SpringBoot-基础使用

1. SpringBoot工程创建

JDK : 8

Maven : 3.5.x

1.1、Maven配置settings.xml

```
<mirrors>
    <mirror>
        <id>aliyunmaven</id>
        <mirrorOf>central</mirrorOf>
        <name>aliyun maven</name>

 <url>https://maven.aliyun.com/repository/public
</url>
    </mirror>
</mirrors>
```

```
<profiles>
    <profile>
        <id>jdk-1.8</id>
        <activation>
            <activeByDefault>true</activeByDefault>
            <jdk>1.8</jdk>
        </activation>
        <properties>

 <maven.compiler.source>1.8</maven.compiler.source>

 <maven.compiler.target>1.8</maven.compiler.target>
```

```
 <maven.compiler.compilerVersion>1.8</maven.compiler
.compilerVersion>
        </properties>
    </profile>
</profiles>
```

1.2、清理Maven仓库脚本

```
@echo off
rem create by NettQun

rem 这里写你的仓库路径
set
REPOSITORY_PATH=D:\programmingSoftware\maven\local_r
epository
rem 正在搜索...
for /f "delims=" %%i in ('dir /b /s
"%REPOSITORY_PATH%\*lastUpdated*"') do (
    echo %%i
    del /s /q "%%i"
)
rem 搜索完毕
pause
```

创建一个bat文件，然后复制上述脚本进去，修改其中maven本地仓库的地址，保存后双击执行即可。

1.3 创建流程

> 继承父工程

在**pom.xml**中添加一下配置，继承spring-boot-starter-parent这个父工程

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-
parent</artifactId>
    <version>2.5.0</version>
</parent>
```

添加依赖

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

创建启动类

创建一个类在其实加上@SpringBootApplication注解标识为启动类。

```
@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) {

 SpringApplication.run(HelloApplication.class,
args);
    }
}
```

定义Controller

创建Controller,主要Controller要放在启动类所在包或者其子包下。

```java
@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String hello(){
        return "hello";
    }
}
```

> 运行测试

直接**运行启动类的main方法**即可。`localhost:8080/hello`

1.4 常见问题及解决方案

> 访问时404

检查Controller是不是在启动类所在的包或者其子包下，如果不是需要进行修改。

> 依赖爆红

配置阿里云镜像后刷新maven项目让其下载。

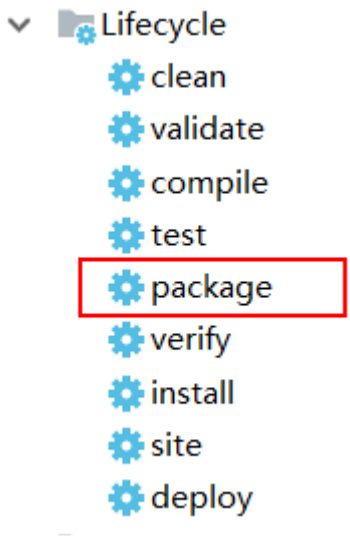
1.5 打包运行

我们可以把springboot的项目打成jar包直接去运行。

> 添加maven插件 **`pom.xml`**

```
<build>
    <plugins>
        <!--springboot打包插件-->
        <plugin>

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

> maven打包



> 运行jar包

在jar包所在目录执行命令,即可运行。mybatis_plus_test-1.0-SNAPSHOT.jar

```
java -jar mybatis_plus_test-1.0-SNAPSHOT.jar
```

2.起步依赖

SpringBoot依靠父项目中的版本锁定和starter机制让我们能更轻松的实现对依赖的管理。

2.0 依赖冲突及其解决方案

2.0.1 依赖冲突

一般程序在运行时发生类似于 java.lang.ClassNotFoundException, Method not found: '……'，或者莫名其妙的异常信息，这种情况一般很大可能就是 jar包依赖冲突的问题引起的了。

一般在是A依赖C(低版本)，B也依赖C(高版本)。 都是他们依赖的又是不同版本的C的时候会出现。

2.0.2 解决方案

如果出现了类似于 java.lang.ClassNotFoundException, Method not found: 这些异常检查相关的依赖冲突问题，排除掉低版本的依赖，留下高版本的依赖。

2.1 版本锁定

我们的SpringBoot模块都需要继承一个父工程：**spring-boot-starter-parent。在spring-boot-starter-parent的父工程spring-boot-dependencies中对常用的依赖进行了版本锁定。**这样我们在添加依赖时，很多时候都不需要添加依赖的版本号了。

我们也可以采用覆盖properties配置或者直接指定版本号的方式修改依赖的版本。

例如：直接指定版本号pom.xml

```
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjweaver</artifactId>
    <version>1.7.2</version>
</dependency>
```

覆盖properties配置pom.xml

```
<properties>
    <aspectj.version>1.7.2</aspectj.version>
</properties>
```

2.2 starter机制

当我们需要使用某种功能时只需要引入对应的starter即可。一个starter针对一种特定的场景，其内部引入了该场景所需的依赖。这样我们就不需要单独引入多个依赖了。

命名规律

- 官方starter都是以 `spring-boot-starter`开头后面跟上场景名称。例如：spring-boot-starter-data-jpa
- 非官方starter则是以 `场景名-spring-boot-starter`的格式，例如：mybatis-spring-boot-starter

3.YML配置

3.1.简介

YAML (YAML Ain't a Markup Language)YAML不是一种标记语言，**通常以.yml为后缀的文件**，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，一种专门用来写配置文件的语言。

YAML试图用一种比XML更敏捷的方式，来完成XML所完成的任务。

例如：

```yaml
student:
    name: sangeng
    age: 15
```

```xml
<student>
    <name>sangeng</name>
    <age>15</age>
</student>
```

3.2.语法

3.2.1.约定

- k: v 表示键值对关系，**冒号后面必须有一个空格**
- 使用空格的缩进表示层级关系，空格数目不重要，**只要是左对齐的一列数据，都是同一个层级的**
- 大小写敏感
- 缩进时**不允许使用Tab键，只允许使用空格**。
- java中对于驼峰命名法，**可用原名或使用-代替驼峰**，如java中的lastName属性,**在yml中使用lastName或 last-name都可正确映射,用横杠分隔开。**

```yaml
user:
  lastName: Smith
```

```yaml
user:
  last-name: Smith
```

3.2.2.键值关系

> 普通值(字面量)

k: v：字面量直接写；

字符串默认不用加上单引号或者双绰号；

"": 双引号；转意字符能够起作用

name: "sangeng \n caotang": 输出；sangeng 换行 caotang

'': 单引号；不会解析转义字符。转义字符在单引号中会被视为普通文本字符。

```
name1: sangeng
name2: 'sangeng  \n caotang'  # 解析为 sangeng (新行)
caotang
name3: "sangeng  \n caotang"  # 解析为 sangeng \n
caotang
age: 15
flag: true
```

> 日期

```
date: 2019/01/01
```

> 对象(属性和值)、Map(键值对)

多行写法：在下一行来写对象的属性和值的关系，**注意缩进**

```
student:
  name: zhangsan
  age: 20
```

行内写法：

```
student: {name: zhangsan,age: 20}
```

> **数组、list、set**

用- 值表示数组中的一个元素

多行写法：

```yaml
pets:
  - dog
  - pig
  - cat
```

行内写法：

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

对象数组、对象list、对象set

```yaml
students:
 - name: zhangsan
   age: 22
 - name: lisi
   age: 20
 - {name: wangwu,age: 18}
```

3.2.3 占位符赋值

可以使用 **${key:defaultValue}** 的方式来赋值，若key不存在，则会使用defaultValue来赋值。

例如

```yaml
server:
  port: ${myPort:88}
myPort: 80
```

3.3.SpringBoot读取YML

3.3.1 @Value注解

注意使用此注解**只能获取简单类型的值**（8种基本数据类型及其包装类，String,Date）

```yaml
student:
  lastName: hq
```

```java
@RestController
public class TestController {
    @Value("${student.lastName}")
    private String lastName;
    @RequestMapping("/test")
    public String test(){
        System.out.println(lastName);
        return "hi";
    }

}
```

注意：加了@Value的类必须是交由Spring容器管理的，例如@Controller,@Service

3.3.2 @ConfigurationProperties 更常用

> yml配置

```yaml
student:
  lastName: hq
  age: 20
student1:
  lastName: hq1
  age: 17
student2:
  lastName: hq2
  age: 15
```

> 在类上添加注解@Component 和
> @ConfigurationProperties(prefix = "配置前缀")

读取student的值，前缀就是等值匹配

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Component
@ConfigurationProperties(prefix = "student")
public class Student {
    private String lastName;
    private Integer age;
}
```

> 从spring容器中获取Student对象

```
@RestController
public class TestController {

    @Autowired
    private Student student;
    @RequestMapping("/test")
    public String test(){
        System.out.println(student);
        return "hi";
    }
}
```

注意事项：要求对应的属性要有set/get方法，并且key要和成员变量名一致才可以对应的上。

3.4 YML和properties配置的相互转换

我们可以使用一些网站非常方便的实现YML和properties格式配置的相互转换。

转换网站：https://www.toyaml.com/index.html

3.5 属性配置dependency导入

如果使用了@ConfigurationProperties注解，可以增加以下依赖，让我们在书写配置时有相应的提示。

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-configuration-
processor</artifactId>
    <optional>true</optional>
</dependency>
```

注意：添加完依赖加完注解后要运行一次程序才会有相应的提示。

SpringBoot-常见场景

1.热部署（不太常用）

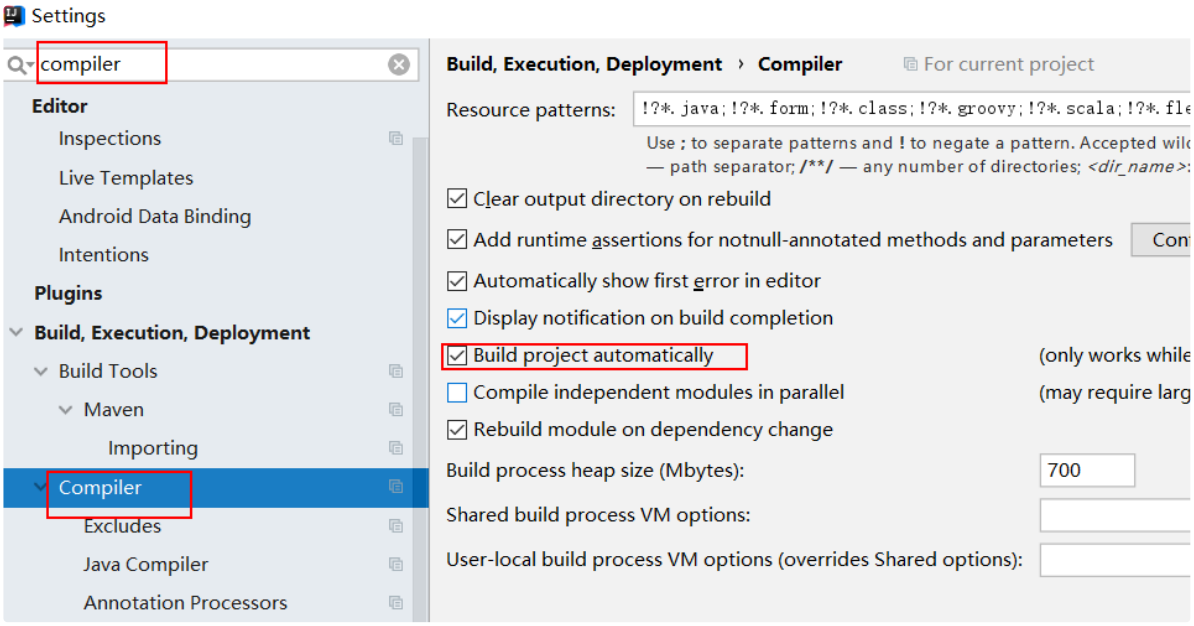
SpringBoot为我们提供了一个方便我们开发测试的工具dev-tools。使用后可以实现热部署的效果。当我们运行了程序后对程序进行了修改，程序会自动重启。

原理是使用了两个ClassLoder,一个ClassLoader加载哪些不会改变的类(第三方jar包),另一个ClassLoader加载会更改的类.称之为Restart ClassLoader,这样在有代码更改的时候,原来的Restart Classloader被丢弃,重新创建一个Restart ClassLoader,由于需要加载的类相比较少,所以实现了较快的重启。

1.1 准备工作

设置IDEA自动编译

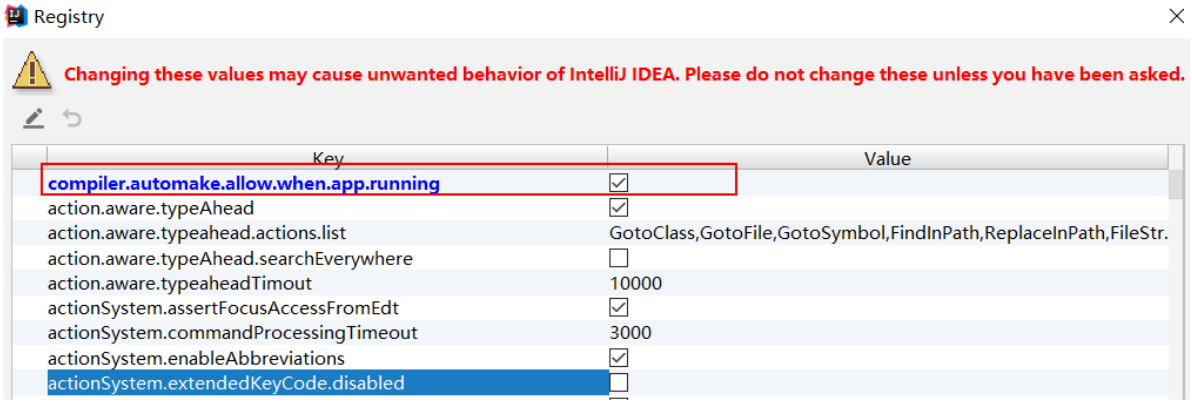
在idea中的setting做下面配置



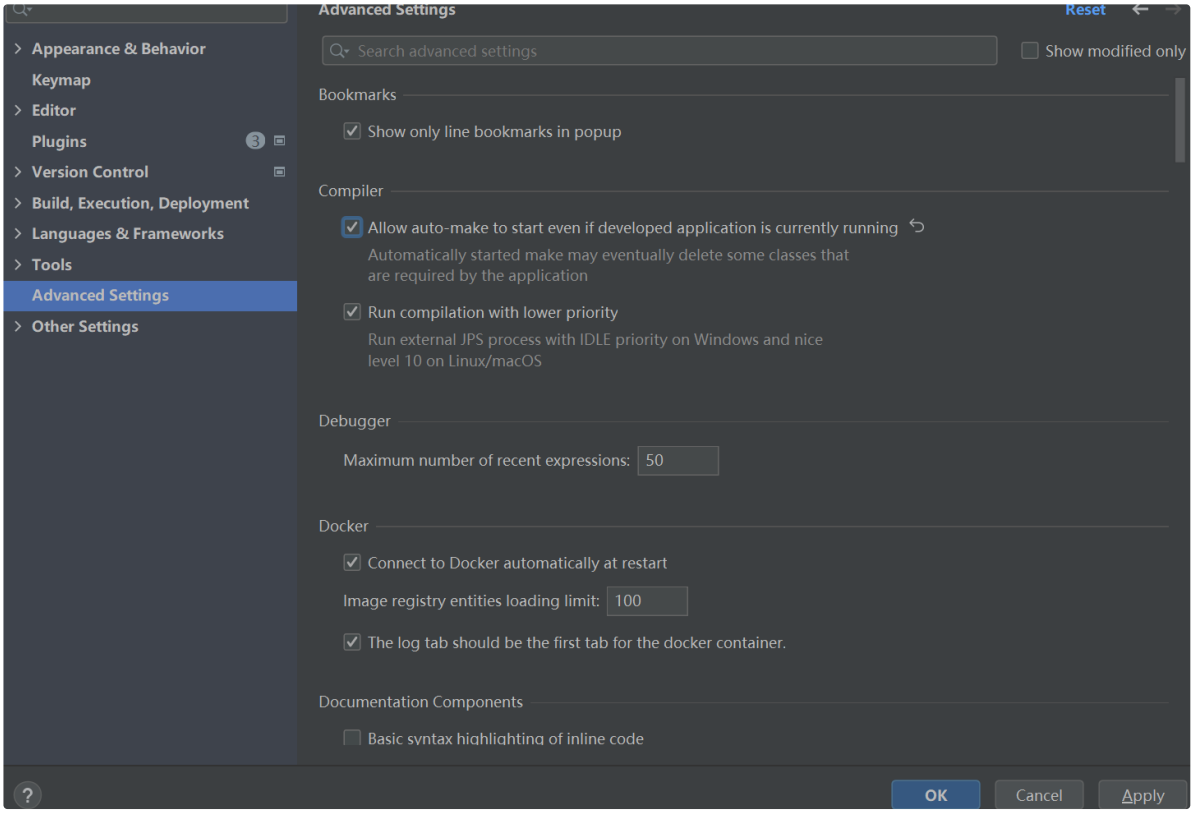
设置允许程序运行时自动启动

关闭刚才的页面，ctrl + shift + alt + / 这组快捷键后会有一个小弹窗，点击Registry 就会进入下面的界面，找到下面的配置项并勾选，勾选后直接点close

2021版本



2022版本



1.2使用

> 添加依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

> 触发热部署

当我们在修改完代码或者静态资源后可以切换到其它软件，让IDEA自动进行编译，自动编译后就会触发热部署。或者使用Ctrl+F9手动触发重新编译。

2.Junit进行单元测试

我们可以使用SpringBoot整合Junit进行单元测试。

Spring Boot 2.2.0 版本开始引入 JUnit 5 作为单元测试默认库。

Junit5功能相比于Junit4也会更强大。但是本课程是SpringBoot的课程，所以主要针对SpringBoot如何整合Junit进行单元测试做讲解。暂不针对Junit5的新功能做介绍。如有需要会针对Junit5录制专门的课程进行讲解。

2.1 使用

> 添加依赖

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

> 编写测试类

```
import com.sangeng.controller.HelloController;
import org.junit.jupiter.api.Test;
import
org.springframework.beans.factory.annotation.Autowir
ed;
import
org.springframework.boot.test.context.SpringBootTest
;

@SpringBootTest
public class ApplicationTest {

    @Autowired
```

```
    private HelloController helloController;

    @Test
    public void testJunit(){
        System.out.println(1);
        System.out.println(helloController);
    }
}
```

> **注意：测试类所在的包需要和启动类是在同一个包下。否则就要使用如下写法指定启动类。**

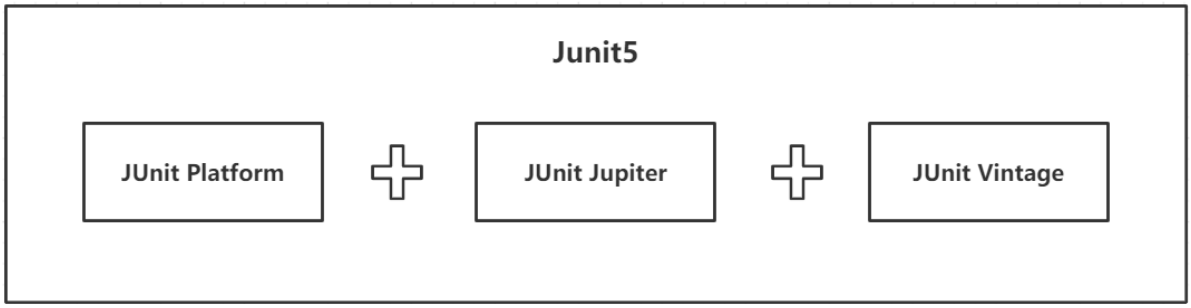
```
import com.sangeng.controller.HelloController;
import org.junit.jupiter.api.Test;
import
org.springframework.beans.factory.annotation.Autowir
ed;
import
org.springframework.boot.test.context.SpringBootTest
;

//classes属性来指定启动类
@SpringBootTest(classes = HelloApplication.class)
public class ApplicationTest {

    @Autowired
    private HelloController helloController;

    @Test
    public void testJunit(){
        System.out.println(1);
        System.out.println(helloController);
    }
}
```

2.2 兼容老版本

如果是对老项目中的SpringBoot进行了**版本升级**会发现之前的**单元测试代码出现了一些问题**。

因为Junit5和之前的Junit4有比较大的不同。

先看一张图：



从上图可以看出 **JUnit 5 = JUnit Platform + JUnit Jupiter + JUnit Vintage**

- **JUnit Platform**： 这是Junit提供的平台功能模块，通过它，其它的测试引擎也可以接入
- **JUnit JUpiter**：这是JUnit5的核心，是一个基于JUnit Platform的引擎实现，它包含许多丰富的新特性来使得自动化测试更加方便和强大。
- **JUnit Vintage**：这个模块是兼容JUnit3、JUnit4版本的测试引擎，使得旧版本的自动化测试也可以在JUnit5下正常运行。

虽然Junit5包含了**JUnit Vintage**来兼容JUnit3和Junit4，但是**SpringBoot 2.4 以上版本对应的spring-boot-starter-test移除了默认对 Vintage 的依赖。**所以当我们仅仅依赖spring-boot-starter-test时会发现之前我们使用的@Test注解和@RunWith注解都不能使用了。**我们可以单独在依赖vintage来进行兼容。**

```
<dependency>
    <groupId>org.junit.vintage</groupId>
    <artifactId>junit-vintage-engine</artifactId>
    <scope>test</scope>
</dependency>
```

注意：org.junit.Test对应的是Junit4的版本，就搭配@RunWith注解来使用。

SpringBoot2.2.0之前版本的写法。 **import org.junit.Test 导入的Test类不同， JUnit5是import org.junit.jupiter.api.Test;**

```
import com.sangeng.controller.HelloController;
import org.junit.Test;
import org.junit.runner.RunWith;
import
org.springframework.beans.factory.annotation.Autowir
ed;
import
org.springframework.boot.test.context.SpringBootTest
;
import
org.springframework.test.context.junit4.SpringRunner
;

//classes属性来指定启动类
@SpringBootTest
@RunWith(SpringRunner.class)
public class ApplicationTest {

    @Autowired
    private HelloController helloController;

    @Test
    public void testJunit(){
        System.out.println(1);
        System.out.println(helloController);
    }
}
```

3.整合mybatis

3.1 准备工作

> 数据准备

```sql
/*Table structure for table `user` */
DROP TABLE IF EXISTS `user`;
CREATE TABLE `user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT NULL,
  `age` int(11) DEFAULT NULL,
  `address` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=10 DEFAULT
CHARSET=utf8;

/*Data for the table `user` */

insert  into `user`(`id`,`username`,`age`,`address`)
values (2,'pdd',25,'上海'),(3,'UZI',19,'上海11'),
(4,'RF',19,NULL),(6,'三更',14,'请问2'),
(8,'test1',11,'cc'),(9,'test2',12,'cc2');

/*!40101 SET SQL_MODE=@OLD_SQL_MODE */;
/*!40014 SET
FOREIGN_KEY_CHECKS=@OLD_FOREIGN_KEY_CHECKS */;
/*!40014 SET UNIQUE_CHECKS=@OLD_UNIQUE_CHECKS */;
/*!40111 SET SQL_NOTES=@OLD_SQL_NOTES */;
```

> 实体类(建立在entity下)

```
import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private Integer id;
    private String username;
    private Integer age;
    private String address;
}
```

3.2 整合步骤

github: https://github.com/mybatis/spring-boot-starter/

> 依赖

```
<!--mybatis启动器-->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-
starter</artifactId>
    <version>2.2.0</version>
</dependency>
<!--mysql驱动-->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

> 配置数据库信息

```
spring:
    datasource:
        driver-class-name: com.mysql.cj.jdbc.Driver
        password: 302305
#         url:
jdbc:mysql://localhost:3306/mybatis_plus_db?
characterEncoding=utf-8&serverTimezone=UTC
        url:
jdbc:mysql://localhost:3306/springboot_test?
characterEncoding=utf-8&serverTimezone=UTC
        username: root
```

> 配置mybatis相关配置

classpath代表类加载路径，如resource;*.xml代表以文件以.xml结尾

type-aliases-package存放实体类位置。 默认情况下，MyBatis 会使用实体类的非限定类名来作为它的别名，如将 com.example.entity.User 的别名设置为 User 或 user

```
mybatis:
  mapper-locations: classpath*:/mapper/**/*.xml
  type-aliases-package: com.hq.entity    # 配置哪个包下的类有默认的别名
```

> 编写Mapper接口

注意在接口上加上@Mapper 和@Repository 注解

@Repository可以防止userMapper出现莫名爆红

```java
@Repository
@Mapper
public interface UserMapper {
    public List<User> findAll();
}
```

编写mapper接口对应的xml文件

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper
3.0//EN" "http://mybatis.org/dtd/mybatis-3-
mapper.dtd" >
<mapper namespace="com.hq.bootMapper.UserMapper">
    <select id="findAll"
resultType="com.hq.entity.User">
        select * from user
    </select>
</mapper>
```

测试

```java
@SpringBootTest(classes = HelloApplication.class)
public class SpringMyTest {

    @Autowired
    UserMapper userMapper;


    @Test
    public void tesMapper(){
        System.out.println(userMapper.findAll());
    }
}
```

4.Web开发

4.1 静态资源访问

从SpringBoot官方文档中我们可以知道，我们可以把静态资源放到 `resources/static` （或者 `resources/public` 或者 `resources/resources` 或者 `resources/META-INF/resources`）中即可。

例如我们想访问文件：resources/static/index.html 只需要在访问时资源路径写成/index.html即可。

例如我们想访问文件：resources/static/pages/login.html 访问的资源路径写成： /pages/login.html

4.1.1 修改静态资源访问路径

SpringBoot默认的静态资源路径匹配为/** 。如果想要修改可以通过 `spring.mvc.static-path-pattern` 这个配置进行修改。

例如想让访问静态资源的url必须前缀有/res。例如/res/index.html才能访问到static目录中的。我们可以修改如下：

在application.yml中

```yaml
spring:
  mvc:
    static-path-pattern: /res/** #修改静态资源访问路径
```

4.1.2 修改静态资源存放目录

我们可以修改 spring.web.resources.static-locations 这个配置来修改静态资源的存放目录。

例如：

```yaml
spring:
  web:
    resources:
      static-locations:
        - classpath:/sgstatic/
        - classpath:/static/
```

4.2 @RequestMapping设置请求映射规则

该注解可以加到方法上或者是类上。（查看其源码可知）

我们可以用其来设定所能匹配请求的要求。只有符合了设置的要求，请求才能被加了该注解的方法或类处理。

4.2.1 指定请求路径 path=? value=?

path或者value属性都可以用来指定请求路径。

例如：我们期望让请求的资源路径为**/test/testPath**的请求能够被**testPath**方法处理则可以写如下代码

```java
@RestController
@RequestMapping("/test")
public class HelloController {
    @RequestMapping("/testPath")
    public String testPath(){
        return "testPath";
    }
}
```

```
@RestController
public class HelloController {

    @RequestMapping("/test/testPath")
    public String testPath(){
        return "testPath";
    }
}
```

4.2.2 指定请求方式 method = RequestMethod.POST

method属性可以用来指定可处理的请求方式。

例如：我们期望让请求的资源路径为**/test/testMethod**的**POST**请求能够被**testMethod**方法处理。则可以写如下代码

```
@RestController
@RequestMapping("/test")
public class TestController {

    @RequestMapping(value = "/testMethod",method =
RequestMethod.POST)
    public String testMethod(){
        System.out.println("testMethod处理了请求");
        return "testMethod";
    }
}
```

注意：我们可以也可以运用如下注解来进行替换

- @PostMapping 等价于 @RequestMapping(method = RequestMethod.POST)
- @GetMapping 等价于 @RequestMapping(method = RequestMethod.GET)
- @PutMapping 等价于 @RequestMapping(method = RequestMethod.PUT)

- `@DeleteMapping` 等价于 `@RequestMapping(method = RequestMethod.DELETE)`

例如：上面的需求我们可以使用下面的写法实现

```
@RestController
@RequestMapping("/test")
public class TestController {

    @PostMapping(value = "/testMethod")
    public String testMethod(){
        System.out.println("testMethod处理了请求");
        return "testMethod";
    }
}
```

4.2.3 指定请求参数 params = "code"

我们可以**使用params属性来对请求参数进行一些限制。可以要求必须具有某些参数，或者是某些参数必须是某个值，或者是某些参数必须不是某个值。**

例如：我们期望让请求的资源路径为**/test/testParams**的**GET**请求,并且请求参数中**具有code参数**的请求能够被testParams方法处理。则可以写如下代码

```
@RestController
@RequestMapping("/test")
public class TestController {
    @RequestMapping(value = "/testParams",method =
RequestMethod.GET,params = "code")
    public String testParams(){
        System.out.println("testParams处理了请求");
        return "testParams";
    }
}
```

如果是**要求不能有code这个参数**可以把改成如下形式

```
@RestController
@RequestMapping("/test")
public class TestController {
    @RequestMapping(value = "/testParams",method =
RequestMethod.GET,params = "!code")
    public String testParams(){
        System.out.println("testParams处理了请求");
        return "testParams";
    }
}
```

如果**要求有code这参数**，并且**这参数值必须是某个值**可以改成如下形式

```
@RestController
@RequestMapping("/test")
public class TestController {
    @RequestMapping(value = "/testParams",method =
RequestMethod.GET,params = "code=sgct")
    public String testParams(){
        System.out.println("testParams处理了请求");
        return "testParams";
    }
}
```

如果**要求有code这参数**，并且这参数值必须**不是某个值**可以改成如下形式

```
@RestController
@RequestMapping("/test")
public class TestController {
    @RequestMapping(value = "/testParams",method =
RequestMethod.GET,params = "code≠sgct")
    public String testParams(){
        System.out.println("testParams处理了请求");
        return "testParams";
    }
}
```

4.2.4 指定请求头 headers = "deviceType"

我们可以使用**headers**属性来对请求头进行一些限制。

例如：我们期望让请求的资源路径为**/test/testHeaders的**GET**请求，并且请求头中**具有**deviceType**的请求能够被testHeaders方法处理。则可以写如下代码

```
@RestController
@RequestMapping("/test")
public class TestController {

    @RequestMapping(value = "/testHeaders",method =
RequestMethod.GET,headers = "deviceType")
    public String testHeaders(){
        System.out.println("testHeaders处理了请求");
        return "testHeaders";
    }
}
```

如果是要求不能有**deviceType**这个请求头可以把改成如下形式

```
@RestController
@RequestMapping("/test")
public class TestController {

    @RequestMapping(value = "/testHeaders",method =
RequestMethod.GET,headers = "!deviceType")
    public String testHeaders(){
        System.out.println("testHeaders处理了请求");
        return "testHeaders";
    }
}
```

如果要求有deviceType这个请求头，并且其值必须**是某个值**可以改成如下形式

```
@RestController
@RequestMapping("/test")
public class TestController {

    @RequestMapping(value = "/testHeaders",method =
RequestMethod.GET,headers = "deviceType=ios")
    public String testHeaders(){
        System.out.println("testHeaders处理了请求");
        return "testHeaders";
    }
}
```

如果要求有deviceType这个请求头，并且其值必须**不是某个值**可以改成如下形式

```
@RestController
@RequestMapping("/test")
public class TestController {

    @RequestMapping(value = "/testHeaders",method =
RequestMethod.GET,headers = "deviceType≠ios")
    public String testHeaders(){
        System.out.println("testHeaders处理了请求");
        return "testHeaders";
    }
}
```

4.2.5 指定请求头Content-Type consumes = "multipart/from-data"

我们可以使用**consumes**属性来对**Content-Type**这个请求头进行一些限制。

我们期望让请求的资源路径为**/test/testConsumes**的POST请求，并且**请求头中的Content-Type**头必须为 **multipart/from-data** 的请求能够被testConsumes方法处理。则可以写如下代码

```
@RequestMapping(value = "/testConsumes",method =
RequestMethod.POST,consumes = "multipart/from-data")
public String testConsumes(){
    System.out.println("testConsumes处理了请求");
    return "testConsumes";
}
```

如果我们要求请求头Content-Type的值必须**不能为某个multipart/from-data**则可以改成如下形式：

```
@RequestMapping(value = "/testConsumes",method =
RequestMethod.POST,consumes = "!multipart/from-
data")
public String testConsumes(){
    System.out.println("testConsumes处理了请求");
    return "testConsumes";
}
```

4.3 获取请求参数 @PathVariable @RequestBody @RequestParam

4.3.1 获取路径参数 @PathVariable

RestFul风格的接口一些参数是在请求路径上的。类似： /user/1 这里的1就是id。如果我们想获取这种格式的数据可以使用**@PathVariable**来实现。

```
@RequestMapping(value = "/user/{id}",method =
RequestMethod.GET)
```

要求定义个RestFul风格的接口，该接口可以用来根据id查询用户。请求路径要求为 /user ，请求方式要求为GET。而请求参数id要写在请求路径上，**例如 /user/1 这里的1就是id**。

我们可以定义如下方法，通过如下方式来获取路径参数：请求地址http://localhost:8080/Hello/user/1

```
@RestController
public class UserController {

    @RequestMapping(value = "/user/{id}",method =
RequestMethod.GET)
    public String findUserById(
@PathVariable("id")Integer id){
        System.out.println("findUserById");
        System.out.println(id);
        return "findUserById";
    }
}
```

```
@PathVariable("id") Integer
id,@PathVariable("name") String name
```

如果这个接口，想根据id和username查询用户。请求路径要求为/user ，请求方式要求为GET。而请求参数id和name要写在请求路径上，例如/**user/1/zs 这里的1就是id，zs是name**

我们可以定义如下方法，通过如下方式来获取路径参数：请求地址http://localhost:8080/Hello/user/1/messi

```
@RestController
public class UserController {
    @RequestMapping(value =
"/user/{id}/{name}",method = RequestMethod.GET)
    public String findUser(@PathVariable("id")
Integer id,@PathVariable("name") String name){
        System.out.println("findUser");
        System.out.println(id);
        System.out.println(name);
        return "findUser";
    }
}
```

4.3.2 获取请求体中的Json格式参数 @RequestBody

RestFul风格的接口一些比较复杂的参数会转换成Json通过请求体传递过来。我们可以使用**@RequestBody**注解获取请求体中的数据。

4.3.2.1 使用

> 单个对象

要求定义个RestFul风格的接口，该接口可以用来新建用户。请求路径要求为 /user ，请求方式要求为POST。用户数据会转换成json通过请求体传递。请求体数据

```
{"username":"hq","age":15,"address":"testaddress"}
```

1.获取参数封装成实体对象

如果我们想把Json数据获取出来封装User对象,我们可以这样定义方法:


```
@RestController
public class UserController {
    @RequestMapping(value = "/user",method =
RequestMethod.POST)
    public String insertUser(@RequestBody User user)
{
        System.out.println("insertUser");
        System.out.println(user);
        return "insertUser";
    }
}
```

User实体类如下：

```
@Data
@AllArgsConstructor
@NoArgsConstructor
public class User {
    private Integer id;
    private String username;
    private Integer age;
    private String address;
}
```

2.获取参数封装成Map集合

也可以把该数据获取出来封装成Map集合：

```
@RequestMapping(value = "/userMap",method =
RequestMethod.POST)
public String insertUserMapMap(@RequestBody Map map)
{
    System.out.println("insertUser");
    System.out.println(map);
    return "insertUser";
}
```

> 多个对象

如果请求体传递过来的数据是一个User集合转换成的json，Json数据可以这样定义：

```
[{"username":"hq","age":15,"address":"testaddress"},
{"username":"mp","age":16,"address":"testAddress"}]
```

方法定义：

```
@RequestMapping(value = "/users",method =
RequestMethod.POST)
public String insertUsers(@RequestBody List<User>
users){
    System.out.println("insertUsers");
    System.out.println(users);
    return "insertUser";
}
```

4.3.2.3 注意事项

如果需要使用**@RequestBody**来获取请求体中Json并且进行转换，要求**请求头 Content-Type 的值要为： application/json** 。

4.3.3 获取QueryString格式参数 @RequestParam

我们可以使用**@RequestParam**来获取QueryString格式的参数。要求定义个接口，该接口请求路径要求为 /testRequestParam，请求方式无要求。参数为id和name和likes。使用QueryString的格式传递。

> 参数单独的获取

如果我们想把**id，name，likes单独获取出来**可以使用如下写法：在方法中定义方法参数，方法参数名要和请求参数名一致，这种情况下我们可以省略**@RequestParam**注解。

请求地址http://localhost:8080/Hello/testRequestParam?id=21&name=hqTest&likes=test1&likes=test2&likes=test3

```
@RequestMapping("/testRequestParam")
public String testRequestParam(Integer id, String
name, String[] likes){
    System.out.println("testRequestParam");
    System.out.println(id);
    System.out.println(name);
    System.out.println(Arrays.toString(likes));
    return "testRequestParam";
}
```

如果方法参数名和请求参数名不一致，我们可以加上**@RequestParam**注解
例如：

```
@RequestMapping("/testRquestParam")
public String testRquestParam(@RequestParam("id")
Integer uid,@RequestParam("name") String name,
@RequestParam("likes")String[] likes){
    System.out.println("testRquestParam");
    System.out.println(uid);
    System.out.println(name);
    System.out.println(Arrays.toString(likes));
    return "testRquestParam";
}
```

> 获取参数封装成实体对象

如果我们想把这些参数封装到一个User对象中可以使用如下写法：

```
@RequestMapping("/testRquestParam")
public String testRquestParam(User user){
    System.out.println("testRquestParam");
    System.out.println(user);
    return "testRquestParam";
}
```

User类定义如下：

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class User {
    private Integer id;
    private String name;
    private Integer age;
    private String[] likes;
}
```

测试时请求url如下：

```
http://localhost:8080/testRquestParam?id=1&name=三更草堂&likes=编程&likes=录课&likes=烫头
```

注意：实体类中的成员变量要和请求参数名对应上。并且要提供对应的set/get方法。

4.3.4 相关注解其他属性 required defaultValue

> required

- 代表是否必须，**默认值为true也就是必须要有对应的参数**。如果没有就会报错。
- 如果对应的参数**可传可不传则可以把其设置为false**

```
@RequestMapping("/testRquestParam")
public String testRquestParam(@RequestParam(value =
"id",required = false) Integer
uid,@RequestParam("name") String name,
@RequestParam("likes")String[] likes){
    System.out.println("testRquestParam");
    System.out.println(uid);
    System.out.println(name);
    System.out.println(Arrays.toString(likes));
    return "testRquestParam";
}
```

> defaultValue

如果对应的参数没有，我们可以用**defaultValue属性设置默认值。**

```
@RequestMapping("/testRquestParam")
public String testRquestParam(@RequestParam(value =
"id",required = false,defaultValue = "777") Integer
uid,@RequestParam("name") String name,
@RequestParam("likes")String[] likes){
    System.out.println("testRquestParam");
    System.out.println(uid);
    System.out.println(name);
    System.out.println(Arrays.toString(likes));
    return "testRquestParam";
}
```

4.4 响应体响应数据@ResponseBody

无论是RestFul风格还是我们之前web阶段接触过的异步请求，都需要**把数据转换成Json放入响应体中。**

- 我们的SpringMVC为我们提供了**@ResponseBody来非常方便的把Json放到响应体中。**
- **@ResponseBody可以加在类上和方法上**

- 要求定义个RestFul风格的接口，该接口可以用来根据id查询用户。请求路径要求为 `/response/user` ，请求方式要求为GET。
- 而请求参数id要写在请求路径上，例如 `/response/user/1` 这里的1就是id。
- 要求获取参数id,去查询对应id的用户信息（模拟查询即可，可以选择直接new一个User对象），并且转换成json响应到响应体中。

```
@Controller
@RequestMapping("/response")
public class ResponseController {

    @RequestMapping("/user/{id}")
    @ResponseBody
    public User findById(@PathVariable("id") Integer
id){
        User user = new User(id, "三更草堂", 15,
null);
        return user;
    }
}
```

4.5 跨域请求

4.5.1 什么是跨域

浏览器出于安全的考虑，使用 XMLHttpRequest对象发起 **HTTP请求时必须遵守同源策略，否则就是跨域的HTTP请求**，默认情况下是被禁止的。**同源策略**要求源相同才能正常进行通信，即协议、域名、端口号都完全一致。

4.5.2 CORS解决跨域

CORS是一个W3C标准，全称是"跨域资源共享"（Cross-origin resource sharing），允许浏览器向跨源服务器，发出XMLHttpRequest请求，从而克服了AJAX只能同源使用的限制。

它通过服务器增加一个**特殊的Header[Access-Control-Allow-Origin]来告诉客户端跨域的限制，如果浏览器支持CORS、并且判断Origin通过的话**，就会允许XMLHttpRequest发起跨域请求。

4.5.3 SpringBoot使用CORS解决跨域

方式1：使用@CrossOrigin

可以在支持跨域的方法上或者是Controller上加上@CrossOrigin注解

```
@RestController
@RequestMapping("/user")
@CrossOrigin
public class UserController {

    @Autowired
    private UserServcie userServcie;

    @RequestMapping("/findAll")
    public ResponseResult findAll(){
        //调用service查询数据 ，进行返回
        List<User> users = userServcie.findAll();

        return new ResponseResult(200,users);
    }
}
```

方式2：使用 WebMvcConfigurer 的 addCorsMappings 方法配置 CorsInterceptor

```
@Configuration
public class CorsConfig implements WebMvcConfigurer
{

    @Override
```

```
    public void addCorsMappings(CorsRegistry
registry) {
        // 设置允许跨域的路径
        registry.addMapping("/**")
                // 设置允许跨域请求的域名
                .allowedOriginPatterns("*")
                // 是否允许cookie
                .allowCredentials(true)
                // 设置允许的请求方式
                .allowedMethods("GET", "POST",
"DELETE", "PUT")
                // 设置允许的header属性
                .allowedHeaders("*")
                // 跨域允许时间
                .maxAge(3600);
    }
}
```

4.6 拦截器（登录）

4.6.0 登录案例

4.6.0.1 思路分析

- 在前后端分离的场景中，很多时候会采用token的方案进行登录校验。
- 登录成功时，后端会根据一些用户信息生成一个token字符串返回给前端。
- 前端会存储这个token。以后前端发起请求时如果有token就会把token放在请求头中发送给后端。
- 后端接口就可以获取请求头中的token信息进行解析，如果解析不成功说明token超时了或者不是正确的token，相当于是未登录状态。
- 如果解析成功，说明前端是已经登录过的。

4.6.0.2 Token生成方案-JWT

本案例采用目前企业中运用比较多的JWT来生成token。

使用时先引入相关依赖

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.9.0</version>
</dependency>
```

然后可以使用下面的工具类来生成和解析token

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.JwtBuilder;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;
import java.util.Date;
import java.util.UUID;

/**
 * JWT工具类
 */
public class JwtUtil {

    //有效期为
    public static final Long JWT_TTL = 60 * 60
*1000L;// 60 * 60 *1000  一个小时
    //设置秘钥明文
    public static final String JWT_KEY = "sangeng";

    /**
     * 创建token
     * @param id
```

```
     * @param subject
     * @param ttlMillis
     * @return
     */
    public static String createJWT(String id, String
subject, Long ttlMillis) {

        SignatureAlgorithm signatureAlgorithm =
SignatureAlgorithm.HS256;
        long nowMillis = System.currentTimeMillis();
        Date now = new Date(nowMillis);
        if(ttlMillis==null){
            ttlMillis=JwtUtil.JWT_TTL;
        }
        long expMillis = nowMillis + ttlMillis;
        Date expDate = new Date(expMillis);
        SecretKey secretKey = generalKey();

        JwtBuilder builder = Jwts.builder()
                .setId(id)                //唯一的ID
                .setSubject(subject)   // 主题  可以是
JSON数据
                .setIssuer("sg")     // 签发者
                .setIssuedAt(now)       // 签发时间
                .signWith(signatureAlgorithm,
secretKey) //使用HS256对称加密算法签名, 第二个参数为秘钥
                .setExpiration(expDate);// 设置过期时
间
        return builder.compact();
    }

    /**
     * 生成加密后的秘钥 secretKey
     * @return
     */
    public static SecretKey generalKey() {
```

```
        byte[] encodedKey =
Base64.getDecoder().decode(JwtUtil.JWT_KEY);
        SecretKey key = new
SecretKeySpec(encodedKey, 0, encodedKey.length,
"AES");
        return key;
    }

    /**
     * 解析
     *
     * @param jwt
     * @return
     * @throws Exception
     */
    public static Claims parseJWT(String jwt) throws
Exception {
        SecretKey secretKey = generalKey();
        return Jwts.parser()
                .setSigningKey(secretKey)
                .parseClaimsJws(jwt)
                .getBody();
    }


}
```

4.6.0.3 登录接口实现

> 数据准备

```sql
DROP TABLE IF EXISTS `sys_user`;
CREATE TABLE `sys_user` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `username` varchar(50) DEFAULT NULL,
  `password` varchar(50) DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=3 DEFAULT
CHARSET=utf8;

/*Data for the table `sys_user` */

insert  into `sys_user`(`id`,`username`,`password`)
values (1,'root','root'),(2,'sangeng','caotang');
```

> 实体类

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
public class SystemUser {

    private Integer id;
    private String username;
    private String password;
}
```

> SystemUserController

```java
@RestController
@RequestMapping("/sys_user")
public class SystemUserController {
    private final SystemUserService
systemUserService;
```

```
    public SystemUserController(SystemUserService
systemUserService) {
        this.systemUserService = systemUserService;
    }

    @PostMapping("/login")
    public ResponseResult login(@RequestBody
SystemUser user) {
        //校验用户名密码是否正确
        SystemUser loginUser =
systemUserService.login(user);
        Map<String, Object> map;
        if (loginUser != null) {
            //如果正确 生成token返回
            map = new HashMap<>();
            String token =
JwtUtil.createJWT(UUID.randomUUID().toString(),

String.valueOf(loginUser.getId()), null);
            map.put("token", token);
        } else {
            //如果不正确 给出相应的提示
            return new ResponseResult(300, "用户名或密
码错误，请重新登录");
        }
        return new ResponseResult(200, "登录成功",
map);
    }
}
```

Service

```
public interface SystemUserService {

    public SystemUser login(SystemUser user);
}
```

```
@Service
public class SystemUserServiceImpl implements
SystemUserService {

    private final SystemUserMapper systemUserMapper;

    public SystemUserServiceImpl(SystemUserMapper
systemUserMapper) {
        this.systemUserMapper = systemUserMapper;
    }

    @Override
    public SystemUser login(SystemUser user) {
        SystemUser loginUser =
systemUserMapper.login(user);
        return loginUser;
    }
}
```

> dao

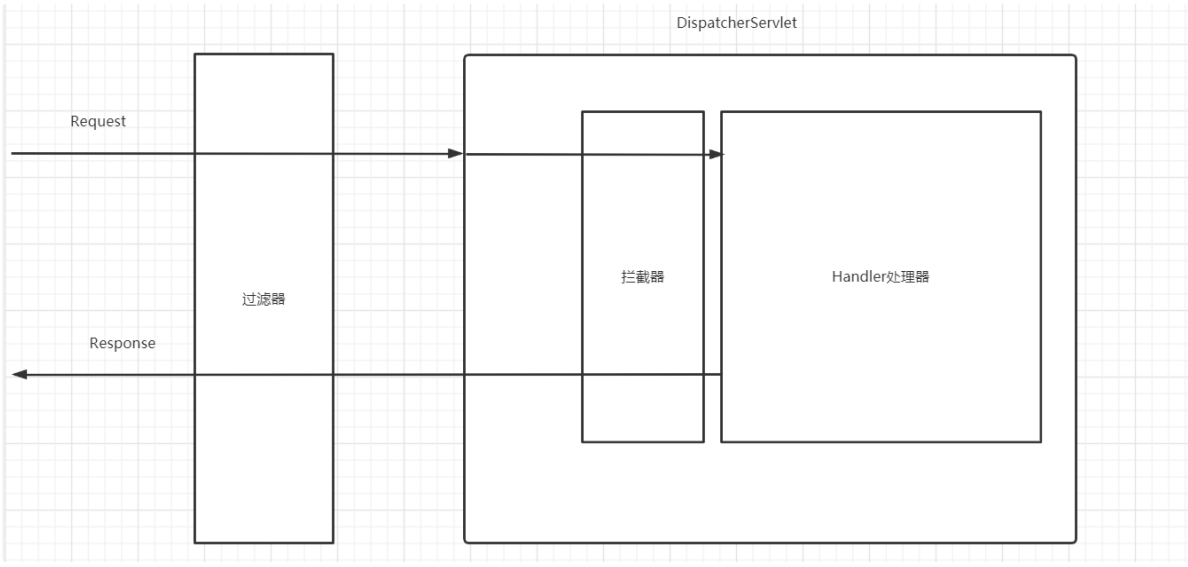
```
@Mapper
@Repository
public interface SystemUserMapper {

    SystemUser login(SystemUser user);

    List<SystemUser> findAll();
}
```

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper
3.0//EN" "http://mybatis.org/dtd/mybatis-3-
mapper.dtd" >
<mapper namespace="com.hq.mapper.SystemUserMapper">

    <select id="login"
resultType="com.hq.entity.SystemUser">
        select * from sys_user where username = #
{username} and password = #{password}
    </select>

    <select id="findAll"
resultType="com.hq.entity.SystemUser">
        select * from sys_user
    </select>
</mapper>
```

4.6.1 拦截器的概念

如果我们想在**多个Handler方法执行之前或者之后都进行一些处理**，甚至某些情况下需要拦截掉，不让Handler方法执行。那么可以使用SpringMVC为我们提供的拦截器。

详情见 https://space.bilibili.com/663528522 SpringMVC课程中拦截器相关章节。



①创建类实现HandlerInterceptor接口

```
public class LoginInterceptor implements
HandlerInterceptor {
}
```

②实现方法

```
@Component
public class LoginInterceptor implements
HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest
request, HttpServletResponse response,
                             Object handler) throws
Exception {
        //获取请求头中的token
        String token = request.getHeader("token");
        //判断token是否为空，如果为空也代表未登录 提醒重新
登录 (401)
        if(!StringUtils.hasText(token)){

 response.sendError(HttpServletResponse.SC_UNAUTHORI
ZED);
            return false;
        }
        //解析token看看是否成功
        try {
            Claims claims = JwtUtil.parseJWT(token);
            String subject = claims.getSubject();
            System.out.println(subject);
        } catch (Exception e) {
            e.printStackTrace();
            //如果解析过程中没有出现异常说明是登录状态
            //如果出现了异常，说明未登录，提醒重新登录
 (401)
```

```
 response.sendError(HttpServletResponse.SC_UNAUTHORIZED);
            return false;
        }
        return true;
    }
}
```

③配置拦截器

```
@Configuration
public class LoginConfig implements WebMvcConfigurer{

    @Autowired
    private LoginInterceptor loginInterceptor;

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(loginInterceptor)//添加拦截器
                .addPathPatterns("/**")   //配置拦截路径

.excludePathPatterns("/sys_user/login");//配置排除路径
    }
}
```

> 请求方式

4.7 异常统一处理 @ControllerAdvice

定义异常处理方法，使用**@ExceptionHandler标识可以处理的异常。**

```
@ControllerAdvice
public class MyControllerAdvice {

    @ExceptionHandler(RuntimeException.class)
    @ResponseBody
    public ResponseResult handlerException(Exception
e){
        //获取异常信息，存放如ResponseResult的msg属性
        String message = e.getMessage();
        ResponseResult result = new
ResponseResult(300,message);
        //把ResponseResult作为返回值返回，要求到时候转换成
json存入响应体中
        return result;
    }
}
```

4.8 获取web原生对象 request response session

我们之前在web阶段我们经常要**使用到request对象，response，session对象**等。我们也**可以通过SpringMVC获取到这些对象。**

```
@RequestMapping("/getRequestAndResponse")
public void getRequestAndResponse(HttpServletRequest
request,

 HttpServletResponse response, HttpSession session){
    System.out.println(request.getMethod());
}
```

4.9 自定义参数解析（@CurrentUserId String userId）

如果我们想实现像获取请求体中的数据那样，在Handler方法的参数上增加一个@RepuestBody注解就可以获取到对应的数据的话。

可以**使用HandlerMethodArgumentResolver来实现自定义的参数解析。**

> 定义用来标识的注解

```
@Target(ElementType.PARAMETER)
@Retention(RetentionPolicy.RUNTIME)
public @interface CurrentUserId {

}
```

> 创建类实现HandlerMethodArgumentResolver接口并重写其中的方法

注意加上@Component注解注入Spring容器

```
@Component
public class UserIdArgumentResolver implements
HandlerMethodArgumentResolver {

    //判断方法参数使用能使用当前的参数解析器进行解析
    @Override
    public boolean supportsParameter(MethodParameter
parameter) {
        //如果方法参数有加上CurrentUserId注解，就能把被我
们的解析器解析
        return
parameter.hasParameterAnnotation(CurrentUserId.class
);
    }
```

```
    //进行参数解析的方法，可以在方法中获取对应的数据，然后把数据作为返回值返回。方法的返回值就会赋值给对应的方法参数
    @Override
    public Object resolveArgument(MethodParameter parameter, ModelAndViewContainer mavContainer, NativeWebRequest webRequest, WebDataBinderFactory binderFactory) throws Exception {
        //获取请求头中的token
        String token = webRequest.getHeader("token");
        if(StringUtils.hasText(token)){
            //解析token，获取userId
            Claims claims = JwtUtil.parseJWT(token);
            String userId = claims.getSubject();
            //返回结果
            return userId;
        }
        return null;
    }
}
```

配置参数解析器

```
@Configuration
public class ArgumentResolverConfig implements
WebMvcConfigurer {

    @Autowired
    private UserIdArgumentResolver
userIdArgumentResolver;

    @Override
    public void
addArgumentResolvers(List<HandlerMethodArgumentResol
ver> resolvers) {
        resolvers.add(userIdArgumentResolver);
    }
}
```

> 测试

在需要获取UserId的方法中增加对应的方法参数然后使用@CurrentUserId进行标识即可获取到数据

```
@RestController
@RequestMapping("/user")
public class UserController {

    private final SystemUserService
systemUserService;

    public UserController(SystemUserService
systemUserService) {
        this.systemUserService = systemUserService;
    }

    @RequestMapping("/findAll")
    public ResponseResult findAll(@CurrentUserId
String userId) throws Exception {
```

```
        System.out.println(userId + "abc");
        //调用service查询数据 , 进行返回s
        List<SystemUser> users =
systemUserService.findAll();

        return new ResponseResult(200,users);
    }
}
```

4.10 声明式事务 @Transactional

直接在需要事务控制的方法上加上对应的注解**@Transactional**

```
@Service
public class UserServiceImpl implements UserServcie
{

    @Autowired
    private UserMapper userMapper;

    @Override
    public List<User> findAll() {
        return userMapper.findAll();
    }

    @Override
    @Transactional
    public void insertUser() {
        //添加2个用户到数据库
        User user = new User(null,"sg666",15,"上海");
        User user2 = new User(null,"sg777",16,"北
京");
        userMapper.insertUser(user);
        System.out.println(1/0);
        userMapper.insertUser(user2);
    }
```

```
}
```

4.11 AOP

AOP详细知识学习见：https://space.bilibili.com/663528522 中的Spring教程

在SpringBoot中默认是开启AOP功能的。如果不想开启AOP功能可以使用如下配置设置为false

```yaml
spring:
  aop:
    auto: false
```

4.11.1 使用步骤

> 添加依赖

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

> 自定义注解

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface InvokeLog {
}
```

> 定义切面类

切面@Aspect

- `@Aspect`是一个用于声明一个类为切面类的注解。在Spring AOP中，切面可以包含多个通知（Advice）和切点（Pointcut），用于定义在何时（在方法调用之前、之后、环绕等时机）以及在何处（符合特定条件的方法）执行特定的行为。

@Component

- `@Component`是Spring中的一个注解，用来标识一个类为Spring容器管理的一个Bean。在这里，**`@Component`使得这个切面类成为Spring管理的一个Bean**，这样Spring容器就可以识别它是一个切面，并根据里面定义的通知和切点来应用AOP代理。

切点@Pointcut

- `@Pointcut("@annotation(com.hq.annotation.InvokeLog)")`定义了一个切点，名为`pt`。**这个切点的触发条件是任何被`@InvokeLog`注解标记的方法。**`@InvokeLog`应该是一个自定义注解，用于标记那些需要被这个切面处理的方法。

环绕通知 @Around

- **`@Around("pt()")`定义了一个环绕通知，它绑定到了上面定义的`pt()`切点上。**环绕通知是AOP中的一种通知类型，它可以在目标方法执行前后执行代码，并且决定是否继续执行目标方法，或者修改返回值等。

方法 printInvokeLog(ProceedingJoinPoint joinPoint)

- 这个方法是实际执行的通知代码。`ProceedingJoinPoint`是环绕通知的一个特殊参数，它允许通知方法执行目标方法。
- 方法首先通过`joinPoint.getSignature()`获取到了被调用方法的签名，然后转换为`MethodSignature`类型以获取更具体的方法信息，如方法名。
- `System.out.println(methodName + "即将被调用");`在目标方法调用之前打印日志。
- `proceed = joinPoint.proceed();`调用目标方法，并保存返回值。如果目标方法顺利完成，接下来会打印“被调用完了”的日志。
- 如果在目标方法执行过程中抛出异常，异常会被捕获，并在控制台打印出来，同时打印出“出现了异常”的日志。
- 通知方法返回目标方法的返回值（如果有的话）。

```
@Aspect   //标识这是一个切面类
```

```java
@Component
public class InvokeLogAspect {

    //确定切点

 @Pointcut("@annotation(com.hq.annotation.InvokeLog)
")
    public void pt(){
    }

    @Around("pt()")
    public Object printInvokeLog(ProceedingJoinPoint
joinPoint){
        //目标方法调用前
        Object proceed = null;
        MethodSignature signature =
(MethodSignature) joinPoint.getSignature();
        String methodName =
signature.getMethod().getName();
        System.out.println(methodName+"即将被调用");
        try {
            proceed = joinPoint.proceed();
            //目标方法调用后
            System.out.println(methodName+"被调用完
了");
        } catch (Throwable throwable) {
            throwable.printStackTrace();
            //目标方法出现异常了
            System.out.println(methodName+"出现了异
常");
        }
        return proceed;
    }
}
```

在需要正确的地方增加对应的注解

```
@Service
public class UserServiceImpl implements UserServcie
{

    @Autowired
    private UserMapper userMapper;

    @Override
    @InvokeLog   //需要被增强方法需要加上对应的注解
    public List<User> findAll() {
        return userMapper.findAll();
    }
}
```

4.11.2 切换动态代理 proxy-target-class: false

有的时候我们需要修改AOP的代理方式。

我们可以使用以下方式修改：在配置文件中**配置spring.aop.proxy-target-class为false这为使用jdk动态代理**。该配置默认值为true，代表使用cglib动态代理。

```
@SpringBootApplication
@EnableAspectJAutoProxy(proxyTargetClass = false)//
修改代理方式
public class WebApplication {
    public static void main(String[] args) {
        ConfigurableApplicationContext context =
SpringApplication.run(WebApplication.class, args);
    }
}
```

如果想生效还需要在配置文件中做如下配置

```yaml
spring:
  aop:
    proxy-target-class: false #切换动态代理的方式
```

4.12 模板引擎相关-Thymeleaf

4.12.1 快速入门

4.12.1.1依赖

```
        <!--thymeleaf依赖-->
        <dependency>

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

4.12.1.2定义Controller

在controller中往域中存数据，并且跳转

```java
@Controller
public class ThymeleafController {

    @Autowired
    private UserServcie userServcie;

    @RequestMapping("/thymeleaf/users")
    public String users(Model model){
        //获取数据
        List<User> users = userServcie.findAll();
        //望域中存入数据
        model.addAttribute("users",users);
        model.addAttribute("msg","hello thymeleaf");
```

```
        //页面跳转
        return "table-standard";
    }
}
```

4.12.1.3 htmL

在**resources\templates**下存放模板页面。

在html标签中加上 xmlns:th="http://www.thymeleaf.org"

获取域中的name属性的值可以使用： ${name} 注意要在th开头的属性中使用

```
<html lang="en" class="no-ie"
xmlns:th="http://www.thymeleaf.org">
 .....
 <div class="panel-heading" th:text="${msg}">Kitchen
Sink</div>
```

如果需要引入静态资源，需要使用如下写法。

```
    <link rel="stylesheet"
th:href="@{/app/css/bootstrap.css}">
    <!-- Vendor CSS -->
    <link rel="stylesheet"
th:href="@{/vendor/fontawesome/css/font-
awesome.min.css}">
    <link rel="stylesheet"
th:href="@{/vendor/animo/animate+animo.css}">
    <link rel="stylesheet"
th:href="@{/vendor/csspinner/csspinner.min.css}">
    <!-- START Page Custom CSS -->
    <!-- END Page Custom CSS -->
    <!-- App CSS -->
    <link rel="stylesheet"
th:href="@{/app/css/app.css}">
    <!-- Modernizr JS Script -->
    <script
th:src="@{/vendor/modernizr/modernizr.js}"
type="application/javascript"></script>
    <!-- FastClick for mobiles -->
    <script
th:src="@{/vendor/fastclick/fastclick.js}"
type="application/javascript"></script>
```

遍历语法：遍历的语法　th:each="自定义的元素变量名称 ： ${集合变量名称}"

```
<tr th:each="user:${users}">
    <td th:text="${user.id}"></td>
    <td th:text="${user.username}"></td>
    <td th:text="${user.age}"></td>
    <td th:text="${user.address}"></td>
</tr>
```

5.整合Redis todo

①依赖

```
<!--redis-->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-
redis</artifactId>
</dependency>
```

②配置Redis地址和端口号

```
spring:
  redis:
    host: 127.0.0.1 #redis服务器ip地址
    port: 6379  #redis端口号
```

③注入RedisTemplate使用

```
@Autowired
private StringRedisTemplate redisTemplate;

@Test
public void testRedis(){
    redisTemplate.opsForValue().set("name","三更");
}
```

6.环境切换

6.1 为什么要使用profile

在实际开发环境中，我们存在开发环境的配置，部署环境的配置，测试环境的配置等等，里面的配置信息很多时，例如：端口、上下文路径、数据库配置等等，若每次切换环境时，我们都需要进行修改这些配置信息时，会比较麻烦，profile的出现就是为了解决这个问题。它可以让我们针对不同的环境进行不同的配置，然后可以通过激活、指定参数等方式快速切换环境。

6.2 使用

6.2.1 创建profile配置文件

- 我们可以用**application-xxx.yml**的命名方式 创建配置文件，其中xxx可以根据自己的需求来定义。
- 我们需要一个**测试环境**的配置文件，则可以命名为：**application-test.yml**
- 需要一个**生产环境**的配置文件，可以命名为：**application-prod.yml**

6.2.2 激活环境

- 我们可以再**application.yml**文件中使用**spring.profiles.active**属性来配置激活哪个环境。
- 也可以使用虚拟机参数来指定激活环境。例如 ： **-Dspring.profiles.active=test**
- 也可以使用命令行参数来激活环境。例如： java -jar mybatis_plus_test-1.0-SNAPSHOT.jar **--spring.profiles.active=test**

```yaml
spring:
    profiles:
        active: test # application-test.yml
```

7.日志

```yaml
debug: true #开启日志
logging:
  level:
    com.hq: debug #设置日志级别
```

8.指标监控

我们在日常开发中需要对程序内部的运行情况进行监控， **比如：健康度、运行指标、日志信息、线程状况等等** 。而SpringBoot的监控Actuator就可以帮我们解决这些问题。

8.1 使用

①添加依赖

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

②访问监控接口

localhost:8080/actuator

③配置启用监控端点

```yaml
management:
  endpoints:
    enabled-by-default: true #配置启用所有端点
    web:
      exposure:
        include: "*" #web端暴露所有端点
```

8.2 常用端点

端点名称	描述
beans	显示应用程序中所有Spring Bean的完整列表。
health	显示应用程序运行状况信息。
info	显示应用程序信息。
loggers	显示和修改应用程序中日志的配置。
metrics	显示当前应用程序的“指标”信息。
mappings	显示所有@RequestMapping路径列表。
scheduledtasks	显示应用程序中的计划任务。

8.3 图形化界面 SpringBoot Admin 需要新建项目

①创建SpringBoot Admin Server应用

要求引入spring-boot-admin-starter-server依赖

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-
server</artifactId>
</dependency>
```

然后在启动类上加上@EnableAdminServer注解

②配置SpringBoot Admin client应用

在需要监控的应用中加上spring-boot-admin-starter-client依赖

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-
client</artifactId>
    <version>2.3.1</version>
</dependency>
```

然后配置SpringBoot Admin Server的地址

```
spring:
  boot:
    admin:
      client:
        url: http://localhost:8888 #配置 Admin Server
的地址
```

若依项目学习

1、前端对axios的封装

```
// 是否需要设置 token
const isToken = (config.headers || {}).isToken ===
false
```

1. 检查 config.headers：首先，表达式检查 config.headers 是否存在。如果 config.headers 为 null 或 undefined，则表达式会评估右侧的空对象 {}。
2. 获取 isToken 值：接着，表达式尝试获取 isToken 属性的值。如果 config.headers 存在且包含 isToken 字段，它将获取该值；如果 config.headers 不存在，将会尝试从空对象中获取 isToken，这自然会返回 undefined。
3. 比较结果：最后，表达式检查获取到的 isToken 值是否严格等于 false。

1.1、Get前端封装

request.js拦截器对我们发送的请求进行了封装，当我们发送Get请求，那么我们携带参数的时候可以放在路径上，也可以用param，对应下面的源码。请求会被构造成如下形式：http://xxxxx:port/aaa?key1=value1&key2=value2。

```
// get请求映射params参数，如果params为空，if就不会执行
if (config.method === 'get' && config.params) {
 let url = config.url + '?' +
tansParams(config.params);
 // 移除拼接后URL末尾的多余字符（通常是一个多余的'&'）
 url = url.slice(0, -1);
 config.params = {};
 config.url = url;
}
```

1.2、Post前端封装

request.js拦截器对我们发送的请求进行了封装，当我们发送Post请求，主要参数就是url和data。

```
if (!isRepeatSubmit && (config.method === 'post' ||
config.method === 'put')) {
 // 如果 config.data 是一个对象，则将其转换为 JSON 字符
串。否则，直接使用 config.data。
 const requestObj = {
   url: config.url,
   data: typeof config.data === 'object' ?
JSON.stringify(config.data) : config.data,
   time: new Date().getTime()
 }
 // 处理重复提交
 const sessionObj =
cache.session.getJSON('sessionObj')
 if (sessionObj === undefined || sessionObj === null
|| sessionObj === '') {
```

```
    cache.session.setJSON('sessionObj', requestObj)
  } else {
    const s_url = sessionObj.url;                    //
请求地址
    const s_data = sessionObj.data;                  //
请求数据
    const s_time = sessionObj.time;                  //
请求时间
    const interval = 1000;                           //
间隔时间(ms)，小于此时间视为重复提交
    // 如果上一个请求的数据和 URL 与当前请求相同，并且两次请
求的时间差小于 interval (即1000毫秒)，则判断为重复提交。
    if (s_data === requestObj.data && requestObj.time
- s_time < interval && s_url === requestObj.url) {
      const message = '数据正在处理，请勿重复提交';
      console.warn(`[${s_url}]: ` + message)
      return Promise.reject(new Error(message))
    } else {
      cache.session.setJSON('sessionObj', requestObj)
    }
  }
}
```

1.3、传参方式

- list()方法:是通过SysFileInfo对象 (get:params) 作为参数传递查询条件,再通过TableDataInfo和getDataTable()进行分页和数据处理。
- getInfo()方法:是通过文件id(fileId)的路径变量方式传参。
- add()方法:是通过@RequestBody注解的SysFileInfo对象 (post:data)直接作为参数传递要新增的数据。

```
// 查询文件信息列表
export function listInfo(query) {
  return request({
    url: '/pill/info/list',
```

```
    method: 'get',
    params: query
  })
}


// 查询文件信息详细
export function getInfo(fileId) {
  return request({
    url: '/pill/info/' + fileId,
    method: 'get'
  })
}

// 新增文件信息
export function addInfo(data) {
  return request({
    url: '/pill/info',
    method: 'post',
    data: data
  })
}
```

```
/**
  * 查询文件信息列表
  */
@PreAuthorize("@ss.hasPermi('pill:info:list')")
@GetMapping("/list")
public TableDataInfo list(SysFileInfo sysFileInfo)
{
    List<String> res = new ArrayList<>();
    ...
    return getDataTable(list);
}

/**
  * 获取文件信息详细信息
```

```
    */
@PreAuthorize("@ss.hasPermi('pill:info:query')")
@GetMapping(value = "/{fileId}")
public AjaxResult getInfo(@PathVariable("fileId")
Long fileId)
{
    return
success(sysFileInfoService.selectSysFileInfoByFileId
(fileId));
}

/**
  * 新增文件信息
  */
@PreAuthorize("@ss.hasPermi('pill:info:add')")
@PostMapping
public AjaxResult add(@RequestBody SysFileInfo
sysFileInfo)
{
    return
toAjax(sysFileInfoService.insertSysFileInfo(sysFileI
nfo));
}
```

1.4、反向代理解决跨域问题

假设环境变量 `VUE_APP_BASE_API` 的值为 `/api`。这意味着，如果你的 Vue.js 应用尝试向 `/api/some-endpoint` 发起请求，这个请求会被代理。

- 原始请求路径：`/api/some-endpoint`
- 代理目标：`http://localhost:8080`
- 经过 `pathRewrite` 后，请求路径变为：`/some-endpoint`
- 因此，最终请求的 URL 是：`http://localhost:8080/some-endpoint`

- 在正则表达式中，`'^'` 符号用于表示匹配字符串的开始。如果请求的路径是 `/some/api/users`，这个正则表达式不会匹配这个路径，因为 `/api` 不是在路径的开头。

```
proxy: {
      // detail:
https://cli.vuejs.org/config/#devserver-proxy
      [process.env.VUE_APP_BASE_API]: { // 使用环境变
量 VUE_APP_BASE_API 的值作为代理规则的键
        target: `http://localhost:8080`, // 设置代理目
标地址
        changeOrigin: true,
        pathRewrite: {
          ['^' + process.env.VUE_APP_BASE_API]: ''
        }
      }
    },
```

2、登陆实现

2.1、验证码逻辑

前端调用get请求方法，后端解析完成，得到返回值（uuid,img）之后，赋值渲染给前端页面

```
<script>
    getCode() {
      getCodeImg().then(res => {
        this.captchaEnabled = res.captchaEnabled ===
undefined ? true : res.captchaEnabled;
        if (this.captchaEnabled) {
          this.codeUrl = "" +
res.img;
          this.loginForm.uuid = res.uuid;
        }
      });
    },
</script>
```

```
// 获取验证码
export function getCodeImg() {
  return request({
    url: '/captchaImage',
    headers: {
      isToken: false
    },
    method: 'get',
    timeout: 20000
  })
}
```

```
@GetMapping("/captchaImage")
public AjaxResult getCode(HttpServletResponse
response) throws IOException {
    ...
    ajax.put("uuid", uuid);
    ajax.put("img",
Base64.encode(os.toByteArray()));
    return ajax;
}
```

2.2、login逻辑

1. **this.$refs**：在Vue.js中，$refs用于直接访问组件模板中的DOM元素或子组件。这里this.$refs.loginForm直接引用了模板中的登录表单组件，以便调用其validate方法。在这段代码中，validate方法是Element UI框架（一个流行的Vue.js UI库）中的el-form组件的一个方法。validate方法用于校验表单项是否符合预定义的规则，这是在loginForm中定义的规则（）。
2. **this.$store**：$store是Vuex提供的一个属性，用于访问Vue应用中的全局状态管理。this.$store.dispatch用于触发状态管理中的action，这里用于处理登录逻辑。
3. **this.$router**：$router是Vue Router的实例，用于编程式地导航到不同的URL。this.$router.push用于在**登录成功后改变当前路由，从而导航到不同的页面。**

```
<script>
    handleLogin() {
      // `validate`方法用于校验表单项是否符合预定义的规则
      this.$refs.loginForm.validate(valid => {
        if (valid) {
          this.loading = true;
          // 这一部分代码检查是否勾选了“记住
我”（rememberMe）。如果是，则使用Cookies存储用户名、
          // 加密后的密码和记住我选项，有效期30天。如果没有
勾选，则移除这些Cookies。
          if (this.loginForm.rememberMe) {
            Cookies.set("username",
this.loginForm.username, { expires: 30 });
            Cookies.set("password",
encrypt(this.loginForm.password), { expires: 30 });
            Cookies.set('rememberMe',
this.loginForm.rememberMe, { expires: 30 });
          } else {
            Cookies.remove("username");
            Cookies.remove("password");
            Cookies.remove('rememberMe');
          }
```

```
          this.$store.dispatch("Login",
this.loginForm).then(() => {
            this.$router.push({ path: this.redirect
|| "/" }).catch(()=>{});
          }).catch(() => {
            this.loading = false;
            if (this.captchaEnabled) {
              this.getCode();
            }
          });
        }
      });
    }
</script>
```

$store是Vuex提供的一个属性，用于访问Vue应用中的全局状态管理。具体的action定义在user.js中,这段代码展示了在Vuex中处理异步操作（如登录），并在操作成功或失败时更新应用的状态。在Vuex中，commit是用来触发（或“提交”）一个mutation的方法。Vuex的mutations是改变应用状态（store中的state）的唯一方式。当你想修改store中的状态时，你需要通过调用mutation来实现，而commit就是用来调用这个mutation的。这里的commit函数用于执行一个名为SET_TOKEN的mutation，并传递res.token作为参数。SET_TOKEN是在Vuex的mutations部分定义的一个函数，它负责将用户的登录令牌保存到Vuex的状态中。虽然Vuex中的状态是全局的，但它是特定于每个用户的浏览器会话的。每个用户的Vuex状态是独立的，不同用户之间的状态不会相互影响。

```
actions: {
    // 登录,`commit`是用来触发（或“提交”）一个`mutation`
的方法
    Login({ commit }, userInfo) {
      const username = userInfo.username.trim()
      const password = userInfo.password
      const code = userInfo.code
      const uuid = userInfo.uuid
```

```
      // 返回一个新的 Promise 对象，这允许异步调用 Login
action 的组件能够得知登录操作何时成功或失败。
      return new Promise((resolve, reject) => {
        login(username, password, code,
uuid).then(res => {
          // 成功登录时：setToken(res.token) 保存令牌
          setToken(res.token)
          commit('SET_TOKEN', res.token)
          resolve()
        }).catch(error => {
          reject(error)
        })
      })
    },
}
```

异步调用的是login方法。

```
// 登录方法
export function login(username, password, code,
uuid) {
  const data = {
    username,
    password,
    code,
    uuid
  }
  return request({
    url: '/login',
    headers: {
      isToken: false
    },
    method: 'post',
    data: data
  })
}
```

后端执行login方法,返回token.

```
@PostMapping("/login")
public AjaxResult login(@RequestBody LoginBody
loginBody)
{
    AjaxResult ajax = AjaxResult.success();
    // 生成令牌
    String token =
loginService.login(loginBody.getUsername(),

 loginBody.getPassword(), loginBody.getCode(),

 loginBody.getUuid());
    ajax.put(Constants.TOKEN, token);
    return ajax;
}
```

2.3、参数设置自动加载

@PostConstruct注解用来标注一个非静态的void()方法,这个方法会在服务器加载Servlet的时候运行,并且只会被服务器调用一次。

```
@Service
public class SysConfigServiceImpl implements
ISysConfigService
    @PostConstruct
    public void init()
    {
        loadingConfigCache();
    }
}
```

2.4、登录认证逻辑

在登陆方法中，我们使用authenticationManager进行认证。**Spring Security的核心逻辑全在过滤器中，过滤器里会调用各种组件完成功能！**AuthenticationContextHolder存入authenticationToken，一边后续校验使用。

```
// 用户验证
Authentication authentication = null;
try {
    UsernamePasswordAuthenticationToken
authenticationToken = new

 UsernamePasswordAuthenticationToken(username,
password);

 AuthenticationContextHolder.setContext(authenticati
onToken);
    // 该方法会去调用
UserDetailsServiceImpl.loadUserByUsername
    authentication =
authenticationManager.authenticate(authenticationTok
en);
}
catch (Exception e) {}
```

若需要获得AuthenticationManager的实例，则需要写个类，该类继承WebSecurityConfigurerAdapter类，然后代码注入bean可以获得认证管理器。通过覆写 authenticationManagerBean 方法并使用 @Bean 注解，您显式地告诉Spring框架创建并管理 AuthenticationManager 的实例，使得它可以被其他部分的应用程序（如其他Bean或配置类）所使用。

```
@Resource
private AuthenticationManager authenticationManager;

@Bean
@Override
public AuthenticationManager
authenticationManagerBean() throws Exception
{
    return super.authenticationManagerBean();
}
```

UserDetailsServiceImpl类实现了UserDetialsService 接口，并重写了loadUserByUsername方法，该方法返回UserDetails接口实例对象，LoginUser类实现了UserDetails接口，并重写了接口内要求的方法getPassword和getUsername。UserDetailsServiceImpl类中的loadUserByUsername方法主要是去数据库查询，通过username找到需要检索的用户，然后对用户做简单校验，通过AuthenticationContextHolder取出token，获取用户名称和密码。对于PasswordEncoder，项目使用的BCryptPasswordEncoder，只需要注入到Spring容器即可被SpringSecurity找到。

```
public UserDetails loadUserByUsername(String
username) throws UsernameNotFoundException
{
    SysUser user =
userService.selectUserByUserName(username);
    ...
    passwordService.validate(user);
    return createLoginUser(user);
}

public void validate(SysUser user)
{
    Authentication
usernamePasswordAuthenticationToken =
AuthenticationContextHolder.getContext();
```

```
    String username =
usernamePasswordAuthenticationToken.getName();
    String password =
usernamePasswordAuthenticationToken.getCredentials()
.toString();
    ...
    if (!matches(user, password))
    {
        ...
    }
    ...
}

public boolean matches(SysUser user, String
rawPassword)
{
    return
SecurityUtils.matchesPassword(rawPassword,
user.getPassword());
}

public static boolean matchesPassword(String
rawPassword, String encodedPassword)
{
    BCryptPasswordEncoder passwordEncoder = new
BCryptPasswordEncoder();
    return passwordEncoder.matches(rawPassword,
encodedPassword);
}

@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder()
{
    return new BCryptPasswordEncoder();
}
```

认证完成之后，login方法拿着经过认证的authentication实例生成了token，并把token和LoginUser存到了Redis，再把token返回给了前端。前端拿到token就立刻存到了cookie中，并且下次给后端发请求的时候带上了token表明自己的身份。每一次后端被请求时，都会执行JwtAuthenticationTokenFilter过滤器，该过滤器是在从请求request的请求头（key为Authorization的条目）中获得token，然后验证token，验证成功则直接封装一个已经成功认证的信息authenticationToken，并存在上下文中。

3、菜单逻辑

permission.js中，router.beforeEach 是一个全局前置守卫，每次路由变更之前都会执行。

- GetInfo异步获取用户信息。
- 获取成功后，GenerateRoutes生成基于角色的路由表，并动态添加路由。
- 如果获取用户信息失败，LogOut执行注销操作并重定向到登录页面。

```
router.beforeEach((to, from, next) => {
    store.dispatch('GetInfo').then(() => {
      isRelogin.show = false

 store.dispatch('GenerateRoutes').then(accessRoutes
=> {
        router.addRoutes(accessRoutes)
        next({ ...to, replace: true })
      })
    }).catch(err => {
      store.dispatch('LogOut').then(() => {
        Message.error(err)
        next({ path: '/' })
      })
    })
}
```

3.1、获取用户权限和角色

这段代码是一个Vuex action，名为GetInfo，用于异步获取用户信息并更新Vuex store中的状态。检查返回的角色信息是否有效。如果是有效的非空数组，则提交 SET_ROLES 和 SET_PERMISSIONS mutations 来更新用户的角色和权限信息。如果无有效角色，则默认赋予一个 'ROLE_DEFAULT' 角色。

```
// 获取用户信息
GetInfo({ commit, state }) {
    return new Promise((resolve, reject) => {
        getInfo().then(res => {
            const user = res.user
            const avatar = (user.avatar == "" ||
user.avatar == null) ?

 require("@/assets/images/profile.jpg") :
process.env.VUE_APP_BASE_API + user.avatar;
            if (res.roles && res.roles.length > 0) {
// 验证返回的roles是否是一个非空数组
                commit('SET_ROLES', res.roles)
                commit('SET_PERMISSIONS',
res.permissions)
            } else {
                commit('SET_ROLES',
['ROLE_DEFAULT'])
            }
            commit('SET_NAME', user.userName)
            commit('SET_AVATAR', avatar)
            resolve(res)
        }).catch(error => {
            reject(error)
        })
    })
},
```

后端查询到数据（角色和权限）后，返回给前端。

```
@GetMapping("getInfo")
public AjaxResult getInfo() {
    SysUser user =
SecurityUtils.getLoginUser().getUser();
    // 角色集合
    Set<String> roles =
permissionService.getRolePermission(user);
    // 权限集合
    Set<String> permissions =
permissionService.getMenuPermission(user);
    AjaxResult ajax = AjaxResult.success();
    ajax.put("user", user);
    ajax.put("roles", roles);
    ajax.put("permissions", permissions);
    return ajax;
}
```

3.2、生成动态路由

```
// 生成路由
GenerateRoutes({ commit }) {
    return new Promise(resolve => {
        // 向后端请求路由数据
        getRouters().then(res => {
            const sdata =
JSON.parse(JSON.stringify(res.data))
            const rdata =
JSON.parse(JSON.stringify(res.data))
            const sidebarRoutes =
filterAsyncRouter(sdata)
            const rewriteRoutes =
filterAsyncRouter(rdata, false, true)
            const asyncRoutes =
filterDynamicRoutes(dynamicRoutes);
            rewriteRoutes.push({ path: '*',
redirect: '/404', hidden: true })
```

```
            router.addRoutes(asyncRoutes);
            commit('SET_ROUTES', rewriteRoutes)
            commit('SET_SIDEBAR_ROUTERS',
constantRoutes.concat(sidebarRoutes))
            commit('SET_DEFAULT_ROUTES',
sidebarRoutes)
            commit('SET_TOPBAR_ROUTES',
sidebarRoutes)
            resolve(rewriteRoutes)
        })
    })
}
}
}
```

后端查询到数据(权限树)后，返回给前端。

```
@GetMapping("getRouters")
public AjaxResult getRouters()
{
    Long userId = SecurityUtils.getUserId();
    List<SysMenu> menus =
menuService.selectMenuTreeByUserId(userId);
    return
AjaxResult.success(menuService.buildMenus(menus));
}
```

4、通用下载方法

index.vue里面定义了button按钮和methods方法,传入三个参数，这里的三个点表示展开运算符(spread syntax),它将一个对象或数组展开,将所有可遍历属性拷贝到当前对象中。

```
<el-col :span="1.5">
```

```
        <el-button
          type="warning"
          plain
          icon="el-icon-download"
          size="mini"
          @click="handleExport"
          v-hasPermi="['system:dict:export']"
        >导出</el-button>
</el-col>

<script>
export default {
    methods: {
        /** 导出按钮操作 */
        handleExport() {
            this.download('system/dict/type/export',
{
                ...this.queryParams
            }, `type_${new Date().getTime()}.xlsx`)
        }
    };
</script>
```

main.js全局挂载

```
Vue.prototype.download = download
```

request.js里面定义了download方法

```
// 通用下载方法
export function download(url, params, filename,
config) {
  downloadLoadingInstance = Loading.service({ text:
"正在下载数据，请稍候", spinner: "el-icon-loading",
background: "rgba(0, 0, 0, 0.7)", })
  return service.post(url, params, {
```

```
    transformRequest: [(params) => { return
tansParams(params) }],
    headers: { 'Content-Type': 'application/x-www-
form-urlencoded' },
    responseType: 'blob', // 设置返回响应类型为blob二进
制数据
    ...config
  }).then(async (data) => {
    const isLogin = await blobValidate(data);
    if (isLogin) {
      const blob = new Blob([data])
      saveAs(blob, filename) // 调用文件下载的api,自动
下载文件
    }
    downloadLoadingInstance.close();
  }).catch((r) => {
    console.error(r)
    Message.error('下载文件出现错误，请联系管理员！')
    downloadLoadingInstance.close();
  })
}
```

后端执行具体的方法，response负责将内存里生成的Excel文件的二进制数据写入输出流,传输给浏览器,以实现文件的下载。

```
@PostMapping("/export")
public void export(HttpServletResponse response,
SysDictType dictType)
{
    List<SysDictType> list =
dictTypeService.selectDictTypeList(dictType);
    ExcelUtil<SysDictType> util = new
ExcelUtil<SysDictType>(SysDictType.class);
    util.exportExcel(response, list, "字典类型");
}
```

5、异步任务管理器

首先有一个工厂类，定义了一系列方法。工厂模式是用来创建对象的一种最常用的设计模式，不暴露创建对象的具体逻辑，而是将将逻辑封装在一个函数中，由工厂管理对象的创建逻辑，调用方不需要知道具体的创建过程，只管使用，**而降低调用者因为创建逻辑导致的错误**；将new操作简单封装，遇到new的时候就应该考虑是否用工厂模式。

```
public static TimerTask recordLogininfor(final
String username, final String status,
                                         final
String message, final Object... args)
{
    final UserAgent userAgent =

 UserAgent.parseUserAgentString(ServletUtils.getRequ
est().getHeader("User-Agent"));
    final String ip = IpUtils.getIpAddr();
    return new TimerTask()
    {
        @Override
        public void run()
        {
            ...
            // 插入数据

 SpringUtils.getBean(ISysLogininforService.class).in
sertLogininfor(logininfor);
        }
    };
}
```

异步任务管理器AsyncManager

1. 使用ScheduledExecutorService线程池调度和执行任务。
2. 使用单例模式,保证全局只有一个AsyncManager实例。

3. 提供execute方法,可以向线程池提交一个TimerTask任务,操作会有10毫秒的延迟。

```
public class AsyncManager
{
    ...

    /**
     * 单例模式
     */
    private AsyncManager(){}

    private static AsyncManager me = new
AsyncManager();

    public static AsyncManager me() {
        return me;
    }

    /**
     * 执行任务
     *
     * @param task 任务
     */
    public void execute(TimerTask task) {
        executor.schedule(task, OPERATE_DELAY_TIME,
TimeUnit.MILLISECONDS);
    }
    ...
}
```

线程池ScheduledExecutorService在项目里已经进行配置。这里创建了一个ScheduledThreadPoolExecutor实例,并且通过ThreadFactory设置了线程工厂,给线程池中的线程设置了命名模式"schedule-pool-%d",并设置为守护线程。 最后还override了afterExecute方法,添加了异常处理。**守护线程 (Daemon Thread) 也**

被称之为后台线程或服务线程，守护线程是为用户线程服务的，当程序中的用户线程全部执行结束之后，守护线程也会跟随结束。

```
/**
  * 执行周期性或定时任务
  */
@Bean(name = "scheduledExecutorService")
protected ScheduledExecutorService
scheduledExecutorService()
{
    return new
ScheduledThreadPoolExecutor(corePoolSize,
                                                new
BasicThreadFactory.Builder().namingPattern("schedule
-pool-%d").daemon(true).build(),
                                                new
ThreadPoolExecutor.CallerRunsPolicy())
    {
        @Override
        protected void afterExecute(Runnable r,
Throwable t)
        {
            super.afterExecute(r, t);
            Threads.printException(r, t);
        }
    };
}

// 具体执行的代码
AsyncManager.me().execute(AsyncFactory.recordLoginin
for(username, Constants.LOGIN_FAIL,
MessageUtils.message("user.jcaptcha.expire")));
```

##