InfoQ

- En
- 中文
- 日本
- Fr
- Br

966,690 四月 独立访问用户

- 语言 & 开发
  - Java
  - Clojure
  - Scala
  - .Net
  - 移动
  - Android
  - iOS
  - HTML 5
  - JavaScript
  - 函数式编程
  - Web API

特别专题 语言 & 开发

分布式MySQL集群方案的探索与思考


本文整理自ArchSummit微信大讲堂张成远线上群分享内容。

浏览所有 **语言 & 开发**

- 架构 & 设计
  - 架构
  - 企业架构

- 性能和可伸缩性
- 设计
- 案例分析
- 设计模式
- 安全

特别专题 架构 & 设计

分布式MySQL集群方案的探索与思考


本文整理自ArchSummit微信大讲堂张成远线上群分享内容。

浏览所有 **架构 & 设计**

- 数据科学
  - 大数据
  - NoSQL
  - 数据库

特别专题 数据科学

架构师特刊：Hadoop十年回顾

Hadoop于2006年1月28日诞生，至今已有10年，它改变了企业对数据的存储、处理和分析的过程，加速了大数据的发展，形成了自己的极其火爆的技术生态圈，并受到非常广泛的应用。在2016年Hadoop十岁生日之际，InfoQ策划了一个Hadoop热点系列文章，为大家梳理Hadoop这十年的变化，技术圈的生态状况，回顾以前，激励以后。

浏览所有 **数据科学**

- 文化 & 方法
  - Agile
  - 领导能力
  - 团队协作
  - 测试
  - 用户体验
  - Scrum
  - 精益

特别专题 文化 & 方法

IT业也是制造业，日本IT产业奇葩在哪儿

以实业立国，日本IT产业模式跟中国有诸多不同。日本IT产业奇葩在哪儿？让在日工作的InfoQ社区编辑为你解答。

浏览所有 **文化 & 方法**

- DevOps
  - 持续交付
  - 自动化操作
  - 云计算

特别专题 DevOps

架构师（2016年5月）

本期主要内容：如何在云平台构建大规模分布式系统，携程移动App架构优化之旅，10亿红包从天而降，揭秘微信摇一摇背后的技术细节，云计算时代来了，没有狂欢盛宴只有整个IT业的呜咽

浏览所有 **DevOps**

架构
移动
Docker
云计算
大数据
架构师
运维
QCon
ArchSummit
AWS
腾讯云
活动大本营

深入学习微框架：Spring Boot

作者 Dan Woods ，译者 张卫滨 发布于 2014年5月13日 | **注意:**GMTC全球移动技术大会2016年6月24-25日，了解更多详情！ 5 讨论

Spring Boot是由Pivotal团队提供的全新框架，其设计目的是用来简化新Spring应用的初始搭建以及开发过程。该框架使用了特定的方式来进行配置，从而使开发人员不再需要定义样板化的配置。通过这种方式，Boot致力于在蓬勃发展的快速应用开发领域（rapid application development）成为领导者。

多年以来，Spring IO平台饱受非议的一点就是大量的XML配置以及复杂的依赖管理。在去年的SpringOne 2GX会议上，Pivotal的CTO Adrian Colyer回应了这些批评，并且特别提到该平台将来的目标之一就是实现免XML配置的开发体验。Boot所实现的功能超出了这个任务的描述，开发人员不仅不再需要编写XML，而且在一些场景中甚至不需要编写繁琐的import语句。在对外公开的beta版本刚刚发布之时，Boot描述了如何使用该框架在140个字符内实现可运行的web应用，从而获得了极大的关注度，该样例发表在tweet上。

然而，Spring Boot并不是要成为Spring IO平台里面众多“Foundation”层项目的替代者。Spring Boot的目标不在于为已解决的问题域提供新的解决方案，而是为平台带来另一种开发体验，从而简化对这些已有技术的使用。对于已经熟悉Spring生态系统的开发人员来说，Boot是一个很理想的选择，不过对于采用Spring技术的新人来说，Boot提供一种更简洁的方式来使用这些技术。

在追求开发体验的提升方面，Spring Boot，甚至可以说整个Spring生态系统都使用到了Groovy编程语言。Boot所提供的众多便捷功能，都是借助于Groovy强大的MetaObject协议、可插拔的AST转换过程以及内置的依赖解决方案引擎所实现的。在其核心的编译模型之中，Boot使用Groovy来构建工程文件，所以它可以使用通用的导入和样板方法（如类的main方法）对类所生成的字节码进行装饰（decorate）。这样使用Boot编写的应用就能保持非常简洁，却依然可以提供众多的功能。

安装Boot

从最根本上来讲，Spring Boot就是一些库的集合，它能够被任意项目的构建系统所使用。简便起见，该框架也提供了命令行界面，它可以用来运行和测试Boot应用。框架的发布版本，包括集成的CLI（命令行界面），可以在Spring仓库中手动下载和安装。一种更为简便的方式是使用Groovy环境管理器（Groovy enVironment Manager，GVM），它会处理Boot版本的安装和管理。Boot及其CLI可以通过GVM的命令行`gvm install springboot`进行安装。在OS X上安装Boot可以使用Homebrew包管理器。为了完成安装，首先要使用`brew tap pivotal/tap`切换到Pivotal仓库中，然后执行`brew install springboot`命令。

要进行打包和分发的工程会依赖于像Maven或Gradle这样的构建系统。为了简化依赖图，Boot的功能是模块化的，通过导入Boot所谓的“starter”模块，可以将许多的依赖添加到工程之中。为了更容易地管理依赖版本和使用默认配置，框架提供了一个parent POM，工程可以继承它。Spring Boot工程的样例POM文件定义如程序清单1所示。

程序清单1

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

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <!-- Inherit defaults from Spring Boot -->
```

```
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.0.0.RC1</version>
    </parent>

    <!-- Add typical dependencies for a web application -->
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-actuator</artifactId>
        </dependency>
    </dependencies>

    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/libs-snapshot</url>
        </repository>
    </repositories>

    <pluginRepositories>
        <pluginRepository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/libs-snapshot</url>
        </pluginRepository>
    </pluginRepositories>

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

为了实现更为简单的构建配置，开发人员可以使用Gradle构建系统中简洁的Groovy DSL，如程序清单1.1所示。

程序清单1.1

```
buildscript {
  repositories {
    maven { url "http://repo.spring.io/libs-snapshot" }
    mavenCentral()
  }
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.0.RC1")
  }
}

apply plugin: 'java'
apply plugin: 'spring-boot'

repositories {
  mavenCentral()
  maven { url "http://repo.spring.io/libs-snapshot"  }
}

dependencies {
  compile 'org.springframework.boot:spring-boot-starter-actuator:1.0.0.RC1'
}
```

为了快速地搭建和运行Boot工程，Pivotal提供了称之为“Spring Initializr” 的web界面，用于下载预先定义好的Maven或Gradle构建配置。我们也可以使用Lazybones模板实现快速起步，在执行`lazybones create spring-boot-actuator my-app`命令后，它会为Boot应用创建必要的工程结构以及gradle构建文件。

开发Spring Boot应用

Spring Boot在刚刚公开宣布之后就将一个样例发布到了Twitter上，它目前成为了最流行的一个应用样例。它的全部描述如程序清单1.2所示，一个非常简单的Groovy文件可以生成功能强大的以Spring为后端的web应用。

程序清单1.2

```
@RestController
class App {
  @RequestMapping("/")
  String home() {
    "hello"
  }
```

```
}
```

这个应用可以通过`spring run App.groovy`命令在Spring Boot CLI中运行。Boot会分析文件并根据各种“编译器自动配置（compiler auto-configuration）”标示符来确定其意图是生成Web应用。然后，它会在一个嵌入式的Tomcat中启动Spring应用上下文，并且使用默认的8080端口。打开浏览器并导航到给定的URL，随后将会加载一个页面并展现简单的文本响应：“hello”。提供默认应用上下文以及嵌入式容器的这些过程，能够让开发人员更加关注于开发应用以及业务逻辑，从而不用再关心繁琐的样板式配置。

Boot能够自动确定类所需的功能，这一点使其成为了强大的快速应用开发工具。当应用在Boot CLI中执行时，它们在使用内部的Groovy编译器进行构建，这个编译器可以在字节码生成的时候以编码的方式探查并修改类。通过这种方式，使用CLI的开发人员不仅可以省去定义默认配置，在一定程度上甚至可以不用定义特定的导入语句，它们可以在编译的过程中识别出来并自动进行添加。除此之外，当应用在CLI中运行时，Groovy内置的依赖管理器，“Grape”，将会解析编译期和运行时的类路径依赖，与Boot编译器的自动配置机制类似。这种方式不仅使得框架更加对用户友好，而且能够让不同版本的Spring Boot与特定版本的来自于Spring IO平台的库相匹配，这样一来开发人员就不用关心如何管理复杂的依赖图和版本结构了。另外，它还有助于快速原型的开发并生成概念原型的工程代码。

对于不是使用CLI构建的工程，Boot提供了许多的“starter”模块，它们定义了一组依赖，这些依赖能够添加到构建系统之中，从而解析框架及其父平台所需的特定类库。例如，`spring-boot-starter-actuator`依赖会引入一组基本的Spring项目，从而实现应用的快速配置和即时可用。关于这种依赖，值得强调的一点就是当开发Web应用，尤其是RESTful Web服务的时候，如果包含了`spring-boot-starter-web`依赖，它就会为你提供启动嵌入式Tomcat容器的自动化配置，并且提供对微服务应用有价值的端点信息，如服务器信息、应用指标（metrics）以及环境详情。除此之外，如果引入`spring-boot-starter-security`模块的话，actuator会自动配置Spring Security，从而为应用提供基本的认证以及其他高级的安全特性。它还会为应用结构引入一个内部的审计框架，这个框架可以用来生成报告或其他的用途，比如开发认证失败的锁定策略。

为了阐述在Java Maven工程中，如何快速地使Spring Web工程准备就绪，考虑一下程序清单1.3中的应用程序代码。

程序清单1.3

```
package com.infoq.springboot;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.web.bind.annotation.*;

@RestController
@EnableAutoConfiguration
public class Application {

  @RequestMapping("/")
  public String home() {
    return "Hello";
  }
```

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

在Application类上的@EnableAutoConfiguration注解会告知Boot要采用一种特定的方式来对应用进行配置。这种方法会将其他样板式的配置均假设为框架默认的约定，因此能够聚焦于如何尽快地使应用准备就绪以便运行起来。Application类是可运行的，因此，当我们以Java Application的方式运行这个类时，就能启动该应用及其嵌入式的容器，这样也能实现即时地开发。

为了发布版本而构建工程时，Boot的Maven和Gradle插件可以嵌入（hook）到这些构建系统的打包过程中，以生成可执行的“胖jar包（fat jar）”，这种jar包含了工程的所有依赖并且能够以可运行jar的方式执行。使用Maven打包Boot应用只需运行mvn package命令，与之类似，使用Gradle时，执行gradle build命令将会在构建的目标地址下生成可运行的jar。

开发微服务

Boot对Spring应用的开发进行了简化，提供了模块化方式导入依赖的能力，强调了开发RESTful Web服务的功能并提供了生成可运行jar的能力，这一切都清晰地表明在开发可部署的微服务方面Boot框架是一个强大的工具。正如前面的例子所示，借助于Boot，让一个RESTful Web工程运行起来是一件很容易的事情；不过，为了了解Boot所有潜在的功能，我们会阐述在开发完整功能的微服务时，会遇到的所有繁琐的事情。在企业级基础设施领域，微服务是一种越来越流行的应用架构，因为它能够实现快速开发、更小的代码库、企业级集成以及模块化部署。有众多的框架致力于该领域的开发，该章节将会讨论使用Boot如何简化这一过程。

数据访问

我们可以基于各种目的来构建微服务，但有一点是肯定的，那就是大多数都需要读取和写入数据库的能力。Spring Boot使数据库集成变成了一项非常简单的任务，因为它具有自动配置Spring Data以访问数据库的能力。只需在你的工程中将spring-boot-starter-data-jpa包含进来，Boot的自动配置引擎就能探测到你的工程需要数据访问功能，并且会在Spring应用上下文中创建必要的Bean，这样你就可以使用Repository和服务了。为了更具体地阐述这一点，请参见程序清单1.4中的Gradle构建文件，它列出了一个基于Groovy的微服务web应用的构建结构，该应用使用了Spring Data对JPA的支持来实现数据访问。

程序清单1.4

```
buildscript {
  repositories {
    maven { url "http://repo.spring.io/libs-snapshot" }
    mavenCentral()
  }
```

```
  dependencies {
    classpath("org.springframework.boot:spring-boot-gradle-plugin:1.0.0.RC1")
  }
}

apply plugin: 'groovy'
apply plugin: 'spring-boot'

repositories {
  mavenCentral()
  maven { url "http://repo.spring.io/libs-snapshot"  }
}

ext {
  springBootVersion = "1.0.0.RC1"
}

dependencies {
  compile 'org.codehaus.groovy:groovy-all:2.2.1'
  compile "org.springframework.boot:spring-boot-starter-web:$springBootVersion"
  compile "org.springframework.boot:spring-boot-starter-data-jpa:$springBootVersion"
  compile "org.springframework.boot:spring-boot-starter-actuator:$springBootVersion"
}
```

在这个配置中，Boot的actuator模块提供了对hsqldb的依赖，这会搭建所有必要的依赖——包括模式的创建——因此Spring Data可以使用这个内存数据库作为数据源。这种简便的方式能够让开发人员免于在开发期创建和管理复杂的XML配置，进而能够快速地开发数据库驱动的微服务。如果在classpath中有H2或Derby数据库的话，这种自动化配置也会生效。Boot所提供的另一个便利之处就是能够快速简便地使用相关数据启动应用的数据库模式。这在开发期是非常有用的，此时数据库可能是在内存中或者是不稳定的，开发人员需要保证的是在应用启动的时候能够访问到这些特定的数据。为了阐述这一点，考虑一下程序清单1.5中的示例JPA实体，它代表了微服务所提供的“User”数据结构。

程序清单1.5

```
@Entity
class User {
  @Id
  @GeneratedValue
  Long id

  String username
  String firstName
  String lastName
  Date createdDate
  Date lastAccessed
```

```
  Boolean isActive = Boolean.TRUE
}
```

为了启用代表User对象的通用数据，我们只需创建一个名为schema.sql或data.sql的文件，并将其包含在classpath之中。这个文件会在模式创建完成之后执行，所以基于程序清单1.5所给出的实体，我们可以使用SQL语句启用一个用户账号，如程序清单1.6所示。

程序清单1.6

```
insert into user(username, first_name, last_name, created_date) values ('danveloper', 'Dan', 'Woods', now())
```

在启动的时候，我们所提供的SQL代码会执行，这样就能确保有一个测试账号可以使用。微服务此时已经具有了数据访问的起始点，程序清单1.7展现了如何按照Spring Data的开发模式创建Repository接口，该接口会作为User实体的数据访问对象（Data Access Object）。

程序清单1.7

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

CrudRepository提供了一些通用的接口方法来创建、查询、更新以及删除对象和对象集合。应用所需的其他特定功能可以按照Spring Data的Repository开发约定进行定义。一旦UserRepository接口创建成功，Boot的spring-data-jpa层会在工程中探测到它，并将其添加到Spring应用上下文之中，这样对于controller和sevice对象来说，它就成为可以进行自动注入的可选对象。这种自动化的配置只有在Boot应用要求按照这种方式初始化的时候才生效，这是通过存在@EnableAutoConfiguration注解来标识的。借助程序清单1.8中所实现的controller，微服务现在就可以定义RESTful端点了，服务的使用者可以获取到User的列表或单个User。

程序清单1.8

```
@RestController
@EnableAutoConfiguration
@RequestMapping("/user")
class UserController {

  @Autowired
  UserRepository repository

  @RequestMapping(method=[RequestMethod.GET])
  def get(Long id) {
    id ? repository.findOne(id) : repository.findAll()
  }

  public static void main(String[] args) {
```

```
    SpringApplication.run UserController, args
  }
}
```

在启动的时候，应用将会输出日志，表明Hibernate按照User实体的定义创建数据库结构，在应用初始化的最后，Boot还会从schema.sql文件中导入数据。

在开发微服务应用时，需要特别注意的一点是使用了@RequestMapping注解。这不是Boot特定的注解。不过，因为Boot安装了自己的端点以监控应用的性能、健康情况以及配置，所以需要确保应用的代码不要与这些内置的提供详情的路径解析相冲突。鉴于此，如果有从请求路径中解析属性的需求（在我们的场景中，也就是user的id属性），那么我们需要仔细考虑这个动态的属性解析会对微服务的其他行为产生什么影响。在本例中，只是简单地将controller映射到/user端点而不是根上下文，就能允许Boot的端点也可以进行访问。

微服务所提供的数据并不一定全部适合关系型结构，针对这一点Spring Boot也暴露了一些模块，从而让开发人员可以使用Spring Data的MongoDB和Redis项目，不过依然采取特定的方式来进行配置。Spring Data用来定义数据访问对象（Data Access Object）的高层框架，这样快速切换JPA与非JPA数据源会变得非常容易。参见程序清单1.9，它展现了一个重新定义的UserRepository接口，这个接口设计为使用MongoDB取代JPA。

程序清单1.9

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

MongoRepository接口也扩展了CrudRepository，因此微服务的Controller代码，也就是程序清单1.8所示并不需要修改。为了实现与MongoDB的集成，工程唯一要做的就是在应用的classpath中包含spring-boot-starter-data-mongodb。程序清单1.4所示的Gradle构建文件需要稍微调整一下依赖的部分，如程序清单1.10所示。

程序清单1.10

```
dependencies {
  compile 'org.codehaus.groovy:groovy-all:2.2.1'
  compile "org.springframework.boot:spring-boot-starter-web:$springBootVersion"
  compile "org.springframework.boot:spring-boot-starter-data-mongodb:$springBootVersion"
  compile "org.springframework.boot:spring-boot-starter-actuator:$springBootVersion"
}
```

MongoDB依赖都置于classpath之中以后，Boot将会自动配置Spring Data连接到localhost上的数据库，并且默认的数据库名为test。在这个库中，将会自动创建User集合（按照MongoDB的标准），微服务现在就能使用MongoDB作为后端了。对非JPA的数据存储来说，初始化数据比其他的方式更为简单，这主要是因为它不能针对MongoDB的文档存储和Redis的键值存储运行SQL文件。鉴于Spring Data会使用这些存储的持久化实例，这就意味着开发期创建的数据需要在重启后保留。为了持久化数据，我们需要修改微服务的controller，这样服务的使用者就能创建User实例了。我们也可以将微服务的UserController进行修改，使其符合通用的RESTful API结构，让controller以不同的方式处理不同的HTTP方法。程序清单1.11展现了为

controller添加创建新User实例的功能。

程序清单1.11

```
@RestController
@RequestMapping("/user")
@EnableAutoConfiguration
class UserController {

  @Autowired
  UserRepository repository

  @RequestMapping(method=[RequestMethod.GET])
  def get(Long id) {
    id ? repository.findOne(id) : repository.findAll()
  }

  @RequestMapping(method=[RequestMethod.POST])
  def create(@RequestBody User user) {
    repository.save user
    user
  }

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

当微服务的使用者往应用的端点上发送一个HTTP POST请求时，Spring将会把请求体转换为User实例。代码接下来会使用UserRepository将这个对象存储到MongoDB集合之中。使用curl创建User实例的样例如程序清单1.12所示。

程序清单1.12

```
curl -v -H "Content-Type: application/json" -d "{ \"username\": \"danveloper\", \"firstName\": \"Dan\", \"lastName\": \"Wooc
```

按照Boot针对Mongo数据源的特定配置，新的User实例默认会持久化到本地Mongo实例的test数据库的user集合之中。如果我们打开web浏览器并对微服务发起一个HTTP GET请求，我们就能看到所创建的user存在于返回的列表之中。

配置

我们可以很快地重写Spring Boot的默认配置。默认情况下，应用的配置可以使用Java属性文件来进行定义，这个文件名为application.properties

并且位于应用的classpath根目录下。不过，一种更好的方式是使用 YAML配置，它提供了结构化以及嵌套的配置。在应用的运行时类路径之中包含snakeyaml之后，你的工程就可以在application.yml文件中直接定义配置了。为了详述这一点，考虑程序清单1.13的示例YAML配置，这里列出了应用的嵌入式HTTP服务器（默认是Tomcat，也可选择Jetty）的各种设置项。

程序清单1.13

```
# Server settings (ServerProperties)
server:
  port: 8080
  address: 127.0.0.1
  sessionTimeout: 30
  contextPath: /

  # Tomcat specifics
  tomcat:
    accessLogEnabled: false
    protocolHeader: x-forwarded-proto
    remoteIpHeader: x-forwarded-for
    basedir:
    backgroundProcessorDelay: 30 # secs
```

允许重写Boot的自动化配置，这一点能够使你的应用从原型转化为真正的产品，Boot使用相同的application.yml文件进行配置，这样就会非常容易。自动化配置的指令被设计的尽可能简短，所以当使用actuator构建微服务时，会安装一个配置属性的端点，也就是/configprops，当确定哪些指令需要重写时可以进行参考。如果我们的微服务要使用持久化数据源，如MySQL，那么只需将MySQL的Java驱动添加到运行时classpath中，然后在application.yml中添加必要的配置指令即可，如程序清单1.14所示。

程序清单1.14

```
spring:
  datasource:
    driverClassName: com.mysql.jdbc.Driver
    url: jdbc:mysql://localhost:3306/proddb
    username: root
    password
```

在一些场景下你可能需要更为灵活的配置，Boot允许你通过Java的系统属性（System properties）重写很多它的默认配置。例如，如果你的应用需要在部署到产品化环境中使用不同的数据库用户，那么username配置指令可以通过标准的Java系统属性传入到应用之中，而这需要切换到命令行中执行-Dspring.datasource.username=user。关于这一点更为现实的场景是云部署环境，如Cloud Foundry或Heroku，这些平台需要应用启动特定的HTTP端口，这一点通过操作系统的环境变量可以实现。Boot能够从系统属性继承得到配置，这样你的应用就可以在命令行中使用-Dserver.port=$PORT来得到HTTP端口。在开发微服务时，这是一种相当有用的特性，因为它可以让微服务应用运行在各种环境配置之中。

外部化配置

微服务必须要支持的很重要的一点就是*外部化*配置。这种配置可以包含任何的内容，从占位符信息到数据库配置等等，在初始规划和构建应用原型时，这是必须要考虑的架构内容。在Spring IO平台中，已经存在各种导入配置的策略，但是应用能够以多种方式使用配置所造成的后果往往是产生冗长的编码性耦合。

Boot一个很棒的特性在于它能管理外部化的配置并将其转换为对象结构，这个对象可以在整个应用上下文中使用。创建一个简单老式的Java/Groovy对象（Plain Old Java/Groovy Object），并使用`@ConfigurationProperties`注解，那么这个对象就能使用Boot配置结构中预先定义的`name`名下的配置项。更具体一点来讲，考虑一下程序清单1.15中的POGO，它能够得到`application.key`下的配置指令。

程序清单1.15

```
@ConfigurationProperties(name = "application")
class ApplicationProperties {
  String name
  String version
}
```

当`ApplicationProperties`对象在Spring上下文中创建完成之后，Boot将会识别出它是一个配置对象，并且会按照运行时classpath之中`application.properties`或`application.yml`文件中的配置指令填充它的属性。因此，如果我们在微服务的`application.yml`文件中添加`application`内容区的话，如程序清单1.16所示，那么我们就可以在应用的其他部分以编程的方式访问这些配置指令。

程序清单1.16

```
application:
  name: sb-ms-custdepl
  version: 0.1-CUSTOMER
```

这些配置指令可以有各种用途，要访问这些指令的唯一要求就是代表它们的POJO/POGO必须是Spring应用上下文的成员。Boot能够将一个controller作为Spring Java配置对象，这样就能很容易地管理配置bean与应用上下文的集成，如程序清单1.17所示。

程序清单1.17

```
@RestController
@Configuration
@RequestMapping("/appinfo")
@EnableAutoConfiguration
class AppInfoController {

  @Autowired
```

```
  ApplicationProperties applicationProperties

  @RequestMapping(method=[RequestMethod.GET])
  def get() {
    [
      name: applicationProperties.name,
      version: applicationProperties.version
    ]
  }

  @Bean
  ApplicationProperties applicationProperties() {
    new ApplicationProperties()
  }

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

程序清单1.17中的样例代码可能有些牵强，不过，即便是在更为复杂的场景下，如何使用Boot来访问应用特定配置的原则是相同的。配置类也支持嵌套式的对象图，这样来自于配置中的深层数据就能更便利地进行访问，也有了更好的语义。例如，如果我们想要得到的配置指令是application.根下的那些metrics key，那么可以在ApplicationProperties POGO中添加一个嵌套对象来表示这些值，如程序清单1.18所示。

程序清单1.18

```
@ConfigurationProperties(name = "application")
class ApplicationProperties {
  String name
  String version

  final Metrics metrics = new Metrics()

  static class Metrics {
    String dbExecutionTimeKey
  }
}
```

现在，我们的application.yml文件可以如程序清单1.19所示，它在application.代码块中包含了metrics配置。

程序清单1.19

```
application:
```

```
  name: sb-ms-custdepl
  version: 0.1-CUSTOMER
  metrics:
    dbExecutionTimeKey: user.get.db.time
```

当我们需要访问application.metrics.dbExecutionTimeKey的值时，能够以编程的方式通过ApplicationProperties对象来进行访问。

为了在整个应用之中使用application.properties或application.yml文件中的这些配置指令，我们并不是必须要将其转换为对象图。Boot也为Spring应用上下文提供了PropertySourcesPlaceholderConfiguration，这样的话，来自于application.properties或application.yml文件的指令或者来自于Java系统的重写属性都可以作为Spring属性占位符来使用。Spring的这种机制能够让你以一种特定的语法来为属性定义占位符值，如果Spring发现了占位符配置的话，就会用这个配置来进行填充。作为示例，我们可以在controller中使用@Value注解来直接访问application.metrics.dbExecutionTimeKey，如程序清单1.20所示。

程序清单1.20

```
@RestController
@RequestMapping("/user")
@EnableAutoConfiguration
class UserController {

  @Autowired
  UserRepository repository

  @Autowired
  GaugeService gaugeService

  @Value('${application.metrics.dbExecutionTimeKey}')
  String dbExecutionKey

  @RequestMapping(method=[RequestMethod.GET])
  def get(Long id) {
    def start = new Date().time
    def result = id ? repository.findOne(id) : repository.findAll()
    gaugeService.submit dbExecutionKey, new Date().time - start
    result
  }

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

关于应用指标的报告，后面会有更为详细的介绍，但现在重要的一点在于，理解@Value注解如何与Spring属性占位符一起使用，使Boot能够自动注入值，从而满足这个微服务的特定配置需求。

安全

在微服务的开发中，对于完备安全场景的需求会持续增长。为了满足这种需求，Boot引入了强大完整的Spring Security，并且提供了自动配置的功能，以快速简便地启用安全层。只需在应用的classpath中包含spring-boot-starter-security模块就能使Boot引入一些安全特性，如跨站脚本防护（cross-site scripting protection）并且会添加头信息以防止点击劫持（click-jacking）。除此之外，添加一条简单的配置指令就能启用基本认证来保护你的应用，如程序清单1.21所示。

程序清单1.21

```
security:
  basic:
    enabled: true
```

Boot会为你提供一个默认的用户账号user和默认角色USER，并且会在应用启动的时候在控制台上输出随机生成的密码。就像Boot的其他功能那样，对于内置的user账号，我们可以很容易地指定不同的用户名和密码（分别为“secured”和“foo”），这需要通过明确定义的配置指令来实现，如程序清单1.22所示。

程序清单1.22

```
security:
  basic:
    enabled: true
  user:
    name: secured
    password: foo
```

对于简单的内部应用或开发原型来说，Boot内置的基础设施能够快速地在微服务中启用基本认证，这是非常有用的。随着需求的演化，你的应用毫无疑问会需要更细粒度的安全特性，如保护端点只能由特定的角色访问。从这个角度来看，我们可能希望具有USER角色的调用者只能读取数据（即**GET**请求），而对具有ADMIN角色的调用者可以读取和写入数据（即**POST**请求）。为了做到这一点，我们需要在工程的application.yml文件中禁用Boot的基本认证自动配置功能，并且定义我们自己的user和admin账号以及对应的角色。当你的需求超过Boot所提供的默认功能时，它通常很快就能实现，这可以作为佐证这一点的又一个例子。为了更具体地阐述这一点，考虑一下程序清单1.23中的代码。这个样例可以阐述如何发挥Spring Security所有潜在的功能以及更为复杂的认证策略，如基于JDBC后端、OpenID或单点登录（Single-Sign On）。

程序清单1.23

```
@RestController
@RequestMapping("/user")
@Configuration
@EnableGlobalMethodSecurity(securedEnabled = true)
@EnableAutoConfiguration
class UserController extends WebSecurityConfigurerAdapter {

  @Autowired
  UserRepository repository

  @RequestMapping(method = [GET])
  @Secured(['ROLE_USER'])
  def get(Long id) {
    id ? repository.findOne(id) : repository.findAll()
  }

  @RequestMapping(method = [POST])
  @Secured(['ROLE_ADMIN'])
  def create(@RequestBody User user) {
    repository.save user
    user
  }

  @Override
  void configure(AuthenticationManagerBuilder auth) {
    auth
    .inMemoryAuthentication()
    .withUser "user" password "password" roles "USER" and() withUser "admin" password "password" roles "USER", "ADMIN"
  }

  @Override
  void configure(HttpSecurity http) throws Exception {
    BasicAuthenticationEntryPoint entryPoint = new BasicAuthenticationEntryPoint()
    entryPoint.realmName = "Spring Boot"
    http.exceptionHandling().authenticationEntryPoint(entryPoint)
    http.requestMatchers().antMatchers("/**").anyRequest()
    .and().httpBasic().and().anonymous().disable().csrf().disable()
  }

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

在程序清单1.23的样例之中，应用现在被明确地配置为要基于user和admin用户账号进行访问，它们的密码都是password，具有的角色分别是USER和ADMIN。微服务的**GET**和**POST**端点分别通过USER和ADMIN角色进行保护，这就意味着普通用户可以访问只读的数据，而执行读取-写入操作的话，需要admin用户凭证。

对于微服务来说，基本认证是很好的一个选择，因为它遵循了很实用且广泛使用的认证协议。换句话说，很多的API调用者，包括移动应用，能够很容易地使用这一点来访问你的微服务。当你的认证需求超过了基本认证的功能时（如OpenID或OAuth），微服务可以使用Spring Security的全部功能来满足你的需求。

消息集成

在任何的应用中，消息（messaging）都是一种很强大的工具，在一点上，微服务当然也不能例外。使用消息驱动架构开发的应用能够更好地支持可重用性和扩展性。Spring Boot能够让开发人员在编写微服务时将消息作为架构的核心组成部分，它使用到了Spring IO平台的企业集成模式（Enterprise Integration Patterns）实现，即Spring Integration。Spring Integration提供了开发消息驱动架构的基本结构，以及与分布式企业平台集成的模块。这种能力使得微服务可以使用来自抽象消息源的业务对象，这些消息源可以在应用内部，也可能是组织机构内部的其他服务所提供的。

尽管Boot并没有提供明确的Spring上下文自动化配置，但是它为Spring Integration提供了一个starter模块，它会负责引入Spring Integration项目的一系列依赖。这些依赖包括Spring Integration的核心库（Core library）、HTTP模块（用来进行面向HTTP的企业集成）、IP模块（用来进行基于Socket的集成操作）、File模块（用于进行文件系统集成）以及Stream模块（用于支持使用Stream的操作，如stdin和stdout）。这个starter模块为开发人员提供了健壮的消息功能的工具集，可以使已有的基础设施适应微服务API。

除了starter模块，Boot也为通过CLI构建的应用提供了编译器自动配置的功能。对于需要快速构建微服务原型并验证可行性的开发者来说，这种方式提供了一些捷径。使用企业级平台的应用可以快速地进行开发，在转移到正式的工程和构建系统之前，就能确认其价值。使用Spring Boot和Spring Integration使一个消息驱动的微服务运行起来非常简单，如程序清单1.24的样例代码所示。

程序清单1.24

```
@RestController
@EnableIntegrationPatterns
class App {

  @Bean
  def userLookupChannel() {
    new DirectChannel()
  }

  @Bean
  def userTemplate() {
    new MessagingTemplate(userLookupChannel())
```

```
  }

  @RequestMapping(method=[RequestMethod.GET])
  def get(@RequestParam(required=false) Long id) {
    userTemplate().convertSendAndReceive( id ? id : "")
  }
}

class User {
  Long id
}

@MessageEndpoint
class UserLookupObject {

  @ServiceActivator(inputChannel="userLookupChannel")
  def get(Long id) {
    id ? new User(id:id) : new User()
  }
}
```

使用消息驱动的方式来进行微服务的开发能提供很大的代码可重用性，并且能够与底层的服务提供者实现相解耦。在更为正式的场景之中，程序清单1.24的代码可能会负责组合数据，这些数据可能来自于数据库调用和企业组织中某个外部的服务集成。Spring Integration具有内置的设施用来进行负载路由（payload routing）和处理器链（handler chaining），这对于组合不同的数据来说，是一个很有吸引力的方案，我们的微服务可以作为一个数据的提供者（provider）。

提供度量指标

微服务最重要的一个特性可能就是为报表终端（reporting agent）提供度量指标。不像那些功能完备的Web应用，微服务是轻量级的，设计时可能就不会规划提供报表界面或完备的接口来分析服务的活动。这种类型的操作最好是留给专门进行数据聚合和分析的应用，这些数据能够用来进行稳定性、性能以及商务智能的监控。基于这样的前提，微服务应该为这些工具提供端点，从而更加容易地获取有关该服务活动的数据。而报表工具负责将数据聚合到一个视图或报告中，对于关心数据的人这才是有意义的。

微服务的一些指标如稳定性和性能，对所有的应用都是通用的，但是与业务操作相关的指标必须由应用本身来具体进行管理。针对这一点，Spring Boot的`actuator`模块为开发人员提供了一种机制，允许开发人员通过`/metrics`端点以编码的方式暴露微服务状态的细节。Boot将指标拆分为“counter”和“gauge”两种类别：counter是所有以Number类型来展现的指标，而gauge是衡量双精度计算的指标。为了让微服务的开发人员更加容易地使用指标，Boot暴露了`CounterService`和`GaugeService`，它们可以自动织入到应用上下文之中。请参见程序清单1.25的样例，它阐述了如何通过`CounterService`对外暴露点击数。

程序清单1.25

```
@RestController
@RequestMapping("/user")
@EnableAutoConfiguration
class UserController {

  @Autowired
  UserRepository repository

  @Autowired
  CounterService counterService

  @RequestMapping(method = [GET])
  def get() {
    get(null)
  }

  @RequestMapping(value="/{id}", method = [GET])
  def get(@PathVariable Long id) {
    counterService.increment id ? "queries.by.id.$id" : "queries.without.id"
    id ? repository.findOne(id) : repository.findAll()
  }
}
```

在点击/user端点时，有可能提供ID也有可能不提供ID，/metrics端点都会在counter.父节点下记录新的key。例如，如果我们只是查询/user端点而不带有ID的话，那么就会注册counter.queries.without.id指标。类似的，如果我们带有ID的话，那么就会看到有一个counter.queries.by.id.<id>的key，它能用来标记对于给定的ID已经进行了多少次查询。这些指标可能会有助于掌握最经常访问的User对象并指导要采取的行为，如缓存或数据库索引。类似于递增指标的数值，CounterService也允许将指标的值将为零。这对于跟踪打开的连接数或其他频率分布（histographic）的测量都是很有用处的。

gauge是稍微有所不同的一种类型指标，它会进行探索性的计算或基于请求来确定值。如GaugeService的JavaDocs所述，“gauge”可以测量任意的值，从方法执行的次数到会议室的温度。当需要为报表工具暴露细节时，这种类型的测量尤其适合于使用GaugeService。gauge的指标会在/metrics端点之下进行访问，并且带有gauge.前缀。它们的注册方式与counter有些差别，如程序清单1.26所示。

程序清单1.26

```
@RestController
@RequestMapping("/user")
@EnableAutoConfiguration
class UserController {
```

```
  @Autowired
  UserRepository repository

  @Autowired
  CounterService counterService

  @RequestMapping(method = [GET])
  def get() {
    get(null)
  }

  @RequestMapping(value="/{id}", method = [GET])
  def get(@PathVariable Long id) {
    def start = new Date().time
    def result = id ? repository.findOne(id) : repository.findAll()
    def time = new Date().time - start
    gaugeService.submit("user.get.db.time", time.doubleValue())
    result
  }
}
```

默认情况下，指标会存储在一个易失的内存数据库之中，但Boot同时也为应用上下文提供了`MetricsRepository`实现，它能支持更为持久化的行为。Boot自带了一个`RedisMetricsRepository`，它能够自动织入进来，从而将指标存储到Redis值存储之中，另外，可以编写自定义的实现将指标存储到任意的数据存储形式之中。

Boot还提供了对Coda Hale Metrics库的支持，它会将以特定名称开头的指标强制转换为对应的Metrics类型。例如，如果有一个指标是以`histogram.`开头，那么这个值将会作为`Histogram`对象类型。这种自动化的强制转换对于`meter.`和`timer.`key也是有效的，而普通的指标将会作为`Gauge`类型。

一旦微服务的指标在Boot中进行了注册，那么报表工具就可以通过`/metrics`端点来检索它们。已命名的指标可以通过`/metrics`端点获取，只需将指标的key名作为查询字符串的一部分即可。例如，如果*只是*访问gauge指标下的“user.get.db.time”，报表工具可以针对`/metrics/gauge.user.get.db.time`进行查询。

打包Boot应用

正如前面所讨论的，Boot提供了Maven和Gradle插件，它为构建系统的打包阶段提供了一种钩子（hook），以产生所谓的“胖jar”，在这种jar中包含了工程的所有依赖。当这个胖jar包执行时，应用将会运行在与工程开发期相同的嵌入式容器之中。这种简便的方式能够让开发人员省去很多麻烦，因为他们的部署包在开发期和运行时环境之中具有相同的依赖结构。这也能够缓解运维团队的焦虑，他们不用担心部署的场景，因为在部署时一个错误配置的运行时容器可能会带有某个特定的依赖，而在项目的开发期所依赖的可能是另外一个。

为了在Maven下执行打包，只需执行`mvn package`命令。Spring Boot的插件会备份工程所创建的原始jar并且在文件名上添加“.original”。在这里，能够得到可运行的jar，文件符合Maven artifact的命名约定，它可以按照工程最合适的方式进行部署。使用Gradle构建Boot工程同样很简单，只需执行标准的`gradle build`命令即可。类似于Maven，Boot插件在原有的打包任务之后使用Gradle安装了一个生命周期事件，并且会在`build/libs`目录下创建胖jar包。对所生成的胖jar包进行检查的一种方式就是所有依赖的jar都会位于归档文件的`lib/`目录下。

打包完成之后，胖jar包就能够像其他可运行的jar文件那样在命令行中执行了，也就是使用`$JAVA_HOME/bin/java -jar path/to/myproject.jar`命令。启动后，Boot应用的日志将会显示在控制台上。

对于需要部署到传统servlet容器之中的应用，Boot提供了一种方式以编码的方式初始化Web配置。为了使用这一点，Boot提供了可选的`WebApplicationInitializer`，它会使用servlet容器来注册应用，这会通过Servlet 3.0 API以编码的方式注册servlet并且会用到`ServletContext`。通过提供`SpringBootServletInitializer`的子类，Boot应用能够使用嵌入的Spring上下文来注册配置，这个Spring上下文是在容器初始化的时候创建的。为了阐述这个功能，考虑程序清单1.27中的示例代码。

程序清单1.27

```
@RestController
@EnableAutoConfiguration
class Application extends SpringBootServletInitializer {

  @RequestMapping(method = RequestMethod.GET)
  String get() {
    "home"
  }

  static void main(String[] args) {
    SpringApplication.run this, args
  }

  @Override
  SpringApplicationBuilder configure(SpringApplicationBuilder application) {
    application.sources Application
  }
}
```

`Application`类中被重写的`configure`方法就是使用嵌入式的Spring上下文注册应用的地方。在更为正式的场景之中，这个方法可能会用来注册Spring Java配置类，它会定义应用中所有controller和服务的bean。

当将应用打包部署到servlet容器之中时，工程要构建为一个war文件。在Maven工程中，为了适应这一点，需要移除Boot插件，并且packaging需要明确定义为“war”类型，如程序清单1.28所示。

程序清单1.28

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

    <groupId>com.example</groupId>
    <artifactId>myproject</artifactId>
    <version>1.0.0-SNAPSHOT</version>
    <packaging>war</packaging>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.0.0.RC1</version>
    </parent>

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

    <repositories>
        <repository>
            <id>spring-snapshots</id>
            <url>http://repo.spring.io/libs-snapshot</url>
        </repository>
    </repositories>
</project>
```

对这个工程执行`mvn install`命令会在`target`目录下生成`myproject-1.0.0-SNAPSHOT.war`文件。使用Gradle构建的工程可以使用Gradle War Plugin，它为构建war文件暴露了一个`war`任务。类似于Maven的配置，Boot Gradle工程也需要移除所包含的Boot插件。产生war文件的示例Gradle构建脚本如程序清单1.29所示。

程序清单1.29

```
apply plugin: 'java'
apply plugin: 'war'

repositories {
    mavenCentral()
    maven { url "http://repo.spring.io/snapshot" }
    maven { url "http://repo.spring.io/milestone" }
}

ext {
  springBootVersion = '1.0.0.BUILD-SNAPSHOT'
}

dependencies {
    compile "org.springframework.boot:spring-boot-starter-web:${springBootVersion}"
    compile "org.springframework.boot:spring-boot-starter-actuator:${springBootVersion}"
}
```

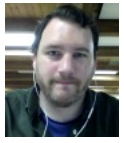
对于Boot工程，使用这个构建脚本运行Gradle的war任务将会在build/libs目录下产生war文件。

不管是Maven还是Gradle的配置，一旦war文件产生，它就可以部署到任意兼容Servlet 3.0的应用服务器之中。部分兼容的容器包括Tomcat 7+、Jetty 8、Glassfish 3.x、JBoss AS 6.x/7.x以及Websphere 8.0。

延伸阅读

Spring Boot团队已经编写了完整的指导和样例来阐述框架的功能。Blog文章、参考资料以及API文档都可以在Spring.IO网站上找到。项目的GitHub页面上可以找到示例的工程，更为具体的细节可以阅读Spring Boot的参考手册。SpringSourceDev YouTube频道有一个关于Spring Boot的webinar，它概述了这个项目的目标和功能。在去年在伦敦举行的Groovy & Grails Exchange上，David Dawson做了一个使用Spring Boot开发微服务的演讲。

关于作者



Daniel Woods是Netflix的高级软件工程师，负责开发持续交付和云部署工具。他擅长JVM栈相关的技术，活跃在Groovy、Grails和Spring社区。可以通过电子邮件地址danielpwoods@gmail.com或Twitter @danveloper联系到Daniel。

查看原文链接：Exploring Micro-frameworks: Spring Boot

- 领域
- **语言 & 开发**
- 专栏
- Spring框架
- Java

相关内容

Spring Boot 1.3发布，包含DevTools和ASCII Art特性

通过Spring Session实现新一代的Session管理

开发者眼中的Spring与Java EE

JavaOne 2015概览

Java 9发布在即，Oracle OpenJDK着手优化Unsafe类

告诉我们您的想法

请输入主题

信息

允许的HTML标签: a,b,br,blockquote,i,li,pre,u,ul,p

当有人回复此评论时请E-mail通知我

发送信息

社区评论 Watch Thread

感谢译者和编辑，挑个刺 2015年1月4日 03:51 by "尹 文友"

1，“程序清单1.27” 排版有误，导致代码部分丢失。
2，翻译质量如果再提高些，读起来会轻松些。当然了，也更此文信息含量较大有关。

- 回复
- 回到顶部

Re: 感谢译者和编辑，挑个刺 2015年5月14日 03:03 by "Wang Peter"

作为介绍Spring Boot功能的文章挺好，但是作为入门的话内容介绍的粒度有点粗。

- 回复
- 回到顶部

用它进行新应用构建，很慢吗？ 2015年7月22日 08:31 by "李 鹏"

Spring用了有三年了，用它来进行应用的构建，很慢吗？

- 回复
- 回到顶部

用它进行新应用构建，很慢吗？ 2015年7月22日 08:31 by "李 鹏"

Spring用了有三年了，用它来进行应用的构建，很慢吗？

- 回复
- 回到顶部

信息量太大 2015年8月21日 12:52 by "Tiange Ye"

翻译读起来也有些吃力，作为Spring Boot入门的文章，我觉得这篇文章适合一些 tianmaying.com/tutorial/spring-boot-overview

- 回复
- 回到顶部

关闭

by

发布于

- 查看
- 回复
- 回到顶部

关闭

主题 您的回复

引用原消息

允许的HTML标签: a,b,br,blockquote,i,li,pre,u,ul,p

当有人回复此评论时请E-mail通知我

发送信息 取消

关闭

主题 您的回复

允许的HTML标签: a,b,br,blockquote,i,li,pre,u,ul,p

当有人回复此评论时请E-mail通知我

取消

关闭

OK

相关内容

- 通过Spring Session实现新一代的Session管理 2015年12月7日 
- Java 9发布在即，Oracle OpenJDK着手优化Unsafe类 2016年5月7日
- Ehcache 3.0发布，修补了API并支持非堆存储 2016年5月5日
- 关于设计低延迟应用的一些经验 2016年4月18日
- JetBrains发布了IntelliJ IDEA 2016.1 2016年4月18日
- 诊断Java代码中常见的数据库性能热点问题 2016年4月28日 
- Android Studio 2.0新特性：即时运行和云测试实验室 2016年4月13日
- 架构师（2016年3月） 2016年3月14日 

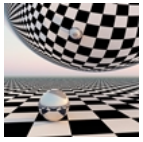
- IAP:HTTP的替代者，更快、更丰富 2016年4月19日 

- 从案例学RxAndroid开发（下） 2016年4月8日 
- 模块化编程和Jigsaw项目最新早期访问版本使用教程 2016年3月10日 


技术 干货
活动大本营
Geekbang
InfoQ
免费 大会 课程 沙龙

相关内容

- 从Java代码到机器代码：如何编写高度优化的Java程序 2016年3月9日 
- OCP Oracle Java SE 8专业程序员认证学习指南——认证与深入学习的阶梯 2016年3月8日 
- 通过Baratine将Lucene库暴露为微服务 2016年2月25日
- 通过使用Byte Buddy，便捷地创建Java Agent 2016年2月22日

- Java Web应用中调优线程池的重要性 2016年2月16日 
- Java 9终于要包含Jigsaw项目了 2016年2月3日 
- sun.misc.Unsafe的后启示录 2016年1月26日 
- Scala模式匹配的亮点——Martin Odersky访谈（四） 2016年1月21日 
- 使用Java 8的CompletableFuture实现函数式的回调 2016年1月19日 
- Java反序列化漏洞被忽略的大规模杀伤利用 2016年1月18日 
- 深入理解Android（三）：Xposed详解 2016年1月4日 


技术干货
活动大本营
Geekbang
InfoQ
免费 大会 课程 沙龙

相关内容

- 全端Web开发：快速开发实践 2015年12月24日 

- Java多线程编程模式实战指南之Promise模式 2015年12月10日 

- 使用Ratpack与Spring Boot构建高性能JVM微服务 2015年12月10日 
- 深入探索Java 8 Lambda表达式 2015年11月27日 

- Scala类型系统的目的——Martin Odersky访谈（三） 2015年11月17日 
- 以什么姿势进入DataMining会少走弯路？ 2016年5月12日
- Mercurial 3.8版本发布：为Mercurial 指令服务器提供最新的快速客户端 2016年5月12日
- Instagram的持续部署实践 2016年5月12日
- 深入JVM彻底剖析ygc越来越慢的原因（下） 2016年5月12日 
- 对话Linus：Linux 25岁啦 2016年5月12日 
- 基于OpenStack的云测试平台 2016年5月12日 

InfoQ每周精要

通过个性化定制的新闻邮件、 RSS Feeds和InfoQ业界邮件通知，保持您对感兴趣的社区内容的时刻关注。

您的邮箱 订阅

语言 & 开发

苹果强推IPv6，你的App符合新规吗？

以什么姿势进入DataMining会少走弯路？

Mercurial 3.8版本发布：为Mercurial 指令服务器提供最新的快速客户端

架构 & 设计

以什么姿势进入DataMining会少走弯路？

分布式MySQL集群方案的探索与思考

深入JVM彻底剖析ygc越来越慢的原因（下）

文化 & 方法

Agile Games 2016主旨发言—更快、更廉价、更好的培训设计

DeepMind回应有关大规模访问病人数据的报道，向InfoQ透露其与NHS的合作细节

IT业也是制造业，日本IT产业奇葩在哪儿

数据科学

14个价值10亿美元以上的大数据公司

DeepMind回应有关大规模访问病人数据的报道，向InfoQ透露其与NHS的合作细节

Google云端机器学习和Tensor Flow的Alpha测试版本发布

DevOps

Instagram的持续部署实践

GitLab揭露严重漏洞，提供补丁

CoreOS公司B轮融资2800万美元，投资阵营由GV牵头

InfoQ每周精要

通过个性化定制的新闻邮件、RSS Feeds和InfoQ业界邮件通知，保持您对感兴趣的社区内容的时刻关注。

您的邮箱 订阅

- 属于您的个性化RSS
- InfoQ官方微博
- InfoQ官方微信
- 社区新闻和热点

特别专题

- 活动大本营
- 月刊：《架构师》
- AWS专区
- 百度技术沙龙专区
- 腾讯云专区
- 七牛专区
- EGO超级极客邦
- StuQ提升你技能
- 信息无障碍参考文档

定制您感兴趣的技术领域

- 语言 & 开发
- 架构 & 设计
- 数据科学
- 文化 & 方法
- DevOps

这会影响您在主页和RSS订阅中看到的内容。点击“偏好设置”可选择更多精彩定制内容。

提供反馈 feedback@cn.infoq.com
错误报告 bugs@cn.infoq.com
商务合作 sales@cn.infoq.com
内容合作 editors@cn.infoq.com
Marketing marketing@infoq.com

BT