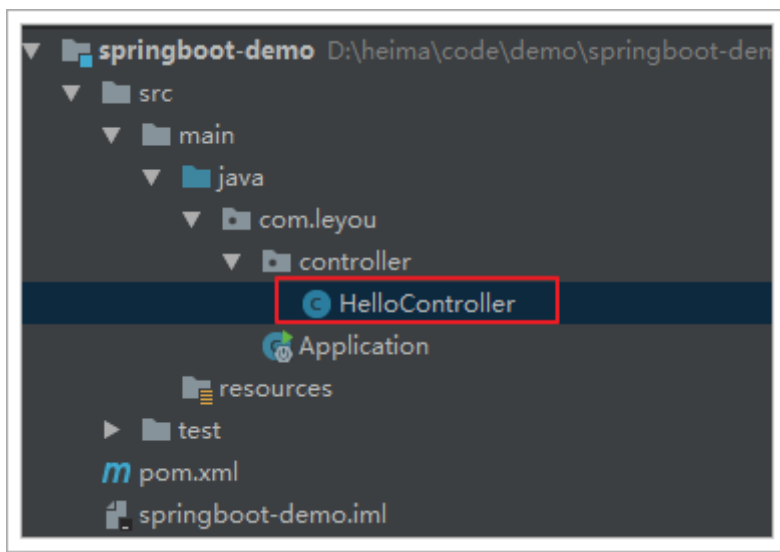
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

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

2.4.编写controller

接下来，我们就可以像以前那样开发SpringMVC的项目了！

我们编写一个controller：



代码：

```
package com.leyou.controller;

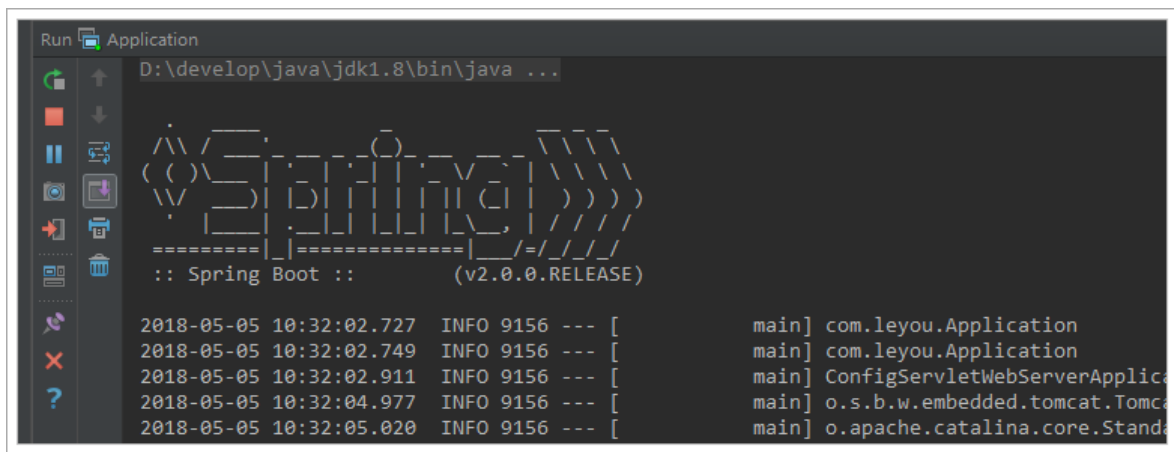
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class HelloController {

    @GetMapping("hello")
    public String hello(){
        return "hello, spring boot!";
    }
}
```

2.5.启动测试

接下来，我们运行main函数，查看控制台：

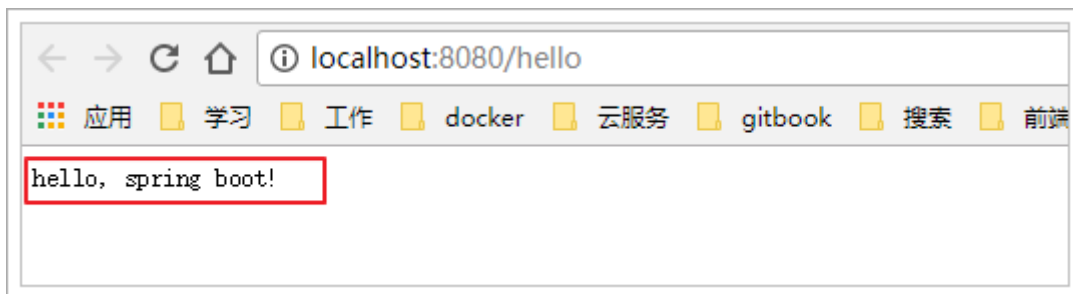


并且可以看到监听的端口信息：

```
com.leyou.Application : Starting Application on DESKTOP-8V9MV76 with PID 2068 (
com.leyou.Application : No active profile set, falling back to default profiles
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat initialized with port(s): 8080 (http) ①
o.apache.catalina.core.StandardService : Starting service [Tomcat]
org.apache.catalina.core.StandardEngine : Starting Servlet engine: [Apache Tomcat/9.0.16]
o.a.catalina.core.AprLifecycleListener : The APR based Apache Tomcat Native library which allows
o.a.c.c.C.[Tomcat] [localhost]. [/] ②
o.s.web.context.ContextLoader : Root WebApplicationContext: initialization completed in
o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 8080 (http) with context pat
com.leyou.Application : Started Application in 1.92 seconds (JVM running for 3.
```

- 1) 监听的端口是8080
- 2) SpringMVC的映射路径是：/

打开页面访问：<http://localhost:8080/hello>

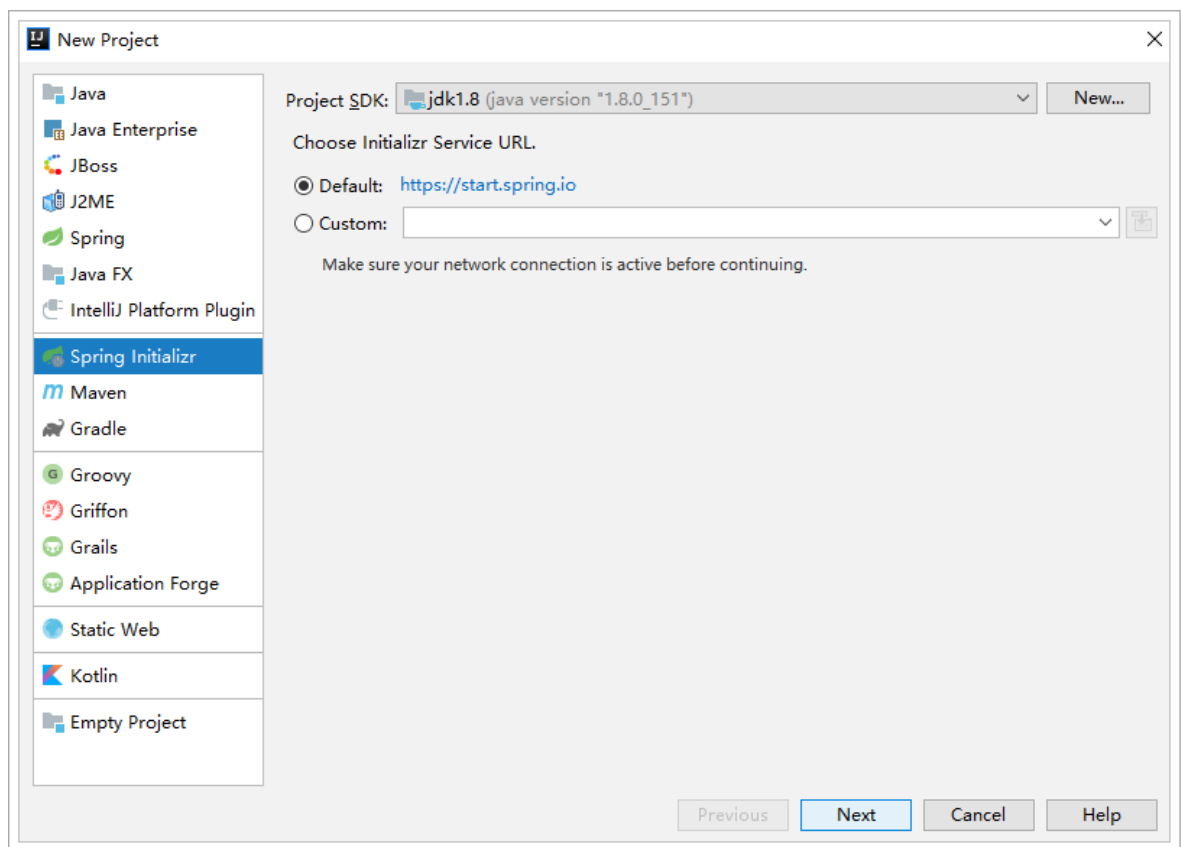


测试成功了！

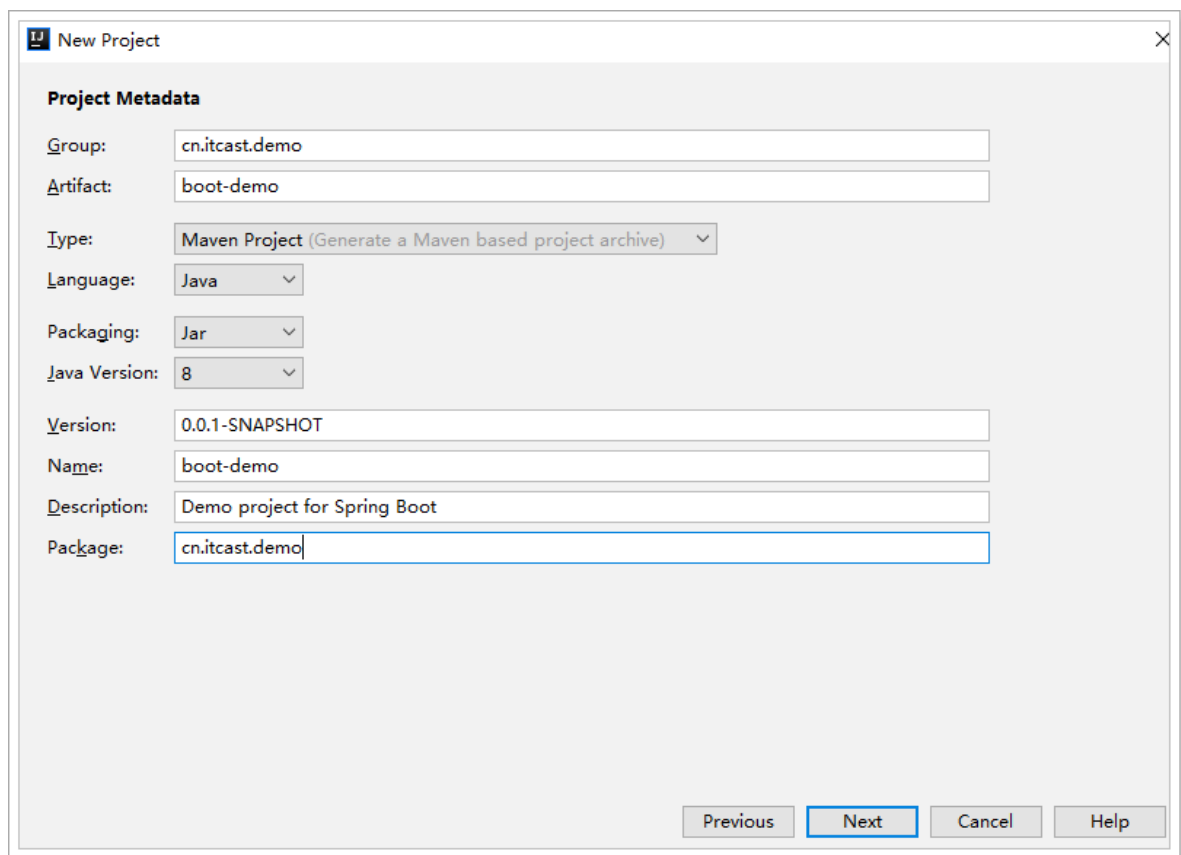
2.6. SpringBoot的项目模板

自己搭建SpringBoot项目需要编写启动类、pom、测试类等内容，虽然已经很简洁，但是也会消耗时间。所以Spring提供了一套快速搭建项目的脚手架，非常方便：

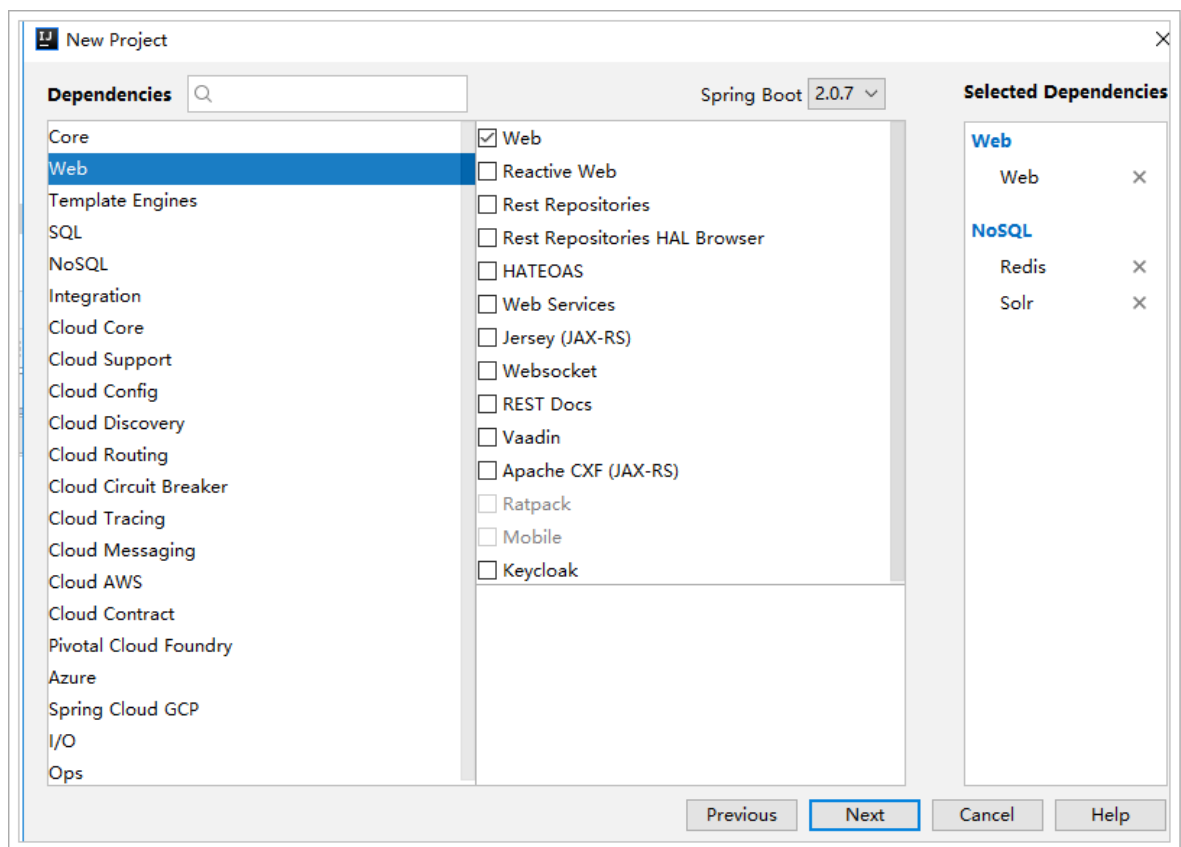
- 1) 创建新工程，选择spring方式：



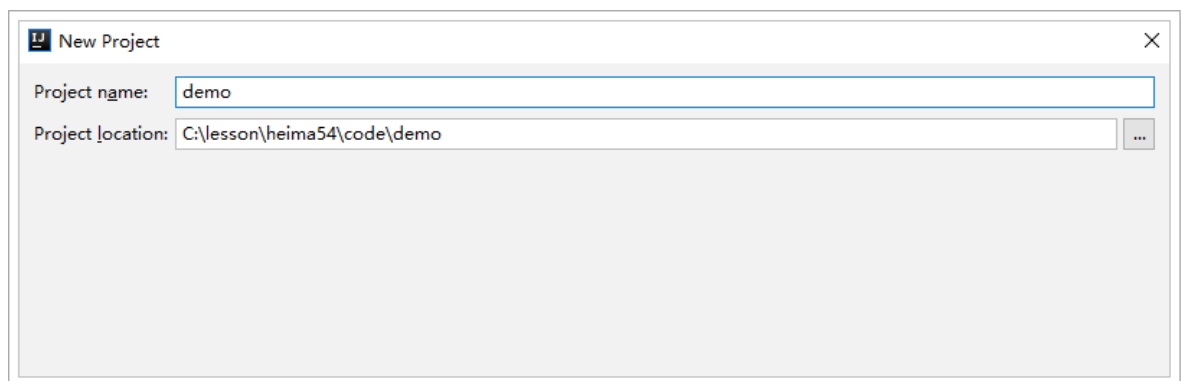
2) 编写项目信息



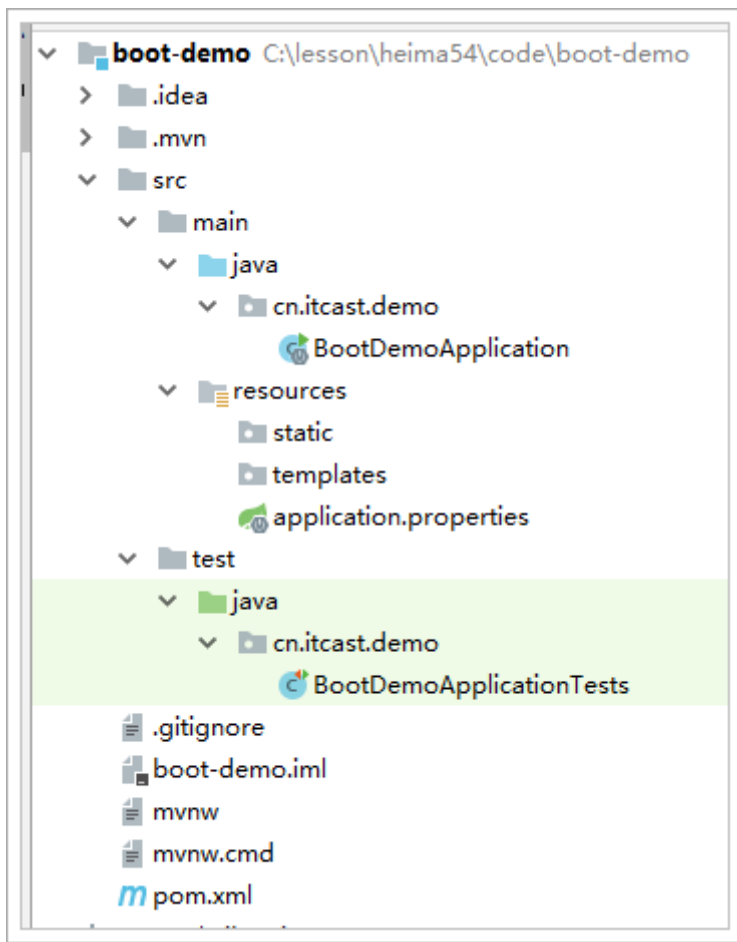
3) 勾选所需依赖



4) 填写项目路径:



这样就可以得到完整项目，并且依赖、配置、启动类、测试类等都已经自动完成：



pom文件:

```
<?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>
  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.1.3.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
  <groupId>cn.itcast.demo</groupId>
  <artifactId>boot-demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>boot-demo</name>
  <description>Demo project for Spring Boot</description>

  <properties>
    <java.version>1.8</java.version>
  </properties>

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

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

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

<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>

</project>
```

(学习阶段不推荐使用这种方式创建项目)

3.Java配置

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？

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配置的优秀。

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.spring纯注解基本知识

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

需求：定义一个简单的User类，定义配置文件，User对象中的属性值从配置文件中获取

User类如下：

在com.leyou.pojo包下创建一个User类

```
package com.leyou.pojo;

public class User {
    private String username;
    private String password;
    private Integer age;
    setter...getter...
}
```

3.3 @Value方式

第一步：在resources下创建一个user.properties文件，里面的内容如下：

```
user.username=zhangsan
user.password=123456
user.age=18
```

第二步：在cn.itcast.config包下创建一个配置类，并使用@Configuration声明是一个配置类，在配置类中创建User对象

```
package com.leyou.config;

import com.leyou.pojo.User;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;

@Configuration //声明当前类是一个配置类
@PropertySource("classpath:user.properties") //加载配置文件
public class MyConfig {

    @Value("${user.username}") //使用@Value注解获取值
    private String username;

    @Value("${user.password}") //使用@Value注解获取值
    private String password;
```

```

@Value("${user.age}") //使用@Value注解获取值
private Integer age;

@Bean //创建User对象，交给spring容器 User对象中的值从配置文件中获取
public User createUser(){
    User user = new User();
    user.setUsername(username);
    user.setPassword(password);
    user.setAge(age);
    return user;
}
}

```

第三步：测试

在HelloController中添加一个方法，验证User中是否有值

```

@RestController
public class HelloController {

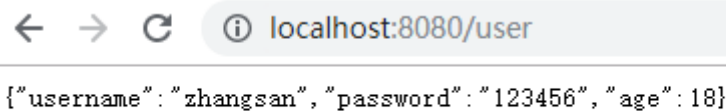
    @GetMapping("hello")
    public String hello(){
        return "hello, spring boot!";
    }

    @Autowired
    private User user;

    @GetMapping("user")
    public User user(){
        return user;
    }
}

```

浏览器测试：成功



localhost:8080/user

{\"username\": \"zhangsan\", \"password\": \"123456\", \"age\": 18}

3.4 Environment获取数据方式

spring中的Environment用来表示整个应用运行时的环境，可以使用Environment获取整个运行环境中的配置信息：方法是：environment.getProperty（配置文件中的key），返回的一律都是字符串，可以根据需要转换。

```

@Configuration
@PropertySource("classpath:user.properties")
public class MyConfig {

```

```

@Autowired
private Environment environment; //注意是spring包下的
@Bean
public User createUser(){
    User user = new User();
    user.setUsername(environment.getProperty("user.username"));
    user.setPassword(environment.getProperty("user.password"));
    user.setAge(Integer.parseInt(environment.getProperty("user.age")));
    return user;
}
}

```

3.5 @ConfigurationProperties方式

spring约定的、非常简洁的配置方式

首先约定，配置信息需要写在一个application.properties的文件中

第一步：在resources中创建一个application.properties文件，文件内容如下(和user.properties中的内容一样)：

```

user.username=zhangsan
user.password=123456
user.age=18

```

第二步：修改配置类如下：

```

@Configuration
public class MyConfig {
    @Bean
    @ConfigurationProperties(prefix = "user") //前缀
    public User createUser(){
        User user = new User();
        return user;
    }
}

```

第三步：测试仍然可以获取数据，很神奇！

← → ↻ ⓘ localhost:8080/user

```
{"username": "zhangsan", "password": "123456", "age": 18}
```

分析：要想简单的获取值，条件是User类中的属性名称和配置文件中的key名称是要一致的！！

3.6 SpringBoot支持的配置文件说明

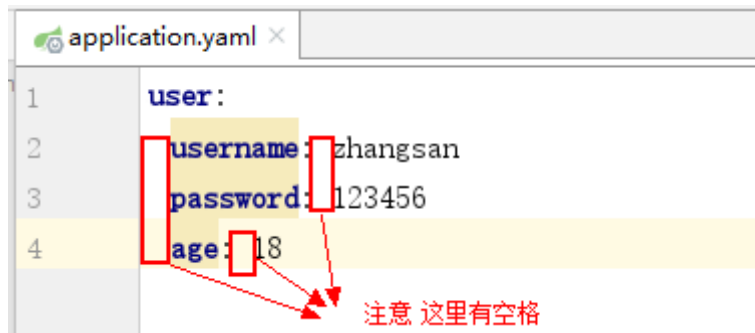
3.6.1 yaml&yml

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

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。

yml和yaml基本格式是一样的：

```
user:
  username: zhangsan
  password: 123456
  age: 18
```



yml 和yaml 比properties 强大的是可以在配置文件中定义一个数组或集合

举例如下：

第一步：在User类中添加3个属性，

```
public class User {
    private String username;
    private String password;
    private Integer age;
    private List<String> girlNames; //一个字符串集合
    private String[] girls; //一个数组
    private List<User> girlList; // 一个集合
    setter....gette....
}
```

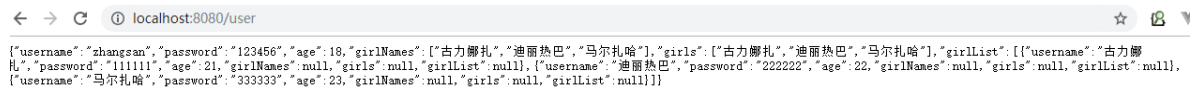
第二步：在application.yml或application.yaml中添加如下配置

```
user:
  username: zhangsan
  password: 123456
  age: 18
girl:
  username: 铁锤妹妹
  password: 111111
  age: 20
girlNames:
  - 古力娜扎
  - 迪丽热巴
```



```
- 马尔扎哈
girls:
- 古力娜扎
- 迪丽热巴
- 马尔扎哈
girlList[0]:
  username: 古力娜扎
  password: 111111
  age: 21
girlList[1]:
  username: 迪丽热巴
  password: 222222
  age: 22
girlList[2]:
  username: 马尔扎哈
  password: 333333
  age: 23
```

第三步：测试



```
{
  "username": "zhangsan",
  "password": "123456",
  "age": 18,
  "girlNames": ["古力娜扎", "迪丽热巴", "马尔扎哈"],
  "girls": ["古力娜扎", "迪丽热巴", "马尔扎哈"],
  "girlList": [
    {
      "username": "古力娜扎",
      "password": "111111",
      "age": 21,
      "girlNames": null,
      "girls": null,
      "girlList": null
    },
    {
      "username": "迪丽热巴",
      "password": "222222",
      "age": 22,
      "girlNames": null,
      "girls": null,
      "girlList": null
    },
    {
      "username": "马尔扎哈",
      "password": "333333",
      "age": 23,
      "girlNames": null,
      "girls": null,
      "girlList": null
    }
  ]
}
```

3.6.2 优先级

在项目中其实只出现一种配置文件就可以了，但是如果真的有properties、yaml、yml三种配置文件时，那它们被加载的优先级是：

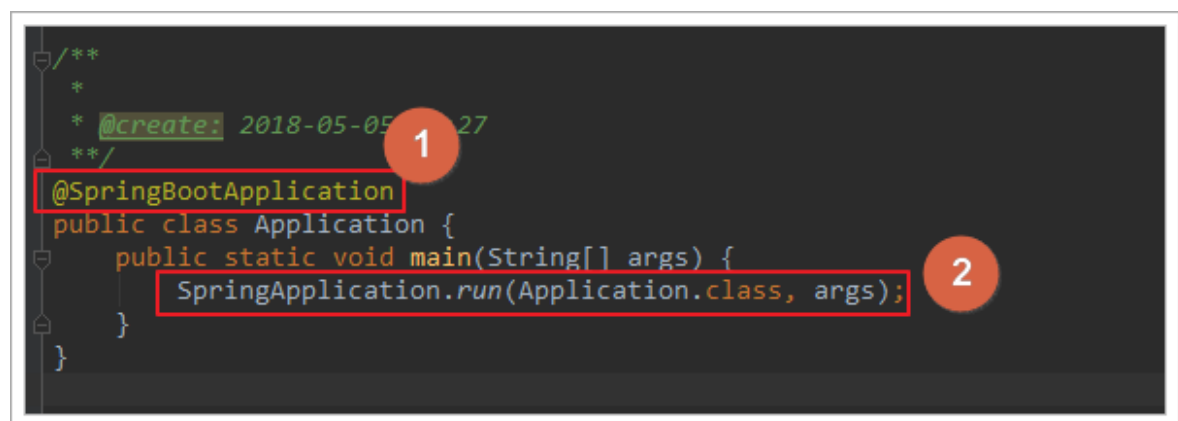
properties > yml > yaml

自己可以在三种配置文件中添加一样的属性，看哪个在生效。（自己测试！）

4.自动配置原理

使用SpringBoot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：



```
/**
 *
 * @create: 2018-05-05 27
 */
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

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} and
 * 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。

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SpringBoot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，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 {

    /**
     * Alias for {@link #basePackages}.
     * <p>Allows for more concise annotation declarations if no other attributes
     * are needed &mdash; for example, {@code @ComponentScan("org.my.pkg")}
     * instead of {@code @ComponentScan(basePackages = "org.my.pkg")}.
     */
    @AliasFor("basePackages")
    String[] value() default {};
```

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：

```
/unchecked, rawtypes/  
public SpringApplication(ResourceLoader resourceLoader, Class<?>... primarySources) {  
    this.resourceLoader = resourceLoader;  
    Assert.notNull(primarySources, "PrimarySources must not be null");  
    this.primarySources = new LinkedHashSet<>(Arrays.asList(primarySources));  
    this.webApplicationType = deduceWebApplicationType();  
    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 = Thread.currentThread().getContextClassLoader();  
    // 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：

```
* @throws IllegalArgumentException if an error occurs while loading factory names  
*/  
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 ?  
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :  
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));  
        result = new LinkedMultiValueMap<>();  
    }  
}
```

通过类加载器加载某个文件，读取内容

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：

```

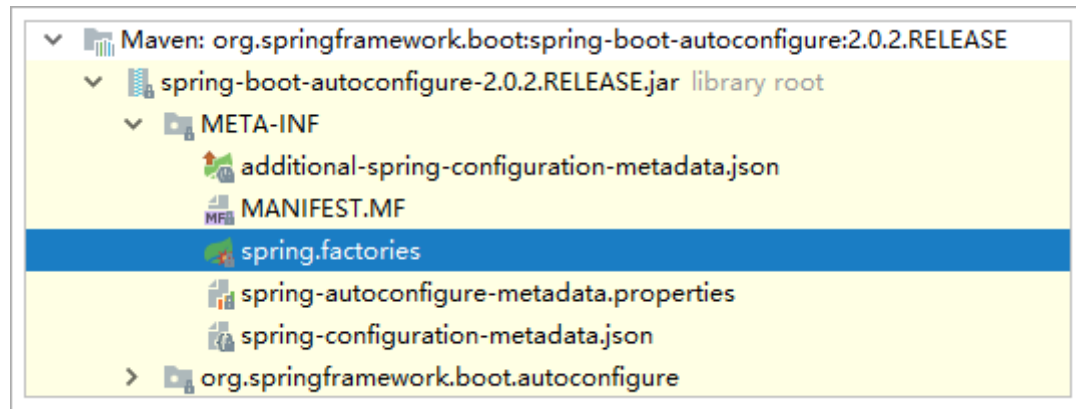
public abstract class SpringFactoriesLoader {

    /**
     * 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。

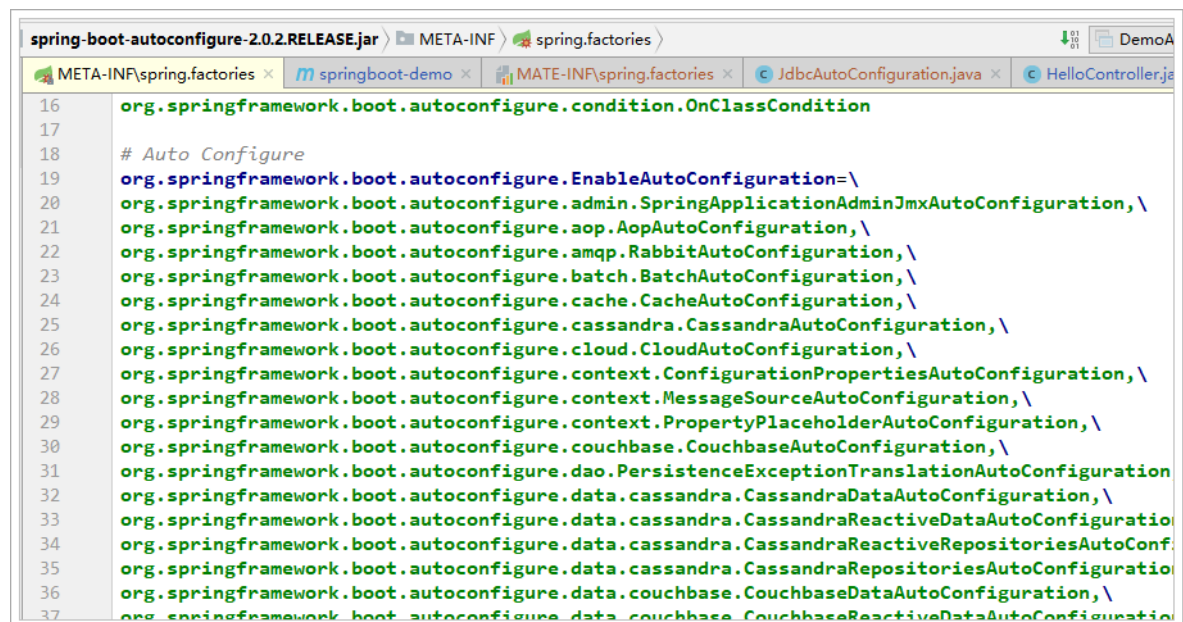
而在Spring的一个依赖包：spring-boot-autoconfigure中，就有这样的文件：



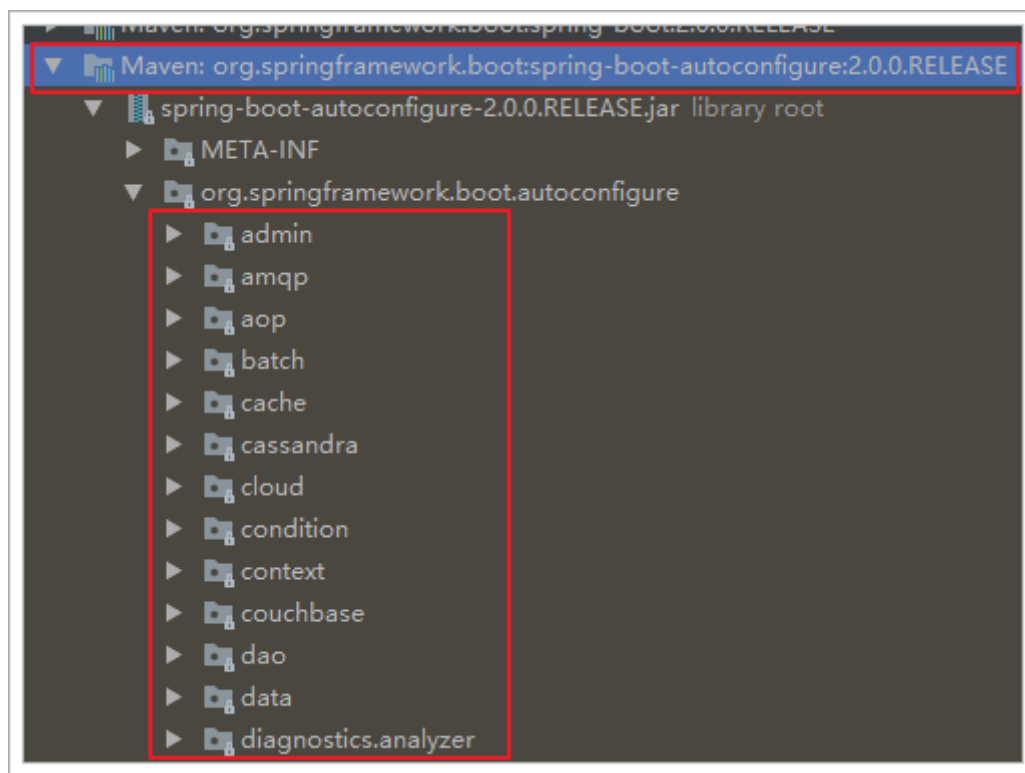
以后我们引入的任何第三方启动器，只要实现自动配置，也都会有类似文件。

4.2.1 默认配置类

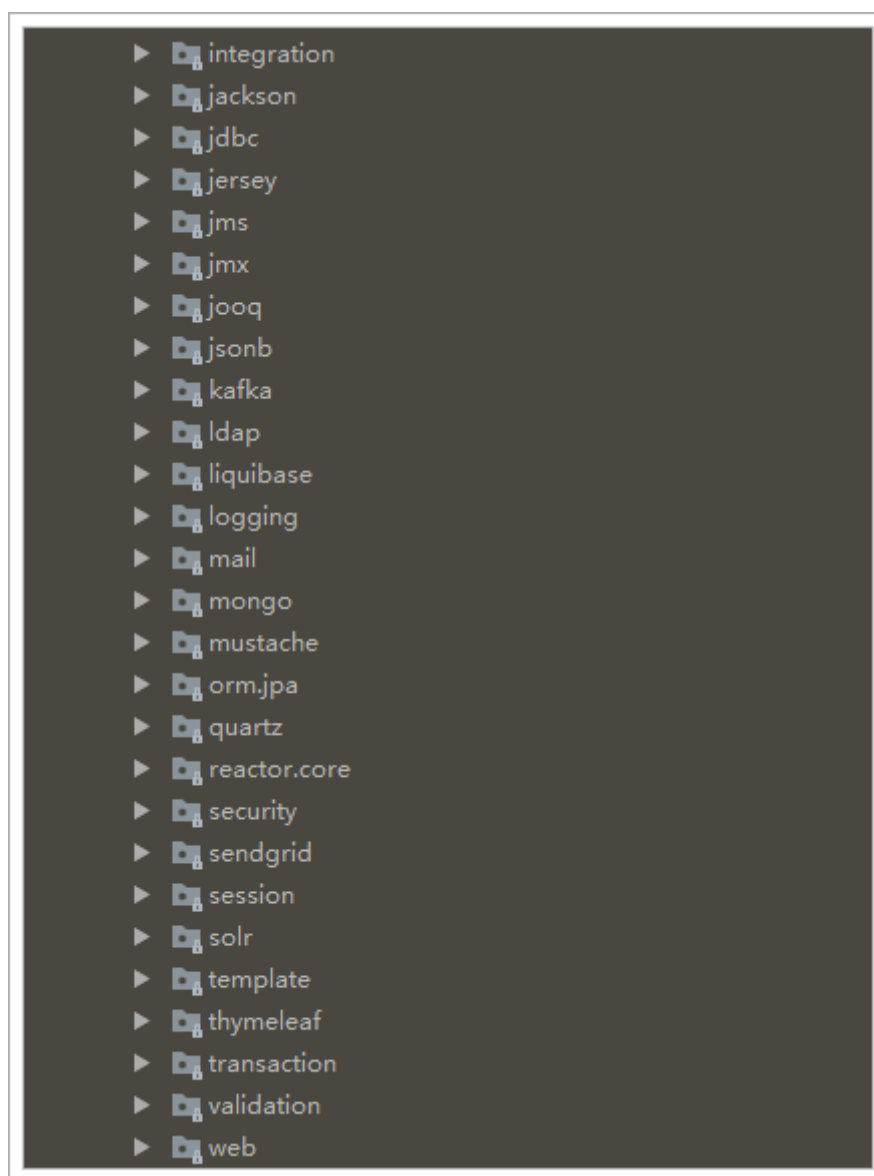
我们打开刚才的spring.factories文件：



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：



还有：

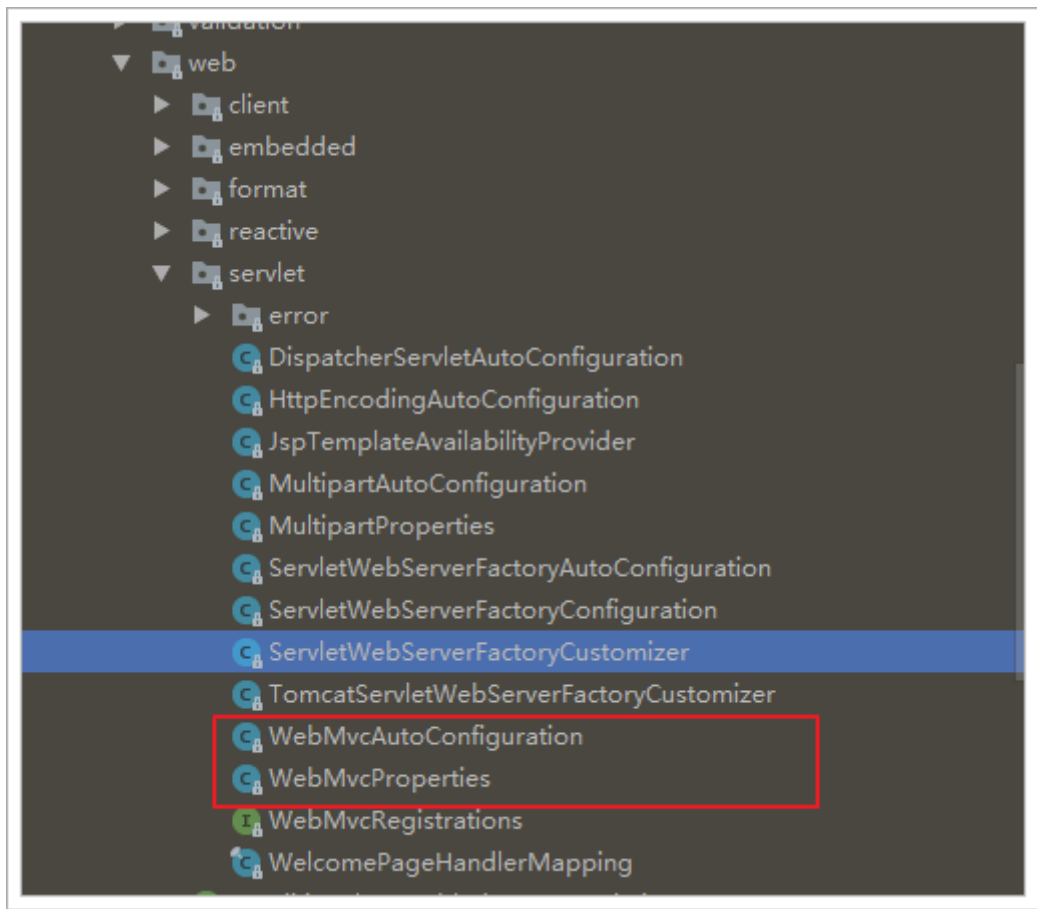


非常多，几乎涵盖了现在主流的开源框架，例如：

- 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：


```

* @author Phillip Webb
* @author Dave Syer
* @author Andy Wilkinson
* @author Sébastien Deleuze
* @author Eddú Meléndez
* @author Stephane Nicoll
* @author Kristine Jetzke
* @author Bruce Brouwer
*/
@Configuration
@ConditionalOnWebApplication(type = Type.SERVLET)
@ConditionalOnClass({ Servlet.class, DispatcherServlet.class, WebMvcConfigurer.class })
@ConditionalOnMissingBean(WebMvcConfigurationSupport.class)
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE + 10)
@AutoConfigureAfter({ DispatcherServletAutoConfiguration.class,
    ValidationAutoConfiguration.class })
public class WebMvcAutoConfiguration {

    public static final String DEFAULT_PREFIX = "";

    public static final String DEFAULT_SUFFIX = "";

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

    @Bean

```

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`WebMVConfigurationSupport`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：

```

@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;
}

```

处理器适配器 (HandlerAdapter)：


```

@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
@Primary
@Override
public RequestMappingHandlerMapping requestMappingHandlerMapping() {
    // Must be @Primary for MvcUriComponentsBuilder to work
    return super.requestMappingHandlerMapping();
}

```

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？

```

// 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 })
public static class WebMvcAutoConfigurationAdapter
    implements WebMvcConfigurer, ResourceLoaderAware {

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

    private final ResourceProperties resourceProperties;

    private final WebMvcProperties mvcProperties;

```

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：

```

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

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

public String getPrefix() { return this.prefix; }

public void setPrefix(String prefix) { this.prefix = prefix; }

public String getSuffix() { return this.suffix; }

public void setSuffix(String suffix) { this.suffix = 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:

```

@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/" };
}

```

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。完成自动配置
- 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，那么会覆盖默认自动配置的Bean
- 我们还可以通过配置application.properties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

5.SpringBoot实践

接下来，我们来看看如何用SpringBoot来玩转以前的SSM,我们会在数据库引入一张用户表tb_user和实体类User

tb_user:

详见资料中：《tb_user.sql》文件。

就在刚才的springboot-demo工程中做开发就可以！

user实体类：放入到com.leyou.pojo包下

```
package com.leyou.pojo;

import lombok.Data;
import java.util.Date;

@Data //注意要记得添加lombok的依赖
public class User{
    // id
    private Long id;
    // 用户名
    private String userName;
    // 密码
    private String password;
    // 姓名
    private String name;
    // 年龄
    private Integer age;
    // 性别, 1男性, 2女性
    private Integer sex;
    // 出生日期
    private Date birthday;
    // 创建时间
    private Date created;
    // 更新时间
    private Date updated;
    // 备注
    private String note;
}
```

5.1.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。

5.1.0.日志控制

日志级别控制：

```
logging:
  level:
    com.leyou: debug
    org.springframework: debug
```

其中:

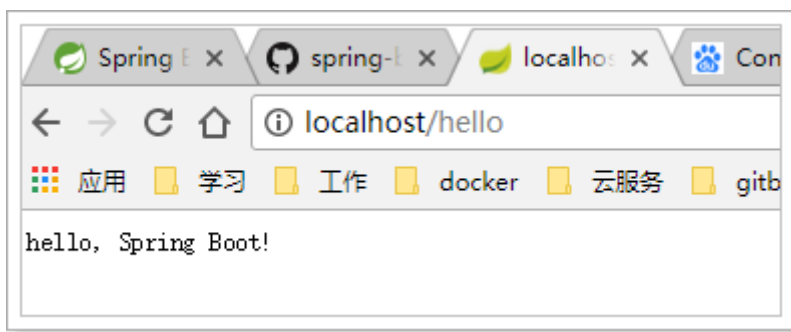
- logging.level: 是固定写法, 说明下面是日志级别配置, 日志相关其它配置也可以使用。
- com.leyou和org.springframework是指定包名, 后面的配置仅对这个包有效。
- debug: 日志的级别

5.1.1.修改端口

查看SpringBoot的全局属性可知, 端口通过以下方式配置:

```
# 映射端口
server.port=80
```

重启服务后测试:



5.1.2.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:

```
@ConfigurationProperties(prefix = "spring.resources", ignoreUnknownFields = false)
public class ResourceProperties {

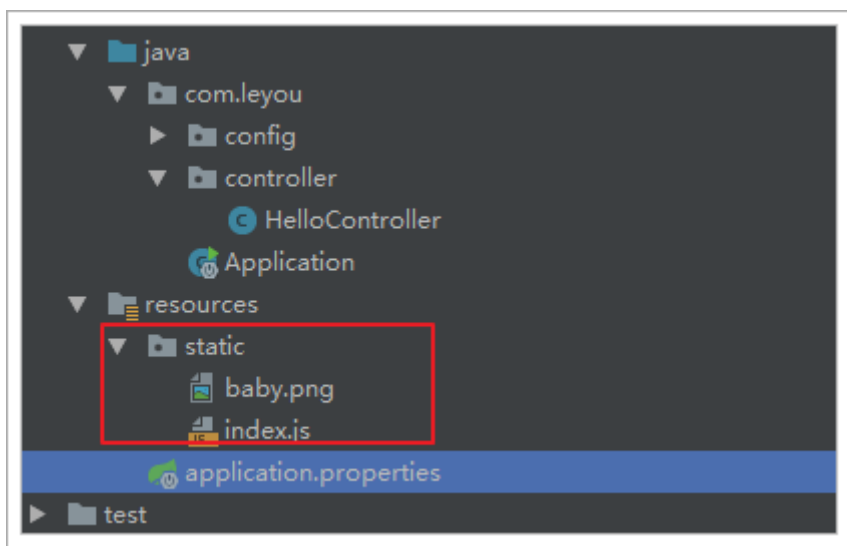
    private static final String[] CLASSPATH_RESOURCE_LOCATIONS = {
        "classpath:/META-INF/resources/", "classpath:/resources/",
        "classpath:/static/", "classpath:/public/" };
}
```

默认的静态资源路径为:

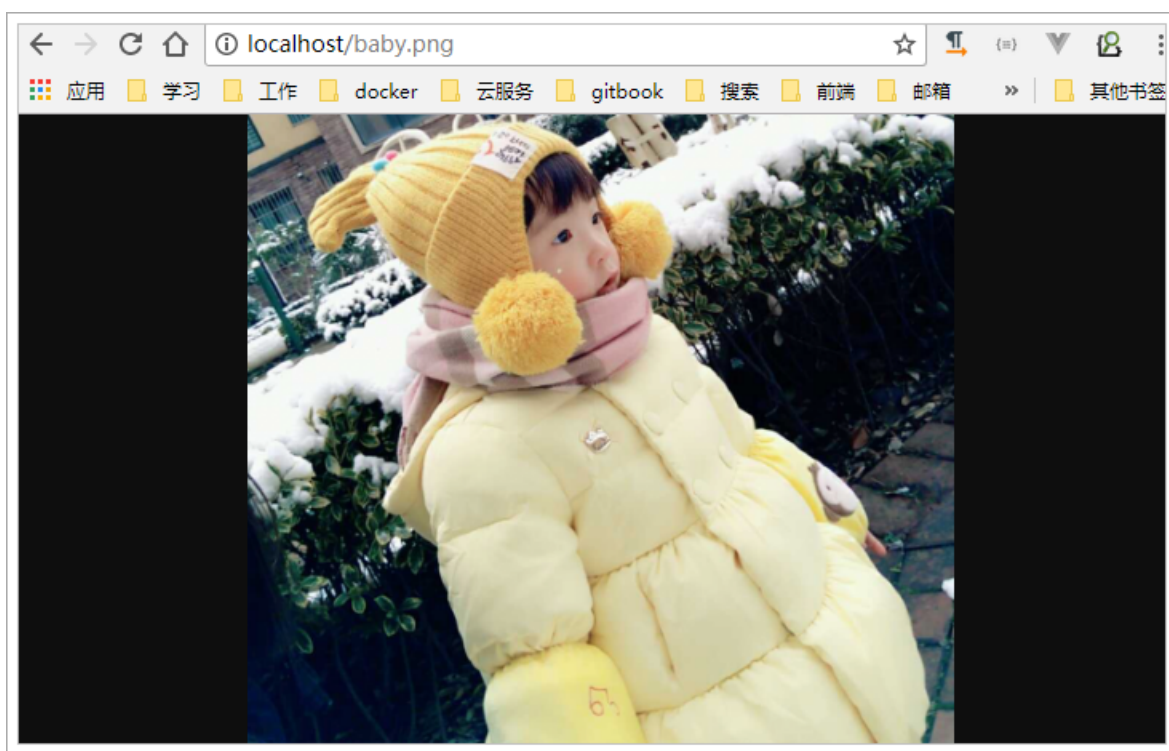
- classpath:/META-INF/resources/
- classpath:/resources/
- classpath:/static/
- classpath:/public

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:



5.1.3.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。

首先我们定义一个拦截器:

```
@Slf4j
public class LoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
        log.debug("preHandle method is now running!");
        return true;
    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) {
        log.debug("postHandle method is now running!");
    }

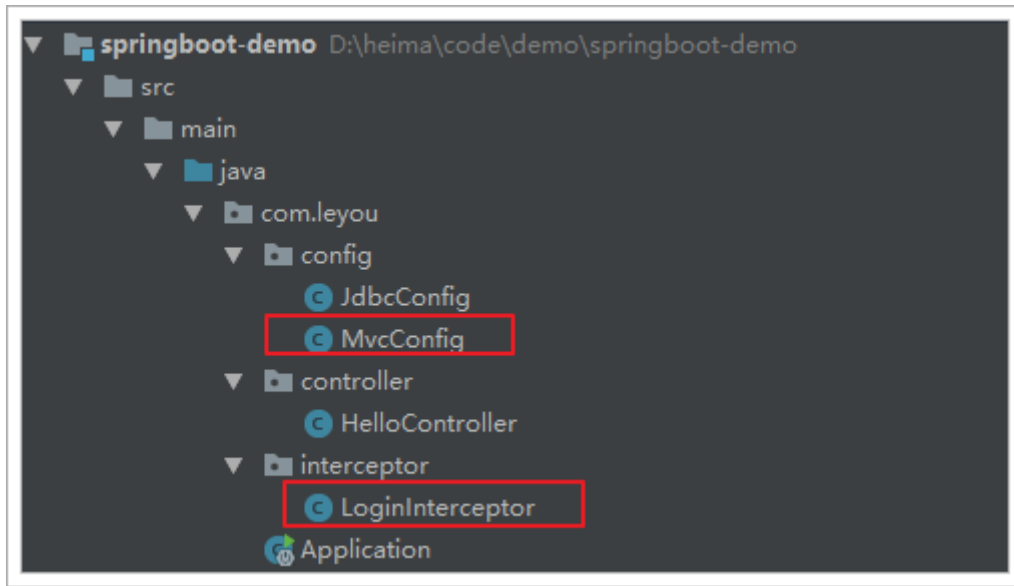
    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) {
        log.debug("afterCompletion method is now running!");
    }
}
```

然后, 我们定义配置类, 注册拦截器:

```
@Configuration
public class MvcConfig implements WebMvcConfigurer{

    /**
     * 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) {
        // 通过registry来注册拦截器, 通过addPathPatterns来添加拦截路径
        registry.addInterceptor(new LoginInterceptor()).addPathPatterns("/**");
    }
}
```

结构如下：



接下来运行并查看日志：

你会发现日志中什么都没有，因为我们记录的log级别是debug，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com.leyou包的日志级别为debug
logging:
  level:
    com.leyou=debug
```

再次运行查看：

```
2018-05-05 17:50:01.811 DEBUG 4548 --- [p-nio-80-exec-1]
com.leyou.interceptor.LoginInterceptor : preHandle method is now running!
2018-05-05 17:50:01.854 DEBUG 4548 --- [p-nio-80-exec-1]
com.leyou.interceptor.LoginInterceptor : postHandle method is now running!
2018-05-05 17:50:01.854 DEBUG 4548 --- [p-nio-80-exec-1]
com.leyou.interceptor.LoginInterceptor : afterCompletion method is now
running!
```

5.2.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：

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

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>5.1.47</version>
</dependency>
```

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

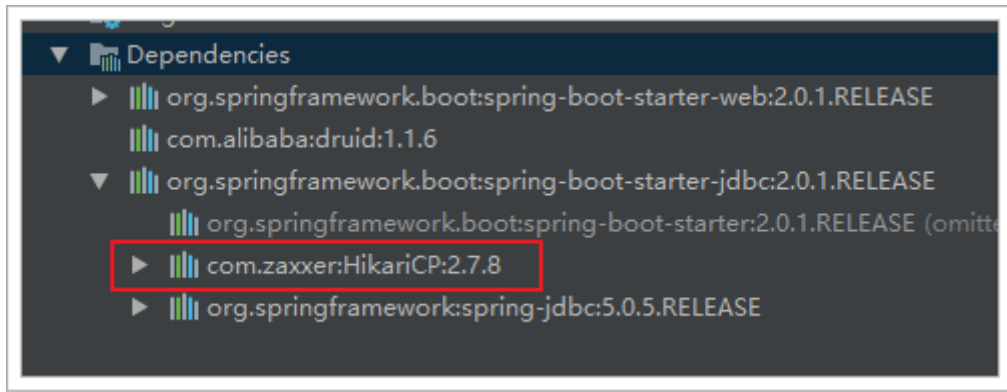
```
@Service
public class UserService {

    public User queryById(Long id){
        // 开始查询
        return new User();
    }

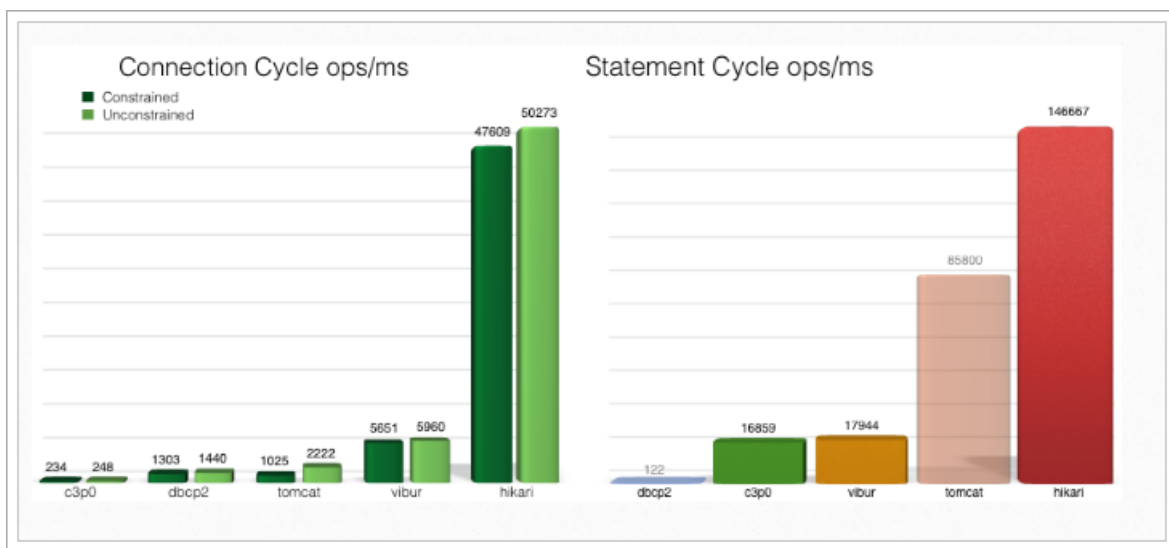
    @Transactional
    public void deleteById(Long id){
        // 开始删除
        System.out.println("删除了: " + id);
    }
}
```

5.3.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：



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：

```
spring:
  datasource:
    driver-class-name: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/mydb01
    username: root
    password: 123
```

5.4.整合mybatis

5.4.1.mybatis

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

```
<!--mybatis -->
<dependency>
  <groupId>org.mybatis.spring.boot</groupId>
  <artifactId>mybatis-spring-boot-starter</artifactId>
  <version>2.0.1</version>
</dependency>
```

配置，一些文件位置相关的，mybatis知道，需要我们来指定：

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

另外，Mapper接口的位置在application.yml中并不能配置，Mapper接口的扫描有两种实现方式：

方式一

我们需要给每一个Mapper接口添加 `@Mapper` 注解，由Spring来扫描这些注解，完成Mapper的动态代理。

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

方式二

在启动类上添加扫描包注解(推荐):

```
@SpringBootApplication
@MapperScan("cn.itcast.mapper")
public class Application {
    public static void main(String[] args) {
        // 启动代码
        SpringApplication.run(Application.class, args);
    }
}
```

这种方式的好处是，不用给每一个Mapper都添加注解。

以下代码示例中，我们将采用@MapperScan扫描方式进行。

5.4.2.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>2.1.5</version>
</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。

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://mapperhelper.github.io/docs/>

另外，我们需要把启动类上的@MapperScan注解修改为通用mapper中自带的：

```

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import tk.mybatis.spring.annotation.MapperScan;

/**
 * @author:
 * @create: 2018-07-11 09:04
 */
@SpringBootApplication
@MapperScan("cn.itcast.mapper")
public class DemoApplication {
    public static void main(String[] args) {
        // 启动springboot
        SpringApplication.run(DemoApplication.class, args);
    }
}

```

接下来就是通用mapper的使用步骤了：

1) 继承Mapper接口

```

public interface UserMapper extends Mapper<User>{
}

```

2) 在实体类上加JPA注解：

```

@Data
@Table(name = "tb_user")
public class User{
    @Id
    @KeySql(useGeneratedKeys = true) // 开启自增主键回显功能
    private Long id;
    private String userName;
    private String password;
    private String name;
    private Integer age;
    private Integer sex;
    private Date birthday;
    private Date created;
    private Date updated;
    private String note;
}

```

此时，对UserService的代码进行简单改造：

```
@Service
public class UserService {

    @Autowired
    private UserMapper userMapper;

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

}
```

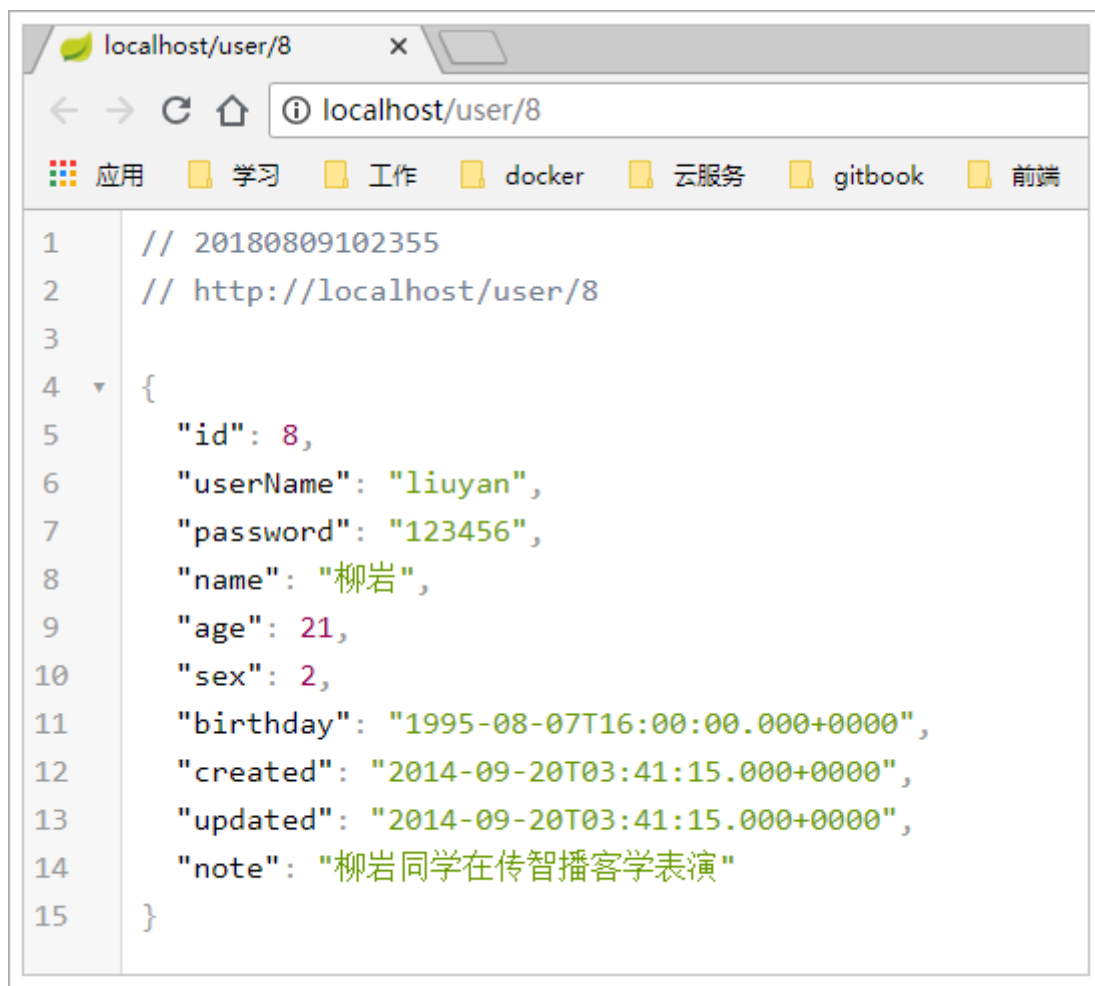
5.5.启动测试

将controller进行简单改造：

```
@RestController
public class HelloController {
    @Autowired
    private UserService userService;

    @GetMapping("/user/{id}")
    public User hello(@PathVariable("id") Long id) {
        User user = this.userService.queryById(id);
        return user;
    }
}
```

我们启动项目，查看：



The image shows a web browser window with the address bar displaying 'localhost/user/8'. Below the address bar is a navigation bar with several tabs: '应用' (Applications), '学习' (Learning), '工作' (Work), 'docker', '云服务' (Cloud Services), 'gitbook', and '前端' (Frontend). The main content area of the browser displays a JSON object, with line numbers 1 through 15 on the left side of the code editor. The JSON object contains the following fields: 'id' (8), 'userName' ('liuyan'), 'password' ('123456'), 'name' ('柳岩'), 'age' (21), 'sex' (2), 'birthday' ('1995-08-07T16:00:00.000+0000'), 'created' ('2014-09-20T03:41:15.000+0000'), 'updated' ('2014-09-20T03:41:15.000+0000'), and 'note' ('柳岩同学在传智播客学表演').

```
1 // 20180809102355
2 // http://localhost/user/8
3
4 {
5   "id": 8,
6   "userName": "liuyan",
7   "password": "123456",
8   "name": "柳岩",
9   "age": 21,
10  "sex": 2,
11  "birthday": "1995-08-07T16:00:00.000+0000",
12  "created": "2014-09-20T03:41:15.000+0000",
13  "updated": "2014-09-20T03:41:15.000+0000",
14  "note": "柳岩同学在传智播客学表演"
15 }
```