



# SpringBoot Cookbook

极客学院出版

# 前言

---

## 内容来源

本书的内容主要来自 [《Spring Boot Cookbook》](#) 一书，我在阅读这本书的过程中进行了简单的总结和翻译，另外，本书还包括了我自己在项目中使用Spring Boot的实践总结。

最开始是一些博客文章，记录于简书：[杜琪的博客](#)，后来经极客学院编辑朋友推荐和帮忙，编写成这本小册子，希望大家喜欢。

## 进度记录

- 2016年1月19日，更新文章中的链接
- 2016年1月6日，在极客学院wiki上线
- 2016年1月5日，项目启动

## 感谢支持

- [极客学院 Wiki](#) 提供图文教程托管服务

## 离线版本

敬请期待

# 目录

---

前言 . . . . .	1
Spring Boot的自动配置、Command-line Runner . . . . .	6
RESTful by Spring Boot with MySQL . . . . .	11
Spring Boot: Data Rest Service . . . . .	17
Spring Boot: 定制servlet filters . . . . .	22
Spring Boot: 定制拦截器 . . . . .	24
Spring Boot: 定制HTTP消息转换器 . . . . .	26
Spring Boot: 定制PropertyEditors . . . . .	28
Spring Boot: 定制type Formatters . . . . .	30
Spring Boot: 定制URL匹配规则 . . . . .	34
Spring Boot: 定制static path mappings . . . . .	36
通过EmbeddedServletContainerCustomizer接口调优Tomcat . . . . .	38
选择Spring Boot项目的内嵌容器 . . . . .	40
让你的Spring Boot工程支持HTTP和HTTPS . . . . .	42
了解Spring Boot的自动配置 . . . . .	50
Spring Boot: 定制自己的starter . . . . .	53
配置是否初始化Bean的方法 . . . . .	57
通过@Enable*注解触发Spring Boot配置 . . . . .	59
Spring Boot应用的测试——Mockito . . . . .	63
初始化数据库和导入数据 . . . . .	69
在测试中使用内存数据库 . . . . .	73
利用Mockito模拟DB . . . . .	76
在Spring Boot项目中使用Spock框架 . . . . .	80
Spring Boot应用的打包和部署 . . . . .	87

Docker with Spring Boot. . . . .	89
Spring Boot应用的健康监控. . . . .	96
Spring Boot Admin的使用. . . . .	104
通过JMX监控Spring Boot应用 . . . . .	113
<b>第 8 章 [第八章 Spring Boot实践总结] . . . . .</b>	<b>117</b>
Spring Boot with Mysql . . . . .	118
Restful: Spring Boot with Mongodb. . . . .	125
Spring Boot with Redis . . . . .	130

HTML



T



unity



HTML



- [Spring Boot的自动配置、Command-line Runner](#)
- [RESTful by Spring Boot with MySQL](#)
- [Spring Boot: Data Rest Service](#)

## Spring Boot的自动配置、Command-line Runner

---

接下来关于SpringBoot的一系列文章和例子，都来自《Spring Boot Cookbook》这本书，本文的主要内容是start.spring.io的使用、Spring Boot的自动配置以及CommandRunner的角色和应用场景。

### 1. start.spring.io的使用

首先带你浏览<http://start.spring.io/>，在这个网址中有一些Spring Boot提供的组件，然后会给你展示如何让你的Spring工程变得“Bootiful”，我称之为“Boot化”。

在网站[Spring Initializr](#)上填对应的表单，描述Spring Boot项目的主要信息，例如Project Metadata、Dependency等。在Project Dependencies区域，你可以根据应用程序的功能需要选择相应的starter。

Spring Boot starters可以简化Spring项目的库依赖管理，将某一特定功能所需要的依赖库都整合在一起，就形成一个starter，例如：连接数据库、springmvc、spring测试框架等等。简单来说，spring boot使得你的pom文件从此变得很清爽且易于管理。

常用的starter以及用处可以列举如下：

- **spring-boot-starter**: 这是核心Spring Boot starter，提供了大部分基础功能，其他starter都依赖于它，因此没有必要显式定义它。
- **spring-boot-starter-actuator**: 主要提供监控、管理和审查应用程序的功能。
- **spring-boot-starter-jdbc**: 该starter提供对JDBC操作的支持，包括连接数据库、操作数据库，以及管理数据库连接等等。
- **spring-boot-starter-data-jpa**: JPA starter提供使用Java Persistence API(例如Hibernate等)的依赖库。
- **spring-boot-starter-data-\***: 提供对MongoDB、Data-Rest或者Solr的支持。
- **spring-boot-starter-security**: 提供所有Spring-security的依赖库。
- **spring-boot-starter-test**: 这个starter包括了spring-test依赖以及其他测试框架，例如JUnit和Mockito等等。
- **spring-boot-starter-web**: 该starter包括web应用程序的依赖库。

## How do

首先我们要通过start.spring.io创建一个图书目录管理程序，它会记录出版图书的记录，包括作者、审阅人、出版社等等。我们将这个项目命名为BookPub，具体的操作步骤如下：

- 点击“Switch to the full version.”，展示完整页面；
- Group设置为：org.test；
- Artifact设置为：bookpub；
- Name设置为：BookPub；
- Package Name设置为：org.test.bookpub；
- Packaging代表打包方式，我们选jar；
- Spring Boot Version选择最新的1.3.0；
- 创建Maven工程，当然，对Gradle比较熟悉的同学可以选择Gradle工程。
- 点击“Generate Project”下载工程包。

利用IDEA导入下载的工程，可以看到pom文件的主体如下所示：

```
<groupId>com.test</groupId>
<artifactId>bookpub</artifactId>
<version>0.0.1-SNAPSHOT</version>
<packaging>jar</packaging>

<name>BookPub</name>
<description>Demo project for Spring Boot</description>

<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.3.0.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
</parent>

<properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <java.version>1.8</java.version>
</properties>

<dependencies>
    <dependency>
```

```

<groupId>org.springframework.boot</groupId>
<artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
</dependency>
</dependencies>

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

```

## 2. Spring Boot的自动配置

在Spring Boot项目中，`xxxApplication.java`会作为应用程序的入口，负责程序启动以及一些基础性的工作。`@SpringBootApplication`是这个注解是该应用程序入口的标志，然后有熟悉的`main`函数，通过`SpringApplication.run(xxxApplication.class, args)`来运行Spring Boot应用。打开`SpringBootApplication`注解可以发现，它是由其他几个类组合而成的：`@Configuration`（等同于spring中的xml配置文件，使用Java文件做配置可以检查类型安全）、`@EnableAutoConfiguration`（自动配置，稍后细讲）、`@ComponentScan`（组件扫描，大家非常熟悉的，可以自动发现和装配一些Bean）。

我们在`pom`文件里可以看到，`com.h2database`这个库起作用的范围是`runtime`，也就是说，当应用程序启动时，如果Spring Boot在`classpath`下检测到`org.h2.Driver`的存在，会自动配置H2数据库连接。现在启动应用程序来观察，以验证我们的想法。打开`shell`，进入项目文件夹，利用`mvn spring-boot:run`启动应用程序，如下图所示。

图片 1.1 Spring Boot 的自动配置

可以看到类似Building JPA container EntityManagerFactory for persistence unit 'default'、HHH000412: Hibernate Core {4.3.11.Final}、HHH000400: Using dialect: org.hibernate.dialect.H2Dialect这些Info信息；由于我们之前选择了jdbc和jpa等starters，Spring Boot将自动创建JPA容器，并使用Hibernate4.3.11，使用H2Dialect管理H2数据库（内存数据库）。

### 3. 使用Command-line runners

我们新建一个StartupRunner类，该类实现CommandLineRunner接口，这个接口只有一个函数：public void run(String... args)，最重要的是：这个方法会在应用程序启动后首先被调用。

How do

- 在 `src/main/java/org/test/bookpub/` 下建立 `StartRunner` 类，代码如下：

```
package com.test.bookpub;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;

public class StartupRunner implements CommandLineRunner {
    protected final Logger logger = LoggerFactory.getLogger(StartupRunner.class);
```

```
@Override  
public void run(String... strings) throws Exception {  
    logger.info("hello");  
}  
}
```

- 在BookPubApplication类中创建bean对象，代码如下：

```
@Bean  
public StartupRunner schedulerRunner() {  
    return new StartupRunner();  
}
```

还是用 `mvn spring-boot:run` 命令启动程序，可以看到hello的输出。对于那种只需要在应用程序启动时执行一次的任务，非常适合利用Command line runners来完成。Spring Boot应用程序在启动后，会遍历CommandLineRunner接口的实例并运行它们的run方法。也可以利用@Order注解（或者实现Order接口）来规定所有CommandLineRunner实例的运行顺序。

利用command-line runner的这个特性，再配合依赖注入，可以在应用程序启动时后首先引入一些依赖bean，例如data source、rpc服务或者其他模块等等，这些对象的初始化可以放在run方法中。不过，需要注意的是，在run方法中执行初始化动作的时候一旦遇到任何异常，都会使得应用程序停止运行，因此最好利用try/catch语句处理可能遇到的异常。

## RESTful by Spring Boot with MySQL

---

现在的潮流是前端承担越来越多的责任：MVC中的V和C，后端只需要负责提供数据M，但是后端有更重要的任务：高并发、提供各个维度的扩展能力（负载均衡、数据表切分、服务分离）、更清晰的API设计。Spring Boot框架提供的机制便于工程师实现标准的RESTful接口，本文主要讨论如何编写Controller代码，另外还涉及了MySQL的数据库操作，之前我也写过一篇关于Mysql的文章[link](#)，但是这篇文章加上了CRUD的操作。

先回顾下之前的文章中我们用到的例子：图书信息管理系统，主要的领域对象有book、author、publisher和reviewer。

首先我们要在pom文件中添加对应的starter，即 `spring-boot-starter-web`，对应的xml代码示例为：

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

然后我们要创建控制器（Controller），先在项目根目录下创建controller包，一般为每个实体类对象创建一个控制器，例如BookController。

`@RestController`注解是`@Controller`和`@ResponseBody`的合集，表示这是个控制器bean，并且是将函数的返回值直接填入HTTP响应体中，是REST风格的控制器。`@RequestMapping("/books")`表示该控制器处理所有“/books”的URL请求，具体由那个函数处理，要根据HTTP的方法来区分：GET表示查询、POST表示提交、PUT表示更新、DELETE表示删除。

- 查询所有图书记录：利用`@Autowired`导入`BookRepository`的Bean，直接调用`bookRepository.findAllBooks()`即可。我们的返回值形式如下。关于RESTful返回值形式的设计，后续会有专门的文章讨论。

```
{
    "message": "get all books",
    "book": [
        {
            "isbn": "9781-1234-5678",
            "title": "你愁啥",
            "description": "这是一本奇怪的书",
            "author": {
                "firstName": "冯",
                "lastName": "pp"
            },
            "publisher": {
                "name": "大锤出版社"
            }
        }
    ]
}
```

```

    },
    "reviewers": []
},
{
  "isbn": "9781-1234-1111",
  "title": "别吵吵",
  "description": "哈哈哈",
  "author": {
    "firstName": "杜琪",
    "lastName": "琪"
  },
  "publisher": {
    "name": "大锤出版社"
  },
  "reviewers": []
}
]
}

```

- 根据isbn查询图书记录：根据isbn查询一本书的记录，调用bookRepository.findBookByIsbn() 即可。返回值形式如下：

```

{
  "message": "get book with isbn(9781-1234-5678)",
  "book": {
    "isbn": "9781-1234-5678",
    "title": "你愁啥",
    "description": "这是一本奇怪的书",
    "author": {
      "firstName": "冯",
      "lastName": "pp"
    },
    "publisher": {
      "name": "大锤出版社"
    },
    "reviewers": []
  }
}

```

- 添加图书记录，客户端的图书信息封装成json字符串传递过来，因此利用@RequestBody获取POST请求体，由于book记录中有外链记录，因此要首先解析出author对象和publisher对象，并将它们存入数据库；然后才生成book对象，并调用bookRepository.save(book) 将book记录存入数据库。该接口的返回值会把刚添加的图书信息返回给客户端，形式类似于getBookByIsbn这个接口。

- 更新图书书名，这里简单以这个接口作为更新的例子。主要步骤是先取出对应isbn的book对象，然后 book. setTitle(title) 更新book信息，然后调用bookRepository. save(book) 更新该对象的信息，通过@PathVariable 修饰的参数title与URL中用“{title}”的值对应。
- 删除图书记录；给定图书的isbn直接删除即可。

最后，放上完整的Controller代码：

```
package com.test.bookpub.controller;

import com.alibaba.fastjson.JSONObject;
import com.test.bookpub.domain.Author;
import com.test.bookpub.domain.Book;
import com.test.bookpub.domain.Publisher;
import com.test.bookpub.repository.AuthorRepository;
import com.test.bookpub.repository.BookRepository;
import com.test.bookpub.repository.PublisherRepository;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.*;

/**
 * @author duqi
 * @create 2015-12-02 18:18
 */
@RestController
@RequestMapping("/books")
public class BookController {
    private static final Logger logger = LoggerFactory.getLogger(BookController.class);

    @Autowired
    private BookRepository bookRepository;
    @Autowired
    public AuthorRepository authorRepository;
    @Autowired
    public PublisherRepository publisherRepository;

    @RequestMapping(method = RequestMethod.GET)
    public Iterable<Book> getAllBooks() {
        return bookRepository.findAll();
    }
}
```

```
@RequestMapping(value = "/{isbn}", method = RequestMethod.GET)
public Map<String, Object> getBook(@PathVariable String isbn) {
    Book book = bookRepository.findBookByIsbn(isbn);

    Map<String, Object> response = new LinkedHashMap<>();
    response.put("message", "get book with isbn(" + isbn + ")");
    response.put("book", book);
    return response;
}

@RequestMapping(method = RequestMethod.POST)
public Map<String, Object> addBook(@RequestBody JSONObject bookJson) {
    JSONObject authorJson = bookJson.getJSONObject("author");
    Author author = new Author(authorJson.getString("firstName"), authorJson.getString("lastName"));
    authorRepository.save(author);

    String isbn = bookJson.getString("isbn");
    JSONObject publisherJson = bookJson.getJSONObject("publisher");
    Publisher publisher = new Publisher(publisherJson.getString("name"));
    publisherRepository.save(publisher);

    String title = bookJson.getString("title");
    String desc = bookJson.getString("desc");
    Book book = new Book(author, isbn, publisher, title);
    book.setDescription(desc);
    bookRepository.save(book);

    Map<String, Object> response = new LinkedHashMap<>();
    response.put("message", "book add successfully");
    response.put("book", book);
    return response;
}

@RequestMapping(value = "/{isbn}", method = RequestMethod.DELETE)
public Map<String, Object> deleteBook(@PathVariable String isbn) {
    Map<String, Object> response = new LinkedHashMap<>();
    try {
        bookRepository.deleteBookByIsbn(isbn);
    } catch (NullPointerException e) {
        logger.error("the book is not in database");
        response.put("message", "delete failure");
        response.put("code", 0);
    }

    response.put("message", "delete successfully");
    response.put("code", 1);
}
```

```

        return response;
    }

    @RequestMapping(value = "/{isbn}/{title}", method = RequestMethod.PUT)
    public Map<String, Object> updateBookTitle(@PathVariable String isbn, @PathVariable String title) {
        Map<String, Object> response = new LinkedHashMap<>();
        Book book = null;
        try {
            book = bookRepository.findBookByIsbn(isbn);
            book.setTitle(title);
            bookRepository.save(book);
        } catch (NullPointerException e) {
            response.put("message", "can not find the book");
            return response;
        }

        response.put("message", "book update successfully");
        response.put("book", book);
        return response;
    }
}

```

有三个问题需要补充探讨

现在我要说下Controller的角色，大家可以看到，我这里将很多业务代码混淆在Controller的代码中。实际上，根据[程序员必知之前端演进史](#)一文所述Controller层应该做的事是： 处理请求的参数 渲染和重定向 选择Model和Service 处理Session和Cookies，我基本上认同这个观点，最多再加上OAuth验证（利用拦截器实现即可）。而真正的业务逻辑应该单独分处一层来处理，即常见的service层；

今天遇到一个类似参考资料2中的错误，我经过查找后发现是Jackson解析我的对象的时候出现了无限递归解析，究其原因，是因为外链： 解析book的时候，需要解析author，但是在author中又有books选项，所以造成死循环，解决的办法就是在author中的books属性上加上注解： @JsonBackReference； 同样需要在Publisher类中的books属性加上@JsonBackReference注解。

上述演示的Controller代码还有两个问题： 返回值形式不统一； 并没有遵循标准的API设计（例如update方法实际上应该由客户端返回更新过的完整对象，这样就可以直接调用save方法），后续，我会参考[RESTful API 设计指南](#)进行学习，对API的设计进行自己的学习总结，读者朋友，你也需要自己实践和学习哦，有问题的可以找我讨论。

## 参考资料

1. [repository中的update方法](#)

2. [使用spring data创建REST应用](#)
3. [遇到的一个错误: at com.fasterxml.jackson.databind.ser.BeanSerializer.serialize](#)
4. [SPRING BOOT: DATA ACCESS WITH JPA, HIBERNATE AND MYSQL](#)

## Spring Boot: Data Rest Service

---

在文章[RESTful by Spring Boot with MySQL](#)通过在Controller中引入BookRepository来对外提供REST API。Spring Boot还可以通过 `spring-boot-starter-data-rest` 来对外提供REST API，可以免于编写对应的Controller，且具备分页和排序的功能。

### 实践

- 在pom文件中添加依赖项

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

- 在包com.test.bookpub.repository下创建AuthorRepository接口，该接口继承自PagingAndSortingRepository，并用@RepositoryRestResource注解修饰。代码如下：

```
package com.test.bookpub.repository;

import com.test.bookpub.domain.Author;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource
public interface AuthorRepository
    extends PagingAndSortingRepository<Author, Long> {
}
```

- 可以看出，实际编写的代码很少，同样套路，为Publisher和Reviewer也添加类似的接口。PublisherRepository的代码如下：

```
package com.test.bookpub.repository;

import com.test.bookpub.domain.Publisher;
import org.springframework.data.repository.PagingAndSortingRepository;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource
public interface PublisherRepository
```

```
    extends PagingAndSortingRepository<Publisher, Long> {
}
```

ReviewerRepository的代码如下：

```
package com.test.bookpub.repository;
import org.springframework.data.repository.PagingAndSortingRepository;
import com.test.bookpub.domain.Publisher.Reviewer;
import org.springframework.data.rest.core.annotation.RepositoryRestResource;

@RepositoryRestResource public interface ReviewerRepoistory
    extends PagingAndSortingRepository<Reviewer, Long> {
}
```

- 启动应用程序，并访问 `http://localhost:8080/authors`，将会得到如下结果

```
~ >>> http http://localhost:8080/authors
HTTP/1.1 200 OK
Content-Type: application/hal+json; charset=UTF-8
Date: Mon, 07 Dec 2015 05:10:17 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

{
    "_embedded": {
        "authors": []
    },
    "_links": {
        "profile": {
            "href": "http://localhost:8080/profile/authors"
        },
        "self": {
            "href": "http://localhost:8080/authors"
        }
    },
    "page": {
        "number": 0,
        "size": 20,
        "totalElements": 0,
        "totalPages": 0
    }
}
```

图片 1.2 访问author信息

## 分析

显然，通过继承PagingAndSortingRepository接口，比直接写Controller能提供更多的功能：分页查询和对查询结果排序。

@RepositoryRestResource注解让编程人员可以直接通过repository提供数据接口，在这个“前端负责V和C，后端负责提供数据”的时代，非常方便；并且，可以通过给该注解传入参数来改变URL。

只要在项目的classpath中包含spring-boot-starter-data-rest，同时就包含了spring-hateoas库支持，这个库可以提供ALPS元数据——一种数据格式，可以用于描述应用级别的API语义。

## 参考资料：

1. [ALPS主页](#)
2. [Spring Data Rest + Spring Security](#)

HTML



T



unity



HTML



- [Spring Boot: 定制servlet filters](#)
- [Spring Boot: 定制拦截器](#)
- [Spring Boot: 定制HTTP消息转换器](#)
- [Spring Boot: 定制PropertyEditors](#)
- [Spring Boot: 定制type Formatters](#)

## Spring Boot: 定制servlet filters

在实际的web应用程序中，经常需要在请求（request）外面增加包装用于：记录调用日志、排除有XSS威胁的字符、执行权限验证等等。除了上述提到的之外，Spring Boot自动添加了OrderedCharacterEncodingFilter和HiddenHttpMethodFilter，并且我们在自己的项目中还可以增加别的过滤器。

Spring Boot、Spring Web和Spring MVC等其他框架，都提供了很多servlet 过滤器可使用，我们需要在配置文件中定义这些过滤器为bean对象。现在假设我们的应用程序运行在一台负载均衡代理服务器后方，因此需要将代理服务器发来的请求包含的IP地址转换成真正的用户IP。Tomcat 8 提供了对应的过滤器：RemoteIpFilter。通过将RemoteFilter这个过滤器加入过滤器调用链即可使用它。

### How Do

一般在写简单的例子时，不需要单独定义配置文件，只需要将对应的bean对象定义在Application类中即可。正式的项目中一般会有单独的web配置文件，我们在项目的 com.test.bookpub （与BookpubApplication.java同级）下建立WebConfiguration.java文件，并用@Configuration注解修饰。

```
package com.test.bookpub;

import org.apache.catalina.filters.RemoteIpFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

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

通过 mvn spring-boot:run 启动项目，可以在终端中看到如下的输出信息，证明RemoteIPFilter已经添加成功。

```
2015-12-07 14:02:50,302 [INFO] 3881 --- [ost-startStop-1] o.s.b.f.config.PropertiesFactoryBean : Loading properties file from class path resource [rest-default-messages.properties]
2015-12-07 14:02:50,908 [INFO] 3881 --- [ost-startStop-1] o.s.b.c.e.ServletRegistrationBean : Mapping servlet: 'dispatcherServlet' to: [/]
2015-12-07 14:02:50,939 [INFO] 3881 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'characterEncodingFilter' to: [//*]
2015-12-07 14:02:50,940 [INFO] 3881 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'hiddenHttpMethodFilter' to: [//*]
2015-12-07 14:02:50,940 [INFO] 3881 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'httpPutFormContentFilter' to: [//*]
2015-12-07 14:02:50,940 [INFO] 3881 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'requestContextFilter' to: [//*]
2015-12-07 14:02:50,940 [INFO] 3881 --- [ost-startStop-1] o.s.b.c.embedded.FilterRegistrationBean : Mapping filter: 'remoteIpFilter' to: [//*]
2015-12-07 14:02:51,276 [INFO] 3881 --- [main] s.w.s.m.m.a.RequestMappingHandlerAdapter : Looking for @ControllerAdvice: org.springframework.boot.context.embedded.AnnotationCon
figEmbeddedWebApplicationContext@59a7f60a; startup date: Mon Dec 07 14:02:44 CST 2015; root of context hierarchy
```

图片 2.1 RemoteIPFilter

## 分析

项目的主类——BookPubApplication，可以看到由@SpringBootApplication注解修饰，这包含了@ComponentScan、@Configuration和@EnableAutoConfiguration注解。在[Spring Boot的自动配置、Command-line Runner](#)一文中曾对这个三个注解做详细解释，@ComponentScan让Spring Boot扫描到WebConfiguration类并把它加入到程序上下文中，因此，我们在WebApplication中定义的Bean就跟在BookPubApplication中定义一样。

方法 `@Bean public RemoteIpFilter remoteIpFilter() { ... }` 返回一个RemoteIPFilter类的spring bean。当Spring Boot监测到有javax.servlet.Filter的bean时就会自动加入过滤器调用链。从上图中还可以看到，该Spring Boot项目一次加入了这几个过滤器：characterEncodingFilter（用于处理编码问题）、hiddenHttpMethodFilter（隐藏HTTP函数）、httpPutFormContentFilter、requestContextFilter（请求上下文），以及我们刚才自定义的RemoteIPFilter。

所有过滤器的调用顺序跟添加的顺序相反，过滤器的实现是责任链模式，具体的原理分析可以参考：[责任链模式](#)

## Spring Boot：定制拦截器

---

Servlet 过滤器属于Servlet API，和Spring关系不大。除了使用过滤器包装web请求，Spring MVC还提供*Handler Interceptor*（拦截器）工具。根据文档，HandlerInterceptor的功能跟过滤器类似，但拦截器提供更精细的控制能力：在request被响应之前、request被响应之后、视图渲染之前以及request全部结束之后。我们不能通过拦截器修改request内容，但是可以通过抛出异常（或者返回false）来暂停request的执行。

Spring MVC中常用的拦截器有：*LocaleChangeInterceptor*（用于国际化配置）和*ThemeChangeInterceptor*。我们也可以增加自己定义的拦截器，可以参考这篇文章中提供的[demo](#)

### How Do

添加拦截器不仅是在WebConfiguration中定义bean，Spring Boot提供了基础类WebMvcConfigurerAdapter，我们项目中的WebConfiguration类需要继承这个类。

1. 继承WebMvcConfigurerAdapter；
2. 为LocaleChangeInterceptor添加@Bean定义，这仅仅是定义了一个interceptor spring bean，但是Spring boot不会自动将它加入到调用链中。
3. 拦截器需要手动加入调用链。

修改后完整的WebConfiguration代码如下：

```
package com.test.bookpub;

import org.apache.catalina.filters.RemoteIpFilter;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.web.servlet.config.annotation.InterceptorRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurerAdapter;
import org.springframework.web.servlet.i18n.LocaleChangeInterceptor;

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

    @Bean    public LocaleChangeInterceptor localeChangeInterceptor() {
        return new LocaleChangeInterceptor();
    }
}
```

```
}

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

}
```

使用 `mvn spring-boot:run` 运行程序，然后通过 `httpie` 访问 `http://localhost:8080/books?locale=foo`，在终端看到如下错误信息。

```
Servlet.service() for servlet [dispatcherServlet] in context with path []
threw exception [Request processing failed; nested exception is
java.lang.UnsupportedOperationException: Cannot change HTTP accept
header - use a different locale resolution strategy] with root cause
```

PS：这里发生错误并不是因为我们输入的 `locale` 是错误的，而是因为默认的 `locale` 修改策略不允许来自浏览器的请求修改。发生这样的错误说明我们之前定义的拦截器起作用了。

## 分析

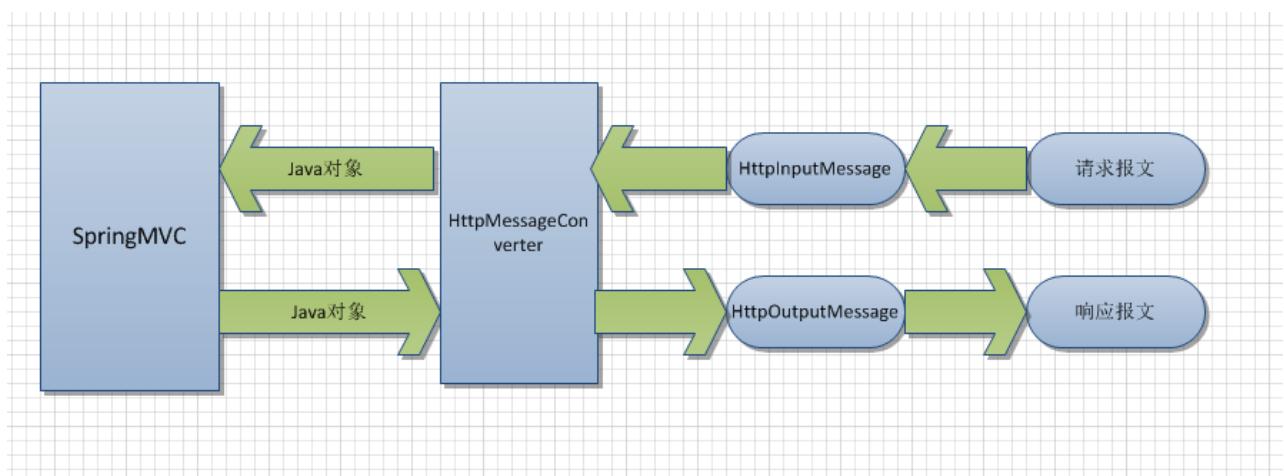
在我们的示例项目中，覆盖并重写了 `addInterceptors(InterceptorRegistry registry)` 方法，这是典型的回调函数——利用该函数的参数 `registry` 来添加自定义的拦截器。

在 Spring Boot 的自动配置阶段，Spring Boot 会扫描所有 `WebMvcConfigurer` 的实例，并顺序调用其中的回调函数，这表示：如果我们想对配置信息做逻辑上的隔离，可以在 Spring Boot 项目中定义多个 `WebMvcConfigurer` 的实例。

## Spring Boot：定制HTTP消息转换器

在构建RESTful数据服务过程中，我们定义了controller、repositories，并用一些注解修饰它们，但是到现在为止我们还没执行过对象的转换——将java实体对象转换成HTTP的数据输出流。Spring Boot底层通过`HttpMessageConverters`依靠Jackson库将Java实体类输出为JSON格式。当有多个转换器可用时，根据消息对象类型和需要的内容类型选择最适合的转换器使用。

在[SpringMVC源码剖析（五）-消息转换器HttpMessageConverter](#)一文中，有一张图可以很清楚得表示消息转换器的位置。



图片 2.2 消息转换器的位置

消息转换器的目标是：HTTP输入请求格式向Java对象的转换；Java对象向HTTP输出请求的转换。有的消息转换器只支持多个数据类型，有的只支持多个输出格式，还有的两者兼备。例如：`MappingJackson2HttpMessageConverter`可以将Java对象转换为`application/json`，而`ProtobufHttpMessageConverter`仅支持`com.google.protobuf.Message`类型的输入，但是可以输出`application/json`、`application/xml`、`text/plain`和`application/x-protobuf`这么多格式。

### How Do

在项目中有三种办法配置消息转换器，主要区别是可定制性和易用度的衡量。 1. 在`WebConfiguration`类中加入`@Bean`定义

```
@Bean public ByteArrayHttpMessageConverter byteArrayHttpMessageConverter() {
    return new ByteArrayHttpMessageConverter();
}
```

- 重写 (override) `configureMessageConverters`方法，扩展现有的消息转换器链表；

```
@Override public
void configureMessageConverters(List<HttpMessageConverter<?>> converters) {
    converters.add(new ByteArrayHttpMessageConverter());
}
```

- 更多的控制，可以重写`extendMessageConverters`方法，首先清空转换器列表，再加入自定义的转换器。

```
@Override public
void extendMessageConverters(List<HttpMessageConverter<?>> converters) {
    converters.clear();
    converters.add(new ByteArrayHttpMessageConverter());
}
```

## 分析

Spring提供了多种方法完成同样的任务，选择哪个取决于我们更侧重便捷性还是更侧重可定制性。

上述提到的三种方法各有什么不同呢？

通过`@Bean`定义`HttpMessageConverter`是向项目中添加消息转换器最简便的办法，这类似于之前提到的添加`Servlet Filters`。如果Spring扫描到`HttpMessageConverter`类型的bean，就会将它自动添加到调用链中。推荐让项目中的`WebConfiguration`继承自`WebMvcConfigurerAdapter`。

通过重写`configureMessageConverters`方法添加自定义的转换器很方便，但有一个弱点：如果项目中存在多个`WebMvcConfigurers`的实例（我们自己定义的，或者Spring Boot默认提供的），不能确保重写后的`configureMessageConverters`方法按照固定顺序执行。

如果需要更精细的控制：清除其他消息转换器或者清楚重复的转换器，可以通过重写`extendMessageConverters`完成，仍然有这种可能：别的`WebMvcConfigurer`实例也可以重写这个方法，但是这种几率非常小。

## Spring Boot：定制PropertyEditors

---

在[Spring Boot：定制HTTP消息转换器](#)一文中我们学习了如何配置消息转换器用于HTTP请求和响应数据，实际上，在一次请求的完成过程中还发生了其他的转换，我们这次关注将参数转换成多种类型的对象，如：字符串转换成Date对象或字符串转换成Integer对象。

在编写控制器中的action方法时，Spring允许我们使用具体的数据类型定义函数签名，这是通过*PropertyEditor*实现的。*PropertyEditor*本来是JDK提供的API，用于将文本值转换成给定的类型，结果Spring的开发人员发现它恰好满足Spring的需求——将URL参数转换成函数的参数类型。

针对常用的类型（Boolean、Currency和Class），Spring MVC已经提供了很多*PropertyEditor*实现。假设我们需要创建一个Isbn类并用它作为函数中的参数。

### How Do

- 考虑到*PropertyEditor*属于工具范畴，选择在项目根目录下增加一个包——utils。在这个包下定义Isbn类和IsbnEditor类，各自代码如下： Isbn类：

```
package com.test.bookpub.utils;

public class Isbn {
    private String isbn;

    public Isbn(String isbn) {
        this.isbn = isbn;
    }

    public String getIsbn() {
        return isbn;
    }
}
```

- IsbnEditor类，继承*PropertyEditorSupport*类，*setAsText*完成字符串到具体对象类型的转换，*getAsText*完成具体对象类型到字符串的转换。

```
package com.test.bookpub.utils;
import org.springframework.util.StringUtils;
import java.beans.PropertyEditorSupport;

public class IsbnEditor extends PropertyEditorSupport {
    @Override
```

```

public void setAsText(String text) throws IllegalArgumentException {
    if (StringUtils.hasText(text)) {
        setValue(new Isbn(text.trim()));
    } else {
        setValue(null);
    }
}

@Override    public String getAsText() {
    Isbn isbn = (Isbn) getValue();
    if (isbn != null) {
        return isbn.getIsbn();
    } else {
        return "";
    }
}
}

```

- 在BookController中增加initBinder函数，通过@InitBinder注解修饰，则可以针对每个web请求创建一个editor实例。

```

@InitBinder
public
void initBinder(WebDataBinder binder) {
    binder.registerCustomEditor(Isbn.class, new IsbnEditor());
}

```

- 修改BookController中对应的函数

```

@RequestMapping(value = "/{isbn}", method = RequestMethod.GET)
public Map<String, Object> getBook(@PathVariable Isbn isbn) {
    Book book = bookRepository.findBookByIsbn(isbn.getIsbn());
    Map<String, Object> response = new LinkedHashMap<>();
    response.put("message", "get book with isbn(" + isbn.getIsbn() + ")");
    response.put("book", book);    return response;
}

```

运行程序，通过Httpie访问 `http localhost:8080/books/9781-1234-1111`，可以得到正常结果，跟之前用String表示isbn时没什么不同，说明我们编写的IsbnEditor已经起作用了。

## 分析

Spring提供了很多默认的editor，我们也可以通过继承PropertyEditorSupport实现自己定制化的editor。

由于PropertyEditor是非线程安全的。通过@InitBinder注解修饰的initBinder函数，会为每个web请求初始化一个editor实例，并通过WebDataBinder对象注册。

## Spring Boot：定制type Formatters

---

考虑到*PropertyEditor*的无状态和非线程安全特性，Spring 3增加了一个*Formatter*接口来替代它。*Formatters*提供和*PropertyEditor*类似的功能，但是提供线程安全特性，并且专注于完成字符串和对象类型的互相转换。

假设在我们的程序中，需要根据一本书的ISBN字符串得到对应的book对象。通过这个类型格式化工具，我们可以在控制器的方法签名中定义Book参数，而URL参数只需要包含ISBN号和数据库ID。

### How Do

- 首先在项目根目录下创建*formatters*包
- 然后创建BookFormatter，它实现了Formatter接口，实现两个函数：parse用于将字符串ISBN转换成book对象；print用于将book对象转换成该book对应的ISBN字符串。

```
package com.test.bookpub.formatters;

import com.test.bookpub.domain.Book;
import com.test.bookpub.repository.BookRepository;
import org.springframework.format.Formatter;
import java.text.ParseException;
import java.util.Locale;

public class BookFormatter implements Formatter<Book> {
    private BookRepository repository;

    public BookFormatter(BookRepository repository) {
        this.repository = repository;
    }

    @Override
    public Book parse(String bookIdentifier, Locale locale) throws ParseException {
        Book book = repository.findBookByIsbn(bookIdentifier);
        return book != null ? book : repository.findOne(Long.valueOf(bookIdentifier));
    }

    @Override
    public String print(Book book, Locale locale) {
        return book.getIsbn();
    }
}
```

- 在WebConfiguration中添加我们定义的formatter，重写（@Override修饰）`addFormatter(FormatterRegistry registry)`函数。

```
@Autowired
private BookRepository bookRepository;
@Override
public void addFormatters(FormatterRegistry registry) {
    registry.addFormatter(new BookFormatter(bookRepository));
}
```

- 最后，需要在BookController中新加一个函数`getReviewers`，根据一本书的ISBN号获取该书的审阅人。

```
@RequestMapping(value = "/{isbn}/reviewers", method = RequestMethod.GET)
public List<Reviewer> getReviewers(@PathVariable("isbn") Book book) {
    return book.getReviewers();
}
```

- 通过`mvn spring-boot:run`运行程序
- 通过httpie访问URL——`http://localhost:8080/books/9781-1234-1111/reviewers`，得到的结果如下：

```
HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Date: Tue, 08 Dec 2015 08:15:31 GMT
Server: Apache-Coyote/1.1
Transfer-Encoding: chunked

[]
```

## 分析

`Formatter`工具的目标是提供跟`PropertyEditor`类似的功能。通过`FormatterRegistry`将我们自己的`formatter`注册到系统中，然后Spring会自动完成文本表示的book和book实体对象之间的互相转换。由于`Formatter`是无状态的，因此不需要为每个请求都执行注册`formatter`的动作。

**使用建议：**如果需要通用类型的转换——例如`String`或`Boolean`，最好使用`PropertyEditor`完成，因为这种需求可能不是全局需要的，只是某个Controller的定制功能需求。

我们在WebConfiguration中引入（`@Autowired`）了`BookRepository`（需要用它创建`BookFormatter`实例），Spring给配置文件提供了使用其他bean对象的能力。Spring本身会确保`BookRepository`先创建，然后在`WebConfiguration`类的创建过程中引入。

HTML



T



unity



HTML



- Spring Boot: 定制URL匹配规则
- Spring Boot: 定制static path mappings
- 通过EmbeddedServletContainerCustomizer接口调优Tomcat
- 选择Spring Boot项目的内嵌容器
- 让你的Spring Boot工程支持HTTP和HTTPS

## Spring Boot：定制URL匹配规则

构建web应用程序时，并不是所有的URL请求都遵循默认的规则。有时，我们希望RESTful URL匹配的时候包含定界符“.”，这种情况在Spring中可以称之为“定界符定义的格式”；有时，我们希望识别斜杠的存在。Spring提供了接口供开发人员按照需求定制。

在之前的几篇文章中，可以通过WebConfiguration类来定制程序中的过滤器、格式化工具等等，同样得，也可以在这个类中用类似的办法配置“路径匹配规则”。

假设ISBN格式允许通过定界符“.”分割图书编号和修订号，形如[isbn-number].[revision]

### How Do

- 在WebConfiguration类中添加对应的配置，代码如下：

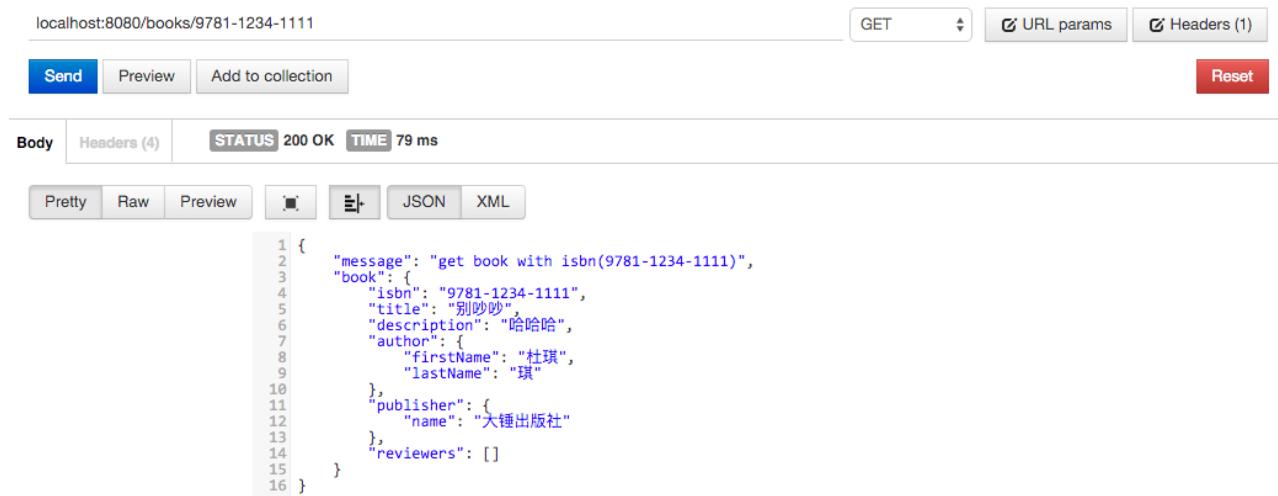
```
@Override
public void configurePathMatch(PathMatchConfigurer configurer) {
    configurer.setUseSuffixPatternMatch(false);
    configurer.setUseTrailingSlashMatch(true);
}
```

- 通过mvn spring-boot:run启动应用程序
- 访问http://localhost:8080/books/9781-1234-1111.1



图片 3.1 在路径匹配时，不使用后缀模式匹配（.\*）

- 访问http://localhost:8080/books/9781-1234-1111



图片 3.2 使用正确的URL访问的结果

## 分析

`configurePathMatch(PathMatchConfigurer configurer)`函数让开发人员可以根据需求定制URL路径的匹配规则。

- `configurer.setUseSuffixPatternMatch(false)`表示设计人员希望系统对外暴露的URL不会识别和匹配.\*后缀。在这个例子中，就意味着Spring会将9781-1234-1111.1当做一个{isbn}参数传给BookController。- `configurer.setUseTrailingSlashMatch(true)`表示系统不区分URL的最后一个字符是否是斜杠/。在这个例子中，就意味着 `http://localhost:8080/books/9781-1234-1111` 和 `http://localhost:8080/books/9781-1234-1111/` 含义相同。

如果需要定制path匹配发生的过程，可以提供自己定制的`PathMatcher`和`UrlPathHelper`，但是这种需求并不常见。

## Spring Boot: 定制static path mappings

在[Spring Boot: 定制URL匹配规则](#)一文中我们展示了如何调整URL请求匹配到对应的控制器方法的规则。类似得，也可以控制应用程序对静态文件（前提是被打包进部署包）的处理。

假设我们需要通过URL `http://localhost:8080/internal/application.properties` 对外暴露当前程序的配置。

### How Do

- 在WebConfiguration类中添加相应的配置，代码如下：

```
@Override
public void addResourceHandlers(ResourceHandlerRegistry registry) {
    registry.addResourceHandler("/internal/**").
        addResourceLocations("/classpath:/");
}
```

- 通过 `mvn spring-boot:run` 启动应用程序
- 通过postman访问 `http://localhost:8080/internal/application.properties` 就得到下列的结果

The screenshot shows a Postman interface with the following details:

- Request URL:** `http://localhost:8080/internal/application.properties`
- Method:** GET
- Headers:** (1)
- Body:** (Pretty, Raw, Preview, JSON, XML)
- Status:** 200 OK
- Time:** 44 ms

The response body (Pretty) is:

```

1 spring.datasource.driverClassName=com.mysql.jdbc.Driver
2 #spring.datasource.url=jdbc:mysql://192.168.99.100:3306/springbootcookbook
3 spring.datasource.url=jdbc:mysql://localhost:3306/springbootcookbook
4 spring.datasource.username=root
5 spring.datasource.password=root
6
7 # Specify the DBMS
8 spring.jpa.database = MYSQL
9 # Show or not log for each sql query
10 spring.jpa.show-sql = true
11 # Hibernate ddl auto (create, create-drop, update)
12 spring.jpa.hibernate.ddl-auto = create-drop
13 # Naming strategy
14 spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
15
16 # stripped before adding them to the entity manager)
17 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

```

图片 3.3 通过配置项对外暴露程序的配置信息

## 分析

通过`*addResourceHandlers(ResourceHandlerRegistry registry)` \*方法可以为应用程序中位于classpath路径下或文件系统下的静态资源配置对应的URL，供其他人通过浏览器访问。在这个例子中，我们规定所有以“/internal”开头的URL请求会在classpath:/目录下查找信息。

- `registry.addResourceHandler("/internal/")`方法添加一个资源处理器，用于注册程序中的静态资源，该函数返回一个`ResourceHandlerRegistration`对象，这个对象可以进一步配置。`/internal/`字符串是一个路径模式串，`PathMatcher`接口用它匹配对应的URL请求，这里默认使用`AntPathMatcher`进行匹配。
- 由上个方法返回的`ResourceHandlerRegistration`实例调用`addResourceLocations("/classpath:/")`方法来规定从哪个目录下加载资源文件。这个目录路径或者是有效的文件系统路径，或者是classpath路径。

PS：通过`setCachePeriod(Integer cachePeriod)`方法可以设置资源处理器的缓存周期——每隔cachePeriod秒就缓存一次。

## 通过EmbeddedServletContainerCustomizer接口调优Tomcat

通过在application.properties设置对应的key-value对，可以配置Spring Boot应用程序的很多特性，例如POST、SSL、MySQL等等。如果需要更加复杂的调优，则可以利用Spring Boot提供的EmbeddedServletContainerCustomizer接口通过编程方式和修改配置信息。

尽管可以通过application.properties设置server.session-timeout属性来配置服务器的会话超时时间，这里我们用EmbeddedServletContainerCustomizer接口修改，来说明该接口的用法。

### How Do

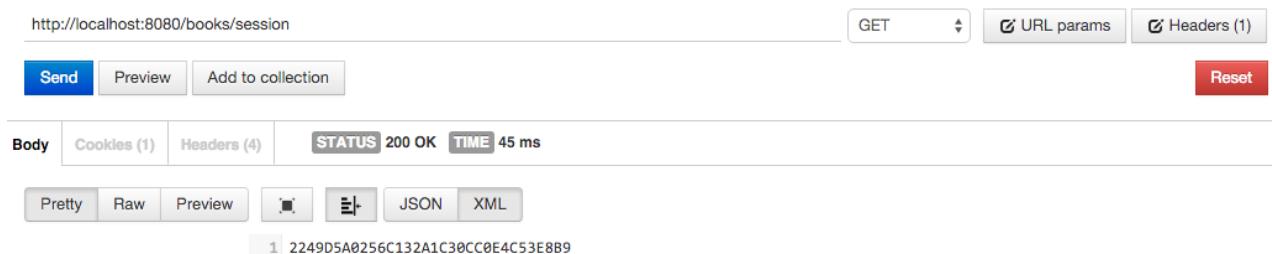
- 假设我们希望设置会话的超时时间为1分钟。在WebConfiguration类中增加EmbeddedServletContainerCustomizer类型的spring bean，代码如下：

```
@Bean
public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer() {
    return new EmbeddedServletContainerCustomizer() {
        @Override
        public void customize(ConfigurableEmbeddedServletContainer container) {
            container.setSessionTimeout(1, TimeUnit.MINUTES);
        }
    };
}
```

- 在BookController中添加一个getSessionId(HttpServletRequest request)函数，直接返回request.getSession().getId()。

```
@RequestMapping(value = "/session", method = RequestMethod.GET)
public String getSessionId(HttpServletRequest request) {
    return request.getSession().getId();
}
```

- 通过mvn spring-boot:run启动应用
- 通过postman访问http://localhost:8080/books/session，得到的结果如下



图片 3.4 获取session

1分钟以后再次调用这个接口，则发现返回的session id已经改变。

## 分析

除了可以使用上面这个写法，对于使用Java 8的开发人员，还可以使用lambda表达式处理，就不需要创建一个`EmbeddedServletContainerCustomizer`实例了。代码如下：

```
//对于Java 8来说可以用lambda表达式,而不需要创建该接口的一个实例.
@Bean
public EmbeddedServletContainerCustomizer embeddedServletContainerCustomizer() {
    return (ConfigurableEmbeddedServletContainer container) -> {
        container.setSessionTimeout(1, TimeUnit.MINUTES);
    };
}
```

在程序启动阶段，Spring Boot检测到`custoimer`实例的存在，然后就会调用`invoke(...)`方法，并向内传递一个`se rvlet`对象的实例。在我们这个例子中，实际上传入的是`TomcatEmbeddedServletContainerFactory`容器对象，但是如果使用`Jutty`或者`Undertow`容器，就会用对应的容器对象。

## 选择Spring Boot项目的内嵌容器

Spring Boot工程的默认web容器是Tomcat，但是开发人员可以根据需要修改，例如使用Jetty或者Undertow，Spring Boot提供了对应的starters。

### How Do

- 在pom文件中排除tomcat的starter

```
<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>
```

- 增加Jetty依赖

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

- 通过 `mvn spring-boot:run` 命令启动，可以看到Jetty已经启动。

```
2015-12-08 22:09:46.436 [main] DEBUG o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started
2015-12-08 22:09:46.465 [main] DEBUG o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 28 ms
2015-12-08 22:09:46.511 [main] DEBUG o.e.jetty.server.ServerConnector : Started ServerConnector@32a0aa78{HTTP/1.1}{0.0.0.0:8080}
2015-12-08 22:09:46.515 [main] DEBUG s.b.c.e.j.JettyEmbeddedServletContainer : Jetty started on port(s) 8080 (http/1.1)
Hibernate: select count(*) as col_0_0_ from book book0_
2015-12-08 22:09:46.965 [main] DEBUG com.test.bookpub.StartupRunner : Number of books: 0
2015-12-08 22:09:46.967 [main] DEBUG com.test.bookpub.BookPubApplication : Started BookPubApplication in 15.009 seconds (JVM running for 21.888)
Hibernate: select count(*) as col_0_0_ from book book0_
```

图片 3.5 Jetty容器启动

PS：如果您使用的gradle，则可以参考官方文档的做法——[Use Jetty instead of Tomcat](#)

## 分析

支持上述切换的原因是Spring Boot的自动配置。我首先在`spring-boot-starter-web`依赖中排除Tomcat依赖，免得它跟Jetty形成依赖冲突。Spring Boot根据在classpath下扫描到的容器类的类型决定使用哪个web容器。

在IDEA中查看`EmbeddedServletContainerAutoConfiguration`类的内部结构，可以看到`@ConditionalOnClass({Servlet.class, Server.class, Loader.class, WebApplicationContext.class})`这样的条件匹配注解，如果在Jetty的Jar包中可以找到上述三个类的实例，则决定使用jetty容器。

```
@Configuration
@ConditionalOnClass({Servlet.class, Server.class, Loader.class, WebApplicationContext.class})
@ConditionalOnMissingBean(value = {EmbeddedServletContainerFactory.class}, search = SearchStrategy.CURRENT)
public static class EmbeddedJetty {
    public EmbeddedJetty() {
    }
    @Bean
    public JettyEmbeddedServletContainerFactory jettyEmbeddedServletContainerFactory() {
        // 返回容器工厂实例，用于构造容器实例
        return new JettyEmbeddedServletContainerFactory();
    }
}
```

同样得，可以看到对Tomcat也存在类似的判断和使用代码：

```
@Configuration
@ConditionalOnClass({Servlet.class, Tomcat.class})
@ConditionalOnMissingBean(value = {EmbeddedServletContainerFactory.class}, search = SearchStrategy.CURRENT)
public static class EmbeddedTomcat {
    public EmbeddedTomcat() {
    }
    @Bean
    public TomcatEmbeddedServletContainerFactory tomcatEmbeddedServletContainerFactory() {
        return new TomcatEmbeddedServletContainerFactory();
    }
}
```

## 让你的Spring Boot工程支持HTTP和HTTPS

---

如今，企业级应用程序的常见场景是同时支持HTTP和HTTPS两种协议，这篇文章考虑如何让Spring Boot应用程序同时支持HTTP和HTTPS两种协议。

### 准备

为了使用HTTPS连接器，需要生成一份Certificate keystore，用于加密和机密浏览器的SSL沟通。

如果你使用Unix或者Mac OS，可以通过下列命令：`keytool -genkey -alias tomcat -keyalg RSA`，在生成过程中可能需要你填入一些自己的信息，例如我的机器上反馈如下：

```

-validity      -- valid days
~ >>> keytool -genkey -alias tomcat -keyalg RSA
输入密钥库口令：
再次输入新口令：
您的名字与姓氏是什么？
[Unknown]: duqi
您的组织单位名称是什么？
[Unknown]: org.test
您的组织名称是什么？
[Unknown]: test
您所在的城市或区域名称是什么？
[Unknown]: hz
您所在的省/市/自治区名称是什么？
[Unknown]: zj
该单位的双字母国家/地区代码是什么？
[Unknown]: china
CN=duqi, OU=org.test, O=test, L=hz, ST=zj, C=china是否正确？
[否]: y

输入 <tomcat> 的密钥口令
(如果和密钥库口令相同，按回车):
再次输入新口令：
它们不匹配。请重试
输入 <tomcat> 的密钥口令
(如果和密钥库口令相同，按回车):
再次输入新口令：
~ >>> ls .keystore
.keystore
~ >>> ls -al .keystore

```

图片 3.6 生成keystore

可以看出，执行完上述命令后在home目录下多了一个新的.keystore文件。

### How Do

- 首先在resources目录下新建一个配置文件*tomcat.https.properties*，用于存放HTTPS的配置信息；

```

custom.tomcat.https.port=8443
custom.tomcat.https.secure=true
custom.tomcat.https.scheme=https

```

```
custom.tomcat.https.ssl=true
custom.tomcat.https.keystore=${user.home}/.keystore
custom.tomcat.https.keystore-password=changeit
```

- 然后在WebConfiguration类中创建一个静态类*TomcatSslConnectorProperties*;

```
@ConfigurationProperties(prefix = "custom.tomcat.https")
public static class TomcatSslConnectorProperties {
    private Integer port;
    private Boolean ssl = true;
    private Boolean secure = true;
    private String scheme = "https";
    private File keystore;
    private String keystorePassword;
    //这里为了节省空间，省略了getters和setters，读者在实践的时候要加上

    public void configureConnector(Connector connector) {
        if (port != null) {
            connector.setPort(port);
        }
        if (secure != null) {
            connector.setSecure(secure);
        }
        if (scheme != null) {
            connector.setScheme(scheme);
        }
        if (ssl != null) {
            connector.setProperty("SSLEnabled", ssl.toString());
        }
        if (keystore != null && keystore.exists()) {
            connector.setProperty("keystoreFile", keystore.getAbsolutePath());
            connector.setProperty("keystorePassword", keystorePassword);
        }
    }
}
```

- 通过注解加载*tomcat.https.properties*配置文件，并与*TomcatSslConnectorProperties*绑定，用注解修饰WebConfiguration类;

```
@Configuration
@PropertySource("classpath:/tomcat.https.properties")
@EnableConfigurationProperties(WebConfiguration.TomcatSslConnectorProperties.class)
public class WebConfiguration extends WebMvcConfigurerAdapter {...}
```

- 在WebConfiguration类中创建`EmbeddedServletContainerFactory`类型的Spring bean，并用它添加之前创建的HTTPS连接器。

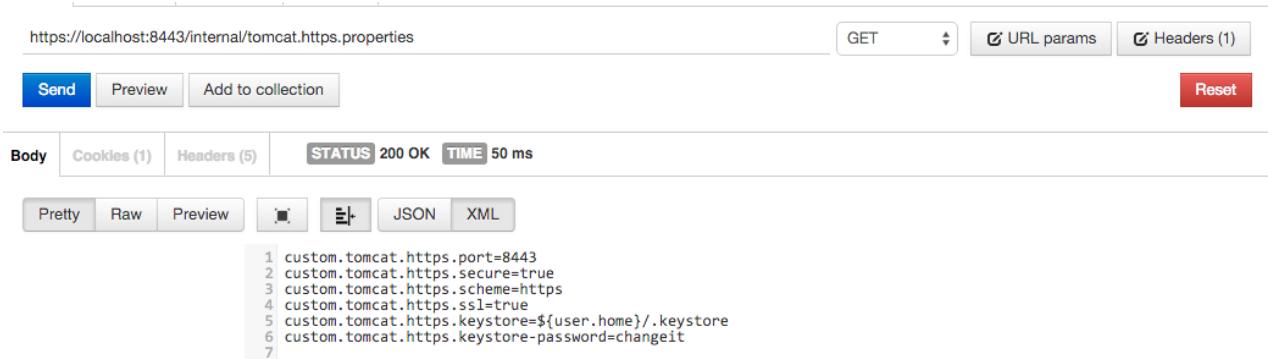
```

@Bean
public EmbeddedServletContainerFactory servletContainer(TomcatSslConnectorProperties properties) {
    TomcatEmbeddedServletContainerFactory tomcat = new TomcatEmbeddedServletContainerFactory();
    tomcat.addAdditionalTomcatConnectors(createSslConnector(properties));
    return tomcat;
}

private Connector createSslConnector(TomcatSslConnectorProperties properties) {
    Connector connector = new Connector();
    properties.configureConnector(connector);
    return connector;
}

```

- 通过`mvn spring-boot:run`启动应用程序；
- 在浏览器中访问URL `https://localhost:8443/internal/tomcat.https.properties`



图片 3.7 支持HTTPS协议

- 在浏览器中访问URL `http://localhost:8080/internal/application.properties`

```

1 spring.datasource.driverClassName=com.mysql.jdbc.Driver
2 #spring.datasource.url=jdbc:mysql://192.168.99.100:3306/springbootcookbook
3 spring.datasource.url=jdbc:mysql://localhost:3306/springbootcookbook
4 spring.datasource.username=root
5 spring.datasource.password=root
6
7 # Specify the DBMS
8 spring.jpa.database = MYSQL
9 # Show or not log for each sql query
10 spring.jpa.show-sql = true
11 # Hibernate ddl auto (create, create-drop, update)
12 spring.jpa.hibernate.ddl-auto = create-drop
13 # Naming strategy
14 spring.jpa.hibernate.naming-strategy = org.hibernate.cfg.ImprovedNamingStrategy
15 # stripped before adding them to the entity manager)
16 spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect
17
18
19

```

图片 3.8 同时支持HTTP协议

## 分析

根据之前的文章和官方文档，Spring Boot已经对外开放了很多服务器配置，这些配置信息通过Spring Boot内部的*ServerProperties*类完成绑定，若要参考Spring Boot的通用配置项，请[点击这里](#)

Spring Boot不支持通过application.properties同时配置HTTP连接器和HTTPS连接器。在[官方文档70.8](#)中提到一种方法，是将属性值硬编码在程序中。

因此我们这里新建一个配置文件*tomcat.https.properties*来实现，但是这并不符合“Spring Boot风格”，后续有可能应该会支持“通过application.properties同时配置HTTP连接器和HTTPS连接器”。我添加的*TomcatSslConnectorProperties*是模仿Spring Boot中的*ServerProperties*的使用机制实现的，这里使用了自定义的属性前缀*custom.tomcat*而没有用现有的*server.*前缀，因为*ServerProperties*禁止在其他的配置文件中使用该命名空间。

*@ConfigurationProperties(prefix = "custom.tomcat.https")*这个注解会让Spring Boot自动将*custom.tomcat.https*开头的属性绑定到*TomcatSslConnectorProperties*这个类的成员上（确保该类的getters和setters存在）。值得一提的是，在绑定过程中Spring Boot会自动将属性值转换成合适的数据类型，例如*custom.tomcat.https.keystore*的值会自动绑定到File对象keystore上。

使用*@PropertySource("classpath:/tomcat.https.properties")*来让Spring Boot加载*tomcat.https.properties*文件中的属性。

使用*@EnableConfigurationProperties(WebConfiguration.TomcatSslConnectorProperties.class)*让Spring Boot自动创建一个属性对象，包含上述通过*@PropertySource*导入的属性。

在属性值导入内存，并构建好*TomcatSslConnectorProperties*实例后，需要创建一个*EmbeddedServletContainerFactory*类型的Spring bean，用于创建*EmbeddedServletContainer*。

通过*createSslConnector*方法可以构建一个包含了我们指定的属性值的连接器，然后通过*tomcat.addAdditionalTomcatConnectors(createSslConnector(properties))*;设置tomcat容器的HTTPS连接器。

## 参考资料

1. 配置SSL

HTML



T



unity



HTML



- [了解Spring Boot的自动配置](#)
- [Spring Boot: 定制自己的starter](#)
- [配置是否初始化Bean的方法](#)
- [通过@Enable\\*注解触发Spring Boot配置](#)

## 了解Spring Boot的自动配置

Spring Boot的自动配置给开发者带来了很大的便利，当开发人员在pom文件中添加starter依赖后，maven或者gradle会自动下载很多jar包到classpath中。当Spring Boot检测到特定类的存在，就会针对这个应用做一定的配置，自动创建和织入需要的spring bean到程序上下文中。

在之前的文章中，我们只是在pom文件中增加各种starter的依赖，例如：`spring-boot-starter-data-jpa`, `spring-boot-starter-web`, `spring-boot-starter-data-test`等等。接下来将在之前的工程的基础上，观察在程序的引导启动过程中，Spring Boot通过自动配置机制帮我们做了哪些工作。

### How Do

- Spring Boot启动时将自动配置的信息通过DEBUG级别的日志打印到控制台。可以通过设置环境变量（DEBUG）或者程序属性（--debug）设置程序的日志输出级别。
- 在项目目录下运行 `DEBUG=true mvn spring-boot:run` 启动应用程序；
- 在后台可以看到DEBUG级别的日志输出，在启动日志的最后，可以看到类似AUTO-CONFIGURATION REPORT的样子。

```

2015-12-09 19:41:56.556 DEBUG 16283 --- [           main] AutoConfigurationReportLoggingInitializer : 

-----
AUTO-CONFIGURATION REPORT
-----

Positive matches:
-----
AopAutoConfiguration matched
- @ConditionalOnClass classes found: org.springframework.context.annotation.EnableAspectJAutoProxy,org.aspectj.lang.annotation.Aspect,org.aspectj.lang.reflect.Advice (OnClassCondition)
- matched (OnPropertyCondition)

AopAutoConfiguration.JdkDynamicAutoProxyConfiguration matched
- matched (OnPropertyCondition)

DataSourceAutoConfiguration matched
- @ConditionalOnClass classes found: javax.sql.DataSource,org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType (OnClassCondition)

DataSourceAutoConfiguration.DataSourceInitializerConfiguration matched
- @ConditionalOnMissingBean (types: org.springframework.boot.autoconfigure.jdbc.DataSourceInitializer; SearchStrategy: all) found no beans (OnBeanCondition)

```

图片 4.1 Positive matches

```

-----
Negative matches:
-----
ActiveMQAutoConfiguration did not match
- required @ConditionalOnClass classes not found: javax.jms.ConnectionFactory,org.apache.activemq.ActiveMQConnectionFactory (OnClassCondition)

AopAutoConfiguration.CglibAutoProxyConfiguration did not match
- @ConditionalOnProperty missing required properties spring.aop.proxy-target-class (OnPropertyCondition)

ArtemisAutoConfiguration did not match
- required @ConditionalOnClass classes not found: javax.jms.ConnectionFactory,org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory (OnClassCondition)

AtomikosJtaConfiguration did not match
- required @ConditionalOnClass classes not found: com.atomikos.icatch.jta.UserTransactionManager (OnClassCondition)

```

图片 4.2 Negative matches

## 分析

可以看到，后台输出的自动配置信息特别多，好几页屏幕，没办法一一分析，在这里选择一个positive match和negative match进行分析。

Spring Boot通过配置信息指出：特定配置项被选中的原因、列出匹配到对应类的配置项（positive match）、不包括某个配置项的原因（negative match）。现在以`DataSourceAutoConfiguration`举例说明：

- `@ConditionalOnClass` 表示对应的类在classpath目录下存在时，才会去解析对应的配置文件，对于`DataSourceAutoConfiguration`来说就是指：只有`javax.sql.DataSource`和`org.springframework.jdbc.datasource.embedded.EmbeddedDatabaseType`类都能存在时，就会配置对应的数据库资源。
- `@ConditionalOnMissingClass` 表示对应的类在classpath目录下找不到。
- `OnClassCondition`用于表示匹配的类型（positive or negative）

`OnClassCondition`是最普遍的浏览探测条件，除此之外，Spring Boot也使用别的探测条件，如：`OnBeanCondition`用于检测指定bean实例存在与否、`OnPropertyCondition`用于检查指定属性是否存在等等。

符合`negative match`代表一些配置类（`xxxConfiguration`之类的），它们虽然存在于classpath目录，但是修饰它们的注解中依赖的其他类不存在。导入如果在pom文件中导入`spring-boot-autoconfigure`包，则`GsonAutoConfiguration`就会出现在classpath目录下，但是该配置类被`@ConditionalOnClass(Gson.class)`修饰，而`com.google.gson.Gson`类不在classpath目录。

```

@Configuration
@ConditionalOnClass({Gson.class})
public class GsonAutoConfiguration {
    public GsonAutoConfiguration() {
    }

    @Bean
    @ConditionalOnMissingBean
    public Gson gson() {
        return new Gson();
    }
}

```

## 总结

- `@ConditionalOnClass`: 该注解的参数对应的类必须存在，否则不解析该注解修饰的配置类；

- `@ConditionalOnMissingBean`: 该注解表示, 如果存在它修饰的类的bean, 则不需要再创建这个bean; 可以给该注解传入参数例如`@ConditionOnMissingBean(name = "example")`, 这个表示如果name为“example”的bean存在, 这该注解修饰的代码块不执行。

## 参考资料

1. [Spring Boot实战：自动配置原理分析](#)

## Spring Boot：定制自己的starter

在学习Spring Boot的过程中，接触最多的就是starter。可以认为starter是一种服务——使得使用某个功能的开发者不需要关注各种依赖库的处理，不需要具体的配置信息，由Spring Boot自动通过classpath路径下的类发现需要的Bean，并织入bean。举个例子，`spring-boot-starter-jdbc`这个starter的存在，使得我们只需要在BookPubApplication下用`@Autowired`引入DataSource的bean就可以，Spring Boot会自动创建DataSource的实例。

这里我们会用一个不太规范的starter展示Spring Boot的自动配置的运行原理。[Spring Boot的自动配置、Command-line Runner](#)一文中曾利用StartupRunner类在程序运行启动后首先查询数据库中书的数目，现在换个需求：在系统启动后打印各个实体的数量。

### How Do

- 新建一个模块`db-count-starter`，然后修改`db-count-starter`模块下的pom文件，增加对应的库。

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot</artifactId>
        <!-- version继承父模块的-->
    </dependency>
    <dependency>
        <groupId>org.springframework.data</groupId>
        <artifactId>spring-data-commons</artifactId>
        <version>1.9.3.RELEASE</version>
    </dependency></dependencies>
```

- 新建包结构`com/test/bookpubstarter/dbcount`，然后新建`DbCountRunner`类，实现`CommandLineRunner`接口，在`run`方法中输出每个实体的数量。

```
package com.test.bookpubstarter.dbcount;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.data.repository.CrudRepository;
import java.util.Collection;

public class DbCountRunner implements CommandLineRunner {
    protected final Logger logger = LoggerFactory.getLogger(DbCountRunner.class);
```

```

private Collection<CrudRepository> repositories;

public DbCountRunner(Collection<CrudRepository> repositories) {
    this.repositories = repositories;
}

@Override
public void run(String... strings) throws Exception {
    repositories.forEach(crudRepository -> {
        logger.info(String.format("%s has %s entries",
            getRepositoryName(crudRepository.getClass()),
            crudRepository.count()));
    });
}

private static String getRepositoryName(Class crudRepositoryClass) {
    for (Class repositoryInterface : crudRepositoryClass.getInterfaces()) {
        if (repositoryInterface.getName().startsWith("com.test.bookpub.repository")) {
            return repositoryInterface.getSimpleName();
        }
    }
    return "UnknownRepository";
}
}

```

- 增加自动配置文件*DbCountAutoConfiguration*

```

package com.test.bookpubstarter.dbcount;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.data.repository.CrudRepository;
import java.util.Collection;

@Configuration
public class DbCountAutoConfiguration {
    @Bean
    public DbCountRunner dbCountRunner(Collection<CrudRepository> repositories) {
        return new DbCountRunner(repositories);
    }
}

```

- 在src/main/resources目录下新建META-INF文件夹，然后新建*spring.factories*文件，这个文件用于告诉Spring Boot去找指定的自动配置文件，因此它的内容是

```

org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
com.test.bookpubstarter.dbcount.DbCountAutoConfiguration

```

- 在之前的程序基础上，在顶层pom文件中增加starter的依赖

```
<dependency>
    <groupId>com.test</groupId>
    <artifactId>db-count-starter</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
```

- 把StartupRunner相关的注释掉，然后在main函数上右键Run *BookPubApplication.main(...)*，可以看出我们编写的starter被主程序使用了。

```
2015-12-10 16:32:41.571 INFO 20844 --- [           main] o.s.d.r.w.BasePathAwareHandlerMapping : Mapped "({@profile}/{repository})",methods=[GET],produces=[application/json]
2015-12-10 16:32:41.589 INFO 20844 --- [           main] m.m.a.ExceptionHandlerExceptionResolver : Detected @ExceptionHandler methods in repositoryRestExceptionHandler
2015-12-10 16:32:41.809 INFO 20844 --- [           main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
2015-12-10 16:32:42.468 INFO 20844 --- [           main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8080 (http) 8443 (https)
Hibernate: select count(*) as col_0_0 from publisher reviewer publisher_0_
2015-12-10 16:32:42.760 INFO 20844 --- [           main] c.t.b.dbcount.DbCountRunner : ReviewerRepository has 0 entries
Hibernate: select count(*) as col_0_0 from book book0_
2015-12-10 16:32:42.771 INFO 20844 --- [           main] c.t.b.dbcount.DbCountRunner : BookRepository has 0 entries
Hibernate: select count(*) as col_0_0 from author author0_
2015-12-10 16:32:42.786 INFO 20844 --- [           main] c.t.b.dbcount.DbCountRunner : AuthorRepository has 0 entries
Hibernate: select count(*) as col_0_0 from publisher publisher0_
2015-12-10 16:32:42.800 INFO 20844 --- [           main] c.t.b.dbcount.DbCountRunner : PublisherRepository has 0 entries
2015-12-10 16:32:42.803 INFO 20844 --- [           main] com.test.bookpub.BookPubApplication : Started BookPubApplication in 12.716 seconds (JVM running for 13.589)
```

图片 4.3 自己的starter简单演示.png

## 分析

正规的starter是一个独立的工程，然后在maven中新仓库注册发布，其他开发人员就可以使用你的starter了。

常见的starter会包括下面几个方面的内容：

- 自动配置文件，根据classpath是否存在指定的类来决定是否要执行该功能的自动配置。
- spring.factories，非常重要，指导Spring Boot找到指定的自动配置文件。
- endpoint：可以理解为一个admin，包含对服务的描述、界面、交互（业务信息的查询）
- health indicator：该starter提供的服务的健康指标

在应用程序启动过程中，Spring Boot使用*SpringFactoriesLoader*类加载器查找*org.springframework.boot.autoconfigure.EnableAutoConfiguration*关键字对应的Java配置文件。Spring Boot会遍历在各个jar包种META-INF目录下的*spring.factories*文件，构建成一个配置文件链表。除了*EnableAutoConfiguration*关键字对应的配置文件，还有其他类型的配置文件：

- org.springframework.context.ApplicationContextInitializer*
- org.springframework.context.ApplicationListener*
- org.springframework.boot.SpringApplicationRunListener*
- org.springframework.boot.env.PropertySourceLoader*

- org.springframework.boot.autoconfigure.template.TemplateAvailabilityProvider
- org.springframework.test.context.TestExecutionListener

Spring Boot的starter在编译时不需要依赖Spring Boot的库。这个例子中依赖spring boot并不是因为自动配置要用spring boot，而仅仅是因为需要实现`CommandLineRunner`接口。

## 两个需要注意的点

1. `@ConditionalOnMissingBean`的作用是：只有对应的bean在系统中都没有被创建，它修饰的初始化代码块才会执行，用户自己手动创建的bean优先；
2. Spring Boot starter如何找到自动配置文件（xxxxAutoConfiguration之类的文件）？
  - `spring.factories`: 由Spring Boot触发探测classpath目录下的类，进行自动配置；
  - `@Enable`: 有时需要由starter的用户触发\*查找自动配置文件的过程。

## 配置是否初始化Bean的方法

---

在[Spring Boot：定制自己的starter](#)一文提到，只要DbCountRunner这个类在classpath路径下，Spring Boot会自动创建对应的spring bean并添加到应用程序上下文中。

在文章最后提到，Spring Boot的自动配置机制依靠`@ConditionalOnMissingBean`注解判断是否执行初始化代码，即如果用户已经创建了bean，则相关的初始化代码不再执行。

现在在上篇文章的基础上进行演示，看看`@ConditionalOnMissingBean`注解的作用。

### How Do

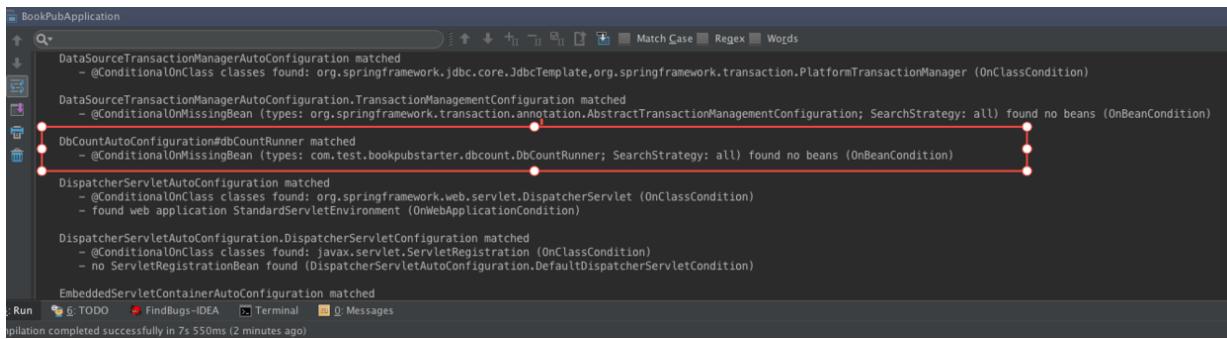
- 在pom文件中增加依赖

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

- 在`DbCountAutoConfiguration`类中添加`@ConditionalOnMissingBean`注解，如下所示：

```
@Configuration
public class DbCountAutoConfiguration {
    @Bean
    @ConditionalOnMissingBean
    public DbCountRunner dbCountRunner(Collection<CrudRepository> repositories) {
        return new DbCountRunner(repositories);
    }
}
```

- 启动应用程序后，看到跟上篇文章相同的结果；
- 修改日志级别为DEBUG，可以看到`DbCountAutoConfiguration`属于Positive match组。

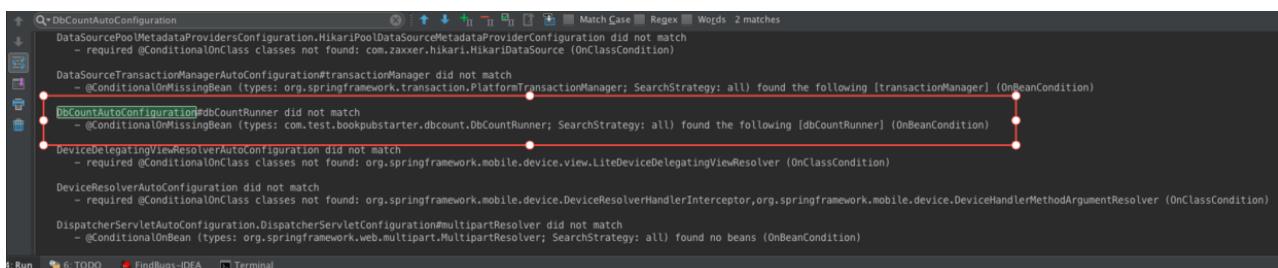


图片 4.4 DbCountAutoConfiguration的自动配置信息

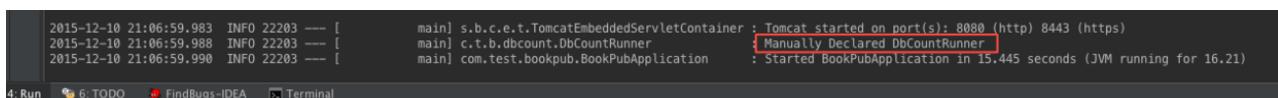
- 在BookPubApplication类中定义DbCountRunner的spring bean

```
@Bean
public DbCountRunner dbCountRunner(Collection<CrudRepository> repositories) {
    return new DbCountRunner(repositories) {
        @Override
        public void run(String... strings) throws Exception {
            logger.info("Manually Declared DbCountRunner");
        }
    };
}
```

- 再次运行程序，观察结果，可以看到这个配置信息放在Negative matchs组中，显示判断条件不匹配，因为已经找到dbCountRunner这个bean。



图片 4.5 手动配置的Bean优先



图片 4.6 修改后的日志信息，显示手动配置bean

## 通过@Enable\*注解触发Spring Boot配置

在[Spring Boot: 定制自己的starter](#)一文最后提到，触发Spring Boot的配置过程有两种方法：

1. `spring.factories`: 由Spring Boot触发探测classpath目录下的类，进行自动配置；
2. `@Enable`: 有时需要由starter的用户触发\*查找自动配置文件的过程。

### How Do

- 接着上篇文章的例子，首先将`spring.factories`中的内容注释掉

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

#com.test.bookpubstarter.dbcount.DbCountAutoConfiguration
```

- 创建元注解（meta-annotation），即在`db-count-starter/src/main/java/org/test/bookpubstarter/dbcount`目录下新建`EnableDbCounting.java`文件。

```
package com.test.bookpubstarter.dbcount;

import org.springframework.context.annotation.Import;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Import(DbCountAutoConfiguration.class)
@Documented
public @interface EnableDbCounting {
}
```

- 在`BookPubApplication`类中删去之前手动创建的`DbCountRunner`的spring bean，然后用`@EnableDbCounting`注解修饰`BookPubApplication`类。

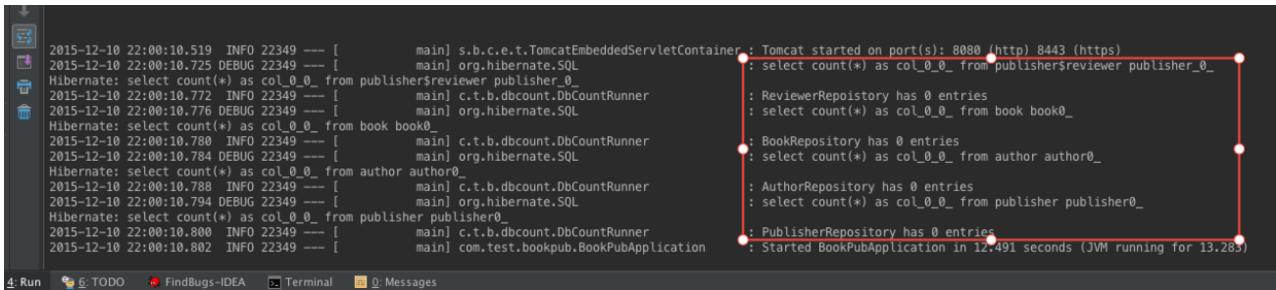
```
package com.test.bookpub;

import com.test.bookpubstarter.dbcount.EnableDbCounting;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
@EnableDbCounting
```

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

- 启动应用程序，设置日志级别为DEBUG



图片 4.7 由starter的用户手动触发配置

可以看出我们自己定义的注解起作用了。如果没有spring.factories，那么在程序启动的时候Spring Boot的自动配置机制不会试图解析DbCountAutoConfiguration类。一般来说，@Component注解的作用范围就是在BookPubApplication所在的目录以及各个子目录，即com.test.bookpub.\*，而DbCountAutoConfiguration是在org.test.bookpubstarter.dbcount目录下，因此不会被扫描到。

@EnableDbCounting注解通过@Import(DbCountAutoConfiguration.class)找到对应的配置类，因此通过用@EnableDbCounting修饰BookPubApplication，就是告诉Spring Boot在启动过程中要把DbCountAutoConfiguration加入到应用上下文中。

HTML



T



unity



HTML



- [Spring Boot应用的测试——Mockito](#)
- [初始化数据库和导入数据](#)
- [在测试中使用内存数据库](#)
- [利用Mockito模拟DB](#)
- [在Spring Boot项目中使用Spock框架](#)

## Spring Boot应用的测试——Mockito

---

Spring Boot可以和大部分流行的测试框架协同工作：通过Spring JUnit创建单元测试；生成测试数据初始化数据库用于测试；Spring Boot可以跟BDD（Behavior Driven Development）工具、Cucumber和Spock协同工作，对应用程序进行测试。

进行软件开发的时候，我们会写很多代码，不过，再过六个月（甚至一年以上）你知道自己的代码怎么运作么？通过测试（单元测试、集成测试、接口测试）可以保证系统的可维护性，当我们修改了某些代码时，通过回归测试可以检查是否引入了新的bug。总得来说，测试让系统不再是一个黑盒子，让开发人员确认系统可用。

在web应用程序中，对Controller层的测试一般有两种方法：（1）发送http请求；（2）模拟http请求对象。第一种方法需要配置回归环境，通过修改代码统计的策略来计算覆盖率；第二种方法是比较正规的思路，但是在我目前经历过的项目中用得不多，今天总结下如何用Mock对象测试Controller层的代码。

在之前的几篇文章中，我们都使用bookpub这个应用程序作为例子，今天也不例外，准备测试它提供的RESTful接口是否能返回正确的响应数据。这种测试不同于单元测试，需要为之初始化完整的应用程序上下文、所有的spring bean都织入以及数据库中需要有测试数据，一般来说这种测试称之为集成测试或者接口测试。

### How Do

通过spirng.io新建的Spring Boot项目提供了一个空的测试文件——*BookPubApplicationTest.java*，内容是：

```
@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = BookPubApplication.class)
public class BookPubApplicationTests {
    @Test
    public void contextLoads() {
    }
}
```

- 在pom文件中增加*spring-boot-starter-test*依赖，添加jsonPath依赖

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

```
<artifactId>json-path</artifactId>
</dependency>
```

- 在BookPubApplicationTest中添加测试用例

```
package com.test.bookpub;

import com.test.bookpub.domain.Book;
import com.test.bookpub.repository.BookRepository;
import org.junit.Before; import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.boot.test.TestRestTemplate;
import org.springframework.boot.test.WebIntegrationTest;
import org.springframework.http.MediaType;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.web.servlet.MockMvc;
import org.springframework.test.web.servlet.setup.MockMvcBuilders;
import org.springframework.web.client.RestTemplate;
import org.springframework.web.context.WebApplicationContext;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertNotNull;
import static org.hamcrest.Matchers.containsString;
import static org.springframework.test.web.servlet.request.MockMvcRequestBuilders.get;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.content;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.jsonPath;
import static org.springframework.test.web.servlet.result.MockMvcResultMatchers.status;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = BookPubApplication.class)
@WebIntegrationTest("server.port:0")
public class BookPubApplicationTests {

    @Autowired
    private WebApplicationContext context;
    @Autowired
    private BookRepository bookRepository;
    @Value("${local.server.port}")
    private int port;

    private MockMvc mockMvc;
    private RestTemplate restTemplate = new TestRestTemplate();

    @Before
    public void setupMockMvc() {
```

```

        mockMvc = MockMvcBuilders.webAppContextSetup(context).build();
    }

    @Test
    public void contextLoads() {
        assertEquals(1, bookRepository.count());
    }

    @Test
    public void webappBookIsbnApi() {
        Book book = restTemplate.getForObject("http://localhost:" + port + "/books/9876-5432-1111", Book.class);
        assertNotNull(book);
        assertEquals("中文测试", book.getPublisher().getName());
    }

    @Test
    public void webappPublisherApi() throws Exception {
        //MockHttpServletRequestBuilder.accept方法是设置客户端可识别的内容类型
        //MockHttpServletRequestBuilder.contentType, 设置请求头中的Content-Type字段, 表示请求体的内容类型
        mockMvc.perform(get("/publishers/1")
            .accept(MediaType.APPLICATION_JSON_UTF8))

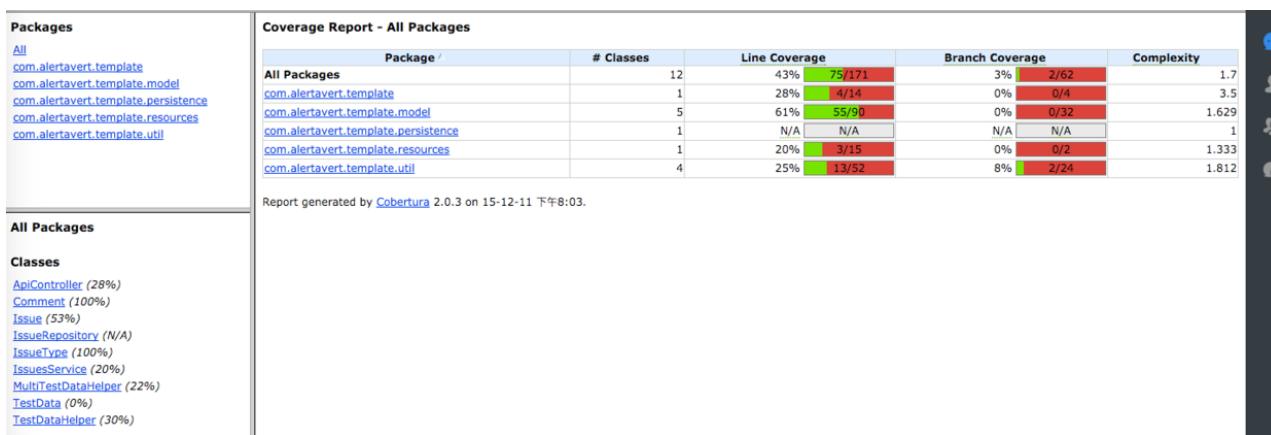
            .andExpect(status().isOk())
            .andExpect(content().string(containsString("中文测试")))
            .andExpect(jsonPath("$.name").value("中文测试"));
    }
}

```

- spring boot项目的代码覆盖率 使用cobertura，参考项目的github地址：[spring boot template](#)

```
# To create test coverage reports (in target/site/cobertura)
```

```
mvn clean cobertura:cobertura test
```



图片 5.1 cobertura统计代码覆盖率

```

    1 // Copyright Marco Massenzio (c) 2014.
  2 // This code is licensed according to the terms of the Apache 2 License.
  3 // See http://www.apache.org/licenses/LICENSE-2.0
  4
  5 package com.alertavert.template.model;
  6
  7 import java.util.Date;
  8
  9 /**
 10  * Value class, encapsulates the concept of a comment to an Issue
 11  *
 12  * This is an immutable class, once created all values cannot be modified.
 13  *
 14  * Created by marco on 12/22/14.
 15  */
 16 public class Comment {
 17     private final String content;
 18     private final String commenter;
 19     private final Date created;
 20
 21     private Comment() {
 22         created = new Date();
 23         commenter = "";
 24         content = "";
 25     }
 26
 27
 28     public Comment(String content, String commenter) {
 29         this.created = new Date();
 30         this.commenter = commenter;
 31         this.content = content;
 32     }
 33
 34
 35     public String getContent() {
 36         return content;
 37     }
 38
 39     public String getCommenter() {
 40         return commenter;
 41     }
 42
 43     public Date getCreated() {
 44         return created;
 45     }
 46 }

```

图片 5.2 单击某个类进去，可以看到详细信息

## 分析

首先分析在BookPubApplicationTests类中用到的注解：

- `@RunWith(SpringJUnit4ClassRunner.class)`，这是JUnit的注解，通过这个注解让`SpringJUnit4ClassRunner`这个类提供Spring测试上下文。
- `@SpringApplicationConfiguration(classes = BookPubApplication.class)`，这是Spring Boot注解，为了进行集成测试，需要通过这个注解加载和配置Spring应用上下文。这是一个元注解（meta-annotation），它包含了`@ContextConfiguration(loader = ApplicationContextLoader.class)`这个注解，测试框架通过这个注解使用Spring Boot框架的`ApplicationContextLoader`加载器创建应用上下文。
- `@WebIntegrationTest("server.port:0")`，这个注解表示当前的测试是集成测试（integration test），因此需要初始化完整的上下文并启动应用程序。这个注解一般和`@SpringApplicationConfiguration`一起出现。`server.port:0`指的是让Spring Boot在随机端口上启动Tomcat服务，随后在测试中程序通过`@Value("${local.server.port}")`获得这个端口号，并赋值给port变量。当在Jenkins或其他持续集成服务器上运行测试程序时，这种随机获取端口的能力可以提供测试程序的并行性。

了解完测试类的注解，再看看测试类的内部。由于这是Spring Boot的测试，因此我们可通过`@Autowired`注解织入任何由Spring管理的对象，或者是通过`@Value`设置指定的环境变量的值。在现在这个测试类中，我们定义了`WebApplicationContext`和`BookRepository`对象。

每个测试用例用@Test注解修饰。在第一个测试用例——contextLoads()方法中，我仅仅需要确认BookRepository连接已经建立，并且数据库中已经包含了对应的测试数据。

第二个测试用例用来测试我们提供的RESTful URL——通过ISBN查询一本书，即“/books/{isbn}”。在这个测试用例中我们使用TestRestTemplate对象发起RESTful请求。

第三个测试用例中展示了如何通过MockMvc对象实现跟第二个测试类似的功能。Spring测试框架提供MockMvc对象，可以在不需要客户端-服务端请求的情况下进行MVC测试，完全在服务端这边就可以执行Controller的请求，跟启动了测试服务器一样。

测试开始之前需要建立测试环境，setup方法被@Before修饰。通过MockMvcBuilders工具，使用WebApplicationContext对象作为参数，创建一个MockMvc对象。

MockMvc对象提供一组工具函数用来执行assert判断，都是针对web请求的判断。这组工具的使用方式是函数的链式调用，允许程序员将多个测试用例链接在一起，并进行多个判断。在这个例子中我们用到下面的一些工具函数：

- perform(get(...))建立web请求。在我们的第三个用例中，通过MockMvcRequestBuilder执行GET请求。
- andExpect(...)可以在perform(...)函数调用后多次调用，表示对多个条件的判断，这个函数的参数类型是ResultMatcher接口，在MockMvcResultMatchers这个类中提供了很多返回ResultMatcher接口的工具函数。这个函数使得可以检测同一个web请求的多个方面，包括HTTP响应状态码(response status)，响应的内容类型(content type)，会话中存放的值，检验重定向、model或者header的内容等等。这里需要通过第三方库json-path检测JSON格式的响应数据：检查json数据包含正确的元素类型和对应的值，例如jsonPath("\$.name").value("中文测试")用于检查在根目录下有一个名为name的节点，并且该节点对应的值是“中文测试”。

## 一个字符乱码问题

- 问题描述：通过spring-boot-starter-data-rest建立的repository，取出的汉字是乱码。
- 分析：使用postman和httpie验证都没问题，说明是Mockmvc的测试用例写得不对，应该主动设置客户端如何解析HTTP响应，用get.accept方法设置客户端可识别的内容类型，修改后的测试用例如下：

```
@Test
public void webappPublisherApi() throws Exception {
    //MockHttpServletRequestBuilder.accept方法是设置客户端可识别的内容类型
    //MockHttpServletRequestBuilder.contentType, 设置请求头中的Content-Type字段, 表示请求体的内容类型
    mockMvc.perform(get("/publishers/1")
        .accept(MediaType.APPLICATION_JSON_UTF8))
```

```
.andExpect(status().isOk())
.andExpect(content().string(containsString("中文测试")))
.andExpect(jsonPath("$.name").value("中文测试"));
}
```

## 参考资料

1. [基于Spring-WS的Restful API的集成测试](#)
2. [J2EE要懂的小事—图解HTTP协议](#)
3. [Integration Testing a Spring Boot Application](#)
4. [spring boot project template](#)

## 初始化数据库和导入数据

在[Spring Boot应用的测试](#)一文中，我们在StarterRunner类的run(...)方法中给数据库中添加一些初始数据。尽管通过编程方式添加初始数据比较快捷方便，但长期来看这并不是一个好办法——特别是当需要添加的数据量很大时。我们开发最好把数据库准备、数据库修改和数据库的配置与将要运行的程序代码分离，尽管这仅仅是为测试用例做准备。Spring Boot已经提供了相应的支持来完成这个任务。

我们在之前的应用程序基础上进行实验。Spring Boot提供两种方法来定义数据库的表结构以及添加数据。第一种方法是使用Hibernate提供的工具来创建表结构，该机制会自动搜索@Entity实体对象并创建对应的表，然后使用import.sql文件导入测试数据；第二种方法是利用旧的Spring JDBC，通过schema.sql文件定义数据库的表结构、通过data.sql导入测试数据。

### How Do

- 首先，将现有的“编程式初始化数据”的代码注释掉，因此在StarterRunner中run方法中注释掉下列代码：

```
@Override
public void run(String... strings) throws Exception {
//    Author author = new Author("du", "qi");
//    authorRepository.save(author);
//    Publisher publisher = new Publisher("中文测试");
//    publisherRepository.save(publisher);
//    Book book = new Book(author, "9876-5432-1111", publisher, "瞅啥");
//    book.setDescription("23333333, 这是一本有趣的书");
//    bookRepository.save(book);
    logger.info("Number of books: " + bookRepository.count());
}
```

- 在resources目录下新建import.sql文件(注意，SQL语句中指定的字段要与Hibernate自动生成的表的字段相同)，该文件的内容如下：

```
INSERT INTO author (id, first_name, last_name) VALUES (1, 'Alex', 'Antonov');
INSERT INTO publisher (id, name) VALUES (1, 'Packt');
INSERT INTO book (isbn, title, author, publisher) VALUES ('9876-5432-1111', 'Spring Boot Recipes', 1, 1);
```

- 现在运行测试用例，发现可以通过；
- 第二种方法是获取Spring JDBC的支持，需要我们提供schema.sql和data.sql文件。现在可以将import.sql重命名为data.sql，然后再创建新的文件schema.sql。在删除数据表时，需要考虑依赖关系，例如表A依赖表B，则先删除表B。创建数据库关系的内容如下：

```
-- clear context
DROP TABLE IF EXISTS `book_reviewers`;
DROP TABLE IF EXISTS `reviewer`;
DROP TABLE IF EXISTS `book`;
DROP TABLE IF EXISTS `author`;
DROP TABLE IF EXISTS `publisher`;

-- Create syntax for TABLE 'author'
CREATE TABLE `author` (
    `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    `first_name` VARCHAR(255) DEFAULT NULL ,
    `last_name` VARCHAR(255) DEFAULT NULL ,
    PRIMARY KEY (`id`)
);

-- Create syntax for TABLE 'publisher'
CREATE TABLE `publisher` (
    `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    `name` VARCHAR(255) DEFAULT NULL ,
    PRIMARY KEY (`id`)
);

-- Create syntax for TABLE 'book'
CREATE TABLE `book` (
    `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    `description` VARCHAR(255) DEFAULT NULL ,
    `isbn` VARCHAR(255) DEFAULT NULL ,
    `title` VARCHAR(255) DEFAULT NULL ,
    `author` BIGINT(20) DEFAULT NULL ,
    `publisher` BIGINT(20) DEFAULT NULL ,
    PRIMARY KEY (`id`),
    CONSTRAINT `FK_publisher` FOREIGN KEY (`publisher`) REFERENCES `publisher` (`id`),
    CONSTRAINT `FK_author` FOREIGN KEY (`author`) REFERENCES `author` (`id`)
);

-- Create syntax for TABLE 'reviewer'
CREATE TABLE `reviewer` (
    `id` BIGINT(20) NOT NULL AUTO_INCREMENT,
    `first_name` VARCHAR(255) DEFAULT NULL ,
    `last_name` VARCHAR(255) DEFAULT NULL ,
    PRIMARY KEY (`id`)
);

-- Create syntax for TABLE 'book_reviewers'
CREATE TABLE `book_reviewers` (
```

```

`books` BIGINT(20) NOT NULL ,
`reviewers` BIGINT(20) NOT NULL ,
CONSTRAINT `FK_book` FOREIGN KEY (`books`) REFERENCES `book` (`id`),
CONSTRAINT `FK_reviewer` FOREIGN KEY (`reviewers`) REFERENCES `reviewer` (`id`)
);

```

- 我们手动创建了数据库表结构，因此需要关掉Hibernate的自动创建开关，即在`application.properties`中设置`spring.jpa.hibernate.ddl-auto = none`
- 运行测试，发现测试可以正常通过。

Note: 个人建议是使用Hibernate的自动创建机制，当然这会少一点可定制性；最近更流行的是Mybatis，mybatis-spring-boot也可以使用，mybatis的可定制性更强。

## 分析

在Spring社区中常常可以通过使用各种组件，例如Spring JDBC、Spring JPA with Hibernate，或者Flyway、Liquidbase这类数据库迁移工具，都能实现类似的功能。

Note: Flyway和Liquidbase都提供数据库的增量迁移功能。当项目中需要管理数据库的增量变动，并且需要快速切换到指定的数据版本时，非常适合使用Flyway和Liquidbase，更多的信息可以参考<http://flywaydb.org/>和<http://www.liquibase.org/>。

在上文中我们使用了两种不同的方法来初始化数据库和填充测试数据

### 使用Spring JPA with Hibernate初始化数据库

这种方法中，由Hibernate库完成大部分工作，我们只需要配置合适的配置项，并创建对应的实体类的定义。在这个方案中我们主要使用以下配置项：`- spring.jpa.hibernate.ddl-auto=create-drop`配置项告诉Hibernate通过@Entity模型的定义自动推断数据库定义并创建合适的表。在程序启动时，经由Hibernate计算出的schema会用来创建表结构，在程序结束时这些表也被删除。即使程序强制退出或者奔溃，在重新启动的时候也会先把之前的表删除，并重新创建——因此“create-drop”这种配置不适合生产环境。PS:如果程序没有显式配置`spring.jpa.hibernate.ddl-auto`属性，Spring Boot会给H2这类的嵌入式数据库配置`create-drop`，因此需要仔细斟酌这个配置项。

- 在classpath下创建`import.sql`文件供Hibernate使用，该文件中的内容是一些SQL语句，将会在应用程序启动时运行。尽管该文件中可以写任何有效的SQL语句，不过建议只写数据操作语句，例如INSERT、UPDATE等等。

## 使用Spring JDBC初始化数据库

如果项目中没有用JPA或者你不想依赖Hibernate库，Spring提供另外一种方法来设置数据库，当然，首先需要提供`spring-boot-starter-jdbc`依赖。

- `spring.jpa.hibernate.ddl-auto=none`表示Hibernate不会自动创建数据库表结构。在生产环境中最好用这个设置，能够避免你不小心将数据库全部删除（那一定是一个噩梦）。
- `schema.sql`文件包含创建数据库表结构的SQL语句，在应用程序启动过程中，需要创建数据库表结构时，执行该文件中的DDL语句。Hibernate会自动删除已经存在的表，如果我们希望只有某个表不存在的时候才创建它，可以在该文件开头最好先使用`DROP TABLE IF EXISTS`删除可能存在的表，再使用`CREATE TABLE IF NOT EXISTS`创建表。这种用法可以灵活得定义数据库中的表结构，因此在生产环境中用更安全。
- `data.sql`的作用跟上一个方法的`import.sql`一样，用于存放数据导入的SQL语句。

考虑到这是Spring的特性，我们可以不只是全局定义数据库定义文件，还可以针对不同的数据库定义不同的文件。例如，可以定义给Oracle数据库使用的`schema-oracle.sql`，给MySQL数据库用的`schema-mysql.sql`文件；对于`data.sql`文件，则可以由不同数据库共用。如果你希望覆盖Spring Boot的自动推断，可以配置`spring.datasource.platform`属性。

Tip: 如果你希望使用别的名字代替`schema.sql`或者`data.sql`，Spring Boot也提供了对应的配置属性，即`spring.datasource.schema`和`spring.datasource.data`。

## 在测试中使用内存数据库

---

在[初始化数据库和导入数据](#)一文中，我们探索了在Spring Boot项目中如何创建数据库的表结构，以及如何往数据库中填充初始数据。在程序开发过程中常常会在环境配置上浪费很多时间，例如在一个存在数据库组件的应用程序中，测试用例运行之前必须保证数据库中的表结构正确，并且已经填入初始数据。对于良好的测试用例，还需要保证数据库在执行用例前后状态不改变。

在之前应用的基础上，*schema.sql*文件中包含创建数据库表结构的SQL语句、*data.sql*文件中包含填充初始数据的SQL语句。这篇文章将//todo

### How Do

- 在src/test/resources目录下创建*test-data.sql*文件，用于导入测试数据

```
INSERT INTO author(first_name, last_name) VALUES ("Greg", "Turnquist");
INSERT INTO book(isbn, title, author, publisher) VALUES ('9781-78439-302-1', 'Learning Spring Boot', 2, 1)
```

- 修改*BookPubApplicationTests*文件，添加数据源属性（ds），是否需要导入测试数据的标志（loadDataFixtures），添加loadDataFixtures方法。

```
public class BookPubApplicationTests {
    ...
    @Autowired
    private DataSource ds;
    private static boolean loadDataFixtures = true;
    private MockMvc mockMvc;
    private RestTemplate restTemplate = new TestRestTemplate();

    @Before
    public void setupMockMvc() {
        ...
    }

    @Before
    public void loadDataFixtures() {
        if (loadDataFixtures) {
            ResourceDatabasePopulator populator = new
                ResourceDatabasePopulator(context.getResource("classpath:/test-data.sql"));
            DatabasePopulatorUtils.execute(populator, ds);
            loadDataFixtures = false;
        }
    }
}
```

```

        }
    }

@Test
public void contextLoads() {
    assertEquals(2, bookRepository.count());
}

@Test    public void webappBookIsbnApi() {
    ...
}

@Test    public void webappPublisherApi() throws Exception {
    ...
}
}

```

- 运行单元测试，可以通过
- Spring Boot会搜集resources目录下的所有*data.sql*文件进行数据导入，由于测试代码有自己的resource目录，因此在这个目录下再创建一个*data.sql\**文件，内容是：

```

INSERT INTO author (id, first_name, last_name) VALUES (3, "William", "Shakespeare");
INSERT INTO publisher (id, name) VALUES (2, "Classical Books");
INSERT INTO book(isbn, title, author, publisher) VALUES ('978-1-23456-789-1', "Romeo and Juliet", 3, 2);

```

- 由于又新加了一个book记录，因此需要修改BookPubApplicationTest

```

@Test
public void contextLoads() {
    assertEquals(3, bookRepository.count());
}

```

- 至此我们还都是使用外部数据库——MySQL，现在尝试使用内存数据库H2，因此在*src/test/resources*目录下添加*application.properties*文件，内容是：

```

spring.datasource.url=\  jdbc:h2:mem:testdb;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
spring.jpa.hibernate.ddl-auto=None

```

- 执行测试用例，可以通过。需要注意的是：Spring Boot仅仅会加载一个*application.properties*文件，由于此处我在*src/test/resources*目录下新建了*application.properties*文件，所以之前的那个（在*src/main/resources*目录下）不会被加载。

## 分析

我们通过Spring的ResourceDatabasePopulator和DatabasePopulatorUtils类加载test-data.sql文件，在test-data.sql文件中的数据仅仅对当前所在的\*Test.java文件有效。Spring Boot自身去处理schema.sql和data.sql文件时也是依靠这两个类，这里我们不过是显式指定了我们希望执行的脚本文件。

- 创建setup方法——*loadDataFixtures()*，并用@Before注解修饰，表示在测试用例之前运行该方法。
- 在*loadDataFixtures()*方法中，首先通过context.getResources方法加载test-data.sql文件，然后通过*DatabasePopulatorUtils.execute(populator, ds)*执行该文件中的SQL语句。
- 为了避免每个@Test修饰的测试用例之前都导入数据，因此引入一个标志变量——*loadDataFixtures*（初始化为true），因此该方法只执行一次。

## 利用Mockito模拟DB



图片 5.3 mockito.jpg

前两篇文章的主要内容是：为了给执行测试，如何建立数据库表和导入初始数据。这里我们将学习如何利用Mockito框架和一些注解模拟（mock）*Repository*实例，从而使得测试用例不依赖外部的数据库服务。

我们需要创建一个Spring Boot配置类，在该类中定义用于测试的Spring Bean；我们通过注解指示Spring Boot何时加载测试配置类以及何时执行该类中的代码。在改配置类中，我们将使用Mockito框架创建一些带预定义方法的mock对象，Spring Boot在执行测试用例之前会将这些对象织入。

### How Do

- 首先创建一个注解用于标识仅用于测试的配置类，可以按照如下方法修改BookPubApplication类。可以看出，关键语句`@ComponentScan(excludeFilters = @ComponentScan.Filter(UsedForTesting.class))`表示：程序正式运行时不扫描`@UsedForTesting`修饰的类。

```

@Configuration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = @ComponentScan.Filter(UsedForTesting.class))
@EnableScheduling
public class BookPubApplication {
    public static void main(String[] args) {
        SpringApplication.run(BookPubApplication.class, args);
    }
    @Bean
    public StartupRunner schedulerRunner() {
        return new StartupRunner();
    }
}

@interface UsedForTesting {}

```

- 在 src/test/java/com/test/bookpub 目录下创建 *TestMockBeansConfig* 文件，内容是：

```
package com.test.bookpub;

import com.test.bookpub.repository.PublisherRepository;
import org.mockito.Mockito;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;

@Configuration
@UsedForTesting
public class TestMockBeansConfig {
    @Bean
    @Primary
    public PublisherRepository createMockPublisherRepository() {
        return Mockito.mock(PublisherRepository.class);
    }
}
```

- 新建一个测试类——*PublisherRepositoryTests*，主要是因为 BookPubApplicationTest 中的内容太多太乱了（在实际项目中我们会严格限制每个测试类中的内容）。

```
package com.test.bookpub;
import com.test.bookpub.repository.PublisherRepository;
import org.junit.After; import org.junit.Before; import org.junit.Test;
import org.junit.runner.RunWith; import org.mockito.Mockito;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.IntegrationTest;
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import static org.junit.Assert.assertEquals;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = {
    BookPubApplication.class,
    TestMockBeansConfig.class
})
@IntegrationTest
public class PublisherRepositoryTests {
    @Autowired
    private PublisherRepository repository;

    @Before
    public void setupPublisherRepositoryMock() {
```

```

    Mockito.when(repository.count())
        .thenReturn(1L);
}

@Test
public void publishersExist() {
    assertEquals(1, repository.count());
}

@Before
public void resetPublisherRepositoryMock() {
    Mockito.reset(repository);
}
}

```

## 分析

OK，分析下刚刚发生了什么。首先，我们从对BookPubApplication.java的修改开始：

- `@SpringBootApplication`被三个注解替换：`@Configuration`, `@EnableAutoConfiguration`和`@ComponentScan(excludeFilters = @ComponentScan.Filter(UsedForTesting.class))`，这么做的原因是可以给`@ComponentScan`注解增加`excludeFilters`属性，通过这个属性，我们提示Spring Boot在正式运行时忽略被`@UsedForTesting`修饰的类。
- `@UsedForTesting`注解定义在BookPubApplication.java文件中，用于修饰`TestMockBeansConfig`类。

接下来看看在`TestMockBeansConfig`中的操作，

- `@Configuration`注解说明这是一个配置类，该类含有应用程序上下文，如果被其他配置文件引入，则该类中定义的Spring Bean应该加入到已经创建的应用上下文。
- 修饰`createMockPublisherRepository`方法的注解`@Primary`表示：如果在织入的时候发现有多个`PublisherRepository`的Spring Bean，则让Spring Boot优先使用该方法返回的Spring Bean。在应用程序启动时，Spring Boot根据`@RepositoryRestResource`注解，已经生成一个`PublisherRepository`的实例，但是这里我们希望应用程序不使用这个真实的实例，而使用Mockito框架模拟出的`PublisherRepository`实例。

最后看下我们的测试用例，主要关注`setupPublisherRepositoryMock`方法和`resetPublisherRepositoryMock`方法：

- `setupPublisherRepositoryMock`方法被`@Before`注解修饰，表示在测试用例运行之前被调用，在这个方法中我们配置了mock对象的行为：如果收到`repository.count()`调用，则返回1。Mockito框架提供了很多DSL形式的语句，可以用于定义这些容易理解的规则。

- `resetPublisherRepositoryMock`方法被`@After`注解修饰，在测试用例执行过后调用，用于清楚之前对repository的设置。

## 在Spring Boot项目中使用Spock框架

Spock框架是基于Groovy语言的测试框架，Groovy与Java具备良好的互操作性，因此可以在Spring Boot项目中使用该框架写优雅、高效以及DSL化的测试用例。Spock通过@RunWith注解与JUnit框架协同使用，另外，Spock也可以和Mockito ([Spring Boot应用的测试——Mockito](#)) 协同使用。

在这个小节中我们会利用Spock、Mockito一起编写一些测试用例（包括对Controller的测试和对Repository的测试），感受下Spock的使用。

### How Do

- 根据[Building an Application with Spring Boot](#)这篇文章的描述，`spring-boot-maven-plugin`这个插件同时也支持在Spring Boot框架中使用Groovy语言。
- 在pom文件中添加Spock框架的依赖

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

- 在src/test目录下创建groovy文件夹，在groovy文件夹下创建com/test/bookpub包。
- 在com/test/bookpub目录下创建SpockBookRepositorySpecification.groovy文件，内容是：

```
@WebAppConfiguration
@ContextConfiguration(classes = [BookPubApplication.class,
    TestMockBeansConfig.class], loader = ApplicationContextLoader.class)
class SpockBookRepositorySpecification extends Specification {
    @Autowired
```

```

private ConfigurableApplicationContext context;
@Shared
boolean sharedSetupDone = false;
@.Autowired
private DataSource ds;
@Autowired
private BookRepository bookRepository;
@Autowired
private PublisherRepository publisherRepository;
@Shared
private MockMvc mockMvc;

void setup() {
    if (!sharedSetupDone) {
        mockMvc = MockMvcBuilders.webAppContextSetup(context).build();
        sharedSetupDone = true;
    }
    ResourceDatabasePopulator populator = new
        ResourceDatabasePopulator(context.getResource("classpath:/packt-books.sql"));
    DatabasePopulatorUtils.execute(populator, ds);
}

@Transactional
def "Test RESTful GET"() {
    when:
    def result = mockMvc.perform(get("/books/${isbn}"));

    then:
    result.andExpect(status().isOk())
    result.andExpect(content().string(containsString("title")));

    where:
    isbn           | title
    "978-1-78398-478-7"|"Orchestrating Docker"
    "978-1-78528-415-1"|"Spring Boot Recipes"
}

@Transactional
def "Insert another book"() {
    setup:
    def existingBook = bookRepository.findBookByIsbn("978-1-78528-415-1")
    def newBook = new Book("978-1-12345-678-9", "Some Future Book",
        existingBook.getAuthor(), existingBook.getPublisher())

    expect:

```

```

bookRepository.count() == 3

when:
def savedBook = bookRepository.save(newBook)

then:
bookRepository.count() == 4
savedBook.id > -1
}
}

```

- 执行测试用例，测试通过
- 接下来试验下Spock如何与mock对象一起工作，之前的文章中我们已经在*TestMockBeansConfig*类中定义了*PublisherRepository*的Spring Bean，如下所示，由于@Primary的存在，使得在运行测试用例时Spring Boot优先使用Mockito框架模拟出的实例。

```

@Configuration
@UsedForTesting
public class TestMockBeansConfig {
    @Bean
    @Primary
    public PublisherRepository createMockPublisherRepository() {
        return Mockito.mock(PublisherRepository.class);
    }
}

```

- 在*BookController.java*中添加*getBooksByPublisher*接口，代码如下所示：

```

@.Autowired
public PublisherRepository publisherRepository;

@RequestMapping(value = "/publisher/{id}", method = RequestMethod.GET)
public List<Book> getBooksByPublisher(@PathVariable("id") Long id) {
    Publisher publisher = publisherRepository.findOne(id);
    Assert.notNull(publisher);
    return publisher.getBooks();
}

```

- 在*SpockBookRepositorySpecification.groovy*文件中添加对应的测试用例，

```

def "Test RESTful GET books by publisher"() {
    setup:
    Publisher publisher = new Publisher("Strange Books")
    publisher.setId(999)
    Book book = new Book("978-1-98765-432-1",

```

```

    "Mytery Book",
    new Author("Jhon", "Done"),
    publisher)
publisher.setBooks([book])
Mockito.when(publisherRepository.count()).
    thenReturn(1L);
Mockito.when(publisherRepository.findOne(1L)).
    thenReturn(publisher)

when:
def result = mockMvc.perform(get("/books/publisher/1"))

then:
result.andExpect(status().isOk())
result.andExpect(content().string(containsString("Strange Books")))

cleanup:
Mockito.reset(publisherRepository)
}

```

- 运行测试用例，发现可以测试通过，在控制器将对象转换成JSON字符串装入HTTP响应体时，依赖Jackson库执行转换，可能会有循环依赖的问题——在模型关系中，一本书依赖一个出版社，一个出版社有包含多本书，在执行转换时，如果不进行特殊处理，就会循环解析。我们这里通过@JsonBackReference注解阻止循环依赖。

## 分析

可以看出，通过Spock框架可以写出优雅而强大的测试代码。

首先看SpockBookRepositorySpecification.groovy文件，该类继承自Specification类，告诉JUnit这个类是测试类。查看Specification类的源码，可以发现它被@RunWith(Sputnik.class)注解修饰，这个注解是连接Spock与JUnit的桥梁。除了引导JUnit，Specification类还提供了很多测试方法和mocking支持。

Note: 关于Spock的文档见[这里](#): [Spock Framework Reference Documentation](#)

根据《单元测试的艺术》一书中提到的，单元测试包括：准备测试数据、执行待测试方法、判断执行结果三个步骤。Spock通过setup、expect、when和then等标签将这些步骤放在一个测试用例中。

- setup: 这个块用于定义变量、准备测试数据、构建mock对象等；
- expect: 一般跟在setup块后使用，包含一些assert语句，检查在setup块中准备好的测试环境
- when: 在这个块中调用要测试的方法；

- `then`：一般跟在`when`后使用，尽可以包含断言语句、异常检查语句等等，用于检查要测试的方法执行后结果是否符合预期；
- `cleanup`：用于清除`setup`块中对环境做的修改，即将当前测试用例中的修改回滚，在这个例子中我们对`publisherRepository`对象执行重置操作。

Spock也提供了`setup()`和`cleanup()`方法，执行一些给所有测试用例使用的准备和清除动作，例如在这个例子中我们使用`setup`方法：（1）mock出web运行环境，可以接受http请求；（2）加载`packt-books.sql`文件，导入预定义的测试数据。web环境只需要Mock一次，因此使用`sharedSetupDone`这个标志来控制。

通过`@Transactional`注解可以实现事务操作，如果某个方法被该注解修饰，则与之相关的`setup()`方法、`cleanup()`方法都被定义在一个事务内执行操作：要么全部成功、要且回滚到初始状态。我们依靠这个方法保证数据库的整洁，也避免了每次输入相同的数据。

HTML



T



unity



HTML



- [Spring Boot应用的打包和部署](#)
- [Docker with Spring Boot](#)

Spring Boot应用的打包和部署

现在的IT开发，[Devops](#)渐渐获得技术管理人员支持、云计算从ECS转向Docker容器技术、微服务的概念和讨论也越来越热，以上这些研究方面，最终都聚焦于软件的打包、分发和部署上。

[Twelve-Factor App] 开发方法这一系列的博文主要讲述了一个现代的SaaS应用是如何被构建和部署的，其中一个关键的原则是：分离配置定义和应用程序。

DevOps开发模型要求开发人员管理应用程序的开发、测试、打包和部署等所有流程，当然，必须确保这些步骤的执行足够简单和可控，否则开发人员都没有时间维护软件和开发新功能了。要实现DevOps模型，需要简洁、隔离的应用程序包，这种应用程序自带运行容器、可以当做进程一样一键运行，并且不需要重新构建就部署到不同的机器上。

## 一、创建基于Spring Boot框架的可执行Jar包

Spring Boot开发的应用可以打包为单独的JAR包，然后通过 `java -jar <name>.jar` 命令运行。接下来我们基于之前练习使用的应用程序，看看如何构建Spring Boot Uber JAR。

Note: Uber JAR是将应用程序打包到单独的jar包中，该jar包包含了应用程序依赖的所有库和二进制包。

## How Do

- 通过 `mvn clean package` 命令打包应用程序
  - 通过命令 `java -jar target/bookpub-0.0.1-SNAPSHOT.jar` 运行程序

图片 6-1 运行 Jar 启动 Java web 应用

## 分析

如上所示，打包成可执行的jar包这种方法相当直观，背后的工作由spring-boot-maven-plugin插件实现：先通过maven-shade-plugin生成一个包含依赖的jar，再通过spring-boot-maven-plugin插件把spring boot loader相关的类，还有MANIFEST.MF打包到jar里。关于Spring Boot的启动原理分析，详见[spring boot应用启动原理分析](#)一文。

总结下Spring Boot应用的启动流程：（1）spring boot应用打包之后，生成一个fat jar，里面包含了应用依赖的jar包，还有Spring boot loader相关的类；（2）Fat jar的启动Main函数是JarLauncher，它负责创建一个LaunchedURLClassLoader来加载/lib下面的jar，并以一个新线程启动应用的Main函数。

## 二、创建Docker镜像

可以参考我之前写的一篇文章：[Docker with Spring Boot](#)

## Docker with Spring Boot

---

前段时间在我厂卷爷的指导下将Docker在我的实际项目中落地，最近几个小demo都尽量熟悉docker的使用，希望通过这篇文章分享我截止目前的使用经验（如有不准确的表述，欢迎帮我指出）。本文的主要内容是关于Java应用程序的docker化，首先简单介绍了docker和docker-compose，然后利用两个案例进行实践说明。

简单说说Docker，现在云计算领域火得一塌糊涂的就是它了吧。Docker的出现是为了解决PaaS的问题：运行环境与具体的语言版本、项目路径强关联，因此干脆利用Docker技术进行资源隔离，构造出跟随应用发布的运行环境，这样就解决了语言版本的限制问题。PaaS的出现是为了让运维人员不需要管理一台虚拟机，IaaS的出现是为了让运维人员不需要管理物理机。云计算，说到底都是俩字——运维。

云计算领域的技术分为虚拟化技术和资源管理两个方面，正好对应我们今天要讲的两个工具：Docker和docker-compose。Docker的主要概念有：容器、镜像、仓库；docker-compose是fig的后续版本，负责将多个docker服务整合起来，对外提供一致服务。

### 1. Spring Boot应用的docker化

首先看Spring Boot应用程序的docker化，由于Spring Boot内嵌了tomcat、Jetty等容器，因此我们对docker镜像的要求就是需要java运行环境。我的应用代码的的Dockerfile文件如下：

```
#基础镜像：仓库是java，标签用8u66-jdk

FROM java:8u66-jdk
#当前镜像的维护者和联系方式

MAINTAINER duqi duqi@example.com
#将打包好的spring程序拷贝到容器中的指定位置

ADD target/bookpub-0.0.1-SNAPSHOT.jar /opt/bookpub-0.0.1-SNAPSHOT.jar
#容器对外暴露8080端口

EXPOSE 8080
#容器启动后需要执行的命令

CMD java -Djava.security.egd=file:/dev/./urandom -jar /opt/bookpub-0.0.1-SNAPSHOT.jar
```

因为目前的示例程序比较简单，这个Dockerfile并没有在将应用程序的数据存放在宿主机上。如果你的应用程序需要写文件系统，例如日志，最好利用 `VOLUME /tmp` 命令，这个命令的效果是：在宿主机的`/var/lib/docker`目录下创建一个临时文件并把它链接到容器中的`/tmp`目录。

把这个Dockerfile放在项目的根目录下即可，后续通过 docker-compose build 统一构建：基础镜像是只读的，然后会在该基础镜像上增加新的可写层来供我们使用，因此java镜像只需要下载一次。

docker-compose是用来做docker服务编排，参看《Docker从入门到实践》中的解释：

Compose 项目目前在 Github 上进行维护，目前最新版本是 1.2.0。Compose 定位是“defining and running complex applications with Docker”，前身是 Fig，兼容 Fig 的模板文件。

Dockerfile 可以让用户管理一个单独的应用容器；而 Compose 则允许用户在一个模板（YAML 格式）中定义一组相关联的应用容器（被称为一个 project，即项目），例如一个 Web 服务容器再加上后端的数据库服务容器等。

单个docker用起来确实没什么用，docker技术的关键在于持续交付，通过与jenkins的结合，可以实现这样的效果：开发人员提交push，然后jenkins就自动构建并测试刚提交的代码，这就是我理解的持续交付。

## 2. spring boot + redis + mongodb

在这个项目中，我启动三个容器：web、redis和mongodb，然后将web与redis连接，web与mongodb连接。首先要进行redis和mongodb的docker化，redis镜像的Dockerfile内容是：

```
FROM      ubuntu:14.04
RUN      apt-get update
RUN      apt-get -y install redis-server
EXPOSE   6379
ENTRYPOINT ["/usr/bin/redis-server"]
```

Mongodb镜像的Dockerfile内容是，docker官方给了mongodb的docker化教程，我直接拿来用了，参见[Dockerizing MongoDB](#)：

```
# Format: FROM repository[:version]

FROM      ubuntu:14.04
# Format: MAINTAINER Name <email@addr.ess>

MAINTAINER duqi duqi@example.com
# Installation:

# Import MongoDB public GPG key AND create a MongoDB list file

RUN apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 7F0CEB10
RUN echo "deb http://repo.mongodb.org/apt/ubuntu $(lsb_release -sc) /mongodb-org/3.0 multiverse" | tee /etc/apt/sources.list.d/mongodb-org-3.0.list
# Update apt-get sources AND install MongoDB
```

```

RUN apt-get update && apt-get install -y mongodb-org
# Create the MongoDB data directory

RUN mkdir -p /data/db
# Expose port 27017 from the container to the host

EXPOSE 27017
# Set usr/bin/mongod as the dockerized entry-point application

ENTRYPOINT ["/usr/bin/mongod"]

```

使用docker-compose编排三个服务，具体的模板文件如下：

```

web:
  build: .
  ports:
    - "49161:8080"
  links:
    - redis
    - mongodb

redis:
  image: duqi/redis
  ports:
    - "6379:6379"

mongodb:
  image: duqi/mongodb
  ports:
    - "27017:27017"

```

架构比较简单，第一个区块的build，表示docker中的命令“`docker build .`”，用于构建web镜像；ports这块表示将容器的8080端口与宿主机（IP地址是：192.168.99.100）的49161对应。因为现在docker不支持原生的os x，因此在mac下使用docker，实际上是在mac上的一台虚拟机（docker-machine）上使用docker，这台机器的地址就是192.168.99.100。参见：[在mac下使用docker](#)

links表示要连接的服务，redis与下方的redis区块对应、mongodb与下方的mongodb区块对应。redis和mongodb类似，首先说明要使用的镜像，然后规定端口映射。

那么，如何运行呢？ 1. 命令 `docker-compose build`，表示构建web服务，目前我用得比较土，就是编译jar之后还需要重新更新docker，优雅点不应该这样。

```
~/I/dailyReport >>> docker-compose build
redis uses an image, skipping
mongodb uses an image, skipping
Building web
Step 1 : FROM java:8u66-jdk
--> de4a13c84f53
Step 2 : MAINTAINER duqi duqi.dq@alibaba-inc.com
--> Using cache
--> 771a7227241b
Step 3 : ADD target/dailyReport-1.0-SNAPSHOT.jar /opt/dailyReport-1.0-SNAPSHOT.jar
--> 88569eb9e3a3
Removing intermediate container 28999d133249
Step 4 : EXPOSE 8080
--> Running in 915c1eb75d60
--> 14bad58b53b9
Removing intermediate container 915c1eb75d60
Step 5 : CMD java -Djava.security.egd=file:/dev/.urandom -jar /opt/dailyReport-1.0-SNAPSHOT.jar
--> Running in e0058610cbbd
--> d5599e0dbac8
Removing intermediate container e0058610cbbd
Successfully built d5599e0dbac8
```

图片 6.2 docker-compose build

2. 命令 `docker-compose up`，表示启动web服务，可以看到mongodb、redis和web依次启动，启动后用 `docker ps` 查看当前的运行容器。

```
~/I/mongodb >>> docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED            STATUS              PORTS               NAMES
d21a1d7cfde3        dailyreport_web   "/bin/sh -c 'java -Dj"   22 seconds ago    Up 21 seconds      0.0.0.0:49161->8080/tcp   dailyreport_web_1
bdb694371c75        duqi/mongodb     "/usr/bin/mongod"    10 days ago       Up 22 seconds      0.0.0.0:27017->27017/tcp   dailyreport_mongodb_1
394935bf42c3        duqi/redis       "/usr/bin/redis-serv"  13 days ago       Up 22 seconds      0.0.0.0:6379->6379/tcp   dailyreport_redis_1
```

图片 6.3 docker ps

特别注意，在配置文件中写redis和mongodb的url时，要用虚拟机的地址，即192.168.99.100。例如，redis的一个配置应该为：`spring.redis.host=192.168.99.100`。

### 3. spring boot + mysql

拉取mysql镜像的指令是：`docker run --name db001 -p 3306:3306 -e MYSQL_ROOT_PASSWORD=admin -d mysql:5.7`，表示启动的docker容器名字是db001，登录密码一定要设定，`-d`表示设置Mysql版本。

我的docker-compose模板文件是：

```
web:
  build: .
  ports:
  - "49161:8080"
  links:
  - mysql

mysql:
```

```
image: mysql:5.7
environment:
  MYSQL_ROOT_PASSWORD: admin
  MYSQL_DATABASE: springbootcookbook
ports:
- "3306:3306"
```

主要内容跟之前的类似，主要讲下mysql部分，通过environment来设置进入mysql容器后的环境变量，即连接数据库的密码MYSQL\_ROOT\_PASSWORD，使用的数据库名称MSYQL\_DATABASE等等。

一直想写这篇文章做个总结，写来发现还是有点薄，对于docker我还需要系统得学习，不过，针对上面的例子，我都是亲自实践过的，大家有什么问题可以与我联系。

## 参考资料

1. [Docker从入门到实践](#)
2. [Docker – Initialize mysql database with schema](#)
3. [使用Docker搭建基础的mysql应用](#)
4. [Spring Boot with docker](#)

HTML



T



unity



HTML



- [Spring Boot应用的健康监控](#)
- [Spring Boot Admin的使用](#)
- [通过JMX监控Spring Boot应用](#)

## Spring Boot应用的健康监控

在之前的系列文章中我们学习了如何进行Spring Boot应用的功能开发，以及如何写单元测试、集成测试等，然而，在实际的软件开发中需要做的不仅如此：还包括对应用程序的监控和管理。

正如飞行员不喜欢盲目飞行，程序员也需要实时看到自己的应用目前的运行情况。如果给定一个具体的时间，我们希望知道此时CPU的利用率、内存的利用率、数据库连接是否正常以及在给定时间段内有多少客户请求等指标；不仅如此，我们希望通过图表、控制面板来展示上述信息。最重要的是：老板和业务人员希望看到的是图表，这些比较直观易懂。

首先，这篇文章讲介绍如何定制自己的health indicator。

### How Do

- 在pom文件中添加*spring-boot-starter-actuator*依赖

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

- *spring-boot-starter-actuator*这个库让我们可以访问应用的很多信息，包括：/env、/info、/metrics、/health等。现在运行程序，然后在浏览器中访问：<http://localhost:8080/health>，将可以看到下列内容。



```
{
  "status" : "UP",
  "diskSpace" : {
    "status" : "UP",
    "total" : 120108089344,
    "free" : 31699034112,
    "threshold" : 10485760
  },
  "db" : {
    "status" : "UP",
    "database" : "MySQL",
    "hello" : 1
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/health"
    }
  }
}
```

图片 7.1 acatuator库提供监控信息

- 除了/health可以访问，其他的Endpoints也可以访问，例如/info：首先在application.properties文件中添加对应的属性值，符号@包围的属性值来自pom.xml文件中的元素节点。

```
info.build.artifact=@project.artifactId@
info.build.name=@project.name@
info.build.description=@project.description@
info.build.version=@project.version@
```

- 要获取配置文件中的节点值，需要在pom文件中进行一定的配置，首先在节点里面添加：

```
<resources>
  <resource>
    <directory>src/main/resources</directory>
    <filtering>true</filtering>
  </resource>
</resources>
```

然后在节点里面增加对应的插件：

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.6</version>
  <configuration>
    <delimiters>
```

```

<delimiter>@</delimiter>
</delimiters>
<useDefaultDelimiters>false</useDefaultDelimiters>
</configuration>
</plugin>

```

- 然后运行应用程序，访问`http://localhost:8080/info`，可以看到下列信息



图片 7.2 `http://localhost:8080/info`

- 除了使用系统默认的监控信息，我们还可以定义自己的health indicator。使用[Spring Boot：定制自己的starter](#)一文中做过的db-count-starter作为观察对象，我们希望监控每个数据库接口的运行状况：如果某个接口返回的个数大于等于0，则表示系统正常，表示为UP状态；否则，可能该接口发生异常，表示为DOWN状态。首先，将DbCountRunner类中的getRepositoryName方法由private转为protected，然后在db-count-starter这个模块中也添加actuator依赖。

- 在`db-count-starter/src/main/com/test/bookpubstarter`目录下创建`DbCountHealthIndicator.java`文件

```

public class DbCountHealthIndicator implements HealthIndicator {
    private CrudRepository crudRepository;
    public DbCountHealthIndicator(CrudRepository crudRepository) {
        this.crudRepository = crudRepository;
    }
    @Override
    public Health health() {
        try {
            long count = crudRepository.count();
            if (count >= 0) {
                return Health.up().withDetail("count", count).build();
            } else {
                return Health.unknown().withDetail("count", count).build();
            }
        }
    }
}

```

```
        }
    } catch (Exception e) {
        return Health.down(e).build();
    }
}
}
```

- 最后，还需要注册刚刚创建的健康监控器，在DbCountAutoConfiguration.java中增加如下定义：

```
@Autowired
private HealthAggregator healthAggregator;
@Bean
public HealthIndicator dbCountHealthIndicator(Collection<CrudRepository> repositories) {
    CompositeHealthIndicator compositeHealthIndicator = new
        CompositeHealthIndicator(healthAggregator);
    for (CrudRepository repository: repositories) {
        String name = DbCountRunner.getRepositoryName(repository.getClass());
        compositeHealthIndicator.addHealthIndicator(name, new DbCountHealthIndicator(repository));
    }
    return compositeHealthIndicator;
}
```

- 运行程序，然后访问http://localhost:8080/health，则可以看到如下结果



```
{
  "status" : "UP",
  "dbCount" : {
    "status" : "UP",
    "BookRepository" : {
      "status" : "UP",
      "count" : 1
    },
    "ReviewerRepository" : {
      "status" : "UP",
      "count" : 1
    },
    "AuthorRepository" : {
      "status" : "UP",
      "count" : 1
    },
    "PublisherRepository" : {
      "status" : "UP",
      "count" : 1
    }
  },
  "diskSpace" : {
    "status" : "UP",
    "total" : 120108089344,
    "free" : 31662907392,
    "threshold" : 10485760
  },
  "db" : {
    "status" : "UP",
    "database" : "MySQL",
    "hello" : 1
  },
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/health"
    }
  }
}
```

图片 7.3 自定义的health indicator

## 分析

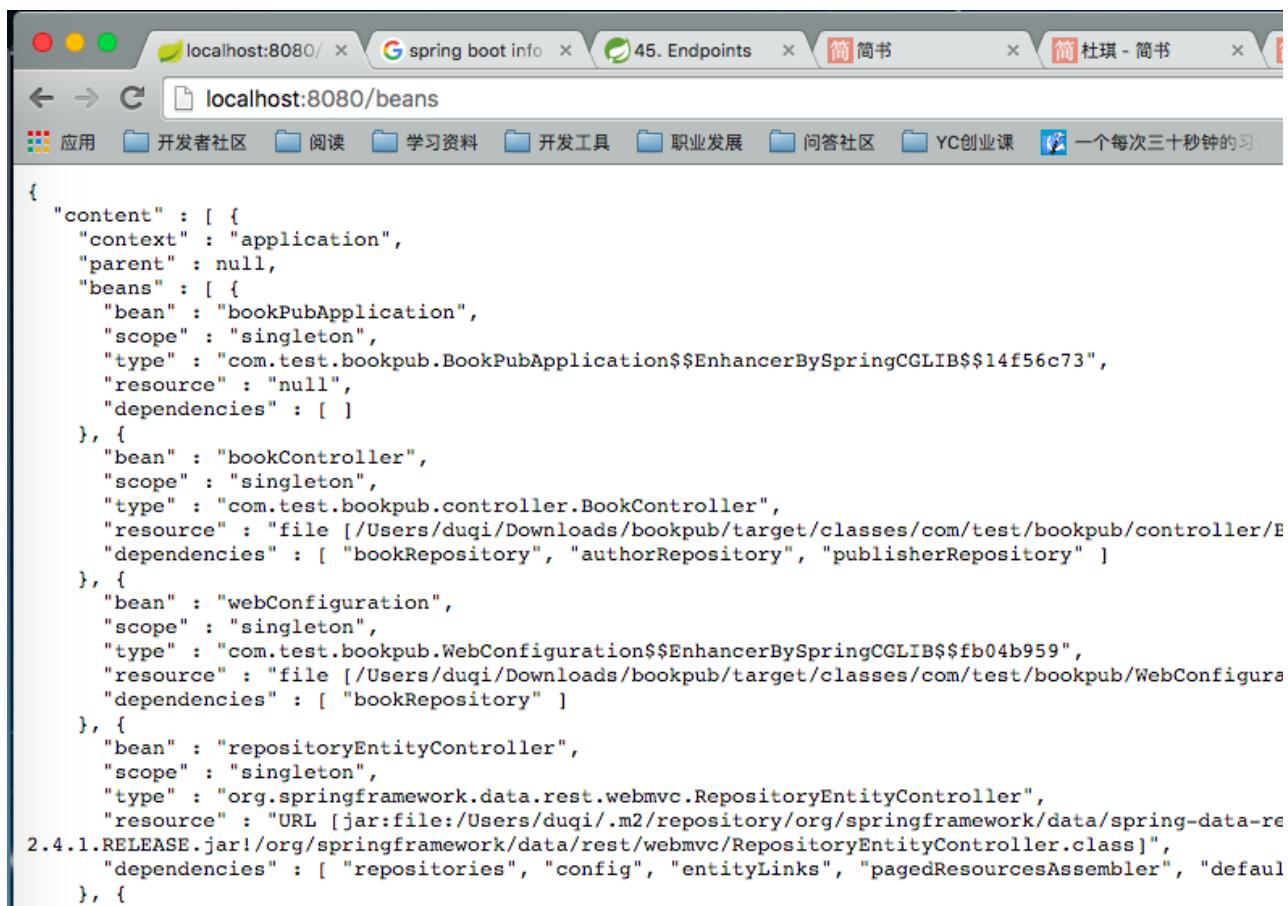
Spring Boot Autuator这个库包括很多自动配置，对外开放了很多endpoints，通过这些endpoints可以访问应用的运行时状态： - /env提供应用程序的环境变量，如果你在调试时想知道某个配置项在运行时的值，可以通过这个endpoint访问——访问`http://localhost:8080/env`，可以看到很多方面的配置，例如， class path resource s—[tomcat.https.properties]、applicationConfig—[classpath:/application.properties]、commonsConfi

g、systemEnvironment、systemProperties等。这些变量的值由Environment实例中的PropertySource实例保存，根据这些属性值所在的层次，有可能在运行时已经做了值替换，跟配置文件中的不一样了。为了确认某个属性的具体值，例如book.count.rate属性，可以访问`http://localhost:8080/env/book.counter.rate`来查询，如果跟配置文件中的不一样，则可能是被系统变量或者命令行参数覆盖了。EnvironmentEndpoint类负责实现上述功能，有兴趣可以再看看它的源码；`- /configprops`提供不同配置对象，例如`WebConfiguration.TomcatSslConnectionProperties`，它与`/env`不同的地方在于它会表示出与配置项绑定的对象。尝试下访问`http://localhost:8080/configprops`，然后在网页中查询`custom.tomcat.https`，可以看到我们之前用于配置`TomcatSslConnector`对象的属性值(参见：[让你的Spring Boot工程支持HTTP和HTTPS](#))。

TomcatSslConnector对应的属性值

图片 7.4 TomcatSslConnector对应的属性值

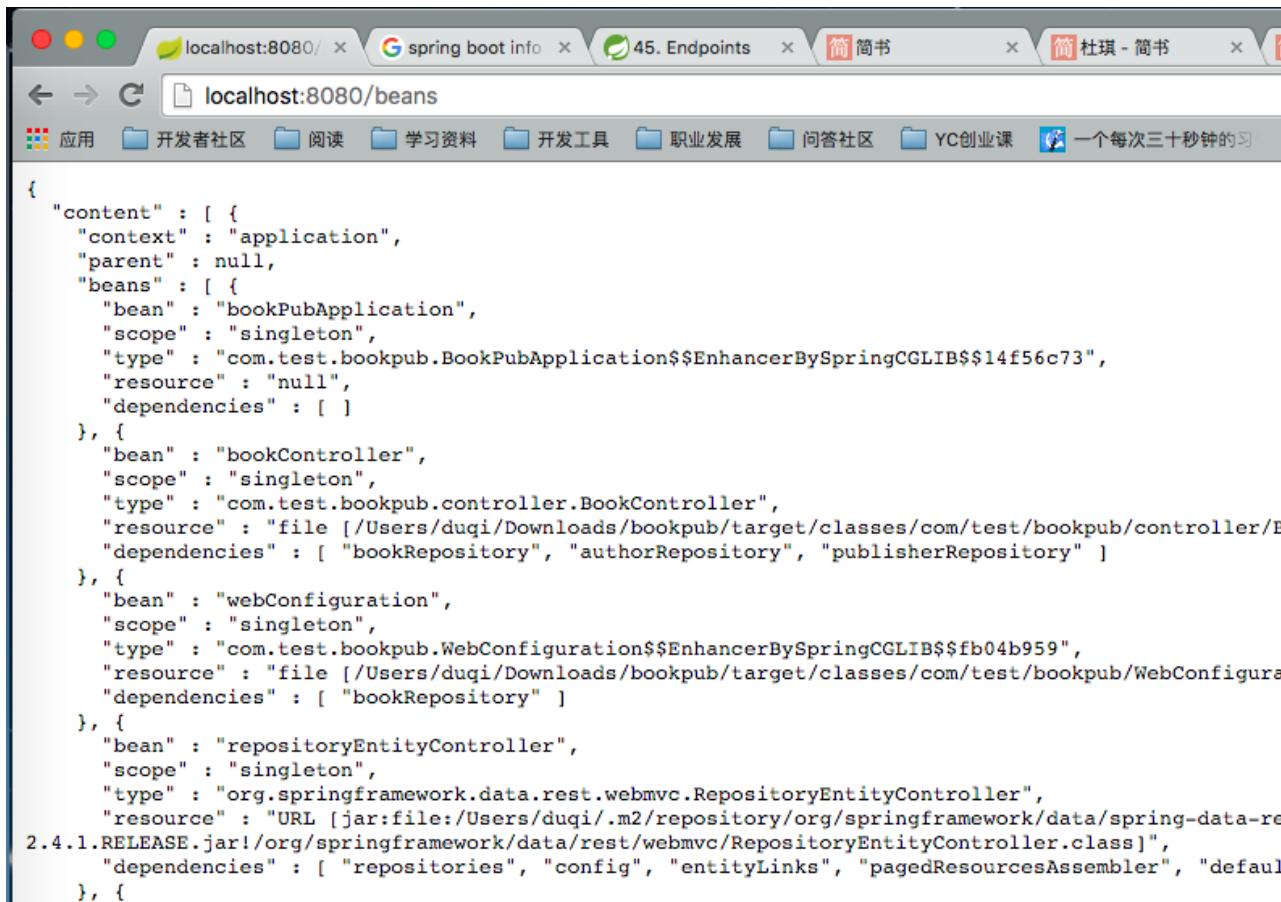
`- /autoconfig`以web形式对外暴露AutoConfiguration 信息，这些信息的解释可以参考[Spring Boot：定制自己的starter](#)一文，这样我们就不需要通过“修改应用程序的日志级别和查看应用的启动信息”来查看应用的自动配置情况了。`- /beans`，这个endpoint列出所有由Spring Boot创建的bean。



```
{
  "content": [
    {
      "context": "application",
      "parent": null,
      "beans": [
        {
          "bean": "bookPubApplication",
          "scope": "singleton",
          "type": "com.test.bookpub.BookPubApplication$$EnhancerBySpringCGLIB$$14f56c73",
          "resource": "null",
          "dependencies": []
        },
        {
          "bean": "bookController",
          "scope": "singleton",
          "type": "com.test.bookpub.controller.BookController",
          "resource": "file [/Users/duqi/Downloads/bookpub/target/classes/com/test/bookpub/controller/E",
          "dependencies": [ "bookRepository", "authorRepository", "publisherRepository" ]
        },
        {
          "bean": "webConfiguration",
          "scope": "singleton",
          "type": "com.test.bookpub.WebConfiguration$$EnhancerBySpringCGLIB$$fb04b959",
          "resource": "file [/Users/duqi/Downloads/bookpub/target/classes/com/test/bookpub/WebConfigura",
          "dependencies": [ "bookRepository" ]
        },
        {
          "bean": "repositoryEntityController",
          "scope": "singleton",
          "type": "org.springframework.data.rest.webmvc.RepositoryEntityController",
          "resource": "URL [jar:file:/Users/duqi/.m2/repository/org/springframework/data/spring-data-re",
          "dependencies": [ "repositories", "config", "entityLinks", "pagedResourcesAssembler", "defaul",
        }
      ]
    }
  ]
}
```

图片 7.5 /beans显示所有Spring Boot创建的bean

`- /mapping`，这个endpoint显示当前应用支持的URL映射，该映射关系由HandlerMapping类维护，通过这个endpoint可以查询某个URL的路由信息。



```
{
  "content": [
    {
      "context": "application",
      "parent": null,
      "beans": [
        {
          "bean": "bookPubApplication",
          "scope": "singleton",
          "type": "com.test.bookpub.BookPubApplication$$EnhancerBySpringCGLIB$$14f56c73",
          "resource": "null",
          "dependencies": []
        },
        {
          "bean": "bookController",
          "scope": "singleton",
          "type": "com.test.bookpub.controller.BookController",
          "resource": "file [/Users/duqi/Downloads/bookpub/target/classes/com/test/bookpub/controller/E",
          "dependencies": [ "bookRepository", "authorRepository", "publisherRepository" ]
        },
        {
          "bean": "webConfiguration",
          "scope": "singleton",
          "type": "com.test.bookpub.WebConfiguration$$EnhancerBySpringCGLIB$$fb04b959",
          "resource": "file [/Users/duqi/Downloads/bookpub/target/classes/com/test/bookpub/WebConfigura",
          "dependencies": [ "bookRepository" ]
        },
        {
          "bean": "repositoryEntityController",
          "scope": "singleton",
          "type": "org.springframework.data.rest.webmvc.RepositoryEntityController",
          "resource": "URL [jar:file:/Users/duqi/.m2/repository/org/springframework/data/spring-data-re",
          "dependencies": [ "repositories", "config", "entityLinks", "pagedResourcesAssembler", "defaul",
        }
      ]
    }
  ]
}
```

图片 7.6 /mappings查看URL映射

- /info，这个endpoint显示应用程序的基本描述，在之前的实践例子中我们看过它的返回信息，属性值来自`application.properties`，同时也可以使用占位符获取`pom.xml`文件中的信息。任何以`info.`开头的属性都会在访问`http://localhost:8080/info`时显示。 - /health提供应用程序的健康状态，或者是某个核心模块的健康状态。 - /metrics，这个endpoint显示Metrics子系统管理的信息，后面的文章会详细介绍它。

上述各个endpoint是Spring Boot Actuator提供的接口和方法，接下来看看我们自己定制的`HealthIndicator`，我们只需要实现`HealthIndicator`接口，Spring Boot会收集该接口的实现，并加入到`/health`这个endpoint中。

在我们的例子中，我们为每个`CrudRepository`实例都创建了一个`HealthIndicator`实例，为此我们创建了一个`CompositeHealthIndicator`实例，由这个实例管理所有的`DbHealthIndicator`实例。作为一个`composite`，它会提供一个内部的层次关系，从而可以返回JSON格式的数据。

代码中的`HealthAggregator`实例的作用是：它维护一个map，告诉`CompositeHealthIndicator`如何决定所有`HealthIndicator`代表的整体的状态。例如，除了一个repository返回DOWN其他的都返回UP，这时候这个composite indicator作为一个整体应该返回UP还是DOWN，`HealthAggregator`实例的作用就在这里。

Spring Boot使用的默认的*HealthAggregator*实现是*OrderedHealthAggregator*, 它的策略是手机所有的内部状态, 然后选出在DOWN、OUT\_OF\_SERVICE、UP和UNKNOWN中间具有最低优先级的那个状态。这里使用策略设计模式, 因此具体的状态判定策略可以改变和定制, 例如我们可以创建定制的*HealthAggregator*:

最后需要考虑下安全问题, 通过这些endpoints暴露出很多应用的信息, 当然, Spring Boot也提供了配置项, 可以关闭指定的endpoint——在application.properties中配置.enable=false;

还可以通过设置management.port=-1关闭endpoint的HTTP访问接口, 或者是设置其他的端口, 供内部的admin服务访问; 除了控制端口, 还可以设置仅仅让本地访问, 只需要设置management.address=127.0.0.1; 通过设置management.context-path=/admin, 可以设置指定的根路径。综合下, 经过上述设置, 在本地访问http://127.0.0.1/admin/health来访问健康状态。

可以在防火墙上屏蔽掉不是/admin/\*的endpoints访问请求, 更进一步, 利用Spring Security可以配置验证信息, 这样要访问当前应用的endpoints必须使用用户名和密码登陆。

## 参考资料

1. [Endpoints](#)
2. [Complete Guide for Spring Boot Actuator](#)

## Spring Boot Admin的使用

---

上一篇文章中了解了Spring Boot提供的监控接口，例如：/health、/info等等，实际上除了之前提到的信息，还有其他信息也需要监控：当前处于活跃状态的会话数量、当前应用的并发数、延迟以及其他度量信息。这次我们了解如何利用Spring-boot-admin对应用信息进行可视化，如何添加度量信息。

### 准备

spring-boot-admin的Github地址在：<https://github.com/codecentric/spring-boot-admin>，它在Spring Boot Actuator的基础上提供简洁的可视化WEB UI。

- 首先在[start.spring.io](https://start.spring.io)中创建简单的admin应用，主要步骤如下：
  - Group: org.sample.admin
  - Artifact: spring-boot-admin-web
  - Name: Spring Boot Admin Web
  - Description: Spring Boot Admin Web Application
  - Package Name: org.sample.admin
  - \*\*Type: \*\* Maven Project
  - Packaging: Jar
  - Java Version: 1.8
  - Language: Java
  - Spring Boot Version: 1.3.1
- 在Ops组选项中选择Actuator
- 选择Generate Project下载应用
- 使用IDEA打开工程，在pom.xml文件中添加下列依赖

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-server</artifactId>
  <version>1.3.2</version>
</dependency>
<dependency>
```

```
<groupId>de.codecentric</groupId>
<artifactId>spring-boot-admin-server-ui</artifactId>
<version>1.3.2</version>
</dependency>
```

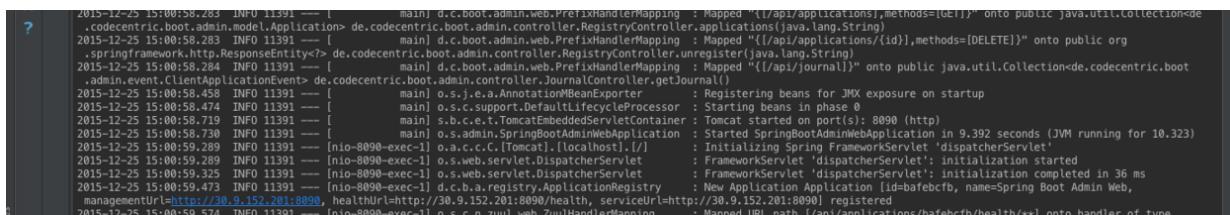
- 在`SpringBootAdminWebApplication.java`文件中添加`@EnableAdminServer`注解

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

- 在`application.properties`文件中添加如下配置

```
server.port = 8090
spring.application.name=Spring Boot Admin Web
spring.boot.admin.url=http://localhost:${server.port}
spring.jackson.serialization.indent_output=true
endpoints.health.sensitive=false
```

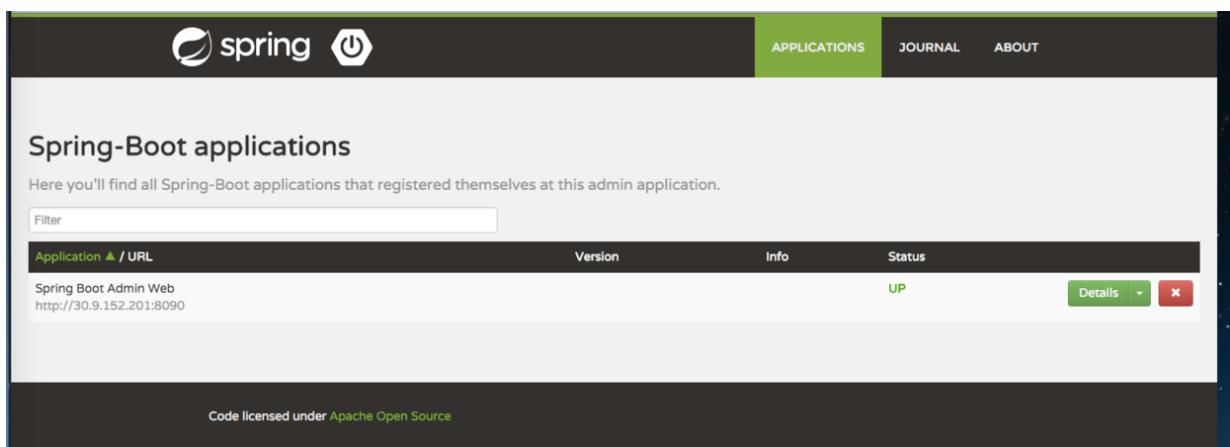
- 启动应用程序，在后台给定如下提示：



```
2015-12-25 15:00:58.283 INFO 11391 --- [main] d.c.boot.admin.web.PrefixHandlerMapping : Mapped "{(/api/applications),methods=[GET]}" onto public java.util.Collection<de.codecentric.boot.admin.model.Application> de.codecentric.boot.admin.controller.RegistryController.getApplications(java.lang.String)
2015-12-25 15:00:58.283 INFO 11391 --- [main] d.c.boot.admin.web.PrefixHandlerMapping : Mapped "{(/api/applications/{id}),methods=[DELETE]}" onto public org.springframework.http.ResponseEntity<> de.codecentric.boot.admin.controller.RegistryController.unregister(java.lang.String)
2015-12-25 15:00:58.284 INFO 11391 --- [main] d.c.boot.admin.web.PrefixHandlerMapping : Mapped "{(/api/journal)}" onto public java.util.Collection<de.codecentric.boot.admin.event.ClientApplicationEvent> de.codecentric.boot.admin.controller.JournalController.getJournal()
2015-12-25 15:00:58.458 INFO 11391 --- [main] o.s.j.e.a.AnnotationBeanExporter : Registering beans for JMX exposure on startup
2015-12-25 15:00:58.474 INFO 11391 --- [main] o.s.c.support.DefaultLifecycleProcessor : Starting beans in phase 0
2015-12-25 15:00:58.719 INFO 11391 --- [main] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on port(s): 8090 (http)
2015-12-25 15:00:58.730 INFO 11391 --- [main] o.s.admin.SpringApplication : Started SpringBootAdminWebApplication in 9.392 seconds (JVM running for 10.323)
2015-12-25 15:00:59.289 INFO 11391 --- [nio-8090-exec-1] o.a.c.c.[Tomcat].[localhost].[/] : Initializing Spring FrameworkServlet 'dispatcherServlet'
2015-12-25 15:00:59.289 INFO 11391 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization started
2015-12-25 15:00:59.325 INFO 11391 --- [nio-8090-exec-1] o.s.web.servlet.DispatcherServlet : FrameworkServlet 'dispatcherServlet': initialization completed in 36 ms
2015-12-25 15:00:59.473 INFO 11391 --- [nio-8090-exec-1] d.c.b.a.registry.ApplicationRegistry : New Application Application [id:bafebcfc, name=Spring Boot Admin Web, managementUrl=http://30.9.152.201:8090, healthUrl=http://30.9.152.201:8090/health, serviceUrl=http://30.9.152.201:8090] registered
2015-12-25 15:00:59.572 INFO 11391 --- [nio-8090-exec-1] d.c.b.a.handler.JournalHandlerMapping : Mapped URL path [/api/applications/{id}/journal] onto handler of type
```

图片 7.7 spring-boot-admin应用启动日志

- 在浏览器中访问上图中提示的地址，可以看到下图的信息



图片 7.8 spring-boot-admin应用

## How Do

- 启动Admin Web应用后，现在可以添加针对BookPub应用的度量信息了。在文章[Spring Boot应用的健康监控](#)中，我们曾定制自己的Health Indicator，用来监控四个数据库接口的健康状态，这次我将利用spring-boot-admin对这些信息进行可视化管理。
- 在db-count-starter模块下添加代码，首先在`db-count-starter/src/main/java/com/test/bookpubstarter/dbcount`目录下添加`DbCountMetrics`类：

```
public class DbCountMetrics implements PublicMetrics {
    private Collection<CrudRepository> repositories;
    public DbCountMetrics(Collection<CrudRepository> repositories) {
        this.repositories = repositories;
    }
    @Override
    public Collection<Metric<?>> metrics() {
        List<Metric<?>> metrics = new LinkedList<>();
        for (CrudRepository repository: repositories) {
            String name =
DbCountRunner.getRepositoryName(repository.getClass());
            String metricName = "counter.datasource." + name;
            metrics.add(new Metric(metricName, repository.count()));
        }
        return metrics;
    }
}
```

- 在`DbCountAutoConfiguration`定义对应的Bean，由Spring Boot完成自动注册

```
@Bean
public PublicMetrics dbCountMetrics(Collection<CrudRepository> repositories) {
    return new DbCountMetrics(repositories);
}
```

- 启动BookPub应用，访问`http://localhost:8080/metrics`，可以看到`DbCountMetrics`已经添加到metrics列表中了。



```
{
  "_links" : {
    "self" : {
      "href" : "http://localhost:8080/metrics"
    }
  },
  "mem" : 385024,
  "mem.free" : 180577,
  "processors" : 4,
  "instance.uptime" : 23611,
  "uptime" : 39718,
  "systemload.average" : 4.46337890625,
  "heap.committed" : 385024,
  "heap.init" : 131072,
  "heap.used" : 204446,
  "heap" : 1864192,
  "threads.peak" : 20,
  "threads.daemon" : 16,
  "threads.totalStarted" : 23,
  "threads" : 19,
  "classes" : 9656,
  "classes.loaded" : 9656,
  "classes.unloaded" : 0,
  "gc.ps_scavenge.count" : 10,
  "gc.ps_scavenge.time" : 154,
  "gc.ps_marksweep.count" : 2,
  "gc.ps_marksweep.time" : 184,
  "counter.datasource.PublisherRepository" : 1,
  "counter.datasource.ReviewerRepository" : 1,
  "counter.datasource.BookRepository" : 1,
  "counter.datasource.AuthorRepository" : 1,
  "httpsessions.max" : -1,
  "httpsessions.active" : 0,
  "datasource.primary.active" : 0,
  "datasource.primary.usage" : 0.0
}
```

图片 7.9 新添加的DbCountMetrics

- 在db-count-starter模块下的pom文件中添加spring-boot-admin-starter-client依赖,

```
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-client</artifactId>
  <version>1.3.2</version>
</dependency>
```

- 在BookPub应用下的application.properties中配置下列属性值

```
spring.application.name=@project.description@  
server.port=8080  
spring.boot.admin.url=http://localhost:8090
```

- 启动BookPub应用，然后在浏览器中访问`http://localhost:8090`

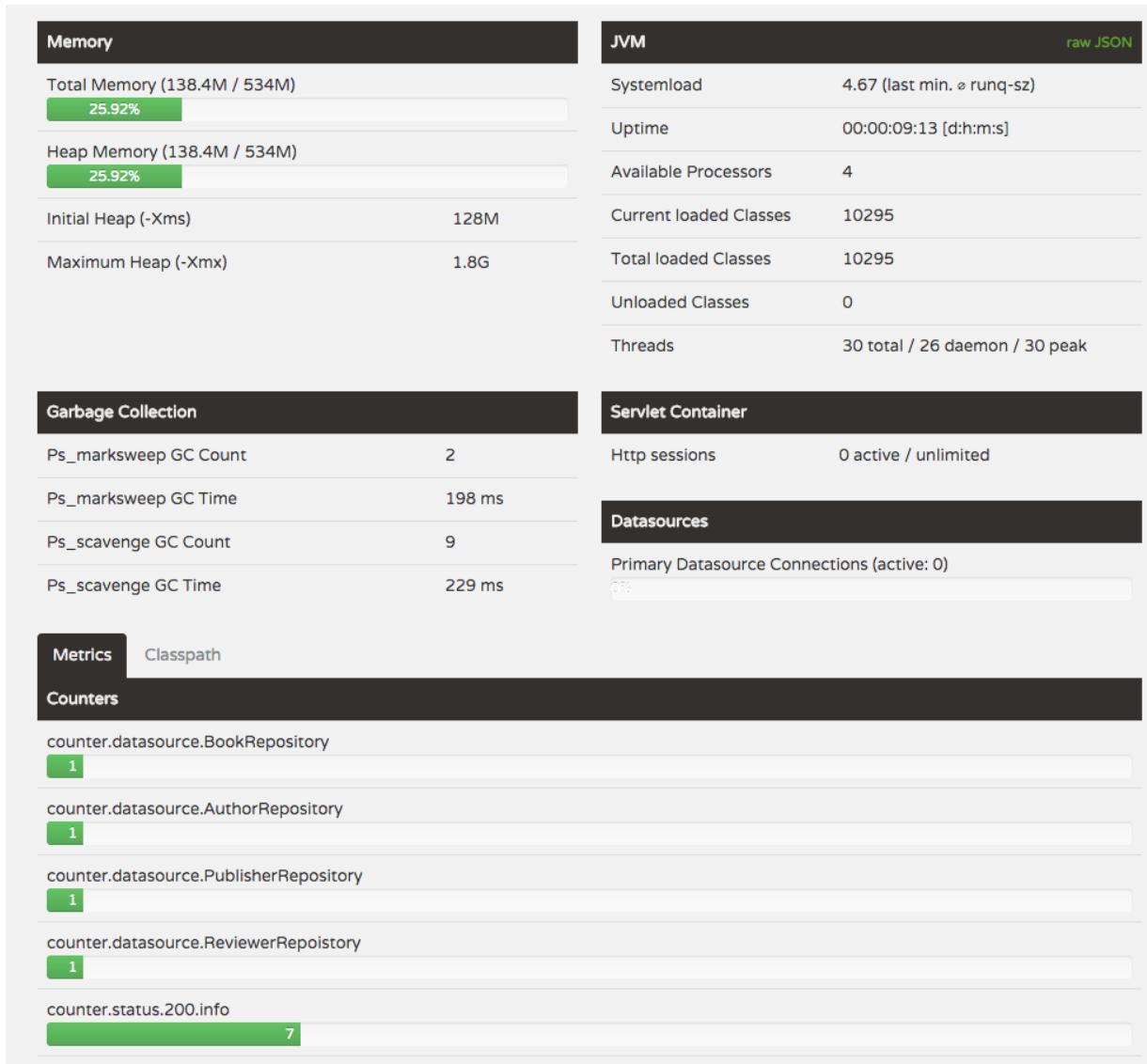
The screenshot shows the Spring Boot Admin web interface. At the top, there is a navigation bar with the Spring logo, followed by links for APPLICATIONS, JOURNAL, and ABOUT. The main content area is titled "Spring-Boot applications" and contains a sub-header: "Here you'll find all Spring-Boot applications that registered themselves at this admin application." Below this is a table listing two applications:

Application ▲ / URL	Version	Info	Status	Actions
BookPub Catalog Application with Spring Boot http://30.9.152.201:8080		build: description: BookPub Catalog Application with Spring Boot name: BookPub ...	UP	<button>Details</button> <button>X</button>
Spring Boot Admin Web http://30.9.152.201:8090			UP	<button>Details</button> <button>X</button>

At the bottom of the page, there is a footer note: "Code licensed under Apache Open Source".

图片 7.10 监控BookPub应用

- 点击右侧的“Details”，可以看到该应用的详细信息



图片 7.11 BookPub应用的详细信息

## 分析

Spring Boot Admin就是将Spring Boot Actuator中提供的endpoint信息可视化表示，在BookPub应用（被监控）的这一端，只需要进行一点配置即可。  
– spring-boot-admin-starter-client，作为客户端，用于与Spring Boot Admin Web的服务器沟通；  
– spring.boot.admin.url=http://localhost:8090用于将当前应用注册到Spring Boot Admin。

如果希望通过Web控制系统的日志级别，则需要在应用中添加Jolokia JMX库（org.jolokia:jolokia-core），同时在项目资源目录下添加logback.xml文件，内容如下：

```
<configuration>
<include resource="org/springframework/boot/logging/logback/base.xml"/>
```

```
<jmxConfigurator/>
</configuration>
```

然后再次启动BookPub应用，然后在Spring Boot Admin的页面中查看LOGGING，则可以看到如下页面：

Component	TRACE	DEBUG	<b>INFO</b>	WARN	ERROR	OFF
ROOT						
com.test.bookpub.BookPubApplication						
com.test.bookpub.StartupRunner						
com.test.bookpub.controller.BookController						
com.test.bookpubstarter.dbcount.DbCountRunner						
de.codecentric.boot.admin.services.ApplicationRegistrar						
org.apache.catalina.core.ContainerBase						
org.apache.catalina.core.ContainerBase.[Tomcat]						

图片 7.12 通过Spring Boot Admin修改日志级别

Spring Boot提供的度量工具功能强大且具备良好的扩展性，除了我们配置的DbCountMetrics，还监控BookPub应用的其他信息，例如内存消耗、线程数量、系统时间以及http会话数量。

## gague和counter的定制

gague和counter度量通过GagueService和CountService实例提供，这些实例可以导入到任何Spring管理的对象中，用于度量应用信息。例如，我们可以统计某个方法的调用次数，如果要统计所有RESTful接口的调用次数，则可以通过AOP实现，在调用指定的接口之前，首先调用counterService.increment("objectName.methodName.invoked");，某个方法被调用之后，则对它的统计值+1。具体的实验步骤如下：

- 在pom文件中添加AOP依赖

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

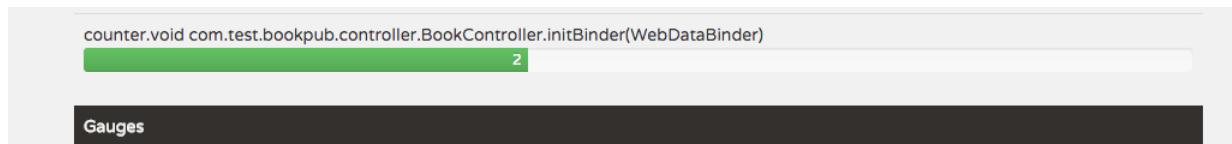
- 在BookPub应用中添加Aspect组件，表示在每个Controller的方法调用之前，首先增加调用次数。

```

@Aspect
@Component
public class ServiceMonitor {
    @Autowired
    private CounterService counterService;
    @Before("execution(* com.test.bookpub.controller.*.*(..))")
    public void countServiceInvoke(JoinPoint joinPoint) {
        counterService.increment(joinPoint.getSignature() + "");
    }
}

```

- 在application.properties中设置打开AOP功能: spring.aop.auto=true 然后启动BookPub应用，通过浏览器访问 <http://localhost:8080/books/9876-5432-1111>，然后再去Spring Boot Admin后台查看对应信息，发现该方法的调用次数已经被统计好了



图片 7.13 统计接口的调用次数

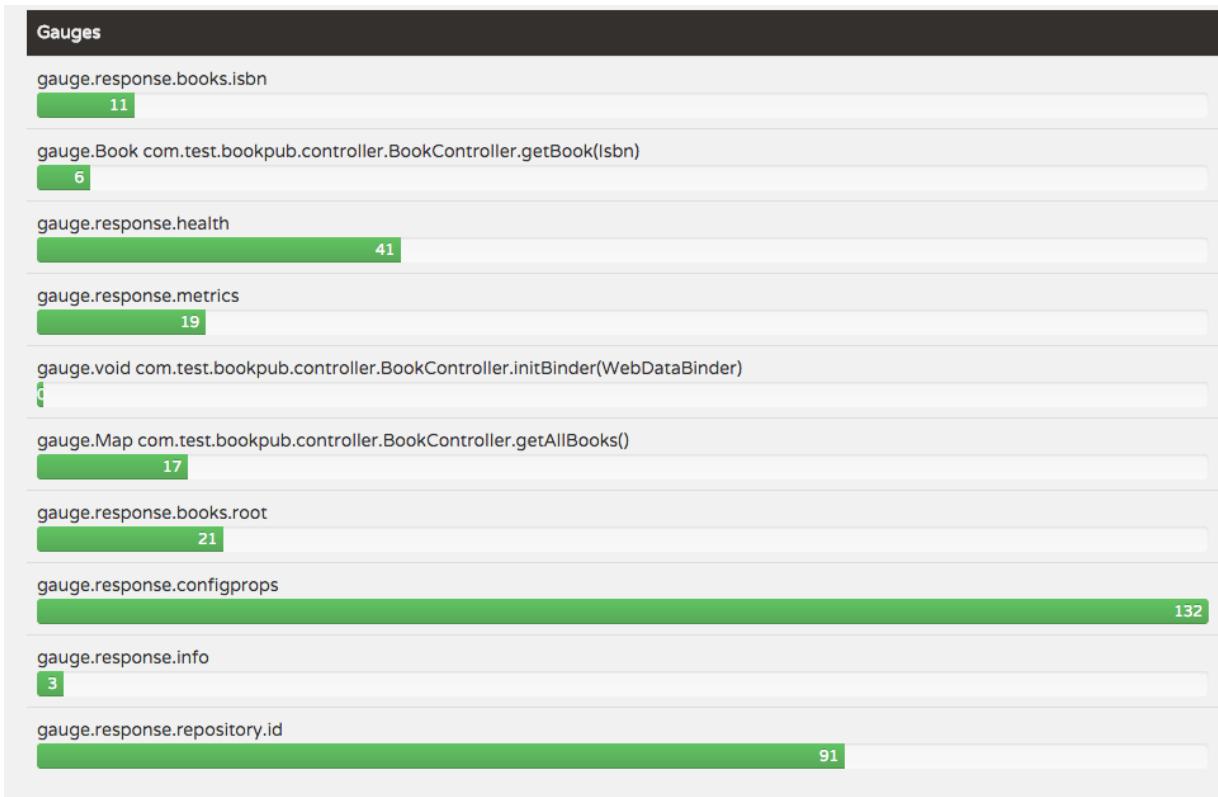
如果希望统计每个接口的调用时长，则需要借助GaugeService来实现，同样使用AOP实现，则需要环绕通知：在接口调用之前，利用long start = System.currentTimeMillis();，在接口调用之后，计算耗费的时间，单位是ms，然后使用gaugeService.submit(latency)更新该接口的调用延时。 – 在ServiceMonitor类中添加对应的监控代码

```

@Autowired
private GaugeService gaugeService;
@Around("execution(* com.test.bookpub.controller.*.*(..))")
public void latencyService(ProceedingJoinPoint pjp) throws Throwable {
    long start = System.currentTimeMillis();
    pjp.proceed();
    long end = System.currentTimeMillis();
    gaugeService.submit(pjp.getSignature().toString(), end - start);
}

```

- 然后在Spring Boot Admin后台可以看到对应接口的调用延迟



图片 7.14 统计接口的调用延时

这两个service可以应付大多数应用需求，如果需要监控其他的度量信息，则可以定制我们自己的Metrics，例如在之前的例子中我们要统计四个数据库接口的调用状态，则我们定义了`DbCountMetrics`，该类实现了`PublishMetrics`，在这个类中我们统计每个数据库接口的记录数量。

`PublishMetrics`这个接口只有一个方法：`Collection<Metric<?>> metrics();`，在该方法中定义具体的监控信息；该接口的实现类需要在配置文件中通过@Bean注解，让Spring Boot在启动过程中初始化，并自动注册到`MetricsEndpoint`处理器中，这样每次有访问`/metrics`的请求到来时，就会执行对应的`metrics`方法。

## 参考资料

1. [Chapter 6. 使用Spring进行面向切面编程（AOP）](#)

## 通过JMX监控Spring Boot应用

在[Spring Boot应用的健康监控](#)一文中，我们通过Spring Boot Actuator对外暴露应用的监控信息，除了使用HTTP获取JSON格式的数据之外，还可以通过JMX监控应用，Spring Boot也提供了对JMX监控的支持。JMX监控对外暴露的信息相同，不过是使用MBeans容器将应用数据封装管理。

接下来我们看下如何利用JMX获取应用状态信息，以及如何使用Jolokia JMX库对外暴露MBeans的HTTP访问URL。

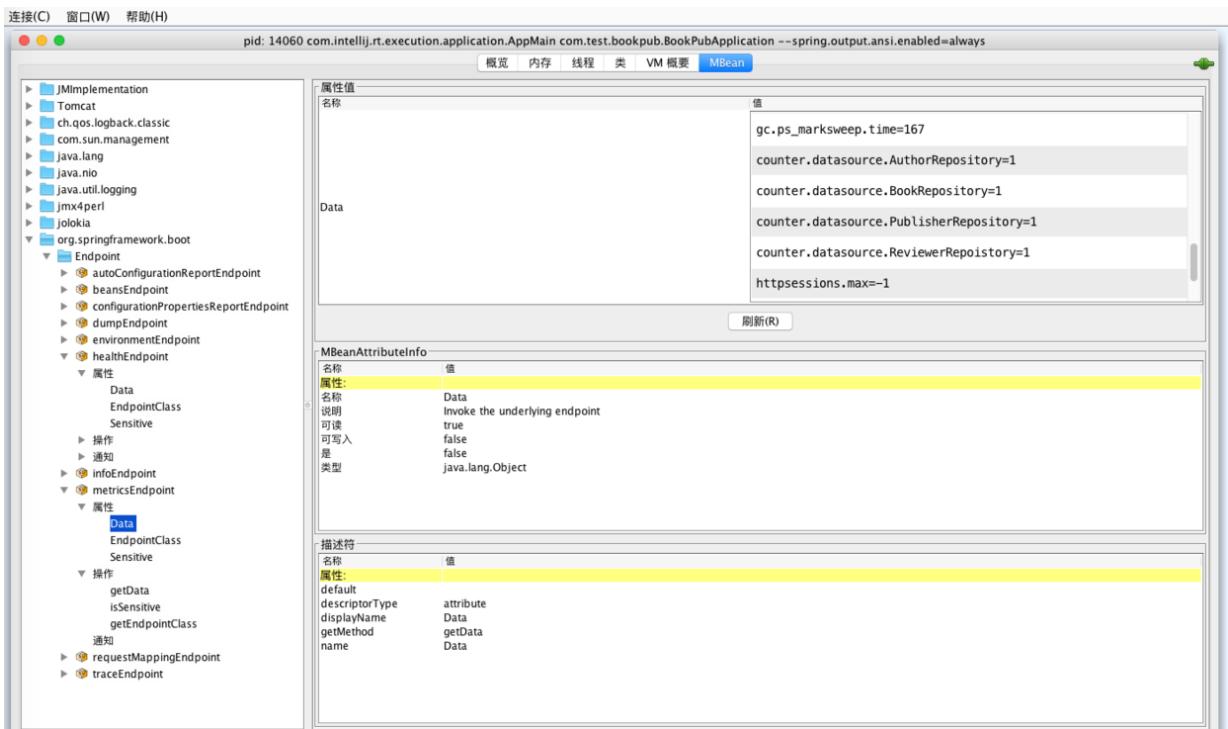
### Get Ready

在BookPub应用的pom文件中添加jolokia-core依赖

```
<!-- JMX monitor -->
<dependency>
    <groupId>org.jolokia</groupId>
    <artifactId>jolokia-core</artifactId>
</dependency>
```

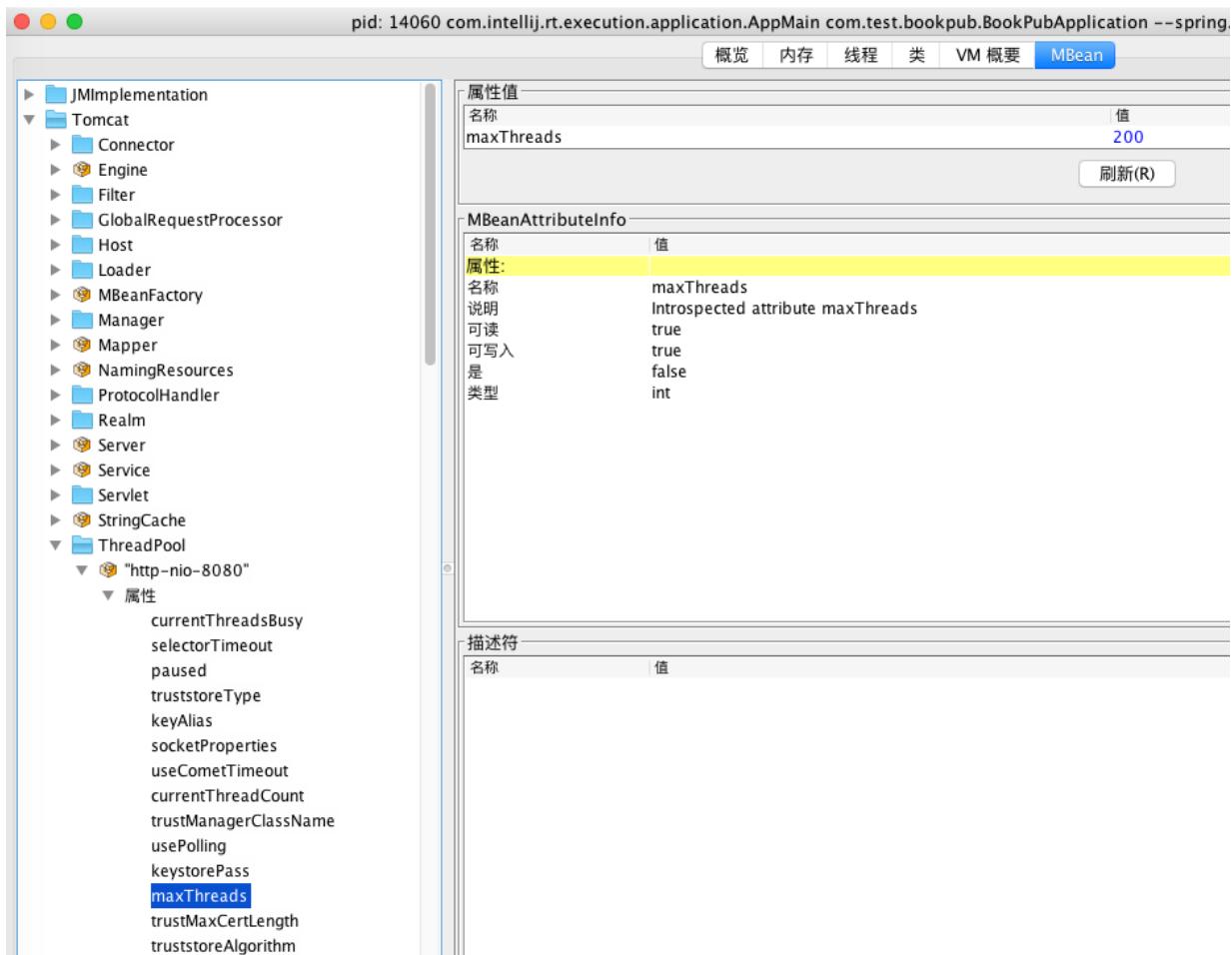
### How Do

- 启动BookPub应用，然后在命令行中执行 `jconsole` 命令启动“Java管理和监视控制台”，然后选择`org.springframework.boot`节点下的`Endpoint`，可以看到如下信息



图片 7.15 Java管理和监视控制台

- 在Tomcat节点下选择ThreadPool，然后在选择http-nio-8080节点，在这个节点下选择maxThreads属性，可以看到如下信息



图片 7.16 查看应用的最大并发线程数

3. 除了通过JMX获取信息，也暴露了对应的HTTP接口访问Mbeans对象的信息，例如，我们在postman中访问`http://localhost:8080/jolokia/read/Tomcat:type=ThreadPool, name=%22http-nio-8080%22/maxThreads`，也可以得到对应的信息

The screenshot shows a POST request in Postman. The URL is `http://localhost:8080/jolokia/read/Tomcat:type=ThreadPool, name=%22http-nio-8080%22/maxThreads`. The method is set to 'GET'. The response status is '200 OK' and the time taken is '63 ms'. The response body is a JSON object:

```

1 {
2   "request": {
3     "mbean": "Tomcat:name=\"http-nio-8080\",type=ThreadPool",
4     "attribute": "maxThreads",
5     "type": "read"
6   },
7   "value": 200,
8   "timestamp": 1451117165,
9   "status": 200
10 }

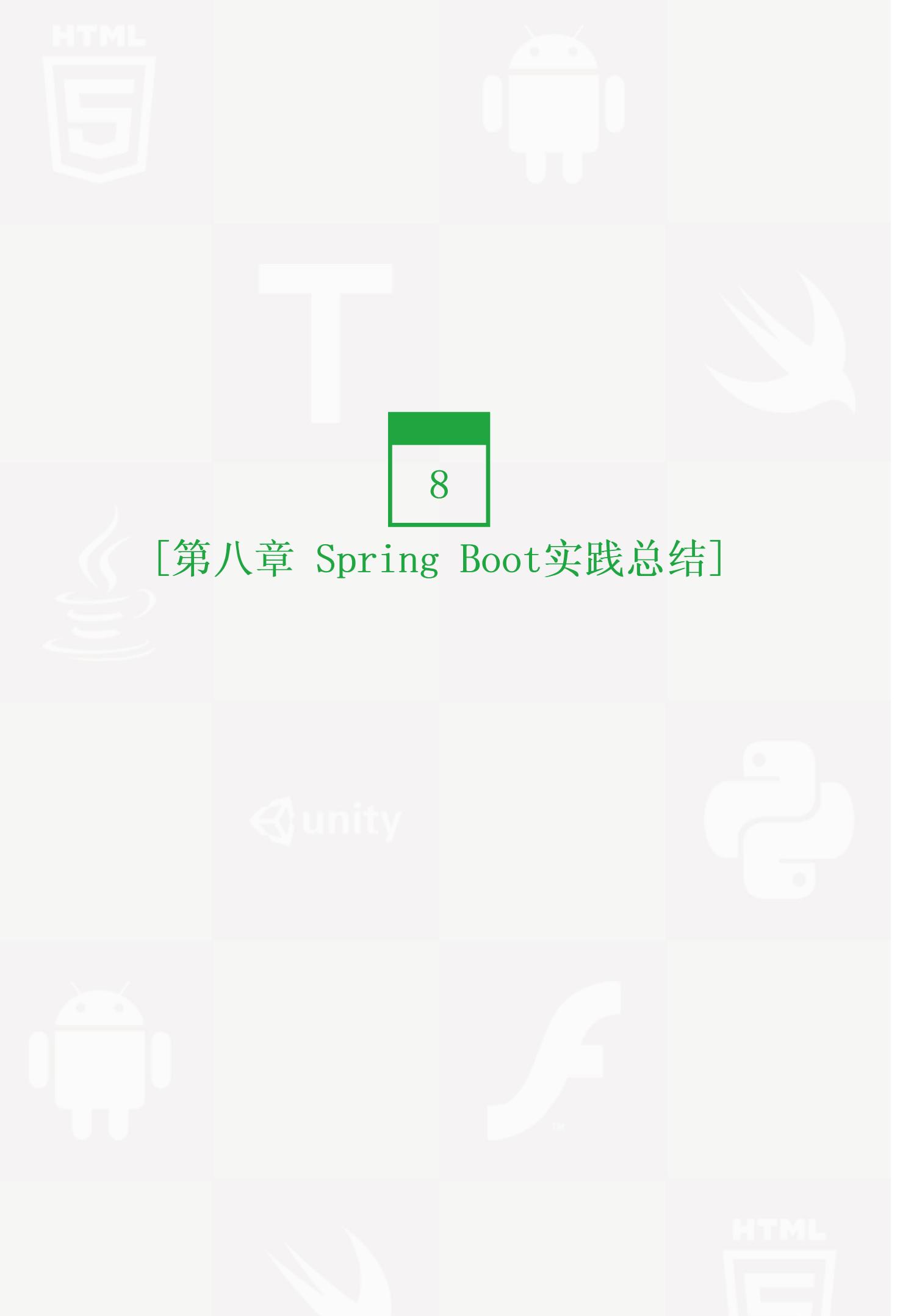
```

图片 7.17 通过HTTP访问应用的最大并发线程数

## 分析

只要添加了Spring Boot Actuator库，所有相关的endpoint和管理服务都打开了，包括JMX，我们可以通过设置`endpoints.jmx.enabled=false`禁止对外提供基于JMX的endpoints；或者通过设置`spring.jmx.enabled=false`禁止对外提供Spring MBeans。

在类路径中存在的Jolokia库会触发Spring Boot的*JolokiaAutoConfiguration*，这个自动配置类会自动配置可以接受/jolokia请求的*JolokiaMvcEndPoint*；也可以通过在application.properties中设置jolokia.config.系列的属性配置不同的Jolokia配置。完整的Jolokia配置地址在：<https://jolokia.org/reference/html/agents.html#agent-war-init-params>。如果你希望定制自己的Jolokia配置，则可以通过设置`endpoints.jolokia.enabled=false`让Spring Boot应用忽略自身提供的配置。



8

## [第八章 Spring Boot实践总结]

## Spring Boot with Mysql

---

Spring Boot大大简化了持久化任务，几乎不需要写SQL语句，之前我写过一篇关于Mongodb的——[RESTful:Spring Boot with Mongodb](#)。

本文将会演示如何在Spring Boot项目中使用mysql数据库。

### 1. 建立数据库连接 (database connection)

在上篇文章中我们新建了一个Spring Boot应用程序，添加了jdbc和data-jpa等starters，以及一个h2数据库依赖，这里我们将配置一个H2数据库。

对于H2、HSQL或者Derby这类嵌入型数据库，只要在pom文件中添加对应的依赖就可以，不需要额外的配置。当spring boot在classpath下发现某个数据库依赖存在且在代码中有关于Datasource Bean的定义时，就会自动创建一个数据库连接。我们通过修改之前的bookpub程序说明这个问题，需要修改StartupRunner.java文件，代码如下：

```
package org.test.bookpub;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;

import javax.sql.DataSource;

public class StartupRunner implements CommandLineRunner {
    protected static final Logger logger = LoggerFactory.getLogger(StartupRunner.class);

    @Autowired
    private DataSource ds;

    @Override
    public void run(String... strings) throws Exception {
        logger.info("Datasource: " + ds.toString());
    }
}
```

启动应用程序，可以看到如下输出：driverClassName=org.h2.Driver；因此，可以证明，Spring Boot根据我们自动织入DataSource的代码，自动创建并初始化了一个H2数据库。不过，这个数据库并没什么用，因为存放其中的数据会在系统停止后就丢失。通过修改配置，我们可以将数据存放在磁盘上。关于H2数据库的配置文件如下：

```
spring.datasource.url = jdbc:h2:~/test;DB_CLOSE_DELAY=-1;DB_CLOSE_ON_EXIT=FALSE
spring.datasource.username = sa
spring.datasource.password =
```

然后启动应用程序，并检查你的home目录下是否存在test.mv.db文件。通过“~/test”，就告诉Spring Boot，H2数据库的数据会存放在test.mv.db这个文件中。

综上，可以看出，Spring Boot试图通过spring.datasource分组下的一系列配置项来简化用户对数据库的使用，我们经常使用的配置项有：url，username，password以及driver-class-name等等。PS：driverClassName或者driver-class-name都可以，Spring Boot会在内部进行统一处理。

最常用的开源数据库是Mysql，在Spring Boot通过下列配置项来配置mysql：

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

如果希望通过Hibernate依靠Entity类自动创建数据库和数据表，则还需要加上配置项——`spring.jpa.hibernate.ddl-auto=create-drop`。PS：在生产环境中不要使用create-drop，这样会在程序启动时先删除旧的，再自动创建新的，最好使用update；还可以通过设置`spring.jpa.show-sql = true`来显示自动创建表的SQL语句，通过`spring.jpa.database = MYSQL`指定具体的数据，如果不明确指定Spring boot会根据classpath中的依赖项自动配置。

要使用MySQL，需要引入对应的connector，因此，首先在pom文件中添加如下依赖：

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

在Spring项目中，如果数据比较简单，我们可以考虑使用JdbcTemplate，而不是直接定义Datasource，配置jdbc的代码如下：

```
@Autowired
private JdbcTemplate jdbcTemplate;
```

只要定义了上面这个代码，Spring Boot会自动创建一个Datasource对象，然后再创建一个jdbcTemplate对象来管理datasource，通过jdbcTemplate操作数据库可以减少大量模板代码。如果你对SpringBoot的原理感兴趣，可以在org.springframework.boot.autoconfigure.jdbc.DataSourceAutoConfiguration类中查看其具体实现。

## 2. 创建数据仓库服务 (data repository service)

连接数据库并直接执行SQL语句这种思路非常古老，早在很多年前就已经出现了ORM（Object Relational Mapping）框架来简化这部分工作，最有名的是Hibernate，但是现在更火的好像是Mybatis。关于Spring Boot和Mybatis的整合，可以参考：[mybatis-spring-boot](#)。我们这里使用Hibernate进行演示。我们将会增加一些实体类，这些实体类决定了数据库的表结构，还要定义一个CrudRepository接口，用于操作数据。

示例程序是一个图书管理系统，显然，数据库中应该具备以下领域对象（domain object）：Book、Author、Reviewers以及Publisher。首先在src/main/java/org/test/bookpub下建立包domain，然后再在这个包下建立相应的实体类。具体代码列举如下（为了节省空间，省去了getter和setter）：

- Book.java

```
package com.test.bookpub.domain;

import javax.persistence.*;
import java.util.List;

@Entity
public class Book {
    @Id
    @GeneratedValue
    private Long id;
    private String isbn;
    private String title;
    private String description;

    @ManyToOne
    private Author author;
    @ManyToOne
    private Publisher publisher;

    @ManyToMany
    private List<Publisher.Reviewer> reviewers;

    protected Book() { }

    public Book(Author author, String isbn, Publisher publisher, String title) {
        this.author = author;
        this.isbn = isbn;
        this.publisher = publisher;
        this.title = title;
    }
}
```

```
    }  
}
```

- Author.java

```
package com.test.bookpub.domain;  
  
import javax.persistence.*;  
import java.util.List;  
  
@Entity  
public class Author {  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String firstName;  
    private String lastName;  
    @OneToMany(mappedBy = "author")  
    private List<Book> books;  
  
    protected Author() {  
    }  
  
    public Author(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
}
```

- Publisher.java

```
package com.test.bookpub.domain;  
  
import javax.persistence.*;  
import java.util.List;  
  
@Entity  
public class Publisher {  
    @Id  
    @GeneratedValue  
    private Long id;  
    private String name;  
  
    @OneToMany(mappedBy = "publisher")  
    private List<Book> books;
```

```

protected Publisher() { }

public Publisher(String name) {
    this.name = name;
}

@Entity
public class Reviewer {
    @Id
    @GeneratedValue
    private Long id;
    private String firstName;
    private String lastName;

    protected Reviewer() { }

    public Reviewer(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}
}

```

- repository层：创建完实体类，还需要创建BookRepository接口，该接口继承自CrudRepository，这个接口放在 src/main/java/com/test/bookpub/repository 包中，具体代码如下：

```

package com.test.bookpub.repository;

import com.test.bookpub.domain.Book;
import org.springframework.data.repository.CrudRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface BookRepository extends CrudRepository<Book, Long> {
    Book findBookByIsbn(String isbn);
}

```

- 织入BookRepository，最后需要再StartupRunner中定义BookRepository对象，并自动织入。

```

package com.test.bookpub;

import com.test.bookpub.repository.BookRepository;
import org.slf4j.Logger;

```

```

import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;

public class StartupRunner implements CommandLineRunner {
    protected final Logger logger = LoggerFactory.getLogger(StartupRunner.class);

    @Autowired
    private BookRepository bookRepository;

    @Override
    public void run(String... strings) throws Exception {
        logger.info("Number of books: " + bookRepository.count());
    }
}

```

可能读者朋友你也注意到了，到此为止，我们都没有写一行SQL语句，也没有在代码中涉及到数据库连接、建立查询等方面的内容。只有实体类上的各种注解表明我们在于数据库做交互：@Entity，@Repository，@Id，@GeneratedValue，@ManyToOne，@ManyToMany以及@OneToMany，这些注解属于Java Persistence API。我们通过CrudRepository接口的子接口与数据库交互，同时由Spring建立对象与数据库表、数据库表中的数据之间的映射关系。下面依次说明这些注解的含义和使用：

- @Entity，说明被这个注解修饰的类应该与一张数据库表相对应，表的名称可以由类名推断，当然了，也可以明确配置，只要加上`@Table(name = "books")`即可。需要特别注意，每个Entity类都应该有一个protected访问级别的无参构造函数，用于给Hibernate提供初始化的入口。
- @Id and @GeneratedValue：@Id注解修饰的属性应该作为表中的主键处理、@GeneratedValue修饰的属性应该由数据库自动生成，而不需要明确指定。
- @ManyToOne，@ManyToMany表明具体的数据存放在其他表中，在这个例子里，书和作者是多对一的关系，书和出版社是多对一的关系，因此book表中的author和publisher相当于数据表中的外键；并且在Publisher中通过@OneToMany（mapped = "publisher"）定义一个反向关联（1——>n），表明book类中的publisher属性与这里的books形成对应关系。
- @Repository 用来表示访问数据库并操作数据的接口，同时它修饰的接口也可以被component scan机制探测到并注册为bean，这样就可以在其他模块中通过@Autowired织入。
- CrudRepository，直接查看源代码，CrudRepository的代码如下：

```

public interface CrudRepository<T, ID extends Serializable>
    extends Repository<T, ID> {

    <S extends T> S save(S entity); //保存给定的entity
}

```

```

T findOne(ID primaryKey); //根据给定的id查询对应的entity

Iterable<T> findAll(); //查询所有entity

Long count(); //返回entity的个数

void delete(T entity); //删除给定的entity

boolean exists(ID primaryKey); //判断给定id的entity是否存在

// ... more functionality omitted.
}

```

我们可以添加自定义的接口函数，JPA会提供对应的SQL查询，例如，在本例中的BookRepository中可以增加findBookByIsbn(String isbn)函数，JPA会自动创建对应的SQL查询——根据isbn查询图书，这种将方法名转换为SQL语句的机制十分方便且功能强大，例如你可以增加类似findByNameIgnoringCase(String name)这种复杂查询。

最后，我们利用 `mvn spring-boot:run` 运行应用程序，观察下Hibernate是如何建立数据库连接，如何检测数据表是否存在以及如何自动创建表的过程。

```

- [zsh] .../ads/bookpub [zsh]
2015-12-01 16:42:53.215 INFO 13458 --- [main] o.hibernate.jpa.internal.util.LogHelper : HHH000204: Processing PersistenceUnitInfo [
    name: default
    ...
]
2015-12-01 16:42:53.353 INFO 13458 --- [main] org.hibernate.Version : HHH000412: Hibernate Core {4.3.11.Final}
2015-12-01 16:42:53.355 INFO 13458 --- [main] org.hibernate.cfg.Environment : HHH000206: hibernate.properties not found
2015-12-01 16:42:53.358 INFO 13458 --- [main] org.hibernate.cfg.Environment : HHH000021: Bytecode provider name : javassist
2015-12-01 16:42:53.666 INFO 13458 --- [main] org.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations {4.0.5.Final}
2015-12-01 16:42:54.355 INFO 13458 --- [main] org.hibernate.dialect.Dialect : HHH000400: Using dialect: org.hibernate.dialect.MySQL5Dialect
2015-12-01 16:42:54.553 INFO 13458 --- [main] o.h.h.i.ast.ASTQueryTranslatorFactory : HHH000397: Using ASTQueryTranslatorFactory
2015-12-01 16:42:54.715 INFO 13458 --- [main] org.hibernate.tuple.PojoInstantiator : HHH000182: No default (no-argument) constructor for class: com.test.bookpub.domain.Publisher$Reviewer (class must be instantiated by Interceptor)
2015-12-01 16:42:54.901 INFO 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : HHH000227: Running hbm2ddl schema export
Hibernate: alter table book drop foreign key FK_rai096pq6coaklief212hpi3d
2015-12-01 16:42:54.919 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : HHH000389: Unsuccessful: alter table book drop foreign key FK_rai096pq6coaklief212hpi3d
2015-12-01 16:42:54.919 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : Table 'springbootcookbook.book' doesn't exist
Hibernate: alter table book drop foreign key FK_1xigoeftluhur7kxne9udlvf
2015-12-01 16:42:54.926 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : HHH000389: Unsuccessful: alter table book drop foreign key FK_1xigoeftluhur7kxne9udlvf
2015-12-01 16:42:54.926 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : Table 'springbootcookbook.book' doesn't exist
Hibernate: alter table book_reviewers drop foreign key FK_2quc75ksvstox5xoso7fekc2p
2015-12-01 16:42:54.921 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : HHH000389: Unsuccessful: alter table book_reviewers drop foreign key FK_2quc75ksvstox5xoso7fekc2p
2015-12-01 16:42:54.922 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : Table 'springbootcookbook.book_reviewers' doesn't exist
Hibernate: alter table book_reviewers drop foreign key FK_8tskt84he1c0us9wbu9vc8cid
2015-12-01 16:42:54.924 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : HHH000389: Unsuccessful: alter table book_reviewers drop foreign key FK_8tskt84he1c0us9wbu9vc8cid
2015-12-01 16:42:54.925 ERROR 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : Table 'springbootcookbook.book_reviewers' doesn't exist
Hibernate: drop table if exists author
Hibernate: drop table if exists book
Hibernate: drop table if exists book_reviewers
Hibernate: drop table if exists publisher
Hibernate: drop table if exists publisher$reviewer
Hibernate: create table author (id bigint not null auto_increment, first_name varchar(255), last_name varchar(255), primary key (id))
Hibernate: create table book (id bigint not null auto_increment, description varchar(255), isbn varchar(255), title varchar(255), author bigint, publisher bigint, primary key (id))
Hibernate: create table book_reviewers (book bigint not null, reviewers bigint not null)
Hibernate: create table publisher (id bigint not null auto_increment, name varchar(255), primary key (id))
Hibernate: create table publisher$reviewer (id bigint not null auto_increment, first_name varchar(255), last_name varchar(255), primary key (id))
Hibernate: alter table book add constraint FK_rai096pq6coaklief212hpi3d foreign key (author) references author (id)
Hibernate: alter table book add constraint FK_1xigoeftluhur7kxne9udlvf foreign key (publisher) references publisher (id)
Hibernate: alter table book_reviewers add constraint FK_2quc75ksvstox5xoso7fekc2p foreign key (reviewers) references publisher$reviewer (id)
Hibernate: alter table book_reviewers add constraint FK_8tskt84he1c0us9wbu9vc8cid foreign key (book) references book (id)
2015-12-01 16:42:55.479 INFO 13458 --- [main] org.hibernate.tool.hbm2ddl.SchemaExport : HHH000230: Schema export complete
2015-12-01 16:42:56.157 INFO 13458 --- [main] o.s.j.e.a.AnnotationMBeanExporter : Registering beans for JMX exposure on startup
Hibernate: select count(*) as col_0_0_ from book book0_
2015-12-01 16:42:56.405 INFO 13458 --- [main] com.test.bookpub.StartupRunner : Number of books: 0
2015-12-01 16:42:56.408 INFO 13458 --- [main] com.test.bookpub.BookPubApplication : Started BookPubApplication in 5.477 seconds (JVM running for 10.305)

```

图片 8.1 spring with mysql

### 3. 参考资料

- <http://docs.spring.io/spring-data/data-commons/docs/current/reference/html/>

## Restful: Spring Boot with Mongodb

---

### 为什么是mongodb?

继续之前的dailyReport项目，今天的任务是选择mongodb作为持久化存储。

关于nosql和rdbms的对比以及选择，我参考了不少资料，关键一点在于：nosql可以轻易扩展表的列，对于业务快速变化的应用场景非常适合；rdbms则需要安装关系型数据库模式对业务进行建模，适合业务场景已经成熟的系统。我目前的这个项目——dailyReport，我暂时没法确定的是，对于一个report，它的属性应该有哪些：date、title、content、address、images等等，基于此我选择mongodb作为该项目的持久化存储。

### 如何将mongodb与spring boot结合使用

- 修改Pom文件，增加mongodb支持

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

- 重新设计Report实体类，id属性是给mongodb用的，用@Id注解修饰；重载toString函数，使用String.format输出该对象。

```
import org.springframework.data.annotation.Id;

/**
 * @author duqi
 * @create 2015-11-17 19:31
 */
public class Report {

    @Id
    private String id;

    private String date;
    private String content;
    private String title;

    public Report() {
```

```
}

public Report(String date, String title, String content) {
    this.date = date;
    this.title = title;
    this.content = content;
}

public String getId() {
    return id;
}

public void setId(String id) {
    this.id = id;
}

public String getTitle() {
    return title;
}

public void setTitle(String title) {
    this.title = title;
}

public String getDate() {
    return date;
}

public void setDate(String dateStr) {
    this.date = dateStr;
}

public String getContent() {
    return content;
}

public void setContent(String content) {
    this.content = content;
}

@Override
public String toString() {
    return String.format("Report[id=%s, date=%s', content=%s', title=%s']", id, date, content, title);
}
```

- 增加ReportRepository接口，它继承自MongoRepository接口，MongoRepository接口包含了常用的CRUD操作，例如：save、insert、findAll等等。我们可以定义自己的查找接口，例如根据report的title搜索，具体的ReportRepository接口代码如下：

```
import org.springframework.data.mongodb.repository.MongoRepository;

import java.util.List;

/**
 * Created by duqi on 15/11/22.
 */
public interface ReportRepository extends MongoRepository<Report, String> {
    Report findByTitle(String title);
    List<Report> findByDate(String date);
}
```

- 修改ReportService代码，增加createReport函数，该函数根据Controller传来的Map参数初始化一个Report对象，并调用ReportRepository将数据save到mongodb中；对于getReportDetails函数，仍然开启缓存，如果没有缓存的时候则利用findByTitle接口查询mongodb数据库。

```
import com.javadu.dailyReport.domain.Report;
import com.javadu.dailyReport.domain.ReportRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

import java.util.Map;

/**
 * @author duqi
 * @create 2015-11-17 20:05
 */

@Service
public class ReportService {

    @Autowired
    private ReportRepository repository;

    public Report createReport(Map<String, Object> reportMap) {
        Report report = new Report(reportMap.get("date").toString(),
            reportMap.get("title").toString(),
            reportMap.get("content").toString());
    }
}
```

```

        repository.save(report);
        return report;
    }

    @Cacheable(value = "reportcache", keyGenerator = "wiselyKeyGenerator")
    public Report getReportDetails(String title) {
        System.out.println("无缓存的时候调用这里---数据库查询, title=" + title);
        return repository.findByTitle(title);
    }
}

```

## Restful接口

Controller只负责URL到具体Service的映射，而在Service层进行真正的业务逻辑处理，我们这里的业务逻辑异常简单，因此显得Service层可有可无，但是如果业务逻辑复杂起来（比方说要通过RPC调用一个异地服务），这些操作都需要再service层完成。总体来讲，Controller层只负责：转发请求 + 构造Response数据；在需要进行权限验证的时候，也在Controller层利用aop完成。

一般将对于Report（某个实体）的所有操作放在一个Controller中，并用@RestController和@RequestMapping("/report")注解修饰，表示所有xxxx/report开头的URL会由这个ReportController进行处理。

### . POST

对于增加report操作，我们选择POST方法，并使用@RequestBody修饰POST请求的请求体，也就是createReport函数的参数；

### . GET

对于查询report操作，我们选择GET方法，URL的形式是：“xxx/report/\${report's title}”，使用@PathVariable修饰url输入的参数，即title。

```

import com.javadu.dailyReport.domain.Report;
import com.javadu.dailyReport.service.ReportService;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.LinkedHashMap;
import java.util.Map;

/**
 * @author duqi

```

```

* @create 2015-11-17 20:10
*/

```

```

@RestController
@RequestMapping("/report")
public class ReportController {
    private static final Logger logger = LoggerFactory.getLogger(ReportController.class);

    @Autowired
    ReportService reportService;

    @RequestMapping(method = RequestMethod.POST)
    public Map<String, Object> createReport(@RequestBody Map<String, Object> reportMap) {
        logger.info("createReport");
        Report report = reportService.createReport(reportMap);

        Map<String, Object> response = new LinkedHashMap<String, Object>();
        response.put("message", "Report created successfully");
        response.put("report", report);

        return response;
    }

    @RequestMapping(method = RequestMethod.GET, value = "/{reportTitle}")
    public Report getReportDetails(@PathVariable("reportTitle") String title) {
        logger.info("getReportDetails");
        return reportService.getReportDetails(title);
    }
}

```

Update和delete操作我这里就不一一讲述了，留个读者作为练习

## 参考资料

1. [sql vs nosql: what you need to know](#)
2. [Accessing data with Mongodb](#)
3. [Spring Boot: Restful API using Spring Boot and Mongodb](#)

## Spring Boot with Redis

---

### Spring Boot简介

Spring Boot是为了简化Spring开发而生，从Spring 3.x开始，Spring社区的发展方向就是弱化xml配置文件而加大注解的戏份。最近召开的SpringOne2GX2015大会上显示：Spring Boot已经是Spring社区中增长最迅速的框架，前三名是：Spring Framework，Spring Boot和Spring Security，这个应该是未来的趋势。

我学习Spring Boot，是因为通过cli工具，spring boot开始往flask（python）、express（nodejs）等web框架发展和靠近，并且Spring Boot几乎不需要写xml配置文件。感兴趣的同学可以根据[spring boot quick start](#)这篇文章中的例子尝试下。

学习新的技术最佳途径是看官方文档，现在Spring boot的release版本是1.3.0-RELEASE，相应的参考文档是[Spring Boot Reference Guide \(1.3.0-RELEASE\)](#)，如果有绝对英文比较吃力的同学，可以参考中文版[Spring Boot参考指南](#)。在前段时间阅读一篇技术文章，介绍如何阅读ios技术文档，我从中也有所收获，那就是我们应该重视spring.io上的guides部分——[Getting Started Guides](#)，这部分都是一些针对特定问题的demo，值得学习。

### Spring Boot的项目结构

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

如上所示，Spring boot项目的结构划分为web->service->domain，其中domain文件夹可类比与业务模型和数据存储，即xxxBean和Dao层；service层是业务逻辑层，web是控制器。比较特别的是，这种类型的项目有自己的入口，即主类，一般命名为Application.java。Application.java不仅提供入口功能，还提供一些底层服务，例如缓存、项目配置等等。

## 例子介绍

本文的例子是取自我的side project之中，日报（report）的查询，试图利用Redis作为缓存，优化查询效率。

## 知识点解析

### 1. 自定义配置

Spring Boot允许外化配置，这样你可以在不同的环境下使用相同的代码。你可以使用properties文件、yaml文件，环境变量和命令行参数来外化配置。使用@Value注解，可以直接将属性值注入到你的beans中。Spring Boot使用一个非常特别的PropertySource来允许对值进行合理的覆盖，按照优先考虑的顺序排位如下：

1. 命令行参数
2. 来自java:comp/env的JNDI属性
3. Java系统属性（System.getProperties()）
4. 操作系统环境变量
5. 只有在random.\*里包含的属性会产生一个RandomValuePropertySource
6. 在打包的jar外的应用程序配置文件（application.properties, 包含YAML和profile变量）
7. 在打包的jar内的应用程序配置文件（application.properties, 包含YAML和profile变量）
8. 在@Configuration类上的@PropertySource注解
9. 默认属性（使用SpringApplication.setDefaultProperties指定）

**使用场景：**可以将一个application.properties打包在Jar内，用来提供一个合理的默认name值；当运行在生产环境时，可以在Jar外提供一个application.properties文件来覆盖name属性；对于一次性的测试，可以使用特病的命令行开关启动，而不需要重复打包jar包。

具体的例子操作过程如下：

- 新建配置文件（application.properties）

```
spring.redis.database=0
spring.redis.host=localhost
spring.redis.password= # Login password of the redis server.
spring.redis.pool.max-active=8
spring.redis.pool.max-idle=8
spring.redis.pool.max-wait=-1
spring.redis.pool.min-idle=0
spring.redis.port=6379
spring.redis.sentinel.master= # Name of Redis server.
spring.redis.sentinel.nodes= # Comma-separated list of host:port pairs.
spring.redis.timeout=0
```

- 使用@PropertySource引入配置文件

```
@Configuration
@PropertySource(value = "classpath:/redis.properties")
@EnableCaching
public class CacheConfig extends CachingConfigurerSupport {
    .....
}
```

- 使用@Value引用属性值

```
@Configuration
@PropertySource(value = "classpath:/redis.properties")
@EnableCaching
public class CacheConfig extends CachingConfigurerSupport {
    @Value("${spring.redis.host}")
    private String host;
    @Value("${spring.redis.port}")
    private int port;
    @Value("${spring.redis.timeout}")
    private int timeout;
    .....
}
```

## 2. redis使用

- 添加pom配置

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

- 编写CacheConfig

```
@Configuration
@PropertySource(value = "classpath:/redis.properties")
@EnableCaching
public class CacheConfig extends CachingConfigurerSupport {
    @Value("${spring.redis.host}")
    private String host;
    @Value("${spring.redis.port}")
    private int port;
    @Value("${spring.redis.timeout}")
```

```

private int timeout;

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

@Bean
public JedisConnectionFactory redisConnectionFactory() {
    JedisConnectionFactory factory = new JedisConnectionFactory();
    factory.setHostName(host);
    factory.setPort(port);
    factory.setTimeout(timeout); //设置连接超时时间
    return factory;
}

@Bean
public CacheManager cacheManager(RedisTemplate redisTemplate) {
    RedisCacheManager cacheManager = new RedisCacheManager(redisTemplate);
    // Number of seconds before expiration. Defaults to unlimited (0)
    cacheManager.setDefaultExpiration(10); //设置key-value超时时间
    return cacheManager;
}

@Bean
public RedisTemplate<String, String> redisTemplate(RedisConnectionFactory factory) {
    StringRedisTemplate template = new StringRedisTemplate(factory);
    setSerializer(template); //设置序列化工具，这样ReportBean不需要实现Serializable接口
    template.afterPropertiesSet();
    return template;
}

private void setSerializer(StringRedisTemplate template) {
    Jackson2JsonRedisSerializer jackson2JsonRedisSerializer = new Jackson2JsonRedisSerializer(Object.class);
    ObjectMapper om = new ObjectMapper();
    om.setVisibility(PropertyAccessor.ALL, JsonAutoDetect.Visibility.ANY);
    om.enableDefaultTyping(ObjectMapper.DefaultTyping.NON_FINAL);
    jackson2JsonRedisSerializer.setObjectMapper(om);
    template.setValueSerializer(jackson2JsonRedisSerializer);
}

```

```

    }
}
}
```

- 启动缓存，使用@Cacheable注解在需要缓存的接口上即可

```

@Service
public class ReportService {
    @Cacheable(value = "reportcache", keyGenerator = "wiselyKeyGenerator")
    public ReportBean getReport(Long id, String date, String content, String title) {
        System.out.println("无缓存的时候调用这里---数据库查询");
        return new ReportBean(id, date, content, title);
    }
}
```

- 测试验证

- 运行方法如下：
  - mvn clean package
  - java -jar target/dailyReport-1.0-SNAPSHOT.jar
- 验证缓存起作用：
  - 访问：http://localhost:8080/report/test
  - 访问：http://localhost:8080/report/test2
- 验证缓存失效（10s+后执行）：
  - 访问：http://localhost:8080/report/test2

## 参考资料

- [spring boot quick start](#)
- [Spring Boot参考指南](#)
- [Spring Boot Reference Guide\(1.3.0-RELEASE\)](#)
- [Getting Started Guides](#)
- [Caching Data in Spring Using Redis](#)
- [Spring boot使用Redis做缓存](#)
- [redis设计与实现](#)



中国最大的IT职业在线教育平台



更多信息请访问

<http://wiki.jikexueyuan.com/project/spring-boot/>