Table of Contents

简介

在您第1次接触和学习Spring框架的时候，是否因为其繁杂的配置而退却了？在你第n次使用Spring框架的时候，是否觉得一堆反复黏贴的配置有一些厌烦？那么您就不妨来试试使用Spring Boot来让你更易上手，更简单快捷地构建Spring应用！

Spring Boot让我们的Spring应用变的更轻量化。比如：你可以仅仅依靠一个Java类来运行一个Spring引用。你也可以打包你的应用为jar并通过使用java -jar来运行你的Spring Web应用。

Spring Boot的主要优点：

- 为所有Spring开发者更快的入门
- 开箱即用，提供各种默认配置来简化项目配置
- 内嵌式容器简化Web项目
- 没有冗余代码生成和XML配置的要求

快速入门

本章主要目标完成Spring Boot基础项目的构建，并且实现一个简单的Http请求处理，通过这个例子对Spring Boot有一个初步的了解，并体验其结构简单、开发快速的特性。

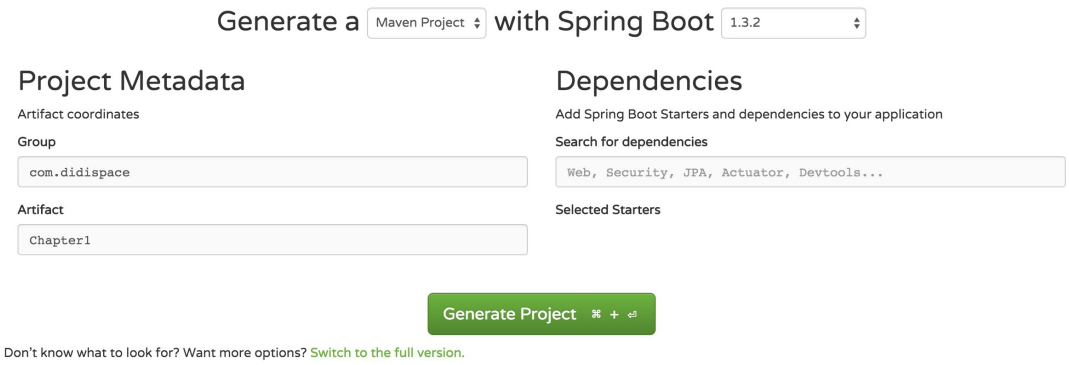
系统要求：

本书所有章节样例采用**Spring Boot 1.3.2**版本调试通过使用。

- Java 7及以上（本系列文章采用 `Java 1.8.0_73` ）
- Spring Framework 4.1.5及以上

使用**Maven**构建项目

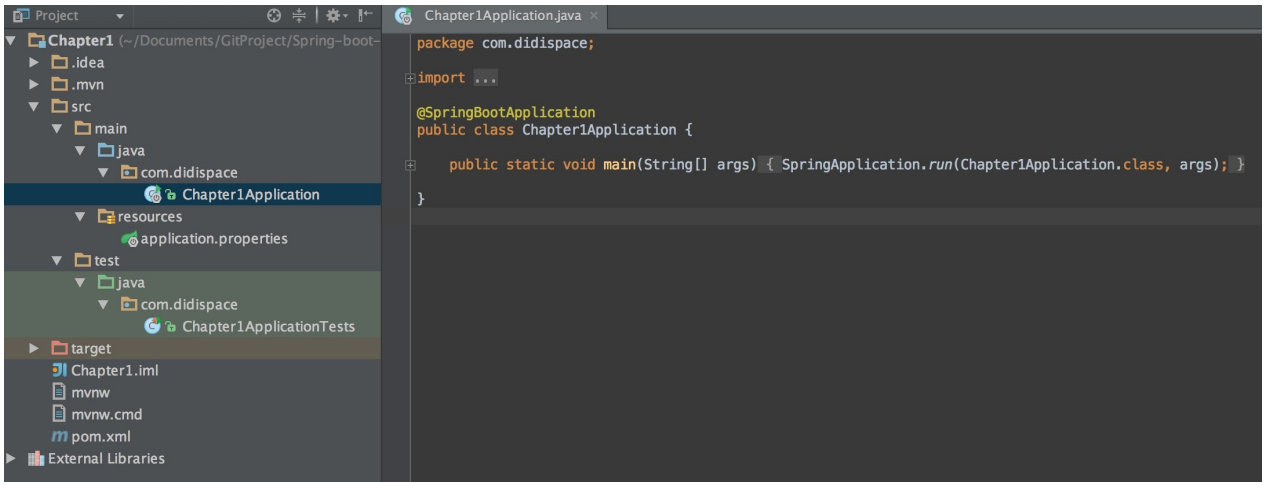
1. 通过 `SPRING INITIALIZR` 工具产生基础项目
   i. 访问： `http://start.spring.io/`
   ii. 选择构建工具 `Maven Project` 、Spring Boot版本 `1.3.2` 以及一些工程基本信息，可参考下图所示



   iii. 点击 `Generate Project` 下载项目压缩包
2. 解压项目包，并用IDE以 `Maven` 项目导入，以 `IntelliJ IDEA 14` 为例：
   i. 菜单中选择 `File` --> `New` --> `Project from Existing Sources...`
   ii. 选择解压后的项目文件夹，点击 `OK`
   iii. 点击 `Import project from external model` 并选择 `Maven` ，点击 `Next` 到底为止。
   iv. 若你的环境有多个版本的JDK，注意到选择 `Java SDK` 的时候请选

择 `Java 7` 以上的版本

项目结构解析



通过上面步骤完成了基础项目的创建，如上图所示，Spring Boot的基础结构共三个文件（具体路径根据用户生成项目时填写的Group所有差异）：

- `src/main/java` 下的程序入口：`Chapter1Application`
- `src/main/resources` 下的配置文件：`application.properties`
- `src/test/` 下的测试入口：`Chapter1ApplicationTests`

生成的 `Chapter1Application` 和 `Chapter1ApplicationTests` 类都可以直接运行来启动当前创建的项目，由于目前该项目未配合任何数据访问或Web模块，程序会在加载完Spring之后结束运行。

引入Web模块

当前的 `pom.xml` 内容如下，仅引入了两个模块：

- `spring-boot-starter` ：核心模块，包括自动配置支持、日志和YAML
- `spring-boot-starter-test` ：测试模块，包括JUnit、Hamcrest、Mockito

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

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

引入Web模块，需添加 `spring-boot-starter-web` 模块：

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

更多starter模块可见 `附件1:Starter POMs`

编写HelloWorld服务

- 创建 `package` 命名为 `com.didispace.web` （根据实际情况修改）
- 创建 `HelloController` 类，内容如下

```
@RestController
public class HelloController {

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

}
```

- 启动主程序，打开浏览器访问 `http://localhost:8080/hello` ，可以看到页面输出 `Hello World`

编写单元测试用例

打开的 `src/test/` 下的测试入口 `Chapter1ApplicationTests` 类。下面编写一个简单的单元测试来模拟http请求，具体如下：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MockServletContext.cla
ss)
@WebAppConfiguration
public class Chapter1ApplicationTests {

    private MockMvc mvc;

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

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

}
```

使用 `MockServletContext` 来构建一个空的 `WebApplicationContext` ，这样我们创建的 `HelloController` 就可以在 `@Before` 函数中创建并传递到 `MockMvcBuilders.standaloneSetup（）` 函数中。

- 注意引入下面内容，让 `status` 、 `content` 、 `equalTo` 函数可用

```
import static org.hamcrest.Matchers.equalTo;
import static org.springframework.test.web.servlet.result.MockMv
cResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMv
cResultMatchers.status;
```

至此已完成本章目标，通过Maven构建了一个空白Spring Boot项目，再通过引入web模块实现了一个简单的请求处理。

系统构建

基本要求：

- Java 7及以上
- Spring Framework 4.1.5及以上

如何支持Java 6

官方建议使用 `Java 8` ，若一定要使用 `Java 6` ，我们需要做一些额外的配置。

使用Tomcat

由于Spring Boot默认使用的是 `Tomcat 8` ，该容器要求 `Java 7` 版本以上，为了支持 `Java 6` 我们需要修改tomcat版本为7，通过修改Maven配置文件 `pom.xml` ，修改内容如下：

```
<properties>
    <tomcat.version>7.0.59</tomcat.version>
</properties>
```

使用Jetty

若您使用的是 `jetty` ,除了加入jetty的版本属性外，还要 `spring-boot-starter-web` 去除 `spring-boot-starter-tomcat` 的依赖，再引入 `spring-boot-starter-jetty` 的依赖，具体下面：

```
<properties>
    <jetty.version>8.1.15.v20140411</jetty.version>
    <jetty-jsp.version>2.2.0.v201112011158</jetty-jsp.version>
</properties>

<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifact
Id>
            </exclusion>
        </exclusions>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
        <exclusions>
            <exclusion>
                <groupId>org.eclipse.jetty.websocket</groupId>
                <artifactId>*</artifactId>
            </exclusion>
        </exclusions>
    </dependency>
</dependencies>
```

Servlet容器要求

推荐使用以下版本的Servlet容器

名称	**Servlet**版本	**Java**版本
Tomcat 8	3.1	Java 7+
Tomcat 7	3.0	Java 6+
Jetty 9	3.1	Java 7+
Jetty 8	3.0	Java 6+
Undertow 1.1	3.1	Java 7+

当然，你也可以将Spring Boot应用部署在其他支持Servlet 3.0以上的容器中。

工程结构

Spring Boot框架本身并没有对工程结构有特别的要求，但是按照最佳实践的工程结构可以帮助我们减少可能会遇见的坑，并且您会在后面章节中体会到该工程结构可以免去不少特殊的配置工作。

典型示例

- root package结构：`com.example.myproject`
- 应用主类`Application.java`置于root package下，通常我们会在应用主类中做一些框架配置扫描等配置，我们放在root package下可以帮助程序减少手工配置来加载到我们希望被Spring加载的内容
- 实体（Entity）与数据访问层（Repository）置于`com.example.myproject.domain`包下
- 逻辑层（Service）置于`com.example.myproject.service`包下
- Web层（web）置于`com.example.myproject.controller`包下

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

框架配置

我们在使用Spring的时候，需要做一系列的配置，通常使用XML或者Java来配置。

Spring Boot更偏向于使用Java来进行配置，从本文开始的入门项目中可以看到，就没有出现XML格式的Spring配置文件。

加载配置

从入门例子中的主类看，`SpringApplication.run`函数传入的`Chapter1Application.class`类就是要读取的配置类，该函数通过传入配置类来进行Spring的基础配置加载。

```
@SpringBootApplication
public class Chapter1Application {

    public static void main(String[] args) {
        SpringApplication.run(Chapter1Application.class, args);
    }

}
```

Spring中的配置类需要以`@Configuration`注解修饰，这里的`@SpringBootApplication`能被正常加载，是由于其整合了`@Configuration`，`@Configuration`，`@Configuration`三个注解。

多配置文件

从`SpringApplication.run`函数的传参看，可以加载多个配置class，所以框架配置内容不是必须都配置于程序主类之中。

当我们创建多个配置类时，可以有两种方法进行引用：

- 在主类中通过`@Import`注解来引入附加的配置类
- 在主类中配置`@ComponentScan`对附加配置类所在的package进行扫描加载

自动配置

Spring Boot自动配置的意图是根据项目中JAR的依赖关系自动配置你的Spring应用程序。例如：当HSQLDB在你的classpath中，并且你没有配置数据源的bean，那么spring boot会自动的为你配置一个嵌入式数据源。

使用自动配置功能：需要在配置类上添加 `@EnableAutoConfiguration` 或者 `@SpringBootApplication` 注解。

替代特定的自动配置

Spring Boot的自动配置通常并不能完全覆盖我们实际项目的配置需要，有的时候我们需要定义一些配置来取代自动配置的部分特定内容。例如：我们需要配置一个自己的数据源bean来替代默认嵌入式数据源。

如果你需要找出当前程序用了哪些自动化配置？那么可以通过bebug模式启动，控制台中会记录自动配置的报告。

禁用特定的自动配置

如果你发现有些自动配置内容你不想要，那么你可以通过 `@EnableAutoConfiguration` 注解来禁用他们

```
@Configuration
@EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.cl
ass})
public class MyConfiguration {

}
```

依赖注入

在Spring Boot中可以自由的使用Spring的注解来定义bean和进行依赖注入配置。

在配置文件中我们可以通过 `@ComponentScan` 配置来发现和创建bean，在关联类中通过 `@Autowired` 来注入bean。

如果你的工程结构是按上一节中的建议来构建的（主函数配置与**root package**下），那么 **`@ComponentScan`** 就不需要配置任何参数。你应用中的所有配置的构建（ **`@Component`** , **`@Service`** , **`@Repository`** , **`@Controller`** 等）都会自动地被注册为**Spring Bean**

> 欠你一个例子！！

具体的Spring IOC详见Spring Framework文档。

开发工具

这里的开发工具不是IDE，而是Spring Boot提供的一个模块 `spring-boot-devtools` ，它包含了一套能够帮助我们快速开发调试的效率工具。

我们只需要在Maven配置文件中加入下面依赖就可以使用它。

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

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

默认属性

自动重启

该工程在spring boot开发过程中非常有用，当工程文件发生变化的时候工程能够自动重启生效变化的内容。

除了引入 `spring-boot-devtools` 模块之外，还需修改pom.xml如下内容：

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

重启触发机制

在Eclipse和IntelliJ IDEA中重启的触发机制有所区别：

- Eclipse在保存修改过的文件时触发
- IntelliJ IDEA在build项目的时候触发

注意不要以**java -jar**的方式执行，不然不会启用**“**自动重启**”**功能，可以用**mvn spring-boot:run**来运行

全局设置

远程应用

开发进阶

Web开发

Spring Boot对于Web开发有相当好的支持。开发者只需要在 `pom.xml` 中加入 `spring-boot-starter-web` 模块，在默认配置和嵌入式容器的帮助下能快速运行起来，就如我们的快速入门案例。

本章节在之前的例子基础上对Web开发在几个常用方面做一些详细介绍。

Spring Boot的Web模块的核心实际上就是Spring Web MVC框架（后文简称：Spring MVC）。由于Spring MVC内容较多，本文无法完全覆盖所有Spring MVC的内容，主要介绍一些示例中需要用到的内容，更多Spring MVC的用法还需查看Spring Web MVC框架的官方文档。

构建RESTful API

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.cla
ss)
@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\",\"age\":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。

渲染**Web**视图

在之前的示例中，我们都是通过@RestController来处理请求，所以返回的内容为json对象。那么如果需要渲染html页面的时候，要如何实现呢？

模板引擎

在动态HTML实现上Spring Boot依然可以完美胜任，并且提供了多种模板引擎的默认配置支持，所以在推荐的模板引擎下，我们可以很快的上手开发动态网站。

Spring Boot提供了默认配置的模板引擎主要有以下几种：

- Thymeleaf
- FreeMarker
- Velocity
- Groovy
- Mustache

Spring Boot建议使用这些模板引擎，避免使用**JSP**，若一定要使用**JSP**将无法实现**Spring Boot**的多种特性，具体可见后文：支持**JSP**的配置

当你使用上述模板引擎中的任何一个，它们默认的模板配置路径为：`src/main/resources/templates`。当然也可以修改这个路径，具体如何修改，可在后续各模板引擎的配置属性中查询并修改。

Thymeleaf

Thymeleaf是一个XML/XHTML/HTML5模板引擎，可用于Web与非Web环境中的应用开发。它是一个开源的Java库，基于Apache License 2.0许可，由Daniel Fernández创建，该作者还是Java加密库Jasypt的作者。

Thymeleaf提供了一个用于整合Spring MVC的可选模块，在应用开发中，你可以使用Thymeleaf来完全代替JSP或其他模板引擎，如Velocity、FreeMarker等。Thymeleaf的主要目标在于提供一种可被浏览器正确显示的、格式良好的模板创建

方式，因此也可以用作静态建模。你可以使用它创建经过验证的XML与HTML模板。相对于编写逻辑或代码，开发者只需将标签属性添加到模板中即可。接下来，这些标签属性就会在DOM（文档对象模型）上执行预先制定好的逻辑。

示例模板：

```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</td>
      <th th:text="#{msgs.headers.price}">Price</td>
    </tr>
  </thead>
  <tbody>
    <tr th:each="prod : ${allProducts}">
      <td th:text="${prod.name}">Oranges</td>
      <td th:text="${#numbers.formatDecimal(prod.price,1,2)}">0.
99</td>
    </tr>
  </tbody>
</table>
```

可以看到Thymeleaf主要以属性的方式加入到html标签中，浏览器在解析html时，当检查到没有的属性时候会忽略，所以Thymeleaf的模板可以通过浏览器直接打开展现，这样非常有利于前后端的分离。

在Spring Boot中使用Thymeleaf，只需要引入下面依赖，并在默认的模板路径 `src/main/resources/templates` 下编写模板文件即可完成。

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

在完成配置之后，举一个简单的例子，在快速入门工程的基础上，举一个简单的示例来通过Thymeleaf渲染一个页面。

```
@Controller
public class HelloController {

    @RequestMapping("/")
    public String index(ModelMap map) {
        // 加入一个属性，用来在模板中读取
        map.addAttribute("host", "http://blog.didispace.com");
        // return模板文件的名称，对应src/main/resources/templates/i
ndex.html
        return "index";
    }

}
```

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8" />
    <title></title>
</head>
<body>
<h1 th:text="${host}">Hello World</h1>
</body>
</html>
```

如上页面，直接打开html页面展现Hello World，但是启动程序后，访问 `http://localhost:8080/` ，则是展示Controller中host的值： `http://blog.didispace.com` ，做到了不破坏HTML自身内容的数据逻辑分离。

更多Thymeleaf的页面语法，还请访问Thymeleaf的官方文档查询使用。

Thymeleaf的默认参数配置

如有需要修改默认配置的时候，只需复制下面要修改的属性到 `application.properties` 中，并修改成需要的值，如修改模板文件的扩展名，修改默认的模板路径等。

```
# Enable template caching.
spring.thymeleaf.cache=true
# Check that the templates location exists.
spring.thymeleaf.check-template-location=true
# Content-Type value.
spring.thymeleaf.content-type=text/html
# Enable MVC Thymeleaf view resolution.
spring.thymeleaf.enabled=true
# Template encoding.
spring.thymeleaf.encoding=UTF-8
# Comma-separated list of view names that should be excluded fro
m resolution.
spring.thymeleaf.excluded-view-names=
# Template mode to be applied to templates. See also StandardTem
plateModeHandlers.
spring.thymeleaf.mode=HTML5
# Prefix that gets prepended to view names when building a URL.
spring.thymeleaf.prefix=classpath:/templates/
# Suffix that gets appended to view names when building a URL.
spring.thymeleaf.suffix=.html  spring.thymeleaf.template-resolve
r-order= # Order of the template resolver in the chain. spring.t
hymeleaf.view-names= # Comma-separated list of view names that c
an be resolved.
```

FreeMarker

待补充

Velocity

待补充

支持JSP的配置

Spring Boot并不建议使用，但如果一定要使用，可以参考下面工程作为脚手架。

https://github.com/spring-projects/spring-boot/tree/v1.3.2.RELEASE/spring-boot-samples/spring-boot-sample-web-jsp

静态资源访问

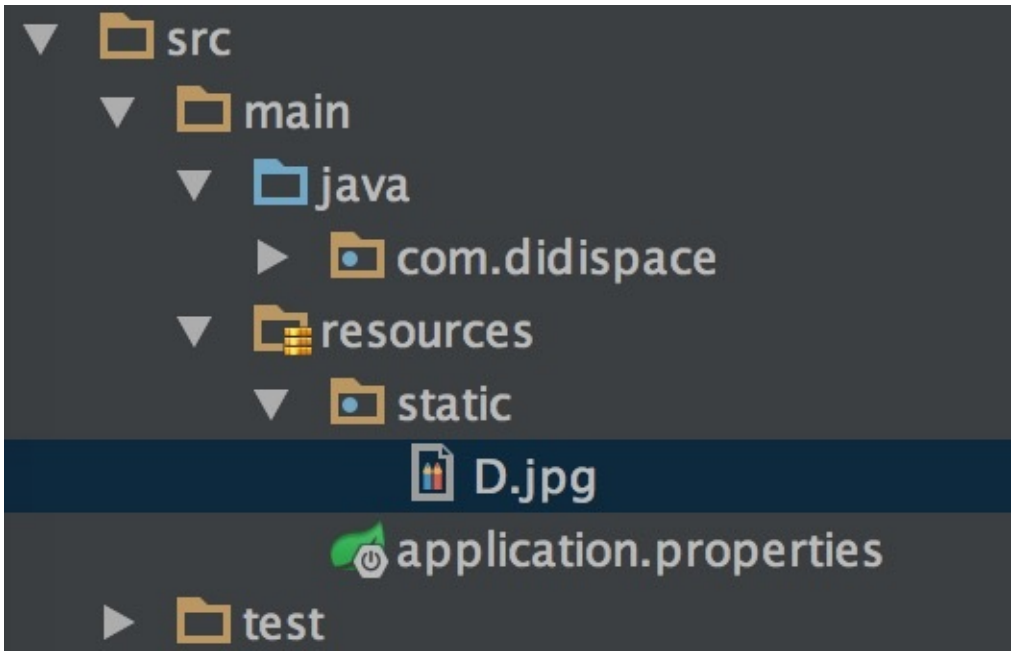
在我们开发Web应用的时候，需要引用大量的js、css、图片等静态资源。

默认配置

Spring Boot默认提供静态资源目录位置需置于classpath下，目录名需符合如下规则：

- /static
- /public
- /resources
- /META-INF/resources

举例：我们可以在 `src/main/resources/` 目录下创建 `static` ，在该位置放置一个图片文件。启动程序后，尝试访问 `http://localhost:8080/D.jpg` 。如能显示图片，配置成功。



自定义配置

。。。

数据访问

数据源配置

在我们访问数据库的时候，需要先配置一个数据源，下面分别介绍一下几种不同的数据库配置方式。

首先，为了连接数据库需要引入jdbc支持，在 `pom.xml` 中引入如下配置：

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

嵌入式数据库支持

嵌入式数据库通常用于开发和测试环境，不推荐用于生产环境。Spring Boot提供自动配置的嵌入式数据库有H2、HSQL、Derby，你不需要提供任何连接配置就能使用。

比如，我们可以在 `pom.xml` 中引入如下配置使用HSQL

```
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <scope>runtime</scope>
</dependency>
```

连接生产数据源

以MySQL数据库为例，先引入MySQL连接的依赖包，在 `pom.xml` 中加入：

```
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <version>5.1.21</version>
</dependency>
```

在 `src/main/resources/application.properties` 中配置数据源信息

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

连接JNDI数据源

当你将应用部署于应用服务器上的时候想让数据源由应用服务器管理，那么可以使用如下配置方式引入JNDI数据源。

```
spring.datasource.jndi-name=java:jboss/datasources/customers
```

使用JdbcTemplate

Spring Boot中JdbcTemplate是自动配置的，你可以直接使用 `@Autowired` 来注入到你自己的bean中来使用。

举例：我们在创建 `User` 表，包含属性 `name` 、 `age` ，下面来编写数据访问对象和单元测试用例。

- 定义包含有插入、删除、查询的抽象接口UserService

```
public interface UserService {

    /**
     * 新增一个用户
     * @param name
     * @param age
     */
    void create(String name, Integer age);

    /**
     * 根据name删除一个用户高
     * @param name
     */
    void deleteByName(String name);

    /**
     * 获取用户总量
     */
    Integer getAllUsers();

    /**
     * 删除所有用户
     */
    void deleteAllUsers();

}
```

- 通过JdbcTemplate实现UserService中定义的数据访问操作

```
@Service
public class UserServiceImpl implements UserService {

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public void create(String name, Integer age) {
        jdbcTemplate.update("insert into USER(NAME, AGE) values(
?, ?)", name, age);
    }

    @Override
    public void deleteByName(String name) {
        jdbcTemplate.update("delete from USER where NAME = ?", n
ame);
    }

    @Override
    public Integer getAllUsers() {
        return jdbcTemplate.queryForObject("select count(1) from
 USER", Integer.class);
    }

    @Override
    public void deleteAllUsers() {
        jdbcTemplate.update("delete from USER");
    }
}
```

- 创建对UserService的单元测试用例，通过创建、删除和查询来验证数据库操作的正确性。

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

    @Autowired
    private UserService userSerivce;

    @Before
    public void setUp() {
        // 准备，清空user表
        userSerivce.deleteAllUsers();
    }

    @Test
    public void test() throws Exception {
        // 插入5个用户
        userSerivce.create("a", 1);
        userSerivce.create("b", 2);
        userSerivce.create("c", 3);
        userSerivce.create("d", 4);
        userSerivce.create("e", 5);

        // 查数据库，应该有5个用户
        Assert.assertEquals(5, userSerivce.getAllUsers().intValu
e());

        // 删除两个用户
        userSerivce.deleteByName("a");
        userSerivce.deleteByName("e");

        // 查数据库，应该有5个用户
        Assert.assertEquals(3, userSerivce.getAllUsers().intValu
e());

    }

}
```

上面介绍的`JdbcTemplate`只是最基本的几个操作，更多其他数据访问操作的使用请参考：*JdbcTemplate API*

通过上面这个简单的例子，我们可以看到在Spring Boot下访问数据库的配置依然秉承了框架的初衷：简单。我们只需要在pom.xml中加入数据库依赖，再到application.properties中配置连接信息，不需要像Spring应用中创建JdbcTemplate的Bean，就可以直接在自己的对象中注入使用。

本文完整示例

使用Spring-data-jpa

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").g
etAge().longValue());

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

多数据源配置

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。

配置两个数据源

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.Drive
r

spring.datasource.secondary.url=jdbc:mysql://localhost:3306/test
2
spring.datasource.secondary.username=root
spring.datasource.secondary.password=root
spring.datasource.secondary.driver-class-name=com.mysql.jdbc.Dri
ver
```

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 dataSourc
e) {
        return new JdbcTemplate(dataSource);
    }

    @Bean(name = "secondaryJdbcTemplate")
    public JdbcTemplate secondaryJdbcTemplate(
            @Qualifier("secondaryDataSource") DataSource dataSou
rce) {
        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) valu
es(?, ?, ?)", 1, "aaa", 20);
        jdbcTemplate1.update("insert into user(id,name,age) valu
es(?, ?, ?)", 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) valu
es(?, ?, ?)", 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("s
elect 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("s
```

```
elect 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Reposit
ory所在位置
public class PrimaryConfig {

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

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

    @Primary
    @Bean(name = "entityManagerFactoryPrimary")
    public LocalContainerEntityManagerFactoryBean entityManagerF
```

```
actoryPrimary (EntityManagerFactoryBuilder builder) {
        return builder
                .dataSource(primaryDataSource)
                .properties(getVendorProperties(primaryDataSourc
e))
                .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 d
ataSource) {
        return jpaProperties.getHibernateProperties(dataSource);
    }

    @Primary
    @Bean(name = "transactionManagerPrimary")
    public PlatformTransactionManager transactionManagerPrimary(
EntityManagerFactoryBuilder builder) {
        return new JpaTransactionManager(entityManagerFactoryPri
mary(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Reposit
ory所在位置
public class SecondaryConfig {
```

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

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

    @Bean(name = "entityManagerFactorySecondary")
    public LocalContainerEntityManagerFactoryBean entityManagerF
actorySecondary (EntityManagerFactoryBuilder builder) {
        return builder
                .dataSource(secondaryDataSource)
                .properties(getVendorProperties(secondaryDataSou
rce))
                .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 d
ataSource) {
        return jpaProperties.getHibernateProperties(dataSource);
    }

    @Bean(name = "transactionManagerSecondary")
    PlatformTransactionManager transactionManagerSecondary(Entit
yManagerFactoryBuilder builder) {
        return new JpaTransactionManager(entityManagerFactorySec
ondary(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