



Table of Contents

简介	1.1
1. 使用Spring Boot搭建你的第一个应用程序	1.2
2. 如何在Spring boot中修改默认端口	1.3
3. spring-boot-starters介绍	1.4
4. 自定义parent POM	1.5
5. 使用spring boot创建fat jar	1.6
6. spring boot 使用maven和fat jar/war运行应用程序的对比	1.7
7. Maven Wrapper简介	1.8
8. Spring Boot注解	1.9
9. Spring Boot @EnableAutoConfiguration和 @Configuration的区别	1.10
10. 自定义spring boot的自动配置	1.11
11. 在Spring Boot中配置web app	1.12
12. 从Spring迁移到Spring Boot	1.13
13. Spring Boot中的测试	1.14
14. Spring Boot的TestRestTemplate使用	1.15
15. Spring Boot Actuator	1.16
16. Spring Boot中的Properties	1.17
17. SpringBoot @ConfigurationProperties详解	1.18
18. 在Spring Boot中加载初始化数据	1.19
19. Spring Boot的exit code	1.20
20. Shutdown SpringBoot App	1.21
21. Spring boot 自定义banner	1.22
22. Spring Boot filter	1.23
23. Spring Boot中使用Swagger CodeGen生成REST client	1.24
24. Spring Boot中使用@JsonComponent	1.25
25. Spring Boot国际化支持	1.26
26. Spring Boot devtool的使用	1.27
27. Spring Boot Admin的使用	1.28
28. 将Spring Boot应用程序注册成为系统服务	1.29
29. Spring Boot 之Spring data JPA简介	1.30
30. Spring Boot JPA中java 8 的应用	1.31
31. Spring Boot中Spring data注解的使用	1.32
32. 在Spring Boot使用H2内存数据库	1.33

33. 在Spring Boot中使用内存数据库	1.34
34. Spring Boot JPA中使用@Entity和@Table	1.35
35. Spring Boot JPA的查询语句	1.36
36. Spring Boot JPA中关联表的使用	1.37
37. Spring Boot JPA 中transaction的使用	1.38

www.flydean.com

简介

Spring boot 2.X 实战教程（2020版）

本文系列的介绍了Spring boot 2.X相关的知识点和实际例子，希望大家能够喜欢。

许可

MIT

更多教程

欢迎关注我的公众号:程序那些事，更多精彩等着您！

更多内容请访问 www.flydean.com

www.flydean.com

使用Spring Boot搭建你的第一个应用程序

Spring Boot是Spring平台的约定式的应用框架，使用Spring Boot可以更加方便简洁的开发基于Spring的应用程序，本篇文章通过一个实际的例子，来一步一步的演示如何创建一个基本的Spring Boot程序。

依赖配置

本例子使用Maven来做包的依赖管理，在pom.xml文件中我们需要添加Spring boot依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

同时我们要构建一个web应用程序，所以需要添加web依赖：

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

ORM框架，我们使用spring自带的jpa，数据库使用内存数据库H2：

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

main程序配置

接下来我们需要创建一个应用程序的主类：

```
@SpringBootApplication
public class App {
```

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

这里我们使用了注解：`@SpringBootApplication`。它等同于三个注解：`@Configuration`, `@EnableAutoConfiguration`, 和 `@ComponentScan`同时使用。

最后，我们需要在resources目录中添加属性文件：`application.properties`。在其中我们定义程序启动的端口：

```
server.port=8081
```

MVC配置

spring MVC可以配合很多模板语言使用，这里我们使用Thymeleaf。

首先需要添加依赖：

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

然后在`application.properties`中添加如下配置：

```
spring.thymeleaf.cache=false  
spring.thymeleaf.enabled=true  
spring.thymeleaf.prefix=classpath:/templates/  
spring.thymeleaf.suffix=.html  
  
spring.application.name=Bootstrap Spring Boot
```

然后创建一个home页面：

```
<html>  
<head><title>Home Page</title></head>  
<body>  
    <h1>Hello !</h1>  
    <p>Welcome to <span th:text="${appName}">Our App</span></p>  
</body>  
</html>
```

最后创建一个Controller指向这个页面：

```
@Controller
public class SimpleController {
    @Value("${spring.application.name}")
    String appName;

    @GetMapping("/")
    public String homePage(Model model) {
        model.addAttribute("appName", appName);
        return "home";
    }
}
```

安全配置

本例主要是搭一个基本完整的框架，所以必须的安全访问控制也是需要的。我们使用Spring Security来做安全控制，加入依赖如下：

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

当spring-boot-starter-security加入依赖之后，应用程序所有的入库会被默认加入权限控制，在本例中，我们还用不到这些权限控制，所以需要自定义SecurityConfig，放行所有的请求：

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .anyRequest()
            .permitAll()
            .and().csrf().disable();
    }
}
```

上例中，我们permit all请求。

后面我会详细的关于Spring Security的教程。这里先不做深入讨论。

存储

本例中，我们定义一个Book类，那么需要定义相应的Entity类：

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(nullable = false, unique = true)
    private String title;

    @Column(nullable = false)
    private String author;
}
```

和相应的Repository类：

```
public interface BookRepository extends CrudRepository<Book, Long> {
    List<Book> findByTitle(String title);
}
```

最后，我们需要让应用程序发现我们配置的存储类，如下：

```
@EnableJpaRepositories("com.flydean.learn.repository")
@EntityScan("com.flydean.learn.entity")
@SpringBootApplication
public class App {

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

这里，我们使用@EnableJpaRepositories 来扫描repository类。

使用@EntityScan来扫描JPA entity类。

为了方便起见，我们使用内存数据库H2. 一旦H2在依赖包里面，Spring boot会自动检测到，并使用它。我们需要配置一些H2的属性：

```
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.url=jdbc:h2:mem:bootapp;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=
```

和安全一样，存储也是一个非常重要和复杂的课题，我们也会在后面的文章中讨论。

Web 页面和Controller

有了Book entity， 我们需要为Book写一个Controller， 主要做增删改查的操作， 如下所示：

```
@RestController
@RequestMapping("/api/books")
public class BookController {

    @Autowired
    private BookRepository bookRepository;

    @GetMapping
    public Iterable findAll() {
        return bookRepository.findAll();
    }

    @GetMapping("/title/{bookTitle}")
    public List findByTitle(@PathVariable String bookTitle) {
        return bookRepository.findByTitle(bookTitle);
    }

    @GetMapping("/{id}")
    public Book findOne(@PathVariable Long id) {
        return bookRepository.findById(id)
            .orElseThrow(BookNotFoundException::new);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Book create(@RequestBody Book book) {
        return bookRepository.save(book);
    }

    @DeleteMapping("/{id}")
    public void delete(@PathVariable Long id) {
        bookRepository.findById(id)
            .orElseThrow(BookNotFoundException::new);
        bookRepository.deleteById(id);
    }

    @PutMapping("/{id}")
    public Book updateBook(@RequestBody Book book, @PathVariable Long id) {
        if (book.getId() != id) {
            throw new BookIdMismatchException("ID mismatch!");
        }
        bookRepository.findById(id)
            .orElseThrow(BookNotFoundException::new);
        return bookRepository.save(book);
    }
}
```

这里我们使用@RestController注解，表示这个Controller是一个API，不涉及到页面的跳转。

@RestController是@Controller和@ResponseBody的集合。

异常处理

基本上我们的程序已经完成了，但是在Controller中，我们定义了一些自定义的异常：

```
public class BookNotFoundException extends RuntimeException {  
  
    public BookNotFoundException(String message, Throwable cause) {  
        super(message, cause);  
    }  
    // ...  
}
```

那么怎么处理这些异常呢？我们可以使用@ControllerAdvice来拦截这些异常：

```
@ControllerAdvice  
public class RestExceptionHandler extends ResponseEntityExceptionHandler {  
  
    @ExceptionHandler({ BookNotFoundException.class })  
    protected ResponseEntity<Object> handleNotFound(  
        Exception ex, WebRequest request) {  
        return handleExceptionInternal(ex, "Book not found",  
            new HttpHeaders(), HttpStatus.NOT_FOUND, request);  
    }  
  
    @ExceptionHandler({ BookIdMismatchException.class,  
        ConstraintViolationException.class,  
        DataIntegrityViolationException.class })  
    public ResponseEntity<Object> handleBadRequest(  
        Exception ex, WebRequest request) {  
        return handleExceptionInternal(ex, ex.getLocalizedMessage(),  
            new HttpHeaders(), HttpStatus.BAD_REQUEST, request);  
    }  
}
```

这种异常捕获也叫做全局异常捕获。

测试

我们的Book API已经写好了，接下来我们需要写一个测试程序来测试一下。

这里我们使用@SpringBootTest：

```
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
public class SpringContextTest {

    @Test
    public void contextLoads() {
        log.info("contextLoads");
    }
}
```

webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT的作用是表示测试时候使用的Spring boot应用程序端口使用自定义在application.properties中的端口。

接下来我们使用RestAssured来测试BookController:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = SpringBootTest.WebEnvironment.DEFINED_PORT)
public class SpringBootBootstrapTest {

    private static final String API_ROOT
        = "http://localhost:8081/api/books";

    private Book createRandomBook() {
        Book book = new Book();
        book.setTitle(randomAlphabetic(10));
        book.setAuthor(randomAlphabetic(15));
        return book;
    }

    private String createBookAsUri(Book book) {
        Response response = RestAssured.given()
            .contentType(MediaType.APPLICATION_JSON_VALUE)
            .body(book)
            .post(API_ROOT);
        return API_ROOT + "/" + response.jsonPath().get("id");
    }

    @Test
    public void whenGetAllBooks_thenOK() {
        Response response = RestAssured.get(API_ROOT);

        assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    }

    @Test
    public void whenGetBooksByTitle_thenOK() {
        Book book = createRandomBook();
```

```
createBookAsUri(book);
Response response = RestAssured.get(
    API_ROOT + "/title/" + book.getTitle());

assertEquals(HttpStatus.OK.value(), response.getStatusCode());
assertTrue(response.as(List.class)
    .size() > 0);
}

@Test
public void whenGetCreatedBookById_thenOK() {
    Book book = createRandomBook();
    String location = createBookAsUri(book);
    Response response = RestAssured.get(location);

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());
    assertEquals(book.getTitle(), response.jsonPath()
        .get("title"));
}

@Test
public void whenGetNotExistBookById_thenNotFound() {
    Response response = RestAssured.get(API_ROOT + "/" + randomNumeric(4));

    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatusCode());
}

@Test
public void whenCreateNewBook_thenCreated() {
    Book book = createRandomBook();
    Response response = RestAssured.given()
        .contentType(MediaType.APPLICATION_JSON_VALUE)
        .body(book)
        .post(API_ROOT);

    assertEquals(HttpStatus.CREATED.value(), response.getStatusCode());
}

@Test
public void whenInvalidBook_thenError() {
    Book book = createRandomBook();
    book.setAuthor(null);
    Response response = RestAssured.given()
        .contentType(MediaType.APPLICATION_JSON_VALUE)
        .body(book)
        .post(API_ROOT);

    assertEquals(HttpStatus.BAD_REQUEST.value(), response.getStatusCode());
}

@Test
public void whenUpdateCreatedBook_thenUpdated() {
```

```
Book book = createRandomBook();
String location = createBookAsUri(book);
book.setId(Long.parseLong(location.split("api/books/")[1]));
book.setAuthor("newAuthor");
Response response = RestAssured.given()
    .contentType(MediaType.APPLICATION_JSON_VALUE)
    .body(book)
    .put(location);

assertEquals(HttpStatus.OK.value(), response.getStatusCode());

response = RestAssured.get(location);

assertEquals(HttpStatus.OK.value(), response.getStatusCode());
assertEquals("newAuthor", response.jsonPath()
    .get("author"));

}

@Test
public void whenDeleteCreatedBook_thenOk() {
    Book book = createRandomBook();
    String location = createBookAsUri(book);
    Response response = RestAssured.delete(location);

    assertEquals(HttpStatus.OK.value(), response.getStatusCode());

    response = RestAssured.get(location);
    assertEquals(HttpStatus.NOT_FOUND.value(), response.getStatusCode());
}
```

写好了测试类，运行就行了。

结论

你的第一个Spring Boot程序就完成了，后面的文章我们会继续丰富和改善这个基本框架，欢迎继续关注。

本文章的例子代码可以参考github: [bootstrap-sample-app](#)

如何在Spring boot中修改默认端口

介绍

Spring boot为应用程序提供了很多属性的默认值。但是有时候，我们需要自定义某些属性，比如：修改内嵌服务器的端口号。

本篇文章就来讨论这个问题。

使用Property文件

第一种方式，也是最常用的方式就是在属性文件中，覆盖默认的配置。对于服务器的端口来说，该配置就是：server.port。

默认情况下，server.port值是8080。 我们可以在application.properties中这样修改为8081：

```
server.port=8081
```

如果你使用的是application.yml，那么需要这样配置：

```
server:  
  port : 8081
```

这两个文件都会在Spring boot启动的时候被加载。

如果同一个应用程序需要在不同的环境中使用不同的端口，这个时候你就需要使用到Spring Boot的profile概念，不同的profile使用不同的配置文件。

比如你在application-dev.properties中：

```
server.port=8081
```

在application-qa.properties 中：

```
server.port=8082
```

在程序中指定

我们可以在程序中直接指定应用程序的端口，如下所示：

```
@SpringBootApplication  
public class CustomApplication {
```

```
public static void main(String[] args) {
    SpringApplication app = new SpringApplication(CustomApplication.class);
    app.setDefaultProperties(Collections
        .singletonMap("server.port", "8083"));
    app.run(args);
}
```

另外一种自定义服务的方法就是实现WebServerFactoryCustomizer接口：

```
@Component
public class ServerPortCustomizer
    implements WebServerFactoryCustomizer<ConfigurableWebServerFactory> {

    @Override
    public void customize(ConfigurableWebServerFactory factory) {
        factory.setPort(8086);
        //        factory.setAddress("");
    }
}
```

使用ConfigurableWebServerFactory可以自定义包括端口在内的其他很多服务器属性。

使用命令行参数

如果应用程序被打包成jar，我们也可以在命令行运行时候，手动指定 server.port。

```
java -jar spring-5.jar --server.port=8083
```

或者这样：

```
java -jar -Dserver.port=8083 spring-5.jar
```

值生效的顺序

上面我们将了这么多修改自定义端口的方式，那么他们的生效顺序是怎么样的呢？

1. 内置的server配置
2. 命令行参数
3. property文件
4. @SpringBootApplication配置的主函数

更多教程请参考 [flydean的博客](#)

www.flydean.com

Spring Boot Starters介绍

对于任何一个复杂项目来说，依赖关系都是一个非常需要注意和消息的方面，虽然重要，但是我们也不需要花太多的时间在上面，因为依赖毕竟只是框架，我们重点需要关注的还是程序业务本身。

这就是为什么会有Spring Boot starters的原因。Starter POMs 是一系列可以被引用的依赖集合，只需要引用一次就可以获得所有需要使用到的依赖。

Spring Boot有超过30个starts, 本文将介绍比较常用到的几个。

Web Start

如果我们需要开发MVC程序或者REST服务，那么我们需要使用到Spring MVC, Tomcat,JSON等一系列的依赖。但是使用Spring Boot Start,一个依赖就够了：

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

现在我们就可以创建REST Controller了：

```
@RestController
public class GenericEntityController {
    private List<GenericEntity> entityList = new ArrayList<>();

    @RequestMapping("/entity/all")
    public List<GenericEntity> findAll() {
        return entityList;
    }

    @RequestMapping(value = "/entity", method = RequestMethod.POST)
    public GenericEntity addEntity(GenericEntity entity) {
        entityList.add(entity);
        return entity;
    }

    @RequestMapping("/entity/findby/{id}")
    public GenericEntity findById(@PathVariable Long id) {
        return entityList.stream().
            filter(entity -> entity.getId().equals(id)).
            findFirst().get();
    }
}
```

这样我们就完成了一个非常简单的Spring Web程序。

Test Starter

在测试中，我们通常会用到Spring Test, JUnit, Hamcrest, 和 Mockito这些依赖，Spring也有一个starter集合：

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

注意，你并不需要指定artifact的版本号，因为这一切都是从spring-boot-starter-parent 的版本号继承过来的。后面升级的话，只需要升级parent的版本即可。具体的应用可以看下本文的例子。

接下来让我们测试一下刚刚创建的controller：

这里我们使用mock。

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = Application.class)
@WebAppConfiguration
public class SpringBootApplicationIntegrationTest {
    @Autowired
    private WebApplicationContext webApplicationContext;
    private MockMvc mockMvc;

    @Before
    public void setupMockMvc() {
        mockMvc = MockMvcBuilders.webAppContextSetup(webApplicationContext).build();
    }

    @Test
    public void givenRequestHasBeenCalled_whenMeetsAllOfGivenConditions_thenCorrect()
            throws Exception {
        MediaType contentType = new MediaType(MediaType.APPLICATION_JSON.getType(),
                MediaType.APPLICATION_JSON.getSubtype(), Charset.forName("utf8"));
        mockMvc.perform(MockMvcRequestBuilders.get("/entity/all"))
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.content().contentType(contentType))
                .andExpect(jsonPath("$.size()", hasSize(4)));
    }
}
```

上面的例子，我们测试了/entity/all接口，并且验证了返回的JSON。

这里@WebAppConfiguration 和 MockMVC 是属于 spring-test 模块, hasSize 是一个Hamcrest 的匹配器, @Before 是一个 JUnit 注解.所有的一切，都包含在一个starter中。

Data JPA Starter

如果想使用JPA, 我们可以这样:

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

我们接下来创建一个repository:

```
public interface GenericEntityRepository extends JpaRepository<GenericEntity, Long>
{}
```

然后是JUnit测试:

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringBootTest(classes = Application.class)
public class SpringBootJPATest {

    @Autowired
    private GenericEntityRepository genericEntityRepository;

    @Test
    public void givenGenericEntityRepository_whenSaveAndRetreiveEntity_thenOK() {
        GenericEntity genericEntity =
            genericEntityRepository.save(new GenericEntity("test"));
        GenericEntity foundedEntity =
            genericEntityRepository.findById(genericEntity.getId()).orElse(null);
        assertNotNull(foundedEntity);
        assertEquals(genericEntity.getValue(), foundedEntity.getValue());
    }
}
```

这里我们测试了JPA自带的save, findById方法。可以看到我们没有做任何的配置, Spring boot自动帮我们完成了所有操作。

Mail Starter

在企业开发中，发送邮件是一件非常常见的事情，如果直接使用 Java Mail API会比较复杂。如果使用 Spring boot：

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

这样我们就可以直接使用JavaMailSender，前提是需要配置mail的连接属性如下：

```
spring.mail.host=localhost
spring.mail.port=25
spring.mail.default-encoding=UTF-8
```

接下来我们来写一些测试案例。

为了发送邮件，我们需要一个简单的SMTP服务器。在本例中，我们使用Wiser。

```
<dependency>
    <groupId>org.subethamail</groupId>
    <artifactId>subethasmtplib</artifactId>
    <version>3.1.7</version>
    <scope>test</scope>
</dependency>
```

下面是如何发送的代码：

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = Application.class)
public class SpringBootMailTest {
    @Autowired
    private JavaMailSender javaMailSender;

    private Wiser wiser;

    private String userTo = "user2@localhost";
    private String userFrom = "user1@localhost";
    private String subject = "Test subject";
    private String textMail = "Text subject mail";

    @Before
    public void setUp() throws Exception {
        final int TEST_PORT = 25;
        wiser = new Wiser(TEST_PORT);
        wiser.start();
    }

    @After
```

```
public void tearDown() throws Exception {
    wiser.stop();
}

@Test
public void givenMail_whenSendAndReceived_thenCorrect() throws Exception {
    SimpleMailMessage message = composeEmailMessage();
    javaMailSender.send(message);
    List<WiserMessage> messages = wiser.getMessages();

    assertThat(messages, hasSize(1));
    WiserMessage wiserMessage = messages.get(0);
    assertEquals(userFrom, wiserMessage.getEnvelopeSender());
    assertEquals(userTo, wiserMessage.getEnvelopeReceiver());
    assertEquals(subject, getSubject(wiserMessage));
    assertEquals(textMail, getMessage(wiserMessage));
}

private String getMessage(WiserMessage wiserMessage)
    throws MessagingException, IOException {
    return wiserMessage.getMimeMessage().getContent().toString().trim();
}

private String getSubject(WiserMessage wiserMessage) throws MessagingException {
    return wiserMessage.getMimeMessage().getSubject();
}

private SimpleMailMessage composeEmailMessage() {
    SimpleMailMessage mailMessage = new SimpleMailMessage();
    mailMessage.setTo(userTo);
    mailMessage.setReplyTo(userFrom);
    mailMessage.setFrom(userFrom);
    mailMessage.setSubject(subject);
    mailMessage.setText(textMail);
    return mailMessage;
}
}
```

在上面的例子中，@Before 和 @After 分别用来启动和关闭邮件服务器。

结论

本文介绍了一些常用的starts，具体例子可以参考 [spring-boot-starts](#)

更多教程请参考 [flydean的博客](#)

www.flydean.com

自定义parent POM

概述

在之前的Spring Boot例子中，我们都会用到这样的parent POM。

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.2.2.RELEASE</version>
  <relativePath/> <!-- lookup parent from repository -->
</parent>
```

这个parent指定了spring-boot所需要的依赖。但是有时候如果我们的项目已经有一个parent了，这时候需要引入spring boot该怎么处理呢？

本文将会解决这个问题。

不使用Parent POM来引入Spring boot

parent pom.xml主要处理的是依赖和使用的插件管理。使用起来会非常简单，这也是我们在Spring boot中常用的方式。

在实际中，如果我们因为种种原因，不能使用Spring boot自带的parent,那么我们可以这样做：

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.2.2.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

将spring-boot-dependencies作为一个依赖放入dependencyManagement标签即可。注意，这里的scope要使用import。

接下来，我们就可以随意使用spring boot的依赖了，例如：

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

另一方面，如果不使用parent POM，Spring boot自带的plugin，需要我们自己引入：

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

覆盖依赖项版本

如果我们需要使用和parent POM中定义的不同的依赖项版本，则可以在dependencyManagement中重写。

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-data-jpa</artifactId>
      <version>1.5.5.RELEASE</version>
    </dependency>
  </dependencies>
  // ...
</dependencyManagement>
```

当然，你也可以在每次引入依赖的时候，指定所需要的版本。

使用spring boot创建fat jar APP

介绍

在很久很久以前，我们部署web程序的方式是怎么样的呢？配置好服务器，将自己写的应用程序打包成war包，扔进服务器中指定的目录里面。当然免不了要配置一些负责的xml和自定义一些servlet。

现在有了spring boot，一切都变了，我们可以将web应用程序打包成fat jar包，直接运行就行了。

本文将会关注于怎么使用Spring Boot创建一个fat jar包。

所有你需要做的就是添加如下依赖：

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

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

build和run

有了上面的配置，只需要使用

```
mvn clean install
```

就可以生成相应的jar包了。

如果要运行它，使用：

```
java -jar <artifact-name>
```

即可。非常简洁。

如果你要在服务器上面永久运行该服务，即使登录的用户退出服务器，则可以使用nohup命令：

```
nohup java -jar <artifact-name>
```

fat jar和 fat war

在上面的例子中，所有的依赖jar包都会被打包进入这一个fat jar中，如果你使用了tomcat,那么tomcat也会被打包进去。

但有时候我们还是需要打包成war包，部署在服务器中，这种情况只需要将pom.xml中的packaging属性修改为war即可。

更多配置

大多情况下，我们不需要额外的配置，如果我们有多个main class,我们需要指定具体的哪个类：

```
<properties>
    <start-class>com.flydean.FatJarApp</start-class>
</properties>
```

如果你没有从spring-boot-starter-parent继承，那么你需要将main class添加到maven plugin中：

```
<plugins>
    <plugin>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-maven-plugin</artifactId>
        <configuration>
            <mainClass>com.flydean.FatJarApp</mainClass>
            <layout>ZIP</layout>
        </configuration>
    </plugin>
</plugins>
```

有些情况下，你需要告诉maven来unpack一些依赖：

```
<plugin>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-maven-plugin</artifactId>
    <configuration>
        <requiresUnpack>
            <dependency>
                <groupId>org.jruby</groupId>
                <artifactId>jruby-complete</artifactId>
            </dependency>
        </requiresUnpack>
    </configuration>
</plugin>
```

本文的代码请参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-fatjar>

www.flydean.com

使用maven和fat jar/war运行应用程序的对比

简介

上篇文章我们介绍了Spring boot的fat jar/war包，jar/war包都可以使用java -jar命令来运行，而maven也提供了mvn spring-boot:run命令来运行应用程序，下面我们看看两者有什么不同。

Spring Boot Maven Plugin

上篇文章我们提到了Spring Boot Maven Plugin，通过使用该插件，可以有效的提高部署效率，并打包成为fat jar/war包。

在打包成fat jar/war包的时候，背后实际上做了如下的事情：

1. 管理了classpath的配置，这样我们在运行java -jar的时候不用手动指定-cp。
2. 使用了自定义的ClassLoader来加载和定位所有的外部jar包依赖。并且所有的依赖jar包已经被包含在这个fat包里面了。
3. 通过manifest自动查找main()，这样我们就不需要在java -jar中手动指定main方法。

使用Maven命令来运行应用程序

要使用maven命令来运行应用程序可以在程序的根目录下面执行：

```
mvn spring-boot:run
```

它会自动下载所需要的依赖，并运行，运行日志如下：

```
mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] -----< com.flydean:springboot-fatjar >-----
[INFO] Building springboot-fatjar 0.0.1-SNAPSHOT
[INFO] -----[ jar ]-----
[INFO]
[INFO] >>> spring-boot-maven-plugin:2.2.2.RELEASE:run (default-cli) > test-compile
@ springboot-fatjar >>>
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:resources (default-resources) @ springboot-
fatjar ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 0 resource
[INFO] Copying 0 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:compile (default-compile) @ springboot-fatja
r ---
```

```
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:3.1.0:testResources (default-testResources) @ springboot-fatjar ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.

[INFO]
[INFO] --- maven-compiler-plugin:3.8.1:testCompile (default-testCompile) @ springboot-fatjar ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] <<< spring-boot-maven-plugin:2.2.2.RELEASE:run (default-cli) < test-compile
@ springboot-fatjar <<<
[INFO]
[INFO]
[INFO] --- spring-boot-maven-plugin:2.2.2.RELEASE:run (default-cli) @ springboot-fatjar ---
[INFO] Attaching agents: []
```

作为fat jar/war包运行应用程序

如果想打包成fat jar/war, 需要使用Maven Spring Boot plugin, 如下所示, 否则打包出来的jar包并不包含外部依赖:

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

如果我们的代码包含了多个main class, 需要手动指定具体使用哪一个, 有两种设置方式:

```
<plugin>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-maven-plugin</artifactId>
  <executions>
    <execution>
      <configuration>
        <mainClass>com.flydean.FatJarApp</mainClass>
      </configuration>
    </execution>
  </executions>
</plugin>
```

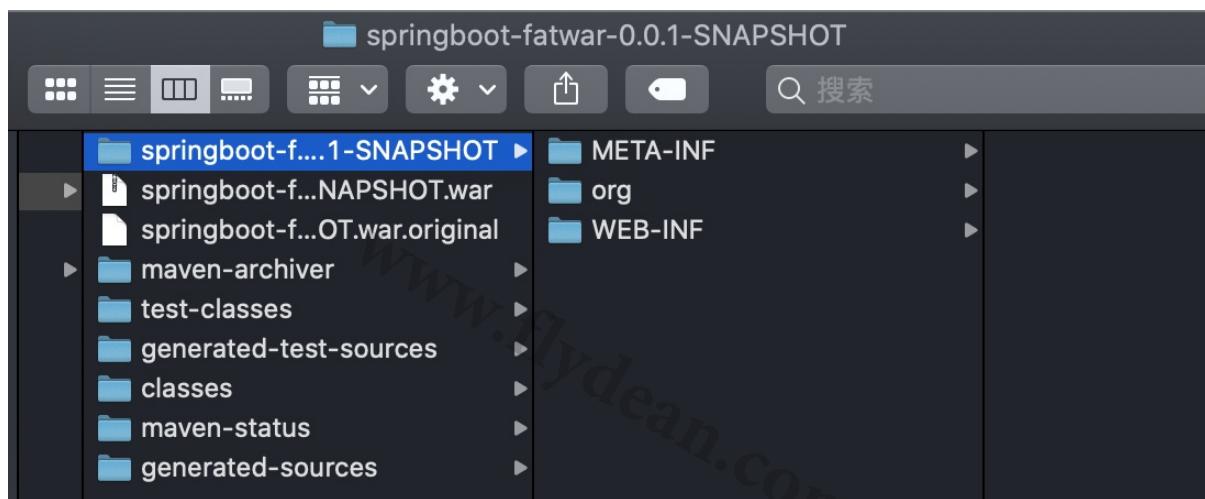
或者设置start-class属性：

```
<properties>
    <start-class>com.flydean.FatJarApp</start-class>
</properties>
```

使用 mvn clean package 即可打包程序，然后使用java -jar target/springboot-fatwar-0.0.1-SNAPSHOT.war 即可运行。

详解War文件

将打包好的war文件解压，我们看下War文件的结构：



里面有三部分：

- META-INF, 里面包含有自动生成的MANIFEST.MF
- WEB-INF/classes, 包含了编译好的class文件
- WEB-INF/lib, 包含了war的依赖jar包和嵌入的Tomcat jar包。
- WEB-INF/lib-provided, 包含了embedded模式运行所需要但是在部署模式不需要的额外的依赖包。
- org/springframework/boot/loader, 里面是Spring boot自定义的类加载器，这些类加载器负责加载外部依赖，并且使他们在运行时可用。

我们再看下MANIFEST.MF文件的内容：

```
Manifest-Version: 1.0
Implementation-Title: springboot-fatwar
Implementation-Version: 0.0.1-SNAPSHOT
Start-Class: com.flydean.FatWarApp
Spring-Boot-Classes: WEB-INF/classes/
Spring-Boot-Lib: WEB-INF/lib/
Build-Jdk-Spec: 1.8
Spring-Boot-Version: 2.2.2.RELEASE
Created-By: Maven Archiver 3.4.0
Main-Class: org.springframework.boot.loader.WarLauncher
```

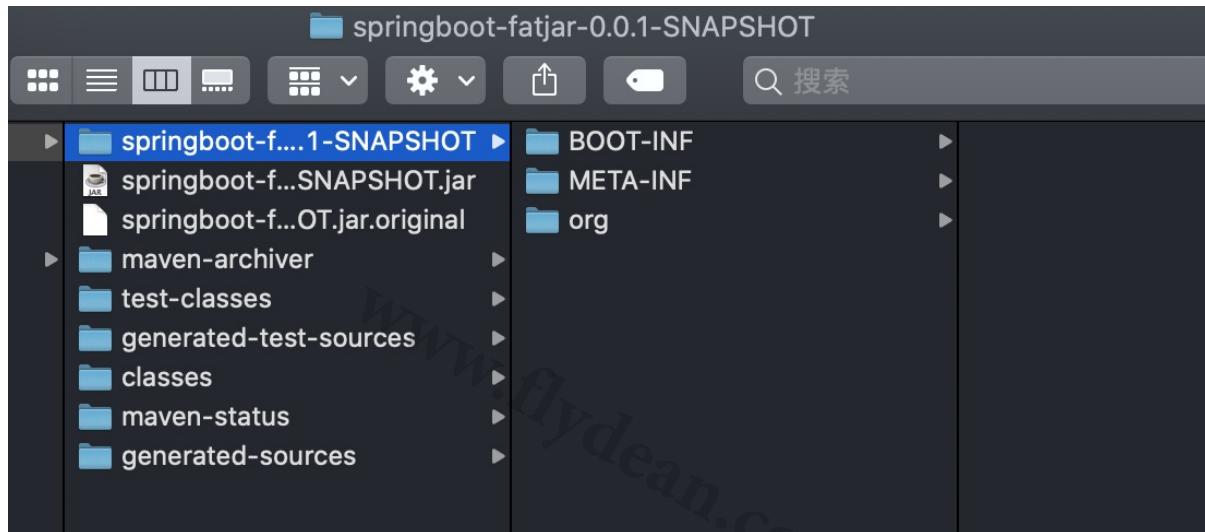
主要关注两行：

```
Start-Class: com.flydean.FatWarApp  
Main-Class: org.springframework.boot.loader.WarLauncher
```

一个是启动类就是我们自己写的，一个main类这个是Spring boot自带的。

详解jar文件

我们再来看下jar文件：



jar文件和war文件有一点不同，没有WEB-INF，改成了BOOT-INF。

- 我们所有的自己的class都在BOOT-INF/classes下面。
- 外部依赖在BOOT-INF/lib下。

我们再看下MANIFEST.MF文件有什么不同：

```
Manifest-Version: 1.0  
Implementation-Title: springboot-fatjar  
Implementation-Version: 0.0.1-SNAPSHOT  
Start-Class: com.flydean.FatJarApp  
Spring-Boot-Classes: BOOT-INF/classes/  
Spring-Boot-Lib: BOOT-INF/lib/  
Build-Jdk-Spec: 1.8  
Spring-Boot-Version: 2.2.2.RELEASE  
Created-By: Maven Archiver 3.4.0  
Main-Class: org.springframework.boot.loader.PropertiesLauncher
```

我们可以看到Start-Class还是一样的，但是Main-Class是不一样的。

如何选择

既然有两种方式来运行应用程序，一种是使用mvn命令，一种是使用fat jar/war文件，那我们该怎么选择呢？

通常情况下，如果我们是在线下的开发环境，可以直接使用mvn命令，mvn命令需要依赖于源代码，我们可以不断的修改源代码，方便开发。

如果是在线上环境，那么我们就需要使用fat jar/war了，这样的外部依赖比较小，我们不需要在线上环境部署maven环境，也不需要源代码，只要一个java的运行时环境就可以了。

本文的代码请参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-fatwar>

更多教程请参考 [flydean的博客](#)

www.flydean.com

Maven Wrapper简介

简介

开发java项目少不了要用到maven或者gradle,对比gradle而言, 可能maven要更加常用一些。要使用maven那就要安装maven,如果有些用户不想安装maven怎么办? 或者说用户不想全局安装maven,那么可以使用项目级别的Maven Wrapper来实现这个功能。

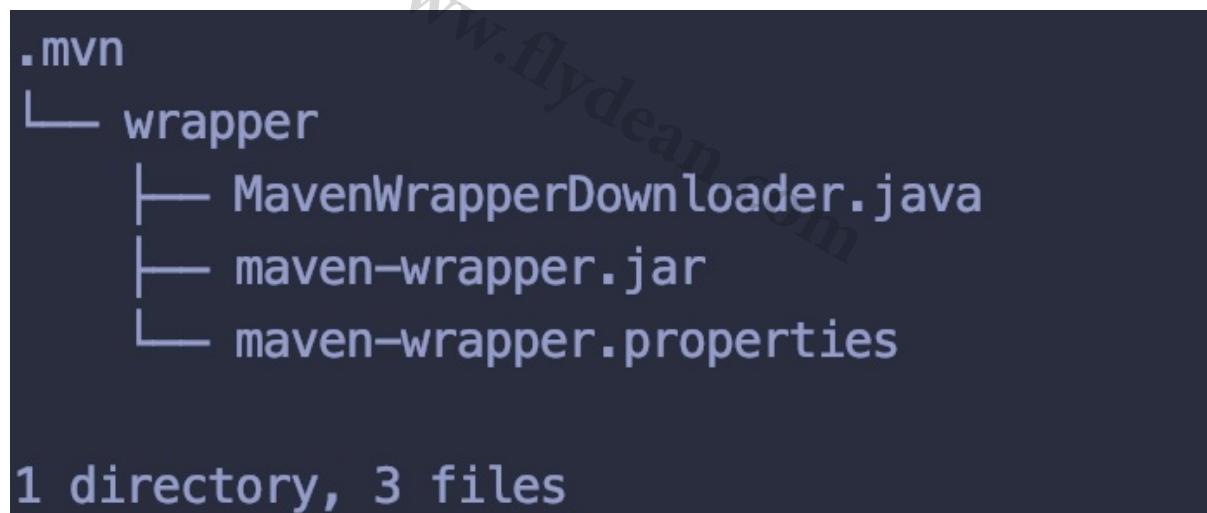
如果大家使用IntelliJ IDEA来开发Spring boot项目, 如果选择从Spring Initializr来创建项目, 则会在项目中自动应用Maven Wrapper。简单点说就是在项目目录下面会多出两个文件: mvnw 和 mvnw.cmd。

Maven Wrapper的结构

mvnw是Linux系统的启动文件。

mvnw.cmd是windows系统的启动文件。

本文不会详细讲解启动文件的内部信息, 有兴趣的小伙伴可以自行去研究。除了这两个启动文件, 在项目中还会生成一个.mvn的隐藏文件夹。如下图所示:



我们再看下 .mvn(wrapper/maven-wrapper.properties) :

```
distributionUrl=https://repo.maven.apache.org/maven2/org/apache/maven/apache-maven/
3.6.3/apache-maven-3.6.3-bin.zip
wrapperUrl=https://repo.maven.apache.org/maven2/io/takari/maven-wrapper/0.5.6/maven-
wrapper-0.5.6.jar
```

这个文件指定了maven和maven wrapper的版本。

下载Maven Wrapper

如果不是使用IntelliJ IDEA， 我们该怎么样下载Maven Wrapper呢？

在程序的主目录下面：

```
mvn -N io.takari:maven:wrapper
```

如果要指定maven版本：

```
mvn -N io.takari:maven:wrapper -Dmaven=3.5.2
```

-N 意思是 -non-recursive， 只会在主目录下载一次。

使用

Maven Wrapper的使用和maven命令是一样的， 比如：

```
./mvnw clean install  
./mvnw spring-boot:run
```

更多教程请参考 [flydean的博客](http://www.flydean.com)

Spring Boot注解

简介

Spring Boot通过自动配置让我们更加简单的使用Spring。在本文中我们将会介绍org.springframework.boot.autoconfigure 和org.springframework.boot.autoconfigure.condition 里面经常会用到的一些注解。

@SpringBootApplication

首先我们看一下@SpringBootApplication:

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

@SpringBootApplication被用在Spring Boot应用程序的Main class中，表示整个应用程序是Spring Boot。

@SpringBootApplication实际上是@Configuration, @EnableAutoConfiguration 和 @ComponentScan的集合。

@EnableAutoConfiguration

@EnableAutoConfiguration 意味着开启了自动配置。这意味着Spring Boot会去在classpath中查找自动配置的beans，并且自动应用他们。

注意， @EnableAutoConfiguration需要和@Configuration配合使用。

```
@Configuration
@EnableAutoConfiguration
public class VehicleFactoryConfig {
```

条件自动配置

有时候，我们在自定义自动配置的时候，希望根据某些条件来开启自动配置，Spring Boot 提供了一些有用的注解。

这些注解可以和@Configuration 类 或者 @Bean 方法一起使用。

@ConditionalOnClass 和 @ConditionalOnMissingClass

这两个注解的意思是，如果注解参数中的类存在或者不存在则Spring会去实例化自动配置的bean。

```
@Configuration  
@ConditionalOnClass(dataSource.class)  
public class MySQLAutoconfiguration {  
}
```

@ConditionalOnBean 和 @ConditionalOnMissingBean

这两个和上面的区别在于，这两个是判断是否有实例化的bean存在。

```
@Bean  
@ConditionalOnBean(name = "dataSource")  
LocalContainerEntityManagerFactoryBean entityManagerFactory() {  
    // ...  
}
```

@ConditionalOnProperty

使用这个注解我们可以判断Property的某些属性是不是需要的值：

```
@Bean  
@ConditionalOnProperty(  
    name = "usemysql",  
    havingValue = "local"  
)  
DataSource dataSource() {  
    // ...  
}
```

@ConditionalOnResource

只有当某些resource存在的时候，才会起作用。

```
@ConditionalOnResource(resources = "classpath:mysql.properties")  
Properties additionalProperties() {  
    // ...  
}
```

@ConditionalOnWebApplication 和 @ConditionalOnNotWebApplication

这两个注解通过判断是否web应用程序。

```
@Bean  
@ConditionalOnWebApplication  
HealthCheckController healthCheckController() {  
    // ...  
    return null;  
}
```

@ConditionalExpression

这个注解可以使用SpEL构造更加复杂的表达式：

```
@Bean  
@ConditionalOnExpression("${usemysql} && ${mysqlserver == 'local'}")  
DataSource dataSource() {  
    // ...  
}
```

@Conditional

还有一种更加复杂的应用叫@Conditional，它的参数是一个自定义的condition类。

```
@Bean  
@Conditional(HibernateCondition.class)  
Properties newAdditionalProperties() {  
    //...  
    return null;  
}
```

```
public class HibernateCondition implements Condition {  
    @Override  
    public boolean matches(ConditionContext conditionContext, AnnotatedTypeMetadata  
    annotatedTypeMetadata) {  
        return false;  
    }  
}
```

这个类需要实现matches方法。

Spring Boot @EnableAutoConfiguration和@Configuration的区别

@Configuration的区别

在Spring Boot中，我们会使用@SpringBootApplication来开启Spring Boot程序。在之前的文章中我们讲到了@SpringBootApplication相当于@EnableAutoConfiguration, @ComponentScan, @Configuration三者的集合。

其中@Configuration用在类上面，表明这个是个配置类，如下所示：

```
@Configuration
public class MySQLAutoconfiguration {
    ...
}
```

而@EnableAutoConfiguration则是开启Spring Boot的自动配置功能。什么是自动配置功能呢？简单点说就是Spring Boot根据依赖中的jar包，自动选择实例化某些配置。

接下来我们看一下@EnableAutoConfiguration是怎么工作的。

先看一下@EnableAutoConfiguration的定义：

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

    String ENABLED_OVERRIDE_PROPERTY = "spring.boot.enableautoconfiguration";

    /**
     * Exclude specific auto-configuration classes such that they will never be applied.
     * @return the classes to exclude
     */
    Class<?>[] exclude() default {};

    /**
     * Exclude specific auto-configuration class names such that they will never be applied.
     * @return the class names to exclude
     * @since 1.3.0
     */
    String[] excludeName() default {};

}
```

注意这一行：@Import(AutoConfigurationImportSelector.class)

AutoConfigurationImportSelector实现了ImportSelector接口，并会在实例化时调用selectImports。下面是其方法：

```
public String[] selectImports(AnnotationMetadata annotationMetadata) {
    if (!isEnabled(annotationMetadata)) {
        return NO_IMPORTS;
    }
    AutoConfigurationMetadata autoConfigurationMetadata = AutoConfigurationMeta
dataLoader
        .loadMetadata(this.beanClassLoader);
    AutoConfigurationEntry autoConfigurationEntry = getAutoConfigurationEntry(a
utoConfigurationMetadata,
        annotationMetadata);
    return StringUtils.toStringArray(autoConfigurationEntry.getConfigurations()
);
}
```

这个方法中的getCandidateConfigurations会从类加载器中查找所有的META-INF/spring.factories，并加载其中实现了@EnableAutoConfiguration的类。有兴趣的朋友可以具体研究一下这个方法的实现。

```
private static Map<String, List<String>> loadSpringFactories(@Nullable ClassLoader
classLoader) {
    MultiValueMap<String, String> result = cache.get(classLoader);
    if (result != null) {
        return result;
    }

    try {
        Enumeration<URL> urls = (classLoader != null ?
            classLoader.getResources(FACTORIES_RESOURCE_LOCATION) :
            ClassLoader.getSystemResources(FACTORIES_RESOURCE_LOCATION));
        result = new LinkedMultiValueMap<>();
        while (urls.hasMoreElements()) {
            URL url = urls.nextElement();
            UrlResource resource = new UrlResource(url);
            Properties properties = PropertiesLoaderUtils.loadProperties(resour
ce);
            for (Map.Entry<?, ?> entry : properties.entrySet()) {
                String factoryTypeName = ((String) entry.getKey()).trim();
                for (String factoryImplementationName : StringUtils.commaDelimi
tedListToStringArray((String) entry.getValue())) {
                    result.add(factoryTypeName, factoryImplementationName.trim(
));
                }
            }
        }
        cache.put(classLoader, result);
    }
```

```

        return result;
    }
    catch (IOException ex) {
        throw new IllegalArgumentException("Unable to load factories from location [" +
            FACTORIES_RESOURCE_LOCATION + "]", ex);
    }
}

```

我们再看一下spring-boot-autoconfigure-2.2.2.RELEASE.jar中的META-INF/spring.factories。

spring.factories里面的内容是key=value形式的，我们重点关注一下EnableAutoConfiguration：

```

# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.admin.SpringApplicationAdminJmxAutoConfigura
tion,\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.cache.CacheAutoConfiguration,\
org.springframework.boot.autoconfigure.cassandra.CassandraAutoConfiguration,\
org.springframework.boot.autoconfigure.cloud.CloudServiceConnectorsAutoConfiguratio
n,\
org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfigura
tion,\
...

```

这里只列出了一部分内容，根据上面的代码，所有的EnableAutoConfiguration的实现都会被自动加载。这就是自动加载的原理了。

如果我们仔细去看具体的实现：

```

@Configuration(proxyBeanMethods = false)
@AutoConfigureAfter(JmxAutoConfiguration.class)
@ConditionalOnProperty(prefix = "spring.application.admin", value = "enabled", hav
ingValue = "true",
    matchIfMissing = false)
public class SpringApplicationAdminJmxAutoConfiguration {

```

可以看到里面使用了很多@Conditional* 的注解，这种注解的作用就是判断该配置类在什么时候能够起作用。

自定义spring boot的自动配置

上篇文章我们讲了spring boot中自动配置的深刻含义和内部结构，这篇文章我们讲一下怎么写出一个自己的自动配置。为了方便和通用起见，这篇文章将会实现一个mysql数据源的自动配置。

添加Maven依赖

我们需要添加mysql和jpa的数据源：

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-data-jpa</artifactId>
    </dependency>
    <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>8.0.18</version>
    </dependency>
</dependencies>
```

创建自定义 Auto-Configuration

我们知道 Auto-Configuration实际上就是一种配置好的@Configuration,所以我们要创建一个MySQL的@Configuration，如下：

```
@Configuration
public class MySQLAutoconfiguration {
```

下一步就是将这个配置类注册到resources下面的/META-INF/spring.factories作为org.springframework.boot.autoconfigure.EnableAutoConfiguration的一个实现：

```
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.flydean.config.MySQLAutoconfiguration
```

如果我们希望自定义的@Configuration拥有最高的优先级，我们可以添加@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE) 如下所示：

```
@Configuration
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
public class MySQLAutoconfiguration {
```

注意，自动配置的bean只有在该bean没有在应用程序中配置的时候才会自动被配置。如果应用程序中已经配置了该bean，则自动配置的bean会被覆盖。

添加Class Conditions

我们的mysqlConfig只有在DataSource这个类存在的时候才会被自动配置。则可以使用@ConditionalOnClass。如果某个类不存在的时候生效则可以使用@ConditionalOnMissingClass。

```
@Configuration
@AutoConfigureOrder(Ordered.HIGHEST_PRECEDENCE)
@ConditionalOnClass(DataSource.class)
public class MySQLAutoconfiguration {
}
```

添加 bean Conditions

如果我们需要的不是类而是bean的实例，则可以使用@ConditionalOnBean 和@ConditionalOnMissingBean。

在本例中，我们希望当dataSource的bean存在的时候实例化一个LocalContainerEntityManagerFactoryBean：

```
@Bean
@ConditionalOnBean(name = "dataSource")
@ConditionalOnMissingBean
public LocalContainerEntityManagerFactoryBean entityManager() {
    LocalContainerEntityManagerFactoryBean em
        = new LocalContainerEntityManagerFactoryBean();
    em.setDataSource(dataSource());
    em.setPackagesToScan("com.flydean.config.example");
    em.setJpaVendorAdapter(new HibernateJpaVendorAdapter());
    if (additionalProperties() != null) {
        em.setJpaProperties(additionalProperties());
    }
    return em;
}
```

同样的，我们可以定义一个transactionManager，只有当JpaTransactionManager不存在的时候才创建：

```
@Bean
@ConditionalOnMissingBean(type = "JpaTransactionManager")
JpaTransactionManager transactionManager(EntityManagerFactory entityManagerFactory) {
    JpaTransactionManager transactionManager = new JpaTransactionManager();
    transactionManager.setEntityManagerFactory(entityManagerFactory);
    return transactionManager;
```

```
}
```

Property Conditions

如果我们想在Spring配置文件中的某个属性存在的情况下实例化bean，则可以使用`@ConditionalOnProperty`。首先我们需要加载这个Spring的配置文件：

```
@PropertySource("classpath:mysql.properties")
public class MySQLAutoconfiguration {
    //...
}
```

我们希望属性文件里usemysql=local的时候创建一个DataSource，则可以这样写：

```
@Bean
@ConditionalOnProperty(
    name = "usemysql",
    havingValue = "local")
@ConditionalOnMissingBean
public DataSource dataSource() {
    DriverManagerDataSource dataSource = new DriverManagerDataSource();

    dataSource.setDriverClassName("com.mysql.cj.jdbc.Driver");
    dataSource.setUrl("jdbc:mysql://localhost:3306/myDb?createDatabaseIfNotExist=true");
    dataSource.setUsername("mysqluser");
    dataSource.setPassword("mysqlpass");
    return dataSource;
}
```

Resource Conditions

当我们需要根据resource文件是否存在来实例化bean的时候，可以使用`@ConditionalOnResource`。

```
@ConditionalOnResource(
    resources = "classpath:mysql.properties")
@Conditional(HibernateCondition.class)
Properties additionalProperties() {
    Properties hibernateProperties = new Properties();

    hibernateProperties.setProperty("hibernate.hbm2ddl.auto",
        env.getProperty("mysql-hibernate.hbm2ddl.auto"));
    hibernateProperties.setProperty("hibernate.dialect",
        env.getProperty("mysql-hibernate.dialect"));
    hibernateProperties.setProperty("hibernate.show_sql",
        env.getProperty("mysql-hibernate.show_sql") != null
```

```

        ? env.getProperty("mysql-hibernate.show_sql") : "false");
    return hibernateProperties;
}

```

我们需要在mysql.properties添加相应的配置：

```

mysql-hibernate.dialect=org.hibernate.dialect.MySQLDialect
mysql-hibernate.show_sql=true
mysql-hibernate.hbm2ddl.auto=create-drop

```

Custom Conditions

除了使用@Condition** 之外，我们还可以继承SpringBootCondition来实现自定义的condition。如下所示：

```

public class HibernateCondition extends SpringBootCondition {

    private static String[] CLASS_NAMES
        = { "org.hibernate.ejb.HibernateEntityManager",
            "org.hibernate.jpa.HibernateEntityManager" };

    @Override
    public ConditionOutcome getMatchOutcome(ConditionContext context,
                                              AnnotatedTypeMetadata metadata) {

        ConditionMessage.Builder message
            = ConditionMessage.forCondition("Hibernate");
        return Arrays.stream(CLASS_NAMES)
            .filter(className -> ClassUtils.isPresent(className, context.getClassLoader()))
            .map(className -> ConditionOutcome
                .match(message.found("class"))
                    .items(ConditionMessage.Style.NORMAL, className))
            .findAny()
            .orElseGet(() -> ConditionOutcome
                .noMatch(message.didNotFind("class", "classes"))
                    .items(ConditionMessage.Style.NORMAL, Arrays.asList(
                        CLASS_NAMES)));
    }
}

```

测试

接下来我们可以测试了：

```

@RunWith(SpringRunner.class)
@SpringBootTest(

```

```
        classes = AutoconfigurationApplication.class)
@EnableJpaRepositories(
    basePackages = { "com.flydean.repository" })
public class AutoconfigurationTest {

    @Autowired
    private MyUserRepository userRepository;

    @Test
    public void whenSaveUser_thenOk() {
        MyUser user = new MyUser("user@email.com");
        userRepository.save(user);
    }
}
```

这里我们因为没有自定义dataSource所以会自动使用自动配置里面的mysql数据源。

停止自动配置

如果我们不想使用刚刚创建的自动配置该怎么做呢？在@SpringBootApplication中exclude MySQLAutoConfiguration.class即可：

```
@SpringBootApplication(exclude={MySQLAutoConfiguration.class})
```

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-custom-autoconfig>

更多教程请参考 [flydean的博客](#)

在Spring Boot中配置web app

本文将会介绍怎么在Spring Boot中创建和配置一个web应用程序。

添加依赖

如果要使用Spring web程序，则需要添加如下依赖：

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

配置端口

正如我们之前文章中提到的，要想配置端口需要在application.properties文件中配置如下：

```
server.port=8083
```

如果你是用的是yaml文件，则：

```
server:
  port: 8083
```

或者通过java文件的形式：

```
@Component
public class CustomizationBean implements
    WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory container) {
        container.setPort(8083);
    }
}
```

配置Context Path

默认情况下，Spring MVC的context path是'/'，如果你想修改，那么可以在配置文件application.properties中修改：

```
server.servlet.contextPath=/springbootapp
```

如果是yaml文件：

```
server:
  servlet:
    contextPath:/springbootapp
```

同样的，可以在java代码中修改：

```
@Component
public class CustomizationBean
implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
    public void customize(ConfigurableServletWebServerFactory container) {
        container.setContextPath("/springbootapp");
    }
}
```

配置错误页面

默认情况下Spring Boot会开启一个whitelabel的功能来处理错误，这个功能本质上是自动注册一个BasicErrorController如果你没有指定错误处理器的话。同样的，这个错误控制器也可以自定义：

```
@RestController
public class MyCustomErrorController implements ErrorController {

    private static final String PATH = "/error";

    @GetMapping(value=PATH)
    public String error() {
        return "Error haven";
    }

    @Override
    public String getErrorPath() {
        return PATH;
    }
}
```

当然，和之前讲过的自定义服务器信息一样，你也可以自定义错误页面，就像在web.xml里面添加error-page：

```
@Component
public class CustomizationBean
implements WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

    @Override
```

```

public void customize(ConfigurableServletWebServerFactory container) {
    container.addErrorPages(new ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
    container.addErrorPages(new ErrorPage("/errorHaven"));
}
}

```

通过这个功能，你可以对错误进行更加细致的分类。

在程序中停止Spring Boot

SpringApplication提供了一个静态的exit()方法，可以通过它来关停一个Spring Boot应用程序：

```

@.Autowired
public void shutDown(ApplicationContext applicationContext) {
    SpringApplication.exit(applicationContext, new ExitCodeGenerator() {
        @Override
        public int getExitCode() {
            return 0;
        }
    });
}

```

第二个参数是一个ExitCodeGenerator的实现，主要用来返回ExitCode。

配置日志级别

我们可以在配置文件中这样配置日志级别：

```

logging.level.org.springframework.web: DEBUG
logging.level.org.hibernate: ERROR

```

注册Servlet

有时候我们需要将程序运行在非嵌套的服务器中，这时候有可能会需要自定义servlet的情况，Spring Boot 也提供了非常棒的支持，我们只需要在ServletRegistrationBean中，注册servlet即可：

```

@Bean
public ServletRegistrationBean servletRegistrationBean() {

    ServletRegistrationBean bean = new ServletRegistrationBean(
        new SpringHelloWorldServlet(), "/springHelloWorld/*");
    bean.setLoadOnStartup(1);
    bean.addInitParameter("message", "SpringHelloWorldServlet special message");
;
    return bean;
}

```

切换嵌套服务器

默认情况下， Spring Boot会使用tomcat作为嵌套的内部服务器，如果想切换成jetty则可以这样：

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

exclude自带的Tomcat，并额外添加spring-boot-starter-jetty即可。

本文的例子可参考：<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-config-webapp>

更多教程请参考 [flydean的博客](#)

从Spring迁移到Spring Boot

Spring Boot给我们的开发提供了一系列的便利，所以我们可能会希望将老的Spring项目转换为新的Spring Boot项目，本篇文章将会探讨如何操作。

请注意，Spring Boot并不是取代Spring，它只是添加了一些自动配置的东西，从而让Spring程序更快更好

添加Spring Boot starters

要想添加Spring Boot，最简单的办法就是添加Spring Boot Starters。

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.2.2.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

添加应用程序入口

每一个Spring Boot程序都需要一个应用程序入口，通常是一个使用@SpringBootApplication注解的main程序：

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

@SpringBootApplication注解是下列注解的组合：

@Configuration，@EnableAutoConfiguration，@ComponentScan。

默认情况下@SpringBootApplication会扫描本package和子package的所有类。所以一般来说SpringBootApplication会放在顶层包下面。

Import Configuration和Components

Spring Boot通常使用自动配置，但是我们也可以手动Import现有的java配置或者xml配置。

对于现有的配置，我们有两个选项，一是将这些配置移动到主Application同级包或者子包下面，方便自动扫描。二是显式导入。

我们看一下怎么显示导入：

```
@SpringBootApplication  
@ComponentScan(basePackages="com.flydean.config")  
@Import(UserRepository.class)  
public class Application {  
    //...  
}
```

如果是xml文件，你也可以这样使用@ImportResource导入：

```
@SpringBootApplication  
@ImportResource("applicationContext.xml")  
public class Application {  
    //...  
}
```

迁移应用程序资源

默认情况下Spring Boot 会查找如下的资源地址：

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

想要迁移的话 我们可以迁移现有资源到上诉的资源地址，也可以使用下面的方法：

```
spring.resources.staticLocations=classpath:/images/,classpath:/jsp/
```

迁移应用程序属性文件

Spring Boot 会在如下的地方查找application.properties或者application.yml 文件：

- * 当前目录
- * 当前目录的/config子目录
- * 在classpath中的/config目录
- * classpath root

我们可以将属性文件移动到上面提到的路径下面。

迁移Spring Web程序

如果要迁移Spring Web程序，我们需要如下几步：

1. 添加spring-boot-starter-web依赖:

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

通过Spring Boot的自动配置，会自动检测classpath中的依赖包，从而自动开启@EnableWebMvc，同时创建一个DispatcherServlet。

如果我们在@Configuration类中使用了@EnableWebMvc注解，则自动配置会失效。

该自动配置同时自动配置了如下3个bean:

- HttpMessageConverter用来转换JSON 和 XML。
- /error mapping用来处理所有的错误
- /static, /public, /resources 或者 /META-INF/resources的静态资源支持。
- 配置View模板

对于web页面，通常不再推荐JSP，而是使用各种模板技术来替换：Thymeleaf, Groovy, FreeMarker, Mustache。我们要做的就是添加如下依赖：

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

template文件在/resources/templates下面。

如果我们仍然需要是用JSP，则需要显示配置如下：

```
spring.mvc.view.prefix=/WEB-INF/views/
spring.mvc.view.suffix=.jsp
```

Spring Boot中的测试

简介

本篇文章我们将会探讨一下怎么在SpringBoot使用测试， Spring Boot有专门的spring-boot-starter-test， 通过使用它可以很方便的在Spring Boot进行测试。

本文将从repository, service, controller, app四个层级来详细描述测试案例。

添加maven依赖

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>test</scope>
</dependency>
```

我们添加spring-boot-starter-test和com.h2database总共两个依赖。H2数据库主要是为了测试方便。

Repository测试

本例中，我们使用JPA，首先创建Entity和Repository：

```
@Entity
@Table(name = "person")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Size(min = 3, max = 20)
    private String name;

    // standard getters and setters, constructors
}
```

```
@Repository
public interface EmployeeRepository extends JpaRepository<Employee, Long> {

    public Employee findByName(String name);
```

```
}
```

测试JPA，我们需要使用@DataJpaTest：

```
@RunWith(SpringRunner.class)
@DataJpaTest
public class EmployeeRepositoryIntegrationTest {

    @Autowired
    private TestEntityManager entityManager;

    @Autowired
    private EmployeeRepository employeeRepository;

    // write test cases here

}
```

@RunWith(SpringRunner.class) 是Junit和Spring Boot test联系的桥梁。

@DataJpaTest为persistence layer的测试提供了如下标准配置：

- 配置H2作为内存数据库
- 配置Hibernate, Spring Data, 和 DataSource
- 实现@EntityScan
- 开启SQL logging

下面是我们的测试代码：

```
@Test
public void whenFindByName_thenReturnEmployee() {
    // given
    Employee alex = new Employee("alex");
    entityManager.persist(alex);
    entityManager.flush();

    // when
    Employee found = employeeRepository.findByName(alex.getName());

    // then
    assertThat(found.getName())
        .isEqualTo(alex.getName());
}
```

在测试中，我们使用了TestEntityManager。TestEntityManager提供了一些通用的对Entity操作的方法。上面的例子中我们使用TestEntityManager向Employee插入了一条数据。

Service测试

在实际的应用程序中，Service通常要使用到Repository。但是在测试中我们可以Mock一个Repository，而不用使用真实的Repository。

先看一下Service：

```
@Service
public class EmployeeServiceImpl implements EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Override
    public Employee getEmployeeByName(String name) {
        return employeeRepository.findByName(name);
    }
}
```

我们再看一下怎么Mock Repository。

```
@RunWith(SpringRunner.class)
public class EmployeeServiceImplIntegrationTest {

    @TestConfiguration
    static class EmployeeServiceImplTestContextConfiguration {

        @Bean
        public EmployeeService employeeService() {
            return new EmployeeServiceImpl();
        }
    }

    @Autowired
    private EmployeeService employeeService;

    @MockBean
    private EmployeeRepository employeeRepository;

    // write test cases here
}
```

看下上面的例子，我们首先使用了@TestConfiguration专门用在测试中的配置信息，在@TestConfiguration中，我们实例化了一个EmployeeService Bean，然后在EmployeeServiceImplIntegrationTest自动注入。

我们还是用了@MockBean，用来Mock一个EmployeeRepository。

我们看下Mock的实现：

```

@Before
public void setUp() {
    Employee alex = new Employee("alex");

    Mockito.when(employeeRepository.findByName(alex.getName()))
        .thenReturn(alex);
}

@Test
public void whenValidName_thenEmployeeShouldBeFound() {
    String name = "alex";
    Employee found = employeeService.getEmployeeByName(name);

    assertThat(found.getName())
        .isEqualTo(name);
}

```

上面的代码中，我们使用Mockito来Mock要返回的数据，然后在接下来的测试中使用。

测试Controller

和测试Service一样，Controller使用到了Service：

```

@RestController
@RequestMapping("/api")
public class EmployeeRestController {

    @Autowired
    private EmployeeService employeeService;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeService.getAllEmployees();
    }
}

```

但是在测试的时候，我们并不需要使用真实的Service，我们需要Mock它。

```

@RunWith(SpringRunner.class)
@WebMvcTest(EmployeeRestController.class)
public class EmployeeControllerIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @MockBean
    private EmployeeService service;
}

```

```
// write test cases here
```

为了测试Controller，我们需要使用到@WebMvcTest，他会为Spring MVC 自动配置所需的组件。

通常情况下@WebMvcTest 会和@MockBean一起使用来提供Mock的具体实现。

@WebMvcTest也提供了自动配置的MockMvc，它为测试MVC Controller提供了更加简单的方式，而不需要启动完整的HTTP server。

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

    Employee alex = new Employee("alex");

    List<Employee> allEmployees = Arrays.asList(alex);

    given(service.getAllEmployees()).willReturn(allEmployees);

    mvc.perform(get("/api/employees")
        .contentType(MediaType.APPLICATION_JSON))
        .andExpect(status().isOk())
        .andExpect(jsonPath("$.size()", hasSize(1)))
        .andExpect(jsonPath("$.name", is(alex.getName())));
}
```

given(service.getAllEmployees()).willReturn(allEmployees); 这一行代码提供了mock的输出。方便后面的测试使用。

@SpringBootTest的集成测试

上面我们讲的都是单元测试，这一节我们讲一下集成测试。

```
@RunWith(SpringRunner.class)
@SpringBootTest(
    webEnvironment = SpringBootTest.WebEnvironment.RANDOM_PORT,
    classes = TestApplication.class)
@Autowired
@TestPropertySource(
    locations = "classpath:application-integrationtest.properties")
public class EmployeeAppIntegrationTest {

    @Autowired
    private MockMvc mvc;

    @Autowired
    private EmployeeRepository repository;
}
```

集成测试需要使用@SpringBootTest，在@SpringBootTest中可以配置webEnvironment，同时如果我们需要自定义测试属性文件可以使用@TestPropertySource。

下面是具体的测试代码：

```
@After
public void resetDb() {
    repository.deleteAll();
}

@Test
public void givenEmployees_whenGetEmployees_thenStatus200() throws Exception {
    createTestEmployee("bob");
    createTestEmployee("alex");

    // @formatter:off
    mvc.perform(get("/api/employees").contentType(MediaType.APPLICATION_JSON))
        .andDo(print())
        .andExpect(status().isOk())
        .andExpect(content().contentTypeCompatibleWith(MediaType.APPLICATION_JSON))
        .andExpect(jsonPath("$.size()", hasSize(greaterThanOrEqualTo(2))))
        .andExpect(jsonPath("$.name", is("bob")))
        .andExpect(jsonPath("$.name", is("alex")));
    // @formatter:on
}

//
private void createTestEmployee(String name) {
    Employee emp = new Employee(name);
    repository.saveAndFlush(emp);
}
```

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-test>

Spring Boot的TestRestTemplate使用

TestRestTemplate和RestTemplate很类似，不过它是专门用在测试环境中的，本文我们将会讲述TestRestTemplate的一些常用方法。

如果我们在测试环境中使用@SpringBootTest，则可以直接使用TestRestTemplate。

添加maven依赖

要使用TestRestTemplate，我们需要首先添加如下的maven依赖：

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

TestRestTemplate VS RestTemplate

TestRestTemplate和RestTemplate的功能很类似，都可以用来和HTTP API进行交互。实际上TestRestTemplate就是RestTemplate的封装。我们看下TestRestTemplate的代码：

```
public class TestRestTemplate {

    private final RestTemplateBuilder builder;

    private final HttpClientOption[] httpClientOptions;

    private final RestTemplate restTemplate;

    ...

    public void setUriTemplateHandler(UriTemplateHandler handler) {
        this.restTemplate.setUriTemplateHandler(handler);
    }

    ...
}
```

以setUriTemplateHandler为例，我们看到实际上TestRestTemplate调用了restTemplate里面的具体方法。

我们看一下TestRestTemplate基本的使用：

```
@Test
public void testGet (){
    TestRestTemplate testRestTemplate = new TestRestTemplate();
    ResponseEntity<String> response = testRestTemplate.
```

```
        getForEntity(FOO_RESOURCE_URL + "/1", String.class);

    assertThat(response.getStatusCode(), equalTo(HttpStatus.OK));
}
```

使用Basic Auth Credentials

TestRestTemplate封装了基本的Auth Credentials，我们可以这样使用：

```
TestRestTemplate testRestTemplate
= new TestRestTemplate("user", "passwd");
ResponseEntity<String> response = testRestTemplate.
    getForEntity(URL_SECURED_BY_AUTHENTICATION, String.class);

assertThat(response.getStatusCode(), equalTo(HttpStatus.OK));
```

使用HttpClientOption

HttpClientOption提供了如下几个选项：ENABLE_COOKIES, ENABLE_REDIRECTS, 和 SSL。

我们看下TestRestTemplate怎么使用：

```
TestRestTemplate testRestTemplate = new TestRestTemplate("user",
    "passwd", TestRestTemplate.HttpClientOption.ENABLE_COOKIES);
ResponseEntity<String> response = testRestTemplate.
    getForEntity(URL_SECURED_BY_AUTHENTICATION, String.class);

assertThat(response.getStatusCode(), equalTo(HttpStatus.OK));
```

如果我们不需要认证，则可以这样使用：

```
TestRestTemplate(TestRestTemplate.HttpClientOption.ENABLE_COOKIES)
```

我们也可以在创建TestRestTemplate之后添加认证：

```
TestRestTemplate testRestTemplate = new TestRestTemplate();
ResponseEntity<String> response = testRestTemplate.withBasicAuth(
    "user", "passwd").getForEntity(URL_SECURED_BY_AUTHENTICATION,
    String.class);

assertThat(response.getStatusCode(), equalTo(HttpStatus.OK));
```

使用RestTemplateBuilder

RestTemplateBuilder为我们提供了自定义RestTemplate的机会，我们可以使用它来对RestTemplate进行封装：

```
RestTemplateBuilder restTemplateBuilder = new RestTemplateBuilder();
restTemplateBuilder.configure(restTemplate);
TestRestTemplate testRestTemplate = new TestRestTemplate(restTemplateBuilder);
ResponseEntity<String> response = testRestTemplate.getForEntity(
    FOO_RESOURCE_URL + "/1", String.class);

assertThat(response.getStatusCode(), equalTo(HttpStatus.OK));
```

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-testRestTemplate>

Spring Boot Actuator

Spring Boot Actuator 在Spring Boot第一个版本发布的时候就有了，它为Spring Boot提供了一系列产品级的特性：监控应用程序，收集元数据，运行情况或者数据库状态等。

使用Spring Boot Actuator我们可以直接使用这些特性而不需要自己去实现，它是用HTTP或者JMX来和外界交互。

开始使用Spring Boot Actuator

要想使用Spring Boot Actuator，需要添加如下依赖：

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

开始使用Actuator

配好上面的依赖之后，我们使用下面的主程序入口就可以使用Actuator了：

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

启动应用程序，访问<http://localhost:8080/actuator>:

```
{"_links": {"self": {"href": "http://localhost:8080/actuator", "templated": false}, "health": {"href": "http://localhost:8080/actuator/health", "templated": false}, "health-path": {"href": "http://localhost:8080/actuator/health/{path}", "templated": true}, "info": {"href": "http://localhost:8080/actuator/info", "templated": false}}}
```

我们可以看到actuator默认开启了两个入口：/health和/info。

如果我们在配置文件里面这样配置，则可以开启actuator所有的入口：

```
management.endpoints.web.exposure.include=*
```

重启应用程序，再次访问<http://localhost:8080/actuator>:

```
{"_links": {"self": {"href": "http://localhost:8080/actuator", "templated": false}, "beans": {"href": "http://localhost:8080/actuator/beans", "templated": false}, "caches-cache": {"href": "http://localhost:8080/actuator/caches/{cache}", "templated": true}, "caches": {"href": "http://localhost:8080/actuator/caches", "templated": false}, "health": {"href": "http://localhost:8080/actuator/health", "templated": false}, "health-path": {"href": "http://localhost:8080/actuator/health/{*path}", "templated": true}, "info": {"href": "http://localhost:8080/actuator/info", "templated": false}, "conditions": {"href": "http://localhost:8080/actuator/conditions", "templated": false}, "configprops": {"href": "http://localhost:8080/actuator/configprops", "templated": false}, "env": {"href": "http://localhost:8080/actuator/env", "templated": false}, "env-toMatch": {"href": "http://localhost:8080/actuator/env/{toMatch}", "templated": true}, "loggers-name": {"href": "http://localhost:8080/actuator/loggers/{name}", "templated": true}, "loggers": {"href": "http://localhost:8080/actuator/loggers", "templated": false}, "heapdump": {"href": "http://localhost:8080/actuator/heapdump", "templated": false}, "threaddump": {"href": "http://localhost:8080/actuator/threaddump", "templated": false}, "metrics": {"href": "http://localhost:8080/actuator/metrics", "templated": false}, "metrics-requiredMetricName": {"href": "http://localhost:8080/actuator/metrics/{requiredMetricName}", "templated": true}, "scheduledtasks": {"href": "http://localhost:8080/actuator/scheduledtasks", "templated": false}, "mappings": {"href": "http://localhost:8080/actuator/mappings", "templated": false}}}}
```

我们可以看到actuator暴露的所有入口。

Health Indicators

Health入口是用来监控组件的状态的，通过上面的入口，我们可以看到Health的入口如下：

```
"health": {"href": "http://localhost:8080/actuator/health", "templated": false}, "health-path": {"href": "http://localhost:8080/actuator/health/{*path}", "templated": true},
```

有两个入口，一个是总体的health，一个是具体的health-path。

我们访问一下<http://localhost:8080/actuator/health>:

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

上面的结果实际上是隐藏了具体的信息，我们可以通过设置

```
management.endpoint.health.show-details=ALWAYS
```

来开启详情，开启之后访问如下：

```
{"status": "UP", "components": {"db": {"status": "UP", "details": {"database": "H2", "result": 1, "validationQuery": "SELECT 1"}}, "diskSpace": {"status": "UP", "details": {"total": 250685575168, "free": 12428898304, "threshold": 10485760}}, "ping": {"status": "UP"}}}
```

其中的components就是health-path,我们可以访问具体的某一个components如<http://localhost:8080/actuator/health/db>:

```
{"status": "UP", "details": {"database": "H2", "result": 1, "validationQuery": "SELECT 1"}}
```

就可以看到具体某一个component的信息。

这些Health components的信息都是收集实现了HealthIndicator接口的bean来的。

我们看下怎么自定义HealthIndicator:

```
@Component
public class CustHealthIndicator implements HealthIndicator {

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

    public int check() {
        // Our logic to check health
        return 0;
    }
}
```

再次查看<http://localhost:8080/actuator/health>, 我们会发现多了一个Cust的组件:

```
"components": {"cust": {"status": "UP"}}
```

在Spring Boot 2.X之后, Spring添加了Reactive的支持, 我们可以添加ReactiveHealthIndicator如下:

```
@Component
public class DownstreamServiceHealthIndicator implements ReactiveHealthIndicator {

    @Override
    public Mono<Health> health() {
        return checkDownstreamServiceHealth().onErrorResume(
            ex -> Mono.just(new Health.Builder().down(ex).build())
        );
    }

    private Mono<Health> checkDownstreamServiceHealth() {
        // we could use WebClient to check health reactively
    }
}
```

```

        return Mono.just(new Health.Builder().up().build());
    }
}

```

再次查看<http://localhost:8080/actuator/health>, 可以看到又多了一个组件:

```
"downstreamService": {"status": "UP"}
```

/info 入口

info显示了App的大概信息, 默认情况下是空的。我们可以这样自定义:

```

info.app.name=Spring Sample Application
info.app.description=This is my first spring boot application
info.app.version=1.0.0

```

查看: <http://localhost:8080/actuator/info>

```
{"app": {"name": "Spring Sample Application", "description": "This is my first spring boot application", "version": "1.0.0"}}
```

/metrics 入口

/metrics提供了JVM和操作系统的一些信息, 我们看下metrics的目录, 访问:

<http://localhost:8080/actuator/metrics>:

```
{"names": ["jvm.memory.max", "jvm.threads.states", "jdbc.connections.active", "process.files.max", "jvm.gc.memory.promoted", "system.load.average.1m", "jvm.memory.used", "jvm.gc.max.data.size", "jdbc.connections.max", "jdbc.connections.min", "jvm.gc.pause", "jvm.memory.committed", "system.cpu.count", "logback.events", "http.server.requests", "jvm.buffer.memory.used", "tomcat.sessions.created", "jvm.threads.daemon", "system.cpu.usage", "jvm.gc.memory.allocated", "hikaricp.connections.idle", "hikaricp.connections.pending", "jdbc.connections.idle", "tomcat.sessions.expired", "hikaricp.connections", "jvm.threads.live", "jvm.threads.peak", "hikaricp.connections.active", "hikaricp.connections.creation", "process.uptime", "tomcat.sessions.rejected", "process.cpu.usage", "jvm.classes.loaded", "hikaricp.connections.max", "hikaricp.connections.min", "jvm.classes.unloaded", "tomcat.sessions.active.current", "tomcat.sessions.alive.max", "jvm.gc.live.data.size", "hikaricp.connections.usage", "hikaricp.connections.timeout", "process.files.open", "jvm.buffer.count", "jvm.buffer.total.capacity", "tomcat.sessions.active.max", "hikaricp.connections.acquire", "process.start.time"]}
```

访问其中具体的某一个组件如下<http://localhost:8080/actuator/metrics/jvm.memory.max>:

```
{"name": "jvm.memory.max", "description": "The maximum amount of memory in bytes that
```

```
can be used for memory management", "baseUnit": "bytes", "measurements": [{"statistic": "VALUE", "value": 3.456106495E9}], "availableTags": [{"tag": "area", "values": ["heap", "no heap"]}, {"tag": "id", "values": ["Compressed Class Space", "PS Survivor Space", "PS Old Gen", "Metaspace", "PS Eden Space", "Code Cache"]}]}]
```

Spring Boot 2.X 的metrics是通过Micrometer来实现的，Spring Boot会自动注册MeterRegistry。有关Micrometer和Spring Boot的结合使用我们会在后面的文章中详细讲解。

自定义Endpoint

Spring Boot的Endpoint也是可以自定义的：

```
@Component
@Endpoint(id = "features")
public class FeaturesEndpoint {

    private Map<String, String> features = new ConcurrentHashMap<>();

    @ReadOperation
    public Map<String, String> features() {
        return features;
    }

    @ReadOperation
    public String feature(@Selector String name) {
        return features.get(name);
    }

    @WriteOperation
    public void configureFeature(@Selector String name, String value) {
        features.put(name, value);
    }

    @DeleteOperation
    public void deleteFeature(@Selector String name) {
        features.remove(name);
    }
}
```

访问<http://localhost:8080/actuator/>，我们会发现多了一个入口：

<http://localhost:8080/actuator/features/>。

上面的代码中@ReadOperation对应的是GET，@WriteOperation对应的是PUT，@DeleteOperation对应的是DELETE。

@Selector后面对应的是路径参数，比如我们可以这样调用configureFeature方法：

```
POST /actuator/features/abc HTTP/1.1
Host: localhost:8080
Content-Type: application/json
User-Agent: PostmanRuntime/7.18.0
Accept: /*
Cache-Control: no-cache
Postman-Token: dbb46150-9652-4a4a-95cb-3a68c9aa8544, 8a033af4-c199-4232-953b-d22dad7
8c804
Host: localhost:8080
Accept-Encoding: gzip, deflate
Content-Length: 15
Connection: keep-alive
cache-control: no-cache

{"value":true}
```

注意，这里的请求BODY是以JSON形式提供的：

```
{"value":true}
```

请求URL：/actuator/features/abc 中的abc就是@Selector 中的 name。

我们再看一下GET请求：

<http://localhost:8080/actuator/features/>

```
{"abc":"true"}
```

这个就是我们之前PUT进去的值。

扩展现有的Endpoints

我们可以使用@EndpointExtension (@EndpointWebExtension或者@EndpointJmxExtension) 来实现对现有EndPoint的扩展：

```
@Component
@EndpointWebExtension(endpoint = InfoEndpoint.class)
public class InfoWebEndpointExtension {

    private InfoEndpoint delegate;

    // standard constructor

    @ReadOperation
    public WebEndpointResponse<Map> info() {
        Map<String, Object> info = this.delegate.info();
        Integer status = getStatus(info);
        return WebEndpointResponse.of(info).status(status);
    }
}
```

```
        return new WebEndpointResponse<>(info, status);
    }

    private Integer getStatus(Map<String, Object> info) {
        // return 5xx if this is a snapshot
        return 200;
    }
}
```

上面的例子扩展了InfoEndpoint。

本文所提到的例子可以参考：<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-actuator>

Spring Boot中的Properties

简介

本文我们将会讨论怎么在Spring Boot中使用Properties。使用Properties有两种方式，一种是java代码的注解，一种是xml文件的配置。本文将会主要关注java代码的注解。

使用注解注册一个Properties文件

注册Properties文件我们可以使用@PropertySource注解，该注解需要配合@Configuration一起使用。

```
@Configuration  
@PropertySource("classpath:foo.properties")  
public class PropertiesWithJavaConfig {  
    // ...  
}
```

我们也可以使用placeholder来动态选择属性文件：

```
@PropertySource({  
    "classpath:persistence-${envTarget:mysql}.properties"  
})
```

@PropertySource也可以多次使用来定义多个属性文件：

```
@PropertySource("classpath:foo.properties")  
@PropertySource("classpath:bar.properties")  
public class PropertiesWithJavaConfig {  
    // ...  
}
```

我们也可以使用@PropertySources来包含多个@PropertySource：

```
@PropertySources({  
    @PropertySource("classpath:foo.properties"),  
    @PropertySource("classpath:bar.properties")  
})  
public class PropertiesWithJavaConfig {  
    // ...  
}
```

使用属性文件

最简单直接的使用办法就是使用@Value注解:

```
@Value( "${jdbc.url}" )
private String jdbcUrl;
```

我们也可以给属性添加默认值:

```
@Value( "${jdbc.url:aDefaultUrl}" )
private String jdbcUrl;
```

如果要在代码中使用属性值, 我们可以从Environment API中获取:

```
@Autowired
private Environment env;
...
dataSource.setUrl(env.getProperty("jdbc.url"));
```

Spring Boot中的属性文件

默认情况下Spring Boot会读取application.properties文件作为默认的属性文件。当然, 我们也可以在命令行提供一个不同的属性文件:

```
java -jar app.jar --spring.config.location=classpath:/another-location.properties
```

如果是在测试环境中, 我们可以使用@TestPropertySource来指定测试的属性文件:

```
@RunWith(SpringRunner.class)
@TestPropertySource("/foo.properties")
public class FilePropertyInjectionUnitTests {

    @Value("${foo}")
    private String foo;

    @Test
    public void whenFilePropertyProvided_thenProperlyInjected() {
        assertThat(foo).isEqualTo("bar");
    }
}
```

除了属性文件, 我们也可以直接以key=value的形式:

```
@RunWith(SpringRunner.class)
@TestPropertySource(properties = {"foo=bar"})
public class PropertyInjectionUnitTests {
```

```

@Value("${foo}")
private String foo;

@Test
public void whenPropertyProvided_thenProperlyInjected() {
    assertThat(foo).isEqualTo("bar");
}
}

```

使用@SpringBootTest，我们也可以使用类似的功能：

```

@RunWith(SpringRunner.class)
@SpringBootTest(properties = {"foo=bar"}, classes = SpringBootPropertiesTestApplication.class)
public class SpringBootPropertyInjectionIntegrationTest {

    @Value("${foo}")
    private String foo;

    @Test
    public void whenSpringBootPropertyProvided_thenProperlyInjected() {
        assertThat(foo).isEqualTo("bar");
    }
}

```

@ConfigurationProperties

如果我们有一组属性，想将这些属性封装成一个bean，则可以考虑使用@ConfigurationProperties。

```

@ConfigurationProperties(prefix = "database")
public class Database {
    String url;
    String username;
    String password;

    // standard getters and setters
}

```

属性文件如下：

```

database.url=jdbc:postgresql:/localhost:5432/instance
database.username=foo
database.password=bar

```

Spring Boot将会自动将这些属性文件映射成java bean的属性，我们需要做的就是定义好prefix。

yaml文件

Spring Boot也支持yaml形式的文件，yaml对于层级属性来说更加友好和方便，我们可以看下properties文件和yaml文件的对比：

```
database.url=jdbc:postgresql:/localhost:5432/instance
database.username=foo
database.password=bar
secret: foo
```

```
database:
  url: jdbc:postgresql:/localhost:5432/instance
  username: foo
  password: bar
  secret: foo
```

注意yaml文件不能用在@PropertySource中。如果你使用@PropertySource，则必须指定properties文件。

Properties环境变量

我们可以这样传入property环境变量：

```
java -jar app.jar --property="value"
```

```
~~shell java -Dproperty.name="value" -jar app.jar
```

或者这样：

```
~~~shell
export name=value
java -jar app.jar
```

环境变量有什么用呢？当指定了特定的环境变量时候，Spring Boot会自动去加载application-environment.properties文件，Spring Boot默认的属性文件也会被加载，只不过优先级比较低。

java代码配置

除了注解和默认的属性文件，java也可以使用PropertySourcesPlaceholderConfigurer来在代码中显示加载：

```
@Bean
public static PropertySourcesPlaceholderConfigurer properties(){
```

```
PropertySourcesPlaceholderConfigurer pspc  
    = new PropertySourcesPlaceholderConfigurer();  
Resource[] resources = new ClassPathResource[ ]  
{ new ClassPathResource( "foo.properties" ) };  
pspc.setLocations( resources );  
pspc.setIgnoreUnresolvablePlaceholders( true );  
return pspc;  
}
```

本文的例子可以参考：<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-properties>

www.flydean.com

SpringBoot @ConfigurationProperties详解

简介

本文将会详细讲解@ConfigurationProperties在Spring Boot中的使用。

添加依赖关系

首先我们需要添加Spring Boot依赖：

```
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <relativePath/> <!-- lookup parent from repository -->
</parent>
```

一个简单的例子

@ConfigurationProperties需要和@Configuration配合使用，我们通常在一个POJO里面进行配置：

```
@Data
@Configuration
@ConfigurationProperties(prefix = "mail")
public class ConfigProperties {

    private String hostName;
    private int port;
    private String from;
}
```

上面的例子将会读取properties文件中所有以mail开头的属性，并和bean中的字段进行匹配：

```
#Simple properties
mail.hostname=host@mail.com
mail.port=9000
mail.from=mailer@mail.com
```

Spring的属性名字匹配支持很多格式，如下所示所有的格式都可以和hostName进行匹配：

```
mail.hostName
mail.hostname
mail.host_name
mail.host-name
mail.HOST_NAME
```

如果你不想使用@Configuration， 那么需要在@EnableConfigurationProperties注解中手动导入配置文件如下：

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

我们也可以在@ConfigurationPropertiesScan中指定Config文件的路径：

```
@SpringBootApplication
@ConfigurationPropertiesScan("com.flydean.config")
public class ConfigPropApp {
    public static void main(String[] args) {
        SpringApplication.run(ConfigPropApp.class, args);
    }
}
```

这样的话程序只会在com.flydean.config包中查找config文件。

属性嵌套

我们可以嵌套class, list, map, 下面我们举个例子，先创建一个普通的POJO：

```
@Data
public class Credentials {
    private String authMethod;
    private String username;
    private String password;
}
```

然后创建一个嵌套的配置文件：

```
@Data
@Configuration
@ConfigurationProperties(prefix = "nestmail")
public class NestConfigProperties {
    private String host;
    private int port;
    private String from;
    private List<String> defaultRecipients;
    private Map<String, String> additionalHeaders;
    private Credentials credentials;
```

```
}
```

对应的属性文件如下：

```
# nest Simple properties
nestmail.hostname=mailer@mail.com
nestmail.port=9000
nestmail.from=mailer@mail.com

#List properties
nestmail.defaultRecipients[0]=admin@mail.com
nestmail.defaultRecipients[1]=owner@mail.com

#Map Properties
nestmail.additionalHeaders.redelivery=true
nestmail.additionalHeaders.secure=true

#Object properties
nestmail.credentials.username=john
nestmail.credentials.password=password
nestmail.credentials.authMethod=SHA1
```

@ConfigurationProperties和@Bean

@ConfigurationProperties也可以和@Bean一起使用如下所示：

```
@Data
public class Item {
    private String name;
    private int size;
}
```

看下怎么使用：

```
@Data
@Configuration
public class BeanConfigProperties {
    @Bean
    @ConfigurationProperties(prefix = "item")
    public Item item() {
        return new Item();
    }
}
```

属性验证

@ConfigurationProperties可以使用标准的JSR-303格式来做属性验证。我们举个例子：

```
@Data  
@Validated  
@Configuration  
@ConfigurationProperties(prefix = "mail")  
public class ConfigProperties {  
  
    @NotEmpty  
    private String hostName;  
    @Min(1025)  
    @Max(65536)  
    private int port;  
    @Pattern(regexp = "^[a-zA-Z0-9._%+-]+@[a-zA-Z.-]+\\.[a-zA-Z]{2,6}$")  
    private String from;  
}
```

如果我们的属性不满足上诉条件，可能出现如下异常：

```
Binding to target org.springframework.boot.context.properties.bind.BindException: Failed to bind properties under 'mail' to com.flydean.config.ConfigProperties$$EnhancerBySpringCGLIB$$f0f87cb9 failed:  
  
Property: mail.port  
Value: 0  
Reason: 最小不能小于1025  
  
Property: mail.hostName  
Value: null  
Reason: 不能为空  
  
Action:  
  
Update your application's configuration  
  
Process finished with exit code 1
```

属性转换

@ConfigurationProperties也支持多种属性转换，下面我们以Duration和DataSize为例：

我们定义两个Duration的字段：

```
@ConfigurationProperties(prefix = "conversion")  
public class PropertyConversion {
```

```

private Duration timeInDefaultUnit;
private Duration timeInNano;
...
}

```

在属性文件中定义这两个字段：

```

conversion.timeInDefaultUnit=10
conversion.timeInNano=9ns

```

我们看到上面的属性可以带单位的。可选的单位是：ns, us, ms, s, m, h 和 d，分别对应纳秒，微妙，毫秒，秒，分钟，小时和天。默认单位是毫秒。我们也可以在注解中指定单位：

```

@DurationUnit(ChronoUnit.DAYS)
private Duration timeInDays;

```

对应的配置文件如下：

```

conversion.timeInDays=2

```

下面我们再看看DataSize怎么使用：

```

private DataSize sizeInDefaultUnit;

private DataSize sizeInGB;

@DataSizeUnit(DataUnit.TERABYTES)
private DataSize sizeInTB;

```

对应的属性文件：

```

conversion.sizeInDefaultUnit=300
conversion.sizeInGB=2GB
conversion.sizeInTB=4

```

Datasize支持B, KB, MB, GB 和TB。

自定义Converter

同样的Spring Boot也支持自定义属性转换器。我们先定义一个POJO类：

```

public class Employee {
    private String name;
    private double salary;
}

```

对应的属性文件：

```
conversion.employee=john,2000
```

我们需要自己实现一个Converter接口的转换类：

```
@Component
@ConfigurationPropertiesBinding
public class EmployeeConverter implements Converter<String, Employee> {

    @Override
    public Employee convert(String from) {
        String[] data = from.split(",");
        return new Employee(data[0], Double.parseDouble(data[1]));
    }
}
```

本文的例子可以参看: <https://github.com/ddean2009/learn-springboot2/tree/master/springboot-ConfigurationProperties>

更多教程请参考 [flydean的博客](#)

在Spring Boot中加载初始化数据

在Spring Boot中，Spring Boot会自动搜索映射的Entity，并且创建相应的table，但是有时候我们希望自定义某些内容，这时候我们就需要使用到data.sql和schema.sql。

依赖条件

Spring Boot的依赖我们就不将了，因为本例将会有数据库的操作，我们这里使用H2内存数据库方便测试：

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

我们需要使用JPA来创建Entity：

```
@Entity
public class Country {
    @Id
    @GeneratedValue(strategy = IDENTITY)
    private Integer id;

    @Column(nullable = false)
    private String name;

    ...
}
```

我们这样定义repository：

```
public interface CountryRepository extends CrudRepository<Country, Integer> {
    List<Country> findByName(String name);
}
```

如果这时候我们启动Spring Boot程序，将会自动创建Country表。

data.sql文件

上面我们创建好了数据表格，我们可以使用data.sql来加载文件：

```
INSERT INTO country (name) VALUES ('India');
```

```
INSERT INTO country (name) VALUES ('Brazil');
INSERT INTO country (name) VALUES ('USA');
INSERT INTO country (name) VALUES ('Italy');
```

在data.sql文件中我们插入了4条数据，可以写个测试例子测试一下：

```
@RunWith(SpringRunner.class)
@SpringBootTest(classes = LoadIniDataApp.class)
public class SpringBootInitialLoadIntegrationTest {

    @Autowired
    private CountryRepository countryRepository;

    @Test
    public void testInitDataForTestClass() {
        assertEquals(4, countryRepository.count());
    }
}
```

schema.sql 文件

有时候我们需要自定义数据库的schema，这时候我们可以使用到schema.sql文件。

先看一个schema.sql的例子：

```
CREATE TABLE country (
    id   INTEGER      NOT NULL AUTO_INCREMENT,
    name VARCHAR(128) NOT NULL,
    PRIMARY KEY (id)
);
```

Spring Boot会自动加载这个schema文件。

我们需要关闭spring boot的schema自动创建功能以防冲突：

```
spring.jpa.hibernate.ddl-auto=none
```

spring.jpa.hibernate.ddl-auto 有如下几个选项：

- **create** : 首先drop现有的tables，然后创建新的tables
- **update** : 这个模式不会删除现有的tables，它会比较现有的tables和新的注解或者xml配置是否一致，然后更新。
- **create-drop** : 和create很类似，不同的是会在程序运行完毕后自动drop掉tables。通常用在单元测试中。
- **validate** : 只会做table是否存在验证，不存在则会报错。
- **none** : 关闭ddl自动生成功能。

如果Spring Boot没有检测到自定义的schema manager的话，则会自动使用create-drop模式。否则使用none模式。

@Sql注解

@Sql是测试包中的一个注解，可以显示的导入要执行的sql文件，它可以用在class上或者方法之上，如下所示：

```
    @Test
    @Sql({"classpath:new_country.sql"})
    public void testLoadDataForTestCase() {
        assertEquals(6, countryRepository.count());
    }
```

上面的例子将会显示的导入classpath中的new_country.sql文件。

@Sql有以下几个属性：

- config：用来配置SQL脚本，我们在下面的@SqlConfig详细讲解。
- executionPhase：可以选择脚本是在BEFORE_TEST_METHOD 或者 AFTER_TEST_METHOD 来执行。
- statements：可以接受内联的sql语句
- scripts：可以指明要执行脚本的路径

@SqlConfig注解

@SqlConfig主要用在class级别或者用在@Sql注解的config属性中：

```
    @Sql(scripts = {"classpath:new_country2.sql"},
          config = @SqlConfig(encoding = "utf-8", transactionMode = SqlConfig.TransactionMode.ISOLATED))
```

@SqlConfig有以下几个属性：

- blockCommentStartDelimiter：指明了SQL脚本的开始标记。
- blockCommentEndDelimiter：SQL脚本的结束标记。
- commentPrefix：SQL 脚本的注释标记
- dataSource：javax.sql.DataSource的名字，指定该脚本将会在什么datasource下执行
- encoding：SQL 文件的编码
- errorMode：脚本遇到错误的处理模式
- separator：分隔符
- transactionManager：指定的PlatformTransactionManager
- transactionMode：事务模式

@Sql是可以多个同时使用的，如下所示：

```
@Test  
@Sql({ "classpath:new_country.sql"})  
@Sql(scripts = {"classpath:new_country2.sql"},  
      config = @SqlConfig(encoding = "utf-8", transactionMode = SqlConfig.TransactionMode.ISOLATED))  
public void testLoadDataForTestCase() {  
    assertEquals(6, countryRepository.count());  
}
```

本文的例子可以参考：<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-load-init-data>

更多教程请参考 [flydean的博客](#)

Spring Boot的exit code

任何应用程序都有exit code，这个code是int值包含负值，在本文中我们将会探讨Spring Boot中的exit code。

Spring Boot的exit code

Spring Boot如果启动遇到错误，则会返回1.正常退出的话则会返回0.

Spring Boot向JVM注册了shutdown hooks来保证应用程序优雅的退出。Spring Boot还提供了org.springframework.boot.ExitCodeGenerator接口，来方便自定义退出code.

自定义Exit Codes

Spring Boot提供了三种方式来让我们自定义exit code。

ExitCodeGenerator, ExitCodeExceptionMapper和ExitCodeEvent。下面我们分别来讲解。

ExitCodeGenerator

实现ExitCodeGenerator接口，我们需要自己实现getExitCode()方法来自定义返回代码：

```
@SpringBootApplication
public class ExitCodeApp implements ExitCodeGenerator {
    public static void main(String[] args) {
        System.exit(SpringApplication.exit(SpringApplication.run(ExitCodeApp.class,
            args)));
    }

    @Override
    public int getExitCode() {
        return 11;
    }
}
```

这里我们调用了System.exit方法来返回特定的代码。

ExitCodeExceptionMapper

如果我们遇到runtime exception的时候，可以使用ExitCodeExceptionMapper来做错误代码的映射如下：

```
@Bean
CommandLineRunner createException() {
    return args -> Integer.parseInt("test") ;
```

```
}

@Bean
ExitCodeExceptionMapper exitCodeToExceptionMapper() {
    return exception -> {
        // set exit code base on the exception type
        if (exception.getCause() instanceof NumberFormatException) {
            return 80;
        }
        return 1;
    };
}
```

上面的例子我们创建了一个CommandLineRunner bean，在实例化的过程中会抛出NumberFormatException，然后在ExitCodeExceptionMapper中，我们会捕捉到这个异常，返回特定的返回值。

ExitCodeEvent

我们还可以使用ExitCodeEvent来捕捉异常事件如下所示：

```
@Bean
DemoListener demoListenerBean() {
    return new DemoListener();
}

private static class DemoListener {
    @EventListener
    public void exitEvent(ExitCodeEvent event) {
        System.out.println("Exit code: " + event.getExitCode());
    }
}
```

当应用程序退出的时候，exitEvent() 方法会被调用。

本文的例子可以参考：<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-exitcode>

Shutdown SpringBoot App

Spring Boot使用ApplicationContext来创建，初始化和销毁所用的bean。本文将会讲解如何shut down一个spring boot应用程序。

Shutdown Endpoint

Spring Boot actuator自带了shutdown的endpoint。首先我们添加pom依赖：

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

接下来我们需要开启shutdown的配置：

```
management.endpoints.web.exposure.include=*
management.endpoint.shutdown.enabled=true
```

上面的配置对外暴露了 /shutdown 接口。我们可以直接这样调用：

```
curl -X POST localhost:8080/actuator/shutdown
```

close Application Context

我们也可以直接调用Application Context的close() 方法来关闭Application Context。

```
@SpringBootApplication
public class ConfigurableApp {

    public static void main(String[] args) {
        ConfigurableApplicationContext ctx = new
            SpringApplicationBuilder(ConfigurableApp.class).web(WebApplicationT
ype.NONE).run();
        System.out.println("Spring Boot application started");
        ctx.getBean(TerminateBean.class);
        ctx.close();
    }
}
```

为了验证App是否被关闭，我们可以在TerminateBean中添加@PreDestroy来监测App是否被关闭：

```

@Component
public class TerminateBean {

    @PreDestroy
    public void onDestroy() throws Exception {
        System.out.println("Spring Container is destroyed!");
    }
}

```

这是程序的输出：

```

2020-02-03 23:12:08.583  INFO 30527 --- [           main] com.flydean.ConfigurableApp       : Started ConfigurableApp in 2.922 seconds (JVM running for 3.559)
Spring Boot application started
Spring Container is destroyed!

```

还有一种办法就是暴露close接口如下所示：

```

@RestController
public class ShutdownController implements ApplicationContextAware {

    private ApplicationContext context;

    @PostMapping("/shutdownContext")
    public void shutdownContext() {
        ((ConfigurableApplicationContext) context).close();
    }

    @Override
    public void setApplicationContext(ApplicationContext ctx) throws BeansException
    {
        this.context = ctx;
    }
}

```

这样我们就可以通过/shutdownContext接口来关闭ApplicationContext。

退出SpringApplication

上篇文章我们讲过可以通过实现ExitCodeGenerator 接口来返回特定的exit code：

```

@SpringBootApplication
public class ExitCodeApp implements ExitCodeGenerator {
    public static void main(String[] args) {
        System.exit(SpringApplication.exit(SpringApplication.run(ExitCodeApp.class,

```

```
    args));
}

@Override
public int getExitCode() {
    return 11;
}
}
```

从外部程序kill App

熟悉shell的同学都知道如果想在外部kill一个程序，需要知道该App的pid，Spring Boot也可以很方便的生成pid：

```
@SpringBootApplication
public class KillApp {
    public static void main(String[] args) {
        SpringApplicationBuilder app = new SpringApplicationBuilder(KillApp.class)
            .web(WebApplicationType.NONE);
        app.build().addListeners(new ApplicationPidFileWriter("./bin/shutdown.pid"))
    };
    app.run();
}
}
```

上面的程序将会在./bin/shutdown.pid生成应用程序的pid,供shell使用。

我们可以这样使用：

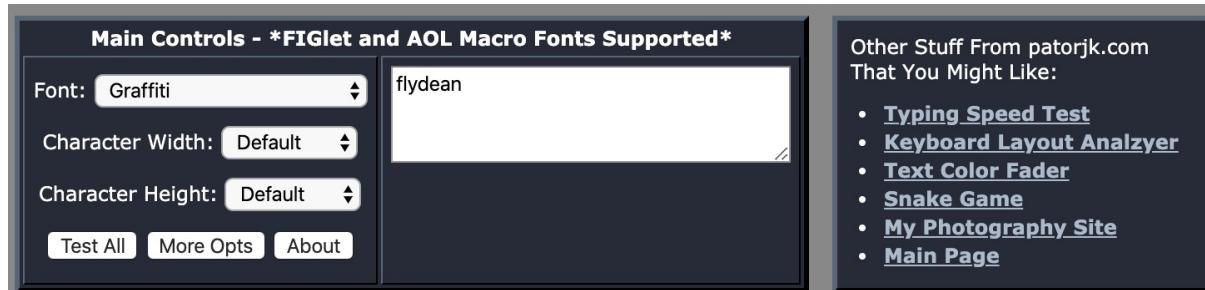
```
kill $(cat ./bin/shutdown.pid)
```

本文的例子可以参考 <https://github.com/ddean2009/learn-springboot2/tree/master/springboot-shutdown>

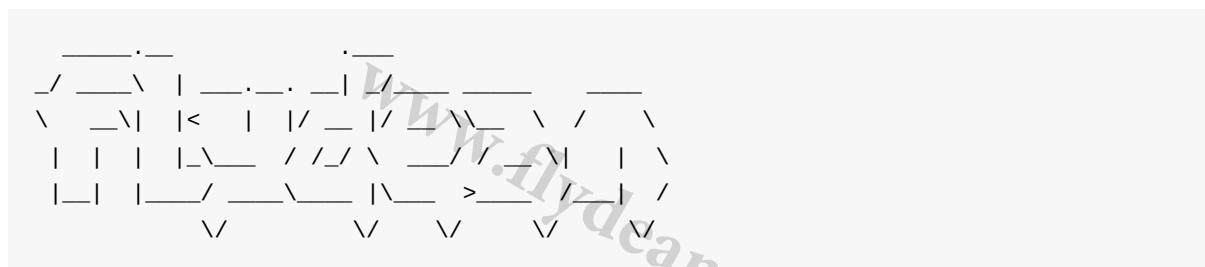
Spring Boot 自定义banner

Spring Boot启动的时候会在命令行生成一个banner，其实这个banner是可以自己修改的，本文将会讲解如何修改这个banner。

首先我们需要将banner保存到一个文件中，网上有很多可以生成banner文件的网站，比如：patorjk.com/software/taag



我们生成了如下的banner：



将其保存为banner.txt，放在 resource目录下。

接下来我们需要指定使用该banner文件，在application.properties文件中定义如下：

```
spring.banner.location=classpath:banner.txt
```

启动看看效果：

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_171.jdk/Contents/Home/bin/java
_____._____
/_ __ \ | < | | / _ | / _ \| \| \ \| | \ |
| | | | | \__ / / / \ \ / / \ \ \ | | \ |
|__| | __/ / __\__ | \__ >__ / / | / / \ \ \
```

除了使用txt文件，我们也可以使用图片如下：

```
spring.banner.image.location=classpath:banner.gif
spring.banner.image.width= //TODO
spring.banner.image.height= //TODO
```

```
spring.banner.image.margin= //TODO  
spring.banner.image.invert= //TODO
```

可以自定义图片的其他一些属性。好了，本文就介绍到这里。

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-customer-banner>

更多教程请参考 [flydean的博客](#)

www.flydean.com

在Spring Boot中自定义filter

本文我们将会讲解如何在Spring Boot中自定义filter并指定执行顺序。

定义Filter很简单，我们只需要实现Filter接口即可，同时我们可指定@Order来确定其执行顺序，我们定义两个filter如下：

```
@Slf4j
@Component
@Order(1)
public class TransactionFilter implements Filter {

    @Override
    public void doFilter(
        ServletRequest request,
        ServletResponse response,
        FilterChain chain) throws IOException, ServletException

    {

        HttpServletRequest req = (HttpServletRequest) request;
        log.info(
            "Starting a transaction for req : {}",
            req.getRequestURI());

        chain.doFilter(request, response);
        log.info(
            "Committing a transaction for req : {}",
            req.getRequestURI());
    }

    // other methods
}
```

```
@Slf4j
@Component
@Order(2)
public class RequestResponseLoggingFilter implements Filter {

    @Override
    public void doFilter(
        ServletRequest request,
        ServletResponse response,
        FilterChain chain) throws IOException, ServletException {

        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse res = (HttpServletResponse) response;
```

```
    log.info(
        "Logging Request  {} : {}", req.getMethod(),
        req.getRequestURI());
    chain.doFilter(request, response);
    log.info(
        "Logging Response :{}",
        res.getContentType());
}

// other methods
}
```

注意在Spring Boot中我们需要使用@Component来实例化Filter从而在Spring Boot中生效。

@Order指定了两个filter的顺序。

上面的例子我们指定了两个filter对于所有的url生效，如果我们希望filter对于特定的某些url生效该怎么办呢？

我们可使用FilterRegistrationBean来手动注册对于的Filter：

```
@Bean
public FilterRegistrationBean<UrlFilter> loggingFilter(){
    FilterRegistrationBean<UrlFilter> registrationBean
        = new FilterRegistrationBean<>();

    registrationBean.setFilter(new UrlFilter());
    registrationBean.addUrlPatterns("/users/*");

    return registrationBean;
}
```

上面我们同时指定了filter对应的urlPattern。

本文的例子可以参考 <https://github.com/ddean2009/learn-springboot2/tree/master/springboot-filter>

更多教程请参考 [flydean的博客](#)

Spring Boot中使用Swagger CodeGen生成REST client

Swagger是一个非常好用的API工具，我们会使用Swagger来暴露API给外界测试，那么有没有简单的办法来生成对应的调client呢？

Swagger CodeGen是一个REST 客户端生成工具，它可以从Open API的规范定义文件中生成对应的 REST Client代码。本文我们将会举例说明如何通过OpenAPI 规范定义文件自动生成REST Client。

什么是Open API规范定义文件呢？

OpenAPI规范（OAS）为RESTful API定义了一个与语言无关的标准接口，使人类和计算机都可以发现和理解服务的功能，而无需访问源代码，文档或通过网络流量检查。正确定义后，使用者可以使用最少的实现逻辑来理解远程服务并与之交互。

然后，文档生成工具可以使用OpenAPI定义来显示API，代码生成工具可以使用各种编程语言，测试工具和许多其他用例来生成服务器和客户端。

值得一提的是OpenAPI规范最早也是Swagger提出来的，后面被捐赠给了社区。

推荐的OpenAPI 文档名字通常为openapi.json 或者 openapi.yaml。

我们看一个swagger自带的 petstore open api 例子：<https://petstore.swagger.io/v2/swagger.json>

```
{
  "swagger": "2.0",
  "info": {
    "description": "This is a sample server Petstore server. You can find out more about Swagger at [http://swagger.io](http://swagger.io) or on [irc.freenode.net, # swagger](http://swagger.io/irc/). For this sample, you can use the api key `specia l-key` to test the authorization filters.",
    "version": "1.0.3",
    "title": "Swagger Petstore",
    "termsOfService": "http://swagger.io/terms/",
    "contact": {
      "email": "apiteam@swagger.io"
    },
    "license": {
      "name": "Apache 2.0",
      "url": "http://www.apache.org/licenses/LICENSE-2.0.html"
    }
  },
  "host": "petstore.swagger.io",
  "basePath": "/v2",
  "tags": [
    ...
  ],
  "schemes": [
    "https",
    "http"
  ]
}
```

```

    "http"
],
"paths": {
  ...
  "definitions": {
    ...
  },
  "externalDocs": {
    "description": "Find out more about Swagger",
    "url": "http://swagger.io"
  }
}
}

```

我们可以看到在这个open API 定义文件里面包含了我们在swagger界面上看到的一切， paths, definitions等。

生成Rest Client

有了Open Api定义文件之后，我们就可以使用 swagger-codegen-cli 来生成对应的rest client文件了。

目前为止，最新的swagger-codegen-cli版本是2.4.12， 我们可以从这里下载

<https://search.maven.org/classic/remotecontent?filepath=io/swagger/swagger-codegen-cli/2.4.12/swagger-codegen-cli-2.4.12.jar>。

下载到本地之后，我们可以通过如下命令来生成rest client:

```

java -jar swagger-codegen-cli-2.4.12.jar generate \
-i http://petstore.swagger.io/v2/swagger.json \
--api-package com.flydean.client.api \
--model-package com.flydean.client.model \
--invoker-package com.flydean.client.invoker \
--group-id com.flydean \
--artifact-id springboot-generate-restclient \
--artifact-version 0.0.1-SNAPSHOT \
-l java \
--library resttemplate \
-o springboot-generate-restclient

```

上述的参数包含：

- -i 指定了open api 定义文件的地址
- --api-package, --model-package, --invoker-package 指定了生成文件的package
- --group-id, --artifact-id, --artifact-version 指定生成的maven 项目的属性
- -l 指明生成的代码编程语言
- --library 指定了实际的实现框架
- -o 指定输出文件目录

Swagger Codegen 支持如下的Java 库:

- jersey1 – Jersey1 + Jackson
- jersey2 – Jersey2 + Jackson
- feign – OpenFeign + Jackson
- okhttp-gson – OkHttp + Gson
- retrofit (Obsolete) – Retrofit1/OkHttp + Gson
- retrofit2 – Retrofit2/OkHttp + Gson
- rest-template – Spring RestTemplate + Jackson
- rest-easy – Resteasy + Jackson

在Spring Boot中使用

我们把生成的代码拷贝到我们的Spring Boot项目中。然后通过下面的代码来启动应用程序:

```
@SpringBootApplication
public class GenerateClientApp {

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

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }

}
```

我们再定义一个controller:

```
@RestController
public class PetController {

    @Autowired
    private PetApi petApi;

    @GetMapping("/api/findAvailablePets")
    public List<Pet> findAvailablePets() {
        return petApi.findPetsByStatus(Arrays.asList("available"));
    }
}
```

现在通过curl localhost:8080/api/findAvailablePets就可以远程调用<http://petstore.swagger.io/v2/swagger.json> 里面暴露的接口了。

API Client 配置

默认情况下ApiClient是默认的不需要认证的，如果需要认证，可以自定义ApiClient如下：

```
@Bean
public ApiClient apiClient() {
    ApiClient apiClient = new ApiClient();

    OAuth petStoreAuth = (OAuth) apiClient.getAuthentication("petstore_auth");
    petStoreAuth.setAccessToken("special-key");

    return apiClient;
}
```

使用Maven plugin

除了使用cli命令之外，我们还可以在pom中添加plugin来实现这个功能：

```
<build>
    <plugins>
        <plugin>
            <groupId>io.swagger</groupId>
            <artifactId>swagger-codegen-maven-plugin</artifactId>
            <version>2.4.12</version>
            <executions>
                <execution>
                    <goals>
                        <goal>generate</goal>
                    </goals>
                    <configuration>
                        <inputSpec>swagger.json</inputSpec>
                        <language>java</language>
                        <library>resttemplate</library>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

在线生成API

我们可以通过<http://generator.swagger.io>来在线生成API代码：

```
curl -X POST -H "content-type:application/json" \
-d '{"swaggerUrl":"http://petstore.swagger.io/v2/swagger.json"}' \
http://generator.swagger.io/api/gen/clients/java
```

该命令会返回一个包含代码的zip包供你下载。

本文的例子可以参考 <https://github.com/ddean2009/learn-springboot2/tree/master/springboot-generate-restclient>

更多教程请参考 [flydean的博客](#)

www.flydean.com

Spring Boot中使用@JsonComponent

@JsonComponent 是Spring boot的核心注解，使用@JsonComponent 之后就不需要手动将Jackson的序列化和反序列化手动加入ObjectMapper了。使用这个注解就够了。

序列化

假如我们有个User类，它里面有一个Color属性：

```
@Data
@AllArgsConstructor
public class User {
    private Color favoriteColor;
}
```

接下来我们来创建针对User的序列化组件，我们需要实现JsonSerializer接口：

```
@JsonComponent
public class UserJsonSerializer extends JsonSerializer<User> {

    @Override
    public void serialize(User user, JsonGenerator jsonGenerator,
                          SerializerProvider serializerProvider) throws IOException

        ,
        JsonProcessingException {

        jsonGenerator.writeStartObject();
        jsonGenerator.writeStringField(
            "favoriteColor",
            getColorAsWebColor(user.getFavoriteColor()));
        jsonGenerator.writeEndObject();
    }

    private static String getColorAsWebColor(Color color) {
        int r = (int) Math.round(color.getRed() * 255.0);
        int g = (int) Math.round(color.getGreen() * 255.0);
        int b = (int) Math.round(color.getBlue() * 255.0);
        return String.format("#%02x%02x%02x", r, g, b);
    }
}
```

在上面的类中，我们自定义了序列化的方法。接下来我们测试一下：

```
@JsonTest
@RunWith(SpringRunner.class)
public class UserJsonSerializerTest {
```

```

    @Autowired
    private ObjectMapper objectMapper;

    @Test
    public void testSerialization() throws JsonProcessingException {
        User user = new User(Color.ALICEBLUE);
        String json = objectMapper.writeValueAsString(user);

        assertEquals("{\"favoriteColor\":\"#f0f8ff\"}", json);
    }
}

```

反序列化

同样的，我们实现JsonDeserializer接口：

```

@JsonComponent
public class UserJsonDeserializer extends JsonDeserializer<User> {

    @Override
    public User deserialize(JsonParser jsonParser,
                           DeserializationContext deserializationContext) throws IOException,
    JsonProcessingException {
        TreeNode treeNode = jsonParser.getCodec().readTree(jsonParser);
        TextNode favoriteColor
            = (TextNode) treeNode.get("favoriteColor");
        return new User(Color.web(favoriteColor.asText()));
    }
}

```

我们看下怎么调用：

```

@JsonTest
@RunWith(SpringRunner.class)
public class UserJsonDeserializerTest {

    @Autowired
    private ObjectMapper objectMapper;

    @Test
    public void testDeserialize() throws IOException {
        String json = "{\"favoriteColor\":\"#f0f8ff\"}"
        User user = objectMapper.readValue(json, User.class);

        assertEquals(Color.ALICEBLUE, user.getFavoriteColor());
    }
}

```

```
    }
}
```

在同一个class中序列化和反序列化

```
@JsonComponent
public class UserCombinedSerializer {

    public static class UserJsonSerializer
        extends JsonSerializer<User> {

        @Override
        public void serialize(User user, JsonGenerator jsonGenerator,
            SerializerProvider serializerProvider) throws IOException,
            JsonProcessingException {

            jsonGenerator.writeStartObject();
            jsonGenerator.writeStringField(
                "favoriteColor", getColorAsWebColor(user.getFavoriteColor()));
            jsonGenerator.writeEndObject();
        }

        private static String getColorAsWebColor(Color color) {
            int r = (int) Math.round(color.getRed() * 255.0);
            int g = (int) Math.round(color.getGreen() * 255.0);
            int b = (int) Math.round(color.getBlue() * 255.0);
            return String.format("#%02x%02x%02x", r, g, b);
        }
    }

    public static class UserJsonDeserializer
        extends JsonDeserializer<User> {

        @Override
        public User deserialize(JsonParser jsonParser,
            DeserializationContext deserializationContext)
            throws IOException, JsonProcessingException {

            TreeNode treeNode = jsonParser.getCodec().readTree(jsonParser);
            TextNode favoriteColor = (TextNode) treeNode.get(
                "favoriteColor");
            return new User(Color.web(favoriteColor.asText()));
        }
    }
}
```

为了方便，我们还可以在同一个类中定义两个内部类来实现序列化和反序列化。如上所示。

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-jsoncomponent>

更多教程请参考 [flydean的博客](#)

www.flydean.com

Spring Boot国际化支持

国际化支持应该是所有的做国际化网站都需要考虑的一个问题，Spring Boot为国际化提供了强有力的支持，本文将会通过一个例子来讲解Spring Boot的国际化。

添加Maven支持

Spring Boot本身就支持国际化，我们这里添加一个模板支持来通过页面来展示，我们这里添加thymeleaf模板：

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

LocaleResolver

我们需要为系统指定一个默认的LocaleResolver：

```
@Bean
public LocaleResolver localeResolver() {
    SessionLocaleResolver slr = new SessionLocaleResolver();
    slr.setDefaultLocale(Locale.US);
    return slr;
}
```

上面的例子中我们自定义了一个SessionLocaleResolver，并且指定了默认的Locale。

LocaleChangeInterceptor

接下来，我们定义一个LocaleChangeInterceptor来接收Locale的变动。这里我们通过lang参数来接收。

```
@Bean
public LocaleChangeInterceptor localeChangeInterceptor() {
    LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
    lci.setParamName("lang");
    return lci;
}
```

当然，我们需要将这个Interceptor注册到SpringMVC中：

```
@Override
```

```
public void addInterceptors(InterceptorRegistry registry) {  
    registry.addInterceptor(localeChangeInterceptor());  
}
```

定义Message Sources

默认情况下，Spring Boot会在src/main/resources查找message文件，默认的message文件是messages.properties,如果指定了某种语言，那么就是messages_XX.properties，其中XX是Local code。

messages.properties是key value的格式，如果在对应的local文件中没找到相应的key，则会在默认的messages.properties中查找。

我们默认定义英语的messages.properties如下：

```
greeting=Hello! Welcome to our website!  
lang.change=Change the language  
lang.eng=English  
lang.fr=French
```

同时我们定义一个法语的message文件messages_fr.properties：

```
greeting=Bonjour! Bienvenue sur notre site!  
lang.change=Changez la langue  
lang.eng=Anglais  
lang.fr=Francais
```

Controller文件

我们定义一个跳转的controller文件：

```
@Controller  
public class PageController {  
  
    @GetMapping("/international")  
    public String getInternationalPage() {  
        return "international";  
    }  
}
```

html文件

相应的html文件如下：

```
<!DOCTYPE html>
```

```
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <meta charset="ISO-8859-1" />
    <title>Home</title>
    <script src="https://ajax.googleapis.com/ajax/libs/jquery/3.1.1/jquery.min.js">
</script>
    <script>
        $(document).ready(function() {
            $("#locales").change(function () {
                var selectedOption = $('#locales').val();
                if (selectedOption != ''){
                    window.location.replace('international?lang=' + selectedOption)
                }
            });
        });
    </script>
</head>
<body>
<h1 th:text="#{greeting}"></h1>

<br /><br />
<span th:text="#{lang.change}"></span>:
<select id="locales">
    <option value=""></option>
    <option value="en" th:text="#{lang.eng}"></option>
    <option value="fr" th:text="#{lang.fr}"></option>
</select>
</body>
</html>
```

运行应用程序

好了，接下来我们可以运行了。

如果我们访问<http://localhost:8080/international?lang=en>，则会读取默认的英语资源：

Hello! Welcome to our website!

Change the language:

通过切换到法语环境：<http://localhost:8080/international?lang=fr>，我们可以看到：

Bonjour! Bienvenue sur notre site!

Changez la langue:

环境已经切换过来了。

本文的例子可以参考: <https://github.com/ddean2009/learn-springboot2/tree/master/springboot-Internationalization>

更多教程请参考 [flydean的博客](#)

Spring Boot devtool的使用

Spring Boot为我们提供了一个便捷的开发Spring Boot应用程序的环境，同时为了方便我们的开发Spring Boot应用程序，Spring Boot 推出了Spring Boot devtool的工具来方便我们更加快速的开发和测试Spring Boot应用程序。

我们将会从下面几个方面来详细讲解Spring Boot devtool的功能。

添加Spring Boot devtool依赖

添加Spring Boot devtool依赖很简单：

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

这样就添加好了，我们可以开始使用Spring boot devtool带给我们的优秀功能了。

默认属性

Spring Boot为我们提供了很多自动配置来提高我们开发的效率，比如会缓存模板引擎例如thymeleaf，但是如果我们在开发过程中可能需要快速的看到修改的结果，这个时候我们就需要这个缓存配置了，这时候我们就需要配置：

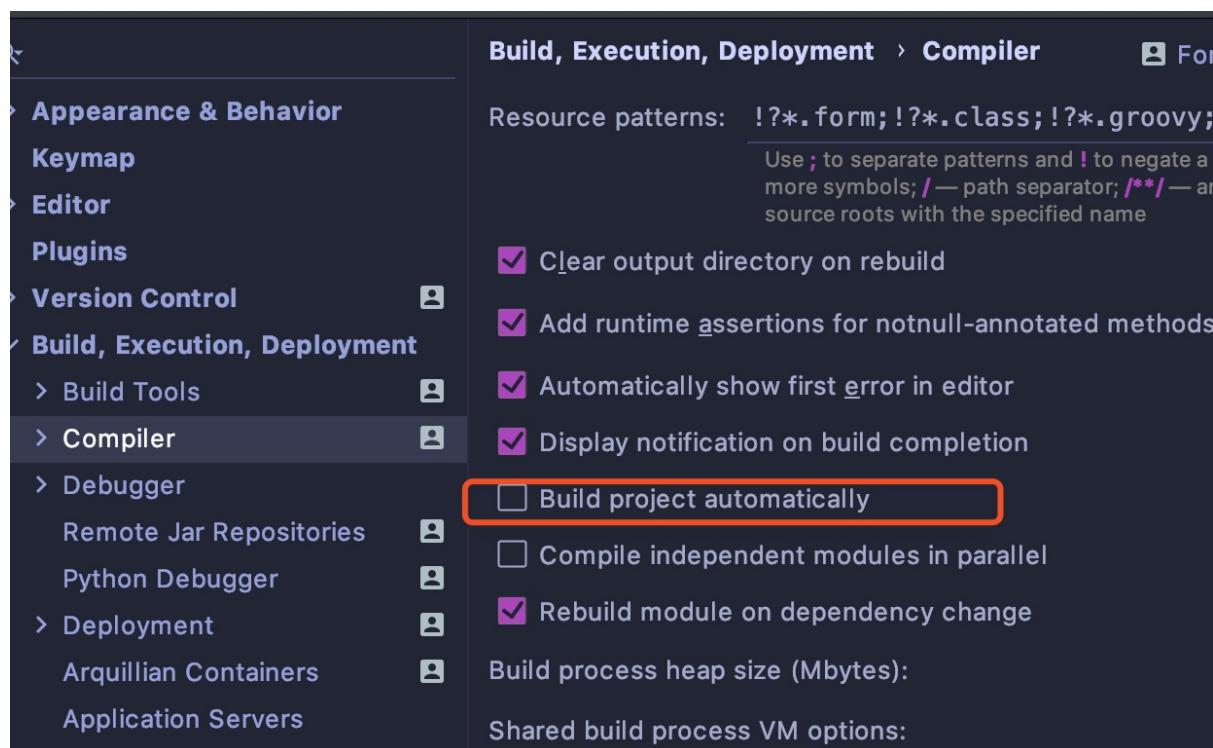
```
spring.thymeleaf.cache=false
```

如果添加了spring-boot-devtools，上述的配置就不需要手动添加，devtool会自动帮我们添加好。

自动重启

在开发过程中，如果我们修改了某些java文件，我们可能需要重启下项目来观看修改后的结果，如果使用spring-boot-devtools，当classpath中有文件变动时候，devtools会自动帮你重启服务器。

注意，这里的重启的条件是classpath的文件要有变化，如果你在使用IDEA开发的话，请勾选“Build project automatically”选项，如下图示所示，否则你需要重新build项目来使重启生效。



Live Reload

Live Reload主要针对资源文件的，我们的APP启动之后，可以看到一个：

```
o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on port 357
29
```

当资源文件变动的时候，方便前端刷新。

如果要用到这个live reload的功能，需要在chrome浏览器中安装一个Remote Live Reload 的插件。

Home > Extensions > RemoteLiveReload



RemoteLiveReload

Offered by: remoteliveread

★★★★★ 32 | [Developer Tools](#) | [6,051 users](#)

[Add to Chrome](#)

这个插件主要是通过引入的脚本livereload.js在 livereload 服务和浏览器之间建立了一个 WebSocket 连接。每当监测到文件的变动，livereload 服务就会向浏览器发送一个信号，浏览器收到信号后就刷新页面，实现了实时刷新的效果。

全局配置

spring-boot-devtools 提供了一个全局配置文件，方便你的开发环境配置，该文件在\$HOME 目录下面的 .spring-boot-devtools.properties 。

本文的例子可以参考 <https://github.com/ddean2009/learn-springboot2/tree/master/springboot-devtool>

更多教程请参考 [flydean的博客](#)

www.flydean.com

Spring Boot Admin的使用

前面的文章我们讲了Spring Boot的Actuator。但是Spring Boot Actuator只是提供了一个个的接口，需要我们自行集成到监控程序中。今天我们将会讲解一个优秀的监控工具Spring Boot Admin。它采用图形化的界面，让我们的Spring Boot管理更加简单。

先上图给大家看一下Spring Boot Admin的界面：

The screenshot shows the Spring Boot Admin web interface. On the left is a sidebar with navigation links: Insights, Details, Performance, Environment, Classes, Configuration Properties, Scheduled Tasks, Log Configuration, JVM, Mappings, and Caches. The main area has a title 'spring-boot-application' with an ID 'a973ff14be49'. Below it are three links: 'http://localhost:8080/' (status), 'http://localhost:8080/actuator' (metrics), and 'http://localhost:8080/actuator/health' (health). The 'Details' tab is selected, showing a table with three rows: 'user.name' (admin), 'user.password' (*****), and 'startup' (2020-02-12T23:34:41.436+08:00). To the right is the 'Health' section, which lists various components and their status: Instance (UP), db (UP), database (H2), result (1), validationQuery (SELECT 1), diskSpace (UP), total (251 GB), free (7.88 GB), and threshold (10.5 MB).

从界面上面我们可以看到Spring Boot Admin提供了众多强大的监控功能。那么开始我们的学习吧。

配置Admin Server

既然是管理程序，肯定有一个server，配置server很简单，我们添加这个依赖即可：

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-server</artifactId>
    <version>2.2.2</version>
</dependency>
```

同时我们需要在main程序中添加@EnableAdminServer来启动admin server。

```
@EnableAdminServer
@SpringBootApplication
public class SpringBootAdminServerApplication {

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

```
}
```

配置admin client

有了server，我们接下来配置需要监控的client应用程序，在本文中，我们自己监控自己，添加client依赖如下：

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-starter-client</artifactId>
    <version>2.2.2</version>
</dependency>
```

我们需要为client指定要注册到的admin server：

```
spring.boot.admin.client.url=http://localhost:8080
```

因为Spring Boot Admin依赖于 Spring Boot Actuator, 从Spring Boot2 之后，我们需要主动开启暴露的主键，如下：

```
management.endpoints.web.exposure.include=*
management.endpoint.health.show-details=always
```

配置安全主键

通常来说，我们需要一个登陆界面，以防止未经授权的人访问。spring boot admin提供了一个UI供我们使用，同时我们添加Spring Security依赖：

```
<dependency>
    <groupId>de.codecentric</groupId>
    <artifactId>spring-boot-admin-server-ui-login</artifactId>
    <version>1.5.7</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

添加了Spring Security，我们需要自定义一些配置：

```
@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    private final AdminServerProperties adminServer;

    public WebSecurityConfig(AdminServerProperties adminServer) {
```

```
this.adminServer = adminServer;
}

@Override
protected void configure(HttpSecurity http) throws Exception {
    SavedRequestAwareAuthenticationSuccessHandler successHandler =
        new SavedRequestAwareAuthenticationSuccessHandler();
    successHandler.setTargetUrlParameter("redirectTo");
    successHandler.setDefaultTargetUrl(this.adminServer.getContextPath() + "/");
;

    http
        .authorizeRequests()
            .antMatchers(this.adminServer.getContextPath() + "/assets/**").permitAll()
            .antMatchers(this.adminServer.getContextPath() + "/login").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin()
            .loginPage(this.adminServer.getContextPath() + "/login")
            .successHandler(successHandler)
            .and()
        .logout()
            .logoutUrl(this.adminServer.getContextPath() + "/logout")
            .and()
        .httpBasic()
            .and()
        .csrf()
            .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
            .ignoringRequestMatchers(
                new AntPathRequestMatcher(this.adminServer.getContextPath() +
                    "/instances", HttpMethod.POST.toString()),
                new AntPathRequestMatcher(this.adminServer.getContextPath() +
                    "/instances/*", HttpMethod.DELETE.toString()),
                new AntPathRequestMatcher(this.adminServer.getContextPath() + "/actuator/**"))
            .and()
        .rememberMe()
            .key(UUID.randomUUID().toString())
            .tokenValiditySeconds(1209600);
    }
}
```

接下来，我们在配置文件中指定服务器的用户名和密码：

```
spring.boot.admin.client.username=admin
spring.boot.admin.client.password=admin
```

作为一个客户端，连接服务器的时候，我们也要提供相应的认证信息如下：

```
spring.boot.admin.client.instance.metadata.user.name=admin  
spring.boot.admin.client.instance.metadata.user.password=admin  
  
spring.boot.admin.client.username=admin  
spring.boot.admin.client.password=admin
```

好了，登录页面和权限认证也完成了。

Hazelcast集群

Spring Boot Admin 支持Hazelcast的集群，我们先添加依赖如下：

```
<dependency>  
    <groupId>com.hazelcast</groupId>  
    <artifactId>hazelcast</artifactId>  
    <version>3.12.2</version>  
</dependency>
```

然后添加Hazelcast的配置：

```
@Configuration  
public class HazelcastConfig {  
  
    @Bean  
    public Config hazelcast() {  
        MapConfig eventStoreMap = new MapConfig("spring-boot-admin-event-store")  
            .setInMemoryFormat(InMemoryFormat.OBJECT)  
            .setBackupCount(1)  
            .setEvictionPolicy(EvictionPolicy.NONE)  
            .setMergePolicyConfig(new MergePolicyConfig(PutIfAbsentMapMergePolicy.class.getName(), 100));  
  
        MapConfig sentNotificationsMap = new MapConfig("spring-boot-admin-application-store")  
            .setInMemoryFormat(InMemoryFormat.OBJECT)  
            .setBackupCount(1)  
            .setEvictionPolicy(EvictionPolicy.LRU)  
            .setMergePolicyConfig(new MergePolicyConfig(PutIfAbsentMapMergePolicy.class.getName(), 100));  
  
        Config config = new Config();  
        config.addMapConfig(eventStoreMap);  
        config.addMapConfig(sentNotificationsMap);  
        config.setProperty("hazelcast.jmx", "true");  
  
        config.getNetworkConfig()
```

```
.getJoin()
.getMulticastConfig()
.setEnabled(false);
TcpIpConfig tcpIpConfig = config.getNetworkConfig()
.getJoin()
.getTcpIpConfig();
tcpIpConfig.setEnabled(true);
tcpIpConfig.setMembers(Collections.singletonList("127.0.0.1"));
return config;
}
}
```

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-admin>

将Spring Boot应用程序注册成为系统服务

在之前的文章中，我们提到了很多Spring Boot的技巧，那么当我们创建好了Spring Boot应用程序之后，怎么在生成环境中运行呢？如果只是以原始的java -jar的方式来运行的话，不能保证程序的健壮性和稳定性，最好的办法是将程序注册成为服务来使用。

本文将会讲解如何将Spring Boot应用程序注册成为Linux和windows的服务。

前期准备

首先我们需要将应用程序打包成为一个可执行的jar包，我们需要添加如下依赖：

```
<packaging>jar</packaging>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
</parent>

<dependencies>
    ...
</dependencies>

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

这里的packaging我们需要选择jar。添加spring-boot-maven-plugin是为了将app打包成为可执行的jar包。

打包成可执行jar包

写好了应用程序，我们可以执行：

```
mvn clean package
```

来打包应用程序，这里我们打包之后的jar包名字为：springboot-run-as-service-0.0.1-SNAPSHOT.jar。

注册成为linux服务

在linux中，我们可以选择System V init script或者Systemd 配置文件，前者逐渐在被后者替代。

为了安全起见，我们需要创建一个运行用户，并给jar包赋予相应的权限：

```
$ sudo useradd flydean  
$ sudo passwd flydean  
$ sudo chown flydean:flydean your-app.jar  
$ sudo chmod 500 your-app.jar
```

System V Init

创建一个文件链接到init.d目录，如下：

```
sudo ln -s /path/to/your-app.jar /etc/init.d/your-app
```

接下来我们就可以启动应用程序了：

```
sudo service your-app start
```

service命令支持start, stop, restart 和 status。同时它还提供了如下的功能：

- your-app 将会以flydean用户启动
- 程序运行的pid存储在/var/run/your-app/your-app.pid
- 应用程序的日志在/var/log/your-app.log

Systemd

使用Systemd，我们需要在 /etc/systemd/system 创建一个your-app.service文件：

```
[Unit]  
Description=A Spring Boot application  
After=syslog.target  
  
[Service]  
User=flydean  
ExecStart=/path/to/your-app.jar SuccessExitStatus=100  
  
[Install]  
WantedBy=multi-user.target
```

接下来我们可以使用systemctl start|stop|restart|status your-app来管理你的服务了。

Upstart

Upstart是一个事件驱动的服务管理器，如果你使用Ubuntu，将会被默认安装。

我们来创建一个your-app.conf：

```
# Place in /home/{user}/.config/upstart

description "Some Spring Boot application"

respawn # attempt service restart if stops abruptly

exec java -jar /path/to/your-app.jar
```

在Windows中安装

在windows中，我们也有很多方式，如下：

Windows Service Wrapper

Windows Service Wrapper 又叫 winsw是一个开源软件，winsw需要和一个配置文件your-app.xml配合使用：

```
<service>
  <id>MyApp</id>
  <name>MyApp</name>
  <description>This runs Spring Boot as a Service.</description>
  <env name="MYAPP_HOME" value="%BASE%" />
  <executable>java</executable>
  <arguments>-Xmx256m -jar "%BASE%\your-app.jar"</arguments>
  <logmode>rotate</logmode>
</service>
```

注意，你需要修改winsw.exe成为your-app.exe来和your-app.xml配合使用。

Java Service Wrapper

Java Service Wrapper 提供了非常强大的配置，他可以让你的应用程序在windows和Linux下面使用。有兴趣的同学可以自行去学习。

Spring Boot 之Spring data JPA简介

JPA的全称是Java Persistence API (JPA)，他是一个存储API的标准，而Spring data JPA就是对JPA的一种实现，可以让我们方便的对数据进行存取。按照约定好的方法命名规则写dao层接口，从而在不实现接口的情况下，实现对数据库的访问和操作。同时提供了很多除了CRUD之外的功能，如分页、排序、复杂查询等等。

Spring data JPA可以看做是对Hibernate的二次封装。本文将会以一个具体的例子来讲解，怎么在Spring Boot中使用Spring data JPA。

添加依赖

我们要添加如下的Spring data JPA依赖，为了方便测试，我们添加一个h2的内存数据库：

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

添加entity bean

我们来创建一个entity bean：

```
@Entity
@Data
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long id;

    @Column(nullable = false, unique = true)
    private String title;

    @Column(nullable = false)
    private String author;
}
```

创建 Dao

```
public interface BookRepository extends JpaRepository<Book, Long> {  
    List<Book> findByTitle(String title);  
  
    @Query("SELECT b FROM Book b WHERE LOWER(b.title) = LOWER(:title)")  
    Book retrieveByTitle(@Param("title") String title);  
}
```

所有的Dao都需要继承Repository接口， Repository是一个空的接口：

```
@Indexed  
public interface Repository<T, ID> {  
}
```

如果要使用默认的通用的一些实现，则可以继承CrudRepository、 PagingAndSortingRepository和 JpaRepository。

上面的例子中我们继承了JpaRepository。

上面的例子中我们创建了一个按Title查找的方法：

```
List<Book> findByTitle(String title);
```

这个方法我们是不需要自己去实现的， Spring Data JPA会帮我们去实现。我们可以使用find...By, read...By, query...By, count...By, 和 get...By的格式定义查询语句， By后面接的就是Entity的属性。除了And，我们还可以使用Or来拼接方法，下面我们再举个例子：

```
interface PersonRepository extends Repository<Person, Long> {  
  
    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);  
  
    // Enabling ignoring case for an individual property  
    List<Person> findByLastnameIgnoreCase(String lastname);  
    // Enabling ignoring case for all suitable properties  
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
  
    // Enabling static ORDER BY for a query
```

```
List<Person> findByLastnameOrderByFirstnameAsc(String lastname);
List<Person> findByLastnameOrderByFirstnameDesc(String lastname);
}
```

当然，处理方法拼接外，我们还可以自定义sql查询语句：

```
@Query("SELECT b FROM Book b WHERE LOWER(b.title) = LOWER(:title)")
Book retrieveByTitle(@Param("title") String title);
```

自定义查询语句给Spring data JPA提供了更大的想象空间。

Spring Data Configuration

要使用Spring Data JPA, 我们还需要在配置文件中指定要扫描的目录，使用@EnableJpaRepositories注解来实现：

```
@Configuration
@EnableJpaRepositories(basePackages = "com.flydean.repository")
public class PersistenceConfig {
```

我们还需要在配置文件中指定数据源的属性：

```
spring.datasource.url=jdbc:h2:mem:db;DB_CLOSE_DELAY=-1
spring.datasource.username=sa
spring.datasource.password=sa
```

测试

有了上面的一切，我们就可以测试我们的数据源了：

```
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {JpaApp.class})
public class BookRepositoryTest {

    @Autowired
    private BookRepository bookRepository;

    @Test
    @Transactional(readOnly=false)
    public void testBookRepository(){
        Book book = new Book();
        book.setTitle(randomAlphabetic(10));
        book.setAuthor(randomAlphabetic(15));
    }
}
```

```
bookRepository.save(book);

bookRepository.findByTitle(book.getTitle()).forEach(e -> log.info(e.toString()));
log.info(bookRepository.retrieveByTitle(book.getTitle()).toString());
}
```

本文的例子可以参考:<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-jpa>

www.flydean.com

Spring Boot JPA中java 8 的应用

上篇文章中我们讲到了如何在Spring Boot中使用JPA。本文我们将会讲解如何在Spring Boot JPA中使用java 8 中的新特习惯如：Optional, Stream API 和 CompletableFuture的使用。

Optional

我们从数据库中获取的数据有可能是空的，对于这样的情况Java 8 提供了Optional类，用来防止出现空值的情况。我们看下怎么在Repository 中定义一个Optional的方法：

```
public interface BookRepository extends JpaRepository<Book, Long> {  
  
    Optional<Book> findOneByTitle(String title);  
}
```

我们看下测试方法怎么实现：

```
@Test  
public void testFindOneByTitle(){  
  
    Book book = new Book();  
    book.setTitle("title");  
    book.setAuthor(randomAlphabetic(15));  
  
    bookRepository.save(book);  
    log.info(bookRepository.findOneByTitle("title").orElse(new Book()).toString());  
}
```

Stream API

为什么会有Stream API呢？ 我们举个例子，如果我们想要获取数据库中所有的Book，我们可以定义如下的方法：

```
public interface BookRepository extends JpaRepository<Book, Long> {  
  
    List<Book> findAll();  
  
    Stream<Book> findAllByTitle(String title);  
}
```

上面的findAll方法会获取所有的Book，但是当数据库里面的数据太多的话，就会消耗过多的系统内存，甚至有可能导致程序崩溃。

为了解决这个问题，我们可以定义如下的方法：

```
Stream<Book> findAllByTitle(String title);
```

当你使用Stream的时候，记得需要close它。我们可以使用java 8 中的try语句来自动关闭：

```
@Test
@Transactional
public void testfindAll() {

    Book book = new Book();
    book.setTitle("titleAll");
    book.setAuthor(randomAlphabetic(15));
    bookRepository.save(book);

    try (Stream<Book> foundBookStream
         = bookRepository.findAllByTitle("titleAll")) {
        assertThat(foundBookStream.count(), equalTo(11));
    }
}
```

这里要注意， 使用Stream必须要在Transaction中使用。否则会报如下错误：

```
org.springframework.dao.InvalidDataAccessApiUsageException: You're trying to execute a streaming query method without a surrounding transaction that keeps the connection open so that the Stream can actually be consumed. Make sure the code consuming the stream uses @Transactional or any other way of declaring a (read-only) transaction.
```

所以这里我们加上了@Transactional 注解。

CompletableFuture

使用java 8 的CompletableFuture， 我们还可以异步执行查询语句：

```
@Async
CompletableFuture<Book> findOneByAuthor(String author);
```

我们这样使用这个方法：

```
@Test
public void testByAuthor() throws ExecutionException, InterruptedException {
    Book book = new Book();
    book.setTitle("titleA");
    book.setAuthor("author");
    bookRepository.save(book);
```

```
    log.info(bookRepository.findOneByAuthor("author").get().toString());  
}
```

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-jpa>

www.flydean.com

Spring Boot中Spring data注解的使用

Spring data JPA为我们提供了很多有用的注解，方便我们来实现各种复杂的功能。本文我们将会从Spring Data Annotations和Spring Data JPA Annotations两部分来讲解。

Spring Data Annotations

Spring Data Annotations是指这些注解来自于spring-data-commons包里面的。Spring Data不仅可以用于JPA，它还有很多其他的数据提供方，JPA只是其中的一个具体实现。

@Transactional

使用@Transactional可以很简单的将方法配置成为Transactional：

```
@Transactional  
void pay() {}
```

@Transactional可以放在方法上，也可以放在class上面，如果放在class上面则说明该class中的所有方法都适用于Transactional。

@NoRepositoryBean

有时候我们在创建父Repository的时候，我们不需要为该父Repository创建一个具体的实现，我们只是想为子Repository提供一个公共的方法而已，这时候，我们就可以在父类上面加入@NoRepositoryBean注解：

```
@NoRepositoryBean  
public interface ParentRepository<T, ID extends Serializable> extends CrudRepository<T, ID> {  
  
    Optional<T> findById(ID id);  
}
```

子类如下：

```
@Repository  
public interface ChildRepository extends ParentRepository<Person, Long> {  
}
```

@Param

我们可以通过使用@Param从而在Query语句中传递参数：

```
@Query("FROM Person p WHERE p.name = :name")
Person findByName(@Param("name") String name);
```

@Id

@Id表示Entity的primary key:

```
class Person {

    @Id
    Long id;

    // ...

}
```

@Transient

通过使用@Transient， 表明Entity的某个字段是不需要被存储的。

```
class Person {

    // ...

    @Transient
    int age;

    // ...

}
```

@CreatedBy, @LastModifiedBy, @CreatedDate, @LastModifiedDate

通过这些注解，我们可以从principals中获得相应的数据：

```
public class Person {

    // ...

    @CreatedBy
    User creator;

    @LastModifiedBy
    User modifier;
```

```
@CreatedDate  
Date createdAt;  
  
@LastModifiedDate  
Date modifiedAt;  
  
// ...  
}
```

因为需要使用到principals，所有这些注解是和Spring Security配合使用的。

Spring Data JPA Annotations

Spring Data JPA Annotations是来自于spring-data-jpa包的。

@Query

通过使用@Query，我们可以自定义SQL语句：

```
@Query("SELECT COUNT(*) FROM Person p")  
long getPersonCount();
```

我们也可以传递参数：

```
@Query("FROM Person p WHERE p.name = :name")  
Person findByName(@Param("name") String name);
```

我们还可以使用native SQL查询：

```
@Query(value = "SELECT AVG(p.age) FROM person p", nativeQuery = true)  
int getAverageAge();
```

@Procedure

通过@Procedure，我们可以调用数据库中的存储过程：

```
@NamedStoredProcedureQueries({  
    @NamedStoredProcedureQuery(  
        name = "count_by_name",  
        procedureName = "person.count_by_name",  
        parameters = {  
            @StoredProcedureParameter(  
                mode = ParameterMode.IN,  
                name = "name",  
                type = String.class),
```

```

        @StoredProcedureParameter(
            mode = ParameterMode.OUT,
            name = "count",
            type = Long.class)
    }
}

class Person {}

```

我们可以在Entity上面添加该注解。然后看下怎么调用：

```

@Procedure(name = "count_by_name")
long getCountByName(@Param("name") String name);

```

@Lock

通过使用@Lock，我们可以选择数据库的隔离方式：

```

@Lock(LockModeType.NONE)
@Query("SELECT COUNT(*) FROM Person p")
long getPersonCount();

```

Lock的值可以有如下几种：

- READ
- WRITE
- OPTIMISTIC
- OPTIMISTIC_FORCE_INCREMENT
- PESSIMISTIC_READ
- PESSIMISTIC_WRITE
- PESSIMISTIC_FORCE_INCREMENT
- NONE

@Modifying

@Modifying表示我们有修改数据库的操作：

```

@Modifying
@Query("UPDATE Person p SET p.name = :name WHERE p.id = :id")
void changeName(@Param("id") long id, @Param("name") String name);

```

@EnableJpaRepositories

通过使用@EnableJpaRepositories，我们来配置Jpa Repository的相关信息：

```
@Configuration  
@EnableJpaRepositories(basePackages = "com.flydean.repository")  
public class PersistenceConfig {  
}
```

默认情况下，我们会在@Configuration类的子类中查找repositories，通过使用basePackages，我们可以指定其他的目录。

本文的例子可以参考:<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-jpa>

www.flydean.com

在Spring Boot使用H2内存数据库

在之前的文章中我们有提到在Spring Boot中使用H2内存数据库方便开发和测试。本文我们将会提供一些更加具体有用的信息来方便我们使用H2数据库。

添加依赖配置

要想使用H2，我们需要添加如下配置：

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

数据库配置

有了上面的依赖，默认情况下Spring Boot会为我们自动创建内存H2数据库，方便我们使用，当然我们也可以使用自己的配置，我们将配置写入application.properties：

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

默认情况下内存数据库会在程序结束之后被销毁，如果我们想永久保存内存数据库需要添加如下配置：

```
spring.datasource.url=jdbc:h2:file:/data/demo
```

这里配置的是数据库的文件存储地址。

添加初始数据

我们可以在resources文件中添加data.sql文件，用来在程序启动时，创建所需的数据库：

```
DROP TABLE IF EXISTS billionaires;
```

```
CREATE TABLE billionaires (
    id INT AUTO_INCREMENT PRIMARY KEY,
    first_name VARCHAR(250) NOT NULL,
    last_name VARCHAR(250) NOT NULL,
    career VARCHAR(250) DEFAULT NULL
);

INSERT INTO billionaires (first_name, last_name, career) VALUES
('Aliko', 'Dangote', 'Billionaire Industrialist'),
('Bill', 'Gates', 'Billionaire Tech Entrepreneur'),
('Folrunsho', 'Alakija', 'Billionaire Oil Magnate');
```

Spring Boot在启动时候会自动加载data.sql文件。这种方式非常方便我们用来测试。

访问H2数据库

虽然是一个内存数据库，我们也可以在外部访问和管理H2，H2提供了一个内嵌的GUI管理程序，我们看下怎么使用。首先需要添加如下权限：

```
spring.h2.console.enabled=true
```

启动程序，我们访问 <http://localhost:8080/h2-console>，得到如下界面：

English ▾ Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded) ▾

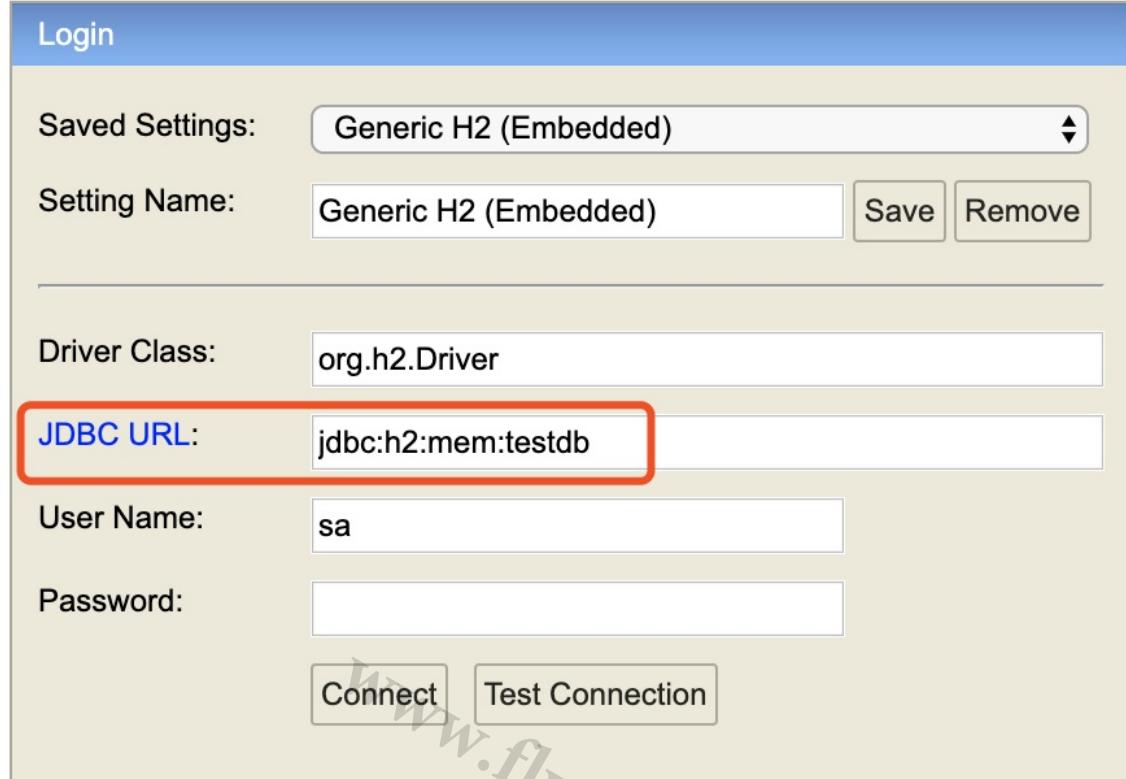
Setting Name: Generic H2 (Embedded)

Driver Class: org.h2.Driver

JDBC URL: **jdbc:h2:mem:testdb**

User Name: sa

Password:



记得填入你在配置文件中配置的地址和密码。

登录之后，我们可以看到如下的管理界面：

The screenshot shows the H2 Database Console interface. On the left, there's a tree view of the database schema:

- jdbc:h2:mem:testdb
- BILLIONAIRES
- INFORMATION_SCHEMA
- Sequences
- Users
- H2 1.4.200 (2019-10-14)

At the top, there are several configuration buttons and dropdowns. Below the schema tree, there are buttons for "Run", "Run Selected", "Auto complete", "Clear", and a text input field for "SQL statement".

Important Commands

	Displays this Help Page
	Shows the Command History
	Executes the current SQL statement
	Executes the SQL statement defined by the text selection
	Auto complete
	Disconnects from the database

Sample SQL Script

Delete the table if it exists	DROP TABLE IF EXISTS TEST;
Create a new table with ID and NAME columns	CREATE TABLE TEST(ID INT PRIMARY KEY, NAME VARCHAR(255));
Add a new row	INSERT INTO TEST VALUES(1, 'Hello');
Add another row	INSERT INTO TEST VALUES(2, 'World');
Query the table	SELECT * FROM TEST ORDER BY ID;
Change data in a row	UPDATE TEST SET NAME='Hi' WHERE ID=1;
Remove a row	DELETE FROM TEST WHERE ID=2;
Help	HELP ...

Adding Database Drivers

我们还可以添加如下配置来管理这个GUI：

```
spring.h2.console.path=/h2-console
spring.h2.console.settings.trace=false
spring.h2.console.settings.web-allow-others=false
```

其中path指定了路径，trace指定是否开启trace output，web-allow-others指定是否允许远程登录。

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-h2>

在Spring Boot中使用内存数据库

所谓内存数据库就是在内存中运行的数据库，不需要将数据存储在文件系统中，但是相对于普通的数据库而言，内存数据库因为数据都在内存中，所以内存的数据库的存取速度会更快。

本文我们将会讨论如何在Spring Boot中使用内存数据库。

H2数据库

H2是一个由java实现的开源内存数据库，它可以支持内存模式和独立模式。如果要使用H2数据库，需要添加如下依赖：

```
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <version>1.4.194</version>
</dependency>
```

我们可以在配置文件中设置更多的H2数据库的连接信息：

```
driverClassName=org.h2.Driver
url=jdbc:h2:mem:myDb;DB_CLOSE_DELAY=-1
username=sa
password=sa
```

默认情况下H2数据库当没有连接的时候会自动关闭，我们可以通过添加DB_CLOSE_DELAY=-1来禁止掉这个功能。

如果我们需要使用Hibernate，我们需要设置如下内容：

```
hibernate.dialect=org.hibernate.dialect.H2Dialect
```

HSQldb

HSQldb是一个开源项目，java写的关系型数据库。它可以支持基本的SQL操作，存储过程和触发器。同样嵌入式或者单独使用。

我们看下怎么添加依赖：

```
<dependency>
    <groupId>org.hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>2.3.4</version>
</dependency>
```

下面是HSQLDB的配置文件：

```
driverClassName=org.hsqldb.jdbc.JDBCDataSource  
url=jdbc:hsqldb:mem:myDb  
username=sa  
password=sa
```

同样的如果使用hibernate需要配置如下属性：

```
hibernate.dialect=org.hibernate.dialect.HSQLDialect
```

Apache Derby

Apache Derby 是由Apache基金会维护的开源项目。

添加依赖：

```
<dependency>  
    <groupId>org.apache.derby</groupId>  
    <artifactId>derby</artifactId>  
    <version>10.13.1.1</version>  
</dependency>
```

配置文件：

```
driverClassName=org.apache.derby.jdbc.EmbeddedDriver  
url=jdbc:derby:memory:myDb;create=true  
username=sa  
password=sa
```

对应的hibernate配置：

```
hibernate.dialect=org.hibernate.dialect.DerbyDialect
```

SQLite

SQLite也是一种内存数据库，我们这样添加依赖：

```
<dependency>  
    <groupId>org.xerial</groupId>  
    <artifactId>sqlite-jdbc</artifactId>  
    <version>3.16.1</version>  
</dependency>
```

配置文件如下：

```
driverClassName=org.sqlite.JDBC  
url=jdbc:sqlite:memory:myDb  
username=sa  
password=sa
```

使用Spring Boot可以很方便的使用上面提到的内存数据库。

更多教程请参考 [flydean的博客](#)

www.flydean.com

Spring Boot JPA中使用@Entity和@Table

本文中我们会讲解如何在Spring Boot JPA中实现class和数据表格的映射。

默认实现

Spring Boot JPA底层是用Hibernate实现的， 默认情况下， 数据库表格的名字是相应的class名字的首字母大写。命名的定义是通过接口ImplicitNamingStrategy来定义的：

```
/**
 * Determine the implicit name of an entity's primary table.
 *
 * @param source The source information
 *
 * @return The implicit table name.
 */
public Identifier determinePrimaryTableName(ImplicitEntityNameSource source);
```

我们看下它的实现ImplicitNamingStrategyJpaCompliantImpl：

```
@Override
public Identifier determinePrimaryTableName(ImplicitEntityNameSource source) {
    if ( source == null ) {
        // should never happen, but to be defensive...
        throw new HibernateException( "Entity naming information was not provided." );
    }

    String tableName = transformEntityName( source.getEntityNaming() );

    if ( tableName == null ) {
        // todo : add info to error message - but how to know what to write since we failed to interpret the naming source
        throw new HibernateException( "Could not determine primary table name for entity" );
    }

    return toIdentifier( tableName, source.getBuildingContext() );
}
```

如果我们要修改系统的默认实现，则可以实现接口PhysicalNamingStrategy：

```
public interface PhysicalNamingStrategy {
    public Identifier toPhysicalCatalogName(Identifier name, JdbcEnvironment jdbcEnvironment);
```

```
public Identifier toPhysicalSchemaName(Identifier name, JdbcEnvironment jdbcEnvironment);

public Identifier toPhysicalTableName(Identifier name, JdbcEnvironment jdbcEnvironment);

public Identifier toPhysicalSequenceName(Identifier name, JdbcEnvironment jdbcEnvironment);

public Identifier toPhysicalColumnName(Identifier name, JdbcEnvironment jdbcEnvironment);
}
```

使用@Table自定义表格名字

我们可以在@Entity中使用@Table来自定义映射的表格名字：

```
@Entity
@Table(name = "ARTICLES")
public class Article {
    // ...
}
```

当然，我们可以将整个名字写在静态变量中：

```
@Entity
@Table(name = Article.TABLE_NAME)
public class Article {
    public static final String TABLE_NAME= "ARTICLES";
    // ...
}
```

在JPQL Queries中重写表格名字

通常我们在@Query中使用JPQL时可以这样用：

```
@Query("select * from Article")
```

其中Article默认是Entity类的Class名称，我们也可以这样来修改它：

```
@Entity(name = "MyArticle")
```

这时候我们可以这样定义JPQL：

```
@Query("select * from MyArticle")
```

更多教程请参考 [flydean的博客](#)

www.flydean.com

Spring Boot JPA的查询语句

之前的文章中，我们讲解了如何使用Spring Boot JPA，在Spring Boot JPA中我们可通过构建查询方法或者通过@Query注解来构建查询语句，本文我们将会更详细的讨论查询语句的构建。

准备工作

首先我们需要添加依赖，这里我们还是使用H2内存数据库：

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

我们创建一个Entity：

```
@Data
@Entity
public class Movie {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    private Long id;
    private String title;
    private String director;
    private String rating;
    private int duration;
}
```

构建初始化data.sql：

```
INSERT INTO movie(id, title, director, rating, duration)
    VALUES(1, 'Godzilla: King of the Monsters', 'Michael Dougherty', 'PG-13', 132)
;
INSERT INTO movie(id, title, director, rating, duration)
    VALUES(2, 'Avengers: Endgame', 'Anthony Russo', 'PG-13', 181);
INSERT INTO movie(id, title, director, rating, duration)
    VALUES(3, 'Captain Marvel', 'Anna Boden', 'PG-13', 123);
INSERT INTO movie(id, title, director, rating, duration)
    VALUES(4, 'Dumbo', 'Tim Burton', 'PG', 112);
INSERT INTO movie(id, title, director, rating, duration)
```

```
VALUES(5, 'Booksmart', 'Olivia Wilde', 'R', 102);
INSERT INTO movie(id, title, director, rating, duration)
VALUES(6, 'Aladdin', 'Guy Ritchie', 'PG', 128);
INSERT INTO movie(id, title, director, rating, duration)
VALUES(7, 'The Sun Is Also a Star', 'Ry Russo-Young', 'PG-13', 100);
```

构建Repository:

```
public interface MovieRepository extends JpaRepository<Movie, Long> {  
}
```

Containing, Contains, IsContaining 和 Like

如果我们想要构建模下面的模糊查询语句：

```
SELECT * FROM movie WHERE title LIKE '%in%';
```

我们可以这样写：

```
List<Movie> findByTitleContaining(String title);
List<Movie> findByTitleContains(String title);
List<Movie> findByTitleIsContaining(String title);
```

将上面的语句添加到Repository中就够了。

我们看下怎么测试：

```
@Slf4j
@RunWith(SpringRunner.class)
@SpringBootTest(classes = {QueryApp.class})
public class MovieRepositoryTest {

    @Autowired
    private MovieRepository movieRepository;

    @Test
    public void TestMovieQuery(){
        List<Movie> results = movieRepository.findByTitleContaining("in");
        assertEquals(3, results.size());

        results = movieRepository.findByTitleIsContaining("in");
        assertEquals(3, results.size());

        results = movieRepository.findByTitleContains("in");
        assertEquals(3, results.size());
    }
}
```

Spring 还提供了Like 关键词，我们可以这样用：

```
List<Movie> findByTitleLike(String title);
```

测试代码：

```
results = movieRepository.findByTitleLike("%in%");  
assertEquals(3, results.size());
```

StartsWith

如果我们需要实现下面这条SQL：

```
SELECT * FROM Movie WHERE Rating LIKE 'PG%';
```

我们可以这样使用：

```
List<Movie> findByRatingStartsWith(String rating);
```

测试代码如下：

```
List<Movie> results = movieRepository.findByRatingStartsWith("PG");  
assertEquals(6, results.size());
```

EndsWith

如果我们要实现下面的SQL：

```
SELECT * FROM Movie WHERE director LIKE '%Burton';
```

可以这样构建：

```
List<Movie> findByDirectorEndsWith(String director);
```

测试代码如下：

```
List<Movie> results = movieRepository.findByDirectorEndsWith("Burton");  
assertEquals(1, results.size());
```

大小写不敏感

要是想实现大小不敏感的功能我们可以这样:

```
List<Movie> findByTitleContainingIgnoreCase(String title);
```

测试代码如下:

```
List<Movie> results = movieRepository.findByTitleContainingIgnoreCase("the");
assertEquals(2, results.size());
```

Not

要想实现Not的功能, 我们可以使用NotContains, NotContaining, 和 NotLike关键词:

```
List<Movie> findByRatingNotContaining(String rating);
```

测试代码如下:

```
List<Movie> results = movieRepository.findByRatingNotContaining("PG");
assertEquals(1, results.size());
```

NotLike:

```
List<Movie> findByDirectorNotLike(String director);
```

测试代码如下:

```
List<Movie> results = movieRepository.findByDirectorNotLike("An%");
assertEquals(5, results.size());
```

@Query

如果我们要实现比较复杂的查询功能, 我们可以使用@Query, 下面是一个命名参数的使用:

```
@Query("SELECT m FROM Movie m WHERE m.title LIKE %:title%")
List<Movie> searchByTitleLike(@Param("title") String title);
```

如果有多个参数, 我们可以这样指定参数的顺序:

```
@Query("SELECT m FROM Movie m WHERE m.rating LIKE ?1%")
List<Movie> searchByRatingStartsWith(String rating);
```

下面是测试代码:

```
List<Movie> results = movieRepository.searchByRatingStartsWith("PG");
assertEquals(6, results.size());
```

在Spring Boot2.4之后，我们可以使用SpEL表达式：

```
@Query("SELECT m FROM Movie m WHERE m.director LIKE ?#{escape([0])} escape ?#{escapeCharacter()}")
List<Movie> searchByDirectorEndsWith(String director);
```

看下怎么使用：

```
List<Movie> results = movieRepository.searchByDirectorEndsWith("Burton");
assertEquals(1, results.size());
```

本文的例子可以参考

更多教程请参考 [flydean的博客](#)

Spring Boot JPA中关联表的使用

本文中，我们会将会通过一个Book和Category的关联关系，来讲解如何在JPA中使用。

添加依赖

我们还是使用H2内存数据库来做测试：

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

<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
```

构建Entity

下面我们构建两个Entity：

```
@Data
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;

    @ManyToOne
    private Category category;
}
```

```
@Data
@Entity
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String name;
```

```

    @OneToOne(mappedBy = "category", cascade = CascadeType.ALL)
    private List<Book> books;
}

```

上面我们定义了两个Entity，Category和Book是一对多的关系。我们通过@ManyToOne和@OneToMany来定义相应的关系。

构建Repository

我们接下来构建相应的Repository:

```

public interface BookRepository extends CrudRepository<Book, Long> {
    long deleteByTitle(String title);

    @Modifying
    @Query("delete from Book b where b.title=:title")
    void deleteBooks(@Param("title") String title);
}

```

```
public interface CategoryRepository extends CrudRepository<Category, Long> {}
```

构建初始数据

为了方便测试，我们先构建需要的数据schema.sql和data.sql:

```

CREATE TABLE book (
    id      BIGINT      NOT NULL AUTO_INCREMENT,
    title  VARCHAR(128) NOT NULL,
    category_id BIGINT,
    PRIMARY KEY (id)
);

```

```

CREATE TABLE category (
    id      BIGINT      NOT NULL AUTO_INCREMENT,
    name   VARCHAR(128) NOT NULL,
    PRIMARY KEY (id)
);

```

```

insert into book(id,title,category_id)
values(1,'The Hobbit',1);
insert into book(id,title,category_id)
values(2,'The Rabbit',1);

insert into category(id,name)
values(1,'category');

```

测试

我们看一下怎么从Book中删除一条数据：

```
@Test  
public void whenDeleteByIdFromRepository_thenDeletingShouldBeSuccessful() {  
    assertThat(bookRepository.count()).isEqualTo(2);  
    bookRepository.deleteById(1L);  
    assertThat(bookRepository.count()).isEqualTo(1);  
}
```

再看一下category的删除：

```
@Test  
public void whenDeletingCategories_thenBooksShouldAlsoBeDeleted() {  
    categoryRepository.deleteAll();  
    assertThat(bookRepository.count()).isEqualTo(0);  
    assertThat(categoryRepository.count()).isEqualTo(0);  
}
```

再看一下book的删除：

```
@Test  
public void whenDeletingBooks_thenCategoriesShouldAlsoBeDeleted() {  
    bookRepository.deleteAll();  
    assertThat(bookRepository.count()).isEqualTo(0);  
    assertThat(categoryRepository.count()).isEqualTo(1);  
}
```

因为我们只在Category中指定了cascade = CascadeType.ALL， 所以删除category的时候可以删除相关联的Book，但是删除Book的时候不会删除相关联的category。

本文的例子可以参考<https://github.com/ddean2009/learn-springboot2/tree/master/springboot-jpa-relation>

Spring Boot JPA 中transaction的使用

transaction是我们在做数据库操作的时候不能回避的一个话题，通过transaction，我们可以保证数据库操作的原子性，一致性，隔离性和持久性。

本文我们将会深入的探讨Spring Boot JPA中@Transactional注解的使用。

通过@Transactional注解，我们可以设置事物的传播级别和隔离级别，同时可以设置timeout, read-only, 和 rollback等特性。

@Transactional的实现

Spring通过创建代理或者操纵字节码来实现事物的创建，提交和回滚操作。如果是代理模式的话，Spring会忽略掉@Transactional的内部方法调用。

如果我们有个方法callMethod，并标记它为@Transactional,那么Spring Boot的实现可能是如下方式：

```
createTransactionIfNecessary();
try {
    callMethod();
    commitTransactionAfterReturning();
} catch (exception) {
    completeTransactionAfterThrowing();
    throw exception;
}
```

@Transactional的使用

@Transactional使用起来很简单，可以放在class上，可以放在interface上，也可以放在方法上面。

如果放在方法上面，那么该方法中的所有public方法都会应用该Transaction。

如果@Transactional放在private方法上面，则Spring Boot将会忽略它。

Transaction的传播级别

传播级别Propagation定义了Transaction的边界，我们可以很方便的在@Transactional注解中定义不同的传播级别。

下面我们来分别看一下Transaction的传播级别。

REQUIRED

REQUIRED是默认的传播级别，下面的两种写法是等价的：

```
@Transactional  
public void deleteBookWithDefaultTransaction(Long id) {  
    bookRepository.deleteBookById(id);  
}  
  
@Transactional(propagation = Propagation.REQUIRED)  
public void deleteBookWithRequired(Long id) {  
}
```

Spring会检测现在是否有一个有效的transaction。如果没有则创建，如果有transaction，则Spring将会把该放方法的业务逻辑附加到已有的transaction中。

我们再看下REQUIRED的伪代码：

```
if (isExistingTransaction()) {  
    if (isValidExistingTransaction()) {  
        validateExisitingAndThrowExceptionIfNotValid();  
    }  
    return existing;  
}  
return createNewTransaction();
```

SUPPORTS

在SUPPORTS的情况下，Spring首先会去检测是否有存在Transaction，如果存在则使用，否则不会使用transaction。

我们看下代码怎么使用：

```
@Transactional(propagation = Propagation.SUPPORTS)  
public void deleteBookWithSupports(Long id) {  
}
```

SUPPORTS的实现伪代码如下：

```
if (isExistingTransaction()) {  
    if (isValidExistingTransaction()) {  
        validateExisitingAndThrowExceptionIfNotValid();  
    }  
    return existing;  
}  
return emptyTransaction;
```

MANDATORY

在MANDATORY情况下，Spring先会去检测是否有一个Transaction存在，如果存在则使用，否则抛出异常。

我们看下代码怎么使用：

```
@Transactional(propagation = Propagation.MANDATORY)
public void deleteBookWithMandatory(Long id) {
}
```

MANDATORY的实现逻辑如下：

```
if (isExistingTransaction()) {
    if (isValidExistingTransaction()) {
        validateExisitingAndThrowExceptionIfNotValid();
    }
    return existing;
}
throw IllegalTransactionStateException;
```

NEVER

如果是NEVER的情况下，如果现在有一个Transaction存在，则Spring会抛出异常。

使用的代码如下：

```
@Transactional(propagation = Propagation.NEVER)
public void deleteBookWithNever(Long id) {
}
```

实现逻辑代码如下：

```
if (isExistingTransaction()) {
    throw IllegalTransactionStateException;
}
return emptyTransaction;
```

NOT_SUPPORTED

如果使用的是NOT_SUPPORTED，那么Spring将会首先暂停现有的transaction，然后在非transaction情况下执行业务逻辑。

我们这样使用：

```
@Transactional(propagation = Propagation.NOT_SUPPORTED)
public void deleteBookWithNotSupported(Long id) {
}
```

REQUIRES_NEW

当REQUIRES_NEW使用时， Spring暂停当前的Transaction，并创建一个新的。

我们看下代码怎么使用：

```
@Transactional(propagation = Propagation.REQUIRES_NEW)
public void deleteBookWithRequiresNew(Long id){}
```

相应的实现代码如下：

```
if (isExistingTransaction()) {
    suspend(existing);
    try {
        return createNewTransaction();
    } catch (exception) {
        resumeAfterBeginException();
        throw exception;
    }
}
return createNewTransaction();
```

NESTED

NESTED顾名思义，是嵌套的Transaction，Spring首先检查transaction是否存在，如果存在则创建一个savepoint，如果我们的程序抛出异常的时候，transaction将会回滚到该savepoint。如果没有transaction，NESTED的表现和REQUIRED一样。

我们看下怎么使用：

```
@Transactional(propagation = Propagation.NESTED)
public void deleteBookWithNested(Long id){}
```

Transaction的隔离级别

隔离级别就是我们之前提到的原子性，一致性，隔离性和持久性。隔离级别描述了改动对其他并发者的可见程度。

隔离级别主要是为了防止下面3个并发过程中可能出现的问题：

1. 脏读：读取一个transaction还没有提交的change
2. 不可重复读：在一个transaction修改数据库中的某行数据时，另外一个transaction多次读取同一行数据，获取到的不同的值。
3. 幻读：在一个transaction添加或者删除数据库的数据时，另外一个transaction做范围查询，获得了不同的数据行数。

READ_UNCOMMITTED

READ_UNCOMMITTED是隔离级别中最低的级别。这个级别下，并发的3个问题都可能出现。

我们这样使用：

```
@Transactional(isolation = Isolation.READ_UNCOMMITTED)
public void deleteBookWithReadUncommitted(Long id){}
```

READ_COMMITTED

READ_COMMITTED可以防止脏读。

我们看下代码：

```
@Transactional(isolation = Isolation.READ_COMMITTED)
public void deleteBookWithReadCommitted(Long id){}
```

REPEATABLE_READ

REPEATABLE_READ可以防止脏读和不可重复读。

使用的代码如下：

```
@Transactional(isolation = Isolation.REPEATABLE_READ)
public void deleteBookWithRepeatableRead(Long id){}
```

SERIALIZABLE

SERIALIZABLE是最严格的基本，可以防止脏读，不可重复读和幻读。

我们看下怎么使用：

```
@Transactional(isolation = Isolation.SERIALIZABLE)
public void deleteBookWithSerializable(Long id){}
```