1.SpringBoot

Spring 大家已经很熟悉了，它能快速的与其它框架进行整合。boot 是启动的意思。SpringBoot 就是启动一个 Spring 构建的项目。SpringBoot 对 Spring 和第三方类库进行了封装，你可以用最少的配置开发项目。大多数的 SpringBoot 项目只需要很少的配置。

有 Spring 的基础，掌握 SpringBoot 很简单。Spring 中有大量的 xml 配置，在 SpringBoot 中可以实现完全的无 xml，将 Spring 中的 xml 配置用 java 的方式配置就可以，大部分的配置都被 SpringBoot 自动装配了，我们只需要定义少部分的内容即可。

后续代码会将 Spring 和 SpringBoot 的语法放在一起进行比较。

2.创建示例工程

演示创建一个 springMVC 的工程。SpringBoot 用一个普通的 java 工程就可以运行 web 应用。它没有 springMVC 的 xml 配置文件，也没有 web.xml

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.7.RELEASE</version>
    <relativePath/>
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

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

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

controller 类：

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

@Controller
public class HelloController {
    @RequestMapping("hello")
    @ResponseBody
    public String helloWorld() {
        return "hello springboot";
    }
}
```

测试类：

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.ComponentScan;

@SpringBootApplication
@ComponentScan("com.spring.test.springboot.controller")
public class StartApplication {
    public static void main(String[] args) {
        SpringApplication.run(StartApplication.class, args);
    }
}
```

@SpringBootApplication：Spring Boot 项目的核心注解，主要目的是开启自动配置。可以将程序以 web 方式运行

@ComponentScan：扫描包，此处扫描的是 controller 所在的包，相当于原来 xml 中的 <context:component-scan base-package=""/>

启动这个 main 方法，通过浏览器访问：

或者配置了 spring-boot-maven-plugin 后，使用 maven 命令启动：spring-boot:run

3.SpringBoot 核心相关内容

3.1 入口类

SpringBoot 通常有一个入口类*Application,内部有一个 main 方法，是启动 SpringBoot 的入口。

使用@SpringBootApplication 注解，并包含 main 方法

3.2 常见注解

@SpringBootApplication ：是 SpringBoot 的核心注解，用于标注程序时一个 SpringBoot 程序。它是一个组合注解，由多个注解组合而成。

@SpringBootApplication=@ComponentScan+@Configuration+@EnableAutoConfiguration

@SpringBootConfiguration： 一个组合注解，相当于传统的 xml 配置文件,包含 @Configuration 注解。在 Spring Boot 项目中推荐使用@SpringBootConfiguration 替代 @Configuration。

@EnableAutoConfiguration ：启用自动配置，该注解会使 Spring Boot 根据项目中依赖的 jar 包自动配置项目的配置项,这也是 springboot 的核心注解之一,我们只需要将项目需要的依赖包加入进来,它会自动帮我们配置这个依赖需要的基本配置。比如我们的项目引入了 spring-boot-starter-web 依赖,springboot 会自动帮我们配置 tomcat 和 springmvc

@ComponentScan：组件扫描，可发现和自动装配一些 bean。默认扫描 @SpringBootApplication 类所在包的同级目录以及它的子目录。

设置不自动装配：

@SpringBootApplication(exclude = {JpaRepositoriesAutoConfiguration.class, RedisAutoConfiguration.class})

注解内部将不需要自动配置的依赖通过 exclude 参数指定即可,可以指定多个类

@Bean： 注解在方法上，声明当前方法返回一个 Bean

@PostConstruct：注解在方法上，构造函数执行后执行。

@PreDestroy： 注解在方法上，在 Bean 销毁前执行。

@Lazy(true): 延迟初始化

@Scope：注解在类上，描述 spring 容器如何创建 Bean 实例。

@Profile：注解在方法类上 在不同情况下选择实例化不同的 Bean 特定环境下生效

@Import：用来导入其他配置类。

@ImportResource：用来加载 xml 配置文件。

3.3 核心配置文件

3.3.1 yml 和 properties

SpringBoot 使用一个全局配置文件 application.properties 或者 application.yml。properties 配置文件见附件。

yml 类似于 xml，但是 yml 没有 xml 中的 标签，而是通过空格来表示层级结构：

```
#相当于 properties 中的 server.port=80
server:
  port: 80
#代表 spring.jpa.下面的属性 每个:之后的代表当前属性下的属性
spring:
  jpa:
    generate-ddl: false
```

```
    show-sql: true
    hibernate:
      ddl-auto: update
    database: mysql
```

SpringBoot 的配置文件可以放在以下几个地方：

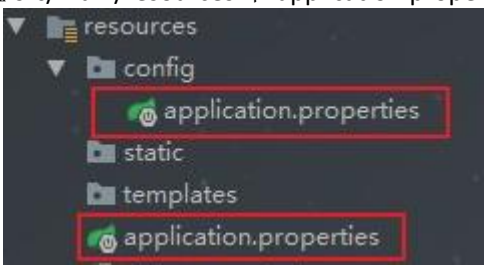
外置，在相对于应用程序运行目录的/congfig 子目录里。(resources/config)

外置，在应用程序运行的目录里(resources)

内置，在 config 包

内置，在 Classpath 根目录

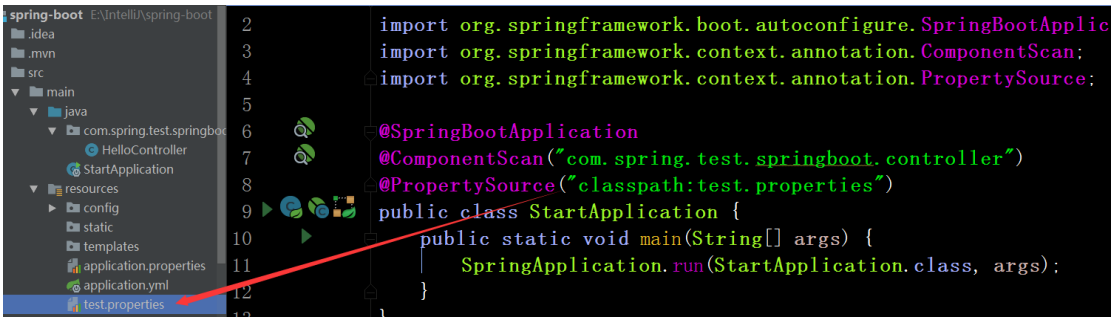
这个列表按照优先级排序，也就是说，src/main/resources/config 下 application.properties 覆盖 src/main/resources 下 application.properties 中相同的属性



如果在相同优先级位置同时有 application.properties 和 application.yml，那么 application.properties 里的属性里面的属性就会覆盖 application.yml

如果自己定义了其它的配置文件，如 test.properties，可以使用@PropertiesSource 注解指定加载配置文件。

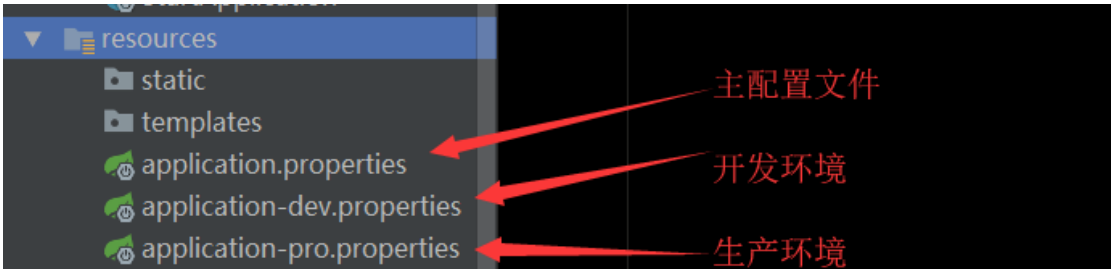
```
@PropertySource("classpath:test.properties")
```



3.3.2 profile 多环境配置

当应用程序需要部署到不同运行环境时，一些配置细节通常会有所不同，最简单的比如日志，生产日志会将日志级别设置为 WARN 或更高级别，并将日志写入日志文件，而开发的时候需要日志级别为 DEBUG，日志输出到控制台即可。

如果按照以前的做法，就是每次发布的时候替换掉配置文件，这样太麻烦了，Spring Boot 的 Profile 就给我们提供了解决方案，命令带上参数就搞定。



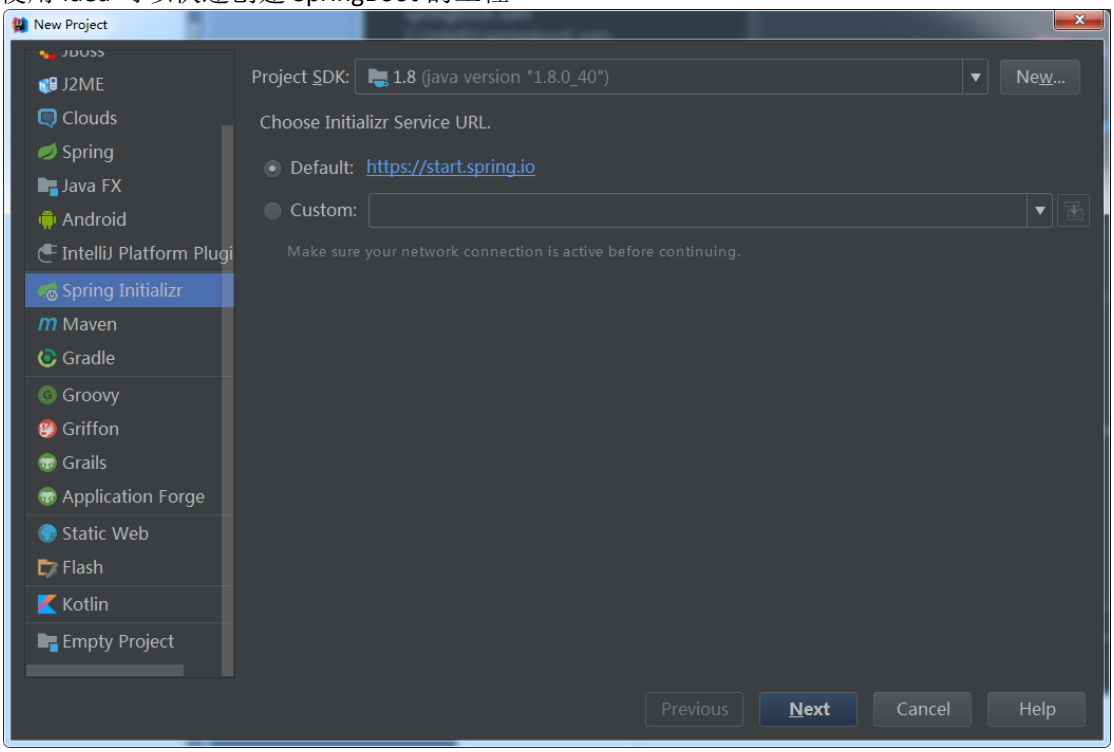
切换的配置文件必须符合 application-xx 的命名，其中 xx 和 application.properties 中指定的属性值对应，在 application.properties 中进行如下配置，系统将会使用 application-dev.properties 中的配置：

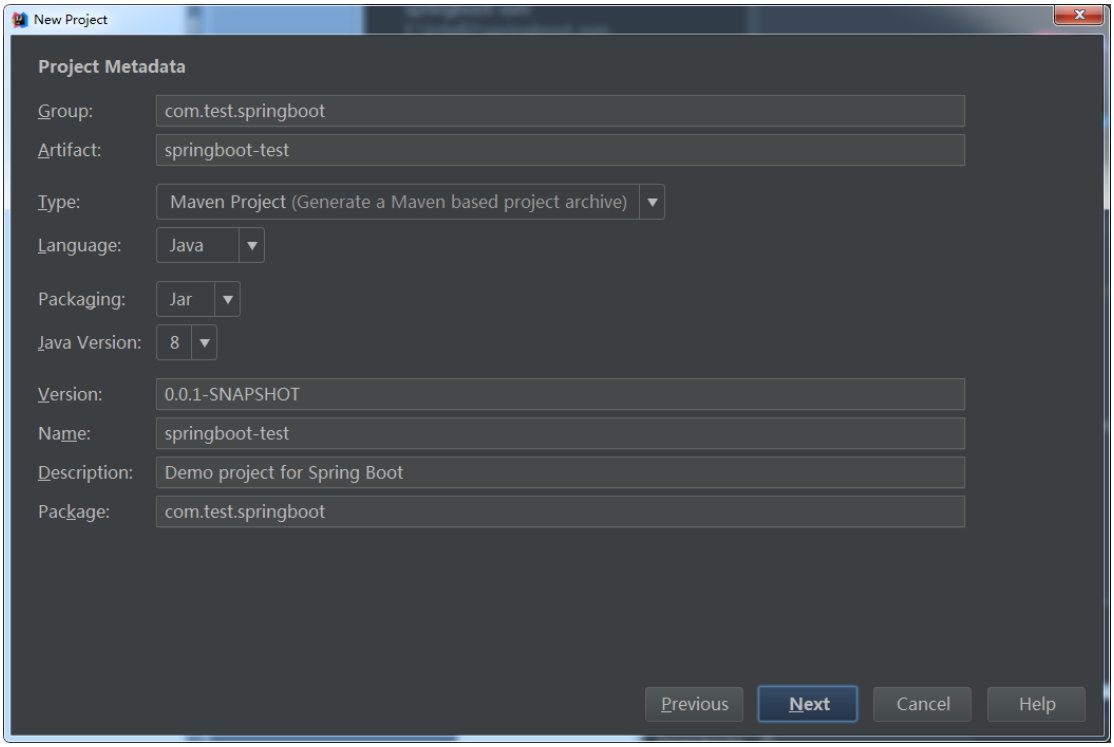
```
spring.profiles.active=dev
```

4.使用 SpringBoot 整合 ssm

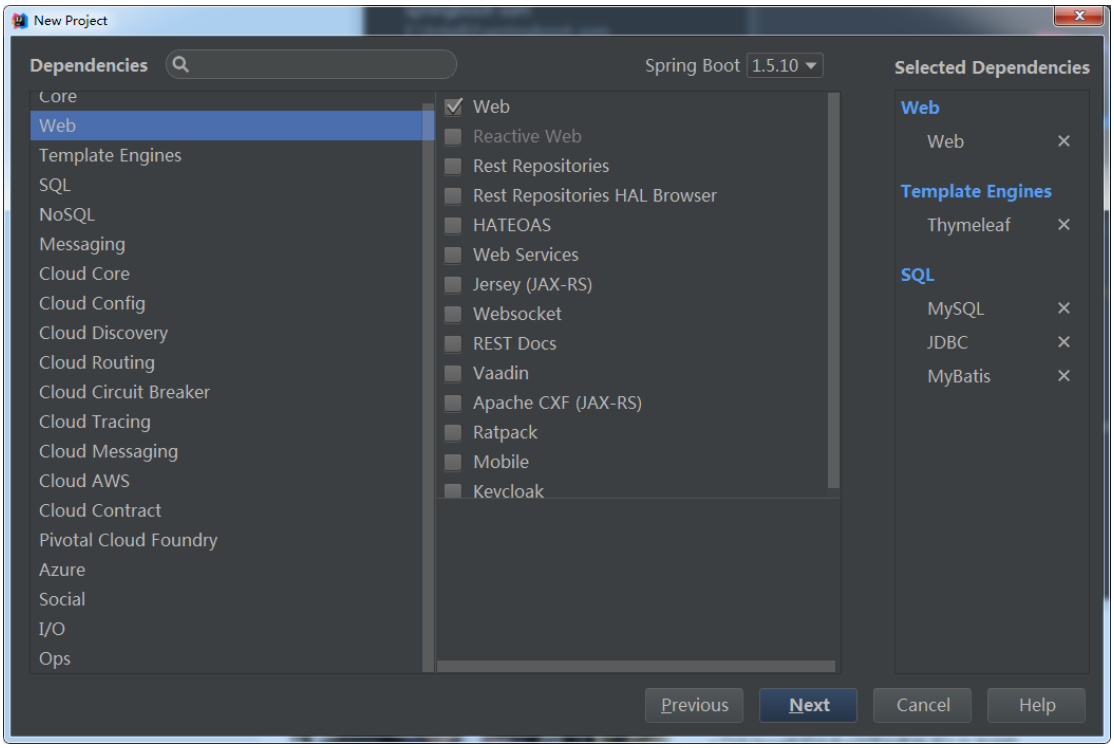
4.1 创建工程

使用 idea 可以快速创建 SpringBoot 的工程





这里选择常用的类库，SpringBoot 将各种框架类库都进行了封装，可以减少 pom 文件中的引用配置：



比如 Spring 和 Mybatis 整合的时候，传统 Spring 项目中需要引入：

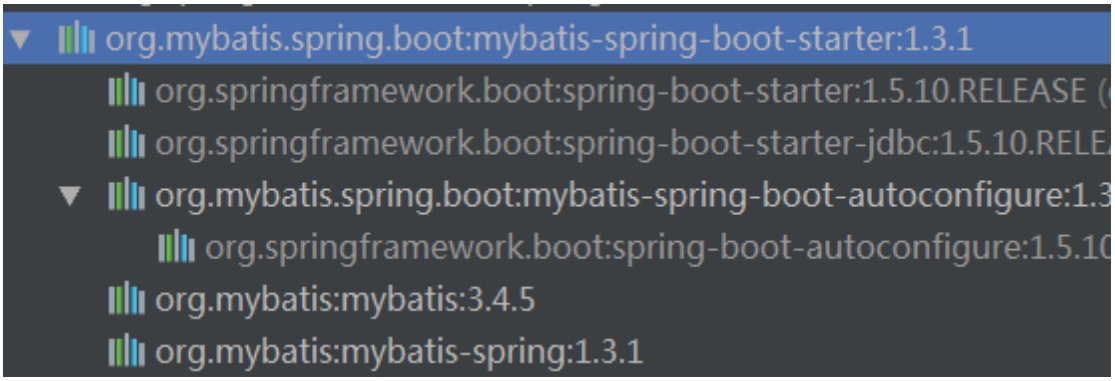
```
<dependency>
    <groupId>org.mybatis</groupId>
    <artifactId>mybatis</artifactId>
    <version>3.4.1</version>
</dependency>
<dependency>
    <groupId>org.mybatis</groupId>
```

```
    <artifactId>mybatis-spring</artifactId>
    <version>1.3.1</version>
</dependency>
```

而在 SpringBoot 中引入的是：

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.3.1</version>
</dependency>
```

可以看到这个类库中除了 mybatis 和 mybatis-spring 之外，还有 spring-boot 的东西



完整的 pom.xml 如下：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.10.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
    <java.version>1.8</java.version>
</properties>

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

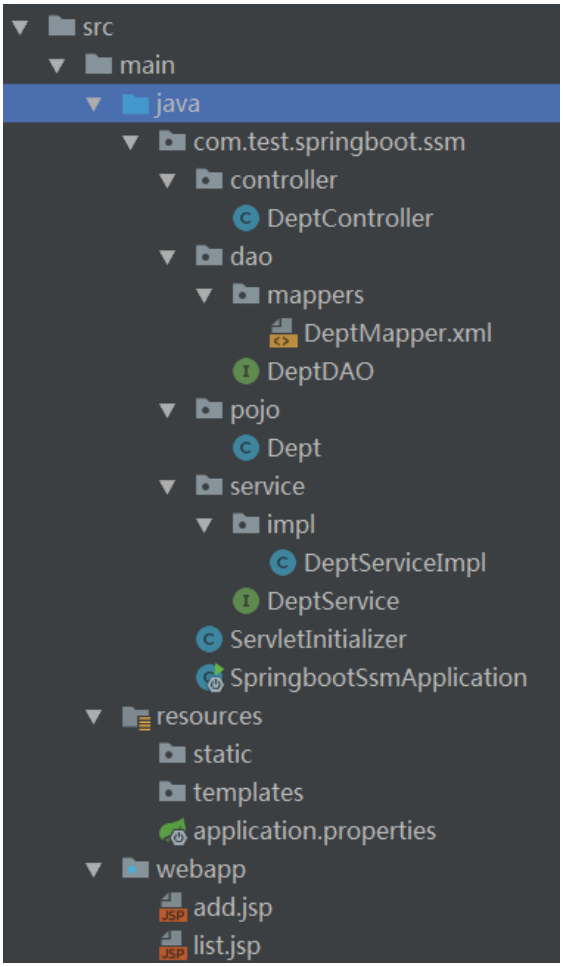
```
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>1.3.1</version>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-tomcat</artifactId>
        <scope>provided</scope>
    </dependency>

    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <scope>runtime</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <!--使用 jsp 页面-->
    <dependency>
        <groupId>org.apache.tomcat.embed</groupId>
        <artifactId>tomcat-embed-jasper</artifactId>
    </dependency>
    <dependency>
        <groupId>jstl</groupId>
        <artifactId>jstl</artifactId>
        <version>1.2</version>
    </dependency>
</dependencies>

<build>
    <finalName>boot</finalName>
    <plugins>
        <plugin>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-maven-plugin</artifactId>
        </plugin>
    </plugins>
    <resources>
        <resource>
            <directory>src/main/java</directory>
            <includes>
                <include>**/*.xml</include>
            </includes>
        </resource>
        <resource>
            <directory>src/main/resources</directory>
            <includes>
                <include>**/*.xml</include>
                <include>**/*.properties</include>
            </includes>
```

```
        </resource>
    </resources>
</build>
```

完整的工程路径如下：



4.2 实体类和 DAO

```
public class Dept {
    private Integer id;
    private String name;

    //getter/setter 方法略
}
```

```
import com.test.springboot.ssm.pojo.Dept;
import org.apache.ibatis.annotations.Insert;
import org.apache.ibatis.annotations.Options;

import java.util.List;

public interface DeptDAO {
    //查询列表，演示使用传统的 mapper 映射文件
```

```
    List<Dept> getDeltList();

    //插入，演示使用注解编写 sql，省略 xml 配置
    @Insert("insert into DEPT(NAME) values(#{name})")
    @Options(useGeneratedKeys = true, keyProperty = "id", keyColumn = "ID")
    void addDept(String name);
}
```

DeptMapper.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.test.springboot.ssm.dao.DeptDAO">

    <resultMap id="deptMap" type="Dept">
        <id property="id" column="ID"/>
        <result property="name" column="NAME"/>
    </resultMap>

    <select id="getDeltList" resultMap="deptMap">
        select ID,NAME from DEPT
    </select>
</mapper>
```

4.3 Service

```
public interface DeptService {
    List<Dept> getDeltList();

    void addDept(String name);
}
```

```
@Service
public class DeptServiceImpl implements DeptService {
    @Autowired
    private DeptDAO deptDAO;

    @Override
    public List<Dept> getDeltList() {
        return deptDAO.getDeltList();
    }

    @Override
    public void addDept(String name) {
        deptDAO.addDept(name);
    }
}
```

4.4 Controller 和页面

```
@Controller
public class DeptController {
    @Autowired
    private DeptService deptService;

    @RequestMapping("list.html")
    public ModelAndView list() {
        List<Dept> deptList = deptService.getDeltList();
        return new ModelAndView("list", "deptList", deptList);
    }

    @RequestMapping("add.html")
    public String add(String name) {
        deptService.addDept(name);
        //添加成功后重定向到列表页
        return "redirect:list.html";
    }
}
```

add.jsp

```
<form action="/add.html" method="post">
    部门名:<input type="text" name="name"/><br/>
    <input type="submit" value="add"/>
</form>
```

list.jsp

```
<c:forEach items="${deptList}" var="dept">
    ${dept.id}-${dept.name}<br/>
</c:forEach>
```

4.5 启动类

到目前为止，项目与传统的 spring 没有任何区别。
传统 spring 项目中需要增加下面两个配置文件,而 SpringBoot 中没有配置文件:
传统 Spring 项目中有以下文件:
spring-config.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
        http://www.springframework.org/schema/context
```

```
       http://www.springframework.org/schema/context/spring-context-4.2.xsd
       http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.0.xsd
       http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.0.xsd">
    <!--扫描@Service 注解-->
    <context:component-scan base-package="com.test.springboot.ssm.service">
        <context:include-filter type="annotation"
expression="org.springframework.stereotype.Service"/>
    </context:component-scan>
    <!--读取配置文件-->
    <context:property-placeholder location="classpath:db.properties"
ignore-unresolvable="true"/>
    <!--从配置文件中获取数据源-->
    <bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName" value="${jdbc.driver}"/>
        <property name="url" value="${jdbc.url}"/>
        <property name="username" value="${jdbc.username}"/>
        <property name="password" value="${jdbc.password}"/>
    </bean>

    <!--spring 管理 session 工厂-->
    <bean id="sqlSessionFactory" class="org.mybatis.spring.SqlSessionFactoryBean">
        <property name="dataSource" ref="dataSource"/>
        <property name="mapperLocations"
value="classpath:com/test/springboot/ssm/dao/mapper/*.xml"/>
        <!--配置实体类别名别名-->
        <property name="typeAliasesPackage" value="com.test.springboot.ssm.pojo"/>
    </bean>

    <!--扫描所有 mybatis 的 dao 接口，生成代理实现类-->
    <bean class="org.mybatis.spring.mapper.MapperScannerConfigurer">
        <property name="basePackage" value="com.test.springboot.ssm.dao"/>
    </bean>

    <!-- 配置事务管理器 -->
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
    </bean>

    <!--事务增强-->
    <tx:advice id="txAdvice" transaction-manager="transactionManager">
        <tx:attributes>
            <!-- 传播行为，匹配的是方法名 -->
            <tx:method name="add*" rollback-for="Exception"/>
            <tx:method name="delete*" rollback-for="Exception"/>
            <tx:method name="update*" rollback-for="Exception"/>
            <tx:method name="get*" propagation="SUPPORTS" read-only="true"/>
            <tx:method name="do*" rollback-for="Exception"/>
        </tx:attributes>
    </tx:advice>
```

```
    <!-- 通过 AOP 配置提供事务增强，让 service 包下所有 Bean 的所有方法拥有事务 -->
    <aop:config>
        <aop:pointcut id="serviceMethod"
                      expression="execution(* com.test.springboot.ssm..*(..))"/>
        <aop:advisor pointcut-ref="serviceMethod" advice-ref="txAdvice"/>
    </aop:config>

</beans>
```

springMVC-servlet.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:mvc="http://www.springframework.org/schema/mvc"
       xsi:schemaLocation="
  http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.2.xsd
  http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.2.xsd
  http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc-4.2.xsd">

    <mvc:annotation-driven/>
    <!--扫描 Controller 所在的包-->
    <context:component-scan base-package="com.ssm.blog.controller">
        <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
    </context:component-scan>

    <!-- 配置视图解析器-->
    <bean
class="org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/"></property><!--前缀-->
        <property name="suffix" value=".jsp"></property><!--后缀-->
    </bean>

</beans>
```

web.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns="http://java.sun.com/xml/ns/javaee"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
          http://java.sun.com/xml/ns/javaee/web-app_3_0.xsd"
         version="3.0">
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>classpath:spring-config.xml</param-value>
    </context-param>
    <!--配置 listener，在启动 Web 容器的时候加载 Spring 的配置-->
    <listener>
```

```
<listener-class>org.springframework.web.context.ContextLoaderListener</listener-cla
ss>
    </listener>
    <filter>
        <filter-name>CharacterEncodingFilter</filter-name>

<filter-class>org.springframework.web.filter.CharacterEncodingFilter</filter-class>
        <init-param>
            <param-name>encoding</param-name>
            <param-value>UTF-8</param-value>
        </init-param>
        <init-param>
            <param-name>forceEncoding</param-name>
            <param-value>true</param-value>
        </init-param>
    </filter>
    <filter-mapping>
        <filter-name>CharacterEncodingFilter</filter-name>
        <url-pattern>/*</url-pattern>
    </filter-mapping>
    <!--配置 DispatcherServlet -->
    <servlet>
        <servlet-name>springMVC</servlet-name>

<servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>springMVC</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
</web-app>
```

而 SpringBoot 中不需要这三个配置文件,写一个启动类，运行 main 方法即可：

```
import org.mybatis.spring.annotation.MapperScan;
import org.springframework.aop.aspectj.AspectJExpressionPointcut;
import org.springframework.aop.support.DefaultPointcutAdvisor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.jdbc.datasource.DataSourceTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;
import org.springframework.transaction.interceptor.TransactionInterceptor;

import javax.sql.DataSource;
import java.util.Properties;

@SpringBootApplication
@EnableTransactionManagement//开启事务管理
@ComponentScan("com.test.springboot.ssm")//扫描注解元素
@MapperScan("com.test.springboot.ssm.dao")//Mybatis 的 DAO 所在包
public class SpringbootSsmApplication {
```

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

    public static final String transactionExecution = "execution (*
com.test.springboot.service..*(..))";

    @Autowired
    private DataSource dataSource;

    //声明式事务
    @Bean
    public DefaultPointcutAdvisor defaultPointcutAdvisor() {
        AspectJExpressionPointcut pointcut = new AspectJExpressionPointcut();
        pointcut.setExpression(transactionExecution);
        DefaultPointcutAdvisor advisor = new DefaultPointcutAdvisor();
        advisor.setPointcut(pointcut);
        Properties attributes = new Properties();
        attributes.setProperty("get*", "PROPAGATION_REQUIRED,-Exception");
        attributes.setProperty("add*", "PROPAGATION_REQUIRED,-Exception");
        attributes.setProperty("update*", "PROPAGATION_REQUIRED,-Exception");
        attributes.setProperty("delete*", "PROPAGATION_REQUIRED,-Exception");
        TransactionInterceptor txAdvice = new TransactionInterceptor(new
DataSourceTransactionManager(dataSource), attributes);
        advisor.setAdvice(txAdvice);
        return advisor;
    }
}
```

数据库等配置信息放到 application.properties 中

```
#数据源的基本信息
spring.datasource.url = jdbc:mysql://localhost:3306/test?characterEncoding=utf-8
spring.datasource.username = root
spring.datasource.password =
spring.datasource.driverClassName = com.mysql.jdbc.Driver

#mybatis 中 mapper 文件的路径
mybatis.mapper-locations=classpath*:com/test/springboot/ssm/dao/mappers/*.xml
#起别名。可省略写 mybatis 的 xml 中的 resultType 的全路径
mybatis.type-aliases-package=com.test.springboot.ssm.pojo

#springMVC 中的视图信息，响应前缀
spring.mvc.view.prefix=/
# 响应页面默认后缀
spring.mvc.view.suffix=.jsp
#DispatcherServlet 中响应的 url-pattern
server.sevlet-path=*.html
server.context-path=/boot

#logging.level.root=debug
logging.level.com.test.springboot.ssm.dao=trace
```

上面的程序只要启动 main 方法就可以访问了。

另外，如果需要打包发布到 tomcat，需要再配置一个 ServletInitializer，否则 tomcat 启动后

会出现 404。

```
import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.web.support.SpringBootServletInitializer;

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

5.启动原理解析

任何一个 SpringBoot 程序都有一个启动类：

```
@SpringBootApplication
public class StartApplication {
    public static void main(String[] args) {
        SpringApplication.run(StartApplication.class, args);
    }
}
```

启动类中包含@SpringBootApplication 注解和 SpringApplication.run()方法

5.1 @SpringBootApplication

@SpringBootApplication 是一个组合注解，除了基本的原信息标注以外，重要的注解有三个：
@Configuration
@EnableAutoConfiguration
@ComponentScan
如下代码等同于使用@SpringBootApplication 注解

```
@Configuration
@EnableAutoConfiguration
@ComponentScan
public class StartApplication {
    public static void main(String[] args) {
        SpringApplication.run(StartApplication.class, args);
    }
}
```

每次写三个注解比较繁琐，所以使用@SpringBootApplication 更方便。

5.1.1 @Configuration

简单的说，SpringBoot 中使用一个@Configuration 注解的类代替 xml 配置文件。
如 spring-config.xml 如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
       ">
    <!--定义 bean-->
</beans>
```

SpringBoot 中写成：

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

@Configuration
public class SpringConfig {
}
```

如果定义一个 bean,xml 中写成：

```
<bean id="dept" class="com.spring.test.springboot.pojo.Dept">
    <property name="id" value="1"/>
</bean>

<bean id="employee" class="com.spring.test.springboot.pojo.Employee">
    <property name="name" value="tom"/>
    <property name="dept" ref="dept"/>
</bean>
```

SpringBoot 中写成：

```
@Bean
public Dept dept() {
    Dept dept = new Dept();
    dept.setId(1);
    return dept;
}

@Bean
public Employee employee() {
    Employee employee = new Employee();
    employee.setName("tom");
    employee.setDept(dept());//注入依赖对象直接调用@Bean 注解的方法
    return employee;
}
```

SpringBoot 中使用@Bean 标注一个方法，该方法的方法名将默认成 bean 的 id。注意@Configuration 的类要被@ComponentScan 扫描到。

5.1.2 @ComponentScan

@ComponentScan 自动扫描并加载符合规则的组件。可以通过 basePackages 指定要扫描的包。如果不指定扫描范围，SpringBoot 默认会从生命@ComponentScan 所在类的包进行扫描。

```
@ComponentScan(basePackages =
"com.spring.test.springboot.controller",includeFilters =
{@ComponentScan.Filter(type= FilterType.ANNOTATION,value=Controller.class)})
```

等同于

```
<context:component-scan base-package="com.spring.test.springboot.controller">
    <context:include-filter type="annotation"
expression="org.springframework.stereotype.Controller"/>
</context:component-scan>
```

5.5.3 @EnableAutoConfiguration

这个注解的作用是将所有符合自动配置条件的 bean 自动加载到 IoC 容器。比如我们的项目引入了 spring-boot-starter-web 依赖,springboot 会自动帮我们配置 tomcat 和 springmvc。@EnableAutoConfigutation 中@Import 了 EnableAutoConfigurationImportSelector，EnableAutoConfigurationImportSelector 类使用了 Spring Core 包的 SpringFactoriesLoader 类的 loadFactoryNamesof()方法。 SpringFactoriesLoader 会查询 META-INF/spring.factories 文件中包含的 JAR 文件。 当找到 spring.factories 文件后，SpringFactoriesLoader 将查询配置文件命名的属性。spring.factories 文件，内容如下：

```
# Initializers
org.springframework.context.ApplicationContextInitializer=\
org.springframework.boot.autoconfigure.logging.AutoConfigurationReportLoggingInitializer

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.redis.RedisAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.jdbc.DataSourceTransactionManagerAutoConfiguration,\
org.springframework.boot.autoconfigure.jms.JmsTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration,\
org.springframework.boot.autoconfigure.mobile.DeviceResolverAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoAutoConfiguration,\
org.springframework.boot.autoconfigure.mongo.MongoTemplateAutoConfiguration,\
org.springframework.boot.autoconfigure.orm.jpa.HibernateJpaAutoConfiguration,\
org.springframework.boot.autoconfigure.reactor.ReactorAutoConfiguration,\
org.springframework.boot.autoconfigure.security.SecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.security.FallbackWebSecurityAutoConfiguration,\
org.springframework.boot.autoconfigure.thymeleaf.ThymeleafAutoConfiguration,\
org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration,\
org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration,\
org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration,\
org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration,\
org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration,\
org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration,\
org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration
```

5.2 SpringApplication

SpringApplication 的 run 方法的实现是我们本次旅程的主要线路，该方法的主要流程大体可以归纳如下：

1） 如果我们使用的是 SpringApplication 的静态 run 方法，那么，这个方法里面首先要创建一个 SpringApplication 对象实例，然后调用这个创建好的 SpringApplication 的实例方法。在 SpringApplication 实例初始化的时候，它会提前做几件事情：

a) 根据 classpath 里面是否存在某个特征类
（org.springframework.web.context.ConfigurableWebApplicationCon

text）来决定是否应该创建一个为 Web 应用使用的 ApplicationContext 类型。

b) 使用 SpringFactoriesLoader 在应用的 classpath 中查找并加载所有可用的 ApplicationContextInitializer。

c) 使用 SpringFactoriesLoader 在应用的 classpath 中查找并加载所有可用的 ApplicationListener。

d) 推断并设置 main 方法的定义类。

2） SpringApplication 实例初始化完成并且完成设置后，就开始执行 run 方法的逻辑了，方法执行伊始，首先遍历执行所有通过 SpringFactoriesLoader 可以查找到并加载的 SpringApplicationRunListener。调用它们的 started()方法，告诉这些 SpringApplicationRunListener，“嘿，SpringBoot 应用要开始执行咯！”。

3） 创建并配置当前 Spring Boot 应用将要使用的 Environment（包括配置要使用的 PropertySource 以及 Profile）。

4） 遍历调用所有 SpringApplicationRunListener 的 environmentPrepared()的方法,告诉他们:“当前 SpringBoot 应用使用的 Environment 准备好了咯!”。

5）如果 SpringApplication 的 showBanner 属性被设置为 true,则打印 banner。

6）根据用户是否明确设置了 applicationContextClass 类型以及初始化阶段的推断结果，决定该为当前 SpringBoot 应用创建什么类型的 ApplicationContext 并创建完成，然后根据条件决定是否添加 ShutdownHook，决定是否使用自定义的 BeanNameGenerator，决定是否使用自定义的 ResourceLoader，当然，最重要的，将之前准备好的 Environment 设置给创建好的 ApplicationContext 使用。

7） ApplicationContext 创建好之后，SpringApplication 会再次借助 Spring-FactoriesLoader，查找并加载 classpath 中所有可用的 ApplicationContext-Initializer，然后遍历调用这些 ApplicationContextInitializer 的 initialize（applicationContext）方法来对已经创建好的 ApplicationContext 进行进一步的处理。

8）遍历调用所有 SpringApplicationRunListener 的 contextPrepared()方法。

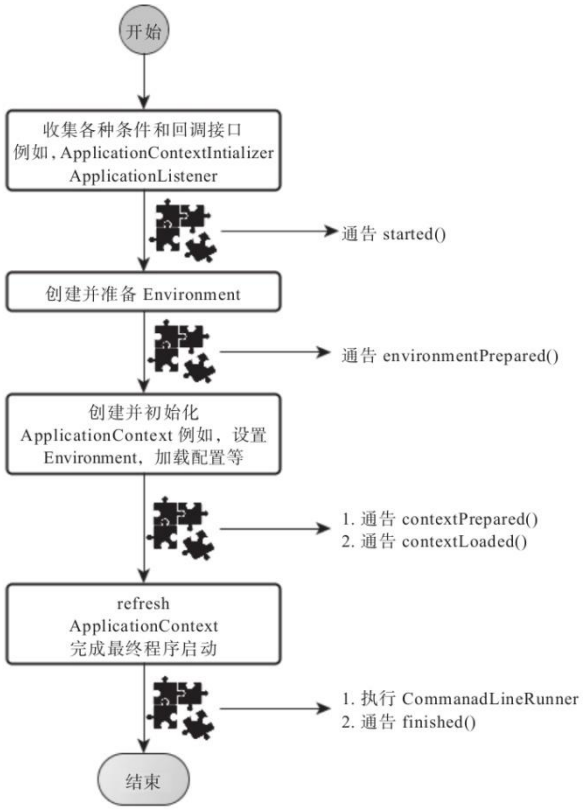
9） 最核心的一步，将之前通过@EnableAutoConfiguration 获取的所有配置以及其他形式的 IoC 容器配置加载到已经准备完毕的 ApplicationContext。

10） 遍历调用所有 SpringApplicationRunListener 的 contextLoaded()方法。

11） 调用 ApplicationContext 的 refresh()方法，完成 IoC 容器可用的最后一道工序。

12）查找当前 ApplicationContext 中是否注册有 CommandLineRunner,如果有，则遍历执行它们。

13） 正常情况下，遍历执行 SpringApplicationRunListener 的 finished()方法、（如果整个过程出现异常，则依然调用所有 SpringApplicationRunListener 的 finished()方法，只不过这种情况下会将异常信息一并传入处理）
去除事件通知点后，整个流程如下：



6.Thymeleaf

SpringBoot 官方不推荐使用 JSP，官方推荐使用 Thymeleaf。

Thymeleaf 是一款用于渲染 XML/XHTML/HTML5 内容的模板引擎。类似 JSP，Velocity，FreeMaker 等，它也可以轻易的与 Spring MVC 等 Web 框架进行集成作为 Web 应用的模板引擎。与其它模板引擎相比，Thymeleaf 最大的特点是能够直接在浏览器中打开并正确显示模板页面，而不需要启动整个 Web 应用。

6.1 搭建示例工程

引入 thymeleaf 的包：

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

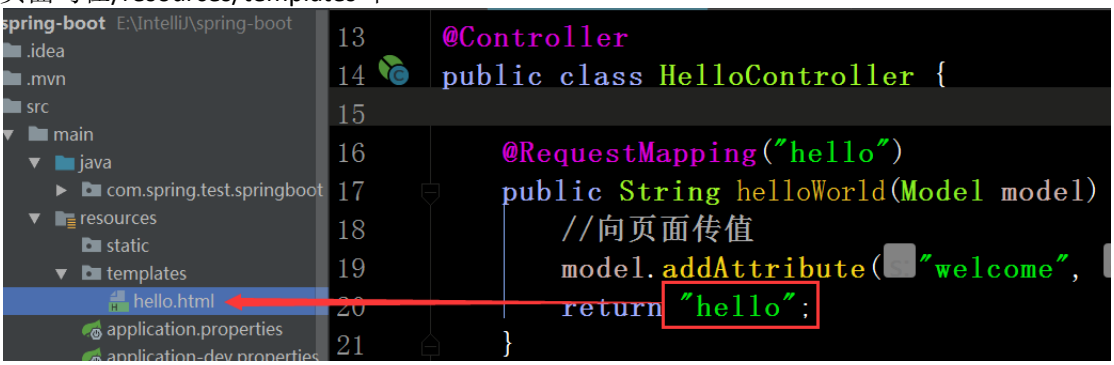
在 application.properties 文件中配置 thymeleaf 的视图解析：

```
spring.thymeleaf.content-type=text/html
spring.thymeleaf.mode =LEGACYHTML5
#开发时关闭缓存,不然没法看到实时页面
spring.thymeleaf.cache=false
#配置静态资源路径
spring.mvc.static-path-pattern=/static/**
```

controller 中的代码和以前的项目一样：

```
@RequestMapping("hello")
public String helloWorld(Model model) {
    //向页面传值
    model.addAttribute("welcome", "hello thymeleaf");
    return "hello";
}
```

页面写在/resources/templates 下



页面 hello.html，页面的文件名与 controller 中方法的返回值一致。注意页面的<html>标签中有一个<html xmlns:th="http://www.thymeleaf.org">，

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    <title>Title</title>
</head>
<body>
<p th:text="${welcome}"></p>
</body>
</html>
```

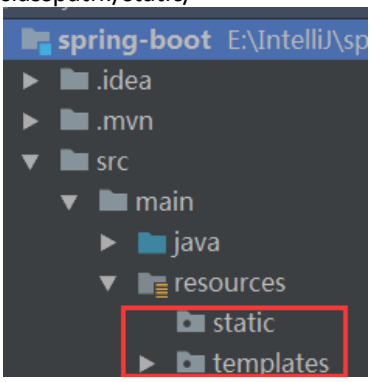
页面中所有动态的内容都使用“th:”前缀。

并且在 thymeleaf 的页面中，html 语法要求很严格，比如标签必须闭合。如果要在解析时自动进行标签补全，需要引入 jar 包：

```
<dependency>
    <groupId>net.sourceforge.nekohtml</groupId>
    <artifactId>nekohtml</artifactId>
    <version>1.9.22</version>
</dependency>
```

6.2 基础语法

spring-boot 很多配置都有默认配置,比如默认页面映射路径为
classpath:/templates/*.html
同样静态文件路径为
classpath:/static/



首先页面的<html>标签要改写：
<html xmlns:th="http://www.thymeleaf.org">

6.2.1 获取变量值

thymeleaf 通过${变量名.属性名}来获取属性值，这个语法和 EL 表达式一样。
页面中所有动态的内容都使用“th:”前缀，并且要写在标签中。
this is tag p
如果直接访问静态页面，会显示“this is tag p”
如果访问动态内容，那么${message}的值会替换掉原来 标签中的静态内容。
常见页面操作如下：

```
@RequestMapping("hello")
public String helloWorld(Model model) {
    //向页面传值,普通文本
    model.addAttribute("text", "hello thymeleaf");
    //html 转义文本
    model.addAttribute("htmlText", "<h1>html</h1>");
    model.addAttribute("ahref", "test");
    List<String> list = new ArrayList<>();
    list.add("a");
    list.add("b");
    model.addAttribute("list", list);

    List<Dept> deptList = new ArrayList<>();
    deptList.add(new Dept(1, "技术部"));
    deptList.add(new Dept(2, "测试部"));
    deptList.add(new Dept(3, "行政部"));
    model.addAttribute("deptList", deptList);
    return "hello";
}
```

```
<p th:text="${text}">我是文本</p>
<p th:utext="${htmlText}">我是转义文本</p>
<p><a th:href="@{{ahref}?pa={text}(ahref=${ahref},text=${text})}">我是a标签</a></p>
我是表格<br/>
<table border="1">
    <tr th:each="dept:${deptList}">
        <td th:text="${dept.id}">id</td>
        <td th:text="${dept.name}">name</td>
    </tr>
</table>
我是下拉框
    <select>
        <option th:each="dept:${deptList}" th:value="${dept.id}"
th:text="${dept.name}" th:selected="${dept.id}==${param.id[0]}"></option>
</select><br/>
    <input th:value="${text}">
<script th:src="@{static/test.js}" type="text/javascript"></script>
```

6.2.2 条件判断

```
<div th:if="${ahref == 'test'}">xxxxxxx</div>
```

6.2.3 其它

格式化日期：

```
<span th:text="${#dates.format(dir.updateTime, 'yyyy-MM-dd HH:mm:ss')}"></span>
```

获取相对路径：

```
<script th:src="@{{path}/static/jquery-1.10.2.min.js (path=${contextPath})}" type="text/javascript"></script>
```

2.x 版本

```
${#httpServletRequest.getContextPath()}
```

script 中获取相对路径：

```
<script type="text/javascript" th:inline="javascript">
    var basePath = [[${#httpServletRequest.getContextPath()}]];
```

如果页面中有 javascript 脚本，脚本中有 json 格式，页面会报错，需要在 js 中加：
<script type="text/javascript" th:inline="none">
绑定事件：
th:onclick="$('#main-content').load([[${child.url}]])"
如果函数中有表达式，是用[[表达式]]