Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Spring Boot致力于在蓬勃发展的快速应用开发领域(rapid application development)成为领导者。

# 介绍

## 1．特点

1. 创建独立的Spring应用程序

2. 嵌入的Tomcat，无需部署WAR文件

3. 简化Maven配置

4. 自动配置Spring

5. 提供生产就绪型功能，如指标，健康检查和外部配置

6. 绝对没有代码生成和对XML没有要求配置

## 2．使用 Spring Boot 有什么好处

回顾我们之前的 SSM 项目，搭建过程还是比较繁琐的，需要：

1．配置 web.xml，加载 spring 和 springmvc

2．配置数据库连接、配置日志文件

3．配置家在配置文件的读取，开启注解

4．配置mapper文件

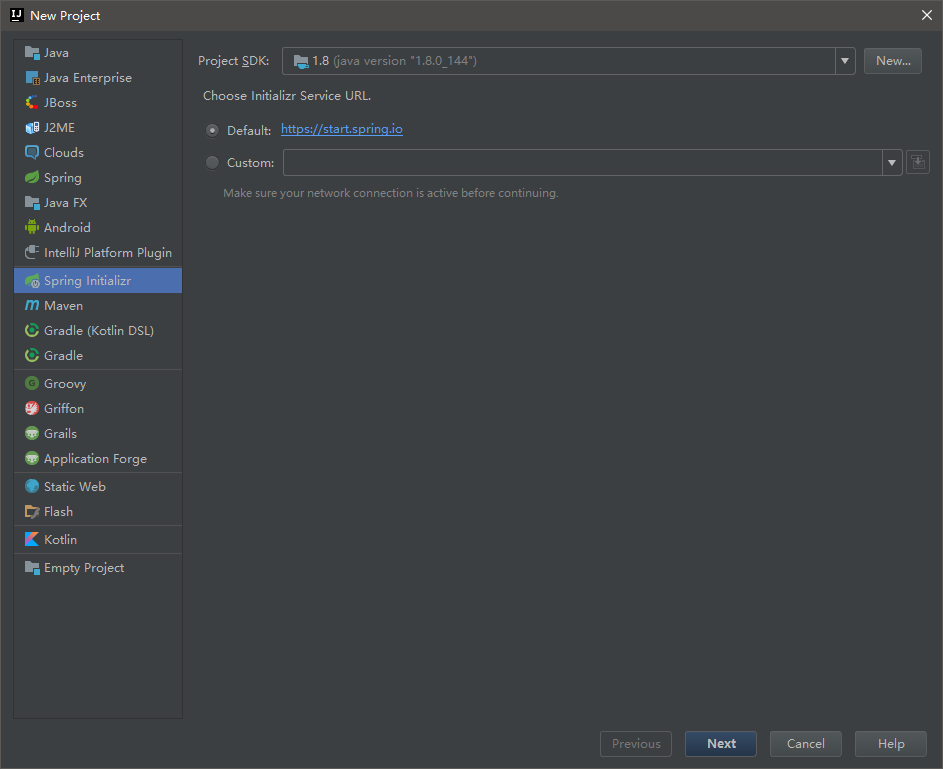
5．.…………..

而使用 Spring Boot 来开发项目则只需要非常少的几个配置就可以搭建起来一个 Web 项目，并且利用 IDEA 可以自动生成生成

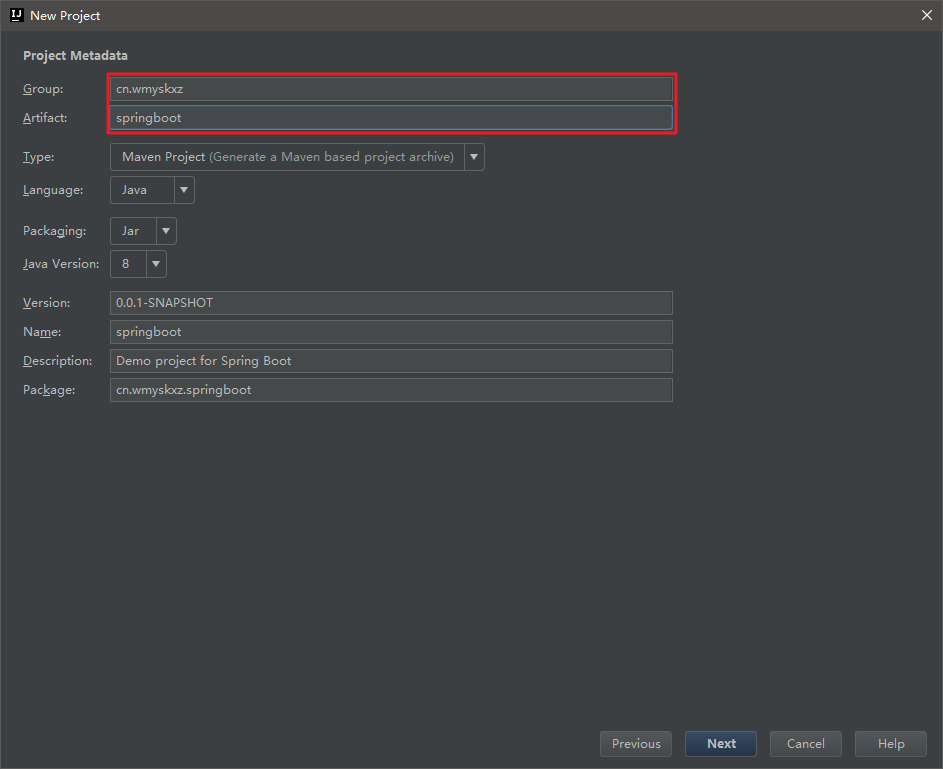
# 二．IDEA通过快速搭建Spring Boot

## 1．新建项目

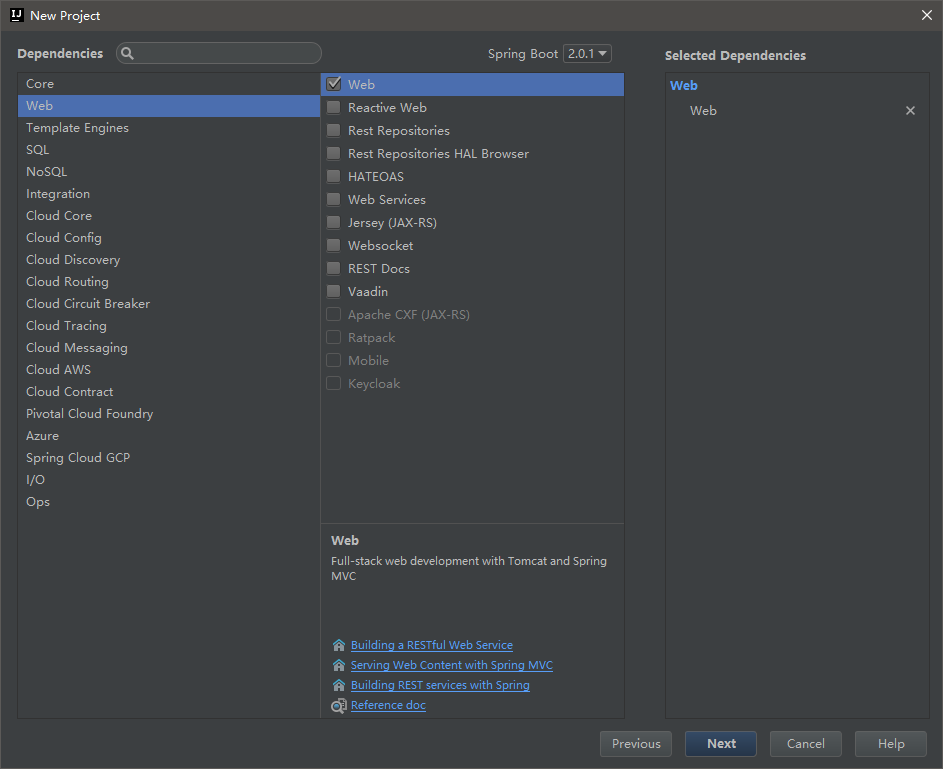
选择 Spring Initializr ，然后选择默认的 url 点击【Next】：



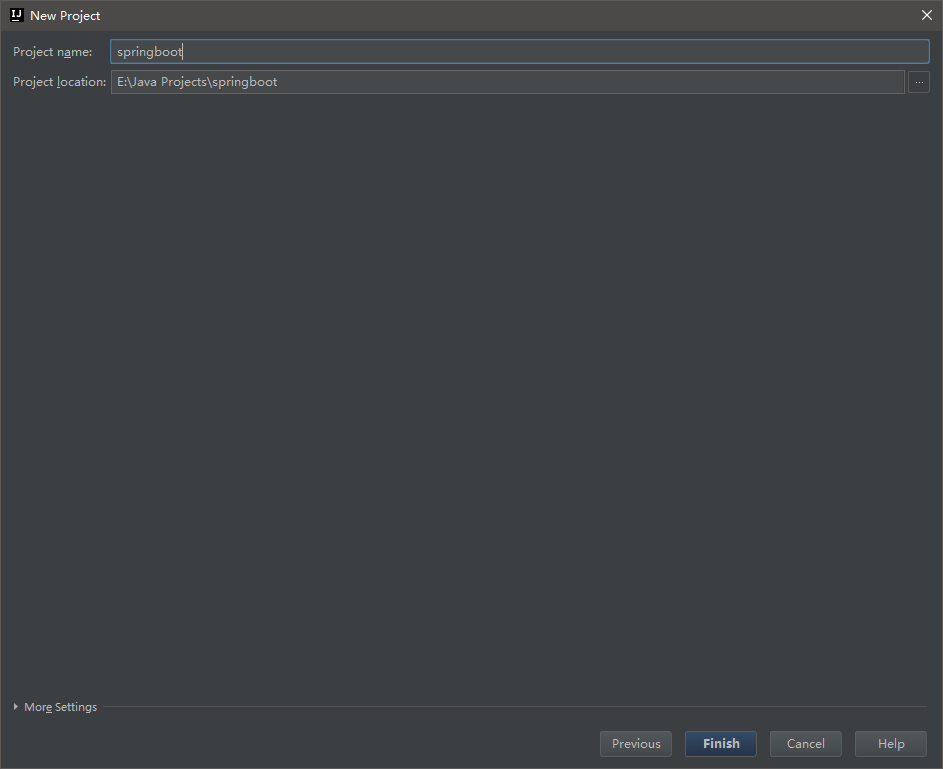
然后修改一下项目的信息：



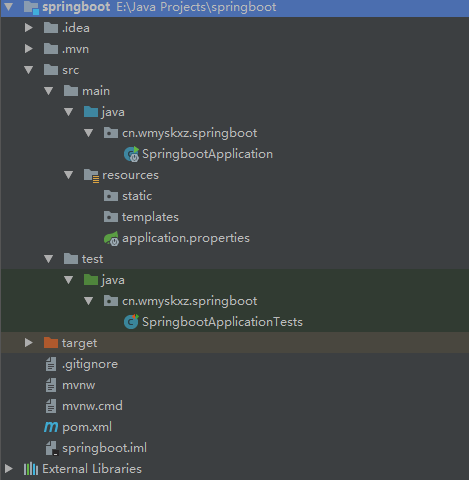
勾选上 Web 模板：



选择好项目的位置，点击【Finish】：



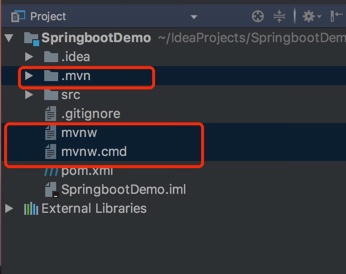
如果是第一次配置 Spring Boot 的话可能需要等待一会儿 IDEA下载相应的依赖包，默认创建好的项目结构如下：



项目结构还是看上去挺清爽的，少了很多配置文件，我们来了解一下默认生成的有什么：

* SpringbootApplication： 一个带有 main() 方法的类，用于启动应用程序
* SpringbootApplicationTests：一个空的 Junit 测试了，它加载了一个使用 Spring Boot 字典配置功能的 Spring 应用程序上下文
* application.properties：一个空的 properties 文件，可以根据需要添加配置属性
* pom.xml： Maven 构建说明文件

进入项目，可以将以下内容删除【也可以不删除】



## 2．解析 Spring Boot 项目

### 2.1 解析 pom.xml 文件

让我们来看看默认生成的 pom.xml 文件中到底有一些什么特别

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  <modelVersion>4.0.0</modelVersion>  <groupId>cn.wmyskxz</groupId>  <artifactId>springboot</artifactId>  <version>0.0.1-SNAPSHOT</version>  <packaging>jar</packaging>  <name>springboot</name>  <description>Demo project for Spring Boot</description>  <!--起步依赖-->  <parent>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-parent</artifactId>  <version>2.0.1.RELEASE</version>  <relativePath/> <!-- lookup parent from repository -->  </parent>  <properties>  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>  <java.version>1.8</java.version>  </properties>  <dependencies>   <!--开发web项目相关依赖-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  </dependency>  <!--springboot单元测试-->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-test</artifactId>  <scope>test</scope>  </dependency>  </dependencies>  <!--maven构建-->  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

### 2.2 应用入口类

Spring Boot 项目通常有一个名为 \*Application 的入口类，入口类里有一个 main 方法， 这个 main 方法其实就是一个标准的 Java应用的入口方法

@SpringBootApplication开启了Spring的组件扫描和springboot的自动配置功能，相当于将以下三个注解组合在了一起

（1）@Configuration：表明该类使用基于Java的配置,将此类作为配置类

（2）@ComponentScan：启用注解扫描

（3）@EnableAutoConfiguration：开启springboot的自动配置功能，让 Spring Boot 根据类路径中的 jar 包依赖为当前项目进行自动配置\*\*，例如，添加了 spring-boot-starter-web 依赖，会自动添加 Tomcat 和 Spring MVC 的依赖，那么 Spring Boot 会对 Tomcat 和 Spring MVC 进行自动配置。

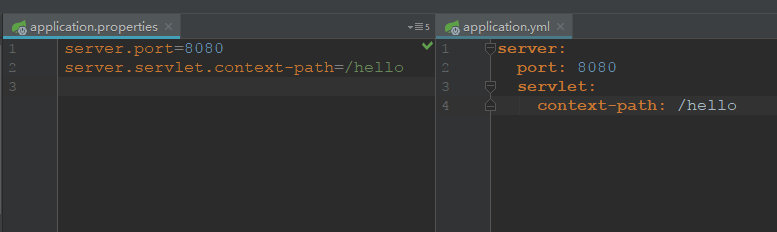
### 2.3 Spring Boot 的配置文件

Spring Boot 使用一个全局的配置文件 application.properties 或 application.yml，放置在【src/main/resources】目录或者类路径的 /config 下。

Spring Boot 不仅支持常规的 properties 配置文件，还支持 yaml 语言的配置文件。yaml 是以数据为中心的语言，在配置数据的时候具有面向对象的特征。

Spring Boot 的全局配置文件的作用是对一些默认配置的配置值进行修改。

* 简单实例如下



我们同样的将 Tomcat 默认端口设置为 8080 ，并将默认的访问路径从 “/” 修改为 “/hello” 时，使用 properties 文件和 yml 文件的区别如上图。

注意： yml 需要在 “:” 后加一个空格，幸好 IDEA 很好地支持了 yml 文件的格式有良好的代码提示，我们可以自己配置多个属性

常用配置

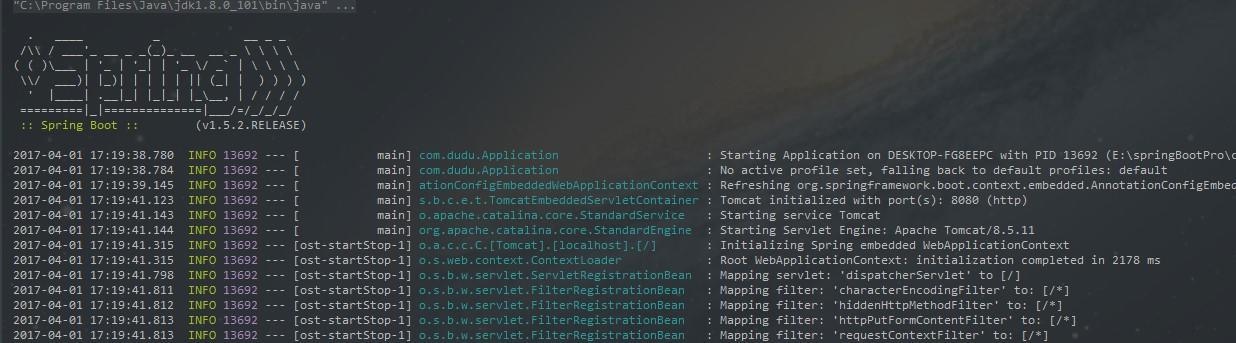
|  |
| --- |
| #设置Tomcat端口，默认8080  server.port=8080  #设置项目ContextPath  server.context-path=/  #设置Tomcat编码  server.tomcat.uri-encoding=UTF-8  #设置视图解析器路径  spring.mvc.view.prefix=/WEB-INF/views/  #设置视图解析器后缀  spring.mvc.view.suffix=.jsp  #数据库配置  spring.datasource.url=jdbc:mysql://localhost:3306/dd?useUnicode=true&characterEncoding=utf-8&zeroDateTimeBehavior=convertToNull&allowMultiQueries=true  spring.datasource.username=root  spring.datasource.password=root  spring.datasource.driver-class-name=com.mysql.jdbc.Driver  #配置.xml文件路径  mybatis.mapper-locations=classpath:/com/dd/wx/mapper/\*.xml  #配置模型路径  mybatis.type-aliases-package=com.dd.wx.model |

### 2.4 配置日志

#### 2.4.1 配置 logback（官方推荐使用）

##### 1．默认配置：

默认情况下，Spring Boot会用Logback来记录日志，并用INFO级别输出到控制台。在运行应用程序和其他例子时，你应该已经看到很多INFO级别的日志了。



从上图可以看到，日志输出内容元素具体如下：

时间日期：精确到毫秒

日志级别：ERROR, WARN, INFO, DEBUG or TRACE

进程ID

分隔符：— 标识实际日志的开始

线程名：方括号括起来（可能会截断控制台输出）

Logger名：通常使用源代码的类名

日志内容

|  |
| --- |
| #logging.path=c:/logs  #logging.file=c:/logs/log2018.txt  #logging.pattern.console="%d-%msg%n"  logging.level.com.dudu=DEBUG：com.dudu包下所有class以DEBUG级别输出  logging.level.root=WARN：root日志以WARN级别输出 |

##### 2．通过额外的文件配置Logback

通过application.properties文件配置Logback,对于大多数Spring Boot应用来说已经足够了，但是对于一些大型的企业应用来说似乎有一些相对复杂的日志需求。

1) 配置日志文件

spring boot 默认会加载 classpath:logback-spring.xml 或者 classpath:logback-spring.groovy。

logback-spring.xml, logback-spring.groovy, logback.xml, logback.groovy

如需要自定义文件名称，在 application.properties 中配置 logging.config 选项即可。

EG:logging.config=classpath:logging-config.xml

在 src/main/resources 下创建 logback-spring.xml 文件

2) 添加日志依赖

假如maven依赖中添加了spring-boot-starter-logging

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-logging</artifactId>

</dependency>

但是呢，实际开发中我们不需要直接添加该依赖。你会发现spring-boot-starter其中包含了 spring-boot-starter-logging，该依赖内容就是 Spring Boot 默认的日志框架 logback。工程中有用到了Thymeleaf，而Thymeleaf依赖包含了spring-boot-starter，最终我只要引入Thymeleaf即可。

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-thymeleaf</artifactId>

</dependency>

简单的日志配置

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8" ?>  <configuration>  <!--输出日志到控制台-->  <appender name="consoleLog" class="ch.qos.logback.core.ConsoleAppender">  <layout>  <pattern>  [%p]%d-%msg%n  </pattern>  </layout>  </appender>  <!--输出日志到文件-->  <appender name="fileLog" class="ch.qos.logback.core.rolling.RollingFileAppender">  <encoder>  <pattern>  [%p]%d-%msg%n  </pattern>  </encoder>  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  <fileNamePattern>  c:/logs/%d.log  </fileNamePattern>  </rollingPolicy>  </appender>  <!--指定最基础的日志输出级别-->  <root level="debug">  <appender-ref ref="consoleLog"></appender-ref>  <appender-ref ref="fileLog"></appender-ref>  </root>  </configuration> |

##### 3．输出日志到控制台

|  |
| --- |
| <?xml version="1.0" encoding="UTF-8"?>  <configuration scan="true" scanPeriod="60 seconds" debug="false">  <contextName>logback-demo</contextName>  <!--输出到控制台 ConsoleAppender-->  <appender name="consoleLog1" class="ch.qos.logback.core.ConsoleAppender">  <!--展示格式 layout-->  <layout class="ch.qos.logback.classic.PatternLayout">  <pattern>  <pattern>%d{HH:mm:ss.SSS} %contextName [%thread] %-5level %logger{36} - %msg%n</pattern>  </pattern>  </layout>  </appender>  <!--输出到控制台 ConsoleAppender-->  <appender name="consoleLog2" class="ch.qos.logback.core.ConsoleAppender">  <encoder>  <pattern>%d -2 %msg%n</pattern>  </encoder>  </appender>  <!--指定最基础的日志输出级别-->  <root level="INFO">  <!--appender将会添加到这个loger-->  <appender-ref ref="consoleLog1"/>  <appender-ref ref="consoleLog2"/>  </root>  </configuration> |

<encoder>表示对日志进行编码：

%d{HH: mm:ss.SSS}——日志输出时间

%thread——输出日志的进程名字，这在Web应用以及异步任务处理中很有用

%-5level——日志级别，并且使用5个字符靠左对齐

%logger{36}——日志输出者的名字

%msg——日志消息

%n——平台的换行符   
ThresholdFilter为系统定义的拦截器，例如我们用ThresholdFilter来过滤掉ERROR级别以下的日志不输出到文件中。如果不用记得注释掉，不然你控制台会发现没日志

##### 4．输出日志到文件

输出到文件 RollingFileAppender

另一种常见的日志输出到文件，随着应用的运行时间越来越长，日志也会增长的越来越多，将他们输出到同一个文件并非一个好办法。RollingFileAppender用于切分文件日志

|  |
| --- |
| <appender name="fileInfoLog" class="ch.qos.logback.core.rolling.RollingFileAppender">  <!--如果只是想要 Info 级别的日志，只是过滤 info 还是会输出 Error 日志，因为 Error 的级别高，  所以我们使用下面的策略，可以避免输出 Error 的日志-->  <filter class="ch.qos.logback.classic.filter.LevelFilter">  <!--过滤 Error-->  <level>ERROR</level>  <!--匹配到就禁止-->  <onMatch>DENY</onMatch>  <!--没有匹配到就允许-->  <onMismatch>ACCEPT</onMismatch>  </filter>  <!--日志名称，如果没有File 属性，那么只会使用FileNamePattern的文件路径规则  如果同时有<File>和<FileNamePattern>，那么当天日志是<File>，明天会自动把今天  的日志改名为今天的日期。即，<File> 的日志都是当天的。  -->  <File>${logback.logdir}/info.${logback.appname}.log</File>  <!--滚动策略，按照时间滚动 TimeBasedRollingPolicy-->  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  <!--文件路径,定义了日志的切分方式——把每一天的日志归档到一个文件中,以防止日志填满整个磁盘空间-->  <FileNamePattern>${logback.logdir}/info.${logback.appname}.%d{yyyy-MM-dd}.log</FileNamePattern>  <!--只保留最近90天的日志-->  <maxHistory>90</maxHistory>  <!--用来指定日志文件的上限大小，那么到了这个值，就会删除旧的日志-->  <!--<totalSizeCap>1GB</totalSizeCap>-->  </rollingPolicy>  <!--日志输出编码格式化-->  <encoder>  <charset>UTF-8</charset>  <pattern>%d [%thread] %-5level %logger{36} %line - %msg%n</pattern>  </encoder>  </appender>  <appender name="fileErrorLog" class="ch.qos.logback.core.rolling.RollingFileAppender">  <!--如果只是想要 Error 级别的日志，那么需要过滤一下，默认是 info 级别的，ThresholdFilter-->  <filter class="ch.qos.logback.classic.filter.ThresholdFilter">  <level>Error</level>  </filter>  <!--日志名称，如果没有File 属性，那么只会使用FileNamePattern的文件路径规则  如果同时有<File>和<FileNamePattern>，那么当天日志是<File>，明天会自动把今天  的日志改名为今天的日期。即，<File> 的日志都是当天的。  -->  <File>${logback.logdir}/error.${logback.appname}.log</File>  <!--滚动策略，按照时间滚动 TimeBasedRollingPolicy-->  <rollingPolicy class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">  <!--文件路径,定义了日志的切分方式——把每一天的日志归档到一个文件中,以防止日志填满整个磁盘空间-->  <FileNamePattern>${logback.logdir}/error.${logback.appname}.%d{yyyy-MM-dd}.log</FileNamePattern>  <!--只保留最近90天的日志-->  <maxHistory>90</maxHistory>  <!--用来指定日志文件的上限大小，那么到了这个值，就会删除旧的日志-->  <!--<totalSizeCap>1GB</totalSizeCap>-->  </rollingPolicy>  <!--日志输出编码格式化-->  <encoder>  <charset>UTF-8</charset>  <pattern>%d [%thread] %-5level %logger{36} %line - %msg%n</pattern>  </encoder>  </appender> |

##### 5．节点介绍

根节点包含的属性

scan : 当此属性设置为true时，配置文件如果发生改变，将会被重新加载，默认值为true。

scanPeriod: 设置监测配置文件是否有修改的时间间隔，如果没有给出时间单位，默认单位是毫秒。当scan为true时，此属性生效。默认的时间间隔为1分钟。

debug: 当此属性设置为true时，将打印出logback内部日志信息，实时查看logback运行状态。默认值为false。

###### 5.1 子节点一 <root>

root节点是必选节点，用来指定最基础的日志输出级别，只有一个level属性。

level:用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF，不能设置为INHERITED或者同义词NULL。   
默认是DEBUG。   
可以包含零个或多个元素，标识这个appender将会添加到这个loger。

<root level="debug">

<appender-ref ref="console" />

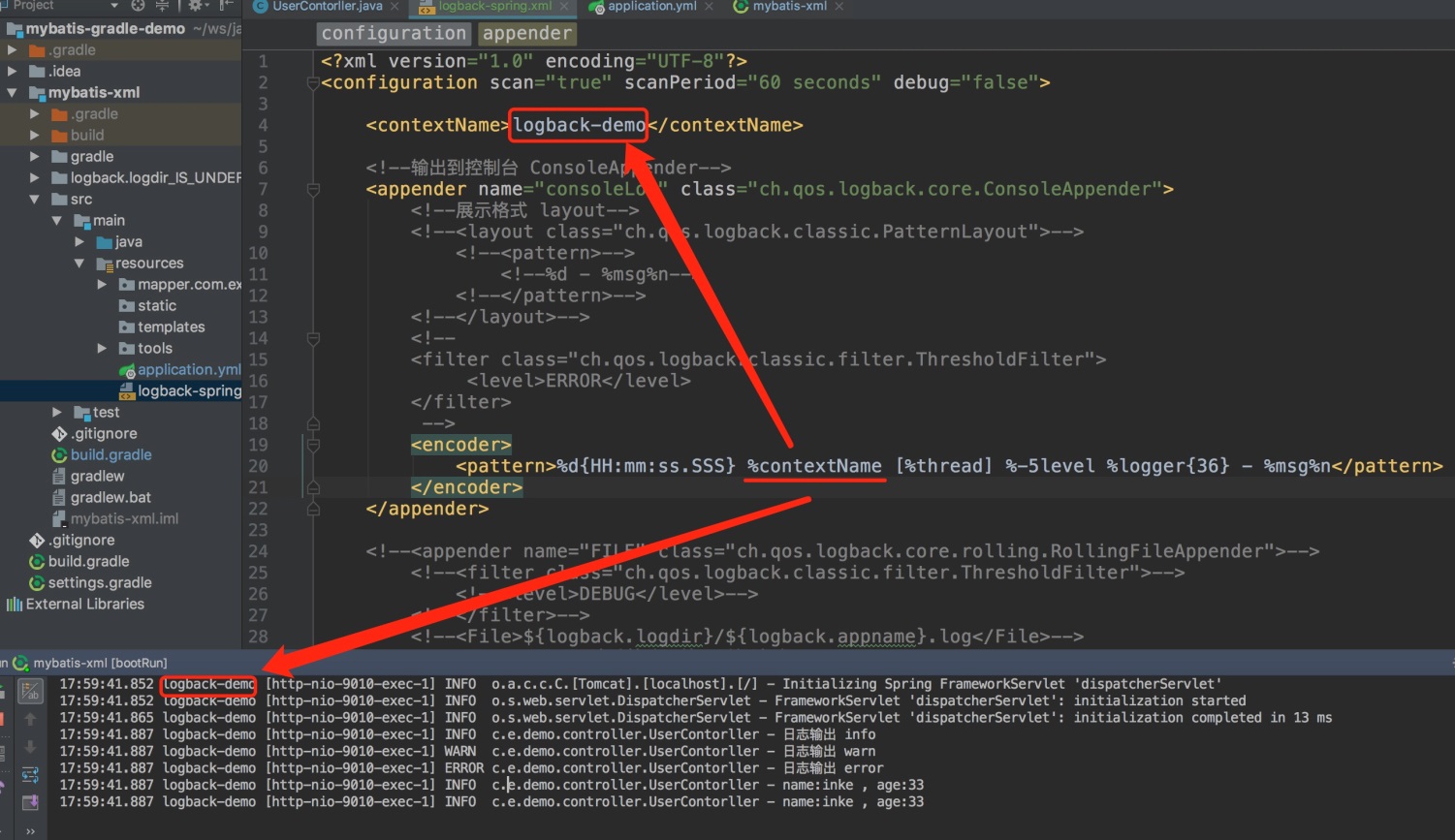
<appender-ref ref="file" />

</root>

###### 5.2 子节点二：<contextName> 设置上下文名称

每个logger都关联到logger上下文，默认上下文名称为“default”。但可以使用设置成其他名字，用于区分不同应用程序的记录。一旦设置，不能修改,可以通过%contextName来打印日志上下文名称，一般来说我们不用这个属性，可有可无。

<contextName>logback</contextName



###### 5.3 子节点三：<property> 设置变量

用来定义变量值的标签， 有两个属性，name和value；其中name的值是变量的名称，value的值时变量定义的值。通过定义的值会被插入到logger上下文中。定义变量后，可以使“${}”来使用变量。

<property name="logback.logdir" value="/Users/inke/dev/log/tomcat"/>

<property name="logback.appname" value="app"/>

###### 5.4 子节点四：<appender>

appender用来格式化日志输出节点，有俩个属性name和class，class用来指定哪种输出策略，常用就是控制台输出策略和文件输出策略。

###### 5.5 子节点五<loger>

<loger>用来设置某一个包或者具体的某一个类的日志打印级别、以及指定<appender>。<loger>仅有一个name属性，一个可选的level和一个可选的addtivity属性。

name: 用来指定受此loger约束的某一个包或者具体的某一个类。

level: 用来设置打印级别，大小写无关：TRACE, DEBUG, INFO, WARN, ERROR, ALL 和 OFF，还有一个特俗值INHERITED或者同义词NULL，代表强制执行上级的级别。如果未设置此属性，那么当前loger将会继承上级的级别。

Addtivity: 是否向上级loger传递打印信息。默认是true。

<!--logback.LogbackDemo：类的全路径 -->

<logger name="com.dudu.controller.LearnController" level="WARN" additivity="false">

<appender-ref ref="console"/>

</logger>

### 2.5 热部署

#### 2.5.1 修改pom文件

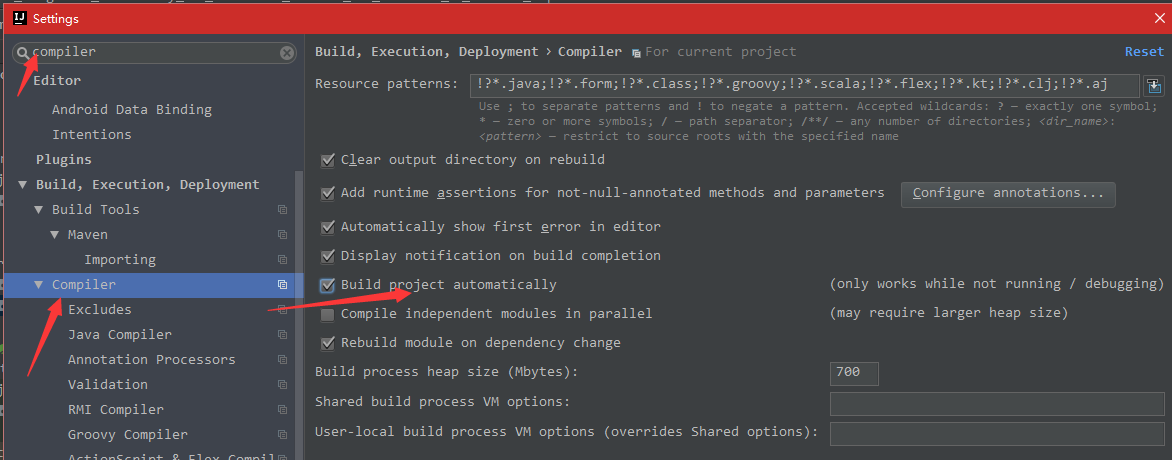
|  |
| --- |
| <!-- 热部署 -->  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-devtools</artifactId>  <optional>true</optional>  <scope>true</scope>  </dependency>  <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>    <!-- 没有该配置，devtools 不生效 -->  <**configuration**>  <!-- 没有该配置，devtools 不生效 -->  <**fork**>true</**fork**>  <**addResources**>true</**addResources**>  </**configuration**>    </plugin>  </plugins>  </build> |

#热部署生效

spring.devtools.restart.enabled=true

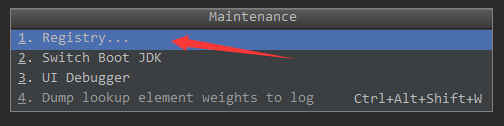
#### 2.5.2 修改idea

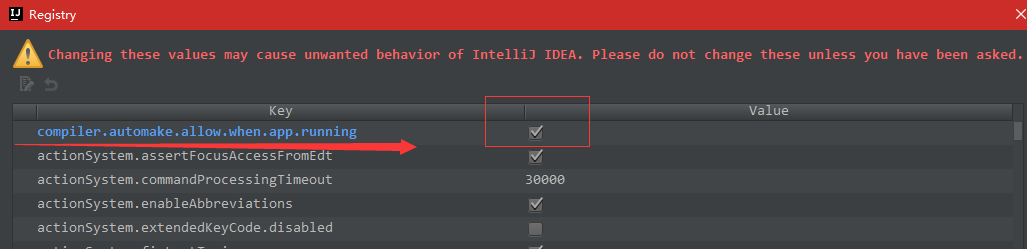
  1） 快捷键： Ctrl+Alt+S



  OK完之后

   2） 快捷键：Ctrl+Shift+Alt+ /





OK之后，再重启项目试试。 你可以发现，你修改java代码，项目会重新部署，修改html文件，刷新浏览器，效果也会修改的。完美实现 “热部署”

# Springboot整合mybatis

## 在pom.xml文件中引入相关依赖

|  |
| --- |
| *<!-- springboot,mybatis 整合包 -->* <**dependency**>  <**groupId**>org.mybatis.spring.boot</**groupId**>  <**artifactId**>mybatis-spring-boot-starter</**artifactId**>  <**version**>1.3.1</**version**> </**dependency**>  *<!-- mysql 驱动包 -->* <**dependency**>  <**groupId**>mysql</**groupId**>  <**artifactId**>mysql-connector-java</**artifactId**> </**dependency**> |

## 在application.properties文件中添加配置信息

|  |
| --- |
| **spring.datasource.driverClassName** = **com.mysql.jdbc.Driver spring.datasource.url** = **jdbc:mysql://localhost:3306/smbms?useUnicode=true&characterEncoding=utf-8 spring.datasource.username** = **root spring.datasource.password** = **accp** |

## 3. 使用PageHelper分页插件

|  |
| --- |
| *<!-- 分页插件-->* <**dependency**>  <**groupId**>com.github.pagehelper</**groupId**>  <**artifactId**>pagehelper-spring-boot-starter</**artifactId**>  <**version**>1.2.3</**version**> </**dependency**> |

|  |
| --- |
| @RequestMapping("/getByGradeNm2")  public List<Grade> getByGradeNm2(String name){  PageHelper.startPage(1,2); //显示第一页2条数据  return gradeService.getByGradeNm(name);  } |

|  |
| --- |
| *#pagehelper分页插件配置* **pagehelper.helperDialect**=**mysql pagehelper.reasonable**=**true pagehelper.supportMethodsArguments**=**true pagehelper.params**=**count=countSql** |

## 4. 配置 Druid 数据源

### 4.1 添加依赖

<!-- springboot,druid 整合包 -->

|  |
| --- |
| *<!-- springboot,druid 整合包 -->* <**dependency**>  <**groupId**>com.alibaba</**groupId**>  <**artifactId**>druid-spring-boot-starter</**artifactId**>  <**version**>1.1.9</**version**> </**dependency**> |

### 4.2 添加配置

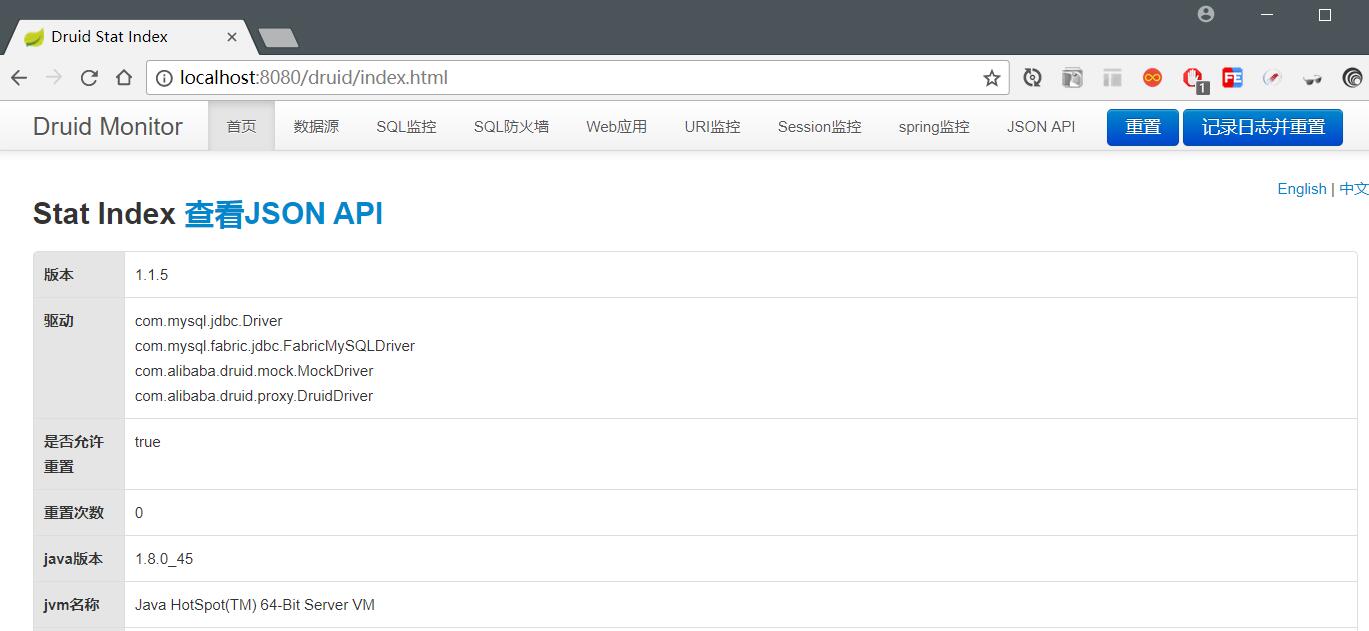
在 application.properties 中添加

|  |
| --- |
| **spring.datasource.driver-class-name**=**com.mysql.jdbc.Driver spring.datasource.url**=**jdbc:mysql://localhost:3306/smbms?useUnicode=true&characterEncoding=utf8&serverTimezone=UTC spring.datasource.username**=**root spring.datasource.password**=**accp** *# 修改数据源* **spring.datasource.type**=**com.alibaba.druid.pool.DruidDataSource spring.datasource.druid.initial-size**=**5  spring.datasource.druid.min-idle**=**5  spring.datasource.druid.max-active**=**20  spring.datasource.druid.max-wait**=**60000  spring.datasource.druid.time-between-eviction-runs-millis**=**60000  spring.datasource.druid.min-evictable-idle-time-millis**=**300000  spring.datasource.druid.validation-query**=**SELECT 1 FROM DUAL  spring.datasource.druid.test-while-idle**=**true  spring.datasource.druid.test-on-borrow**=**false  spring.datasource.druid.test-on-return**=**false  spring.datasource.druid.pool-prepared-statements**=**true  spring.datasource.druid.max-pool-prepared-statement-per-connection-size**=**20  spring.datasource.druid.filters**=**stat,wall** |

## 5、配置 Druid 监控

默认情况下，Druid 的监控统计功能和页面是开启的。

我们启动项目，访问 <http://localhost:8080/druid/index.html>，如下图：

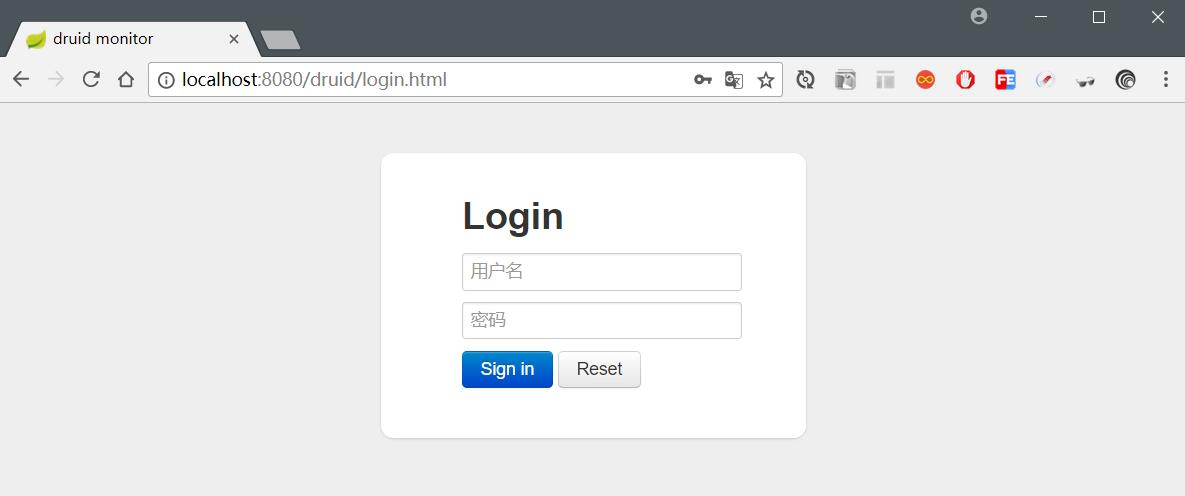


为了保证访问的安全性，我们可以如下配置：

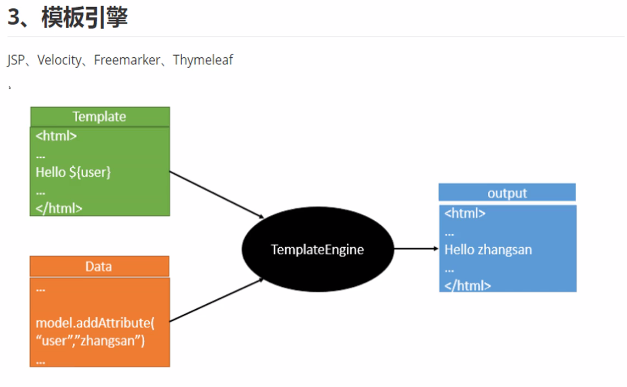
在 application.properties 中添加：

|  |
| --- |
| *## druid 监控* **spring.datasource.druid.web-stat-filter.enabled**=**true spring.datasource.druid.web-stat-filter.url-pattern**=**/\* spring.datasource.druid.web-stat-filter.exclusions**=**\*.js,\*.gif,\*.jpg,\*.png,\*.css,\*.ico,/druid/\*** *## druid 监控页面* **spring.datasource.druid.stat-view-servlet.enabled**=**true spring.datasource.druid.stat-view-servlet.url-pattern**=**/druid/\* spring.datasource.druid.stat-view-servlet.login-username**=**druid spring.datasource.druid.stat-view-servlet.login-password**=**druid123** |

重启项目，再次访问 <http://localhost:8080/druid/index.html> 地址时需要身份验证：



# 整合 Thymeleaf



## 3.1 添加 Thymeleaf 依赖

在pom.xml 文件中添加

|  |
| --- |
| <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-thymeleaf</**artifactId**> </**dependency**> |

## 3.2 添加 Thymeleaf 模板配置

在 application.properties 中添加如下内容：

|  |
| --- |
| spring.thymeleaf.cache=true  spring.thymeleaf.prefix=classpath:/templates/  spring.thymeleaf.suffix=.html  spring.thymeleaf.mode=HTML5  spring.thymeleaf.encoding=UTF-8  spring.thymeleaf.content-type=text/html |

|  |
| --- |
|  |

|  |
| --- |
|  |

上述配置都是默认值，所以也可以不添加。

## 3.3 Thymeleaf 案例演示

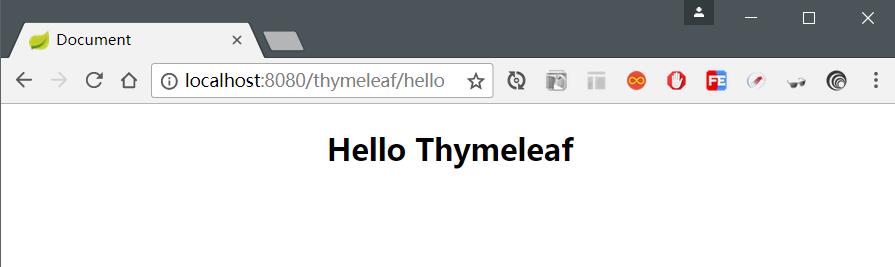
在 controller 包中创建 ThymeleafController:

|  |
| --- |
| @Controller  @RequestMapping("thymeleaf")  public class ThymeleafController {  @RequestMapping("hello")  public String hello(Map<String,Object> map) {  map.put("msg", "Hello Thymeleaf");  return "hello";  }  } |

在 template 目录下创建名为 hello.html 的文件

|  |
| --- |
| <!DOCTYPE html>  <**html lang="en" xmlns:th="http://www.thymeleaf.org"**>  <head>  <meta charset="UTF-8">  <title>Document</title>  <link href="/css/index.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <h2 th:text="${msg}"></h2>  </div>  </body>  </html> |

结果如下：

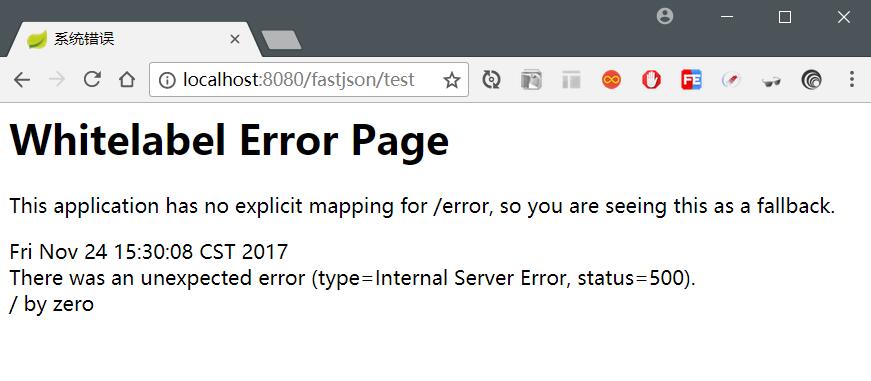
[](http://ow97db1io.bkt.clouddn.com/springboog-5-2.jpg)

## 3.4 错误处理

### 3.4.1 友好页面

先演示非友好页面

结果如下：

[](http://images.extlight.com/springboot-web-09-0.jpg)

当系统报错时，返回到页面的内容通常是一些杂乱的代码段，这种显示对用户来说不友好，因此我们需要自定义一个友好的提示系统异常的页面。在 src/main/resources 下创建 /public/error，在该目录下再创建一个名为 5xx.html 文件，该页面的内容就是当系统报错时返回给用户浏览的内容：

|  |
| --- |
| <!DOCTYPE html>  <html lang="zh">  <head>  <meta charset="UTF-8">  <title>系统错误</title>  <link href="/css/index.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <h2>系统内部错误</h2>  </div>  </body>  </html> |

路径时固定的，Spring Boot 会在系统报错时将返回视图指向该目录下的文件。

如下图：

[](http://images.extlight.com/springboot-web-09-1.jpg)

上边处理的 5xx 状态码的问题，接下来解决 404 状态码的问题。

当出现 404 的情况时，用户浏览的页面也不够友好，因此我们也需要自定义一个友好的页面给用户展示。

在 /public/error 目录下再创建一个名为 404.html 的文件：

|  |
| --- |
| <!DOCTYPE html>  <html lang="zh">  <head>  <meta charset="UTF-8">  <title>访问异常</title>  <link href="/css/index.css" rel="stylesheet"/>  </head>  <body>  <div class="container">  <h2>找不到页面</h2>  </div>  </body>  </html> |

我们请求一个不存在的资源，如：<http://localhost:8080/test2>，结果如下图：

[](http://images.extlight.com/springboot-web-09-2.jpg)

# 五．文件上传和下载

## 5.1上传

方案一：

|  |
| --- |
| *//处理文件上传* @RequestMapping(value=**"/testuploadimg"**, method = RequestMethod.***POST***) @ResponseBody **public** String upload(MultipartFile file) **throws** Exception {  **if**(file.isEmpty()){  **return "false"**;  }  String fileName = file.getOriginalFilename();  **int** size = (**int**) file.getSize();  System.***out***.println(fileName + **"-->"** + size);   String path = **"G:/testFile"** ;  File dest = **new** File(path + **"/"** + fileName);  **if**(!dest.getParentFile().exists()){ *//判断文件父目录是否存在* dest.getParentFile().mkdir();  }  **try** {  file.transferTo(dest); *//保存文件* **return "true"**;  } **catch** (IllegalStateException e) {  *//* ***TODO Auto-generated catch block*** e.printStackTrace();  **return "false"**;  } **catch** (IOException e) {  *//* ***TODO Auto-generated catch block*** e.printStackTrace();  **return "false"**;  } } |

方案二：

添加依赖

|  |
| --- |
| *<dependency>*  *<groupId>org.apache.commons</groupId>*  *<artifactId>commons-io</artifactId>*  *<version>1.3.2</version>*  *</dependency>* |

|  |
| --- |
| */\*\**  *\* 添加*  *\* @return*  *\* <input type="file" name="file" >*  *\*/*  *@RequestMapping("/add")*  *public String add(@RequestParam("file") MultipartFile file, HttpServletRequest request) {*  *//项目根目录*  *String path = request.getServletContext().getRealPath("/");*  *//原始文件名*  *String originalFilename = file.getOriginalFilename(); // config.zip*    *//处理文件名重复的问题*  *SimpleDateFormat sdf = new SimpleDateFormat("yyyyMMddHHmmssSSS");*  *//config-201710311607384-3.zip*  *String newFileName = FilenameUtils.getBaseName(originalFilename)*  *+ "-"*  *+ sdf.format(new Date())*  *+ "-"*  *+ new Random().nextInt(10)*  *+ "."*  *+ FilenameUtils.getExtension(originalFilename);*    *try {*  *file.transferTo(new File(path + newFileName));*  *} catch (Exception e) {*  *e.printStackTrace();*  *}*    *//b.setFileName(originalFilename);*  *//b.setFilePath(path + newFileName);*    *billService.add(b);*  *return "redirect:/xxx";*  *}* |

## 5.2 下载

方案一：

|  |
| --- |
| @RequestMapping(**"download"**) **public** String downLoad(HttpServletResponse response){  String filename=**"ACCP1.jpg"**;  String filePath = **"G:/testFile"** ;  File file = **new** File(filePath + **"/"** + filename);  **if**(file.exists()){ *//判断文件父目录是否存在* response.setContentType(**"application/force-download"**);  response.setHeader(**"Content-Disposition"**, **"attachment;fileName="** + filename);   **byte**[] buffer = **new byte**[1024];  FileInputStream fis = **null**; *//文件输入流* BufferedInputStream bis = **null**;   OutputStream os = **null**; *//输出流* **try** {  os = response.getOutputStream();  fis = **new** FileInputStream(file);  bis = **new** BufferedInputStream(fis);  **int** i = bis.read(buffer);  **while**(i != -1){  os.write(buffer);  i = bis.read(buffer);  }   } **catch** (Exception e) {  *//* ***TODO Auto-generated catch block*** e.printStackTrace();  }  System.***out***.println(**"----------file download"** + filename);  **try** {  bis.close();  fis.close();  } **catch** (IOException e) {  *//* ***TODO Auto-generated catch block*** e.printStackTrace();  }  }  **return null**; } |

方案二：

|  |
| --- |
| @RequestMapping("/download/{id}")  public void download(@PathVariable Long id, HttpServletResponse response) throws Exception {  Bill b = billService.get(id);    //设置下载的文件名  response.setHeader("Content-disposition",  "attachment; filename=" + new String(b.getFileName().getBytes(), "ISO8859-1")); //下载文件名    FileCopyUtils.copy(new FileInputStream(b.getFilePath()), response.getOutputStream());    } |

# 六．整合 JavaMail

本次测试演示带模板的邮件，使用 Freemark 实现邮件的模板。

## 6.1 添加依赖

|  |
| --- |
| *<!--.整合 JavaMail-->* <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-mail</**artifactId**> </**dependency**> <**dependency**>  <**groupId**>org.springframework.boot</**groupId**>  <**artifactId**>spring-boot-starter-freemarker</**artifactId**> </**dependency**> |

## 6.2 添加配置

在 application.properties 中添加

|  |
| --- |
| *# javamail 配置* **spring.mail.host**=**smtp.163.com spring.mail.username**=**it\_car@163.com spring.mail.password**=**gaas12c65 spring.mail.properties.mail.smtp.auth**=**true spring.mail.properties.mail.smtp.starttls.enable**=**true spring.mail.properties.mail.smtp.starttls.required**=**true** |

## 6.3 编码

|  |
| --- |
| **package** com.example.demo.service.impl;  **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.boot.autoconfigure.mail.MailProperties; **import** org.springframework.boot.context.properties.EnableConfigurationProperties; **import** org.springframework.mail.javamail.JavaMailSender; **import** org.springframework.mail.javamail.MimeMessageHelper; **import** org.springframework.stereotype.Component; **import** org.springframework.ui.freemarker.FreeMarkerTemplateUtils; **import** org.springframework.web.servlet.view.freemarker.FreeMarkerConfigurer; **import** java.util.HashMap; **import** java.util.Map;  **import** javax.mail.MessagingException; **import** javax.mail.internet.InternetAddress; **import** javax.mail.internet.MimeMessage; **import** java.io.UnsupportedEncodingException;   @Component @EnableConfigurationProperties(MailProperties.**class**) **public class** JavaMailComponent {   **private static final** String ***template*** = **"mail.ftl"**;  @Autowired  **private** FreeMarkerConfigurer **freeMarkerConfigurer**;   @Autowired  **private** JavaMailSender **javaMailSender**;  @Autowired  **private** MailProperties **mailProperties**;  **public void** sendMail(String email) {  Map<String, Object> map = **new** HashMap<String, Object>();  map.put(**"email"**, email);  **try** {  *// 获取内容* String text = **this**.getTextByTemplate(***template***, map);  *// 发送* **this**.send(email, text);  } **catch** (Exception e) {  e.printStackTrace();  }  }  **private** String getTextByTemplate(String template, Map<String, Object> model) **throws** Exception {  **return** FreeMarkerTemplateUtils  .*processTemplateIntoString*(**this**.**freeMarkerConfigurer**.getConfiguration().getTemplate(template), model);  }  **private** String send(String email, String text) **throws** MessagingException, UnsupportedEncodingException {  MimeMessage message = **this**.**javaMailSender**.createMimeMessage();  MimeMessageHelper helper = **new** MimeMessageHelper(message, **true**, **"UTF-8"**);  InternetAddress from = **new** InternetAddress();  from.setAddress(**this**.**mailProperties**.getUsername());  from.setPersonal(**"月光中的污点"**, **"UTF-8"**);  helper.setFrom(from);  helper.setTo(email);  helper.setSubject(**"SpringBoot 发送的第一封邮件"**);  helper.setText(text, **true**);  **this**.**javaMailSender**.send(message);  **return** text;  } } |

## 6.4 创建mail.ftl模板文件

在 src/main/resources 下的 template 目录下创建名为 mail.ftl 的文件，其内容如下

|  |
| --- |
| <!DOCTYPE **html**> <**html lang="zh"**> <**head**>  <**meta http-equiv="Content-Type" content="text/html; charset=UTF-8"** /> </**head**> <**body**> <**div style="width**: 600**px**; **text-align**: **left**; **margin**: 0 **auto**;**"**>  <**h1 style="color**: **#005da7**;**"**>月光中的污点</**h1**>  <**div style="border-bottom**: 5**px solid #005da7**; **height**: 2**px**; **width**: 100%;**"**></**div**>  <**div style="border**: 1**px solid #005da7**; **font-size**: 16**px**; **line-height**: 50**px**; **padding**: 20**px**;**"**>  <**div**>**${email}**，您好！</**div**>  <**div**>  这是个测试  </**div**>  <**div**>  想了解更多信息，请访问 <**a href="http://www.baidu.com"**>  <http://www.baidu.com>  </**a**>  </**div**>  </**div**> </**div**> </**body**> </**html**> |

## 6.5 测试

|  |
| --- |
| **package** com.example.demo;  **import** com.example.demo.service.impl.JavaMailComponent; **import** org.junit.Test; **import** org.junit.runner.RunWith; **import** org.springframework.beans.factory.annotation.Autowired; **import** org.springframework.boot.test.context.SpringBootTest; **import** org.springframework.test.context.junit4.SpringRunner;  @RunWith(SpringRunner.**class**) @SpringBootTest **public class** MailTest {  @Autowired  **private** JavaMailComponent **javaMailComponent**;   @Test  **public void** test() {  **this**.**javaMailComponent**.sendMail(**"329386467@qq.com"**);  } } |

# 七．整合 shiro实现用户认证

## 7.1 shiro介绍

Shiro是Apache下的一个开源项目，我们称之为Apache Shiro。它是一个很易用与Java项目的的安全框架，提供了认证、授权、加密、会话管理，与spring Security 一样都是做一个权限的安全框架，但是与Spring Security 相比，在于 Shiro 使用了比较简单易懂易于使用的授权方式。shiro属于轻量级框架，相对于security简单的多，也没有security那么复杂。

Shiro能做什么呢？

* 验证用户身份
* 用户访问权限控制，比如：1、判断用户是否分配了一定的安全角色。2、判断用户是否被授予完成某个操作的权限
* 在非 web 或 EJB 容器的环境下可以任意使用Session API
* 可以响应认证、访问控制，或者 Session 生命周期中发生的事件
* 可将一个或以上用户安全数据源数据组合成一个复合的用户 “view”(视图)
* 支持单点登录(SSO)功能
* 支持提供“Remember Me”服务，获取用户关联信息而无需登录

Authentication（认证）, Authorization（授权）, Session Management（会话管理）, Cryptography（加密）被 Shiro 框架的开发团队称之为应用安全的四大基石。那么就让我们来看看它们吧：

* Authentication（认证）：用户身份识别，通常被称为用户“登录”
* Authorization（授权）：访问控制。比如某个用户是否具有某个操作的使用权限。
* Session Management（会话管理）：特定于用户的会话管理,甚至在非web 或 EJB 应用程序。
* Cryptography（加密）：在对数据源使用加密算法加密的同时，保证易于使用。

## 7.2快速上手

### pom包依赖

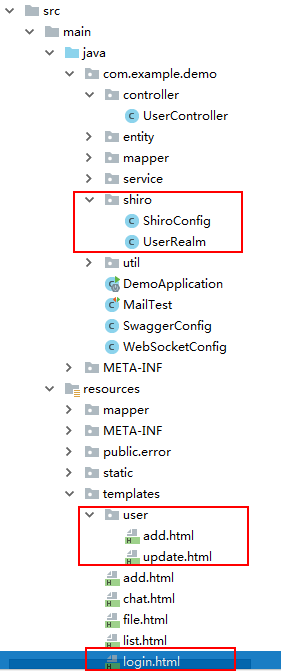
|  |
| --- |
| *<!--springboot整合shiro-->* <**dependency**>  <**groupId**>org.apache.shiro</**groupId**>  <**artifactId**>shiro-spring</**artifactId**>  <**version**>1.4.0</**version**> </**dependency**> |

### 核心api

Subject :用户主体（把操作交给SecurityManager）

SecurityManager:安全管理器（关联Realm）

Realm : Shiro连接数据库的桥梁



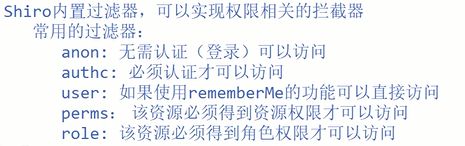
### 自定义Realm类

|  |
| --- |
| **package** com.example.demo.shiro;  **import** org.apache.shiro.authc.AuthenticationException; **import** org.apache.shiro.authc.AuthenticationInfo; **import** org.apache.shiro.authc.AuthenticationToken; **import** org.apache.shiro.authz.AuthorizationInfo; **import** org.apache.shiro.realm.AuthorizingRealm; **import** org.apache.shiro.subject.PrincipalCollection;  */\*\*  \* 自定义Realm类  \*/* **public class** UserRealm **extends** AuthorizingRealm {   */\*\*  \* 执行授权逻辑  \** ***@return*** *\*/* @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  System.***out***.println(**" 执行授权逻辑"**);  **return null**;  }  */\*\*  \* 执行认证逻辑   \** ***@return*** *\*/* @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) **throws** AuthenticationException {  System.***out***.println(**" 执行认证逻辑"**);  **return null**;  } } |

### 编写Shiro配置类（\*）

|  |
| --- |
| **package** com.example.demo.shiro;  **import** org.apache.shiro.spring.web.ShiroFilterFactoryBean; **import** org.apache.shiro.web.mgt.DefaultWebSecurityManager; **import** org.springframework.beans.factory.annotation.Qualifier; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  */\*  配置类  \*/* @Configuration **public class** ShiroConfig {  *//创建ShiroFilterFactoryBean*  @Bean**public** ShiroFilterFactoryBean getShiroFilterFactoryBean(@Qualifier(**"securityManager"**) DefaultWebSecurityManager securityManager){  ShiroFilterFactoryBean shiroFilterFactoryBean = **new** ShiroFilterFactoryBean();  *//设置安全管理器* shiroFilterFactoryBean.setSecurityManager(securityManager);  **return** shiroFilterFactoryBean;  }  *//创建DefaultWebSecurityManager* @Bean(name=**"securityManager"**)  **public** DefaultWebSecurityManager getDefaultWebSecurityManager(@Qualifier(**"userRealm"**) UserRealm userRealm){  DefaultWebSecurityManager securityManager = **new** DefaultWebSecurityManager();  *//关联realm* securityManager.setRealm(userRealm);  **return** securityManager;  }  *//创建Realm* @Bean(name = **"userRealm"**)  **public** UserRealm getRealm()  {  **return new** UserRealm();  } } |

### 添加Shiro内置过滤器



|  |
| --- |
| *//创建ShiroFilterFactoryBean* @Bean **public** ShiroFilterFactoryBean getShiroFilterFactoryBean(@Qualifier(**"securityManager"**) DefaultWebSecurityManager securityManager){  ShiroFilterFactoryBean shiroFilterFactoryBean = **new** ShiroFilterFactoryBean();  *//设置安全管理器* shiroFilterFactoryBean.setSecurityManager(securityManager);  *//添加Shiro内置过滤器* Map<String,String> filterMap = **new** LinkedHashMap<String,String>();  filterMap.put(**"/adds"**,**"authc"**);  filterMap.put(**"/updates"**,**"authc"**);  shiroFilterFactoryBean.setFilterChainDefinitionMap(filterMap);  *//修改调整登录页面* shiroFilterFactoryBean.setLoginUrl(**"/toLogin"**);  **return** shiroFilterFactoryBean; } |

通配符方式

|  |
| --- |
| filterMap.put(**"/test"**,**"anon"**); filterMap.put(**"/\*"**,**"authc"**); |

### 7.3实现用户认证【登录】操作

创建表单

|  |
| --- |
| * <**h3**>用户登录</**h3**> <**form action="login" method="post"**>  name:<**input type="text" name="name"**/><**br**/>  pass:<**input type="password" name="password"**/><**br**/>  <**input type="submit" value="用户登录"**/> </**form**> |

UserController

|  |
| --- |
| @RequestMapping(value=**"/login"**) **public** String login(String name,String password,Model model) {  */\*\*  \* 使用Shiro编写认证操作  \*/  //1.获取Subject对象* Subject subject = SecurityUtils.*getSubject*();  *//2.封装用户数据* UsernamePasswordToken token = **new** UsernamePasswordToken(name,password);  *//3.执行登录方法* **try** {  subject.login(token);*//无异常代表登录成功* **return "redirect:/test"**;  } **catch** (UnknownAccountException e) {  *//登录失败：账户名不存在* model.addAttribute(**"msg"**,**"用户名不存在"**);  **return "login"**;  } **catch** (IncorrectCredentialsException e) {  *//登录失败：密码错误* model.addAttribute(**"msg"**,**"密码错误"**);  **return "login"**;  }  } |

编写userRealm的认证逻辑

|  |
| --- |
| **package** com.example.demo.shiro; **import** org.apache.shiro.authc.\*; **import** org.apache.shiro.authz.AuthorizationInfo; **import** org.apache.shiro.realm.AuthorizingRealm; **import** org.apache.shiro.subject.PrincipalCollection; */\*\*  \* 自定义Realm类  \*/* **public class** UserRealm **extends** AuthorizingRealm {   */\*\*  \* 执行授权逻辑  \** ***@return*** *\*/* @Override  **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  System.***out***.println(**" 执行授权逻辑"**);  **return null**;  }  */\*\*  \* 执行认证逻辑  \** ***@return*** *\*/* @Override  **protected** AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken authenticationToken) **throws** AuthenticationException {  System.***out***.println(**" 执行认证逻辑"**);  *//假如数据库的用户名和密码分别为 accp 123456  //编写Shiro判断逻辑，用户名和密码的判断  //1.判断用户名* UsernamePasswordToken token = (UsernamePasswordToken)authenticationToken;  **if**(!token.getUsername().equals(**"accp"**))  {  *//用户名不存在 Shiro底层会抛出UnknownAccountException异常* **return null**;  }  *//判断密码* **return new** SimpleAuthenticationInfo(**""**,**"123456"**,**""**);  } } |

### 数据库操作

|  |
| --- |
| User user = **this**.**userService**.getByName(token.getUsername()); **if**(user==**null**) {  *//用户名不存在 Shiro底层会抛出UnknownAccountException异常* **return null**; } *//判断密码* **return new** SimpleAuthenticationInfo(**""**,user.getPassword(),**""**); |

### 7.4实现用户授权操作完成页面拦截

在ShiroConfig类中进行设置

|  |
| --- |
| **package** com.example.demo.shiro;  **import** org.apache.shiro.spring.web.ShiroFilterFactoryBean; **import** org.apache.shiro.web.mgt.DefaultWebSecurityManager; **import** org.springframework.beans.factory.annotation.Qualifier; **import** org.springframework.context.annotation.Bean; **import** org.springframework.context.annotation.Configuration;  **import** java.util.LinkedHashMap; **import** java.util.Map;  */\*  配置类  \*/* @Configuration **public class** ShiroConfig {  *//创建ShiroFilterFactoryBean* @Bean  **public** ShiroFilterFactoryBean getShiroFilterFactoryBean(@Qualifier(**"securityManager"**) DefaultWebSecurityManager securityManager){  ShiroFilterFactoryBean shiroFilterFactoryBean = **new** ShiroFilterFactoryBean();  *//设置安全管理器* shiroFilterFactoryBean.setSecurityManager(securityManager);  *//添加Shiro内置过滤器* Map<String,String> filterMap = **new** LinkedHashMap<String,String>();  */\*filterMap.put("/adds","authc");  filterMap.put("/updates","authc");\*/* filterMap.put(**"/test"**,**"anon"**);  filterMap.put(**"/login"**,**"anon"**);   *//授权过滤器  //拦截后会自云烟跳转到未权页面* filterMap.put(**"/adds"**,**"perms[user:add]"**);   filterMap.put(**"/\*"**,**"authc"**);  shiroFilterFactoryBean.setFilterChainDefinitionMap(filterMap);  *//修改调整登录页面* shiroFilterFactoryBean.setLoginUrl(**"/toLogin"**);   *//设置未授权提示页面* shiroFilterFactoryBean.setUnauthorizedUrl(**"/noAuth"**);   **return** shiroFilterFactoryBean;  }  *//创建DefaultWebSecurityManager* @Bean(name=**"securityManager"**)  **public** DefaultWebSecurityManager getDefaultWebSecurityManager(@Qualifier(**"userRealm"**) UserRealm userRealm){  DefaultWebSecurityManager securityManager = **new** DefaultWebSecurityManager();  *//关联realm* securityManager.setRealm(userRealm);  **return** securityManager;  }  *//创建Realm* @Bean(name = **"userRealm"**)  **public** UserRealm getRealm()  {  **return new** UserRealm();  } } |

在Controller中设置未授权页面跳转

|  |
| --- |
| *//未授权提示* @RequestMapping(value = **"/noAuth"**) **public** String noAuth() {  **return "noAuth"**; } |

### 编写资源授权逻辑

UserReam中编写

|  |
| --- |
| */\*\*  \* 执行授权逻辑  \** ***@return*** *\*/* @Override **protected** AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principalCollection) {  System.***out***.println(**" 执行授权逻辑"**);  *//给资源进行授权* SimpleAuthorizationInfo info = **new** SimpleAuthorizationInfo();  *//添加资源授权字符串* info.addStringPermission(**"user:add"**);  **return** info; } |