

# springboot(上课)

## 快速开始spring boot应用

### 官方向导搭建boot应用

1. 地址: <http://start.spring.io/>
2. 设置项目属性:

start.spring.io

SPRING INITIALIZR bootstrap your application now

Generate a **Maven Project** with **Java** and Spring Boot **1.5.10**

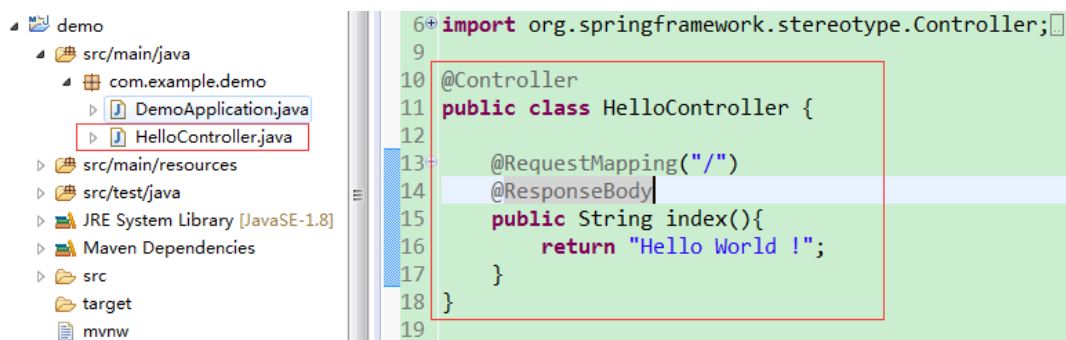
**Project Metadata**  
Artifact coordinates  
**Group**  
com.example  
**Artifact**  
demo

**Dependencies**  
Add Spring Boot Starters and dependencies to your application  
**Search for dependencies**  
Web, Security, JPA, Actuator, Devtools...  
**Selected Dependencies**  
Web

Generate Project alt + G

Don't know what to look for? Want more options? [Switch to the full version.](#)

3. 解压, 拷贝到工作空间, 导入maven项目
4. 写Controller: HelloController.java

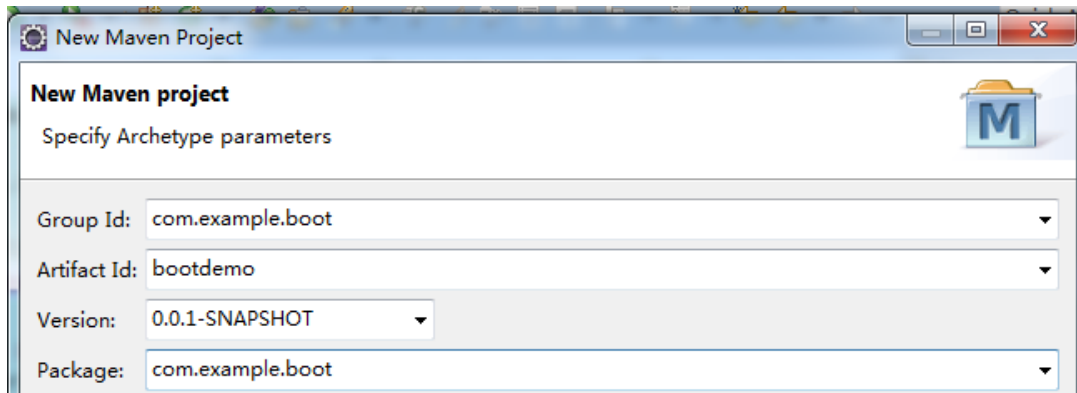
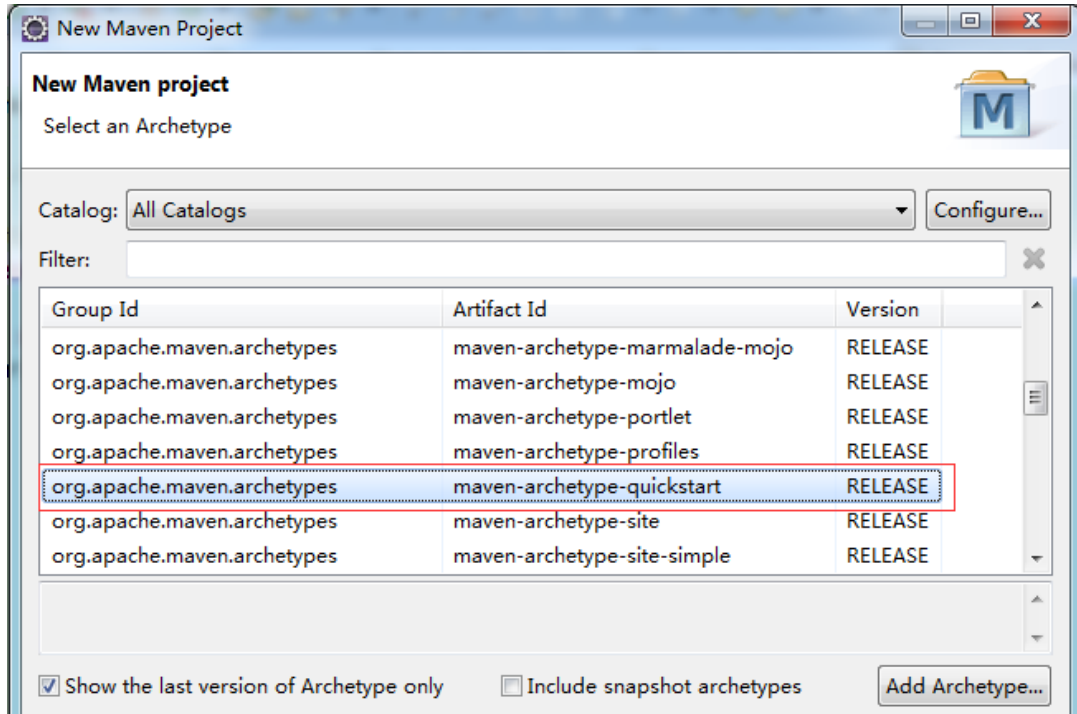


5. 启动Spring Boot入口类: DemoApplication

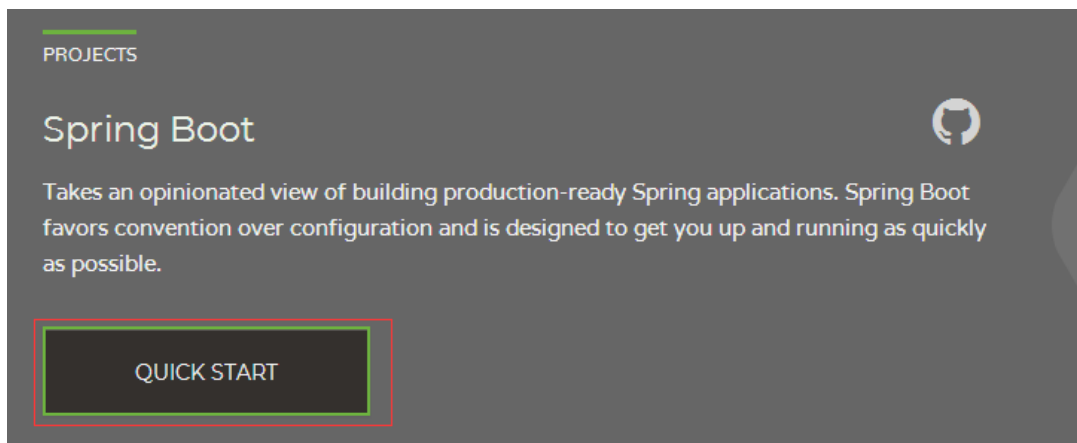
## 普通maven工程搭建boot应用

1. 新建一个普通的maven工程，选择quickstart

【注意：Spring boot是web工程，但是我们这里只需要建立quickstart即可，因为spring boot内嵌了servlet容器】



2. 查看官方文档：<https://projects.spring.io/spring-boot/> 点击quick start



3. 选择版本【1.5.10.RELEASE】，建议：生产环境中选择稳定的版本

4. 拷贝依赖的父pom到自己的工程pom文件中：

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
```

```

3     <artifactId>spring-boot-starter-parent</artifactId>
4     <version>1.5.10.RELEASE</version>
5 </parent>
6 <dependencies>
7     <dependency>
8         <groupId>org.springframework.boot</groupId>
9         <artifactId>spring-boot-starter-web</artifactId>
10    </dependency>
11 </dependencies>

```

5.从上面的第一个boot项目的pom中拷贝项目构建的内容到当前工程中（以下内容为每个maven项目都必须要的）：

```

1 <properties>
2     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
3     <project.reporting.outputEncoding>UTF-
4     8</project.reporting.outputEncoding>
5     <java.version>1.8</java.version>
6 </properties>

```

```

1 <build>
2     <plugins>
3         <plugin>
4             <groupId>org.springframework.boot</groupId>
5             <artifactId>spring-boot-maven-plugin</artifactId>
6         </plugin>
7     </plugins>
8 </build>

```

6.如果项目出现红叉，选择项目 -- 》右键 --》Maven--》Update Project

7.拷贝文档中的事例代码**SampleController.java**到工程中

8.Run as --> Java Application启动SampleController.java

9.浏览器输入：<http://localhost:8080/> 即可

当然，除了以上两种方式搭建boot工程，也可以通过其它工具快速生成，例如idea，sts，spring boot cli等

这些工具集成了spring boot特性，可以一键生成springboot工程骨架

## Starter POM

## 统一父POM管理

## 建立boot-parent工程

好，首先我们建立一个 **boot-parent**的maven工程：

Group Id:	com.example.boot
Artifact Id:	boot-parent
Version:	0.0.1-SNAPSHOT
Package:	com.example.boot

然后修改pom.xml

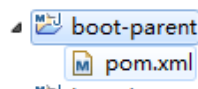
1. packaging改为pom格式:

`<packaging>pom</packaging>`

2. 加入`dependencyManagement`, 同时去掉`version`, 直接使用父pom中的版本即可

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

3. 删除无用的源文件, 只保留pom.xml



4. 修改pom.xml, 加入如下内容, 从上面获取即可:

```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>
```

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

那么我们要成为一个springboot项目, 必须要引入他的父pom对不对:

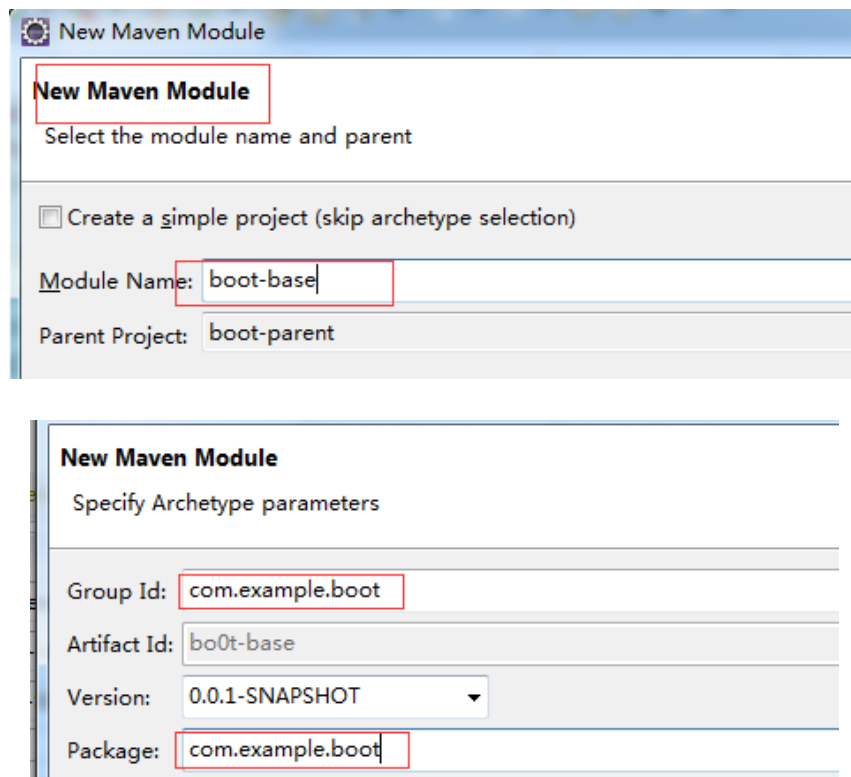
于是加入他的父pom:

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-parent</artifactId>
  <version>1.5.10.RELEASE</version>
</dependency>
```

## 建立boot-base工程：

建立boot-base工程，实现之前的helloworld功能：

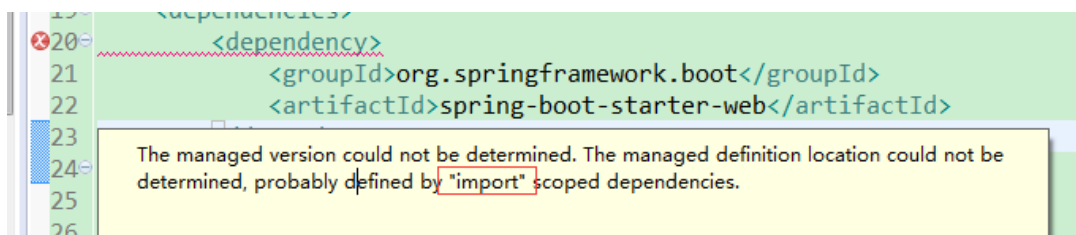
1. 在boot-parent工程上面，建立maven module模块工程



2. 把之前的SampleController.java复制过来，但是会报错，这时候，加入如下内容：

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

1. 如果报错，如下：



2. 需要修改父pom.xml中内容,boot-parent中的pom.xml,加入如下内容：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-parent</artifactId>
    <version>1.5.10.RELEASE</version>
    <type>pom</type>
    <scope>import</scope>
</dependency>
```

3. 启动SampleController，然后访问：<http://localhost:8080/>

spring boot 一个很重要的特点：解决了所有依赖的版本问题

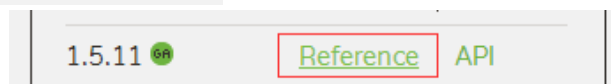
## spring boot 测试

1. 添加测试支持依赖：spring-boot-starter-test

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

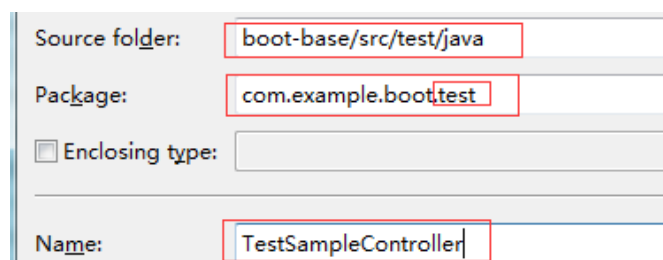
注意：加入这个依赖之后，junit包就可以不用了，因为test的starter中包含了junit

备注：怎么找到所有的starter：



这里面ctrl +f 搜索：starter，就可以看到spring boot中的所有starter

1. 在测试包中建立测试程序类，测试SampleController



3. 编写测试类：

```
@SpringBootTest(classes=SampleController.class)// spring boot test支持
@WebAppConfiguration // 按照web的形式运行测试，因为我们是web项目
@RunWith(SpringJUnit4ClassRunner.class)// Spring junit的启动支持类
public class TestSampleController {

    @Autowired
    private SampleController sampleController;

    @Test
    public void testHome(){
        TestCase.assertEquals(sampleController.home(), "Hello World!");
    }
}
```

so easy

## spring boot 启动注解分析

1. @EnableAutoConfiguration：开启自动配置功能

@ComponentScan(basePackages={"com.example.boot"}) 包扫描

## 2.@SpringBootApplication配置详解：

他是一个组合注解，他内部主要包含三个子注解：@SpringBootConfiguration、@EnableAutoConfiguration、@ComponentScan

**@SpringBootConfiguration**：他继承@Configuration，说明这是一个配置类，什么是配置类呢？就相当于我们以前写的xml配置，例如我们我们的bean标签，用来实例化一个bean。那么在这个配置类中就是实现以前我们xml配置的功能

**@EnableAutoConfiguration**：开启自动配置功能，他会扫描带有@Configuration的类，然后初始化这些配置类中的信息并且加入到应用上下文中去，同时完成一些基本的初始化工作

**@ComponentScan**：组件包扫描，也就是我现在需要扫描哪些包下面的注解，可自动发现和装配一些bean。**默认扫描当前启动类所在包下面的类和下面的所有子包**

## spring boot 热加载/部署

### 1.加入springloaded

```
1 <dependency>
2   <groupId>org.springframework</groupId>
3   <artifactId>springloaded</artifactId>
4 </dependency>
```

### 2.加入spring-boot-devtools

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

3.启动程序，访问浏览器出现第一个结果，然后修改控制器输出内容，再次刷新看到新的结果  
同时在控制台可以看待这样一句话：

**o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 35729**

## 项目打包部署

1.修改boot-parent中pom.xml文件,增加如下内容（当然也可以把下面的内容复制到子模块中也是可以的）

```
1   <build>
2     <plugins>
3       <plugin><!-- 项目的打包发布 -->
4         <groupId>org.springframework.boot</groupId>
5         <artifactId>spring-boot-maven-plugin</artifactId>
6         <configuration>
7           <mainClass>com.example.boot.SpringBootMain</mainClass>
8         </configuration>
9       <executions>
10        <execution>
```

```

11         <goals>
12             <goal>repackage</goal>
13         </goals>
14     </execution>
15 </executions>
16 </plugin>
17 </plugins>
18 </build>

```

2.执行maven install

3.在target目录下面，可以看到打包的jar文件

4.执行java -jar xx.jar

例如：

D:\tools\javaSE1.8\jdk1.8\bin>java.exe -jar

D:\01\_dev\_env\repos\com\example\boot\boot-base\0.0.1-SNAPSHOT\boot-base-0.0.1-SNAPSHOT.jar

【注意：执行jar的jdk版本需要与jar打包编译的版本一致。如果配置了环境变量，直接使用java命令打包即可】

这就是微架构，一个程序打包之后轻轻松松在如任何地方一执行就完成了。

## Spring Boot属性配置文件详解

### 修改端口

application.properties:

```

1 server.port=8888

```

另外，也可以直接在运行jar包的时候修改

java -jar xx.jar --server.port=8888

### 自定义属性及获取

1.application.properties中[文件改成UTF-8]:

```

1 teacher.id=1
2 teacher.name=zhangsan

```

2.@Value("\${属性名}")获取对应的属性值

```

1 @Controller
2 public class SampleController {
3     @Value("${teacher.name}")
4     private String teacherName;

```



```

5     @RequestMapping("/")
6     @ResponseBody
7     public String home() {
8         return "Hello World!" + this.teacherName;
9     }
10 }

```

## 参数引用

application.properties

```

1 teacher.id=1
2 teacher.name=zhangsan
3 teacher.info=Teacher ${teacher.name}'s number is ${teacher.id}

```

## 随机内容生成

```

1 # 随机字符串
2 random.string=${random.value}
3 # 随机int
4 random.number=${random.int}
5 # 随机long
6 random.long=${random.long}
7 # 1-20的随机数
8 random.b=${random.int[1,20]}

```

## 多环境配置

我们在开发应用时，通常一个项目会被部署到不同的环境中，比如：开发、测试、生产等。其中每个环境的数据库地址、服务器端口等等配置都会不同，对于多环境的配置，大部分构建工具或是框架解决的基本思路是一致的，通过配置多份不同环境的配置文件，再通过打包命令指定需要打包的内容之后进行区分打包，Spring Boot也提供了支持

在Spring Boot中多环境配置文件名需要满足application-{profile}.properties的格式，其中{profile}对应你的环境标识，比如：

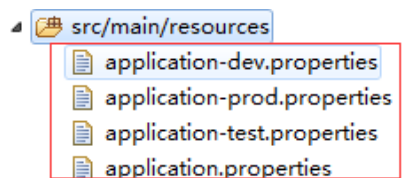
- application-dev.properties：开发环境
- application-test.properties：测试环境
- application-prod.properties：生产环境

至于哪个具体的配置文件会被加载，需要在application.properties文件中通过

spring.profiles.active属性来设置，其值对应{profile}值。

比如：spring.profiles.active=dev就会加载application-dev.properties配置文件中的内容

案例：



在dev, test, prod这三个文件均都设置不同的server.port端口属性，如：dev环境设置为

8081，test环境设置为8082，prod环境设置为8083

application.properties中设置spring.profiles.active=dev，就是说默认以dev环境设置

总结：

1.application.properties中配置通用内容，并设置spring.profiles.active=dev，以

开发环境为默认配置

2.application-{profile}.properties中配置各个环境不同的内容

## Spring boot 集成模板引擎实现web应用

### 静态资源访问

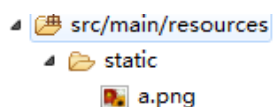
静态资源：js, css, html, 图片，音视频等

静态资源路径：是指系统可以直接访问的路径，且路径下的所有文件均可被用户直接读取。

Spring Boot默认提供静态资源目录位置需置于classpath下，目录名需符合如下规则：

```
1 /static
2 /public
3 /resources
4 /META-INF/resources
```

案例：在classpath下面创建static目录，并且加入一个图片a.png



加入之后，然后不需要重启直接访问：<http://localhost:8081/a.png>

修改默认的静态资源目录：`spring.resources.static-locations`

## 模板引擎

Spring Boot强烈建议使用模板引擎渲染html页面，避免使用JSP，若一定要使用JSP将无法实现Spring Boot的多种特性。

老师在这里讲两种模板引擎的集成：Thymeleaf(spring boot推荐), FreeMarker

【师傅领进门，修行靠个人，哈哈】

### Thymeleaf

Spring boot默认的模板配置路径为：`src/main/resources/templates`。当然也可以修改这个路径

集成Thymeleaf步骤：

1.修改pom.xml，增加如下依赖：

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

2.编写Controller

```
1 @Controller
2 public class SampleController {
3
4     @RequestMapping("/testThymeleaf")
5     public String testThymeleaf(ModelMap map) {
6         // 设置属性
7         map.addAttribute("name", "zhangsan");
8         // testThymeleaf: 为模板文件的名称
9         // 对应src/main/resources/templates/testThymeleaf.html
10        return "testThymeleaf";
11    }
12 }
```

3.在src/main/resources/下面建立templates/testThymeleaf.html

```
1 <!DOCTYPE html>
2 <html>
3     <head lang="en">
4         <meta charset="UTF-8" />
5         <title>testThymeleaf</title>
6     </head>
7     <body>
8         <h1 th:text="${name}">ABC</h1>
9     </body>
```

10 </html>

4.运行spring boot, 浏览器输入: <http://localhost:8081/testThymeleaf>

### Thymeleaf的默认参数配置 (供参考) :

```
1 # Enable MVC Thymeleaf view resolution.
2 spring.thymeleaf.enabled=true
3 # Enable template caching.
4 spring.thymeleaf.cache=true
5 # Check that the templates location exists.
6 spring.thymeleaf.check-template-location=true
7 # Content-Type value.
8 spring.thymeleaf.content-type=text/html
9 # Template encoding.
10 spring.thymeleaf.encoding=UTF-8
11 # Comma-separated list of view names that should be excluded from
    resolution.
12 spring.thymeleaf.excluded-view-names=
13 # Template mode to be applied to templates. See also
    StandardTemplateModeHandlers.
14 spring.thymeleaf.mode=HTML5
15 # Prefix that gets prepended to view names when building a URL.
16 spring.thymeleaf.prefix=classpath:/templates/
17 # Suffix that gets appended to view names when building a URL.
18 spring.thymeleaf.suffix=.html
19 # Order of the template resolver in the chain.
20 spring.thymeleaf.template-resolver-order=
21 # Comma-separated list of view names that can be resolved.
22 spring.thymeleaf.view-names=
```

## FreeMarker

### 1.修改pom.xml, 增加依赖

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

### 2.写Controller

```
1      @RequestMapping("/testFreemarker")
2      public String testFreemarker(Map<String,String> map) {
3          map.put("name", "张三");
4          return "hello"; //默认为src/main/resources/templates/hello.ftl
5      }
```

3.hello.ftl,目录为: src\main\resources\templates

```
1 <html>
2 <body>
3     hello, ${name}
4 </body>
5 </html>
```

3运行spring boot main, 浏览器输入如下地址:

<http://localhost:8081/testFreemarker>

## 集成Swagger2构建RESTful API文档

[Swagger2提供以下能力]:

- 1.随项目自动生成强大RESTful API文档, 减少工作量
- 2.API文档与代码整合在一起, 便于同步更新API说明
- 3.页面测试功能来调试每个RESTful API

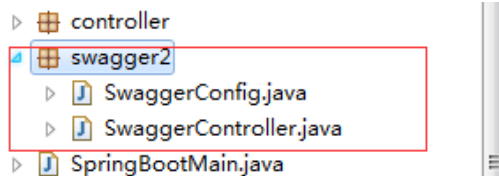
[集成Swagger2步骤]:

- 1.修改pom.xml, 添加Swagger2依赖

```
1 <dependency>
2     <groupId>io.springfox</groupId>
3     <artifactId>springfox-swagger2</artifactId>
4     <version>2.2.2</version>
5 </dependency>
6 <dependency>
7     <groupId>io.springfox</groupId>
8     <artifactId>springfox-swagger-ui</artifactId>
9     <version>2.2.2</version>
10 </dependency>
```

- 2.创建Swagger2配置类

在spring boot启动类所在包或子包中创建Swagger配置类SwaggerConfig.java, 如下:



SwaggerConfig.java内容如下:

```
1 @Configuration
2 @EnableSwagger2
3 public class SwaggerConfig {
```

```

4      @Bean
5      public Docket createRestApi() {
6          return new Docket(DocumentationType.SWAGGER_2)
7              .apiInfo(apiInfo())
8              .select()
9
10         .apis(RequestHandlerSelectors.basePackage("com.example.boot"))// 指定扫描包
           下面的注解
11         .paths(PathSelectors.any())
12         .build();
13     }
14     // 创建api的基本信息
15     private ApiInfo apiInfo() {
16         return new ApiInfoBuilder()
17             .title("集成Swagger2构建RESTful APIs")
18             .description("集成Swagger2构建RESTful APIs")
19             .termsOfServiceUrl("https://www.baidu.com")
20             .contact("zhangsan")
21             .version("1.0.0")
22             .build();
23     }

```

### 3.创建Controller: SwaggerController.java

```

1  @RestController
2  @RequestMapping(value="/swagger")
3  public class SwaggerController {
4      @ApiOperation(value="获取用户信息", notes="根据id来获取用户详细信息")
5      @ApiImplicitParam(name="id", value="用户ID", required=true,
6          dataType="String")
7      @RequestMapping(value="/{id}", method=RequestMethod.GET)
8      public Map<String,String> getInfo(@PathVariable String id) {
9          Map<String ,String> map = new HashMap<String, String>();
10         map.put("name", "张三");
11         map.put("age", "34");
12         return map;
13     }

```

4.启动Spring boot，访问Swagger UI界面：<http://localhost:8081/swagger-ui.html>

5.测试API:

Parameter	Value	Description	Parameter Type	Data Type
id	2	用户ID	body	String

Parameter content type:

### Response Messages

HTTP Status Code	Reason	Response Model	Headers
401	Unauthorized		
403	Forbidden		
404	Not Found		

[Try it out!](#) [Hide Response](#)

集成Swagger2源码成功!

## 统一异常处理

创建全局异常处理类：通过使用@ControllerAdvice定义统一的异常处理类，

@ExceptionHandler用来定义针对的异常类型

1.增加异常类：

```

1 @ControllerAdvice
2 class GlobalExceptionHandler {
3     @ExceptionHandler(value = Exception.class)
4     public ModelAndView defaultErrorHandler(HttpServletRequest req,
5     Exception e)
6     throws Exception {
7         ModelAndView mav = new ModelAndView();
8         mav.addObject("msg", "异常咯...");
9         mav.setViewName("error");
10        return mav;
11    }
12 }
```

2.增加Controller方法，抛出异常：

```

1 @RequestMapping("/exception")
2 public String hello() throws Exception {
3     throw new Exception("发生错误");
4 }
```

3.src/main/resources/templates增加error.html:

```

1 <!DOCTYPE html>
2 <html>
3 <head lang="en">
```

```

4     <meta charset="UTF-8" />
5     <title>统一异常处理</title>
6 </head>
7 <body>
8     <h1>Error</h1>
9     <div th:text="{msg}"></div>
10 </body>
11 </html>

```

## 集成Mybatis

集成步骤：

### 1.修改pom.xml,增加依赖

```

1     <dependency>
2         <groupId>org.mybatis.spring.boot</groupId>
3         <artifactId>mybatis-spring-boot-starter</artifactId>
4         <version>1.1.1</version><!-- 版本号必须需要 -->
5     </dependency>
6     <dependency>
7         <groupId>mysql</groupId>
8         <artifactId>mysql-connector-java</artifactId>
9     </dependency>

```

### 2.mysql的连接配置

application.properties：

```

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

```

### 3.创建表t\_user

```

1 CREATE TABLE `t_user` (
2   `id` int(11) NOT NULL AUTO_INCREMENT,
3   `name` varchar(40) DEFAULT NULL,
4   `age` int(11) DEFAULT NULL,
5   `address` varchar(100) DEFAULT NULL,
6   `phone` varchar(40) DEFAULT NULL,

```



```
7 PRIMARY KEY (`id`)  
8 ) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

## 4.创建User.java文件

包名为：com.example.boot.**bean**

```
1 public class User {  
2     private Integer id;  
3     private String name;  
4     private Integer age;  
5     private String address;  
6     private String phone;  
7     // getter,setter省略  
8 }
```

## 5.创建UserMapper.java接口文件

包名为：com.example.boot.**mybatis**

```
1 @Mapper  
2 public interface UserMapper {  
3     /**根据id查询用户*/  
4     @Select("SELECT * FROM T_USER WHERE ID = #{id}")  
5     User findById(@Param("id") String id);  
6     /**新增用户*/  
7     @Insert("INSERT INTO T_USER(NAME, AGE, ADDRESS, PHONE) VALUES(#{name},  
8     #{age}, #{address}, #{phone})")  
9     int insert(@Param("name") String name, @Param("age") Integer  
10    age,@Param("address") String address,@Param("phone") String phone);  
11 }
```

## 6.测试

```
1 @RunWith(SpringJUnit4ClassRunner.class)  
2 @SpringBootTest(classes = SpringBootMain.class)  
3 public class MybatisTest {  
4     @Autowired  
5     private UserMapper userMapper;  
6     @Test  
7     public void testInsert() throws Exception {  
8         int num = userMapper.insert("zhangsan", 20,"长沙","13100000000");  
9         TestCase.assertEquals(num,1);  
10    }  
11    @Test  
12    public void testFindById() throws Exception {
```

```
13         User u = userMapper.findById(14);
14         TestCase.assertNotNull(u);
15         System.out.println(u.getName());
16     }
17 }
```

注意：测试完毕之后，记得把这个**测试类注释掉**，不然后面构建整个项目的时候会执行test case,导致编译不成功

## 集成redis

集成Redis集成步骤：

### 1.修改pom.xml,增加依赖

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

注意：旧版本spring boot中集成的redis starter为：spring-boot-starter-redis

### 2.redis连接配置

```
1 # REDIS (RedisProperties)
2 # Redis数据库索引（默认为0）
3 spring.redis.database=0
4 # Redis服务器地址
5 spring.redis.host=127.0.0.1
6 # Redis服务器连接端口
7 spring.redis.port=6379
8 # Redis服务器连接密码（默认为空）
9 spring.redis.password=
10 # 连接池最大连接数（使用负值表示没有限制）
11 spring.redis.pool.max-active=8
12 # 连接池最大阻塞等待时间（使用负值表示没有限制）
13 spring.redis.pool.max-wait=-1
14 # 连接池中的最大空闲连接
15 spring.redis.pool.max-idle=8
16 # 连接池中的最小空闲连接
17 spring.redis.pool.min-idle=0
18 # 连接超时时间（毫秒）
19 spring.redis.timeout=0
```

注意：spring.redis.database的配置通常使用0即可，Redis在配置的时候可以设置数据库数量，默认为16，可以理解为数据库的schema

### 3.启动redis

windows:

```
1 redis-server redis.windows.conf
```

### 4.测试

```
1 @RunWith(SpringJUnit4ClassRunner.class)
2 @SpringBootTest(classes = SpringBootMain.class)
3 public class SpringRedisTest {
4     @Autowired
5     private RedisTemplate<String,String> redisTemplate;
6     @Test
7     public void testRedis() throws Exception {
8         ValueOperations<String, String> ops = redisTemplate.opsForValue();
9         ops.set("name", "zhangsan");
10        String value = ops.get("name");
11        System.out.println(value);
12        TestCase.assertEquals("zhangsan", value);
13    }
14 }
15
```

注意：redis中存储**对象**，需要我们自己实现**RedisSerializer<T>**接口来对传入对象进行序列化和反序列化

## 集成RabbitMQ

RabbitMQ是以AMQP协议实现的一种消息中间件产品，

AMQP是Advanced Message Queuing Protocol的简称，它是一个面向消息中间件的

开放式标准应用层协议。AMQP中定义了以下标准特性：

- 1 消息方向
- 2 消息队列
- 3 消息路由（包括：点到点模式和发布-订阅模式）

- 4 可靠性
- 5 安全性

关于AMQP 、RabbitMQ的详细内容不再这里过多介绍，本次课主要讲怎么与Spring boot集成

## 1.安装RabbitMQ[windows]

Erlang/OTP 20.3下载地址：

[http://erlang.org/download/otp\\_win64\\_20.3.exe](http://erlang.org/download/otp_win64_20.3.exe)

Erlang/OTP其它版本下载地址：<http://www.erlang.org/downloads>

RabbitMQ Server 3.7.4下载地址：


<https://dl.bintray.com/rabbitmq/all/rabbitmq-server/3.7.4/rabbitmq-server-3.7.4.exe>

RabbitMQ其它版本下载地址：<https://www.rabbitmq.com/download.html>

关于Linux平台怎么安装，同学们自行百度即可

## 2.启动RabbitMQ Server

RabbitMQ Server安装之后，会自动注册为windows服务，并以默认配置启动起来

名称	描述	状态	启动类型	登录为
 RabbitMQ	Multi-protoco...	已启动	自动	本地系统

所以需要启动的话，直接通过服务的方式启动即可。

## 3.RabbitMQ管理页面

### 1.开启Web管理插件

进入rabbitmq安装目录的sbin目录，在此打开dos命令窗口，执行以下命令

```
1 rabbitmq-plugins enable rabbitmq_management
```

出现如下提示，说明web管理插件安装成功

```
D:\01_dev_env\RabbitMQ\rabbitmq_server-3.7.4\sbin>rabbitmq-plugins enable rabbitmq_management
Enabling plugins on node rabbit@DELL-PC:
rabbitmq_management
The following plugins have been configured:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch
Applying plugin configuration to rabbit@DELL-PC...
The following plugins have been enabled:
  rabbitmq_management
  rabbitmq_management_agent
  rabbitmq_web_dispatch

set 3 plugins.
Offline change; changes will take effect at broker restart.
```

然后重新启动RabbitMQ 服务，打开浏览器并访问：<http://localhost:15672/>，并使用默认用户guest登录，密码也为guest，即可进入管理界面

## 4.Spring Boot整合

### 1.修改pom.xml，增加依赖支持

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

### 2.新增管理用户并设置权限

#### 1.Add a user

▼ Add a user

Username:

Password: ▼

(confirm)

Tags:

Set **Admin** | Monitoring | Policymaker  
Management | Impersonator | None

Add user

username:springboot  
password:123456

2.切换到springboot用户登陆，在All users中，点击Name为springboot， 进入权限设置页面

## Users

▼ All users

Filter:  ☐ Regex ?

Name	Tags	Can access virtual hosts	Has password
guest	administrator	/	•
springboot	administrator	No access	•

3.在权限设置页面，进入Permissions页面，点击“Set permission”

## User: springboot

► Overview

▼ Permissions

Current permissions

Virtual host	Configure regexp	Write regexp	Read regexp	
/	.*	.*	.*	Clear

Set permission

Virtual Host:

Configure regexp:

Write regexp:

Read regexp:

### 3.rabbit mq连接配置

```

1 ## rabbitmq config
2 spring.rabbitmq.host=localhost
3 spring.rabbitmq.port=5672
4 spring.rabbitmq.username=springboot
5 spring.rabbitmq.password=123456
6

```

### 4.创建Rabbit配置类

配置类主要用来配置队列、交换器、路由等高级信息

```

1 import org.springframework.amqp.core.Queue;
2 import org.springframework.context.annotation.Bean;
3 import org.springframework.context.annotation.Configuration;
4 @Configuration
5 public class RabbitConfig {
6     @Bean
7     public Queue firstQueue() {
8         // 创建一个队列，名称为：first
9         return new Queue("first");
10    }
11 }

```

### 5.创建消息产生者类

```

1 @Component
2 public class Sender {
3     @Autowired
4     private AmqpTemplate rabbitTemplate;

```

```

5     public void send() {
6         rabbitTemplate.convertAndSend("first", "test rabbitmq message
!!!");
7     }
8 }

```

说明：通过注入AmqpTemplate接口的实例来实现消息的发送，AmqpTemplate接口定义了一套针对AMQP协议的基础操作

## 6.创建消息消费者

```

1 @Component
2 @RabbitListener(queues = "first")
3 public class Receiver {
4     @RabbitHandler
5     public void process(String msg) {
6         System.out.println("receive msg : " + msg);
7     }
8 }

```

说明：

@RabbitListener注解：定义该类需要监听的队列

@RabbitHandler注解：指定对消息的处理

## 6.创建测试类

```

1 @RunWith(SpringJUnit4ClassRunner.class)
2 @SpringBootTest(classes = SpringBootMain.class)
3 public class RabbitmqTest {
4     @Autowired
5     private Sender sender;
6     @Test
7     public void testRabbitmq() throws Exception {
8         sender.send();
9     }
10 }

```

## 7.启动主程序：SpringBootMain

控制台如果出现以下信息，则说明rabbitmq连接成功

```

1 Created new connection:
   rabbitConnectionFactory#29102d45:0/SimpleConnection@1dcfb5ba
   [delegate=amqp://springboot@127.0.0.1:5672/, localPort= 55088]

```

## 8.运行JUnit测试类

控制台输出：

```
1 receive msg : test rabbitmq message !!!
```

集成Rabbit MQ完毕!

## Spring boot 日志

Java 有很多日志系统，例如，Java Util Logging, Log4J, Log4J2, Logback 等。Spring Boot 也提供了不同的选项，比如日志框架可以用 logback 或 log4j , log4j2等。

### 默认的日志框架 logback

springboot 自带log日志功能 使用的是slf4j(Simple Logging Facade For Java)，它是一个针对于各类Java日志框架的统一Facade抽象

日志实现默认使用的logback

Logback是log4j框架的作者开发的新一代日志框架，它效率更高、能够适应诸多的运行环境，同时天然支持SLF4J。这是默认支持logback的原因

例如，在spring-boot-starter 依赖中，添加了 spring-boot-starter-logging依赖

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

那么， Spring Boot 应用将自动使用 logback 作为应用日志框架， Spring Boot 启动的时候，由 org.springframework.boot.logging.Logging.LoggingApplicationListener 根据情况初始化并使用。

值得注意的是，默认情况下，Spring Boot 使用 logback 作为应用日志框架。因为 spring-boot-starter 其中包含了 spring-boot-starter-logging，该依赖就是 使用Spring Boot 默认的日志框架 logback

```
▲ spring-boot-starter-web : 1.5.10.RELEASE [compile]
  ▲ spring-boot-starter : 1.5.10.RELEASE [compile]
    ▶ spring-boot-starter-logging : 1.5.10.RELEASE [compile]
```

【程序中使用】：

```
1 import org.slf4j.Logger;
2 import org.slf4j.LoggerFactory;
3 private final Logger logger =
  LoggerFactory.getLogger(SampleController.class);
```

### 日志级别



默认情况下，Spring Boot 配置的是INFO 日志级别，也就是会输出INFO级别以上的日志（ERROR, WARN, INFO）。如果需要 Debug 级别的日志。在src/main/resources/application.properties 中配置。

```
1 debug=true
```

此外，配置 logging.level.\* 来具体输出哪些包的日志级别。

例如

```
1 logging.level.root=INFO
2 logging.level.org.springframework.web=DEBUG
3 logging.level.com.example.boot.controller=DEBUG
```

## 日志文件

默认情况下，Spring Boot 日志只会输出到控制台，并不会写入到日志文件，因此，对于正式环境的应用，我们需要通过在 application.properties 文件中配置 logging.file 文件名称和 logging.path 文件路径，将日志输出到日志文件中。

```
1 logging.path = /var/tmp
2 logging.file = xxx.log
3 logging.level.root = info
```

如果只配置 logging.path，在 /var/tmp文件夹生成一个日志文件为 spring.log。如果只配置 logging.file，会在项目的当前路径下生成一个 xxx.log 日志文件。

值得注意的是，日志文件会在 10MB 大小的时候被截断，产生新的日志文件。

## 常用的日志框架 log4j

如果，我们希望使用 log4j 或者 log4j2，我们可以采用类似的方式将它们对应的依赖模块加到 Maven 依赖中。

### 集成log4j2

在spring-boot-dependencies POMs中搜索spring-boot-starter-log4j2

发现Spring boot父Pom中自己提供了这个依赖，于是我们加入如下jar依赖：

```
1      <!-- log4j2 -->
2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter</artifactId>
5          <exclusions>
6              <exclusion>
7                  <groupId>org.springframework.boot</groupId>
8                  <artifactId>spring-boot-starter-logging</artifactId>
9              </exclusion>
10         </exclusions>
11     </dependency>
12     <dependency>
13         <groupId>org.springframework.boot</groupId>
14         <artifactId>spring-boot-starter-log4j2</artifactId>
15     </dependency>
```

日志使用跟上面logback一样。

## 集成log4j

在spring-boot-dependencies POMs中搜索spring-boot-starter-log4j  
发现Spring boot的父Poms中自己并没有提供了这个依赖，我们在  
<http://mvnrepository.com>  
中央仓库中查找spring-boot-starter-log4j



### 1. Spring Boot Log4J Starter

org.springframework.boot » spring-boot-starter-log4j

Spring Boot Log4J Starter

### 1. 加入pom依赖

```
1      <!-- log4j start -->
2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter</artifactId>
5          <exclusions>
6              <exclusion>
7                  <groupId>org.springframework.boot</groupId>
8                  <artifactId>spring-boot-starter-logging</artifactId>
9              </exclusion>
10         </exclusions>
11     </dependency>
12     <dependency>
13         <groupId>org.springframework.boot</groupId>
14         <artifactId>spring-boot-starter-log4j</artifactId>
15         <version>1.3.8.RELEASE</version>
16     </dependency>
17     <!-- log4j end -->
```

### 2. classpath下增加log4j.properties

```
1 log4j.rootCategory=INFO, stdout, file, errorfile
2 log4j.category.com.example.boot=INFO, myFile
3 log4j.logger.error=errorfile
4
5 # 控制台输出
6 log4j.appender.stdout=org.apache.log4j.ConsoleAppender
7 log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
8 log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS}
9 %5p %c{1}:%L - %m%n
10
11 # root日志输出
12 log4j.appender.file=org.apache.log4j.DailyRollingFileAppender
```

```

12 log4j.appender.file.file=logs/all.log
13 log4j.appender.file.DatePattern='.'yyyy-MM-dd
14 log4j.appender.file.layout=org.apache.log4j.PatternLayout
15 log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS}
   %5p %c{1}:%L - %m%n
16
17 # error日志输出
18 log4j.appender.errorfile=org.apache.log4j.DailyRollingFileAppender
19 log4j.appender.errorfile.file=logs/error.log
20 log4j.appender.errorfile.DatePattern='.'yyyy-MM-dd
21 log4j.appender.errorfile.Threshold = ERROR
22 log4j.appender.errorfile.layout=org.apache.log4j.PatternLayout
23 log4j.appender.errorfile.layout.ConversionPattern=%d{yyyy-MM-dd
   HH:mm:ss,SSS} %5p %c{1}:%L - %m%n
24
25 # com.example.boot下的日志输出
26 log4j.appender.myFile=org.apache.log4j.DailyRollingFileAppender
27 log4j.appender.myFile.file=logs/my.log
28 log4j.appender.myFile.DatePattern='.'yyyy-MM-dd
29 log4j.appender.myFile.layout=org.apache.log4j.PatternLayout
30 log4j.appender.myFile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS}
   %5p %c{1}:%L ---- %m%n

```

### 3.代码中使用log4j

```

1 import org.apache.log4j.Logger;
2 private final Logger logger = Logger.getLogger(xxx.class);

```

## 自定义视图映射

在项目开发过程中，经常会涉及页面跳转问题，而且这个页面跳转没有任何业务逻辑过程，只是单纯的路由过程（例如：点击一个按钮跳转到一个页面）

正常的写法是这样的：

```

1 @RequestMapping("/testmvc")
2 public String view(){
3     return "abc";
4 }

```

现在只需要这样统一写，此类必须在启动类所在包或者子包中：

```

1 @Configuration
2 public class WebMvcConfig extends WebMvcConfigurerAdapter{
3     @Override
4     public void addViewControllers(ViewControllerRegistry registry) {

```

```
5 registry.addViewController("/testmvc").setViewName("/abc");
6 }
7 }
```

页面：abc.flt 或者 abc.html

```
1 <html>
2 <body>
3     hello
4 </body>
5 </html>
```

访问<http://localhost:8081/testmvc> 即可访问到这个abc.flt文件

## 自定义Starter

在我们学习SpringBoot时都已经了解到starter是SpringBoot的核心组成部分，SpringBoot为我们提供了尽可能完善的封装，提供了一系列的自动化配置的starter插件，我们在使用spring-boot-starter-web时只需要在pom.xml配置文件内添加依赖就可以了，我们之前传统方式则需要添加很多相关SpringMVC配置文件。而spring-boot-starter-web为我们提供了几乎所有的默认配置，很好的降低了使用框架时的复杂度。

因此在使用xx.starter时你就不用考虑该怎么配置，即便是有一些必要的配置在application.properties配置文件内对应配置就可以了，那好，为什么我在application.properties配置对应属性后xx.starter就可以获取到并作出处理呢？下面我们带着这个疑问来编写我们自定义的starter让我们深入了解SpringBoot

## 创建自己的starter项目

创建普通maven项目，修改pom.xml,增加自动配置依赖

```
1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-autoconfigure</artifactId>
4     <version>1.5.10.RELEASE</version>
5 </dependency>
```

我们这个starter并不做其他复杂逻辑的编写，所以这里的依赖只是添加了spring-boot-autoconfigure，实战开发时可以添加任意依赖到项目中。

## 配置映射参数实体

starter是如何读取application.properties或者application.yml配置文件内需要的配置参数的呢？那么接下来我们就看看如何可以获取自定义的配置信息。

SpringBoot在处理这种事情上早就已经考虑到了，所以提供了一个注解

@ConfigurationProperties，该注解可以完成将application.properties配置文件内的有规则的配置参数映射到实体内的field内，不过需要提供setter方法，自定义配置参数实体代码如下所示：

```
1 @ConfigurationProperties(prefix = "hello")
2 public class HelloProperties{
3     private String msg = "test";
4     public String getMsg() {
5         return msg;
6     }
7     public void setMsg(String msg) {
8         this.msg = msg;
9     }
10 }
11
```

在上面代码中，@ConfigurationProperties注解内我们使用到了属性prefix，该属性配置了读取参数的前缀，根据上面的实体属性对应配置文件内的配置则是hello.msg，当然我们提供了默认值，配置文件内不进行配置时则是使用默认值

## 编写自定义业务

我们为自定义starter提供一个Service，并且提供一个名为sayHello的方法用于返回我们配置的msg内容。代码如下所示：

```
1 public class HelloService{
2     private String msg;
3
4     public String sayHello(){
5         return msg;
6     }
7     public void setMsg(String msg) {
8         this.msg = msg;
9     }
10 }
```

我们Service内的代码比较简单，根据属性参数进行返回格式化后的字符串。

接下来我们开始编写自动配置，这一块是starter的核心部分，配置该部分后在启动项目时才会自动加载配置，当然其中有很多细节性质的配置

## 实现自动化配置

自动化配置其实只是提供实体bean的验证以及初始化，我们先来看看代码：

```
1 @Configuration//开启配置
```

```

2  @EnableConfigurationProperties(HelloProperties.class)//开启使用映射实体对象
3  @ConditionalOnClass(HelloService.class)//存在HelloService时初始化该配置类
4  @ConditionalOnProperty//存在对应配置信息时初始化该配置类
5      (
6          prefix = "hello",//存在配置前缀hello
7          value = "enabled",//开启
8          matchIfMissing = true//缺失检查
9      )
10 public class HelloAutoConfiguration{
11     //application.properties配置文件映射前缀实体对象
12     @Autowired
13     private HelloProperties helloProperties;
14     /**
15      * 根据条件判断不存在HelloService时初始化新bean到SpringIoc
16      * @return
17      */
18     @Bean//创建HelloService实体bean
19     @ConditionalOnMissingBean(HelloService.class)//缺失HelloService实体bean
    时, 初始化HelloService并添加到SpringIoc
20     public HelloService helloService(){
21         System.out.println(">>>The HelloService Not Found, Execute Create
    New Bean.");
22         HelloService helloService = new HelloService();
23         helloService.setMsg(helloProperties.getMsg());//设置消息内容
24         return helloService;
25     }
26 }

```

自动化配置代码中有很多我们之前没有用到的注解配置，我们从上开始讲解

@Configuration：这个配置就不用多做解释了，我们一直在使用

@EnableConfigurationProperties：这是一个开启使用配置参数的注解，value值就是我们配置实体参数映射的ClassType，将配置实体作为配置来源。

SpringBoot内置条件注解

有关@ConditionalOnXxx相关的注解这里要系统的说下，因为这个是我们配置的关键，根据名称我们可以理解为具有Xxx条件，当然它实际的意义也是如此，条件注解是一个系列，下面我们详细做出解释

@ConditionalOnBean：当SpringIoc容器内存在指定Bean的条件

@ConditionalOnClass：当SpringIoc容器内存在指定Class的条件

@ConditionalOnExpression：基于SpEL表达式作为判断条件

@ConditionalOnJava：基于JVM版本作为判断条件

@ConditionalOnJndi：在JNDI存在时查找指定的位置

@ConditionalOnMissingBean：当SpringIoc容器内不存在指定Bean的条件

@ConditionalOnMissingClass：当SpringIoc容器内不存在指定Class的条件

@ConditionalOnNotWebApplication：当前项目不是Web项目的条件

@ConditionalOnProperty：指定的属性是否有指定的值

@ConditionalOnResource：类路径是否有指定的值

@ConditionalOnSingleCandidate：当指定Bean在SpringIoc容器内只有一个，或者虽然有多个但是指定首选的Bean

@ConditionalOnWebApplication：当前项目是Web项目的条件

以上注解都是元注解@Conditional演变而来的，根据不同的条件对应创建以上的具体条件注解。

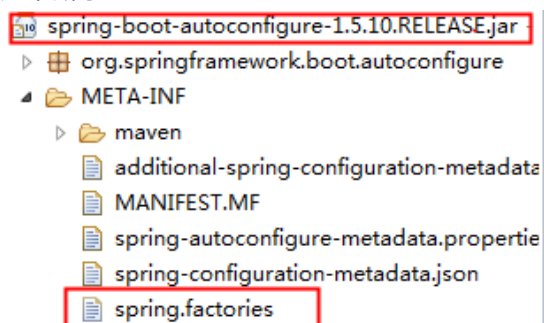
到目前为止我们还没有完成自动化配置starter，我们需要了解SpringBoot 运作原理后才可以完成后续编码。

## Starter自动化运作原理

在注解@SpringBootApplication上存在一个开启自动化配置的注解@EnableAutoConfiguration来完成自动化配置，注解源码如下所示：

```
1 @AutoConfigurationPackage
2 @Import({EnableAutoConfigurationImportSelector.class})
3 public @interface EnableAutoConfiguration {
4     String ENABLED_OVERRIDE_PROPERTY =
5         "spring.boot.enableautoconfiguration";
6
7     Class<?>[] exclude() default {};
8
9     String[] excludeName() default {};
```

在@EnableAutoConfiguration注解内使用到了@import注解来完成导入配置的功能，而EnableAutoConfigurationImportSelector内部则是使用了SpringFactoriesLoader.loadFactoryNames方法进行扫描具有META-INF/spring.factories文件的jar包。我们可以先来看下spring-boot-autoconfigure包内的spring.factories文件内容，如下所示：



可以看到配置的结构形式是Key=>Value形式，多个Value时使用,隔开，那我们在自定义starter内也可以使用这种形式来完成，我们的目的是为了完成自动化配置，所以我们这里Key则是需要使用org.springframework.boot.autoconfigure.EnableAutoConfiguration

## 自定义spring.factories

我们在src/main/resource目录下创建META-INF目录，并在目录内添加文件spring.factories，具体内容如下所示：

```
1 #配置自定义Starter的自动化配置
```

```
2 org.springframework.boot.autoconfigure.EnableAutoConfiguration=com.example
  .HelloAutoConfiguration
```

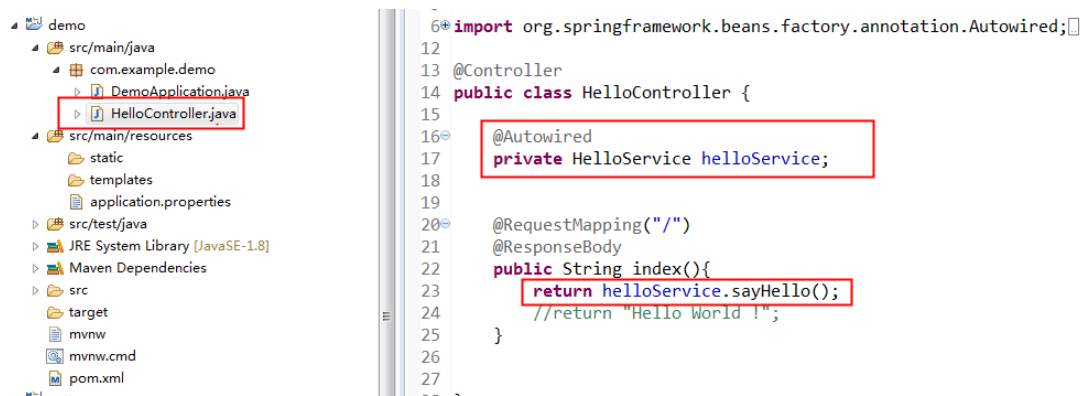
到目前为止我们的自定义starter已经配置完成，下面我们需要新建一个SpringBoot项目来测试我们的自动化配置是否已经生效。

## 创建SpringBoot测试项目

创建spring boot项目，在pom文件中增加自定义的starter依赖

```
1 <dependency>
2     <groupId>com.example</groupId>
3     <artifactId>spring-boot-starter-hello</artifactId>
4     <version>0.0.1-SNAPSHOT</version>
5 </dependency>
```

controller引入自定义starter中的service调用业务：



## 运行测试

在运行项目之前，我们打开application.properties配置文件开启debug模式，查看自动化配置的输出日志，配置内容如下所示：

```
1 #显示debug日志信息
2 debug=true
```

接下来我们启动项目，在控制台查找是否存在我们的HelloAutoConfiguration日志输出

在控制台可以看到我们的自定义starter的自动化配置已经生效了，并且根据@ConditionalOnMissingBean(HelloService.class)做出了条件注入HelloService实体bean到Springloc容器内



我们的配置生效了，到目前为止我相信大家已经明白了我们application.properties配置文件为什么可以作为统一配置入口，为什么配置后可以被对应starter所使用

