目录

Spring Boot

1 Spring Boot 概述

1.1 什么是 Spring Boot

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。简单的来说 Spring Boot 这个项目整合目前很多的流行的第三方框架,并且做了一系列的默认配置,我们无需在进行手动配置,直接拿过来直接使用！比如我们要使用 Spring mvc 开发 web 应用,那么我们可能需要经过以下几步

- 导入 spring mvc 的相关依赖包
- 在 web.xml 文件中配置 Spring mvc 的前端控制器
- 创建一个 spring mvc 的配置文件
- 在 spring mvc 的配置文件中进行相关配置
  1. 配置注解扫描路径
  2. 配置处理器映射器
  3. 配置处理器适配器
  4. 配置视图解析器
- 开发 Controller

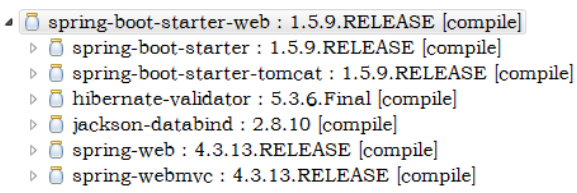
那么现在如果我们使用 Spring Boot 开发我们的 web 应用,那么具体的步骤如下:

- 导入相关的依赖包
- 开发 Controller

单单从开发步骤上讲都比我们的原始开发少了很多,其中的配置部分 Spring Boot 帮我们完成了,不需要我们在进行配置,当然如果我们想更改 Spring Boot 的默认配置也是可以的.极大的简化了我们的开发.

1.2 Spring Boot 的核心功能

- 独立运行的 spring 项目： Spring Boot 可以以 jar 包形式直接运行，如 java -jar xxx.jar 优点是：节省服务器资源
- 内嵌 servlet 容器: Spring Boot 可以选择内嵌 Tomcat，Jetty，这样我们无须以 war 包形式部署项目。
- 提供 starter 简化 Maven 配置: 在 Spring Boot 项目中为我们提供了很多的 spring-boot-starter-xxx 的项目(我们把这个依赖可以称之为起步依赖),我们导入指定的这些项目的坐标,就会自动导入和该模块相关的依赖包:
  例如我们后期再使用 Spring Boot 进行 web 开发我们就需要导入 spring-boot-starter-web 这个项目的依赖,导入这个依赖以后!那么 Spring Boot 就会自动导入 web 开发所需要的其他的依赖包,如下图所示:



- 自动配置 spring: Spring Boot 会根据在类路径中的 jar 包，类，为 jar 包里的类自动配置 Bean，这样会极大减少我们要使用的配置。当然 Spring Boot 只考虑了大部分开发场景，并不是所有的场景，如果在实际的开发中我们需要自动配置 Bean，而 Spring Boot 不能满足，则可以自定义自动配置。
- 准生产的应用监控: Spring Boot 提供基于 http，ssh，telnet 对运行时的项目进行监控
- 无代码生成和 xml 配置: Spring Boot 大量使用 spring4.x 提供的注解新特性来实现无代码生成和 xml 配置。spring4.x 提倡使用 Java 配置和注解配置组合，而 Spring Boot 不需要任何 xml 配置即可实现 spring 的所有配置。

1.3 Spring Boot 的优点和缺点

1.3.1 优点

- 快速构建项目
- 对主流框架无缝集成
- 项目可以独立运行，无需依赖外部 servlet 容器
- 提供运行时的应用监控
- 极大提高了开发，部署效率

1.3.2 缺点

- 资料相对比较少
- 版本迭代较快

2 Spring Boot 入门案例

使用 Spring Boot 开发 web 应用.

2.1 在 pom.xml 添加依赖

在 pom.xml 文件我们需要添加两部分的依赖:

- 让我们的项目继承 spring-boot-starter-parent 的工程
- 加入 spring-boot-starter-web 的依赖

```
<!-- 添加父工程 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>

<dependencies>
    <!-- 加入web开发的支持 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
</dependencies>
```

Spring Boot 项目默认的编译版本是 1.6,如果我们想使用 1.7 的编译版本我们就需要在 pom.xml 文件中定义一个变量

```
<!-- 定义变量 -->
<properties>
    <java.version>1.7</java.version>
</properties>
```

2.2 开发 Controller

```
package cn.itcast.spring.boot.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller
public class HelloSpringBootController {

    @RequestMapping(value = "/hello")
    @ResponseBody
    public String hello() {
        return "Hello Spring Boot" ;
    }

}
```

2.3 开发启动类

```
package cn.itcast.spring.boot.controller;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class HelloSpringBootApplication {
    public static void main(String[] args) {
        // 启动Spring Boot应用
        SpringApplication.run(HelloSpringBootApplication.class, args) ;
    }
}
```

2.4 启动应用

2.4.1 直接运行 main 方法进行启动

2.4.2 以 jar 包的方式运行

以 jar 包的方式运行我们需要导入一个 maven 的插件,然后在进行打包

```
<build>
    <plugins>
            <!-- Spring Boot的maven插件 -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
    </plugins>
</build>
```

通过入门案例的学习我们发现使用 Spring Boot 开发我们的 Spring 应用是非常简单的.那么同时很多同学可能存在一些疑问: 无 xml 的配置怎么实现的呢?自动化配置怎么实现的呢?

3 Spring Boot 核心

3.1 无 xml 配置的实现

自从 spring3.0 以后 spring 提供了很多的注解来替代 xml 文件的配置.最为核心的是下面的两个注解:

- @Configuration 标注该类是一个配置类.类似于我们定义的 applicationContext.xml
- @Bean 类似于我们在之前的 spring 配置文件中配置的<bean id="" class=""/>

有了上面的两个注解我们就可以使用编码的方式来完成 spring 的相关配置,接下来我们就来使用 java 编码的方式来完成 spring 的配置

3.1.1 入门 Demo

3.1.1.1 导入相关依赖

```
<properties>
      <spring.version>4.3.7.RELEASE</spring.version>
</properties>

<dependencies>

      <!-- Spring依赖包 -->
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-context</artifactId>
          <version>${spring.version}</version>
      </dependency>

      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-beans</artifactId>
          <version>${spring.version}</version>
      </dependency>

      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-context-support</artifactId>
          <version>${spring.version}</version>
      </dependency>

</dependencies>
```

```
<build>
       <plugins>
          <!-- maven的编译插件 -->
          <plugin>
             <groupId>org.apache.maven.plugins</groupId>
             <artifactId>maven-compiler-plugin</artifactId>
             <version>2.3.2</version>
             <configuration>
                <source>1.7</source>
                <target>1.7</target>
             </configuration>
          </plugin>
       </plugins>
</build>
```

3.1.1.2 编写 HelloService

```
package com.itheima.javaconfig.service;

public class HelloService {

    public String sayHello() {
        return "Hello JavaConfig" ;
    }

}
```

3.1.1.3 编写配置类

```
package com.itheima.javaconfig.service;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration  // 相当于我们定义的配置文件
public class ApplicationConfiguration {
    @Bean        // 相当于我们在配置文件中定义了一个bean
    public HelloService helloService() {
        return new HelloService() ;
    }
}
```

3.1.1.4 编写测试类

```
package com.itheima.javaconfig.service;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class JavaConfigTest {

    public static void main(String[] args) {

    // 通过AnnotationConfigApplicationContext这个类获取Spring容器
    AnnotationConfigApplicationContext applicationContext = new
  AnnotationConfigApplicationContext(ApplicationConfiguration.class) ;
        HelloService helloService = applicationContext.getBean(HelloService.class) ;
        String result = helloService.sayHello() ;
        System.out.println(result);

    }

}
```

这个时候我们应该就明白了 Spring Boot 项目之所以能做到不需要 xml 文件,是因为它使用了这两个注解替换了之前了 xml 文件的配置.

3.1.2 练习(配置 JdbcTemplate)

需求使用 JdbcTemplate 查询所有的用户数据
导入 user 脚本文件

3.1.2.1 pom.xml 文件的配置

在入门小 Demo 的 pom.xml 文件的配置基础之上加入如下依赖:

```
<properties>
    <druid.version>1.1.6</druid.version>
    <mysql.version>5.1.44</mysql.version>
</properties>

<!-- 加入spring-jdbc的依赖包 -->
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-jdbc</artifactId>
    <version>${spring.version}</version>
</dependency>

<!-- 加入druid的数据源 -->
<dependency>
    <groupId>com.alibaba</groupId>
    <artifactId>druid</artifactId>
    <version>${druid.version}</version>
</dependency>

<!-- 加入mysql的驱动 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>${mysql.version}</version>
</dependency>
```

3.1.2.2 db.properties 文件配置

```
jdbcDriverClassName=com.mysql.jdbc.Driver
jdbcUrl=jdbc:mysql://localhost:3306/spring-boot
jdbcUserName=root
jdbcPassword=1234
```

3.1.2.3 编写实体类

```
private Integer id ;              // 唯一标识
private String userName ;         // 用户名
private String sex ;              // 性别
private String address ;          // 住址
```

3.1.2.4 编写配置类

```
package com.itheima.config;

import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.PropertySource;
import org.springframework.jdbc.core.JdbcTemplate;

import com.alibaba.druid.pool.DruidDataSource;
```

```
@Configuration                                          // 标注该类是一个配置类
@PropertySource(value = {"classpath:db.properties"}) // 加载配置文件
public class JdbcTemplateConfiguration {

    @Value("${jdbcDriverClassName}")
    private String driverClassName ;

    @Value("${jdbcUrl}")
    private String jdbcUrl ;

    @Value("${jdbcUserName}")
    private String jdbcUserName ;

    @Value("${jdbcPassword}")
    private String jdbcPassword ;

    /**
     * 配置数据源
     * @return
     */
    @Bean
    public DataSource dataSource() {
        DruidDataSource dataSource = new DruidDataSource() ;
        dataSource.setDriverClassName(driverClassName);
        dataSource.setUrl(jdbcUrl);
        dataSource.setUsername(jdbcUserName);
        dataSource.setPassword(jdbcPassword);
        return dataSource ;
    }

    @Bean
    public JdbcTemplate jdbcTemplate(DataSource dataSource) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource) ;
        return jdbcTemplate ;
    }

}
```

3.1.2.5 编写测试类

```
package com.itheima.config;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

import org.springframework.context.annotation.AnnotationConfigApplicationContext;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import com.itheima.config.domain.User;

public class JdbcTemplateJavaConfigTest {

    public static void main(String[] args) {

        // 获取spring容器
        AnnotationConfigApplicationContext applicationContext =
                                        new
AnnotationConfigApplicationContext(JdbcTemplateConfiguration.class);

        // 获取JdbcTemplate
        JdbcTemplate jdbcTemplate = applicationContext.getBean(JdbcTemplate.class);

        // 进行查询
        List<User> users = jdbcTemplate.query("select * from user", new RowMapper<User>(){
```

```
            @Override
            public User mapRow(ResultSet rs, int num) throws SQLException {

                User user = new User() ;
                user.setId(rs.getInt("id"));
                user.setUserName(rs.getString("username"));
                user.setSex(rs.getString("sex"));
                user.setAddress(rs.getString("address"));

                return user;
            }

        }) ;

        // 输出
        System.out.println(users);

    }

}
```

通过刚才的练习我们应该对 java 配置的方式有一个更加深刻的认识了.

3.2 自动化配置实现

我们刚才在编写入门案例的时候我们使用的是 Spring mvc 作为我们的表现层框架,但是我们都知道我们要使用 Spring mvc 我们就需要在 web.xml 文件中配置 Spring mvc 的前端控制器(DispatcherServlet). 但是我们刚才在编写入门案例的时候我们并没有去做任何的配置.那么我们为什么可以使用呢?那是因为 Spring Boot 给我们做了自动配置.那么接下来我们就来讲解一下 Spring Boot 中自动配置的原理.

3.2.1 入口类和@SpringBootApplication

Spring Boot 的项目一般都会有*Application 的入口类，入口类中会有 main 方法，这是一个标准的 Java 应用程序的入口方法。Spring Boot 会自动扫描@SpringBootApplication 所在类的同级包以及下级包中的 bean(如果是 jpa 项目还会自动扫描标注@Entity 的实体类)
@SpringBootApplication 注解是 Spring Boot 的核心注解:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class),
        @Filter(type = FilterType.CUSTOM, classes = AutoConfigurationExcludeFilter.class) })
public @interface SpringBootApplication {
```

通过源码我们发现在@SpringBootApplication 这个注解上又定义了一些其他的注解如：@SpringBootConfiguration @EnableAutoConfiguration @ComponentScan 我们把这样的注解称之为组合注解.组合注解作用就是简化我们的注解使用：我们在某一个类上使用了@SpringBootApplication 那么就相当于在该类上使用了该注解上定义的其他的三个注解

3.2.1.1 @SpringBootConfiguration

@SpringBootConfiguration 这是 Spring Boot 项目的配置注解，源码如下:

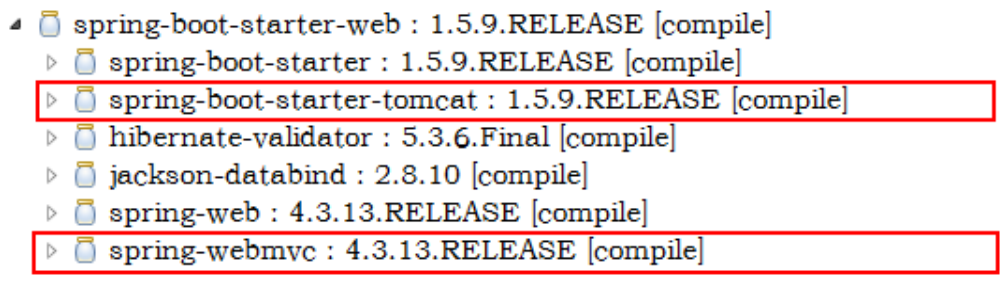
```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Configuration
public @interface SpringBootConfiguration {

}
```

这个注解也是一个组合注解，在 Spring Boot 项目中推荐使用@ SpringBootConfiguration 替代@Configuration

3.2.1.2@EnableAutoConfiguration

启用自动配置，该注解会使 Spring Boot 根据项目中类路径依赖的 jar 包自动配置项目的配置项：例如：我们添加了 spring-boot-starter-web 依赖,会自动添加会自动添加 tomcat 和 springmvc 的依赖



那么 spring boot 项目会自动为我们配置 tomcat 和 springmvc; 具体体现在 tomcat 的配置使用的是 8080 端口,Spring mvc 请求处理方式为/

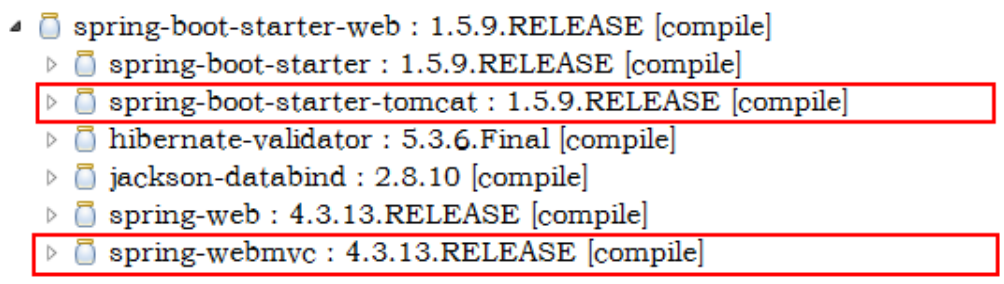
3.2.1.3@ComponentScan

默认扫描@SpringBootApplication 所在类的同级目录以及它的子目录。

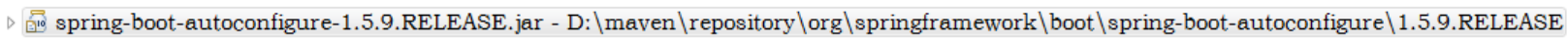
3.2.2 DispatcherServlet 自动配置分析

3.2.2.1DispatcherServlet 自动配置源码解析

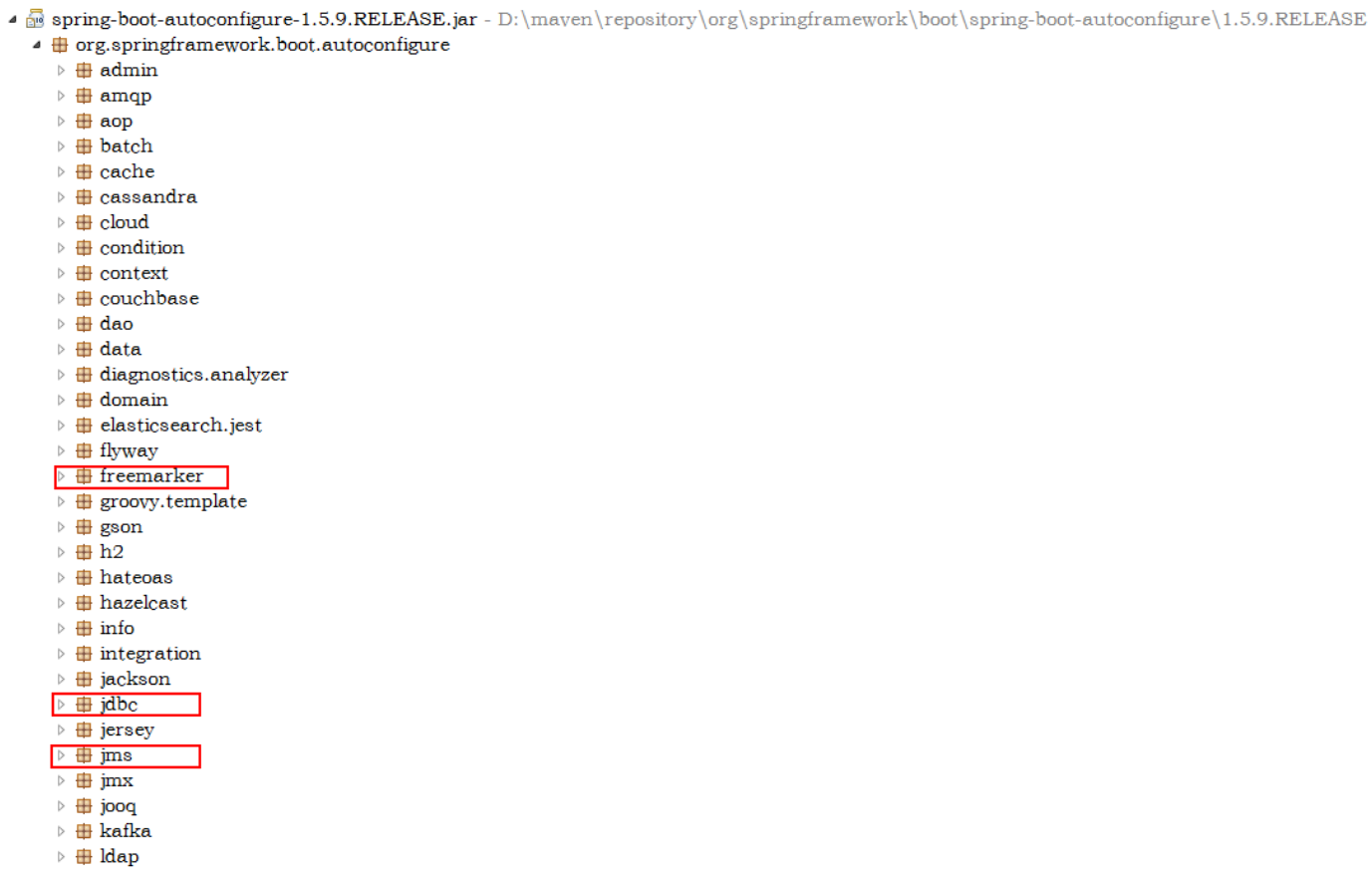
入门案例中我们只是加入了 spring-boot-starter-web 这么一个依赖,就可以实现 DispatcherServlet 的相关配置,这是为什么呢?注意当我们加入了 spring-boot-starter-web 的依赖以后,根据 maven 的依赖传递特性,会自动将一些依赖包加入到我们的项目中; 比如会自动添加 tomcat 和 springmvc 的依赖:



那么这些依赖包的确加入到了我们的项目中,那么怎么完成的自动配置呢? 因为在加入 spring-boot-starter-web 依赖的时候,会自动将另外一个依赖加入进来: 这个依赖包就是



这个包中其实定义了很多技术点的自动配置:



那么和我们 web 开发相关的自动配置类是在 web 包下,在这个 web 包中定义了很多的 xxxAutoConfiguration 这样的类,这些类其实就是用来完成自动配置的; 我们不妨打开 DispatcherServletAutoConfiguration 看看

```
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@Configuration
@ConditionalOnWebApplication
@ConditionalOnClass(DispatcherServlet.class)
@AutoConfigureAfter(EmbeddedServletContainerAutoConfiguration.class)
public class DispatcherServletAutoConfiguration {
```

上面的是关于 DispatcherServletAutoConfiguration 的定义,通过源码我们发现在这个类上定义了很多的好几个注解:

- @AutoConfigureOrder: 这个注解自动配置的顺序定义,取值为 int 的最小值,优先级最高
- @Configuration: 这个表示该类是一个配置类,类似于我们定义了一个 applicationContext.xml 文件
- @ConditionalOnWebApplication: 这是一个条件注解,当前项目是 web 环境的条件下
- @ ConditionalOnClass: 当前类路径下有指定的类的条件下
- @ AutoConfigureAfter: 定义该配置类的载入顺序;该类上表示(后于 EmbeddedServletContainerAutoConfiguration)载入

其实看到这里我们就可以明白,Spring Boot 中的自动配置其实使用的就是这些条件注解来完成的,当满足某一些添加以后就可以启动自

动配置

3.2.2.2常见的条件注解

@ConditionalOnBean	当容器中存在指定 bean 的条件下
@ConditionalOnClass	当类路径下存在指定类的条件下
@ConditionalOnMissingBean	当容器中不存在指定 bean 的条件下
@ConditionalOnMissingClass	当类路径下不存在指定类的条件下
@ConditionalOnProperty	指定的属性是否存在指定的值
@ConditionalOnResource	类路径下是否有指定的值
@ConditionalOnWebApplication	是 web 环境的条件下下
@ConditionalOnNotWebApplication	不是 web 环境的条件下

3.3 自定义自动配置

需求: 自定义自动配置类,完成 HelloService 的自动配置(itheima-spring-starter-hello)

3.3.1 编写 HelloService

```
package com.itheima.spring.starter.service;

public class HelloService {

    private final String DEFAULT_SAYWORD = "hello" ;

    public String sayHello(String sayWord) {
        if(sayWord != null && !sayWord.trim().equals("")) {
            return sayWord ;
        }
        return DEFAULT_SAYWORD ;
    }

}
```

3.3.2 pom.xml 文件

```
<dependencies>

    <!-- 引入springboot自动配置的jar包 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-autoconfigure</artifactId>
        <version>1.5.9.RELEASE</version>
    </dependency>

</dependencies>
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
            <version>2.3.2</version>
            <configuration>
                <source>1.7</source>
                <target>1.7</target>
            </configuration>
        </plugin>
    </plugins>
</build>
```

3.3.3 编写配置类

```
package com.itheima.spring.starter.autoconfig;

import org.springframework.boot.autoconfigure.condition.ConditionalOnClass;
import org.springframework.boot.autoconfigure.condition.ConditionalOnMissingBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import com.itheima.spring.starter.service.HelloService;

@Configuration
@ConditionalOnClass(HelloService.class)
public class HelloServiceAutoConfig {

    @Bean
    @ConditionalOnMissingBean(HelloService.class)
    public HelloService helloStaterService() {
        return new HelloService() ;
    }
}
```

3.3.4 注册自动配置类

在 src/main/resources 目录下创建一个: META-INF 这个文件夹,然后在该文件夹下创建一个文件: spring.factories; 内容如下

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.itheima.spring.starter.autoconfig.HelloServiceAutoConfig
```

Spring boot 项目在运行的时候会读取这个配置文件中的自动配置,来完成自动配置

3.3.5 编写测试工程

3.3.5.1 pom.xml 文件

```
<!-- 定义变量 -->
<properties>
    <java.version>1.7</java.version>
</properties>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>

<dependencies>

    <!-- 导入spring-boot-starter -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>

    <!-- 导入spring boot 测试的依赖包 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <!-- 加入itcast-spring-starter-hello依赖 -->
    <dependency>
        <groupId>com.itheima</groupId>
        <artifactId>itheima-spring-starter-hello</artifactId>
        <version>0.0.1-SNAPSHOT</version>
    </dependency>

</dependencies>
```

3.3.5.2 工程测试

3.3.5.2.1 通过入口类进行测试

```
package com.itheima.autoconfig;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.ConfigurableApplicationContext;

import com.itheima.spring.starter.service.HelloService;

@SpringBootApplication
public class HelloStarterServiceTest {

    public static void main(String[] args) {

        // 获取 Spring 容器
        ConfigurableApplicationContext applicationContext = SpringApplication.run(HelloStarterServiceTest.class, args) ;
        HelloService helloStaterService = applicationContext.getBean(HelloService.class) ;
        String sayHello = helloStaterService.sayHello(null) ;
        System.out.println(sayHello);

    }

}
```

3.3.5.2.2 使用 Junit 进行测试

```
package com.itheima.autoconfig;

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.SpringJUnit4ClassRunner;

import com.itheima.spring.starter.service.HelloService;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = HelloStarterServiceTest.class)
public class ApplicationTest {

    @Autowired
    private HelloService helloService ;

    @Test
    public void testAutoConfig() {
        String sayHello = helloService.sayHello(null) ;
        System.out.println(sayHello);
    }

}
```

3.4 关闭自动配置

如果我们不需要 Spring Boot 自动配置，想关闭某一项的自动配置，该如何设置呢？比如我们不想要 Spring Boot 为我们配置的 DispatcherServlet,如何完成呢?@SpringBootApplication(exclude = {DispatcherServletAutoConfiguration.class})

3.5 更改默认的配置

Spring Boot 使用一个全局的配置文件,这个全局配置文件为 application.properties 或者 application.yml; 放置在 src/main/resources 目录下,这个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
比如我们可以更改 tomcat 的端口号: server.port=8088

我们也可以更改 springmvc 的拦截规则: server.servlet-path=*.do
测试: http://localhost:8088/hello.do
更多的配置参见资料(Spring Boot 常用配置.docx)

3.6 Starters

Spring Boot 为我们提供了简化企业级开发绝大多数场景的 starter pom,只有使用了应用场景所需要的 starter pom,那么相关的技术配置将会生效,我们就可以直接使用 Spring Boot 为我们提供的自动配置的 bean; Spring Boot 官方为我们提供的 starter pom 文件有: https://docs.spring.io/spring-boot/docs/1.5.9.RELEASE/reference/htmlsingle/#using-boot-starter

spring-boot-starter-data-jpa	Starter for using Spring Data JPA with Hibernate	Pom
spring-boot-starter-data-ldap	Starter for using Spring Data LDAP	Pom
spring-boot-starter-data-mongodb	Starter for using MongoDB document-oriented database and Spring Data MongoDB	Pom
spring-boot-starter-data-neo4j	Starter for using Neo4j graph database and Spring Data Neo4j	Pom
spring-boot-starter-data-redis	Starter for using Redis key-value data store with Spring Data Redis and the Jedis client	Pom
spring-boot-starter-data-rest	Starter for exposing Spring Data repositories over REST using Spring Data REST	Pom
spring-boot-starter-data-solr	Starter for using the Apache Solr search platform with Spring Data Solr	Pom
spring-boot-starter-freemarker	Starter for building MVC web applications using FreeMarker views	Pom
spring-boot-starter-groovy-templates	Starter for building MVC web applications using Groovy Templates views	Pom

3.7 @ImportSource

Spring boot 不建议我们使用配置文件的方式来完成配置,但是有的情况下我们不得不需要使用配置文件,那么我们就可以使用 @ImportSource 注解来加载指定的配置文件

4 Web 开发

需求: 使用 Spring Boot 完成用户信息的查询, 页面使用 bootstrap 进行渲染

4.1 静态资源的访问

在我们 web 开发中,存在很多的一些 css，js，图片等等一些静态的资源文件!那么我们应该把这些静态资源存储在什么目录下呢?Spring Boot 项目要求我们将这个静态资源文件存储到 resources 目录下的 static 目录中; 当然这个配置是可以进行更改的,但是不建议更改!在进行访问的时候我们不需要添加上 static 文件目录,Spring Boot 会自动在 static 目录中查找对应的资源文件.

4.2 模板引擎(freemarker)

Spring Boot 项目建议我们使用模板引擎来进行页面视图的渲染,而不建议我们使用 jsp! 因此内嵌的 tomcat 也没有提供 jsp 页面的支持.Spring Boot 提供了大量的模板引擎,包含: Freemarker , Groovy , Thymeleaf , Velocity 和 Mustache. 本次我们就来讲解一下 Spring Boot 继承 freemarker 作为页面视图!

4.2.1 引入指定的依赖包

```
<!-- 加入 spring-boot-starter-freemarker -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-freemarker</artifactId>
</dependency>
```

4.2.2 创建实体类

```
private Integer id ;              // 唯一标识
private String userName ;         // 用户名
private String sex ;              // 性别
private String address ;          // 住址
```

4.2.3 编写模板文件

Spring Boot 项目为我们提供了一个 templates 目录专门用来存储模板文件的. 因此我们需要将指定的模板文件放在该目录下! 模板文件的后缀名默认是 ftl.

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>欢迎登录</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css" />
        <link rel="stylesheet" href="css/bootstrap.min.css" />
        <script type="text/javascript" src="js/jquery-1.11.3.min.js" ></script>
        <script type="text/javascript" src="js/bootstrap.min.js" ></script>
    </head>

    <body class="container">

        <center>
            <h1>用户信息列表</h1>
        </center>
        <table class="table">

            <tr>
                <th>用户id</th>
                <th>名称</th>
                <th>用户性别</th>
                <th>地址</th>
                <th>操作</th>
            </tr>

            <#list users as user>
                <tr>
                    <th>${user.id}</th>
                    <th>${user.userName}</th>
                    <th>${user.sex}</th>
                    <th>${user.address}</th>
                    <th><a href="javascript:void(0);">删除</a></th>
                </tr>
            </#list>

        </table>

    </body>
</html>
```

4.2.4 编写 Controller

学习过 freemarker 的程序员,都知道!freemarker 的原理就是模板 + 数据 =>>> 输出结果! 而我们使用 Spring Boot 我们只需要将数据存储到 Model 对象中.然后返回到结果页面!Spring Boot 会自动的将数据填充到模板完成视图的渲染!

```
package com.itheima.spring.controller;

import java.util.ArrayList;
import java.util.List;

import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

import com.itheima.spring.domain.User;

@Controller
public class UserController {

    @RequestMapping(value = "/findAllUser")
    public String findAllUser(Model model) {

        // 准备数据
        List<User> users = new ArrayList<User>() ;
        users.add(new User(1 , "张三" , "1" , "北京")) ;
```

```
        users.add(new User(2 , "李四" , "2" , "西安")) ;
        users.add(new User(3 , "王五" , "2" , "草滩六路")) ;
        users.add(new User(4 , "赵六" , "1" , "上海")) ;

        // 把数据存储到Model中
        model.addAttribute("users", users) ;

        // 返回逻辑视图名称
        return "user" ;
    }

}
```

4.3 项目热部署

在进行项目开发阶段,我们需要频繁的修改代码,来进行项目的测试!每一次进行项目代码修改的时候,我们都需要进行项目重新启动,这样新添加的代码才可以生效!这种做法比较麻烦.我们就可以使用 Spring Boot 为我们提供的: spring-boot-devtools,来进行项目的热部署!我们只需要引入对应的依赖包即可!

```
<!-- 热部署配置 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
</dependency>
```

4.4 全局异常处理

在进行 web 开发的时候,我们都需要对异常进行处理!Spring mvc 中提供了一个全局异常处理器 HandlerExceptionResolver 对象对异常进行统一的处理! 那么在 Spring Boot 中如何进行异常的统一处理呢? 在 Spring Boot 中给我们提供了两个注解,可以用来完成异常的处理!

- @ControllerAdvice是 controller 的一个辅助类，最常用的就是作为全局异常处理的切面类
- @ExceptionHandler 捕获@RequestMapping 注解的方法抛出的异常，我们可以通过该注解实现自定义异常处理, value 指定需要拦截的异常类型

那么关于方法的返回值取值可以有一下几种:

- 返回 ModelAndView 对象
- 返回 Model 但是需要使用@ResponseBody 注解将其转化成 Json 字符串
- 返回 String 表示返回一个逻辑视图

4.4.1 添加方法

```
@RequestMapping(value = "/findUser/{num}")
@ResponseBody
public String findUser(@PathVariable(value = "num") int num) {

    if(num == 1) {
        return "Hello User " ;
    }else {
        throw new RuntimeException("not find userinfo ..............") ;
    }

}
```

4.4.2 异常处理类

```
package com.itheima.spring.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.bind.annotation.ExceptionHandler;
import org.springframework.web.bind.annotation.ResponseBody;

@ControllerAdvice
public class UserExceptionHandler {

    @ExceptionHandler
    @ResponseBody
```

```
    public String handlerException(Exception exception) {
        System.out.println(exception.getMessage());
        return "异常被UserExceptionHandler处理了....." ;
    }
}
```

4.5 文件上传

4.5.1 模板文件(file-upload.ftl)

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
        <title>用户添加</title>
        <meta name="viewport" content="width=device-width, initial-scale=1">
        <link rel="stylesheet" href="css/bootstrap-theme.min.css" />
        <link rel="stylesheet" href="css/bootstrap.min.css" />
        <script type="text/javascript" src="js/jquery-1.11.3.min.js" ></script>
        <script type="text/javascript" src="js/bootstrap.min.js" ></script>
    </head>

    <body class="container">
        <center>
            <h1>文件上传表单</h1>
        </center>
        <form role="form" action="/fileUpload" enctype="multipart/form-data" method="post">
             <div class="form-group">
               <label for="inputfile">文件输入</label>
               <input type="file" id="inputfile" name="uploadFile">
             </div>
            <button type="submit" class="btn btn-default">提交</button>
        </form>
    </body>
</html>
```

4.5.2 添加方法

```
@RequestMapping(value = "/{path}")
public String path2(@PathVariable(value = "path")String path) {
    return path ;
}
```

4.5.3 上传文件 Controller

```
package com.itheima.spring.controller;

import java.io.File;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.multipart.MultipartFile;

@Controller
public class UploadController {

    @RequestMapping(value = "/fileUpload")
    public String fileUpload(MultipartFile uploadFile) {

        try {
            File file = new File("D:\\images\\" + uploadFile.getOriginalFilename()) ;
            uploadFile.transferTo(file);
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }

        return "success" ;
    }

}
```

4.6 拦截器

拦截器对使用 Spring mvc、Struts 的开发人员来说特别熟悉，因为你只要想去做好一个项目必然会用到它。拦截器在我们平时的项目中用处有很多，如：日志记录、用户登录状态拦截、安全拦截等等。而 Spring Boot 内部集成的是 Spring mvc 控制框架，所以使用起来跟 SpringMVC 没有区别，只是在配置上有点不同。

需求: 当用户请求以/file-upload 这个路径时候,需要验证用户是否登录如果没有登录,跳转到登录页面,让用户进行登录!如果用户已经登录直接放行!

4.6.1 编写拦截器

```
package com.itheima.spring.interceptor;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.HandlerInterceptor;
import org.springframework.web.servlet.ModelAndView;

public class CheckUserLoginInterceptor implements HandlerInterceptor {

    @Override
    public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object
handler) throws Exception {

        String userName = (String) request.getSession().getAttribute("user") ;
        if(userName == null || userName.equals("")) {
            response.sendRedirect("/login");
            return false ;
        }else {
            return true;
        }

    }

    @Override
    public void postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,
ModelAndView modelAndView) throws Exception {

    }

    @Override
    public void afterCompletion(HttpServletRequest request, HttpServletResponse response, Object
handler, Exception ex) throws Exception {

    }

}
```

4.6.2 编写登录页面

```
<!DOCTYPE html>
<html>
    <head>
        <meta charset="UTF-8">
```

```
    <title>用户添加</title>
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="stylesheet" href="css/bootstrap-theme.min.css" />
    <link rel="stylesheet" href="css/bootstrap.min.css" />
    <script type="text/javascript" src="js/jquery-1.11.3.min.js" ></script>
    <script type="text/javascript" src="js/bootstrap.min.js" ></script>
</head>

<body class="container">
    <center>
        <h1>用户登录</h1>
    </center>
    <form role="form" action="/userLogin">
         <div class="form-group">
           <label for="inputfile">用户名</label>
           <input type="text" id="userName" name="userName">
         </div>
         <div class="form-group">
           <label for="inputfile">密码</label>
           <input type="password" id="password" name="password">
         </div>
        <button type="submit" class="btn btn-default">登录</button>
    </form>
</body>
</html>
```

4.6.3 编写登录的 Controller

```
package com.itheima.spring.controller;

import javax.servlet.http.HttpSession;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class LoginController {

    @RequestMapping(value = "/userLogin")
    public String login(String userName , String password , HttpSession session) {
        System.out.println("userName====>>>" + userName + ", password ======>>>>" + password);
        session.setAttribute("user", userName);
        return "redirect:/file-upload" ;
    }

}
```

4.6.4 配置拦截器

```
package com.itheima.spring.interceptor.config;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;

import com.itheima.spring.interceptor.CheckUserLoginInterceptor;

@Configuration
public class InterceptorConfig extends WebMvcConfigurerAdapter {

    @Override
    public void addInterceptors(InterceptorRegistry registry) {
        registry.addInterceptor(new CheckUserLoginInterceptor()).addPathPatterns("/file-upload") ;
    }

}
```

4.7 Jsp 页面的支持(了解)

Spring Boot 不建议我们使用 jsp 页面作为视图的显示层,因此在 Spring Boot 中默认是不支持 jsp 页面的.如果我们还想使用 jsp 页面就需要添加 jsp 页面的支持!

```
<!-- jsp的支持 -->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
    <scope>provided</scope>
</dependency>
```

5 Dao 开发

关于持久层的开发存在很多的技术,我们可以直接使用 Spring 给我们提供的 JdbcTemplate,也可以使用一些第三方的框架,比如 Hibernate 或者 Mybatis.我们本次选定的是 mybatis 进行开发.因此我们所要讲解的知识点是 Spring Boot 和 mybatis 的整合.那么 Spring Boot 和 Mybatis 的整合存在两种方式:

- 第一种：使用 Mybatis 官方提供的 Spring Boot 整合包实现，地址：https://github.com/mybatis/spring-boot-starter
- 第二种：使用 mybatis-spring 整合的方式，也就是我们传统的方式

我们本次选地的是第二种方式,因为第一种方式对于我们后期程序的扩展不是很方便.
需求: 查询所有的用户信息

5.1 pom.xml 文件

```
<!-- 定义变量 -->
<properties>
    <java.version>1.7</java.version>
</properties>

<!-- 添加父工程 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>

<dependencies>

    <!-- 加入spring boot 对jdbc的支持 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jdbc</artifactId>
    </dependency>

    <!-- 配置mybatis 整合 spring 所需要的依赖包 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis-spring</artifactId>
        <version>1.3.1</version>
    </dependency>

    <!-- mybatis的开发包 -->
    <dependency>
        <groupId>org.mybatis</groupId>
        <artifactId>mybatis</artifactId>
        <version>3.2.8</version>
    </dependency>

    <!-- spring boot的测试包 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
```

```
    <!-- 引入mysql的驱动 -->
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>

    <!-- 配置druid数据源 -->
    <dependency>
        <groupId>com.alibaba</groupId>
        <artifactId>druid-spring-boot-starter</artifactId>
        <version>1.1.6</version>
    </dependency>

</dependencies>

<build>
    <plugins>
        <!-- Spring Boot的maven的打包插件 -->
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
</build>
```

5.2 整合配置

我们本次使用的是 Mybatis 的代理开发(所谓的代理开发就是我们程序员必须要编写 Dao 接口,不需要编写实现类,实现类有 Mybatis 通过代理模式给我们生成).那么要使用 Mybatis 的代理开发,我们就需要配置两个东西一个是 SqlSessionFactoryBean 另外一个就是 MapperScannerConfigurer. 那么接下来我们就来配置一下

5.2.1 SqlSessionFactoryBean 配置类

```
package com.itheima.mybatis.config;

import javax.sql.DataSource;

import org.mybatis.spring.SqlSessionFactoryBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.io.Resource;
import org.springframework.core.io.support.PathMatchingResourcePatternResolver;
import org.springframework.core.io.support.ResourcePatternResolver;

@Configuration
public class SqlSessionFactoryBeanConfiguration {

    @Bean(name = "sqlSessionFactory")
    public SqlSessionFactoryBean sessionFactory(DataSource dataSource) {

        // 创建SqlSessionFactoryBean对象
        SqlSessionFactoryBean sessionFactoryBean = new SqlSessionFactoryBean() ;
        sessionFactoryBean.setDataSource(dataSource);

        // 加载配置文件
        ResourcePatternResolver patternResolver = new PathMatchingResourcePatternResolver() ;
        Resource resource = patternResolver.getResource("classpath:mybatis/mybatis-config.xml");
        sessionFactoryBean.setConfigLocation(resource);

        // 返回
        return sessionFactoryBean ;
    }

}
```

5.2.2 MapperScannerConfigurer 配置

```
package com.itheima.mybatis.config;

import org.mybatis.spring.mapper.MapperScannerConfigurer;
import org.springframework.boot.autoconfigure.AutoConfigureAfter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
// 在SqlSessionFactoryBeanConfiguration配置完毕以后进行配置
@AutoConfigureAfter(value = {SqlSessionFactoryBeanConfiguration.class})
public class MapperScannerConfigurerConfiguration {

    @Bean
    public MapperScannerConfigurer mapperScannerConfigurer() {

        // 创建MapperScannerConfigurer对象
        MapperScannerConfigurer mapperScannerConfigurer = new MapperScannerConfigurer() ;
        mapperScannerConfigurer.setSqlSessionFactoryBeanName("sqlSessionFactory");
        mapperScannerConfigurer.setBasePackage("com.itheima.mybatis.springboot.mapper");

        // 返回
        return mapperScannerConfigurer ;

    }

}
```

5.3 application.properties

需要在该文件中配置数据库的链接信息

```
spring.datasource.druid.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.druid.url=jdbc:mysql://localhost:3306/spring-boot
spring.datasource.druid.username=root
spring.datasource.druid.password=1234
```

5.4 User 实体类

```
private Integer id ;                // 唯一标识
private String userName ;           // 用户名
private String sex ;                // 性别
private String address ;            // 住址
```

5.5 UserMapper 接口

```
package com.itheima.mybatis.springboot.mapper;

import java.util.List;

import com.itheima.mybatis.springboot.domain.User;

/**
 * 用户操作持久层接口
 * @author itcast-hly
 *
 */
public interface UserMapper {

    /**
     * 查询所有的用户数据
     * @return
     */
    public abstract List<User> findAllUser() ;
```

```
}
```

5.6 UserMapper.xml(映射文件)

```
<?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.itheima.mybatis.springboot.mapper.UserMapper" >

    <!-- 查询所有的用户信息 -->
    <select id="findAllUser" resultType="com.itheima.mybatis.springboot.domain.User">
        select id , username as userName , sex  , address from user
    </select>

</mapper>
```

5.7 编写入口类

```
package com.itheima.mybatis;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MybatisSpringBootApplication {

    public static void main(String[] args) {

        // 启动应用
        SpringApplication.run(MybatisSpringBootApplication.class, args) ;

    }

}
```

5.8 测试类

```
package com.itheima.mybatis.test;
import java.util.List;
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.SpringJUnit4ClassRunner;

import com.itheima.mybatis.MybatisSpringBootApplication;
import com.itheima.mybatis.springboot.domain.User;
import com.itheima.mybatis.springboot.mapper.UserMapper;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {MybatisSpringBootApplication.class})
public class MybatisSpringBootTest {

    @Autowired
    private UserMapper userMapper ;
    @Test
    public void findAll() {
        List<User> user = userMapper.findAllUser() ;
        System.out.println(user);
    }

}
```

至此 Spring Boot 和 Mybatis 就整合完毕了,也是很简单的

6 其他技术的整合

6.1 整合 Redis

6.1.1 pom.xml 文件

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

6.1.2 application.properties 配置

```
spring.redis.database=0
spring.redis.host=127.0.0.1
spring.redis.port=6379
```

如果要进行集群链接请配置: `spring.redis.cluster.nodes`

6.1.3 测试

使用过 Spring Data Redis 的开发人员应该知道,Spring Data Redis 给我们提供了一个类 StringRedisTemplate 用来简化我们对 Redis 的操作.我们现在使用的是 Spring Boot,那么针对于 StringRedisTemplate 这个类,Spring Boot 给我们进行了自动配置,因此我们只需要注入该类既可以使用了

```
package com.itheima.springboot;

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.data.redis.core.StringRedisTemplate;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(value = SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {RedisSpringBootApplcation.class})
public class RedisSpringBootTest {

    @Autowired
    private StringRedisTemplate redisTemplate ;

    @Test
    public void stringValue() {
        redisTemplate.boundValueOps("itheima").set("黑马程序员");
    }

    @Test
    public void getValue() {
        String value = redisTemplate.boundValueOps("itheima").get() ;
        System.out.println(value);
    }


}
```

6.2 整合 Solr

6.2.1 pom.xml 文件

```
<!-- 配置spring boot 和solr整合的starter -->
<dependency>
```

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

6.2.2 application.propert 配置

```
# solr单节点配置
spring.data.solr.host=http://127.0.0.1:8080/solr
# 集群配置
# spring.data.solr.zk-host=192.168.221.155:2181,192.168.221.155:2182,192.168.221.155:2183
```

6.2.3 测试

Spring Boot 针对 Solr 提供的自动配置类为 SolrClient,因此我们只需要注入 SolrClient 即可

```
package com.itheima.springboot;

import org.apache.solr.client.solrj.SolrClient;
import org.apache.solr.common.SolrInputDocument;
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.SpringJUnit4ClassRunner;

@RunWith(value = SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {RedisSpringBootApplcation.class})
public class RedisSpringBootTest {

    @Autowired
    private SolrClient solrClient ;

    @Test
    public void addDocument() throws Exception {
        SolrInputDocument inputDocument = new SolrInputDocument() ;
        inputDocument.addField("id", "100");
        inputDocument.addField("product_name", "传智播客是一家 IT 教育机构");
        solrClient.add(inputDocument) ;
        solrClient.commit() ;
    }

}
```

至于查询大家如果使用过 solrj 应该是没有问题的.

6.3 整合 ActiveMQ

6.3.1 pom.xml 文件

```
<!-- 加入activemq的支持 -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
```

6.3.2 application.properties 的配置

```
spring.activemq.broker-url=tcp://192.168.80.130:61616
```

6.3.3 配置 Destination

```
package com.itheima.activemq.config;
```

```
import javax.jms.Queue;
import javax.jms.Topic;

import org.apache.activemq.command.ActiveMQQueue;
import org.apache.activemq.command.ActiveMQTopic;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class DestinationConfig {

    @Bean
    public Queue queueDestination() {
        return new ActiveMQQueue("itcast-activemq-queue") ;
    }

    @Bean
    public Topic topicDestination() {
        return new ActiveMQTopic("itcast-activemq-topic") ;
    }

}
```

6.3.4 发送消息类

```
package com.itheima.activemq.service;

import javax.jms.Queue;
import javax.jms.Topic;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Service;

@Service
public class ActiveMQProducer {

    @Autowired
    private JmsTemplate jmsTemplate;

    @Autowired
    private Queue queue;

    @Autowired
    private Topic topic;

    public void sendMessage() {

        for (int x = 0; x < 10000; x++) {

            // 发送消息
            jmsTemplate.convertAndSend(queue, "queue......" + x);
            jmsTemplate.convertAndSend(topic, "topic ......" + x);

            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }

    }

}
```

6.3.5 测试类

```
package com.itheima.activemq.test;

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.SpringJUnit4ClassRunner;

import com.itheima.activemq.ActiveMqApplication;
import com.itheima.activemq.service.ActiveMQProducer;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = {ActiveMqApplication.class})
public class ActivemqApplicationTest {

    @Autowired
    private ActiveMQProducer activeMQProducer ;

    @Test
    public void sendMessage() {
        activeMQProducer.sendMessage();
    }

}
```

6.3.6 消息接收端

```
package com.itheima.springboot;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class ActiveMQConsumer {

    @JmsListener(destination = "itcast-activemq-topic")
    public void topicConsumer(String text) {
        System.out.println("spring boot activemq topicConsumer ========>>>> " + text);
    }

    @JmsListener(destination = "itcast-activemq-queue")
    public void queueConsumer(String text) {
        System.out.println("spring boot activemq queueConsumer ========>>>> " + text);
    }

}
```

6.3.7 小结

通过上面的测试我们发现只能接收到 queue 的消息,不能接收到 topic 的消息!那么因为 Spring Boot 在集成 activemq 的时候默认只支持 queue 的消息!如果我们想让其支持 topic 类型的消息!我们就需要是人如下配置:

```
spring.jms.pub-sub-domain=true
```

加入上述配置了以后,我们发现只能接收 topic 的类型的消息,不能接收到 queue 类型的消息!因为该值一个 boolean 类型的值.默认是 false(支持 queue) , true(支持 topic) 如果我们即想获取 queue 的消息,也想获取到 topic 的消息,那么我们就需要自定义消息监听器

6.3.8 自定义消息监听器

```
package com.itheima.springboot;

import javax.jms.ConnectionFactory;
```

```
import org.apache.activemq.ActiveMQConnectionFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerContainerFactory;

@Configuration
public class ActiveMQConsumerListenterConfig {

    @Bean(name = "activemqConnectionFactory")
    public ConnectionFactory activemqConnectionFactroy() {
        ConnectionFactory connectionFactory = new
ActiveMQConnectionFactory("tcp://192.168.80.130:61616") ;
        return connectionFactory ;
    }

    @Bean(name = "topicJmsListenerContainerFactory")
    public JmsListenerContainerFactory topicJmsListenerContainerFactory(ConnectionFactory
connectionFactory) {
        DefaultJmsListenerContainerFactory defaultJmsListenerContainerFactory = new
DefaultJmsListenerContainerFactory() ;
        defaultJmsListenerContainerFactory.setPubSubDomain(true)  ; // 支持topic
        defaultJmsListenerContainerFactory.setConnectionFactory(connectionFactory);
        return defaultJmsListenerContainerFactory ;
    }

    @Bean(name = "queueJmsListenerContainerFactory")
    public JmsListenerContainerFactory queueJmsListenerContainerFactory(ConnectionFactory
connectionFactory) {
        DefaultJmsListenerContainerFactory defaultJmsListenerContainerFactory = new
DefaultJmsListenerContainerFactory() ;
        defaultJmsListenerContainerFactory.setConnectionFactory(connectionFactory);
        return defaultJmsListenerContainerFactory ;
    }

}
```

修改监听,指定使用的消息监听器

```
package com.itheima.springboot;

import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;

@Component
public class ActiveMQConsumer {

    @JmsListener(destination = "itcast-activemq-topic" , containerFactory = "topicJmsListenerContainerFactory")
    public void topicConsumer(String text) {
        System.out.println("spring boot activemq topicConsumer ========>>>> " + text);
    }

    @JmsListener(destination = "itcast-activemq-queue" , containerFactory = "queueJmsListenerContainerFactory")
    public void queueConsumer(String text) {
        System.out.println("spring boot activemq queueConsumer ========>>>> " + text);
    }

}
```

6.4 整合 dubbo

Spring Boot 和 dubbo 进行整合的时候也是存在两种方式:

- 使用@ImportSouce 注解读取配置文件
- 使用 dubbo-starter **的进行实现**

我们本次使用的是第二种方式,因为第二种方式较第一种方式简单

6.4.1 pom.xml 文件

```
<!-- 定义变量 -->
<properties>
    <java.version>1.7</java.version>
    <dubbo-spring-boot>1.0.0</dubbo-spring-boot>
</properties>

<!-- 添加父工程 -->
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.9.RELEASE</version>
</parent>

<dependencies>

    <!-- spring boot的测试包 -->
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>

    <!-- Spring Boot Dubbo 依赖 -->
    <dependency>
        <groupId>io.dubbo.springboot</groupId>
        <artifactId>spring-boot-starter-dubbo</artifactId>
        <version>${dubbo-spring-boot}</version>
    </dependency>

</dependencies>

<build>
    <plugins>

        <!-- Spring Boot的maven的打包插件 -->
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>

    </plugins>
</build>
```

6.4.2 服务提供方

6.4.2.1 代码

```
package com.itheima.dubbo.service.impl;

import com.alibaba.dubbo.config.annotation.Service;
import com.itheima.dubbo.service.UserService;

@Service
public class UserServiceImpl implements UserService {

    @Override
    public String sayHello() {
        return "sayHello";
    }

}
```

6.4.2.2 application.properties 配置

```
spring.dubbo.application.name=dubbo-provider
spring.dubbo.registry.address=zookeeper://192.168.80.130:2181
spring.dubbo.protocol.name=dubbo
spring.dubbo.protocol.port=20880
spring.dubbo.scan=com.itheima.dubbo.service.impl
```

6.4.3 服务消费方

6.4.3.1 application.properties 配置

```
server.port=8081
spring.dubbo.application.name=dubbo-consumer
spring.dubbo.registry.address=zookeeper://192.168.80.130:2181
spring.dubbo.protocol.name=dubbo
spring.dubbo.protocol.port=20880
spring.dubbo.scan=com.itheima.dubbo.controller
```

6.4.3.2 代码

```
package com.itheima.dubbo.controller;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

import com.alibaba.dubbo.config.annotation.Reference;
import com.itheima.dubbo.service.UserService;

@Controller
public class UserController {

    @Reference
    private UserService userService ;

    @RequestMapping(value = "/sayHello")
    @ResponseBody
    public String sayHello() {
        return userService.sayHello() ;
    }
}
```

7 总结

通过本课程的学习我们应该对 Spring Boot 有了一个深刻的认识！本课程中是对一些主流的技术进行了整合的讲解,其他技术大家可以查找一下资料自行学习!