SpringBoot系列从入门到进阶小册

Spring Boot 快速入门

Spring Boot Web 开发

Spring Boot 数据访问

Spring Boot 日志管理

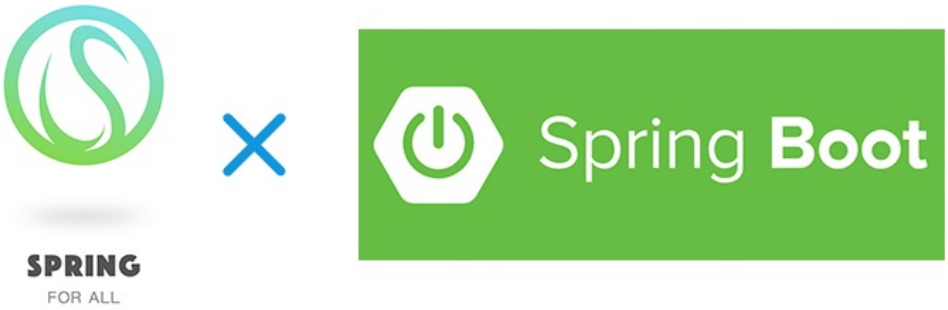
Spring Boot 监控管理

Spring Boot 整合 Dubbo

Spring Boot 整合 Elasticsearch

Spring Boot 监控管理

具体相关代码见：社区开源项目 http://www.spring4all.com/projects



该教程内容不定时更新，如果您有更好的文章推荐，请投稿，微信 139-5868-6678。

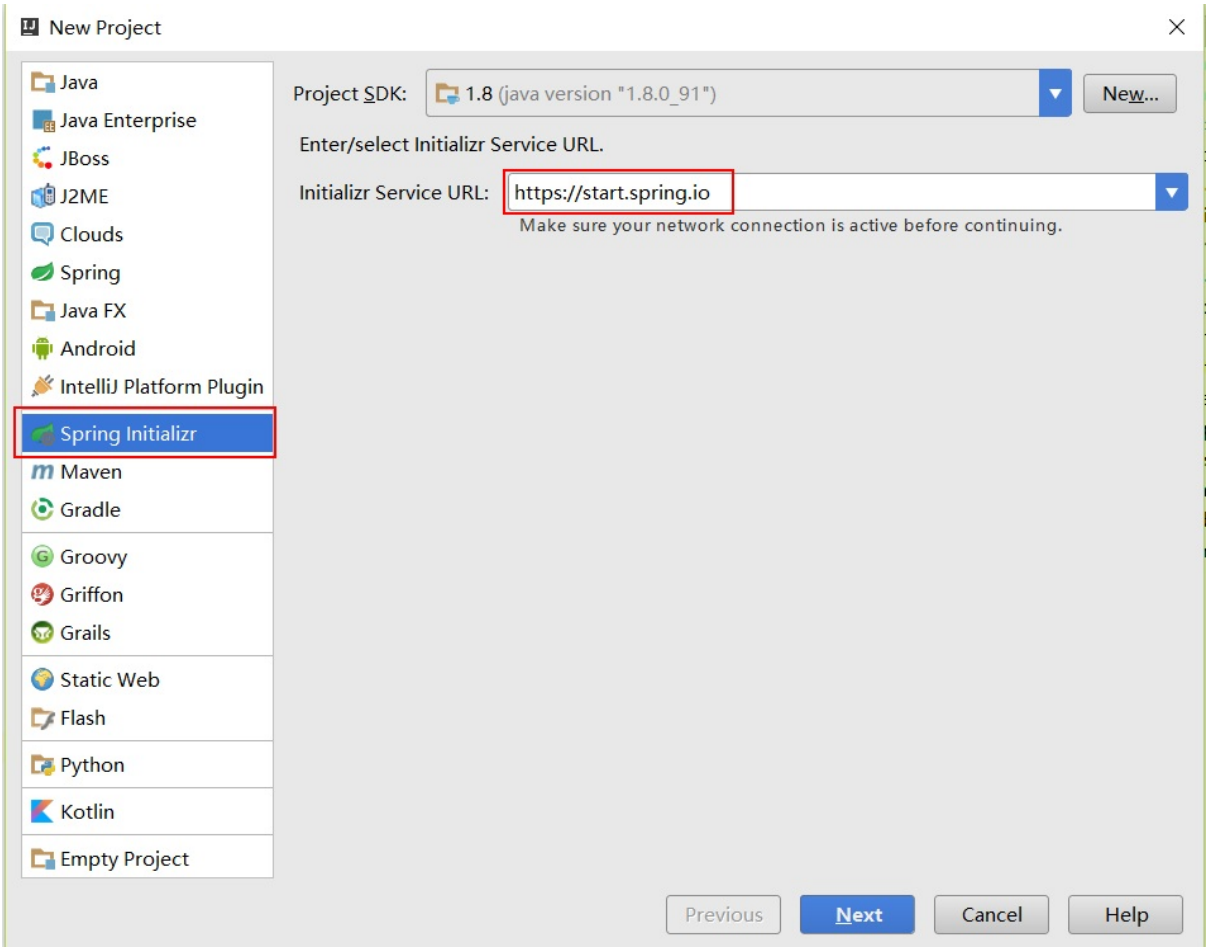
本系列教程由 Spring For All 社区整理，采用 CC BY 3.0 CN 协议 进行许可。 可自由转载、引用，但需署名作者且注明文章出处。

持续学习精进Spring周边相关技术，请长按识别二维码关注“SpringForAll社区”官方公众号

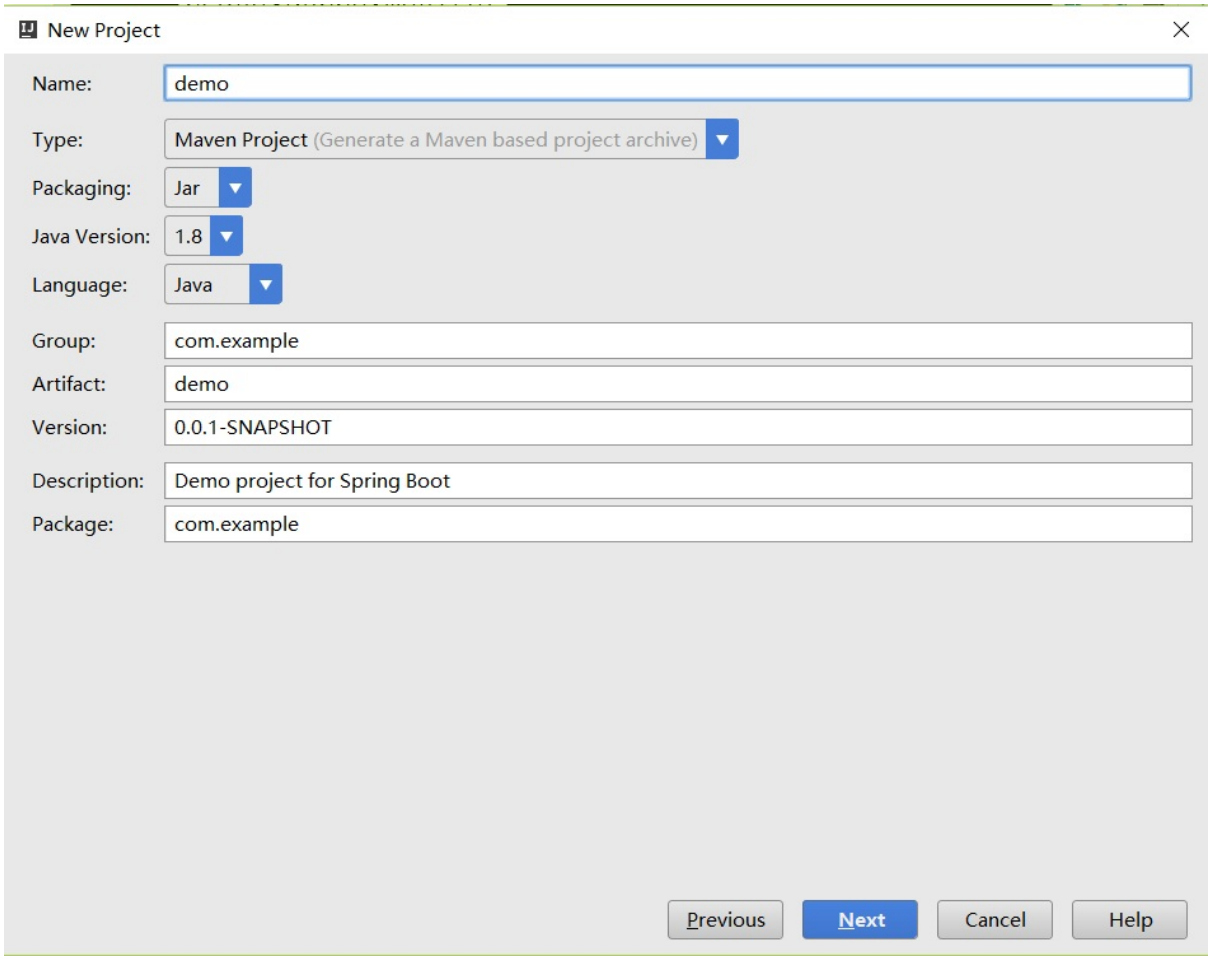
在之前的所有Spring Boot和Spring Cloud相关博文中，都会涉及Spring Boot工程的创建。而创建的方式多种多样，我们可以通过Maven来手工构建或是通过脚手架等方式快速搭建，也可以通过《Spring Boot快速入门》一文中提到的 `SPRING INITIALIZR` 页面工具来创建，相信每位读者都有自己最喜欢和最为熟练的创建方式。

本文我们将介绍嵌入的Intellij中的Spring Initializr工具，它同Web提供的创建功能一样，可以帮助我们快速的构建出一个基础的Spring Boot/Cloud工程。

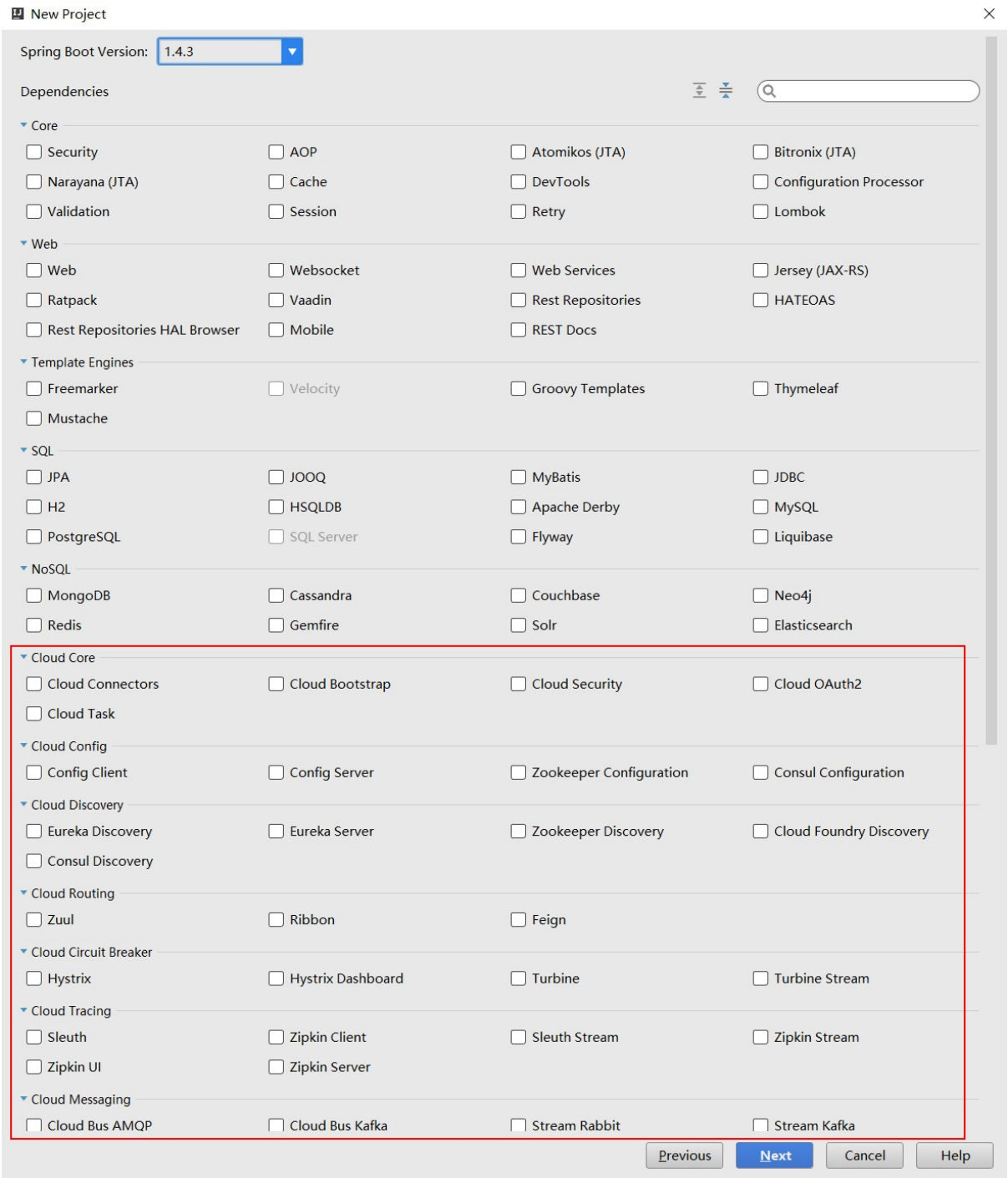
- 菜单栏中选择 `File` => `New` => `Project..` ，我们可以看到如下图所示的创建功能窗口。其中 `Initial Service Url` 指向的地址就是Spring官方提供的Spring Initializr工具地址，所以这里创建的工程实际上也是基于它的Web工具来实现的。



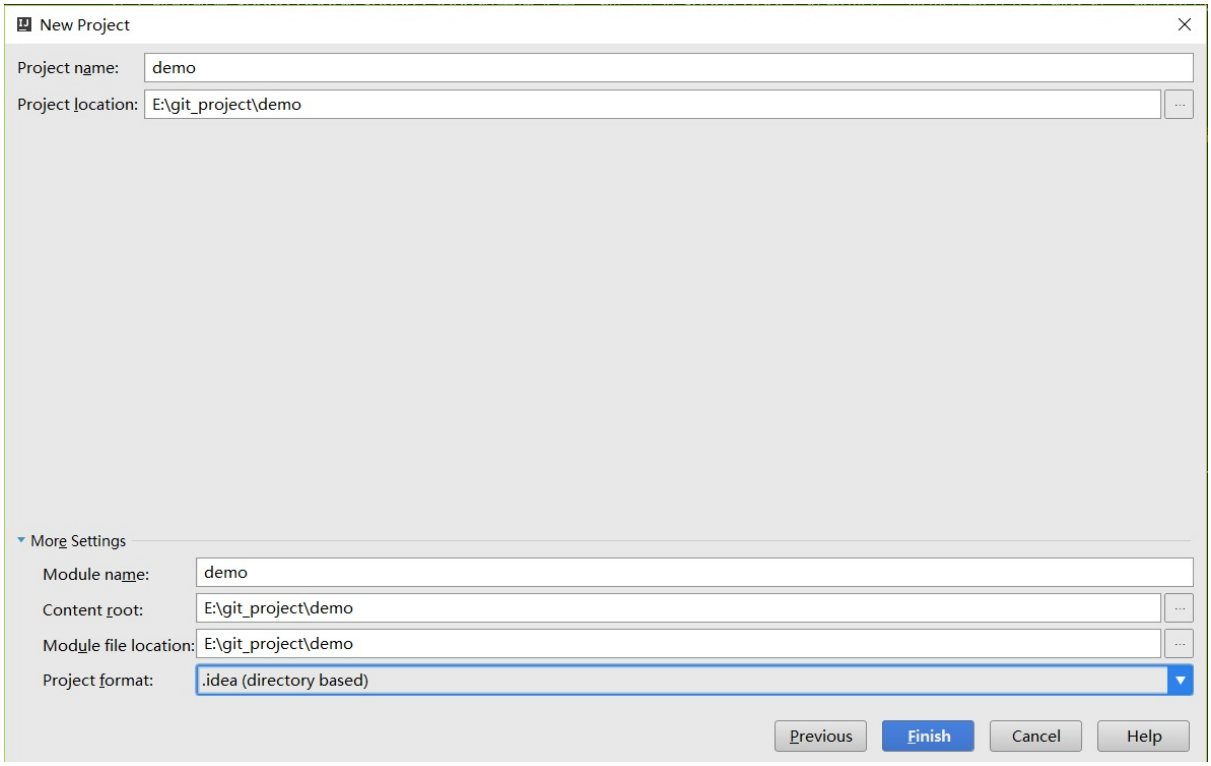
- 点击 `Next` ，等待片刻后，我们可以看到如下图所示的工程信息窗口，在这里我们可以编辑我们想要创建的工程信息。其中， `Type` 可以改变我们要构建的工程类型，比如：Maven、Gradle； `Language` 可以选择：Java、Groovy、Kotlin。



- 点击 `Next` ，进入选择Spring Boot版本和依赖管理的窗口。在这里值的我们关注的是，它不仅包含了Spring Boot Starter POMs中的各个依赖，还包含了Spring Cloud的各种依赖。



- 点击 `Next`，进入最后关于工程物理存储的一些细节。最后，点击 `Finish` 就能完成工程的构建了。



Intellij中的 `Spring Initializr` 虽然还是基于官方Web实现，但是通过工具来进行调用并直接将结果构建到我们的本地文件系统中，让整个构建流程变得更加顺畅，还没有体验过此功能的Spring Boot/Cloud爱好者们不妨可以尝试一下这种不同的构建方式。

摘要: 原创出处:www.bysocket.com 泥瓦匠BYSocket 希望转载，保留摘要，谢谢!

“以前是人放狗看家，现在是狗牵着人散步” — 随笔

Spring Boot 系列文章：《Spring Boot 那些事》

一、Spring Boot 自述

世界上最好的文档来源自官方的《Spring Boot Reference Guide》，是这样介绍的：

> Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can “just run”...Most Spring Boot applications need very little Spring configuration.

Spring Boot(英文中是“引导”的意思)，是用来简化Spring应用的搭建到开发的过程。应用开箱即用，只要通过 “just run”（可能是 java -jar 或 tomcat 或 maven插件run 或 shell脚本），就可以启动项目。二者，Spring Boot 只要很少的Spring配置文件（例如那些xml，property）。 因为“习惯优先于配置”的原则，使得Spring Boot在快速开发应用和微服务架构实践中得到广泛应用。 Javaer装好JDK环境和Maven工具就可以开始学习Boot了~

二、HelloWorld实战详解

首先得有个maven基础项目，可以直接使用Maven骨架工程生成Maven骨架Web项目，即man archetype:generate命令：

```
mvn archetype:generate -DgroupId=springboot -DartifactId=springboot-helloworld -Darchetype
ArtifactId=maven-archetype-quickstart -DinteractiveMode=false
```

2.1 pom.xml配置

代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>springboot</groupId>
    <artifactId>springboot-helloworld</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-helloworld :: HelloWorld Demo</name>

    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
```

```
        <version>1.3.3.RELEASE</version>
    </parent>

    <dependencies>
        <!-- Spring Boot web依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

只要加入一个 Spring Boot 启动父依赖即可。

2.2 Controller层

HelloWorldController的代码如下：

```
/**
 * Spring Boot HelloWorld案例
 *
 * Created by bysocket on 16/4/26.
 */
@RestController
public class HelloWorldController {

    @RequestMapping("/")
    public String sayHello() {
        return "Hello,World!";
    }
}
```

@RestController和@RequestMapping注解是来自SpringMVC的注解，它们不是SpringBoot的特定部分。 1. @RestController：提供实现了REST API，可以服务JSON,XML或者其他。这里是以String的形式渲染出结果。 2. @RequestMapping：提供路由信息，"/"路径的HTTP Request都会被映射到sayHello方法进行处理。 具体参考，世界上最好的文档来源自官方的《Spring Framework Document》

2.3 启动应用类

和第一段描述一样，开箱即用。如下面Application类：

```
/**
 * Spring Boot应用启动类
 *
 * Created by bysocket on 16/4/26.
 */
@SpringBootApplication
public class Application {

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

\1. @SpringBootApplication：Spring Boot 应用的标识 2. Application很简单，一个main函数作为主入口。SpringApplication引导应用，并将Application本身作为参数传递给run方法。具体run方法会启动嵌入式的Tomcat并初始化Spring环境及其各Spring组件。

2.4 Controller层测试类

一个好的程序，不能缺少好的UT。针对HelloWorldController的UT如下：

```
/**
 * Spring Boot HelloWorldController 测试 - {@link HelloWorldController}
 *
 * Created by bysocket on 16/4/26.
 */
public class HelloWorldControllerTest {

    @Test
    public void testSayHello() {
        assertEquals("Hello,World!",new HelloWorldController().sayHello());
    }
}
```

三、运行

Just Run的宗旨，运行很简单，直接右键Run运行Application类。同样你也可以Debug Run。可以在控制台中看到：

```
Tomcat started on port(s): 8080 (http)
Started Application in 5.986 seconds (JVM running for 7.398)
```

然后访问 http://localhost:8080/ ,即可在页面中看到Spring Boot对你 say hello：

```
Hello,World!
```

四、小结

1. Spring Boot pom配置 2. Spring Boot 启动及原理 3. 对应代码分享在 Github 主页

相信很多人选择Spring Boot主要是考虑到它既能兼顾Spring的强大功能，还能实现快速开发的便捷。我们在Spring Boot使用过程中，最直观的感受就是没有了原来自己整合Spring应用时繁多的XML配置内容，替代它的是在 `pom.xml` 中引入模块化的 `Starter POMs` ，其中各个模块都有自己的默认配置，所以如果不是特殊应用场景，就只需要在 `application.properties` 中完成一些属性配置就能开启各模块的应用。

在之前的各篇文章中都有提及关于 `application.properties` 的使用，主要用来配置数据库连接、日志相关配置等。除了这些配置内容之外，本文将具体介绍一些在 `application.properties` 配置中的其他特性和使用方法。

自定义属性与加载

我们在使用Spring Boot的时候，通常也需要定义一些自己使用的属性，我们可以如下方式直接定义：

```
com.didispace.blog.name=程序猿DD
com.didispace.blog.title=Spring Boot教程
```

然后通过 `@Value("${属性名}")` 注解来加载对应的配置属性，具体如下：

```
@Component
public class BlogProperties {

    @Value("${com.didispace.blog.name}")
    private String name;
    @Value("${com.didispace.blog.title}")
    private String title;

    // 省略getter和setter

}
```

按照惯例，通过单元测试来验证BlogProperties中的属性是否已经根据配置文件加载了。

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

    @Autowired
    private BlogProperties blogProperties;


    @Test
    public void getHello() throws Exception {
        Assert.assertEquals(blogProperties.getName(), "程序猿DD");
        Assert.assertEquals(blogProperties.getTitle(), "Spring Boot教程");
    }
```

```
}
```

参数间的引用

在 `application.properties` 中的各个参数之间也可以直接引用来使用，就像下面的设置：

```
com.didispace.blog.name=程序猿DD
com.didispace.blog.title=Spring Boot教程
com.didispace.blog.desc=${com.didispace.blog.name}正在努力写《${com.didispace.blog.title}》
```

`com.didispace.blog.desc` 参数引用了上文中定义的 `name` 和 `title` 属性，最后该属性的值就是 `程序猿DD正在努力写《Spring Boot教程》` 。

使用随机数

在一些情况下，有些参数我们需要希望它不是一个固定的值，比如密钥、服务端口等。Spring Boot的属性配置文件中可以通过 `${random}` 来产生int值、long值或者string字符串，来支持属性的随机值。

```
# 随机字符串
com.didispace.blog.value=${random.value}
# 随机int
com.didispace.blog.number=${random.int}
# 随机long
com.didispace.blog.bignumber=${random.long}
# 10以内的随机数
com.didispace.blog.test1=${random.int(10)}
# 10-20的随机数
com.didispace.blog.test2=${random.int[10,20]}
```

通过命令行设置属性值

相信使用过一段时间Spring Boot的用户，一定知道这条命令： `java -jar xxx.jar --server.port=8888` ，通过使用—server.port属性来设置xxx.jar应用的端口为8888。

在命令行运行时，连续的两个减号 `--` 就是对 `application.properties` 中的属性值进行赋值的标识。所以， `java -jar xxx.jar --server.port=8888` 命令，等价于我们在 `application.properties` 中添加属性 `server.port=8888` ，该设置在样例工程中可见，读者可通过删除该值或使用命令行来设置该值来验证。

通过命令行来修改属性值固然提供了不错的便利性，但是通过命令行就能更改应用运行的参数，那岂不是很不安全？是的，所以Spring Boot也贴心的提供了屏蔽命令行访问属性的设置，只需要这句设置就能屏蔽： `SpringApplication.setAddCommandLineProperties(false)` 。

多环境配置

我们在开发Spring Boot应用时，通常同一套程序会被应用和安装到几个不同的环境，比如：开发、测试、生产等。其中每个环境的数据库地址、服务器端口等等配置都会不同，如果在为不同环境打包时都要频繁修改配置文件的话，那必将是个非常繁琐且容易发生错误的事。

对于多环境的配置，各种项目构建工具或是框架的基本思路是一致的，通过配置多份不同环境的配置文件，再通过打包命令指定需要打包的内容之后进行区分打包，Spring Boot也不例外，或者说更加简单。

在Spring Boot中多环境配置文件名需要满足 `application-{profile}.properties` 的格式，其中 `{profile}` 对应你的环境标识，比如：

- `application-dev.properties` ：开发环境
- `application-test.properties` ：测试环境
- `application-prod.properties` ：生产环境

至于哪个具体的配置文件会被加载，需要在 `application.properties` 文件中通过 `spring.profiles.active` 属性来设置，其值对应 `{profile}` 值。

如： `spring.profiles.active=test` 就会加载 `application-test.properties` 配置文件内容

下面，以不同环境配置不同的服务端口为例，进行样例实验。

- 针对各环境新建不同的配置文件 `application-dev.properties` 、 `application-test.properties` 、 `application-prod.properties`
- 在这三个文件均都设置不同的 `server.port` 属性，如：dev环境设置为1111，test环境设置为2222，prod环境设置为3333
- application.properties中设置 `spring.profiles.active=dev` ，就是说默认以dev环境设置
- 测试不同配置的加载
  - 执行 `java -jar xxx.jar` ，可以观察到服务端口被设置为 `1111` ，也就是默认的开发环境（dev）
  - 执行 `java -jar xxx.jar --spring.profiles.active=test` ，可以观察到服务端口被设置为 `2222` ，也就是测试环境的配置（test）
  - 执行 `java -jar xxx.jar --spring.profiles.active=prod` ，可以观察到服务端口被设置为 `3333` ，也就是生产环境的配置（prod）

按照上面的实验，可以如下总结多环境的配置思路：

- `application.properties` 中配置通用内容，并设置 `spring.profiles.active=dev` ，以开发环境为默认配置
- `application-{profile}.properties` 中配置各个环境不同的内容
- 通过命令行方式去激活不同环境的配置

完整示例chapter2-1-1

http://git.oschina.net/didispace/SpringBoot-Learning


持续更新...
关注 社区 · 轻松学习
长按识别ME

摘要: 原创出处 www.bysocket.com 「泥瓦匠BYSocket」欢迎转载，保留摘要，谢谢！

『 仓廪实而知礼节，衣食足而知荣辱 - 管仲 』

本文提纲

一、自动配置

二、自定义属性

三、random.* 属性

四、多环境配置

运行环境：JDK 7 或 8，Maven 3.0+

技术栈：SpringBoot 1.5+

一、自动配置

Spring Boot 提供了对应用进行自动化配置。相比以前 XML 配置方式，很多显式方式申明是不需要的。二者，大多数默认的配置足够实现开发功能，从而更快速开发。

什么是**自动配置**?

Spring Boot 提供了默认的配置，如默认的 Bean ，去运行 Spring 应用。它是非侵入式的，只提供一个默认实现。

大多数情况下，自动配置的 Bean 满足了现有的业务场景，不需要去覆盖。但如果自动配置做的不够好，需要覆盖配置。比如通过命令行动态指定某个 jar ，按不同环境启动（这个例子在第 4 小节介绍）。那怎么办？这里先要考虑到配置的优先级。

Spring Boot 不单单从 application.properties 获取配置，所以我们可以在程序中多种设置配置属性。按照以下列表的优先级排列：

1.命令行参数

2.java:comp/env 里的 JNDI 属性

3.JVM 系统属性

4.操作系统环境变量

5.RandomValuePropertySource 属性类生成的 random.* 属性

6.应用以外的 application.properties（或 yml）文件

7.打包在应用内的 application.properties（或 yml）文件

8.在应用 @Configuration 配置类中，用 @PropertySource 注解声明的属性文件

9.SpringApplication.setDefaultProperties 声明的默认属性

可见，命令行参数优先级最高。这个可以根据这个优先级，可以在测试或生产环境中快速地修改配置参数值，而不需要重新打包和部署应用。

还有第 6 点，根据这个在多 moudle 的项目中，比如常见的项目分 api 、service、dao 等 moudles，往往会加一个 deploy moudle 去打包该业务各个子 moudle，应用以外的配置优先。

二、自定义属性

泥瓦匠喜欢按着代码工程来讲解知识。git clone 下载工程 springboot-learning-example ， 项目地址见 GitHub - https://github.com/JeffLi1993/springboot-learning-example。

a. 编译工程

在项目根目录 springboot-learning-example，运行 maven 指令：

```
cd springboot-learning-example
mvn clean install
```

b. 运行工程 test 方法

运行 springboot-properties 工程 org.spring.springboot.property.PropertiesTest 测试类的 getHomeProperties 方法。可以在控制台看到输出，这是通过自定义属性获取的值:

```
HomeProperties{province='ZheJiang', city='WenLing', desc='dev: I'm living in ZheJiang WenL
ing.'}
```

怎么定义自定义属性呢?

首先项目结构如下：

```
├── pom.xml
└── src
    ├── main
    │   ├── java
    │   │   └── org
    │   │       └── spring
    │   │           └── springboot
    │   │               ├── Application.java
    │   │               └── property
    │   │                   ├── HomeProperties.java
    │   │                   └── UserProperties.java
    │   └── resources
    │       ├── application-dev.properties
    │       ├── application-prod.properties
    │       └── application.properties
    └── test
        ├── java
        │   └── org
        │       └── spring
        │           └── springboot
        │               └── property
        │                   ├── HomeProperties1.java
```

```
        │                   └── PropertiesTest.java
        └── resouorces
            └── application.yml
```

在 application.properties 中对应 HomeProperties 对象字段编写属性的 KV 值：

```
## 家乡属性 Dev
home.province=ZheJiang
home.city=WenLing
home.desc=dev: I'm living in ${home.province} ${home.city}.
```

这里也可以通过占位符，进行属性之间的引用。

然后，编写对应的 HomeProperties Java 对象：

```
/**
 * 家乡属性
 *
 * Created by bysocket on 17/04/2017.
 */
@Component
@ConfigurationProperties(prefix = "home")
public class HomeProperties {

    /**
     * 省份
     */
    private String province;

    /**
     * 城市
     */
    private String city;

    /**
     * 描述
     */
    private String desc;

    public String getProvince() {
        return province;
    }

    public void setProvince(String province) {
        this.province = province;
    }

    public String getCity() {
        return city;
```

```
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getDesc() {
        return desc;
    }

    public void setDesc(String desc) {
        this.desc = desc;
    }

    @Override
    public String toString() {
        return "HomeProperties{" +
                "province='" + province + '\'' +
                ", city='" + city + '\'' +
                ", desc='" + desc + '\'' +
                '}';
    }
}
```

通过 @ConfigurationProperties(prefix = "home”) 注解，将配置文件中以 home 前缀的属性值自动绑定到对应的字段中。同是用 @Component 作为 Bean 注入到 Spring 容器中。

如果不是用 application.properties 文件，而是用 application.yml 的文件，对应配置如下：

```
## 家乡属性
home:
  province: 浙江省
  city: 温岭松门
  desc: 我家住在${home.province}的${home.city}
```

键值对冒号后面，必须空一格。

注意这里，就有一个坑了：

application.properties 配置中文值的时候，读取出来的属性值会出现乱码问题。但是 application.yml 不会出现乱码问题。原因是，Spring Boot 是以 iso-8859 的编码方式读取 application.properties 配置文件。

注意这里，还有一个坑：

如果定义一个键值对 user.name=xxx ,这里会读取不到对应写的属性值。为什么呢？Spring Boot 的默认 StandardEnvironment 首先将会加载 “systemEnvironment" 作为首个PropertySource. 而 source 即为System.getProperties().当 getProperty时,按照读取顺序,返回 “systemEnvironment" 的值.即 System.getProperty("user.name")

（Mac 机子会读自己的登录账号，这里感谢我的死党 http://rapharino.com/）

三、random.* 属性

Spring Boot 通过 RandomValuePropertySource 提供了很多关于随机数的工具类。概括可以生成随机字符串、随机 int 、随机 long、某范围的随机数。

运行 springboot-properties 工程 org.spring.springboot.property.PropertiesTest 测试类的 randomTestUser 方法。多次运行，可以发现每次输出不同 User 属性值：

```
UserProperties{id=-3135706105861091890, age=41, desc='泥瓦匠叫做3cf8fb2507f64e361f62700bcbd
17770', uuid='582bcc01-bb7f-41db-94d5-c22aae186cb4'}
```

application.yml 方式的配置如下（ application.properties 形式这里不写了）：

```
## 随机属性
user:
  id: ${random.long}
  age: ${random.int[1,200]}
  desc: 泥瓦匠叫做${random.value}
  uuid: ${random.uuid}
```

四、多环境配置

很多场景的配置，比如数据库配置、Redis 配置、注册中心和日志配置等。在不同的环境，我们需要不同的包去运行项目。所以看项目结构，有两个环境的配置：

```
application-dev.properties：开发环境
application-prod.properties：生产环境
```

Spring Boot 是通过 application.properties 文件中，设置 spring.profiles.active 属性，比如 ，配置了 dev ,则加载的是 application-dev.properties ：

```
# Spring Profiles Active
spring.profiles.active=dev
```

那运行 springboot-properties 工程中 Application 应用启动类，从控制台中可以看出，是加载了 application-dev.properties 的属性输出：

```
HomeProperties{province='ZheJiang', city='WenLing', desc='dev: I'm living in ZheJiang WenL
ing.'}
```

将 spring.profiles.active 设置成 prod，重新运行，可得到 application-prod.properties的属性输出：

```
HomeProperties{province='ZheJiang', city='WenLing', desc='prod: I'm living in ZheJiang Wen
```

```
Ling.'}
```

根据优先级，顺便介绍下 jar 运行的方式，通过设置 -Dspring.profiles.active=prod 去指定相应的配置:

```
mvn package
java -jar -Dspring.profiles.active=prod springboot-properties-0.0.1-SNAPSHOT.jar
```

五、小结

常用的样板配置在 Spring Boot 官方文档给出，我们常在 application.properties（或 yml）去配置各种常用配置：

http://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html

感谢资料：

http://blog.didispace.com/springbootproperties/

https://docs.spring.io/spring-boot/docs

首先，回顾并详细说明一下在快速入门中使用的 `@Controller` 、 `@RestController` 、 `@RequestMapping` 注解。如果您对Spring MVC不熟悉并且还没有尝试过快速入门案例，建议先看一下快速入门的内容。

- `@Controller` ： 修饰class，用来创建处理http请求的对象
- `@RestController` ： Spring4之后加入的注解，原来在 `@Controller` 中返回json需要 `@ResponseBody` 来配合，如果直接用 `@RestController` 替代 `@Controller` 就不需要再配置 `@ResponseBody` ，默认返回json格式。
- `@RequestMapping` ： 配置url映射

下面我们尝试使用Spring MVC来实现一组对User对象操作的RESTful API，配合注释详细说明在Spring MVC中如何映射HTTP请求、如何传参、如何编写单元测试。

RESTful API具体设计如下：

请求类型	URL	功能说明
GET	/users	查询用户列表
POST	/users	创建一个用户
GET	/users/id	根据id查询一个用户
PUT	/users/id	根据id更新一个用户
DELETE	/users/id	根据id删除一个用户

User实体定义：

```
public class User {

    private Long id;
    private String name;
    private Integer age;
```

```
    // 省略setter和getter

}
```

实现对User对象的操作接口

```
@RestController
@RequestMapping(value="/users")     // 通过这里配置使下面的映射都在/users下
public class UserController {

    // 创建线程安全的Map
    static Map<Long, User> users = Collections.synchronizedMap(new HashMap<Long, User>());


    @RequestMapping(value="/", method=RequestMethod.GET)
    public List<User> getUserList() {
        // 处理"/users/"的GET请求，用来获取用户列表
        // 还可以通过@RequestParam从页面中传递参数来进行查询条件或者翻页信息的传递
        List<User> r = new ArrayList<User>(users.values());
        return r;
    }

    @RequestMapping(value="/", method=RequestMethod.POST)
    public String postUser(@ModelAttribute User user) {
        // 处理"/users/"的POST请求，用来创建User
        // 除了@ModelAttribute绑定参数之外，还可以通过@RequestParam从页面中传递参数
        users.put(user.getId(), user);
        return "success";
    }

    @RequestMapping(value="/{id}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long id) {
        // 处理"/users/{id}"的GET请求，用来获取url中id值的User信息
        // url中的id可通过@PathVariable绑定到函数的参数中
        return users.get(id);
    }

    @RequestMapping(value="/{id}", method=RequestMethod.PUT)
    public String putUser(@PathVariable Long id, @ModelAttribute User user) {
        // 处理"/users/{id}"的PUT请求，用来更新User信息
        User u = users.get(id);
        u.setName(user.getName());
        u.setAge(user.getAge());
        users.put(id, u);
        return "success";
    }

    @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
    public String deleteUser(@PathVariable Long id) {
        // 处理"/users/{id}"的DELETE请求，用来删除User
```

```
        users.remove(id);
        return "success";
    }

}
```

下面针对该Controller编写测试用例验证正确性，具体如下。当然也可以通过浏览器插件等进行请求提交验证。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MockServletContext.class)
@WebAppConfiguration
public class ApplicationTests {

    private MockMvc mvc;

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

    @Test
    public void testUserController() throws Exception {
        // 测试UserController
        RequestBuilder request = null;

        // 1、get查一下user列表，应该为空
        request = get("/users/");
        mvc.perform(request)
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("[]")));

        // 2、post提交一个user
        request = post("/users/")
                .param("id", "1")
                .param("name", "测试大师")
                .param("age", "20");
        mvc.perform(request)
                .andExpect(content().string(equalTo("success")));

        // 3、get获取user列表，应该有刚才插入的数据
        request = get("/users/");
        mvc.perform(request)
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("[{\"id\":1,\"name\":\"测试大师\",\"age
\":20}]")));

        // 4、put修改id为1的user
        request = put("/users/1")
                .param("name", "测试终极大师")
```

```
                .param("age", "30");
        mvc.perform(request)
                .andExpect(content().string(equalTo("success")));

        // 5、get一个id为1的user
        request = get("/users/1");
        mvc.perform(request)
                .andExpect(content().string(equalTo("{\"id\":1,\"name\":\"测试终极大师\",\"a
ge\":30}")));

        // 6、del删除id为1的user
        request = delete("/users/1");
        mvc.perform(request)
                .andExpect(content().string(equalTo("success")));

        // 7、get查一下user列表，应该为空
        request = get("/users/");
        mvc.perform(request)
                .andExpect(status().isOk())
                .andExpect(content().string(equalTo("[]")));

    }

}
```

至此，我们通过引入web模块（没有做其他的任何配置），就可以轻松利用Spring MVC的功能，以非常简洁的代码完成了对User对象的RESTful API的创建以及单元测试的编写。其中同时介绍了Spring MVC中最为常用的几个核心注解：`@Controller`，`@RestController`，`RequestMapping`以及一些参数绑定的注解：`@PathVariable`，`@ModelAttribute`，`@RequestParam`等。

完整案例

http://git.oschina.net/didispace/SpringBoot-Learning


持续更新...
关注 社区 · 轻松学习
长按识别ME

摘要: 原创出处:www.bysocket.com 泥瓦匠BYSocket 希望转载，保留摘要，谢谢!

> “怎样的人生才是没有遗憾的人生？我的体会是：
>
> （1）拥有健康；
>
> （2）创造“难忘时刻”；
>
> （3）尽力做好自己，不必改变世界；
>
> （4）活在当下。”
>
> -《向死而生》李开复

Spring Boot 系列文章：《Spring Boot 那些事》

基于上一篇《Springboot 整合 Mybatis 的完整 Web 案例》，这边我们着重在 控制层 讲讲。讲讲如何在 Springboot 实现 Restful 服务，基于 HTTP / JSON 传输。

一、运行 springboot-restful 工程

git clone 下载工程 springboot-learning-example ，项目地址见 GitHub - https://github.com/JeffLi1993/springboot-learning-example。下面开始运行工程步骤（Quick Start）：

1.数据库准备

a.创建数据库 springbootdb：

```
CREATE DATABASE springbootdb;
```

b.创建表 city：(因为我喜欢徒步)

```
DROP TABLE IF EXISTS  `city`;
CREATE TABLE `city` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '城市编号',
  `province_id` int(10) unsigned  NOT NULL COMMENT '省份编号',
  `city_name` varchar(25) DEFAULT NULL COMMENT '城市名称',
  `description` varchar(25) DEFAULT NULL COMMENT '描述',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

c.插入数据

```
INSERT city VALUES (1 ,1,'温岭市','BYSocket 的家在温岭。');
```

2. springboot-restful 工程项目结构介绍

> springboot-restful 工程项目结构如下图所示：
>
> org.spring.springboot.controller - Controller 层

> org.spring.springboot.dao - 数据操作层 DAO
>
> org.spring.springboot.domain - 实体类
>
> org.spring.springboot.service - 业务逻辑层
>
> Application - 应用启动类
>
> application.properties - 应用配置文件，应用启动会自动读取配置

3.改数据库配置

打开 application.properties 文件， 修改相应的数据源配置，比如数据源地址、账号、密码等。（如果不是用 MySQL，自行添加连接驱动 pom，然后修改驱动名配置。）

4.编译工程

在项目根目录 springboot-learning-example，运行 maven 指令：

mvn clean install

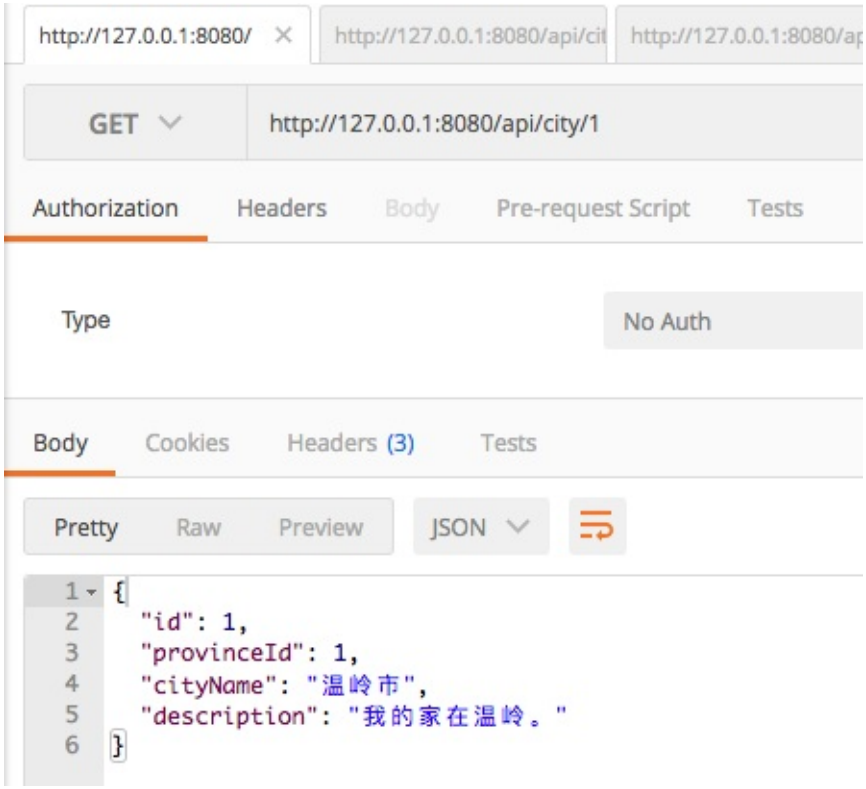
5.运行工程

右键运行 springboot-restful 工程 Application 应用启动类的 main 函数。

用 postman 工具可以如下操作，

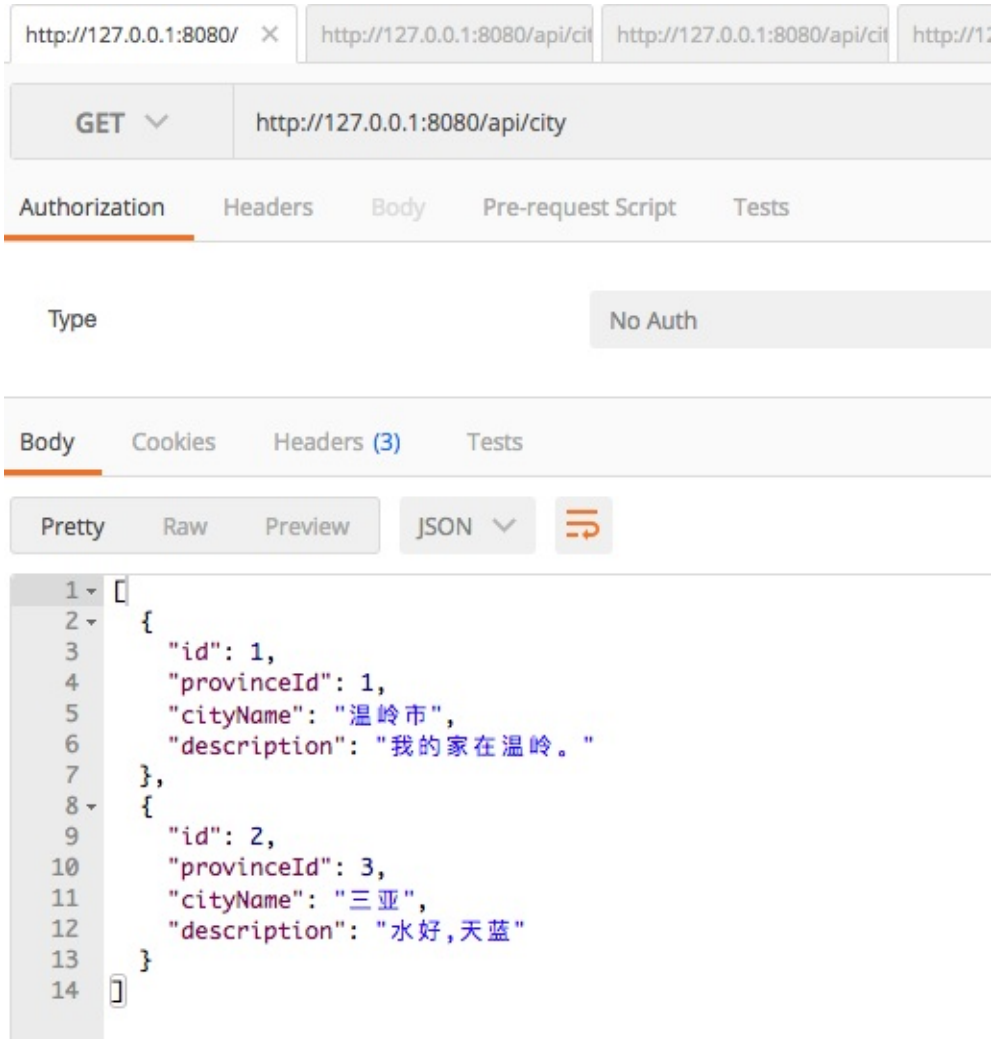
根据 ID，获取城市信息

GET http://127.0.0.1:8080/api/city/1



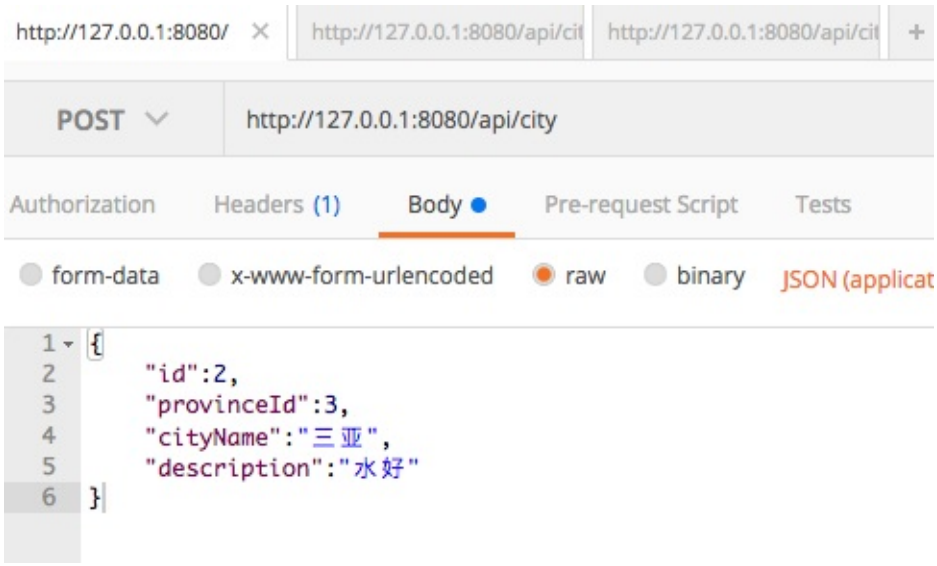
获取城市列表

GET http://127.0.0.1:8080/api/city



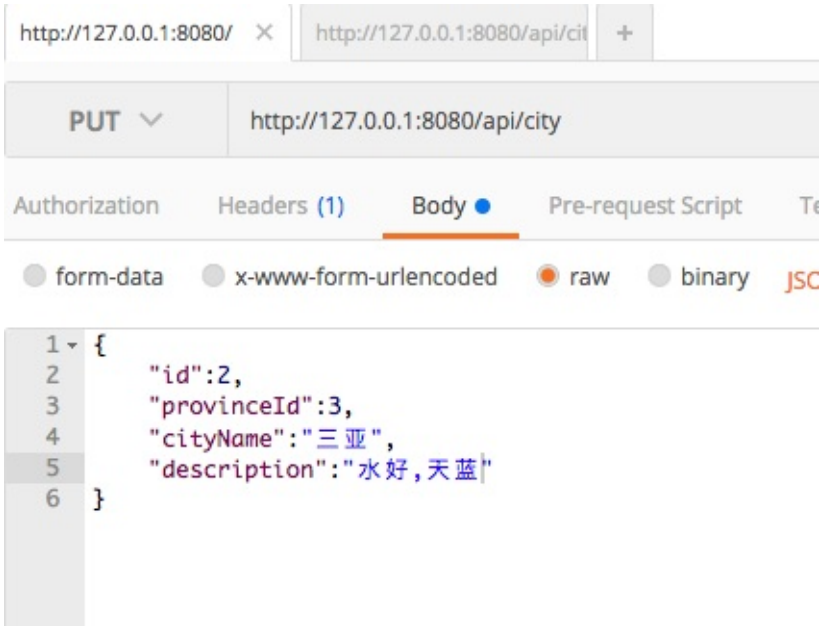
新增城市信息

POST http://127.0.0.1:8080/api/city



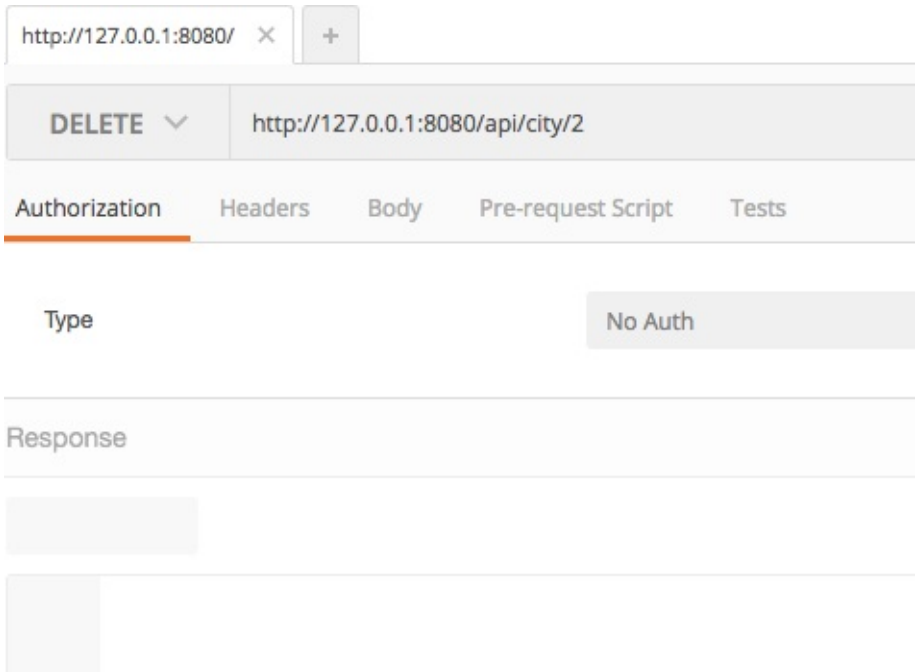
更新城市信息

PUT http://127.0.0.1:8080/api/city



删除城市信息

DELETE http://127.0.0.1:8080/api/city/2



二、springboot-restful 工程控制层实现详解

1.什么是 REST?

REST 是属于 WEB 自身的一种架构风格，是在 HTTP 1.1 规范下实现的。Representational State Transfer 全称翻译为表现层状态转化。Resource：资源。比如 newsfeed；Representational：表现形式，比如用JSON，富文本等；State Transfer：状态变化。通过HTTP 动作实现。

理解 REST ,要明白五个关键要素：

> 资源（Resource）
>
> 资源的表述（Representation）
>
> 状态转移（State Transfer）
>
> 统一接口（Uniform Interface）
>
> 超文本驱动（Hypertext Driven）

6 个主要特性：

> 面向资源（Resource Oriented）
>
> 可寻址（Addressability）
>
> 连通性（Connectedness）
>
> 无状态（Statelessness）
>
> 统一接口（Uniform Interface）
>
> 超文本驱动（Hypertext Driven）

具体这里就不一一展开，详见 http://www.infoq.com/cn/articles/understanding-restful-style

2.Spring 对 REST 支持实现

CityRestController.java 城市 Controller 实现 Restful HTTP 服务

```
public class CityRestController {

    @Autowired
    private CityService cityService;

    @RequestMapping(value = "/api/city/{id}", method = RequestMethod.GET)
    public City findOneCity(@PathVariable("id") Long id) {
        return cityService.findCityById(id);
    }

    @RequestMapping(value = "/api/city", method = RequestMethod.GET)
    public List<City> findAllCity() {
        return cityService.findAllCity();
    }

    @RequestMapping(value = "/api/city", method = RequestMethod.POST)
    public void createCity(@RequestBody City city) {
        cityService.saveCity(city);
    }

    @RequestMapping(value = "/api/city", method = RequestMethod.PUT)
    public void modifyCity(@RequestBody City city) {
```

```
        cityService.updateCity(city);
    }

    @RequestMapping(value = "/api/city/{id}", method = RequestMethod.DELETE)
    public void modifyCity(@PathVariable("id") Long id) {
        cityService.deleteCity(id);
    }
}
```

具体 Service 、dao 层实现看代码GitHub https://github.com/JeffLi1993/springboot-learning-example/tree/master/springboot-restful

代码详解：

@RequestMapping 处理请求地址映射。

> method - 指定请求的方法类型：POST/GET/DELETE/PUT 等
>
> value - 指定实际的请求地址
>
> consumes - 指定处理请求的提交内容类型，例如 Content-Type 头部设置application/json, text/html
>
> produces - 指定返回的内容类型

@PathVariable URL 映射时，用于绑定请求参数到方法参数

@RequestBody 这里注解用于读取请求体 boy 的数据，通过 HttpMessageConverter 解析绑定到对象中

3.HTTP 知识补充

> GET 请求获取Request-URI所标识的资源
>
> POST 在Request-URI所标识的资源后附加新的数据
>
> HEAD 请求获取由Request-URI所标识的资源的响应消息报头
>
> PUT 请求服务器存储一个资源，并用Request-URI作为其标识
>
> DELETE 请求服务器删除Request-URI所标识的资源
>
> TRACE 请求服务器回送收到的请求信息，主要用于测试或诊断
>
> CONNECT 保留将来使用
>
> OPTIONS 请求查询服务器的性能，或者查询与资源相关的选项和需求

详情请看《JavaEE 要懂的小事：一、图解Http协议》

三、小结

Springboot 实现 Restful 服务，基于 HTTP / JSON 传输，适用于前后端分离。这只是个小demo，没有加入bean validation这种校验。还有各种业务场景。

推荐：《 Springboot 整合 Mybatis 的完整 Web 案例》

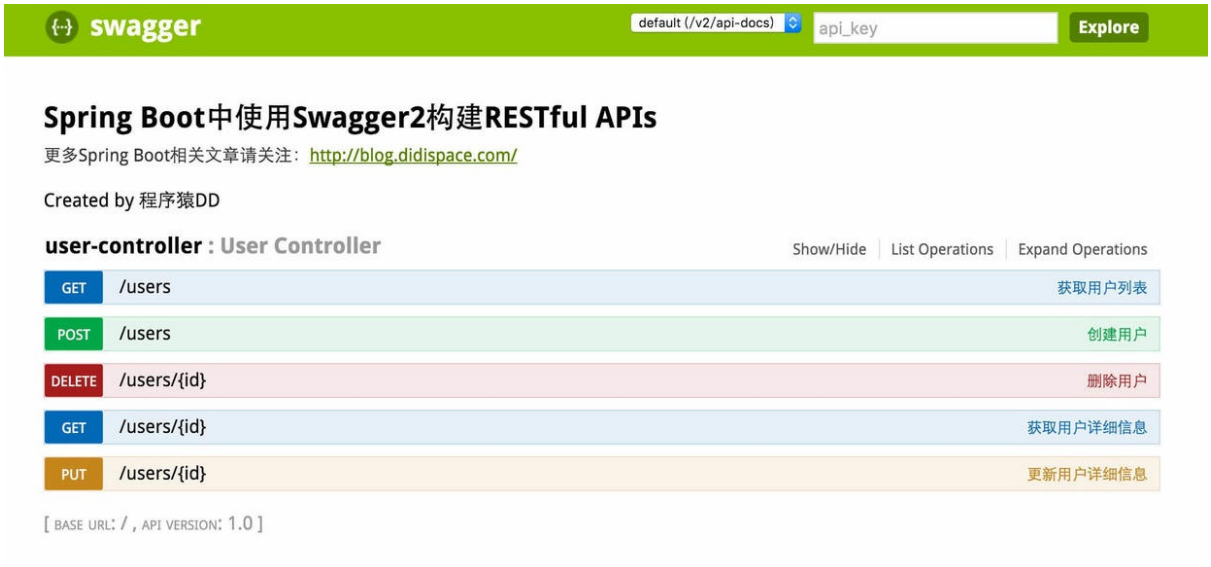
http://www.bysocket.com/?p=1610

由于Spring Boot能够快速开发、便捷部署等特性，相信有很大一部分Spring Boot的用户会用来构建RESTful API。而我们构建RESTful API的目的通常都是由于多终端的原因，这些终端会共用很多底层业务逻辑，因此我们会抽象出这样一层来同时服务于多个移动端或者Web前端。

这样一来，我们的RESTful API就有可能要面对多个开发人员或多个开发团队：IOS开发、Android开发或是Web开发等。为了减少与其他团队平时开发期间的频繁沟通成本，传统做法我们会创建一份RESTful API文档来记录所有接口细节，然而这样的做法有以下几个问题：

- 由于接口众多，并且细节复杂（需要考虑不同的HTTP请求类型、HTTP头部信息、HTTP请求内容等），高质量地创建这份文档本身就是件非常吃力的事，下游的抱怨声不绝于耳。
- 随着时间推移，不断修改接口实现的时候都必须同步修改接口文档，而文档与代码又处于两个不同的媒介，除非有严格的管理机制，不然很容易导致不一致现象。

为了解决上面这样的问题，本文将介绍RESTful API的重磅好伙伴Swagger2，它可以轻松的整合到Spring Boot中，并与Spring MVC程序配合组织出强大RESTful API文档。它既可以减少我们创建文档的工作量，同时说明内容又整合入实现代码中，让维护文档和修改代码整合为一体，可以让我们在修改代码逻辑的同时方便的修改文档说明。另外Swagger2也提供了强大的页面测试功能来调试每个RESTful API。具体效果如下图所示：



下面来具体介绍，如果在Spring Boot中使用Swagger2。首先，我们需要一个Spring Boot实现的RESTful API工程，若您没有做过这类内容，建议先阅读 Spring Boot构建一个较为完成的RESTful APIs和单元测试。

下面的内容我们会以教程样例中的Chapter3-1-1进行下面的实验（Chpater3-1-5是我们的结果工程，亦可参考）。

添加Swagger2依赖

在 `pom.xml` 中加入Swagger2的依赖

```
<dependency>
    <groupId>io.springfox</groupId>
```

```
    <artifactId>springfox-swagger2</artifactId>
    <version>2.2.2</version>
</dependency>
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.2.2</version>
</dependency>
```

创建Swagger2配置类

在 `Application.java` 同级创建Swagger2的配置类 `Swagger2` 。

```
@Configuration
@EnableSwagger2
public class Swagger2 {

    @Bean
    public Docket createRestApi() {
        return new Docket(DocumentationType.SWAGGER_2)
                .apiInfo(apiInfo())
                .select()
                .apis(RequestHandlerSelectors.basePackage("com.didispace.web"))
                .paths(PathSelectors.any())
                .build();
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder()
                .title("Spring Boot中使用Swagger2构建RESTful APIs")
                .description("更多Spring Boot相关文章请关注：http://blog.didispace.com/")
                .termsOfServiceUrl("http://blog.didispace.com/")
                .contact("程序猿DD")
                .version("1.0")
                .build();
    }

}
```

如上代码所示，通过 `@Configuration` 注解，让Spring来加载该类配置。再通过 `@EnableSwagger2` 注解来启用Swagger2。

再通过 `createRestApi` 函数创建 `Docket` 的Bean之后， `apiInfo()` 用来创建该Api的基本信息（这些基本信息会展现在文档页面中）。 `select()` 函数返回一个 `ApiSelectorBuilder` 实例用来控制哪些接口暴露给Swagger来展现，本例采用指定扫描的包路径来定义，Swagger会扫描该包下所有Controller定义的API，并产生文档内容（除了被 `@ApiIgnore` 指定的请求）。

添加文档内容

在完成了上述配置后，其实已经可以生产文档内容，但是这样的文档主要针对请求本身，而描述主要来源于函数等命名产生，对用户并不友好，我们通常需要自己增加一些说明来丰富文档内容。如下所示，我们通过 `@ApiOperation` 注解来给API增加说明、通过 `@ApiImplicitParams` 、 `@ApiImplicitParam` 注解来给参数增加说明。

```
@RestController
@RequestMapping(value="/users")     // 通过这里配置使下面的映射都在/users下，可去除
public class UserController {

    static Map<Long, User> users = Collections.synchronizedMap(new HashMap<Long, User>());

    @ApiOperation(value="获取用户列表", notes="")
    @RequestMapping(value={""}, method=RequestMethod.GET)
    public List<User> getUserList() {
        List<User> r = new ArrayList<User>(users.values());
        return r;
    }

    @ApiOperation(value="创建用户", notes="根据User对象创建用户")
    @ApiImplicitParam(name = "user", value = "用户详细实体user", required = true, dataType = "User")
    @RequestMapping(value="", method=RequestMethod.POST)
    public String postUser(@RequestBody User user) {
        users.put(user.getId(), user);
        return "success";
    }

    @ApiOperation(value="获取用户详细信息", notes="根据url的id来获取用户详细信息")
    @ApiImplicitParam(name = "id", value = "用户ID", required = true, dataType = "Long")
    @RequestMapping(value="/{id}", method=RequestMethod.GET)
    public User getUser(@PathVariable Long id) {
        return users.get(id);
    }

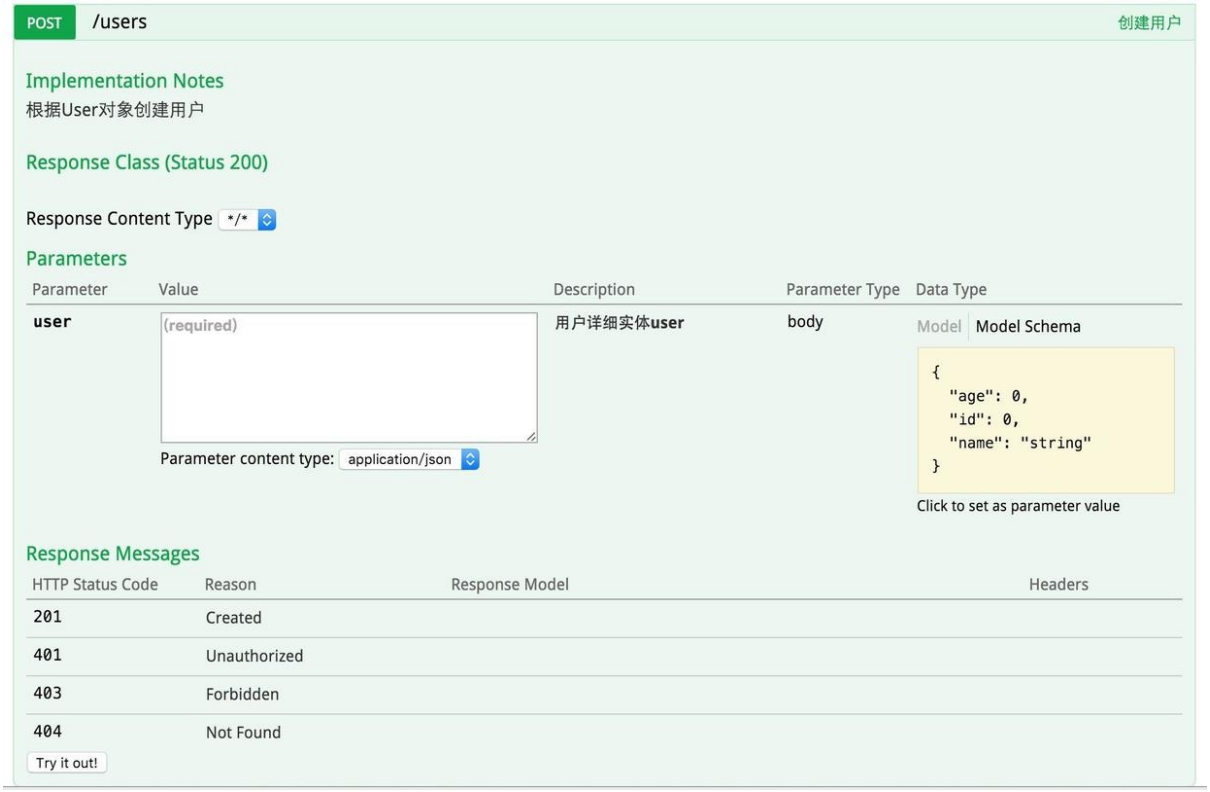
    @ApiOperation(value="更新用户详细信息", notes="根据url的id来指定更新对象，并根据传过来的user信息来更新用户详细信息")
    @ApiImplicitParams({
            @ApiImplicitParam(name = "id", value = "用户ID", required = true, dataType = "Long"),
            @ApiImplicitParam(name = "user", value = "用户详细实体user", required = true, dataType = "User")
    })
    @RequestMapping(value="/{id}", method=RequestMethod.PUT)
    public String putUser(@PathVariable Long id, @RequestBody User user) {
        User u = users.get(id);
        u.setName(user.getName());
        u.setAge(user.getAge());
```

```
        users.put(id, u);
        return "success";
    }

    @ApiOperation(value="删除用户", notes="根据url的id来指定删除对象")
    @ApiImplicitParam(name = "id", value = "用户ID", required = true, dataType = "Long")
    @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
    public String deleteUser(@PathVariable Long id) {
        users.remove(id);
        return "success";
    }

}
```

完成上述代码添加上，启动Spring Boot程序，访问：http://localhost:8080/swagger-ui.html 。就能看到前文所展示的RESTful API的页面。我们可以再点开具体的API请求，以POST类型的/users请求为例，可找到上述代码中我们配置的Notes信息以及参数user的描述信息，如下图所示。



API文档访问与调试

在上图请求的页面中，我们看到user的Value是个输入框？是的，Swagger除了查看接口功能外，还提供了调试测试功能，我们可以点击上图中右侧的Model Schema（黄色区域：它指明了User的数据结构），此时Value中就有了user对象的模板，我们只需要稍适修改，点击下方“Try it out!”按钮，即可完成了一次请求调用！

此时，你也可以通过几个GET请求来验证之前的POST请求是否正确。

相比为这些接口编写文档的工作，我们增加的配置内容是非常少而且精简的，对于原有代码的侵入也在忍受范围之内。因此，在构建RESTful API的同时，加入swagger来对API文档进行管理，是个不错的选择。

完整结果示例可查看Chapter3-1-5。

参考信息

- Swagger官方网站

摘要: 原创出处:www.bysocket.com 泥瓦匠BYSocket 希望转载，保留摘要，谢谢!

> “年轻就不应该让自己过得太舒服” - From yong

一、Springboot 那些事

SpringBoot 很方便的集成 FreeMarker ， DAO 数据库操作层依旧用的是 Mybatis，本文将会一步一步到来如何集成 FreeMarker 以及配置的详解：

Springboot 那些事：

系类文章：

> 《Spring Boot 之 RESRful API 权限控制》
>
> 《Spring Boot 之 HelloWorld详解》
>
> 《Springboot 整合 Mybatis 的完整 Web 案例》
>
> 《Springboot 实现 Restful 服务，基于 HTTP / JSON 传输》
>
> 《Springboot 集成 FreeMarker》

二、运行 springboot-freemarker 工程

git clone 下载工程 springboot-learning-example ， 项目地址见 GitHub - https://github.com/JeffLi1993/springboot-learning-example。下面开始运行工程步骤（Quick Start）：

1.数据库准备

a.创建数据库 springbootdb：

```
CREATE DATABASE springbootdb;
```

b.创建表 city： (因为我喜欢徒步)

```
DROP TABLE IF EXISTS  `city`;
CREATE TABLE `city` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '城市编号',
  `province_id` int(10) unsigned  NOT NULL COMMENT '省份编号',
  `city_name` varchar(25) DEFAULT NULL COMMENT '城市名称',
  `description` varchar(25) DEFAULT NULL COMMENT '描述',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

c.插入数据

```
INSERT city VALUES (1 ,1,'温岭市','BYSocket 的家在温岭。');
```

2. 项目结构介绍

项目结构如下图所示：

> org.spring.springboot.controller - Controller 层
>
> org.spring.springboot.dao - 数据操作层 DAO
>
> org.spring.springboot.domain - 实体类
>
> org.spring.springboot.service - 业务逻辑层
>
> Application - 应用启动类
>
> resources/application.properties - 应用配置文件，应用启动会自动读取配置
>
> resources/web - *.ftl文件，是 FreeMarker 文件配置路径。在 application.properties 配置
>
> resources/mapper - DAO Maper XML 文件

3.改数据库配置

打开 application.properties 文件， 修改相应的数据源配置，比如数据源地址、账号、密码等。（如果不是用 MySQL，pom 自行添加连接驱动依赖，然后修改驱动名配置。）

4.编译工程

在项目根目录 springboot-learning-example，运行 maven 指令：

mvn clean install

5.运行工程

右键运行 springboot-freemarker 工程 Application 应用启动类的 main 函数，然后在浏览器访问：

获取 ID 编号为 1 的城市信息页面：

```
localhost:8080/api/city/1
```

获取城市列表页面：

```
localhost:8080/api/city
```

6.补充

运行环境：JDK 7 或 8，Maven 3.0+

技术栈：SpringBoot、Mybatis、FreeMarker

三、 springboot-freemarker 工程配置详解

具体代码见 GitHub - https://github.com/JeffLi1993/springboot-learning-example

1.pom.xml 依赖

pom.xml 代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>springboot</groupId>
    <artifactId>springboot-freemarker</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-freemarker :: Spring Boot 集成 FreeMarker 案例</name>

    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>

    <properties>
        <mybatis-spring-boot>1.2.0</mybatis-spring-boot>
        <mysql-connector>5.1.39</mysql-connector>
    </properties>

    <dependencies>
        <!-- Spring Boot Freemarker 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-freemarker</artifactId>
        </dependency>

        <!-- Spring Boot Web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Spring Boot Test 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Spring Boot Mybatis 依赖 -->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
```

```
            <version>${mybatis-spring-boot}</version>
        </dependency>

        <!-- MySQL 连接驱动依赖 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql-connector}</version>
        </dependency>

        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

在 pom.xml 依赖中增加 Spring Boot FreeMarker 依赖。

2.配置 FreeMarker

然后在 application.properties 中加入 FreeMarker 相关的配置：

```
## Freemarker 配置
## 文件配置路径
spring.freemarker.template-loader-path=classpath:/web/
spring.freemarker.cache=false
spring.freemarker.charset=UTF-8
spring.freemarker.check-template-location=true
spring.freemarker.content-type=text/html
spring.freemarker.expose-request-attributes=true
spring.freemarker.expose-session-attributes=true
spring.freemarker.request-context-attribute=request
spring.freemarker.suffix=.ftl
```

这是我这块的配置，如果需要更多的 FreeMarker 配置，可以查看下面的详解：

```
spring.freemarker.allow-request-override=false # Set whether HttpServletRequest attributes
 are allowed to override (hide) controller generated model attributes of the same name.
spring.freemarker.allow-session-override=false # Set whether HttpSession attributes are al
lowed to override (hide) controller generated model attributes of the same name.
spring.freemarker.cache=false # Enable template caching.
spring.freemarker.charset=UTF-8 # Template encoding.
spring.freemarker.check-template-location=true # Check that the templates location exists.
spring.freemarker.content-type=text/html # Content-Type value.
spring.freemarker.enabled=true # Enable MVC view resolution for this technology.
spring.freemarker.expose-request-attributes=false # Set whether all request attributes sho
uld be added to the model prior to merging with the template.
```

```
spring.freemarker.expose-session-attributes=false # Set whether all HttpSession attributes
 should be added to the model prior to merging with the template.
spring.freemarker.expose-spring-macro-helpers=true # Set whether to expose a RequestContex
t for use by Spring's macro library, under the name "springMacroRequestContext".
spring.freemarker.prefer-file-system-access=true # Prefer file system access for template
loading. File system access enables hot detection of template changes.
spring.freemarker.prefix= # Prefix that gets prepended to view names when building a URL.
spring.freemarker.request-context-attribute= # Name of the RequestContext attribute for al
l views.
spring.freemarker.settings.*= # Well-known FreeMarker keys which will be passed to FreeMar
ker's Configuration.
spring.freemarker.suffix= # Suffix that gets appended to view names when building a URL.
spring.freemarker.template-loader-path=classpath:/templates/ # Comma-separated list of tem
plate paths.
spring.freemarker.view-names= # White list of view names that can be resolved.
```

3.展示层 Controller 详解

```
/**
 * 城市 Controller 实现 Restful HTTP 服务
 * <p>
 * Created by bysocket on 07/02/2017.
 */
@Controller
public class CityController {

    @Autowired
    private CityService cityService;

    @RequestMapping(value = "/api/city/{id}", method = RequestMethod.GET)
    public String findOneCity(Model model, @PathVariable("id") Long id) {
        model.addAttribute("city", cityService.findCityById(id));
        return "city";
    }

    @RequestMapping(value = "/api/city", method = RequestMethod.GET)
    public String findAllCity(Model model) {
        List<City> cityList = cityService.findAllCity();
        model.addAttribute("cityList",cityList);
        return "cityList";
    }
}
```

a.这里不是走 HTTP + JSON 模式，使用了 @Controller 而不是先前的 @RestController

b.方法返回值是 String 类型，和 application.properties 配置的 Freemarker 文件配置路径下的各个 *.ftl 文件名一致。这样才会准确地把数据渲染到 ftl 文件里面进行展示。

c.用 Model 类，向 Model 加入数据，并指定在该数据在 Freemarker 取值指定的名称。

四、小结

FreeMarker 是常用的模板引擎，很多开发 Web 的必选。

推荐阅读《Springboot 那些事》

在上一篇Spring中使用JdbcTemplate访问数据库 中介绍了一种基本的数据访问方式，结合构建RESTful API和使用Thymeleaf模板引擎渲染Web视图的内容就已经可以完成App服务端和Web站点的开发任务了。

然而，在实际开发过程中，对数据库的操作无非就“增删改查”。就最为普遍的单表操作而言，除了表和字段不同外，语句都是类似的，开发人员需要写大量类似而枯燥的语句来完成业务逻辑。

为了解决这些大量枯燥的数据操作语句，我们第一个想到的是使用ORM框架，比如：Hibernate。通过整合Hibernate之后，我们以操作Java实体的方式最终将数据改变映射到数据库表中。

为了解决抽象各个Java实体基本的“增删改查”操作，我们通常会以泛型的方式封装一个模板Dao来进行抽象简化，但是这样依然不是很方便，我们需要针对每个实体编写一个继承自泛型模板Dao的接口，再编写该接口的实现。虽然一些基础的数据访问已经可以得到很好的复用，但是在代码结构上针对每个实体都会有一堆Dao的接口和实现。

由于模板Dao的实现，使得这些具体实体的Dao层已经变的非常“薄”，有一些具体实体的Dao实现可能完全就是对模板Dao的简单代理，并且往往这样的实现类可能会出现在很多实体上。Spring-data-jpa的出现正可以让这样一个已经很“薄”的数据访问层变成只是一层接口的编写方式。比如，下面的例子：

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

    User findByName(String name);

    @Query("from User u where u.name=:name")
    User findUser(@Param("name") String name);

}
```

我们只需要通过编写一个继承自 `JpaRepository` 的接口就能完成数据访问，下面以一个具体实例来体验Spring-data-jpa给我们带来的强大功能。

使用示例

由于Spring-data-jpa依赖于Hibernate。如果您对Hibernate有一定了解，下面内容可以毫不费力的看懂并上手使用Spring-data-jpa。如果您还是Hibernate新手，您可以先按如下方式入门，再建议回头学习一下Hibernate以帮助这部分的理解和进一步使用。

工程配置

在 `pom.xml` 中添加相关依赖，加入以下内容：

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

在 `application.xml` 中配置：数据库连接信息（如使用嵌入式数据库则不需要）、自动创建表结构的设置，例如使用mysql的情况如下：

```
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=create-drop
```

`spring.jpa.properties.hibernate.hbm2ddl.auto` 是hibernate的配置属性，其主要作用是：自动创建、更新、验证数据库表结构。该参数的几种配置如下：

- `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。
- `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。
- `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。
- `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。

至此已经完成基础配置，如果您有在Spring下整合使用过它的话，相信你已经感受到Spring Boot的便利之处：JPA的传统配置在 `persistence.xml` 文件中，但是这里我们不需要。当然，最好在构建项目时候按照之前提过的最佳实践的工程结构来组织，这样以确保各种配置都能被框架扫描到。

创建实体

创建一个User实体，包含id（主键）、name（姓名）、age（年龄）属性，通过ORM框架其会被映射到数据库表中，由于配置了 `hibernate.hbm2ddl.auto`，在应用启动的时候框架会自动去数据库中创建对应的表。

```
@Entity
public class User {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private Integer age;
```

```
    // 省略构造函数

    // 省略getter和setter

}
```

创建数据访问接口

下面针对User实体创建对应的 `Repository` 接口实现对该实体的数据访问，如下代码：

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

    User findByName(String name);

    User findByNameAndAge(String name, Integer age);

    @Query("from User u where u.name=:name")
    User findUser(@Param("name") String name);

}
```

在Spring-data-jpa中，只需要编写类似上面这样的接口就可实现数据访问。不再像我们以往编写了接口时候还需要自己编写接口实现类，直接减少了我们的文件清单。

下面对上面的 `UserRepository` 做一些解释，该接口继承自 `JpaRepository` ，通过查看 `JpaRepository` 接口的API文档，可以看到该接口本身已经实现了创建（save）、更新（save）、删除（delete）、查询（findAll、findOne）等基本操作的函数，因此对于这些基础操作的数据访问就不需要开发者再自己定义。

在我们实际开发中， `JpaRepository` 接口定义的接口往往还不够或者性能不够优化，我们需要进一步实现更复杂一些的查询或操作。由于本文重点在spring boot中整合spring-data-jpa，在这里先抛砖引玉简单介绍一下spring-data-jpa中让我们兴奋的功能，后续再单独开篇讲一下spring-data-jpa中的常见使用。

在上例中，我们可以看到下面两个函数：

- `User findByName(String name)`
- `User findByNameAndAge(String name, Integer age)`

它们分别实现了按name查询User实体和按name和age查询User实体，可以看到我们这里没有任何类SQL语句就完成了两个条件查询方法。这就是Spring-data-jpa的一大特性：**通过解析方法名创建查询**。

除了通过解析方法名来创建查询外，它也提供通过使用@Query 注解来创建查询，您只需要编写JPQL语句，并通过类似“:name”来映射@Param指定的参数，就像例子中的第三个findUser函数一样。

Spring-data-jpa的能力远不止本文提到的这些，由于本文主要以整合介绍为主，对于Spring-data-jpa的使用只是介绍了常见的使用方式。诸如@Modifying操作、分页排序、原生SQL支持以及与Spring MVC的结合使用等等内容就不在本文中详细展开，这里先挖个坑，后续再补文章填坑，如您对这些感

兴趣可以关注我博客或简书，同样欢迎大家留言交流想法。

单元测试

在完成了上面的数据访问接口之后，按照惯例就是编写对应的单元测试来验证编写的内容是否正确。这里就不多做介绍，主要通过数据操作和查询来反复验证操作的正确性。

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

    @Autowired
    private UserRepository userRepository;

    @Test
    public void test() throws Exception {

        // 创建10条记录
        userRepository.save(new User("AAA", 10));
        userRepository.save(new User("BBB", 20));
        userRepository.save(new User("CCC", 30));
        userRepository.save(new User("DDD", 40));
        userRepository.save(new User("EEE", 50));
        userRepository.save(new User("FFF", 60));
        userRepository.save(new User("GGG", 70));
        userRepository.save(new User("HHH", 80));
        userRepository.save(new User("III", 90));
        userRepository.save(new User("JJJ", 100));

        // 测试findAll, 查询所有记录
        Assert.assertEquals(10, userRepository.findAll().size());

        // 测试findByName, 查询姓名为FFF的User
        Assert.assertEquals(60, userRepository.findByName("FFF").getAge().longValue());

        // 测试findUser, 查询姓名为FFF的User
        Assert.assertEquals(60, userRepository.findUser("FFF").getAge().longValue());

        // 测试findByNameAndAge, 查询姓名为FFF并且年龄为60的User
        Assert.assertEquals("FFF", userRepository.findByNameAndAge("FFF", 60).getName());

        // 测试删除姓名为AAA的User
        userRepository.delete(userRepository.findByName("AAA"));

        // 测试findAll, 查询所有记录, 验证上面的删除是否成功
        Assert.assertEquals(9, userRepository.findAll().size());

    }
```

```
}
```

完整示例

http://git.oschina.net/didispace/SpringBoot-Learning

之前在介绍使用JdbcTemplate和Spring-data-jpa时，都使用了单数据源。在单数据源的情况下，Spring Boot的配置非常简单，只需要在 `application.properties` 文件中配置连接参数即可。但是往往随着业务量发展，我们通常会进行数据库拆分或是引入其他数据库，从而我们需要配置多个数据源，下面基于之前的JdbcTemplate和Spring-data-jpa例子分别介绍两种多数据源的配置方式。

多数据源配置

创建一个Spring配置类，定义两个DataSource用来读取 `application.properties` 中的不同配置。如下例子中，主数据源配置为 `spring.datasource.primary` 开头的配置，第二数据源配置为 `spring.datasource.secondary` 开头的配置。

```
@Configuration
public class DataSourceConfig {

    @Bean(name = "primaryDataSource")
    @Qualifier("primaryDataSource")
    @ConfigurationProperties(prefix="spring.datasource.primary")
    public DataSource primaryDataSource() {
        return DataSourceBuilder.create().build();
    }

    @Bean(name = "secondaryDataSource")
    @Qualifier("secondaryDataSource")
    @Primary
    @ConfigurationProperties(prefix="spring.datasource.secondary")
    public DataSource secondaryDataSource() {
        return DataSourceBuilder.create().build();
    }

}
```

对应的 `application.properties` 配置如下：

```
spring.datasource.primary.url=jdbc:mysql://localhost:3306/test1
spring.datasource.primary.username=root
spring.datasource.primary.password=root
spring.datasource.primary.driver-class-name=com.mysql.jdbc.Driver

spring.datasource.secondary.url=jdbc:mysql://localhost:3306/test2
spring.datasource.secondary.username=root
spring.datasource.secondary.password=root
spring.datasource.secondary.driver-class-name=com.mysql.jdbc.Driver
```

JdbcTemplate支持

对JdbcTemplate的支持比较简单，只需要为其注入对应的datasource即可，如下例子，在创建JdbcTemplate的时候分别注入名为 `primaryDataSource` 和 `secondaryDataSource` 的数据源来区分不同的JdbcTemplate。

```
    @Bean(name = "primaryJdbcTemplate")
    public JdbcTemplate primaryJdbcTemplate(
            @Qualifier("primaryDataSource") DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }

    @Bean(name = "secondaryJdbcTemplate")
    public JdbcTemplate secondaryJdbcTemplate(
            @Qualifier("secondaryDataSource") DataSource dataSource) {
        return new JdbcTemplate(dataSource);
    }
```

接下来通过测试用例来演示如何使用这两个针对不同数据源的JdbcTemplate。

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

    @Autowired
    @Qualifier("primaryJdbcTemplate")
    protected JdbcTemplate jdbcTemplate1;

    @Autowired
    @Qualifier("secondaryJdbcTemplate")
    protected JdbcTemplate jdbcTemplate2;

    @Before
    public void setUp() {
        jdbcTemplate1.update("DELETE  FROM  USER ");
        jdbcTemplate2.update("DELETE  FROM  USER ");
    }

    @Test
    public void test() throws Exception {

        // 往第一个数据源中插入两条数据
        jdbcTemplate1.update("insert into user(id,name,age) values(?, ?, ?)", 1, "aaa", 20
);
        jdbcTemplate1.update("insert into user(id,name,age) values(?, ?, ?)", 2, "bbb", 30
);

        // 往第二个数据源中插入一条数据，若插入的是第一个数据源，则会主键冲突报错
        jdbcTemplate2.update("insert into user(id,name,age) values(?, ?, ?)", 1, "aaa", 20
);
```

```
        // 查一下第一个数据源中是否有两条数据，验证插入是否成功
        Assert.assertEquals("2", jdbcTemplate1.queryForObject("select count(1) from user",
 String.class));

        // 查一下第一个数据源中是否有两条数据，验证插入是否成功
        Assert.assertEquals("1", jdbcTemplate2.queryForObject("select count(1) from user",
 String.class));

    }


}
```

完整示例:Chapter3-2-3

Spring-data-jpa支持

对于数据源的配置可以沿用上例中 `DataSourceConfig` 的实现。

新增对第一数据源的JPA配置，注意两处注释的地方，用于指定数据源对应的 `Entity` 实体和 `Repository` 定义位置，用 `@Primary` 区分主数据源。

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
        entityManagerFactoryRef="entityManagerFactoryPrimary",
        transactionManagerRef="transactionManagerPrimary",
        basePackages= { "com.didispace.domain.p" }) //设置Repository所在位置
public class PrimaryConfig {

    @Autowired @Qualifier("primaryDataSource")
    private DataSource primaryDataSource;

    @Primary
    @Bean(name = "entityManagerPrimary")
    public EntityManager entityManager(EntityManagerFactoryBuilder builder) {
        return entityManagerFactoryPrimary(builder).getObject().createEntityManager();
    }

    @Primary
    @Bean(name = "entityManagerFactoryPrimary")
    public LocalContainerEntityManagerFactoryBean entityManagerFactoryPrimary (EntityManag
erFactoryBuilder builder) {
        return builder
                .dataSource(primaryDataSource)
                .properties(getVendorProperties(primaryDataSource))
                .packages("com.didispace.domain.p") //设置实体类所在位置
                .persistenceUnit("primaryPersistenceUnit")
                .build();
```

```
    }

    @Autowired
    private JpaProperties jpaProperties;

    private Map<String, String> getVendorProperties(DataSource dataSource) {
        return jpaProperties.getHibernateProperties(dataSource);
    }

    @Primary
    @Bean(name = "transactionManagerPrimary")
    public PlatformTransactionManager transactionManagerPrimary(EntityManagerFactoryBuilde
r builder) {
        return new JpaTransactionManager(entityManagerFactoryPrimary(builder).getObject())
;
    }

}
```

新增对第二数据源的JPA配置，内容与第一数据源类似，具体如下：

```
@Configuration
@EnableTransactionManagement
@EnableJpaRepositories(
        entityManagerFactoryRef="entityManagerFactorySecondary",
        transactionManagerRef="transactionManagerSecondary",
        basePackages= { "com.didispace.domain.s" }) //设置Repository所在位置
public class SecondaryConfig {

    @Autowired @Qualifier("secondaryDataSource")
    private DataSource secondaryDataSource;

    @Bean(name = "entityManagerSecondary")
    public EntityManager entityManager(EntityManagerFactoryBuilder builder) {
        return entityManagerFactorySecondary(builder).getObject().createEntityManager();
    }

    @Bean(name = "entityManagerFactorySecondary")
    public LocalContainerEntityManagerFactoryBean entityManagerFactorySecondary (EntityMan
agerFactoryBuilder builder) {
        return builder
                .dataSource(secondaryDataSource)
                .properties(getVendorProperties(secondaryDataSource))
                .packages("com.didispace.domain.s") //设置实体类所在位置
                .persistenceUnit("secondaryPersistenceUnit")
                .build();
    }

    @Autowired
    private JpaProperties jpaProperties;
```

```
    private Map<String, String> getVendorProperties(DataSource dataSource) {
        return jpaProperties.getHibernateProperties(dataSource);
    }

    @Bean(name = "transactionManagerSecondary")
    PlatformTransactionManager transactionManagerSecondary(EntityManagerFactoryBuilder bui
lder) {
        return new JpaTransactionManager(entityManagerFactorySecondary(builder).getObject(
));
    }

}
```

完成了以上配置之后，主数据源的实体和数据访问对象位于： `com.didispace.domain.p` ，次数据源的实体和数据访问接口位于： `com.didispace.domain.s` 。

分别在这两个package下创建各自的实体和数据访问接口

- 主数据源下，创建User实体和对应的Repository接口

```
@Entity
public class User {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private Integer age;

    public User(){}

    public User(String name, Integer age) {
        this.name = name;
        this.age = age;
    }

    // 省略getter、setter

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

}
```

- 从数据源下，创建Message实体和对应的Repository接口

```
@Entity
public class Message {

    @Id
    @GeneratedValue
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String content;

    public Message(){}

    public Message(String name, String content) {
        this.name = name;
        this.content = content;
    }

    // 省略getter、setter

}
public interface MessageRepository extends JpaRepository<Message, Long> {

}
```

接下来通过测试用例来验证使用这两个针对不同数据源的配置进行数据操作。

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

    @Autowired
    private UserRepository userRepository;
    @Autowired
    private MessageRepository messageRepository;

    @Test
    public void test() throws Exception {

        userRepository.save(new User("aaa", 10));
        userRepository.save(new User("bbb", 20));
        userRepository.save(new User("ccc", 30));
        userRepository.save(new User("ddd", 40));
        userRepository.save(new User("eee", 50));

        Assert.assertEquals(5, userRepository.findAll().size());

        messageRepository.save(new Message("o1", "aaaaaaaaaa"));
```

```
        messageRepository.save(new Message("o2", "bbbbbbbbbb"));
        messageRepository.save(new Message("o3", "cccccccccc"));

        Assert.assertEquals(3, messageRepository.findAll().size());

    }

}
```

完整示例:Chapter3-2-4

http://git.oschina.net/didispace/SpringBoot-Learning

Spring Boot中除了对常用的关系型数据库提供了优秀的自动化支持之外，对于很多NoSQL数据库一样提供了自动化配置的支持，包括：Redis, MongoDB, Elasticsearch, Solr和Cassandra。

使用Redis

Redis是一个开源的使用ANSI C语言编写、支持网络、可基于内存亦可持久化的日志型、Key-Value数据库。

- Redis官网
- Redis中文社区

引入依赖

Spring Boot提供的数据访问框架Spring Data Redis基于Jedis。可以通过引入 `spring-boot-starter-redis` 来配置依赖关系。

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

参数配置

按照惯例在 `application.properties` 中加入Redis服务端的相关配置，具体说明如下：

```
# 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=localhost
# 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
```

其中spring.redis.database的配置通常使用0即可，Redis在配置的时候可以设置数据库数量，默认为16，可以理解为数据库的schema

测试访问

通过编写测试用例，举例说明如何访问Redis。

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

    @Autowired
    private StringRedisTemplate stringRedisTemplate;

    @Test
    public void test() throws Exception {

        // 保存字符串
        stringRedisTemplate.opsForValue().set("aaa", "111");
        Assert.assertEquals("111", stringRedisTemplate.opsForValue().get("aaa"));

    }

}
```

通过上面这段极为简单的测试案例演示了如何通过自动配置的 `StringRedisTemplate` 对象进行Redis的读写操作，该对象从命名中就可注意到支持的是String类型。如果有使用过spring-data-redis的开发者一定熟悉 `RedisTemplate` 接口， `StringRedisTemplate` 就相当于 `RedisTemplate` 的实现。

除了String类型，实战中我们还经常会在Redis中存储对象，这时候我们就会想是否可以使用类似 `RedisTemplate` 来初始化并进行操作。但是Spring Boot并支持直接使用，需要我们自己实现 `RedisSerializer` 接口来对传入对象进行序列化和反序列化，下面我们通过一个实例来完成对象的读写操作。

- 创建要存储的对象：User

```
public class User implements Serializable {

    private static final long serialVersionUID = -1L;

    private String username;
    private Integer age;

    public User(String username, Integer age) {
        this.username = username;
        this.age = age;
    }

    // 省略getter和setter

}
```

- 实现对象的序列化接口

```
public class RedisObjectSerializer implements RedisSerializer<Object> {

  private Converter<Object, byte[]> serializer = new SerializingConverter();
  private Converter<byte[], Object> deserializer = new DeserializingConverter();

  static final byte[] EMPTY_ARRAY = new byte[0];

  public Object deserialize(byte[] bytes) {
    if (isEmpty(bytes)) {
      return null;
    }

    try {
      return deserializer.convert(bytes);
    } catch (Exception ex) {
      throw new SerializationException("Cannot deserialize", ex);
    }
  }

  public byte[] serialize(Object object) {
    if (object == null) {
      return EMPTY_ARRAY;
    }

    try {
      return serializer.convert(object);
    } catch (Exception ex) {
      return EMPTY_ARRAY;
    }
  }

  private boolean isEmpty(byte[] data) {
    return (data == null || data.length == 0);
  }
}
```

- 配置针对User的RedisTemplate实例

```
@Configuration
public class RedisConfig {

    @Bean
    JedisConnectionFactory jedisConnectionFactory() {
        return new JedisConnectionFactory();
    }

    @Bean
```

```
    public RedisTemplate<String, User> redisTemplate(RedisConnectionFactory factory) {
        RedisTemplate<String, User> template = new RedisTemplate<String, User>();
        template.setConnectionFactory(jedisConnectionFactory());
        template.setKeySerializer(new StringRedisSerializer());
        template.setValueSerializer(new RedisObjectSerializer());
        return template;
    }


}
```

- 完成了配置工作后，编写测试用例实验效果

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

    @Autowired
    private RedisTemplate<String, User> redisTemplate;

    @Test
    public void test() throws Exception {

        // 保存对象
        User user = new User("超人", 20);
        redisTemplate.opsForValue().set(user.getUsername(), user);

        user = new User("蝙蝠侠", 30);
        redisTemplate.opsForValue().set(user.getUsername(), user);

        user = new User("蜘蛛侠", 40);
        redisTemplate.opsForValue().set(user.getUsername(), user);

        Assert.assertEquals(20, redisTemplate.opsForValue().get("超人").getAge().longValue(
));
        Assert.assertEquals(30, redisTemplate.opsForValue().get("蝙蝠侠").getAge().longValu
e());
        Assert.assertEquals(40, redisTemplate.opsForValue().get("蜘蛛侠").getAge().longValu
e());

    }

}
```

当然spring-data-redis中提供的数据操作远不止这些，本文仅作为在Spring Boot中使用redis时的配置参考，更多对于redis的操作使用，请参考Spring-data-redis Reference。

- 本文完整示例chapter3-2-5


持续更新...
关注 社区 · 轻松学习
长按识别ME

前段时间分享了关于Spring Boot中使用Redis的文章，除了Redis之后，我们在互联网产品中还经常会用到另外一款著名的NoSQL数据库MongoDB。

下面就来简单介绍一下MongoDB，并且通过一个例子来介绍Spring Boot中对MongoDB访问的配置和使用。

MongoDB简介

MongoDB是一个基于分布式文件存储的数据库，它是一个介于关系数据库和非关系数据库之间的产品，其主要目标是在键/值存储方式（提供了高性能和高度伸缩性）和传统的RDBMS系统（具有丰富的功能）之间架起一座桥梁，它集两者的优势于一身。

MongoDB支持的数据结构非常松散，是类似json的bson格式，因此可以存储比较复杂的数据类型，也因为他的存储格式也使得它所存储的数据在Nodejs程序应用中使用非常流畅。

既然称为NoSQL数据库，Mongo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。

但是，MongoDB也不是万能的，同MySQL等关系型数据库相比，它们在针对不同的数据类型和事务要求上都存在自己独特的优势。在数据存储的选择中，坚持多样化原则，选择更好更经济的方式，而不是自上而下的统一化。

较常见的，我们可以直接用MongoDB来存储键值对类型的数据，如：验证码、Session等；由于MongoDB的横向扩展能力，也可以用来存储数据规模会在未来变的非常巨大的数据，如：日志、评论等；由于MongoDB存储数据的弱类型，也可以用来存储一些多变json数据，如：与外系统交互时经常变化的JSON报文。而对于一些对数据有复杂的高事务性要求的操作，如：账户交易等就不适合使用MongoDB来存储。

MongoDB官网

访问MongoDB

在Spring Boot中，对如此受欢迎的MongoDB，同样提供了自配置功能。

引入依赖

Spring Boot中可以通过在 `pom.xml` 中加入 `spring-boot-starter-data-mongodb` 引入对mongodb的访问支持依赖。它的实现依赖 `spring-data-mongodb` 。是的，您没有看错，又是 `spring-data` 的子项目，之前介绍过 `spring-data-jpa` 、 `spring-data-redis` ，对于mongodb的访问， `spring-data` 也提供了强大的支持，下面就开始动手试试吧。

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

快速开始使用Spring-data-mongodb

若MongoDB的安装配置采用默认端口，那么在自动配置的情况下，我们不需要做任何参数配置，就能马上连接上本地的MongoDB。下面直接使用 `spring-data-mongodb` 来尝试对mongodb的存取操作。（记得mongod启动您的mongodb）

- 创建要存储的User实体，包含属性：id、username、age

```
public class User {

    @Id
    private Long id;

    private String username;
    private Integer age;

    public User(Long id, String username, Integer age) {
        this.id = id;
        this.username = username;
        this.age = age;
    }

    // 省略getter和setter

}
```

- 实现User的数据访问对象：UserRepository

```
public interface UserRepository extends MongoRepository<User, Long> {

    User findByUsername(String username);

}
```

- 在单元测试中调用

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

    @Autowired
    private UserRepository userRepository;

    @Before
    public void setUp() {
        userRepository.deleteAll();
    }

    @Test
    public void test() throws Exception {
```

```
        // 创建三个User，并验证User总数
        userRepository.save(new User(1L, "didi", 30));
        userRepository.save(new User(2L, "mama", 40));
        userRepository.save(new User(3L, "kaka", 50));
        Assert.assertEquals(3, userRepository.findAll().size());

        // 删除一个User，再验证User总数
        User u = userRepository.findOne(1L);
        userRepository.delete(u);
        Assert.assertEquals(2, userRepository.findAll().size());

        // 删除一个User，再验证User总数
        u = userRepository.findByUsername("mama");
        userRepository.delete(u);
        Assert.assertEquals(1, userRepository.findAll().size());

    }

}
```

参数配置

通过上面的例子，我们可以轻而易举的对MongoDB进行访问，但是实战中，应用服务器与MongoDB通常不会部署于同一台设备之上，这样就无法使用自动化的本地配置来进行使用。这个时候，我们也可以方便的配置来完成支持，只需要在application.properties中加入mongodb服务端的相关配置，具体示例如下：

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

在尝试此配置时，记得在*mongo*中对*test*库创建具备读写权限的用户（用户名为*name*，密码为*pass*），不同版本的用户创建语句不同，注意查看文档做好准备工作

若使用mongodb 2.x，也可以通过如下参数配置，该方式不支持mongodb 3.x。

```
spring.data.mongodb.host=localhost spring.data.mongodb.port=27017
```


持续更新...
关注 社区 · 轻松学习
长按识别ME

摘要: 原创出处:www.bysocket.com 泥瓦匠BYSocket 希望转载，保留摘要，谢谢!

推荐一本书《腾讯传》。

新年第一篇 Springboot 技术文诞生。泥瓦匠准备写写 Springboot 相关最佳实践。一方面总结下一些 Springboot 相关，一方面和大家交流交流 Springboot 框架。

现在业界互联网流行的数据操作层框架 Mybatis，下面详解下 Springboot 如何整合 Mybatis ，这边没有使用 Mybatis Annotation 这种，是使用 xml 配置 SQL。因为我觉得 SQL 和业务代码应该隔离，方便和 DBA 校对 SQL。二者 XML 对较长的 SQL 比较清晰。

一、运行 springboot-mybatis 工程

git clone 下载工程 springboot-learning-example ，项目地址见 GitHub。下面开始运行工程步骤（Quick Start）：

1.数据库准备

a.创建数据库 springbootdb：

```
CREATE DATABASE springbootdb;
```

b.创建表 city： (因为我喜欢徒步)

```
DROP TABLE IF EXISTS  `city`;
CREATE TABLE `city` (
  `id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '城市编号',
  `province_id` int(10) unsigned  NOT NULL COMMENT '省份编号',
  `city_name` varchar(25) DEFAULT NULL COMMENT '城市名称',
  `description` varchar(25) DEFAULT NULL COMMENT '描述',
  PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

c.插入数据

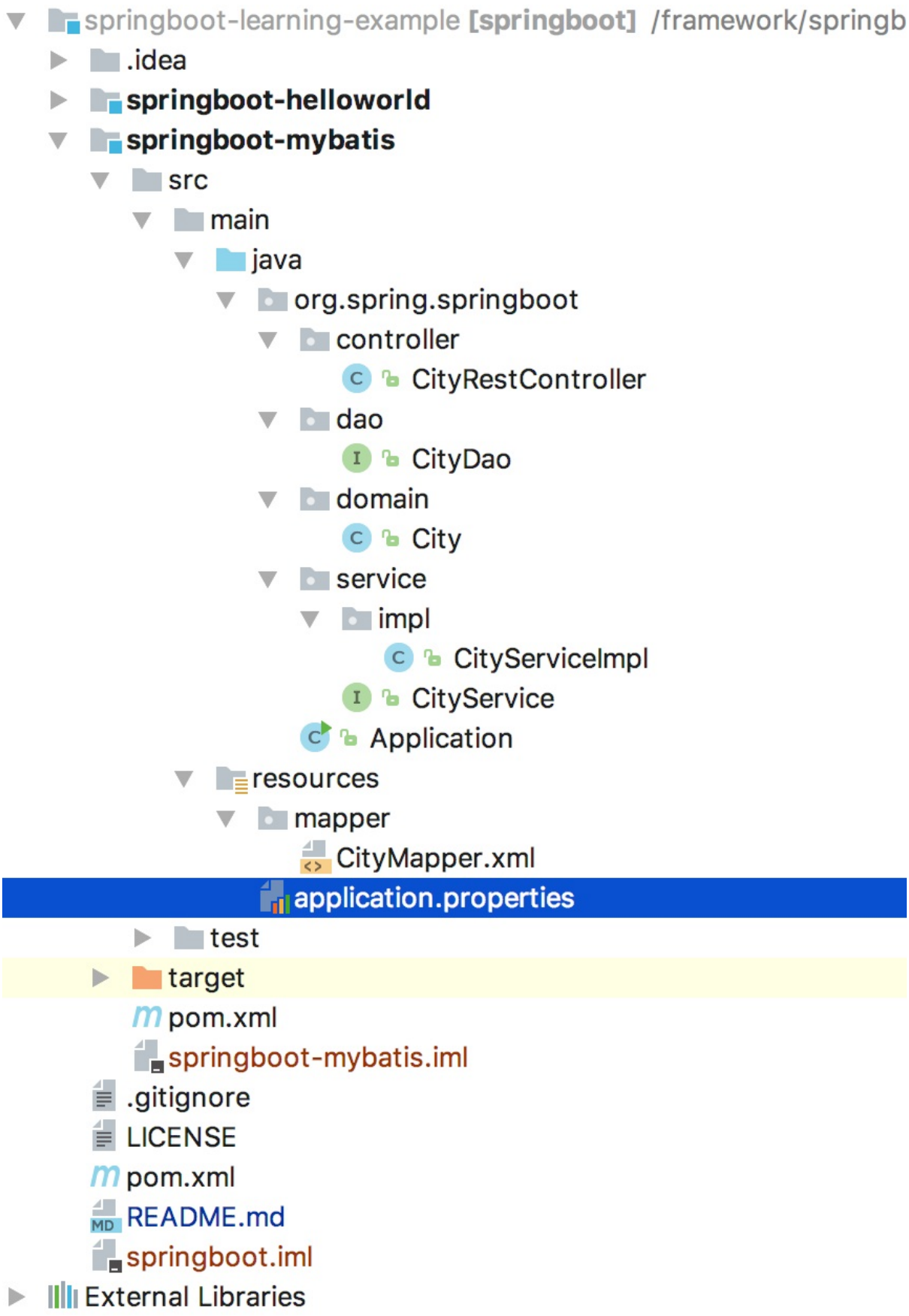
```
INSERT city VALUES (1 ,1,'温岭市','BYSocket 的家在温岭。');
```

2. 项目结构介绍

项目结构如下图所示：

```
org.spring.springboot.controller - Controller 层
org.spring.springboot.dao - 数据操作层 DAO
org.spring.springboot.domain - 实体类
org.spring.springboot.service - 业务逻辑层
Application - 应用启动类
```

```
application.properties - 应用配置文件，应用启动会自动读取配置
```



3.改数据库配置

打开 application.properties 文件， 修改相应的数据源配置，比如数据源地址、账号、密码等。（如果不是用 MySQL，自行添加连接驱动 pom，然后修改驱动名配置。）

4.编译工程

在项目根目录 springboot-learning-example，运行 maven 指令：

```
mvn clean install
```

5.运行工程

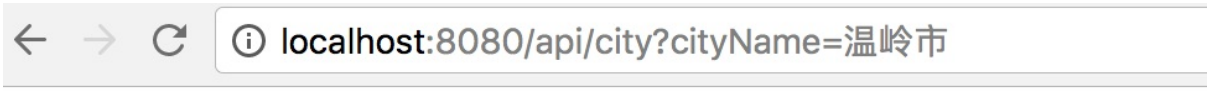
右键运行 Application 应用启动类的 main 函数，然后在浏览器访问：

```
http://localhost:8080/api/city?cityName=温岭市
```

可以看到返回的 JSON 结果：

```
{
    "id": 1,
    "provinceId": 1,
    "cityName": "温岭市",
    "description": "我的家在温岭。"
}
```

如图：



{
    "id": 1,
    "provinceId": 1,
    "cityName": "温岭市",
    "description": "我的家在温岭。"
}

二、springboot-mybatis 工程配置详解

1.pom 添加 Mybatis 依赖

```
<!-- Spring Boot Mybatis 依赖 -->
<dependency>
    <groupId>org.mybatis.spring.boot</groupId>
    <artifactId>mybatis-spring-boot-starter</artifactId>
    <version>${mybatis-spring-boot}</version>
</dependency>
```

整个工程的 pom.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>springboot</groupId>
    <artifactId>springboot-mybatis</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-mybatis :: 整合 Mybatis Demo</name>

    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>

    <properties>
        <mybatis-spring-boot>1.2.0</mybatis-spring-boot>
        <mysql-connector>5.1.39</mysql-connector>
    </properties>

    <dependencies>

        <!-- Spring Boot Web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Spring Boot Test 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
```

```
        <!-- Spring Boot Mybatis 依赖 -->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>${mybatis-spring-boot}</version>
        </dependency>

        <!-- MySQL 连接驱动依赖 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql-connector}</version>
        </dependency>

        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

2.在 application.properties 应用配置文件，增加 Mybatis 相关配置

```
## Mybatis 配置
mybatis.typeAliasesPackage=org.spring.springboot.domain
mybatis.mapperLocations=classpath:mapper/*.xml
```

mybatis.typeAliasesPackage 配置为 org.spring.springboot.domain，指向实体类包路径。
mybatis.mapperLocations 配置为 classpath 路径下 mapper 包下，* 代表会扫描所有 xml 文件。

mybatis 其他配置相关详解如下：

```
mybatis.config = mybatis 配置文件名称
mybatis.mapperLocations = mapper xml 文件地址
mybatis.typeAliasesPackage = 实体类包路径
mybatis.typeHandlersPackage = type handlers 处理器包路径
mybatis.check-config-location = 检查 mybatis 配置是否存在，一般命名为 mybatis-config.xml
mybatis.executorType = 执行模式。默认是 SIMPLE
```

3.在 Application 应用启动类添加注解 MapperScan Application.java 代码如下：

```
/**
 * Spring Boot 应用启动类
 *
 * Created by bysocket on 16/4/26.
```

```
 */
// Spring Boot 应用的标识
@SpringBootApplication
// mapper 接口类扫描包配置
@MapperScan("org.spring.springboot.dao")
public class Application {

    public static void main(String[] args) {
        // 程序启动入口
        // 启动嵌入式的 Tomcat 并初始化 Spring 环境及其各 Spring 组件
        SpringApplication.run(Application.class,args);
    }
}
```

mapper 接口类扫描包配置注解 MapperScan： 用这个注解可以注册 Mybatis mapper 接口类。

4.添加相应的 City domain类、CityDao mapper接口类 City.java:

```
/**
 * 城市实体类
 *
 * Created by bysocket on 07/02/2017.
 */
public class City {

    /**
     * 城市编号
     */
    private Long id;

    /**
     * 省份编号
     */
    private Long provinceId;

    /**
     * 城市名称
     */
    private String cityName;

    /**
     * 描述
     */
    private String description;

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
```

```
        this.id = id;
    }

    public Long getProvinceId() {
        return provinceId;
    }

    public void setProvinceId(Long provinceId) {
        this.provinceId = provinceId;
    }

    public String getCityName() {
        return cityName;
    }

    public void setCityName(String cityName) {
        this.cityName = cityName;
    }

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

CityDao.java:

```
/**
 * 城市 DAO 接口类
 *
 * Created by bysocket on 07/02/2017.
 */
public interface CityDao {

    /**
     * 根据城市名称，查询城市信息
     *
     * @param cityName 城市名
     */
    City findByName(@Param("cityName") String cityName);
}
```

其他不明白的，可以 git clone 下载工程 springboot-learning-example ， 工程代码注解很详细。
https://github.com/JeffLi1993/springboot-learning-example。

三、其他

利用 Mybatis-generator自动生成代码 http://www.cnblogs.com/yjmyzz/p/4210554.html Mybatis 通用 Mapper3 https://github.com/abel533/Mapper Mybatis 分页插件 PageHelper https://github.com/pagehelper/Mybatis-PageHelper

『 公司需要人、产品、业务和方向，方向又要人、产品、业务和方向，方向… 循环』

本文提纲 一、前言 二、运行 springboot-mybatis-annotation 工程 三、springboot-mybatis-annotation 工程配置详解 四、小结

运行环境：JDK 7 或 8、Maven 3.0+ **技术栈**：SpringBoot 1.5+、SpringBoot Mybatis Starter 1.2+ 、MyBatis 3.4+

前言

距离第一篇 Spring Boot 系列的博文 3 个月了。《Springboot 整合 Mybatis 的完整 Web 案例》第一篇出来是 XML 配置 SQL 的形式。虽然 XML 形式是我比较推荐的，但是注解形式也是方便的。尤其一些小系统，快速的 CRUD 轻量级的系统。

这里感谢晓春 http://xchunzhao.tk/ 的 Pull Request，提供了 springboot-mybatis-annotation 的实现。

一、运行 springboot-mybatis-annotation 工程

由于这篇文章和 《Springboot 整合 Mybatis 的完整 Web 案例》 类似，所以运行这块环境配置大家参考另外一篇兄弟文章。

然后Application 应用启动类的 main 函数，然后在浏览器访问：

```
http://localhost:8080/api/city?cityName=温岭市
```

可以看到返回的 JSON 结果：

```
{
  "id": 1,
  "provinceId": 1,
  "cityName": "温岭市",
  "description": "我的家在温岭。"
}
```

三、springboot-mybatis-annotation 工程配置详解

1.pom 添加 Mybatis 依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/mave
n-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>springboot</groupId>
```

```
    <artifactId>springboot-mybatis-annotation</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <name>springboot-mybatis-annotation</name>
    <description>Springboot-mybatis :: 整合Mybatis Annotation Demo</description>

    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>

    <properties>
        <mybatis-spring-boot>1.2.0</mybatis-spring-boot>
        <mysql-connector>5.1.39</mysql-connector>
    </properties>

    <dependencies>

        <!-- Spring Boot Web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Spring Boot Test 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Spring Boot Mybatis 依赖 -->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>${mybatis-spring-boot}</version>
        </dependency>

        <!-- MySQL 连接驱动依赖 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql-connector}</version>
        </dependency>

        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
```

```
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>


</project>
```

2.在 CityDao 城市数据操作层接口类添加注解 @Mapper、@Select 和 @Results

```
/**
* 城市 DAO 接口类
*
* Created by xchunzhao on 02/05/2017.
*/
@Mapper // 标志为 Mybatis 的 Mapper
public interface CityDao {

/**
* 根据城市名称，查询城市信息
*
* @param cityName 城市名
*/
@Select("SELECT * FROM city where city_name=${cityName}")
// 返回 Map 结果集
@Results({
@Result(property = "id", column = "id"),
@Result(property = "provinceId", column = "province_id"),
@Result(property = "cityName", column = "city_name"),
@Result(property = "description", column = "description"),
})
City findByName(@Param("cityName") String cityName);
}
```

@Mapper 标志接口为 MyBatis Mapper 接口 @Select 是 Select 操作语句 @Results 标志结果集，以及与库表字段的映射关系

其他的注解可以看 org.apache.ibatis.annotations 包提供的，如图：

可以 git clone 下载工程 springboot-learning-example， springboot-mybatis-annotation 工程代码注解很详细。 https://github.com/JeffLi1993/springboot-learning-example。

四、小结

注解不涉及到配置，更近贴近 0 配置。再次感谢晓春 http://xchunzhao.tk/ 的 Pull Request~


持续更新...
关注 社区 · 轻松学习
长按识别ME

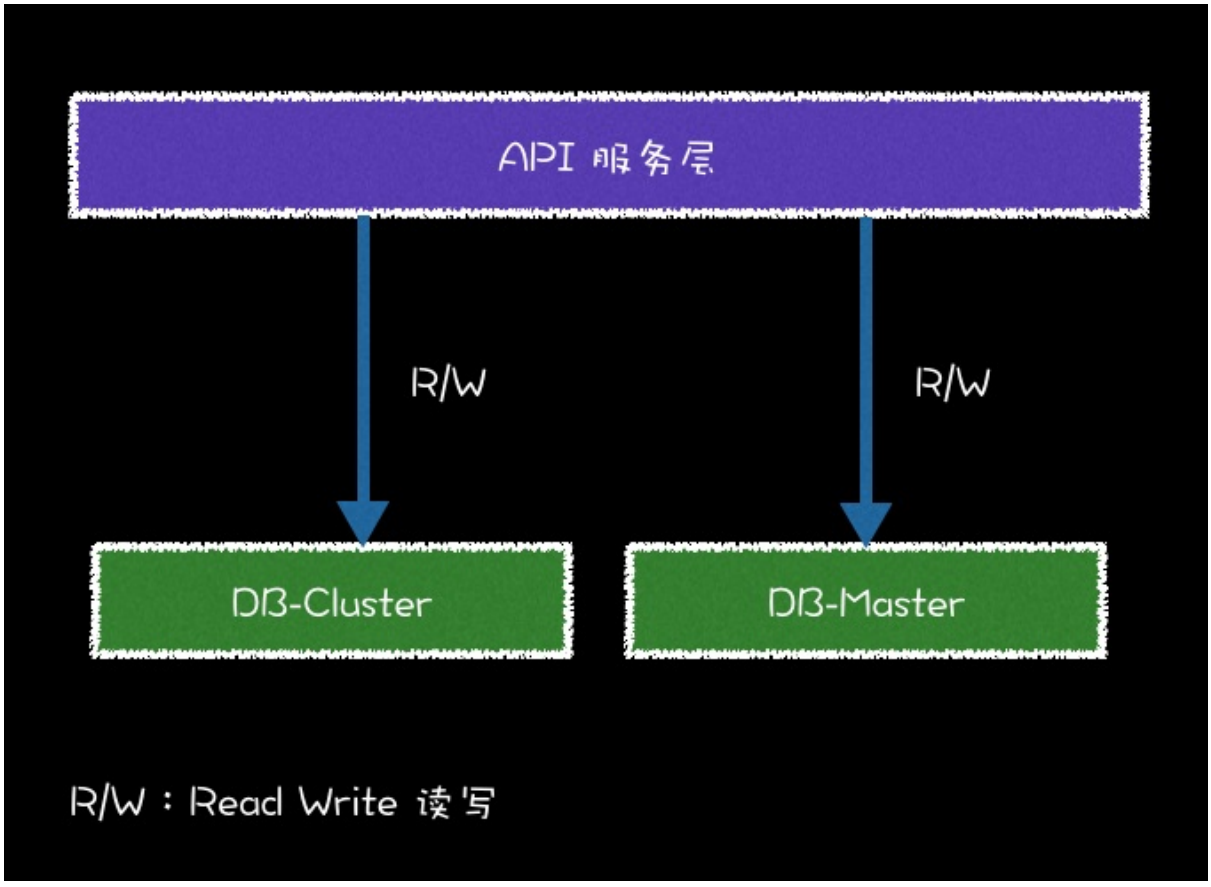
摘要: 原创出处:www.bysocket.com 泥瓦匠BYSocket 希望转载，保留摘要，谢谢!

“清醒时做事，糊涂时跑步，大怒时睡觉，独处时思考”

本文提纲 一、多数据源的应用场景 二、运行 springboot-mybatis-mutil-datasource 工程案例 三、springboot-mybatis-mutil-datasource 工程代码配置详解

一、多数据源的应用场景

目前，业界流行的数据操作框架是 Mybatis，那 Druid 是什么呢？ Druid 是 Java 的数据库连接池组件。Druid 能够提供强大的监控和扩展功能。比如可以监控 SQL ，在监控业务可以查询慢查询 SQL 列表等。Druid 核心主要包括三部分： 1. DruidDriver 代理 Driver，能够提供基于 Filter－Chain 模式的插件体系。 2. DruidDataSource 高效可管理的数据库连接池 3. SQLParser 当业务数据量达到了一定程度，DBA 需要合理配置数据库资源。即配置主库的机器高配置，把核心高频的数据放在主库上；把次要的数据放在从库，低配置。开源节流嘛，就这个意思。把数据放在不同的数据库里，就需要通过不同的数据源进行操作数据。这里我们举个 springboot-mybatis-mutil-datasource 工程案例： user 用户表在主库 master 上，地址表 city 在从库 cluster 上。下面实现获取 根据用户名获取用户信息，包括从库的地址信息 REST API，那么需要从主库和从库中分别获取数据，并在业务逻辑层组装返回。逻辑如图：



下面就运行这个案例。

二、运行 springboot-mybatis-mutil-datasource 工程案例

git clone 下载工程 springboot-learning-example，项目地址见 GitHub - https://github.com/JeffLi1993/springboot-learning-example。下面开始运行工程步骤（Quick Start）：
1.数据库准备 a.创建 cluster 数据库 springbootdb：

```
CREATE DATABASE springbootdb;
```

b.创建表 city：(因为我喜欢徒步)

```
DROP TABLE IF EXISTS `city`;
CREATE TABLE `city` (
`id` int(10) unsigned NOT NULL AUTO_INCREMENT COMMENT '城市编号',
`province_id` int(10) unsigned NOT NULL COMMENT '省份编号',
`city_name` varchar(25) DEFAULT NULL COMMENT '城市名称',
`description` varchar(25) DEFAULT NULL COMMENT '描述',
PRIMARY KEY (`id`)
) ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

c.插入数据

```
INSERT city VALUES (1 ,1,'温岭市','BYSocket 的家在温岭。');
```

然后，再创建一个 master 数据库 a.创建数据库 springbootdb_cluster：

```
CREATE DATABASE springbootdb_cluster;
```

b.创建表 user：

```
DROP TABLE IF EXISTS `city`;
CREATE TABLE user
(
id INT(10) unsigned PRIMARY KEY NOT NULL COMMENT '用户编号' AUTO_INCREMENT,
user_name VARCHAR(25) COMMENT '用户名称',
description VARCHAR(25) COMMENT '描述'
)ENGINE=InnoDB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

c.插入数据

```
INSERT user VALUES (1 ,'泥瓦匠','他有一个小网站 bysocket.com');
```

（以上数据库创建无先后顺序） **2. 项目结构介绍** 项目结构如下图所示：
org.spring.springboot.config.ds - 配置层，这里是数据源的配置，包括 master 和 cluster 的数据源配置 org.spring.springboot.controller - Controller 层 org.spring.springboot.dao - 数据操作层 DAO，细分了 master 和 cluster 包下的 DAO 操作类 org.spring.springboot.domain - 实体类 org.spring.springboot.service - 业务逻辑层 Application - 应用启动类 application.properties - 应用配置

文件，应用启动会自动读取配置 **3.改数据库配置** 打开 application.properties 文件， 修改相应的主从数据源配置，比如数据源地址、账号、密码等。（如果不是用 MySQL，自行添加连接驱动 pom，然后修改驱动名配置。） **4.编译工程** 在项目根目录 springboot-learning-example，运行 maven 指令：

```
mvn clean install
```

5.运行工程 右键运行 Application 应用启动类（位置：/springboot-learning-example/springboot-mybatis-mutil-datasource/src/main/java/org/spring/springboot/Application.java）的 main 函数，这样就成功启动了 springboot-mybatis-mutil-datasource 案例。 在浏览器打开：http://localhost:8080/api/user?userName=泥瓦匠 浏览器返回 JSON 结果：

```
{
    "id": 1,
    "userName": "泥瓦匠",
    "description": "他有一个小网站 bysocket.com",
    "city": {
        "id": 1,
        "provinceId": 1,
        "cityName": "温岭市",
        "description": "BYSocket 的家在温岭。"
    }
}
```

这里 city 结果体来自 cluster 库，user 结果体来自 master 库。

三、springboot-mybatis-mutil-datasource 工程代码配置详解

代码共享在我的 GitHub 上： https://github.com/JeffLi1993/springboot-learning-example/tree/master/springboot-mybatis-mutil-datasource 首先代码工程结构如下：org.spring.springboot.config.ds 包包含了多数据源的配置，同样有第三个数据源，按照前几个复制即可 resources/mapper 下面有两个模块，分别是 Mybatis 不同数据源需要扫描的 mapper xml 目录

```
├── pom.xml
└── src
    └── main
        ├── java
        │   └── org
        │       └── spring
        │           └── springboot
        │               ├── Application.java
        │               ├── config
        │               │   └── ds
        │               │       ├── ClusterDataSourceConfig.java
        │               │       └── MasterDataSourceConfig.java
        │               ├── controller
        │               │   └── UserRestController.java
        │               ├── dao
```

```
        │               │   ├── cluster
        │               │   │   └── CityDao.java
        │               │   └── master
        │               │       └── UserDao.java
        │               ├── domain
        │               │   ├── City.java
        │               │   └── User.java
        │               └── service
        │                   ├── UserService.java
        │                   └── impl
        │                       └── UserServiceImpl.java
        └── resources
            ├── application.properties
            └── mapper
                ├── cluster
                │   └── CityMapper.xml
                └── master
                    └── UserMapper.xml
```

1. 依赖 pom.xml Mybatis 通过 Spring Boot Mybatis Starter 依赖 Druid 是数据库连接池依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>springboot</groupId>
    <artifactId>springboot-mybatis-mutil-datasource</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-mybatis-mutil-datasource :: Spring Boot 实现 Mybatis 多数据源配置</name
>

    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>

    <properties>
        <mybatis-spring-boot>1.2.0</mybatis-spring-boot>
        <mysql-connector>5.1.39</mysql-connector>
        <druid>1.0.18</druid>
    </properties>

    <dependencies>

        <!-- Spring Boot Web 依赖 -->
```

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

        <!-- Spring Boot Test 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Spring Boot Mybatis 依赖 -->
        <dependency>
            <groupId>org.mybatis.spring.boot</groupId>
            <artifactId>mybatis-spring-boot-starter</artifactId>
            <version>${mybatis-spring-boot}</version>
        </dependency>

        <!-- MySQL 连接驱动依赖 -->
        <dependency>
            <groupId>mysql</groupId>
            <artifactId>mysql-connector-java</artifactId>
            <version>${mysql-connector}</version>
        </dependency>

        <!-- Druid 数据连接池依赖 -->
        <dependency>
            <groupId>com.alibaba</groupId>
            <artifactId>druid</artifactId>
            <version>${druid}</version>
        </dependency>

        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

2. application.properties 配置两个数据源配置 数据源配置会被数据源数据源配置如下

```
## master 数据源配置
master.datasource.url=jdbc:mysql://localhost:3306/springbootdb?useUnicode=true&characterEn
coding=utf8
master.datasource.username=root
master.datasource.password=123456
master.datasource.driverClassName=com.mysql.jdbc.Driver
```

```
## cluster 数据源配置
cluster.datasource.url=jdbc:mysql://localhost:3306/springbootdb_cluster?useUnicode=true&ch
aracterEncoding=utf8
cluster.datasource.username=root
cluster.datasource.password=123456
cluster.datasource.driverClassName=com.mysql.jdbc.Driver
```

3. 数据源配置 多数据源配置的时候注意，必须要有一个主数据源，即 MasterDataSourceConfig 配置：

```
@Configuration
// 扫描 Mapper 接口并容器管理
@MapperScan(basePackages = MasterDataSourceConfig.PACKAGE, sqlSessionFactoryRef = "masterS
qlSessionFactory")
public class MasterDataSourceConfig {

    // 精确到 master 目录，以便跟其他数据源隔离
    static final String PACKAGE = "org.spring.springboot.dao.master";
    static final String MAPPER_LOCATION = "classpath:mapper/master/*.xml";

    @Value("${master.datasource.url}")
    private String url;

    @Value("${master.datasource.username}")
    private String user;

    @Value("${master.datasource.password}")
    private String password;

    @Value("${master.datasource.driverClassName}")
    private String driverClass;

    @Bean(name = "masterDataSource")
    @Primary
    public DataSource masterDataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driverClass);
        dataSource.setUrl(url);
        dataSource.setUsername(user);
        dataSource.setPassword(password);
        return dataSource;
    }

    @Bean(name = "masterTransactionManager")
    @Primary
    public DataSourceTransactionManager masterTransactionManager() {
        return new DataSourceTransactionManager(masterDataSource());
    }
```

```
    @Bean(name = "masterSqlSessionFactory")
    @Primary
    public SqlSessionFactory masterSqlSessionFactory(@Qualifier("masterDataSource") DataSo
urce masterDataSource)
            throws Exception {
        final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
        sessionFactory.setDataSource(masterDataSource);
        sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver()
                .getResources(MasterDataSourceConfig.MAPPER_LOCATION));
        return sessionFactory.getObject();
    }
}
```

@Primary 标志这个 Bean 如果在多个同类 Bean 候选时，该 Bean 优先被考虑。「多数据源配置的时候注意，必须要有一个主数据源，用 @Primary 标志该 Bean」 @MapperScan 扫描 Mapper 接口并容器管理，包路径精确到 master，为了和下面 cluster 数据源做到精确区分 @Value 获取全局配置文件 application.properties 的 kv 配置,并自动装配 sqlSessionFactoryRef 表示定义了 key ，表示一个唯一 SqlSessionFactory 实例 同理可得，从数据源 ClusterDataSourceConfig 配置如下：

```
@Configuration
// 扫描 Mapper 接口并容器管理
@MapperScan(basePackages = ClusterDataSourceConfig.PACKAGE, sqlSessionFactoryRef = "cluste
rSqlSessionFactory")
public class ClusterDataSourceConfig {

    // 精确到 cluster 目录，以便跟其他数据源隔离
    static final String PACKAGE = "org.spring.springboot.dao.cluster";
    static final String MAPPER_LOCATION = "classpath:mapper/cluster/*.xml";

    @Value("${cluster.datasource.url}")
    private String url;

    @Value("${cluster.datasource.username}")
    private String user;

    @Value("${cluster.datasource.password}")
    private String password;

    @Value("${cluster.datasource.driverClassName}")
    private String driverClass;

    @Bean(name = "clusterDataSource")
    public DataSource clusterDataSource() {
        DruidDataSource dataSource = new DruidDataSource();
        dataSource.setDriverClassName(driverClass);
        dataSource.setUrl(url);
        dataSource.setUsername(user);
        dataSource.setPassword(password);
        return dataSource;
```

```
    }

    @Bean(name = "clusterTransactionManager")
    public DataSourceTransactionManager clusterTransactionManager() {
        return new DataSourceTransactionManager(clusterDataSource());
    }

    @Bean(name = "clusterSqlSessionFactory")
    public SqlSessionFactory clusterSqlSessionFactory(@Qualifier("clusterDataSource") Data
Source clusterDataSource)
            throws Exception {
        final SqlSessionFactoryBean sessionFactory = new SqlSessionFactoryBean();
        sessionFactory.setDataSource(clusterDataSource);
        sessionFactory.setMapperLocations(new PathMatchingResourcePatternResolver()
                .getResources(ClusterDataSourceConfig.MAPPER_LOCATION));
        return sessionFactory.getObject();
    }
}
```

上面数据配置分别扫描 Mapper 接口，org.spring.springboot.dao.master（对应 xml classpath:mapper/master ） 和 org.spring.springboot.dao.cluster（对应 xml classpath:mapper/cluster ） 包中对应的 UserDAO 和 CityDAO 。 都有 @Mapper 标志为 Mybatis 的并通过容器管理的 Bean。Mybatis 内部会使用反射机制运行去解析相应 SQL。 **3.业务层 biz** biz 照常注入了两个 DAO，如同以前一样正常工作。不用关心和指定到具体说明数据源。

```
/**
 * 用户业务实现层
 *
 * Created by bysocket on 07/02/2017.
 */
@Service
public class UserServiceImpl implements UserService {

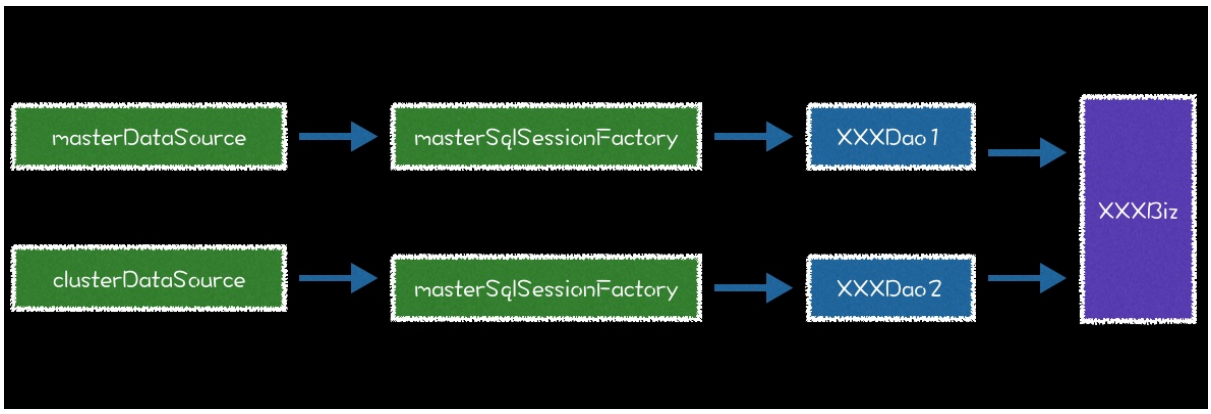
    @Autowired
    private UserDao userDao; // 主数据源

    @Autowired
    private CityDao cityDao; // 从数据源

    @Override
    public User findByName(String userName) {
        User user = userDao.findByName(userName);
        City city = cityDao.findByName("温岭市");
        user.setCity(city);
        return user;
    }
}
```

四、小结

多数据源适合的场景很多。不同的 DataSource ， 不同的 SqlSessionFactory 和 不同的 DAO 层，在业务逻辑层做 整合。总结的架构图如下：



同样，如果实战中，大家遇到什么，或者建议《Spring boot 那些事》还需要一起交流的。请点击留言。

Spring Boot在所有内部日志中使用Commons Logging，但是默认配置也提供了对常用日志的支持，如：Java Util Logging，Log4J, Log4J2和Logback。每种Logger都可以通过配置使用控制台或者文件输出日志内容。

格式化日志

默认的日志输出如下：

```
2016-04-13 08:23:50.120  INFO 37397 --- [ main] org.hibernate.Version : HHH000412: Hiberna
te Core {4.3.11.Final}
```

输出内容元素具体如下：

- 时间日期： 精确到毫秒
- 日志级别： ERROR, WARN, INFO, DEBUG or TRACE
- 进程ID
- 分隔符： `---` 标识实际日志的开始
- 线程名： 方括号括起来（可能会截断控制台输出）
- Logger名： 通常使用源代码的类名
- 日志内容

控制台输出

在Spring Boot中默认配置了 `ERROR` 、 `WARN` 和 `INFO` 级别的日志输出到控制台。

我们可以通过两种方式切换至 `DEBUG` 级别：

- 在运行命令后加入 `--debug` 标志，如： `$ java -jar myapp.jar --debug`
- 在 `application.properties` 中配置 `debug=true` ，该属性置为true的时候，核心Logger（包含嵌入式容器、hibernate、spring）会输出更多内容，但是你自己应用的日志并不会输出为DEBUG级别。

多彩输出

如果你的终端支持ANSI，设置彩色输出会让日志更具可读性。通过在 `application.properties` 中设置 `spring.output.ansi.enabled` 参数来支持。

- NEVER：禁用ANSI-colored输出（默认项）
- DETECT：会检查终端是否支持ANSI，是的话就采用彩色输出（推荐项）
- ALWAYS：总是使用ANSI-colored格式输出，若终端不支持的时候，会有很多干扰信息，不推荐使用

文件输出

Spring Boot默认配置只会输出到控制台，并不会记录到文件中，但是我们通常生产环境使用时都需要以文件方式记录。

若要增加文件输出，需要在 `application.properties` 中配置 `logging.file` 或 `logging.path` 属性。

- logging.file，设置文件，可以是绝对路径，也可以是相对路径。如： `logging.file=my.log`
- logging.path，设置目录，会在该目录下创建spring.log文件，并写入日志内容，
  如： `logging.path=/var/log`

日志文件会在10Mb大小的时候被截断，产生新的日志文件，默认级别为：ERROR、WARN、INFO

级别控制

在Spring Boot中只需要在 `application.properties` 中进行配置完成日志记录的级别控制。

配置格式： `logging.level.*=LEVEL`

- `logging.level` ： 日志级别控制前缀， `*` 为包名或Logger名
- `LEVEL` ： 选项TRACE, DEBUG, INFO, WARN, ERROR, FATAL, OFF

举例：

- `logging.level.com.didispace=DEBUG` ： `com.didispace` 包下所有class以DEBUG级别输出
- `logging.level.root=WARN` ： root日志以WARN级别输出

自定义日志配置

由于日志服务一般都在ApplicationContext创建前就初始化了，它并不是必须通过Spring的配置文件控制。因此通过系统属性和传统的Spring Boot外部配置文件依然可以很好的支持日志控制和管理。

根据不同的日志系统，你可以按如下规则组织配置文件名，就能被正确加载：

- Logback： `logback-spring.xml` ， `logback-spring.groovy` ， `logback.xml` ， `logback.groovy`
- Log4j： `log4j-spring.properties` ， `log4j-spring.xml` ， `log4j.properties` ， `log4j.xml`
- Log4j2： `log4j2-spring.xml` ， `log4j2.xml`
- JDK (Java Util Logging)： `logging.properties`

Spring Boot官方推荐优先使用带有 `-spring` 的文件名作为你的日志配置（如使用 `logback-spring.xml` ，而不是 `logback.xml` ）

自定义输出格式

在Spring Boot中可以通过在 `application.properties` 配置如下参数控制输出格式：

- logging.pattern.console：定义输出到控制台的样式（不支持JDK Logger）
- logging.pattern.file：定义输出到文件的样式（不支持JDK Logger）


持续更新...
关注 社区 · 轻松学习
长按识别ME

之前在Spring Boot日志管理 一文中主要介绍了Spring Boot中默认日志工具（logback）的基本配置内容。对于很多习惯使用log4j的开发者，Spring Boot依然可以很好的支持，只是需要做一些小小的配置功能。

引入log4j依赖

在创建Spring Boot工程时，我们引入了 `spring-boot-starter` ，其中包含了 `spring-boot-starter-logging` ，该依赖内容就是Spring Boot默认的日志框架Logback，所以我们在引入log4j之前，需要先排除该包的依赖，再引入log4j的依赖，就像下面这样：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <exclusions>
        <exclusion>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-logging</artifactId>
        </exclusion>
    </exclusions>
</dependency>

<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-log4j</artifactId>
</dependency>
```

配置log4j.properties

在引入了log4j依赖之后，只需要在 `src/main/resources` 目录下加入 `log4j.properties` 配置文件，就可以开始对应用的日志进行配置使用。

控制台输出

通过如下配置，设定root日志的输出级别为INFO，appender为控制台输出stdout

```
# LOG4J配置
log4j.rootCategory=INFO, stdout

# 控制台输出
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L -
%m%n
```

输出到文件

在开发环境，我们只是输出到控制台没有问题，但是到了生产或测试环境，或许持久化日志内容，方便追溯问题原因。可以通过添加如下的appender内容，按天输出到不同的文件中去，同时还需要为 `log4j.rootCategory` 添加名为file的appender，这样root日志就可以输出到 `logs/all.log` 文件中了。

```
#
log4j.rootCategory=INFO, stdout, file

# root日志输出
log4j.appender.file=org.apache.log4j.DailyRollingFileAppender
log4j.appender.file.file=logs/all.log
log4j.appender.file.DatePattern='.'yyyy-MM-dd
log4j.appender.file.layout=org.apache.log4j.PatternLayout
log4j.appender.file.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L - %m
%n
```

分类输出

当我们日志量较多的时候，查找问题会非常困难，常用的手段就是对日志进行分类，比如：

- 可以按不同package进行输出。通过定义输出到 `logs/my.log` 的appender，并对 `com.didispace` 包下的日志级别设定为DEBUG级别、appender设置为输出到 `logs/my.log` 的名为 `didifile` 的appender。

```
# com.didispace包下的日志配置
log4j.category.com.didispace=DEBUG, didifile

# com.didispace下的日志输出
log4j.appender.didifile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.didifile.file=logs/my.log
log4j.appender.didifile.DatePattern='.'yyyy-MM-dd
log4j.appender.didifile.layout=org.apache.log4j.PatternLayout
log4j.appender.didifile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L
---- %m%n
```

- 可以对不同级别进行分类，比如对ERROR级别输出到特定的日志文件中，具体配置可以如下。

```
log4j.logger.error=errorfile
# error日志输出
log4j.appender.errorfile=org.apache.log4j.DailyRollingFileAppender
log4j.appender.errorfile.file=logs/error.log
log4j.appender.errorfile.DatePattern='.'yyyy-MM-dd
log4j.appender.errorfile.Threshold = ERROR
log4j.appender.errorfile.layout=org.apache.log4j.PatternLayout
log4j.appender.errorfile.layout.ConversionPattern=%d{yyyy-MM-dd HH:mm:ss,SSS} %5p %c{1}:%L
 - %m%n
```

本文主要介绍如何在spring boot中引入log4j，以及一些基础用法，对于更多log4j的用法，还请参考log4j官方网站

本文完整示例Chapter4-2-2，可以通过运行单元测试来观察对不同内容的分类记录情况。

AOP为Aspect Oriented Programming的缩写，意为：面向切面编程，通过预编译方式和运行期动态代理实现程序功能的统一维护的一种技术。AOP是Spring框架中的一个重要内容，它通过对既有程序定义一个切入点，然后在其前后切入不同的执行内容，比如常见的有：打开数据库连接/关闭数据库连接、打开事务/关闭事务、记录日志等。基于AOP不会破坏原来程序逻辑，因此它可以很好的对业务逻辑的各个部分进行隔离，从而使得业务逻辑各部分之间的耦合度降低，提高程序的可重用性，同时提高了开发的效率。

下面主要讲两个内容，一个是如何在Spring Boot中引入Aop功能，二是如何使用Aop做切面去统一处理Web请求的日志。

以下所有操作基于chapter4-2-2工程进行。

准备工作

因为需要对web请求做切面来记录日志，所以先引入web模块，并创建一个简单的hello请求的处理。

- `pom.xml` 中引入web模块

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

- 实现一个简单请求处理：通过传入name参数，返回“hello xxx”的功能。

```
@RestController
public class HelloController {

    @RequestMapping(value = "/hello", method = RequestMethod.GET)
    @ResponseBody
    public String hello(@RequestParam String name) {
        return "Hello " + name;
    }

}
```

下面，我们可以对上面的/hello请求，进行切面日志记录。

引入AOP依赖

在Spring Boot中引入AOP就跟引入其他模块一样，非常简单，只需要在 `pom.xml` 中加入如下依赖：

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

在完成了引入AOP依赖包后，一般来说并不需要去做其他配置。也许在Spring中使用过注解配置方式的人会问是否需要在程序主类中增加 `@EnableAspectJAutoProxy` 来启用，实际并不需要。

可以看下面关于AOP的默认配置属性，其中 `spring.aop.auto` 属性默认是开启的，也就是说只要引入了AOP依赖后，默认已经增加了 `@EnableAspectJAutoProxy` 。

```
# AOP
spring.aop.auto=true # Add @EnableAspectJAutoProxy.
spring.aop.proxy-target-class=false # Whether subclass-based (CGLIB) proxies are to be cre
ated (true) as
 opposed to standard Java interface-based proxies (false).
```

而当我们需要使用CGLIB来实现AOP的时候，需要配置 `spring.aop.proxy-target-class=true` ，不然默认使用的是标准Java的实现。

实现Web层的日志切面

实现AOP的切面主要有以下几个要素：

- 使用 `@Aspect` 注解将一个java类定义为切面类
- 使用 `@Pointcut` 定义一个切入点，可以是一个规则表达式，比如下例中某个package下的所有函数，也可以是一个注解等。
- 根据需要在切入点不同位置的切入内容
  - 使用 `@Before` 在切入点开始处切入内容
  - 使用 `@After` 在切入点结尾处切入内容
  - 使用 `@AfterReturning` 在切入点return内容之后切入内容（可以用来对处理返回值做一些加工处理）
  - 使用 `@Around` 在切入点前后切入内容，并自己控制何时执行切入点自身的内容
  - 使用 `@AfterThrowing` 用来处理当切入内容部分抛出异常之后的处理逻辑

```
@Aspect
@Component
public class WebLogAspect {

    private Logger logger = Logger.getLogger(getClass());

    @Pointcut("execution(public * com.didispace.web..*.*(..))")
    public void webLog(){}

    @Before("webLog()")
    public void doBefore(JoinPoint joinPoint) throws Throwable {
        // 接收到请求，记录请求内容
        ServletRequestAttributes attributes = (ServletRequestAttributes) RequestContextHol
der.getRequestAttributes();
        HttpServletRequest request = attributes.getRequest();

        // 记录下请求内容
```

```
        logger.info("URL : " + request.getRequestURL().toString());
        logger.info("HTTP_METHOD : " + request.getMethod());
        logger.info("IP : " + request.getRemoteAddr());
        logger.info("CLASS_METHOD : " + joinPoint.getSignature().getDeclaringTypeName() +
"." + joinPoint.getSignature().getName());
        logger.info("ARGS : " + Arrays.toString(joinPoint.getArgs()));

    }

    @AfterReturning(returning = "ret", pointcut = "webLog()")
    public void doAfterReturning(Object ret) throws Throwable {
        // 处理完请求，返回内容
        logger.info("RESPONSE : " + ret);
    }

}
```

可以看上面的例子，通过 `@Pointcut` 定义的切入点为 `com.didispace.web` 包下的所有函数（对web层所有请求处理做切入点），然后通过 `@Before` 实现，对请求内容的日志记录（本文只是说明过程，可以根据需要调整内容），最后通过 `@AfterReturning` 记录请求返回的对象。

通过运行程序并访问： `http://localhost:8080/hello?name=didi` ，可以获得下面的日志输出

```
2016-05-19 13:42:13,156  INFO WebLogAspect:41 - URL : http://localhost:8080/hello
2016-05-19 13:42:13,156  INFO WebLogAspect:42 - HTTP_METHOD : http://localhost:8080/hello
2016-05-19 13:42:13,157  INFO WebLogAspect:43 - IP : 0:0:0:0:0:0:0:1
2016-05-19 13:42:13,160  INFO WebLogAspect:44 - CLASS_METHOD : com.didispace.web.HelloCont
roller.hello
2016-05-19 13:42:13,160  INFO WebLogAspect:45 - ARGS : [didi]
2016-05-19 13:42:13,170  INFO WebLogAspect:52 - RESPONSE:Hello didi
```

优化：AOP切面中的同步问题

在WebLogAspect切面中，分别通过doBefore和doAfterReturning两个独立函数实现了切点头部和切点返回后执行的内容，若我们想统计请求的处理时间，就需要在doBefore处记录时间，并在doAfterReturning处通过当前时间与开始处记录的时间计算得到请求处理的消耗时间。

那么我们是否可以在WebLogAspect切面中定义一个成员变量来给doBefore和doAfterReturning一起访问呢？是否会有同步问题呢?

的确，直接在这里定义基本类型会有同步问题，所以我们可以引入ThreadLocal对象，像下面这样进行记录：

```
@Aspect
@Component
public class WebLogAspect {

    private Logger logger = Logger.getLogger(getClass());
```

```
    ThreadLocal<Long> startTime = new ThreadLocal<>();

    @Pointcut("execution(public * com.didispace.web..*.*(..))")
    public void webLog(){}

    @Before("webLog()")
    public void doBefore(JoinPoint joinPoint) throws Throwable {
        startTime.set(System.currentTimeMillis());

        // 省略日志记录内容
    }

    @AfterReturning(returning = "ret", pointcut = "webLog()")
    public void doAfterReturning(Object ret) throws Throwable {
        // 处理完请求，返回内容
        logger.info("RESPONSE : " + ret);
        logger.info("SPEND TIME : " + (System.currentTimeMillis() - startTime.get()));
    }


}
```

优化：AOP切面的优先级

由于通过AOP实现，程序得到了很好的解耦，但是也会带来一些问题，比如：我们可能会对Web层做多个切面，校验用户，校验头信息等等，这个时候经常会碰到切面的处理顺序问题。

所以，我们需要定义每个切面的优先级，我们需要 `@Order(i)` 注解来标识切面的优先级。**i的值越小，优先级越高**。假设我们还有一个切面是 `CheckNameAspect` 用来校验name必须为didi，我们为其设置 `@Order(10)` ，而上文中WebLogAspect设置为 `@Order(5)` ，所以WebLogAspect有更高的优先级，这个时候执行顺序是这样的：

- 在 `@Before` 中优先执行 `@Order(5)` 的内容，再执行 `@Order(10)` 的内容
- 在 `@After` 和 `@AfterReturning` 中优先执行 `@Order(10)` 的内容，再执行 `@Order(5)` 的内容

所以我们可以这样子总结：

- 在切入点前的操作，按order的值由小到大执行
- 在切入点后的操作，按order的值由大到小执行

本文完整示例Chapter4-2-4


持续更新...
关注 社区 · 轻松学习
长按识别ME

在Spring Boot的众多Starter POMs中有一个特殊的模块，它不同于其他模块那样大多用于开发业务功能或是连接一些其他外部资源。它完全是一个用于暴露自身信息的模块，所以很明显，它的主要作用是用于监控与管理，它就是： `spring-boot-starter-actuator` 。

`spring-boot-starter-actuator` 模块的实现对于实施微服务的中小团队来说，可以有效地减少监控系统在采集应用指标时的开发量。当然，它也并不是万能的，有时候我们也需要对其做一些简单的扩展来帮助我们实现自身系统个性化的监控需求。下面，在本文中，我们将详解的介绍一些关于 `spring-boot-starter-actuator` 模块的内容，包括它的原生提供的端点以及一些常用的扩展和配置方式。

初识Actuator

下面，我们可以通过对快速入门中实现的Spring Boot应用增加 `spring-boot-starter-actuator` 模块功能，来对它有一个直观的认识。

在现有的Spring Boot应用中引入该模块非常简单，只需要在 `pom.xml` 的 `dependencies` 节点中，新增 `spring-boot-starter-actuator` 的依赖即可，具体如下：

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

通过增加该依赖之后，重新启动应用。此时，我们可以在控制台中看到如下图所示的输出：

```
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/trace || /trace.json],methods=[GET],produces=[application
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/info || /info.json],methods=[GET],produces=[application/j
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env/{name:.*}],methods=[GET],produces=[application/json]}
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/env || /env.json],methods=[GET],produces=[application/jso
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/dump || /dump.json],methods=[GET],produces=[application/j
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/autoconfig || /autoconfig.json],methods=[GET],produces=[a
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/health || /health.json],produces=[application/json]}" ont
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/mappings || /mappings.json],methods=[GET],produces=[appli
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/beans || /beans.json],methods=[GET],produces=[application
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/configprops || /configprops.json],methods=[GET],produces=
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/metrics/{name:.*}],methods=[GET],produces=[application/js
o.s.b.a.e.mvc.EndpointHandlerMapping     : Mapped "{[/metrics || /metrics.json],methods=[GET],produces=[applica
```

上图显示了一批端点定义，这些端点并非我们自己在程序中创建，而是由 `spring-boot-starter-actuator` 模块根据应用依赖和配置自动创建出来的监控和管理端点。通过这些端点，我们可以实时的获取应用的各项监控指标，比如：访问 `/health` 端点，我们可以获得如下返回的应用健康信息：

```
{
    "status": "UP",
    "diskSpace": {
        "status": "UP",
        "total": 491270434816,
        "free": 383870214144,
        "threshold": 10485760
    }
}
```

原生端点

通过在快速入门示例中添加 `spring-boot-starter-actuator` 模块，我们已经对它有了一个初步的认识。接下来，我们详细介绍一下 `spring-boot-starter-actuator` 模块中已经实现的一些原生端点。如果根据端点的作用来说，我们可以原生端点分为三大类：

- 应用配置类：获取应用程序中加载的应用配置、环境变量、自动化配置报告等与Spring Boot应用密切相关的配置类信息。
- 度量指标类：获取应用程序运行过程中用于监控的度量指标，比如：内存信息、线程池信息、HTTP请求统计等。
- 操作控制类：提供了对应用的关闭等操作类功能。

下面我们来详细了解一下这三类端点都分别可以为我们提供怎么样的有用信息和强大功能，以及我们如何去扩展和配置它们。

应用配置类

由于Spring Boot为了改善传统Spring应用繁杂的配置内容，采用了包扫描和自动化配置的机制来加载原本集中于xml文件中的各项内容。虽然这样的做法，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。而这类端点就可以帮助我们轻松的获取一系列关于Spring 应用配置内容的详细报告，比如：自动化配置的报告、Bean创建的报告、环境属性的报告等。

- /autoconfig：该端点用来获取应用的自动化配置报告，其中包括所有自动化配置的候选项。同时还列出了每个候选项自动化配置的各个先决条件是否满足。所以，该端点可以帮助我们方便的找到一些自动化配置为什么没有生效的具体原因。该报告内容将自动化配置内容分为两部分：
  - `positiveMatches` 中返回的是条件匹配成功的自动化配置
  - `negativeMatches` 中返回的是条件匹配不成功的自动化配置

```
{
    "positiveMatches": { // 条件匹配成功的
        "EndpointWebMvcAutoConfiguration": [
            {
                "condition": "OnClassCondition",
                "message": "@ConditionalOnClass classes found: javax.servlet.Servlet,o
rg.springframework.web.servlet.DispatcherServlet"
            },
            {
                "condition": "OnWebApplicationCondition",
                "message": "found web application StandardServletEnvironment"
            }
        ],
        ...
    },
    "negativeMatches": {  // 条件不匹配成功的
        "HealthIndicatorAutoConfiguration.DataSourcesHealthIndicatorConfiguration": [
            {
                "condition": "OnClassCondition",
```

```
                "message": "required @ConditionalOnClass classes not found: org.spring
framework.jdbc.core.JdbcTemplate"
            }
        ],
        ...
    }
}
```

从如上示例中我们可以看到，每个自动化配置候选项中都有一系列的条件，比如上面没有成功匹配的 `HealthIndicatorAutoConfiguration.DataSourcesHealthIndicatorConfiguration` 配置，它的先决条件就是需要在工程中包含 `org.springframework.jdbc.core.JdbcTemplate` 类，由于我们没有引入相关的依赖，它就不会执行自动化配置内容。所以，当我们发现有一些期望的配置没有生效时，就可以通过该端点来查看没有生效的具体原因。

- /beans：该端点用来获取应用上下文中创建的所有Bean。

```
[
    {
        "context": "hello:dev:8881",
        "parent": null,
        "beans": [
            {
                "bean": "org.springframework.boot.autoconfigure.web.DispatcherServletA
utoConfiguration$DispatcherServletConfiguration",
                "scope": "singleton",
                "type": "org.springframework.boot.autoconfigure.web.DispatcherServletA
utoConfiguration$DispatcherServletConfiguration$$EnhancerBySpringCGLIB$$3440282b",
                "resource": "null",
                "dependencies": [
                    "serverProperties",
                    "spring.mvc.CONFIGURATION_PROPERTIES",
                    "multipartConfigElement"
                ]
            },
            {
                "bean": "dispatcherServlet",
                "scope": "singleton",
                "type": "org.springframework.web.servlet.DispatcherServlet",
                "resource": "class path resource [org/springframework/boot/autoconfigu
re/web/DispatcherServletAutoConfiguration$DispatcherServletConfiguration.class]",
                "dependencies": []
            }
        ]
    }
]
```

如上示例中，我们可以看到在每个bean中都包含了下面这几个信息：

- bean：Bean的名称

    - scope：Bean的作用域
    - type：Bean的Java类型
    - reource：class文件的具体路径
    - dependencies：依赖的Bean名称
- /configprops：该端点用来获取应用中配置的属性信息报告。从下面该端点返回示例的片段中，我们看到返回了关于该短信的配置信息，`prefix`属性代表了属性的配置前缀，`properties`代表了各个属性的名称和值。所以，我们可以通过该报告来看到各个属性的配置路径，比如我们要关闭该端点，就可以通过使用`endpoints.configprops.enabled=false`来完成设置。

```
{
    "configurationPropertiesReportEndpoint": {
        "prefix": "endpoints.configprops",
        "properties": {
            "id": "configprops",
            "sensitive": true,
            "enabled": true
        }
    },
    ...
}
```

- /env：该端点与`/configprops`不同，它用来获取应用所有可用的环境属性报告。包括：环境变量、JVM属性、应用的配置配置、命令行中的参数。从下面该端点返回的示例片段中，我们可以看到它不仅返回了应用的配置属性，还返回了系统属性、环境变量等丰富的配置信息，其中也包括了应用还没有没有使用的配置。所以它可以帮助我们方便地看到当前应用可以加载的配置信息，并配合`@ConfigurationProperties`注解将它们引入到我们的应用程序中来进行使用。另外，为了配置属性的安全，对于一些类似密码等敏感信息，该端点都会进行隐私保护，但是我们需要让属性名中包含：password、secret、key这些关键词，这样该端点在返回它们的时候会使用`*`来替代实际的属性值。

```
{
    "profiles": [
        "dev"
    ],
    "server.ports": {
        "local.server.port": 8881
    },
    "servletContextInitParams": {

    },
    "systemProperties": {
        "idea.version": "2016.1.3",
        "java.runtime.name": "Java(TM) SE Runtime Environment",
        "sun.boot.library.path": "C:\\Program Files\\Java\\jdk1.8.0_91\\jre\\bin",
        "java.vm.version": "25.91-b15",
        "java.vm.vendor": "Oracle Corporation",
        ...
```

```
    },
    "systemEnvironment": {
        "configsetroot": "C:\\WINDOWS\\ConfigSetRoot",
        "RABBITMQ_BASE": "E:\\tools\\rabbitmq",
        ...
    },
    "applicationConfig: [classpath:/application-dev.properties]": {
        "server.port": "8881"
    },
    "applicationConfig: [classpath:/application.properties]": {
        "server.port": "8885",
        "spring.profiles.active": "dev",
        "info.app.name": "spring-boot-hello",
        "info.app.version": "v1.0.0",
        "spring.application.name": "hello"
    }
}
```

- /mappings：该端点用来返回所有Spring MVC的控制器映射关系报告。从下面的示例片段中，我们可以看该报告的信息与我们在启用Spring MVC的Web应用时输出的日志信息类似，其中 `bean` 属性标识了该映射关系的请求处理器， `method` 属性标识了该映射关系的具体处理类和处理函数。

```
{
    "/webjars/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**": {
        "bean": "resourceHandlerMapping"
    },
    "/**/favicon.ico": {
        "bean": "faviconHandlerMapping"
    },
    "{[/hello]}": {
        "bean": "requestMappingHandlerMapping",
        "method": "public java.lang.String com.didispace.web.HelloController.index()"
    },
    "{[/mappings || /mappings.json],methods=[GET],produces=[application/json]}": {
        "bean": "endpointHandlerMapping",
        "method": "public java.lang.Object org.springframework.boot.actuate.endpoint.m
vc.EndpointMvcAdapter.invoke()"
    },
    ...
}
```

- /info：该端点用来返回一些应用自定义的信息。默认情况下，该端点只会返回一个空的json内容。我们可以在 `application.properties` 配置文件中通过 `info` 前缀来设置一些属性，比如下面这样：

```
info.app.name=spring-boot-hello
info.app.version=v1.0.0
```

再访问 `/info` 端点，我们可以得到下面的返回报告，其中就包含了上面我们在应用自定义的两个参数。

```
{
    "app": {
        "name": "spring-boot-hello",
        "version": "v1.0.0"
    }
}
```

度量指标类

上面我们所介绍的应用配置类端点所提供的信息报告在应用启动的时候都已经基本确定了其返回内容，可以说是一个静态报告。而度量指标类端点提供的报告内容则是动态变化的，这些端点提供了应用程序在运行过程中的一些快照信息，比如：内存使用情况、HTTP请求统计、外部资源指标等。这些端点对于我们构建微服务架构中的监控系统非常有帮助，由于Spring Boot应用自身实现了这些端点，所以我们可以很方便地利用它们来收集我们想要的信息，以制定出各种自动化策略。下面，我们就来分别看看这些强大的端点功能。

- /metrics：该端点用来返回当前应用的各类重要度量指标，比如：内存信息、线程信息、垃圾回收信息等。

```
{
  "mem": 541305,
  "mem.free": 317864,
  "processors": 8,
  "instance.uptime": 33376471,
  "uptime": 33385352,
  "systemload.average": -1,
  "heap.committed": 476672,
  "heap.init": 262144,
  "heap.used": 158807,
  "heap": 3701248,
  "nonheap.committed": 65856,
  "nonheap.init": 2496,
  "nonheap.used": 64633,
  "nonheap": 0,
  "threads.peak": 22,
  "threads.daemon": 20,
  "threads.totalStarted": 26,
  "threads": 22,
  "classes": 7669,
  "classes.loaded": 7669,
  "classes.unloaded": 0,
```

```
  "gc.ps_scavenge.count": 7,
  "gc.ps_scavenge.time": 118,
  "gc.ps_marksweep.count": 2,
  "gc.ps_marksweep.time": 234,
  "httpsessions.max": -1,
  "httpsessions.active": 0,
  "gauge.response.beans": 55,
  "gauge.response.env": 10,
  "gauge.response.hello": 5,
  "gauge.response.metrics": 4,
  "gauge.response.configprops": 153,
  "gauge.response.star-star": 5,
  "counter.status.200.beans": 1,
  "counter.status.200.metrics": 3,
  "counter.status.200.configprops": 1,
  "counter.status.404.star-star": 2,
  "counter.status.200.hello": 11,
  "counter.status.200.env": 1
}
```

从上面的示例中，我们看到有这些重要的度量值：

- 系统信息：包括处理器数量 `processors` 、运行时间 `uptime` 和 `instance.uptime` 、系统平均负载 `systemload.average` 。
- `mem.*` ： 内存概要信息，包括分配给应用的总内存数量以及当前空闲的内存数量。这些信息来自 `java.lang.Runtime` 。
- `heap.*` ： 堆内存使用情况。这些信息来自 `java.lang.management.MemoryMXBean` 接口中 `getHeapMemoryUsage` 方法获取的 `java.lang.management.MemoryUsage` 。
- `nonheap.*` ： 非堆内存使用情况。这些信息来自 `java.lang.management.MemoryMXBean` 接口中 `getNonHeapMemoryUsage` 方法获取的 `java.lang.management.MemoryUsage` 。
- `threads.*` ： 线程使用情况，包括线程数、守护线程数（daemon）、线程峰值（peak）等，这些数据均来自 `java.lang.management.ThreadMXBean` 。
- `classes.*` ： 应用加载和卸载的类统计。这些数据均来自 `java.lang.management.ClassLoadingMXBean` 。
- `gc.*` ： 垃圾收集器的详细信息，包括垃圾回收次数 `gc.ps_scavenge.count` 、垃圾回收消耗时间 `gc.ps_scavenge.time` 、标记-清除算法的次数 `gc.ps_marksweep.count` 、标记-清除算法的消耗时间 `gc.ps_marksweep.time` 。这些数据均来自 `java.lang.management.GarbageCollectorMXBean` 。
- `httpsessions.*` ： Tomcat容器的会话使用情况。包括最大会话数 `httpsessions.max` 和活跃会话数 `httpsessions.active` 。该度量指标信息仅在引入了嵌入式Tomcat作为应用容器的时候才会提供。
- `gauge.*` ： HTTP请求的性能指标之一，它主要用来反映一个绝对数值。比如上面示例中的 `gauge.response.hello: 5` ，它表示上一次 `hello` 请求的延迟时间为5毫秒。
- `counter.*` ： HTTP请求的性能指标之一，它主要作为计数器来使用，记录了增加量和减少量。如上示例中 `counter.status.200.hello: 11` ，它代表了 `hello` 请求返回 `200` 状态的次数

为11。
对于 `gauge.*` 和 `counter.*` 的统计，这里有一个特殊的内容请求 `star-star` ，它代表了对静态资源的访问。这两类度量指标非常有用，我们不仅可以使用它默认的统计指标，还可以在程序中轻松的增加自定义统计值。只需要通过注入 `org.springframework.boot.actuate.metrics.CounterService` 和 `org.springframework.boot.actuate.metrics.GaugeService` 来实现自定义的统计指标信息。比如：我们可以像下面这样自定义实现对 `hello` 接口的访问次数统计。

```
@RestController
public class HelloController {

    @Autowired
    private CounterService counterService;

    @RequestMapping("/hello")
    public String greet() {
        counterService.increment("didispace.hello.count");
        return "";
    }

}
```

`/metrics` 端点可以提供应用运行状态的完整度量指标报告，这项功能非常的实用，但是对于监控系统中的各项监控功能，它们的监控内容、数据收集频率都有所不同，如果我们每次都通过全量获取报告的方式来收集，略显粗暴。所以，我们还可以通过 `/metrics/{name}` 接口来更细粒度的获取度量信息，比如我们可以通过访问 `/metrics/mem.free` 来获取当前可用内存数量。

- /health：该端点在一开始的示例中我们已经使用过了，它用来获取应用的各类健康指标信息。
在 `spring-boot-starter-actuator` 模块中自带实现了一些常用资源的健康指标检测器。这些检测器都通过 `HealthIndicator` 接口实现，并且会根据依赖关系的引入实现自动化装配，比如用于检测磁盘的 `DiskSpaceHealthIndicator` 、检测DataSource连接是否可用的 `DataSourceHealthIndicator` 等。有时候，我们可能还会用到一些Spring Boot的Starter POMs中还没有封装的产品来进行开发，比如：当使用RocketMQ作为消息代理时，由于没有自动化配置的检测器，所以我们需要自己来实现一个用来采集健康信息的检测器。比如，我们可以在Spring Boot的应用中，为 `org.springframework.boot.actuate.health.HealthIndicator` 接口实现一个对RocketMQ的检测器类：

```
@Component
public class RocketMQHealthIndicator implements HealthIndicator {

    @Override
    public Health health() {
        int errorCode = check();
        if (errorCode != 0) {
          return Health.down().withDetail("Error Code", errorCode).build();
        }
```

```
            return Health.up().build();
        }

        private int check() {
            // 对监控对象的检测操作
        }
}
```

通过重写 `health()` 函数来实现健康检查，返回的 `Heath` 对象中，共有两项内容，一个是状态信息，除了该示例中的 `UP` 与 `DOWN` 之外，还有 `UNKNOWN` 和 `OUT_OF_SERVICE` ，可以根据需要来实现返回；还有一个详细信息，采用Map的方式存储，在这里通过 `withDetail` 函数，注入了一个ErrorCode信息，我们也可以填入一下其他信息，比如，检测对象的IP地址、端口等。重新启动应用，并访问 `/health` 接口，我们在返回的JSON字符串中，将会包含了如下信息：

```
"rocketMQ": {
  "status": "UP"
}
```

- /dump：该端点用来暴露程序运行中的线程信息。它使用 `java.lang.management.ThreadMXBean` 的 `dumpAllThreads` 方法来返回所有含有同步信息的活动线程详情。
- /trace：该端点用来返回基本的HTTP跟踪信息。默认情况下，跟踪信息的存储采用 `org.springframework.boot.actuate.trace.InMemoryTraceRepository` 实现的内存方式，始终保留最近的100条请求记录。它记录的内容格式如下：

```
[
    {
        "timestamp": 1482570022463,
        "info": {
            "method": "GET",
            "path": "/metrics/mem",
            "headers": {
                "request": {
                    "host": "localhost:8881",
                    "connection": "keep-alive",
                    "cache-control": "no-cache",
                    "user-agent": "Mozilla/5.0 (Windows NT 10.0; WOW64) AppleWebKit/53
7.36 (KHTML, like Gecko) Chrome/53.0.2785.143 Safari/537.36",
                    "postman-token": "9817ea4d-ad9d-b2fc-7685-9dff1a1bc193",
                    "accept": "*/*",
                    "accept-encoding": "gzip, deflate, sdch",
                    "accept-language": "zh-CN,zh;q=0.8"
                },
                "response": {
                    "X-Application-Context": "hello:dev:8881",
                    "Content-Type": "application/json;charset=UTF-8",
                    "Transfer-Encoding": "chunked",
```

```
                    "Date": "Sat, 24 Dec 2016 09:00:22 GMT",
                    "status": "200"
                }
            }
        }
    },
    ...
]
```

操作控制类

仔细的读者可能会发现，我们在“初识Actuator”时运行示例的控制台中输出的所有监控端点，已经在介绍应用配置类端点和度量指标类端点时都讲解完了。那么还有哪些是操作控制类端点呢？实际上，由于之前介绍的所有端点都是用来反映应用自身的属性或是运行中的状态，相对于操作控制类端点没有那么敏感，所以他们默认都是启用的。而操作控制类端点拥有更强大的控制能力，如果要使用它们的话，需要通过属性来配置开启。

在原生端点中，只提供了一个用来关闭应用的端点： `/shutdown` 。我们可以通过如下配置开启它：

```
endpoints.shutdown.enabled=true
```

在配置了上述属性之后，只需要访问该应用的 `/shutdown` 端点就能实现关闭该应用的远程操作。由于开放关闭应用的操作本身是一件非常危险的事，所以真正在线上使用的时候，我们需要对其加入一定的保护机制，比如：定制Actuator的端点路径、整合Spring Security进行安全校验等。

摘要: 原创出处:www.bysocket.com 泥瓦匠BYSocket 希望转载，保留摘要，谢谢!

“看看星空，会觉得自己很渺小，可能我们在宇宙中从来就是一个偶然。所以，无论什么事情，仔细想一想，都没有什么大不了的。这能帮助自己在遇到挫折时稳定心态，想得更开。”-《腾讯传》

本文提纲

一、为啥整合 Dubbo 实现 SOA

Dubbo 不单单只是高性能的 RPC 调用框架，更是 SOA 服务治理的一种方案。

核心：

1. 远程通信，向本地调用一样调用远程方法。
2. 集群容错
3. 服务自动发现和注册，可平滑添加或者删除服务提供者。

我们常常使用 Springboot 暴露 HTTP 服务，并走 JSON 模式。但慢慢量大了，一种 SOA 的治理方案。这样可以暴露出 Dubbo 服务接口，提供给 Dubbo 消费者进行 RPC 调用。下面我们详解下如何集成 Dubbo。

二、运行 springboot-dubbo-server 和 springboot-dubbo-client 工程

运行环境：JDK 7 或 8，Maven 3.0+ 技术栈：SpringBoot 1.5+、Dubbo 2.5+、ZooKeeper 3.3+

1.ZooKeeper 服务注册中心

ZooKeeper 是一个分布式的，开放源码的分布式应用程序协调服务。它是一个为分布式应用提供一致性服务的软件，提供的功能包括：配置维护、域名服务、分布式同步、组服务等。

下载 ZooKeeper ，地址 http://www.apache.org/dyn/closer.cgi/zookeeper 解压 ZooKeeper

```
tar zxvf zookeeper-3.4.8.tar.gz
```

在 conf 目录新建 zoo.cfg ，照着该目录的 zoo_sample.cfg 配置如下。

```
cd zookeeper-3.3.6/conf
vim zoo.cfg
```

zoo.cfg 代码如下（自己指定 log 文件目录）：

```
tickTime=2000
```

```
dataDir=/javaee/zookeeper/data
dataLogDir=/javaee/zookeeper/log
clientPort=2181
```

在 bin 目录下，启动 ZooKeeper：

```
cd zookeeper-3.3.6/bin
./zkServer.sh start
```

2. git clone 下载工程 springboot-learning-example

项目地址见 GitHub - https://github.com/JeffLi1993/springboot-learning-example：

```
git clone git@github.com:JeffLi1993/springboot-learning-example.git
```

然后，Maven 编译安装这个工程：

```
cd springboot-learning-example
mvn clean install
```

3.运行 springboot-dubbo-server Dubbo 服务提供者工程

右键运行 springboot-dubbo-server 工程 ServerApplication 应用启动类的 main 函数。Console 中出现如下表示项目启动成功。这里表示 Dubbo 服务已经启动成功，并注册到 ZK （ZooKeeper）中。

4.运行 springboot-dubbo-client Dubbo 服务消费者工程

右键运行 springboot-dubbo-client 工程 ClientApplication 应用启动类的 main 函数。Console 中出现如下：

```
...
2017-03-01 16:31:38.473  INFO 9896 --- [           main] o.s.j.e.a.AnnotationMBeanExporter
        : Registering beans for JMX exposure on startup
2017-03-01 16:31:38.538  INFO 9896 --- [           main] s.b.c.e.t.TomcatEmbeddedServletCo
ntainer : Tomcat started on port(s): 8081 (http)
2017-03-01 16:31:38.547  INFO 9896 --- [           main] org.spring.springboot.ClientAppli
cation  : Started ClientApplication in 6.055 seconds (JVM running for 7.026)
City{id=1, provinceId=2, cityName='温岭', description='是我的故乡'}
```

最后打印的城市信息，就是通过 Dubbo 服务接口调用获取的。顺利运行成功，下面详解下各个代码及配置。

三、springboot-dubbo-server 和 springboot-dubbo-client 工程配置详解

代码都在 GitHub 上， https://github.com/JeffLi1993/springboot-learning-example。

1.详解 springboot-dubbo-server Dubbo 服务提供者工程

springboot-dubbo-server 工程目录结构

```
├── pom.xml
└── src
    └── main
        ├── java
        │   └── org
        │       └── spring
        │           └── springboot
        │               ├── ServerApplication.java
        │               ├── domain
        │               │   └── City.java
        │               └── dubbo
        │                   ├── CityDubboService.java
        │                   └── impl
        │                       └── CityDubboServiceImpl.java
        └── resources
            └── application.properties
```

a.pom.xml 配置 pom.xml 中依赖了 spring-boot-starter-dubbo 工程，该项目地址是 https://github.com/teaey/spring-boot-starter-dubbo。

pom.xml 配置如下

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>springboot</groupId>
    <artifactId>springboot-dubbo-server</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>springboot-dubbo 服务端:: 整合 Dubbo/ZooKeeper 详解 SOA 案例</name>

    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>

    <properties>
        <dubbo-spring-boot>1.0.0</dubbo-spring-boot>
```

```
    </properties>

    <dependencies>

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

        <!-- Spring Boot Web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <!-- Spring Boot Test 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

b.application.properties 配置

```
## Dubbo 服务提供者配置
spring.dubbo.application.name=provider
spring.dubbo.registry.address=zookeeper://127.0.0.1:2181
spring.dubbo.protocol.name=dubbo
spring.dubbo.protocol.port=20880
spring.dubbo.scan=org.spring.springboot.dubbo
```

这里 ZK 配置的地址和端口，就是上面本机搭建的 ZK 。如果有自己的 ZK 可以修改下面的配置。配置解释如下：

```
spring.dubbo.application.name 应用名称
spring.dubbo.registry.address 注册中心地址
spring.dubbo.protocol.name 协议名称
spring.dubbo.protocol.port 协议端口
```

```
spring.dubbo.scan dubbo 服务类包目录
```

c.CityDubboServiceImpl.java 城市业务 Dubbo 服务层实现层类

```
// 注册为 Dubbo 服务
@Service(version = "1.0.0")
public class CityDubboServiceImpl implements CityDubboService {

    public City findCityByName(String cityName) {
        return new City(1L,2L,"温岭","是我的故乡");
    }
}
```

@Service 注解标识为 Dubbo 服务，并通过 version 指定了版本号。

d.City.java 城市实体类 实体类通过 Dubbo 服务之间 RPC 调用，则需要实现序列化接口。最好指定下 serialVersionUID 值。

2.详解 springboot-dubbo-client Dubbo 服务消费者工程 springboot-dubbo-client 工程目录结构

```
├── pom.xml
└── src
    └── main
        ├── java
        │   └── org
        │       └── spring
        │           └── springboot
        │               ├── ClientApplication.java
        │               ├── domain
        │               │   └── City.java
        │               └── dubbo
        │                   ├── CityDubboConsumerService.java
        │                   └── CityDubboService.java
        └── resources
            └── application.properties
```

pom.xml 、 CityDubboService.java、City.java 没有改动。Dubbo 消费者通过引入接口实现 Dubbo 接口的调用。

a.application.properties 配置

```
## 避免和 server 工程端口冲突
server.port=8081

## Dubbo 服务消费者配置
spring.dubbo.application.name=consumer
spring.dubbo.registry.address=zookeeper://127.0.0.1:2181
spring.dubbo.scan=org.spring.springboot.dubbo
```

因为 springboot-dubbo-server 工程启动占用了 8080 端口，所以这边设置端口为 8081。

b.CityDubboConsumerService.java 城市 Dubbo 服务消费者

```
@Component
public class CityDubboConsumerService {

    @Reference(version = "1.0.0")
    CityDubboService cityDubboService;

    public void printCity() {
        String cityName="温岭";
        City city = cityDubboService.findCityByName(cityName);
        System.out.println(city.toString());
    }
}
```

@Reference(version = “1.0.0”) 通过该注解，订阅该接口版本为 1.0.0 的 Dubbo 服务。 这里将 CityDubboConsumerService 注入 Spring 容器，是为了更方便的获取该 Bean，然后验证这个 Dubbo 调用是否成功。

c.ClientApplication.java 客户端启动类

```
@SpringBootApplication
public class ClientApplication {

    public static void main(String[] args) {
        // 程序启动入口
        // 启动嵌入式的 Tomcat 并初始化 Spring 环境及其各 Spring 组件
        ConfigurableApplicationContext run = SpringApplication.run(ClientApplication.class
, args);
        CityDubboConsumerService cityService = run.getBean(CityDubboConsumerService.class)
;
        cityService.printCity();
    }
}
```

解释下这段逻辑，就是启动后从 Bean 容器中获取城市 Dubbo 服务消费者 Bean。然后调用该 Bean 方法去验证 Dubbo 调用是否成功。

四、小结

还有涉及到服务的监控，治理。这本质上和 SpringBoot 无关，所以这边不做一一介绍。感谢阿里 teaey 提供的 starter-dubbo 项目。


持续更新...
关注 社区 · 轻松学习
长按识别ME

继续上一篇：《 Springboot 整合 Dubbo/ZooKeeper 》，在 Spring Boot 使用 Dubbo Activate 扩展点。这是一个群友问的，我总结下，分享给更多人。

本文提纲 一、什么是 Dubbo Activate 注解 二、使用 Dubbo Activate 三、小结

运行环境：JDK 7 或 8，Maven 3.0+ **技术栈**：SpringBoot 1.5+、Dubbo 2.5+、ZooKeeper 3.3+

一、什么是 Dubbo Activate 注解

@Activate 是一个 Duboo 框架提供的注解。在 Dubbo 官方文档上有记载： 对于集合类扩展点，比如：Filter, InvokerListener, ExportListener, TelnetHandler, StatusChecker等， 可以同时加载多个实现，此时，可以用自动激活来简化配置。

用 @Activate 来实现一些 Filter ，可以具体如下：

1. 无条件自动激活

直接使用默认的注解即可

```
import com.alibaba.dubbo.common.extension.Activate;
import com.alibaba.dubbo.rpc.Filter;

@Activate // 无条件自动激活
public class XxxFilter implements Filter {
    // ...
}
```

1. 配置 xxx 参数，并且参数为有效值时激活，比如配了cache="lru"，自动激活 CacheFilter ```Java import com.alibaba.dubbo.common.extension.Activate; import com.alibaba.dubbo.rpc.Filter;

@Activate("xxx") // 当配置了xxx参数，并且参数为有效值时激活，比如配了cache="lru"，自动激活 CacheFilter。 public class XxxFilter implements Filter { // … }

```
3. 只对提供方激活，group 可选 provider 或 consumer
```Java
import com.alibaba.dubbo.common.extension.Activate;
import com.alibaba.dubbo.rpc.Filter;

@Activate(group = "provider", value = "xxx")
// 只对提供方激活，group可选"provider"或"consumer"
public class XxxFilter implements Filter {
    // ...
}
```

二、使用 Dubbo Activate 注解

基于以前的 springboot-dubbo-server 和 springboot-dubbo-client 工程，GitHub 地址：https://github.com/JeffLi1993/springboot-learning-example 。
```

这里我们在消费端，既 springboot-dubbo-client 工程上添加一个 Filter。代码如下：

```
package com.xxx;

import com.alibaba.dubbo.rpc.Filter;
import com.alibaba.dubbo.rpc.Invoker;
import com.alibaba.dubbo.rpc.Invocation;
import com.alibaba.dubbo.rpc.Result;
import com.alibaba.dubbo.rpc.RpcException;


public class XxxFilter implements Filter {
    public Result invoke(Invoker<?> invoker,
Invocation invocation) throws RpcException {
        // before filter ...
        Result result = invoker.invoke(invocation);
        // after filter ...
        return result;
    }
}
```

启动 client 工程发现，Console 报错，出现：

```
Caused by: java.lang.IllegalStateException: No such extension dubboConsumerFilter for filt
er/com.alibaba.dubbo.rpc.Filter
```

发现这个 Filter 初始化时，报错了。证明没有配置成功。

原来根据官方文档中描述，我们需要配置扩展点配置文件。

在 META-INF 中配置：

```
xxx=com.xxx.XxxFilter
```

Maven 项目目录结构

```
src
 |-main
    |-java
        |-com
            |-xxx
                |-XxxFilter.java (实现Filter接口)
    |-resources
        |-META-INF
            |-dubbo
                |-com.alibaba.dubbo.rpc.Filter (纯文本文件，内容为：xxx=com.xxx.XxxFilter)
```

三、小结

调用拦截扩展的应用场景很多，比如黑白名单，比如 IP 等。

摘要: 原创出处 www.bysocket.com 「泥瓦匠BYSocket 」欢迎转载，保留摘要，谢谢！

『 与其纠结，不如行动学习。Innovate ， And out execute ! 』

本文提纲 一、前言 二、applications.properties 配置清单 三、@Service 服务提供者常用配置 四、@Reference 服务消费者常用配置 五、小结

运行环境：JDK 7 或 8、Maven 3.0+ 技术栈：SpringBoot 1.5+、、Dubbo 2.5+

一、前言

在泥瓦匠出的 Springboot 整合 Dubbo/ZooKeeper 详解 SOA 案例

Spring Boot 中如何使用 Dubbo Activate 扩展点

两篇文章后，很多人跟我聊 Spring Boot 整合 Dubbo 的细节问题。当然最多的是配置问题，比如 Q：如果一个程序既提供服务又是消费者怎么配置 scan package？ A（群友周波）： 就是 com.xxx.provider 生产者，com.xxx.consumer 消费者，那么 scan package 就设置到 com.xxx

Q：如何设置消费者调用生产者的超时时间？ A：目前不能通过 application.properties 定义。@Reference timeout

Q：consumer 怎么配置接入多个 provider？ A：@Reference 可以指定不同的 register。register （注册中心 like provider container）里面可以对应多个 provider

Q： @Service(version = “1.0.0”) 这个 1.0.0 可以从 application.properties 配置文件中读取吗？可以区分不同的环境，可以统一升级管理 A：占时还没有解决… 但是应用环境，如：dev/test/run 可以使用下面的配置，在 application.properties 定义 spring.dubbo.application.environment

Spring Boot 整合 Dubbo 的项目依赖了 spring-boot-starter-dubbo 工程，该项目地址是 https://github.com/teaey/spring-boot-starter-dubbo。 感谢作者~

二、applications.properties 配置清单

根据 starter 工程源码，可以看出 application.properties 对应的 Dubbo 配置类 DubboProperties 。

```
@ConfigurationProperties(prefix = "spring.dubbo")
public class DubboProperties {

    private String scan;

    private ApplicationConfig application;

    private RegistryConfig registry;

    private ProtocolConfig protocol;
}
```

包括了扫描路径、应用配置类、注册中心配置类和服务协议类

所以具体常用配置下 扫描包路径：指的是 Dubbo 服务注解的服务包路径

```
## Dubbo 配置
# 扫描包路径
spring.dubbo.scan=org.spring.springboot.dubbo
```

应用配置类：关于 Dubbo 应用级别的配置

这里注意多个注册中心的配置方式。下面介绍单个注册中心的配置方式。

```
## Dubbo 应用配置
# 应用名称
spring.dubbo.application.name=xxx

# 模块版本
spring.dubbo.application.version=xxx

# 应用负责人
spring.dubbo.application.owner=xxx

# 组织名(BU或部门)
spring.dubbo.application.organization=xxx

# 分层
spring.dubbo.application.architecture=xxx

# 环境，如：dev/test/run
spring.dubbo.application.environment=xxx

# Java代码编译器
spring.dubbo.application.compiler=xxx

# 日志输出方式
spring.dubbo.application.logger=xxx

# 注册中心 0
spring.dubbo.application.registries[0].address=zookeeper:#127.0.0.1:2181=xxx
# 注册中心 1
spring.dubbo.application.registries[1].address=zookeeper:#127.0.0.1:2181=xxx

# 服务监控
spring.dubbo.application.monitor.address=xxx
```

注册中心配置类：常用 ZooKeeper 作为注册中心进行服务注册。

```
## Dubbo 注册中心配置类
# 注册中心地址
spring.dubbo.application.registries.address=xxx
```

```
# 注册中心登录用户名
spring.dubbo.application.registries.username=xxx

# 注册中心登录密码
spring.dubbo.application.registries.password=xxx

# 注册中心缺省端口
spring.dubbo.application.registries.port=xxx

# 注册中心协议
spring.dubbo.application.registries.protocol=xxx

# 客户端实现
spring.dubbo.application.registries.transporter=xxx

spring.dubbo.application.registries.server=xxx

spring.dubbo.application.registries.client=xxx

spring.dubbo.application.registries.cluster=xxx

spring.dubbo.application.registries.group=xxx

spring.dubbo.application.registries.version=xxx

# 注册中心请求超时时间(毫秒)
spring.dubbo.application.registries.timeout=xxx

# 注册中心会话超时时间(毫秒)
spring.dubbo.application.registries.session=xxx

# 动态注册中心列表存储文件
spring.dubbo.application.registries.file=xxx

# 停止时等候完成通知时间
spring.dubbo.application.registries.wait=xxx

# 启动时检查注册中心是否存在
spring.dubbo.application.registries.check=xxx

# 在该注册中心上注册是动态的还是静态的服务
spring.dubbo.application.registries.dynamic=xxx

# 在该注册中心上服务是否暴露
spring.dubbo.application.registries.register=xxx

# 在该注册中心上服务是否引用
spring.dubbo.application.registries.subscribe=xxx
```

服务协议配置类：

```
## Dubbo 服务协议配置


# 服务协议
spring.dubbo.application.protocol.name=xxx

# 服务IP地址(多网卡时使用)
spring.dubbo.application.protocol.host=xxx

# 服务端口
spring.dubbo.application.protocol.port=xxx

# 上下文路径
spring.dubbo.application.protocol.contextpath=xxx

# 线程池类型
spring.dubbo.application.protocol.threadpool=xxx

# 线程池大小(固定大小)
spring.dubbo.application.protocol.threads=xxx

# IO线程池大小(固定大小)
spring.dubbo.application.protocol.iothreads=xxx

# 线程池队列大小
spring.dubbo.application.protocol.queues=xxx

# 最大接收连接数
spring.dubbo.application.protocol.accepts=xxx

# 协议编码
spring.dubbo.application.protocol.codec=xxx

# 序列化方式
spring.dubbo.application.protocol.serialization=xxx

# 字符集
spring.dubbo.application.protocol.charset=xxx

# 最大请求数据长度
spring.dubbo.application.protocol.payload=xxx

# 缓存区大小
spring.dubbo.application.protocol.buffer=xxx

# 心跳间隔
spring.dubbo.application.protocol.heartbeat=xxx

# 访问日志
spring.dubbo.application.protocol.accesslog=xxx
```

```
# 网络传输方式
spring.dubbo.application.protocol.transporter=xxx

# 信息交换方式
spring.dubbo.application.protocol.exchanger=xxx

# 信息线程模型派发方式
spring.dubbo.application.protocol.dispatcher=xxx

# 对称网络组网方式
spring.dubbo.application.protocol.networker=xxx

# 服务器端实现
spring.dubbo.application.protocol.server=xxx

# 客户端实现
spring.dubbo.application.protocol.client=xxx

# 支持的telnet命令，多个命令用逗号分隔
spring.dubbo.application.protocol.telnet=xxx

# 命令行提示符
spring.dubbo.application.protocol.prompt=xxx

# status检查
spring.dubbo.application.protocol.status=xxx

# 是否注册
spring.dubbo.application.protocol.status=xxx
```

三、@Service 服务提供者常用配置

常用 @Service 配置的如下

```
version 版本
group 分组
provider 提供者
protocol 服务协议
monitor 服务监控
registry 服务注册
…
```

四、@Reference 服务消费者常用配置

常用 @Reference 配置的如下

```
version 版本
```

```
group 分组
timeout 消费者调用提供者的超时时间
consumer 服务消费者
monitor 服务监控
registry 服务注册
```

五、小结

主要介绍了 Spring Boot Dubbo 整合中的细节问题大集合。

运行环境：JDK 7 或 8，Maven 3.0+ 技术栈：SpringBoot 1.5+， Spring Data Elasticsearch 1.5+，ElasticSearch 2.3.2

本文提纲

- 一、搜索实战场景需求
- 二、运行 spring-data-elasticsearch-query 工程
- 三、spring-data-elasticsearch-query 工程代码详解

一、搜索实战场景需求

搜索的场景会很多，常用的搜索场景，需要搜索的字段很多，但每个字段匹配到后所占的权重又不同。比如电商网站的搜索，搜到商品名称和商品描述，自然商品名称的权重远远大于商品描述。而且单词匹配肯定不如短语匹配。这样就出现了新的需求，如何确定这些短语，即自然分词。那就利用分词器，即可得到所需要的短语，然后进行搜索。

下面介绍短语如何进行按权重分匹配搜索。

二、运行 spring-data-elasticsearch-query 工程

1. 后台起守护线程启动 Elasticsearch

```
cd elasticsearch-2.3.2/
./bin/elasticsearch -d
```

git clone 下载工程 springboot-elasticsearch，项目地址见 GitHub - https://github.com/JeffLi1993/ …ample。

下面开始运行工程步骤（Quick Start）：

2. 项目结构介绍

```
org.spring.springboot.controller - Controller 层
org.spring.springboot.repository - ES 数据操作层
org.spring.springboot.domain - 实体类
org.spring.springboot.service - ES 业务逻辑层
Application - 应用启动类
application.properties - 应用配置文件，应用启动会自动读取配置
```

本地启动的 ES，就不需要改配置文件了。如果连测试 ES 服务地址，需要修改相应 配置

3.编译工程

在项目根目录 spring-data-elasticsearch-query，运行 maven 指令：

```
mvn clean install
```

4.运行工程

右键运行 Application 应用启动类（位置：org/spring/springboot/Application.java）的 main 函数，这样就成功启动了 spring-data-elasticsearch-query 案例。 用 Postman 工具新增两个城市

a. 新增城市信息

```
POST http://127.0.0.1:8080/api/city
{
    "id”:"1",
    "score":"5",
    "name":"上海",
    "description":"上海是个热城市"
}
POST http://127.0.0.1:8080/api/city
{
    "id":"2",
    "score”:"4",
    "name”:”温岭",
    "description":”温岭是个沿海城市"
}
```

下面是实战搜索语句的接口：

```
GET http://localhost:8080/api/city ... nt%3D城市
```

获取返回结果： 返回 JSON 如下：

```
[
    {
        "id": 2,
        "name": "温岭",
        "description": "温岭是个沿海城市",
        "score": 4
    },
    {
        "id": 1,
        "name": "上海",
        "description": "上海是个好城市",
        "score": 3
    }
]
```

应用的控制台中，日志打印出查询语句的 DSL ：

```
DSL  =
{
 "function_score" : {
   "functions" : [ {
     "filter" : {
       "match" : {
         "name" : {
           "query" : "城市",
           "type" : "phrase"
         }
       }
     },
     "weight" : 1000.0
   }, {
     "filter" : {
       "match" : {
         "description" : {
           "query" : "城市",
           "type" : "phrase"
         }
       }
     },
     "weight" : 500.0
   } ],
   "score_mode" : "sum",
   "min_score" : 10.0
 }
}
```

三、spring-data-elasticsearch-query 工程代码详解

具体代码见 GitHub - https://github.com/JeffLi1993/springboot-learning-example

1.pom.xml 依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/ma ... gt%3B
    <modelVersion>4.0.0</modelVersion>
    <groupId>springboot</groupId>
    <artifactId>spring-data-elasticsearch-crud</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-data-elasticsearch-crud :: spring-data-elasticsearch - 基本案例 </name>
    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
```

```
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>
    <dependencies>
        <!-- Spring Boot Elasticsearch 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
        </dependency>
        <!-- Spring Boot Web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

这里依赖的 spring-boot-starter-data-elasticsearch 版本是 1.5.1.RELEASE，对应的 spring-data-elasticsearch 版本是 2.1.0.RELEASE。对应官方文档：http://docs.spring.io/spring-d … html/。后面数据操作层都是通过该 spring-data-elasticsearch 提供的接口实现。

application.properties 配置 ES 地址

```
# ES
spring.data.elasticsearch.repositories.enabled = true
spring.data.elasticsearch.cluster-nodes = 127.0.0.1:9300
```

默认 9300 是 Java 客户端的端口。9200 是支持 Restful HTTP 的接口。 更多配置：

- spring.data.elasticsearch.cluster-name Elasticsearch 集群名。(默认值: elasticsearch)
- spring.data.elasticsearch.cluster-nodes 集群节点地址列表，用逗号分隔。如果没有指定，就启动一个客户端节点。
- spring.data.elasticsearch.propertie 用来配置客户端的额外属性。
- spring.data.elasticsearch.repositories.enabled 开启 Elasticsearch 仓库。(默认值:true。)

3. ES 数据操作层

```
/**
 * ES 操作类
 * <p>
 * Created by bysocket on 17/05/2017.
 */
```

```
public interface CityRepository extends ElasticsearchRepository<City, Long> {
}
```

接口只要继承 ElasticsearchRepository 接口类即可，具体使用的是该接口的方法：

```
    Iterable<T> search(QueryBuilder query);
    Page<T> search(QueryBuilder query, Pageable pageable);
    Page<T> search(SearchQuery searchQuery);
    Page<T> searchSimilar(T entity, String[] fields, Pageable pageable);
```

4. 实体类

```
/**
 * 城市实体类
 * <p>
 * Created by bysocket on 03/05/2017.
 */
@Document(indexName = "province", type = "city")
public class City implements Serializable {
    private static final long serialVersionUID = -1L;
    /**
     * 城市编号
     */
    private Long id;
    /**
     * 城市名称
     */
    private String name;
    /**
     * 描述
     */
    private String description;
    /**
     * 城市评分
     */
    private Integer score;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
```

```
    }
    public Integer getScore() {
        return score;
    }
    public void setScore(Integer score) {
        this.score = score;
    }
}
```

注意 a. City 属性名不支持驼峰式。 b. indexName 配置必须是全部小写，不然会出异常。 org.elasticsearch.indices.InvalidIndexNameException: Invalid index name [provinceIndex], must be lowercase

5. 城市 ES 业务逻辑实现类

代码如下：

```
/**
 * 城市 ES 业务逻辑实现类
 * <p>
 * Created by bysocket on 20/06/2017.
 */
@Service
public class CityESServiceImpl implements CityService {
    private static final Logger LOGGER = LoggerFactory.getLogger(CityESServiceImpl.class);
    /* 分页参数 */
    Integer PAGE_SIZE = 12;          // 每页数量
    Integer DEFAULT_PAGE_NUMBER = 0; // 默认当前页码
    /* 搜索模式 */
    String SCORE_MODE_SUM = "sum"; // 权重分求和模式
    Float  MIN_SCORE = 10.0F;      // 由于无相关性的分值默认为 1 ，设置权重分最小值为 10
    @Autowired
    CityRepository cityRepository; // ES 操作类
    public Long saveCity(City city) {
        City cityResult = cityRepository.save(city);
        return cityResult.getId();
    }
    @Override
    public List<City> searchCity(Integer pageNumber, Integer pageSize, String searchConten
t) {
        // 校验分页参数
        if (pageSize == null || pageSize <= 0) {
            pageSize = PAGE_SIZE;
        }
        if (pageNumber == null || pageNumber < DEFAULT_PAGE_NUMBER) {
            pageNumber = DEFAULT_PAGE_NUMBER;
        }
        LOGGER.info("\n searchCity: searchContent [" + searchContent + "] \n ");
        // 构建搜索查询
```

```
        SearchQuery searchQuery = getCitySearchQuery(pageNumber,pageSize,searchContent);
        LOGGER.info("\n searchCity: searchContent [" + searchContent + "] \n DSL  = \n " +
 searchQuery.getQuery().toString());
        Page<City> cityPage = cityRepository.search(searchQuery);
        return cityPage.getContent();
    }
    /**
     * 根据搜索词构造搜索查询语句
     *
     * 代码流程:
     *      - 权重分查询
     *      - 短语匹配
     *      - 设置权重分最小值
     *      - 设置分页参数
     *
     * @param pageNumber 当前页码
     * @param pageSize 每页大小
     * @param searchContent 搜索内容
     * @return
     */
    private SearchQuery getCitySearchQuery(Integer pageNumber, Integer pageSize,String sea
rchContent) {
        // 短语匹配到的搜索词，求和模式累加权重分
        // 权重分查询 https://www.elastic.co/guide/c ... .html
        //   - 短语匹配 https://www.elastic.co/guide/c ... .html
        //   - 字段对应权重分设置，可以优化成 enum
        //   - 由于无相关性的分值默认为 1 ，设置权重分最小值为 10
        FunctionScoreQueryBuilder functionScoreQueryBuilder = QueryBuilders.functionScoreQ
uery()
                .add(QueryBuilders.matchPhraseQuery("name", searchContent),
                ScoreFunctionBuilders.weightFactorFunction(1000))
                .add(QueryBuilders.matchPhraseQuery("description", searchContent),
                ScoreFunctionBuilders.weightFactorFunction(500))
                .scoreMode(SCORE_MODE_SUM).setMinScore(MIN_SCORE);
        // 分页参数
        Pageable pageable = new PageRequest(pageNumber, pageSize);
        return new NativeSearchQueryBuilder()
                .withPageable(pageable)
                .withQuery(functionScoreQueryBuilder).build();
    }
}
```

可以看到该过程实现了，短语精准匹配以及匹配到根据字段权重分求和，从而实现按权重搜索查询。代码流程如下:

- 权重分查询
- 短语匹配
- 设置权重分最小值
- 设置分页参数

注意：

- 字段对应权重分设置，可以优化成 enum
- 由于无相关性的分值默认为 1 ，设置权重分最小值为 10

权重分查询文档：https://www.elastic.co/guide/c … .html。 短语匹配文档：https://www.elastic.co/guide/c … .html。

四、小结

Elasticsearch 还提供很多高级的搜索功能。这里提供下需要经常逛的相关网站： Elasticsearch 中文社区 https://elasticsearch.cn/topic/elasticsearch Elasticsearch: 权威指南-在线版 https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html

本文目录 一、Elasticsearch 基本术语 1.1 文档（Document）、索引（Index）、类型（Type）文档三要素 1.2 集群（Cluster）、节点（Node）、分片（Shard）分布式三要素 二、Elasticsearch 工作原理 2.1 文档存储的路由 2.2 如何健康检查 2.3 如何水平扩容 三、小结

> Spring For All 社区（spring4all.com）是新组建的关于 Spring 的纯技术交流社区。来社区找我吧。

一、Elasticsearch 基本术语

1.1 文档（Document）、索引（Index）、类型（Type）文档三要素 文档（Document） 文档，在面向对象观念就是一个对象。在 ES 里面，是一个大 JSON 对象，是指定了唯一 ID 的最底层或者根对象。文档的位置由 _index、_type 和 _id 唯一标识。

索引（Index） 索引，用于区分文档成组，即分到一组的文档集合。索引，用于存储文档和使文档可被搜索。比如项目存索引 project 里面，交易存索引 sales 等。

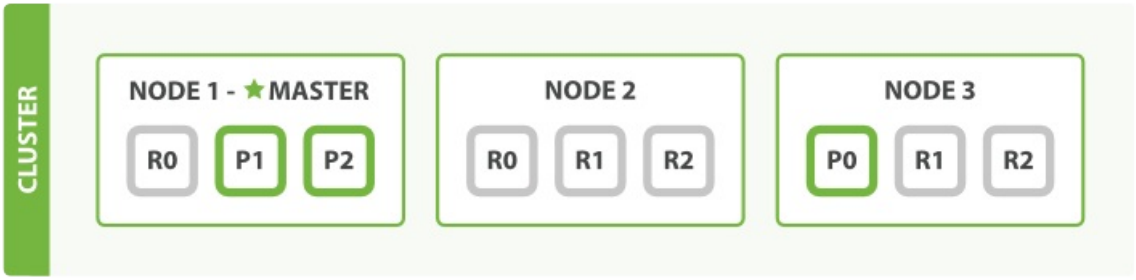
类型（Type） 类型，用于区分索引中的文档，即在索引中对数据逻辑分区。比如索引 project 的项目数据，根据项目类型 ui 项目、插画项目等进行区分。

和关系型数据库 MySQL 做个类比： Document 类似于 Record Type 类似于 Table Index 类似于 Database

1.2 集群（Cluster）、节点（Node）、分片（Shard）分布式三要素 集群（Cluster） 服务器集群大家都知道，这里 ES 也是类似的。多个 ElasticSearch 运行实例（节点）组合的组合体是 ElasticSearch 集群。 ElasticSearch 是天然的分布式，通过水平扩容为集群添加更多节点。 集群是去中心化的，有一个主节点（Master）。主节点是动态选举，因此不会出现单点故障。

那分片和节点的配置呢?

节点（Node） 一个 ElasticSearch 运行实例就是节点。顺着集群来，任何节点都可以被选举成为主节点。主节点负责集群内所以变更，比如索引的增加、删除等。所以集群不会因为主节点流量的增大成为瓶颈。因为任何节点都会成为主节点。 下面有 3 个节点，第 1 个节点有：2 个主分片和 1 个副分片。如图：



那么，只有一个节点的 ElasticSearch 服务会存在瓶颈。如图：

分片（Shard） 分片，是 ES 节点中最小的工作单元。分片仅仅保存全部数据的一部分，分片的集合是 ES 的索引。分片包括主分片和副分片，主分片是副分片的拷贝。主分片和副分片地工作基本没有大的区别。 在索引中全文搜索，然后会查询到每个分片，将每个分配的结果进行全局地收集处理，并返回。

二、Elasticsearch 工作原理

2.1 文档存储的路由 当索引到一个文档（如：报价系统），具体的文档数据（如：报价数据）会存储到一个分片。具体文档数据会被切分，并分别存储在分片 1 或者 分片 2 … 那么如何确定存在哪个分片呢?

存储路由过程由下面地公式决定：

```
shard = hash(routing) % number_of_primary_shards
```

routing 是可变值，支持自定义，默认文档 _id。 hash 函数生成数字，经过取余算法得到余数，那么这个余数就是分片的位置。 这是不是有点负载均衡的类似。

2.2 如何健康检查 集群名，集群的健康状态

```
GET http://127.0.0.1:9200/_cluster/stats
{
    "cluster_name": "elasticsearch",
    "status": "green",
    "timed_out": false,
    "number_of_nodes": 1,
    "number_of_data_nodes": 1,
    "active_primary_shards": 0,
    "active_shards": 0,
    "relocating_shards": 0,
    "initializing_shards": 0,
    "unassigned_shards": 0
}
```

status 字段是需要我们关心的。状态可能是下列三个值之一：

> green 所有的主分片和副本分片都已分配。你的集群是 100% 可用的。 yellow 所有的主分片已经分片了，但至少还有一个副本是缺失的。不会有数据丢失，所以搜索结果依然是完整的。高可用会弱化把 yellow 想象成一个需要及时调查的警告。 red 至少一个主分片（以及它的全部副本）都

> 在缺失中。这意味着你在缺少数据：搜索只能返回部分数据，而分配到这个分片上的写入请求会返回一个异常。

active_primary_shards 集群中的主分片数量 active_shards 所有分片的汇总值 relocating_shards 显示当前正在从一个节点迁往其他节点的分片的数量。通常来说应该是 0，不过在 Elasticsearch 发现集群不太均衡时，该值会上涨。比如说：添加了一个新节点，或者下线了一个节点。 initializing_shards 刚刚创建的分片的个数。 unassigned_shards 已经在集群状态中存在的分片。

2.3 如何水平扩容 主分片在索引创建已经确定。读操作可以同时被主分片和副分片处理。因此，更多的分片，会拥有更高的吞吐量。自然，需要增加更多的硬件资源支持吞吐量。说明，这里无法提高性能，因为每个分片获得的资源会变少。动态调整副本分片数，按需伸缩集群，比如把副本数默认值为 1 增加到 2：

```
PUT /blogs/_settings
{
    "number_of_replicas" : 2
}
```

三、小结 简单初探了下 ElasticSearch 的相关内容。后面会主要落地到实战，关于 spring-data-elasticsearch 这块的实战。

本文目录 一、spring-data-elasticsearch 是什么？ 1.1 Spring Data 1.2 Spring Data Elasticsearch 二、spring-data-elasticsearch 快速入门 2.1 pom.xml 依赖 2.2 ElasticsearchRepository 2.3 ElasticsearchTemplate 2.4 使用案例 三、spring-data-elasticsearch 和 elasticsearch 版本 四、小结

> 这里我们只是把人生大致分成“学习阶段”以及之后的“工作阶段”。 -《未来简史》

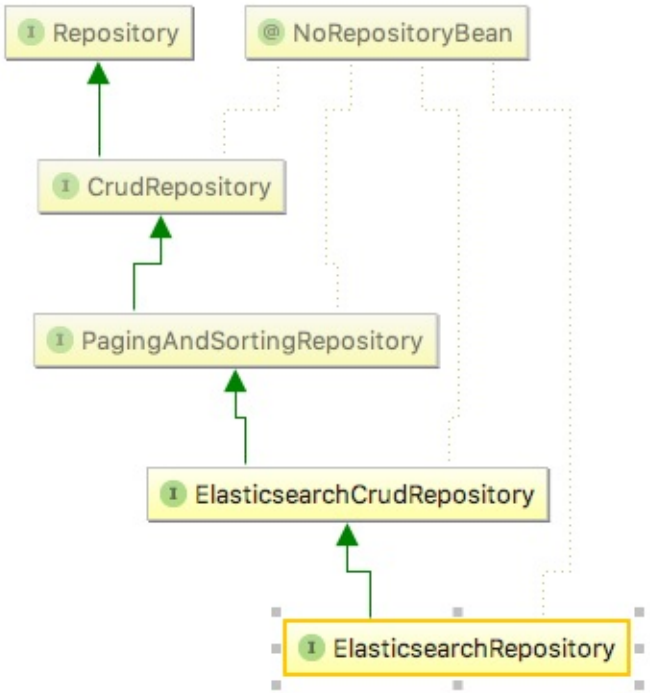
一、spring-data-elasticsearch 是什么？ 1.1 Spring Data 要了解 spring-data-elasticsearch 是什么，首先了解什么是 Spring Data。 Spring Data 基于 Spring 为数据访问提供一种相似且一致性的编程模型，并保存底层数据存储的。

1.2 Spring Data Elasticsearch spring-data-elasticsearch 是 Spring Data 的 Community modules 之一，是 Spring Data 对 Elasticsearch 引擎的实现。 Elasticsearch 默认提供轻量级的 HTTP Restful 接口形式的访问。相对来说，使用 HTTP Client 调用也很简单。但 spring-data-elasticsearch 可以更快的支持构建在 Spring 应用上，比如在 application.properties 配置 ES 节点信息和 spring-boot-starter-data-elasticsearch 依赖，直接在 Spring Boot 应用上使用。

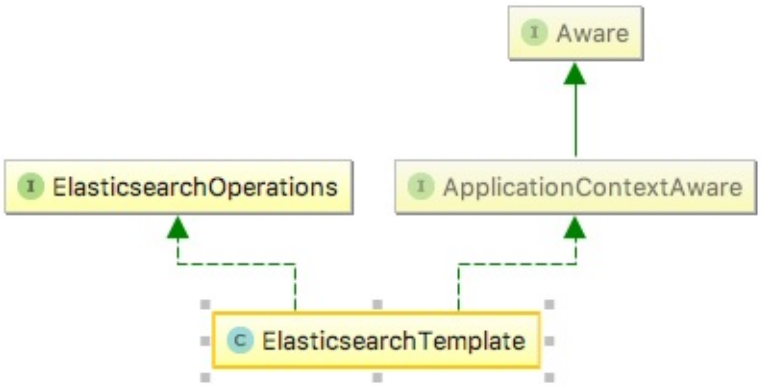
二、spring-data-elasticsearch 快速入门 2.1 pom.xml 依赖

```
<dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-elasticsearch</artifactId>
    <version>x.y.z.RELEASE</version>
</dependency>
```

2.2 ElasticsearchRepository ES 通用的存储接口的一种默认实现。Spring 根据接口定义的方法名，具体执行对应的数据存储实现。 ElasticsearchRepository 继承 ElasticsearchCrudRepository，ElasticsearchCrudRepository 继承 PagingAndSortingRepository。所以一般 CRUD 带分页已经支持。如图：



2.3 ElasticsearchTemplate ES 数据操作的中心支持类。和 JdbcTemplate 一样，几乎所有操作都可以使用 ElasticsearchTemplate 来完成。 ElasticsearchTemplate 实现了 ElasticsearchOperations 和 ApplicationContextAware 接口。ElasticsearchOperations 接口提供了 ES 相关的操作，并将 ElasticsearchTemplate 加入到 Spring 上下文。如图：



2.4 使用案例 拿官方案例来吧，详细介绍了 Book ES 对象的接口实现。 可以看出，book 拥有 name 和 price 两个属性。下面支持 name 和 price 列表 ES 查询，分页查询，范围查询等。还有可以利用注解实现 DSL 操作。

```
public interface BookRepository extends Repository<Book, String> {
    List<Book> findByNameAndPrice(String name, Integer price);
    List<Book> findByNameOrPrice(String name, Integer price);
    Page<Book> findByName(String name,Pageable page);
    Page<Book> findByNameNot(String name,Pageable page);
    Page<Book> findByPriceBetween(int price,Pageable page);
    Page<Book> findByNameLike(String name,Pageable page);
    @Query("{\"bool\" : {\"must\" : {\"term\" : {\"message\" : \"?0\"}}}}")
    Page<Book> findByMessage(String message, Pageable pageable);
}
```

三、spring-data-elasticsearch 和 elasticsearch 版本 SpringBoot 1.5+ 目前仅支持 ElasticSearch 2.3.2，所以如果想要使用最新的 ES。可以通过默认的轻量级的 HTTP 去调用实现。其版本对应如下:

spring data elasticsearch elasticsearch 3.0.0.BUILD-SNAPSHOT 5.4.0 2.0.4.RELEASE 2.4.0 2.0.0.RELEASE 2.2.0 1.4.0.M1 1.7.3 1.3.0.RELEASE 1.5.2 1.2.0.RELEASE 1.4.4 1.1.0.RELEASE 1.3.2 1.0.0.RELEASE 1.1.1

四、小结 本小结介绍了 spring-data-elasticsearch 是概述以及它的入门，还有 spring-data-elasticsearch 核心接口及版本的情况。

资料： 项目地址 https://github.com/spring-proj … earch 官方文档 http://docs.spring.io/spring-data/elasticsearch/docs/current/reference/html/


持续更新...
关注 社区 · 轻松学习
长按识别ME

『 风云说：能分享自己职位的知识的领导是个好领导。 』 **运行环境**：JDK 7 或 8，Maven 3.0+ **技术栈**：SpringBoot 1.5+， Spring Data Elasticsearch 1.5+ ，ElasticSearch 2.3.2 **本文提纲** 一、spring-data-elasticsearch-crud 的工程介绍 二、运行 spring-data-elasticsearch-crud 工程 三、spring-data-elasticsearch-crud 工程代码详解

一、spring-data-elasticsearch-crud 的工程介绍 spring-data-elasticsearch-crud 的工程，介绍 Spring Data Elasticsearch 简单的 ES 操作。Spring Data Elasticsearch 可以跟 JPA 进行类比。其使用方法也很简单。

二、运行 spring-data-elasticsearch-crud 工程 注意的是这里使用的是 ElasticSearch 2.3.2。是因为版本对应关系 https://github.com/spring-projects/spring-data-elasticsearch/wiki/Spring-Data-Elasticsearch---Spring-Boot---version-matrix;

Spring Boot Version (x) Spring Data Elasticsearch Version (y) Elasticsearch Version (z) x <= 1.3.5 y <= 1.3.4 z <= 1.7.2 *x >= 1.4.x 2.0.0 <=y < 5.0.0* ***2.0.0 <= z < 5.0.0*** \ - 只需要你修改下对应的 pom 文件版本号 ** - 下一个 ES 的版本会有重大的更新

1. 后台起守护线程启动 Elasticsearch

```
cd elasticsearch-2.3.2/
./bin/elasticsearch -d
```

git clone 下载工程 springboot-elasticsearch ，项目地址见 GitHub - https://github.com/JeffLi1993/ …ample。 下面开始运行工程步骤（Quick Start）：

1. 项目结构介绍

```
org.spring.springboot.controller - Controller 层
org.spring.springboot.repository - ES 数据操作层
org.spring.springboot.domain - 实体类
org.spring.springboot.service - ES 业务逻辑层
Application - 应用启动类
application.properties - 应用配置文件，应用启动会自动读取配置
```

本地启动的 ES ，就不需要改配置文件了。如果连测试 ES 服务地址，需要修改相应配置

3.编译工程 在项目根目录 spring-data-elasticsearch-crud，运行 maven 指令：

```
mvn clean install
```

4.运行工程 右键运行 Application 应用启动类（位置：/springboot-learning-example/springboot-elasticsearch/src/main/java/org/spring/springboot/Application.java）的 main 函数，这样就成功启动了 springboot-elasticsearch 案例。 用 Postman 工具新增两个城市

a. 新增城市信息

```
POST http://127.0.0.1:8080/api/city
{
```

```
    "id”:"1",
    "score":"5",
    "name":"上海",
    "description":"上海是个热城市"
}
POST http://127.0.0.1:8080/api/city
{
    "id":"2",
    "score”:"4",
    "name”:”温岭",
    "description":”温岭是个沿海城市"
}
```

可以打开 ES 可视化工具 head 插件：http://localhost:9200/_plugin/head/： （如果不知道怎么安装，请查阅 《Elasticsearch 和插件 elasticsearch-head 安装详解》 http://www.bysocket.com/?p=1744 。） 在「数据浏览」tab，可以查阅到 ES 中数据是否被插入，插入后的数据格式如下：

```
{
"_index": "cityindex",
"_type": "city",
"_id": "1",
"_version": 1,
"_score": 1,
"_source": {
  "id":"2",
    "score”:"4",
    "name”:”温岭",
    "description":”温岭是个沿海城市"
}
}
```

下面是基本查询语句的接口： a. 普通查询，查询城市描述

```
GET http://localhost:8080/api/city ... on%3D温岭
```

返回 JSON 如下：

```
[
    {
        "id": 2,
        "name": "温岭",
        "description": "温岭是个沿海城市",
        "score": 4
    }
]
```

b. AND 语句查询

```
GET http://localhost:8080/api/city ... on%3D温岭&score=4
```

返回 JSON 如下:

```
[
    {
        "id": 2,
        "name": "温岭",
        "description": "温岭是个沿海城市",
        "score": 4
    }
]
```

如果换成 score=5 ， 就没有结果了。

c. OR 语句查询

```
GET http://localhost:8080/api/city ... on%3D上海&score=4
```

返回 JSON 如下：

```
[
    {
        "id": 2,
        "name": "温岭",
        "description": "温岭是个沿海城市",
        "score": 4
    },
    {
        "id": 1,
        "name": "上海",
        "description": "上海是个好城市",
        "score": 3
    }
]
```

d. NOT 语句查询

```
GET http://localhost:8080/api/city ... on%3D温州
```

返回 JSON 如下：

```
[
    {
        "id": 2,
        "name": "温岭",
        "description": "温岭是个沿海城市",
```

```
        "score": 4
    },
    {
        "id": 1,
        "name": "上海",
        "description": "上海是个好城市",
        "score": 3
    }
]
```

e. LIKE 语句查询

```
GET http://localhost:8080/api/city ... on%3D城市
```

返回 JSON 如下：

```
[
    {
        "id": 2,
        "name": "温岭",
        "description": "温岭是个沿海城市",
        "score": 4
    },
    {
        "id": 1,
        "name": "上海",
        "description": "上海是个好城市",
        "score": 3
    }
]
```

三、spring-data-elasticsearch-crud 工程代码详解 具体代码见 GitHub - https://github.com/JeffLi1993/springboot-learning-example

1.pom.xml 依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/ma ... gt%3B
    <modelVersion>4.0.0</modelVersion>
    <groupId>springboot</groupId>
    <artifactId>spring-data-elasticsearch-crud</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-data-elasticsearch-crud :: spring-data-elasticsearch - 基本案例 </name>
    <!-- Spring Boot 启动父依赖 -->
    <parent>
```

```
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>
    <dependencies>
        <!-- Spring Boot Elasticsearch 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
        </dependency>
        <!-- Spring Boot Web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
</project>
```

这里依赖的 spring-boot-starter-data-elasticsearch 版本是 1.5.1.RELEASE，对应的 spring-data-elasticsearch 版本是 2.1.0.RELEASE。对应官方文档：http://docs.spring.io/spring-d … html/。后面数据操作层都是通过该 spring-data-elasticsearch 提供的接口实现。

2. application.properties 配置 ES 地址

```
# ES
spring.data.elasticsearch.repositories.enabled = true
spring.data.elasticsearch.cluster-nodes = 127.0.0.1:9300
默认 9300 是 Java 客户端的端口。9200 是支持 Restful HTTP 的接口。
```

更多配置： spring.data.elasticsearch.cluster-name Elasticsearch 集群名。(默认值: elasticsearch) spring.data.elasticsearch.cluster-nodes 集群节点地址列表，用逗号分隔。如果没有指定，就启动一个客户端节点。 spring.data.elasticsearch.propertie 用来配置客户端的额外属性。 spring.data.elasticsearch.repositories.enabled 开启 Elasticsearch 仓库。(默认值:true。)

3. ES 数据操作层

```
/**
 * ES 操作类
 * <p>
 * Created by bysocket on 17/05/2017.
 */
public interface CityRepository extends ElasticsearchRepository<City, Long> {
    /**
```

```
     * AND 语句查询
     *
     * @param description
     * @param score
     * @return
     */
    List<City> findByDescriptionAndScore(String description, Integer score);
    /**
     * OR 语句查询
     *
     * @param description
     * @param score
     * @return
     */
    List<City> findByDescriptionOrScore(String description, Integer score);
    /**
     * 查询城市描述
     *
     * 等同于下面代码
     * @Query("{\"bool\" : {\"must\" : {\"term\" : {\"description\" : \"?0\"}}}}")
     * Page<City> findByDescription(String description, Pageable pageable);
     *
     * @param description
     * @param page
     * @return
     */
    Page<City> findByDescription(String description, Pageable page);
    /**
     * NOT 语句查询
     *
     * @param description
     * @param page
     * @return
     */
    Page<City> findByDescriptionNot(String description, Pageable page);
    /**
     * LIKE 语句查询
     *
     * @param description
     * @param page
     * @return
     */
    Page<City> findByDescriptionLike(String description, Pageable page);
}
```

接口只要继承 ElasticsearchRepository 类即可。默认会提供很多实现，比如 CRUD 和搜索相关的实现。类似于 JPA 读取数据，是使用 CrudRepository 进行操作 ES 数据。支持的默认方法有： count(), findAll(), findOne(ID), delete(ID), deleteAll(), exists(ID), save(DomainObject), save(Iterable)。

另外可以看出，接口的命名是遵循规范的。常用命名规则如下： 关键字 方法命名 And findByNameAndPwd Or findByNameOrSex Is findById Between findByIdBetween Like findByNameLike NotLike findByNameNotLike OrderBy findByIdOrderByXDesc Not findByNameNot

4. 实体类

```
/**
 * 城市实体类
 * <p>
 * Created by bysocket on 03/05/2017.
 */
@Document(indexName = "province", type = "city")
public class City implements Serializable {
    private static final long serialVersionUID = -1L;
    /**
     * 城市编号
     */
    private Long id;
    /**
     * 城市名称
     */
    private String name;
    /**
     * 描述
     */
    private String description;
    /**
     * 城市评分
     */
    private Integer score;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public Integer getScore() {
        return score;
```

```
    }
    public void setScore(Integer score) {
        this.score = score;
    }
}
```

注意 a. City 属性名不支持驼峰式。 b. indexName 配置必须是全部小写，不然会出异常。org.elasticsearch.indices.InvalidIndexNameException: Invalid index name [provinceIndex], must be lowercase

四、小结 预告下 下一篇《深入浅出 spring-data-elasticsearch - 实战案例详解》，会带来实战项目中涉及到的权重分 & 短语精准匹配的讲解。

『 热烈的爱情到订婚早已是定点，婚一结一切了结。现在订了婚，彼此间还留着情感发展的余地，这是桩好事。- 《我们仨》 』

运行环境：JDK 7 或 8，Maven 3.0+ 技术栈：SpringBoot 1.5+， Spring Data Elasticsearch 1.5+ ，ElasticSearch 2.3.2

本文提纲 一、搜索实战场景需求 二、运行 spring-data-elasticsearch-query 工程 三、spring-data-elasticsearch-query 工程代码详解

一、搜索实战场景需求 搜索的场景会很多，常用的搜索场景，需要搜索的字段很多，但每个字段匹配到后所占的权重又不同。比如电商网站的搜索，搜到商品名称和商品描述，自然商品名称的权重远远大于商品描述。而且单词匹配肯定不如短语匹配。这样就出现了新的需求，如何确定这些短语，即自然分词。那就利用分词器，即可得到所需要的短语，然后进行搜索。 下面介绍短语如何进行按权重分匹配搜索。

二、运行 spring-data-elasticsearch-query 工程

1. 后台起守护线程启动 Elasticsearch

```
cd elasticsearch-2.3.2/
./bin/elasticsearch -d
```

git clone 下载工程 springboot-elasticsearch ，项目地址见 GitHub - https://github.com/JeffLi1993/ …ample。 下面开始运行工程步骤（Quick Start）：

1. 项目结构介绍

```
org.spring.springboot.controller - Controller 层
org.spring.springboot.repository - ES 数据操作层
org.spring.springboot.domain - 实体类
org.spring.springboot.service - ES 业务逻辑层
Application - 应用启动类
application.properties - 应用配置文件，应用启动会自动读取配置
```

本地启动的 ES ，就不需要改配置文件了。如果连测试 ES 服务地址，需要修改相应配置

3.编译工程 在项目根目录 spring-data-elasticsearch-query，运行 maven 指令：

```
mvn clean install
```

4.运行工程 右键运行 Application 应用启动类（位置：org/spring/springboot/Application.java）的 main 函数，这样就成功启动了 spring-data-elasticsearch-query 案例。 用 Postman 工具新增两个城市

a. 新增城市信息

```
POST http://127.0.0.1:8080/api/city
{
    "id”:"1",
    "score":"5",
```

```
    "name":"上海",
    "description":"上海是个热城市"
}
POST http://127.0.0.1:8080/api/city
{
    "id":"2",
    "score”:"4",
    "name”:”温岭",
    "description":”温岭是个沿海城市"
}
```

下面是实战搜索语句的接口： GET http://localhost:8080/api/city … nt%3D城市 获取返回结果： 返回 JSON 如下：

```
[
    {
        "id": 2,
        "name": "温岭",
        "description": "温岭是个沿海城市",
        "score": 4
    },
    {
        "id": 1,
        "name": "上海",
        "description": "上海是个好城市",
        "score": 3
    }
]
```

应用的控制台中，日志打印出查询语句的 DSL：

```
 DSL  =
 {
  "function_score" : {
    "functions" : [ {
      "filter" : {
        "match" : {
          "name" : {
            "query" : "城市",
            "type" : "phrase"
          }
        }
      },
      "weight" : 1000.0
    }, {
      "filter" : {
        "match" : {
          "description" : {
            "query" : "城市",
```

```
            "type" : "phrase"
          }
        }
      },
      "weight" : 500.0
    } ],
    "score_mode" : "sum",
    "min_score" : 10.0
  }
}
```

三、spring-data-elasticsearch-query 工程代码详解 具体代码见 GitHub - https://github.com/JeffLi1993/springboot-learning-example

1.pom.xml 依赖

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
hema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd
/ma ... gt%3B
    <modelVersion>4.0.0</modelVersion>
    <groupId>springboot</groupId>
    <artifactId>spring-data-elasticsearch-crud</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <name>spring-data-elasticsearch-crud :: spring-data-elasticsearch - 基本案例 </name>
    <!-- Spring Boot 启动父依赖 -->
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.5.1.RELEASE</version>
    </parent>
    <dependencies>
        <!-- Spring Boot Elasticsearch 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-elasticsearch</artifactId>
        </dependency>
        <!-- Spring Boot Web 依赖 -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!-- Junit -->
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
        </dependency>
    </dependencies>
```

```
</project>
```

这里依赖的 spring-boot-starter-data-elasticsearch 版本是 1.5.1.RELEASE，对应的 spring-data-elasticsearch 版本是 2.1.0.RELEASE。对应官方文档：http://docs.spring.io/spring-d … html/。后面数据操作层都是通过该 spring-data-elasticsearch 提供的接口实现。

1. application.properties 配置 ES 地址

```
# ES
spring.data.elasticsearch.repositories.enabled = true
spring.data.elasticsearch.cluster-nodes = 127.0.0.1:9300
```

默认 9300 是 Java 客户端的端口。9200 是支持 Restful HTTP 的接口。 更多配置：
spring.data.elasticsearch.cluster-name Elasticsearch 集群名。(默认值: elasticsearch)
spring.data.elasticsearch.cluster-nodes 集群节点地址列表，用逗号分隔。如果没有指定，就启动一个客户端节点。 spring.data.elasticsearch.propertie 用来配置客户端的额外属性。
spring.data.elasticsearch.repositories.enabled 开启 Elasticsearch 仓库。(默认值:true。)

3. ES 数据操作层

```
/**
 * ES 操作类
 * <p>
 * Created by bysocket on 17/05/2017.
 */
public interface CityRepository extends ElasticsearchRepository<City, Long> {
}
```

接口只要继承 ElasticsearchRepository 接口类即可，具体使用的是该接口的方法：

```
    Iterable<T> search(QueryBuilder query);
    Page<T> search(QueryBuilder query, Pageable pageable);
    Page<T> search(SearchQuery searchQuery);
    Page<T> searchSimilar(T entity, String[] fields, Pageable pageable);
```

1. 实体类

```
/**
 * 城市实体类
 * <p>
 * Created by bysocket on 03/05/2017.
 */
@Document(indexName = "province", type = "city")
public class City implements Serializable {
    private static final long serialVersionUID = -1L;
    /**
     * 城市编号
```

```
     */
    private Long id;
    /**
     * 城市名称
     */
    private String name;
    /**
     * 描述
     */
    private String description;
    /**
     * 城市评分
     */
    private Integer score;
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getDescription() {
        return description;
    }
    public void setDescription(String description) {
        this.description = description;
    }
    public Integer getScore() {
        return score;
    }
    public void setScore(Integer score) {
        this.score = score;
    }
}
```

注意 a. City 属性名不支持驼峰式。 b. indexName 配置必须是全部小写，不然会出异常。 org.elasticsearch.indices.InvalidIndexNameException: Invalid index name [provinceIndex], must be lowercase

1. 城市 ES 业务逻辑实现类 代码如下：

```
/**
 * 城市 ES 业务逻辑实现类
 * <p>
 * Created by bysocket on 20/06/2017.
```

```
 */
@Service
public class CityESServiceImpl implements CityService {
    private static final Logger LOGGER = LoggerFactory.getLogger(CityESServiceImpl.class);
    /* 分页参数 */
    Integer PAGE_SIZE = 12;          // 每页数量
    Integer DEFAULT_PAGE_NUMBER = 0; // 默认当前页码
    /* 搜索模式 */
    String SCORE_MODE_SUM = "sum"; // 权重分求和模式
    Float  MIN_SCORE = 10.0F;      // 由于无相关性的分值默认为 1 ，设置权重分最小值为 10
    @Autowired
    CityRepository cityRepository; // ES 操作类
    public Long saveCity(City city) {
        City cityResult = cityRepository.save(city);
        return cityResult.getId();
    }
    @Override
    public List<City> searchCity(Integer pageNumber, Integer pageSize, String searchConten
t) {
        // 校验分页参数
        if (pageSize == null || pageSize <= 0) {
            pageSize = PAGE_SIZE;
        }
        if (pageNumber == null || pageNumber < DEFAULT_PAGE_NUMBER) {
            pageNumber = DEFAULT_PAGE_NUMBER;
        }
        LOGGER.info("\n searchCity: searchContent [" + searchContent + "] \n ");
        // 构建搜索查询
        SearchQuery searchQuery = getCitySearchQuery(pageNumber,pageSize,searchContent);
        LOGGER.info("\n searchCity: searchContent [" + searchContent + "] \n DSL  = \n " +
 searchQuery.getQuery().toString());
        Page<City> cityPage = cityRepository.search(searchQuery);
        return cityPage.getContent();
    }
    /**
     * 根据搜索词构造搜索查询语句
     *
     * 代码流程：
     *      - 权重分查询
     *      - 短语匹配
     *      - 设置权重分最小值
     *      - 设置分页参数
     *
     * @param pageNumber 当前页码
     * @param pageSize 每页大小
     * @param searchContent 搜索内容
     * @return
     */
    private SearchQuery getCitySearchQuery(Integer pageNumber, Integer pageSize,String sea
rchContent) {
        // 短语匹配到的搜索词，求和模式累加权重分
```

```
        // 权重分查询 https://www.elastic.co/guide/c ... .html
        //   - 短语匹配 https://www.elastic.co/guide/c ... .html
        //   - 字段对应权重分设置，可以优化成 enum
        //   - 由于无相关性的分值默认为 1 ，设置权重分最小值为 10
        FunctionScoreQueryBuilder functionScoreQueryBuilder = QueryBuilders.functionScoreQ
uery()
                .add(QueryBuilders.matchPhraseQuery("name", searchContent),
                ScoreFunctionBuilders.weightFactorFunction(1000))
                .add(QueryBuilders.matchPhraseQuery("description", searchContent),
                ScoreFunctionBuilders.weightFactorFunction(500))
                .scoreMode(SCORE_MODE_SUM).setMinScore(MIN_SCORE);
        // 分页参数
        Pageable pageable = new PageRequest(pageNumber, pageSize);
        return new NativeSearchQueryBuilder()
                .withPageable(pageable)
                .withQuery(functionScoreQueryBuilder).build();
    }
}
```

可以看到该过程实现了，短语精准匹配以及匹配到根据字段权重分求和，从而实现按权重搜索查询。代码流程如下: - 权重分查询 - 短语匹配 - 设置权重分最小值 - 设置分页参数

注意： - 字段对应权重分设置，可以优化成 enum - 由于无相关性的分值默认为 1 ，设置权重分最小值为 10

权重分查询文档：https://www.elastic.co/guide/c … .html。 短语匹配文档：https://www.elastic.co/guide/c … .html。

四、小结 Elasticsearch 还提供很多高级的搜索功能。这里提供下需要经常逛的相关网站：Elasticsearch 中文社区 https://elasticsearch.cn/topic/elasticsearch Elasticsearch: 权威指南-在线版 https://www.elastic.co/guide/cn/elasticsearch/guide/current/index.html

图文简介

快速开始

1、Spring Boot 应用暴露监控指标【版本 1.5.7.RELEASE】

首先，添加依赖如下依赖：

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

        <dependency>
            <groupId>io.prometheus</groupId>
            <artifactId>simpleclient_spring_boot</artifactId>
            <version>0.0.26</version>
        </dependency>
```

然后，在启动类 `Application.java` 添加如下注解：

```
@SpringBootApplication
@EnablePrometheusEndpoint
@EnableSpringBootMetricsCollector
public class Application {

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

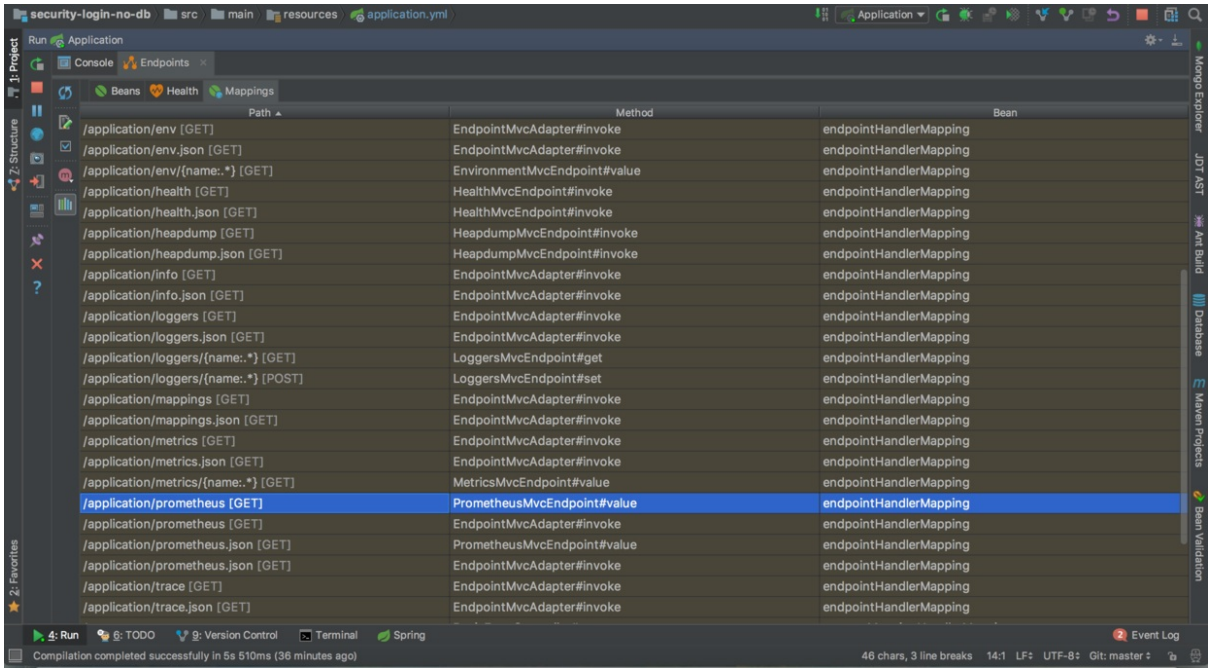
}
```

最后，配置默认的登录账号和密码，在 `application.yml` 中：

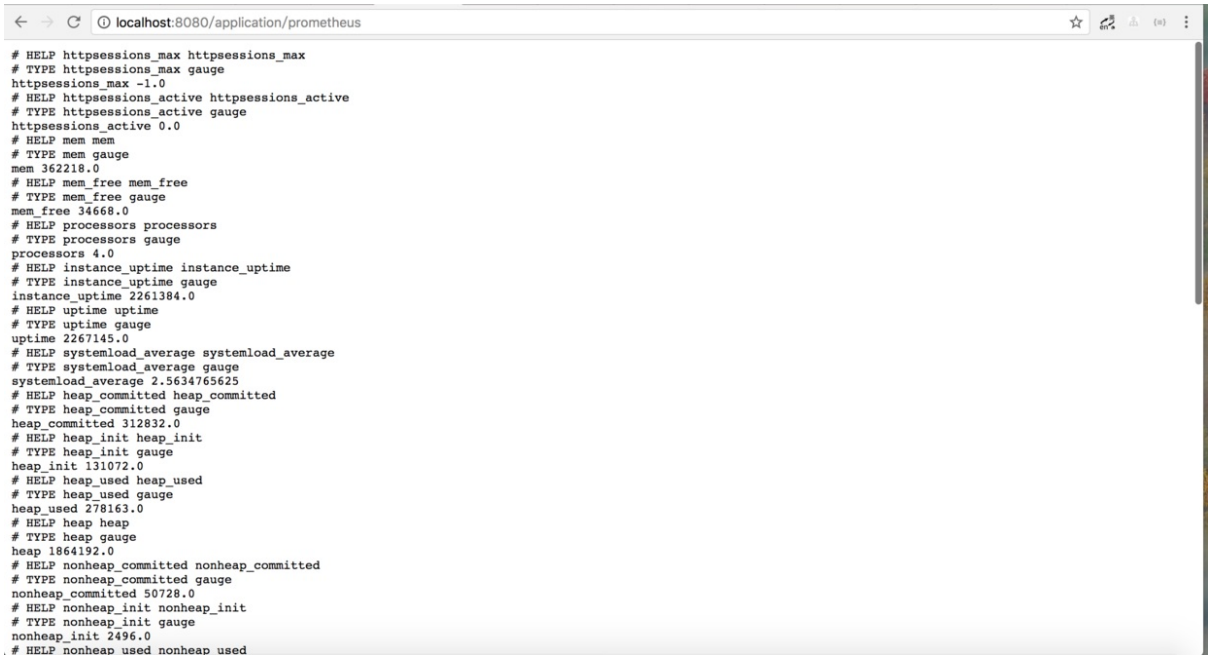
```
security:
  user:
    name: user
    password: pwd
```

> 提示：不建议配置 `management.security.enabled: false`

启动应用程序后，会看到如下一系列的 `Mappings`



利用账号密码访问 http://localhost:8080/application/prometheus ，可以看到 Prometheus 格式的指标数据



2、Prometheus 采集 Spring Boot 指标数据

首先，获取 Prometheus 的 Docker 镜像：

```
$ docker pull prom/prometheus
```

然后，编写配置文件 `prometheus.yml` ：

```
global:
  scrape_interval: 10s
```

```
  scrape_timeout: 10s
  evaluation_interval: 10m
scrape_configs:
  - job_name: spring-boot
    scrape_interval: 5s
    scrape_timeout: 5s
    metrics_path: /application/prometheus
    scheme: http
    basic_auth:
      username: user
      password: pwd
    static_configs:
      - targets:
        - 127.0.0.1:8080  #此处填写 Spring Boot 应用的 IP + 端口号
```

接着，启动 Prometheus：

```
$ docker run -d \
--name prometheus \
-p 9090:9090 \
-m 500M \
-v "$(pwd)/prometheus.yml":/prometheus.yml \
-v "$(pwd)/data":/data \
prom/prometheus \
-config.file=/prometheus.yml \
-log.level=info
```

最后，访问 http://localhost:9090/targets，检查 Spring Boot 采集状态是否正常。

3、Grafana 可视化监控数据

首先，获取 Grafana 的 Docker 镜像：

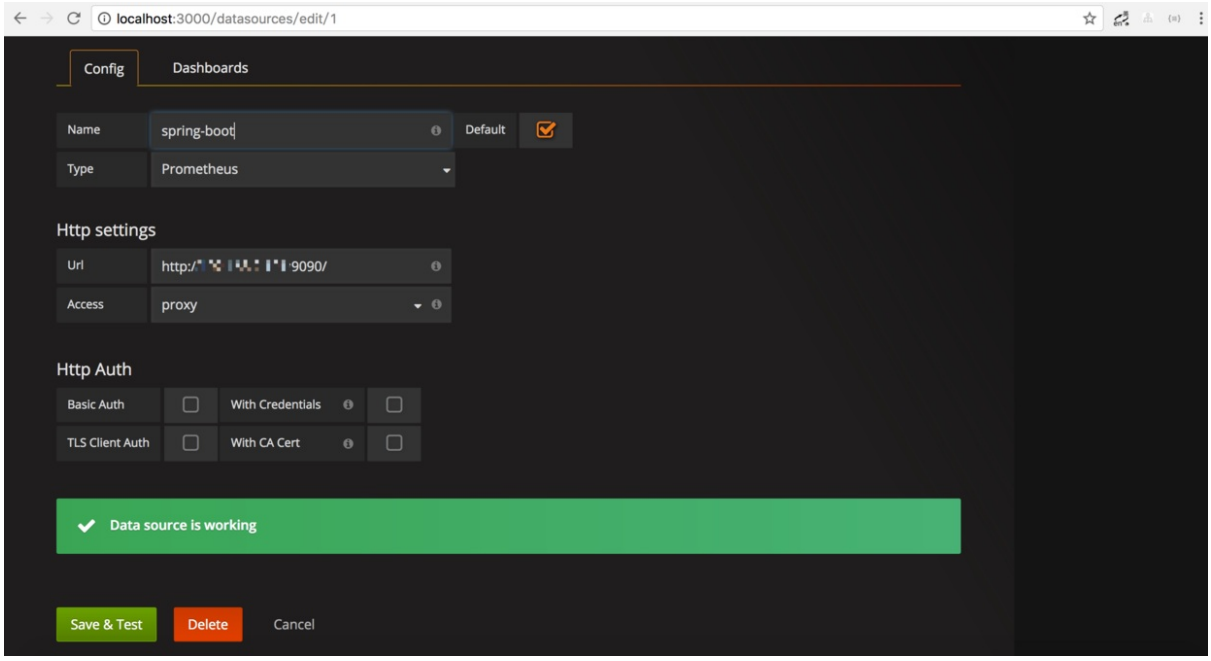
```
$ docker pull grafana/grafana
```

然后，启动 Grafana：

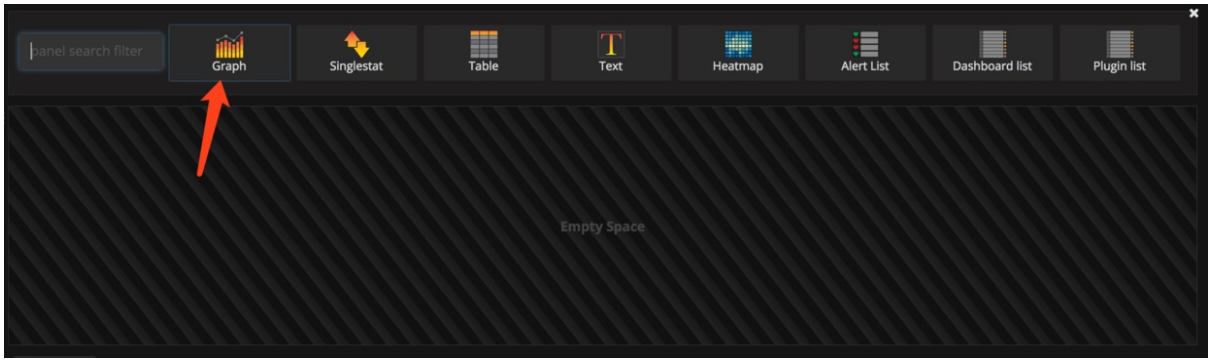
```
$ docker run --name grafana -d -p 3000:3000 grafana/grafana
```

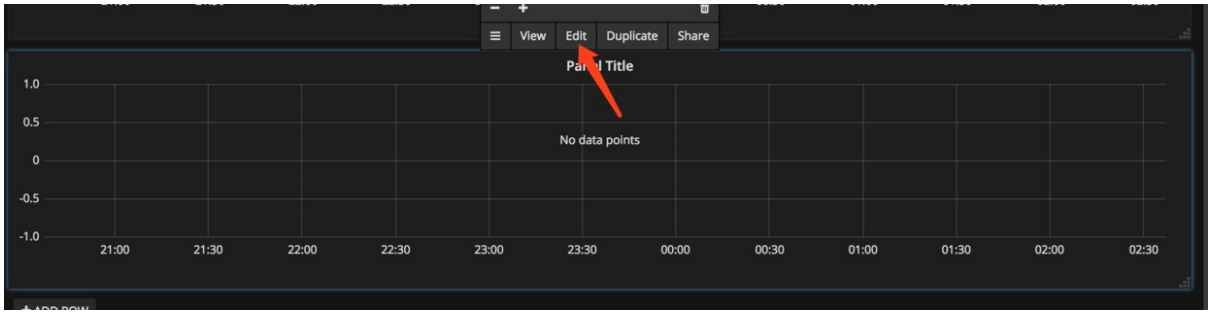
接着，访问 http://localhost:3000/ 配置 Prometheus 数据源：

> Grafana 登录账号 admin 密码 admin

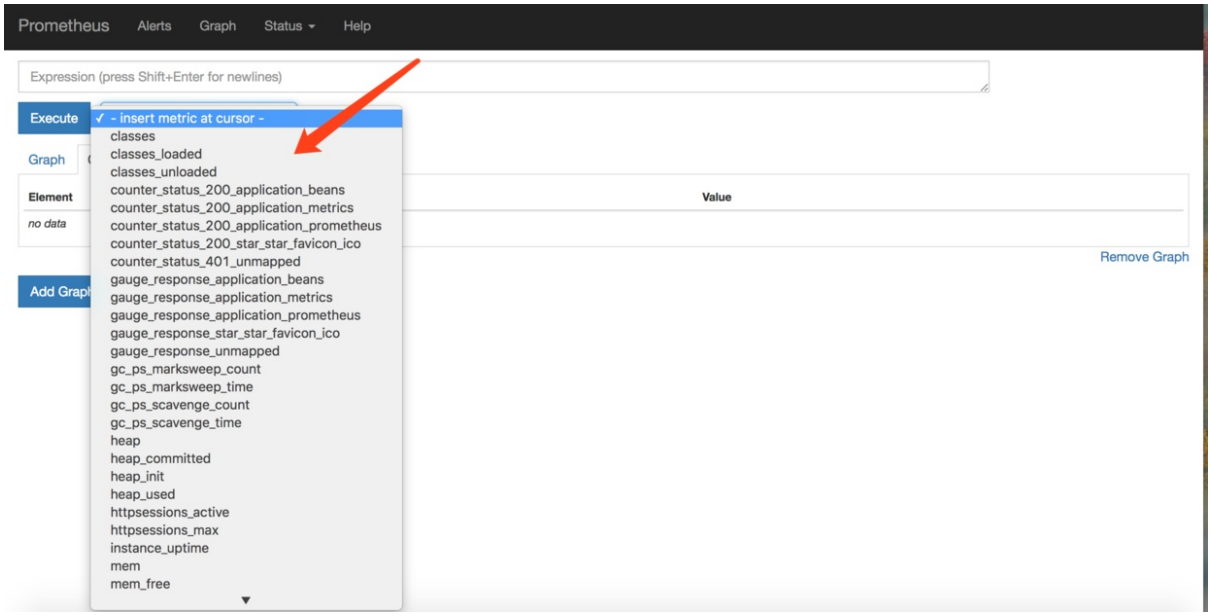


最后，配置单个指标的可视化监控面板：





提示，此处不能任意填写，只能填已有的指标点，具体的可以在 Prometheus 的首页看到，即 http://localhost:9090/graph



多配置几个指标之后，即可有如下效果：

参考文档

- prometheus 官方文档
- Grafana Docker 安装
- Spring Boot 官方文档
- https://prometheus.io/docs/introduction/overview/
- http://docs.grafana.org/installation/docker/
- http://projects.spring.io/spring-boot/