SpringBoot

第一章 Spring Boot 框架初相识

1-1 Spring Boot 简介

1、Spring boot 是 Spring 家族中的一个全新的框架，它用来简化 Spring 应用程序的创建和开发过程，也可以说 Spring boot 能简化我们之前采用 Spring mvc + Spring + MyBatis（SSM） 框架进行开发的过程；

2、在以往我们采用 Spring mvc + Spring + MyBatis 框架进行开发的时候，搭建和整合三大框架，我们需要做很多工作，比如配置 web.xml，配置 Spring，配置 MyBatis，并将它们整合在一起等，而 Spring boot 框架对此开发过程进行了革命性的颠覆，抛弃了繁琐的 xml 配置过程，采用大量的默认配置简化我们的开发过程；

3、所以采用 Spring boot 可以非常容易和快速地创建基于 Spring 框架的应用程序，它让编码变简单了，配置变简单了，部署变简单了，监控变简单了；

4、正因为 Spring boot 它化繁为简，让开发变得极其简单和快速，所以在业界备受关注；

5、Spring boot 在国内的关注趋势图：http://t.cn/ROQLquP

1-2 Spring Boot 的特性

1、能够快速创建基于 Spring 的应用程序；

2、能够直接使用 java main 方法启动内嵌的 Tomcat 服务器运行 Spring boot 程序，不需要单独部署到外部的 tomcat 中运行；

3、提供约定的 starter POM 来简化 Maven 配置，让 Maven 的配置变得简单；

4、根据项目的 Maven 依赖配置，Spring boot 自动配置 Spring、Spring mvc 等；

5、提供了程序的健康检查等功能；

6、基本可以完全不使用 XML 配置文件，采用注解配置；

1-3 Spring Boot 四大核心

1、自动配置：针对很多 Spring 应用程序和常见的应用功能，Spring Boot 能自动提供相关配置；

2、起步依赖：告诉 Spring Boot 需要什么功能，它就能引入需要的依赖库；

3、Actuator：让你能够深入运行中的 Spring Boot 应用程序，一探 Spring boot 程序的内部信息；

4、命令行界面：这是 Spring Boot 的可选特性，主要针对 Groovy 语言使用；

注：Groovy 是一种基于 JVM (Java 虚拟机) 的敏捷开发语言；
它结合了 Python、Ruby 和 Smalltalk 的许多强大的特性，Groovy 代码能够与

Java 代码很好地结合，也能用于扩展现有代码；
由于其运行在 JVM 上的特性，Groovy 可以使用其他 Java 语言编写的库；

第二章 Spring Boot 框架初体验

2-1 Spring Boot 基础开发环境

1、Spring boot 目前分为两大版本系列，1.x 系列和 2.x 系列，目前 Spring Boot 最新正式版为 2.0.3.RELEASE；

2、如果是使用 eclipse，推荐安装 Spring Tool Suite (STS) 插件；

3、如果使用 IDEA 旗舰版，自带了 Springboot 插件；

4、推荐使用 Maven 3.0+，Maven 目前最新版本为 3.5.2；

5、推荐使用 Java 8，Spring boot 1.x 系列的版本兼容 Java 6，Spring boot 2.x 系列需要至少 Java 8；

2-2 第一个 Spring Boot 程序

快速开发一个 Spring boot 程序步骤如下：

1、创建一个 Spring boot 项目；

（1）可以采用方式一： 使用 eclipse 的 Spring Tool Suite (STS) 插件或者 IDEA 自带的插件创建；

（2）可以采用方式二：直接使用 Maven 创建项目的方式创建；

2、加入 Spring boot 的父级和起步依赖；

（1）父级依赖：

```
<parent>
```

```
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.0.3.RELEASE</version>
    <relativePath />
</parent>
```

加入 Spring boot 父级依赖可以简化我们项目的 Maven 配置；

（2）起步依赖：

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

加入 Spring boot 的起步依赖也可以简化我们项目的 Maven 配置；

3、创建 Spring boot 的入口 main 方法

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

4、创建一个 Spring mvc 的 Controller

```
@Controller
public class HelloController {
    @RequestMapping("/sayHi")
    public @ResponseBody String sayHi () {

        return "Hi, Spring boot";

    }
}
```

5、运行 Spring boot 的入口 main 方法

通过 eclipse、idea 右键运行 main 方法；

至此，第一个 Spring boot 程序开发完成；

2-3 第一个 Spring Boot 程序解析

1、Spring Boot 的父级依赖spring-boot-starter-parent配置之后，当前的项目就是 Spring Boot 项目；

2、spring-boot-starter-parent 是一个特殊的 starter 依赖，它用来提供相关的 Maven 默认依赖，使用它之后，常用的 jar 包依赖可以省去 version 配置；

3、Spring Boot 提供了哪些默认 jar 包的依赖，可查看该父级依赖的 pom 文件；

4、如果不想使用某个默认的依赖版本，可以通过 pom.xml 文件的属性配置覆盖各个依赖项，比如覆盖 Spring 版本：

```
<properties>
    <spring.version>4.3.10.RELEASE</spring.version>
</properties>
```

5、@SpringBootApplication 注解是 Spring Boot 项目的核心注解，主要作用是开启 Spring 自动配置；

6、main 方法是一个标准的 Java 程序的 main 方法，主要作用是作为项目启动运行的入口；

7、@Controller 及 @ResponseBody 依然是我们之前的 Spring mvc，因为 Spring boot 的里面依然是使用我们的 Spring mvc + Spring + MyBatis 等框架；

2-4 Spring Boot 的属性配置文件

Spring boot 的核心配置文件用于配置 Spring boot 程序，有两种格

式的配置文件：

1、.properties 文件

键值对的 properties 属性配置文件；

2、.yml 文件

一种 yaml 格式的配置文件；

.properties 配置举例：

```
#配置内嵌的服务器端口
server.port=8080
#配置应用访问路径
server.servlet.context-path=/springboot-web
```

2-5 Spring Boot 的.yml 配置文件

yml 是一种 yaml 格式的配置文件，主要采用一定的空格、换行等格式排版进行配置；

yaml 是一种直观的能够被计算机识别的的数据序列化格式，容易被人类阅读，yaml 类似于 xml，但是语法比 xml 简洁很多；

值与前面的冒号配置项必须要有一个空格；

yml 后缀也可以使用 yaml 后缀；

```
server:
  port: 9090
  servlet:
      context-path: /springboot-web
```

2-6 Spring Boot 多环境配置文件

多环境配置文件是指当我们项目中有多套配置文件时，在运行的时候

究竟使用哪一套配置？SpringBoot 给我们提供了一种配置方式，可以指定激活使用哪一套文件；

```
#比如配置开发环境
spring.profiles.active=dev
application-dev.properties
#比如配置生产环境
spring.profiles.active=product
application-product.properties
```

2-7 Spring boot 自定义配置文件

我们可以在 Spring boot 的核心配置文件中自定义配置，然后采用如下注解去读取配置的属性值；

1、@Value 注解

用于逐个读取自定义的配置，比如：

```
@Value("${wkcto.site}")
private String site;

@Value("${wkcto.tel}")
private String tel;
```

2、@ConfigurationProperties

用于将整个文件映射成一个对象，比如：

```
@Component
@ConfigurationProperties(prefix="wkcto")
public class MyConfig {
    private String site;
    private String tel;
    public String getSite() {
        return name;
    }
    public void setSite(String site) {
        this.site = site;
```

```
    }
    public String getTel() {
        return tel;
    }
    public void setTel(String tel) {
        this.tel = tel;
    }
}
```

2-8 Spring Boot 非 web 应用程序

在 Spring Boot 框架中，要创建一个非 Web 应用程序（纯 Java 程序）：

方式一：

创建纯 Java 项目的起步依赖：

```
<!-- Springboot 开发 java 项目的起步依赖 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
</dependency>
```

直接在 main 方法中，根据 SpringApplication.run()方法获取返回的 Spring 容器对象，再获取业务 bean 进行调用；

```
public static void main(String[] args) {
    ApplicationContext context =
    SpringApplication.run(Application.class, args);
    HelloService helloService =
    (HelloService)context.getBean("helloService");
    String hi = helloService.getMessage("springboot main");
    System.out.println(hi);
}
```

方式二：

1、Spring boot 的入口类实现 CommandLineRunner 接口；

2、覆盖 CommandLineRunner 接口的 run()方法，run 方法中编写

具体的处理逻辑即可；

```
@Autowired
private HelloService helloService;
@Override
public void run(String... args) throws Exception {
    System.out.println("hello world!");
    String ss = helloService.getMessage("aaa111");
    System.out.println(ss);
}
```

2-9 Spring Boot 日志 LOGO

关闭 spring logo 图标 日志输出：

```
SpringApplication springApplication =
new SpringApplication(Application.class);
springApplication.setBannerMode(Banner.Mode.OFF);
springApplication.run(args);
```

如何修改启动的 logo 日志：

在 src/main/resources 放入 banner.txt 文件

利用网站生成图标：http://patorjk.com/software/taag/

将生成好的图标文字粘贴到 banner.txt 文件中；

2-10 Spring Boot 热部署插件

在实际开发中，我们修改某些代码逻辑功能或页面都需要重启应用，

这无形中降低了开发效率；

热部署是指当我们修改代码后，服务能自动重启加载新修改的内容，

这样大大提高了我们开发的效率；

Spring boot 热部署通过添加一个插件实现；

插件为：spring-boot-devtools，在 Maven 中配置如下：

```
<!-- springboot 开发自动热部署 -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
    <optional>true</optional>
</dependency>
```

注意：该热部署插件在实际使用中会有一些小问题，有时候明明已经重启，但没有生效，这种情况下，手动重启一下程序；

第三章 Spring Boot 框架 Web 开发

3-1 Spring Boot 使用 Spring MVC

Spring boot 下的 Spring mvc 和之前的 Spring mvc 使用是完全一样的：

@Controller

即为 Spring mvc 的注解，处理 http 请求；

@RestController

Spring 4 后新增注解，是@Controller 与@ResponseBody 的组合注解，用于返回字符串或 json 数据；

@RequestMapping

接收请求路径映射；

@RequestParam

接收请求参数；

@ResponseBody

返回字符串或 json 格式的数据，不返回 JSP；

3-2 Spring Boot 使用 JSP

在 Spring boot 中使用 jsp，按如下步骤进行：

1、在 pom.xml 文件中配置依赖项

```
<!--引入 Spring Boot 内嵌的 Tomcat 对 JSP 的解析包-->
<dependency>
    <groupId>org.apache.tomcat.embed</groupId>
    <artifactId>tomcat-embed-jasper</artifactId>
</dependency>
<!-- servlet 依赖的 jar 包 start -->
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>javax.servlet-api</artifactId>
</dependency>
<!-- servlet 依赖的 jar 包 start -->

<!-- jsp 依赖 jar 包 start -->
<dependency>
   <groupId>javax.servlet.jsp</groupId>
   <artifactId>javax.servlet.jsp-api</artifactId>
   <version>2.3.1</version>
</dependency>
<!-- jsp 依赖 jar 包 end -->

<!--jstl 标签依赖的 jar 包 start -->
<dependency>
   <groupId>javax.servlet</groupId>
   <artifactId>jstl</artifactId>
</dependency>
<!--jstl 标签依赖的 jar 包 end -->
```

2、在 application.properties 文件配置 spring mvc 视图展示为 jsp：

```
spring.mvc.view.prefix=/
```

```
spring.mvc.view.suffix=.jsp
```

3、在 src/main 下创建 webapp 目录，并在该目录下新建 jsp 页面；

4、配置 pom.xml 的 resources：

```
<resources>
    <resource>
        <directory>src/main/java</directory>
        <includes>
            <include>**/*.xml</include>
        </includes>
    </resource>
    <resource>
        <directory>src/main/resources</directory>
        <includes>
            <include>**/*.*</include>
        </includes>
    </resource>
    <resource>
        <directory>src/main/webapp</directory>
        <targetPath>META-INF/resources</targetPath>
        <includes>
            <include>**/*.*</include>
        </includes>
    </resource>
</resources>
```

3-3 Spring Boot 使用 Interceptor

1、按照 Spring mvc 的方式编写一个拦截器类；

2、编写一个配置类 implements WebMvcConfigurer 接口

3、为该配置类添加@Configuration 注解，标注此类为一个配置类，让 Spring boot 扫描到；

4、覆盖其中的方法并添加已经编写好的拦截器：

```
@Override
public void addInterceptors(InterceptorRegistry registry) {
    //对/api/** 和 /api/login 链接的请求进行拦截
```

```
    registry.addInterceptor(new
LoginInterceptor()).addPathPatterns("/api/**").excludePathPatt
erns("/api/login");
}
```

3-4 Spring Boot 使用 Servlet

方式一

通过注解方式实现;

1、使用 Servlet3 的注解方式编写一个 Servlet

```
@WebServlet(urlPatterns="/myServlet")
public class MyServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        resp.getWriter().print("hello word");
        resp.getWriter().flush();
        resp.getWriter().close();
    }
    @Override
    protected      void      doPost(HttpServletRequest      req,
HttpServletResponse resp) throws ServletException, IOException {
        this.doGet(req, resp);
    }
}
```

2、在 main 方法的主类上添加注解:

```
@ServletComponentScan(basePackages="com.wkcto.servlet")
```

方式二

通过 Spring boot 的配置类实现;

1、编写一个普通的 Servlet

```
public class HeServlet extends HttpServlet {
    @Override
    public void doGet(HttpServletRequest req, HttpServletResponse
resp) throws ServletException, IOException {
        resp.getWriter().print("hello word");
        resp.getWriter().flush();
```

```
        resp.getWriter().close();
    }
    @Override
    protected      void      doPost(HttpServletRequest      req,
HttpServletResponse resp) throws ServletException, IOException {
        this.doGet(req, resp);
    }
}
```

2、编写一个 Springboot 的配置类;

```
@Configuration
public class ServletConfig {
    @Bean
    public ServletRegistrationBean heServletRegistrationBean(){
        ServletRegistrationBean      registration      =      new
ServletRegistrationBean(new HeServlet(), "/servlet/heServlet");
        return registration;
    }
}
```

3-5 Spring Boot 使用 Filter

方式一

通过注解方式实现;

1、编写一个 Servlet3 的注解过滤器;

```
@WebFilter(urlPatterns="/*")
public class MyFilter implements Filter {
    @Override
    public    void    init(FilterConfig    filterConfig)    throws
ServletException {
    }
    @Override
    public void doFilter(ServletRequest request, ServletResponse
response, FilterChain chain)
            throws IOException, ServletException {

        System.out.println("您已进入 filter 过滤器...");

        chain.doFilter(request, response);
    }
    @Override
```

```
    public void destroy() {
    }
}
```

2、在 main 方法的主类上添加注解：

```
@ServletComponentScan(basePackages={"com.wkcto.springboot.servlet", "com.wkcto.springboot.filter"})
```

方式二

通过 Spring boot 的配置类实现；

1、编写一个普通的 Filter

```
public class HeFilter implements Filter {
    @Override
    public void init(FilterConfig filterConfig) throws ServletException {
    }
    @Override
    public void doFilter(ServletRequest request, ServletResponse response, FilterChain chain)
            throws IOException, ServletException {
        System.out.println("he 您已进入 filter 过滤器...");
        chain.doFilter(request, response);
    }
    @Override
    public void destroy() {
    }
}
```

2、编写一个 Springboot 的配置类；

```
@Configuration
public class ServletConfig {
    @Bean
    public FilterRegistrationBean heFilterRegistration() {
        FilterRegistrationBean registration = new FilterRegistrationBean(new HeFilter());
        registration.addUrlPatterns("/*");
        return registration;
    }
}
```

3-6 Spring Boot 配置字符编码

1、第一种方式是使用传统的 Spring 提供给的字符编码过滤器：

```
@Bean
public FilterRegistrationBean filterRegistrationBean() {
    FilterRegistrationBean registrationBean = new
FilterRegistrationBean();
    CharacterEncodingFilter characterEncodingFilter = new
CharacterEncodingFilter();
    characterEncodingFilter.setForceEncoding(true);
    characterEncodingFilter.setEncoding("UTF-8");
    registrationBean.setFilter(characterEncodingFilter);
    registrationBean.addUrlPatterns("/*");
    return registrationBean;
}
```

注意：只有当 spring.http.encoding.enabled=false 配置成 false 后，过滤器才会起作用；

2、第二种方式是在 application.properties 中配置字符编码：

从 springboot1.4.2 之后开始新增的一种字符编码设置；

```
spring.http.encoding.charset=UTF-8
spring.http.encoding.enabled=true
spring.http.encoding.force=true
```

第四章 Spring Boot 集成 MyBatis

4-1 Spring Boot 集成 MyBatis 配置

Spring boot 集成 MyBatis 的步骤如下：

1、在 pom.xml 中配置相关 jar 依赖；

```
<!-- 加载 mybatis 整合 springboot -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
```

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

<!-- MySQL 的 jdbc 驱动包 -->
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
</dependency>
```

2、在 Springboot 的核心配置文件 application.properties 中配置数据源：

```
spring.datasource.username=root
spring.datasource.password=123456
spring.datasource.driver-class-name=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://192.168.230.128:3306/workdb
?useUnicode=true&characterEncoding=utf8&useSSL=false
```

3、在 MyBatis 的 Mapper 接口中添加@Mapper 注解；

或者 在运行的主类上添加

@MapperScan("com.wkcto.springboot.mapper") 注解包扫描；

4-2 Spring Boot 集成 MyBatis 测试

启动 SpringBoot 程序测试

4-3 Spring Boot 事务管理

Spring Boot 使用事务非常简单，底层依然采用的是 Spring 本身提供的事务管理；

1、在入口类中使用注解 **@EnableTransactionManagement** 开启事务支持；

2、在访问数据库的 Service 方法上添加注解 **@Transactional** 即

可；

第五章 Spring Boot 集成 Redis

5-1 Spring Boot 集成 Redis 步骤

1、在 pom.xml 中配置相关的 jar 依赖；

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

2、在 Springboot 核心配置文件 application.properties 中配置 redis 连接信息：

```
spring.redis.host=192.168.230.128
spring.redis.port=6379
spring.redis.password=123456
```

3、配置了上面的步骤，Spring boot 将自动配置 RedisTemplate，在需要操作 redis 的类中注入 redisTemplate;

在使用的类中注入：

```
@Autowired
private RedisTemplate<String, String> redisTemplate;
@Autowired
private RedisTemplate<Object, Object> redisTemplate;
```

spring boot 帮我们注入的 redisTemplate 类，泛型里面只能写 <String, String>、<Object, Object>

5-2 Spring Boot 集成 Redis 测试

启动程序，进行访问测试；

设置 key 的序列化方式为字符串，增强 key 的可读性；

5-3 高并发条件下缓存穿透问题分析

在项目中使用缓存通常是先检查缓存中是否存在，如果存在直接返回缓存内容，如果不存在就直接查询数据库，然后将查询出来的数据缓存到缓存中，最终返回查询结果；

但是如果大量用户请求查询的某一个数据，而该数据在缓存中不存在，就会造成大量的用户请求都去查询 DB，这样缓存就失去了意义，在并发流量大时，可能导致 DB 压力过大而失去响应；

5-4 高并发条件下缓存穿透问题复现

按照常规的代码实现方式，多线程并发条件下多个请求落入到了数据库；

5-5 高并发条件下缓存穿透问题处理

通过锁机制，避免请求穿透缓存直接落入到数据库；

5-6 高并发条件下缓存穿透问题测试

运行程序，观察是否有多个请求落入到数据库；

5-7 SpringBoot 集成 Redis 哨兵模式

SpringBoot 配置：

```
redis.password=123456
redis.sentinel.master=mymaster
redis.sentinel.nodes=192.168.179.128:26380,192.168.179.128:263
82,192.168.179.128:26384
```

配置 Redis 主从模式

```
include /usr/local/redis-3.2.9/redis.conf
port 6380
pidfile "/var/run/redis_6380.pid"
logfile "/var/run/6380.log"
dir "/run"
dbfilename "dump6380.rdb"
daemonize yes
protected-mode no
requirepass "123456"
masterauth "123456"
```

```
include /usr/local/redis-3.2.9/redis.conf
port 6381
pidfile "/var/run/redis_6381.pid"
logfile "/var/run/6381.log"
dir "/run"
dbfilename "dump6381.rdb"
daemonize yes
protected-mode no
requirepass "123456"
masterauth "123456"
```

```
include /usr/local/redis-3.2.9/redis.conf
port 6382
pidfile "/var/run/redis_6382.pid"
logfile "/var/run/6382.log"
dir "/run"
dbfilename "dump6382.rdb"
slaveof 192.168.93.128 6380
daemonize yes
protected-mode no
requirepass "123456"
masterauth "123456"
```

配置 Redis 哨兵模式

```
protected-mode no
sentinel monitor mymaster 192.168.230.128 6380 2
sentinel auth-pass mymaster 123456
```

第六章 Spring Boot 集成 Dubbo

6-1 集成前的准备

1、阿里巴巴提供的 dubbo 集成 springboot 开源项目；

https://github.com/alibaba

2、我们将采用该项目提供的 jar 包进行集成；

```
<!--添加 dubbo 集成 springboot 依赖-->
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

6-2 服务接口项目的开发

按照 Dubbo 官方开发建议，创建一个接口项目，该项目只定义接口和 model 类；

```
public interface UserService {
    public String sayHi (String name);
}
```

6-3 服务提供者的开发

1、创建一个 Springboot 项目并配置好相关的依赖；

2、加入 springboot 与 dubbo 集成的起步依赖：

```
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

3、在 Springboot 的核心配置文件 application.properties 中配置 dubbo 的信息：

```
server.port=9090
spring.application.name=z-miaosha-service
spring.dubbo.appname=z-miaosha-service
spring.dubbo.registry=zookeeper://192.168.230.128:2181
```

由于使用了 zookeeper 作为注册中心，则需要加入 zookeeper 的客户端 jar 包：

```
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
```

4、编写 Dubbo 的接口实现类：

```
@Service //该注解是 dubbo 的
@Component //该注解是 spring 的
public class UserServiceImpl implements UserService {
    @Override
    public String sayHi(String name) {
        return "Hi, " + name;
    }
}
```

5、编写一个入口 main 程序启动 Dubbo 服务提供者：

```
@SpringBootApplication
@EnableDubboConfiguration //开启 dubbo 配置支持
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

6-4 服务消费者的开发

1、创建一个 Springboot 项目并配置好相关的依赖；

2、加入 springboot 与 dubbo 集成的起步依赖：

```
<dependency>
    <groupId>com.alibaba.spring.boot</groupId>
    <artifactId>dubbo-spring-boot-starter</artifactId>
    <version>2.0.0</version>
</dependency>
```

3、在 Springboot 的核心配置文件 application.properties 中配置 dubbo 的信息：

```
# WEB 服务端口
server.port=9090
# dubbo 配置
spring.application.name=dubbo-spring-boot-starter
spring.dubbo.appname=springboot-dubbo-consumer
spring.dubbo.registry=zookeeper://192.168.91.129:2181
```

由于使用了 zookeeper 作为注册中心，则需要加入 zookeeper 的客户端 jar 包：

```
<dependency>
    <groupId>com.101tec</groupId>
    <artifactId>zkclient</artifactId>
    <version>0.10</version>
</dependency>
```

4、编写一个 Controller 类，调用远程的 Dubbo 服务：

```
@Controller
public class UserController {
    @Reference //使用 dubbo 的注解引用远程的 dubbo 服务
    private UserService userService;
    @RequestMapping("/sayHi")
    public @ResponseBody
    String sayHi () {
        return userService.sayHi("spring boot dubbo.....");
    }
}
```

5、编写一个入口 main 程序启动 Dubbo 服务提供者：

```
@SpringBootApplication
@EnableDubboConfiguration //开启 dubbo 配置支持
public class SpringbootApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringbootApplication.class, args);
    }
}
```

6-5 整体集成测试

运行服务提供者、服务消费者、整体测试；

第七章 Spring Boot 实现 RESTfull

7-1 认识 RESTFull

RESTFull 一种互联网软件架构设计的风格，但它并不是标准，它只是提出了一组客户端和服务器交互时的架构理念和设计原则，基于这种理念和原则设计的接口可以更简洁，更有层次；

任何的技术都可以实现这种理念，如果一个架构符合 REST 原则，就称它为 RESTFull 架构；

REST 这个词，最早是由 Roy Thomas Fielding 在他 2000 年的博士论文中提出的；

比如我们要访问一个 http 接口：

http://localhost:8080/api/order?id=1521&status=1

采用 RESTFull 风格则 http 地址为：

http://localhost:8080/api/order/1521/1

7-2 Spring Boot 开发 RESTFull

Spring boot 开发 RESTFull 主要是一个核心的注解实现：

@PathVariable

获取 url 中的数据；

该注解是实现 RESTFull 最主要的一个注解；

7-3 Spring Boot 下的 HTTP 请求注解

@GetMapping

RequestMapping 和 Get 请求方法的组合；

@PostMapping

RequestMapping 和 Post 请求方法的组合；

@PutMapping

RequestMapping 和 Put 请求方法的组合；

@DeleteMapping

RequestMapping 和 Delete 请求方法的组合；

在 Restfull 理念中：

1、@PostMapping

接收和处理 Post 方式的请求，增加操作；

2、@DeleteMapping

接收 delete 方式的请求，删除操作，可以使用 GetMapping 代替；

3、@PutMapping

接收 put 方式的请求，修改操作，可以用 PostMapping 代替；

4、@ GetMapping

接收 get 方式的请求，查询操作；

第八章 Spring Boot 打包与部署

8-1 Spring Boot 程序打 war 包

1、程序入口类需扩展继承 SpringBootServletInitializer 类；

2、程序入口类覆盖如下方法：

```
@Override
protected SpringApplicationBuilder
configure(SpringApplicationBuilder application) {
    return application.sources(SpringbootApplication.class);
}
```

3、更新打包方式包为 war，在 pom.xml 中修改

```
<packaging>war</packaging>
```

4、配置 springboot 打包的插件

```
<!-- Springboot 打包的插件 -->
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
</plugin>
```

5、在项目中通过 Maven install 在本地 maven 仓库安装成一个 war 包，然后将 war 包部署到 tomcat 下运行；

8-2 Spring Boot 程序打 Jar 包

1、Spring boot 程序打包，在 pom.xml 文件加入如下 Spring boot 的 maven 插件：

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

上面这个插件打包有小问题（插件使用 1.4.2 版本，其他版本目前测试有问题）

2、在项目中使用 Maven install 在本地 maven 仓库安装成一个 jar；

3、使用 java -jar 运行第 2 步生成的 jar 包，从而可以启动 Spring boot 程序；

4、访问第 3 步运行起来的 spring boot 程序；

8-3 Spring boot 程序部署运行总结

1、在 IDEA 中直接运行 spring boot 程序的 main 方法 (开发阶段)；

2、用 maven 将 spring boot 安装为一个 jar 包，使用 Java 命令运行：java -jar spring-boot-xxx.jar

可以将该命令封装到一个 Linux 的一个 shell 脚本中（上线部署）

3、使用 Spring boot 的 maven 插件将 Springboot 程序打成 war 包，单独部署在 tomcat 中运行（上线部署）；

第九章 Spring Boot 服务监控

9-1 Spring boot Actuator 简介

在生产环境中，需要实时或定期监控服务的可用性，spring-boot 的 actuator 功能提供了很多监控所需的接口；

actuator 是 spring boot 提供的对应用系统的自省和监控的集成功能，可以对应用系统进行配置查看、健康检查、相关功能统计等；

9-2 Spring boot Actuator 集成配置

如何使用该功能呢?

1、在项目的 Maven 中添加如下依赖：

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

2、application.properties 或 application.yml 配置文件中指定监控的 HTTP 端口；

如果不指定，则使用和 server 相同的端口，比如：

```
#服务运行的端口
server.port=8080
#actuator 监控的端口（端口可配可不配）
management.server.port=8100
#actuator 监控访问的上下文（上下文可配可不配）
management.server.servlet.context-path=/springboot-web
#默认只开启了 health 和 info，设置为*，则包含所有 we 入口端点
management.endpoints.web.exposure.include=*
```

访问举例：

http://localhost:8080/springboot-web/actuator/health

9-3 Spring boot Actuator 监控接口

actuator 提供的主要功能：

HTTP 方法 路径 描述 是否为 web 入口

GET /configprops 查看配置属性，包括默认配置 true

GET /beans 查看 bean 及其关系列表 true

GET /mappings 查看所有 url 映射 true

GET /env 查看所有环境变量 true

GET /health 查看应用健康指标 false

GET /info 查看应用信息 false

GET /metrics 查看应用基本指标 true

GET /metrics/{name} 查看具体指标 true

JMX /shutdown 关闭应用 true

其中：

1、/shutdown 需要在配置文件中开启才能生效：

```
management.endpoint.shutdown.enabled=true
```

2、/info 需要自己在 application.properties 配置文件中添加信息：

```
info.contact.email=wkcto@wkcto.com
info.contact.phone=010-84846003
```

然后请求才会有数据；

第十章 Spring Boot 集成 Thymeleaf

10-1 认识 Thymeleaf

Thymeleaf 是一个流行的模板引擎，该模板引擎采用 Java 语言开发；

模板引擎是一个技术名词，是跨领域跨平台的概念，在 Java 语言体系下有模板引擎，在 C#、PHP 语言体系下也有模板引擎，甚至在 JavaScript 中也会用到模板引擎技术；

Java 生态下的模板引擎有 Thymeleaf 、Freemaker、Velocity、Beetl（国产） 等；

Thymeleaf 模板既能用于 web 环境下，也能用于非 web 环境下，在非 web 环境下，它能直接显示模板上的静态数据，在 web 环境下，它能像 JSP 一样从后台接收数据并替换掉模板上的静态数据；

Thymeleaf 它是基于 HTML 的，以 HTML 标签为载体，Thymeleaf 要寄托在 HTML 的标签下实现对数据的展示；

Thymeleaf 的官方网站：http://www.thymeleaf.org

Spring boot 集成了 thymeleaf 模板技术，并且 spring boot 官方也推荐使用 thymeleaf 来替代 JSP 技术；

thymeleaf 是另外的一种模板技术，它本身并不属于 springboot，springboot 只是很好地集成这种模板技术，作为前端页面的数据展示；

10-2 Spring boot 集成 Thymeleaf

1、第一步：在 Maven 中引入 Thymeleaf 的依赖，加入以下依赖配置即可：

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

2、第二步：在 Spring boot 的核心配置文件 application.properties 中对 Thymeleaf 进行配置：

```
#开发阶段，建议关闭 thymeleaf 的缓存
spring.thymeleaf.cache=false
```

3、第三步：写一个 Controller 去映射到模板页面（和 SpringMVC 基本一致），比如：

```
@RequestMapping("/index")
public String index (Model model) {
    model.addAttribute("data", "Spring boot 集成 Thymeleaf! ");
    //return 中就是你页面的名字（不带.html 后缀）
    return "index";
}
```

4、第四步：在 src/main/resources 的 templates 下新建一个 index.html 页面用于展示数据：

HTML 页面的<html>元素中加入以下属性：

```
<html xmlns:th="http://www.thymeleaf.org">
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>Spring boot 集成 Thymeleaf</title>
</head>
<body>
<p th:text="${data}">Spring boot 集成 Thymeleaf</p>
</body>
</html>
```

Springboot 使用 thymeleaf 作为视图展示，约定将模板文件放置在 src/main/resource/templates 目录下，静态资源放置在 src/main/resource/static 目录下

10-3 Thymeleaf 标准表达式

Thymeleaf 的标准表达式主要有如下几类：

1、标准变量表达式

语法：${...}

变量表达式用于访问容器（tomcat）上下文环境中的变量，功能和 JSTL 中的 ${} 相同；

Thymeleaf 中的变量表达式使用 ${变量名} 的方式获取其中的数据

比如在 Spring mvc 的 Controllar 中使用 model.addAttribute 向前端传输数据，代码如下：

```
@RequestMapping(value="/userinfo")
public String userinfo (Model model) {
    User user = new User();
    user.setId(1);
    user.setNick("zhangsanf");
    user.setPhone("13700020000");
    user.setAddress("beijing");
    model.addAttribute("user", user);
    model.addAttribute("hello", "helloworld");
    return "user";
}
```

前端接收代码：

```
<td th:text="${user.nick}">x</td>
<td th:text="${user.phone}">137xxxxxxxx</td>
<td th:text="${user.email}">xxx@xx.com</td>
```

```
<td th:text="${user.address}">北京.xxx</td>
<span th:text="${hello}">你好</span>
```

其中，th:text="" 是 Thymeleaf 的一个属性，用于文本的显示；

2、选择变量表达式

语法：*{...}

选择变量表达式，也叫星号变量表达式，使用 th:object 属性来绑定对象，比如：

```
@RequestMapping(value="/userinfo")
public String userinfo (Model model) {
    User user = new User();
    user.setId(1);
    user.setNick("zhangsanfeng");
    user.setPhone("13700020000");
    user.setAddress("beijing");
    model.addAttribute("user", user);
    model.addAttribute("hello", "helloworld");
    return "user";
}
```

前端接收代码

```
<div th:object="${user}" >
    <p>nick: <span th:text="*{nick}">张</span></p>
    <p>phone: <span th:text="*{phone}" >三</span></p>
    <p>email: <span th:text="*{email}" >北京</span></p>
    <p>address: <span th:text="*{address}" >北京</span></p>
</div>
```

选择表达式首先使用 th:object 来邦定后台传来的 User 对象，然后使用 * 来代表这个对象，后面 {} 中的值是此对象中的属性；

选择变量表达式 *{...} 是另一种类似于变量表达式 ${...} 表示变量的方法；

选择变量表达式在执行时是在选择的对象上求解，而${...}是在上下文的变量 Model 上求解；

通过 th:object 属性指明选择变量表达式的求解对象；

上述代码等价于：

```
<div>
    <p>nick: <span th:text="${user.nick}">张</span></p>
    <p>phone: <span th:text="${user.phone}" >三</span></p>
    <p>email: <span th:text="${user.email}" >北京</span></p>
    <p>address: <span th:text="${user.address}" >北京</span></p>
</div>
```

标准变量表达式和选择变量表达式可以混合一起使用，比如：

```
<div th:object="${user}" >
    <p>nick: <span th:text="*{nick}">张</span></p>
    <p>phone: <span th:text="${user.phone}" >三</span></p>
    <p>email: <span th:text="${user.email}" >北京</span></p>
    <p>address: <span th:text="*{address}" >北京</span></p>
</div>
```

也可以不使用 th:object 进行对象的选择，而直接使用 *{...} 获取数据，比如：

```
<div>
    <p>nick: <span th:text="*{user.nick}">张</span></p>
    <p>phone: <span th:text="*{user.phone}" >三</span></p>
    <p>email: <span th:text="*{user.email}" >北京</span></p>
    <p>address: <span th:text="*{user.address}" >北京</span></p>
</div>
```

3、URL 表达式

语法：@{...}

URL 表达式可用于 <script src="...">、<link href="...">、<a href="...">、<form action="...">等

1、绝对 URL，比如：

```
<a th:href="@{'http://xxx/boot/info?id='+${user.id}}">查看</a>
```

2、相对 URL，相对于页面，比如：

```
<a th:href="@{'user/info?id='+${user.id}}">查看</a>
```

3、相对 URL，相对于项目上下文，比如：

```
<a th:href="@{'/user/info?id='+${user.id}}">查看</a>
```

（项目的上下文名会被自动添加）

10-4 Thymeleaf 常见属性

如下为 thymeleaf 的常见属性：

th:action

定义后台控制器的路径，类似<form>标签的 action 属性，比如：

```
<form id="login" th:action="@{/login}">......</form>
```

th:method

设置请求方法，比如：

```
<form id="login" th:action="@{/login}" th:method="post">
......
</form>
```

th:href

定义超链接，比如：

```
<a class="login" th:href="@{/login}">登录</a>
```

th:src

用于外部资源引入，比如<script>标签的 src 属性，<img>标签的 src 属性，常与@{}表达式结合使用；

```
<script th:src="@{/static/js/jquery-2.4.min.js}"></script>
<img th:src="@{/static/image/logo.png}"/>
```

th:id

类似 html 标签中的 id 属性，比如：

```
<span th:id="${hello}">aaa</span>
```

th:name

设置表单名称，比如：

```
<input th:type="text" th:id="userName" th:name="userName">
```

th:value

类似 html 标签中的 value 属性，能对某元素的 value 属性进行赋值，比如：

```
<input type="hidden" id="userId" name="userId"
th:value="${userId}">
```

th:attr

该属性也是用于给 HTML 中某元素的某属性赋值，但该方式写法不够优雅，比如上面的例子可以写成如下形式：

```
<input type="hidden" id="userId" name="userId"
th:attr="value=${userId}">
```

th:text

用于文本的显示，比如：

```
<input type="text" id="realName" name="reaName"
th:text="${realName}">
```

th:each

这个属性非常常用，比如从后台传来一个对象集合那么就可以使用此属性遍历输出，它与 JSTL 中的<c: forEach>类似，此属性既可以循环遍历集合，也可以循环遍历数组及 Map，比如：

```
<tr th:each="user, interStat : ${userlist}">
    <td th:text="${interStat.index}"></td>
    <td th:text="${user.id}"></td>
    <td th:text="${user.nick}"></td>
    <td th:text="${user.phone}"></td>
    <td th:text="${user.email}"></td>
    <td th:text="${user.address}"></td>
</tr>
```

以上代码解读如下：

th:each="user, iterStat : ${userlist}" 中的 ${userList} 是后台传来的 Key，

user 是${userList} 中的一个数据，

iterStat 是 ${userList} 循环体的信息，

其中 user 及 iterStat 自己可以随便写；

interStat 是循环体的信息，通过该变量可以获取如下信息：

index、size、count、even、odd、first、last，其含义如下：

index: 当前迭代对象的 index（从 0 开始计算）

count: 当前迭代对象的个数（从 1 开始计算）

size: 被迭代对象的大小

current: 当前迭代变量

even/odd: 布尔值，当前循环是否是偶数/奇数（从 0 开始计算）

first: 布尔值，当前循环是否是第一个

last: 布尔值，当前循环是否是最后一个

注意：循环体信息 interStat 也可以不定义，则默认采用迭代变量加上 Stat 后缀，即 userStat

Map 类型的循环：

```
<div th:each="myMapVal : ${myMap}">
    <span th:text="${myMapVal.key}"></span>
    <span th:text="${myMapVal.value}"></span>
    <br/>
</div>
```

${myMapVal.key} 是获取 map 中的 key，${myMapVal.value} 是获取 map 中的 value；

数组类型的循环：

```
<div th:each="myArrayVal : ${myArray}">
    <div th:text="${myArrayVal}"></div>
</div>
```

th:if

条件判断，比如后台传来一个变量，判断该变量的值，1 为男，2 为女：

```
<span th:if="${sex} == 1" >
   男：<input type="radio" name="se"  th:value="男" />
</span>
<span th:if="${sex} == 2">
   女：<input type="radio" name="se" th:value="女"  />
</span>
```

th:unless

th:unless 是 th:if 的一个相反操作，上面的例子可以改写为：

```
<span th:unless="${sex} == 1" >
   女：<input type="radio" name="se"  th:value="女" />
</span>
<span th:unless="${sex} == 2">
   男：<input type="radio" name="se" th:value="男"  />
</span>
```

th:switch/th:case

switch，case 判断语句，比如：

```
<div th:switch="${sex}">
    <p th:case="1">性别：男</p>
    <p th:case="2">性别：女</p>
    <p th:case="*">性别：未知</p>
</div>
```

一旦某个 case 判断值为 true，剩余的 case 则都当做 false，“*”表示默认的 case，前面的 case 都不匹配时候，执行默认的 case；

th:object

用于数据对象绑定

通常用于选择变量表达式（星号表达式）

th:onclick

点击事件，th:onclick="'getCollect()'"

th:style

设置样式，th:style="'display:none;'"

th:inline

内联文本、内联脚本

th:inline 有三个取值类型（text, javascript 和 none)

该属性使用内联表达式[[...]]展示变量数据，比如：

```
<span th:inline="text">Hello, [[${user.nick}]]</span>
```

等同于：

```
<span>Hello, <span th:text="${user.nick}"></span></span>
```

th:inline 写在任何父标签都可以，比如如下也是可以的：

```
<body th:inline="text">
...
<span>[[${user.nick}]]</span>
...
</body>
```

内联脚本

```
<script th:inline="javascript" type="text/javascript">
    var user = [[${user.phone}]];
    alert(user);
</script>
<script th:inline="javascript" type="text/javascript">
    var msg  = "Hello," + [[${user.phone}]];
    alert(msg);
</script>
```

10-5 Thymeleaf 字面量

文本字面量

用单引号'...'包围的字符串为文本字面量，比如：

```
<a th:href="@{'api/getUser?id=' + ${user.id}}">修改</a>
```

数字字面量

```
<p>今年是<span th:text="2017">1949</span>年</p>
<p>20 年后，将是<span th:text="2017 + 20">1969</span>年</p>
```

布尔字面量

true 和 false

```
<p th:if="${isFlag == true}">
    执行操作
</p>
```

null 字面量

```
<p th:if="${userlist == null}">
    userlist 为空
</p>
<p th:if="${userlist != null}">
    userlist 不为空
</p>
```

10-6 Thymeleaf 字符串拼接

一种是字面量拼接：

```
<span th:text="'当前是第'+${sex}+'页 ,共'+${sex}+'页'"></span>
```

另一种更优雅的方式，使用“|”减少了字符串的拼接：

```
<span th:text="|当前是第${sex}页，共${sex}页|"></span>
```

10-7 Thymeleaf 三元运算判断

```
<span th:text="${sex eq 1} ? '男' : '女'">未知</span>
```

10-8 Thymeleaf 运算和关系判断

算术运算：+ , - , * , / , %

关系比较: > , < , >= , <= (gt , lt , ge , le)

相等判断: == , != (eq , ne)

10-9 Thymaleaf 内置对象

1、模板引擎提供了一组内置的对象，这些内置的对象可以直接在模板中使用，这些对象由#号开始引用:

2、官方手册:

http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

#request:

相当于 HttpServletRequest 对象，这是 3.x 版本，若是 2.x 版本使用 #httpServletRequest;

```
${#request.getContextPath()}
${#request.getAttribute("phone")}
```

#session:

相当于 HttpSession 对象，这是 3.x 版本，若是 2.x 版本使用 #httpSession;

需要在后台 controller 中设置了 session

```
${#session.getAttribute("phone")}
${#session.id}
${#session.lastAccessedTime}
```

Thymaleaf 表达式功能对象

1、模板引擎提供的一组功能性内置对象，可以在模板中直接使用这些对象提供的功能方法:

2、工作中常使用的数据类型，如集合，时间，数值，可以使用 thymeleaf 的提供的功能性对象来处理它们;

3、内置功能对象前都需要加#号，内置对象一般都以 s 结尾；

4、官方手册：

http://www.thymeleaf.org/doc/tutorials/3.0/usingthymeleaf.html

#dates: java.util.Date 对象的实用方法，

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

#calendars: 和 dates 类似，但是 java.util.Calendar 对象；

#numbers: 格式化数字对象的实用方法；

#strings: 字符串对象的实用方法： contains, startsWith, prepending/appending 等；

#objects: 对 objects 操作的实用方法；

#bools: 对布尔值求值的实用方法；

#arrays: 数组的实用方法；

#lists: list 的实用方法，比如

```
<span th:text="${#lists.size(datas)}"></span>
```

#sets: set 的实用方法；

#maps: map 的实用方法；

#aggregates: 对数组或集合创建聚合的实用方法；

第十一章 Spring Boot 综合案例

10-1 综合案例需求

通过上面内容的学习，我们完成一个综合案例：

采用 Springboot + dubbo + mybatis + redis + thymeleaf 实现对数据库的增删改查以及缓存操作；

具体需求如下：

MySQL 数据库中有一张表 u_user_info；

前端使用 thymeleaf 模板技术展示数据；

后端使用 spring boot + dubbo + mybatis + redis 实现对数据库数据的增删改查以及缓存操作；

查询数据后将数据放入 redis 缓存中，减少对数据库的直接访问；

主要目的是练习 Spring Boot 如何集成各类技术进行项目开发；

10-2 综合案例实现

创建 SpringBoot 项目，整合各大技术，进行代码开发；

10-3 综合案例总结

1、采用 Spring Boot 开发实质上也是一个常规的 Spring 项目开发；

2、由于 Spring Boot 提供 main 方法启动程序和自动化配置，可以简化开发过程，提高开发效率；

3、Spring Boot 项目开发代码的实现依然是使用 Spring mvc + spring +dubbo + mybatis + redis 等，当然 Spring Boot 能集成几乎所有的开源项目；

4、采用 Spring Boot 开发，需要掌握大量的注解，所以日常开发中注意对注解的积累；