

1. springboot概念

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

2. 入门案例

创建工程
添加依赖
 添加父工程坐标
 添加web启动器
 管理jdk版本
 完整pom
启动类
编写controller
启动测试

3. 全注解配置和属性注入

回顾历史
spring全注解配置
SpringBoot的属性注入
更优雅的注入

4. 自动配置原理

@SpringBootApplication
 @SpringBootConfiguration
 @ComponentScan
 @EnableAutoConfiguration
默认配置原理
总结

1. springboot概念

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

什么是SpringBoot

springboot是spring快速开发脚手架，通过约定大于配置的方式，快速构建和启动spring项目

为什么要学习SpringBoot

spring的缺点：

- 复杂的配置，
 项目各种配置是开发时的损耗，写配置挤占了写应用程序逻辑的时间。
- 混乱的依赖管理。

项目的依赖管理非常的繁琐。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，这是一个棘手的问题。并且，一旦选错了依赖的版本，随之而来的就是各种的不兼容的bug。

spring boot 可以解决上面2个问题

SpringBoot的特点

Spring Boot 特点:

- 快速开发spring应用的框架
- 内嵌tomcat和jetty容器，不需要单独安装容器，jar包直接发布一个web应用
- 简化maven配置，parent这种方式，一站式引入需要的各种依赖
- 基于注解的零配置思想
- 和各种流行框架，spring web mvc，mybatis，spring cloud无缝整合

更多细节，大家可以到[官网](#)查看。

总结

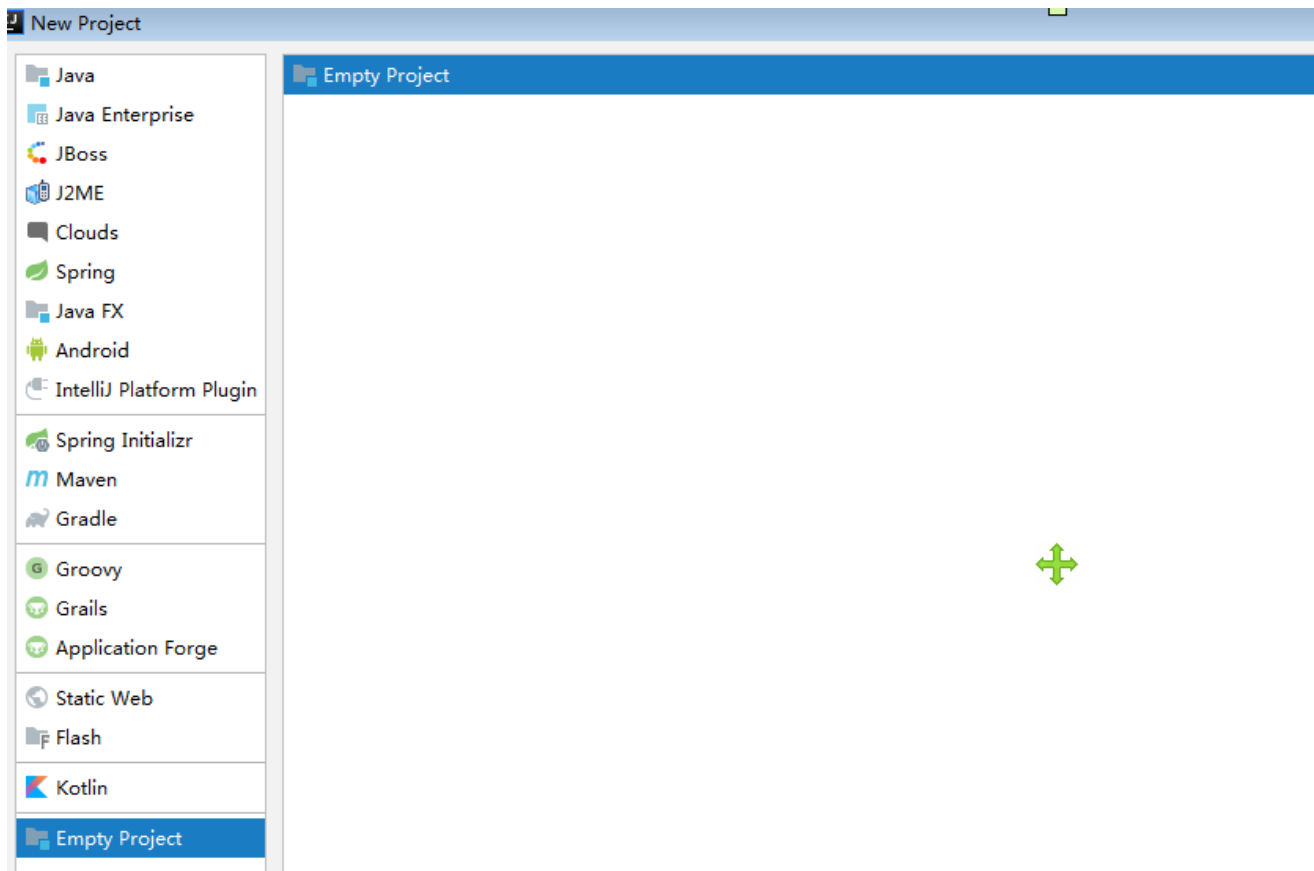
spring boot 是spring快速开发脚手架，通过约定大于配置，优化了混乱的依赖管理，和复杂的配置，让我们用java -jar方式，运行启动java web项目

2. 入门案例

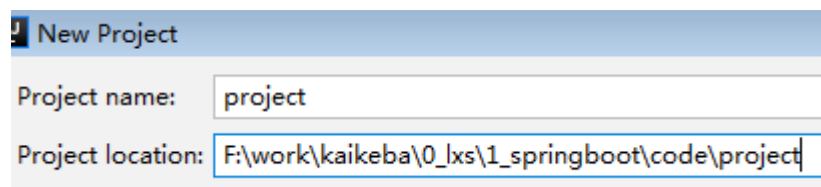
需求：创建HelloController,在页面中打印hello spring boot...

创建工程

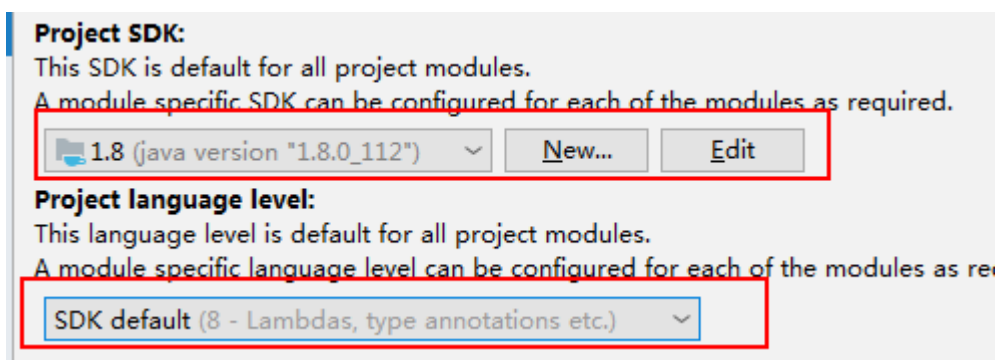
我们先新建一个空的工程：



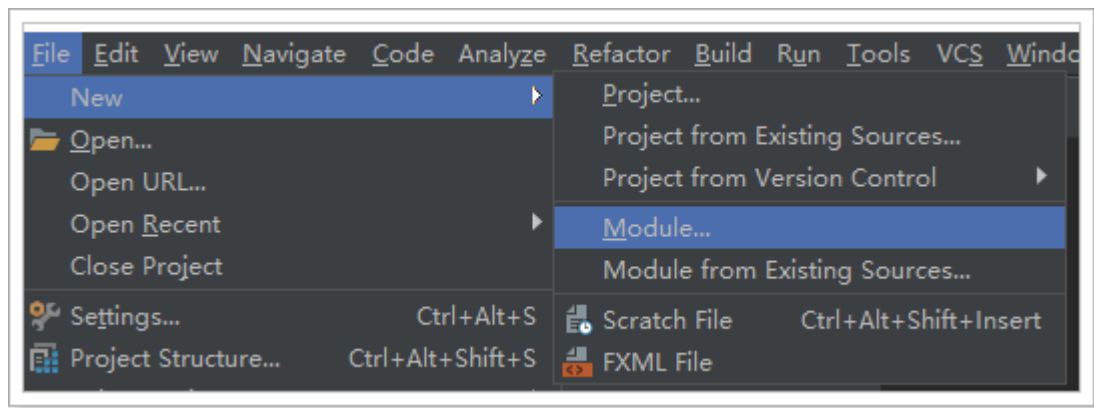
工程名为project:



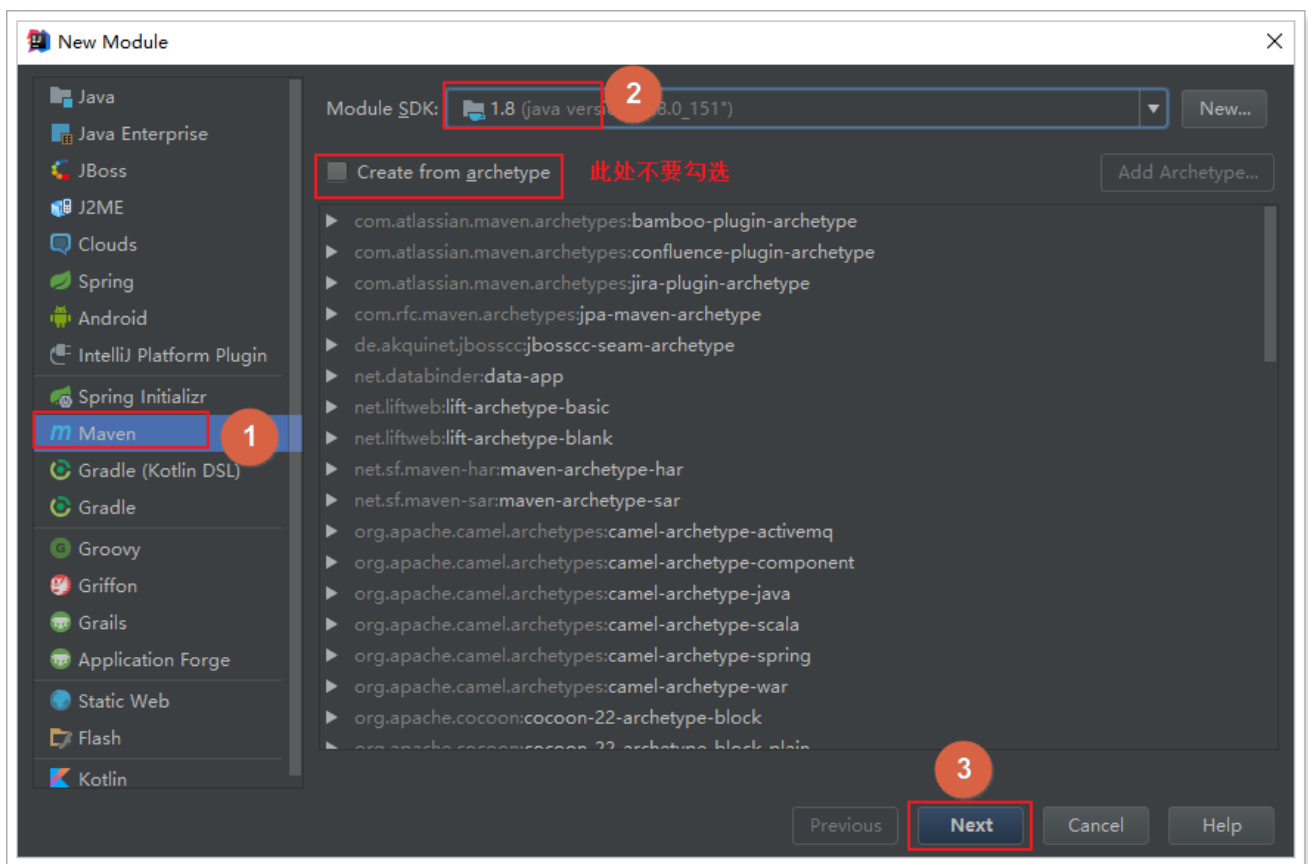
设置jdk版本为1.8:



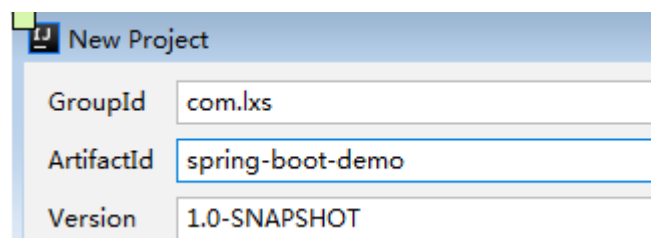
新建一个module:



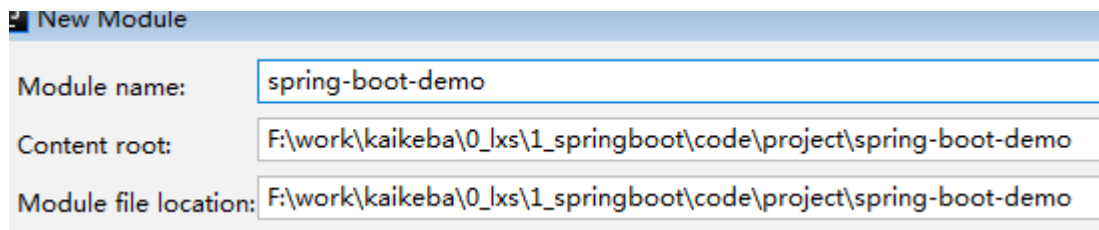
使用maven来构建:



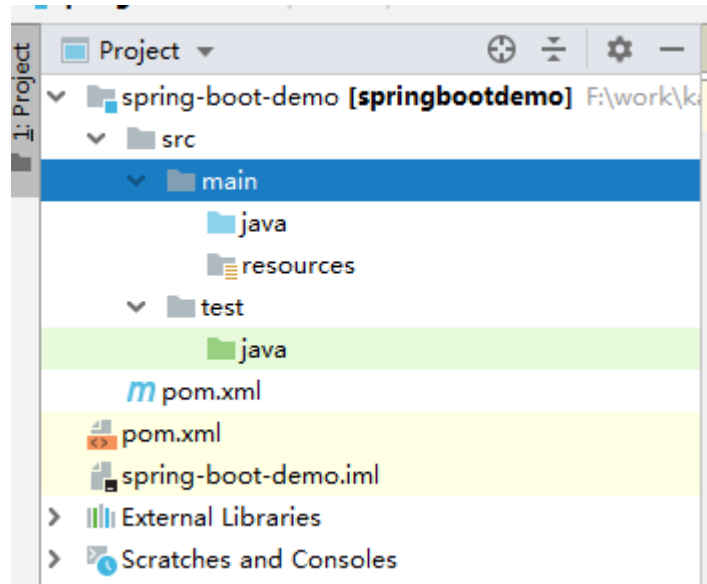
然后填写项目坐标:



目录结构:



项目结构：



添加依赖

SpringBoot提供了一个名为spring-boot-starter-parent的构件，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！

添加父工程坐标

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.0.0.RELEASE</version>
</parent>
```

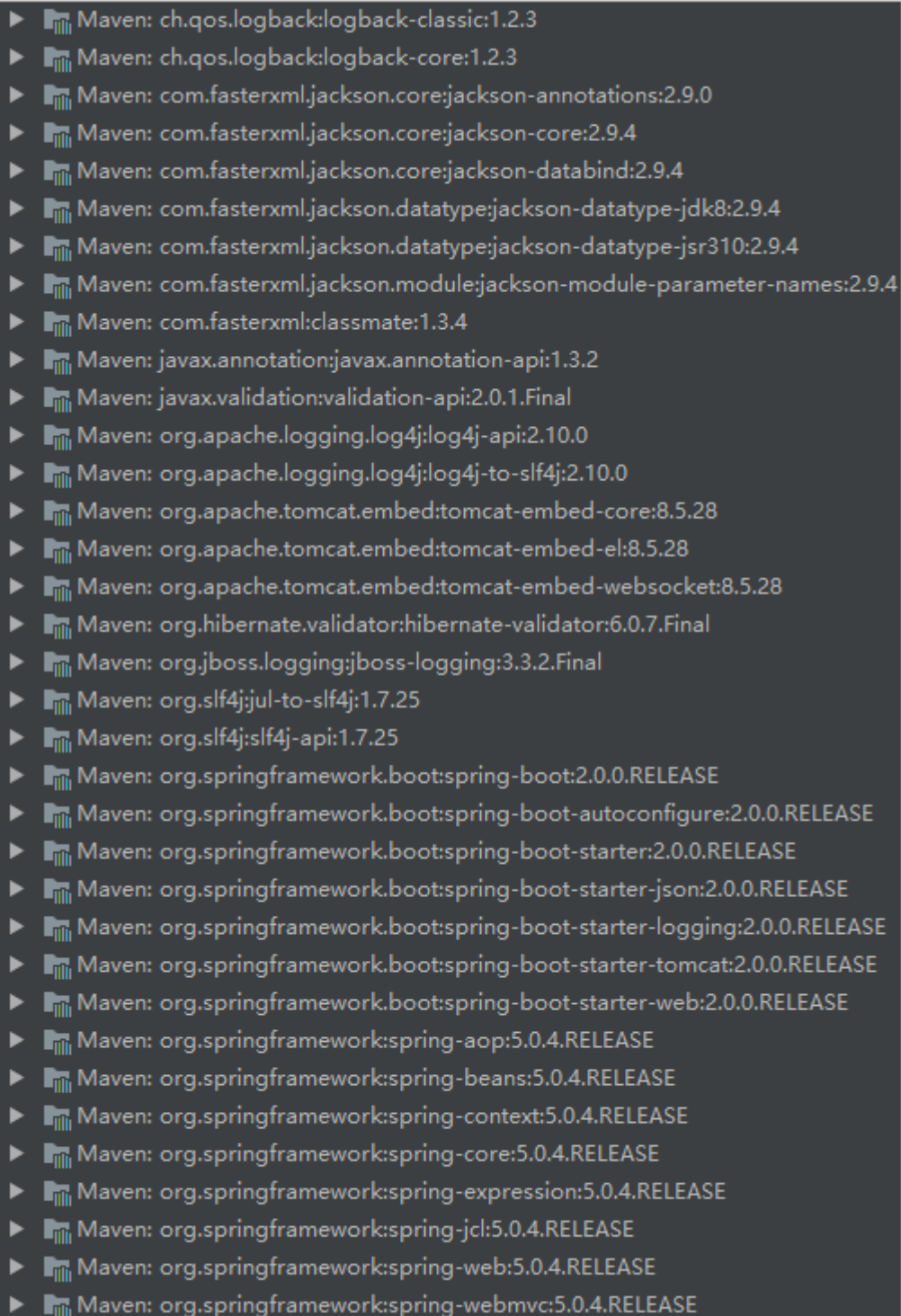
添加web启动器

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。

管理jdk版本

默认情况下，maven工程的jdk版本是1.5，而我们开发使用的是1.8，因此这里我们需要修改jdk版本，只需要简单的添加以下属性即可：

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

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.lxs.demo</groupId>
  <artifactId>springboot-demo</artifactId>
  <version>1.0-SNAPSHOT</version>

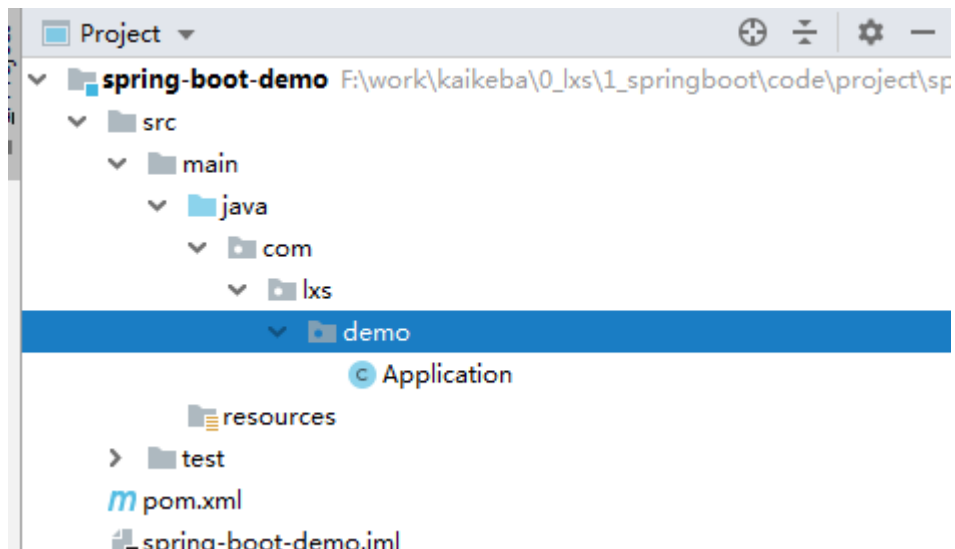
  <properties>
    <java.version>1.8</java.version>
  </properties>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.0.RELEASE</version>
  </parent>

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

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：



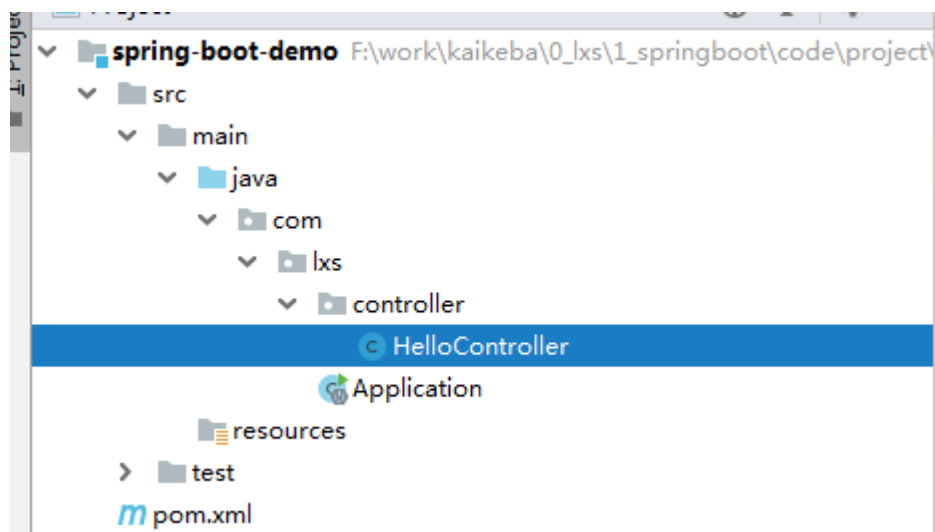
然后编写main函数:

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

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:



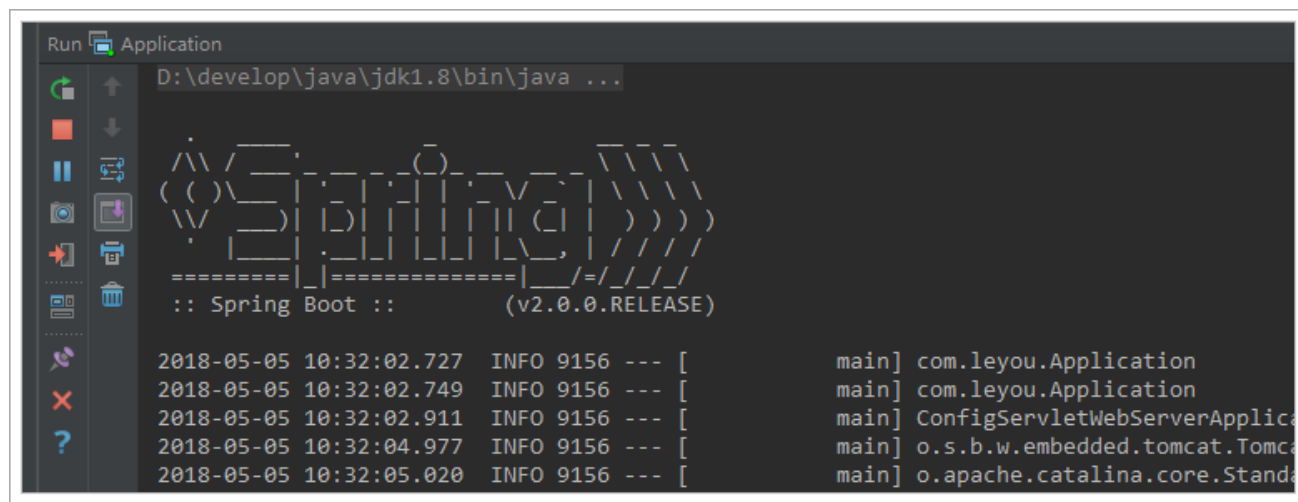
代码:


```
@RestController
public class HelloController {

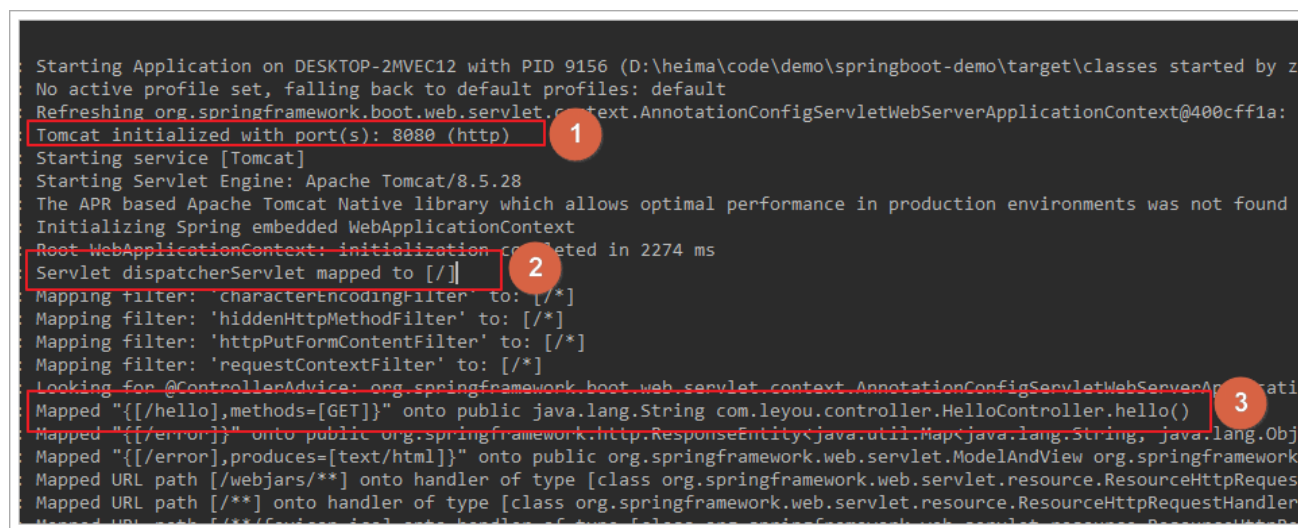
    @GetMapping("/hello")
    public String hello(){
        return "hello, spring boot!";
    }
}
```

启动测试

接下来，我们运行main函数，查看控制台：

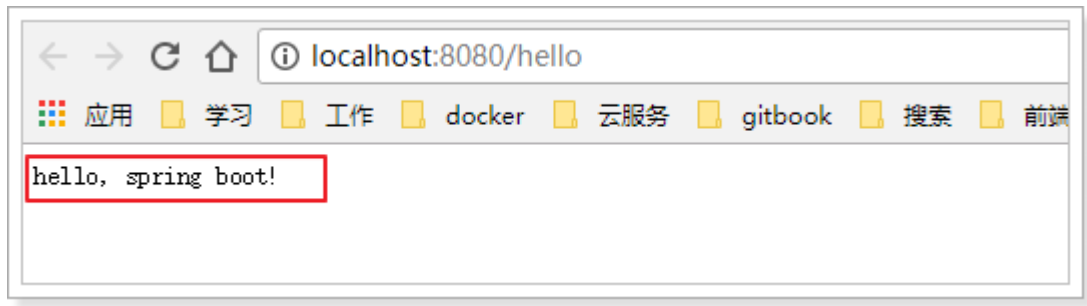


并且可以看到监听的端口信息：



- 1) 监听的端口是8080
- 2) SpringMVC的映射路径是: /
- 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. 全注解配置和属性注入

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？

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配置的优雅。

spring全注解配置

spring全注解配置主要靠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：

首先引入Druid连接池依赖：

```
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>druid</artifactId>
  <version>1.1.10</version>
</dependency>
```

创建一个jdbc.properties文件，编写jdbc属性(可以拷贝)：

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

然后编写代码：

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

    @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。默认的对象名id=方法名，可以通过 `@Bean("自定义名字")`，来指定新的对象名

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

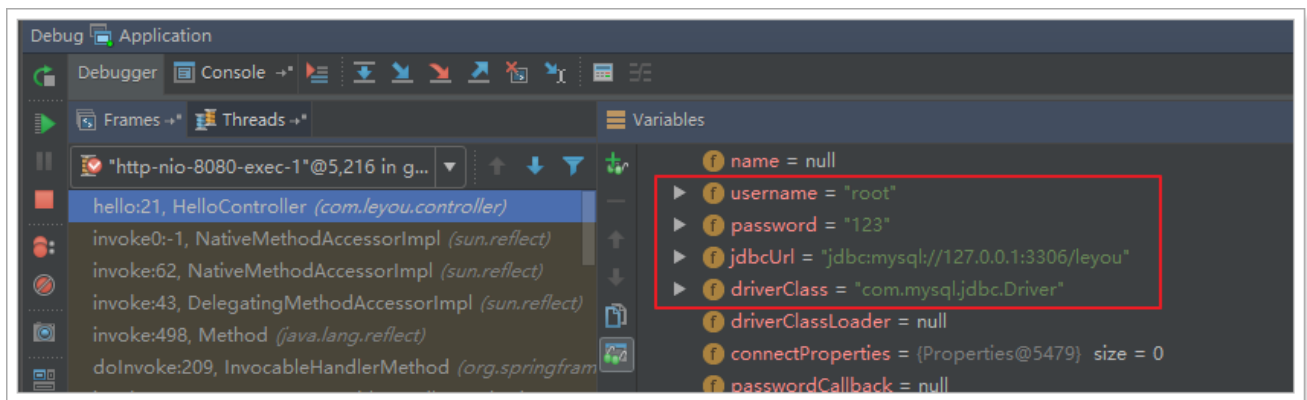
我们在 `HelloController` 中测试：

```
@RestController
public class HelloController {

    @Autowired
    private DataSource dataSource;

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

然后Debug运行并查看：



属性注入成功了！

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。

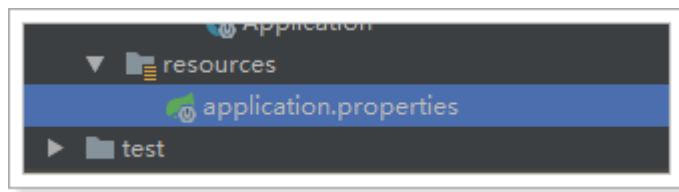
1) 我们新建一个类，用来进行属性注入：

```

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

```

- 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SpringBoot默认读取的属性文件名：



2) 在JdbcConfig中使用这个属性：

```

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

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

```

- 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注入JdbcProperties：
 - @Autowired注入

```

@Autowired
private JdbcProperties prop;

```

- 构造函数注入

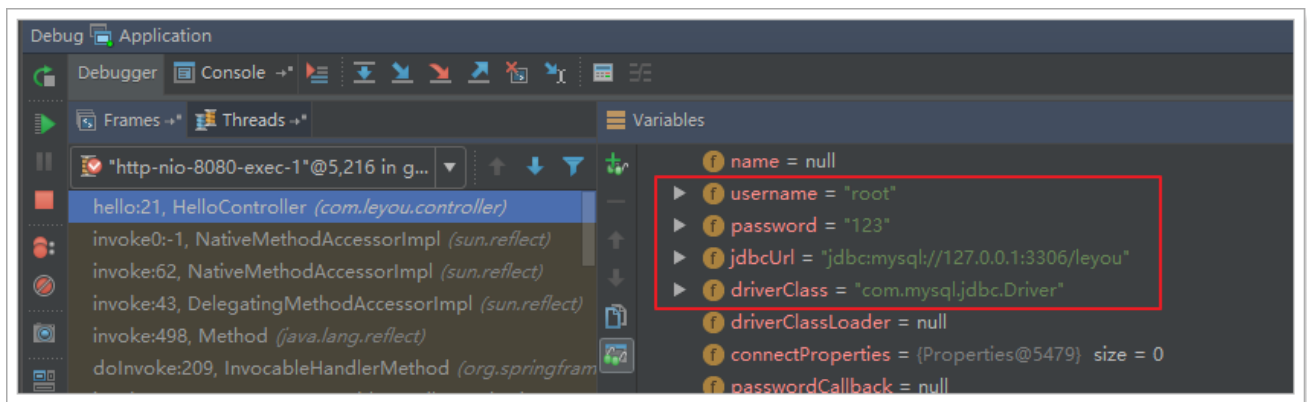
```
private JdbcProperties prop;  
public JdbcConfig(JdbcProperties prop){  
    this.prop = prop;  
}
```

- 声明有@Bean的方法参数注入

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

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：

24.7.5 @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）。

更优雅的注入

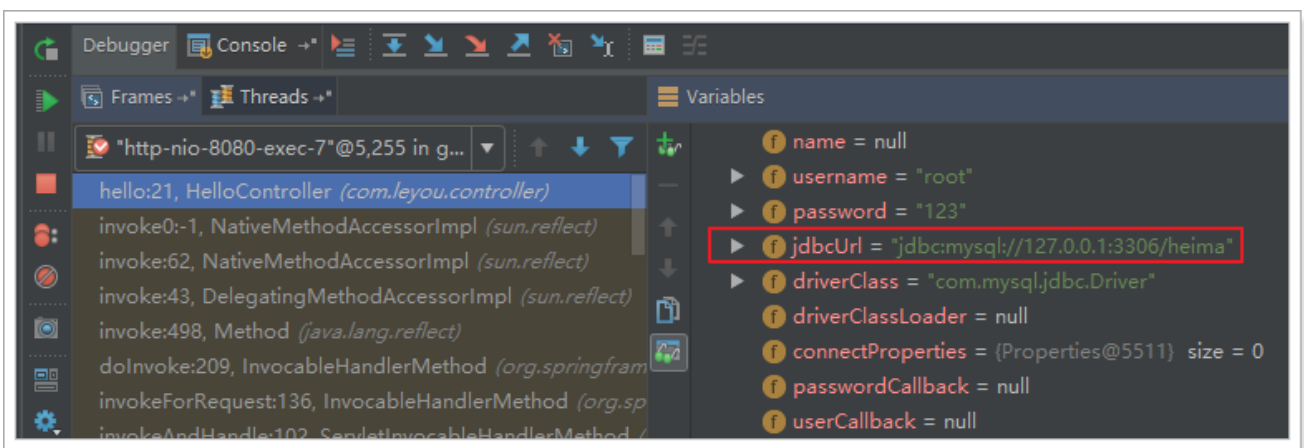
事实上，如果一段属性只有一个Bean需要使用，我们无需将其注入到一个类（JdbcProperties）中。而是直接在需要的地方声明即可：

```
@Configuration
public class JdbcConfig {

    @Bean
    // 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：/lxs，再次测试：



4. 自动配置原理

通过刚才的案例看到，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？

这些都是从springboot启动器开始的

```

5
6  @SpringBootApplication
7  public class Application {
8
9      public static void main(String[] args) {
10         SpringApplication.run(Application.class, args);
11     }
12 }

```

我们重点关注@SpringBootApplication注解

@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

@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 @SpringBootApplication} 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 SpringBootApplication {
}

```

通过这段我们可以看出，在这个注解上面，又有一个 `@Configuration` 注解。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。

@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")}</p>
     * 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。**

@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：

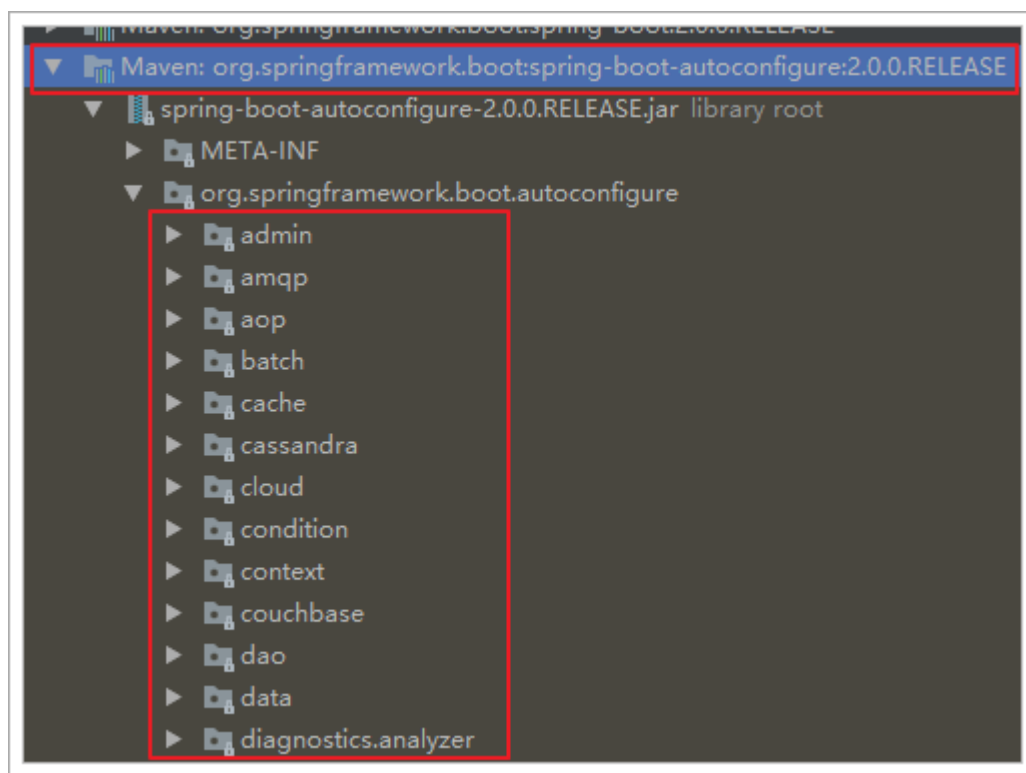
第二级的注解 `@EnableAutoConfiguration`，告诉SpringBoot基于你所添加的依赖，去“猜测”你想要如何配置Spring。比如我们引入了 `spring-boot-starter-web`，而这个启动器中帮我们添加了 `tomcat`、`SpringMVC` 的依赖。此时自动配置就知道你是要开发一个web应用，所以就帮你完成了web及SpringMVC的默认配置了！

总结，SpringBoot内部对大量的第三方库进行了默认配置，我们引入对应库所需的依赖，那么默认配置就会生效。

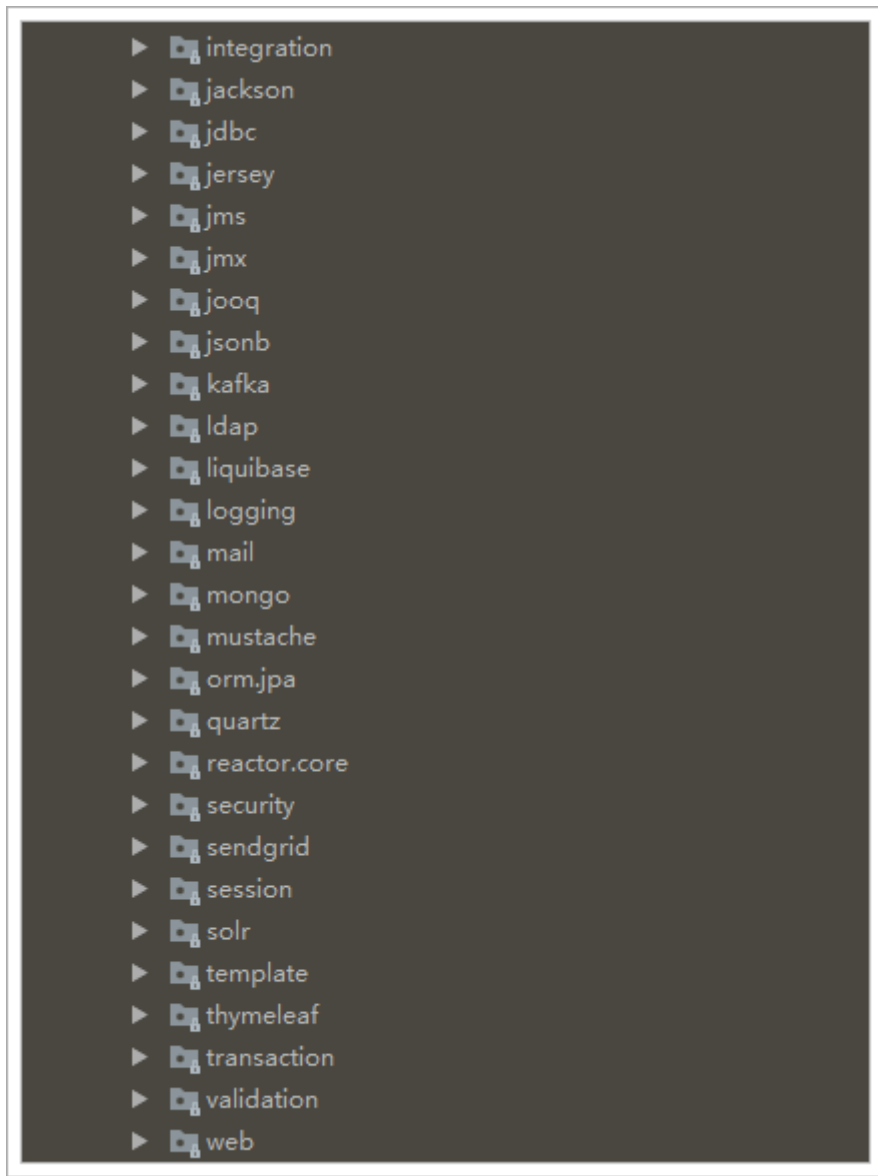
默认配置原理

`@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，springboot如何做到的？

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：

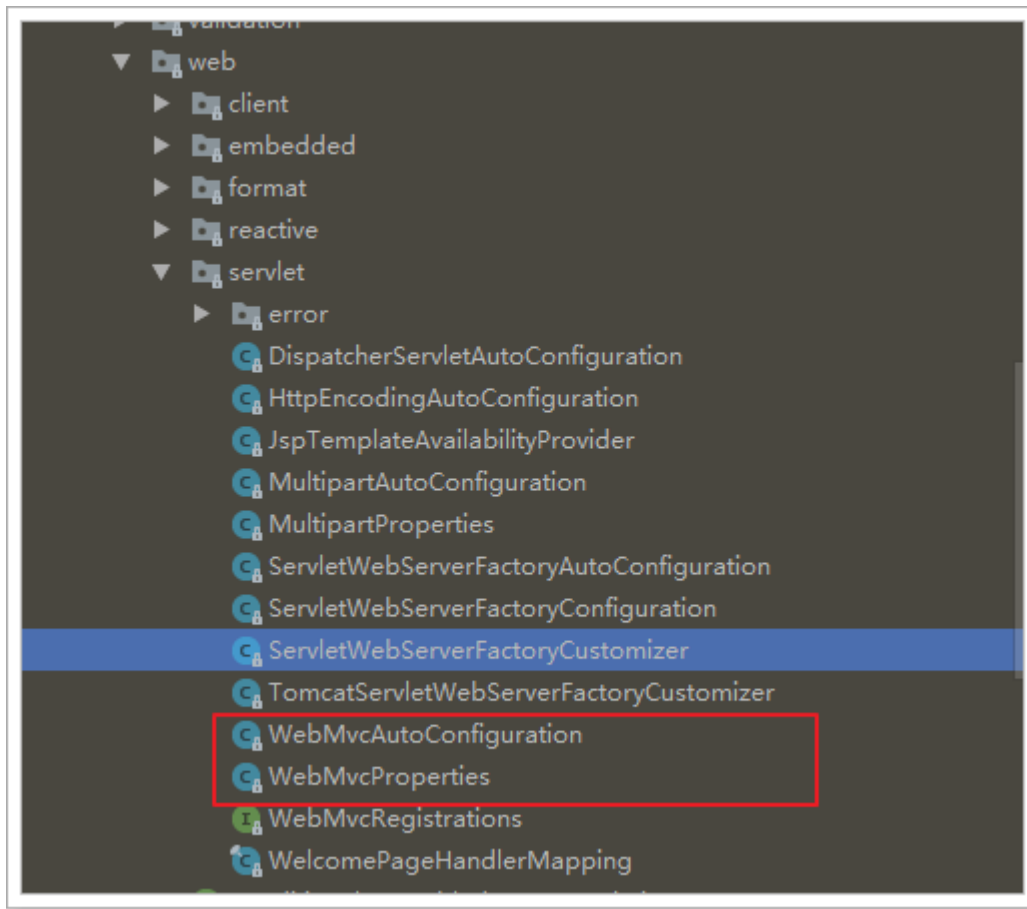


非常多，几乎涵盖了现在主流的开源框架，例如：

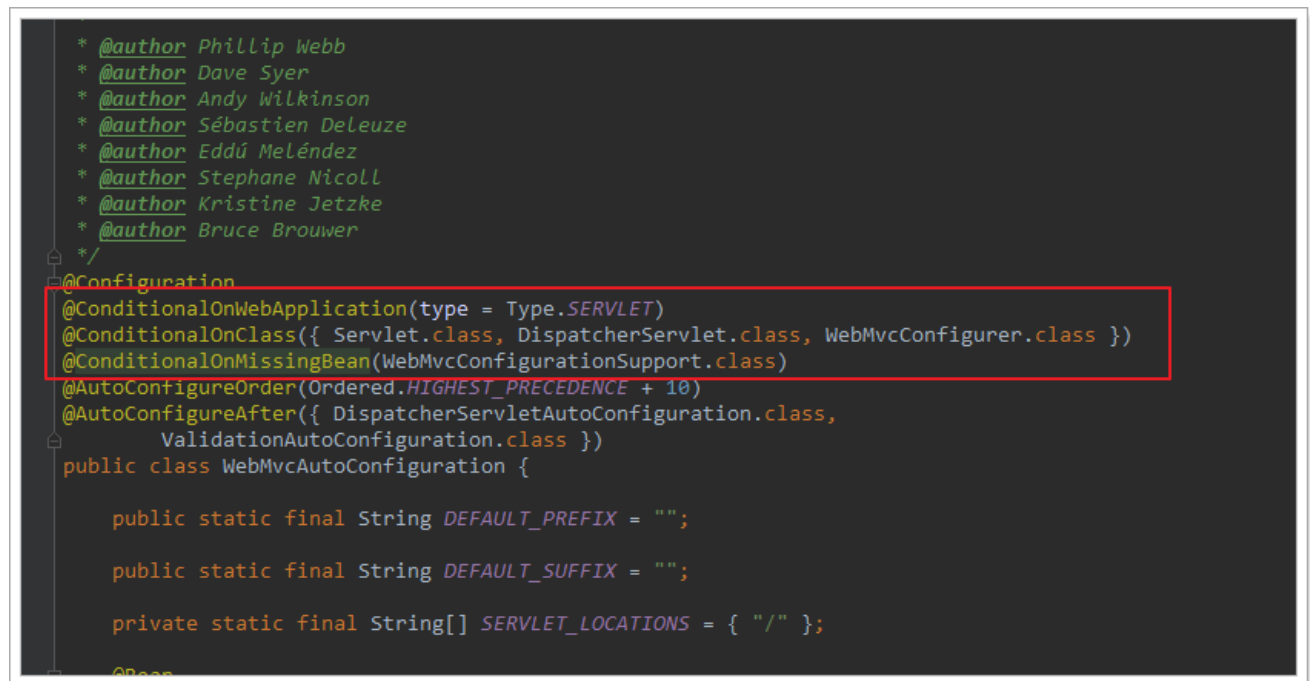
- redis
- jms
- amqp
- jdbc
- jackson
- mongodb
- jpa
- solr
- elasticsearch

... 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
- `@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：

```
@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。

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：

- 引入了相关依赖
- 没有自定义配置类