Springboot框架

一. SpringBoot介绍

1.1 概念

> Springboot采用了 `习惯优于配置/约定大于配置` 的理念快速的搭建项目的开发环境，我们无需或者进行很少的相关spring配置就能够快速将项目运行起来。

1.2 优点

- 能够快速搭建项目
- 内置了一些配置不需要J集成
- 提高了开发效率、部署效率
- 内置tomcat等

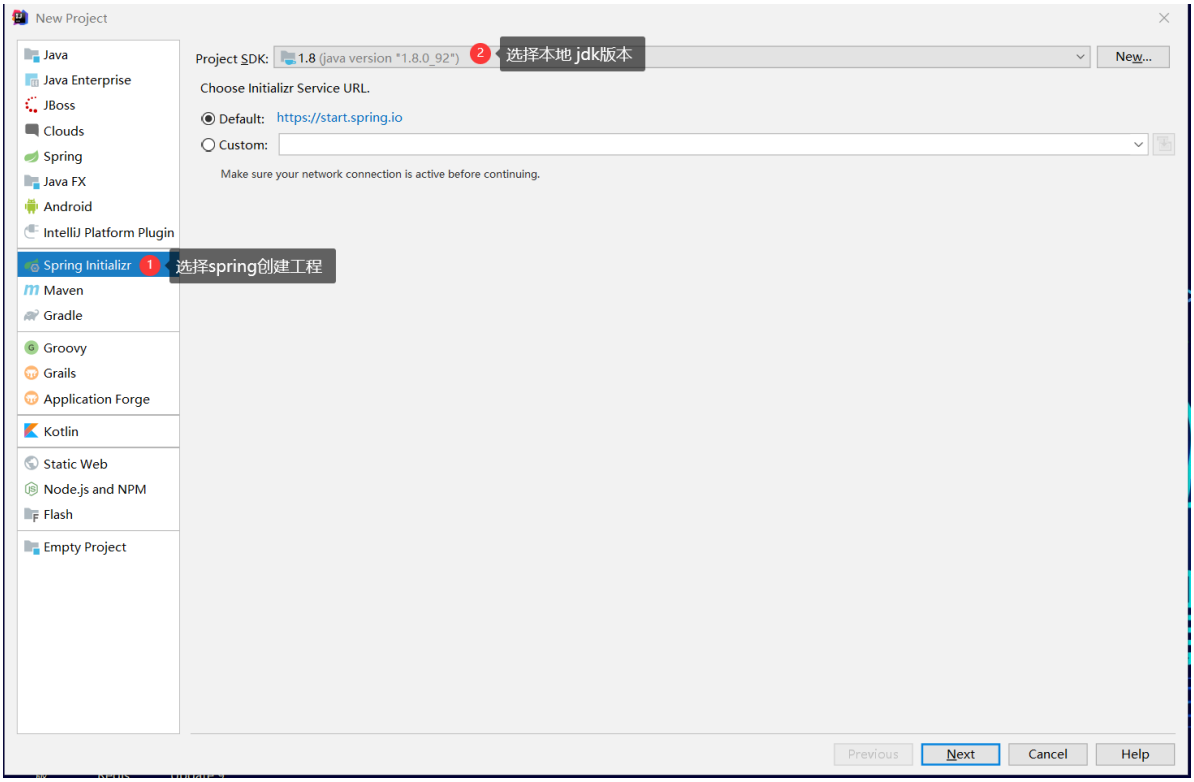
1.3 缺点

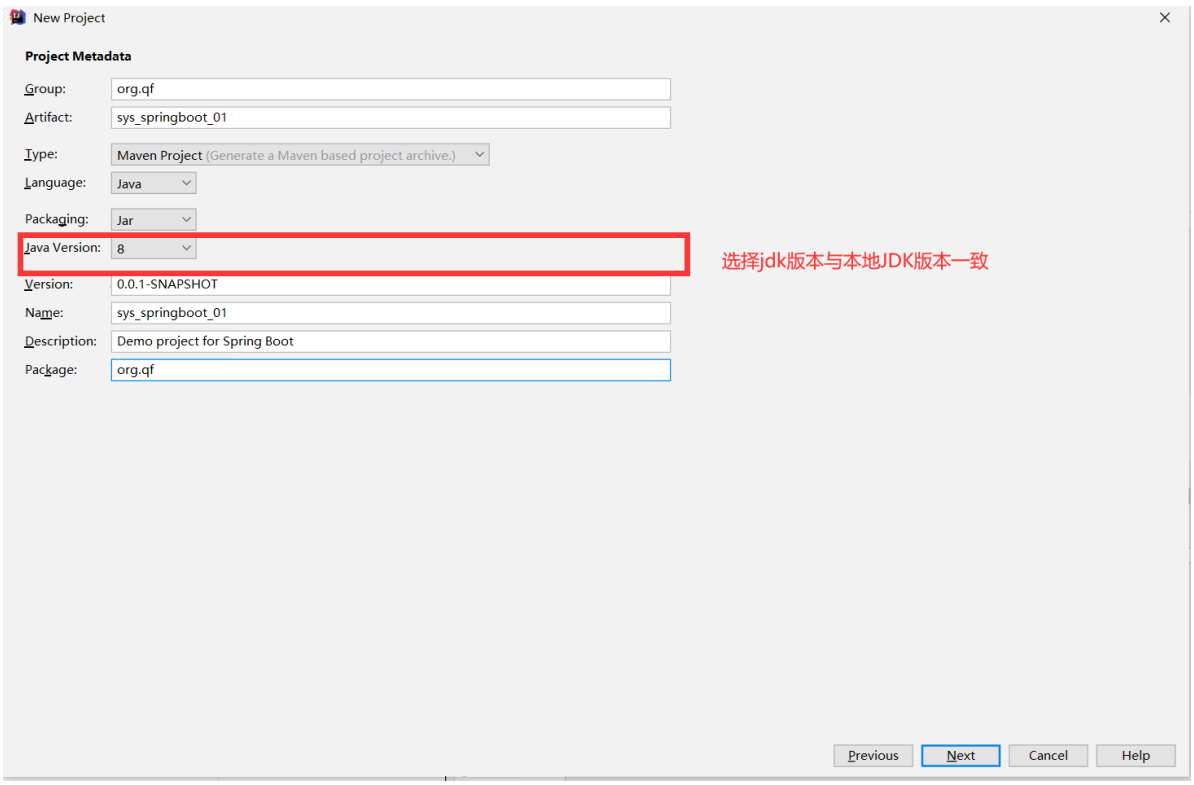
- 由于配置都是内置的，报错时定位比较困难
- 版本迭代更新，有些版本改动还是比较大。

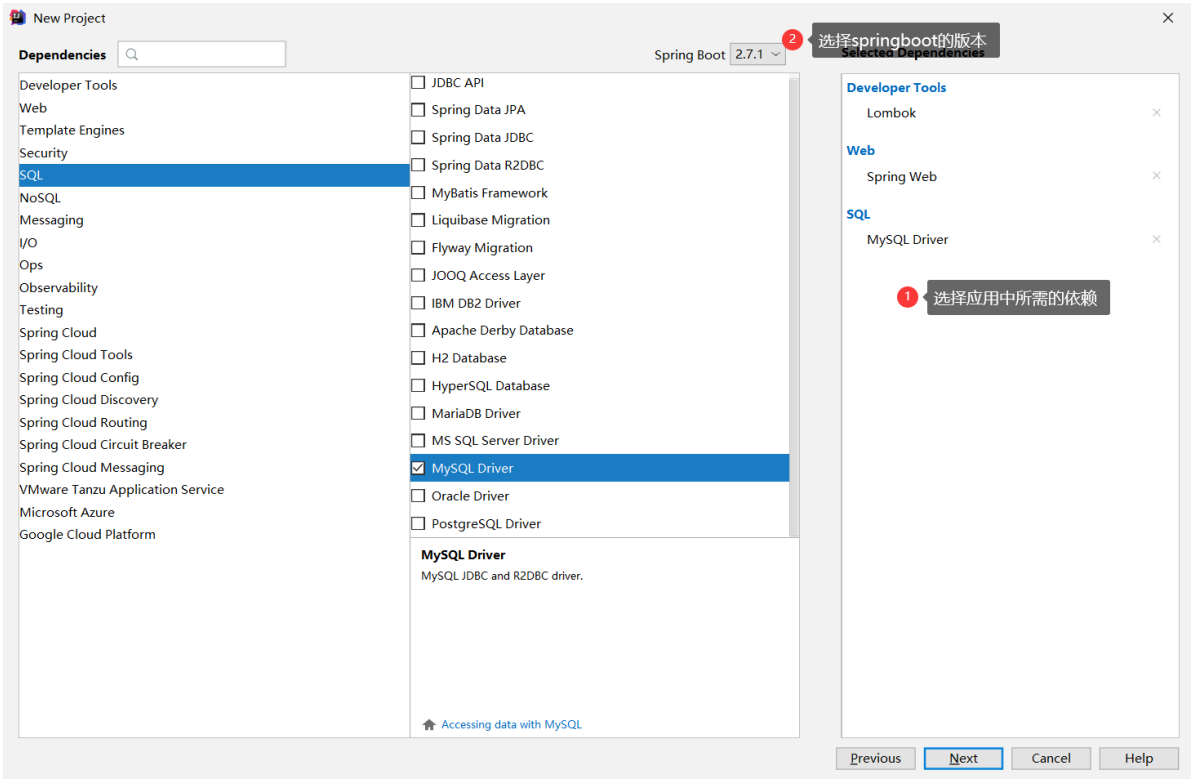
二、 第一个Springboot应用

> springboot应用需要依赖远程服务器进行创建。
>
> 远程服务器：
>
> - spring官网： **https://start.spring.io**
> - ali: **https://start.aliyun.com**

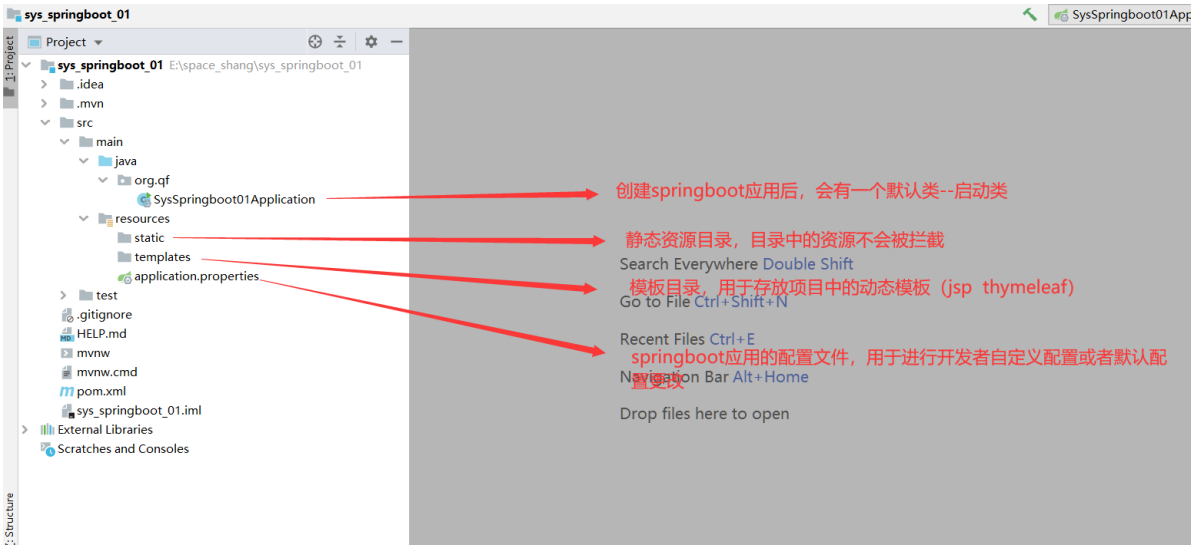
2.1 创建项目


New Project
Java
Java Enterprise
JBoss
Clouds
Spring
Java FX
Android
IntelliJ Platform Plugin
Spring Initializr
Maven
Gradle
Groovy
Grails
Application Forge
Kotlin
Static Web
Node.js and NPM
Flash
Empty Project
Project SDK:
1.8 (java version "1.8.0_92")
选择本地 jdk版本
New...
Choose Initializr Service URL.
Default: https://start.spring.io
Custom:
Make sure your network connection is active before continuing.
选择spring创建工程
Previous
Next
Cancel
Help


New Project
Project Metadata
Group: org.qf
Artifact: sys_springboot_01
Type: Maven Project (Generate a Maven based project archive.)
Language: Java
Packaging: Jar
Java Version: 8
Version: 0.0.1-SNAPSHOT
Name: sys_springboot_01
Description: Demo project for Spring Boot
Package: org.qf
选择jdk版本与本地JDK版本一致
Previous
Next
Cancel
Help



2.2 配置项目



2.3 整合MyBatis

> springboot帮助我们完成通用性配置，连接地址、用户名、密码等等

2.3.1 MyBatis介绍

> - 官网地址：
>
> https://mybatis.net.cn/
>
> MyBatis 是一款优秀的持久层框架，它支持自定义 SQL、存储过程以及高级映射。MyBatis 免除了几乎所有的 JDBC 代码以及设置参数和获取结果集的工作。MyBatis 可以通过简单的 XML 或注解来配置和映射原始类型、接口和 Java POJO（Plain Old Java Objects，普通老式 Java 对象）为数据库中的记录。

2.3.2 整合mybatis

- mysql的依赖驱动
- 在springboot主配置文件application.properties文件中配置数据源及路径

```
# 配置数据源（key必须按照springboot的要求）
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/db_test?serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=123456


# 配置映射文件路径及实体类的包名
mybatis.mapper-locations=classpath:mappers/*Mapper.xml
# 实体类
mybatis.type-aliases-package=org.qf.entity
```

- 在springboot启动类通过@MapperScan注解指定Dao接口的包名

```
package org.qf;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@MapperScan("org.qf.dao")
@SpringBootApplication
public class SysSpringboot01Application {

    public static void main(String[] args) {
        SpringApplication.run(SysSpringboot01Application.class, args);
    }

}
```

2.4 启动项目

> springboot应用自带servlet容器---Tomcat ,因此不需要配置tomcat，运行启动类即可启动一个springboot应用。

2.5 注册功能

2.5.1 实体类(Users)

```
package org.qf.entity;

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

/**
 * 实体类：  一张表对应一个实体类，把表映射为具体的类，类中的属性就是表中的字段名
 *
 * 封装
 */
@Data    //lombok 生成get和set方法
```

```
@AllArgsConstructor  // 有参构造函数
@NoArgsConstructor  //无参构造
public class Users {

    private Integer id;   //编号

    private String username;   //用户名

    private String password;  //密码

}
```

2.5.2 dao接口

```
package org.qf.dao;

import org.qf.entity.Users;
import org.springframework.stereotype.Repository;


/**
 * 用户接口 ： 操作数据库
 */
@Repository
public interface UsersMapper {

    /**
     * 注册功能
     * @param username
     * @param password
     * @return
     */
    public int regiter(String username,String password);
}
```

2.5.3 mapper映射文件

> 跟dao接口一一对应

```
<?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">
<!--

   namespace: 指向的是对应的接口
-->
<mapper namespace="org.qf.dao.UsersMapper">

    <!--注册-->
    <insert id="regiter">
        insert into users(username, password)values(#{username},#{password})
    </insert>

</mapper>
```

2.5.4 service(业务层结构)

```
package org.qf.service;

/**
 * 用户业务层： 处理业务功能，调用dao层接口
 */
public interface UsersService {

    /**
     * 注册功能
     * @param username
     * @param password
     * @return
     */
    public boolean register(String username,String password);



}
```

2.5.5 service(业务层实现类)

```
package org.qf.service.impl;

import org.qf.dao.UsersMapper;
import org.qf.entity.Users;
import org.qf.service.UsersService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service  //将这个类交给spring容器管理
public class UsersServiceImpl implements UsersService {

    @Autowired
    private UsersMapper usersMapper;   //注入dao层

    //方法重写
    @Override
    public boolean register(String username, String password) {
        int regiter = usersMapper.regiter(username, password);
        if(regiter>0){
            return true;
        }else{
            return false;
        }
    }


}
```

2.5.6 控制器

```
package org.qf.controller;

import org.qf.service.UsersService;
import org.springframework.stereotype.Controller;
```

```java
import org.springframework.web.bind.annotation.RequestMapping;

import javax.annotation.Resource;
import javax.annotation.Resources;

@Controller
public class UsersController {

   @Resource
    private UsersService usersService;


   @RequestMapping("/login")
    public String login(String username,String password){
       boolean register = usersService.register(username, password);
       if(register){
           System.out.print("注册成功!");
       }else{
           System.out.print("注册失败! ");
       }
       return "";

   }

}
```

三、 Springboot原理

3.1 starter

> 一个starter就是一个开发场景的支持（依赖+配置）
>
> - springboot为我们提供了简化企业级绝大多数场景的支持（提供了很多的starter），我们在进行项目开发的过程中只需要引入对应的starter（创建springboot应用时可选择），相关的依赖和配置就会内置到项目中

3.2 starter依赖

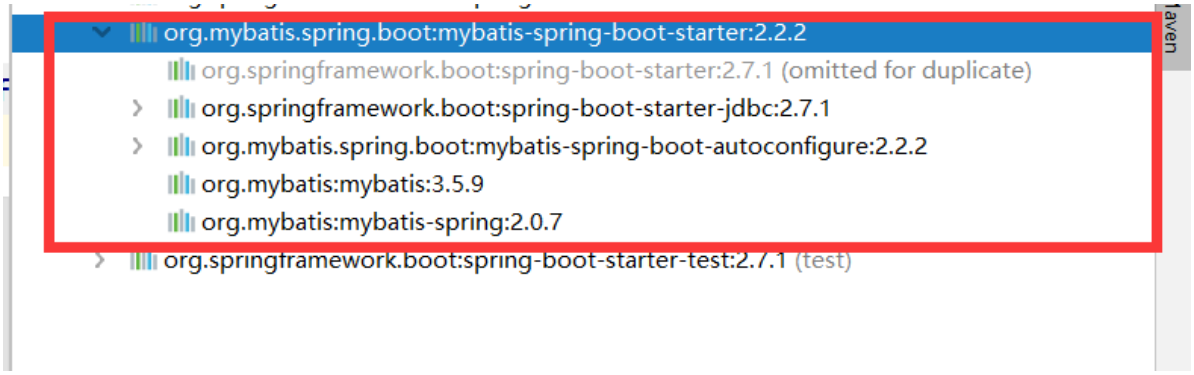
> 一个starter依赖表示的不是一个依赖，而是某种开发环境所需的一组依赖。

- Spring web--------springboot--- stater-web
- Mybatis----------springboot---stater--mybatis

3.3 starter配置

> 一个starter不仅包含所需依赖，还包含了其所需的对应的配置

- 依赖



- 配置:

```
public class MybatisAutoConfiguration implements InitializingBean {
    private static final Logger logger = LoggerFactory.getLogger(MybatisAutoConfiguration.class);
    private final MybatisProperties properties;
    private final Interceptor[] interceptors;
    private final TypeHandler[] typeHandlers;
    private final LanguageDriver[] languageDrivers;
    private final ResourceLoader resourceLoader;
    private final DatabaseIdProvider databaseIdProvider;
    private final List<ConfigurationCustomizer> configurationCustomizers;
    private final List<SqlSessionFactoryBeanCustomizer> sqlSessionFactoryBeanCustomizers;

    @Bean
    @ConditionalOnMissingBean
    public SqlSessionFactory sqlSessionFactory(DataSource dataSource) throws Exception {
        SqlSessionFactoryBean factory = new SqlSessionFactoryBean();
        factory.setDataSource(dataSource);
        factory.setVfs(SpringBootVFS.class);
        if (StringUtils.hasText(this.properties.getConfigLocation())) {
            factory.setConfigLocation(this.resourceLoader.getResource(this.properties.getConfigLocation()));
        }

        this.applyConfiguration(factory);
        if (this.properties.getConfigurationProperties() != null) {
            factory.setConfigurationProperties(this.properties.getConfigurationProperties());
        }
```

3.4 案例

> 引入redis场景

- 添加依赖

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

- 在service中可以直接注入redis客户端

```
import org.springframework.data.redis.core.StringRedisTemplate;
import org.springframework.stereotype.Service;

import javax.annotation.Resource;

@Service
public class UserServiceImpl {


    @Resource
    private StringRedisTemplate stringRedisTemplate;
}
```

3.5 springboot应用的pom文件

3.5.1 基于spring官方服务器创建的springboot应用

- 继承spring-boot-starter-parent.pom

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.7.1</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
```

- 继承spring-boot-dependencies 已经对主流的框架的版本进行了声明

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-dependencies</artifactId>
  <version>2.7.1</version>
</parent>
```

- 引入了maven对springboot应用支持的插件spirng-boot

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

3.5.2 基于ali服务器创建的Springboot应用

```
<?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
https://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.qf</groupId>
    <artifactId>sys_springboot_0714_02</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>sys_springboot_0714_02</name>
```

```
    <description>Demo project for Spring Boot</description>

    <properties>
        <java.version>1.8</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
        <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
        <!-- springboot引用的pom没有继承spring-boot-starter-parent.pom，因此版本需要在当前
pom中定义-->
        <spring-boot.version>2.3.7.RELEASE</spring-boot.version>
    </properties>
```

3.6 Java配置方式

> 如果我们需要在springboot应用中整合一种新的开发场景，只需在pom.xml中引入对应的starter即可。
>
> 一个starter不仅包含依赖、还包含相应的配置，starter中包含的配置都是通过Java类实现的-----Java配置方式

3.6.1 Spring版本发展

> 随着spring版本的迭代，配置方式也在方式变化

- spring 1.x
  - 所有的bean的配置只能通过xml方式配置
- spring 2.x
  - 基于JDK1.5对注解的支持，spirng2.x开发支持注解
  - 业务开发使用注解： @service @controller
- spring 3.x
  - Spring开始提供了基于Java的配置方式
- spring 4.x
  - xml、注解、Java

3.6.2 xml配置

```
 <bean id="users" class="org.qf.entity.Users">
        <property name="userId" value="1"></property>
        <property name="userName" value="aa"></property>
 </bean>
```

3.6.3 注解配置

```
@Component
public class Users {

}
```

3.6.4 Java配置方式

- 创建配置类

```
package org.qf.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import java.util.Date;

@Configuration
public class SpringConfig {

    @Bean
    public Date getDate(){
        return new Date();
    }
}
```

3.7 Springboot自动配置

```
 private static Map<String, List<String>> loadSpringFactories(ClassLoader classLoader)
{
        Map<String, List<String>> result = (Map)cache.get(classLoader);
        if (result != null) {
            return result;
        } else {
            HashMap result = new HashMap();

            //springboot基础依赖包含这个文件：springboot内置的自动配置类路径
            // 其他第三方starter也包含这个文件，包含第三方环境的自动配置类的路径
            try {
                Enumeration urls = classLoader.getResources("META-
INF/spring.factories");
 }
```

3.8 全局配置文件

> Springboot针对不同的开发场景提供默认的属性配置，如果默认的配置不能满足开发的需要，我们需要对属性配置进行修改。

- springboot应用提供了一个全局配置文件applicaiton.properties用于进行自定义配置
- 全局配置文件支持两种语法配置
    - properties键值对配置
    - yaml语法的配置

3.8.1 基于properties配置

```properties
# 应用名称
spring.application.name=sys_springboot_0714_02

# 数据源
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
spring.datasource.username=root
spring.datasource.password=123456

# 配置映射文件路径及实体类的包名
mybatis.mapper-locations=classpath:mappers/*Mapper.xml
# 实体类
mybatis.type-aliases-package=org.qf.entity
```

3.8.2 基于yaml配置

```yaml
# 数据源
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/db?characterEncoding=utf-8
    username: root
    password: 123456
mybatis:
# 配置映射文件路径及实体类的包名
  mapper-locations: classpath:mappers/*Mapper.xml
# 实体类
  type-aliases-package: org.qf.entity
```

3.8.3 常用的全局配置

```yaml
server:
  port: 8090
  servlet:
    context-path: /demo
```

3.9 自定义Banner

- 在springboot应用启动的时候是有一个默认启动图案的
- 这个默认图案支持自定义配置
  - 在 resources目录下创建 一个banner.txt文件
  - 在banner.txt文件自定义图案　http://patorjk.com/software/taag/

四、Springboot整合JSP

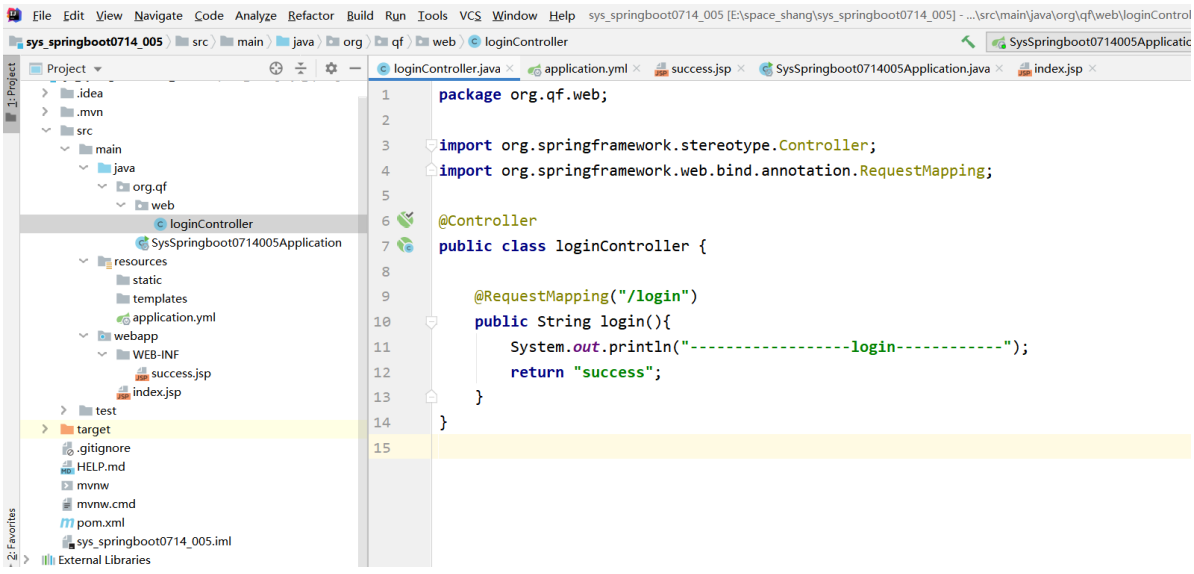
> springboot应用默认支持的动态网页技术是Thymeleaf ,并不支持JSP ,因此在springboot应用想要使用jsp需要通过手动整合来实现

4.1 导入依赖

```
        <!-- 添加servlet依赖模块 -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>javax.servlet-api</artifactId>
        </dependency>
        <!-- 添加jstl标签库依赖模块 -->
        <dependency>
            <groupId>javax.servlet</groupId>
            <artifactId>jstl</artifactId>
        </dependency>
        <!--添加tomcat依赖模块.-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-tomcat</artifactId>
        </dependency>
        <!-- 使用jsp引擎，springboot内置tomcat没有此依赖 -->
        <dependency>
            <groupId>org.apache.tomcat.embed</groupId>
            <artifactId>tomcat-embed-jasper</artifactId>
        </dependency>
```

4.2 创建jsp页面

- 修改pom文件 打包方式 为war
- 在main中创建webapp目录
- 在webapp创建.jsp页面

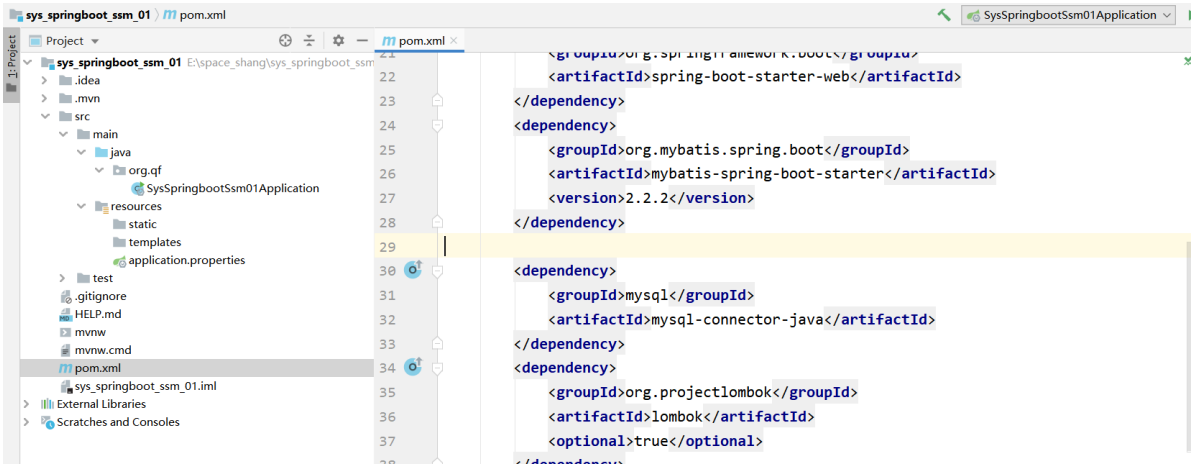


五、springboot整合SSM

5.1 创建springboot项目

- 创建项目时添加依赖
  - lombok
  - spring.web
  - mysql driver

- mybatis



5.2 进行mybatis所需的配置

- 将默认创建的application.properties改为application.yml
- 完成mybatis的自定义配置

```yaml
# 数据库的配置
spring:
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    url: jdbc:mysql://localhost:3306/db?serverTimezone=UTC
    username: root
    password: 123456

mybatis:
  type-aliases-package: org.qf.entitmappersy
  mapper-locations: classpath:/*Mapper.xml
```

5.3 在启动类配置DAO扫描

- @MapperScan

```java
package org.qf;

import org.mybatis.spring.annotation.MapperScan;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@MapperScan("org.qf.dao")
@SpringBootApplication
public class SysSpringbootSsm01Application {

    public static void main(String[] args) {
        SpringApplication.run(SysSpringbootSsm01Application.class, args);
    }

}
```

5.4 整合druid连接池

> 在springboot中整合mybatis的时候，默认继承自带的连接池，企业中的比较广泛是druid

5.4.1 添加依赖

```
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid-spring-boot-starter</artifactId>
    <version>1.2.11</version>
</dependency>
```

5.4.2 配置druid数据源

```
spring:
  datasource:
    druid:
      driver-class-name: com.mysql.cj.jdbc.Driver
      url: jdbc:mysql://localhost:3306/db?serverTimezone=UTC
      username: root
      password: 123456
```

六、 Thymeleaf

> Thymeleaf是一种类似于JSP的 动态网页技术

6.1 Thymeleaf简介

- JSP必须依赖Tomcat运行，不能直接运行在浏览器 中
- HTML可以直接运行在浏览器中，但是不能接收控制器传递的数据
- Thymeleaf是一种既保留了HTML的后缀能够直接在浏览器运行的能力，又实现了JSP显示动态数据的功能

6.2 Thymeleaf的使用

> springboot应用对thymeleaf提供了良好的支持

6.2.1 添加thymeleaf的依赖

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

6.2.2 创建thymeleaf模板

> thymeleaf模板就是HTML文件

- Springboot应用中resources/templates目录就是用来存放页面模板的
- 重要说明

    - static目录下的资源被定义为静态资源，Springboot默认放行
- templates目录下的文件 会被定义为动态网页或者模板，springboot会拦截templates中定义的资源，如果将HTML文件定义在templates目录下，则必须通过控制器跳转访问。
- templates目录创建HTML页面
- 创建 controller，用于转发跳转页面请求

6.3 Thymeleaf基本语法

> 如果要在thymeleaf模板中获取控制器传递的数据，需要使用th标签

6.3.1 在thymeleaf模板页面引入th标签的命名空间

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8">
    <title>登录页面</title>
</head>
<body>
    <h1>成功</h1>
</body>
</html>
```

6.3.2 th:text

> 在几乎所有的html双标签都可以使用th:text，将接收的数据显示在标签的内容中

```
<h2>欢迎<span th:text="${name}">xxx</span></h2>
```

6.3.3 th:object 和*

```
<div th:object="${u}">
      <p th:text="*{userId}"></p>
      <p th:text="*{userName}"></p>
      <p th:text="*{userPwd}"></p>
  </div>
```

6.3.4 th:each 循环

```
<tr th:each="u:${list}">
    <td th:text="${u.userId}"></td>
    <td th:text="${u.userName}"></td>
    <td th:text="${u.userPwd}"></td>
    <td th:text="${u.userImg}"></td>
</tr>
```

6.3.5 超链接传值

```
<a th:href="@{/userById(id=${u.userId})}">查看</a>
```