目錄

Spring Boot 开源电子书

阅读地址

Github 托管地址

- Spring Boot Book 目前正在书写中, 将总结我之前的文章和对 Spring Boot 的理解 ，也欢迎大家一起加入和完善。
- 如果发现不通顺或者有歧义的地方, 可以在评论里指出来, 我们会及时改正的.
- 本文对应的示例代码：spring-boot-examples
- 欢迎大家加入 Spring Boot 交流群，群号：**755543204**，暗号：springboot
- 建议使用GitBook Editor编辑

Spring Boot 学习资源

- 微服务技术架构和大数据治理实战
- 快速学习 Spring Boot 技术栈
- Spring Boot 中文导航
- Spring Boot 经典学习示例
- Spring Boot 问答-知乎
- Spring Cloud 中文导航

如何参与

任何问题都欢迎直接联系我 ityouknow@126.com

Gitbook 提供了非常棒的在线编辑功能, 所以想贡献的同学可以直接联系我申请权限！

许可证

贡献者列表

成员	联系方式	**Github**
ityouknow	ityouknow@126.com	https://github.com/ityouknow

继续学习

如果你想及时了解 Spring Boot 信息，可以访问我的博客：www.ityouknow.com，也可以关注我的公众号：

关于本书

计划分三个阶段来更新此开源书籍：

- 第一个阶段，先同步自己博客中关于 Spring Boot 1.0 的文章
- 第二个阶段，新增博客中 Spring Boot 2.0 的文章
- 第三个阶段，完善自己对 Spring Boot 的理解，以及开发过程中需要的实践经验

目前还在第一个阶段...

什么是spring boot

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。用我的话来理解，就是spring boot其实不是什么新的框架，它默认配置了很多框架的使用方式，就像maven整合了所有的jar包，spring boot整合了所有的框架（不知道这样比喻是否合适）。

使用spring boot有什么好处

其实就是简单、快速、方便！平时如果我们需要搭建一个spring web项目的时候需要怎么做呢？

- 1）配置web.xml，加载spring和spring mvc
- 2）配置数据库连接、配置spring事务
- 3）配置加载配置文件的读取，开启注解
- 4）配置日志文件
- ...
- 配置完成之后部署tomcat 调试
- ...

现在非常流行微服务，如果我这个项目仅仅只是需要发送一个邮件，如果我的项目仅仅是生产一个积分；我都需要这样折腾一遍!

但是如果使用spring boot呢？
很简单，我仅仅只需要非常少的几个配置就可以迅速方便的搭建起来一套web项目或者是构建一个微服务！

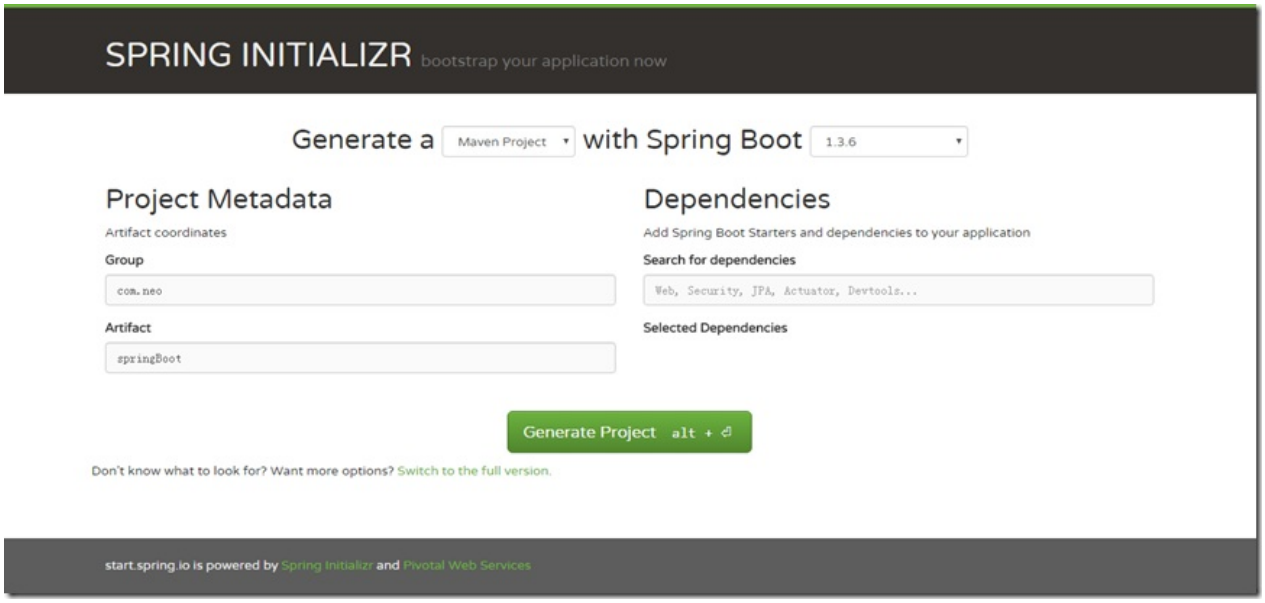
使用sping boot到底有多爽，用下面这幅图来表达

快速入门

说了那么多，手痒痒的很，马上来一发试试!

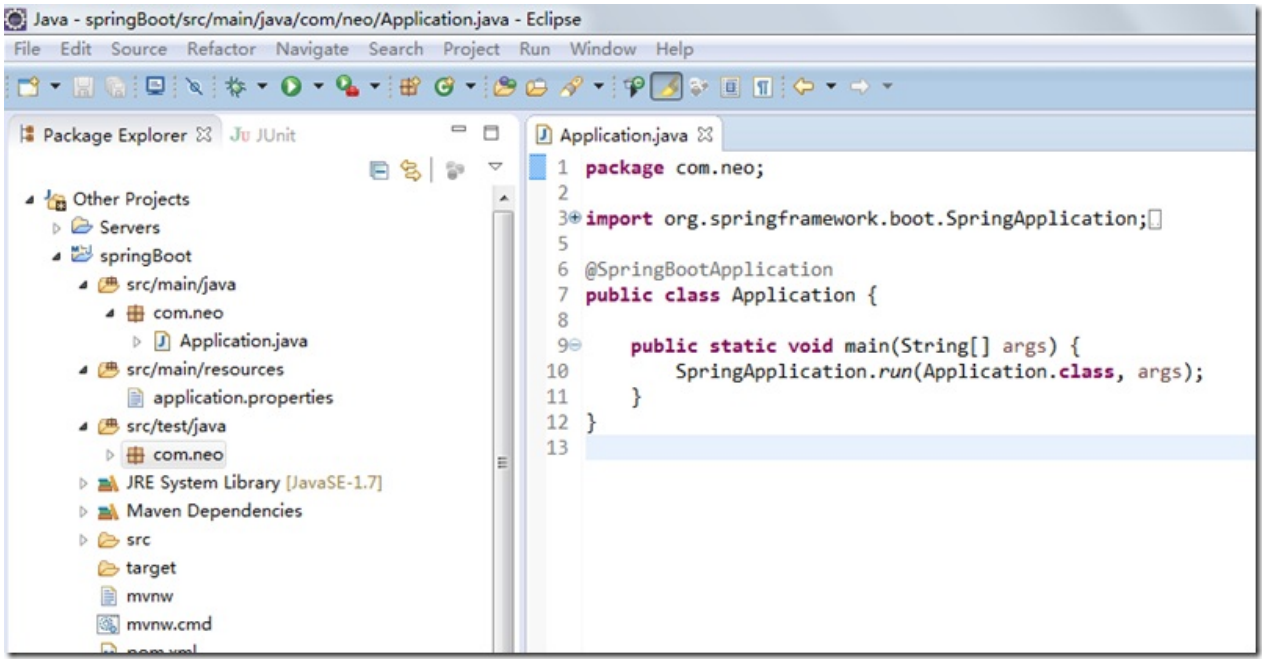
maven构建项目

- 1、访问http://start.spring.io/
- 2、选择构建工具Maven Project、Spring Boot版本1.3.6以及一些工程基本信息，点击“Switch to the full version.”java版本选择1.7，可参考下图所示：



- 3、点击Generate Project下载项目压缩包
- 4、解压后，使用eclipse，Import -> Existing Maven Projects -> Next ->选择解压后的文件夹-> Finsh，OK done!

项目结构介绍



如上图所示，Spring Boot的基础结构共三个文件:

- src/main/java 程序开发以及主程序入口
- src/main/resources 配置文件
- src/test/java 测试程序

另外，spingboot建议的目录结果如下：

root package结构： `com.example.myproject`

```
com
  +- example
    +- myproject
      +- Application.java
      |
      +- domain
      |  +- Customer.java
      |  +- CustomerRepository.java
      |
      +- service
      |  +- CustomerService.java
      |
      +- controller
      |  +- CustomerController.java
      |
```

- 1、Application.java 建议放到根目录下面,主要用于做一些框架配置
- 2、domain目录主要用于实体（Entity）与数据访问层（Repository）
- 3、service 层主要是业务类代码
- 4、controller 负责页面访问控制

采用默认配置可以省去很多配置，当然也可以根据自己的喜欢来进行更改
最后，启动Application main方法，至此一个java项目搭建好了！

引入**web**模块

1、pom.xml中添加支持web的模块：

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

pom.xml文件中默认有两个模块：

`spring-boot-starter` ：核心模块，包括自动配置支持、日志和YAML；

`spring-boot-starter-test` ：测试模块，包括JUnit、Hamcrest、Mockito。

2、编写controller内容：

```
@RestController
public class HelloWorldController {
    @RequestMapping("/hello")
    public String index() {
        return "Hello World";
    }
}
```

`@RestController` 的意思就是controller里面的方法都以json格式输出，不用再写什么jackjson配置的了！

3、启动主程序，打开浏览器访问http://localhost:8080/hello，就可以看到效果了，有木有很简单！

如何做单元测试

打开的src/test/下的测试入口，编写简单的http请求来测试；使用mockmvc进行，利用MockMvcResultHandlers.print()打印出执行结果。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class HelloTests {


    private MockMvc mvc;

    @Before
    public void setUp() throws Exception {
        mvc = MockMvcBuilders.standaloneSetup(new HelloWorldCont
roller()).build();
    }

    @Test
    public void getHello() throws Exception {
        mvc.perform(MockMvcRequestBuilders.get("/hello").accept(
MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("Hello World"
)));
    }

}
```

开发环境的调试

热启动在正常开发项目中已经很常见了吧，虽然平时开发web项目过程中，改动项目启重启总是报错；但springBoot对调试支持很好，修改之后可以实时生效，需要添加以下的配置：

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

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

该模块在完整的打包环境下运行的时候会被禁用。如果你使用java -jar启动应用或者用一个特定的classloader启动，它会认为这是一个“生产环境”。

总结

使用spring boot可以非常方便、快速搭建项目，使我们不用关心框架之间的兼容性，适用版本等各种问题，我们想使用任何东西，仅仅添加一个配置就可以，所以使用sping boot非常适合构建微服务。

上篇文章介绍了Spring boot初级教程：spring boot(一)：入门篇-%E5%85%A5%E9%97%A8%E7%AF%87.html)，方便大家快速入门、了解实践Spring boot特性；本篇文章接着上篇内容继续为大家介绍spring boot的其它特性（有些未必是spring boot体系栈的功能，但是是spring特别推荐的一些开源技术本文也会介绍），对了这里只是一个大概的介绍，特别详细的使用我们会在其它的文章中来展开说明。

web开发

spring boot web开发非常的简单，其中包括常用的json输出、filters、property、log等

json 接口开发

在以前的spring 开发的时候需要我们提供json接口的时候需要做那些配置呢

1. 添加 jackjson 等相关jar包
2. 配置spring controller扫描
3. 对接的方法添加@ResponseBody

就这样我们会经常由于配置错误，导致406错误等等，spring boot如何做呢，只需要类添加 `@RestController` 即可，默认类中的方法都会以json的格式返回

```
@RestController
public class HelloWorldController {
    @RequestMapping("/getUser")
    public User getUser() {
        User user=new User();
        user.setUserName("小明");
        user.setPassWord("xxxx");
        return user;
    }
}
```

如果我们需要使用页面开发只要使用 `@Controller` ，下面会结合模板来说明

自定义Filter

我们常常在项目中会使用filters用于录调用日志、排除有XSS威胁的字符、执行权限验证等等。Spring Boot自动添加了OrderedCharacterEncodingFilter和HiddenHttpMethodFilter，并且我们可以自定义Filter。

两个步骤：

1. 实现Filter接口，实现Filter方法
2. 添加 `@Configuration` 注解，将自定义Filter加入过滤链

好吧，直接上代码

```
@Configuration
public class WebConfiguration {
    @Bean
    public RemoteIpFilter remoteIpFilter() {
        return new RemoteIpFilter();
    }

    @Bean
    public FilterRegistrationBean testFilterRegistration() {

        FilterRegistrationBean registration = new FilterRegistra
tionBean();
        registration.setFilter(new MyFilter());
        registration.addUrlPatterns("/*");
        registration.addInitParameter("paramName", "paramValue")
;
        registration.setName("MyFilter");
        registration.setOrder(1);
        return registration;
    }

    public class MyFilter implements Filter {
        @Override
        public void destroy() {
            // TODO Auto-generated method stub
        }

        @Override
        public void doFilter(ServletRequest srequest, ServletRes
```

```
ponse sresponse, FilterChain filterChain)
                throws IOException, ServletException {
            // TODO Auto-generated method stub
            HttpServletRequest request = (HttpServletRequest) sr
equest;
            System.out.println("this is MyFilter,url :"+request.
getRequestURI());
            filterChain.doFilter(srequest, sresponse);
        }

        @Override
        public void init(FilterConfig arg0) throws ServletExcept
ion {
            // TODO Auto-generated method stub
        }
    }
}
```

自定义Property

在web开发的过程中，我经常需要自定义一些配置文件，如何使用呢

配置在application.properties中

```
com.neo.title=纯洁的微笑
com.neo.description=分享生活和技术
```

自定义配置类

```
@Component
public class NeoProperties {
    @Value("${com.neo.title}")
    private String title;
    @Value("${com.neo.description}")
    private String description;

    //省略getter settet方法

    }
```

log配置

配置输出的地址和输出级别

```
logging.path=/user/local/log
logging.level.com.favorites=DEBUG
logging.level.org.springframework.web=INFO
logging.level.org.hibernate=ERROR
```

path为本机的log地址，`logging.level` 后面可以根据包路径配置不同资源的log级别

数据库操作

在这里我重点讲述mysql、spring data jpa的使用，其中mysql 就不用说了大家很熟悉，jpa是利用Hibernate生成各种自动化的sql，如果只是简单的增删改查，基本上不用手写了，spring内部已经帮大家封装实现了。

下面简单介绍一下如何在spring boot中使用

1、添加相jar包

```
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
     <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
```

2、添加配置文件

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

spring.jpa.properties.hibernate.hbm2ddl.auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.My
SQL5InnoDBDialect
spring.jpa.show-sql= true
```

其实这个hibernate.hbm2ddl.auto参数的作用主要用于：自动创建|更新|验证数据库表结构,有四个值：

1. create： 每次加载hibernate时都会删除上一次的生成的表，然后根据你的model类再重新来生成新表，哪怕两次没有任何改变也要这样执行，这就是导致数据库表数据丢失的一个重要原因。
2. create-drop ：每次加载hibernate时根据model类生成表，但是sessionFactory一关闭,表就自动删除。
3. update：最常用的属性，第一次加载hibernate时根据model类会自动建立起表的结构（前提是先建立好数据库），以后加载hibernate时根据 model类自动更新表结构，即使表结构改变了但表中的行仍然存在不会删除以前的行。要注意的是当部署到服务器后，表结构是不会被马上建立起来的，是要等 应用第一次运行起来后才会。
4. validate ：每次加载hibernate时，验证创建数据库表结构，只会和数据库中的表进行比较，不会创建新表，但是会插入新值。

`dialect` 主要是指定生成表名的存储引擎为InneoDB
`show-sql` 是否打印出自动生产的SQL，方便调试的时候查看

3、添加实体类和Dao

```
@Entity
public class User implements Serializable {

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue
    private Long id;
    @Column(nullable = false, unique = true)
    private String userName;
    @Column(nullable = false)
    private String passWord;
    @Column(nullable = false, unique = true)
    private String email;
    @Column(nullable = true, unique = true)
    private String nickName;
    @Column(nullable = false)
    private String regTime;

    //省略getter settet方法、构造方法

}
```

dao只要继承JpaRepository类就可以，几乎可以不用写方法，还有一个特别有尿性的功能非常赞，就是可以根据方法名来自动的生产SQL，比如 `findByUserName` 会自动生产一个以 `userName` 为参数的查询方法，比如 `findAlll` 自动会查询表里面的所有数据，比如自动分页等等。。

Entity中不映射成列的字段得加**@Transient** 注解，不加注解也会映射成列

```
public interface UserRepository extends JpaRepository<User, Long
> {
    User findByUserName(String userName);
    User findByUserNameOrEmail(String username, String email);
```

4、测试

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(Application.class)
public class UserRepositoryTests {

    @Autowired
    private UserRepository userRepository;

    @Test
    public void test() throws Exception {
        Date date = new Date();
        DateFormat dateFormat = DateFormat.getDateTimeInstance(D
ateFormat.LONG, DateFormat.LONG);
        String formattedDate = dateFormat.format(date);

        userRepository.save(new User("aa1", "aa@126.com", "aa",
"aa123456",formattedDate));
        userRepository.save(new User("bb2", "bb@126.com", "bb",
"bb123456",formattedDate));
        userRepository.save(new User("cc3", "cc@126.com", "cc",
"cc123456",formattedDate));

        Assert.assertEquals(9, userRepository.findAll().size());
        Assert.assertEquals("bb", userRepository.findByUserNameO
rEmail("bb", "cc@126.com").getNickName());
        userRepository.delete(userRepository.findByUserName("aa1"
));
    }

}
```

当让 spring data jpa 还有很多功能，比如封装好的分页，可以自己定义SQL，主从分离等等，这里就不详细讲了

thymeleaf模板

Spring boot 推荐使用来代替jsp,thymeleaf模板到底是什么来头呢，让spring大哥来推荐，下面我们来聊聊

Thymeleaf 介绍

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

好了，你们说了我们已经习惯使用了什么 velocity,FreMaker，beetle之类的模版，那么到底好在哪里呢？ 比一比吧 Thymeleaf是与众不同的，因为它使用了自然的模板技术。这意味着Thymeleaf的模板语法并不会破坏文档的结构，模板依旧是有效的XML文档。模板还可以用作工作原型，Thymeleaf会在运行期替换掉静态值。Velocity与FreeMarker则是连续的文本处理器。 下面的代码示例分别使用Velocity、FreeMarker与Thymeleaf打印出一条消息：

```
Velocity: <p>$message</p>
FreeMarker: <p>${message}</p>
Thymeleaf: <p th:text="${message}">Hello World!</p>
```

注意，由于**Thymeleaf**使用了**XML DOM**解析器，因此它并不适合于处理大规模的**XML**文件。

URL

URL在Web应用模板中占据着十分重要的地位，需要特别注意的是Thymeleaf对于URL的处理是通过语法@{...}来处理的。Thymeleaf支持绝对路径URL：

```
<a th:href="@{http://www.thymeleaf.org}">Thymeleaf</a>
```

条件求值

```
<a th:href="@{/login}" th:unless=${session.user != null}>Login</a
>
```

for循环

```
<tr th:each="prod : ${prods}">
      <td th:text="${prod.name}">Onions</td>
      <td th:text="${prod.price}">2.41</td>
      <td th:text="${prod.inStock}? #{true} : #{false}">yes</td>
</tr>
```

就列出这几个吧

页面即原型

在Web开发过程中一个绕不开的话题就是前端工程师与后端工程师的写作，在传统Java Web开发过程中，前端工程师和后端工程师一样，也需要安装一套完整的开发环境，然后各类Java IDE中修改模板、静态资源文件，启动/重启/重新加载应用服务器，刷新页面查看最终效果。

但实际上前端工程师的职责更多应该关注于页面本身而非后端，使用JSP，Velocity等传统的Java模板引擎很难做到这一点，因为它们必须在应用服务器中渲染完成后才能在浏览器中看到结果，而Thymeleaf从根本上颠覆了这一过程，通过属性进行模板渲染不会引入任何新的浏览器不能识别的标签，例如JSP中的，不会在Tag内部写表达式。整个页面直接作为HTML文件用浏览器打开，几乎就可以看到最终的效果，这大大解放了前端工程师的生产力，它们的最终交付物就是纯的HTML/CSS/JavaScript文件。

Gradle 构建工具

spring 项目建议使用Gradle进行构建项目，相比maven来讲 Gradle更简洁，而且gradle更时候大型复杂项目的构建。gradle吸收了maven和ant的特点而来，不过目前maven仍然是Java界的主流，大家可以先了解了解。

一个使用gradle配置的项目

```
buildscript {
    repositories {
        maven { url "http://repo.spring.io/libs-snapshot" }
        mavenLocal()
    }
    dependencies {
        classpath("org.springframework.boot:spring-boot-gradle-p
lugin:1.3.6.RELEASE")
    }
}

apply plugin: 'java'  //添加 Java 插件, 表明这是一个 Java 项目
apply plugin: 'spring-boot' //添加 Spring-boot支持
apply plugin: 'war'  //添加 War 插件, 可以导出 War 包
apply plugin: 'eclipse' //添加 Eclipse 插件, 添加 Eclipse IDE 支持,
 Intellij Idea 为 "idea"

war {
    baseName = 'favorites'
    version =  '0.1.0'
}

sourceCompatibility = 1.7  //最低兼容版本 JDK1.7
targetCompatibility = 1.7  //目标兼容版本 JDK1.7

repositories {     //  Maven 仓库
    mavenLocal()        //使用本地仓库
    mavenCentral()      //使用中央仓库
    maven { url "http://repo.spring.io/libs-snapshot" } //使用远
程仓库
}

dependencies {   // 各种 依赖的jar包
    compile("org.springframework.boot:spring-boot-starter-web:1.
```

```
3.6.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-thymel
eaf:1.3.6.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-data-j
pa:1.3.6.RELEASE")
    compile group: 'mysql', name: 'mysql-connector-java', versio
n: '5.1.6'
    compile group: 'org.apache.commons', name: 'commons-lang3',
version: '3.4'
    compile("org.springframework.boot:spring-boot-devtools:1.3.6
.RELEASE")
    compile("org.springframework.boot:spring-boot-starter-test:1
.3.6.RELEASE")
    compile 'org.webjars.bower:bootstrap:3.3.6'
    compile 'org.webjars.bower:jquery:2.2.4'
    compile("org.webjars:vue:1.0.24")
    compile 'org.webjars.bower:vue-resource:0.7.0'

}

bootRun {
    addResources = true
}
```

WebJars

WebJars是一个很神奇的东西，可以让大家以jar包的形式来使用前端的各种框架、组件。

什么是WebJars

什么是WebJars？WebJars是将客户端（浏览器）资源（JavaScript，Css等）打成jar包文件，以对资源进行统一依赖管理。WebJars的jar包部署在Maven中央仓库上。

为什么使用

我们在开发Java web项目的时候会使用像Maven，Gradle等构建工具以实现对jar包版本依赖管理，以及项目的自动化管理，但是对于JavaScript，Css等前端资源包，我们只能采用拷贝到webapp下的方式，这样做就无法对这些资源进行依赖管理。那么WebJars就提供给我们这些前端资源的jar包形势，我们就可以进行依赖管理。

如何使用

1、 WebJars主官网 查找对于的组件，比如Vuejs

```
<dependency>
    <groupId>org.webjars.bower</groupId>
    <artifactId>vue</artifactId>
    <version>1.0.21</version>
</dependency>
```

2、页面引入

```
<link th:href="@{/webjars/bootstrap/3.3.6/dist/css/bootstrap.css
}" rel="stylesheet"></link>
```

就可以正常使用了！

参考：

新一代Java模板引擎Thymeleaf

Spring Boot参考指南-中文版

Thymeleaf 是新一代的模板引擎，在spring4.0中推荐使用thymeleaf来做前端模版引擎。

thymeleaf 介绍

简单说， Thymeleaf 是一个跟 Velocity、FreeMarker 类似的模板引擎，它可以完全替代 JSP 。相较与其他的模板引擎，它有如下三个极吸引人的特点：

- 1.Thymeleaf 在有网络和无网络的环境下皆可运行，即它可以让美工在浏览器查看页面的静态效果，也可以让程序员在服务器查看带数据的动态页面效果。这是由于它支持 html 原型，然后在 html 标签里增加额外的属性来达到模板+数据的展示方式。浏览器解释 html 时会忽略未定义的标签属性，所以 thymeleaf 的模板可以静态地运行；当有数据返回到页面时，Thymeleaf 标签会动态地替换掉静态内容，使页面动态显示。
- 2.Thymeleaf 开箱即用的特性。它提供标准和spring标准两种方言，可以直接套用模板实现JSTL、 OGNL表达式效果，避免每天套模板、该jstl、改标签的困扰。同时开发人员也可以扩展和创建自定义的方言。
- 3.Thymeleaf 提供spring标准方言和一个与 SpringMVC 完美集成的可选模块，可以快速的实现表单绑定、属性编辑器、国际化等功能。

标准表达式语法

它们分为四类：

- 1.变量表达式
- 2.选择或星号表达式
- 3.文字国际化表达式
- 4.URL表达式

变量表达式

变量表达式即OGNL表达式或Spring EL表达式(在Spring术语中也叫model attributes)。如下所示：

```
${session.user.name}
```

它们将以HTML标签的一个属性来表示：

```
<span th:text="${book.author.name}">
<li th:each="book : ${books}">
```

选择(星号)表达式

选择表达式很像变量表达式，不过它们用一个预先选择的对象来代替上下文变量容器(map)来执行，如下：
`*{customer.name}`

被指定的object由th:object属性定义：

```
<div th:object="${book}">
  ...
  <span th:text="*{title}">...</span>
  ...
</div>
```

文字国际化表达式

文字国际化表达式允许我们从一个外部文件获取区域文字信息(.properties)，用Key索引Value，还可以提供一组参数(可选).

```
#{main.title}
#{message.entrycreated(${entryId})}
```

可以在模板文件中找到这样的表达式代码：

```
<table>
  ...
  <th th:text="#{header.address.city}">...</th>
  <th th:text="#{header.address.country}">...</th>
  ...
</table>
```

URL表达式

URL表达式指的是把一个有用的上下文或回话信息添加到URL，这个过程经常被叫做URL重写。

`@{/order/list}`

URL还可以设置参数：

`@{/order/details(id=${orderId})}`

相对路径：

`@{../documents/report}`

让我们看这些表达式：

```
    <form th:action="@{/createOrder}">
    <a href="main.html" th:href="@{/main}">
```

变量表达式和星号表达有什么区别吗？

如果不考虑上下文的情况下，两者没有区别；星号语法评估在选定对象上表达，而不是整个上下文

什么是选定对象？就是父标签的值，如下：

```
  <div th:object="${session.user}">
    <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>
    <p>Surname: <span th:text="*{lastName}">Pepper</span>.</p>
    <p>Nationality: <span th:text="*{nationality}">Saturn</span>.
</p>
  </div>
```

这是完全等价于：

```
  <div th:object="${session.user}">
      <p>Name: <span th:text="${session.user.firstName}">Sebasti
an</span>.</p>
      <p>Surname: <span th:text="${session.user.lastName}">Pepper
</span>.</p>
      <p>Nationality: <span th:text="${session.user.nationality}"
>Saturn</span>.</p>
  </div>
```

当然，美元符号和星号语法可以混合使用：

```
  <div th:object="${session.user}">
      <p>Name: <span th:text="*{firstName}">Sebastian</span>.</p>

        <p>Surname: <span th:text="${session.user.lastName}">Pep
per</span>.</p>
      <p>Nationality: <span th:text="*{nationality}">Saturn</span
>.</p>
  </div>
```

表达式支持的语法

字面（**Literals**）

- 文本文字（Text literals）：`'one text', 'Another one!',…`
- 数字文本（Number literals）：`0, 34, 3.0, 12.3,…`
- 布尔文本（Boolean literals）：`true, false`
- 空（Null literal）：`null`
- 文字标记（Literal tokens）：`one, sometext, main,…`

文本操作（**Text operations**）

- 字符串连接(String concatenation): `+`
- 文本替换（Literal substitutions）：`|The name is ${name}|`

算术运算（**Arithmetic operations**）

- 二元运算符（Binary operators）：`+, -, *, /, %`
- 减号（单目运算符）Minus sign (unary operator): `-`

布尔操作（**Boolean operations**）

- 二元运算符（Binary operators）：`and, or`
- 布尔否定（一元运算符）Boolean negation (unary operator): `!, not`

比较和等价**(Comparisons and equality)**

- 比较（Comparators）：`>, <, >=, <= (gt, lt, ge, le)`
- 等值运算符（Equality operators）：`==, != (eq, ne)`

条件运算符（**Conditional operators**）

- If-then: `(if) ? (then)`
- If-then-else: `(if) ? (then) : (else)`
- Default: (value) ?: `(defaultvalue)`

所有这些特征可以被组合并嵌套：

```
'User is of type ' + (${user.isAdmin()} ? 'Administrator' : (${u
ser.type} ?: 'Unknown'))
```

常用**th**标签都有那些？

关键字	功能介绍	案例
th:id	替换id	`<input th:id="'xxx' + ${collect.id}"/>`
th:text	文本替换	`<p th:text="${collect.description}">description<`
th:utext	支持html的文本替换	`<p th:utext="${htmlcontent}">conten</p>`
th:object	替换对象	`<div th:object="${session.user}">`

th:value	属性赋值	`<input th:value="${user.name}" />`
th:with	变量赋值运算	`<div th:with="isEven=${prodStat.count}%2==0" </div>`
th:style	设置样式	`th:style="'display:' + @{(${sitrue} ? 'none' 'inline-block')} + ''"`
th:onclick	点击事件	`th:onclick="'getCollect()'"`
th:each	属性赋值	`tr th:each="user,userStat:${users}">`
th:if	判断条件	`<a th:if="${userId == collect.userId}" >`
th:unless	和th:if判断相反	`<a th:href="@{/login}" th:unless=${session.u != null}>Login</a>`
th:href	链接地址	`<a th:href="@{/login}" th:unless=${session.u != null}>Login</a> />`
th:switch	多路选择 配合th:case使用	`<div th:switch="${user.role}">`
th:case	th:switch的一个分支	`<p th:case="'admin'">User is an administrator</p>`
th:fragment	布局标签，定义一个代码片段，方便其它地方引用	`<div th:fragment="alert">`
th:include	布局标签，替换内容到引入的文件	`<head th:include="layout :: htmlhead" th:with="title='xx'"></head> />`
th:replace	布局标签，替换整个标签到	`<div th:replace="fragments/header :: title"> </div>`

	引入的文件	
th:selected	selected选择框选中	th:selected="(${xxx.id} == ${configObj.dd})"
th:src	图片类地址引入	
th:inline	定义js脚本可以使用变量	<script type="text/javascript" th:inline="javascript">
th:action	表单提交的地址	<form action="subscribe.html" th:action="@{/subscribe}">
th:remove	删除某个属性	<tr th:remove="all"> 1.all:删除包含标签和所有的子。2.body:不包含标记删除,但删除其所有的孩子。3.tag 标记的删除,但不删除它的孩子。4.all-but-first:删除所含标签的孩子,除了第一个。5.none:什么也不做。这个值是的动态评估。
th:attr	设置标签属性，多个属性可以用逗号分隔	比如 th:attr="src=@{/image/aa.jpg},title=#{logo}" ，此标签不太优雅，一般用的比较少。

还有非常多的标签，这里只列出最常用的几个,由于一个标签内可以包含多个th:x属性，其生效的优先级顺序为:

include,each,if/unless/switch/case,with,attr/attrprepend/attrappend,value/href,src ,etc,text/utext,fragment,remove。

几种常用的使用方法

1、赋值、字符串拼接

```
 <p  th:text="${collect.description}">description</p>
 <span th:text="'Welcome to our application, ' + ${user.name} +
'!'">
```

字符串拼接还有另外一种简洁的写法

```
<span th:text="|Welcome to our application, ${user.name}!|">
```

2、条件判断 If/Unless

Thymeleaf中使用th:if和th:unless属性进行条件判断，下面的例子中，`<a>` 标签只有在 `th:if` 中条件成立时才显示：

```
<a th:if="${myself=='yes'}" > </i> </a>
<a th:unless=${session.user != null} th:href="@{/login}" >Login</
a>
```

th:unless于th:if恰好相反，只有表达式中的条件不成立，才会显示其内容。

也可以使用 `(if) ? (then) : (else)` 这种语法来判断显示的内容

3、for 循环

```
  <tr  th:each="collect,iterStat : ${collects}">
     <th scope="row" th:text="${collect.id}">1</th>
     <td >
        <img th:src="${collect.webLogo}"/>
     </td>
     <td th:text="${collect.url}">Mark</td>
     <td th:text="${collect.title}">Otto</td>
     <td th:text="${collect.description}">@mdo</td>
     <td th:text="${terStat.index}">index</td>
 </tr>
```

iterStat称作状态变量，属性有：

- index:当前迭代对象的index（从0开始计算）
- count: 当前迭代对象的index(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

4、URL

URL在Web应用模板中占据着十分重要的地位，需要特别注意的是Thymeleaf对于URL的处理是通过语法@{...}来处理的。 如果需要Thymeleaf对URL进行渲染，那么务必使用th:href，th:src等属性，下面是一个例子

```
<!-- Will produce 'http://localhost:8080/standard/unread' (plus
rewriting) -->
 <a  th:href="@{/standard/{type}(type=${type})}">view</a>

<!-- Will produce '/gtvg/order/3/details' (plus rewriting) -->
<a href="details.html" th:href="@{/order/{orderId}/details(order
Id=${o.id})}">view</a>
```

设置背景

```
<div th:style="'background:url(' + @{/<path-to-image>} + ');'"></
div>
```

根据属性值改变背景

```
 <div class="media-object resource-card-image"  th:style="'backg
round:url(' + @{(${collect.webLogo}=='' ? 'img/favicon.png' : ${
collect.webLogo})} + ')'" ></div>
```

几点说明：

- 上例中URL最后的 `(orderId=${o.id})` 表示将括号内的内容作为URL参数

处理，该语法避免使用字符串拼接，大大提高了可读性

- `@{...}` 表达式中可以通过 `{orderId}` 访问Context中的orderId变量
- `@{/order}` 是Context相关的相对路径，在渲染时会自动添加上当前Web应用的Context名字，假设context名字为app，那么结果应该是/app/order

5、内联js

内联文本：[[...]]内联文本的表示方式，使用时，必须先用th:inline="text/javascript/none"激活，th:inline可以在父级标签内使用，甚至作为body的标签。内联文本尽管比th:text的代码少，不利于原型显示。

```
<script th:inline="javascript">
/*<![CDATA[*/
...
var username = /*[[${sesion.user.name}]]*/ 'Sebastian';
var size = /*[[${size}]]*/ 0;
...
/*]]>*/
</script>
```

js附加代码：

```
/*[+
var msg = 'This is a working application';
+]*/
```

js移除代码：

```
/*[- */
var msg = 'This is a non-working template';
/* -]*/
```

6、内嵌变量

为了模板更加易用，Thymeleaf还提供了一系列Utility对象（内置于Context中），可以通过#直接访问：

- dates ： *java.util.Date*的功能方法类。
- calendars : 类似*#dates*，面向*java.util.Calendar*
- numbers : 格式化数字的功能方法类
- strings : 字符串对象的功能类，*contains,startWiths,prepending/appending*等等。
- objects: 对*objects*的功能类操作。
- bools: 对布尔值求值的功能方法。
- arrays：对数组的功能类方法。
- lists: 对*lists*功能类方法
- sets
- maps

  ...

下面用一段代码来举例一些常用的方法：

dates

```
/*
 * Format date with the specified pattern
 * Also works with arrays, lists or sets
 */
${#dates.format(date, 'dd/MMM/yyyy HH:mm')}
${#dates.arrayFormat(datesArray, 'dd/MMM/yyyy HH:mm')}
${#dates.listFormat(datesList, 'dd/MMM/yyyy HH:mm')}
${#dates.setFormat(datesSet, 'dd/MMM/yyyy HH:mm')}

/*
 * Create a date (java.util.Date) object for the current date an
d time
 */
${#dates.createNow()}

/*
 * Create a date (java.util.Date) object for the current date (t
ime set to 00:00)
 */
${#dates.createToday()}
```

strings

```
/*
 * Check whether a String is empty (or null). Performs a trim()
operation before check
 * Also works with arrays, lists or sets
 */
${#strings.isEmpty(name)}
${#strings.arrayIsEmpty(nameArr)}
${#strings.listIsEmpty(nameList)}
${#strings.setIsEmpty(nameSet)}

/*
 * Check whether a String starts or ends with a fragment
 * Also works with arrays, lists or sets
 */
${#strings.startsWith(name,'Don')}                  // also arra
y*, list* and set*
${#strings.endsWith(name,endingFragment)}           // also arra
y*, list* and set*

/*
 * Compute length
 * Also works with arrays, lists or sets
 */
${#strings.length(str)}

/*
 * Null-safe comparison and concatenation
 */
${#strings.equals(str)}
${#strings.equalsIgnoreCase(str)}
${#strings.concat(str)}
${#strings.concatReplaceNulls(str)}

/*
 * Random
 */
${#strings.randomAlphanumeric(count)}
```

使用thymeleaf布局

使用thymeleaf布局非常的方便

定义代码片段

```
<footer th:fragment="copy">
&copy; 2016
</footer>
```

在页面任何地方引入：

```
<body>
  <div th:include="footer :: copy"></div>
  <div th:replace="footer :: copy"></div>
 </body>
```

th:include 和 th:replace区别，include只是加载，replace是替换

返回的HTML如下：

```
<body>
   <div> &copy; 2016 </div>
  <footer>&copy; 2016 </footer>
</body>
```

下面是一个常用的后台页面布局，将整个页面分为头部，尾部、菜单栏、隐藏栏，点击菜单只改变content区域的页面

```
<body class="layout-fixed">
  <div th:fragment="navbar"  class="wrapper"  role="navigation">
    <div th:replace="fragments/header :: header">Header</div>
    <div th:replace="fragments/left :: left">left</div>
    <div th:replace="fragments/sidebar :: sidebar">sidebar</div>
    <div layout:fragment="content" id="content" ></div>
    <div th:replace="fragments/footer :: footer">footer</div>
  </div>
</body>
```

任何页面想使用这样的布局值只需要替换中见的 content模块即可

```
 <html xmlns:th="http://www.thymeleaf.org" layout:decorator="lay
out">
   <body>
      <section layout:fragment="content">
    ...
```

也可以在引用模版的时候传参

```
<head th:include="layout :: htmlhead" th:with="title='Hello'"></
head>
```

layout 是文件地址，如果有文件夹可以这样写 fileName/layout:htmlhead
htmlhead 是指定义的代码片段 如 `th:fragment="copy"`

源码案例

这里有一个开源项目几乎使用了这里介绍的所有标签和布局，大家可以参考：

示例代码-github

示例代码-码云

参考

thymeleaf官方指南
新一代Java模板引擎Thymeleaf
Thymeleaf基本知识
thymeleaf总结文章
Thymeleaf 模板的使用
thymeleaf 学习笔记

这篇文章介绍如何使用jpa和thymeleaf做一个增删改查的示例。

快速上手

配置文件

pom包配置

pom包里面添加jpa和thymeleaf的相关包引用

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

在**application.properties**中添加配置

```
spring.datasource.url=jdbc:mysql://127.0.0.1/test?useUnicode=tru
e&characterEncoding=utf-8&serverTimezone=UTC&useSSL=true
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.jdbc.Driver

spring.jpa.properties.hibernate.hbm2ddl.auto=update
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.My
SQL5InnoDBDialect
spring.jpa.show-sql= true

spring.thymeleaf.cache=false
```

其中 `propertiesspring.thymeleaf.cache=false` 是关闭thymeleaf的缓存，不然在开发过程中修改页面不会立刻生效需要重启，生产可配置为true。

在项目resources目录下会有两个文件夹：static目录用于放置网站的静态内容如css、js、图片；templates目录用于放置项目使用的页面模板。

启动类

启动类需要添加Servlet的支持

```
@SpringBootApplication
public class JpaThymeleafApplication extends SpringBootServletIn
itializer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicati
onBuilder application) {
        return application.sources(JpaThymeleafApplication.class
);
    }

    public static void main(String[] args) throws Exception {
        SpringApplication.run(JpaThymeleafApplication.class, arg
s);
    }
}
```

数据库层代码

实体类映射数据库表

```
@Entity
public class User {
    @Id
    @GeneratedValue
    private long id;
    @Column(nullable = false, unique = true)
    private String userName;
    @Column(nullable = false)
    private String password;
    @Column(nullable = false)
    private int age;
    ...
}
```

继承JpaRepository类会自动实现很多内置的方法，包括增删改查。也可以根据方法名来自动生成相关sql，具体可以参考：springboot(五)：spring data jpa的使用-spring-data-jpa%E7%9A%84%E4%BD%BF%E7%94%A8.html)

```
public interface UserRepository extends JpaRepository<User, Long
> {
    User findById(long id);
    Long deleteById(Long id);
}
```

业务层处理

service调用jpa实现相关的增删改查，实际项目中service层处理具体的业务代码。

```
@Service
public class UserServiceImpl implements UserService{

    @Autowired
    private UserRepository userRepository;

    @Override
    public List<User> getUserList() {
        return userRepository.findAll();
    }

    @Override
    public User findUserById(long id) {
        return userRepository.findById(id);
    }

    @Override
    public void save(User user) {
        userRepository.save(user);
    }

    @Override
    public void edit(User user) {
        userRepository.save(user);
    }

    @Override
    public void delete(long id) {
        userRepository.delete(id);
    }
}
```

Controller负责接收请求，处理完后将页面内容返回给前端。

```
@Controller
public class UserController {

    @Resource
    UserService userService;
```

```
@RequestMapping("/")
public String index() {
    return "redirect:/list";
}

@RequestMapping("/list")
public String list(Model model) {
    List<User> users=userService.getUserList();
    model.addAttribute("users", users);
    return "user/list";
}

@RequestMapping("/toAdd")
public String toAdd() {
    return "user/userAdd";
}

@RequestMapping("/add")
public String add(User user) {
    userService.save(user);
    return "redirect:/list";
}

@RequestMapping("/toEdit")
public String toEdit(Model model,Long id) {
    User user=userService.findUserById(id);
    model.addAttribute("user", user);
    return "user/userEdit";
}

@RequestMapping("/edit")
public String edit(User user) {
    userService.edit(user);
    return "redirect:/list";
}


@RequestMapping("/delete")
```

```
    public String delete(Long id) {
        userService.delete(id);
        return "redirect:/list";
    }
}
```

- `return "user/userEdit";` 代表会直接去resources目录下找相关的文件。
- `return "redirect:/list";` 代表转发到对应的controller，这个示例就相当于删除内容之后自动调整到list请求，然后再输出到页面。

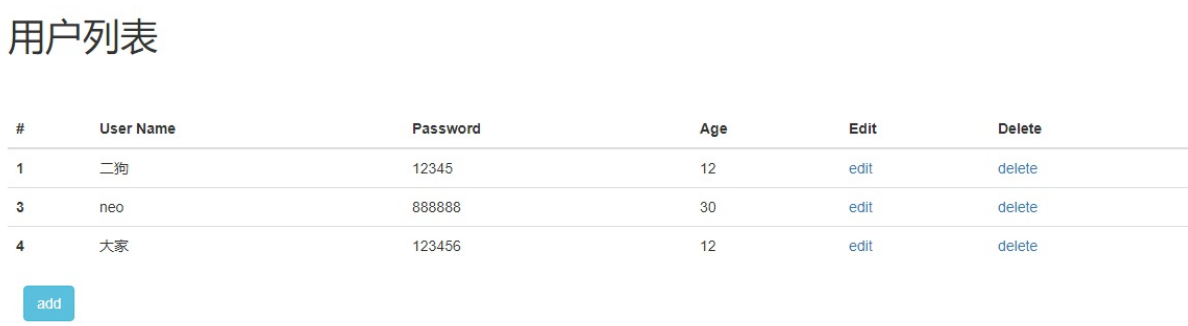
页面内容

list列表

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>userList</title>
    <link rel="stylesheet" th:href="@{/css/bootstrap.css}"></link
>
</head>
<body class="container">
<br/>
<h1>用户列表</h1>
<br/><br/>
<div class="with:80%">
    <table class="table table-hover">
        <thead>
        <tr>
            <th>#</th>
            <th>User Name</th>
            <th>Password</th>
            <th>Age</th>
            <th>Edit</th>
            <th>Delete</th>
        </tr>
        </thead>
        <tbody>
```

```
        <tr  th:each="user : ${users}">
            <th scope="row" th:text="${user.id}">1</th>
            <td th:text="${user.userName}">neo</td>
            <td th:text="${user.password}">Otto</td>
            <td th:text="${user.age}">6</td>
            <td><a th:href="@{/toEdit(id=${user.id})}">edit</a></
td>
            <td><a th:href="@{/delete(id=${user.id})}">delete</a>
</td>
        </tr>
        </tbody>
    </table>
</div>
<div class="form-group">
    <div class="col-sm-2 control-label">
        <a href="/toAdd" th:href="@{/toAdd}" class="btn btn-info"
>add</a>
    </div>
</div>

</body>
</html>
```

效果图：



`<tr th:each="user : ${users}">` 这里会从controler层model set的对象去获取相关的内容，`th:each` 表示会循环遍历对象内容。

其实还有其它的写法，具体的语法内容可以参考这篇文章：springboot(四)：thymeleaf使用详解-thymeleaf%E4%BD%BF%E7%94%A8%E8%AF%A6%E8%A7%A3.html)

修改页面：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="UTF-8"/>
    <title>user</title>
    <link rel="stylesheet" th:href="@{/css/bootstrap.css}"></link
>
</head>
<body class="container">
<br/>
<h1>修改用户</h1>
<br/><br/>
<div class="with:80%">
    <form class="form-horizontal"   th:action="@{/edit}" th:obje
ct="${user}"  method="post">
        <input type="hidden" name="id" th:value="*{id}" />
        <div class="form-group">
            <label for="userName" class="col-sm-2 control-label">
userName</label>
            <div class="col-sm-10">
                <input type="text" class="form-control" name="us
erName"  id="userName" th:value="*{userName}" placeholder="userN
ame"/>
            </div>
        </div>
        <div class="form-group">
            <label for="password" class="col-sm-2 control-label"
 >Password</label>
            <div class="col-sm-10">
                <input type="password" class="form-control" name=
"password" id="password"  th:value="*{password}" placeholder="Pa
ssword"/>
            </div>
        </div>
        <div class="form-group">
            <label for="age" class="col-sm-2 control-label">age</
label>
            <div class="col-sm-10">
```

```
                <input type="text" class="form-control" name="ag
e"  id="age" th:value="*{age}" placeholder="age"/>
            </div>
        </div>
        <div class="form-group">
            <div class="col-sm-offset-2 col-sm-10">
                <input type="submit" value="Submit" class="btn b
tn-info" />
                     
                <a href="/toAdd" th:href="@{/list}" class="btn b
tn-info">Back</a>
            </div>

        </div>
    </form>
</div>
</body>
</html>
```

添加页面和修改类似就不在贴代码了。

效果图：

这样一个使用jpa和thymeleaf的增删改查示例就完成了。

上传文件是互联网中常常应用的场景之一，最典型的情况就是上传头像等，今天就带着带着大家做一个Spring Boot上传文件的小案例。

1、pom包配置

我们使用Spring Boot最新版本1.5.9、jdk使用1.8、tomcat8.0。

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

<properties>
    <java.version>1.8</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-starter-thymeleaf</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

引入了`spring-boot-starter-thymeleaf`做页面模板引擎，写一些简单的上传示例。

2、启动类设置

```
@SpringBootApplication
public class FileUploadWebApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(FileUploadWebApplication.class, ar
gs);
    }

    //Tomcat large file upload connection reset
    @Bean
    public TomcatEmbeddedServletContainerFactory tomcatEmbedded()
 {
        TomcatEmbeddedServletContainerFactory tomcat = new Tomca
tEmbeddedServletContainerFactory();
        tomcat.addConnectorCustomizers((TomcatConnectorCustomize
r) connector -> {
            if ((connector.getProtocolHandler() instanceof Abstr
actHttp11Protocol<?>)) {
                //-1 means unlimited
                ((AbstractHttp11Protocol<?>) connector.getProtoc
olHandler()).setMaxSwallowSize(-1);
            }
        });
        return tomcat;
    }

}
```

tomcatEmbedded这段代码是为了解决，上传文件大于10M出现连接重置的问题。此异常内容GlobalException也捕获不到。

详细内容参考：Tomcat large file upload connection reset

3、编写前端页面

上传页面

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<body>
<h1>Spring Boot file upload example</h1>
<form method="POST" action="/upload" enctype="multipart/form-dat
a">
    <input type="file" name="file" /><br/><br/>
    <input type="submit" value="Submit" />
</form>
</body>
</html>
```

非常简单的一个Post请求，一个选择框选择文件，一个提交按钮，效果如下：

上传结果展示页面：

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
<body>
<h1>Spring Boot - Upload Status</h1>
<div th:if="${message}">
    <h2 th:text="${message}"/>
</div>
</body>
</html>
```

效果图如下：

4、编写上传控制类

访问localhost自动跳转到上传页面：

```
@GetMapping("/")
public String index() {
    return "upload";
}
```

上传业务处理

```
@PostMapping("/upload")
public String singleFileUpload(@RequestParam("file") MultipartFi
le file,
                               RedirectAttributes redirectAttrib
utes) {
    if (file.isEmpty()) {
        redirectAttributes.addFlashAttribute("message", "Please
select a file to upload");
        return "redirect:uploadStatus";
    }

    try {
        // Get the file and save it somewhere
        byte[] bytes = file.getBytes();
        Path path = Paths.get(UPLOADED_FOLDER + file.getOriginal
Filename());
        Files.write(path, bytes);

        redirectAttributes.addFlashAttribute("message",
                "You successfully uploaded '" + file.getOriginal
Filename() + "'");

    } catch (IOException e) {
        e.printStackTrace();
    }

    return "redirect:/uploadStatus";
}
```

上面代码的意思就是，通过 `MultipartFile` 读取文件信息，如果文件为空跳转到结果页并给出提示；如果不为空读取文件流并写入到指定目录，最后将结果展示到页面。

`MultipartFile` 是Spring上传文件的封装类，包含了文件的二进制流和文件属性等信息，在配置文件中也可对相关属性进行配置，基本的配置信息如下：

- `spring.http.multipart.enabled=true` #默认支持文件上传.
- `spring.http.multipart.file-size-threshold=0` #支持文件写入磁盘.
- `spring.http.multipart.location=` # 上传文件的临时目录

- `spring.http.multipart.max-file-size=1Mb` # 最大支持文件大小
- `spring.http.multipart.max-request-size=10Mb` # 最大支持请求大小

最常用的是最后两个配置内容，限制文件上传大小，上传时超过大小会抛出异常：

Spring Boot - Upload Status

org.apache.tomcat.util.http.fileupload.FileUploadBase$SizeLimitExceededException: the request was rejected because its size (108105436) exceeds the configured maximum (10485760)

更多配置信息参考这里：Common application properties

5、异常处理

```
@ControllerAdvice
public class GlobalExceptionHandler {

    @ExceptionHandler(MultipartException.class)
    public String handleError1(MultipartException e, RedirectAtt
ributes redirectAttributes) {
        redirectAttributes.addFlashAttribute("message", e.getCau
se().getMessage());
        return "redirect:/uploadStatus";
    }
}
```

设置一个 `@ControllerAdvice` 用来监控 `Multipart` 上传的文件大小是否受限，当出现此异常时在前端页面给出提示。利用 `@ControllerAdvice` 可以做很多东西，比如全局的统一异常处理等，感兴趣的同学可以下来了解。

6、总结

这样一个使用Spring Boot上传文件的简单Demo就完成了，感兴趣的同学可以将示例代码下载下来试试吧。

在我们的项目开发过程中，经常需要定时任务来帮助我们来做一些内容，springboot默认已经帮我们实行了，只需要添加相应的注解就可以实现

1、pom包配置

pom包里面只需要引入springboot starter包即可

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

2、启动类启用定时

在启动类上面加上 `@EnableScheduling` 即可开启定时

```
@SpringBootApplication
@EnableScheduling
public class Application {

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

3、创建定时任务实现类

定时任务1：

```
@Component
public class SchedulerTask {

    private int count=0;

    @Scheduled(cron="*/6 * * * * ?")
    private void process(){
        System.out.println("this is scheduler task runing  "+(co
unt++));
    }

}
```

定时任务2：

```
@Component
public class Scheduler2Task {

    private static final SimpleDateFormat dateFormat = new Simpl
eDateFormat("HH:mm:ss");

    @Scheduled(fixedRate = 6000)
    public void reportCurrentTime() {
        System.out.println("现在时间：" + dateFormat.format(new Da
te()));
    }

}
```

结果如下：

```
this is scheduler task runing  0
现在时间：09:44:17
this is scheduler task runing  1
现在时间：09:44:23
this is scheduler task runing  2
现在时间：09:44:29
this is scheduler task runing  3
现在时间：09:44:35
```

参数说明

`@Scheduled` 参数可以接受两种定时的设置，一种是我们常用的`cron="*/6 * * * * ?"`,一种是 `fixedRate = 6000`，两种都表示每隔六秒打印一下内容。

fixedRate 说明

- `@Scheduled(fixedRate = 6000)` ：上一次开始执行时间点之后6秒再执行
- `@Scheduled(fixedDelay = 6000)` ：上一次执行完毕时间点之后6秒再执行
- `@Scheduled(initialDelay=1000, fixedRate=6000)` ：第一次延迟1秒后执行，之后按fixedRate的规则每6秒执行一次

发送邮件应该是网站的必备功能之一，什么注册验证，忘记密码或者是给用户发送营销信息。最早期的时候我们会使用JavaMail相关api来写发送邮件的相关代码，后来spring推出了JavaMailSender更加简化了邮件发送的过程，在之后springboot对此进行了封装就有了现在的spring-boot-starter-mail,本章文章的介绍主要来自于此包。

简单使用

1、pom包配置

pom包里面添加spring-boot-starter-mail包引用

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

2、在application.properties中添加邮箱配置

```
spring.mail.host=smtp.qiye.163.com //邮箱服务器地址
spring.mail.username=xxx@oo.com //用户名
spring.mail.password=xxyyooo    //密码
spring.mail.default-encoding=UTF-8

mail.fromMail.addr=xxx@oo.com  //以谁来发送邮件
```

3、编写mailService,这里只提出实现类。

```
@Component
public class MailServiceImpl implements MailService{

    private final Logger logger = LoggerFactory.getLogger(this.g
etClass());

    @Autowired
    private JavaMailSender mailSender;

    @Value("${mail.fromMail.addr}")
    private String from;

    @Override
    public void sendSimpleMail(String to, String subject, String
 content) {
        SimpleMailMessage message = new SimpleMailMessage();
        message.setFrom(from);
        message.setTo(to);
        message.setSubject(subject);
        message.setText(content);

        try {
            mailSender.send(message);
            logger.info("简单邮件已经发送。");
        } catch (Exception e) {
            logger.error("发送简单邮件时发生异常！", e);
        }

    }
}
```

4、编写test类进行测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MailServiceTest {

    @Autowired
    private MailService MailService;

    @Test
    public void testSimpleMail() throws Exception {
        MailService.sendSimpleMail("ityouknow@126.com","test sim
ple mail"," hello this is simple mail");
    }
}
```

至此一个简单的文本发送就完成了。

加点料

但是在正常使用的过程中，我们通常在邮件中加入图片或者附件来丰富邮件的内容，下面讲介绍如何使用springboot来发送丰富的邮件。

发送html格式邮件

其它都不变在MailService添加sendHtmlMail方法.

```
public void sendHtmlMail(String to, String subject, String conte
nt) {
    MimeMessage message = mailSender.createMimeMessage();

    try {
        //true表示需要创建一个multipart message
        MimeMessageHelper helper = new MimeMessageHelper(message
, true);
        helper.setFrom(from);
        helper.setTo(to);
        helper.setSubject(subject);
        helper.setText(content, true);

        mailSender.send(message);
        logger.info("html邮件发送成功");
    } catch (MessagingException e) {
        logger.error("发送html邮件时发生异常！", e);
    }
}
```

在测试类中构建html内容，测试发送

```
@Test
public void testHtmlMail() throws Exception {
    String content="<html>\n" +
            "<body>\n" +
            "    <h3>hello world ! 这是一封Html邮件!</h3>\n" +
            "</body>\n" +
            "</html>";
    MailService.sendHtmlMail("ityouknow@126.com","test simple ma
il",content);
}
```

发送带附件的邮件

在MailService添加sendAttachmentsMail方法.

```
public void sendAttachmentsMail(String to, String subject, Strin
g content, String filePath){
    MimeMessage message = mailSender.createMimeMessage();

    try {
        MimeMessageHelper helper = new MimeMessageHelper(message
, true);
        helper.setFrom(from);
        helper.setTo(to);
        helper.setSubject(subject);
        helper.setText(content, true);

        FileSystemResource file = new FileSystemResource(new Fil
e(filePath));
        String fileName = filePath.substring(filePath.lastIndexO
f(File.separator));
        helper.addAttachment(fileName, file);

        mailSender.send(message);
        logger.info("带附件的邮件已经发送。");
    } catch (MessagingException e) {
        logger.error("发送带附件的邮件时发生异常！", e);
    }
}
```

> 添加多个附件可以使用多条 `helper.addAttachment(fileName, file)`

在测试类中添加测试方法

```
@Test
public void sendAttachmentsMail() {
    String filePath="e:\\tmp\\application.log";
    mailService.sendAttachmentsMail("ityouknow@126.com", "主题：
带附件的邮件", "有附件，请查收！", filePath);
}
```

发送带静态资源的邮件

邮件中的静态资源一般就是指图片，在MailService添加sendAttachmentsMail方法.

```
public void sendInlineResourceMail(String to, String subject, St
ring content, String rscPath, String rscId){
    MimeMessage message = mailSender.createMimeMessage();

    try {
        MimeMessageHelper helper = new MimeMessageHelper(message
, true);
        helper.setFrom(from);
        helper.setTo(to);
        helper.setSubject(subject);
        helper.setText(content, true);

        FileSystemResource res = new FileSystemResource(new File
(rscPath));
        helper.addInline(rscId, res);

        mailSender.send(message);
        logger.info("嵌入静态资源的邮件已经发送。");
    } catch (MessagingException e) {
        logger.error("发送嵌入静态资源的邮件时发生异常！", e);
    }
}
```

在测试类中添加测试方法

```
@Test
public void sendInlineResourceMail() {
    String rscId = "neo006";
    String content="<html><body>这是有图片的邮件：<img src=\'cid:"
+ rscId + "\' ></body></html>";
    String imgPath = "C:\\Users\\summer\\Pictures\\favicon.png";

    mailService.sendInlineResourceMail("ityouknow@126.com", "主题
：这是有图片的邮件", content, imgPath, rscId);
}
```

> 添加多个图片可以使用多条 `<img src='cid:" + rscId + "' >` 和 `helper.addInline(rscId, res)` 来实现

到此所有的邮件发送服务已经完成了。

邮件系统

上面发送邮件的基础服务就这些了，但是如果我们要做成一个邮件系统的话还需要考虑以下几个问题：

邮件模板

我们会经常收到这样的邮件：

```
尊敬的neo用户：

                恭喜您注册成为xxx网的用户，同时感谢您对xxx的关注与支持并欢迎您使用xx的产品与服务。
                ...
```

其中只有neo这个用户名在变化，其它邮件内容均不变，如果每次发送邮件都需要手动拼接的话会不够优雅，并且每次模板的修改都需要改动代码的话也很不方便，因此对于这类邮件需求，都建议做成邮件模板来处理。模板的本质很简单，就是在模板中替换变化的参数，转换为html字符串即可，这里以 `thymeleaf` 为例来演示。

1、pom中导入thymeleaf的包

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

2、在resorces/templates下创建emailTemplate.html

```
<!DOCTYPE html>
<html lang="zh" xmlns:th="http://www.thymeleaf.org">
    <head>
        <meta charset="UTF-8"/>
        <title>Title</title>
    </head>
    <body>
        您好,这是验证邮件,请点击下面的链接完成验证,<br/>
        <a href="#" th:href="@{ http://www.ityouknow.com/neo/{id}(id=${id}) }">激活账号</a>
    </body>
</html>
```

3、解析模板并发送

```
@Test
public void sendTemplateMail() {
    //创建邮件正文
    Context context = new Context();
    context.setVariable("id", "006");
    String emailContent = templateEngine.process("emailTemplate", context);

    mailService.sendHtmlMail("ityouknow@126.com","主题：这是模板邮件",emailContent);
}
```

发送失败

因为各种原因，总会有邮件发送失败的情况，比如：邮件发送过于频繁、网络异常等。在出现这种情况的时候，我们一般会考虑重新重试发送邮件，会分为以下几个步骤来实现：

- 1、接收到发送邮件请求，首先记录请求并且入库。
- 2、调用邮件发送接口发送邮件，并且将发送结果记录入库。
- 3、启动定时系统扫描时间段内，未发送成功并且重试次数小于3次的邮件，进行再次发送

异步发送

很多时候邮件发送并不是我们主业务必须关注的结果，比如通知类、提醒类的业务可以允许延时或者失败。这个时候可以采用异步的方式来发送邮件，加快主交易执行速度，在实际项目中可以采用MQ发送邮件相关参数，监听到消息队列之后启动发送邮件。

这篇文章我们来学习如何使用Spring Boot集成Apache Shiro。安全应该是互联网公司的一道生命线，几乎任何的公司都会涉及到这方面的需求。在Java领域一般有Spring Security、Apache Shiro等安全框架，但是由于Spring Security过于庞大和复杂，大多数公司会选择Apache Shiro来使用，这篇文章会先介绍一下Apache Shiro，在结合Spring Boot给出使用案例。

Apache Shiro

What is Apache Shiro?

Apache Shiro是一个功能强大、灵活的，开源的安全框架。它可以干净利落地处理身份验证、授权、企业会话管理和加密。

Apache Shiro的首要目标是易于使用和理解。安全通常很复杂，甚至让人感到很痛苦，但是Shiro却不是这样子的。一个好的安全框架应该屏蔽复杂性，向外暴露简单、直观的API，来简化开发人员实现应用程序安全所花费的时间和精力。

Shiro能做什么呢？

- 验证用户身份
- 用户访问权限控制，比如：1、判断用户是否分配了一定的安全角色。2、判断用户是否被授予完成某个操作的权限
- 在非 web 或 EJB 容器的环境下可以任意使用Session API
- 可以响应认证、访问控制，或者 Session 生命周期中发生的事件
- 可将一个或以上用户安全数据源数据组合成一个复合的用户 "view"(视图)
- 支持单点登录(SSO)功能
- 支持提供“Remember Me”服务，获取用户关联信息而无需登录

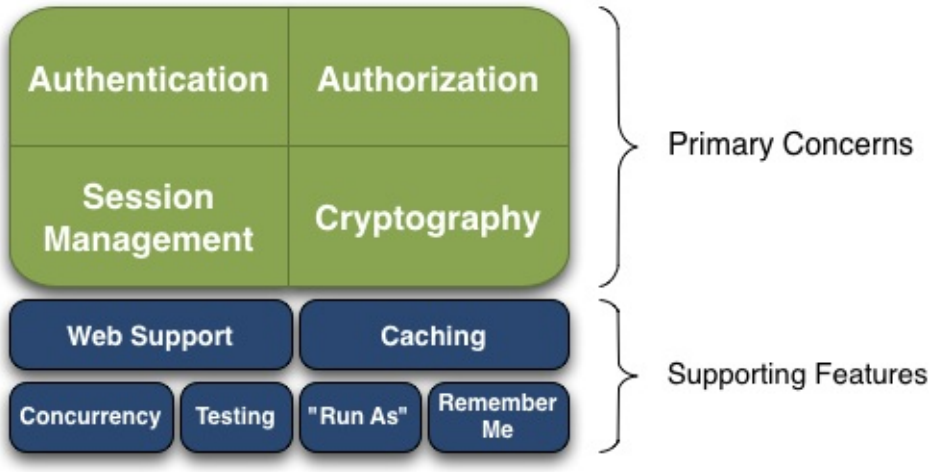
  …

等等——都集成到一个有凝聚力的易于使用的API。

Shiro 致力在所有应用环境下实现上述功能，小到命令行应用程序，大到企业应用中，而且不需要借助第三方框架、容器、应用服务器等。当然 Shiro 的目的是尽量的融入到这样的应用环境中去，但也可以在它们之外的任何环境下开箱即用。

Apache Shiro Features 特性

Apache Shiro是一个全面的、蕴含丰富功能的安全框架。下图为描述Shiro功能的框架图：



Authentication（认证）, Authorization（授权）, Session Management（会话管理）, Cryptography（加密）被 Shiro 框架的开发团队称之为应用安全的四大基石。那么就让我们来看看它们吧：

- **Authentication**（认证）：用户身份识别，通常被称为用户“登录”
- **Authorization**（授权）：访问控制。比如某个用户是否具有某个操作的使用权限。
- **Session Management**（会话管理）：特定于用户的会话管理,甚至在非web或 EJB 应用程序。
- **Cryptography**（加密）：在对数据源使用加密算法加密的同时，保证易于使用。

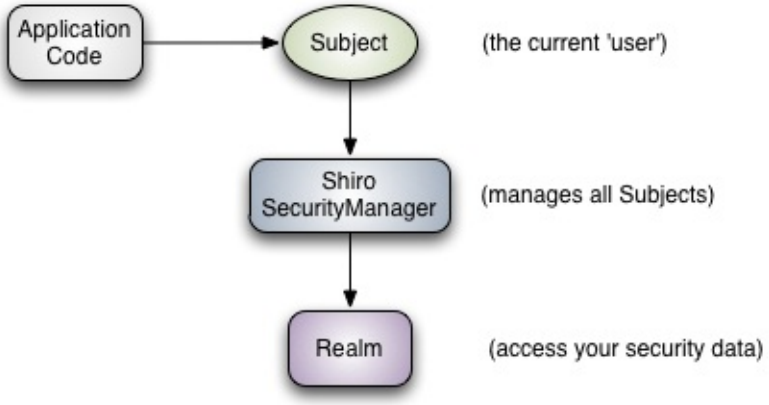
还有其他的功能来支持和加强这些不同应用环境下安全领域的关注点。特别是对以下的功能支持：

- Web支持：Shiro 提供的 web 支持 api ，可以很轻松的保护 web 应用程序的安全。
- 缓存：缓存是 Apache Shiro 保证安全操作快速、高效的重要手段。
- 并发：Apache Shiro 支持多线程应用程序的并发特性。
- 测试：支持单元测试和集成测试，确保代码和预想的一样安全。
- "Run As"：这个功能允许用户假设另一个用户的身份(在许可的前提下)。
- "Remember Me"：跨 session 记录用户的身份，只有在强制需要时才需要登录。

> 注意： Shiro不会去维护用户、维护权限，这些需要我们自己去设计/提供，然后通过相应的接口注入给Shiro

High-Level Overview 高级概述

在概念层，Shiro 架构包含三个主要的理念：Subject,SecurityManager和 Realm。下面的图展示了这些组件如何相互作用，我们将在下面依次对其进行描述。



- Subject：当前用户，Subject 可以是一个人，但也可以是第三方服务、守护进程帐户、时钟守护任务或者其它--当前和软件交互的任何事件。
- SecurityManager：管理所有Subject，SecurityManager 是 Shiro 架构的核心，配合内部安全组件共同组成安全伞。
- Realms：用于进行权限信息的验证，我们自己实现。Realm 本质上是一个特定的安全 DAO：它封装与数据源连接的细节，得到Shiro 所需的相关的数据。在配置 Shiro 的时候，你必须指定至少一个Realm 来实现认证（authentication）和/或授权（authorization）。

我们需要实现Realms的Authentication 和 Authorization。其中 Authentication 是用来验证用户身份，Authorization 是授权访问控制，用于对用户进行的操作授权，证明该用户是否允许进行当前操作，如访问某个链接，某个资源文件等。

快速上手

基础信息

pom包依赖

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

        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-thymeleaf</artifactId
>
        </dependency>
        <dependency>
            <groupId>net.sourceforge.nekohtml</groupId>
            <artifactId>nekohtml</artifactId>
            <version>1.9.22</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.apache.shiro</groupId>
            <artifactId>shiro-spring</artifactId>
            <version>1.4.0</version>
        </dependency>
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <scope>runtime</scope>
        </dependency>
    </dependencies>
```

重点是 shiro-spring包

配置文件

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

    jpa:
      database: mysql
      show-sql: true
      hibernate:
        ddl-auto: update
        naming:
          strategy: org.hibernate.cfg.DefaultComponentSafeNaming
Strategy
      properties:
         hibernate:
            dialect: org.hibernate.dialect.MySQL5Dialect

    thymeleaf:
       cache: false
       mode: LEGACYHTML5
```

thymeleaf的配置是为了去掉html的校验

页面

我们新建了六个页面用来测试：

- index.html ：首页
- login.html ：登录页
- userInfo.html ： 用户信息页面
- userInfoAdd.html ：添加用户页面
- userInfoDel.html ：删除用户页面
- 403.html ： 没有权限的页面

除过登录页面其它都很简单，大概如下：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Title</title>
</head>
<body>
<h1>index</h1>
</body>
</html>
```

RBAC

RBAC 是基于角色的访问控制（Role-Based Access Control ）在 RBAC 中，权限与角色相关联，用户通过成为适当角色的成员而得到这些角色的权限。这就极大地简化了权限的管理。这样管理都是层级相互依赖的，权限赋予给角色，而把角色又赋予用户，这样的权限设计很清楚，管理起来很方便。

采用jpa技术来自动生成基础表格，对应的entity如下：

用户信息

```
@Entity
public class UserInfo implements Serializable {
    @Id
    @GeneratedValue
    private Integer uid;
    @Column(unique =true)
    private String username;//帐号
    private String name;//名称（昵称或者真实姓名，不同系统不同定义）
    private String password; //密码;
    private String salt;//加密密码的盐
    private byte state;//用户状态,0:创建未认证（比如没有激活，没有输入验证码等等）--等待验证的用户 , 1:正常状态,2：用户被锁定.
    @ManyToMany(fetch= FetchType.EAGER)//立即从数据库中进行加载数据;
    @JoinTable(name = "SysUserRole", joinColumns = { @JoinColumn(name = "uid") }, inverseJoinColumns ={@JoinColumn(name = "roleId") })
    private List<SysRole> roleList;// 一个用户具有多个角色

    // 省略 get set 方法
 }
```

角色信息

```
@Entity
public class SysRole {
    @Id@GeneratedValue
    private Integer id; // 编号
    private String role; // 角色标识程序中判断使用,如"admin",这个是唯一的:
    private String description; // 角色描述,UI界面显示使用
    private Boolean available = Boolean.FALSE; // 是否可用,如果不可用将不会添加给用户

    //角色 -- 权限关系：多对多关系;
    @ManyToMany(fetch= FetchType.EAGER)
    @JoinTable(name="SysRolePermission",joinColumns={@JoinColumn(name="roleId")},inverseJoinColumns={@JoinColumn(name="permissionId")})
    private List<SysPermission> permissions;

    // 用户 - 角色关系定义;
    @ManyToMany
    @JoinTable(name="SysUserRole",joinColumns={@JoinColumn(name="roleId")},inverseJoinColumns={@JoinColumn(name="uid")})
    private List<UserInfo> userInfos;// 一个角色对应多个用户

    // 省略 get set 方法
 }
```

权限信息

```
@Entity
public class SysPermission implements Serializable {
    @Id@GeneratedValue
    private Integer id;//主键.
    private String name;//名称.
    @Column(columnDefinition="enum('menu','button')")
    private String resourceType;//资源类型，[menu|button]
    private String url;//资源路径.
    private String permission; //权限字符串,menu例子：role:*，butto
n例子：role:create,role:update,role:delete,role:view
    private Long parentId; //父编号
    private String parentIds; //父编号列表
    private Boolean available = Boolean.FALSE;
    @ManyToMany
    @JoinTable(name="SysRolePermission",joinColumns={@JoinColumn
(name="permissionId")},inverseJoinColumns={@JoinColumn(name="rol
eId")})
    private List<SysRole> roles;

    // 省略 get set 方法
 }
```

根据以上的代码会自动生成user_info（用户信息表）、sys_role（角色表）、sys_permission（权限表）、sys_user_role（用户角色表）、sys_role_permission（角色权限表）这五张表，为了方便测试我们给这五张表插入一些初始化数据：

```
INSERT INTO `user_info` (`uid`,`username`,`name`,`password`,`sal
t`,`state`) VALUES ('1', 'admin', '管理员', 'd3c59d25033dbf980d29
554025c23a75', '8d78869f470951332959580424d4bf4f', 0);
INSERT INTO `sys_permission` (`id`,`available`,`name`,`parent_id`
,`parent_ids`,`permission`,`resource_type`,`url`) VALUES (1,0,'
用户管理',0,'0/','userInfo:view','menu','userInfo/userList');
INSERT INTO `sys_permission` (`id`,`available`,`name`,`parent_id`
,`parent_ids`,`permission`,`resource_type`,`url`) VALUES (2,0,'
用户添加',1,'0/1','userInfo:add','button','userInfo/userAdd');
INSERT INTO `sys_permission` (`id`,`available`,`name`,`parent_id`
,`parent_ids`,`permission`,`resource_type`,`url`) VALUES (3,0,'
用户删除',1,'0/1','userInfo:del','button','userInfo/userDel');
INSERT INTO `sys_role` (`id`,`available`,`description`,`role`) V
ALUES (1,0,'管理员','admin');
INSERT INTO `sys_role` (`id`,`available`,`description`,`role`) V
ALUES (2,0,'VIP会员','vip');
INSERT INTO `sys_role` (`id`,`available`,`description`,`role`) V
ALUES (3,1,'test','test');
INSERT INTO `sys_role_permission` VALUES ('1', '1');
INSERT INTO `sys_role_permission` (`permission_id`,`role_id`) VA
LUES (1,1);
INSERT INTO `sys_role_permission` (`permission_id`,`role_id`) VA
LUES (2,1);
INSERT INTO `sys_role_permission` (`permission_id`,`role_id`) VA
LUES (3,2);
INSERT INTO `sys_user_role` (`role_id`,`uid`) VALUES (1,1);
```

Shiro 配置

首先要配置的是ShiroConfig类，Apache Shiro 核心通过 Filter 来实现，就好像SpringMvc 通过DispachServlet 来主控制一样。 既然是使用 Filter 一般也就能猜到，是通过URL规则来进行过滤和权限校验，所以我们需要定义一系列关于URL的规则和访问权限。

ShiroConfig

```
@Configuration
```

```
public class ShiroConfig {
    @Bean
    public ShiroFilterFactoryBean shirFilter(SecurityManager sec
urityManager) {
        System.out.println("ShiroConfiguration.shirFilter()");
        ShiroFilterFactoryBean shiroFilterFactoryBean = new Shir
oFilterFactoryBean();
        shiroFilterFactoryBean.setSecurityManager(securityManage
r);
        //拦截器.
        Map<String,String> filterChainDefinitionMap = new Linked
HashMap<String,String>();
        // 配置不会被拦截的链接 顺序判断
        filterChainDefinitionMap.put("/static/**", "anon");
        //配置退出 过滤器,其中的具体的退出代码Shiro已经替我们实现了
        filterChainDefinitionMap.put("/logout", "logout");
        //<!-- 过滤链定义，从上向下顺序执行，一般将/**放在最为下边 -->:
这是一个坑呢，一不小心代码就不好使了;
        //<!-- authc:所有url都必须认证通过才可以访问; anon:所有url都都
可以匿名访问-->
        filterChainDefinitionMap.put("/**", "authc");
        // 如果不设置默认会自动寻找Web工程根目录下的"/login.jsp"页面
        shiroFilterFactoryBean.setLoginUrl("/login");
        // 登录成功后要跳转的链接
        shiroFilterFactoryBean.setSuccessUrl("/index");

        //未授权界面;
        shiroFilterFactoryBean.setUnauthorizedUrl("/403");
        shiroFilterFactoryBean.setFilterChainDefinitionMap(filte
rChainDefinitionMap);
        return shiroFilterFactoryBean;
    }

    @Bean
    public MyShiroRealm myShiroRealm(){
        MyShiroRealm myShiroRealm = new MyShiroRealm();
        return myShiroRealm;
    }
```

```
    @Bean
    public SecurityManager securityManager(){
        DefaultWebSecurityManager securityManager =  new Default
WebSecurityManager();
        securityManager.setRealm(myShiroRealm());
        return securityManager;
    }
}
```

Filter Chain定义说明：

- 1、一个URL可以配置多个Filter，使用逗号分隔
- 2、当设置多个过滤器时，全部验证通过，才视为通过
- 3、部分过滤器可指定参数，如perms，roles

Shiro内置的FilterChain

Filter Name	Class
anon	org.apache.shiro.web.filter.authc.AnonymousFilter
authc	org.apache.shiro.web.filter.authc.FormAuthenticationFilter
authcBasic	org.apache.shiro.web.filter.authc.BasicHttpAuthenticationFilter
perms	org.apache.shiro.web.filter.authz.PermissionsAuthorizationFilter
port	org.apache.shiro.web.filter.authz.PortFilter
rest	org.apache.shiro.web.filter.authz.HttpMethodPermissionFilter
roles	org.apache.shiro.web.filter.authz.RolesAuthorizationFilter
ssl	org.apache.shiro.web.filter.authz.SslFilter
user	org.apache.shiro.web.filter.authc.UserFilter

- anon:所有url都都可以匿名访问
- authc: 需要认证才能进行访问
- user:配置记住我或认证通过可以访问

登录认证实现

在认证、授权内部实现机制中都有提到，最终处理都将交给Real进行处理。因为在Shiro中，最终是通过Realm来获取应用程序中的用户、角色及权限信息的。通常情况下，在Realm中会直接从我们的数据源中获取Shiro需要的验证信息。可以说，Realm是专用于安全框架的DAO. Shiro的认证过程最终会交由Realm执行，这时会调用Realm的 `getAuthenticationInfo(token)` 方法。

该方法主要执行以下操作:

- 1、检查提交的进行认证的令牌信息
- 2、根据令牌信息从数据源(通常为数据库)中获取用户信息
- 3、对用户信息进行匹配验证。
- 4、验证通过将返回一个封装了用户信息的 `AuthenticationInfo` 实例。
- 5、验证失败则抛出 `AuthenticationException` 异常信息。

而在我们的应用程序中要做的就是自定义一个Realm类，继承AuthorizingRealm抽象类，重载doGetAuthenticationInfo()，重写获取用户信息的方法。

doGetAuthenticationInfo的重写

```
@Override
protected AuthenticationInfo doGetAuthenticationInfo(AuthenticationToken token)
        throws AuthenticationException {
    System.out.println("MyShiroRealm.doGetAuthenticationInfo()");
    //获取用户的输入的账号.
    String username = (String)token.getPrincipal();
    System.out.println(token.getCredentials());
    //通过username从数据库中查找 User对象，如果找到，没找到.
    //实际项目中，这里可以根据实际情况做缓存，如果不做，Shiro自己也是有时间间隔机制，2分钟内不会重复执行该方法
    UserInfo userInfo = userInfoService.findByUsername(username);
    System.out.println("----->>userInfo="+userInfo);
    if(userInfo == null){
        return null;
    }
    SimpleAuthenticationInfo authenticationInfo = new SimpleAuthenticationInfo(
            userInfo, //用户名
            userInfo.getPassword(), //密码
            ByteSource.Util.bytes(userInfo.getCredentialsSalt()), //salt=username+salt
            getName()  //realm name
    );
    return authenticationInfo;
}
```

链接权限的实现

shiro的权限授权是通过继承 `AuthorizingRealm` 抽象类，重载 `doGetAuthorizationInfo();` 当访问到页面的时候，链接配置了相应的权限或者shiro标签才会执行此方法否则不会执行，所以如果只是简单的身份认证没有权限的控制的话，那么这个方法可以不进行实现，直接返回null即可。在这个方法中主要是使用类： `SimpleAuthorizationInfo` 进行角色的添加和权限的添加。

```
@Override
protected AuthorizationInfo doGetAuthorizationInfo(PrincipalCollection principals) {
    System.out.println("权限配置-->MyShiroRealm.doGetAuthorizationInfo()");
    SimpleAuthorizationInfo authorizationInfo = new SimpleAuthorizationInfo();
    UserInfo userInfo  = (UserInfo)principals.getPrimaryPrincipal();
    for(SysRole role:userInfo.getRoleList()){
        authorizationInfo.addRole(role.getRole());
        for(SysPermission p:role.getPermissions()){
            authorizationInfo.addStringPermission(p.getPermission());
        }
    }
    return authorizationInfo;
}
```

当然也可以添加set集合：roles是从数据库查询的当前用户的角色，stringPermissions是从数据库查询的当前用户对应的权限

```
authorizationInfo.setRoles(roles);
authorizationInfo.setStringPermissions(stringPermissions);
```

就是说如果在shiro配置文件中添加了 `filterChainDefinitionMap.put(“/add”, “perms[权限添加]”);` 就说明访问/add这个链接必须要有“权限添加”这个权限才可以访问，如果在shiro配置文件中添加了 `filterChainDefinitionMap.put(“/add”, “roles[100002]，perms[权限添加]”);` 就说明访问 `/add` 这个链接必须要有“权限添加”这个权限和具有“100002”这个角色才可以访问。

登录实现

登录过程其实只是处理异常的相关信息，具体的登录验证交给shiro来处理

```
@RequestMapping("/login")
public String login(HttpServletRequest request, Map<String, Object> map) throws Exception{
    System.out.println("HomeController.login()");
    // 登录失败从request中获取shiro处理的异常信息。
    // shiroLoginFailure:就是shiro异常类的全类名.
    String exception = (String) request.getAttribute("shiroLoginFailure");
    System.out.println("exception=" + exception);
    String msg = "";
    if (exception != null) {
        if (UnknownAccountException.class.getName().equals(exception)) {
            System.out.println("UnknownAccountException -- > 账号不存在：");
            msg = "UnknownAccountException -- > 账号不存在：";
        } else if (IncorrectCredentialsException.class.getName().equals(exception)) {
            System.out.println("IncorrectCredentialsException -- > 密码不正确：");
            msg = "IncorrectCredentialsException -- > 密码不正确：";
        } else if ("kaptchaValidateFailed".equals(exception)) {
            System.out.println("kaptchaValidateFailed -- > 验证码错误");
            msg = "kaptchaValidateFailed -- > 验证码错误";
        } else {
            msg = "else >> "+exception;
            System.out.println("else -- >" + exception);
        }
    }
    map.put("msg", msg);
    // 此方法不处理登录成功,由shiro进行处理
    return "/login";
}
```

其它dao层和service的代码就不贴出来了大家直接看代码。

测试

1、编写好后就可以启动程序，访问 `http://localhost:8080/userInfo/userList` 页面，由于没有登录就会跳转到 `http://localhost:8080/login` 页面。登录之后就会跳转到index页面，登录后，直接在浏览器中输入 `http://localhost:8080/userInfo/userList` 访问就会看到用户信息。上面这些操作时候触发 `MyShiroRealm.doGetAuthenticationInfo()` 这个方法，也就是登录认证的方法。

2、登录admin账户，访问： `http://127.0.0.1:8080/userInfo/userAdd` 显示 `用户添加界面` ，访问 `http://127.0.0.1:8080/userInfo/userDel` 显示 `403没有权限` 。上面这些操作时候触发 `MyShiroRealm.doGetAuthorizationInfo()` 这个方面，也就是权限校验的方法。

3、修改admin不同的权限进行测试

shiro很强大，这仅仅是完成了登录认证和权限管理这两个功能，更多内容以后有时间再做探讨。

这两天启动了一个新项目因为项目组成员一直都使用的是mybatis，虽然个人比较喜欢jpa这种极简的模式，但是为了项目保持统一性技术选型还是定了 mybatis。到网上找了一下关于spring boot和mybatis组合的相关资料，各种各样的形式都有，看的人心累，结合了mybatis的官方demo和文档终于找到了最简的两种模式，花了一天时间总结后分享出来。

orm框架的本质是简化编程中操作数据库的编码，发展到现在基本上就剩两家了，一个是宣称可以不用写一句SQL的hibernate，一个是可以灵活调试动态sql的mybatis,两者各有特点，在企业级系统开发中可以根据需求灵活使用。发现一个有趣的现象：传统企业大都喜欢使用hibernate,互联网行业通常使用mybatis。

hibernate特点就是所有的sql都用Java代码来生成，不用跳出程序去写（看）sql，有着编程的完整性，发展到最顶端就是spring data jpa这种模式了，基本上根据方法名就可以生成对应的sql了，有不太了解的可以看我的上篇文章springboot(五)：spring data jpa的使用-spring-data-jpa%E7%9A%84%E4%BD%BF%E7%94%A8.html)。

mybatis初期使用比较麻烦，需要各种配置文件、实体类、dao层映射关联、还有一大推其它配置。当然mybatis也发现了这种弊端，初期开发了generator可以根据表结果自动生产实体类、配置文件和dao层代码，可以减轻一部分开发量；后期也进行了大量的优化可以使用注解了，自动管理dao层和配置文件等，发展到最顶端就是今天要讲的这种模式了，mybatis-spring-boot-starter就是springboot+mybatis可以完全注解不用配置文件，也可以简单配置轻松上手。

> 现在想想spring boot 就是牛逼呀，任何东西只要关联到spring boot都是化繁为简。

mybatis-spring-boot-starter

官方说明：`MyBatis Spring-Boot-Starter will help you use MyBatis with Spring Boot`
其实就是myBatis看spring boot这么火热也开发出一套解决方案来凑凑热闹,但这一凑确实解决了很多问题，使用起来确实顺畅了许多。mybatis-spring-boot-starter主要有两种解决方案，一种是使用注解解决一切问题，一种是简化后的老传统。

当然任何模式都需要首先引入mybatis-spring-boot-starter的pom文件,现在最新版本是1.1.1（`刚好快到双11了 :)`）

```
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>1.1.1</version>
</dependency>
```

好了下来分别介绍两种开发模式

无配置文件注解版

就是一切使用注解搞定。

1 添加相关maven文件

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.mybatis.spring.boot</groupId>
        <artifactId>mybatis-spring-boot-starter</artifactId>
        <version>1.1.1</version>
    </dependency>
     <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
    </dependency>
     <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <optional>true</optional>
    </dependency>
</dependencies>
```

完整的pom包这里就不贴了，大家直接看源码

2、`application.properties` 添加相关配置

```
mybatis.type-aliases-package=com.neo.entity

spring.datasource.driverClassName = com.mysql.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/test1?useUni
code=true&characterEncoding=utf-8
spring.datasource.username = root
spring.datasource.password = root
```

springboot会自动加载spring.datasource.*相关配置，数据源就会自动注入到sqlSessionFactory中，sqlSessionFactory会自动注入到Mapper中，对了你一切都不用管了，直接拿起来使用就行了。

在启动类中添加对mapper包扫描 `@MapperScan`

```
@SpringBootApplication
@MapperScan("com.neo.mapper")
public class Application {

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

或者直接在Mapper类上面添加注解 `@Mapper` ,建议使用上面那种，不然每个mapper加个注解也挺麻烦的

3、开发Mapper

第三步是最关键的一块，sql生产都在这里

```
public interface UserMapper {

    @Select("SELECT * FROM users")
    @Results({
        @Result(property = "userSex",  column = "user_sex", javaType = UserSexEnum.class),
        @Result(property = "nickName", column = "nick_name")
    })
    List<UserEntity> getAll();

    @Select("SELECT * FROM users WHERE id = #{id}")
    @Results({
        @Result(property = "userSex",  column = "user_sex", javaType = UserSexEnum.class),
        @Result(property = "nickName", column = "nick_name")
    })
    UserEntity getOne(Long id);

    @Insert("INSERT INTO users(userName,passWord,user_sex) VALUES(#{userName}, #{passWord}, #{userSex})")
    void insert(UserEntity user);

    @Update("UPDATE users SET userName=#{userName},nick_name=#{nickName} WHERE id =#{id}")
    void update(UserEntity user);

    @Delete("DELETE FROM users WHERE id =#{id}")
    void delete(Long id);

}
```

为了更接近生产我特地将**user_sex**、**nick_name**两个属性在数据库加了下划线和实体类属性名不一致，另外**user_sex**使用了枚举

> - @Select 是查询类的注解，所有的查询均使用这个
> - @Result 修饰返回的结果集，关联实体类属性和数据库字段一一对应，如果实体类属性和数据库属性名保持一致，就不需要这个属性来修饰。
> - @Insert 插入数据库使用，直接传入实体类会自动解析属性到对应的值
> - @Update 负责修改，也可以直接传入对象
> - @delete 负责删除

了解更多属性参考这里

> 注意，使用#符号和$符号的不同：

```
// This example creates a prepared statement, something like sel
ect * from teacher where name = ?;
@Select("Select * from teacher where name = #{name}")
Teacher selectTeachForGivenName(@Param("name") String name);

// This example creates n inlined statement, something like sele
ct * from teacher where name = 'someName';
@Select("Select * from teacher where name = '${name}'")
Teacher selectTeachForGivenName(@Param("name") String name);
```

4、使用

上面三步就基本完成了相关dao层开发，使用的时候当作普通的类注入进入就可以了

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserMapperTest {

    @Autowired
    private UserMapper UserMapper;

    @Test
    public void testInsert() throws Exception {
        UserMapper.insert(new UserEntity("aa", "a123456", UserSe
xEnum.MAN));
        UserMapper.insert(new UserEntity("bb", "b123456", UserSe
xEnum.WOMAN));
        UserMapper.insert(new UserEntity("cc", "b123456", UserSe
xEnum.WOMAN));

        Assert.assertEquals(3, UserMapper.getAll().size());
    }

    @Test
    public void testQuery() throws Exception {
        List<UserEntity> users = UserMapper.getAll();
        System.out.println(users.toString());
    }

    @Test
    public void testUpdate() throws Exception {
        UserEntity user = UserMapper.getOne(3l);
        System.out.println(user.toString());
        user.setNickName("neo");
        UserMapper.update(user);
        Assert.assertTrue(("neo".equals(UserMapper.getOne(3l).ge
tNickName())));
    }
}
```

源码中controler层有完整的增删改查，这里就不贴了

极简xml版本

极简xml版本保持映射文件的老传统，优化主要体现在不需要实现dao的是实现层，系统会自动根据方法名在映射文件中找对应的sql.

1、配置

pom文件和上个版本一样，只是 `application.properties` 新增以下配置

```
mybatis.config-locations=classpath:mybatis/mybatis-config.xml
mybatis.mapper-locations=classpath:mybatis/mapper/*.xml
```

指定了mybatis基础配置文件和实体类映射文件的地址

mybatis-config.xml 配置

```
<configuration>
    <typeAliases>
        <typeAlias alias="Integer" type="java.lang.Integer" />
        <typeAlias alias="Long" type="java.lang.Long" />
        <typeAlias alias="HashMap" type="java.util.HashMap" />
        <typeAlias alias="LinkedHashMap" type="java.util.LinkedH
ashMap" />
        <typeAlias alias="ArrayList" type="java.util.ArrayList"
/>
        <typeAlias alias="LinkedList" type="java.util.LinkedList"
 />
    </typeAliases>
</configuration>
```

这里也可以添加一些mybatis基础的配置

2、添加User的映射文件

```
<mapper namespace="com.neo.mapper.UserMapper" >
    <resultMap id="BaseResultMap" type="com.neo.entity.UserEntit
```

```
y" >
        <id column="id" property="id" jdbcType="BIGINT" />
        <result column="userName" property="userName" jdbcType="
VARCHAR" />
        <result column="passWord" property="passWord" jdbcType="
VARCHAR" />
        <result column="user_sex" property="userSex" javaType="c
om.neo.enums.UserSexEnum"/>
        <result column="nick_name" property="nickName" jdbcType=
"VARCHAR" />
    </resultMap>

    <sql id="Base_Column_List" >
        id, userName, passWord, user_sex, nick_name
    </sql>

    <select id="getAll" resultMap="BaseResultMap"  >
       SELECT
       <include refid="Base_Column_List" />
       FROM users
    </select>

    <select id="getOne" parameterType="java.lang.Long" resultMap=
"BaseResultMap" >
        SELECT
       <include refid="Base_Column_List" />
       FROM users
       WHERE id = #{id}
    </select>

    <insert id="insert" parameterType="com.neo.entity.UserEntity"
 >
       INSERT INTO
              users
              (userName,passWord,user_sex)
           VALUES
              (#{userName}, #{passWord}, #{userSex})
    </insert>

    <update id="update" parameterType="com.neo.entity.UserEntity"
```

```
 >
        UPDATE
                users
        SET
            <if test="userName != null">userName = #{userName},</
if>
            <if test="passWord != null">passWord = #{passWord},</
if>
            nick_name = #{nickName}
        WHERE
                id = #{id}
    </update>

    <delete id="delete" parameterType="java.lang.Long" >
        DELETE FROM
                 users
        WHERE
                 id =#{id}
    </delete>
</mapper>
```

其实就是把上个版本中mapper的sql搬到了这里的xml中了

3、编写Dao层的代码

```
public interface UserMapper {

    List<UserEntity> getAll();

    UserEntity getOne(Long id);

    void insert(UserEntity user);

    void update(UserEntity user);

    void delete(Long id);

}
```

对比上一步这里全部只剩了接口方法

4、使用

使用和上个版本没有任何区别，大家就看代码吧

如何选择

两种模式各有特点，注解版适合简单快速的模式，其实像现在流行的这种微服务模式，一个微服务就会对应一个自己的数据库，多表连接查询的需求会大大的降低，会越来越适合这种模式。

老传统模式比适合大型项目，可以灵活的动态生成SQL，方便调整SQL，也有痛痛快快，洋洋洒洒的写SQL的感觉。

示例代码-github

示例代码-码云

说起多数据源，一般都来解决那些问题呢，主从模式或者业务比较复杂需要连接不同的分库来支持业务。我们项目是后者的模式，网上找了很多，大都是根据jpa来做多数据源解决方案，要不就是老的spring多数据源解决方案，还有的是利用aop动态切换，感觉有点小复杂，其实我只是想找一个简单的多数据支持而已，折腾了两个小时整理出来，供大家参考。

> 废话不多说直接上代码吧

配置文件

pom包就不贴了比较简单该依赖的就依赖，主要是数据库这边的配置：

```
mybatis.config-locations=classpath:mybatis/mybatis-config.xml

spring.datasource.test1.driverClassName = com.mysql.jdbc.Driver
spring.datasource.test1.url = jdbc:mysql://localhost:3306/test1?
useUnicode=true&characterEncoding=utf-8
spring.datasource.test1.username = root
spring.datasource.test1.password = root

spring.datasource.test2.driverClassName = com.mysql.jdbc.Driver
spring.datasource.test2.url = jdbc:mysql://localhost:3306/test2?
useUnicode=true&characterEncoding=utf-8
spring.datasource.test2.username = root
spring.datasource.test2.password = root
```

一个test1库和一个test2库，其中test1位主库，在使用的过程中必须指定主库，不然会报错。

数据源配置

```
@Configuration
@MapperScan(basePackages = "com.neo.mapper.test1", sqlSessionTem
plateRef  = "test1SqlSessionTemplate")
public class DataSource1Config {

    @Bean(name = "test1DataSource")
```

```
    @ConfigurationProperties(prefix = "spring.datasource.test1")
    @Primary
    public DataSource testDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "test1SqlSessionFactory")
    @Primary
    public SqlSessionFactory testSqlSessionFactory(@Qualifier("t
est1DataSource") DataSource dataSource) throws Exception {
        SqlSessionFactoryBean bean = new SqlSessionFactoryBean()
;
        bean.setDataSource(dataSource);
        bean.setMapperLocations(new PathMatchingResourcePatternR
esolver().getResources("classpath:mybatis/mapper/test1/*.xml"));
        return bean.getObject();
    }

    @Bean(name = "test1TransactionManager")
    @Primary
    public DataSourceTransactionManager testTransactionManager(@
Qualifier("test1DataSource") DataSource dataSource) {
        return new DataSourceTransactionManager(dataSource);
    }

    @Bean(name = "test1SqlSessionTemplate")
    @Primary
    public SqlSessionTemplate testSqlSessionTemplate(@Qualifier(
"test1SqlSessionFactory") SqlSessionFactory sqlSessionFactory) t
hrows Exception {
        return new SqlSessionTemplate(sqlSessionFactory);
    }

}
```

最关键的地方就是这块了，一层一层注入,首先创建DataSource，然后创建SqlSessionFactory再创建事务，最后包装到SqlSessionTemplate中。其中需要指定分库的mapper文件地址，以及分库dao层代码

```
@MapperScan(basePackages = "com.neo.mapper.test1", sqlSessionTem
plateRef  = "test1SqlSessionTemplate")
```

这块的注解就是指明了扫描dao层，并且给dao层注入指定的SqlSessionTemplate。所有 `@Bean` 都需要按照命名指定正确。

dao层和xml层

dao层和xml需要按照库来分在不同的目录，比如：test1库dao层在com.neo.mapper.test1包下，test2库在com.neo.mapper.test1

```
public interface User1Mapper {

    List<UserEntity> getAll();

    UserEntity getOne(Long id);

    void insert(UserEntity user);

    void update(UserEntity user);

    void delete(Long id);

}
```

xml层

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN" "ht
tp://mybatis.org/dtd/mybatis-3-mapper.dtd" >
<mapper namespace="com.neo.mapper.test1.User1Mapper" >
    <resultMap id="BaseResultMap" type="com.neo.entity.UserEntit
y" >
        <id column="id" property="id" jdbcType="BIGINT" />
        <result column="userName" property="userName" jdbcType="
VARCHAR" />
        <result column="passWord" property="passWord" jdbcType="
```

```
VARCHAR" />
        <result column="user_sex" property="userSex" javaType="c
om.neo.enums.UserSexEnum"/>
        <result column="nick_name" property="nickName" jdbcType=
"VARCHAR" />
    </resultMap>

    <sql id="Base_Column_List" >
        id, userName, passWord, user_sex, nick_name
    </sql>

    <select id="getAll" resultMap="BaseResultMap"  >
       SELECT
       <include refid="Base_Column_List" />
       FROM users
    </select>

    <select id="getOne" parameterType="java.lang.Long" resultMap=
"BaseResultMap" >
        SELECT
       <include refid="Base_Column_List" />
       FROM users
       WHERE id = #{id}
    </select>

    <insert id="insert" parameterType="com.neo.entity.UserEntity"
 >
       INSERT INTO
              users
              (userName,passWord,user_sex)
           VALUES
              (#{userName}, #{passWord}, #{userSex})
    </insert>

    <update id="update" parameterType="com.neo.entity.UserEntity"
 >
       UPDATE
              users
       SET
           <if test="userName != null">userName = #{userName},</
```

```
if>
            <if test="passWord != null">passWord = #{passWord},</
if>
            nick_name = #{nickName}
        WHERE
                id = #{id}
    </update>

    <delete id="delete" parameterType="java.lang.Long" >
        DELETE FROM
                users
        WHERE
                id =#{id}
    </delete>

</mapper>
```

测试

测试可以使用SpringBootTest,也可以放到Controller中，这里只贴Controller层的使用

```
@RestController
public class UserController {

    @Autowired
    private User1Mapper user1Mapper;

    @Autowired
    private User2Mapper user2Mapper;

    @RequestMapping("/getUsers")
    public List<UserEntity> getUsers() {
        List<UserEntity> users=user1Mapper.getAll();
        return users;
    }

    @RequestMapping("/getUser")
    public UserEntity getUser(Long id) {
        UserEntity user=user2Mapper.getOne(id);
        return user;
    }

    @RequestMapping("/add")
    public void save(UserEntity user) {
        user2Mapper.insert(user);
    }

    @RequestMapping(value="update")
    public void update(UserEntity user) {
        user2Mapper.update(user);
    }

    @RequestMapping(value="/delete/{id}")
    public void delete(@PathVariable("id") Long id) {
        user1Mapper.delete(id);
    }

}
```

在上篇文章springboot(二)：web综合开发-web%E7%BB%BC%E5%90%88%E5%BC%80%E5%8F%91.html)中简单介绍了一下spring data jpa的基础性使用，这篇文章将更加全面的介绍spring data jpa 常见用法以及注意事项

使用spring data jpa 开发时，发现国内对spring boot jpa全面介绍的文章比较少案例也比较零碎，因此写文章总结一下。本人也正在翻译Spring Data JPA 参考指南,有兴趣的同学欢迎联系我，一起加入翻译中！

spring data jpa介绍

首先了解JPA是什么？

JPA(Java Persistence API)是Sun官方提出的Java持久化规范。它为Java开发人员提供了一种对象/关联映射工具来管理Java应用中的关系数据。他的出现主要是为了简化现有的持久化开发工作和整合ORM技术，结束现在Hibernate，TopLink，JDO等ORM框架各自为营的局面。值得注意的是，JPA是在充分吸收了现有Hibernate，TopLink，JDO等ORM框架的基础上发展而来的，具有易于使用，伸缩性强等优点。从目前的开发社区的反应上看，JPA受到了极大的支持和赞扬，其中就包括了Spring与EJB3.0的开发团队。

> 注意:JPA是一套规范，不是一套产品，那么像Hibernate,TopLink,JDO他们是一套产品，如果说这些产品实现了这个JPA规范，那么我们就可以叫他们为JPA的实现产品。

spring data jpa

Spring Data JPA 是 Spring 基于 ORM 框架、JPA 规范的基础上封装的一套JPA应用框架，可使开发者用极简的代码即可实现对数据的访问和操作。它提供了包括增删改查等在内的常用功能，且易于扩展！学习并使用 Spring Data JPA 可以极大提高开发效率！

> spring data jpa让我们解脱了DAO层的操作，基本上所有CRUD都可以依赖于它来实现

基本查询

基本查询也分为两种，一种是spring data默认已经实现，一种是根据查询的方法来自动解析成SQL。

预先生成方法

spring data jpa 默认预先生成了一些基本的CURD的方法，例如：增、删、改等等

1 继承JpaRepository

```
public interface UserRepository extends JpaRepository<User, Long
> {
}
```

2 使用默认方法

```
@Test
public void testBaseQuery() throws Exception {
    User user=new User();
    userRepository.findAll();
    userRepository.findOne(1l);
    userRepository.save(user);
    userRepository.delete(user);
    userRepository.count();
    userRepository.exists(1l);
    // ...
}
```

就不解释了根据方法名就看出意思来

自定义简单查询

自定义的简单查询就是根据方法名来自动生成SQL，主要的语法是 `findXXBy` , `readAXXBy` , `queryXXBy` , `countXXBy` , `getXXBy` 后面跟属性名称：

```
User findByUserName(String userName);
```

也使用一些加一些关键字 `And` 、 `Or`

```
User findByUserNameOrEmail(String username, String email);
```

修改、删除、统计也是类似语法

```
Long deleteById(Long id);

Long countByUserName(String userName)
```

基本上SQL体系中的关键词都可以使用，例如： `LIKE` 、 `IgnoreCase` 、 `OrderBy` 。

```
List<User> findByEmailLike(String email);

User findByUserNameIgnoreCase(String userName);

List<User> findByUserNameOrderByEmailDesc(String email);
```

具体的关键字，使用方法和生产成**SQL**如下表所示

Keyword	Sample	JPQL snip
And	findByLastnameAndFirstname	… where x.lastname = and x.firstnan ?2
Or	findByLastnameOrFirstname	… where x.lastname = x.firstname =
Is,Equals	findByFirstnameIs,findByFirstnameEquals	… where x.firstname =
Between	findByStartDateBetween	… where x.startDate between ?1 a 2

LessThan	findByAgeLessThan	… where x.ag 1
LessThanEqual	findByAgeLessThanEqual	… where x.ag ?1
GreaterThan	findByAgeGreaterThan	… where x.ag 1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.ag ?1
After	findByStartDateAfter	… where x.startDate >
Before	findByStartDateBefore	… where x.startDate <
IsNull	findByAgeIsNull	… where x.ag null
IsNotNull,NotNull	findByAge(Is)NotNull	… where x.ag null
Like	findByFirstnameLike	… where x.firstname lik
NotLike	findByFirstnameNotLike	… where x.firstname n ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname lik (parameter b with appende
EndingWith	findByFirstnameEndingWith	… where x.firstname lik (parameter b with prepend
Containing	findByFirstnameContaining	… where x.firstname lik (parameter b wrapped in %
OrderBy	findByAgeOrderByLastnameDesc	… where x.ag 1 order by x.lastname de
Not	findByLastnameNot	… where x.lastname <:

In	findByAgeIn(Collection ages)	… where x.ag ?1
NotIn	findByAgeNotIn(Collection age)	… where x.ag in ?1
TRUE	findByActiveTrue()	… where x.ac = true
FALSE	findByActiveFalse()	… where x.ac = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firs = UPPER(?1

复杂查询

在实际的开发中我们需要用到分页、删选、连表等查询的时候就需要特殊的方法或者自定义SQL

分页查询

分页查询在实际使用中非常普遍了，spring data jpa已经帮我们实现了分页的功能，在查询的方法中，需要传入参数 `Pageable` ,当查询中有多个参数的时候 `Pageable` 建议做为最后一个参数传入

```
Page<User> findALL(Pageable pageable);

Page<User> findByUserName(String userName,Pageable pageable);
```

`Pageable` 是spring封装的分页实现类，使用的时候需要传入页数、每页条数和排序规则

```
@Test
public void testPageQuery() throws Exception {
    int page=1,size=10;
    Sort sort = new Sort(Direction.DESC, "id");
    Pageable pageable = new PageRequest(page, size, sort);
    userRepository.findALL(pageable);
    userRepository.findByUserName("testName", pageable);
}
```

限制查询

有时候我们只需要查询前N个元素，或者支取前一个实体。

```
ser findFirstByOrderByLastnameAsc();

User findTopByOrderByAgeDesc();

Page<User> queryFirst10ByLastname(String lastname, Pageable page
able);

List<User> findFirst10ByLastname(String lastname, Sort sort);

List<User> findTop10ByLastname(String lastname, Pageable pageabl
e);
```

自定义SQL查询

其实Spring data 觉大部分的SQL都可以根据方法名定义的方式来实现，但是由于某些原因我们想使用自定义的SQL来查询，spring data也是完美支持的；在SQL的查询方法上面使用 `@Query` 注解，如涉及到删除和修改在需要加上 `@Modifying` .也可以根据需要添加 `@Transactional` 对事物的支持，查询超时的设置等

```
@Modifying
@Query("update User u set u.userName = ?1 where u.id = ?2")
int modifyByIdAndUserId(String  userName, Long id);

@Transactional
@Modifying
@Query("delete from User where id = ?1")
void deleteByUserId(Long id);

@Transactional(timeout = 10)
@Query("select u from User u where u.emailAddress = ?1")
    User findByEmailAddress(String emailAddress);
```

多表查询

多表查询在spring data jpa中有两种实现方式，第一种是利用hibernate的级联查询来实现，第二种是创建一个结果集的接口来接收连表查询后的结果，这里主要第二种方式。

首先需要定义一个结果集的接口类。

```
public interface HotelSummary {

    City getCity();

    String getName();

    Double getAverageRating();

    default Integer getAverageRatingRounded() {
        return getAverageRating() == null ? null : (int) Math.ro
und(getAverageRating());
    }

}
```

查询的方法返回类型设置为新创建的接口

```
@Query("select h.city as city, h.name as name, avg(r.rating) as
averageRating "
        - "from Hotel h left outer join h.reviews r where h.city
 = ?1 group by h")
Page<HotelSummary> findByCity(City city, Pageable pageable);

@Query("select h.name as name, avg(r.rating) as averageRating "
        - "from Hotel h left outer join h.reviews r  group by h"
)
Page<HotelSummary> findByCity(Pageable pageable);
```

使用

```
Page<HotelSummary> hotels = this.hotelRepository.findByCity(new
PageRequest(0, 10, Direction.ASC, "name"));
for(HotelSummary summay:hotels){
        System.out.println("Name" +summay.getName());
    }
```

> 在运行中Spring会给接口（HotelSummary）自动生产一个代理类来接收返回的结果，代码汇总使用 `getXX` 的形式来获取

多数据源的支持

同源数据库的多源支持

日常项目中因为使用的分布式开发模式，不同的服务有不同的数据源，常常需要在一个项目中使用多个数据源，因此需要配置sping data jpa对多数据源的使用，一般分一下为三步：

- 1 配置多数据源
- 2 不同源的实体类放入不同包路径
- 3 声明不同的包路径下使用不同的数据源、事务支持

异构数据库多源支持

比如我们的项目中，即需要对mysql的支持，也需要对mongodb的查询等。

实体类声明 `@Entity` 关系型数据库支持类型、声明 `@Document` 为mongodb支持类型，不同的数据源使用不同的实体就可以了

```
interface PersonRepository extends Repository<Person, Long> {
 …
}

@Entity
public class Person {
  …
}

interface UserRepository extends Repository<User, Long> {
 …
}

@Document
public class User {
  …
}
```

但是，如果User用户既使用mysql也使用mongodb呢，也可以做混合使用

```
interface JpaPersonRepository extends Repository<Person, Long> {
 …
}

interface MongoDBPersonRepository extends Repository<Person, Long
> {
 …
}

@Entity
@Document
public class Person {
  …
}
```

也可以通过对不同的包路径进行声明，比如A包路径下使用mysql,B包路径下使用mongoDB

```
@EnableJpaRepositories(basePackages = "com.neo.repositories.jpa"
)
@EnableMongoRepositories(basePackages = "com.neo.repositories.mo
ngo")
interface Configuration { }
```

其它

使用枚举

使用枚举的时候，我们希望数据库中存储的是枚举对应的String类型，而不是枚举的索引值，需要在属性上面添加 `@Enumerated(EnumType.STRING)` 注解

```
@Enumerated(EnumType.STRING)
@Column(nullable = true)
private UserType type;
```

不需要和数据库映射的属性

正常情况下我们在实体类上加入注解 `@Entity` ，就会让实体类和表相关连如果其中某个属性我们不需要和数据库来关联只是在展示的时候做计算，只需要加上 `@Transient` 属性既可。

```
@Transient
private String  userName;
```

源码案例

这里有一个开源项目几乎使用了这里介绍的所有标签和布局，大家可以参考：

示例代码-github

示例代码-码云

参考

Spring Data JPA - Reference Documentation

Spring Data JPA——参考文档 中文版

mongodb是最早热门非关系数据库的之一，使用也比较普遍，一般会用做离线数据分析来使用，放到内网的居多。由于很多公司使用了云服务，服务器默认都开放了外网地址，导致前一阵子大批 MongoDB 因配置漏洞被攻击，数据被删，引起了人们的注意，感兴趣的可以看看这篇文章：场屠戮MongoDB的盛宴反思：超33000个数据库遭遇入侵勒索，同时也说明了很多公司生产中大量使用mongodb。

mongodb简介

MongoDB（来自于英文单词“Humongous”，中文含义为“庞大”）是可以应用于各种规模的企业、各个行业以及各类应用程序的开源数据库。基于分布式文件存储的数据库。由C++语言编写。旨在为WEB应用提供可扩展的高性能数据存储解决方案。MongoDB是一个高性能，开源，无模式的文档型数据库，是当前NoSql数据库中比较热门的一种。

MongoDB是一个介于关系数据库和非关系数据库之间的产品，是非关系数据库当中功能最丰富，最像关系数据库的。他支持的数据结构非常松散，是类似json的bjson格式，因此可以存储比较复杂的数据类型。Mongo最大的特点是他支持的查询语言非常强大，其语法有点类似于面向对象的查询语言，几乎可以实现类似关系数据库单表查询的绝大部分功能，而且还支持对数据建立索引。

传统的关系数据库一般由数据库（database）、表（table）、记录（record）三个层次概念组成，MongoDB是由数据库（database）、集合（collection）、文档对象（document）三个层次组成。MongoDB对于关系型数据库里的表，但是集合中没有列、行和关系概念，这体现了模式自由的特点。

MongoDB中的一条记录就是一个文档，是一个数据结构，由字段和值对组成。MongoDB文档与JSON对象类似。字段的值有可能包括其它文档、数组以及文档数组。MongoDB支持OS X、Linux及Windows等操作系统，并提供了Python，PHP，Ruby，Java及C++语言的驱动程序，社区中也提供了对Erlang及.NET等平台的驱动程序。

MongoDB的适合对大量或者无固定格式的数据进行存储，比如：日志、缓存等。对事物支持较弱，不适用复杂的多文档（多表）的级联查询。文中演示mongodb版本为3.4。

mongodb的增删改查

Spring Boot对各种流行的数据源都进行了封装，当然也包括了mongodb,下面给大家介绍如何在spring boot中使用mongodb：

1、pom包配置

pom包里面添加spring-boot-starter-data-mongodb包引用

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

    </dependency>
</dependencies>
```

2、在application.properties中添加配置

```
spring.data.mongodb.uri=mongodb://name:pass@localhost:27017/test
```

多个IP集群可以采用以下配置：

```
spring.data.mongodb.uri=mongodb://user:pwd@ip1:port1,ip2:port2/database
```

2、创建数据实体

```
public class UserEntity implements Serializable {
        private static final long serialVersionUID = -3258839839
160856613L;
        private Long id;
        private String userName;
        private String passWord;

      //getter、setter省略
}
```

3、创建实体dao的增删改查操作

dao层实现了UserEntity对象的增删改查

```
@Component
public class UserDaoImpl implements UserDao {

    @Autowired
    private MongoTemplate mongoTemplate;

    /**
     * 创建对象
     * @param user
     */
    @Override
    public void saveUser(UserEntity user) {
        mongoTemplate.save(user);
    }

    /**
     * 根据用户名查询对象
     * @param userName
     * @return
     */
    @Override
    public UserEntity findUserByUserName(String userName) {
        Query query=new Query(Criteria.where("userName").is(user
Name));
```

```
        UserEntity user =  mongoTemplate.findOne(query , UserEnt
ity.class);
        return user;
    }

    /**
     * 更新对象
     * @param user
     */
    @Override
    public void updateUser(UserEntity user) {
        Query query=new Query(Criteria.where("id").is(user.getId
()));
        Update update= new Update().set("userName", user.getUser
Name()).set("passWord", user.getPassWord());
        //更新查询返回结果集的第一条
        mongoTemplate.updateFirst(query,update,UserEntity.class)
;
        //更新查询返回结果集的所有
        // mongoTemplate.updateMulti(query,update,UserEntity.cla
ss);
    }

    /**
     * 删除对象
     * @param id
     */
    @Override
    public void deleteUserById(Long id) {
        Query query=new Query(Criteria.where("id").is(id));
        mongoTemplate.remove(query,UserEntity.class);
    }
}
```

4、开发对应的测试方法

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class UserDaoTest {

    @Autowired
    private UserDao userDao;

    @Test
    public void testSaveUser() throws Exception {
        UserEntity user=new UserEntity();
        user.setId(2l);
        user.setUserName("小明");
        user.setPassWord("fffooo123");
        userDao.saveUser(user);
    }

    @Test
    public void findUserByUserName(){
       UserEntity user= userDao.findUserByUserName("小明");
       System.out.println("user is "+user);
    }

    @Test
    public void updateUser(){
        UserEntity user=new UserEntity();
        user.setId(2l);
        user.setUserName("天空");
        user.setPassWord("fffxxxx");
        userDao.updateUser(user);
    }

    @Test
    public void deleteUserById(){
        userDao.deleteUserById(1l);
    }

}
```

5、查看验证结果

可以使用工具mongoVUE工具来连接后直接图形化展示查看，也可以登录服务器用命令来查看

1.登录mongos

```
bin/mongo -host localhost -port 20000
```

2、切换到test库

```
use test
```

3、查询userEntity集合数据

```
db.userEntity.find()
```

根据3查询的结果来观察测试用例的执行是否正确。

到此springboot对应mongodb的增删改查功能已经全部实现。

多数据源mongodb的使用

在多mongodb数据源的情况下，我们换种更优雅的方式来实现

1、pom包配置

添加lombok和spring-boot-autoconfigure包引用

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-autoconfigure</artifactId>
    <version>RELEASE</version>
</dependency>
```

- Lombok - 是一个可以通过简单的注解形式来帮助我们简化消除一些必须有但显得很臃肿的Java代码的工具，通过使用对应的注解，可以在编译源码的时候生成对应的方法。简单试了以下这个工具还挺好玩的，加上注解我们就不用手动写 getter\setter、构建方式类似的代码了。
- spring-boot-autoconfigure - 就是spring boot的自动化配置

2、配置文件使用YAML的形式添加两条数据源，如下：

```
mongodb:
  primary:
    host: 192.168.9.60
    port: 20000
    database: test
  secondary:
    host: 192.168.9.60
    port: 20000
    database: test1
```

3、配置两个库的数据源

封装读取以mongodb开头的两个配置文件

```
@Data
@ConfigurationProperties(prefix = "mongodb")
public class MultipleMongoProperties {

    private MongoProperties primary = new MongoProperties();
    private MongoProperties secondary = new MongoProperties();
}
```

配置不同包路径下使用不同的数据源

第一个库的封装

```
@Configuration
@EnableMongoRepositories(basePackages = "com.neo.model.repository.primary",
        mongoTemplateRef = PrimaryMongoConfig.MONGO_TEMPLATE)
public class PrimaryMongoConfig {

    protected static final String MONGO_TEMPLATE = "primaryMongoTemplate";
}
```

第二个库的封装

```
@Configuration
@EnableMongoRepositories(basePackages = "com.neo.model.repository.secondary",
        mongoTemplateRef = SecondaryMongoConfig.MONGO_TEMPLATE)
public class SecondaryMongoConfig {

    protected static final String MONGO_TEMPLATE = "secondaryMongoTemplate";
}
```

读取对应的配置信息并且构造对应的MongoTemplate

```
@Configuration
public class MultipleMongoConfig {

    @Autowired
    private MultipleMongoProperties mongoProperties;

    @Primary
    @Bean(name = PrimaryMongoConfig.MONGO_TEMPLATE)
    public MongoTemplate primaryMongoTemplate() throws Exception
{
        return new MongoTemplate(primaryFactory(this.mongoProper
ties.getPrimary()));
    }

    @Bean
    @Qualifier(SecondaryMongoConfig.MONGO_TEMPLATE)
    public MongoTemplate secondaryMongoTemplate() throws Excepti
on {
        return new MongoTemplate(secondaryFactory(this.mongoProp
erties.getSecondary()));
    }

    @Bean
    @Primary
    public MongoDbFactory primaryFactory(MongoProperties mongo)
throws Exception {
        return new SimpleMongoDbFactory(new MongoClient(mongo.ge
tHost(), mongo.getPort()),
                mongo.getDatabase());
    }

    @Bean
    public MongoDbFactory secondaryFactory(MongoProperties mongo)
 throws Exception {
        return new SimpleMongoDbFactory(new MongoClient(mongo.ge
tHost(), mongo.getPort()),
                mongo.getDatabase());
    }
}
```

两个库的配置信息已经完成。

4、创建两个库分别对应的对象和Repository

借助lombok来构建对象

```
@Data
@AllArgsConstructor
@NoArgsConstructor
@Document(collection = "first_mongo")
public class PrimaryMongoObject {

    @Id
    private String id;

    private String value;

    @Override
    public String toString() {
        return "PrimaryMongoObject{" + "id='" + id + '\'' + ", v
alue='" + value + '\''
                + '}';
    }
}
```

对应的Repository

```
public interface PrimaryRepository extends MongoRepository<Prima
ryMongoObject, String> {
}
```

继承了 MongoRepository 会默认实现很多基本的增删改查，省了很多自己写dao层的代码

Secondary和上面的代码类似就不贴出来了

5、最后测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class MuliDatabaseTest {

    @Autowired
    private PrimaryRepository primaryRepository;

    @Autowired
    private SecondaryRepository secondaryRepository;

    @Test
    public void TestSave() {

        System.out.println("**********************************
************************");
        System.out.println("测试开始");
        System.out.println("**********************************
************************");

        this.primaryRepository
                .save(new PrimaryMongoObject(null, "第一个库的对象"
));

        this.secondaryRepository
                .save(new SecondaryMongoObject(null, "第二个库的对
象"));

        List<PrimaryMongoObject> primaries = this.primaryReposit
ory.findAll();
        for (PrimaryMongoObject primary : primaries) {
            System.out.println(primary.toString());
        }

        List<SecondaryMongoObject> secondaries = this.secondaryR
epository.findAll();

        for (SecondaryMongoObject secondary : secondaries) {
```

```
            System.out.println(secondary.toString());
        }

        System.out.println("************************************
************************");
        System.out.println("测试完成");
        System.out.println("************************************
************************");
    }

}
```

到此，mongodb多数据源的使用已经完成。

spring boot对常用的数据库支持外，对nosql 数据库也进行了封装自动化。

redis介绍

Redis是目前业界使用最广泛的内存数据存储。相比memcached，Redis支持更丰富的数据结构，例如hashes, lists, sets等，同时支持数据持久化。除此之外，Redis还提供一些类数据库的特性，比如事务，HA，主从库。可以说Redis兼具了缓存系统和数据库的一些特性，因此有着丰富的应用场景。本文介绍Redis在Spring Boot中两个典型的应用场景。

如何使用

1、引入 spring-boot-starter-redis

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

2、添加配置文件

```
# REDIS (RedisProperties)
# Redis数据库索引（默认为0）
spring.redis.database=0
# Redis服务器地址
spring.redis.host=192.168.0.58
# Redis服务器连接端口
spring.redis.port=6379
# Redis服务器连接密码（默认为空）
spring.redis.password=
# 连接池最大连接数（使用负值表示没有限制）
spring.redis.pool.max-active=8
# 连接池最大阻塞等待时间（使用负值表示没有限制）
spring.redis.pool.max-wait=-1
# 连接池中的最大空闲连接
spring.redis.pool.max-idle=8
# 连接池中的最小空闲连接
spring.redis.pool.min-idle=0
# 连接超时时间（毫秒）
spring.redis.timeout=0
```

3、添加cache的配置类

```
@Configuration
@EnableCaching
public class RedisConfig extends CachingConfigurerSupport{

    @Bean
    public KeyGenerator keyGenerator() {
        return new KeyGenerator() {
            @Override
            public Object generate(Object target, Method method,
 Object... params) {
                StringBuilder sb = new StringBuilder();
                sb.append(target.getClass().getName());
                sb.append(method.getName());
                for (Object obj : params) {
                    sb.append(obj.toString());
                }
                return sb.toString();
```

```
            }
        };
    }

    @SuppressWarnings("rawtypes")
    @Bean
    public CacheManager cacheManager(RedisTemplate redisTemplate)
 {
        RedisCacheManager rcm = new RedisCacheManager(redisTempl
ate);
        //设置缓存过期时间
        //rcm.setDefaultExpiration(60);//秒
        return rcm;
    }

    @Bean
    public RedisTemplate<String, String> redisTemplate(RedisConn
ectionFactory factory) {
        StringRedisTemplate template = new StringRedisTemplate(f
actory);
        Jackson2JsonRedisSerializer jackson2JsonRedisSerializer
= new Jackson2JsonRedisSerializer(Object.class);
        ObjectMapper om = new ObjectMapper();
        om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Vi
sibility.ANY);
        om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FI
NAL);
        jackson2JsonRedisSerializer.setObjectMapper(om);
        template.setValueSerializer(jackson2JsonRedisSerializer)
;
        template.afterPropertiesSet();
        return template;
    }

}
```

3、好了，接下来就可以直接使用了

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(Application.class)
public class TestRedis {

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Autowired
    private RedisTemplate redisTemplate;

    @Test
    public void test() throws Exception {
        stringRedisTemplate.opsForValue().set("aaa", "111");
        Assert.assertEquals("111", stringRedisTemplate.opsForVal
ue().get("aaa"));
    }

    @Test
    public void testObj() throws Exception {
        User user=new User("aa@126.com", "aa", "aa123456", "aa",
"123");
        ValueOperations<String, User> operations=redisTemplate.o
psForValue();
        operations.set("com.neox", user);
        operations.set("com.neo.f", user,1,TimeUnit.SECONDS);
        Thread.sleep(1000);
        //redisTemplate.delete("com.neo.f");
        boolean exists=redisTemplate.hasKey("com.neo.f");
        if(exists){
            System.out.println("exists is true");
        }else{
            System.out.println("exists is false");
        }
       // Assert.assertEquals("aa", operations.get("com.neo.f").
getUserName());
    }
}
```

以上都是手动使用的方式，如何在查找数据库的时候自动使用缓存呢，看下面；

4、自动根据方法生成缓存

```
@RequestMapping("/getUser")
@Cacheable(value="user-key")
public User getUser() {
    User user=userRepository.findByUserName("aa");
    System.out.println("若下面没出现“无缓存的时候调用”字样且能打印出数据表示测试成功");
    return user;
}
```

其中value的值就是缓存到redis中的key

共享Session-spring-session-data-redis

分布式系统中，sessiong共享有很多的解决方案，其中托管到缓存中应该是最常用的方案之一，

Spring Session官方说明

Spring Session provides an API and implementations for managing a user's session information.

如何使用

1、引入依赖

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

2、Session配置：

```
@Configuration
@EnableRedisHttpSession(maxInactiveIntervalInSeconds = 86400*30)
public class SessionConfig {
}
```

> maxInactiveIntervalInSeconds: 设置Session失效时间，使用Redis Session之后，原Boot的server.session.timeout属性不再生效

好了，这样就配置好了，我们来测试一下

3、测试

添加测试方法获取sessionid

```
@RequestMapping("/uid")
    String uid(HttpSession session) {
        UUID uid = (UUID) session.getAttribute("uid");
        if (uid == null) {
            uid = UUID.randomUUID();
        }
        session.setAttribute("uid", uid);
        return session.getId();
    }
```

登录redis 输入 `keys '*sessions*'`

```
t<spring:session:sessions:db031986-8ecc-48d6-b471-b137a3ed6bc4
t(spring:session:expirations:1472976480000
```

其中 1472976480000为失效时间，意思是这个时间后session失效， `db031986-8ecc-48d6-b471-b137a3ed6bc4` 为sessionId,登录http://localhost:8080/uid 发现会一致，就说明session 已经在redis里面进行有效的管理了。

如何在两台或者多台中共享session

其实就是按照上面的步骤在另一个项目中再次配置一次，启动后自动就进行了session共享。

RabbitMQ 即一个消息队列，主要是用来实现应用程序的异步和解耦，同时也能起到消息缓冲，消息分发的作用。

消息中间件在互联网公司的使用中越来越多，刚才还看到新闻阿里将RocketMQ捐献给了apache，当然了今天的主角还是讲RabbitMQ。消息中间件最主要的作用是解耦，中间件最标准的用法是生产者生产消息传送到队列，消费者从队列中拿取消息并处理，生产者不用关心是谁来消费，消费者不用关心谁在生产消息，从而达到解耦的目的。在分布式的系统中，消息队列也会被用在很多其它的方面，比如：分布式事务的支持，RPC的调用等等。

以前一直使用的是ActiveMQ，在实际的生产使用中也出现了一些小问题，在网络查阅了很多的资料后，决定尝试使用RabbitMQ来替换ActiveMQ，RabbitMQ的高可用性、高性能、灵活性等一些特点吸引了我们，查阅了一些资料整理出此文。

RabbitMQ介绍

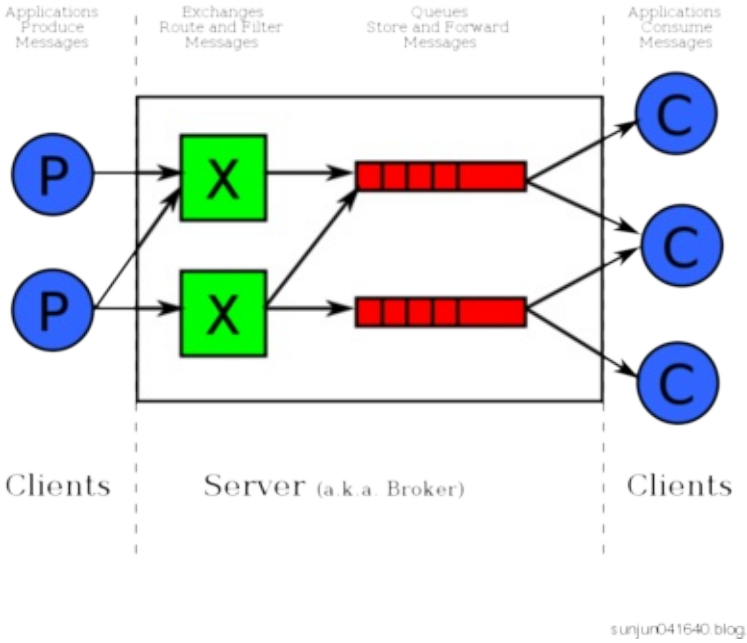
RabbitMQ是实现AMQP（高级消息队列协议）的消息中间件的一种，最初起源于金融系统，用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。RabbitMQ主要是为了实现系统之间的双向解耦而实现的。当生产者大量产生数据时，消费者无法快速消费，那么需要一个中间层。保存这个数据。

AMQP，即Advanced Message Queuing Protocol，高级消息队列协议，是应用层协议的一个开放标准，为面向消息的中间件设计。消息中间件主要用于组件之间的解耦，消息的发送者无需知道消息使用者的存在，反之亦然。AMQP的主要特征是面向消息、队列、路由（包括点对点和发布/订阅）、可靠性、安全。

RabbitMQ是一个开源的AMQP实现，服务器端用Erlang语言编写，支持多种客户端，如：Python、Ruby、.NET、Java、JMS、C、PHP、ActionScript、XMPP、STOMP等，支持AJAX。用于在分布式系统中存储转发消息，在易用性、扩展性、高可用性等方面表现不俗。

相关概念

通常我们谈到队列服务, 会有三个概念： 发消息者、队列、收消息者，RabbitMQ在这个基本概念之上, 多做了一层抽象, 在发消息者和 队列之间, 加入了交换器(Exchange). 这样发消息者和队列就没有直接联系, 转而变成发消息者把消息给交换器, 交换器根据调度策略再把消息再给队列。



- 左侧 P 代表 生产者，也就是往 RabbitMQ 发消息的程序。
- 中间即是 RabbitMQ，其中包括了 交换机 和 队列。
- 右侧 C 代表 消费者，也就是往 RabbitMQ 拿消息的程序。

那么，其中比较重要的概念有 *4* 个，分别为：虚拟主机，交换机，队列，和绑定。

- 虚拟主机：一个虚拟主机持有一组交换机、队列和绑定。为什么需要多个虚拟主机呢？很简单，RabbitMQ当中，用户只能在虚拟主机的粒度进行权限控制。 因此，如果需要禁止A组访问B组的交换机/队列/绑定，必须为A和B分别创建一个虚拟主机。每一个RabbitMQ服务器都有一个默认的虚拟主机“/”。
- 交换机：*Exchange* 用于转发消息，但是它不会做存储 ，如果没有 Queue bind 到 Exchange 的话，它会直接丢弃掉 Producer 发送过来的消息。 这里有一个比较重要的概念：路由键 。消息到交换机的时候，交互机会转发到对应的队列中，那么究竟转发到哪个队列，就要根据该路由键。
- 绑定：也就是交换机需要和队列相绑定，这其中如上图所示，是多对多的关系。

交换机(Exchange)

交换机的功能主要是接收消息并且转发到绑定的队列，交换机不存储消息，在启用ack模式后，交换机找不到队列会返回错误。交换机有四种类型：Direct, topic, Headers and Fanout

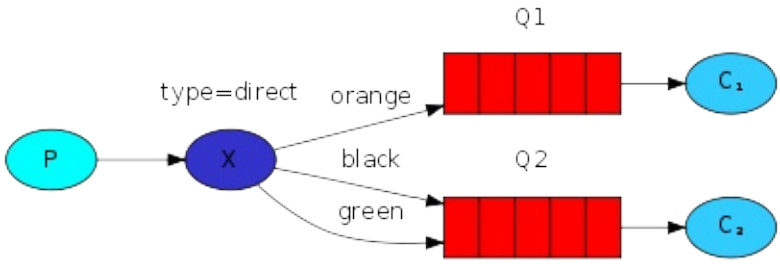
- Direct：direct 类型的行为是"先匹配, 再投送". 即在绑定时设定一个 **routing_key**, 消息的**routing_key** 匹配时, 才会被交换器投送到绑定的队列中

去.

- Topic：按规则转发消息（最灵活）
- Headers：设置header attribute参数类型的交换机
- Fanout：转发消息到所有绑定队列

Direct Exchange

Direct Exchange是RabbitMQ默认的交换机模式，也是最简单的模式，根据key全文匹配去寻找队列。



第一个 X - Q1 就有一个 binding key，名字为 orange； X - Q2 就有 2 个 binding key，名字为 black 和 green。当消息中的 路由键 和 这个 *binding key* 对应上的时候，那么就知道了该消息去到哪一个队列中。

Ps：为什么 X 到 Q2 要有 black，green，2个 binding key呢，一个不就行了吗？ - 这个主要是因为可能又有 Q3，而Q3只接受 black 的信息，而Q2不仅接受black 的信息，还接受 green 的信息。

Topic Exchange

Topic Exchange 转发消息主要是根据通配符。 在这种交换机下，队列和交换机的绑定会定义一种路由模式，那么，通配符就要在这种路由模式和路由键之间匹配后交换机才能转发消息。

在这种交换机模式下：

- 路由键必须是一串字符，用句号（ `.` ） 隔开，比如说 agreements.us，或者 agreements.eu.stockholm 等。
- 路由模式必须包含一个 星号（ `*` ），主要用于匹配路由键指定位置的一个单词，比如说，一个路由模式是这样子：agreements..b.*，那么就只能匹配路由键是这样子的：第一个单词是 agreements，第四个单词是 b。 井号（#）就表示相当于一个或者多个单词，例如一个匹配模式是agreements.eu.berlin.#，那么，以agreements.eu.berlin开头的路由键都是可以的。

具体代码发送的时候还是一样，第一个参数表示交换机，第二个参数表示routing key，第三个参数即消息。如下：

```
rabbitTemplate.convertAndSend("testTopicExchange","key1.a.c.key2"
, " this is  RabbitMQ!");
```

topic 和 direct 类似, 只是匹配上支持了"模式", 在"点分"的 routing_key 形式中, 可以使用两个通配符:

- `*` 表示一个词.
- `#` 表示零个或多个词.

Headers Exchange

headers 也是根据规则匹配, 相较于 direct 和 topic 固定地使用 routing_key , headers 则是一个自定义匹配规则的类型. 在队列与交换器绑定时, 会设定一组键值对规则, 消息中也包括一组键值对(headers 属性), 当这些键值对有一对, 或全部匹配时, 消息被投送到对应队列.

Fanout Exchange

Fanout Exchange 消息广播的模式，不管路由键或者是路由模式，会把消息发给绑定给它的全部队列，如果配置了routing_key会被忽略。

springboot集成RabbitMQ

springboot集成RabbitMQ非常简单，如果只是简单的使用配置非常少，springboot提供了spring-boot-starter-amqp项目对消息各种支持。

简单使用

1、配置pom包，主要是添加spring-boot-starter-amqp的支持

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

2、配置文件

配置rabbitmq的安装地址、端口以及账户信息

```
spring.application.name=spirng-boot-rabbitmq

spring.rabbitmq.host=192.168.0.86
spring.rabbitmq.port=5672
spring.rabbitmq.username=admin
spring.rabbitmq.password=123456
```

3、队列配置

```
@Configuration
public class RabbitConfig {

    @Bean
    public Queue Queue() {
        return new Queue("hello");
    }

}
```

3、发送者

rabbitTemplate是springboot 提供的默认实现

```
public class HelloSender {

    @Autowired
    private AmqpTemplate rabbitTemplate;

    public void send() {
        String context = "hello " + new Date();
        System.out.println("Sender : " + context);
        this.rabbitTemplate.convertAndSend("hello", context);
    }

}
```

4、接收者

```
@Component
@RabbitListener(queues = "hello")
public class HelloReceiver {

    @RabbitHandler
    public void process(String hello) {
        System.out.println("Receiver  : " + hello);
    }

}
```

5、测试

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class RabbitMqHelloTest {

    @Autowired
    private HelloSender helloSender;

    @Test
    public void hello() throws Exception {
        helloSender.send();
    }

}
```

> 注意，发送者和接收者的queue name必须一致，不然不能接收

多对多使用

一个发送者，N个接收者或者N个发送者和N个接收者会出现什么情况呢？

一对多发送
对上面的代码进行了小改造,接收端注册了两个Receiver,Receiver1和Receiver2，发送端加入参数计数，接收端打印接收到的参数，下面是测试代码，发送一百条消息，来观察两个接收端的执行效果

```
@Test
public void oneToMany() throws Exception {
    for (int i=0;i<100;i++){
        neoSender.send(i);
    }
}
```

结果如下：

```
Receiver 1: spirng boot neo queue ****** 11
Receiver 2: spirng boot neo queue ****** 12
Receiver 2: spirng boot neo queue ****** 14
Receiver 1: spirng boot neo queue ****** 13
Receiver 2: spirng boot neo queue ****** 15
Receiver 1: spirng boot neo queue ****** 16
Receiver 1: spirng boot neo queue ****** 18
Receiver 2: spirng boot neo queue ****** 17
Receiver 2: spirng boot neo queue ****** 19
Receiver 1: spirng boot neo queue ****** 20
```

根据返回结果得到以下结论

> 一个发送者，N个接受者,经过测试会均匀的将消息发送到N个接收者中

多对多发送

复制了一份发送者，加入标记，在一百个循环中相互交替发送

```
@Test
    public void manyToMany() throws Exception {
        for (int i=0;i<100;i++){
            neoSender.send(i);
            neoSender2.send(i);
        }
}
```

结果如下：

```
Receiver 1: spirng boot neo queue ****** 20
Receiver 2: spirng boot neo queue ****** 20
Receiver 1: spirng boot neo queue ****** 21
Receiver 2: spirng boot neo queue ****** 21
Receiver 1: spirng boot neo queue ****** 22
Receiver 2: spirng boot neo queue ****** 22
Receiver 1: spirng boot neo queue ****** 23
Receiver 2: spirng boot neo queue ****** 23
Receiver 1: spirng boot neo queue ****** 24
Receiver 2: spirng boot neo queue ****** 24
Receiver 1: spirng boot neo queue ****** 25
Receiver 2: spirng boot neo queue ****** 25
```

> 结论：和一对多一样，接收端仍然会均匀接收到消息

高级使用

对象的支持

springboot以及完美的支持对象的发送和接收，不需要格外的配置。

```
//发送者
public void send(User user) {
    System.out.println("Sender object: " + user.toString());
    this.rabbitTemplate.convertAndSend("object", user);
}

...

//接收者
@RabbitHandler
public void process(User user) {
    System.out.println("Receiver object : " + user);
}
```

结果如下：

```
Sender object: User{name='neo', pass='123456'}
Receiver object : User{name='neo', pass='123456'}
```

Topic Exchange

topic 是RabbitMQ中最灵活的一种方式，可以根据routing_key自由的绑定不同的队列

首先对topic规则配置，这里使用两个队列来测试

```
@Configuration
public class TopicRabbitConfig {

    final static String message = "topic.message";
    final static String messages = "topic.messages";

    @Bean
    public Queue queueMessage() {
        return new Queue(TopicRabbitConfig.message);
    }

    @Bean
    public Queue queueMessages() {
        return new Queue(TopicRabbitConfig.messages);
    }

    @Bean
    TopicExchange exchange() {
        return new TopicExchange("exchange");
    }

    @Bean
    Binding bindingExchangeMessage(Queue queueMessage, TopicExch
ange exchange) {
        return BindingBuilder.bind(queueMessage).to(exchange).wi
th("topic.message");
    }

    @Bean
    Binding bindingExchangeMessages(Queue queueMessages, TopicEx
change exchange) {
        return BindingBuilder.bind(queueMessages).to(exchange).w
ith("topic.#");
    }
}
```

使用queueMessages同时匹配两个队列，queueMessage只匹配"topic.message"队列

```
public void send1() {
    String context = "hi, i am message 1";
    System.out.println("Sender : " + context);
    this.rabbitTemplate.convertAndSend("exchange", "topic.messag
e", context);
}

public void send2() {
    String context = "hi, i am messages 2";
    System.out.println("Sender : " + context);
    this.rabbitTemplate.convertAndSend("exchange", "topic.messag
es", context);
}
```

发送send1会匹配到topic.#和topic.message 两个Receiver都可以收到消息，发送send2只有topic.#可以匹配所有只有Receiver2监听到消息

Fanout Exchange

Fanout 就是我们熟悉的广播模式或者订阅模式，给Fanout交换机发送消息，绑定了这个交换机的所有队列都收到这个消息。

Fanout 相关配置

```
@Configuration
public class FanoutRabbitConfig {

    @Bean
    public Queue AMessage() {
        return new Queue("fanout.A");
    }

    @Bean
    public Queue BMessage() {
        return new Queue("fanout.B");
    }

    @Bean
    public Queue CMessage() {
```

```
        return new Queue("fanout.C");
    }

    @Bean
    FanoutExchange fanoutExchange() {
        return new FanoutExchange("fanoutExchange");
    }

    @Bean
    Binding bindingExchangeA(Queue AMessage,FanoutExchange fanou
tExchange) {
        return BindingBuilder.bind(AMessage).to(fanoutExchange);
    }

    @Bean
    Binding bindingExchangeB(Queue BMessage, FanoutExchange fano
utExchange) {
        return BindingBuilder.bind(BMessage).to(fanoutExchange);
    }

    @Bean
    Binding bindingExchangeC(Queue CMessage, FanoutExchange fano
utExchange) {
        return BindingBuilder.bind(CMessage).to(fanoutExchange);
    }

}
```

这里使用了A、B、C三个队列绑定到Fanout交换机上面，发送端的routing_key写任何字符都会被忽略：

```
public void send() {
        String context = "hi, fanout msg ";
        System.out.println("Sender : " + context);
        this.rabbitTemplate.convertAndSend("fanoutExchange","",
context);
}
```

结果如下：

```
Sender : hi, fanout msg
...
fanout Receiver B: hi, fanout msg
fanout Receiver A  : hi, fanout msg
fanout Receiver C: hi, fanout msg
```

结果说明，绑定到fanout交换机上面的队列都收到了消息

上篇文章介绍了如何使用Spring Boot上传文件，这篇文章我们介绍如何使用Spring Boot将文件上传到分布式文件系统FastDFS中。

这个项目会在上一个项目的基础上进行构建。

1、pom包配置

我们使用Spring Boot最新版本1.5.9、jdk使用1.8、tomcat8.0。

```
<dependency>
    <groupId>org.csource</groupId>
    <artifactId>fastdfs-client-java</artifactId>
    <version>1.27-SNAPSHOT</version>
</dependency>
```

加入了 `fastdfs-client-java` 包，用来调用FastDFS相关的API。

2、配置文件

resources目录下添加 `fdfs_client.conf` 文件

```
connect_timeout = 60
network_timeout = 60
charset = UTF-8
http.tracker_http_port = 8080
http.anti_steal_token = no
http.secret_key = 123456

tracker_server = 192.168.53.85:22122
tracker_server = 192.168.53.86:22122
```

配置文件设置了连接的超时时间，编码格式以及tracker_server地址等信息

详细内容参考：fastdfs-client-java

3、封装FastDFS上传工具类

封装FastDFSFile，文件基础信息包括文件名、内容、文件类型、作者等。

```
public class FastDFSFile {
    private String name;
    private byte[] content;
    private String ext;
    private String md5;
    private String author;
    //省略getter、setter
```

封装FastDFSClient类，包含常用的上传、下载、删除等方法。

首先在类加载的时候读取相应的配置信息，并进行初始化。

```
static {
    try {
        String filePath = new ClassPathResource("fdfs_client.con
f").getFile().getAbsolutePath();;
        ClientGlobal.init(filePath);
        trackerClient = new TrackerClient();
        trackerServer = trackerClient.getConnection();
        storageServer = trackerClient.getStoreStorage(trackerSer
ver);
    } catch (Exception e) {
        logger.error("FastDFS Client Init Fail!",e);
    }
}
```

文件上传

```
public static String[] upload(FastDFSFile file) {
    logger.info("File Name: " + file.getName() + "File Length:"
+ file.getContent().length);

    NameValuePair[] meta_list = new NameValuePair[1];
    meta_list[0] = new NameValuePair("author", file.getAuthor())
;

    long startTime = System.currentTimeMillis();
    String[] uploadResults = null;
    try {
        storageClient = new StorageClient(trackerServer, storage
Server);
        uploadResults = storageClient.upload_file(file.getConten
t(), file.getExt(), meta_list);
    } catch (IOException e) {
        logger.error("IO Exception when uploadind the file:" + f
ile.getName(), e);
    } catch (Exception e) {
        logger.error("Non IO Exception when uploadind the file:"
 + file.getName(), e);
    }
    logger.info("upload_file time used:" + (System.currentTimeMi
llis() - startTime) + " ms");

    if (uploadResults == null) {
        logger.error("upload file fail, error code:" + storageCl
ient.getErrorCode());
    }
    String groupName = uploadResults[0];
    String remoteFileName = uploadResults[1];

    logger.info("upload file successfully!!!" + "group_name:" +
groupName + ", remoteFileName:" + " " + remoteFileName);
    return uploadResults;
}
```

使用FastDFS提供的客户端storageClient来进行文件上传，最后将上传结果返回。

根据groupName和文件名获取文件信息。

```
public static FileInfo getFile(String groupName, String remoteFileName) {
    try {
        storageClient = new StorageClient(trackerServer, storageServer);
        return storageClient.get_file_info(groupName, remoteFileName);
    } catch (IOException e) {
        logger.error("IO Exception: Get File from Fast DFS failed", e);
    } catch (Exception e) {
        logger.error("Non IO Exception: Get File from Fast DFS failed", e);
    }
    return null;
}
```

下载文件

```
public static InputStream downFile(String groupName, String remo
teFileName) {
    try {
        storageClient = new StorageClient(trackerServer, storage
Server);
        byte[] fileByte = storageClient.download_file(groupName,
 remoteFileName);
        InputStream ins = new ByteArrayInputStream(fileByte);
        return ins;
    } catch (IOException e) {
        logger.error("IO Exception: Get File from Fast DFS faile
d", e);
    } catch (Exception e) {
        logger.error("Non IO Exception: Get File from Fast DFS f
ailed", e);
    }
    return null;
}
```

删除文件

```
public static void deleteFile(String groupName, String remoteFil
eName)
        throws Exception {
    storageClient = new StorageClient(trackerServer, storageServ
er);
    int i = storageClient.delete_file(groupName, remoteFileName)
;
    logger.info("delete file successfully!!!" + i);
}
```

使用FastDFS时，直接调用FastDFSClient对应的方法即可。

4、编写上传控制类

从MultipartFile中读取文件信息，然后使用FastDFSClient将文件上传到FastDFS集群中。

```
public String saveFile(MultipartFile multipartFile) throws IOExc
eption {
    String[] fileAbsolutePath={};
    String fileName=multipartFile.getOriginalFilename();
    String ext = fileName.substring(fileName.lastIndexOf(".") + 1
);
    byte[] file_buff = null;
    InputStream inputStream=multipartFile.getInputStream();
    if(inputStream!=null){
        int len1 = inputStream.available();
        file_buff = new byte[len1];
        inputStream.read(file_buff);
    }
    inputStream.close();
    FastDFSFile file = new FastDFSFile(fileName, file_buff, ext)
;
    try {
        fileAbsolutePath = FastDFSClient.upload(file);  //upload
 to fastdfs
    } catch (Exception e) {
        logger.error("upload file Exception!",e);
    }
    if (fileAbsolutePath==null) {
        logger.error("upload file failed,please upload again!");
    }
    String path=FastDFSClient.getTrackerUrl()+fileAbsolutePath[0
]+ "/"+fileAbsolutePath[1];
    return path;
}
```

请求控制，调用上面方法 `saveFile()` 。

```
@PostMapping("/upload") //new annotation since 4.3
public String singleFileUpload(@RequestParam("file") MultipartFi
le file,
                               RedirectAttributes redirectAttrib
utes) {
    if (file.isEmpty()) {
        redirectAttributes.addFlashAttribute("message", "Please
select a file to upload");
        return "redirect:uploadStatus";
    }
    try {
        // Get the file and save it somewhere
        String path=saveFile(file);
        redirectAttributes.addFlashAttribute("message",
                "You successfully uploaded '" + file.getOriginal
Filename() + "'");
        redirectAttributes.addFlashAttribute("path",
                "file path url '" + path + "'");
    } catch (Exception e) {
        logger.error("upload file failed",e);
    }
    return "redirect:/uploadStatus";
}
```

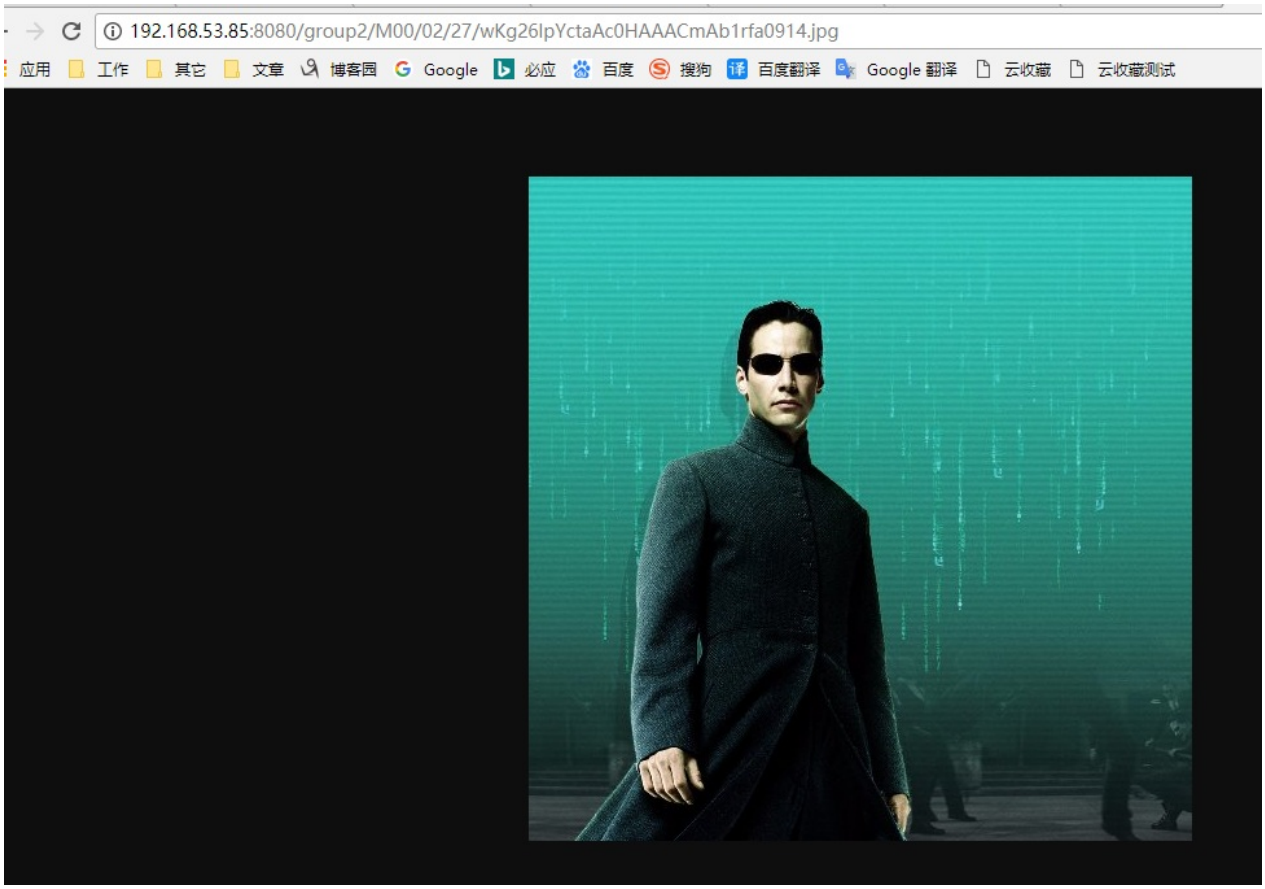
上传成功之后，将文件的路径展示到页面，效果图如下：

Spring Boot - Upload Status

You successfully uploaded 'neo.jpg'

file path url 'http://192.168.53.85:8080/group2/M00/02/27/wKg26lpYctaAc0HAAACmAb1rfa0914.jpg'

在浏览器中访问此Url，可以看到成功通过FastDFS展示：



> 这样使用Spring Boot 集成FastDFS的案例就完成了。

spring boot项目如何测试，如何部署，在生产中有什么好的部署方案吗？这篇文章就来介绍一下spring boot 如何开发、调试、打包到最后的投产上线。

开发阶段

单元测试

在开发阶段的时候最重要的是单元测试了，springboot对单元测试的支持已经很完善了。

1、在pom包中添加spring-boot-starter-test包引用

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

2、开发测试类

以最简单的helloworld为例，在测试类的类头部需要添加：`@RunWith(SpringRunner.class)` 和 `@SpringBootTest` 注解，在测试方法的顶端添加 `@Test` 即可，最后在方法上点击右键run就可以运行。

```
@RunWith(SpringRunner.class)
@SpringBootTest
public class ApplicationTests {

    @Test
    public void hello() {
        System.out.println("hello world");
    }

}
```

实际使用中，可以按照项目的正常使用去注入dao层代码或者是service层代码进行测试验证，spring-boot-starter-test提供很多基础用法，更难得的是增加了对Controller层测试的支持。

```
//简单验证结果集是否正确
Assert.assertEquals(3, userMapper.getAll().size());

//验证结果集，提示
Assert.assertTrue("错误，正确的返回值为200", status == 200);
Assert.assertFalse("错误，正确的返回值为200", status != 200);
```

引入了 `MockMvc` 支持了对Controller层的测试，简单示例如下：

```
public class HelloControlerTests {

    private MockMvc mvc;

    //初始化执行
    @Before
    public void setUp() throws Exception {
        mvc = MockMvcBuilders.standaloneSetup(new HelloControlle
r()).build();
    }

    //验证controller是否正常响应并打印返回结果
    @Test
    public void getHello() throws Exception {
        mvc.perform(MockMvcRequestBuilders.get("/hello").accept(
MediaType.APPLICATION_JSON))
                .andExpect(MockMvcResultMatchers.status().isOk()
)
                .andDo(MockMvcResultHandlers.print())
                .andReturn();
    }

    //验证controller是否正常响应并判断返回结果是否正确
    @Test
    public void testHello() throws Exception {
        mvc.perform(MockMvcRequestBuilders.get("/hello").accept(
MediaType.APPLICATION_JSON))
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("Hello World"
)));
    }

}
```

单元测试是验证你代码第一道屏障，要养成每写一部分代码就进行单元测试的习惯，不要等到全部集成后再进行测试，集成后因为更关注整体运行效果，很容易遗漏掉代码底层的bug.

集成测试

整体开发完成之后进入集成测试，spring boot项目的启动入口在 Application类中，直接运行run方法就可以启动项目，但是在调试的过程中我们肯定需要不断的去调试代码，如果每修改一次代码就需要手动重启一次服务就很麻烦，spring boot非常贴心的给出了热部署的支持，很方便在web项目中调试使用。

pom需要添加以下的配置：

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

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

添加以上配置后，项目就支持了热部署，非常方便集成测试。

投产上线

其实我觉得这个阶段，应该还是比较简单一般分为两种；一种是打包成jar包直接执行，另一种是打包成war包放到tomcat服务器下。

打成jar包

如果你使用的是maven来管理项目，执行以下命令既可以

```
cd 项目跟目录（和pom.xml同级）
mvn clean package
## 或者执行下面的命令
## 排除测试代码后进行打包
mvn clean package  -Dmaven.test.skip=true
```

打包完成后jar包会生成到target目录下，命名一般是 项目名+版本号.jar

启动jar包命令

```
java -jar  target/spring-boot-scheduler-1.0.0.jar
```

这种方式，只要控制台关闭，服务就不能访问了。下面我们使用在后台运行的方式来启动:

```
nohup java -jar target/spring-boot-scheduler-1.0.0.jar &
```

也可以在启动的时候选择读取不同的配置文件

```
java -jar app.jar --spring.profiles.active=dev
```

也可以在启动的时候设置jvm参数

```
java -Xms10m -Xmx80m -jar app.jar &
```

gradle
如果使用的是gradle,使用下面命令打包

```
gradle build
java -jar build/libs/mymodule-0.0.1-SNAPSHOT.jar
```

打成**war**包

打成war包一般可以分两种方式来实现，第一种可以通过eclipse这种开发工具来导出war包，另外一种是使用命令来完成，这里主要介绍后一种

1、maven项目，修改pom包

将

```
<packaging>jar</packaging>
```

改为

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

2、打包时排除tomcat.

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

在这里将scope属性设置为provided，这样在最终形成的WAR中不会包含这个JAR包，因为Tomcat或Jetty等服务器在运行时将会提供相关的API类。

3、注册启动类

创建ServletInitializer.java，继承SpringBootServletInitializer ，覆盖configure()，把启动类Application注册进去。外部web应用服务器构建Web Application Context的时候，会把启动类添加进去。

```
public class ServletInitializer extends SpringBootServletInitial
izer {
    @Override
    protected SpringApplicationBuilder configure(SpringApplicati
onBuilder application) {
        return application.sources(Application.class);
    }
}
```

最后执行

```
mvn clean package  -Dmaven.test.skip=true
```

会在target目录下生成：项目名+版本号.war文件，拷贝到tomcat服务器中启动即可。

gradle

如果使用的是gradle,基本步奏一样，build.gradle中添加war的支持，排除spring-boot-starter-tomcat：

```
...

apply plugin: 'war'

...

dependencies {
    compile("org.springframework.boot:spring-boot-starter-web:1.
4.2.RELEASE"){
        exclude mymodule:"spring-boot-starter-tomcat"
    }
}
...
```

再使用构建命令

```
gradle build
```

war会生成在build\libs 目录下。

生产运维

查看JVM参数的值

可以根据java自带的jinfo命令：

```
jinfo -flags pid
```

来查看jar 启动后使用的是什么gc、新生代、老年代分批的内存都是多少，示例如下：

```
-XX:CICompilerCount=3 -XX:InitialHeapSize=234881024 -XX:MaxHeapS
ize=3743416320 -XX:MaxNewSize=1247805440 -XX:MinHeapDeltaBytes=5
24288 -XX:NewSize=78118912 -XX:OldSize=156762112 -XX:+UseCompres
sedClassPointers -XX:+UseCompressedOops -XX:+UseFastUnorderedTim
eStamps -XX:+UseParallelGC
```

- `-XX:CICompilerCount` ：最大的并行编译数
- `-XX:InitialHeapSize` 和 `-XX:MaxHeapSize` ：指定JVM的初始和最大堆内存大小
- `-XX:MaxNewSize` ： JVM堆区域新生代内存的最大可分配大小
- ...
- `-XX:+UseParallelGC` ：垃圾回收使用Parallel收集器

如何重启

简单粗暴

直接kill掉进程再次启动jar包

```
ps -ef|grep java
##拿到对于Java程序的pid
kill -9 pid
## 再次重启
Java -jar  xxxx.jar
```

当然这种方式比较传统和暴力，所以建议大家使用下面的方式来管理

脚本执行

如果使用的是maven,需要包含以下的配置

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <executable>true</executable>
    </configuration>
</plugin>
```

如果使用是gradle，需要包含下面配置

```
springBoot {
    executable = true
}
```

启动方式：

1、可以直接 `./yourapp.jar` 来启动

2、注册为服务

也可以做一个软链接指向你的jar包并加入到 `init.d` 中，然后用命令来启动。

init.d 例子:

```
ln -s /var/yourapp/yourapp.jar /etc/init.d/yourapp
chmod +x /etc/init.d/yourapp
```

这样就可以使用 `stop` 或者是 `restart` 命令去管理你的应用。

```
/etc/init.d/yourapp start|stop|restart
```

或者

```
service yourapp start|stop|restart
```

微服务的特点决定了功能模块的部署是分布式的，大部分功能模块都是运行在不同的机器上，彼此通过服务调用进行交互，前后台的业务流会经过很多个微服务的处理和传递，出现了异常如何快速定位是哪个环节出现了问题？

在这种框架下，微服务的监控显得尤为重要。本文主要结合Spring Boot Actuator，跟大家一起分享微服务Spring Boot Actuator的常见用法，方便我们在日常中对我们的微服务进行监控治理。

Actuator监控

Spring Boot使用“习惯优于配置的理念”，采用包扫描和自动化配置的机制来加载依赖jar中的Spring bean,不需要任何Xml配置，就可以实现Spring的所有配置。虽然这样做能让我们的代码变得非常简洁，但是整个应用的实例创建和依赖关系等信息都被离散到了各个配置类的注解上，这使得我们分析整个应用中资源和实例的各种关系变得非常的困难。

Actuator是Spring Boot提供的对应用系统的自省和监控的集成功能，可以查看应用配置的详细信息，例如自动化配置信息、创建的Spring beans以及一些环境属性等。

Actuator监控只需要添加以下依赖就可以完成

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

为了保证actuator暴露的监控接口的安全性，需要添加安全控制的依赖spring-boot-start-security依赖，访问应用监控端点时，都需要输入验证信息。Security依赖，可以选择不加，不进行安全管理，但不建议这么做。

Actuator 的 REST 接口

Actuator监控分成两类：原生端点和用户自定义端点；自定义端点主要是指扩展性，用户可以根据自己的实际应用，定义一些比较关心的指标，在运行期进行监控。

原生端点是在应用程序里提供众多 Web 接口，通过它们了解应用程序运行时的内部状况。原生端点又可以分成三类：

- 应用配置类：可以查看应用在运行期的静态信息：例如自动配置信息、加载的springbean信息、yml文件配置信息、环境信息、请求映射信息；
- 度量指标类：主要是运行期的动态信息，例如堆栈、请求连、一些健康指标、metrics信息等；
- 操作控制类：主要是指shutdown,用户可以发送一个请求将应用的监控功能关闭。

Actuator 提供了 13 个接口，具体如下表所示。

HTTP 方法	路径	描述
GET	/autoconfig	提供了一份自动配置报告，记录哪些自动配置条件通过了，哪些没通过
GET	/configprops	描述配置属性(包含默认值)如何注入Bean
GET	/beans	描述应用程序上下文里全部的Bean，以及它们的关系
GET	/dump	获取线程活动的快照
GET	/env	获取全部环境属性
GET	/env/{name}	根据名称获取特定的环境属性值
GET	/health	报告应用程序的健康指标，这些值由HealthIndicator的实现类提供
GET	/info	获取应用程序的定制信息，这些信息由info打头的属性提供
GET	/mappings	描述全部的URI路径，以及它们和控制器(包含Actuator端点)的映射关系
GET	/metrics	报告各种应用程序度量信息，比如内存用量和HTTP请求计数
GET	/metrics/{name}	报告指定名称的应用程序度量值
POST	/shutdown	关闭应用程序，要求endpoints.shutdown.enabled设置为true
GET	/trace	提供基本的HTTP请求跟踪信息(时间戳、HTTP头等)

快速上手

相关配置

项目依赖

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

配置文件

```
server:
  port: 8080
management:
  security:
    enabled: false #关掉安全认证
  port: 8088 #管理端口调整成8088
  context-path: /monitor #actuator的访问路径
endpoints:
  shutdown:
    enabled: true

info:
   app:
      name: spring-boot-actuator
      version: 1.0.0
```

- `management.security.enabled=false` 默认有一部分信息需要安全验证之后才可以查看，如果去掉这些安全认证，直接设置management.security.enabled=false
- `management.context-path=/monitor` 代表启用单独的url地址来监控Spring Boot应用，为了安全一般都启用独立的端口来访问后端的监控信息
- `endpoints.shutdown.enabled=true` 启用接口关闭Spring Boot

配置完成之后，启动项目就可以继续验证各个监控功能了。

命令详解

autoconfig

Spring Boot的自动配置功能非常便利，但有时候也意味着出问题比较难找出具体的原因。使用 autoconfig 可以在应用运行时查看代码了某个配置在什么条件下生效，或者某个自动配置为什么没有生效。

启动示例项目，访问：`http://localhost:8088/monitor/autoconfig` 返回部分信息如下：

```
{
    "positiveMatches": {
     "DevToolsDataSourceAutoConfiguration": {
            "notMatched": [
                {
                    "condition": "DevToolsDataSourceAutoConfigur
ation.DevToolsDataSourceCondition",
                    "message": "DevTools DataSource Condition di
d not find a single DataSource bean"
                }
            ],
            "matched": [ ]
        },
        "RemoteDevToolsAutoConfiguration": {
            "notMatched": [
                {
                    "condition": "OnPropertyCondition",
                    "message": "@ConditionalOnProperty (spring.d
evtools.remote.secret) did not find property 'secret'"
                }
            ],
            "matched": [
                {
                    "condition": "OnClassCondition",
                    "message": "@ConditionalOnClass found requir
ed classes 'javax.servlet.Filter', 'org.springframework.http.ser
ver.ServerHttpRequest'; @ConditionalOnMissingClass did not find
unwanted class"
                }
            ]
        }
    }
}
```


configprops

查看配置文件中设置的属性内容，以及一些配置属性的默认值。

启动示例项目，访问：`http://localhost:8088/monitor/configprops` 返回部分信息如下：

```
{
  ...
  "environmentEndpoint": {
    "prefix": "endpoints.env",
    "properties": {
      "id": "env",
      "sensitive": true,
      "enabled": true
    }
  },
  "spring.http.multipart-org.springframework.boot.autoconfigure.
web.MultipartProperties": {
    "prefix": "spring.http.multipart",
    "properties": {
      "maxRequestSize": "10MB",
      "fileSizeThreshold": "0",
      "location": null,
      "maxFileSize": "1MB",
      "enabled": true,
      "resolveLazily": false
    }
  },
  "infoEndpoint": {
    "prefix": "endpoints.info",
    "properties": {
      "id": "info",
      "sensitive": false,
      "enabled": true
    }
  }
  ...
}
```

beans

根据示例就可以看出，展示了bean的别名、类型、是否单例、类的地址、依赖等信息。

启动示例项目，访问：`http://localhost:8088/monitor/beans` 返回部分信息如下：

```
[
  {
    "context": "application:8080:management",
    "parent": "application:8080",
    "beans": [
      {
        "bean": "embeddedServletContainerFactory",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.context.embedded.tomca
t.TomcatEmbeddedServletContainerFactory",
        "resource": "null",
        "dependencies": [

        ]
      },
      {
        "bean": "endpointWebMvcChildContextConfiguration",
        "aliases": [

        ],
        "scope": "singleton",
        "type": "org.springframework.boot.actuate.autoconfigure.
EndpointWebMvcChildContextConfiguration$$EnhancerBySpringCGLIB$$
a4a10f9d",
        "resource": "null",
        "dependencies": [

        ]
      }
  }
]
```


dump

/dump 接口会生成当前线程活动的快照。这个功能非常好，方便我们在日常定位问题的时候查看线程的情况。 主要展示了线程名、线程ID、线程的状态、是否等待锁资源等信息。

启动示例项目，访问：`http://localhost:8088/monitor/dump` 返回部分信息如下：


```
[
  {
    "threadName": "http-nio-8088-exec-6",
    "threadId": 49,
    "blockedTime": -1,
    "blockedCount": 0,
    "waitedTime": -1,
    "waitedCount": 2,
    "lockName": "java.util.concurrent.locks.AbstractQueuedSynchr
onizer$ConditionObject@1630a501",
    "lockOwnerId": -1,
    "lockOwnerName": null,
    "inNative": false,
    "suspended": false,
    "threadState": "WAITING",
    "stackTrace": [
      {
        "methodName": "park",
        "fileName": "Unsafe.java",
        "lineNumber": -2,
        "className": "sun.misc.Unsafe",
        "nativeMethod": true
      },
      {
        "methodName": "park",
        "fileName": "LockSupport.java",
        "lineNumber": 175,
        "className": "java.util.concurrent.locks.LockSupport",
        "nativeMethod": false
      },
      {
        "methodName": "await",
        "fileName": "AbstractQueuedSynchronizer.java",
```

```
        "lineNumber": 2039,
        "className": "java.util.concurrent.locks.AbstractQueuedS
ynchronizer$ConditionObject",
        "nativeMethod": false
      },
       ...
      {
        "methodName": "getTask",
        "fileName": "ThreadPoolExecutor.java",
        "lineNumber": 1067,
        "className": "java.util.concurrent.ThreadPoolExecutor",
        "nativeMethod": false
      },
      {
        "methodName": "runWorker",
        "fileName": "ThreadPoolExecutor.java",
        "lineNumber": 1127,
        "className": "java.util.concurrent.ThreadPoolExecutor",
        "nativeMethod": false
      },
      {
        "methodName": "run",
        "fileName": "ThreadPoolExecutor.java",
        "lineNumber": 617,
        "className": "java.util.concurrent.ThreadPoolExecutor$Wo
rker",
        "nativeMethod": false
      },
      {
        "methodName": "run",
        "fileName": "TaskThread.java",
        "lineNumber": 61,
        "className": "org.apache.tomcat.util.threads.TaskThread$
WrappingRunnable",
        "nativeMethod": false
      },
      {
        "methodName": "run",
        "fileName": "Thread.java",
        "lineNumber": 745,
```

```
        "className": "java.lang.Thread",
        "nativeMethod": false
      }
    ],
    "lockedMonitors": [

    ],
    "lockedSynchronizers": [

    ],
    "lockInfo": {
      "className": "java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject",
      "identityHashCode": 372286721
    }
  }
  ...
]
```

env

展示了系统环境变量的配置信息，包括使用的环境变量、JVM 属性、命令行参数、项目使用的jar包等信息。和configprops不同的是，configprops关注于配置信息，env关注运行环境信息。

启动示例项目，访问：`http://localhost:8088/monitor/env` 返回部分信息如下：

```
{
  "profiles": [

  ],
  "server.ports": {
    "local.management.port": 8088,
    "local.server.port": 8080
  },
  "servletContextInitParams": {

  },
  "systemProperties": {
    "com.sun.management.jmxremote.authenticate": "false",
    "java.runtime.name": "Java(TM) SE Runtime Environment",
    "spring.output.ansi.enabled": "always",
    "sun.boot.library.path": "C:\\Program Files\\Java\\jdk1.8.0_
101\\jre\\bin",
    "java.vm.version": "25.101-b13",
    "java.vm.vendor": "Oracle Corporation",
    "java.vendor.url": "http://java.oracle.com/",
    "java.rmi.server.randomIDs": "true",
    "path.separator": ";",
    "java.vm.name": "Java HotSpot(TM) 64-Bit Server VM",
    "file.encoding.pkg": "sun.io",
    "user.country": "CN",
    "user.script": "",
    "sun.java.launcher": "SUN_STANDARD",
    "sun.os.patch.level": "",
    "PID": "5268",
    "com.sun.management.jmxremote.port": "60093",
    "java.vm.specification.name": "Java Virtual Machine Spe
```


为了避免敏感信息暴露到 /env 里，所有名为password、secret、key(或者名字中最后一段是这些)的属性在 /env 里都会加上“*”。举个例子，如果有一个属性名字是database.password，那么它在/env中的显示效果是这样的：

```
"database.password":"******"
```

/env/{name}用法

就是env的扩展 可以获取指定配置信息，比如：`http://localhost:8088/monitor/env/java.vm.version`,返回：`{"java.vm.version":"25.101-b13"}`

health

可以看到 HealthEndPoint 给我们提供默认的监控结果，包含 磁盘检测和数据库检测

启动示例项目，访问：`http://localhost:8088/monitor/health` 返回部分信息，下面的JSON响应是由状态、磁盘空间和db。描述了应用程序的整体健康状态,UP 表明应用程序是健康的。磁盘空间描述总磁盘空间,剩余的磁盘空间和最小阈值。`application.properties` 阈值是可配置的

```
{
  "status": "UP",
  "diskSpace": {
    "status": "UP",
    "total": 209715195904,
    "free": 183253909504,
    "threshold": 10485760
  }
  "db": {
        "status": "UP",
        "database": "MySQL",
        "hello": 1
    }
}
```

> 其实看 Spring Boot-actuator 源码，你会发现 HealthEndPoint 提供的信息不仅限于此，org.springframework.boot.actuate.health 包下 你会发现 ElasticsearchHealthIndicator、RedisHealthIndicator、RabbitHealthIndicator 等

info

info就是我们自己配置在配置文件中以Info开头的配置信息，比如我们在示例项目中的配置是：

```
info:
   app:
      name: spring-boot-actuator
      version: 1.0.0
```

启动示例项目，访问：`http://localhost:8088/monitor/info` 返回部分信息如下：

```
{
  "app": {
    "name": "spring-boot-actuator",
    "version": "1.0.0"
  }
}
```

mappings

描述全部的URI路径，以及它们和控制器的映射关系

启动示例项目，访问：`http://localhost:8088/monitor/mappings` 返回部分信息如下：

```
{
  "/**/favicon.ico": {
    "bean": "faviconHandlerMapping"
  },
  "{[/hello]}": {
    "bean": "requestMappingHandlerMapping",
    "method": "public java.lang.String com.neo.controller.HelloC
ontroller.index()"
  },
  "{[/error]}": {
    "bean": "requestMappingHandlerMapping",
    "method": "public org.springframework.http.ResponseEntity<ja
va.util.Map<java.lang.String, java.lang.Object>> org.springframe
work.boot.autoconfigure.web.BasicErrorController.error(javax.ser
vlet.http.HttpServletRequest)"
  }
}
```


metrics

最重要的监控内容之一，主要监控了JVM内容使用、GC情况、类加载信息等。

启动示例项目，访问：`http://localhost:8088/monitor/metrics` 返回部分信息如下：

```
{
  "mem": 337132,
  "mem.free": 183380,
  "processors": 4,
  "instance.uptime": 254552,
  "uptime": 259702,
  "systemload.average": -1.0,
  "heap.committed": 292864,
  "heap.init": 129024,
  "heap.used": 109483,
  "heap": 1827840,
  "nonheap.committed": 45248,
  "nonheap.init": 2496,
  "nonheap.used": 44269,
  "nonheap": 0,
  "threads.peak": 63,
  "threads.daemon": 43,
  "threads.totalStarted": 83,
  "threads": 46,
  "classes": 6357,
  "classes.loaded": 6357,
  "classes.unloaded": 0,
  "gc.ps_scavenge.count": 8,
  "gc.ps_scavenge.time": 99,
  "gc.ps_marksweep.count": 1,
  "gc.ps_marksweep.time": 43,
  "httpsessions.max": -1,
  "httpsessions.active": 0
}
```


对 `/metrics` 接口提供的信息进行简单分类如下表：

分类	前缀	报告内容
垃圾收集器	gc.*	已经发生过的垃圾收集次数，以及垃圾收集所耗的时间，适用于标记-清理垃圾收集器和并行垃圾收集器(数据源自java.lang.management. GarbageCollectorMXBean)
内存	mem.*	分配给应用程序的内存数量和空闲的内存数量(数据源自java.lang. Runtime)
堆	heap.*	当前内存用量(数据源自 java.lang.management.MemoryUsage)
类加载器	classes.*	JVM类加载器加载与卸载的类的数量(数据源自 java.lang. management.ClassLoadingMXBean
系统	processors、 instance.uptime、 uptime、 systemload.average	系统信息，例如处理器数量(数据源自 java.lang.Runtime)、运行时间(数据源自 java.lang.management.RuntimeMXBean)、平均负载(数据源自 java.lang.management.OperatingSystemMXBe
线程池	thread.*	线程、守护线程的数量，以及JVM启动后的线程数量峰值(数据源自 java.lang .management.ThreadMXBean)
数据源	datasource.*	数据源连接的数量(源自数据源的元数据，仅当Spring应用程序上下文里存在 DataSource Bean的时候才会有这个信息)
Tomcat会话	httpsessions.*	Tomcat的活跃会话数和最大会话数(数据源自嵌入式Tomcat的Bean，仅在使用嵌入式Tomcat服务运行应用程序时才有这个信息)
HTTP	counter.status.、 *gauge.response.*	多种应用程序服务HTTP请求的度量值与计数器

解释说明：

- 请注意，这里的一些度量值，比如数据源和Tomcat会话，仅在应用程序中运行特定组件时才有数据。你还可以注册自己的度量信息。
- HTTP的计数器和度量值需要做一点说明。counter.status 后的值是HTTP状态码，随后是所请求的路径。举个例子，counter.status.200.metrics 表明/metrics端点返回 200(OK) 状态码的次数。
- HTTP的度量信息在结构上也差不多，却在报告另一类信息。它们全部以 gauge.response 开头,，表明这是HTTP响应的度量信息。前缀后是对应的路径。度量值是以毫秒为单位的时间，反映了最近处理该路径请求的耗时。

- 这里还有几个特殊的值需要注意。root路径指向的是根路径或/。star-star代表了那些Spring 认为是静态资源的路径，包括图片、JavaScript和样式表，其中还包含了那些找不到的资源。这就是为什么你经常会看到counter.status.404.star-star，这是返回了HTTP 404 (NOT FOUND) 状态的请求数。
- `/metrics` 接口会返回所有的可用度量值，但你也可能只对某个值感兴趣。要获取单个值，请求时可以在URL后加上对应的键名。例如，要查看空闲内存大小,可以向 `/metrics/mem.free` 发一 个GET请求。例如访问：`http://localhost:8088/monitor/metrics/mem.free` ，返回：`{"mem.free":178123}` 。

shutdown

开启接口优雅关闭Spring Boot应用，要使用这个功能首先需要在配置文件中开启：

```
endpoints:
  shutdown:
    enabled: true
```

配置完成之后，启动示例项目，访问：`http://localhost:8088/monitor/shutdown` 返回部分信息如下：

```
{
    "message": "Shutting down, bye..."
}
```

此时你会发现应用已经被关闭。

trace

/trace 接口能报告所有Web请求的详细信息，包括请求方法、路径、时间戳以及请求和响应的头信息，记录每一次请求的详细信息。

启动示例项目，先访问一次：`http://localhost:8080/hello` ，再到浏览器执行：`http://localhost:8088/monitor/trace` 查看返回信息：

```
[
  {
    "timestamp": 1516780334777,
    "info": {
      "method": "GET",
      "path": "/hello",
      "headers": {
        "request": {
          "host": "localhost:8080",
          "connection": "keep-alive",
          "cache-control": "max-age=0",
          "user-agent": "Mozilla/5.0 (Windows NT 10.0; Win64; x6
4) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/63.0.3239.84 Sa
fari/537.36",
          "upgrade-insecure-requests": "1",
          "accept": "text/html,application/xhtml+xml,application
/xml;q=0.9,image/webp,image/apng,*/*;q=0.8",
          "accept-encoding": "gzip, deflate, br",
          "accept-language": "zh-CN,zh;q=0.9",
          "cookie": "UM_distinctid=16053ba344f1cd-0dc220c44cc94-
b7a103e-13c680-16053ba3450751; Hm_lvt_0fb30c642c5f6453f17d881f52
9a1141=1513076406,1514961720,1515649377; CNZZDATA1260945749=2322
52692-1513233181-%7C1516085149; Hm_lvt_6d8e8bb59814010152d98507a
18ad229=1515247964,1515296008,1515672972,1516086283"
        },
        "response": {
          "X-Application-Context": "application:8080",
          "Content-Type": "text/html;charset=UTF-8",
          "Content-Length": "11",
          "Date": "Wed, 24 Jan 2018 07:52:14 GMT",
          "status": "200"
        }
      },
      "timeTaken": "4"
    }
  }
]
```


上述信息展示了，/hello请求的详细信息。

其它配置

敏感信息访问限制

根据上面表格，鉴权为false的，表示不敏感，可以随意访问，否则就是做了一些保护，不能随意访问。

```
endpoints.mappings.sensitive=false
```

这样需要对每一个都设置，比较麻烦。敏感方法默认是需要用户拥有ACTUATOR角色，因此，也可以设置关闭安全限制：

```
management.security.enabled=false
```

或者配合Spring Security做细粒度控制。

启用和禁用接口

虽然Actuator的接口都很有用，但你不一定需要全部这些接口。默认情况下，所有接口(除 了/shutdown)都启用。比如要禁用 /metrics 接口，则可以设置如下：

```
endpoints.metrics.enabled = false
```

如果你只想打开一两个接口，那就先禁用全部接口，然后启用那几个你要的，这样更方便。

```
endpoints.enabled = false
endpoints.metrics.enabled = true
```

上一篇文章《springboot(十九)：使用Spring Boot Actuator监控应用》介绍了Spring Boot Actuator的使用，Spring Boot Actuator提供了对单个Spring Boot的监控，信息包含：应用状态、内存、线程、堆栈等等，比较全面的监控了Spring Boot应用的整个生命周期。

但是这样监控也有一些问题：第一，所有的监控都需要调用固定的接口来查看，如果全面查看应用状态需要调用很多接口，并且接口返回的Json信息不方便运营人员理解；第二，如果Spring Boot应用集群非常大，每个应用都需要调用不同的接口来查看监控信息，操作非常繁琐低效。在这样的背景下，就诞生了另外一个开源软件：**Spring Boot Admin**。

什么是Spring Boot Admin?

Spring Boot Admin 是一个管理和监控Spring Boot 应用程序的开源软件。每个应用都认为是一个客户端，通过HTTP或者使用 Eureka注册到admin server中进行展示，Spring Boot Admin UI部分使用AngularJs将数据展示在前端。

Spring Boot Admin 是一个针对spring-boot的actuator接口进行UI美化封装的监控工具。他可以：在列表中浏览所有被监控spring-boot项目的基本信息，详细的Health信息、内存信息、JVM信息、垃圾回收信息、各种配置信息（比如数据源、缓存列表和命中率）等，还可以直接修改logger的level。

这篇文章给大家介绍如何使用Spring Boot Admin对Spring Boot应用进行监控。

监控单体应用

这节给大家展示如何使用Spring Boot Admin监控单个Spring Boot应用。

Admin Server端

项目依赖

```
<dependencies>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server</artifactId>
    <version>1.5.6</version>
  </dependency>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server-ui</artifactId>
    <version>1.5.6</version>
  </dependency>
</dependencies>
```

配置文件

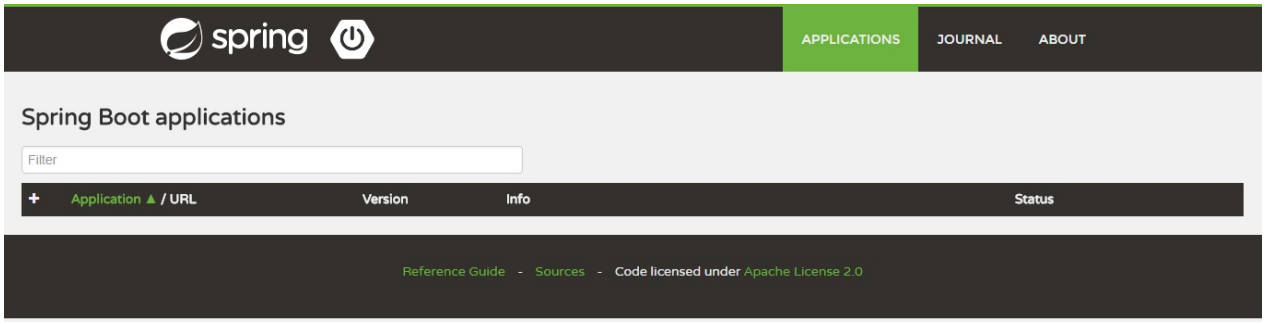
```
server.port=8000
```

服务端设置端口为：8000。

启动类

```
@Configuration
@EnableAutoConfiguration
@EnableAdminServer
public class AdminServerApplication {

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

完成上面三步之后，启动服务端，浏览器访问 `http://localhost:8000` 可以看到以下界面：



示例代码

Admin Client端

项目依赖

```
<dependencies>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>1.5.6</version>
  </dependency>
</dependencies>
```

配置文件

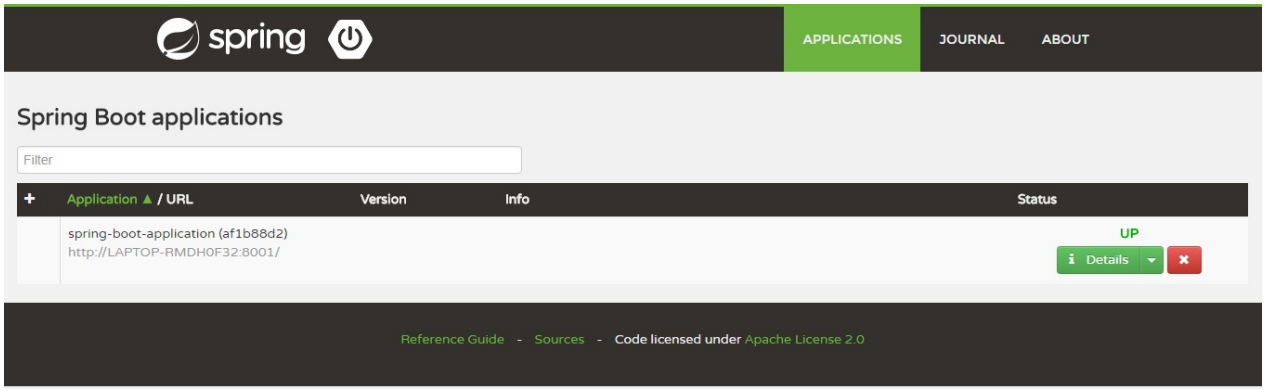
```
server.port=8001

spring.boot.admin.url=http://localhost:8000
management.security.enabled=false
```

- `spring.boot.admin.url` 配置Admin Server的地址
- `management.security.enabled=false` 关闭安全验证

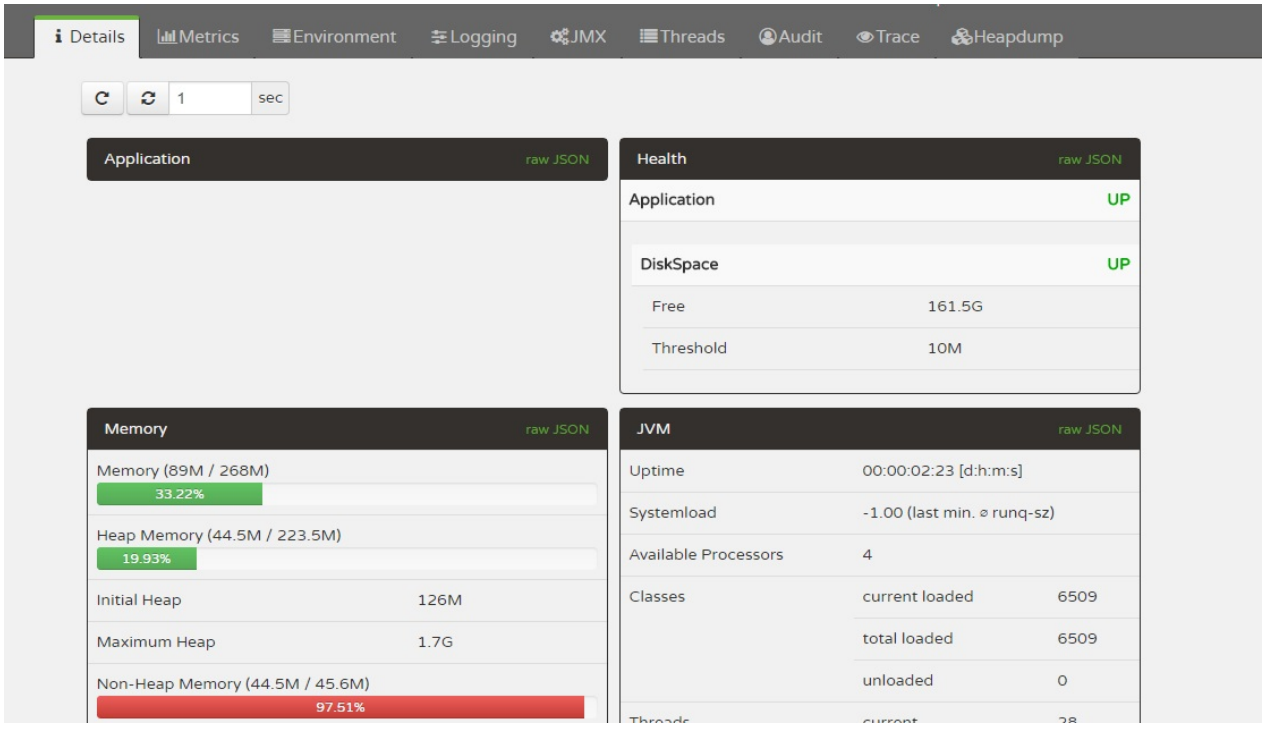
启动类

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

配置完成之后，启动Client端服务，再次访问服务：`http://localhost:8000` 可以看到客户端的相关信息。



首页会展示被监控的各个服务，点击详情可以查看某个服务的具体监控信息



通过上图可以看出，Spring Boot Admin以图形化的形式展示了应用的各项信息，这些信息大多都来自于Spring Boot Actuator提供的接口。

监控微服务

如果我们使用的是单个Spring Boot应用，就需要在每一个被监控的应用中配置Admin Server的地址信息；如果应用都注册在Eureka中就不需要再对每个应用进行配置，Spring Boot Admin会自动从注册中心抓取应用的相关信息。

这里使用四个示例项目来演示：

- spring-boot-admin-server Admin Server端
- spring-cloud-eureka 注册中心
- spring-cloud-producer 应用一，Admin Client端
- spring-cloud-producer-2 应用二，Admin Client端

首先启动注册中心spring-cloud-eureka，如果对Eureka不了解的同学可以查看这篇文章springcloud(二)：注册中心Eureka

Server端

示例项目：spring-boot-admin-server

项目依赖

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server</artifactId>
    <version>1.5.6</version>
  </dependency>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server-ui</artifactId>
    <version>1.5.6</version>
  </dependency>
</dependencies>
```

增加了对eureka的支持

配置文件

```
server:
  port: 8000
spring:
  application:
    name: admin-server
eureka:
  instance:
    leaseRenewalIntervalInSeconds: 10
  client:
    registryFetchIntervalSeconds: 5
    serviceUrl:
      defaultZone: ${EUREKA_SERVICE_URL:http://localhost:8761}/e
ureka/

management.security.enabled: false
```

配置文件中添加了eureka的相关配置

启动类

```
@Configuration
@EnableAutoConfiguration
@EnableDiscoveryClient
@EnableAdminServer
public class AdminServerApplication {

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

上述步骤完成之后，启动Server端。

Client端

示例项目：spring-cloud-producer和spring-cloud-producer-2

项目依赖

```
<dependencies>
  <dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-eureka</artifactId>
  </dependency>
  <dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>1.5.6</version>
  </dependency>
</dependencies>
```

配置文件

```
server:
  port: 9000
spring:
  application:
    name: producer
eureka:
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
management:
  security:
    enabled: false
```

我们发现配置文件中并没有添加Admin Server的相关配置

启动类

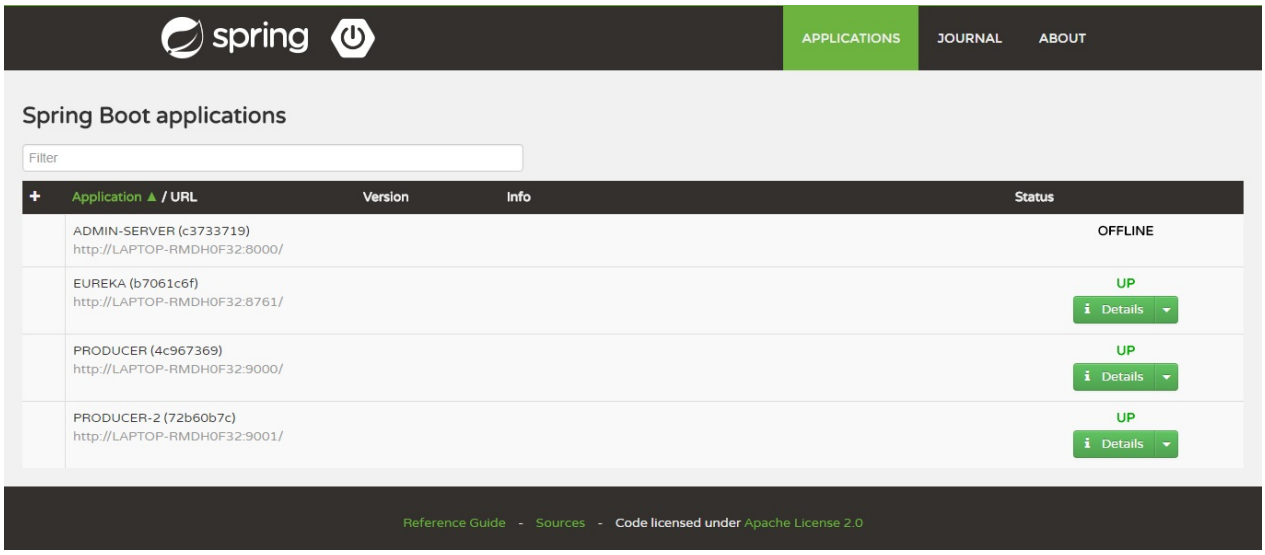
```
@SpringBootApplication
@EnableDiscoveryClient
public class ProducerApplication {

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

Web层

```
@RequestMapping("/hello")
  public String index(@RequestParam String name) {
      logger.info("request one/two  name is "+name);
      return "hello "+name+"，this is first messge";
  }
```

web层添加了/hello的请求方法，方法中使用one/two区别是哪个应用。spring-cloud-producer-2和spring-cloud-producer代码类似，具体大家可以查看示例代码。

完成上面配置之后，分别启动项目：spring-cloud-producer和spring-cloud-producer-2，浏览器访问 `http://localhost:8000` 可以看到以下界面：



从上图可以看出Admin Server监控了四个实例，包括Server自己，注册中心、两个PRODUCER。说明Admin Server自动从服务中心抓取了所有的实例信息并进行了监控。点击Detail可以具体查看某一个示例的监控信息。

示例代码

邮件告警

Spring Boot Admin将微服务中所有应用信息在后台进行了展示，非常方便我们对微服务整体的监控和治理。但是我们的运营人员也不可能一天24小时盯着监控后台，因此如果服务有异常的时候，有对应的邮件告警就太好了，其实Spring Boot Admin也给出了支持。

我们对上面的示例项目spring-boot-admin-server进行改造。

添加依赖

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

增加了邮件发送的starter包

配置文件

```
spring:
  mail:
    host: smtp.qq.com
    username: xxxxx@qq.com
    password: xxxx
    properties:
      mail:
        smtp:
          auth: true
          starttls:
            enable: true
            required: true
  boot:
    admin:
      notify:
        mail:
          from: xxxx@qq.com
          to: xxxx@qq.com
# http://codecentric.github.io/spring-boot-admin/1.5.6/#mail-not
ifications
```

在配置文件中添加邮件发送相关信息：邮件的发送者、接受者、协议、移动授权码等。关于Spring Boot邮件发送，可以参考springboot(十)：邮件服务

配置完成后，重新启动项目spring-boot-admin-server，这样Admin Server就具备了邮件告警的功能，默认情况下Admin Server对Eureka中的服务上下线都进行了监控，当服务上下线的时候我们就会收到如下邮件：

当然这只是最基本的邮件监控，在实际的使用过程中，需要根据我们的情况对邮件告警内容进行自定义，比如监控堆内存的使用情况，当到达一定比例的时候进行告警等。