



黑马程序员
www.itheima.com

传智播客旗下
高端IT教育品牌

改变中国IT教育，我们正在行动

Spring Boot

传智播客.黑马程序员



一、Spring Boot 简介

（一）什么是 Spring Boot

Spring 诞生时是 Java 企业版 (Java Enterprise Edition , JEE , 也称 J2EE) 的轻量级代替品。无需开发重量级的 Enterprise JavaBean (EJB) , Spring 为企业级 Java 开发提供了一种相对简单的方法 , 通过依赖注入和面向切面编程 , 用简单的 Java 对象 (Plain Old Java Object , POJO) 实现了 EJB 的功能。

虽然 Spring 的组件代码是轻量级的 , 但它的配置却是重量级的。一开始 , Spring 用 XML 配置 , 而且是很多 XML 配置。Spring 2.5 引入了基于注解的组件扫描 , 这消除了大量针对应用程序自身组件的显式 XML 配置。Spring 3.0 引入了基于 Java 的配置 , 这是一种类型安全的可重构配置方式 , 可以代替 XML。

所有这些配置都代表了开发时的损耗。因为在思考 Spring 特性配置和解决业务问题之间需要进行思维切换 , 所以写配置挤占了写应用程序逻辑的时间。和所有框架一样 , Spring 实用 , 但与此同时它要求的回报也不少。

除此之外 , 项目的依赖管理也是件吃力不讨好的事情。决定项目里要用哪些库就已经够让人头痛的了 , 你还要知道这些库的哪个版本和其他库不会有冲突 , 这难题实在太棘手。并且 , 依赖管理也是一种损耗 , 添加依赖不是写应用程序代码。一旦选错了依赖的版本 , 随之而来的不兼容问题毫无疑问会是生产力杀手。

Spring Boot 让这一切成为了过去。

Spring Boot 是 Spring 社区较新的一个项目。该项目的目的是帮助开发者更容易的创建基于 Spring 的应用程序和服务 , 让更多的人更快的对 Spring 进行入



门体验，为 Spring 生态系统提供了一种固定的、约定优于配置风格的框架。

Spring Boot 具有如下特性：

- (1) 为基于 Spring 的开发提供更快的入门体验
- (2) 开箱即用，没有代码生成，也无需 XML 配置。同时也可以修改默认值来满足特定的需求。
- (3) 提供了一些大型项目中常见的非功能性特性，如嵌入式服务器、安全、指标，健康检测、外部配置等。
- (4) Spring Boot 并不是不对 Spring 功能上的增强，而是提供了一种快速使用 Spring 的方式。

(二) Spring Boot 四大神器

Spring Boot 有四大神器，分别是 auto-configuration、Starters、CLI、Actuator

(1) auto-configuration (自动配置)：针对很多 Spring 应用程序常见的应用功能，Spring Boot 能自动提供相关配置。

(2) Starters (起步依赖)：告诉 Spring Boot 需要什么功能，它就能引入需要的库。

(3) CLI (命令行界面)：这是 Spring Boot 的可选特性，借此你只需写代码就能完成完整的应用程序，无需传统项目构建。

(4) Actuator (监控工具)：让你能够深入运行中的 Spring Boot 应用程序，一探究竟。



二、Spring Boot 入门

准备工作：需要重新配置 Maven 的本地仓库，配套的资料文件夹中已经提供。

（一）起步依赖

创建 Maven 工程 spring-boot_Demo（打包方式 jar）

在 pom.xml 中添加如下依赖

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.0.RELEASE</version>
</parent>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

我们会惊奇地发现，我们的工程自动添加了好多好多 jar 包，而这些 jar 包正式我们做开发时需要导入的 jar 包。因为这些 jar 包被我们刚才引入的 spring-boot-starter-web 所引用了，所以我们引用 spring-boot-starter-web 后会自动把依赖传递过来。

（二）变更 JDK 版本

我们发现默认情况下工程的 JDK 版本是 1.6，而我们通常使用 1.7 的版本，所以我们需要在 pom.xml 中添加以下配置

```
<properties>
  <java.version>1.7</java.version>
</properties>
```




(四) Spring MVC 实现 Hello World 输出

我们现在开始使用 spring MVC 框架，实现 json 数据的输出。如果按照我们原来的做法，需要在 web.xml 中添加一个 DispatcherServlet 的配置，再添加一个 spring 的配置文件，配置文件中需要添加如下配置

```
<!-- 使用组件扫描，不用将 controller 在 spring 中配置 -->
<context:component-scan base-package="cn.itcast.demo.controller" />

<!-- 使用注解驱动不用在下边定义映射器和适配配置器 -->
<mvc:annotation-driven />

<!-- 注解适配器 -->
<bean
class="org.springframework.web.servlet.mvc.method.annotation.RequestMappingHandlerAdapter"
">
    <property name="messageConverters">
        <list>
            <bean
class="org.springframework.http.converter.json.MappingJacksonHttpMessageConverter"></bean>
        </list>
    </property>
</bean>
```

但是我们用 SpringBoot，这一切都省了。我们直接写 Controller 类

```
package cn.itcast.demo.controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
@RestController
public class HelloWorldController {

    @RequestMapping("/info")
    public String info(){
        return "HelloWorld";
    }
}
```

我们运行启动类或通过 maven 命令来运行程序



```
spring-boot:run
```

在浏览器地址栏输入 <http://localhost:8080/info> 即可看到运行结果

（五）热部署

我们在开发中反复修改类、页面等资源，每次修改后都是需要重新启动才生效，这样每次启动都很麻烦，浪费了大量的时间，能不能在我修改代码后不重启就能生效呢？可以，在 pom.xml 中添加如下配置就可以实现这样的功能，我们称之为热部署。

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

赶快试试看吧，是不是很爽。 😊

三、Spring Boot 整合

（一）整合 Spring Data JPA

需求：使用 Spring Boot + Spring MVC + Spring Data JPA + EasyUI 框架组合实现部门列表查询，效果如下：



编号	名称	电话
4	销售精英部	12121212
5	仓储部	89223322
7	财务中心	14141414
8	售后服务中心	90909090
9	市场营销部	99819182
10	策划部	80112211
11	人事科	12212121
12	行政科	111
17	审计科	12311



准备工作，在 MySQL 中创建库 erpdb ,erpdb 中创建表 dep,字段信息如下：

	Field Name	Datatype	Len	Default	PK?	Not Null?	Unsigned	Auto Incr?	Zerofill	Comment
*	uuid	bigint	20		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	部门编号
	name	varchar	30		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	部门名称
	tele	varchar	30		<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	电话

1. 后端代码

(1) 创建工程，引入起步依赖

创建 maven 的 jar 工程，在 pom.xml 中添加以下配置

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>1.4.0.RELEASE</version>
  <relativePath/>
</parent>
<properties>
  <java.version>1.7</java.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-devtools</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
  </dependency>
</dependencies>
```

(2) 创建配置文件

在 src/main/resources 下添加 application.properties 配置文件，内容如下：



```
#DB Configuration:
spring.datasource.driverClassName = com.mysql.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/erpdb
spring.datasource.username = root
spring.datasource.password = 123456

#JPA Configuration:
spring.jpa.database=MySQL
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.naming_strategy=org.hibernate.cfg.ImprovedNamingStrategy
```

此文件用于覆盖 spring boot 默认配置信息

(3) 创建实体类

创建包 cn.itcast.erp.entity，在此包下创建实体类 Dep，内容如下：

```
package cn.itcast.erp.entity;
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Dep {
    @Id
    private Long uuid;
    private String name;
    private String tele;

    //getter and setter...
}
```

(4) 创建数据访问接口

创建 cn.itcast.erp.dao 包，在此包下创建数据访问接口 DepDao

```
package cn.itcast.erp.dao;
import org.springframework.data.jpa.repository.JpaRepository;
import cn.itcast.erp.entity.Dep;
public interface DepDao extends JpaRepository<Dep, Long>{
}
```

(5) 创建业务逻辑接口



```
package cn.itcast.erp.service;
import java.util.List;
import cn.itcast.erp.entity.Dep;
public interface DepService {

    public List<Dep> findAll();

}
```

(6) 创建业务逻辑实现类

```
package cn.itcast.erp.service.impl;
import java.util.List;
import javax.annotation.Resource;
import org.springframework.stereotype.Service;
import cn.itcast.erp.dao.DepDao;
import cn.itcast.erp.entity.Dep;
import cn.itcast.erp.service.DepService;

@Service
public class DepServiceImpl implements DepService {

    @Resource
    private DepDao depDao;

    @Override
    public List<Dep> findAll() {

        return depDao.findAll();

    }

}
```

(7) 创建 Controller

创建 cn.itcast.erp.controller 包 ,在此包下创建控制器类 DepController ,内容如下：

```
package cn.itcast.erp.controller;
import java.util.List;
import javax.annotation.Resource;

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



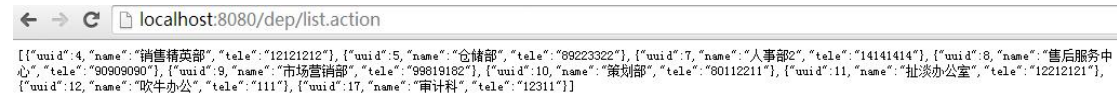
```
import org.springframework.web.bind.annotation.ResponseBody;
import cn.itcast.erp.entity.Dep;
import cn.itcast.erp.service.DepService;

@Controller
@ResponseBody
@RequestMapping("/dep")
public class DepController {

    @Resource
    private DepService depService;

    @RequestMapping("/list")
    public List<Dep> list(){
        return depService.findAll();
    }
}
```

运行引导类 Application，打开浏览器输入 <http://localhost:8080/dep/list>，会看到浏览器上有 json 数据的输出



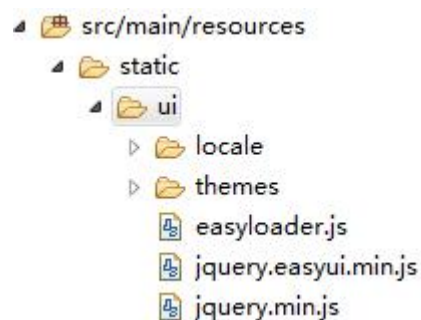
localhost:8080/dep/list.action

```
[{"uuid":4,"name":"销售精英部","tele":"12121212"}, {"uuid":5,"name":"仓储部","tele":"89223322"}, {"uuid":7,"name":"人事部2","tele":"14141414"}, {"uuid":8,"name":"售后服务中心","tele":"90909090"}, {"uuid":9,"name":"市场营销部","tele":"99819182"}, {"uuid":10,"name":"策划部","tele":"80112211"}, {"uuid":11,"name":"扯淡办公室","tele":"12212121"}, {"uuid":12,"name":"吹牛办公","tele":"111"}, {"uuid":17,"name":"审计科","tele":"12311"}]
```

2 前端代码

(1) 添加 EasyUI 框架

在 `src/main/resources` 下创建 `static` 文件夹，将资源文件夹中的 `ui` 文件夹拷贝过来。



```
src/main/resources
├── static
│   └── ui
│       ├── locale
│       ├── themes
│       ├── easyloader.js
│       ├── jquery.easyui.min.js
│       └── jquery.min.js
```

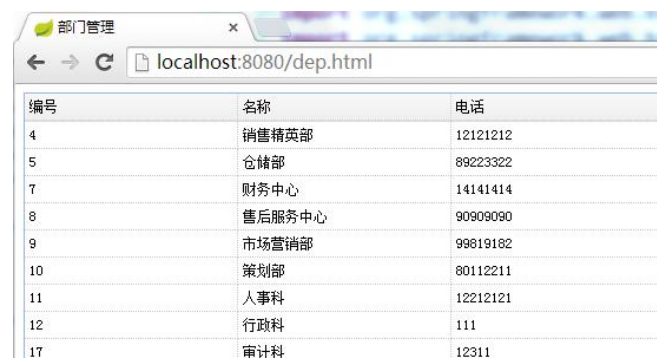
(2) 创建页面



在 static 文件夹下创建页面 dep.html ,内容如下：

```
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>部门管理</title>
<link rel="stylesheet" type="text/css"
href="ui/themes/default/easyui.css">
<link rel="stylesheet" type="text/css" href="ui/themes/icon.css">
<script type="text/javascript" src="ui/jquery.min.js"></script>
<script type="text/javascript" src="ui/jquery.easyui.min.js"></script>
<script type="text/javascript"
src="ui/locale/easyui-lang-zh_CN.js"></script>
<script type="text/javascript">
    $(function(){
        $('#grid').datagrid({
            url:'dep/list',
            columns:[[
                {field:'uuid',title:'编号',width:200},
                {field:'name',title:'名称',width:200},
                {field:'tele',title:'电话',width:200}
            ]]
        });
    });
</script>
</head>
<body>
<table id="grid"></table>
</body>
</html>
```

地址栏输入 http://localhost:8080/dep.html ,效果完成



编号	名称	电话
4	销售精英部	12121212
5	仓储部	89223322
7	财务中心	14141414
8	售后服务中心	90909090
9	市场营销部	99819182
10	策划部	80112211
11	人事科	12212121
12	行政科	111
17	审计科	12311



(二) 整合 Redis

1. 列表数据存入缓存

需求：基于上例代码，将列表数据缓存到 Redis

(1) 在 pom.xml 中引入 Redis 起步依赖

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

(2) 修改引导类 Application, 添加注解

```
@SpringBootApplication
@EnableCaching
public class Application{

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

}
```

此注解作用是开启缓存。

(3) 让实体类实现可序列化接口

```
import javax.persistence.Entity;
import javax.persistence.Id;
@Entity
public class Dep implements Serializable{

    @Id
    private Long uuid;
    private String name;
    private String tele;
```

(4) 在业务逻辑类 DepServiceImpl 的 findAll 方法上添加缓存注解

```
@Cacheable(value = "DepCache", key="'dep.findAll'")
public List<Dep> findAll() {
    System.out.println("findAll()从数据库中提取数据");
    return depDao.findAll();
}
```



测试：启动 redis 服务，在浏览器输入 `http://localhost:8080/dep/list` 可看到 json 字符串，第一次访问在控制台可以看到 `findAll()从数据库中提取数据`，第二次以后访问不会出此提示，表示从缓存数据中提取。

2. 删除数据后更新缓存

需求：当删除数据后清除缓存数据，再次访问列表时从数据库中提取最新数据。

(1) 在业务逻辑层接口 DepService 中添加方法

```
public void delete(Long id);
```

(2) 在业务逻辑类 DepServiceImpl 中添加方法

```
@CacheEvict(value="DepCache", key="'dep.findAll'")
public void delete(Long id){
    System.out.println("删除数据 ID"+id);
    depDao.delete(id);
}
```

@CacheEvict 注解用于清除缓存

3. 指定 Redis 主机 IP

很多情况下 Redis 与 web 应用并非部署在一台机器上，这就需要远程调用 Redis。

我们在 application.properties 中，添加以下配置，指定 Redis 所在的主机

```
spring.redis.host=192.168.80.10
```

(三) 整合 ActiveMQ

1. 使用内嵌服务

(1) 在 pom.xml 中引入 ActiveMQ 起步依赖

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



(2) 在 Application 类中新增方法

```
@Bean
public Queue queue() {
    return new ActiveMQQueue("itcast.queue");
}
```

(3) 创建消息生产者

```
package cn.itcast.demo.controller;
import javax.annotation.Resource;
import javax.jms.Queue;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
@Controller
@ResponseBody
public class QueueController {

    @Resource
    private JmsMessagingTemplate jmsMessagingTemplate;

    @Resource
    private Queue queue;

    /**
     * 消息生产者
     */
    @RequestMapping("/send")
    public void send() {
        this.jmsMessagingTemplate.convertAndSend(this.queue, "我是一个好消息:");
    }
}
```

(4) 创建消息消费者

```
package cn.itcast.demo;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
```



```
/**
 * 消息消费者.
 * @author Angel --守护天使
 * @version v.0.1
 * @date 2016 年 8 月 23 日
 */
@Component
public class Consumer {

    @JmsListener(destination = "itcast.queue")
    public void readItcastQueue(String text) {
        System.out.println(text);
    }
}
```

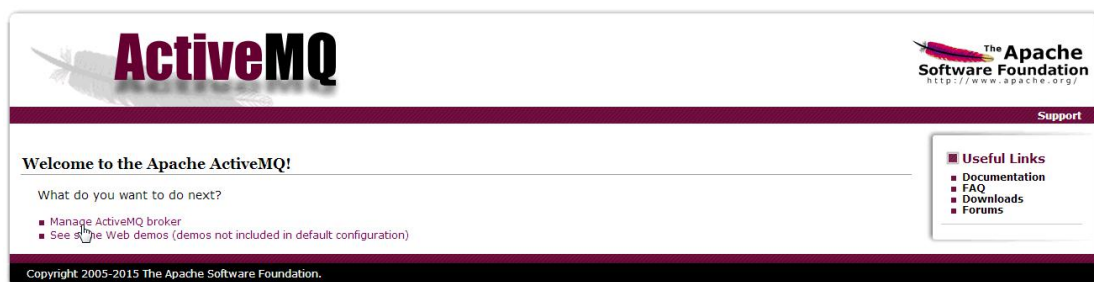
测试：启动服务后，在浏览器执行 <http://localhost:8080/send> 即可看到控制台输出消息提示。Spring Boot 内置了 ActiveMQ 的服务，所以我们不用单独启动也可以执行应用程序。

2. 使用外部服务

(1) 在虚拟机中部署 activeMQ 的服务

我的虚拟机的 IP 是 192.168.80.10，启动后，在本地输入管理界面的地址：

<http://192.168.80.10:8161>



(2) 在 src/main/resources 下创建 application.properties

```
spring.activemq.broker-url=tcp://192.168.80.10:61616
```

指定 ActiveMQ 的地址



(四) 整合 JUnit

(1) 在 pom.xml 中引入依赖

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

(2) 在 src/test/java 下编写单元测试类，测试刚才我们做的消息推送

```
package cn.itcast.erp.test;
import javax.annotation.Resource;
import javax.jms.Queue;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.jms.core.JmsMessagingTemplate;
import org.springframework.test.context.junit4.SpringRunner;
import cn.itcast.erp.Application;
@RunWith(SpringRunner.class)
@SpringBootTest(classes=Application.class)
public class ErpTest {

    @Resource
    private Queue queue;

    @Resource
    private JmsMessagingTemplate jmsMessagingTemplate;

    @Test
    public void testQueue() {
        System.out.println("Test -activeMQ");
        jmsMessagingTemplate.convertAndSend(this.queue, "我是一个好消息: ")
    }
}
```

@SpringBootTest 注解的 class 属性要指定引导类的 class

SpringRunner 与 SpringJUnit4ClassRunner 相同，只是看起来短一些而已。



四、Spring Boot 进阶

(一) servlet 配置

我们在开发中有时会用到自己写的 servlet, web2.5 规范的做法是在 web.xml 中进行配置, 而 web3.0 不推荐采用 web.xml 进行配置, 那如何配置我们自己编写的 servlet 呢?

准备工作: 创建 cn.itcast.demo.servlet 包, 包下建立 InfoServlet 类

```
package cn.itcast.demo.servlet;
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
public class InfoServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest request ,HttpServletResponse
response) throws IOException {
        response.getWriter().print("info...");
    }
}
```

1. 注解配置

(1) 在 Application 类中添加注解, 开启 servlet 注解扫描

```
@SpringBootApplication
@WebServletComponentScan
public class Application {

    public static void main(String[] args) {

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

(2) 在 InfoServlet 中添加注解

```
@WebServlet(urlPatterns="/servlet/info")
public class InfoServlet extends HttpServlet {
```



2. 代码配置

在 Application 类添加方法

```
@Bean
public ServletRegistrationBean infoServletBean(){
    return new ServletRegistrationBean(new InfoServlet(),
        "/servlet/info");
}
```

打开浏览器，地址栏输入 <http://localhost:8080/servlet/info> 即可看到输出结果

(二) 过滤器配置

过滤器的配置也有两种方式：注解方式和代码方式

准备工作：创建 cn.itcast.demo.filter 包，包下创建 MyFilter 类

```
package cn.itcast.demo.filter;
import java.io.IOException;
import javax.servlet.Filter;
import javax.servlet.FilterChain;
import javax.servlet.FilterConfig;
import javax.servlet.ServletException;
import javax.servlet.ServletRequest;
import javax.servlet.ServletResponse;
public class MyFilter implements Filter {
    @Override
    public void destroy() {
    }
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
        FilterChain filterChain)
        throws IOException, ServletException {
        System.out.println("do myFilter!!!");
        filterChain.doFilter(request, response);
    }
    @Override
    public void init(FilterConfig arg0) throws ServletException {
    }
}
```



1. 注解方式

(1) 在过滤器类上添加注解

```
@WebFilter(urlPatterns="/servlet/*")  
public class MyFilter implements Filter {
```

(2) 在引导类 Application 上添加注解

```
@ServletComponentScan
```

2. 代码方式

在 Application 类中添加方法

```
@Bean  
public FilterRegistrationBean myFilterBean(){  
    FilterRegistrationBean bean=new FilterRegistrationBean();  
    bean.setFilter(new MyFilter());  
    bean.addUrlPatterns("/servlet/*");  
    return bean;  
}
```

打开浏览器，地址栏输入 <http://localhost:8080/servlet/info> 即可在控制台看到输出结果

(三) 配置文件读取

1. 读取核心配置文件

(1) 在 spring-boot_Demo 工程的 src/main/resources 下构建核心配置文件 application.properties，内容如下

```
name=\u4F20\u667A\u64AD\u5BA2  
url=http://www.itcast.cn
```

(2) 在 HelloWorldController 中添加

```
@Resource  
private Environment env;
```

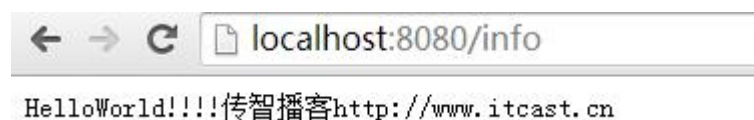
(3) 读取属性用 env 的 getProperty 方法，参数为配置文件中的 key



例如：

```
@RequestMapping("/info")
public String info(){
    return
    "HelloWorld!!!!"+env.getProperty("name")+env.getProperty("url");
}
```

测试一下吧



2.读取自定义配置文件

(1)在 spring-boot_Demo 工程的 src/main/resources 下构建自定义配置文件 mail.properties, 内容如下

```
mail.host=smtp.sina.com
mail.port=25
mail.username=itcast
mail.password=heima
```

(2) 在 cn.itcast.demo 创建 MailProperties 类

```
package cn.itcast.demo;
import
org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
@Configuration
@ConfigurationProperties(locations="classpath:mail.properties",prefix="
mail")
public class MailProperties {

    private String host;
    private Integer port;
    private String username;
    private String password;

    //省略 getter 和 setter

}
```

(3) 在 HelloWorldController 中添加如下代码



```
@Resource
private MailProperties mailProperties;

@RequestMapping("/mailInfo")
public String mailInfo(){
    return mailProperties.getHost()+"<br>"
        +mailProperties.getPort()+"<br>"
        +mailProperties.getUsername()+"<br>"
        +mailProperties.getPassword();
}
```

在浏览器中输入 `http://localhost:8080/mailInfo` 即可看到以下运行结果

```
smtp.sina.com
25
itcast
heima
```

五、Spring Boot 部署

(一) 构建 jar 文件

(1) 我们在 erp 工程的 pom.xml 中，添加 maven 插件

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

(2) 我们目前的工程采用的是 jar 的打包方式，所以我们在执行 package 命令后，会产生一个 jar 包。

我们进入到这个目录用压缩软件打开此 jar 包，看以下它的内部结构



其中我们发现了一个叫 lib 的文件夹，打开 lib 文件夹，我们惊奇地发现此文件夹下全是 erp 工程依赖的 jar 包，甚至还有 tomcat。这种包含有 jar 包的 jar 包，我们称之为 fatJAR(胖 jar 包)

(3) 由于 fatJAR 本身就包括 tomcat，我们就不需要另外部署了，直接在命令行就可以把我们的应用启动起来，在命令行，进入到 jar 包所在的目录，我们可以通过以下命令来执行此 jar 包。

```
java -jar erp-0.0.1-SNAPSHOT.jar
```

在控制台会出现启动信息，在浏览器访问程序。

(二) 构建 war 文件

spring-boot 默认提供内嵌的 tomcat，所以打包直接生成 jar 包，用 java -jar 命令就可以启动。但是，有时候我们更希望一个 tomcat 来管理多个项目，这种情况下就需要项目是 war 格式的包而不是 jar 格式的包。

我们按照以下步骤完成对 erp 工程的改造

(1) 修改 pom.xml

将打包方式修改为 war

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

添加依赖



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

spring-boot-starter-tomcat 是原来被传递过来的依赖，默认会打到包里，所以我们再次引入此依赖，并指定依赖范围为 provided，这样 tomcat 相关的 jar 就不会打包到 war 里了。

(2) 添加 ServletInitializer

```
package cn.itcast.erp;

import org.springframework.boot.builder.SpringApplicationBuilder;
import org.springframework.boot.context.web.SpringBootServletInitializer;

public class ServletInitializer extends SpringBootServletInitializer {

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

由于我们采用 web3.0 规范，是没有 web.xml 的，而此类的作用与 web.xml 相同。

(3) 运行 package 打包命令生成 war 包

生成后将 war 包放入 tomcat，启动 tomcat，测试完成的功能是否可以使用。



六、Spring Boot 原理

（一）起步依赖原理解析

打开 spring-boot-starter-parent 下的 pom.xml ,会发现此文件中已经定了很多 dependencyManagement 和 dependency , dependencyManagement 用来锁定版本 , dependency 中有常用的 jar 包 , spring-boot-starter-parent 的父 pom 是 spring-boot-dependencies。我们打开 spring-boot-dependencies 下的 pom.xml ,会发现此配置中定义了更多的配置信息 ,有常用的软件版本定义和依赖配置。打开 spring-boot-starter-web 下的 pom.xml ,会发现其中配置了 web 开发所需的依赖。

其实起步依赖的原理很简单 , 就是 Maven 的依赖传递的应用。

（二）自动配置原理解析

我们首先看一下启动类的 @SpringBootApplication 注解

```
@SpringBootApplication
public class Application {

    public static void main(String[] args) {

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

这个注解实际上是一个组合注解

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = @Filter(type =
public @interface SpringBootApplication {
```

其中有个 @EnableAutoConfiguration 注解



```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration {
```

其中有个@Import(EnableAutoConfigurationImportSelector.class) 此句话的作用是导入自动配置类，导入哪些？我们接着看

打开 EnableAutoConfigurationImportSelector 这个类：

```
@Override
public String[] selectImports(AnnotationMetadata metadata) {
    if (!isEnabled(metadata)) {
        return NO_IMPORTS;
    }
    try {
        AnnotationAttributes attributes = getAttributes(metadata);
        List<String> configurations = getCandidateConfigurations(metadata,
            attributes);
        configurations = removeDuplicates(configurations);
        Set<String> exclusions = getExclusions(metadata, attributes);
        configurations.removeAll(exclusions);
        configurations = sort(configurations);
        recordWithConditionEvaluationReport(configurations, exclusions);
        return configurations.toArray(new String[configurations.size()]);
    }
    catch (IOException ex) {
        throw new IllegalStateException(ex);
    }
}
```

```
protected List<String> getCandidateConfigurations(AnnotationMetadata metadata,
    AnnotationAttributes attributes) {
    List<String> configurations = SpringFactoriesLoader.loadFactoryNames(
        getSpringFactoriesLoaderFactoryClass(), getBeanClassLoader());
    Assert.notEmpty(configurations,
        "No auto configuration classes found in META-INF/spring.factories. If you "
        + "are using a custom packaging, make sure that file is correct.");
    return configurations;
}
```

SpringFactoriesLoader.loadFactoryNames 方法的作用就是从 META-INF/spring.factories 文件中读取指定类对应的类名称列表，spring.factories 文件中有关自动配置的配置信息如下：

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
```



```
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfiguration,\norg.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\n
```

..... 由于篇幅有限，省略下面的大段配置信息

这个配置信息中出现大量的以 Configuration 为结尾的类名称 这些类就是存有自动配置信息的类，而 SpringApplication 在获取这些类名后再加载。

任何一个自动配置信息类，一般都有如下的条件注解：

- (1) @ConditionalOnBean：当容器里面有指定的 Bean 条件下
- (2) @ConditionalOnClass：当类路径下有指定的类的条件下
- (3) @ConditionalOnProperty:指定的属性是否有执行的值
- (4) @ConditionalOnResource：类路径是否有指定的值
- (5) @ConditionalOnWebApplication：当前项目是 Web 项目的条件下
- (6) @ConditionalOnMissingBean：当容器里没有指定 Bean 的情况下

我们以 ActiveMQ 的自动配置信息类为例来讲解一下自动配置的含义 源码如下：

```
@Configuration\n@AutoConfigureBefore(JmsAutoConfiguration.class)\n@AutoConfigureAfter({ JndiConnectionFactoryAutoConfiguration.class })\n@ConditionalOnClass({ ConnectionFactory.class,\n    ActiveMQConnectionFactory.class })\n@ConditionalOnMissingBean(ConnectionFactory.class)\n@EnableConfigurationProperties(ActiveMQProperties.class)\n@Import({ ActiveMQXAConnectionFactoryConfiguration.class,\n    ActiveMQConnectionFactoryConfiguration.class })\npublic class ActiveMQAutoConfiguration {\n\n}\n
```

解释：

@Configuration 标注此类为配置类

@ConditionalOnClass({ConnectionFactory.class, ActiveMQConnectionFactory.class })表示此类配置在 ConnectionFactory 和 ActiveMQConnectionFactory 都存在的



情况下生效

@ConditionalOnMissingBean(ConnectionFactory.class)

其功能为如果存在指定 name 的 bean，则该注解标注的 bean 不创建

@EnableConfigurationProperties(ActiveMQProperties.class) 表示读取属性类 ActiveMQProperties

ActiveMQProperties 类部分代码如下：

```
@ConfigurationProperties(prefix = "spring.activemq")
public class ActiveMQProperties{
    private String brokerUrl;
    private boolean inMemory = true;
    private String user;
    private String password;
    //代码略 .....
}
```

prefix 表示属性前缀

七、综合案例-会员卡管理系统

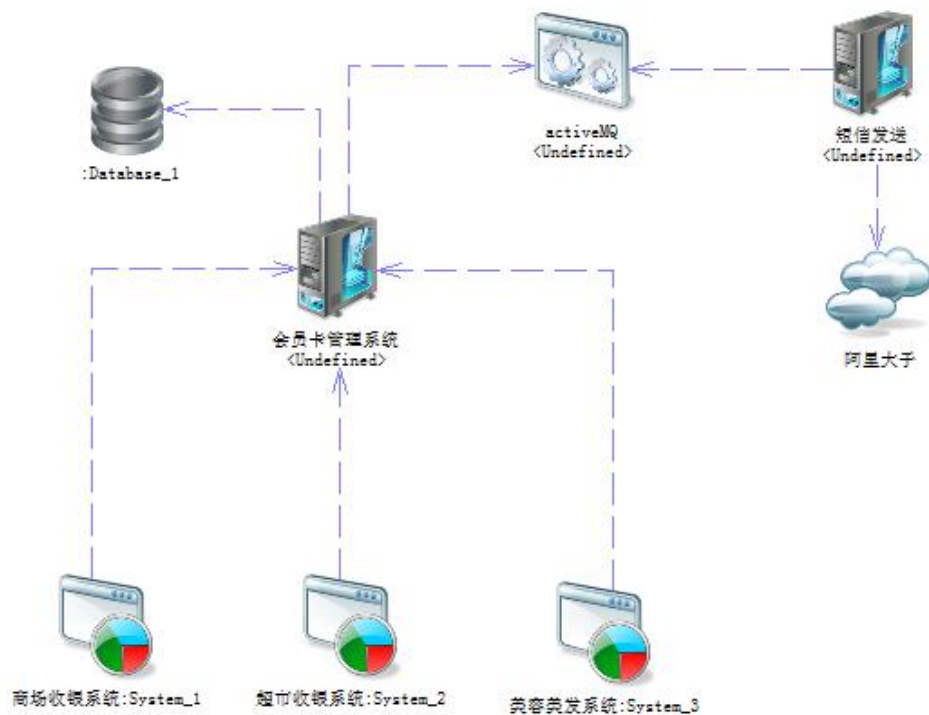
(一) 需求分析

某大型连锁企业旗下包括商场、超市、美容、美发、健身会馆等多元化企业，现提出将旗下所有企业的会员积分纳入统一的平台进行管理，要求开发一套云会员管理平台。

当商场、超市、美容店、美发店等基层商铺产生会员积分变动时，通过调用《会员管理平台》实现积分的变更和积分变更日志的记录。

当会员积分发生变更时，将消息推送到 activeMQ 的队列中，另有一个程序负责读取队列的消息，并调用“阿里大于”提供的短信 SDK 发送短信。

架构图如下：



要求实现的功能：

- (1) 会员卡的查询
- (2) 新增会员
- (3) 增加会员积分（微服务）
- (4) 将积分变动消息推送给 ActiveMQ
- (5) 从 ActiveMQ 中提取消息调用阿里大于发送短信

准备工作：

在 MySQL 中创建数据库 memberdb

建表 member ,用于存放会员信息

Field Name	Datatype	Len	Default	PK?	Not Null?	Unsigned	Auto Incr?	Zero fill	Comment
sn	varchar	10		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	会员卡号
name	varchar	20		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	会员姓名
sex	varchar	10		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	性别
phone	varchar	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	电话
point	int	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	积分余额



建表 point_log ，用于存放积分变更的信息

Field Name	Datatype	Len	Default	PK?	Not Null?	Unsigned	Auto Incr?	Zero fill	Comment
id	bigint	20		<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	ID
exetime	datetime			<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	执行时间
sn	varchar	10		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	会员卡号
point	int	11		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	变动积分数
remark	varchar	200		<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	备注

(二) 搭建工程框架

实现步骤：

(1) 创建 jar 工程 memberpro ，在 pom.xml 中引入依赖与插件

```
<parent>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-parent</artifactId>
<version>1.4.0.RELEASE</version>
<relativePath/>
</parent>
<properties>
<java.version>1.7</java.version>
</properties>
<dependencies>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-devtools</artifactId>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
<groupId>mysql</groupId>
<artifactId>mysql-connector-java</artifactId>
<version>5.1.6</version>
</dependency>
<dependency>
<groupId>org.springframework.boot</groupId>
```




```
<artifactId>spring-boot-starter-redis</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-activemq</artifactId>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.alibaba</groupId>
  <artifactId>fastjson</artifactId>
  <version>1.1.37</version>
</dependency>
</dependencies>

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

(2) 在 src/main/resources 下创建属性文件 application.properties

```
#DB Configuration:
spring.datasource.driverClassName = com.mysql.jdbc.Driver
spring.datasource.url =
jdbc:mysql://localhost:3306/memberdb?characterEncoding=utf-8
spring.datasource.username = root
spring.datasource.password = 123456
#JPA Configuration:
spring.jpa.database=MySQL
spring.jpa.show-sql=true
spring.jpa.generate-ddl=true
spring.jpa.hibernate.ddl-auto=update
spring.jpa.hibernate.naming_strategy=org.hibernate.cfg.ImprovedNamingStrategy
```



(3) 编写引导类

```
package cn.itcast.member;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

(三) 会员卡查询

1. 编写后端代码

(1) 编写实体类 Member

```
package cn.itcast.member.entity;
import javax.persistence.Entity;
import javax.persistence.Id;
/**
 * 会员实体类
 * @author Administrator
 *
 */
@Entity
public class Member {

    @Id
    private String sn;//会员卡号
    private String name;//会员姓名
    private String sex;//会员性别
    private String phone;//会员电话
    private Integer point;//会员积分

    //getter and setter
}
```

(2) 编写数据访问层接口 MemberDao

```
package cn.itcast.member.dao;
import java.util.List;
```




```
import org.springframework.data.jpa.repository.JpaRepository;
import cn.itcast.member.entity.Member;
public interface MemberDao extends JpaRepository<Member, String> {

    /**
     * 根据卡号查询会员信息
     * @param sn
     * @return
     */
    public List<Member> findBySn(String sn);
}
```

(3) 编写业务逻辑层接口 MemberService

```
package cn.itcast.member.service;
import java.util.List;
import cn.itcast.member.entity.Member;
public interface MemberService {

    /**
     * 根据会员卡号查询会员信息
     * @param sn
     * @return
     */
    public List<Member> findBySn(String sn);
}
```

(4) 编写业务逻辑类 MemberServiceImpl

```
package cn.itcast.member.service.impl;
import java.util.List;
import javax.annotation.Resource;
import org.springframework.stereotype.Service;
import cn.itcast.member.dao.MemberDao;
import cn.itcast.member.entity.Member;
import cn.itcast.member.service.MemberService;
@Service
public class MemberServiceImpl implements MemberService{

    @Resource
    private MemberDao memberDao;
```



```
@Override
public List<Member> findBySn(String sn) {
    return memberDao.findBySn(sn);
}
}
```

(5) 编写控制层代码 MemberController

```
package cn.itcast.member.controller;
import java.util.List;
import javax.annotation.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import cn.itcast.member.entity.Member;
import cn.itcast.member.service.MemberService;
@Controller
@ResponseBody
@RequestMapping("/member")
public class MemberController {

    @Resource
    private MemberService memberService;

    @RequestMapping("/sn/{sn}")
    public List<Member> findBySn(@PathVariable String sn){
        return memberService.findBySn(sn);
    }
}
```

浏览器测试：



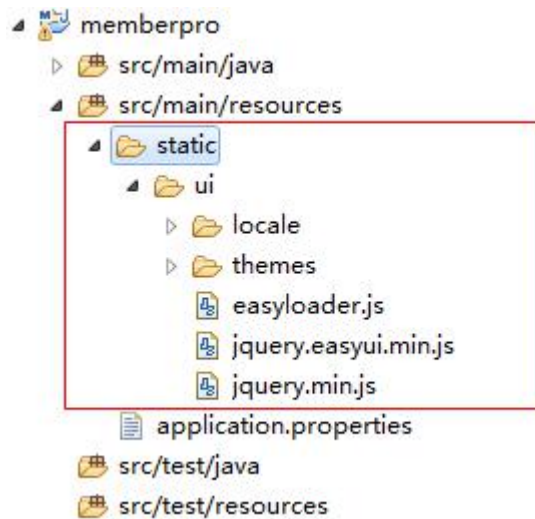
localhost:8080/member/sn/000001

[{"sn": "000001", "name": "孙悟空", "sex": "男", "phone": "12345", "point": 100}]

2. 编写前端代码

(1) 添加 easyUI 框架

在 src/main/resources 创建 static 文件夹，将 ui 文件夹拷贝到 static 文件夹中



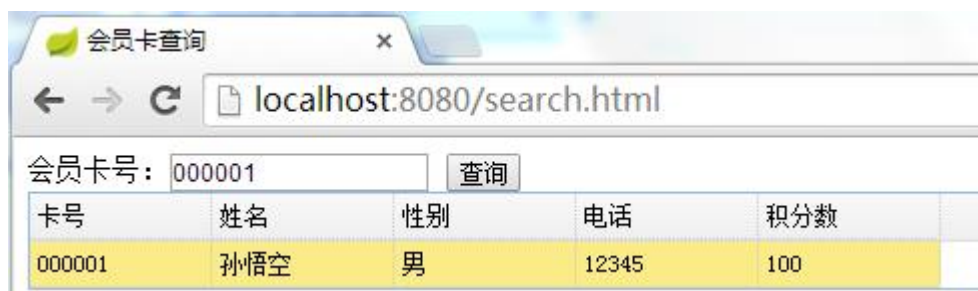
(2) 编写会员卡查询页面 search.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>会员卡查询</title>
<link rel="stylesheet" type="text/css"
href="ui/themes/default/easyui.css">
<link rel="stylesheet" type="text/css" href="ui/themes/icon.css">
<script type="text/javascript" src="ui/jquery.min.js"></script>
<script type="text/javascript" src="ui/jquery.easyui.min.js"></script>
<script type="text/javascript"
src="ui/locale/easyui-lang-zh_CN.js"></script>
<script type="text/javascript">
$(function(){
    //初始化表格列信息
    $('#grid').datagrid({
        columns:[[
            {field:'sn',title:'卡号',width:100} ,
            {field:'name',title:'姓名',width:100} ,
            {field:'sex',title:'性别',width:100} ,
            {field:'phone',title:'电话',width:100} ,
            {field:'point',title:'积分',width:100}
        ]]
    });
});
```



```
//查询
$('#btnSearch').bind('click',function(){
    $('#grid').datagrid({
        url:'member/sn/'+$('#sn').val(),
    });
});
})
</script>
</head>
<body>
<form>
会员卡号: <input id="sn">
<button id="btnSearch" type="button">查询</button>
</form>
<table id="grid"></table>
</body>
</html>
```

浏览器测试：



（四）新增会员

1. 编写后端代码

（1）在 MemberService 中新增方法定义

```
/**
 * 增加会员
 * @param member
 */
public void save(Member member);
```

（2）在 MemberServiceImpl 中实现该方法

```
@Override
public void save(Member member) {
```



```
        member.setPoint(0); //初始化积分分数
        memberDao.save(member);
    }
```

新增会员后，默认积分数为 0

(3) 在 entity 包新增实体类，用于控制层返回信息

```
package cn.itcast.member.entity;

/**
 * 返回的消息实体
 * @author Administrator
 *
 */
public class Result {

    public Result(boolean success, String message) {
        super();
        this.success = success;
        this.message = message;
    }

    private boolean success; //是否成功
    private String message; //消息

    //getter and setter.....
}
```

(4) 在 MemberController 新增方法

```
/**
 * 新增会员
 * @param member
 * @return
 */
@RequestMapping("/save")
public Result save(Member member){
    try {
        memberService.save(member);
        return new Result(true, "会员保存成功");
    } catch (Exception e) {
        e.printStackTrace();
        return new Result(false, "会员保存失败");
    }
}
```



2. 编写前端代码

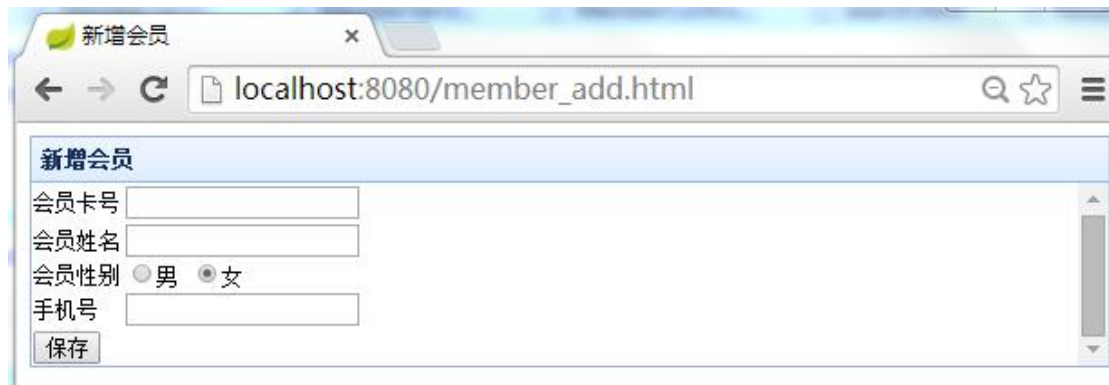
在 static 下创建页面 member_add.html

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>新增会员</title>
<link rel="stylesheet" type="text/css"
href="ui/themes/default/easyui.css">
<link rel="stylesheet" type="text/css" href="ui/themes/icon.css">
<script type="text/javascript" src="ui/jquery.min.js"></script>
<script type="text/javascript" src="ui/jquery.easyui.min.js"></script>
<script type="text/javascript"
src="ui/locale/easyui-lang-zh_CN.js"></script>
<script type="text/javascript">
    $(function(){
        //保存
        $('#btnSave').bind('click',function(){
            //提交表单
            $('#editForm').form('submit',{
                url:'member/save',
                success:function(value){
                    var result=JSON.parse(value);//转换为 json 对象
                    if(result.success){
                        $('#editForm').form('clear');//清空表单
                    }
                    $.messager.alert('提示',result.message);
                }
            });
        });
    });
</script>
</head>
<body>
<div class="easyui-panel" title="新增会员">
<form id="editForm" method="post" >
<table>
    <tr>
        <td>会员卡号</td>
        <td><input name="sn"></td>
```



```
</tr>
<tr>
    <td>会员姓名</td>
    <td><input name="name"></td>
</tr>
<tr>
    <td>会员性别</td>
    <td>
        <input name="sex" type="radio" value="男">男
        <input name="sex" type="radio" value="女">女
    </td>
</tr>
<tr>
    <td>手机号</td>
    <td><input name="phone"></td>
</tr>
</table>
<button type="button" id="btnSave">保存</button>
</form>
</div>
</body>
</html>
```

浏览器测试：



(五) 增加会员积分 (微服务)

(1) 创建实体类 PointLog

```
package cn.itcast.member.entity;
import java.util.Date;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
```



```
import javax.persistence.GenerationType;
import javax.persistence.Id;
/**
 * 积分日志
 * @author Administrator
 *
 */
@Entity
public class PointLog {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Long id;//编号
    private Date exetime;//执行时间
    private String sn;//会员卡号
    private Integer point;//会员积分变动数量
    private String remark;//备注

    //getter and setter
}
```

注意：由于我们数据库表的名称为 point_log ，所以我们的 log 的 L 应该大写

(2) 创建数据访问接口 PointLogDao

```
package cn.itcast.member.dao;
import org.springframework.data.jpa.repository.JpaRepository;
import cn.itcast.member.entity.PointLog;
public interface PointLogDao extends JpaRepository<PointLog, Long> {

}
```

(3) 创建业务逻辑接口 PointLogService

```
package cn.itcast.member.service;
import cn.itcast.member.entity.PointLog;
/**
 * 积分日志业务逻辑接口
 * @author Administrator
 *
 */
public interface PointLogService {

    /**
```




```
    * 增加积分
    * @param point
    */
    public void add(PointLog pointLog);
}
```

(4) 创建业务逻辑实现类 PointLogServiceImpl

```
package cn.itcast.member.service.impl;
import java.util.Date;
import javax.annotation.Resource;
import org.springframework.stereotype.Service;
import cn.itcast.member.dao.MemberDao;
import cn.itcast.member.dao.PointLogDao;
import cn.itcast.member.entity.Member;
import cn.itcast.member.entity.PointLog;
import cn.itcast.member.service.PointLogService;
/**
 * 积分日志业务逻辑类
 */
@Service
public class PointLogServiceImpl implements PointLogService {
    @Resource
    private PointLogDao pointDao;
    @Resource
    private MemberDao memberDao;

    @Override
    public void add(PointLog pointLog) {
        //修改 member 表中的积分
        Member member = memberDao.findOne(pointLog.getSn());
        if(member==null){
            throw new RuntimeException("没有查询结果");
        }
        member.setPoint( member.getPoint()+ pointLog.getPoint() );//加上变动的积分
        if(member.getPoint()<0){
            throw new RuntimeException("积分余额不足");
        }
        pointLog.setExetime(new Date()); //设置执行时间为当前时间
        pointDao.save(pointLog);
    }
}
```



在这段业务逻辑中，首先查询会员卡，如果能够查询则变更积分，并在积分日志表中增加记录

(5) 创建控制器类 PointLogController

```
package cn.itcast.member.controller;
import javax.annotation.Resource;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import cn.itcast.member.entity.PointLog;
import cn.itcast.member.entity.Result;
import cn.itcast.member.service.PointLogService;
@Controller
@ResponseBody
@RequestMapping("/pointlog")
public class PointLogController {

    @Resource
    private PointLogService pointService;

    @RequestMapping("/add")
    public Result add(PointLog pointLog){
        try {
            pointService.add(pointLog);
            return new Result(true, "增加积分成功");
        } catch (RuntimeException e) {
            e.printStackTrace();
            return new Result(false, e.getMessage());
        } catch (Exception e) {

            e.printStackTrace();
            return new Result(false, "增加积分失败");
        }
    }
}
```

浏览器测试：

<http://localhost:8080/pointlog/add?sn=000003&point=100&remark=ok>



(六) 将积分变动信息推送给 ActiveMQ

准备工作，在虚拟机（192.168.80.10）中驱动 activeMQ

(1) 在 Application 类中增加一个 bean

```
@Bean
public Queue sms_queue() {
    return new ActiveMQQueue("itcast.sms");
}
```

(2) 在 application.properties 增加配置

```
spring.activemq.broker-url=tcp://192.168.80.10:61616
sms.templatCode.point=SMS\_49880037
```

(3) 创建用于传递给 activeMQ 的对象

```
package cn.itcast.member.entity;
import java.io.Serializable;
/**
 * 短信
 * @author Administrator
 *
 */
public class Sms implements Serializable{

    private String phone;//手机号
    private String templatCode;//短信模板编号
    private String paramString;//参数 json 字符串

    //getter and setter....
}
```

(4) 在 PointLogServiceImpl 增加属性

```
@Resource
private JmsMessagingTemplate jmsMessagingTemplate;

@Resource
private Queue sms_queue;

@Resource
private Environment env;
```



(5) 在 PointLogServiceImpl 的 add 方法中增加代码

```
//将短信对象推送给 ActiveMQ
try {
    Sms sms=new Sms();
    sms.setPhone(member.getPhone()); //手机号
    sms.setTemplatCode(environment.getProperty("sms.templatCode.point"))
    ; //短信模板

    Map<String, String> map=new HashMap<>();
    map.put("name", member.getName()); //姓名
    map.put("sn", member.getSn()); //卡号
    map.put("point", String.valueOf(pointLog.getPoint())); //变动的积分
    map.put("allpoint", String.valueOf(member.getPoint())); //积分余额

    sms.setParamString(JSON.toJSONString(map));
    jmsMessagingTemplate.convertAndSend(sms_queue, sms );
} catch (Exception e) {
    // TODO Auto-generated catch block
    e.printStackTrace();
}
```

浏览器测试增加会员积分后

← → ↻

{"success": true, "message": "增加积分成功"}

进入 activeMQ (http://192.168.80.10:8161/) 查看到推送的消息。

itcast.sms	1	0	1	0	Browse Active Consumers Active Producers	Send To Purge Delete
------------	---	---	---	---	---	-------------------------

(七) 短信网关

1. 准备工作

(1) 短信平台账号的申请与 SDK 下载

我们使用“阿里大于”提供的短信平台发送短信，关于阿里大于相关的文档

我们在“资源”中提供。



(2) 将 jar 包安装到本地仓库

```
mvn install:install-file -DgroupId=com.taobao.api  
-DartifactId=taobao -Dversion=1.0 -Dpackaging=jar  
-Dfile=d:\taobao-sdk-java-auto.jar
```

2. 代码编写

(1) 创建工程 sms，引入依赖

```
<parent>  
  <groupId>org.springframework.boot</groupId>  
  <artifactId>spring-boot-starter-parent</artifactId>  
  <version>1.4.0.RELEASE</version>  
  <relativePath/>  
</parent>  
<properties>  
  <java.version>1.7</java.version>  
</properties>  
<dependencies>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-activemq</artifactId>  
  </dependency>  
  <dependency>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-web</artifactId>  
  </dependency>  
  <dependency>  
    <groupId>com.taobao.api</groupId>  
    <artifactId>taobao</artifactId>  
    <version>1.0</version>  
  </dependency>  
</dependencies>  
<build>  
  <plugins>  
    <plugin>  
      <groupId>org.springframework.boot</groupId>  
      <artifactId>spring-boot-maven-plugin</artifactId>  
    </plugin>  
  </plugins>  
</build>
```



(2) 编写启动类

```
package cn.itcast.sms;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
@SpringBootApplication
public class Application {

    public static void main(String[] args) {

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

(3) 编写配置文件 application.properties

```
server.port=9000
spring.activemq.broker-url=tcp://192.168.80.10:61616
spring.activemq.packages.trustAll=true

alidayu.sms.appkey=23656001
alidayu.sms.secret=你的秘钥
alidayu.sms.freeSignName=\u9ED1\u9A6C
```

因为我们要同时启动两个项目，所以我们要通过 server.port 来设置 tomcat 的启动端口

spring.activemq.broker-url 设置 activeMQ 的地址

spring.activemq.packages.trustAll=true 设置将所有的对象设置为信任。这样我们才可以取出 activeMQ 中的对象。

alidayu.sms 为前缀的配置是我们自己定义的

alidayu.sms.appkey 为应用程序 KEY

Alidayu.sms.secret 为秘钥

alidayu.sms.freeSignName 为签名

(4) 编写配置类



```
package cn.itcast.sms;

import
org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
/**
 * 短信配置类
 * @author Administrator
 *
 */
@Configuration
@ConfigurationProperties(prefix="alidayu.sms")
public class SmsConfig {

    private String appkey;//应用 KEY
    private String secret;//密钥
    private String freeSignName;//签名

    //getter and setter...
}
```

(5) 编写短信实体类

与 memberpro 工程中的 Sms 类相同，可以直接拷贝过来使用

(6) 编写“消息消费者”类 Consumer

```
package cn.itcast.sms;

import javax.annotation.Resource;
import javax.jms.JMSException;
import org.apache.activemq.command.ActiveMQObjectMessage;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.stereotype.Component;
import com.taobao.api.ApiException;
import com.taobao.api.DefaultTaobaoClient;
import com.taobao.api.TaobaoClient;
import com.taobao.api.request.AlibabaAliqinFcSmsNumSendRequest;
import com.taobao.api.response.AlibabaAliqinFcSmsNumSendResponse;
import cn.itcast.member.entity.Sms;

/**
 * 消息消费者-发送短信
 */
```



```
@Component
public class Consumer {
    @Resource
    private SmsConfig smsConfig;

    @JmsListener(destination="itcast.sms")
    public void sendSms(Sms sms){
        try {
            String url="http://gw.api.taobao.com/router/rest";
            TaobaoClient client = new DefaultTaobaoClient(url,
smsConfig.getAppkey(),smsConfig.getSecret());
            AlibabaAliqinFcSmsNumSendRequest req = new
AlibabaAliqinFcSmsNumSendRequest();
            req.setSmsType( "normal" );
            req.setSmsFreeSignName( smsConfig.getFreeSignName() );//签名
            req.setSmsParamString( sms.getParamString() );//参数字符串
            req.setRecNum( sms.getPhone() );//手机号
            req.setSmsTemplateCode( sms.getTemplatCode());//模板编号
            AlibabaAliqinFcSmsNumSendResponse rsp = client.execute(req);//
发送
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

(八) 部署

(1) 部署会员管理系统

(2) 部署短信网关



附录 A. Spring Boot 应用启动器

spring Boot 应用启动器基本的一共有 44 种，具体如下：

1) spring-boot-starter

这是 Spring Boot 的核心启动器，包含了自动配置、日志和 YAML。

2) spring-boot-starter-actuator

帮助监控和管理应用。

3) spring-boot-starter-amqp

通过 spring-rabbit 来支持 AMQP 协议 (Advanced Message Queuing Protocol) 。

4) spring-boot-starter-aop

支持面向方面的编程即 AOP，包括 spring-aop 和 AspectJ。

5) spring-boot-starter-artemis

通过 Apache Artemis 支持 JMS 的 API (Java Message Service API) 。

6) spring-boot-starter-batch

支持 Spring Batch，包括 HSQLDB 数据库。

7) spring-boot-starter-cache

支持 Spring 的 Cache 抽象。

8) spring-boot-starter-cloud-connectors

支持 Spring Cloud Connectors，简化了在像 Cloud Foundry 或 Heroku 这样的云平台上连接服务。

9) spring-boot-starter-data-elasticsearch

支持 Elasticsearch 搜索和分析引擎，包括 spring-data-elasticsearch。

10) spring-boot-starter-data-gemfire

支持 GemFire 分布式数据存储，包括 spring-data-gemfire。

11) spring-boot-starter-data-jpa

支持 JPA (Java Persistence API)，包括 spring-data-jpa、spring-orm、hibernate。

12) spring-boot-starter-data-MongoDB

支持 MongoDB 数据，包括 spring-data-mongodb。

13) spring-boot-starter-data-rest

通过 spring-data-rest-webmvc，支持通过 REST 暴露 Spring Data 数据仓库。

14) spring-boot-starter-data-solr

支持 Apache Solr 搜索平台，包括 spring-data-solr。



15) spring-boot-starter-freemarker

支持 FreeMarker 模板引擎。

16) spring-boot-starter-groovy-templates

支持 Groovy 模板引擎。

17) spring-boot-starter-hateoas

通过 spring-hateoas 支持基于 HATEOAS 的 RESTful Web 服务。

18) spring-boot-starter-hornetq

通过 HornetQ 支持 JMS。

19) spring-boot-starter-integration

支持通用的 spring-integration 模块。

20) spring-boot-starter-jdbc

支持 JDBC 数据库。

21) spring-boot-starter-jersey

支持 Jersey RESTful Web 服务框架。

22) spring-boot-starter-jta-atomikos

通过 Atomikos 支持 JTA 分布式事务处理。

23) spring-boot-starter-jta-bitronix

通过 Bitronix 支持 JTA 分布式事务处理。

24) spring-boot-starter-mail

支持 javax.mail 模块。

25) spring-boot-starter-mobile

支持 spring-mobile。

26) spring-boot-starter-mustache

支持 Mustache 模板引擎。

27) spring-boot-starter-Redis

支持 Redis 键值存储数据库，包括 spring-redis。

28) spring-boot-starter-security

支持 spring-security。

29) spring-boot-starter-social-facebook

支持 spring-social-facebook

30) spring-boot-starter-social-linkedin

支持 pring-social-linkedin



31) spring-boot-starter-social-twitter

支持 pring-social-twitter

32) spring-boot-starter-test

支持常规的测试依赖，包括 JUnit、Hamcrest、Mockito 以及 spring-test 模块。

33) spring-boot-starter-thymeleaf

支持 Thymeleaf 模板引擎，包括与 Spring 的集成。

34) spring-boot-starter-velocity

支持 Velocity 模板引擎。

35) spring-boot-starter-web

S 支持全栈式 Web 开发，包括 Tomcat 和 spring-webmvc。

36) spring-boot-starter-websocket

支持 WebSocket 开发。

37) spring-boot-starter-ws

支持 Spring Web Services。

Spring Boot 应用启动器面向生产环境的还有 2 种，具体如下：

1) spring-boot-starter-actuator

增加了面向产品上线相关的功能，比如测量和监控。

2) spring-boot-starter-remote-shell

增加了远程 ssh shell 的支持。

最后，Spring Boot 应用启动器还有一些替换技术的启动器，具体如下：

1) spring-boot-starter-jetty

引入了 Jetty HTTP 引擎（用于替换 Tomcat）。

2) spring-boot-starter-log4j

支持 Log4J 日志框架。

3) spring-boot-starter-logging

引入了 Spring Boot 默认的日志框架 Logback。

4) spring-boot-starter-tomcat

引入了 Spring Boot 默认的 HTTP 引擎 Tomcat。

5) spring-boot-starter-undertow

引入了 Undertow HTTP 引擎（用于替换 Tomcat）。



附录 B. Spring Boot 配置文件 application.properties

```
#####COMMON SPRING BOOT PROPERTIES
#####-----CORE PROPERTIES-----
#SPRING CONFIG (ConfigFileApplicationListener)

spring.config.name= # config file name (default to 'application')
spring.config.location= # location of config file

#PROFILES
spring.profiles= # comma list of active profiles

#APPLICATION SETTINGS (SpringApplication)
spring.main.sources=
spring.main.web-environment= # detect by default
spring.main.show-banner=true
spring.main...= # see class for all properties

#LOGGING
logging.path=/var/logs
logging.file=myapp.log
logging.config=

#IDENTITY (ContextIdApplicationContextInitializer)
spring.application.name=
spring.application.index=

#EMBEDDED SERVER CONFIGURATION (ServerProperties)
server.port=8080
server.address= # bind to a specific NIC
server.session-timeout= # session timeout in seconds
server.context-path= # the context path, defaults to '/'
server.servlet-path= # the servlet path, defaults to '/'
server.tomcat.access-log-pattern= # log pattern of the access log
server.tomcat.access-log-enabled=false # is access logging enabled
server.tomcat.protocol-header=x-forwarded-proto # ssl forward headers
server.tomcat.remote-ip-header=x-forwarded-for
server.tomcat.basedir=/tmp # base dir (usually not needed, defaults to tmp)
server.tomcat.background-processor-delay=30; # in seconds
server.tomcat.max-threads = 0 # number of threads in protocol handler
server.tomcat.uri-encoding = UTF-8 # character encoding to use for URL decoding
```



```
#SPRING MVC (HttpMapperProperties)
http.mappers.json-pretty-print=false # pretty print JSON
http.mappers.json-sort-keys=false # sort keys
spring.mvc.locale= # set fixed locale, e.g. enUK
spring.mvc.date-format= # set fixed date format, e.g. dd/MM/yyyy
spring.mvc.message-codes-resolver-format= # PREFIXERRORCODE / POSTFIXERROR_CODE
spring.view.prefix= # MVC view prefix
spring.view.suffix= # ... and suffix
spring.resources.cache-period= # cache timeouts in headers sent to browser
spring.resources.add-mappings=true # if default mappings should be added

#THYMELEAF (ThymeleafAutoConfiguration)
spring.thymeleaf.prefix=classpath:/templates/
spring.thymeleaf.suffix=.html
spring.thymeleaf.mode=HTML5
spring.thymeleaf.encoding=UTF-8
spring.thymeleaf.content-type=text/html # ;charset=<encoding> is added
spring.thymeleaf.cache=true # set to false for hot refresh

#FREEMARKER (FreeMarkerAutoConfiguration)
spring.freemarker.allowRequestOverride=false
spring.freemarker.allowSessionOverride=false
spring.freemarker.cache=true
spring.freemarker.checkTemplateLocation=true
spring.freemarker.contentType=text/html
spring.freemarker.exposeRequestAttributes=false
spring.freemarker.exposeSessionAttributes=false
spring.freemarker.exposeSpringMacroHelpers=false
spring.freemarker.prefix=
spring.freemarker.requestContextAttribute=
spring.freemarker.settings.*=
spring.freemarker.suffix=.ftl
spring.freemarker.templateEncoding=UTF-8
spring.freemarker.templateLoaderPath=classpath:/templates/
spring.freemarker.viewNames= # whitelist of view names that can be resolved

#GROOVY TEMPLATES (GroovyTemplateAutoConfiguration)
spring.groovy.template.allowRequestOverride=false
spring.groovy.template.allowSessionOverride=false
spring.groovy.template.cache=true
spring.groovy.template.configuration.*= # See Groovy's TemplateConfiguration
```



```
spring.groovy.template.contentType=text/html
spring.groovy.template.prefix=classpath:/templates/
spring.groovy.template.suffix=.tpl
spring.groovy.template.templateEncoding=UTF-8
spring.groovy.template.viewNames= # whitelist of view names that can be resolved

#VELOCITY TEMPLATES (VelocityAutoConfiguration)
spring.velocity.allowRequestOverride=false
spring.velocity.allowSessionOverride=false
spring.velocity.cache=true
spring.velocity.checkTemplateLocation=true
spring.velocity.contentType=text/html
spring.velocity.dateToolAttribute=
spring.velocity.exposeRequestAttributes=false
spring.velocity.exposeSessionAttributes=false
spring.velocity.exposeSpringMacroHelpers=false
spring.velocity.numberToolAttribute=
spring.velocity.prefix=
spring.velocity.properties.*=
spring.velocity.requestContextAttribute=
spring.velocity.resourceLoaderPath=classpath:/templates/
spring.velocity.suffix=.vm
spring.velocity.templateEncoding=UTF-8
spring.velocity.viewNames= # whitelist of view names that can be resolved

#INTERNATIONALIZATION (MessageSourceAutoConfiguration)
spring.messages.basename=messages
spring.messages.cacheSeconds=-1
spring.messages.encoding=UTF-8

#SECURITY (SecurityProperties)
security.user.name=user # login username
security.user.password= # login password
security.user.role=USER # role assigned to the user
security.require-ssl=false # advanced settings ...
security.enable-csrf=false
security.basic.enabled=true
security.basic.realm=Spring
security.basic.path= # /**
security.headers.xss=false
security.headers.cache=false
security.headers.frame=false
```



```
security.headers.contentType=false
security.headers.hsts=all # none / domain / all
security.sessions=stateless # always / never / if_required / stateless
security.ignored=false

#DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.name= # name of the data source
spring.datasource.initialize=true # populate using data.sql
spring.datasource.schema= # a schema (DDL) script resource reference
spring.datasource.data= # a data (DML) script resource reference
spring.datasource.platform= # the platform to use in the schema resource (schema-${platform}.sql)
spring.datasource.continueOnError=false # continue even if can't be initialized
spring.datasource.separator=; # statement separator in SQL initialization scripts
spring.datasource.driverClassName= # JDBC Settings...
spring.datasource.url=
spring.datasource.username=
spring.datasource.password=
spring.datasource.max-active=100 # Advanced configuration...
spring.datasource.max-idle=8
spring.datasource.min-idle=8
spring.datasource.initial-size=10
spring.datasource.validation-query=
spring.datasource.test-on-borrow=false
spring.datasource.test-on-return=false
spring.datasource.test-while-idle=
spring.datasource.time-between-eviction-runs-millis=
spring.datasource.min-evictable-idle-time-millis=
spring.datasource.max-wait-millis=

#MONGODB (MongoProperties)
spring.data.mongodb.host= # the db host
spring.data.mongodb.port=27017 # the connection port (defaults to 27107)
spring.data.mongodb.uri=mongodb://localhost/test # connection URL
spring.data.mongo.repositories.enabled=true # if spring data repository support is enabled

#JPA (JpaBaseConfiguration, HibernateJpaAutoConfiguration)
spring.jpa.properties.*= # properties to set on the JPA connection
spring.jpa.openInView=true
spring.jpa.show-sql=true
spring.jpa.database-platform=
spring.jpa.database=
spring.jpa.generate-ddl=false # ignored by Hibernate, might be useful for other vendors
```



```
spring.jpa.hibernate.naming-strategy= # naming classname
spring.jpa.hibernate.ddl-auto= # defaults to create-drop for embedded dbs
spring.data.jpa.repositories.enabled=true # if spring data repository support is enabled

#SOLR (SolrProperties))
spring.data.solr.host=http://127.0.0.1:8983/solr
spring.data.solr.zkHost=
spring.data.solr.repositories.enabled=true # if spring data repository support is enabled

#ELASTICSEARCH (ElasticsearchProperties))
spring.data.elasticsearch.cluster-name= # The cluster name (defaults to elasticsearch)
spring.data.elasticsearch.cluster-nodes= # The address(es) of the server node (comma-separated;
if not specified starts a client node)
spring.data.elasticsearch.local=true # if local mode should be used with client nodes
spring.data.elasticsearch.repositories.enabled=true # if spring data repository support is
enabled

#FLYWAY (FlywayProperties)
flyway.locations=classpath:db/migrations # locations of migrations scripts
flyway.schemas= # schemas to update
flyway.initVersion= 1 # version to start migration
flyway.prefix=V
flyway.suffix=.sql
flyway.enabled=true
flyway.url= # JDBC url if you want Flyway to create its own DataSource
flyway.user= # JDBC username if you want Flyway to create its own DataSource
flyway.password= # JDBC password if you want Flyway to create its own DataSource

#LIQUIBASE (LiquibaseProperties)
liquibase.change-log=classpath:/db/changelog/db.changelog-master.yaml
liquibase.contexts= # runtime contexts to use
liquibase.default-schema= # default database schema to use
liquibase.drop-first=false
liquibase.enabled=true

#JMX
spring.jmx.enabled=true # Expose MBeans from Spring

#ABBIT (RabbitProperties)
spring.rabbitmq.host= # connection host
spring.rabbitmq.port= # connection port
spring.rabbitmq.addresses= # connection addresses (e.g. myhost:9999,otherhost:1111)
```




```
spring.rabbitmq.username= # login user
spring.rabbitmq.password= # login password
spring.rabbitmq.virtualhost=
spring.rabbitmq.dynamic=

#REDIS (RedisProperties)
spring.redis.host=localhost # server host
spring.redis.password= # server password
spring.redis.port=6379 # connection port
spring.redis.pool.max-idle=8 # pool settings ...
spring.redis.pool.min-idle=0
spring.redis.pool.max-active=8
spring.redis.pool.max-wait=-1

#ACTIVEMQ (ActiveMQProperties)
spring.activemq.broker-url=tcp://localhost:61616 # connection URL
spring.activemq.user=
spring.activemq.password=
spring.activemq.in-memory=true # broker kind to create if no broker-url is specified
spring.activemq.pooled=false

#HornetQ (HornetQProperties)
spring.hornetq.mode= # connection mode (native, embedded)
spring.hornetq.host=localhost # hornetQ host (native mode)
spring.hornetq.port=5445 # hornetQ port (native mode)
spring.hornetq.embedded.enabled=true # if the embedded server is enabled (needs
hornetq-jms-server.jar)
spring.hornetq.embedded.serverId= # auto-generated id of the embedded server (integer)
spring.hornetq.embedded.persistent=false # message persistence
spring.hornetq.embedded.data-directory= # location of data content (when persistence is enabled)
spring.hornetq.embedded.queues= # comma separate queues to create on startup
spring.hornetq.embedded.topics= # comma separate topics to create on startup
spring.hornetq.embedded.cluster-password= # customer password (randomly generated by default)

#JMS (JmsProperties)
spring.jms.pub-sub-domain= # false for queue (default), true for topic

#SPRING BATCH (BatchDatabaseInitializer)
spring.batch.job.names=job1,job2
spring.batch.job.enabled=true
spring.batch.initializer.enabled=true
spring.batch.schema= # batch schema to load
```



```
#AOP
spring.aop.auto=
spring.aop.proxy-target-class=

#FILE ENCODING (FileEncodingApplicationListener)
spring.mandatory-file-encoding=false

#SPRING SOCIAL (SocialWebAutoConfiguration)
spring.social.auto-connection-views=true # Set to true for default connection views or false if
you provide your own

#SPRING SOCIAL FACEBOOK (FacebookAutoConfiguration)
spring.social.facebook.app-id= # your application's Facebook App ID
spring.social.facebook.app-secret= # your application's Facebook App Secret

#SPRING SOCIAL LINKEDIN (LinkedInAutoConfiguration)
spring.social.linkedin.app-id= # your application's LinkedIn App ID
spring.social.linkedin.app-secret= # your application's LinkedIn App Secret

#SPRING SOCIAL TWITTER (TwitterAutoConfiguration)
spring.social.twitter.app-id= # your application's Twitter App ID
spring.social.twitter.app-secret= # your application's Twitter App Secret

#SPRING MOBILE SITE PREFERENCE (SitePreferenceAutoConfiguration)
spring.mobile.sitepreference.enabled=true # enabled by default

#SPRING MOBILE DEVICE VIEWS (DeviceDelegatingViewResolverAutoConfiguration)
spring.mobile.devicedelegatingviewresolver.enabled=true # disabled by default
spring.mobile.devicedelegatingviewresolver.normalPrefix=
spring.mobile.devicedelegatingviewresolver.normalSuffix=
spring.mobile.devicedelegatingviewresolver.mobilePrefix=mobile/
spring.mobile.devicedelegatingviewresolver.mobileSuffix=
spring.mobile.devicedelegatingviewresolver.tabletPrefix=tablet/
spring.mobile.devicedelegatingviewresolver.tabletSuffix=

#####=====ACTUATOR PROPERTIES=====

#MANAGEMENT HTTP SERVER (ManagementServerProperties)
management.port= # defaults to 'server.port'
management.address= # bind to a specific NIC
management.contextPath= # default to '/'
```



```
#ENDPOINTS (AbstractEndpoint subclasses)
endpoints.autoconfig.id=autoconfig
endpoints.autoconfig.sensitive=true
endpoints.autoconfig.enabled=true
endpoints.beans.id=beans
endpoints.beans.sensitive=true
endpoints.beans.enabled=true
endpoints.configprops.id=configprops
endpoints.configprops.sensitive=true
endpoints.configprops.enabled=true
endpoints.configprops.keys-to-sanitize=password,secret
endpoints.dump.id=dump
endpoints.dump.sensitive=true
endpoints.dump.enabled=true
endpoints.env.id=env
endpoints.env.sensitive=true
endpoints.env.enabled=true
endpoints.health.id=health
endpoints.health.sensitive=false
endpoints.health.enabled=true
endpoints.info.id=info
endpoints.info.sensitive=false
endpoints.info.enabled=true
endpoints.metrics.id=metrics
endpoints.metrics.sensitive=true
endpoints.metrics.enabled=true
endpoints.shutdown.id=shutdown
endpoints.shutdown.sensitive=true
endpoints.shutdown.enabled=false
endpoints.trace.id=trace
endpoints.trace.sensitive=true
endpoints.trace.enabled=true

#MVC ONLY ENDPOINTS
endpoints.jolokia.path=jolokia
endpoints.jolokia.sensitive=true
endpoints.jolokia.enabled=true # when using Jolokia
endpoints.error.path=/error

#JMX ENDPOINT (EndpointMBeanExportProperties)
endpoints.jmx.enabled=true
```



```
endpoints.jmx.domain= # the JMX domain, defaults to 'org.springframework'
endpoints.jmx.unique-names=false
endpoints.jmx.enabled=true
endpoints.jmx.staticNames=

#JOLOKIA (JolokiaProperties)
jolokia.config.*= # See Jolokia manual

#REMOTE SHELL
shell.auth=simple # jaas, key, simple, spring
shell.command-refresh-interval=-1
shell.command-path-pattern= # classpath:/commands/, classpath:/crash/commands/
shell.config-path-patterns= # classpath:/crash/
shell.disabled-plugins=false # don't expose plugins
shell.ssh.enabled= # ssh settings ...
shell.ssh.keyPath=
shell.ssh.port=
shell.telnet.enabled= # telnet settings ...
shell.telnet.port=
shell.auth.jaas.domain= # authentication settings ...
shell.auth.key.path=
shell.auth.simple.user.name=
shell.auth.simple.user.password=
shell.auth.spring.roles=

#GIT INFO
spring.git.properties= # resource ref to generated git info properties file
```