

SpringBoot基础

本站：IT楠老师 公众号：IT楠说java QQ群：1083478826

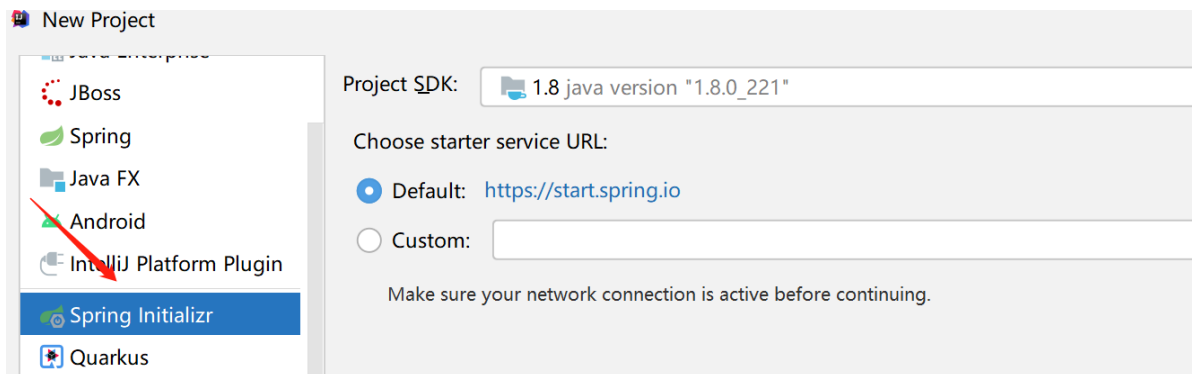
一句话，他就是个框架，先看看怎么用，然后体会用了有什么好处！

一、SpringBoot快速入门

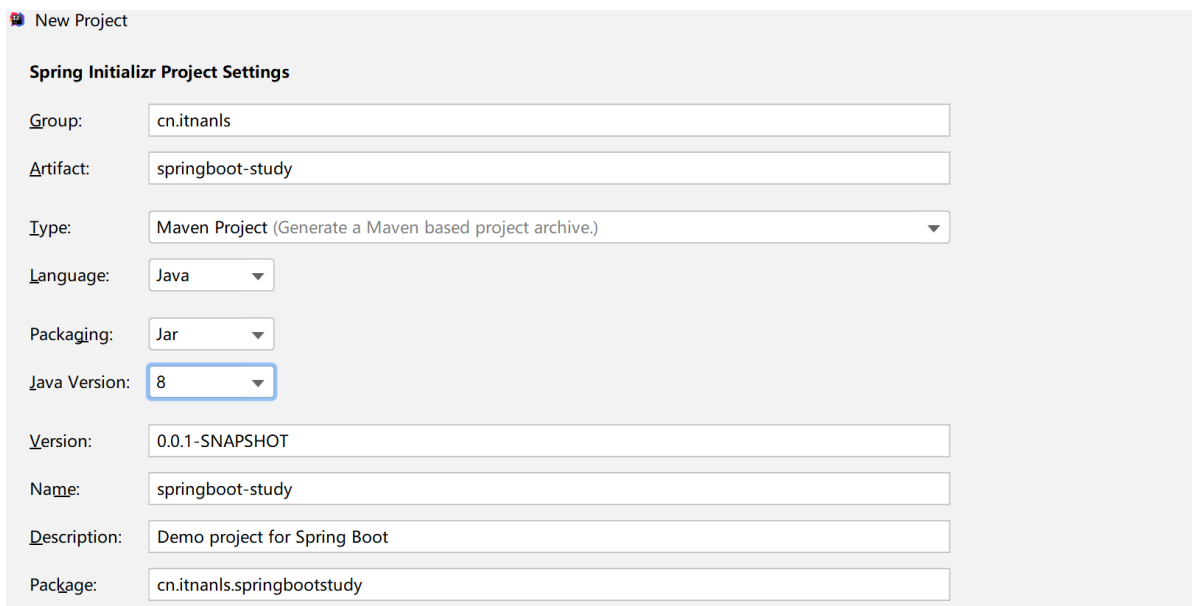
1、创建项目

看看idea能怎么优雅的创建一个springboot项目

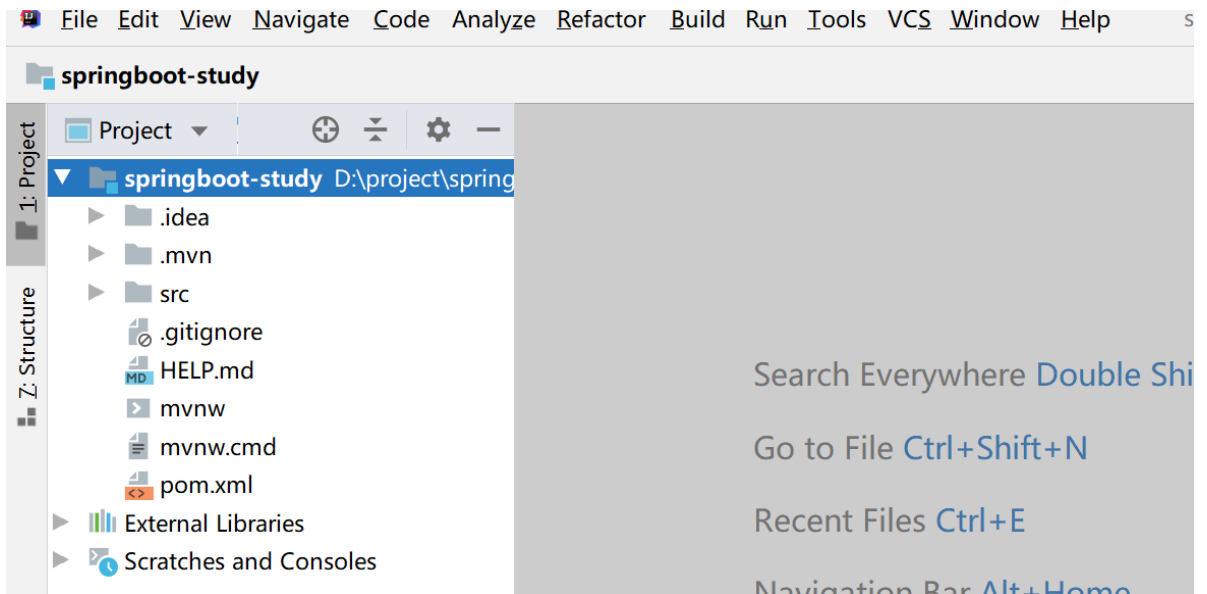
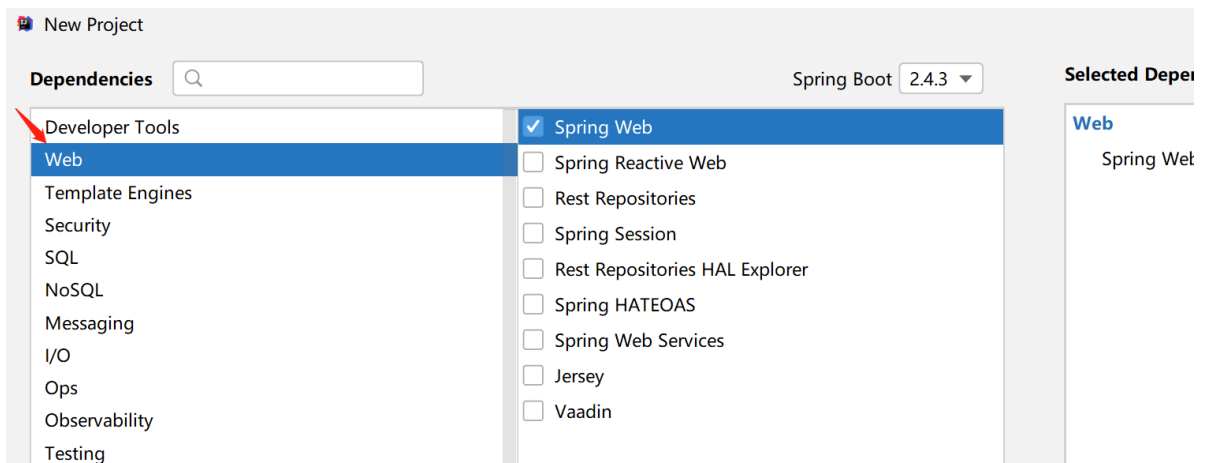
(1) 创建一个项目选择spring initializr



(2) 工程名spring-boot-test



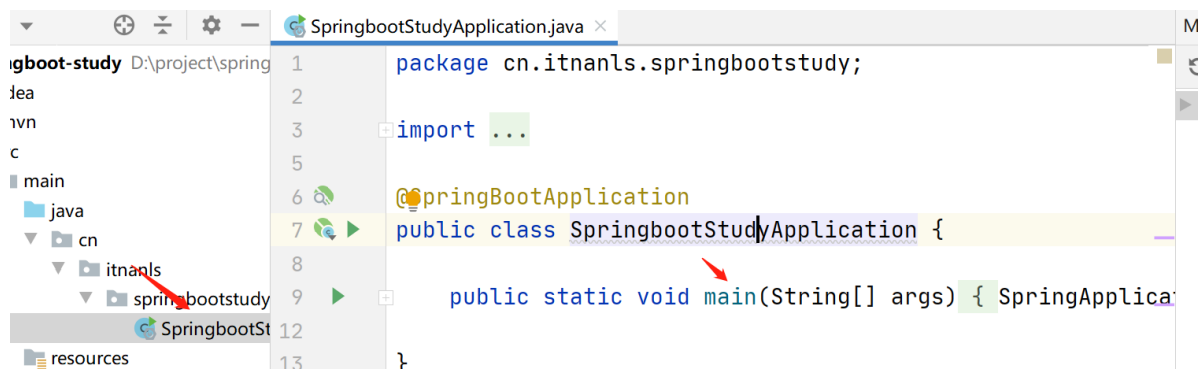
(3) 勾选需要的组件



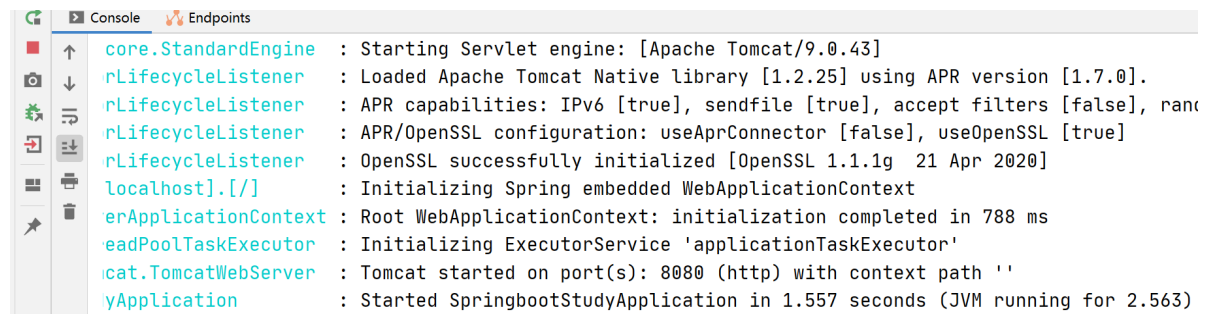
写一个controller

```
1  /**
2   * @author IT楠老师
3   * @date 2020/6/2
4   */
5  @Controller
6  public class UserController {
7
8      @RequestMapping("/")
9      @ResponseBody
10     public String test(){
11         return "hello springboot";
12     }
13
14 }
```

运行引导类的main方法



结果ok!



其他的都一样是不是很快是不是很爽!

2、分析一下

1、springboot工程本质就是个maven工程，他有自己的依赖和自己的插件



重点：都一样!

思考你能不能从一个空工程建立一个springboot工程

- (1) 建一个maven工程
- (2) 引入依赖

1 <dependencies>

```

2      <dependency>
3          <groupId>org.springframework.boot</groupId>
4          <artifactId>spring-boot-starter-web</artifactId>
5      </dependency>
6
7      <dependency>
8          <groupId>org.springframework.boot</groupId>
9          <artifactId>spring-boot-devtools</artifactId>
10         <optional>true</optional>
11         <scope>true</scope>
12     </dependency>
13 </dependencies>
14
15 <build>
16     <plugins>
17         <plugin>
18             <groupId>org.springframework.boot</groupId>
19             <artifactId>spring-boot-maven-plugin</artifactId>
20             <configuration>
21                 <fork>true</fork>
22             </configuration>
23         </plugin>
24     </plugins>
25 </build>

```

(3) 写一个main方法即可，main方法是固定的

```

1 @SpringBootApplication
2 public class SpringbootStudyApplication {
3     public static void main(String[] args) {
4         SpringApplication.run(SpringbootStudyApplication.class, args);
5     }
6 }

```

2、SpringBoot工程热部署

(1) 我们在开发中反复修改类、页面等资源，每次修改后都是需要重新启动才生效，这样每次启动都很麻烦，浪费了大量的时间，我们可以在修改代码后不重启就能生效，在 pom.xml 中添加如下配置就可以实现这样的功能，我们称之为热部署。

```

1 <!--热部署配置-->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-devtools</artifactId>
5     <optional>true</optional>
6     <scope>true</scope>
7 </dependency>
8
9
10 <build>
11     <plugins>
12         <plugin>
13             <groupId>org.springframework.boot</groupId>
14             <artifactId>spring-boot-maven-plugin</artifactId>

```

```

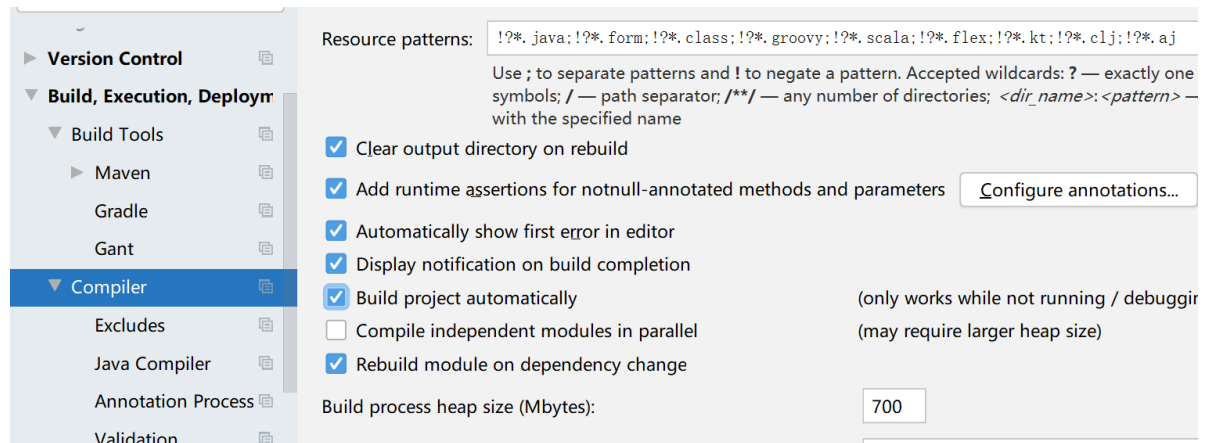
15         <configuration>
16             <fork>true</fork>
17         </configuration>
18     </plugin>
19 </plugins>
20 </build>

```

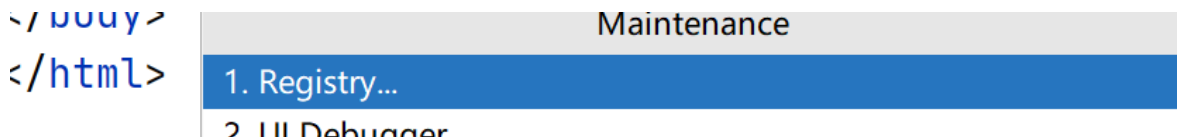
(2) IDEA中配置

当我们修改了类文件后，idea不会自动编译，得修改idea设置。

File-Settings-Compiler-Build Project automatically



ctrl + shift + alt + / ,选择Registry,勾上 Compiler autoMake allow when app running



思考和我们以前的项目有什么不同，直观的两点

1. 以main方法启动，貌似不需要tomcat
2. 没有什么配置文件，xml都没有了
3. 没有了配置文件我们怎么组装呢？

二、SpringBoot基础分析

1、丢了的tomcat

内嵌tomcat原理

pom

```
1 <dependencies>
2   <dependency>
3     <groupId>org.apache.tomcat.embed</groupId>
4     <artifactId>tomcat-embed-core</artifactId>
5     <version>9.0.43</version>
6   </dependency>
7   <dependency>
8     <groupId>org.apache.tomcat.embed</groupId>
9     <artifactId>tomcat-embed-jasper</artifactId>
10    <version>9.0.43</version>
11  </dependency>
12  <dependency>
13    <groupId>org.springframework</groupId>
14    <artifactId>spring-webmvc</artifactId>
15    <version>5.2.9.RELEASE</version>
16  </dependency>
17 </dependencies>
18 <build>
19   <plugins>
20     <plugin>
21       <groupId>org.apache.maven.plugins</groupId>
22       <artifactId>maven-compiler-plugin</artifactId>
23       <version>3.8.1</version>
24       <configuration>
25         <target>1.8</target>
26         <source>1.8</source>
27         <encoding>utf-8</encoding>
28       </configuration>
29     </plugin>
30   </plugins>
31 </build>
```

一个main方法

```
1 /**
2  * @author zn
3  * @date 2021/3/4
4  */
5 public class SpringBootApplication {
6
7     public static void run() {
8         Tomcat tomcat = new Tomcat();
9         Context context = tomcat.addWebapp("/app", "D:/img/");
10    }
```

```

11     AnnotationConfigWebApplicationContext applicationContext
12         = new AnnotationConfigWebApplicationContext();
13     applicationContext.register(AppConfig.class);
14     DispatcherServlet dispatcherServlet = new
DispatcherServlet(applicationContext);
15
16     Wrapper wrapper = tomcat.addServlet("/app", "app",
dispatcherServlet);
17     wrapper.addMapping("/*");
18
19     try {
20         Connector connector = new Connector();
21         connector.setPort(8080);
22         tomcat.getService().addConnector(connector);
23         tomcat.start();
24         tomcat.getServer().await();
25     } catch (LifecycleException e) {
26         e.printStackTrace();
27     }
28 }
29
30 }

```

```

1 @ComponentScan("cn.itnanls")
2 public class AppConfig {
3 }

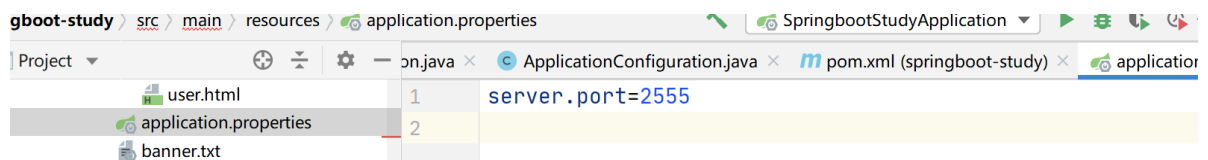
```

2、配置文件

1、真的没有配置文件了吗？其实也有



例如：怎么配一个端口呢？



2、猜一猜，估计是根据约定以编码的方式写了代码里边

同时根据上边的配置我们大致明白了，对于配置而言一般是如下情况

```

1 if(配置里有){
2     return 配置;
3 }else{
4     return 默认;
5 }

```

3、思考一下spring里边的bean是怎么注入的？

通常：spring注入bean有几种方式

可以直接注入Bean (User) 也可以注入factorybean (UserFactorybean)

(1) 配置文件 不说了 <bean> 标签。

(2) 扫包

@Component

@Configuration @Bean

@Service @Controller @Repository

@Import

```
1 // 直接引入一个类
2 @Import(User.class)
3 // 引入一个配置类
4 @Import(UserConfiguration.class)
5 // 引入一个实现了ImportSelector接口的类，返回全类型数组，数组的类都可以引入
6 @Import(UserSelector.class)
7
8
9 public class UserSelector implements ImportSelector {
10     @Override
11     public String[] selectImports(AnnotationMetadata importingClassMetadata)
12     {
13         return new String[]{"cn.itnanls.User"};
14     }
15 }
```

但是前提是你扫描的包下有这些注解！

那你引入的第三方的组件，你的工程并没有扫描到，怎么用啊？

1、自己写配置类，当然可以

2、但是事实上很多组件连配置类都不用写，为什么？

重要特性----自动装配 (starter) 后边讲

配置文件真的可以没有吗？

1、你想改一个端口，真没要修改代码吗？

2、框架知道你的数据库连接吗？

3、修改连接真的要修改代码，重新编译吗？

配置文件真的都有用吗？

1. spring的配置文件太重，我们不需要的是哪些不会经常变动的而已。

2. 但事实上，我们使用的大量组件都需要和spring结合，只要结合就需要注入自己的bean。

3. 到了后来你会发现，配置文件里有了fastjson的配置，druid的配置，mybatis的配置，springmvc的配置....

4. 然而，这些配置往往配置一次以后很少会在此改变。
5. 和装配相关的配置往往没有那么重要！

那合理的做法是什么呢？

- 1、我们预留出一个单独的配置文件，去配置那些必须的或是需要修改的配置。
- 2、其他的配置我们使用java方式去配置，如果组件能够自己自动配置就更好了，springboot也是这么做的。

约定大于配置

mysql启动的端口你知道是多少吗？

tomcat启动的端口你知道吗？

为什么springboot启动默认就在8080端口，你配置了吗？

因为是默认，因为约定俗成。

当所有组件都有默认的配置，都有你知我知的约定，那他具体怎么和spring整合，我们就明白了。

举个例子，比如整合thymeleaf，默认他们试图解析器前缀是"classpath:/templates/"，后缀是".html"

那么在整合时，我们即使什么也不配，我也知道。

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

加一个依赖就搞定了。

但是有一天我想修改他的配置怎么办，有办法。

springboot是提供有了一个配置文件的。

SpringBoot是 **基于约定** 的，所以很多配置都有默认值，但如果想使用自己的配置替换默认配置的话，就可以使用application.properties或者application.yml（application.yaml）进行配置。

SpringBoot默认会从Resources目录下加载application.properties或application.yml（application.yaml）文件

其中，application.properties文件是键值对类型的文件，之前一直在使用，所以此处不在对properties文件的格式进行阐述。除了properties文件外，SpringBoot还可以使用yaml文件进行配置，下面对yaml文件进行讲解。

(1) 配置文件语法

application.yml配置文件

yml配置文件简介

YML文件格式是YAML (YAML Aint Markup Language)编写的文件格式，YAML是一种直观的能够被电脑识别的数据序列化格式，并且容易被人类阅读，容易和脚本语言交互的，可以被支持YAML库的不同的编程语言程序导入，比如：C/C++，Ruby，Python，Java，Perl，C#，PHP等。YML文件是以数据为核心的，比传统的xml方式更加简洁。

YML文件的扩展名可以使用.yml或者.yaml。

yml配置文件的语法

配置普通数据

- 语法：key: value
- 示例代码：
- name: haohao
- 注意：value之前有一个空格

配置对象数据

- 语法：
key:
key1: value1
key2: value2
或者：
key: {key1: value1, key2: value2}

- 示例代码：

```
1 person:  
2   name: haohao  
3   age: 31  
4   addr: beijing  
5  
6 #或者  
7  
8 person: {name: haohao, age: 31, addr: beijing}
```

- 注意：key1前面的空格个数不限定，在yml语法中，相同缩进代表同一个级别

配置Map数据

同上面的对象写法

配置数组 (List、Set) 数据

- 语法：
key:
- value1
- value2
或者：

key: [value1,value2]

- 示例代码:

```
1  city:
2    - beijing
3    - tianjin
4    - shanghai
5    - chongqing
6
7  #或者
8
9  city: [beijing,tianjin,shanghai,chongqing]
10
11 #集合中的元素是对象形式
12 student:
13   - name: zhangsan
14     age: 18
15     score: 100
16   - name: lisi
17     age: 28
18     score: 88
19   - name: wangwu
20     age: 38
21     score: 90
```

- 注意: value1与之间的 - 之间存在一个空格

(2) 常用配置

```
1  # EMBEDDED SERVER CONFIGURATION (ServerProperties)
2  server.port=8080 # Server HTTP port.
3  server.servlet.context-path= demo # Context path of the application.
4  server.servlet.path=/ # Path of the main dispatcher servlet.
5
6
7  # SPRING MVC (WebMvcProperties)
8  spring.mvc.view.prefix= # Spring MVC view prefix.
9  spring.mvc.view.suffix= # Spring MVC view suffix.
10
11 # DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
12 spring.datasource.driver-class-name= # Fully qualified name of the JDBC
13   driver. Auto-detected based on the URL by default.
14 spring.datasource.password= # Login password of the database.
15 spring.datasource.url= # JDBC URL of the database.
16 spring.datasource.username= # Login username of the database.
```

我们可以通过配置application.properties 或者 application.yml 来修改SpringBoot的默认配置

例如:

application.properties文件

```
1  server.port=8888
2  server.servlet.context-path=demo
```

application.yml文件

```
1 server:
2   port: 8888
3   servlet:
4     context-path: /demo
```

(3) 读取配置

使用注解@Value

我们可以通过@Value注解将配置文件中的值映射到一个Spring管理的Bean的字段上

例如:

application.properties配置如下:

```
1 person:
2   name: zhangsan
3   age: 18
```

或者, application.yml配置如下:

```
1 person:
2   name: zhangsan
3   age: 18
```

实体Bean代码如下:

```
1 @Controller
2 public class QuickStartController {
3
4     @Value("${person.name}")
5     private String name;
6     @Value("${person.age}")
7     private Integer age;
8
9
10    @RequestMapping("/quick")
11    @ResponseBody
12    public String quick(){
13        return "springboot 访问成功! name="+name+", age="+age;
14    }
15
16 }
```

浏览器访问地址: <http://localhost:8080/quick> 结果如下:

使用注解@ConfigurationProperties映射

通过注解@ConfigurationProperties(prefix="配置文件中的key的前缀")可以将配置文件中的配置自动与实体进行映射

application.properties配置如下：

```
1 person:
2   name: zhangsan
3   age: 18
```

或者，application.yml配置如下：

```
1 person:
2   name: zhangsan
3   age: 18
```

实体Bean代码如下：

```
1 @Controller
2 @ConfigurationProperties(prefix = "person")
3 public class QuickStartController {
4
5     private String name;
6     private Integer age;
7
8     @RequestMapping("/quick")
9     @ResponseBody
10    public String quick(){
11        return "springboot 访问成功! name="+name+",age="+age;
12    }
13
14    public void setName(String name) {
15        this.name = name;
16    }
17
18    public void setAge(Integer age) {
19        this.age = age;
20    }
21 }
```

浏览器访问地址：<http://localhost:8080/quick> 结果如下：

注意：使用@ConfigurationProperties方式可以进行配置文件与实体字段的自动映射，但需要字段必须提供set方法才可以，而使用@Value注解修饰的字段不需要提供set方法

3、依赖分析

(1) 分析spring-boot-starter-parent

按住Ctrl点击pom.xml中的spring-boot-starter-parent，跳转到了spring-boot-starter-parent的pom.xml，xml配置如下（只摘抄了部分重点配置）：

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-dependencies</artifactId>
4   <version>2.4.3</version>
5 </parent>
```

发现parent中还有parent (spring-boot-dependencies)

```
1 <parent>
2   <groupId>org.springframework.boot</groupId>
3   <artifactId>spring-boot-dependencies</artifactId>
4   <version>2.4.3</version>
5 </parent>
6 <artifactId>spring-boot-starter-parent</artifactId>
7 <packaging>pom</packaging>
8 <name>spring-boot-starter-parent</name>
9 <description>Parent pom providing dependency and plugin management for
applications built with Maven</description>
10 <properties>
11   <java.version>1.8</java.version>
12   <resource.delimiter>@</resource.delimiter>
13   <maven.compiler.source>${java.version}</maven.compiler.source>
14   <maven.compiler.target>${java.version}</maven.compiler.target>
15   <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
16   <project.reporting.outputEncoding>UTF-
8</project.reporting.outputEncoding>
17 </properties>
18 <url>https://spring.io/projects/spring-boot</url>
19 <licenses>
20   <license>
21     <name>Apache License, Version 2.0</name>
22     <url>https://www.apache.org/licenses/LICENSE-2.0</url>
23   </license>
24 </licenses>
25 <developers>
26   <developer>
27     <name>Pivotal</name>
28     <email>info@pivotal.io</email>
29     <organization>Pivotal Software, Inc.</organization>
30     <organizationUrl>https://www.spring.io</organizationUrl>
31   </developer>
32 </developers>
33 <scm>
34   <url>https://github.com/spring-projects/spring-boot</url>
35 </scm>
36 <build>
37   <resources>
38     <resource>
39       <directory>${basedir}/src/main/resources</directory>
40       <filtering>true</filtering>
41       <includes>
42         <include>**/application*.yml</include>
43         <include>**/application*.yaml</include>
44         <include>**/application*.properties</include>
45       </includes>
46     </resource>
```

```

47     <resource>
48         <directory>${basedir}/src/main/resources</directory>
49         <excludes>
50             <exclude>**/application*.yml</exclude>
51             <exclude>**/application*.yaml</exclude>
52             <exclude>**/application*.properties</exclude>
53         </excludes>
54     </resource>
55 </resources>

```

按住Ctrl点击pom.xml中的spring-boot-starter-dependencies，跳转到了spring-boot-starter-dependencies的pom.xml，xml配置如下（只摘抄了部分重点配置）：

```

1  <properties>
2      <activemq.version>5.15.12</activemq.version>
3      <commons-codec.version>1.14</commons-codec.version>
4      <commons-dbcp2.version>2.7.0</commons-dbcp2.version>
5      <commons-lang3.version>3.10</commons-lang3.version>
6      <commons-pool.version>1.6</commons-pool.version>
7      <commons-pool2.version>2.8.0</commons-pool2.version>
8      <ehcache.version>2.10.6</ehcache.version>
9      <ehcache3.version>3.8.1</ehcache3.version>
10     <elasticsearch.version>7.6.2</elasticsearch.version>
11     <freemarker.version>2.3.30</freemarker.version>
12     <hibernate.version>5.4.15.Final</hibernate.version>
13     <hibernate-validator.version>6.1.5.Final</hibernate-validator.version>
14     <hikaricp.version>3.4.5</hikaricp.version>
15     <jedis.version>3.3.0</jedis.version>
16     <junit.version>4.13</junit.version>
17     <junit-jupiter.version>5.6.2</junit-jupiter.version>
18     <kafka.version>2.5.0</kafka.version>
19     <maven-compiler-plugin.version>3.8.1</maven-compiler-plugin.version>
20     <maven-dependency-plugin.version>3.1.2</maven-dependency-plugin.version>
21     <mysql.version>8.0.20</mysql.version>
22     <neo4j-ogm.version>3.2.11</neo4j-ogm.version>
23     <netty.version>4.1.49.Final</netty.version>
24     <netty-tcnative.version>2.0.30.Final</netty-tcnative.version>
25     <quartz.version>2.3.2</quartz.version>
26     <servlet-api.version>4.0.1</servlet-api.version>
27     <slf4j.version>1.7.30</slf4j.version>
28     <snakeyaml.version>1.26</snakeyaml.version>
29     <solr.version>8.5.1</solr.version>
30     ...
31 </properties>
32 <dependencyManagement>
33     ...
34 </dependencyManagement>

```

从上面的spring-boot-starter-dependencies的pom.xml中我们可以发现，一部分坐标的版本、依赖管理、插件管理已经定义好，所以我们的SpringBoot工程继承spring-boot-starter-parent后已经具备版本锁定等配置了。所以起步依赖的作用就是进行依赖的传递。

springboot帮助我们整合了市场上绝大多数多的常用组件，并且使用了组合适版本，尽量规避了因版本不兼容而导致的问题。

(2) 分析spring-boot-starter-web

按住Ctrl点击pom.xml中的spring-boot-starter-web，跳转到了spring-boot-starter-web的pom.xml，xml配置如下（只摘抄了部分重点配置）：

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd"
  xmlns="http://maven.apache.org/POM/4.0.0"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
4   <!-- This module was also published with a richer model, Gradle metadata,
  -->
5   <!-- which should be used instead. Do not delete the following line which
  -->
6   <!-- is to indicate to Gradle or any Gradle module metadata file consumer
  -->
7   <!-- that they should prefer consuming it instead. -->
8   <!-- do_not_remove: published-with-gradle-metadata -->
9   <modelVersion>4.0.0</modelVersion>
10  <groupId>org.springframework.boot</groupId>
11  <artifactId>spring-boot-starter-web</artifactId>
12  <version>2.3.0.RELEASE</version>
13  <name>spring-boot-starter-web</name>
14  <description>Starter for building web, including RESTful, applications
  using Spring MVC. Uses Tomcat as the default embedded
  container</description>
15  <url>https://spring.io/projects/spring-boot</url>
16  <organization>
17    <name>Pivotal Software, Inc.</name>
18    <url>https://spring.io</url>
19  </organization>
20  <licenses>
21    <license>
22      <name>Apache License, Version 2.0</name>
23      <url>http://www.apache.org/licenses/LICENSE-2.0</url>
24    </license>
25  </licenses>
26  <developers>
27    <developer>
28      <name>Pivotal</name>
29      <email>info@pivotal.io</email>
30      <organization>Pivotal Software, Inc.</organization>
31      <organizationUrl>https://www.spring.io</organizationUrl>
32    </developer>
33  </developers>
34  <scm>
35    <connection>scm:git:git://github.com/spring-projects/spring-
  boot.git</connection>
36    <developerConnection>scm:git:ssh://git@github.com/spring-
  projects/spring-boot.git</developerConnection>
37    <url>https://github.com/spring-projects/spring-boot</url>
38  </scm>
39  <issueManagement>
40    <system>GitHub</system>
41    <url>https://github.com/spring-projects/spring-boot/issues</url>
```



```
42 </issueManagement>
43 <dependencyManagement>
44   <dependencies>
45     <dependency>
46       <groupId>org.springframework.boot</groupId>
47       <artifactId>spring-boot-dependencies</artifactId>
48       <version>2.3.0.RELEASE</version>
49       <type>pom</type>
50       <scope>import</scope>
51     </dependency>
52   </dependencies>
53 </dependencyManagement>
54 <dependencies>
55   <dependency>
56     <groupId>org.springframework.boot</groupId>
57     <artifactId>spring-boot-starter</artifactId>
58     <version>2.3.0.RELEASE</version>
59     <scope>compile</scope>
60   </dependency>
61   <dependency>
62     <groupId>org.springframework.boot</groupId>
63     <artifactId>spring-boot-starter-json</artifactId>
64     <version>2.3.0.RELEASE</version>
65     <scope>compile</scope>
66   </dependency>
67   <dependency>
68     <groupId>org.springframework.boot</groupId>
69     <artifactId>spring-boot-starter-tomcat</artifactId>
70     <version>2.3.0.RELEASE</version>
71     <scope>compile</scope>
72   </dependency>
73   <dependency>
74     <groupId>org.springframework</groupId>
75     <artifactId>spring-web</artifactId>
76     <scope>compile</scope>
77   </dependency>
78   <dependency>
79     <groupId>org.springframework</groupId>
80     <artifactId>spring-webmvc</artifactId>
81     <scope>compile</scope>
82   </dependency>
83 </dependencies>
84 </project>
```

从上面的spring-boot-starter-web的pom.xml中我们可以发现，spring-boot-starter-web就是将web开发要使用的spring-web、spring-webmvc等坐标进行了“打包”，这样我们的工程只要引入spring-boot-starter-web起步依赖的坐标就可以进行web开发了，同样体现了依赖传递的作用。

每一项都可以点进去看看！

三、web配置

1、自定义欢迎页

默认回去static中找index.html

```
1 classpath:/static/index.html
2 classpath:/public/index.html
```

2、配置错误页

在static目录下新建error目录

放入对应的错误页面就行了如404.html 500.html

为什么放在这里就行，应为约定

3、自定义favicon

浏览器左上角的图标可以放在静态资源下static中，

[随便找个网站http://www.bitbug.net/生成了放进去就行了](http://www.bitbug.net/)

体会一点感受：

约定的重要性，你我都服从约定，很多代码就不用写了。

你知道垃圾要扔到垃圾箱，那么处理垃圾就变得简单了。

spring做的就是制定垃圾要扔到垃圾箱的规则。

4、定制banner

<http://patorjk.com/software/taag/>

<https://www.degraeve.com/img2txt.php>

创建Banner文件

`src/main/resource/banner.txt`

```
1  ${AnsiColor.BRIGHT_RED}
2  //////////////////////////////////////
3  //                                _ooOoo_                                //
4  //                                o8888888o                                //
5  //                                88" . "88                                //
6  //                                (| ^_^ |)                                //
7  //                                O\  =  /O                                //
8  //                                _____\`---'\`_____/                                //
9  //                                .\" \\\|      /\\| \".                                //
10 //                                /  \\\||| :  |||\\ \                                //
11 //                                / _||| | -:- |||| - \                                //
12 //                                | | \\\ -  /// | |                                //
13 //                                | \_|  '\`---/\'  | |                                //
```

```

14 //          \ .-\_  \`  _/_-./  //
15 //          _- \ . ' /--.--\ ` . _  //
16 //          ."" '< ` _-\<|>/_-.' >'"" .  //
17 //          | | : ` - \ ` ; \ _ / ; \ - ` : | |  //
18 //          \ \ ` - . \ _ - \ / _ - / . - ` / /  //
19 //      =====`- . _ _ - ` _ _ \ _ _ / _ _ - ` -' =====  //
20 //          `-----'  //
21 //      ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^  //
22 //          佛祖保佑      永不宕机      永无BUG  //
23 ///////////////////////////////////////////////////////////////////
24 Spring Boot Version: ${spring-boot.version}${spring-boot.formatted-version}

```

```

1          .:~::~
2          .:~::~:~::~
3          .:~::~:~::~:~::~
4          .:~::~:~::~:~::~:~::~'
5          '.:~::~:~::~:~::~:'
6          .:~::~:~::~:~::~
7          '.:~::~:~::~:~::~..
8          .:~::~:~::~:~::~:~::~
9          `.:~::~:~::~:~::~:~::~:~::~
10         .:~::~ `.:~::~:~::~:~::~'          .:~::~
11         .:~::~'      '.:~::~:~::~'          .:~::~:~::~:~::~
12         .:~::~:~::~'      .:~::~:~::~      .:~::~:~::~:~::~'.:~::~:~::~
13         .:~::~:~::~'      .:~::~:~::~      .:~::~:~::~:~::~' '.:~::~:~::~
14         .:~::~:~::~'      .:~::~:~::~:~::~:~::~'      '.:~::~:~::~
15         .:~::~:~::~'      .:~::~:~::~:~::~:~::~'      `.:~::~:~::~
16         .:~::~:~::~      .:~::~:~::~:~::~:~::~'      `.:~::~:~::~
17         .:~::~:~::~'      '.:~::~:~::~:~::~'      .:~::~:~::~
18         .:~::~:~::~'      '.:~::~:~::~'      .:~::~:~::~'~::~..

```

从上面的内容中可以看到，还使用了一些属性设置：

- `${AnsiColor.BRIGHT_RED}`：设置控制台中输出内容的颜色，可以自定义，具体参考 `org.springframework.boot.ansi.AnsiColor`
- `${spring-boot.version}`：Spring Boot的版本号
- `{spring-boot.formatted-version}`：格式化后的{spring-boot.version}版本信息

不妨自己从run方法进去看看源码：

```

1 public static void main(String[] args) {
2     SpringApplication.run(SpringbootStudyApplication.class, args);
3 }

```

原来在这里定义了。

```
1 | SpringApplicationBannerPrinter类中:
2 | static final String BANNER_LOCATION_PROPERTY = "spring.banner.location";
3 | static final String BANNER_IMAGE_LOCATION_PROPERTY =
4 | "spring.banner.image.location";
5 | static final String DEFAULT_BANNER_LOCATION = "banner.txt";
```

5、其他的服务器配置如jetty

在某个博客看到改Jetty的好处，也真是我现在开发的项目后面要用长连接

好处：

- 1、Jetty适合长连接应用，就是聊天类的长连接
- 2、Jetty更轻量级。这是相对Tomcat而言的。
- 3、jetty更灵活，体现在其可插拔性和可扩展性，更易于开发者对Jetty本身进行二次开发，定制一个适合自身需求的Web Server。
- 4、使用Jetty，需要在spring-boot-starter-web排除spring-boot-starter-tomcat，因为SpringBoot默认使用tomcat

对于配置内置服务器的springBoot，都必定会配置

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

以上配置springBoot的启动web服务器，但默认是Tomcat

所以呢，要配置为jetty要去掉默认tomcat配置

```
1 | <dependency>
2 |     <groupId>org.springframework.boot</groupId>
3 |     <artifactId>spring-boot-starter-web</artifactId>
4 |     <exclusions>
5 |         <exclusion>
6 |             <groupId>org.springframework.boot</groupId>
7 |             <artifactId>spring-boot-starter-tomcat</artifactId>
8 |         </exclusion>
9 |     </exclusions>
10 | </dependency>
```

并且加上jetty启动

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

6、资源定义

默认，就用默认

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

如果需要修改

第一种：在配置文件中配置

```
1 #静态资源访问路径  
2 spring.mvc.static-path-pattern=/img/**  
3 #静态资源映射路径，注意会覆盖  
4 spring.mvc.static-path-pattern=/**  
5 spring.web.resources.static-locations[0]=classpath:/img/  
6 spring.web.resources.static-locations[1]=classpath:/static/  
7 spring.web.resources.static-locations[2]=classpath:/public/
```

点击配置文件发现，源码就是读取static-locations来设置资源目录的：

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

7、web配置（重点）

WebMvcConfigurer配置类其实是Spring内部的一种配置方式，采用JavaBean的形式来代替传统的xml配置文件形式进行针对框架个性化定制，可以自定义一些Handler，Interceptor，ViewResolver，MessageConverter。基于java-based方式的spring mvc配置，需要创建一个配置类并实现WebMvcConfigurer接口；

在Spring Boot 1.5版本都是靠重写WebMvcConfigurerAdapter的方法来添加自定义拦截器，消息转换器等。SpringBoot 2.0 后，该类被标记为@Deprecated（弃用）。官方推荐直接实现WebMvcConfigurer或者直接继承WebMvcConfigurationSupport，方式一实现WebMvcConfigurer接口（推荐），方式二继承WebMvcConfigurationSupport类。

常用的方法：

```
1 /**  
2  * @author zn  
3  * @date 2021/3/3  
4  */  
5 @Configuration  
6 public class ApplicationConfiguration implements webMvcConfigurer {  
7  
8     /**
```

```

9      * 定义拦截器
10     * @param registry
11     */
12     @Override
13     public void addInterceptors(InterceptorRegistry registry) {
14         registry.addInterceptor(new
LoginInterception()).addPathPatterns("/**");
15     }
16
17     /**
18     * 定义资源，不覆盖默认
19     * @param registry
20     */
21     @Override
22     public void addResourceHandlers(ResourceHandlerRegistry registry) {
23
24         registry.addResourceHandler("/img/**").addResourceLocations("classpath:/im
g/");
25
26         registry.addResourceHandler("/file/**").addResourceLocations("file:D:/img/
");
27     }
28
29     /**
30     * 后端解决跨域的配置
31     * @param registry
32     */
33     @Override
34     public void addCorsMappings(CorsRegistry registry) {
35         registry.addMapping("/app/**").allowedOrigins("");
36     }
37
38     /**
39     * 视图跳转控制器
40     * 相当于写了controller
41     * @param registry
42     */
43     @Override
44     public void addViewControllers(ViewControllerRegistry registry) {
45         registry.addViewController("/user").setViewName("user");
46     }
47
48     /**
49     <dependency>
50         <groupId>com.alibaba</groupId>
51         <artifactId>fastjson</artifactId>
52         <version>1.2.46</version>
53     </dependency>
54     * 配置消息转化器
55     * @param converters
56     */
57     @Override
58     public void configureMessageConverters(List<HttpMessageConverter<?>>
converters) {
59
60         FastJsonHttpMessageConverter converter = new
FastJsonHttpMessageConverter();
61         FastJsonConfig config = new FastJsonConfig();

```

```

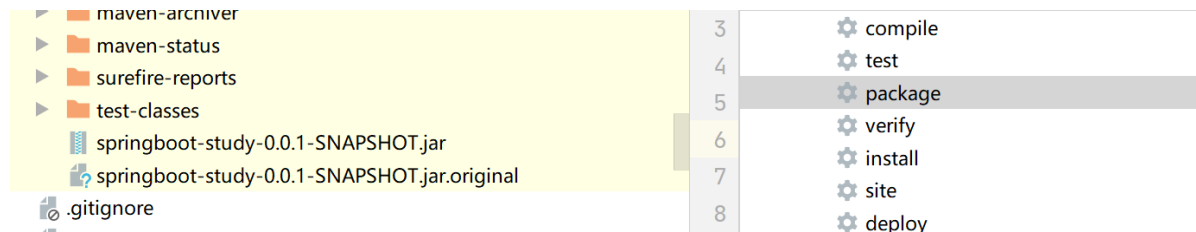
60         config.setSerializerFeatures(
61             // 保留map空的字段
62             SerializerFeature.WriteMapNullValue,
63             // 将String类型的null转成""
64             SerializerFeature.WriteNullStringAsEmpty,
65             // 将Number类型的null转成0
66             SerializerFeature.WriteNullNumberAsZero,
67             // 将List类型的null转成[]
68             SerializerFeature.WriteNullListAsEmpty,
69             // 将Boolean类型的null转成false
70             SerializerFeature.WriteNullBooleanAsFalse,
71             // 避免循环引用
72             SerializerFeature.DisableCircularReferenceDetect
73         );
74
75         converter.setFastJsonConfig(config);
76         converter.setDefaultCharset(Charset.forName("UTF-8"));
77         List<MediaType> mediaTypeList = new ArrayList<>();
78         // 解决中文乱码问题，相当于在Controller上的@RequestMapping中加了个属性
        produces = "application/json"
79         mediaTypeList.add(MediaType.APPLICATION_JSON);
80         converter.setSupportedMediaTypes(mediaTypeList);
81
82         //将fastjson添加到视图消息转换器列表内
83         converters.add(converter);
84     }
85
86
87     /**
88      * 配置视图解析器
89      * @param registry
90      */
91     @Override
92     public void configureViewResolvers(ViewResolverRegistry registry) {
93
94     }
95
96     /**
97      * 参数解析器
98      * @param resolvers
99      */
100    @Override
101    public void addArgumentResolvers(List<HandlerMethodArgumentResolver>
        resolvers) {
102
103    }
104
105    /**
106     * 配置返回值处理器
107     * @param handlers
108     */
109    @Override
110    public void
        addReturnValueHandlers(List<HandlerMethodReturnValueHandler> handlers) {
111
112    }
113
114 }

```

8、打包

jar包

因为是maven项目



标志打包为war

```
1 | <packaging>war</packaging>
```

```
1 | <dependency>
2 |     <groupId>org.springframework.boot</groupId>
3 |     <artifactId>spring-boot-starter-web</artifactId>
4 |     <exclusions>
5 |         <exclusion>
6 |             <groupId>org.slf4j</groupId>
7 |             <artifactId>slf4j-log4j12</artifactId>
8 |         </exclusion>
9 |         <exclusion>
10 |             <groupId>org.springframework.boot</groupId>
11 |             <artifactId>spring-boot-starter-tomcat</artifactId>
12 |         </exclusion>
13 |     </exclusions>
14 | </dependency>
15 | <dependency>
16 |     <groupId>org.springframework.boot</groupId>
17 |     <artifactId>spring-boot-starter-tomcat</artifactId>
18 |     <!-- 打包的时候可以不用包进去，别的设施会提供。事实上该依赖理论上可以参与编译，测试，运行等周期。
19 |          相当于compile，但是打包阶段做了exclude操作-->
20 |     <scope>provided</scope>
21 | </dependency>
```



```

1  /**
2   * @author IT楠老师
3   * @date 2020/6/7
4   */
5  @Component
6  public class ServletInitializer extends SpringBootServletInitializer {
7      @Override
8      protected SpringApplicationBuilder configure(SpringApplicationBuilder
builder) {
9          return builder.sources(MySpringBootApplication.class);
10     }
11 }

```

SpringBootServletInitializer的执行过程，简单来说就是通过SpringApplicationBuilder构建并封装SpringApplication对象，并最终调用SpringApplication的run方法的过程。

spring boot就是为了简化开发的，也就是用注解的方式取代了传统的xml配置。

SpringBootServletInitializer就是原有的web.xml文件的替代。

使用了嵌入式Servlet,默认是不支持jsp。

SpringBootServletInitializer 可以使用外部的Servlet容器，使用步骤：

1. 必须创建war项目，需要创建好web项目的目录。2. 嵌入式Tomcat依赖scope指定provided。3. 编写SpringBootServletInitializer子类，并重写configure方法。

```

public class ServletInitializer extends SpringBootServletInitializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
        return application.sources(SpringBoot04WebJspApplication.class);
    }
}

```

4. 启动服务器。

jar包和war包启动区别

- 1 | jar包: 执行SpringBootApplication的run方法, 启动IOC容器, 然后创建嵌入式Servlet容器

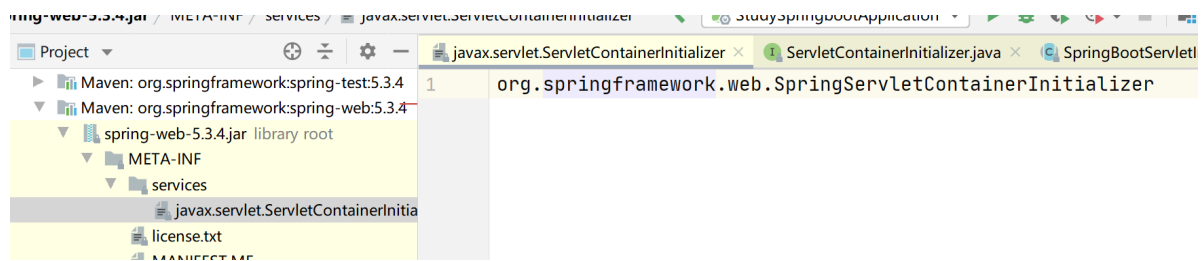
war包: 先是启动Servlet服务器, 服务器启动Springboot应用(SpringBootServletInitizer), 然后启动IOC容器

SpringBootServletInitializer实例执行onStartup方法的时候会通过createRootApplicationContext方法来执行run方法，接下来的过程就同以jar包形式启动的应用的run过程一样了，在内部会创建IOC容器并返回，只是以war包形式的应用在创建IOC容器过程中，不再创建Servlet容器了。

tomcat

1、ServletContainerInitializer

2、META-INF/services/javax.servlet.ServletContainerInitializer 找 spi 调用 onstart



3、servletcontext 配置servlet filter listener

9、启动系统任务

有两个接口CommandLineRunner和ApplicationRunner，实现了这两项接口的类会在系统启动后自动调用执行run方法

```
1  /**
2   * @author IT楠老师
3   * @date 2020/6/7
4   */
5  @Component
6  @Order(1)
7  public class CommandRunnerOne implements CommandLineRunner {
8      //args是获取的main方法传入的参数
9      @Override
10     public void run(String... args) throws Exception {
11         //这里可以做一些事情比如redis预热，系统检查等工作
12         System.out.println("commandOne---->" + Arrays.toString(args));
13     }
14 }
15
16
17 @Component
18 @Order(2)
19 public class CommandRunnerTwo implements CommandLineRunner {
20     //args是获取的main方法传入的参数
21     @Override
22     public void run(String... args) throws Exception {
23         //这里可以做一些事情比如redis预热，系统检查等工作
24         System.out.println("commandTwo---->" + Arrays.toString(args));
25     }
26 }
```

测试

另一个和这个接口用法一样，只是参数不同，有兴趣的同学可以研究一下。

```
▶  = "java.class.version" -> "52.0"
▶  = "sun.management.compiler" -> "HotSpot 64-Bit Tiered Compilers"
▶  = "spring.liveBeansView.mbeanDomain" -> ""
▶  = "os.version" -> "10.0"
▶  = "user.home" -> "C:\Users\51018"
▶  = "user.timezone" -> "Asia/Shanghai"
▶  = "java.awt.printerjob" -> "sun.awt.windows.WPrinterJob"
▶  = "file.encoding" -> "UTF-8"
▶  = "java.specification.version" -> "1.8"
▶  = "java.class.path" -> "C:\Program Files\Java\jdk1.8.0_221\jre\lib\charsets.jar;C:\Program
▶  = "user.name" -> "51018"
▶  = "com.sun.management.jmxremote" -> ""
▶  = "java.vm.specification.version" -> "1.8"
▶  = "sun.java.command" -> "cn.itnanls.springbootstudy.SpringbootStudyApplication"
▶  = "java.home" -> "C:\Program Files\Java\jdk1.8.0_221\jre"
▶  = "sun.arch.data.model" -> "64"
▶  = "user.language" -> "zh"
▶  = "java.specification.vendor" -> "Oracle Corporation"
▶  = "awt.toolkit" -> "sun.awt.windows.WToolkit"
▶  = "java.vm.info" -> "mixed mode"
▶  = "java.version" -> "1.8.0 221"
```

10、配置文件的读取顺序

如果在不同的目录中存在多个配置文件，它的读取顺序是：

- 1、config/application.properties（项目根目录中config目录下）
- 2、config/application.yml
- 3、application.properties（项目根目录下）
- 4、application.yml
- 5、resources/config/application.properties（项目resources目录中config目录下）
- 6、resources/config/application.yml
- 7、resources/application.properties（项目的resources目录下）
- 8、resources/application.yml

有啥好处，打包后我们可以再jar包之外放置配置文件，随时修改

```
1  /**
2   * {@link EnvironmentPostProcessor} that configures the context environment
   by loading
3   * properties from well known file locations. By default properties will be
   loaded from
4   * 'application.properties' and/or 'application.yml' files in the following
   locations:
5   * <ul>
6   * <li>file:./config/</li>
7   * <li>file:./config/{@literal *}</li>
8   * <li>file:./</li>
9   * <li>classpath:config/</li>
10  * <li>classpath:</li>
11  * </ul>
```

```

12  * The list is ordered by precedence (properties defined in locations higher
    in the list
13  * override those defined in lower locations).
14  */
15  @Deprecated
16  public class ConfigFileApplicationListener implements
    EnvironmentPostProcessor, SmartApplicationListener, Ordered {

```

11、SpringBoot Profile多环境配置

多Profile文件

我们在主配置文件编写的时候，文件名可以是 application-{profile}.properties/yml 例如：

- application-dev.yml
- application-prod.yml
- application-test.yml

默认使用application.properties的配置；

yml支持多文档块方式

```

1  spring:
2    profiles:
3      active: dev #指定使用哪个环境
4  ---
5  server:
6    port: 8082
7  spring:
8    profiles: dev
9  ---
10 server:
11   port: 8083
12 spring:
13   profiles: test
14 ---
15 server:
16   port: 8084
17 spring:
18   profiles: prod

```

激活指定Profile

- 在配置文件中指定 spring.profiles.active=dev
- 命令行: java -jar spring-boot-02-config-0.0.1-SNAPSHOT.jar --spring.profiles.active=dev;
- 可以直接在测试的时候，配置传入命令行参数
- 虚拟机参数: -Dspring.profiles.active=dev

优先级

命令行参数>JVM参数>配置文件

12、原生的servlet内容

(1) 第一种方式

由于springboot基于servlet3.0+,内嵌tomcat容器 因此无法像之前一样通过web.xml中配置Filter

```
1  /**
2   * @author itnanls
3   * @date 2021/3/9
4   */
5  @WebServlet(urlPatterns = "/user")
6  public class UserServlet extends HttpServlet {
7      @Override
8      protected void doGet(HttpServletRequest req, HttpServletResponse resp)
9      throws ServletException, IOException {
10         System.out.println("创建了servlet");
11     }
12 }
13 /**
14  * @author itnanls
15  */
16 @SpringBootApplication
17 @ServletComponentScan
18 public class SpringbootStudyApplication {
19     public static void main(String[] args) {
20         SpringApplication.run(SpringbootStudyApplication.class, args);
21     }
22 }
```

```
1  /**
2   * @author itnanls
3   * @date 2021/3/9
4   */
5  @WebFilter("/")
6  public class UserFilter implements Filter {
7      @Override
8      public void doFilter(ServletRequest request, ServletResponse response,
9      FilterChain chain) throws IOException, ServletException {
10         System.out.println("来了过滤器!");
11         doFilter(request, response, chain);
12     }
13 }
14 /**
15  * @author itnanls
16  */
17 @SpringBootApplication
18 @ServletComponentScan
19 public class SpringbootStudyApplication {
20     public static void main(String[] args) {
21         SpringApplication.run(SpringbootStudyApplication.class, args);
22     }
23 }
```

```

1  /**
2   * @author zn
3   * @date 2021/3/9
4   */
5  @WebListener
6  public class UserListener implements HttpSessionListener {
7
8      @Override
9      public void sessionCreated(HttpSessionEvent se) {
10         System.out.println("创建了一个session");
11     }
12 }
13
14
15 @SpringBootApplication
16 @ServletComponentScan
17 public class SpringbootStudyApplication {
18     public static void main(String[] args) {
19         SpringApplication.run(SpringbootStudyApplication.class, args);
20     }
21 }

```

(2) 第二种方式

```

1  /**
2   * @author itnanls
3   * @date 2021/3/9
4   */
5  @Configuration
6  public class ServletConfiguration {
7
8      @Bean
9      public ServletRegistrationBean<UserServlet> servletRegistrationBean(){
10         return new ServletRegistrationBean<>(new UserServlet(), "/user");
11     }
12
13     @Bean
14     public FilterRegistrationBean<UserFilter> filterRegistrationBean(){
15         FilterRegistrationBean<UserFilter> register = new
16         FilterRegistrationBean<>();
17         register.setFilter(new UserFilter());
18         register.setUrlPatterns(Arrays.asList("/admin/*", "/user/*"));
19         return register;
20     }
21
22     @Bean
23     public ServletListenerRegistrationBean<UserListener>
24     servletListenerRegistrationBean(){
25         return new ServletListenerRegistrationBean<>(new UserListener());
26     }
27 }

```

13、观察者设计模式

定义一个自定义事件，继承ApplicationEvent类

```

1  /**
2   * 定义事件
3   *
4   */
5  public class MyApplicationEvent extends ApplicationEvent {
6      private static final long serialVersionUID = 1L;
7      public MyApplicationEvent(Object source) {
8          super(source);
9      }
10 }

```

```

1  /**
2   * 定义事件的监听
3   *
4   */
5  @Component
6  public class MyApplicationListener implements
    ApplicationListener<MyApplicationEvent> {
7
8      public void onApplicationEvent(MyApplicationEvent event) {
9          System.out.println("接收到事件: "+event.getClass());
10     }
11
12 }

```

主类测试:

```

1  @SpringBootApplication
2  public class Application {
3      public static void main(String[] args) {
4          SpringApplication application = new
    SpringApplication(Application.class);
5          ConfigurableApplicationContext context =application.run(args);
6          //发布事件
7          context.publishEvent(new MyApplicationEvent(new Object()));
8          context.close();
9      }
10 }

```

14、定时任务

开启定时任务

```
1 @EnableScheduling
2 public class MySpringBootApplication
3
```

写代码

```
1 /**
2  * @author IT楠老师
3  * @date 2020/6/7
4  */
5 @Component
6 public class MySchedule {
7
8     @Scheduled(fixedDelay = 3000)
9     public void fixedDelay(){
10         System.out.println("fixedDelay:"+new Date());
11     }
12
13     @Scheduled(fixedRate = 3000)
14     public void fixedRate(){
15         System.out.println("fixedRate:"+new Date());
16     }
17
18
19     @Scheduled(initialDelay = 1000,fixedDelay = 2000)
20     public void initialDelay(){
21         System.out.println("initialDelay:"+new Date());
22     }
23
24     @Scheduled(cron = "0 * * * * ?")
25     public void cron(){
26         System.out.println("cron:"+new Date());
27     }
28
29 }
30
```

结果

```
1 fixedRate:Sun Jun 07 11:16:17 CST 2020
2 fixedDelay:Sun Jun 07 11:16:17 CST 2020
3 2020-06-07 11:16:17.337 INFO 11928 --- [           main]
  com.example.MySpringBootApplication : Started MySpringBootApplication
  in 6.028 seconds (JVM running for 8.211)
4 initialDelay:Sun Jun 07 11:16:18 CST 2020
5 fixedRate:Sun Jun 07 11:16:20 CST 2020
6 fixedDelay:Sun Jun 07 11:16:20 CST 2020
7 initialDelay:Sun Jun 07 11:16:20 CST 2020
8 initialDelay:Sun Jun 07 11:16:22 CST 2020
9 fixedRate:Sun Jun 07 11:16:23 CST 2020
10 fixedDelay:Sun Jun 07 11:16:23 CST 2020
11 initialDelay:Sun Jun 07 11:16:24 CST 2020
12 fixedRate:Sun Jun 07 11:16:26 CST 2020
13 fixedDelay:Sun Jun 07 11:16:26 CST 2020
```



```
14 | initialDelay:Sun Jun 07 11:16:26 CST 2020
15 | initialDelay:Sun Jun 07 11:16:28 CST 2020
16 | fixedRate:Sun Jun 07 11:16:29 CST 2020
17 | fixedDelay:Sun Jun 07 11:16:29 CST 2020
18 | initialDelay:Sun Jun 07 11:16:30 CST 2020
19 |
```

区别

- 1、fixedDelay控制方法执行的间隔时间，是以上一次方法执行完开始算起，如上一次方法执行阻塞住了，那么直到上一次执行完，并间隔给定的时间后，执行下一次。
- 2、fixedRate是按照一定的速率执行，是从上一次方法执行开始的时间算起，如果上一次方法阻塞住了，下一次也是不会执行，但是在阻塞这段时间内累计应该执行的次数，当不再阻塞时，一下子把这些全部执行掉，而后再按照固定速率继续执行。
- 3、cron表达式可以定制化执行任务，但是执行的方式是与fixedDelay相近的，也是会按照上一次方法结束时间开始算起。
- 4、initialDelay。如：@Scheduled(initialDelay = 10000,fixedRate = 15000)
这个定时器就是在上一个的基础上加了一个initialDelay = 10000 意思就是在容器启动后,延迟10秒后再执行一次定时器,以后每15秒再执行一次该定时器

quartz自学

五、SpringBoot与整合其他技术

1、SpringBoot整合Junit

添加Junit的起步依赖

```
1 | <dependency>
2 |     <groupId>org.springframework.boot</groupId>
3 |     <artifactId>spring-boot-starter-test</artifactId>
4 |     <scope>test</scope>
5 | </dependency>
```

编写测试类

```
1 | package com.xinzhi.test;
2 |
3 | import com.xinzhi.MySpringBootApplication;
4 | import com.xinzhi.domain.User;
5 | import com.xinzhi.mapper.UserMapper;
6 | import org.junit.Test;
7 | import org.junit.runner.RunWith;
8 | import org.springframework.beans.factory.annotation.Autowired;
9 | import org.springframework.boot.test.context.SpringBootTest;
10 | import org.springframework.test.context.junit4.SpringRunner;
11 |
12 | import java.util.List;
13 |
14 | @SpringBootTest
15 | class StudyspringbootApplicationTests {
16 |
```

```

17     @Autowired
18     private Cat cat;
19
20     @Test
21     void contextLoads() {
22         System.out.println(cat);
23     }
24
25 }

```

控制台打印信息

2、整合Spring Data JPA

JPA是Java Persistence API的简称，中文名Java持久层API，是JDK 5.0注解或XML描述对象 - 关系表的映射关系，并将运行期的实体[对象持久化](#)到数据库中。

JPA是需要Provider来实现其功能的，Hibernate就是JPA Provider中很强的一个，应该说无人能出其右。从功能上来说，JPA就是Hibernate功能的一个子集。Hibernate 从3.2开始，就开始兼容JPA。Hibernate3.2获得了Sun TCK的JPA(Java Persistence API) 兼容认证。

添加Spring Data JPA的起步依赖

```

1 <!-- springBoot JPA的起步依赖 -->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-data-jpa</artifactId>
5 </dependency>

```

添加数据库驱动依赖

```

1 <!-- MySQL连接驱动 -->
2 <dependency>
3     <groupId>mysql</groupId>
4     <artifactId>mysql-connector-java</artifactId>
5 </dependency>

```

在application.properties中配置数据库和jpa的相关属性

```

1 spring:
2     #通用的数据源配置
3     datasource:
4         url: jdbc:mysql://localhost:3306/ssm?
4         useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai
5         driver-class-name: com.mysql.cj.jdbc.Driver
6         username: root
7         password: root
8     jpa:
9         #这个参数是在建表的时候，将默认的存储引擎切换为 InnoDB 用的
10        database-platform: org.hibernate.dialect.MySQL5InnoDBDialect
11        #配置在日志中打印出执行的 SQL 语句信息。
12        show-sql: true
13        hibernate:

```

```

14      #配置指明在程序启动的时候要删除并且创建实体类对应的表
15      ddl-auto: create
16
17  ddl-auto可选参数
18  create 启动时删数据库中的表，然后创建，退出时不删除数据表
19  create-drop 启动时删数据库中的表，然后创建，退出时删除数据表 如果表不存在报错
20  update 如果启动时表格式不一致则更新表，原有数据保留
21  validate 项目启动表结构进行校验 如果不一致则报错

```

创建实体配置实体

```

1  @NoArgsConstructor
2  @Data
3  @Entity
4  @Table(name = "user")
5  public class User {
6
7      @Id
8      @GeneratedValue(strategy=GenerationType.AUTO)//主键生成策略
9      @Column(name="id")
10     private Integer id;
11     @Column(name = "user_name")
12     private String username;
13     @Column(name = "password")
14     private String password;
15     @Column(name = "birthday")
16     private Date birthday;
17
18 }

```

主键生成策略

JPA提供的四种标准用法为TABLE,SEQUENCE,IDENTITY,AUTO.

1、TABLE：使用一个特定的数据库表格来保存主键。

使用一个特定的数据库表格来保存主键,持久化引擎通过关系数据库的一张特定的表格来生成主键,这种策略的好处就是不依赖于外部环境和数据库的具体实现,在不同数据库间可以很容易的进行移植,但由于其不能充分利用数据库的特性,所以不会优先使用。

2、SEQUENCE：根据底层数据库的序列来生成主键，条件是数据库支持序列。

在某些数据库中,不支持主键自增长,比如Oracle,其提供了一种叫做"序列(sequence)"的机制生成主键。此时,GenerationType.SEQUENCE就可以作为主键生成策略。该策略的不足之处正好与TABLE相反,由于只有部分数据库(Oracle,PostgreSQL,DB2)支持序列对象,MySQL不支持序列,所以该策略一般不应用于其他数据库。类似的,该策略一般与另外一个注解一起使用@SequenceGenerator,@SequenceGenerator注解指定了生成主键的序列.然后JPA会根据注解内容创建一个序列(或使用一个现有的序列)。如果不指定序列,则会自动生成一个序列SEQ_GEN_SEQUENCE。

3、IDENTITY：主键由数据库自动生成（主要是自动增长型）

此种主键生成策略就是通常所说的主键自增长,数据库在插入数据时,会自动给主键赋值,比如MySQL可以在创建表时声明"auto_increment" 来指定主键自增长。该策略在大部分数据库中都提供了支持(指定方法或关键字可能不同),但还是少数数据库不支持,所以可移植性略差。使用自增长主键生成策略是只需要声明strategy = GenerationType.IDENTITY即可。

4、AUTO：主键由程序控制。

把主键生成策略交给持久化引擎(persistence engine),持久化引擎会根据数据库在以上三种主键生成策略中选择其中一种。此种主键生成策略比较常用,由于JPA默认的生成策略就是GenerationType.AUTO,所以使用此种策略时,可以显式的指定@GeneratedValue(strategy = GenerationType.AUTO)也可以直接@GeneratedValue。

编写UserRepository

```
1 public interface UserRepository extends JpaRepository<User,Long>{
2     public List<User> findAll();
3 }
```

编写测试类

```
1 @SpringBootTest(classes=MySpringBootApplication.class)
2 public class JpaTest {
3
4     @Autowired
5     private UserRepository userRepository;
6
7     @Test
8     public void test(){
9         List<User> users = userRepository.findAll();
10        System.out.println(users);
11    }
12
13 }
```

控制台打印信息

HQL是Hibernate Query Language的缩写,提供更加丰富灵活、更为强大的查询能力;HQL更接近SQL语句查询语法。

- 对查询条件进行了[面向对象](#)封装,符合编程人员的思维方式,格式: from + 类名 + 类对象 + where + 对象的属性
- **区分大小写,关键字不区分大小写**
- **从下标0开始计算位置(hibernate5之后不支持)**
- **支持命名参数**

```
1 @Query("select u from User1 u where u.username = :aaa")
2 List<User1> findByUsername(@Param("aaa") String aaa);
```

如果想使用原生sql

```
1 @Query(value = "select * from t_users where username =?",nativeQuery = true)
```

分页处理,一切皆对象

```

1 | Sort sort = Sort.by(Sort.Direction.DESC,"id");
2 | Pageable pageable = PageRequest.of(0,3,sort);
3 | Page<User> all = userDao.findAll(pageable);

```

如图1所示，会议提前关闭问题

spring整合hibernate或者spring boot里使用jpa，本质都是hibernate的session操作数据库，默认session会提前关闭，报延迟加载..异常，要等到后台值返回到视图层之后，才关闭session才合理;

spring boot里这样设置即可，延续session到返回视图层

3、SpringBoot整合Mybatis

添加Mybatis的起步依赖

```

1 | <!--mybatis起步依赖-->
2 | <dependency>
3 |     <groupId>org.mybatis.spring.boot</groupId>
4 |     <artifactId>mybatis-spring-boot-starter</artifactId>
5 |     <version>2.1.2</version>
6 | </dependency>
7 | <!-- MySQL连接驱动 -->
8 | <dependency>
9 |     <groupId>mysql</groupId>
10 |    <artifactId>mysql-connector-java</artifactId>
11 | </dependency>
12 | <dependency>
13 |     <groupId>org.projectlombok</groupId>
14 |     <artifactId>lombok</artifactId>
15 | </dependency>

```

添加数据库连接信息

在application.properties中添加数据库的连接信息

```

1 | #DB Configuration:
2 | spring:
3 |     datasource:
4 |         url: jdbc:mysql://localhost:3306/ssm?
           useUnicode=true&characterEncoding=utf8&useSSL=false&serverTimezone=Asia/Shanghai
5 |         driver-class-name: com.mysql.cj.jdbc.Driver
6 |         username: root
7 |         password: root

```

创建user表

在test数据库中创建user表

```

1 | -- -----
2 | -- Table structure for `user`
3 | -- -----
4 | DROP TABLE IF EXISTS `user`;

```

```

5 CREATE TABLE `user` (
6   `id` int(11) NOT NULL AUTO_INCREMENT,
7   `username` varchar(50) DEFAULT NULL,
8   `password` varchar(50) DEFAULT NULL,
9   PRIMARY KEY (`id`)
10 ) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT CHARSET=utf8;
11
12 -----
13 -- Records of user
14 -----
15 INSERT INTO `user` VALUES ('1', 'zhangsan', '123');
16 INSERT INTO `user` VALUES ('2', 'lisi', '123');

```

创建实体Bean

```

1 /**
2  * @author IT楠老师
3  * @date 2020/6/5
4  */
5 @Data
6 @AllArgsConstructor
7 @NoArgsConstructor
8 public class User {
9
10     private int id;
11     private String username;
12     private String password;
13
14 }

```

编写Mapper

```

1 @Mapper
2 public interface UserMapper {
3     public List<User> queryUserList();
4 }

```

注意：@Mapper标记该类是一个mybatis的mapper接口，可以被spring boot自动扫描到spring上下文中

配置Mapper映射文件

在src\main\resources\mapper路径下加入UserMapper.xml配置文件"

```

1 <?xml version="1.0" encoding="utf-8" ?>
2 <!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
3   "http://mybatis.org/dtd/mybatis-3-mapper.dtd" >
4 <mapper namespace="com.xinzhi.dao.UserMapper">
5     <select id="queryUserList" resultType="user">
6         select * from user
7     </select>
8 </mapper>

```

在application.properties中添加mybatis的信息

```
1 #spring集成Mybatis环境
2 mybatis:
3   type-aliases-package: com.example.entity
4   mapper-locations: mapper/*Mapper.xml
```

service层

```
1 /**
2  * @author IT楠老师
3  * @date 2020/6/5
4  */
5 public interface IUserService {
6
7     /**
8      * 获取用户信息
9      * @return
10     */
11     List<User> getAllUsers();
12
13 }
14
15 @Service
16 public class UserServiceImpl implements IUserService {
17
18     @Resource
19     private UserMapper userMapper;
20
21     @Override
22     public List<User> getAllUsers() {
23         return userMapper.queryUserList();
24     }
25 }
```

编写测试Controller

```
1 /**
2  * @author IT楠老师
3  * @date 2020/6/5
4  */
5 @Controller
6 @RequestMapping("/user")
7 public class UserController {
8
9     @Resource
10     private IUserService userService;
11
12     @GetMapping
13     @ResponseBody
14     public List<User> getUsers(){
15         return userService.getAllUsers();
16     }
17 }
```

测试

4、德鲁伊数据源

```
1  /**
2   * @author IT楠老师
3   * @date 2020/6/5
4   */
5  @Configuration
6  public class DruidConfig {
7
8      @ConfigurationProperties(prefix = "spring.datasource")
9      @Bean
10     public DataSource druid() {
11         return new DruidDataSource();
12     }
13 }
```

```
1  spring:
2     datasource:
3         username: root
4         password: root
5         driver-class-name: com.mysql.cj.jdbc.Driver
6         url: jdbc:mysql://127.0.0.1:3306/ssm?
characterEncoding=utf8&useSSL=false&serverTimezone=UTC&rewriteBatchedStateme
nts=true
7         type: com.alibaba.druid.pool.DruidDataSource
8         # 数据源其他配置
9         initialSize: 5
10        minIdle: 5
11        maxActive: 20
12        maxWait: 60000
13        timeBetweenEvictionRunsMillis: 60000
14        minEvictableIdleTimeMillis: 300000
15        validationQuery: SELECT 1 FROM DUAL
16        testWhileIdle: true
17        testOnBorrow: false
18        testOnReturn: false
19        poolPreparedStatements: true
20        # 配置监控统计拦截的filters，去掉后监控界面sql无法统计，'wall'用于防火墙
21        filters: stat,wall
22        maxPoolPreparedStatementPerConnectionSize: 20
23        useGlobalDataSourceStat: true
24        connectionProperties:
druid.stat.mergeSql=true;druid.stat.slowSqlMillis=500
25
26
```

重新运行

看后台已经切换了

还有监控呢？

完善配置文件


```

1  /**
2   * @author IT楠老师
3   * @date 2020/6/5
4   */
5  @Configuration
6  public class DruidConfig {
7
8      /**
9       * 注入数据源
10      * @return
11      */
12      @ConfigurationProperties(prefix = "spring.datasource")
13      @Bean
14      public DataSource druid() {
15          return new DruidDataSource();
16      }
17
18
19      /**
20       * 配置监控
21       * @return
22       */
23      @Bean
24      public ServletRegistrationBean statViewServlet(){
25          ServletRegistrationBean bean = new ServletRegistrationBean(new
StatViewServlet(), "/druid/*");
26          HashMap<String, String> map = new HashMap<>(2);
27          map.put("loginUsername", "xinzhi");
28          map.put("loginPassword", "123456");
29          bean.setInitParameters(map);
30          return bean;
31      }
32
33      @Bean
34      public FilterRegistrationBean webStatFilter(){
35          FilterRegistrationBean<Filter> bean = new FilterRegistrationBean<>
();
36          bean.setFilter(new WebStatFilter());
37          HashMap<String, String> map = new HashMap<>(8);
38          map.put("exclusions", "*.js");
39          bean.setInitParameters(map);
40          bean.setUrlPatterns(Arrays.asList("/*"));
41          return bean;
42      }
43
44  }

```

搞定

5、pagehelper

为了方便分页

引入依赖

```

1 <dependency>
2   <groupId>com.github.pagehelper</groupId>
3   <artifactId>pagehelper-spring-boot-starter</artifactId>
4   <version>1.2.10</version>
5 </dependency>

```

在application.yml中做如下配置

```

1 # 分页配置
2 pagehelper:
3   helper-dialect: mysql
4   reasonable: true
5   support-methods-arguments: true
6   params: count=countSql

```

在代码中使用, (service或controller)

```

1 //这行是重点, 表示从pageNum页开始, 每页pageSize条数据
2 PageHelper.startPage(pageNum, pageSize);
3 List<Tools> list = toolsMapper.findAll();
4 PageInfo<Tools> pageInfo = new PageInfo<Tools>(list);
5 return ServerResponse.createBySuccess("查询成功", pageInfo);

```

开启日志

```

1 mybatis:
2   mapper-locations: mapper/*.xml
3   type-aliases-package: cn.itnanls.studyspringboot.entity
4   configuration:
5     map-underscore-to-camel-case: true
6     log-impl: org.apache.ibatis.logging.stdout.StdOutImpl

```

6、Thymeleaf模板引擎

Thymeleaf整合SpringBoot

在pom.xml文件引入thymeleaf

```

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

```

在application.properties (application.yml) 文件中配置thymeleaf

```

1 | spring.thymeleaf.prefix=classpath:/templates/
2 | spring.thymeleaf.check-template-location=true
3 | spring.thymeleaf.suffix=.html
4 | spring.thymeleaf.encoding=UTF-8
5 | spring.thymeleaf.content-type=text/html
6 | spring.thymeleaf.mode=HTML5
7 | spring.thymeleaf.cache=false

```

新建编辑控制层代码HelloController，在request添加了name属性，返回到前端hello.html再使用thymeleaf取值显示。

```

1 | @GetMapping("/login")
2 | public String toLogin(HttpServletRequest request){
3 |     request.setAttribute("name", "zhangsan");
4 |     return "login";
5 | }

```

新建编辑模板文件，在resources文件夹下的templates目录，用于存放HTML等模板文件，在这新增hello.html，添加如下代码。

```

1 | <!DOCTYPE html>
2 | <html lang="en" xmlns:th="http://www.thymeleaf.org">
3 | <head>
4 |     <meta charset="UTF-8"/>
5 |     <title>springboot-thymeleaf demo</title>
6 | </head>
7 |
8 | <body>
9 |     <p th:text="'hello, ' + ${name} + '!'" />
10 | </body>
11 | </html>

```

切记：使用Thymeleaf模板引擎时，必须在html文件上方添加该行代码使用支持Thymeleaf。

```

1 | <html lang="en" xmlns:th="http://www.thymeleaf.org">

```

5. 启动项目，访问<http://localhost:8080/user/login>，看到如下显示证明SpringBoot整合Thymeleaf成功。

博客学习该模板引擎语法

https://blog.csdn.net/qq_24598601/article/details/89190411

路径映射

如果觉得写controller太麻烦一次性多映射一些常用的地址

```

1  @Override
2  protected void addViewControllers(ViewControllerRegistry registry) {
3      registry.addViewController("/login").setViewName("login.html");
4      registry.addViewController("/register").setViewName("register.html");
5      super.addViewControllers(registry);
6  }

```

7、集成Swagger

(1) POM添加依赖

```

1  <dependency>
2      <groupId>io.springfox</groupId>
3      <artifactId>springfox-boot-starter</artifactId>
4      <version>3.0.0</version>
5  </dependency>

```

(2) Application启动应用类上面加入@EnableOpenApi注解

```

1  @EnableOpenApi

```

(3) Swagger3Config的配置

```

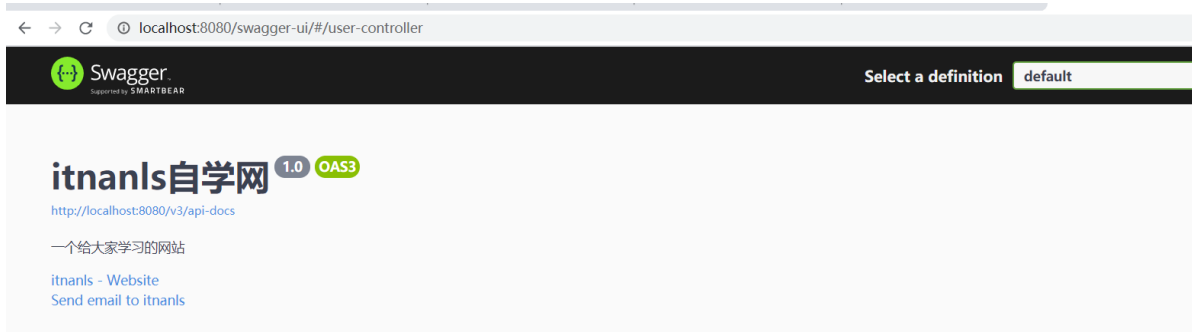
1  /**
2   * @author itnanls
3   * @date 2021/3/11
4   */
5  @Configuration
6  public class Swagger3Config {
7      @Bean
8      public Docket createRestApi() {
9          return new Docket(DocumentationType.OAS_30)
10             .apiInfo(apiInfo())
11             .select()
12             //
13             .apis(RequestHandlerSelectors.withMethodAnnotation(ApiOperation.class))
14             .apis(RequestHandlerSelectors.basePackage("cn.itnanls.springbootstudy.controller"))
15             .paths(PathSelectors.any())
16             .build();
17     }
18     private ApiInfo apiInfo() {
19         return new ApiInfoBuilder()
20             .title("itnanls自学网")
21             .description("一个给大家学习的网站")

```

```

22         .contact(new Contact("itnanls", "www.itnanls.cn",
    "510180298@qq.com"))
23         .version("1.0")
24         .build();
25     }
26
27 }
28

```



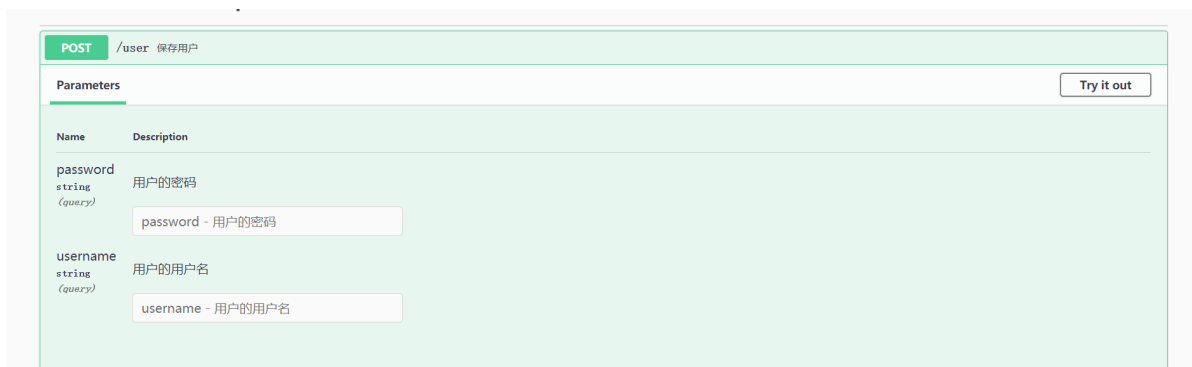
(4) 完善controller

```

1  /**
2   * @author itnanls
3   * @date 2021/3/11
4   */
5  @RestController
6  @RequestMapping("user")
7  @Api(tags = "用户信息的curd操作api")
8  public class UserController {
9
10     @GetMapping("{id}")
11     @ApiOperation("根据id获取用户")
12     @ApiImplicitParam(name = "id", value = "用户的id", paramType =
    "path", dataType = "Integer")
13     @ApiResponse(code = 200, message = "成功")
14     public String getUser(@PathVariable Integer id){
15         return id + "号员工: zhangsan";
16     }
17
18     @PostMapping
19     @ApiOperation("保存用户")
20     @ApiImplicitParams({
21         @ApiImplicitParam(name = "username", value = "用户的用户名",
    paramType = "query", dataType = "String"),
22         @ApiImplicitParam(name = "password", value = "用户的密码")
23     })
24     @ApiResponses({
25         @ApiResponse(code = 200, message = "成功"),
26         @ApiResponse(code = 2001, message = "用户已经存在"),
27         @ApiResponse(code = 2002, message = "密码不合法")
28     })
29     public String getUser(String username, String password){
30         return "Success";
31     }

```

```
32 |  
33 | }  
34 |
```



POST /user 保存用户

Try it out

Parameters

| Name | Description |
|-------------------------------|-------------|
| password string (query) | 用户的密码 |
| username string (query) | 用户的用户名 |

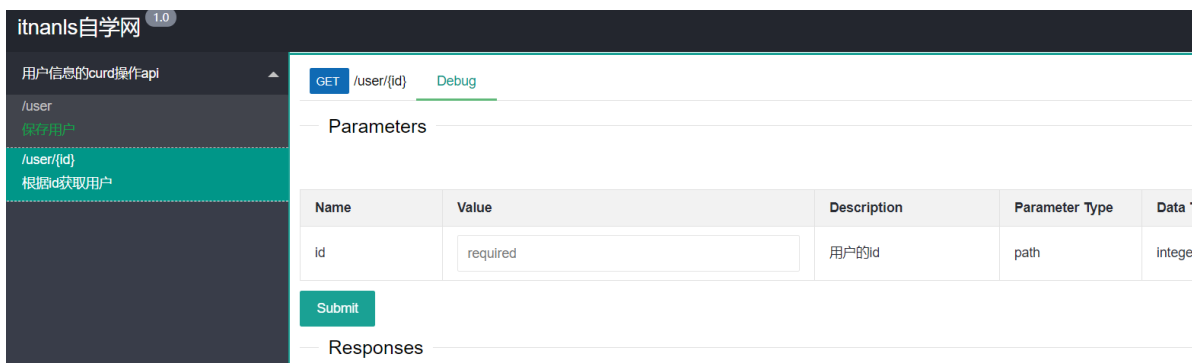
password - 用户的密码

username - 用户的用户名

(5) 换皮肤

访问地址: <http://localhost:8080/docs.html>

```
1 <dependency>  
2   <groupId>com.github.caspar-chen</groupId>  
3   <artifactId>swagger-ui-layer</artifactId>  
4   <version>1.1.3</version>  
5 </dependency>
```



itnanls 自学网 1.0

用户信息的crud操作api

/user

保存用户

/user/{id}

根据id获取用户

GET /user/{id} Debug

Parameters

| Name | Value | Description | Parameter Type | Data 1 |
|------|----------|-------------|----------------|--------|
| id | required | 用户的id | path | Intege |

Submit

Responses

8、SpringBoot整合Redis

Lettuce 和 Jedis 都是Redis的client, 所以他们都可以连接 Redis Server。

Jedis在实现上是直接连接的Redis Server, 如果在多线程环境下是非线程安全的。

每个线程都去拿自己的 Jedis 实例, 当连接数量增多时, 资源消耗阶梯式增大, 连接成本就较高了。

Lettuce的连接是基于Netty的, Netty 是一个多线程、事件驱动的 I/O 框架。连接实例可以在多个线程间共享, 当多线程使用同一连接实例时, 是线程安全的。

所以, 一个多线程的应用可以使用同一个连接实例, 而不用担心并发线程的数量。

当然这个也是可伸缩的设计, 一个连接实例不够的情况也可以按需增加连接实例。

通过异步的方式可以让我们更好的利用系统资源, 而不用浪费线程等待网络或磁盘I/O。

所以 Lettuce 可以帮助我们充分利用异步的优势。

添加redis的起步依赖

```
1 <!-- 配置使用redis启动器 -->
2 <dependency>
3     <groupId>org.springframework.boot</groupId>
4     <artifactId>spring-boot-starter-data-redis</artifactId>
5 </dependency>
```

配置redis的连接信息

```
1 #Redis
2 spring.redis.host=127.0.0.1
3 spring.redis.port=6379
```

注入RedisTemplate测试redis操作

jackson的核心方法：

```
1 String userJson = objectMapper.writeValueAsString(new User());
2 System.out.println(userJson);
3
4 User user = objectMapper.readValue(userJson, User.class);
5 System.out.println(user);
```

```
1 /**
2  * @author IT楠老师
3  * @date 2020/6/6
4  */
5
6 @SpringBootTest
7 public class RedisTest {
8
9     @Resource
10     private UserMapper userMapper;
11
12     @Autowired
13     private RedisTemplate<String, String> redisTemplate;
14
15     @Test
16     public void test() throws JsonProcessingException {
17
18         BoundHashOperations<String, Object, Object> hash =
19         redisTemplate.boundHashOps("user:1");
20         hash.put("username", "zhangsan");
21         hash.put("age", "12");
22         hash.put("password", "1233");
23
24         BoundValueOperations<String, String> userList =
25         redisTemplate.boundValueOps("user:list");
26
27         //从redis缓存中获得指定的数据
```

```

27     String usersJson = userList.get();
28     //如果redis中没有数据的话
29     if(null==usersJson){
30         //查询数据库获得数据
31         List<User> users = userMapper.queryUserList();
32         //转换成json格式字符串
33         ObjectMapper om = new ObjectMapper();
34         usersJson = om.writeValueAsString(users);
35         //将数据存储到redis中，下次在查询直接从redis中获得数据，不用在查询数据库
36         redisTemplate.boundValueOps("user:list").set(usersJson);
37
38         system.out.println("=====从数据库获得数据
=====");
39     }else{
40         system.out.println("=====从redis缓存中获得数据
=====");
41     }
42
43     System.out.println(usersJson);
44
45 }
46 }

```

注入自定义的redisTemplate，因为有一个需求我们经常想直接以String为key，value是Object去存储

```

1  @Bean
2  public RedisTemplate<String,Object> redisTemplate(RedisConnectionFactory
factory){
3      RedisTemplate<String,Object> redisTemplate = new RedisTemplate<>();
4      // 设置不同的序列化方式
5      redisTemplate.setKeySerializer(RedisSerializer.string());
6      redisTemplate.setValueSerializer(RedisSerializer.json());
7      redisTemplate.setHashKeySerializer(RedisSerializer.string());
8      redisTemplate.setHashValueSerializer(RedisSerializer.json());
9      // 设置redis的连接工厂，自动装配时已经注入了
10     redisTemplate.setConnectionFactory(factory);
11     return redisTemplate;
12 }

```

```

1  @Autowired
2  private RedisTemplate<String,Object> redisTemplate;
3
4  @Test
5  void contextLoads() {
6      ValueOperations<String, Object> opsForValue =
redisTemplate.opsForValue();
7      opsForValue.set("user",new User());
8  }

```


9、集成shiro (先学shiro)

shiro是web开发中常用的使用安全管理框架，通过shiro-spring-boot-web-starter方式集成Shiro到springboot2可以简化配置。

(1) 引包

maven方式在项目pom.xml中引入shiro starter包的坐标，这里引用了1.4.1版本

```
1 <dependency>
2     <groupId>org.apache.shiro</groupId>
3     <artifactId>shiro-spring-boot-web-starter</artifactId>
4     <version>1.7.1</version>
5 </dependency>
6
```

(2) 配置

starter已经做了很多自动配置工作，具体可以参考ShiroAutoConfiguration.java、ShiroBeanAutoConfiguration.java和ShiroWebAutoConfiguration.java这几个文件。

这里使用新建shiroConfig.java类方式进行shiro配置。主要配置Realm、url过滤器、密码匹配器和安全管理器这几个组件就可以让shiro正常工作。

```
1 @Configuration
2 public class shiroConfig { // 配置自定义Realm
3     @Bean
4     public UserRealm userRealm() {
5         UserRealm userRealm = new UserRealm();
6         userRealm.setCredentialsMatcher(credentialsMatcher()); //配置使用哈希
7         return userRealm;
8     }
9
10    // 配置url过滤器
11    @Bean
12    public ShiroFilterChainDefinition shiroFilterChainDefinition() {
13        DefaultShiroFilterChainDefinition chainDefinition = new
14        DefaultShiroFilterChainDefinition();
15
16        chainDefinition.addPathDefinition("/captcha", "anon");
17        chainDefinition.addPathDefinition("/logout", "anon");
18        chainDefinition.addPathDefinition("/layuiadmin/**", "anon");
19        chainDefinition.addPathDefinition("/druid/**", "anon");
20        chainDefinition.addPathDefinition("/api/**", "anon");
21        // all other paths require a logged in user
22        chainDefinition.addPathDefinition("/login", "anon");
23        chainDefinition.addPathDefinition("/**", "authc");
24        return chainDefinition;
25    }
26
27    // 设置用于匹配密码的CredentialsMatcher
28    @Bean
```

```

28     public HashedCredentialsMatcher credentialsMatcher() {
29         HashedCredentialsMatcher credentialsMatcher = new
HashedCredentialsMatcher();
30         credentialsMatcher.setHashAlgorithmName(Sha256Hash.ALGORITHM_NAME);
    // 散列算法，这里使用更安全的sha256算法
31         credentialsMatcher.setStoredCredentialsHexEncoded(false); // 数据库
    存储的密码字段使用HEX还是BASE64方式加密
32         credentialsMatcher.setHashIterations(1024); // 散列迭代次数
33         return credentialsMatcher;
34     }
35
36     // 配置security并设置userRealm，避免xxxx required a bean named 'authorizer'
    that could not be found.的报错
37     @Bean
38     public SessionsSecurityManager securityManager() {
39         DefaultWebSecurityManager securityManager = new
DefaultWebSecurityManager();
40         securityManager.setRealm(userRealm());
41         return securityManager;
42     }
43 }

```

(3) 登录页配置

shiro默认的登录页是/login.jsp，需要在项目配置文件application.yml中修改默认登录页等配置。)

```

1 shiro:
2   loginUrl: /toLogin
3   successUrl: /

```

(4) 自定义realm

```

1 /**
2  * @author itnanls
3  * @date 2021/3/10
4  */
5 public class UserRealm extends AuthorizingRealm {
6     @Override
7     protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection
principals) {
8         String username = (String)
SecurityUtils.getSubject().getPrincipal();
9         SimpleAuthorizationInfo authorizationInfo = new
SimpleAuthorizationInfo();
10        // 角色
11        Set<String> roles = new HashSet<>();
12        Set<String> permissions = new HashSet<>();
13        // 权限
14        // 测试用权限
15        if ("admin".equals(username)) {
16            roles.add("admin");
17            permissions.add("op:write");
18        } else {

```

```

19         roles.add("user");
20         permissions.add("op:read");
21     }
22     authorizationInfo.setRoles(roles);
23     authorizationInfo.setStringPermissions(permissions);
24     return authorizationInfo;
25 }
26
27 @Override
28 protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken
token) throws AuthenticationException {
29     String username = (String) token.getPrincipal();
30     // 从数据库查出user
31     // User user = userDao.getByUsername(username);
32     User user = new User();
33     if (user == null) {
34         throw new UnknownAccountException(); // 账号不存在
35     }
36     if (user.getStatus() != 0) {
37         throw new LockedAccountException(); // 账号被锁定
38     }
39     String salt = user.getSalt();
40     SimpleAuthenticationInfo authenticationInfo = new
SimpleAuthenticationInfo(user,
41         user.getPassword(),
42         ByteSource.Util.bytes(salt),
43         getName());
44
45     return authenticationInfo;
46 }
47 }

```

(5) 登录方法

```

1  @ResponseBody
2  @PostMapping("login")
3  public JsonResult doLogin(String username, String password,
HttpServletRequest request) {
4      try {
5          UsernamePasswordToken token = new UsernamePasswordToken(username,
password);
6          SecurityUtils.getSubject().login(token);
7          //addLoginRecord(getLoginUserId(), request); // 记录登录信息
8          return R.ok("登录成功").put("data", map);
9      } catch (IncorrectCredentialsException ice) {
10         return R.error("密码错误");
11     } catch (UnknownAccountException uae) {
12         return R.error("账号不存在");
13     } catch (LockedAccountException e) {
14         return R.error("账号被锁定");
15     } catch (ExcessiveAttemptsException eae) {
16         return JsonResult.error("操作频繁，请稍后再试");
17     }
18 }

```

(6) 接口权限验证

```
1 @RequiresRoles("user")
2 @GetMapping("/user/list.html")
3 public String userList() {
4     return "user/user/list";
5 }
6
7 @RequiresPermissions("op:read")
8 @GetMapping("user/userform.html")
9 public String userForm() {
10     return "user/user/userform";
11 }
```

(7) shiro默认过滤器

shiro提供和多个默认的过滤器，我们可以用这些过滤器来配置过滤指定url的访问权限。)

常见过滤器

| anon | AnonymousFilter | 指定url可以匿名访问 |
|-------------------|--------------------------------|---|
| authc | FormAuthenticationFilter | 指定url需要form表单登录，默认会从请求中获取username、password、rememberMe等参数并尝试登录，如果登录不了就会跳转到loginUrl配置的路径。我们也可以用这个过滤器做默认的登录逻辑，但是一般都是我们自己在控制器写登录逻辑的，自己写的话出错返回的信息都可以定制嘛。 |
| logout | LogoutFilter | 登出过滤器，配置指定url就可以实现退出功能，非常方便 |
| noSessionCreation | NoSessionCreationFilter | 禁止创建会话 |
| perms | PermissionsAuthorizationFilter | 需要指定权限才能访问 |
| roles | RolesAuthorizationFilter | 需要指定角色才能访问 |
| ssl | SslFilter | 需要https请求才能访问 |
| user | UserFilter | 需要已登录或“记住我”的用户才能访问 |

(8) shiro常用注解

可以在控制器类上使用

常见注解

- @RequiresGuest 只有游客可以访问
- @RequiresAuthentication 需要登录才能访问
- @RequiresUser 已登录的用户或“记住我”的用户能访问
- @RequiresRoles 已登录的用户需具有指定的角色才能访问
- @RequiresPermissions 已登录的用户需具有指定的权限才能访问

(9) 整合thymleaf

1、添加依赖

```
1 <dependency>
2     <groupId>com.github.theborakompanioni</groupId>
3     <artifactId>thymeleaf-extras-shiro</artifactId>
4     <version>2.0.0</version>
5 </dependency>
```

2、在ShiroConfig中配置ShiroDialect

```
1 /**
2  * shiro方言 支持shiro标签
3  * @return
4  */
5 @Bean
6 public ShiroDialect shiroDialect() {
7     return new ShiroDialect();
8 }
```

3、html中导入shiro标签

```
1 <!DOCTYPE html>
2 <html lang="en"
3     xmlns:th="http://www.thymeleaf.org"
4     xmlns:shiro="http://www.pollix.at/thymeleaf/shiro">
5 <head>
6     <meta charset="UTF-8">
7     <title>itnan1s</title>
8 </head>
9 <body>
10    <form id="loginForm" >
11        Username: <input type="text" name="username"/> <br/>
12        Password: <input type="password" name="password"/><br/>
13
14        <input type="checkbox" name="rememberMe" value="true"/>Remember Me<br/>
15        <input type="button" id="loginBtn" value="login">
16
17    </form>
18
19    <script src="https://cdn.bootcdn.net/ajax/libs/jquery/2.2.4/jquery.min.js">
20    </script>
```

```

20 <script>
21     $("#loginBtn").click(function () {
22         $.post("login", $("#loginForm").serialize(), function (data) {
23             console.log(data)
24             if(data === "success"){
25                 location.href = "/index";
26             }
27         })
28     })
29
30 </script>
31 </body>
32 </html>

```

登录测试!

10、整合spring-security

1、加入页面映射

```

1  @Configuration
2  public class MvcConfig implements WebMvcConfigurer {
3
4      public void addViewControllers(ViewControllerRegistry registry) {
5          registry.addViewController("/home").setViewName("home");
6          registry.addViewController("/").setViewName("home");
7          registry.addViewController("/hello").setViewName("hello");
8          registry.addViewController("/login").setViewName("login");
9      }
10
11 }

```

2、映入依赖

```

1 <dependency>
2     <groupId>org.springframework.boot</groupId>
3     <artifactId>spring-boot-starter-security</artifactId>
4 </dependency>
5 <dependency>
6     <groupId>org.springframework.security</groupId>
7     <artifactId>spring-security-test</artifactId>
8     <scope>test</scope>
9 </dependency>

```

3、编写配置类

```

1  @Configuration
2  @EnableWebSecurity
3  public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
4      @Override
5      protected void configure(HttpSecurity http) throws Exception {
6          http

```

```

7         .authorizeRequests()
8             .antMatchers("/", "/home").permitAll()
9             .anyRequest().authenticated()
10            .and()
11        .formLogin()
12            .loginPage("/login")
13            .permitAll()
14            .and()
15        .logout()
16            .permitAll();
17    }
18
19    @Bean
20    @Override
21    public UserDetailsService userDetailsService() {
22        UserDetails user =
23            User.withDefaultPasswordEncoder()
24                .username("user")
25                .password("password")
26                .roles("USER")
27                .build();
28
29        return new InMemoryUserDetailsManager(user);
30    }
31 }

```

- authenticated() 保护URL，需要用户登录
- permitAll() 指定URL无需保护，一般应用与静态资源文件
- hasRole(String role) 限制单个角色访问，角色将被增加 "ROLE_" .所以"ADMIN" 将和 "ROLE_ADMIN"进行比较.
- hasAuthority(String authority) 限制单个权限访问
- hasAnyRole(String... roles)允许多个角色访问.
- hasAnyAuthority(String... authorities) 允许多个权限访问.
- access(String attribute) 该方法使用 SpEL表达式, 所以可以创建复杂的限制.
- hasIpAddress(String ipAddressExpression) 限制IP地址或子网

4、login.html

```

1 <!DOCTYPE html>
2 <html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:th="https://www.thymeleaf.org"
3     xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-
  springsecurity3">
4     <head>
5         <title>Spring Security Example </title>
6     </head>
7     <body>
8         <div th:if="${param.error}">
9             Invalid username and password.
10        </div>
11        <div th:if="${param.logout}">
12            You have been logged out.
13        </div>

```

```

14         <form th:action="@{/login}" method="post">
15             <div><label> User Name : <input type="text" name="username"/>
</label></div>
16             <div><label> Password: <input type="password" name="password"/>
</label></div>
17             <div><input type="submit" value="Sign In"/></div>
18         </form>
19     </body>
20 </html>

```

5、hello.html

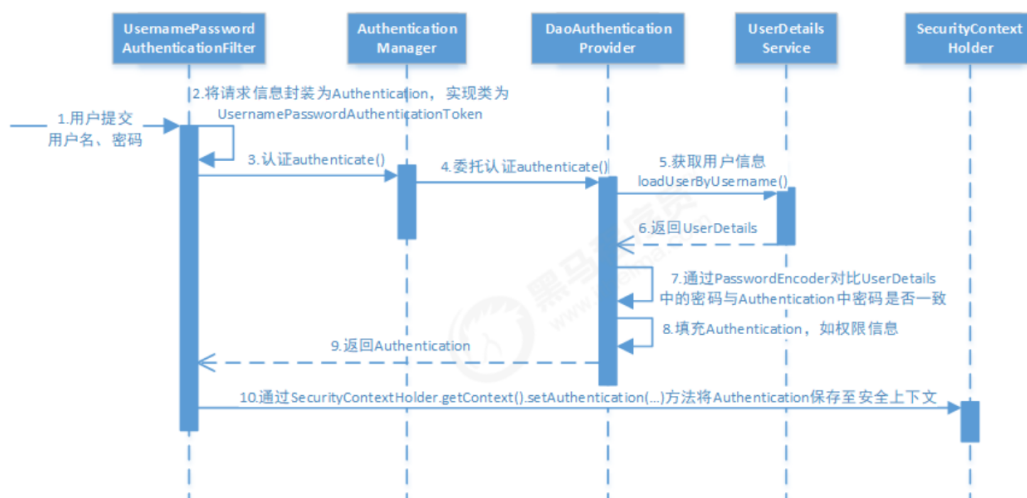
```

1  <!DOCTYPE html>
2  <html xmlns="http://www.w3.org/1999/xhtml"
xmlns:th="https://www.thymeleaf.org"
3      xmlns:sec="https://www.thymeleaf.org/thymeleaf-extras-
springsecurity3">
4      <head>
5          <title>Hello world!</title>
6      </head>
7      <body>
8          <h1 th:inline="text">Hello [[${#httpServletRequest.remoteUser}]]!
</h1>
9          <form th:action="@{/logout}" method="post">
10             <input type="submit" value="Sign out"/>
11         </form>
12     </body>
13 </html>

```

6、自定义

认证流程



```

1  /**
2   * @author itnanls
3   * @date 2021/3/11
4   */

```



```

5  @Service
6  public class SpringDataUserDetailsService implements UserDetailsService {
7      @Override
8      public UserDetails loadUserByUsername(String s) throws
UsernameNotFoundException {
9          return User
10             .withUsername("zhangsan")
11
12             .password("$2a$10$qsqPRUfw5LQUXEQrgOTqbuGMpuRFDqs2up15ZvNm33LA9QXRZfUma")
13             .roles("admin")
14             .authorities("user:add").build();
15 }

```

密码编码器

```

1  @Bean
2  public PasswordEncoder passwordEncoder(){
3      return new BCryptPasswordEncoder();
4  }

```

获取当前用户的方法

```

1  /**
2   * 获取当前登录用户名
3   * @return
4   */
5  private String getUsername(){
6      SecurityContext securityContext = SecurityContextHolder.getContext();
7      return securityContext.getAuthentication().getName();
8  }

```

我们可以在任何 @Configuration 实例上使用 @EnableGlobalMethodSecurity(prePostEnabled = true) 注释来启用基于注解的安全性。

方法注解

@Secured 注释用于指定方法上的角色列表。因此，如果用户至少具有一个指定的角色，则用户能访问该方法。

我们定义一个 getUsername 方法：

```

1  @Secured("ROLE_VIEWER")
2  public String getUsername() {
3      SecurityContext securityContext = SecurityContextHolder.getContext();
4      return securityContext.getAuthentication().getName();
5  }

```

我们可以传入多个

```
1 @Secured({ "ROLE_VIEWER", "ROLE_EDITOR" })
2 public boolean isValidUsername(String username) {
3     return userRoleRepository.isValidUsername(username);
4 }
```

@RoleAllowed注释是JSR-250对@Secured注释的等效注释。

基本上，我们可以像@Secured一样使用@RoleAllowed注释。

```
1 @RolesAllowed("ROLE_VIEWER")
2 public String getUsername2() {
3 }
4
5 @RolesAllowed({ "ROLE_VIEWER", "ROLE_EDITOR" })
6 public boolean isValidUsername2(String username) {
7 }
```

同样，只有当用户至少具有ROLE_VIEWER或ROLE_EDITOR角色之一时，用户才能调用isValidUsername2。

@PreAuthorize和**@PostAuthorize**注释都提供基于表达式的访问控制。

因此，可以使用SpEL (Spring Expression Language) 编写。

@PreAuthorize注释在进入方法之前检查给定的表达式，而**@PostAuthorize**注释在执行方法后验证它并且可能改变结果。

现在，让我们声明一个getUsernameInUpperCase方法，如下所示：

```
1 @PreAuthorize("hasRole('ROLE_VIEWER')")
2 public String getUsernameInUpperCase() {
3     return getUsername().toUpperCase();
4 }
```

```
1 @PreAuthorize("hasRole('ROLE_VIEWER') or hasRole('ROLE_EDITOR')")
2 public boolean isValidUsername3(String username) {
3     //...
4 }
```

而且，我们实际上可以使用method参数作为表达式的一部分：

```
1 @PreAuthorize("#username == authentication.principal.username")
2 public String getMyRoles(String username) {
3     //...
4 }
```

这里，只有当参数username的值与当前主体的用户名相同时，用户才能调用getMyRoles方法。

值得注意的是，@PreAuthorize表达式可以替换为@PostAuthorize表达式。

让我们重写getMyRoles:

```
1 @PostAuthorize("#username == authentication.principal.username")
2 public String getMyRoles2(String username) {
3     //...
4 }
```

但是, 在上一个示例中, 授权将在执行目标方法后延迟。

此外, @ PostAuthorize注释提供了访问方法结果的能力:

```
1 @PostAuthorize
2     ("returnObject.username == authentication.principal.nickName")
3 public CustomUser loadUserDetail(String username) {
4     return userRoleRepository.loadUserByUserName(username);
5 }
```

在此示例中, 如果返回的CustomUser的用户名等于当前身份验证主体的昵称, 则loadUserDetail方法会成功执行。

六、自动装配

1、原理

1) SpringBoot启动的时候加载主配置类, 开启了自动配置功能@EnableAutoConfiguration

2) @EnableAutoConfiguration作用:

- 利用EnableAutoConfigurationImportSelector给容器中导入一些组件;
- 可以插件selectImports()方法的内容;
- List configurations = getCandidateConfigurations(annotationMetadata, attributes)获取候选的配置

SpringFactoriesLoader.loadFactoryNames()会扫描所有jar包类路径下的META-INF/spring.factories文件

把扫描到的这些文件的内容包装成properties对象, 从properties中获取到EnableAutoConfiguration.class类(类名)对应的值, 然后把他们添加在容器中。

将类路径下META-INF/spring.factories里面配置的所有EnableAutoConfiguration的值加入到了容器中

```
1 # Initializers
2 org.springframework.context.ApplicationContextInitializer=\
3 org.springframework.boot.autoconfigure.SharedMetadataReaderFactoryContextIn
4   itializer,\
5 org.springframework.boot.autoconfigure.logging.ConditionEvaluationReportLog
6   gingListener
7
8 # Application Listeners
```

```
7 org.springframework.context.ApplicationListener=\
8 org.springframework.boot.autoconfigure.BackgroundPreinitializer
9
10 # Auto Configuration Import Listeners
11 org.springframework.boot.autoconfigure.AutoConfigurationImportListener=\
12 org.springframework.boot.autoconfigure.condition.ConditionEvaluationReportA
13   utoConfigurationImportListener
14
15 # Auto Configuration Import Filters
16 org.springframework.boot.autoconfigure.AutoConfigurationImportFilter=\
17 org.springframework.boot.autoconfigure.condition.OnBeanCondition,\
18 org.springframework.boot.autoconfigure.condition.OnClassCondition,\
19 org.springframework.boot.autoconfigure.condition.OnWebApplicationCondition
20
21 # Auto Configure
22 org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
23 org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoC
24   onfiguration,\
25 org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
26 org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
27 org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
28 org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
29 org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration
30   ,\
31 org.springframework.boot.autoconfigure.cloud.CloudServiceConnectorsAutoConf
32   igation,\
33 org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoC
34   onfiguration,\
35 org.springframework.boot.autoconfigure.context.MessageSourceAutoConfigurati
36   on,\
37 org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfi
38   guration,\
39 org.springframework.boot.autoconfigure.couchbase.CouchbaseAutoConfiguration
40   ,\
41 org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationA
42   utoConfiguration,\
43 org.springframework.boot.autoconfigure.data.cassandra.CassandraDataAutoConf
44   igation,\
45 org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveData
46   AutoConfiguration,\
47 org.springframework.boot.autoconfigure.data.cassandra.CassandraReactiveRepo
48   sitoriesAutoConfiguration,\
49 org.springframework.boot.autoconfigure.data.cassandra.CassandraRepositories
50   AutoConfiguration,\
51 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseDataAutoConf
52   igation,\
53 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveData
54   AutoConfiguration,\
55 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseReactiveRepo
56   sitoriesAutoConfiguration,\
57 org.springframework.boot.autoconfigure.data.couchbase.CouchbaseRepositories
58   AutoConfiguration,\
59 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchAuto
60   Configuration,\
61 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchData
62   AutoConfiguration,\
63 org.springframework.boot.autoconfigure.data.elasticsearch.ElasticsearchRepo
64   sitoriesAutoConfiguration,\
```

```
45 org.springframework.boot.autoconfigure.data.jdbc.JdbcRepositoriesAutoConfig
    uration,\
46 org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfigur
    ation,\
47 org.springframework.boot.autoconfigure.data ldap.LdapRepositoriesAutoConfig
    uration,\
48 org.springframework.boot.autoconfigure.data.mongo.MongoDataAutoConfiguratio
    n,\
49 org.springframework.boot.autoconfigure.data.mongo.MongoReactiveDataAutoConf
    igation,\
50 org.springframework.boot.autoconfigure.data.mongo.MongoReactiveRepositories
    AutoConfiguration,\
51 org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConf
    igation,\
52 org.springframework.boot.autoconfigure.data.neo4j.Neo4jDataAutoConfiguratio
    n,\
53 org.springframework.boot.autoconfigure.data.neo4j.Neo4jRepositoriesAutoConf
    igation,\
54 org.springframework.boot.autoconfigure.data.solr.SolrRepositoriesAutoConfig
    uration,\
55 org.springframework.boot.autoconfigure.data.redis.RedisAutoConfiguration,\
56 org.springframework.boot.autoconfigure.data.redis.RedisReactiveAutoConfigur
    ation,\
57 org.springframework.boot.autoconfigure.data.redis.RedisRepositoriesAutoConf
    igation,\
58 org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfi
    guration,\
59 org.springframework.boot.autoconfigure.data.web.SpringDataWebAutoConfigurat
    ion,\
60 org.springframework.boot.autoconfigure.elasticsearch.jest.JestAutoConfigura
    tion,\
61 org.springframework.boot.autoconfigure.elasticsearch.rest.RestClientAutoCon
    figuration,\
62 org.springframework.boot.autoconfigure.flyway.FlywayAutoConfiguration,\
63 org.springframework.boot.autoconfigure.freemarker.FreeMarkerAutoConfigurati
    on,\
64 org.springframework.boot.autoconfigure.gson.GsonAutoConfiguration,\
65 org.springframework.boot.autoconfigure.h2.H2ConsoleAutoConfiguration,\
66 org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration,
    \
67 org.springframework.boot.autoconfigure.hazelcast.HazelcastAutoConfiguration
    ,\
68 org.springframework.boot.autoconfigure.hazelcast.HazelcastJpaDependencyAuto
    Configuration,\
69 org.springframework.boot.autoconfigure.http.HttpMessageConvertersAutoConfig
    uration,\
70 org.springframework.boot.autoconfigure.http.codec.CodecsAutoConfiguration,\
71 org.springframework.boot.autoconfigure.influx.InfluxDbAutoConfiguration,\
72 org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration,\
73 org.springframework.boot.autoconfigure.integration.IntegrationAutoConfigura
    tion,\
74 org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration,\
75 org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
76 org.springframework.boot.autoconfigure.jdbc.JdbcTemplateAutoConfiguration,\
77 org.springframework.boot.autoconfigure.jdbc.JndiDataSourceAutoConfiguration
    ,\
78 org.springframework.boot.autoconfigure.jdbc.XADataSourceAutoConfiguration,\
```

```
79 org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAut
oConfiguration,\
80 org.springframework.boot.autoconfigure.jms.JmsAutoConfiguration,\
81 org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
82 org.springframework.boot.autoconfigure.jms.JndiConnectionFactoryAutoConfigu
ration,\
83 org.springframework.boot.autoconfigure.jms.activemq.ActiveMQAutoConfigurati
on,\
84 org.springframework.boot.autoconfigure.jms.artemis.ArtemisAutoConfiguration
,\
85 org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAutoCo
nfiguration,\
86 org.springframework.boot.autoconfigure.jersey.JerseyAutoConfiguration,\
87 org.springframework.boot.autoconfigure.jooq.JooqAutoConfiguration,\
88 org.springframework.boot.autoconfigure.jsonb.JsonbAutoConfiguration,\
89 org.springframework.boot.autoconfigure.kafka.KafkaAutoConfiguration,\
90 org.springframework.boot.autoconfigure.ldap.embedded.EmbeddedLdapAutoConfig
uration,\
91 org.springframework.boot.autoconfigure.ldap.LdapAutoConfiguration,\
92 org.springframework.boot.autoconfigure liquibase.LiquibaseAutoConfiguration
,\
93 org.springframework.boot.autoconfigure.mail.MailSenderAutoConfiguration,\
94 org.springframework.boot.autoconfigure.mail.MailSenderValidatorAutoConfigur
ation,\
95 org.springframework.boot.autoconfigure.mongo.embedded.EmbeddedMongoAutoConf
iguration,\
96 org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
97 org.springframework.boot.autoconfigure.mongo.MongoReactiveAutoConfiguration
,\
98 org.springframework.boot.autoconfigure.mustache.MustacheAutoConfiguration,\
99 org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguratio
n,\
100 org.springframework.boot.autoconfigure.quartz.QuartzAutoConfiguration,\
101 org.springframework.boot.autoconfigure.reactor.core.ReactorCoreAutoConfigur
ation,\
102 org.springframework.boot.autoconfigure.security.servlet.SecurityAutoConfigu
ration,\
103 org.springframework.boot.autoconfigure.security.servlet.SecurityRequestMatc
herProviderAutoConfiguration,\
104 org.springframework.boot.autoconfigure.security.servlet.UserDetailsServiceA
utoConfiguration,\
105 org.springframework.boot.autoconfigure.security.servlet.SecurityFilterAutoC
onfiguration,\
106 org.springframework.boot.autoconfigure.security.reactive.ReactiveSecurityAu
toConfiguration,\
107 org.springframework.boot.autoconfigure.security.reactive.ReactiveUserDetail
sServiceAutoConfiguration,\
108 org.springframework.boot.autoconfigure.sendgrid.SendGridAutoConfiguration,\
109 org.springframework.boot.autoconfigure.session.SessionAutoConfiguration,\
110 org.springframework.boot.autoconfigure.security.oauth2.client.servlet.OAuth
2ClientAutoConfiguration,\
111 org.springframework.boot.autoconfigure.security.oauth2.client.reactive.Reac
tiveOAuth2ClientAutoConfiguration,\
112 org.springframework.boot.autoconfigure.security.oauth2.resource.servlet.OAu
th2ResourceServerAutoConfiguration,\
113 org.springframework.boot.autoconfigure.security.oauth2.resource.reactive.Re
activeOAuth2ResourceServerAutoConfiguration,\
114 org.springframework.boot.autoconfigure.solr.SolrAutoConfiguration,\
```

```
115 org.springframework.boot.autoconfigure.task.TaskExecutionAutoConfiguration,
116 \
117 org.springframework.boot.autoconfigure.task.TaskSchedulingAutoConfiguration
118 ,\
119 org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration
120 ,\
121 org.springframework.boot.autoconfigure.transaction.TransactionAutoConfigura
122 tion,\
123 org.springframework.boot.autoconfigure.transaction.jta.JtaAutoConfiguration
124 ,\
125 org.springframework.boot.autoconfigure.validation.ValidationAutoConfigurati
126 on,\
127 org.springframework.boot.autoconfigure.web.client.RestTemplateAutoConfigura
128 tion,\
129 org.springframework.boot.autoconfigure.web.embedded.EmbeddedWebServerFactor
130 yCustomizerAutoConfiguration,\
131 org.springframework.boot.autoconfigure.web.reactive.HttpHandlerAutoConfigur
132 ation,\
133 org.springframework.boot.autoconfigure.web.reactive.ReactiveWebServerFactor
134 yAutoConfiguration,\
135 org.springframework.boot.autoconfigure.web.reactive.WebFluxAutoConfiguratio
136 n,\
137 org.springframework.boot.autoconfigure.web.reactive.error.ErrorWebFluxAutoC
138 onfiguration,\
139 org.springframework.boot.autoconfigure.web.reactive.function.client.ClientH
140 ttpConnectorAutoConfiguration,\
141 org.springframework.boot.autoconfigure.web.reactive.function.client.WebClie
142 ntAutoConfiguration,\
143 org.springframework.boot.autoconfigure.web.servlet.DispatcherServletAutoCon
144 figuration,\
145 org.springframework.boot.autoconfigure.web.servlet.ServletWebServerFactoryA
146 utoConfiguration,\
147 org.springframework.boot.autoconfigure.web.servlet.error.ErrorMvcAutoConfig
148 uration,\
149 org.springframework.boot.autoconfigure.web.servlet.HttpEncodingAutoConfigur
150 ation,\
151 org.springframework.boot.autoconfigure.web.servlet.MultipartAutoConfigurati
152 on,\
153 org.springframework.boot.autoconfigure.web.servlet.WebMvcAutoConfiguration,
154 \
155 org.springframework.boot.autoconfigure.websocket.reactive.WebSocketReactive
156 AutoConfiguration,\
157 org.springframework.boot.autoconfigure.websocket.servlet.WebSocketServletAu
158 toConfiguration,\
159 org.springframework.boot.autoconfigure.websocket.servlet.WebSocketMessaging
160 AutoConfiguration,\
161 org.springframework.boot.autoconfigure.webservices.WebServicesAutoConfigura
162 tion,\
163 org.springframework.boot.autoconfigure.webservices.client.WebServiceTemplat
164 eAutoConfiguration
165
166 # Failure analyzers
167 org.springframework.boot.diagnostics.FailureAnalyzer=\
168 org.springframework.boot.autoconfigure.diagnostics.analyzer.NoSuchBeanDefin
169 itionFailureAnalyzer,\
170 org.springframework.boot.autoconfigure.jdbc.DataSourceBeanCreationFailureAn
171 alyzer,\
```

```

145 org.springframework.boot.autoconfigure.jdbc.HikariDriverConfigurationFailur
    eAnalyzer,\
146 org.springframework.boot.autoconfigure.session.NonUniqueSessionRepositoryFa
    ilureAnalyzer
147
148 # Template availability providers
149 org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvide
    r=\
150 org.springframework.boot.autoconfigure.freemarker.FreeMarkerTemplateAvaila
    bilityProvider,\
151 org.springframework.boot.autoconfigure.mustache.MustacheTemplateAvailabilit
    yProvider,\
152 org.springframework.boot.autoconfigure.groovy.template.GroovyTemplateAvaila
    bilityProvider,\
153 org.springframework.boot.autoconfigure.thymeleaf.ThymeleafTemplateAvailabil
    ityProvider,\
154 org.springframework.boot.autoconfigure.web.servlet.JspTemplateAvailabilityP
    rovider

```

每一个这样的AutoConfiguration类都是容器中的一个组件，都加入到容器中，用他们来做配置

3) 每一个自动配置类进行自动配置功能;

4) 以HttpEncodingAutoConfiguration (Http编码自动配置)为例解释自动配置原理;

```

1 //表示这是一个配置类没以前编写的配置文件一样，也可以给容器中添加组件
2 @Configuration
3 //启动指定类的ConfigurationProperties功能，将配置文件中对应的值和
  HttpEncodingAutoConfigurationProperties绑定起来;
4 @EnableConfigurationProperties({HttpProperties.class})
5 //Spring底层@Conditional注解，根据不同的条件，如果满足指定的条件，整个配置类里面的配置
  就会生效(即判断当前应用是否是web应用)
6 @ConditionalOnWebApplication(
7     type = Type.SERVLET
8 )
9 //判断当前项目有没有这个类CharacterEncodingFilter; SpringMVC中进行乱码处理的过滤器
10 @ConditionalOnClass({CharacterEncodingFilter.class})
11 @ConditionalOnProperty(
12     prefix = "spring.http.encoding",
13     value = {"enabled"},
14     matchIfMissing = true
15 )
16 public class HttpEncodingAutoConfiguration{
17 //给容器中添加一个组件
18 @Bean
19     @ConditionalOnMissingBean
20     public CharacterEncodingFilter characterEncodingFilter() {
21         CharacterEncodingFilter filter = new
  OrderedCharacterEncodingFilter();
22         filter.setEncoding(this.properties.getCharset().name());
23
24         filter.setForceRequestEncoding(this.properties.shouldForce(org.springframework
    work.boot.autoconfigure.http.HttpProperties.Encoding.Type.REQUEST));
25
26         filter.setForceResponseEncoding(this.properties.shouldForce(org.springframe
    work.boot.autoconfigure.http.HttpProperties.Encoding.Type.RESPONSE));

```



```

25         return filter;
26     }
27 }
28 根据当前不同的条件判断，决定这个配置类是否生效？
29 一旦这个配置类生效，这个配置类就会给容器中添加各种组件，这些组件的属性是从对应的
    properties类中获取的，这些类里面的每一个属性又是和配置文件绑定的。

```

5) 所有在配置文件中能配置的属性都是在xxxxPropertites类中封装着，配置文件能配置什么就可以参照某个功能对应的这个属性类；

```

1  //从配置文件中获取指定的值和bean的属性进行绑定
2  @ConfigurationProperties(
3      prefix = "spring.http"
4  )
5  public class HttpProperties {
6      private boolean logRequestDetails;
7      private final HttpProperties.Encoding encoding = new
        HttpProperties.Encoding();

```

2、自定义starter

新建hello-spring-boot-starter工程

1.pom.xml

```

1  <?xml version="1.0" encoding="UTF-8"?>
2  <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5      http://maven.apache.org/xsd/maven-4.0.0.xsd">
6      <modelVersion>4.0.0</modelVersion>
7      <parent>
8          <groupId>org.springframework.boot</groupId>
9          <artifactId>spring-boot-starter-parent</artifactId>
10         <version>2.3.0.RELEASE</version>
11         <relativePath/> <!-- lookup parent from repository -->
12     </parent>
13     <groupId>com.xinzhi</groupId>
14     <artifactId>hello-spring-boot-starter</artifactId>
15     <version>0.0.1-SNAPSHOT</version>
16     <name>hello-spring-boot-starter</name>
17     <description>Demo project for Spring Boot</description>
18
19     <properties>
20         <java.version>1.8</java.version>
21     </properties>
22
23     <dependencies>
24         <dependency>
25             <groupId>org.springframework.boot</groupId>
26             <artifactId>spring-boot-autoconfigure</artifactId>
27         </dependency>
28     </dependencies>

```

```
27 |
28 | </project>
```

同时删除 启动类、resources下的文件，test文件。

HelloProperties

```
1  /**
2   * @author IT楠老师
3   * @date 2020/6/5
4   */
5  @ConfigurationProperties(prefix = "itnanls.user")
6  public class UserProperties {
7      private String username = "张三";
8      private String password = "123";
9
10     public String getUsername() {
11         return username;
12     }
13
14     public void setUsername(String username) {
15         this.username = username;
16     }
17
18     public String getPassword() {
19         return password;
20     }
21
22     public void setPassword(String password) {
23         this.password = password;
24     }
25 }
```

HelloService

```
1  /**
2   * @author IT楠老师
3   * @date 2020/6/5
4   */
5  @Component
6  public class User {
7
8      @Resource
9      private UserProperties userProperties;
10
11     public void say(){
12         System.out.println(userProperties.getUsername()+"的密码
是: "+userProperties.getPassword());
13     }
14 }
15 }
```

UserAutoConfiguration

```
1  /**
2   * @author IT楠老师
```

```

3      * @date 2020/6/5
4      */
5      @Configuration
6      @ConditionalOnClass(User.class)
7      @ConditionalOnProperty(prefix = "xinzhi.user",value =
      "enable",matchIfMissing = true)
8      @EnableConfigurationProperties(UserProperties.class)
9      public class UserAutoConfiguration {
10
11          @Bean
12          @ConditionalOnMissingBean
13          public User greeter() {
14              return new User();
15          }
16      }

```

5. spring.factories

在 **resources** 下创建文件夹 **META-INF** 并在 **META-INF** 下创建文件 **spring.factories** , 内容如下:

```

1      # Auto Configure
2      org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
3      cn.itnanls.config.UserAutoConfiguration

```

到这儿, 我们的配置自定义的starter就写完了

三、测试自定义starter

我们创建个项目 **hello-spring-boot-starter-test**, 来测试系我们写的stater。

1. pom.xml

```

1      <?xml version="1.0" encoding="UTF-8"?>
2      <project xmlns="http://maven.apache.org/POM/4.0.0"
3      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4      xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
5      https://maven.apache.org/xsd/maven-4.0.0.xsd">
6          <modelVersion>4.0.0</modelVersion>
7          <parent>
8              <groupId>org.springframework.boot</groupId>
9              <artifactId>spring-boot-starter-parent</artifactId>
10             <version>2.3.0.RELEASE</version>
11             <relativePath/> <!-- lookup parent from repository -->
12         </parent>
13         <groupId>com.example</groupId>
14         <artifactId>spring-boot-initializr</artifactId>
15         <version>0.0.1-SNAPSHOT</version>
16         <name>spring-boot-initializr</name>
17         <description>Demo project for Spring Boot</description>
18
19         <properties>
20             <java.version>1.8</java.version>
21         </properties>

```

```

21     <dependencies>
22         <dependency>
23             <groupId>org.springframework.boot</groupId>
24             <artifactId>spring-boot-starter-web</artifactId>
25         </dependency>
26         <dependency>
27             <groupId>com.xinzhi</groupId>
28             <artifactId>hello-spring-boot-starter</artifactId>
29             <version>0.0.1-SNAPSHOT</version>
30         </dependency>
31     </dependencies>
32
33     <build>
34         <plugins>
35             <plugin>
36                 <groupId>org.springframework.boot</groupId>
37                 <artifactId>spring-boot-maven-plugin</artifactId>
38             </plugin>
39         </plugins>
40     </build>
41
42 </project>

```

2. HelloController

```

1  @Controller
2  public class UserController {
3
4      @Resource
5      private User user;
6
7      @GetMapping("/user")
8      @ResponseBody
9      public void getUser(){
10         user.say();
11     }
12 }

```

 1591340121810

3. 修改application.properties

```

1  xinzhi.user.username=李四
2  xinzhi.user.password=abc

```

 1591340475864

B站: IT楠老师 公众号: IT楠说java QQ群: 1083478826 新知大数据

制作不易、如果觉的好不妨打个赏: