

# 1. 目的

创建一个springboot应用程序， 浏览器发送hello， 服务器接受请求并处理， 响应hello world字符串给浏览器

# 2. 工程

## POM依赖

```
<!--导入springBoot相关的依赖-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>

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

## 主程序

```
/**  
 * @SpringBootApplication 来标注一个主程序类，说明这是一个Spring Boot应用  
 **/  
  
@SpringBootApplication  
public class HelloWorldMainApplication {  
    public static void main(String[] args) {  
  
        // Spring应用启动起来  
        SpringApplication.run(HelloWorldMainApplication.class, args);  
    }  
}
```

## controller

```
@Controller  
public class HelloController {  
  
    @ResponseBody  
    @RequestMapping("/hello")  
    public String hello(){  
        return "Hello World!";  
    }  
}
```

## 打包可执行jar

### 引入新依赖

```
<!--这个插件，可以将应用打包成一个可执行的jar包-->
<build>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

## 使用命令打包

```
mvn package。会生成对应的可运行的jar包（如：springBootHello-1.0-SNAPSHOT.jar）
```

```
java -jar springBootHello-1.0-SNAPSHOT.jar
```

及时目标环境没有tomcat也没关系，因为打包的时候已经将相关环境都打包进来了[可以解压看看呀。。]

# 3. Hello探究

## 1. POM文件

父项目以前做依赖管理的。。。

```
<!--导入springBoot相关的依赖-->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>
```

父项目：

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>1.5.9.RELEASE</version>
  <relativePath>../../spring-boot-dependencies</relativePath>
</parent>
```

来真正管理springBoot应用里面的所有依赖版本：Spring boot的版本仲裁中心，以后我们导入依赖默认是不需要写版本：（没有dependencies里面的管理的依赖自然需要声明版本号）

## 2. 启动器

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

spring-boot-starter-web：

Spring-boot-starter:场景启动器，帮我们导入了web相关的依赖。Spring-boot有很多的场景启动器，如：spring-boot-starter-web/spring-boot-starter-mail

Spring-boot将所有的功能都抽取出来了，做出一个个的starters（启动器），只需要在项目里面引入这些starter相关场景的所有依赖都会导入进来，要用什么功能就导入什么场景的启动器

## 3. 主程序类

```
/**  
 * @SpringBootApplication 来标注一个主程序类，说明这是一个Spring Boot应用  
 **/  
  
@SpringBootApplication  
public class HelloWorldMainApplication {  
    public static void main(String[] args) {  
  
        // Spring应用启动起来  
        SpringApplication.run(HelloWorldMainApplication.class, args);  
    }  
}
```

@SpringBootApplication: Spring Boot应用标注在某个类上说明这个类是Spring Boot的主配置类，Spring Boot就应该运行这个类的main方法来启动Spring Boot应用。。。

```
@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 {
```

@SpringBootConfiguration: Spring Boot的配置类，标注在某个类上，表示这是一个Spring Boot的配置类

@Configuration: Spring的配置类，参考下文，配置类也是组件（@Component）

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public @interface Configuration {
```

@EnableAutoConfiguration: 开启自动配置功能；以前需要配置的东西，Spring Boot帮我们自动配置，就是告诉Spring Boot开启自动配置功能，这样自动配置才能生效啊！！！

```
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

@AutoConfigurationPackage: 自动配置包

```
@Import(AutoConfigurationPackages.Registrar.class)
public @interface AutoConfigurationPackage {
```

@Import: Spring的底层注解，给容器中导入一些组件。具体内容由对应的类指定

所有这里的主要目的是将主配置类（@AutoConfigurationPackage标注的类）的所在包及下面所有子包里面的所有组件扫描到Spring容器中；

```
130     public void registerBeanDefinitions(AnnotationMetadata metadata, metadata: StandardAnnotationMetadata@2878
131             BeanDefinitionRegistry registry) { registry: "org.springframework.beans.factory.support.DefaultListab
132             register(registry, new PackageImport(metadata).getPackageName()); metadata: StandardAnnotationMetadata@28
133         }
134
135     @Override
136
```

Variables

```
> {} this = {AutoConfigurationPackages$Registrar@2880}
- metadata = {StandardAnnotationMetadata@2878}
  - annotations = {Annotation[1]@2884}
    - {} 0 = {$Proxy7@2888}
      - h = {AnnotationInvocationHandler@2889}
        - type = {Class@1676} "interface org.springframework.boot.autoconfigure.SpringBootApplication" ... Navigate
        - memberValues = {LinkedHashMap@2890} size = 4
        - memberMethods = {Method[4]@2891}
      - nestedAnnotationsAsMap = true
  - introspectedClass = {Class@475} "class com.luoyu.HelloWorldMainApplication" ... Navigate
- registry = {DefaultListableBeanFactory@2879} "org.springframework.beans.factory.support.DefaultListableBeanFactory@60b4beb4: defining beans [org.s... View
```

```
130     public void registerBeanDefinitions(AnnotationMetadata metadata, metadata: StandardAnnotationMetadata@2878
131             BeanDefinitionRegistry registry) { registry: "org.springframework.beans.factory.support.DefaultListab
132             register(registry, new PackageImport(metadata).getPackageName()); metadata: StandardAnnotationMetadata@28
133         }
134
135     @Override
136
```

Evaluate

Expression:

```
new PackageImport(metadata).getPackageName()
```

Result:

```
result = "com.luoyu"
  value = {char[9]}@3599 [c, o, m, ., l, u, o, y, u]
  hash = 0
```

registerBeanDefinitions: 注册一些bean编译信息

AnnotationMetadata: 注解标注的一些元信息

@Import(EnableAutoConfigurationImportSelector.class), 给容器中导入一些组件

(EnableAutoConfigurationImportSelector.class/开启自动配置导入选择器, 确定导入哪些组件的选择器)

```
public class EnableAutoConfigurationImportSelector
    extends AutoConfigurationImportSelector {
    @Override
    protected boolean isEnabled(AnnotationMetadata metadata) {
        if (getClass().equals(EnableAutoConfigurationImportSelector.class)) {
            return getEnvironment().getProperty(
                EnableAutoConfiguration.ENABLED_OVERRIDE_PROPERTY, Boolean.class,
                defaultValue: true);
        }
        return true;
    }
}
```

```
private ResourceLoader resourceLoader;
@Override
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
        configurations = removeDuplicates(configurations);
        configurations = sort(configurations, autoConfigurationMetadata);
    }
}
```

将所有需要导入的组件以全类名的方式返回；这些组件就会被添加到容器中去

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader);
        AnnotationAttributes attributes = getAttributes(annotationMetadata);
        List<String> configurations = getCandidateConfigurations(annotationMetadata,
            attributes);
    }
}
```

Variables

- ↳ this = {EnableAutoConfigurationImportSelector@2196}
- ↳ annotationMetadata = {StandardAnnotationMetadata@2197}
- ↳ annotations = {Annotation[1]@2811}
- ↳ 0 = {\$Proxy7@2813}
  - ↳ h = {AnnotationInvocationHandler@2814}
    - ↳ type = {Class@1706} "interface org.springframework.boot.autoconfigure.SpringBootApplication" ... Navigate
    - ↳ memberValues = {LinkedHashMap@2815} size = 4
    - ↳ memberMethods = {Method[4]@2816}
- ↳ nestedAnnotationsAsMap = true
- ↳ introspectedClass = {Class@475} "class com.luoyu.HelloWorldMainApplication" ... Navigate

```

public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    try {
        AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMetadataLoader
            .loadMetadata(this.beanClassLoader); // beanClassLoader: Launcher$AppClassLoader@2184
        AnnotationAttributes attributes = getAttributes(annotationMetadata); // attributes: size = 2
        List<String> configurations = getCandidateConfigurations(annotationMetadata, // annotationMetadata: Standard
            attributes); // attributes: size = 2
        configurations = removeDuplicates(configurations); // configurations: size = 96
        configurations = sort(configurations, autoConfigurationMetadata);
        Set<String> exclusions = getExclusions(annotationMetadata, attributes);
        checkExcludedClasses(configurations, exclusions);
        configurations.removeAll(exclusions);
    }
}

```

Variables

```

+ <--> this = {EnableAutoConfigurationImportSelector@2185}
+ <--> annotationMetadata = {StandardAnnotationMetadata@2186}
+ <--> autoConfigurationMetadata = {AutoConfigurationMetadataLoader$PropertiesAutoConfigurationMetadata@2810}
+ <--> attributes = {AnnotationAttributes@2813} size = 2
+ <--> configurations = {ArrayList@2818} size = 96
    > <--> 0 = "org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration"
    > <--> 1 = "org.springframework.boot.autoconfigure.aop.AopAutoConfiguration"
    > <--> 2 = "org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration"
    > <--> 3 = "org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration"
    > <--> 4 = "org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration"
    > <--> 5 = "org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration"
    > <--> 6 = "org.springframework.boot.autoconfigure.cloud.CloudAutoConfiguration"

```

会给容器中导入非常多的自动配置类（xxxAutoConfiguration）,就是给容器中导入这个场景需要的所有组件，并配置好这些组件。。。如何获取到的呢？

```

SpringFactoriesLoader.loadFactoryNames(
    EnableAutoConfiguration.class, getClassLoader());

```

```

public static List<String> loadFactoryNames(Class<?> factoryClass, ClassLoader classLoader) {
    String factoryClassName = factoryClass.getName();
    try {
        Enumeration<URL> urls = (classLoader != null ? classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        List<String> result = new ArrayList<String>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            Properties properties = PropertiesLoaderUtils.loadProperties(new UrlResource(url));
            String factoryClassNames = properties.getProperty(factoryClassName);
            result.addAll(Arrays.asList(StringUtils.commaDelimitedListToStringArray(factoryClassNames)));
        }
    }
    return result;
}

```

使用classLoader类加载器获取一个资源，然后将资源当成properties文件，然后从properties中拿出来factoryClassNames。

Spring Boot在启动的时候从类路径下META-INF/spring.factories中获取EnableAutoConfiguration指定的值，将这些值作为自动配置类导入到容器中，然后自动配置类就会生效，帮我们进行自动配置工作。。。

有了自动配置类，免去了我们手动编写配置注入功能组件等的工作。。。

## 4. 使用Spring Initializer创建

使用idea 的spring initializr进行创建。向导会联网创建Spring Boot项目

- 主程序已经生成好了，我们只需要我们自己的逻辑
- resources文件目录结构：
  - static：保存所有静态资源：js/css/images
  - templates：保存所有的模版页面
  - application.properties：Spring Boot应用的配置文件

### 4.1 配置文件

SpringBoot使用一个全局的配置文件，配置文件名是固定的  
(application.properties/application.yml)

配置文件的作用：修改SpringBoot自动配置的默认值，SpringBoot在底层都给我们自动配置好了

YAML (YAML Ain't Markup Language) ，以数据为中心，比json/xml等更适合做配置文件

- YAML A Markup Language:是一个标记语言
- YAML isn't Markup Language:不是一个标记语言

```
server:  
  port: 8081
```

## 4.2 基本语法

k:(空格)v：表示一对键值对（空格必须有）

以空格的缩进来控制层级关系，只要左对齐的一列数据，都是同一个层级的

属性和值也是大小写敏感。

字面量：普通的值（数字，字符串，布尔）

- 字符串默认不用加单引号或者双引号
- 双引号：不会转义字符串里面的特殊字符；特殊字符会作为本身想表示的意思
- 单引号：会转义字符，特殊字符就是一个普通的字符串数据

对象/Map（属性和值）（键值对）：

```
server:  
  port: 8081  
  address: /abc/def
```

行内写法：

```
server: {port: 8081, address: /abc/def}
```

数据（List/Set）：用-表示数组中的一个元素

```
pets:  
  - cat  
  - dog  
  - pig  
  
pets: [cat,dog,pig]
```

## 4.3 配置文件注入

### 配置文件

```
person:  
    lastName: zhangsan  
    age: 18  
    boss: false  
    birth: 2017/12/12  
    maps: {k1: v1, k2: v2}  
    lists:  
        - lisi  
        - zhaoliu  
dog:  
    name: xiaogou  
    age: 2
```

### Person

```
/**  
 * 将配置文件中配置的每一个属性的值，映射到这个组建中  
 * @ConfigurationProperties:告诉SpringBoot将本类中的所有属性和配置文件中的相关  
配置进行绑定；  
 *           prefix = "person" :配置文件中那个下面的所有属性进行一一映射  
 * 只有这个组件是容器中的组件，才能使用容器提供的功能ConfigurationProperties  
 **/  
  
@Component  
@ConfigurationProperties(prefix = "person")  
public class Person {  
  
    private String lastName;  
    private String age;  
    private Boolean boss;  
    private Date birth;
```

```
    private Map<String, Object> maps;
    private List<Object> list;
    private Dog dog;
}
```

## 单元测试和配置文件处理器

```
<!--Spring Boot进行单元测试的模块-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<!--导入配置文件处理器， 配置文件进行绑定就会有提示-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-processor</artifactId>
    <optional>true</optional>
</dependency>
```

## 单元测试验证

```
/**
 * SpringBoot单元测试
 * 可以在测试期间很方便的类似编码一样进行自动注入等容器的功能
 */
@SpringBootTest
class SpringInitializrDemoApplicationTests {

    @Autowired
    Person person;

    @Test
    void contextLoads() {
        System.out.println(person);
    }
}
```

}

}

# 其他

1. 查看ssh公钥方法:**cd ~/.ssh    cat id\_rsa.pub**

2.