# 微服务使用手册

## 1 [首页](#_bookmark0)

1. [微服务简介](#_bookmark1)
2. [Spring Cloud](#_bookmark2)
   1. [服务发现](#_bookmark3)与监控
      1. [Eureka](#_bookmark4)
      2. [Eureka的高可用](#_bookmark5)
      3. monitor
      4. [monitor安装与使用](#_bookmark7)
      5. monitor常用功能
      6. zipkin
   2. [服务提供者](#_bookmark10)
   3. [服务消费者](#_bookmark11)
      1. [Ribbon](#_bookmark12)

[2.3.2. Feign](#_bookmark13)

* 1. [熔断器](#_bookmark14)
     1. [Hystrix](#_bookmark15)
     2. [Hystrix Dashboard](#_bookmark16)
     3. [Turbine](#_bookmark17)
  2. [配置中心](#_bookmark18)
  3. [API Gateway](#_bookmark19)

1. [使用Docker构建微服务](#_bookmark20)
   1. [Docker介绍](#_bookmark21)
   2. [Docker的安装](#_bookmark22)
   3. [Docker的常用命令](#_bookmark23)
   4. [Dockerfile常用指令](#_bookmark24)
   5. [Docker私有仓库的搭建与使用](#_bookmark25)
   6. [使用Dockerfile构建Docker镜像](#_bookmark26)

1.1

1.2

1.3

1.3.1

1.3.1.1

1.3.1.2

1.3.1.3

1.3.1.4

1.3.1.5

1.3.1.6

1.3.2

1.3.3

1.3.3.1

1.3.3.2

1.3.4

1.3.4.1

1.3.4.2

1.3.4.3

1.3.5

1.3.6

1.4

1.4.1

1.4.2

1.4.3

1.4.4

1.4.5

1.4.6

* 1. [使用Maven插件构建Docker镜像](#_bookmark27)
  2. [Docker Compose](#_bookmark28)
     1. [Docker Compose的安装](#_bookmark29)
     2. [Docker Compose入门示例](#_bookmark30)
     3. [docker-compose.yml常用命令](#_bookmark31)
     4. [docker-compose常用命令](#_bookmark32)

1.4.7

1.4.8

1.4.8.1

1.4.8.2

1.4.8.3

1.4.8.4

使用**Spring Cloud**与**Docker**实战微服务

简介

本文主要是对 Spring Cloud 和 Docker 的学习总结与实战 ，目前在 Git@OSC 以及 Github 上同步更新。

已完成的章节包括：

1. 微服务简介
2. Spring Cloud
   1. 服务发现
      1. Eureka
      2. Eureka的高可用
      3. Consul
      4. Consul安装与使用
      5. Consul常用命令
      6. Consul高可用
   2. 服务提供者
   3. 服务消费者
      1. Ribbon

2.3.2. Feign

* 1. 熔断器
     1. Hystrix
     2. Hystrix Dashboard
     3. Turbine
  2. 配置中心
  3. API Gateway
  4. 2.7 Eureka的高可用

1. 使用Docker构建微服务
   1. Docker介绍
   2. Docker的安装
   3. Docker的常用命令
   4. Dockerfile常用指令
   5. Docker私有仓库的搭建
   6. 使用Dockerfile构建Docker镜像
   7. 使用Maven插件构建Docker镜像
   8. Docker Compose
      1. Docker Compose的安装
      2. Docker Compose入门示例
      3. docker-compose.yml常用命令
      4. docker-compose常用命令
2. 微服务简介

什么是微服务架构

近年来，在软件开发领域关于微服务的讨论呈现出火爆的局面，有人倾向于在系统设计与开发中采用微服务方式实现软件系统的松耦合、跨部门开发，被认为是IT软 件架构的未来方向，Martin Fowler也给微服务架构极高的评价；同时，反对之声也很强烈，持反对观点的人表示微服务增加了系统维护、部署的难度，导致一些功能 模块或代码无法复用，同时微服务允许使用不同的语言和框架来开发各个系统模块，这又会增加系统集成与测试的难度，而且随着系统规模的日渐增长，微服务在 一定程度上也会导致系统变得越来越复杂。尽管一些公司已经在生产系统中采用了 微服务架构，并且取得了良好的效果；但更多公司还是处在观望的态度。

什么是微服务架构呢？简单说就是将一个完整的应用（单体应用）按照一定的拆分规则（后文讲述）拆分成多个不同的服务，每个服务都能独立地进行开发、部署、扩展。服务于服务之间通过注入RESTful api或其他方式调用。大家可以搜索到很多相关介绍和文章。本文暂不细表。在此推荐两个比较好的博客：

<http://microservices.io/><http://martinfowler.com/articles/microservices.html>

1. **Spring Cloud**

### **Spring Cloud** 简介

Spring Cloud是在Spring Boot的基础上构建的，用于简化分布式系统构建的工具集，为开发人员提供快速建立分布式系统中的一些常见的模式。

下面我们将以代码与讲解结合的方式，为大家讲解Spring Cloud中的各种组件。

#### 准备工作

技术储备：

|  |  |
| --- | --- |
| 所需技能 | 备注 |
| Java |  |
| Maven | 文章涉及到大量的代码，均使用Maven构建 |
| Spring Boot | Spring Cloud是在Spring Boot基础上构建的 |

环境准备：

|  |  |
| --- | --- |
| 工具 | 版本或描述 |
| JDK | 1.8 |
| IDE | Eclipse(本例中使用的是4.7). |
|  |  |

本课程所使用的软件及版本：

|  |  |  |
| --- | --- | --- |
| 使用到的软件 | 版本号 | 是否最新版本 |
| Spring Boot | 1.5.4.RELEASE | - |
| Spring Cloud | Brixton.SR4 | - |

Host配置：在生产环境下，我们往往会为每个应用配置一个host，使用host而非IP进行访问。为了更加贴近生产环境，以及后文Docker章节的讲解，我们配 置一下Host。在Windows系统下，

是 C:/Windows/System32/drivers/etc/hosts 文件，在Linux系统下， 是 /etc/hosts 文 件 ：

**Host**配置

127.0.0.1 nriet-eureka nriet-config nriet-gateway nriet-monitor nriet-zipkin

nriet-auth-service

#### 主机规划：

|  |  |  |
| --- | --- | --- |
| 项目名称 | 端口 | 描述 |
| nriet-eureka | 8761 | 服务发现与注册 |
| nriet-config | 8888 | 配置中心服务端 |
| nriet-gateway | 8060 | 网关、代理所有微服务的接口转发 |
| nriet-monitor | 8040 | 微服务监视器，查看微服务健康状况等信息 |
| nriet-zipkin | 9411 | 链路跟踪，查看微服务之间的调用情况 |
| nriet-auth-service | 5000 | Oauth2 微服务身份认证服务 |
| nriet-datacenter-api | 8080 | 业务相关的Api接口 |
| nriet-common |  | 工具包工程 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

父项目的建立

在进入主题之前，我们首先创建一个父项目（nriet-cloud）， 这样可以对项目中的Maven依赖进行统一的管理。

<?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.nriet</groupId>

<artifactId>nriet-cloud</artifactId>

<version>1.0</version>

<packaging>pom</packaging>

<name>${project.artifactId}</name>

<parent>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-parent</artifactId>

<version>1.5.4.RELEASE</version>

<relativePath/>

</parent>

<modules>

<module>nriet-eureka</module>

<module>nriet-config</module>

<module>nriet-gateway</module>

<module>nriet-zipkin</module>

<module>nriet-monitor</module>

<module>nriet-auth-service</module>

<module>nriet-commons</module>

<module>nriet-datacenter-api</module>

</modules>

<properties>

<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>

<java.version>1.8</java.version>

<docker.plugin.version>1.0.0</docker.plugin.version>

<docker.image.prefix>nriet-cloud</docker.image.prefix>

<spring.cloud.version>Dalston.SR4</spring.cloud.version>

<spring-boot-admin.version>1.5.4</spring-boot-admin.version>

</properties>

<dependencies>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-actuator</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-test</artifactId>

<scope>test</scope>

</dependency>

</dependencies>

<dependencyManagement>

<dependencies>

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-server</artifactId>

<version>${spring-boot-admin.version}</version>

</dependency>

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-server-ui</artifactId>

<version>${spring-boot-admin.version}</version>

</dependency>

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-server-ui-hystrix</artifactId>

<version>${spring-boot-admin.version}</version>

</dependency>

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-server-ui-turbine</artifactId>

<version>${spring-boot-admin.version}</version>

</dependency>

<dependency>

<groupId>de.codecentric</groupId>

<artifactId>spring-boot-admin-server-ui-login</artifactId>

<version>${spring-boot-admin.version}</version>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-dependencies</artifactId>

<version>${spring.cloud.version}</version>

<type>pom</type>

<scope>import</scope>

</dependency>

</dependencies>

</dependencyManagement>

<build>

<pluginManagement>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

<executions>

<execution>

<goals>

<goal>repackage</goal>

</goals>

</execution>

</executions>

</plugin>

</plugins>

</pluginManagement>

</build>

</projec

* 1. 服务发现Eureka

关于服务发现

在微服务架构中，服务发现（Service Discovery）是关键原则之一。手动配置每个客户端或某种形式的约定是很难做的，并且很脆弱。Spring Cloud提供了多种服务发现的实现方式，例如：Eureka、Consul、Zookeeper。

Spring Cloud支持得最好的是Eureka，其次是Consul，最次是Zookeeper。

* + 1. **Eureka**

准备工作

在生产环境下，我们往往会为每个应用配置一个host，使用host而非IP进行访问。为了更加贴近生产环境，以及后文Docker章节的讲解，我们首先配置一下

Host

127.0.0.1 nriet-eureka

代码示例

创建一个Maven工程（nriet-eureka），并在pom.xml中加入 如下内容：

|  |  |
| --- | --- |
| <?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>  <parent>  <groupId>com.nriet</groupId>  <artifactId>nriet-cloud</artifactId>  <version>1.0</version>  </parent>  <artifactId>nriet-eureka</artifactId>  <packaging>jar</packaging>  <name>${project.artifactId}</name>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka-server</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency>  </dependencies>  <build>  <finalName>${project.artifactId}</finalName>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> | |
|  |  |

编写Spring Boot启动程序：通过@EnableEurekaServer申明一个注册中心：

/\*\*

* 使用Eureka做服务发现注册。
* @author b\_wangpei

\*/

@SpringBootApplication @EnableEurekaServer

public class RegistryApplication{

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

}

}

在默认情况下，Eureka会将自己也作为客户端尝试注册，所以在单机模式下， 我们需要禁止该行为，只需要在application.yml中如下配置：

# 参考文档：<http://projects.spring.io/spring-cloud/docs/1.0.3/spri>

ng-cloud.html#\_standalone\_mode

# 参考文档：<http://my.oschina.net/buwei/blog/618756>

defaultZone: http://${eureka.instance.hostname}:${server.p

ort}/eureka/

# 指定该Eureka实例的主机名

eureka: instance:

hostname: nriet-eureka client:

registerWithEureka: false fetchRegistry: false

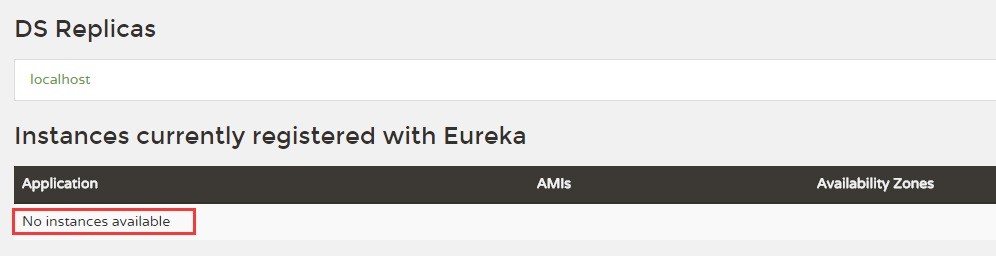
serviceUrl:

# 指定该Eureka实例的端口

server:

port: 8761

启动工程后，访问：http://nriet-eureka:8761/ ，如下图。我们会发现此时还没有服务注册到Eureka上面。



* + 1. **Eureka**的高可用

按照前文对Eureka的讲解，我们即可构建出一个简单的注册中心。但此时的Eureka 是单点的，不适合于大型应用，那么如何实现Eureka的高可用呢？

添加主机名：

127.0.0.1 peer1 peer2

修改application.yml

---

spring:

profiles: peer2

server:

port: 8762

eureka: instance:

hostname: peer2 client:

serviceUrl:

defaultZone: http://peer1:8761/eureka/

# 指定当profile

# 将自己注册到peer2

# 指定profile=p

---

spring:

profiles: peer1

server:

port: 8761 eureka:

instance:

hostname: peer1

client:

serviceUrl:

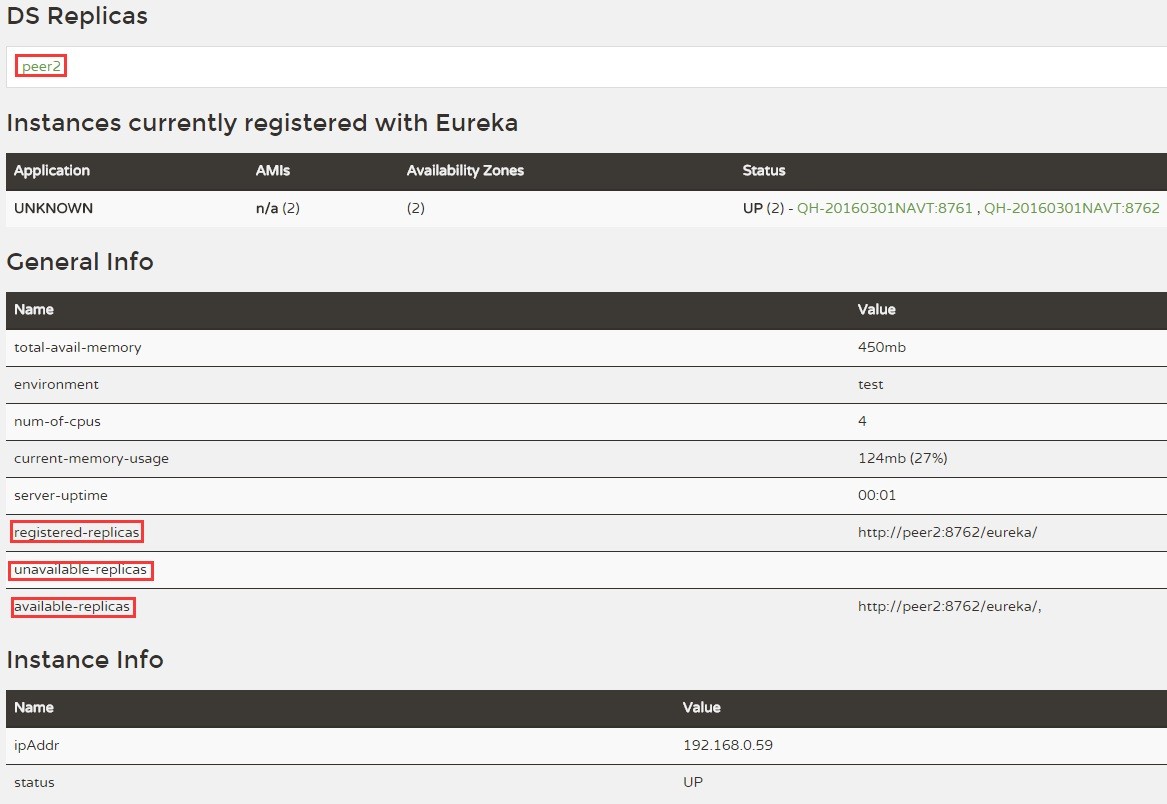
defaultZone: http://peer2:8762/eureka/

分别启动两个Eureka应用：

java -jar nriet-eureka-1.0.jar --spring.profiles.active=peer1

java -jar nriet-eureka-1.0.jar --spring.profiles.active=peer2

访问 http://peer1:8761 ，我们会发现 registered-replicas 中已经有 peer2 节点了，同样地，访问 http://peer2:8762 ，也能发现其中的 registered-replicas 有 peer1 节点，如下图：



我们尝试将 peer2 节点关闭，然后访问 http://peer1:8761 ，会发现此时peer2会被添加到 unavaliable-replicas 一栏中。

注意：

该示例中的 hostname 并非必须的，如果不配置，默认将会使用IP进行查找。

将服务注册到高可用的**Eureka**

如果注册中心是高可用的，那么各个微服务配置只需要将 defaultZone 改为如下即可：

eureka: client:

serviceUrl:

defaultZone: http://peer1:8761/eureka/,http://peer2:8762/e

ureka

* 1. 配置中心config

Spring Cloud Config提供了一种在分布式系统中外部化配置服务器和客户端的支持。配置服务器有一个中心位置，管理所有环境下的应用的外部属性。客户端和服 务器映射到相同Spring Eventment 和 PropertySrouce抽象的概念，所以非常适合

Spring应用，但也可以在任何语言开发的任何应用中使用。在一个应用从开发、测 试到生产的过程中，你可以分别地管理开发、测试、生产环境的配置，并且在迁移 的时候获取相应的配置来运行。

Config Server 存储后端默认使用git存储配置信息，因此可以很容易支持标记配置环境的版本，同时可以使用一个使用广泛的工具管理配置内容。当然添加其他方式 的存储实现也是很容易的。

#### 代码示例

##### 准备工作

我们首先配置Host

127.0.0.1 nriet-config

准备几个配置文件，命名规范为 项目名称-环境名称.yml ，本文在git 仓库：http://172.16.4.52:10080/nriet/config-repo中，创建以下几个文件：

Nriet-datacenter-api-dev.yml

Nriet-datacenter-api-prod.yml

#### 服务器端代码示例

创建一个Maven项目(nriet-config)，在pom.xml文件中添加如下内容：

启动类：

<?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>

<parent>

<groupId>com.nriet</groupId>

<artifactId>nriet-cloud</artifactId>

<version>1.0</version>

</parent>

<artifactId>nriet-config</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<name>${project.artifactId}</name>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-server</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-config-monitor</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-rabbit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-security</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

</plugins>

</build>

</project>

配置文件：application.yml

这样，一个Config Server就完成了。

按照上文，我们成功搭建了Config Server，并测试能够正常获取到git仓库中的配置信息。那么对于一个微服务应用，如何才能获取配置信息呢？

配置服务客户端示例

新建一个Maven项目，在pom.xml中添加如下内容：

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

配置文件：bootstrap.yml

spring:

application:

name: nriet-datacenter-api

cloud:

config:

uri: http://172.16.4.52:8888

profile: dev

fail-fast: **true**

username: user

password: ${CONFIG\_SERVER\_PASSWORD:password}

retry:

initial-interval: 2000

max-interval: 10000

multiplier: 2

max-attempts: 10

配置文件bootstrap.yml中配置微服务配置中心的地址，以及profile值，这个值代表我们读取git上面的文件类型。配置中心会根据spring.application.name和profile来获取配置文件。

#### 配置内容的热加载

如果不重启应用，能够做到配置的刷新吗？答案显然是可以的，本例中提供两种方法。

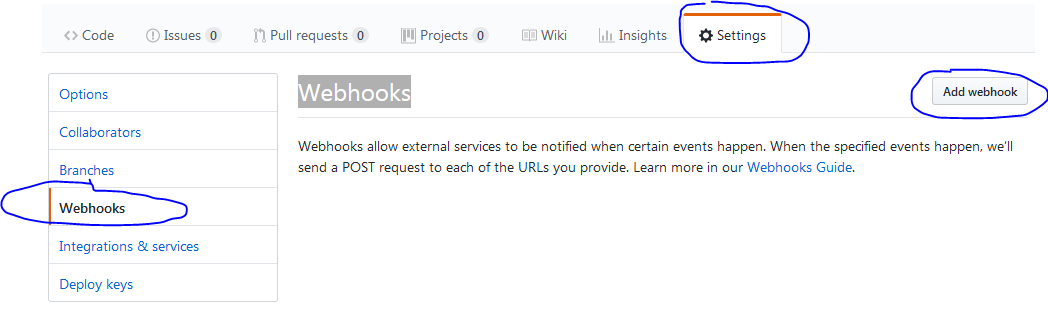
第一种手动刷新：

然后使用命令（本文使用的是curl，Linux和Windows都有curl工具，当然也可以借 助其他工具，例如Postman等）：

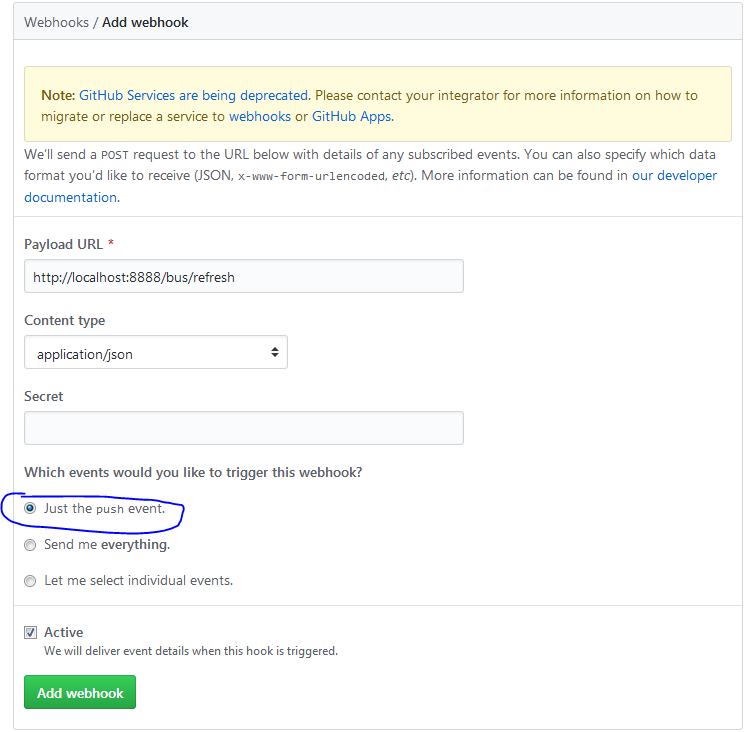
curl -X POST http://localhost:8888/bus/refresh

然后再次访问，将会看到配置已经刷新。

**第二种是git提供的自动刷新服务Webhooks：**

****

**这里以github为例，登录git帐号，选择settings-webhooks-add webhook：**

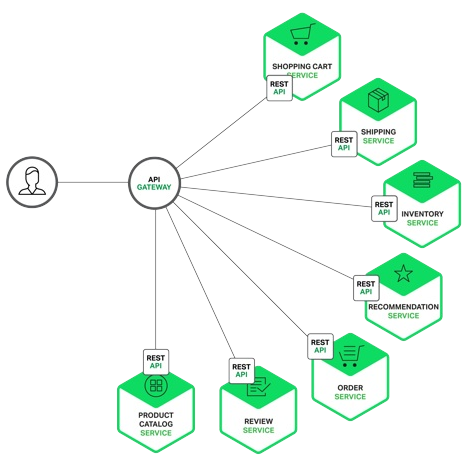
****

**选择push时请求推送配置即完成了配置中心的自动刷新任务，下次只需提交文件push到git中即可。**

* 1. **网关Gateway**

API Gateway是微服务架构中不可或缺的部分。API Gateway的定义以及存在的意义，Chris已经为大家描述过了，本文不再赘述，以下是链接：

使用API Gateway后，客户端和微服务之间的网络图变成下图：



通过API Gateway，可以统一向外部系统提供REST API。Spring Cloud中使用Zuul

作为API Gateway。Zuul提供了动态路由、监控、回退、安全等功能。

下面我们进入Zuul的学习：

#### 准备工作

为了更贴近生产，我们首先配置Host

127.0.0.1 nriet-gateway

#### **Zuul**代码示例

创建Maven项目，在pom.xml中添加如下内容：

<?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>

<parent>

<groupId>com.nriet</groupId>

<artifactId>nriet-cloud</artifactId>

<version>1.0</version>

</parent>

<artifactId>nriet-gateway</artifactId>

<version>1.0</version>

<packaging>jar</packaging>

<name>${project.artifactId}</name>

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-zuul</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-bus-amqp</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-netflix-hystrix-stream</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-sleuth</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-sleuth-stream</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-stream-binder-rabbit</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-starter-aop</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.retry</groupId>

<artifactId>spring-retry</artifactId>

</dependency>

<dependency>

<groupId>org.jolokia</groupId>

<artifactId>jolokia-core</artifactId>

</dependency>

</dependencies>

<build>

<finalName>${project.artifactId}</finalName>

<plugins>

<plugin>

<groupId>org.springframework.boot</groupId>

<artifactId>spring-boot-maven-plugin</artifactId>

</plugin>

<!-- <plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

</plugin> -->

</plugins>

</build>

</project>

启动类：

/\*\*

* 使用@EnableZuulProxy注解激活zuul。
* 跟进该注解可以看到该注解整合了@EnableCircuitBreaker、@EnableDiscov

eryClient，是个组合注解，目的是简化配置。

* @author

\*/

@SpringBootApplication

@EnableDiscoveryClient

@EnableZuulProxy

**public** **class** GatewayApplication {

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

SpringApplication.*run*(GatewayApplication.**class**, args);

}

}

配置文件：application.yml

spring:

application:

name: nriet-gateway #clientID

#配置中心地址

cloud:

config:

uri: http://nriet-config:8888

fail-fast: **true**

username: user

password: ${CONFIG\_SERVER\_PASSWORD:password}

retry:

initial-interval: 2000

max-interval: 10000

multiplier: 2

max-attempts: 10

同样使用配置中心，配置中心application.yml配置：

|  |
| --- |
| server:  port: 8060 #服务端口号  #允许注册中心推送config  management:  security:  enabled: **false**  hystrix:  command:  default:  execution:  isolation:  thread:  timeoutInMilliseconds: 20000 #断路时间  #负载均衡有限选择负载少的服务，同时有超时加权  ribbon:  ReadTimeout: 10000  ConnectTimeout: 10000  #服务注册中心  eureka:  instance:  hostname: nriet-gateway  prefer-ip-address: **true**  statusPageUrlPath: /info #info信息  client:  service-url:  defaultZone: http://user:${REGISTRY\_SERVER\_PASSWORD:password}@nriet-eureka:8761/eureka/ #服务注册地址  #配置注册中心显示info  info:  build:  name: 网关服务  version: v1.0 #可以显示在monitor界面版本信息  #网关转发规则  zuul:  ignoredServices: '\*' #忽略所有没有配置在列表中的服务  routes:  nriet-auth-service: #需要转发的clientID  path: /uaa/\*\* #转发匹配的路径  stripPrefix: **false** #转发时是否去掉前缀  sensitiveHeaders: #转发头部信息  nriet-web:  path: /analyse/\*\*  nriet-datacenter-api:  path: /api/\*\* |

Zuul：标签下面即为网管转发规则，可以自定义访问路径。

* 1. **Oauth2认证服务**

创建Maven项目nriet-auth-service，在pom.xml中添加如下内容：

|  |
| --- |
| <?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>  <parent>  <groupId>com.nriet</groupId>  <artifactId>nriet-cloud</artifactId>  <version>1.0</version>  </parent>  <artifactId>nriet-auth-service</artifactId>  <version>1.0</version>  <packaging>jar</packaging>  <name>${project.artifactId}</name>  <dependencies>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-data-jpa</artifactId>  </dependency>  <dependency>  <groupId>mysql</groupId>  <artifactId>mysql-connector-java</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-oauth2</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-sleuth</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-sleuth-stream</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-ribbon</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-aop</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.retry</groupId>  <artifactId>spring-retry</artifactId>  </dependency>  <dependency>  <groupId>org.jolokia</groupId>  <artifactId>jolokia-core</artifactId>  </dependency>  </dependencies>  <build>  <finalName>${project.artifactId}</finalName>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

创建项目bootstrap.yml，在里面添加配置中心的地址：

|  |
| --- |
| spring:  application:  name: nriet-auth-service #实例的ID  cloud:  #读取配置中心  config:  uri: http://nriet-config:8888  fail-fast: **true** #允许fail-fast错误处理机制  username: user  password: ${CONFIG\_SERVER\_PASSWORD:password}  retry:  initial-interval: 2000  max-interval: 10000  multiplier: 2  max-attempts: 10 |

在配置中心的git上传nriet-auth-service.yml：

|  |
| --- |
| server:  context-path: /uaa #服务根路径  port: 5000 #服务端口号  management:  security:  enabled: **false** #设置false后 配置中心可以推送变更  context-path: /mgmt #认证管理信息地址  eureka:  instance:  health-check-url-path: ${server.context-path}${management.context-path}/health #健康信息  status-page-url-path: ${server.context-path}${management.context-path}/info #服务信息  metadata-map:  management.context-path: ${server.context-path}${management.context-path} #获取管理信息的地址  #注册中心地址  client:  service-url:  defaultZone: http://user:${REGISTRY\_SERVER\_PASSWORD:password}@nriet-eureka:8761/eureka/  #配置注册中心显示info  info:  build:  name: Oath2认证服务  version: v1.0 #可以显示在monitor界面版本信息  #spring-数据源配置  spring:  datasource:  driver-class-name: com.mysql.jdbc.Driver  url: jdbc:mysql://172.16.4.52:13306/oauth?characterEncoding=utf8&useSSL=false  username: root  password: 123456  jpa:  show-sql: **true** #显示通过jpa执行的sql语句 |

创建启动函数AuthApplication：

|  |
| --- |
| /\*\*  \*  \* **@author** b\_wangpei  \*  \*/  @SpringBootApplication  @EnableDiscoveryClient  @EnableAuthorizationServer  **public** **class** AuthApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(AuthApplication.**class**, args);  }  } |

这里@EnableAuthorizationServer注解表示这是一个oath认证服务端；

然后创建config类，加入EnableAuthorizationServer注解，Configuration会让服务启动时spring自动装载这个配置。

|  |
| --- |
| @Configuration  @EnableAuthorizationServer  **public** **class** OAuthConfiguration **extends** AuthorizationServerConfigurerAdapter |

在类中着重需要配置，客户端访问时的认证信息：

|  |
| --- |
| @Override  **public** **void** configure(ClientDetailsServiceConfigurer clients)  **throws** Exception {  clients.jdbc(dataSource)  .passwordEncoder(passwordEncoder)  .withClient("client")  .secret("secret")  .authorizedGrantTypes("password", "refresh\_token") //授权类型  .scopes("read", "write")  .accessTokenValiditySeconds(3599)  .refreshTokenValiditySeconds(2592000)  ;  } |

认证信息与上文中的其他应用中是对应的，这里创建一个客户端id为client以及权限范围scopes等信息。然后再配置客户端允许的帐号信息：

|  |
| --- |
| @Configuration  @Order(-20) //类加载顺序  **protected** **static** **class** AuthenticationManagerConfiguration **extends** GlobalAuthenticationConfigurerAdapter {  @Autowired  **private** DataSource dataSource;  @Override  **public** **void** init(AuthenticationManagerBuilder auth) **throws** Exception {  auth.jdbcAuthentication().dataSource(dataSource)  .withUser("dave").password("secret").roles("USER")  .and()  .withUser("anil").password("password").roles("ADMIN")  ;  }  } |

这是OAuthConfiguration中的一个内部类。

创建一个接口，供客户端验证验证授权用户信息：

|  |
| --- |
| @RestController  @RequestMapping("/")  **public** **class** UserController {  @GetMapping(value = "/current")  **public** Principal getUser(Principal principal) {  **return** principal;  }  } |

同时开放这个接口的访问权限，新建ResourceServerConfiguration.java：

|  |
| --- |
| @Configuration  @EnableResourceServer  **public** **class** ResourceServerConfiguration **extends** ResourceServerConfigurerAdapter {  @Override  **public** **void** configure(HttpSecurity http) **throws** Exception {  http  .requestMatchers().antMatchers("/current")  .and()  .authorizeRequests()  .antMatchers("/current").access("#oauth2.hasScope('read')");  }  } |

这样微服务授权认证服务端已经建立完成了，那么授权客户端需要如何使用呢？因下文中关于消费者章节会说明代码中如何继承基类完成认证工作，这里不再重复说明，大家可以看一下客户端的配置文件：

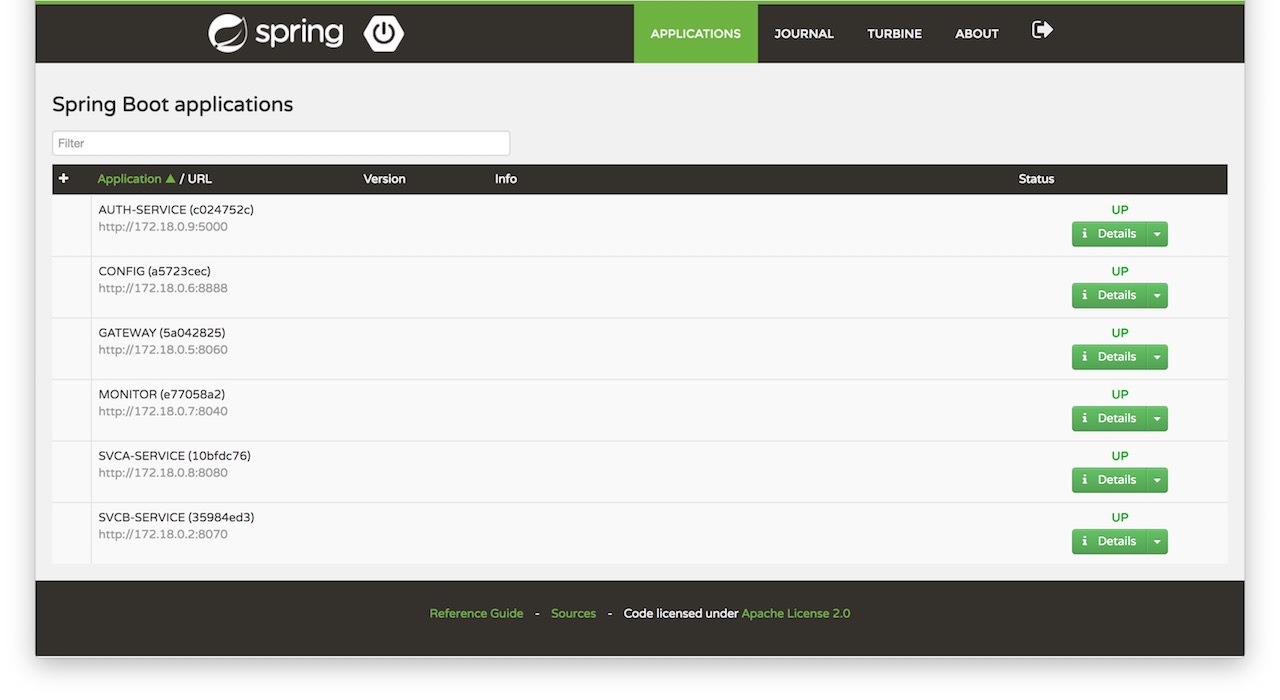
|  |
| --- |
| security:  user:  name: anil  password: password  oauth2:  resource:  userInfoUri: http://172.16.4.52:8060/uaa/current  client:  clientId: client  clientSecret: secret  accessTokenUri: http://172.16.4.52:8060/uaa/oauth/token  grant\_type: password  scope: read%20write  clientAuthenticationScheme: header |

这里的配置项与上文中认证服务端的配置项是一一对应的。

* + 1. **monitor**

分布式微服务健康监控系统，这里使用的是spring-boot-admin。Spring Boot Admin 是一个管理和监控你的 Spring Boot 应用程序的应用程序。 这些应用程序通过 Spring Boot Admin Client（通过 HTTP）注册或者使用 Spring Cloud（例如 Eureka）发现。 UI只是 Spring Boot Actuator 端点上的一个 AngularJs 应用程序。

本次使用Eureka作为服务注册发现程序。



* 1. monitor服务监控

#### 准备工作

创建一个 SpringBoot 项目nriet-monitor，把它当作 server 端，pom如下：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <project xmlns=*"http://maven.apache.org/POM/4.0.0"*  xmlns:xsi=*"http://www.w3.org/2001/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>  <parent>  <groupId>com.nriet</groupId>  <artifactId>nriet-cloud</artifactId>  <version>1.0</version>  </parent>  <artifactId>nriet-monitor</artifactId>  <version>1.0</version>  <packaging>jar</packaging>  <name>${project.artifactId}</name>  <dependencies>  <dependency>  <groupId>de.codecentric</groupId>  <artifactId>spring-boot-admin-server</artifactId>  </dependency>  <dependency>  <groupId>de.codecentric</groupId>  <artifactId>spring-boot-admin-server-ui</artifactId>  </dependency>  <dependency>  <groupId>de.codecentric</groupId>  <artifactId>spring-boot-admin-server-ui-login</artifactId>  </dependency>  <dependency>  <groupId>de.codecentric</groupId>  <artifactId>spring-boot-admin-server-ui-hystrix</artifactId>  </dependency>  <dependency>  <groupId>de.codecentric</groupId>  <artifactId>spring-boot-admin-server-ui-turbine</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-turbine-stream</artifactId>  <exclusions>  <exclusion>  <artifactId>netty-transport-native-epoll</artifactId>  <groupId>io.netty</groupId>  </exclusion>  <exclusion>  <artifactId>netty-codec-http</artifactId>  <groupId>io.netty</groupId>  </exclusion>  </exclusions>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-stream-rabbit</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency>  </dependencies>  <build>  <finalName>${project.artifactId}</finalName>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

我们可以看到，该boot核心依赖为spring-boot-admin-server与spring-boot-admin-server-ui；server负责获取服务注册情况，并监控微服务的运行监控及运行环境等信息，ui部分负责可视化的界面展示。

配置文件application.yml：

|  |
| --- |
| #监控级别  logging:  level:  org.springframework.cloud.netflix.zuul.filters.post.SendErrorFilter: error  #服务端口号  server:  port: 8040  #断路器监控端口号变量  turbine:  stream:  port: 8041  #注册中心  eureka:  instance:  hostname: nriet-monitor #本实例的id  prefer-ip-address: **true** #是否运行ip访问  statusPageUrlPath: /info #info信息  metadata-map:  user.name: ${security.user.name}  user.password: ${security.user.password}  client:  service-url:  defaultZone: http://user:${REGISTRY\_SERVER\_PASSWORD:password}@nriet-eureka:8761/eureka/ #注册中心地址  ##配置注册中心显示info  info:  build:  name: 微服务监控  version: v1.0 #可以显示在monitor界面版本信息  spring:  rabbitmq:  host: rabbitmq  username: nriet  password: nriet2018  #追踪监控项配置  boot:  admin:  routes:  endpoints: env,metrics,trace,dump,jolokia,info,configprops,trace,logfile,refresh,flyway,liquibase,heapdump,loggers,auditevents,hystrix.stream  turbine:  clusters: default  location: http://nriet-monitor:${turbine.stream.port}  #不允许网页直接访问  security:  user:  name: admin  password: ${MONITOR\_SERVER\_PASSWORD:admin} |

monitor的启动与查看

启动monitor

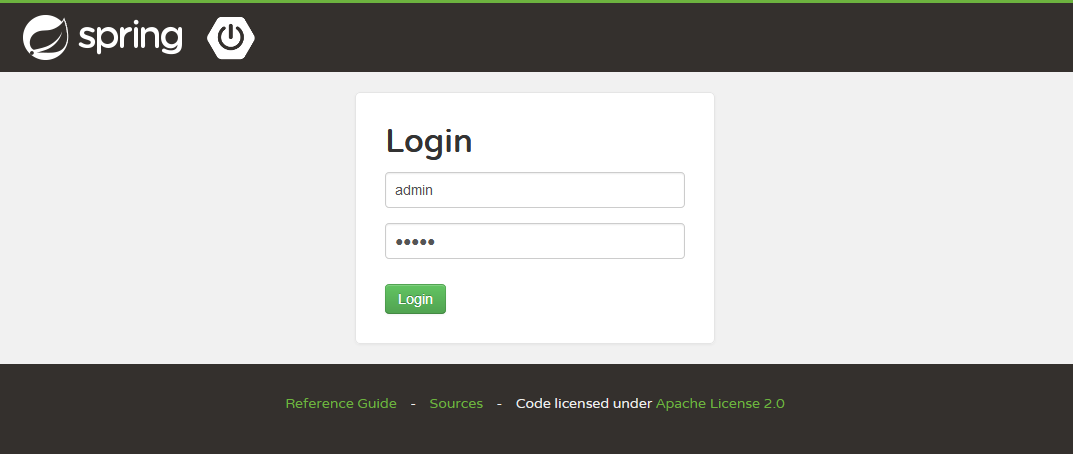
在启动类 SpringbootApplication.java 中引入注解 **@EnableAdminServer** ，

然后运行项目：

访问 <http://localhost:8084/> 即可：

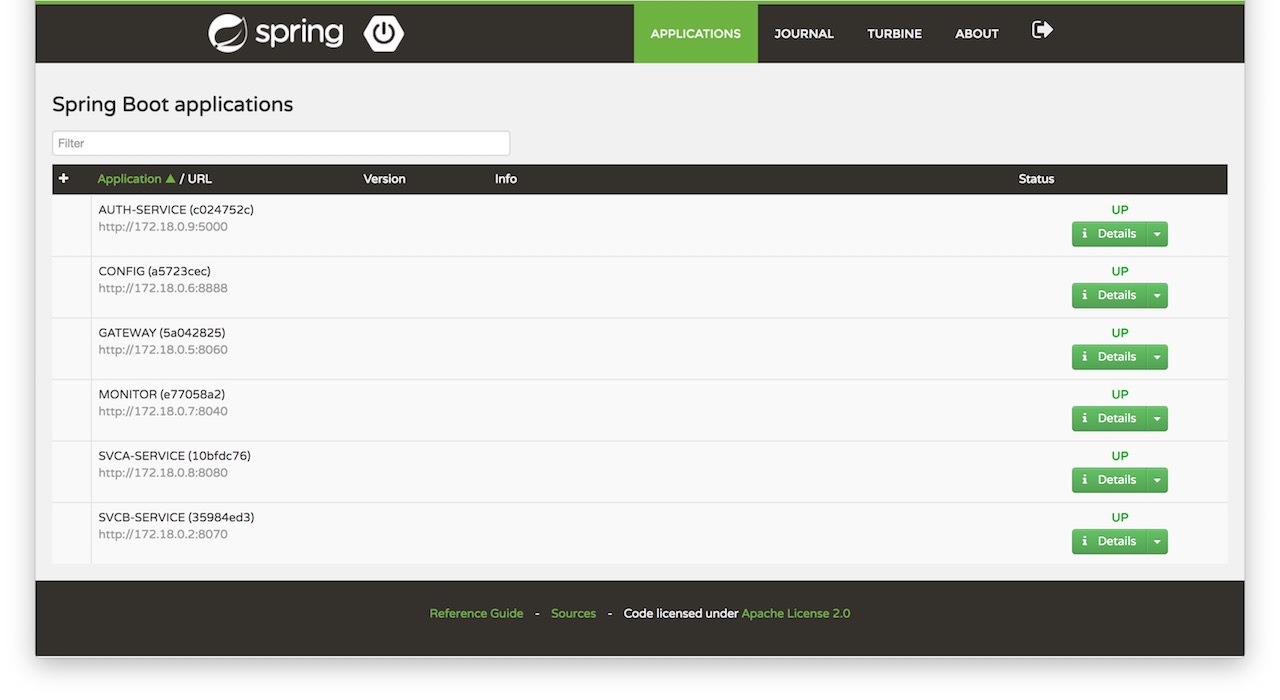
输入 http://localhost:8040/ 访问monitor，可查看到如下界面：

通过配置在文件中的用户名密码访问（默认：admin/admin）：

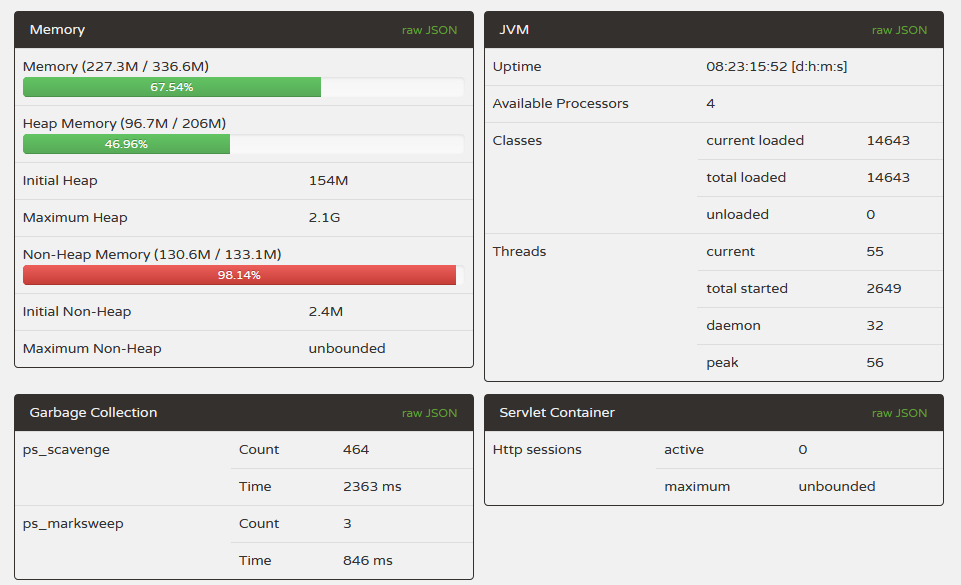


* + 1. monitor常用功能

微服务健康情况监控：



点击图中的 **detail** 按钮：可以看到应用程序的健康值、内存、JVM、GC 等信息：



* 1. zipkin追踪服务

Zipkin是一个链路跟踪工具，可以用来监控微服务集群中调用链路的通畅情况。

创建子项目nriet-zipkin，pom.xml中加入依赖：

|  |
| --- |
| <?xml version=*"1.0"* encoding=*"UTF-8"*?>  <project xmlns=*"http://maven.apache.org/POM/4.0.0"*  xmlns:xsi=*"http://www.w3.org/2001/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>  <parent>  <groupId>com.nriet</groupId>  <artifactId>nriet-cloud</artifactId>  <version>1.0</version>  </parent>  <artifactId>nriet-zipkin</artifactId>  <version>1.0</version>  <packaging>jar</packaging>  <name>${project.artifactId}</name>  <dependencies>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-sleuth</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>  </dependency>  <dependency>  <groupId>io.zipkin.java</groupId>  <artifactId>zipkin-autoconfigure-ui</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-security</artifactId>  </dependency>  </dependencies>  <build>  <finalName>${project.artifactId}</finalName>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build>  </project> |

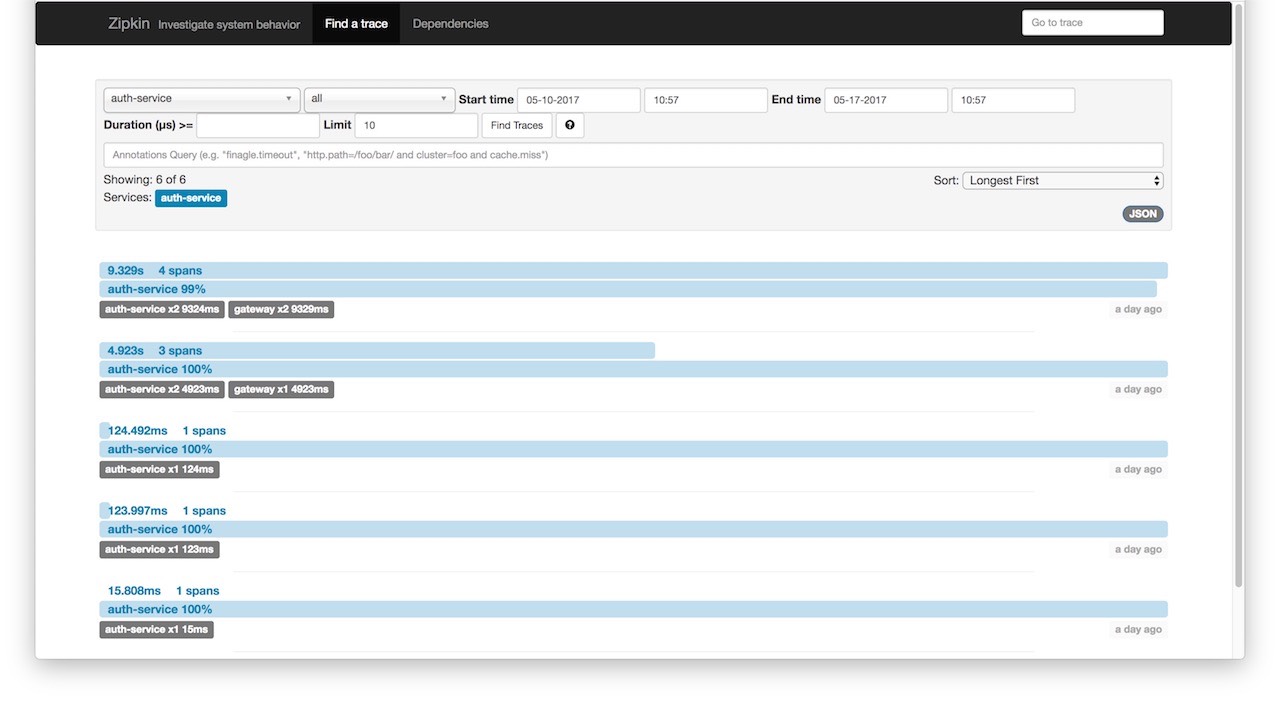
Zipkin提供多种追踪信息获取方式，因本例中使用boot-admin进行服务健康监控，同时使用rabbitmq作为监控信息消息队列，所以在zipkin服务中需要加入rabbitmq的相应配置即可追踪到服务调用情况：

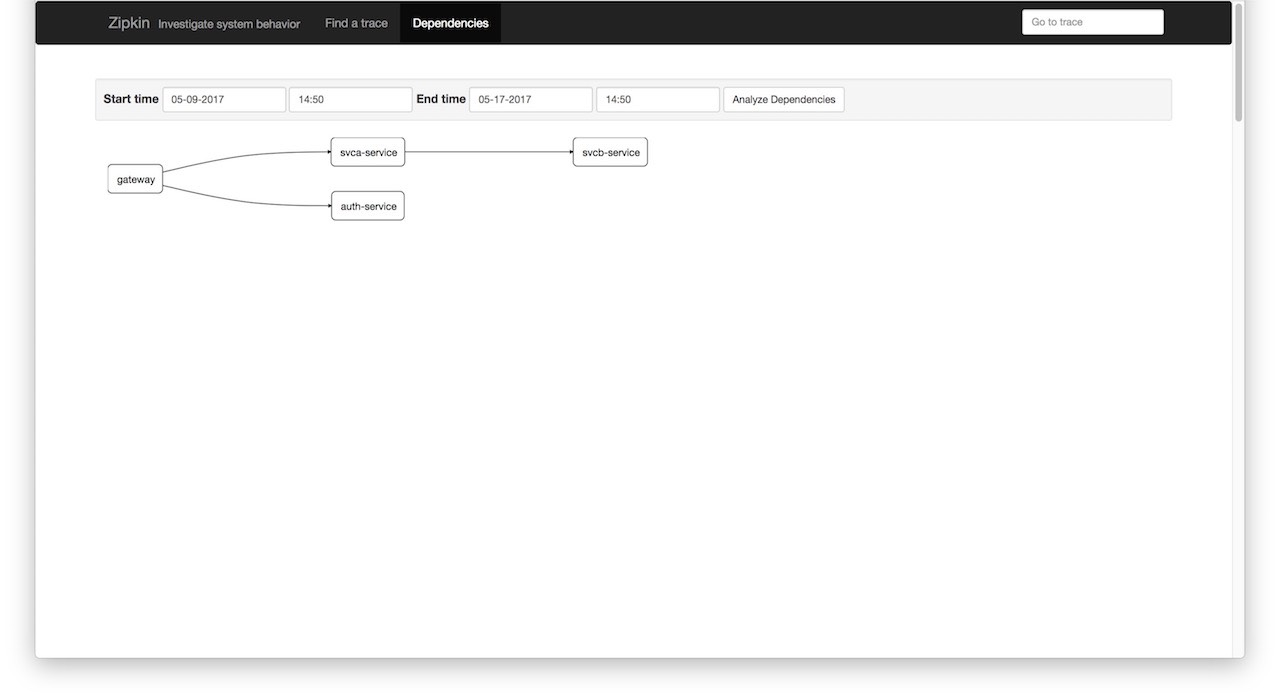
|  |
| --- |
| spring:  application:  name: nriet-zipkin  rabbitmq:  host: rabbitmq  username: nriet  password: nriet2018 |

主函数中需要加入@EnableZipkinStreamServer注解：

|  |
| --- |
| @SpringBootApplication  @EnableZipkinStreamServer  **public** **class** ZipkinApplication {  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(ZipkinApplication.**class**, args);  }  } |

启动后访问地址 <http://localhost:9411/> (admin/admin)





* 1. 服务提供者

#### 服务提供者和服务消费者

下面这张表格，简单描述了服务提供者/消费者是什么：

|  |  |
| --- | --- |
| 名词 | 概念 |
| 服务提供者 | 服务的被调用方（即：为其他服务提供服务的服务） |
| 服务消费者 | 服务的调用方（即：依赖其他服务的服务） |

#### 服务提供者代码示例

这是一个稍微有点复杂的程序。我们使用spring-datacenter-api为例，将该服务注册到注册中心nriet-Eureka中,并通过nriet-gateway进行访问。

创建一个Maven工程，并在pom.xml中添加如下内容：

微服务父类依赖：

<parent>

<groupId>com.nriet</groupId>

<artifactId>nriet-cloud</artifactId>

<version>1.0</version>

</parent>

微服务组件依赖：

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-eureka</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-sleuth</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-sleuth-stream</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-stream-binder-rabbit</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-config</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-bus-amqp</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-feign</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-hystrix</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-netflix-hystrix-stream</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-ribbon</artifactId>  </dependency>  <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-oauth2</artifactId>  </dependency> |

然后在application.yml中配置注册中心的信息：

|  |
| --- |
| eureka:  instance:  hostname: nriet-datacenter-api  instance-id: nriet-datacenter-api #本实例ID  prefer-ip-address: **true** #支持ip访问  statusPageUrlPath: /info #info信息  # ip-address: 172.16.4.51 #ip地址  client:  service-url:  defaultZone: http://user:${REGISTRY\_SERVER\_PASSWORD:password}@nriet-eureka:8761/eureka/ |

注意上面的nriet-eureka可以使用ip地址替代，不然不需要在hostname中进行相应ip地址的配置

编写Spring Boot启动程序，通过@EnableDiscoveryClient注解，即可将nriet-datacenter-api服务注册到nriet-eureka上面去

@SpringBootApplication @EnableDiscoveryClient

public class NrietDatacenterApplication {

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

}

}

安全认证配置，微服务的restapi可以选择是否对外开放，对于有安全性要求的服务，我们可以打开认证服务：

|  |
| --- |
| @Configuration  @EnableResourceServer  @EnableGlobalMethodSecurity(prePostEnabled = **true**, jsr250Enabled = **true**)  **public** **class** DataCenterConfiguration **extends** ResourceServerConfigurerAdapter {  @Override  **public** **void** configure(HttpSecurity http) **throws** Exception {  http.authorizeRequests()  .anyRequest().authenticated();  // .anyRequest().permitAll();  }  } |

Authenticated表示所有的外部请求必须得到微服务的权限认证，认证详情在Oauth章节会有详细解说。如果是permitAll则表示api对外开放无需身份认证。

* 1. 服务消费者

上文我们创建了注册中心，以及服务的提供者nriet-datacenter-api，并将服务提供者注册到了注册中心上。

要想消费nriet-datacenter-api的服务是很简单的，我们只需要使用RestTemplate或者Feign即可.

对于需要请求必须通过微服务认证才能放的api时，需要用到nriet-common这个工具boot，在application主函数中继承基础类：

|  |
| --- |
| @EnableCircuitBreaker  @EnableOAuth2Client  @SpringBootApplication(exclude={DataSourceAutoConfiguration.**class**,HibernateJpaAutoConfiguration.**class**})  **public** **class** ManageWebApp **extends** BasicApplication{  @Autowired  **private** RestTemplateBuilder builder;  // 使用RestTemplateBuilder来实例化RestTemplate对象，spring默认已经注入了RestTemplateBuilder实例  @Bean  **public** RestTemplate restTemplate() {  **return** builder.build();  }    **public** **static** **void** main(String[] args) {  **new** SpringApplicationBuilder(ManageWebApp.**class**).run(args);  }  } |

#### 配置文件application.yml：

|  |
| --- |
| security:  user:  name: anil  password: password  oauth2:  resource:  userInfoUri: http://nriet-gateway:8060/uaa/current  client:  clientId: client  clientSecret: secret  accessTokenUri: http://nriet-gateway:8060/uaa/oauth/token  grant\_type: password  scope: read%20write |

**extends** BasicApplication中含有对tocken的管理及维护任务，restTemplate对象是需要在调用restApi时用到的这里需要提前注入。

接下来需要建立访问对应业务接口的service，这里以testService为例：

|  |
| --- |
| @Service  **public** **class** ITestService **extends** BasicService<User> {    } |
|  |

只要继承BasicService基类，就获得了一个经过重写的restTemplate对象，可以在方法里直接使用this捕获到对应的方法，通过该对象调用restAPI则会通过认证，框架部分将tocken及认证过经过封装，我们开发的时候无需关心。

|  |
| --- |
| this.getForObject(apiURL, rsClass, parameterMap);  this.postForEntity(apiURL, rsClass, parameterMap)； |

apiURL:请求的rest地址

rsClass：返回类型

parameterMap：get/post时发送的参数

### **2.10 Ribbon**

#### **Ribbon**介绍

Ribbon是Netflix发布的开源项目，主要功能是提供客户端的软件负载均衡算法，将Netflix的中间层服务连接在一起。Ribbon客户端组件提供一系列完善的配置项如连接超时，重试等。简单的说，就是在配置文件中列出Load Balancer后面所有的机器，Ribbon会自动的帮助你基于某种规则（如简单轮询，随即连接等）去连接这些 机器。我们也很容易使用Ribbon实现自定义的负载均衡算法。简单地说，Ribbon是 一个客户端负载均衡器。

Ribbon工作时分为两步：第一步先选择 Eureka Server, 它优先选择在同一个Zone 且负载较少的Server；第二步再根据用户指定的策略，在从Server取到的服务注册列表中选择一个地址。其中Ribbon提供了三种策略：轮询、断路器和根据响应时间 加权。

#### **Ribbon**配置示例

创建一个Maven项目，并在pom.xml中加入如下内容：

|  |
| --- |
| <dependency>  <groupId>org.springframework.cloud</groupId>  <artifactId>spring-cloud-starter-ribbon</artifactId>  </dependency> |

当于Spring Cloud应用引入Ribbon和Eureka的时候，会触发Eureka中实现的Ribbon的自动化配置。

接下来消费者需要在restTemplate对象注入的时候加上@LoadBalanced注解，即实现的负载均衡。

负载均衡的微服务一定要注意配置成不同的端口号，但是注册到eureka 的 clientid（spring.application.name）必须一致；

### **2.11 Feign**

#### **Feign**介绍

Feign是一个声明式的web service客户端，它使得编写web service客户端更为容易。创建接口，为接口添加注解，即可使用Feign。Feign可以使用Feign注解或者JAX-RS注解，还支持热插拔的编码器和解码器。Spring Cloud为Feign添加了Spring MVC的注解支持，并整合了Ribbon和Eureka来为使用Feign时提供负载均衡。

#### **Feign**示例

创建一个Maven项目，并在pom.xml添加如下内容：

<dependencies>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-eureka</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-feign</artifactId>

</dependency>

启动类： Application.java

/\*\*

* 使用@EnableFeignClients开启Feign
* @author eacdy

\*/

@SpringBootApplication @EnableFeignClients @EnableDiscoveryClient

public class FeignApplication { public static void main(String[] args) {

SpringApplication.run(FeignApplication.class, args);

}

}

Feign范例测试：

/\*\*

* 使用@FeignClient("nriet-datacenter-api")注解绑定服务，还可以使用url参数指定一个URL。
* @author

\*/

@FeignClient(name = "nriet-datacenter-api")

public interface DatacenterFeignClient {

@RequestMapping("/{id}")

public User findByIdFeign(@RequestParam("id") Long id);

}

application.yml

server: port: 9090

spring: application:

name: consumer-user

eureka:

client: serviceUrl:

defaultZone: http://nriet-eureka:8761/eureka/ instance:

preferIpAddress: true ribbon:

eureka:

enabled: true

同样的，启动该应用，我们会发现和Ribbon示例一样实现了负载均衡。

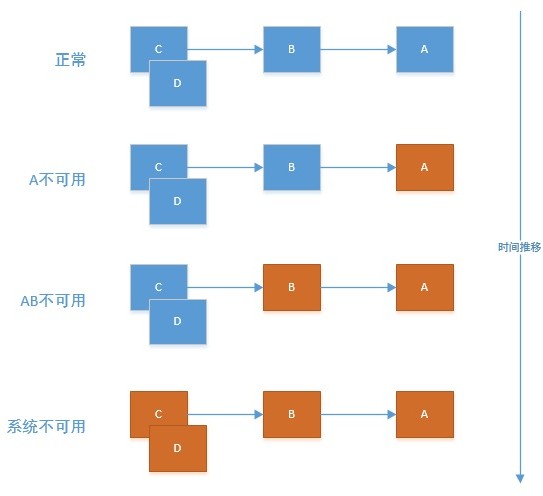
* 1. 熔断器

#### 雪崩效应

在微服务架构中通常会有多个服务层调用，基础服务的故障可能会导致级联故障， 进而造成整个系统不可用的情况，这种现象被称为服务雪崩效应。服务雪崩效应是 一种因“服务提供者”的不可用导致“服务消费者”的不可用,并将不可用逐渐放大的过程。

如果下图所示：A作为服务提供者，B为A的服务消费者，C和D是B的服务消费者。

A不可用引起了B的不可用，并将不可用像滚雪球一样放大到C和D时，雪崩效应就 形成了。

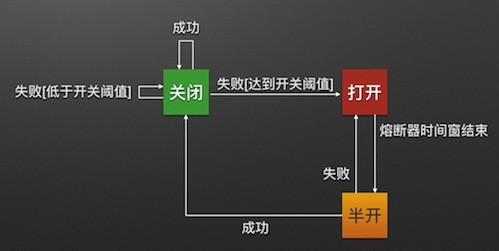


#### 熔断器（**CircuitBreaker**）

熔断器的原理很简单，如同电力过载保护器。它可以实现快速失败，如果它在一段 时间内侦测到许多类似的错误，会强迫其以后的多个调用快速失败，不再访问远程 服务器，从而防止应用程序不断地尝试执行可能会失败的操作，使得应用程序继续 执行而不用等待修正错误，或者浪费CPU时间去等到长时间的超时产生。熔断器也 可以使应用程序能够诊断错误是否已经修正，如果已经修正，应用程序会再次尝试 调用操作。

熔断器模式就像是那些容易导致错误的操作的一种代理。这种代理能够记录最近调 用发生错误的次数，然后决定使用允许操作继续，或者立即返回错误。

熔断器开关相互转换的逻辑如下图：



* 1. **Hystrix**

#### **Hystrix**

在Spring Cloud中使用了Netflix开发的Hystrix来实现熔断器。下面我们依然通过几个简单的代码示例，进入Hystrix的学习：

通用方式使用**Hystrix**

代码示例：

新建一个Maven项目，在pom.xml中添加如下内容：

<!-- 整合ribbon -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-ribbon</artifactId>

</dependency>

<!-- 整合hystrix -->

<dependency>

<groupId>org.springframework.cloud</groupId>

<artifactId>spring-cloud-starter-hystrix</artifactId>

</dependency>

启动类使用@EnableCircuitBreaker注解开启断路器功能：

@Service

public class RibbonHystrixService { @Autowired

private RestTemplate restTemplate;

private static final Logger LOGGER = LoggerFactory.getLogger(R ibbonHystrixService.class);

/\*\*

* 使用@HystrixCommand注解指定当该方法发生异常时调用的方法
* @param id id
* @return 通过id查询到的用户

\*/

@HystrixCommand(fallbackMethod = "fallback") public User findById(Long id) {

return this.restTemplate.getForObject(“APIUrl”);

}

/\*\*

* hystrix fallback方法
* @param id id
* @return 默认的用户

\*/

public User fallback(Long id) {

RibbonHystrixService.LOGGER.info("异常发生，进入fallback方法，接收的参数：id = {}", id);

User user = new User(); user.setId(-1L);

user.setUsername("default username"); user.setAge(0);

return user;

}

}

#### **Feign**使用**Hystrix**

代码示例

在Feign中使用Hystrix是非常简单的事情，因为Feign已经集成了Hystrix。我们只需项目的代码做一点修改，增加一个fallback类作为熔断后调用的实现类，修改为如下即可：

/\*\*

* 使用@FeignClient注解的fallback属性，指定fallback类
* @author

\*/

@FeignClient(name = "nriet-datacenter-api", fallback = Hys trixClientFallback.class)

public interface UserFeignHystrixClient { @RequestMapping("/{id}")

public User findByIdFeign(@RequestParam("id") Long id);

/\*\*

* + 这边采取了和Spring Cloud官方文档相同的做法，将fallback类作为内部类放入Feign的接口中，当然也可以单独写一个fallback类。
  + @author

\*/ @Component

static class HystrixClientFallback implements UserFeignHystrix Client {

private static final Logger LOGGER = LoggerFactory.getLogger (HystrixClientFallback.class);

/\*\*

* + - hystrix fallback方法
    - @param id id
    - @return 默认的用户

\*/ @Override

public User findByIdFeign(Long id) {

HystrixClientFallback.LOGGER.info("异常发生，进入fallback方法

，接收的参数：id = {}", id);

User user = new User(); user.setId(-1L);

user.setUsername("default username"); user.setAge(0);

return user;

}

}

* 1. **Turbine**

#### **Turbine**

在复杂的分布式系统中，相同服务的结点经常需要部署上百甚至上千个，很多时候，运维人员希望能够把相同服务的节点状态以一个整体集群的形式展现出来，这 样可以更好的把握整个系统的状态。 为此，Netflix提供了一个开源项目

（Turbine）来提供把多个hystrix.stream的内容聚合为一个数据源供Dashboard展 示。

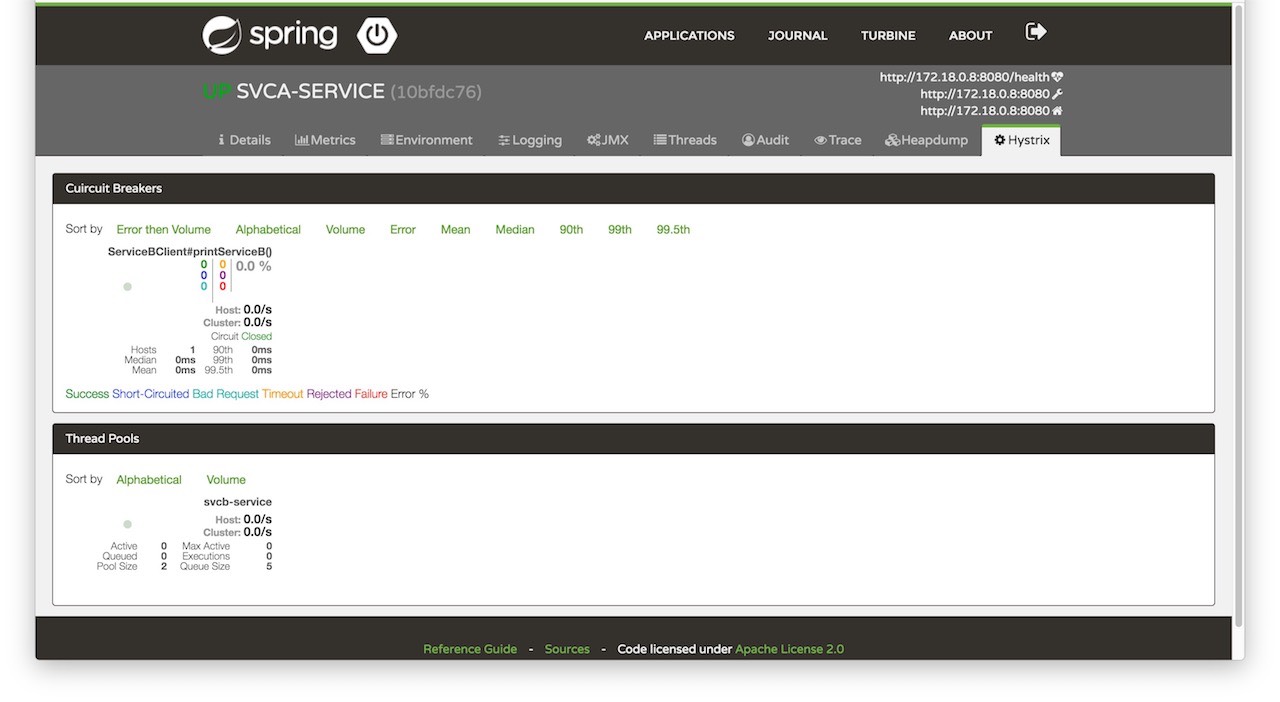
和Hystrix Dashboard一样，Turbine也可以下载war包部署到Web容器，本文不做赘述。下面讨论Spring Cloud是怎么使用Turbine的。

#### 配置示例

新建Maven项目，并在pom.xml中添加如下内容：

本例中Turbine和Hystrix Dashboard部分都集成在monitor内显示：

点击菜单TURBINE



1. 使用**Docker**构建微服务

准备工作

|  |  |  |  |
| --- | --- | --- | --- |
| 安装软件 | 版本 | 功能 | 必要程度 |
| Docker | 1.12.1 | Docker | 是 |
| CentOS7.0 或其他系统 | 7.0 | Docker的宿主机，本章的讲解都是在CentOS 7.0 下进行的。Docker现已支持Windows系统，但考虑到绝大多数Docker容器还是跑在Linux环境下的，故而只讲解Linux环境下的使用。Windows下 的安装使用大致类似，请读者自行研究。 | 是 |
| Maven | 3.3.9 |  | 是 |
| JDK | 8u65 |  | 是 |

注意：

本章的讲解都是在CentOS7下进行的，建议新手使用CentOS 7.x进入学习。

**CentOS**下**JDK 1.8**的安装

1. 到Oracle官网下载好 jdk-8u65-linux-x64.rpm 备用
2. 卸载系统自带java

yum -y remove [上面查出来的东西，多个用空格分隔]

# 如果有结果出来，则说明自带了java

# 查询出已经安装的java

java -version

rpm -qa|grep java

1. 安装JDK

cd /usr

mkdir /usr/java

rpm -ivh jdk-8u65-linux-x64.rpm

1. 配置环境变量：

vim /etc/profile

找 到 ： export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE

HISTCONTROL 这一行，并在其下面一行添加如下内容：

# 设置java环境变量

export JAVA\_HOME=/usr/java/jdk1.8.0\_65 # 根据情况修改

export PATH=$JAVA\_HOME/bin:$PATH

export CLASSPATH=.:$JAVA\_HOME/lib/dt.jar:$JAVA\_HOME/lib/tool s.jar

1. 使环境变量生效：

source /ect/profile

1. 测试：

java -version javac

**CentOS**下**Maven**的安装

Maven的安装比较简单，只需要下载后解压，配置环境变量即可。

1. 下载并解压：

cd /opt

wget <http://apache.fayea.com/maven/maven-3/3.3.9/binaries/ap> ache-maven-3.3.9-bin.tar.gz

tar -zxvf apache-maven-3.3.9-bin.tar.gz

1. 配置环境变量：

vim /etc/profile

找 到 export PATH USER LOGNAME MAIL HOSTNAME HISTSIZE

HISTCONTROL ， 并在其下面一行添加如下内容：

# 设置Maven环境变量

export MAVEN\_HOME=/opt/apache-maven-3.3.9/ export PATH=$MAVEN\_HOME/bin:$PATH

1. 使环境变量生效：

source /etc/profile

1. 测试

mvn -version

1. 修改Maven配置

本地仓库路径配置：

<!-- 本地仓库路径配置 -->

<localRepository>/path/to/local/repo</localRepository>

国内Maven镜像配置：

<mirror>

<id>aliyun</id>

<mirrorOf>central</mirrorOf>

<name>Aliyun Central mirror</name>

<url><http://maven.aliyun.com/nexus/content/groups/public></ url>

</mirror>

**CentOS**下**Git**的安装

yum install git

* 1. **Docker**介绍

**Docker**概览

Docker是一个用于开发、交付和运行应用的开放平台，Docker被设计用于更快地交 付应用。Docker可以将应用程序和基础设施层隔离，并且可以将基础设施当作程序 一样进行管理。使用Docker，可以更快地打包代码、测试以及部署，并且可以减少 从编写到部署运行代码的周期。

Docker将内核容器特性（LXC）、工作流和工具集成，以帮助管理和部署应用。

什么是**Docker**

核心是，Docker了一种在安全隔离的容器中运行近乎所有应用的方式，这种隔离性 和安全性允许你在同一个主机上同时运行多个容器，而容器的这种轻量级特性，无 需消耗运行hpervisor所需的额外负载，意味着你可以节省更多的硬件资源。

基于容器虚拟化的工具或平台可提供：

将应用（包括支撑组件）放入Docker容器中

分发和交付这些容器给团队，便于后续的开发和测试

将容器部署到生产环境中，生产环境可以是本地的数据中心，也可以在云端。

自从上世纪 90 年代硬件虚拟化被主流的技术广泛普及之后，对数据中心而言，发生的最大的变革莫过于容器和容器管理工具，例如：Docker。在过去的一年内，

Docker 技术已经逐渐走向成熟，并且推动了大型初创公司例如 Twitter 和 Airbnb 的发展，甚至在银行、连锁超市、甚至 NASA 的数据中心都赢得了一席之地。当我几年前第一次直到 Docker 的时候，我还对 Docker 的未来持怀疑的态度，我认为他们是把以前的 Linux 容器的概念拿出来包装了一番推向市场。但是使用 Docker 成功进行了几个项目 例如 Spantree 之后，我改变了我的看法：Docker 帮助我们节省了大量的时间和经历，并且已经成为我们技术团队中不可或缺的工具。 GitHub 上面每天都会催生出各式各样的工具、形态各异的语言和千奇百怪的概念。如果你和我一样，没有时间去把他们全部都测试一遍，甚至没有时间去亲自测试 Docker， 那么你可以看一下我的这篇文章：我将会用我们在 Docker 中总结的经验来告诉你什么是 Docker、为什么 Docker 会这么火。

**Docker** 是容器管理工具 Docker 是一个轻量级、便携式、与外界隔离的容器，也是一个可以在容器中很方便地构建、传输、运行应用的引擎。和传统的虚拟化技术不同的是，Docker 引擎并不虚拟出一台虚拟机，而是直接使用宿主机的内核和硬

件，直接在宿主机上运行容器内应用。也正是得益于此，Docker 容器内运行的应用和宿主机上运行的应用性能差距几乎可以忽略不计。 但是 Docker 本身并不是一个容器系统，而是一个基于原有的容器化工具 LXC 用来创建虚拟环境的工具。类似 LXC 的工具已经在生产环境中使用多年，Docker 则基于此提供了更加友好的镜像管理工具和部署工具。

**Docker** 不是虚拟化引擎 Docker 第一次发布的时候，很多人都拿 Docker 和虚拟机

VMware、KVM 和 VirtualBox 比较。尽管从功能上看，Docker 和虚拟化技术致力于解决的问题都差不多，但是 Docker 却是采取了另一种非常不同的方式。虚拟机是虚拟出一套硬件，虚拟机的系统进行的磁盘操作，其实都是在对虚拟出来的磁盘 进行操作。当运行 CPU 密集型的任务时，是虚拟机把虚拟系统里的 CPU 指令“翻译”成宿主机的CPU指令并进行执行。两个磁盘层，两个处理器调度器，两个操作系统消耗的内存，所有虚拟出的这些都会带来相当多的性能损失，一台虚拟机所消 耗的硬件资源和对应的硬件相当，一台主机上跑太多的虚拟机之后就会过载。而Docker 就没有这种顾虑。Docker 运行应用采取的是“容器”的解决方案：使用

namespace 和 CGroup 进行资源限制，和宿主机共享内核，不虚拟磁盘，所有的容器磁盘操作其实都是对 /var/lib/docker/ 的操作。简言之，Docker 其实只是在宿主机中运行了一个受到限制的应用程序。 从上面不难看出，容器和虚拟机的概念并不相同，容器也并不能取代虚拟机。在容器力所不能及的地方，虚拟机可以大显身手。例如：宿主机是 Linux，只能通过虚拟机运行 Windows，Docker 便无法做

到。再例如，宿主机是 Windows，Windows 并不能直接运行 Docker，Windows上的 Docker 其实是运行在 VirtualBox 虚拟机里的。

**Docker** 使用层级的文件系统 前面提到过，Docker 和现有容器技术 LXC 等相比， 优势之一就是 Docker 提供了镜像管理。对于 Docker 而言，镜像是一个静态的、只读的容器文件系统的快照。然而不仅如此，Docker 中所有的磁盘操作都是对特定的Copy-On-Write文件系统进行的。下面通过一个例子解释一下这个问题。 例如我们要建立一个容器运行 JAVA Web 应用，那么我们应该使用一个已经安装了

JAVA 的镜像。在 Dockerfile（一个用于生成镜像的指令文件）中，应该指明“基于

JAVA 镜像”，这样 Docker 就会去 Docker Hub Registry 上下载提前构建好的 JAVA 镜像。然后再 Dockerfile 中指明下载并解压 Apache Tomcat 软件到 /opt/tomcat 文件夹中。这条命令并不会对原有的 JAVA 镜像产生任何影响，而仅仅是在原有镜像上面添加了一个改动层。当一个容器启动时，容器内的所有改动层都会启动，容器 会从第一层中运行 /usr/bin/java 命令，并且调用另外一层中的 /opt/tomcat/bin 命

令。实际上，Dockerfile 中每一条指令都会产生一个新的改动层，即便只有一个文件被改动。如果用过 Git 就能更清楚地认识这一点，每条指令就像是每次 commit， 都会留下记录。但是对于 Docker 来说，这种文件系统提供了更大的灵活性，也可以更方便地管理应用程序。 我们Spantree的团队有一个自己维护的含有 Tomcat 的镜像。发布新版本也非常简单：使用 Dockerfile 将新版本拷贝进镜像从而创建一个新镜像，然后给新镜像贴上版本的标签。不同版本的镜像的不同之处仅仅是一个 90 MB 大小的 WAR 文件，他们所基于的主镜像都是相同的。如果使用虚拟机去维护这些不同的版本的话，还要消耗掉很多不同的磁盘去存储相同的系统，而使用

Docker 就只需要很小的磁盘空间。即便我们同时运行这个镜像的很多实例，我们也只需要一个基础的 JAVA / TOMCAT 镜像。

**Docker** 可以节约时间 很多年前我在为一个连锁餐厅开发软件时，仅仅是为了描述如何搭建环境都需要写一个 12 页的 Word 文档。例如本地 Oracle 数据库，特定版本的 JAVA，以及其他七七八八的系统工具和共享库、软件包。整个搭建过程浪费掉了我们团队每个人几乎一天的时间，如果用金钱衡量的话，花掉了我们上万美金 的时间成本。虽然客户已经对这种事情习以为常，甚至认为这是引入新成员、让成 员适应环境、让自己的员工适应我们的软件所必须的成本，但是相比较起来，我们 宁愿把更多的时间花在为客户构建可以增进业务的功能上面。 如果当时有

Docker，那么构建环境就会像使用自动化搭建工具 Puppet / Chef / Salt / Ansible 一样简单，我们也可以把整个搭建时间周期从一天缩短为几分钟。但是和这些工具不同的地方在于，Docker 可以不仅仅可以搭建整个环境，还可以将整个环境保存成磁盘文件，然后复制到别的地方。需要从源码编译 Node.js 吗？Docker 做得到。

Docker 不仅仅可以构建一个 Node.js 环境，还可以将整个环境做成镜像，然后保存到任何地方。当然，由于 Docker 是一个容器，所以不用担心容器内执行的东西会对宿主机产生任何的影响。 现在新加入我们团队的人只需要运行 docker-compose

up 命令，便可以喝杯咖啡，然后开始工作了。

**Docker** 可以节省开销 当然，时间就是金钱。除了时间外，Docker 还可以节省在基础设施硬件上的开销。高德纳和麦肯锡的研究表明，数据中心的利用率在 6% -

12% 左右。不仅如此，如果采用虚拟机的话，你还需要被动地监控和设置每台虚拟机的 CPU 硬盘和内存的使用率，因为采用了静态分区(static partitioning)所以资源并不能完全被利用。。而容器可以解决这个问题：容器可以在实例之间进行内存和磁盘共享。你可以在同一台主机上运行多个服务、可以不用去限制容器所消耗的资源、可以去限制资源、可以在不需要的时候停止容器，也不用担心启动已经停止的程序时会带来过多的资源消耗。凌晨三点的时候只有很少的人会去访问你的网站， 同时你需要比较多的资源执行夜间的批处理任务，那么可以很简单的便实现资源的交换。 虚拟机所消耗的内存、硬盘、CPU 都是固定的，一般动态调整都需要重启

虚拟机。而用 Docker 的话，你可以进行资源限制，得益于 CGroup，可以很方便动态调整资源限制，让然也可以不进行资源限制。Docker 容器内的应用对宿主机而言只是两个隔离的应用程序，并不是两个虚拟机，所以宿主机也可以自行去分配 资源。

注：

部分翻译自：<https://docs.docker.com/engine/understanding-docker/>[部分参考：http://zhidao.baidu.com/link?](http://zhidao.baidu.com/link?url=4FOwNhnpVC3FP0hOxaC4vrl3fFG27lWRpDEaZ3KJBVL0E29C5O-ty4zqze1On52Uk4kcNrnPd3VEKpKvRs4pNEV-lgo78lmP1_FXffMerdG)

[url=4FOwNhnpVC3FP0hOxaC4vrl3fFG27lWRpDEaZ3KJBVL0E29C5O-](http://zhidao.baidu.com/link?url=4FOwNhnpVC3FP0hOxaC4vrl3fFG27lWRpDEaZ3KJBVL0E29C5O-ty4zqze1On52Uk4kcNrnPd3VEKpKvRs4pNEV-lgo78lmP1_FXffMerdG)

[ty4zqze1On52Uk4kcNrnPd3VEKpKvRs4pNEV-lgo78lmP1\_FXffMerdG](http://zhidao.baidu.com/link?url=4FOwNhnpVC3FP0hOxaC4vrl3fFG27lWRpDEaZ3KJBVL0E29C5O-ty4zqze1On52Uk4kcNrnPd3VEKpKvRs4pNEV-lgo78lmP1_FXffMerdG)

参考文档

Docker介绍：<http://www.lupaworld.com/article-243555-1.html>

Docker 介 绍 ：<http://www.docker.org.cn/book/docker/what-is-docker-16.html> Docker官方文档：<https://docs.docker.com/engine/understanding-docker/>

Docker中文文档：<http://git.oschina.net/widuu/chinese_docker>

Docker介绍：<https://segmentfault.com/a/1190000002609286>

* 1. **Docker**的安装

Docker的安装是比较简单的，笔者原本不想过多提及；但是看到有不少读者对

Docker的安装提出了疑问，故此进行一个安装的总结。

对于Linux用户可以借助其发行版的Linux包管理工具安装，对于Windows和MAC用 户相对麻烦一些，笔者下面以Windows7系统为例，讲述安装过程。笔者强烈建议大家使用**Linux**系统进入本章的学习，第一是比较符合目前Docker的市场趋势，第二Docker本身就是基于Linux的LXC技术。

**CentOS 7.0**下**Docker**的安装

* + 1. 查看内核版本(Docker需要64位版本，同时内核版本在3.10以上，如果版本低 于3.10，需要升级内核)：

uname -r

* + 1. 更新yum包：

yum update

* + 1. 添加yum仓库：

sudo tee /etc/yum.repos.d/docker.repo <<-'EOF' [dockerrepo]

name=Docker Repository baseurl=https://yum.dockerproject.org/repo/main/centos/7/ enabled=1

gpgcheck=1 gpgkey=https://yum.dockerproject.org/gpg EOF

* + 1. 安装Docker

yum install docker-engine

* + 1. 启动Docker

service docker start

* + 1. 使用Docker国内镜像（为Docker镜像下载提速，非必须）

curl -sSL https://get.daocloud.io/daotools/set\_mirror.sh | s h -s [http://fe8a7d6e.m.daocloud.io](http://fe8a7d6e.m.daocloud.io/)

参考：

官方文档：<https://docs.docker.com/engine/installation/linux/centos/>

**CentOS 6.5**下**Docker**的安装

Docker容器最早受到RHEL完善的支持是从最近的CentOS 7.0开始的，官方说明是只能运行于64位架构平台，内核版本为2.6.32-431及以上（即 >= CentOS 6.5，运行docker时实际提示3.10.0及以上）。 需要注意的是CentOS 6.5与7.0的安装是有一点点不同的，CentOS 6.x上Docker的安装包叫docker-io，并且来源于Fedora

epel库，这个仓库维护了大量的没有包含在发行版中的软件，所以先要安装EPEL，而CentOS 7.x的Docker直接包含在官方镜像源的Extras仓库（CentOS-

Base.repo下的[extras]节enable=1启用）。

下面就CentOS 6.5讲解Docker的安装过程，以下是软件版本：

|  |  |
| --- | --- |
| **Linux**版本 | **Docker**版本 |
| CentOS 6.5 X64（只能X64） | 1.7.1 |

升级内核

查看内核版本：

uname -r

结果： 2.6.32-431.el6.x86\_64 ，不满足上文的需求，故此需要升级内核。升级步骤：

1. 导入公钥数字证书

rpm --import https://[www.elrepo.org/RPM-GPG-KEY-elrepo.org](http://www.elrepo.org/RPM-GPG-KEY-elrepo.org)

1. 安装ELRepo

rpm -ivh <http://www.elrepo.org/elrepo-release-6-5.el6.elrepo>

.noarch.rpm

1. 安装kernel长期版本

表示long-term的意思，长期维护版本，也可以将kernel-lt改为kernel-ml，

安装主线版本

# lt

yum --enablerepo=elrepo-kernel install kernel-lt -y

1. 编辑grub.conf文件，修改Grub引导顺序，确认刚安装好的内核在哪个位置，然 后设置default值（从0开始），一般新安装的内核在第一个位置，所以设置

default=0。

vim /etc/grub.conf

# 以下是/etc/grub.conf的内容

default=0 # 修改该值即可

timeout=5 splashimage=(hd0,0)/grub/splash.xpm.gz hiddenmenu

title CentOS (3.10.103-1.el6.elrepo.x86\_64)

1. 重启并查看内核版本，将会发现内核已经更新。

安装**Docker**

1. 禁用selinux，因为selinux和LXC有冲突，故而需要禁用

SELINUX=disabled # 将SELINUX设为disabled，注意修改后最好重启下机

器。

disabled - No SELinux policy is loaded.

enforcing - SELinux security policy is enforced.

permissive - SELinux prints warnings instead of enforc

# #

ing.

#

# 以下是/etc/selinux/config的内容

vim /etc/selinux/config的内容

1. 安 装 Fedora EPEL

yum -y install <http://dl.fedoraproject.org/pub/epel/6/x86_64>

/epel-release-6-8.noarch.rpm

1. 安装Docker

yum install -y docker-io

1. 以守护模式运行Docker

docker -d

1. 如果不报错，那就是启动成功了，如果报以下异常：

docker: relocation error: docker: symbol dm\_task\_get\_info\_wi th\_deferred\_remove, version Base not defined in file libdevm apper.so.1.02 with link time reference

INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)

执行以下内容：

yum upgrade device-mapper-libs

1. 将Docker开机启动

chkconfig docker on

1. 重启机器

其他平台的安装

请参考：<https://docs.docker.com/engine/installation/>

参考文档

Windows：<https://docs.docker.com/engine/installation/windows/> MAC： <https://docs.docker.com/engine/installation/mac/> CentOS：<https://docs.docker.com/engine/installation/linux/centos/>

* 1. **Docker**的常用命令

### 准备工作

* + 1. 对于Window用户，请点击Kitematic左下方的DOCKER CLI按钮，在弹出的命令窗体内输入命令，不要在CMD中测试Docker命令。
    2. 下载镜像，以kitematic/hello-world-nginx为例：

docker pull kitematic/hello-world-nginx

常用命令测试一览表

|  |  |
| --- | --- |
| 命令 | 解释 |
| docker images | 列表本地所有镜像 |
| docker search 关键词 | 在Docker Hub中搜索镜像 |
| docker pull 镜像名称 | 下载Docker镜像 |
| docker rmi 镜像id | 删除Docker镜像。加参数-f表示强制删除。 |
| docker run 镜像名称 | 启动Docker镜像 |
| docker ps | 列表所有运行中的Docker容器。该命令参数比较多，-a： 列表所有容器；-f：过滤；-q 只列表容器的id。 |
| docker version | 查看Docker版本信息 |
| docker info | 查看Docker系统信息，例如：CPU、内存、容器个数等等 |
| docker kill 容器id | 杀死id对应容器 |
| docker start / stop / restart 容器id | 启动、停止、重启指定容器 |
| docker build -t 标签名称 目录 | 构建Docker镜像，-t 表示指定一个标签 |
| docker tag | 为镜像打标签 |

更多命令，请输入 --help 参数查询；如果想看docker命令可输入 docker --

help ；如果想查询 docker run 命令的用法，可输入 docker run --help 。

**docker run**

docker run 应该是我们最常用的命令了，这边讲解一下，便于大家入门。

|  |  |
| --- | --- |
| 参数 | 解释 |
| -d | 后台运行 |
| -P | 随机端口映射 |
| -p | 指定端口映射格式：  ip:hostPort:containerPort  ip::containerPort hostPort:containerPort containerPort |

测试：

1. 启动测试镜像 docker pull kitematic/hello-world-nginx

docker run -d -p 91:80 kitematic/hello-world-nginx

这边解释下docker run的两个参数：

# 后台运行

# 开放容器端口到宿主机端口

-d

-p 宿主机端口:容器端口

1. 访问：http://localhost:91 测试，这里的localhost指的是宿主机的主机名



说明

由于Docker的入门不是本文章的主要话题，同时入门也是非常简单的，所以不做太多赘述了，我们尽量把话题聚焦在微服务上。如果有疑问的童鞋可以给我提Issue， 也可以等待我的“Docker手把手”系列文章。

**TIPS**

* 1. 目前网上很多教程还是boot2docker（项目地

址：<https://github.com/boot2docker/boot2docker>），该项目已废弃了，笔者不建议使用。

* 1. 如果安装不成功，请按照提示解决一下，譬如开启VT技术等等。Docker的 提示是做得非常好的，如果还是安装不上，请给我提Issue，可以远程解决。
  2. 对于Winodws用户，建议点击Kitematic左下角的DOCKER CLI，在弹出来的控制台输入Docker命令，而不要直接在CMD中输入命令。

参考文档：

<http://my.oschina.net/denglz/blog/487332> <https://segmentfault.com/a/1190000000733628> <http://www.oschina.net/translate/nstalling-dockerio-on-centos-64-64-bit> <https://segmentfault.com/a/1190000000735011> <http://www.server110.com/docker/201411/11122.html> [http://dockone.io/article/152 http://www.open- open.com/lib/view/open1422492851548.html](http://www.open-open.com/lib/view/open1422492851548.html)

[Docker常用命令：http://www.infoq.com/cn/articles/docker-command-line- quest/](http://www.infoq.com/cn/articles/docker-command-line-quest/)

* 1. **Dockerfile**常用指令

指令的一般格式为 指令名称 参数 。

## FROM

支持三种格式：



FROM <image>



FROM <image>:<tag>



FROM <image>@<digest>

FROM指令必须指定且需要在Dockerfile其他指令的前面，指定的基础image可以是 官方远程仓库中的，也可以位于本地仓库。后续的指令都依赖于该指令指定的

image。当在同一个Dockerfile中建立多个镜像时，可以使用多个FROM指令。

**MAINTAINER**

格式为：

 MAINTAINER <name>

用于指定维护者的信息。

**RUN**

支持两种格式：

RUN <command>

或 RUN ["executable", "param1", "param2"] 。

RUN <command> 在shell终端中运行命令，在Linux中默认是 /bin/sh -c 在

Windows中是 使用

RUN ["executable", "param1", "param2"]

cmd /s /c

exec执行。指定其他终端可以通过该方式操作，例如： RUN ["/bin/bash", "-

c", "echo hello"] ，该方式必须使用["]而不能使用[']，因为该方式会被转换成一个JSON 数组。

## CMD

支持三种格式：

CMD ["executable","param1","param2"] (推荐使用)

CMD ["param1","param2"] (为ENTRYPOINT指令提供预设参数)

CMD command param1 param2 (在shell中执行)

CMD指令的主要目的是为执行容器提供默认值。每个Dockerfile只有一个CMD命 令，如果指定了多个CMD命令，那么只有一条会被执行，如果启动容器的时候指定 了运行的命令，则会覆盖掉CMD指定的命令。

**LABEL**

格式为：

 LABEL <key>=<value> <key>=<value> <key>=<value> ...

为镜像添加元数据。使用 "和 \ 转换命令行，示例：

LABEL "com.example.vendor"="ACME Incorporated" LABEL com.example.label-with-value="foo" LABEL version="1.0"

LABEL description="This text illustrates \ that label-values can span multiple lines."

**EXPOSE**

格式为：

 EXPOSE <port> [<port>...]

为Docker容器设置对外的端口号。在启动时，可以使用-p选项或者-P选项。 示例：

# 映射一个端口示例

EXPOSE port1

# 相应的运行容器使用的命令

docker run -p port1 image

# 也可以使用-P选项启动

docker run -P image

# 映射多个端口示例

EXPOSE port1 port2 port3

# 相应的运行容器使用的命令

docker run -p port1 -p port2 -p port3 image

# 还可以指定需要映射到宿主机器上的某个端口号

docker run -p host\_port1:port1 -p host\_port2:port2 -p host\_port3

:port3 image

**ENV**

格式为：



ENV <key> <value>



ENV <key>=<value> ...

指定环境变量，会被后续RUN指令使用，并在容器启动后，可以通过 docker inspect 查看这个环境变量，也可以通过 docker run --env <key>=<value> 来修改环境变量

示例：

ENV JAVA\_HOME /path/to/java # 设置环境变量JAVA\_HOME

**ADD**

格式为：



ADD <src>... <dest>



ADD ["<src>",... "<dest>"]

从src目录复制文件到容器的dest。其中src可以是Dockerfile所在目录的相对路径， 也可以是一个URL，还可以是一个压缩包

注意：

* + 1. src必须在构建的上下文内，不能使用例如： ADD ../somethine

/something ，因为 docker build 命令首先会将上下文路径和其子目录发送到docker daemon

* + 1. 如果src是一个URL，同时dest不以斜杠结尾，dest将会被视为文件，src对应 内容文件将会被下载到dest
    2. 如果src是一个URL，同时dest以斜杠结尾，dest将被视为目录，src对应内容 将会被下载到dest目录
    3. 如果src是一个目录，那么整个目录其下的内容将会被拷贝，包括文件系统元数 据
    4. 如果文件是可识别的压缩包格式，则docker会自动解压

**COPY**

格式为：

COPY <src>... <dest>

 COPY ["<src>",... "<dest>"] （shell中执行）

复制本地端的src到容器的dest。和ADD指令类似，COPY不支持URL和压缩包。

**ENTRYPOINT**

格式为：



ENTRYPOINT ["executable", "param1", "param2"]



ENTRYPOINT command param1 param2

指定Docker容器启动时执行的命令，可以多次设置，但是只有最后一个有效。

**VOLUME**

格式为：

 VOLUME ["/data"]

使容器中的一个目录具有持久化存储数据的功能，该目录可以被容器本身使用，也 可以共享给其他容器。当容器中的应用有持久化数据的需求时可以在Dockerfile中使用该指令。

**USER**

格式为：

 USER 用户名

设置启动容器的用户，默认是root用户。

**WORKDIR**

格式为：

 WORKDIR /path/to/workdir

切换目录指令，类似于cd命令，对RUN、CMD、ENTRYPOINT生效。

**ARG**

格式为：

 ARG <name>[=<default value>]

ARG指令定义一个变量。

**ONBUILD**

格式为：

 ONBUILD [INSTRUCTION]

指定当建立的镜像作为其他镜像的基础时，所执行的命令。

其他

STOPSINGAL HEALTHCHECK SHELL 由于并不是很常用，所以不做讲解了。有兴趣的可以前往<https://docs.docker.com/engine/reference/builder/>扩展阅读。

参考文档：

[Dockerfile文档：https://docs.docker.com/engine/reference/builder/#dockerfile- reference](https://docs.docker.com/engine/reference/builder/#dockerfile-reference)

[Dockerfile最佳实践：https://docs.docker.com/engine/userguide/eng- image/dockerfile\_best-practices/#build-cache](https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#build-cache)

Docker书

[籍：http://udn.yyuap.com/doc/docker\_practice/advanced\_network/port\_mapping.h](http://udn.yyuap.com/doc/docker_practice/advanced_network/port_mapping.html)

[tml](http://udn.yyuap.com/doc/docker_practice/advanced_network/port_mapping.html)

Docker书

[籍：https://philipzheng.gitbooks.io/docker\_practice/content/dockerfile/instructions.](https://philipzheng.gitbooks.io/docker_practice/content/dockerfile/instructions.html)

[html](https://philipzheng.gitbooks.io/docker_practice/content/dockerfile/instructions.html)

Dockerfile讲解：<http://blog.csdn.net/qinyushuang/article/details/43342553>

Dockerfile讲解：<http://blog.csdn.net/wsscy2004/article/details/25878223> Dockerfile网络：<http://my.oschina.net/ghm7753/blog/522809>

COPY 和 ADD 的区

别：[http://blog.163.com/digoal@126/blog/static/163877040201410341236664/](http://blog.163.com/digoal%40126/blog/static/163877040201410341236664/)

CMD与ENTRYPOINT的区别：<http://cloud.51cto.com/art/201411/457338.htm>

* 1. **Docker**私有仓库的搭建与使用

和Maven一样，Docker不仅提供了一个中央仓库，同时也允许我们搭建私有仓库。 如果读者对Maven有所了解，将会很容易理解私有仓库的优势：

节省带宽，镜像无需从中央仓库下载，只需从私有仓库中下载即可 对于私有仓库中已有的镜像，提升了下载速度

便于内部镜像的统一管理

下面我们来讲解一下如何搭建、使用私有仓库

准备工作

准备两台安装有Docker的CentOS7的机器，主机规划如下（仅供参考）：

|  |  |  |
| --- | --- | --- |
| 主机 | **IP** | 角色 |
| node0 | 192.168.11.143 | Docker开发机 |
| node1 | 192.168.11.144 | Docker私有仓库 |

安装、使用私有仓库

网上有很多 docker-registry 的教程，但是 docker-registry 已经过时，并且已经2年不维护了。详见<https://github.com/docker/docker-registry>，故而本文不做探讨，对 docker-registry 有兴趣的童鞋可以查阅本节的参考文档。

本节讲解registry V2，registry V2需要Docker版本高于1.6.0。registry V2要求使用

https访问，那么我们先做一些准备，为了方便，这边模拟以域 名 reg.itmuch.com 进行讲解。

使用域名搭建**https**的私有仓库

首先修改两台机器的hosts，配置 192.168.11.144 到 reg.itmuch.com 的映射

echo '192.168.11.144 reg.itmuch.com'>> /etc/hosts

既然使用https，那么我们需要生成证书，本文讲解的是使用openssl自签名证书，当然也可以使用诸如 Let’s Encrypt 等工具生成证书，首先在node1机器上生成key：

mkdir -p ~/certs cd ~/certs

openssl genrsa -out reg.itmuch.com.key 2048

再生成密钥文件：

openssl req -newkey rsa:4096 -nodes -sha256 -keyout reg.itmuch.c om.key -x509 -days 365 -out reg.itmuch.com.crt

会有一些信息需要填写：

Country Name (2 letter code) [XX]:CN

# 你的国家名称

State or Province Name (full name) []:JS

# 省 份

Locality Name (eg, city) [Default City]:NJ

# 所在城市

Organization Name (eg, company) [Default Company Ltd]:ITMUCH

# 组织名称

Organizational Unit Name (eg, section) []:ITMUCH

# 组织单元名称

Common Name (eg, your name or your server's hostname) []:reg.itm uch.com # 域 名

Email Address [][:eacdy0000@126.com](mailto:eacdy0000@126.com)

# 邮 箱

这样自签名证书就制作完成了。

由于是自签名证书，默认是不受Docker信任的，故而需要将证书添加到Docker 的根证书中，Docker在CentOS 7中，证书存放路径

是 /etc/docker/certs.d/域名 ：

node1 端 ：

mkdir -p /etc/docker/certs.d/reg.itmuch.com

cp ~/certs/reg.itmuch.crt /etc/docker/certs.d/reg.itmuch.com/

node0 端：将生成的证书下载到根证书路径

mkdir -p /etc/docker/certs.d/reg.itmuch.com

scp root@192.168.11.144:/root/certs/reg.itmuch.com.crt /etc/dock er/certs.d/reg.itmuch.com/

重 新 启 动 node0 和 node1 的 Docker

service docker restart

在 node1 上启动私有仓库

首先切换到家目录中，这一步不能少，原因是下面的-v 挂载了证书，如果不切换， 将会引用不到证书文件。

cd ~

启动Docker私有仓库（注意：如果直接粘贴运行，请删除掉注释）：

docker run -d -p 443:5000 --restart=always --name registry \

-v `pwd`/certs:/certs \ # 将“当

前目录/certs”挂载到容器的“/certs”

-v /opt/docker-image:/opt/docker-image \

-e STORAGE\_PATH=/opt/docker-image \

# 指定容器内存储镜像的路径

-e REGISTRY\_HTTP\_TLS\_CERTIFICATE=/certs/reg.itmuch.com.crt \

# 指定证书文件

-e REGISTRY\_HTTP\_TLS\_KEY=/certs/reg.itmuch.com.key \

# 指定key文件

registry:2

其中，之所以挂载/opt/docker-image目录，是为了防止私有仓库容器被删除，私有 仓库中的镜像也会丢失。

在 node0 上测试，将镜像push到私服

docker pull kitematic/hello-world-nginx

docker tag kitematic/hello-world-nginx reg.itmuch.com/kitematic/ hello-world-nginx # 为本地镜像打标签

docker push reg.itmuch.com/kitematic/hello-world-nginx

# 将镜像push到私服

会发现如下内容：

The push refers to a repository [reg.itmuch.com/kitematic/hello- world-nginx]

5f70bf18a086: Pushed b51acdd3ef48: Pushed 3f47ff454588: Pushed

....

latest: digest: sha256:d3e1883b703c39556f2f09da14cc3b820f69a4343 6655c882c0c0ded0dda6a4b size: 3226

说明已经push成功。

从私服中下载镜像：

docker pull reg.itmuch.com/kitematic/hello-world-nginx

配置登录认证

在很多场景下，我们需要用户登录后才能访问私有仓库，那么我们可以如下操作： 建立在上文生成证书，同时重启过Docker服务的前提下，我们讲解一下如何配置：

为防止端口冲突，我们首先删除或停止之前启动好的私有仓库：

docker kill registry

在node1机器上安装 httpd-tools ：

yum install httpd-tools

在node机器上创建密码文件，并添加一个用户 testuser ，密码是 testpassword ：

cd ~ mkdir auth

htpasswd -Bbn testuser testpassword > auth/htpasswd

在node1机器上切换到 ~ 目录，并启动私有仓库（注意：如果直接粘贴运 行，请删除掉注释）：

docker run -d -p 443:5000 --restart=always --name registry2 \

-v /opt/docker-image:/var/lib/registry \

# 挂载容器内存储镜像路径到宿主机

-v `pwd`/certs:/certs \

-e REGISTRY\_HTTP\_TLS\_CERTIFICATE=/certs/reg.itmuch.com.crt \

-e REGISTRY\_HTTP\_TLS\_KEY=/certs/reg.itmuch.com.key \

-v `pwd`/auth:/auth \

-e "REGISTRY\_AUTH=htpasswd" \

-e "REGISTRY\_AUTH\_HTPASSWD\_REALM=Registry Realm" \

-e REGISTRY\_AUTH\_HTPASSWD\_PATH=/auth/htpasswd \ registry:2

测试：

docker push reg.itmuch.com/kitematic/hello-world-nginx

提示：

461f75075df2: Image push failed no basic auth credentials

说明需要认证。

我们登陆一下，执行：

docker login reg.imuch.com

再次执行

docker push reg.itmuch.com/kitematic/hello-world-nginx

就可以正常push镜像到私有仓库了。

注意：如果想要从私有仓库上下载镜像，同样需要登录。

参考文档：

官方文档：[https://docs.docker.com/registry/deploying/#/running-a-domain-registry](https://docs.docker.com/registry/deploying/%23/running-a-domain-registry)

Docker Registry V2 htpasswd认证方式搭

建：<http://www.tuicool.com/articles/vMZZveM>

Docker Registry V2搭建：<http://www.tuicool.com/articles/6jEJZj> Docker Registry V2搭建：<http://tomhat.iteye.com/blog/2304098>

Docker Registry V1搭建：<http://blog.csdn.net/wsscy2004/article/details/26279569>

非认证的Docker Registry V1搭

建：<http://blog.csdn.net/wangtaoking1/article/details/44180901>

带认证的Docker Registry V1搭

建：<http://snoopyxdy.blog.163.com/blog/static/601174402015823741997/>

Docker专题汇总：<http://www.zimug.com/360.html>

Docker疑难解答：<https://segmentfault.com/q/1010000000938076>

* 1. 使用**Dockerfile**构建**Docker**镜像

下面我们以microservice-discovery-eureka项目为例，我们首先执行

mvn clean package # 使用Maven打包项目

将项目构建成jar包： microservice-discovery-eureka-0.0.1-SNAPSHOT.jar

，那么如果我们想要启动项目则只需要在 microservice-discovery-eureka- 0.0.1-SNAPSHOT.jar 所在的目录（即项目的target目录）执行：

java -jar microservice-discovery-eureka-0.0.1-SNAPSHOT.jar

那么如果我们现在想要将项目在Docker容器中运行，需要怎么做呢？

使用**Dockerfile**构建**Docker**镜像

在 microservice-discovery-eureka-0.0.1-SNAPSHOT.jar 所在目录（默认即：项目构建后的target目录，当然也可以将jar文件拷贝到其他任意路

径），创建文件，命名为Dockerfile

# 基于哪个镜像

FROM java:8

# 将本地文件夹挂载到当前容器

VOLUME /tmp

# 拷贝文件到容器，也可以直接写成ADD microservice-discovery-eureka-0.0

.1-SNAPSHOT.jar /app.jar

ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar RUN bash -c 'touch /app.jar'

# 开放8761端口

EXPOSE 8761

# 配置容器启动后执行的命令

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-j ar","/app.jar"]

构建docker镜像，执行：

# 格式：docker build -t 标签

docker build -t eacdy/test1 .

名称 Dockerfile的相对位置

构建成功： Successfully built a7cc6f4de088 。

启动镜像

docker run -p 8761:8761 eacdy/test1

访问 http://Docker宿主机IP:8761 ，我们会发现Eureka能够正常被访问。

参考文档

基于Dockerfile搭建JAVA Tomcat运行环

境：<http://www.blogjava.net/yongboy/archive/2013/12/16/407643.html>

Docker实现增量发布之前期准

备：<http://blog.csdn.net/zssureqh/article/details/52009043>

Dockerfile详解：<http://blog.csdn.net/wsscy2004/article/details/25878223> Dockerfile RUN/CMD/ENTRYPOINT命令详

[解：http://blog.163.com/digoal@126/blog/static/163877040201410411715832](http://blog.163.com/digoal%40126/blog/static/163877040201410411715832/)

[/](http://blog.163.com/digoal%40126/blog/static/163877040201410411715832/)

如何使用Dockerfile构建镜

像：<http://blog.csdn.net/qinyushuang/article/details/43342553>

* 1. 使用**Maven**插件构建**Docker**镜像

### 工具

工欲善其事，必先利其器。笔者经过调研，有以下几款Docker的Maven插件进入笔 者视野：

|  |  |
| --- | --- |
| 插件名称 | 官方地址 |
| docker-maven-plugin | <https://github.com/spotify/docker-maven-plugin> |
| docker-maven-plugin | <https://github.com/fabric8io/docker-maven-plugin> |
| docker-maven-plugin | <https://github.com/bibryam/docker-maven-plugin> |

笔者从Stars、文档易用性以及更新频率三个纬度考虑，选用了第一款。

使用插件构建**Docker**镜像

简单使用

我们以之前的项目：microservice-discovery-eureka为例：

在pom.xml中添加下面这段

|  |  |
| --- | --- |
| <build>  <plugins>  <!-- docker的maven插件，官网：https://github.com/spoti  fy/docker-maven-plugin -->  <plugin>  <groupId>com.spotify</groupId>  <artifactId>docker-maven-plugin</artifactId>  <version>0.4.12</version>  <configuration>  <!-- 注意imageName一定要是符合正则[a-z0-9-\_.]的  ，否则构建不会成功 -->  <!-- 详见：https://github.com/spotify/docker-  maven-plugin Invalid repository name ... only [a-z0-9-\_.] are allowed-->  <imageName>microservice-discovery-eureka</im  ageName>  <baseImage>java</baseImage>  <entryPoint>["java", "-jar", "/${project.bui ld.finalName}.jar"]</entryPoint>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}  </directory>  <include>${project.build.finalName}.  jar</include>  </resource>  </resources>  </configuration>  </plugin>  </plugins>  </build> | |
|  |  |

执行命令：

mvn clean package docker:build

我们会发现控制台有类似如下内容：

[INFO] Building image microservice-discovery-eureka Step 1 : FROM java

Pulling from library/java

Digest: sha256:581a4afcbbedd8fdf194d597cb5106c1f91463024fb3a49a2 d9f025165eb675f

Status: Downloaded newer image for java:latest

---> ea40c858f006

Step 2 : ADD /microservice-discovery-eureka-0.0.1-SNAPSHOT.jar /

/

---> d1c174083bca

Removing intermediate container 91913d847c20

Step 3 : ENTRYPOINT java -jar /microservice-discovery-eureka-0.0

.1-SNAPSHOT.jar

---> Running in 0f2aeccdfd46

---> d57b027ca65a

Removing intermediate container 0f2aeccdfd46 Successfully built d57b027ca65a

[INFO] Built microservice-discovery-eureka

[INFO] ---------------------------------------------------------

--------------- [INFO] BUILD SUCCESS

[INFO] ---------------------------------------------------------

---------------

[INFO] Total time: 01:38 min

[INFO] Finished at: 2016-09-18T01:05:05-07:00

[INFO] Final Memory: 40M/198M

恭喜，构建成功了。

我们执行 docker images 会发现该镜像已经被构建成功：

microservice-discovery-eureka latest d57b027ca65a

About a minute ago 681.5 MB

IMAGE ID

TAG

SIZE

REPOSITORY

CREATED

启动镜像

docker run -p 8761:8761 microservice-discovery-eureka

我们会发现该Docker镜像会很快地启动。

访问测试

访问http://Docker宿主机IP:8761 ，能够正常看到Eureka界面。

使用**Dockerfile**进行构建

上文讲述的方式是最简单的方式，很多时候，我们还是要借助Dockerfile进行构建的，首先我们在/microservice-discovery-eureka/src/main/docker目录下，建立文件

Dockerfile

FROM java:8 VOLUME /tmp

ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar RUN bash -c 'touch /app.jar'

EXPOSE 9000

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-j ar","/app.jar"]

修改pom.xml

|  |  |
| --- | --- |
| <build>  <plugins>  <!-- docker的maven插件，官网：https://github.com/spoti  fy/docker-maven-plugin -->  <plugin>  <groupId>com.spotify</groupId>  <artifactId>docker-maven-plugin</artifactId>  <version>0.4.12</version>  <configuration>  <!-- 注意imageName一定要是符合正则[a-z0-9-\_.]的  ，否则构建不会成功 -->  <!-- 详见：https://github.com/spotify/docker-  maven-plugin Invalid repository name ... only [a-z0-9-\_.] are allowed-->  <imageName>microservice-discovery-eureka-doc  kerfile</imageName>  <!-- 指定Dockerfile所在的路径 -->  <dockerDirectory>${project.basedir}/src/main  /docker</dockerDirectory>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}  </directory>  <include>${project.build.finalName}.  jar</include>  </resource>  </resources>  </configuration>  </plugin>  </plugins>  </build> | |
|  |  |

其他步骤一样。这样即可使用Dockerfile进行构建Docker镜像啦。

将**Docker**镜像**push**到**DockerHub**上

首先修改Maven的全局配置文件settings.xml，添加以下段落

<servers>

<server>

<id>docker-hub</id>

<username>你的DockerHub用户名</username>

<password>你的DockerHub密码</password>

<configuration>

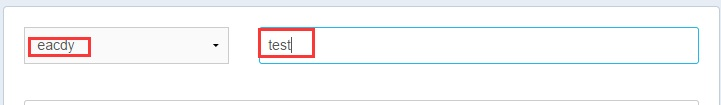
<email>你的DockerHub邮箱</email>

</configuration>

</server>

</servers>

在DockerHub上创建repo,例如：test，如下图



项目pom.xml修改为如下：注意imageName的路径要和repo的路径一致

<build>

<plugins>

<!-- docker的maven插件，官网：https://github.com/spoti

fy/docker-maven-plugin -->

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.12</version>

<configuration>

<!-- 注意imageName一定要是符合正则[a-z0-9-\_.]的

，否则构建不会成功 -->

<!-- 详见：https://github.com/spotify/docker-

maven-plugin Invalid repository

name ... only [a-z0-9-\_.] are allowed -->

<!-- 如果要将docker镜像push到DockerHub上去的话， 这边的路径要和repo路径一致 -->

<imageName>eacdy/test</imageName>

<!-- 指定Dockerfile所在的路径 -->

<dockerDirectory>${project.basedir}/src/main

|  |  |
| --- | --- |
| /docker</dockerDirectory>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}  </directory>  <include>${project.build.finalName}.  jar</include>  </resource>  </resources>  <!-- 以下两行是为了docker push到DockerHub使用的  。 -->  <serverId>docker-hub</serverId>  <registryUrl>https://index.docker.io/v1/</re  gistryUrl>  </configuration>  </plugin>  </plugins>  </build> | |
|  |  |

执行命令：

mvn clean package docker:build -DpushImage

搞定，等构建成功后，我们会发现Docker镜像已经被push到DockerHub上了。

将镜像**push**到私有仓库

在很多场景下，我们需要将镜像push到私有仓库中去，这边为了讲解的全面性，私 有仓库采用的是配置登录认证的私有仓库。

和push镜像到DockerHub中一样，我们首先需要修改Maven的全局配置文件

settings.xml，添加以下段落

<servers>

<server>

<id>docker-registry</id>

<username>你的DockerHub用户名</username>

<password>你的DockerHub密码</password>

<configuration>

<email>你的DockerHub邮箱</email>

</configuration>

</server>

</servers>

将项目的pom.xml改成如下，

<plugin>

<groupId>com.spotify</groupId>

<artifactId>docker-maven-plugin</artifactId>

<version>0.4.12</version>

<configuration>

<!-- 路径为：私有仓库地址/你想要的镜像路径 -->

<imageName>reg.itmuch.com/test-pull-registry</imageName>

<dockerDirectory>${project.basedir}/src/main/docker</dockerD irectory>

<resources>

<resource>

<targetPath>/</targetPath>

<directory>${project.build.directory}</directory>

<include>${project.build.finalName}.jar</include>

</resource>

</resources>

<!-- 与maven配置文件settings.xml一致 -->

<serverId>docker-registry</serverId>

</configuration>

</plugin>

执行：

mvn clean package docker:build -DpushImage

稍等片刻，将会push成功。

如果想要从私服上下载该镜像，执行：

docker login reg.itmuch.com # 然后输入账号和密码

docker pull reg.itmuch.com/test-pull-registry

将插件绑定在某个**phase**执行

在很多场景下，我们有这样的需求，例如执行 mvn clean package 时，自动地为我们构建docker镜像，可以吗？答案是肯定的。我们只需要将插件的 goal 绑定在某个phase即可。

所谓的phase和goal，可以这样理解：maven命令格式是： mvn phase:goal ， 例 如 mvn package docker:build 那 么 ， package 和 docker 都是phase， build 则 是 goal 。

下面是示例：

|  |  |
| --- | --- |
| <build>  <plugins>  <plugin>  <groupId>com.spotify</groupId>  <artifactId>docker-maven-plugin</artifactId>  <executions>  <execution>  <id>build-image</id>  <phase>package</phase>  <goals>  <goal>build</goal>  </goals>  </execution>  </executions>  <configuration>  <imageName>${docker.image.prefix}/${project.artifactId}  </imageName>  <baseImage>java</baseImage>  <entryPoint>["java", "-jar", "/${project.build.finalNa me}.jar"]</entryPoint>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}</directory>  <include>${project.build.finalName}.jar</include>  </resource>  </resources>  </configuration>  </plugin>  </plugins>  </build> | |
|  |  |

如上，我们只需要添加：

<executions>

<execution>

<id>build-image</id>

<phase>package</phase>

<goals>

<goal>build</goal>

</goals>

</execution>

</executions>

即可。本例指的是讲docker的build目标，绑定在package这个phase上。也就是 说，用户只需要执行 mvn package ，就自动执行了 mvn docker:build 。

### 常见异常

连接不上**2375**（一般在**Win7**上出现）

Connect to localhost:2375 [localhost/127.0.0.1, localhost/0:0:0: 0:0:0:0:1] failed: Connection refused: connect -> [Help 1]

解决步骤：

输 入 docker-machine env

$Env:DOCKER\_TLS\_VERIFY = "1"

$Env:DOCKER\_HOST = "tcp://192.168.99.100:2376"

$Env:DOCKER\_CERT\_PATH = "C:\Users\Administrator\.docker\machine\ machines\default

为插件添加配置

<!-- 解决Connect to localhost:2375的问题的其中一种方式，注意要跟docker

-machine env相一致 -->

<dockerHost>https://192.168.99.100:2376</dockerHost>

<dockerCertPath>C:\Users\Administrator\.docker\machine\machi nes\default</dockerCertPath>

修改后插件配置变为：

|  |  |
| --- | --- |
| <plugin>  <groupId>com.spotify</groupId>  <artifactId>docker-maven-plugin</artifactId>  <version>0.4.12</version>  <configuration>  <imageName>eacdy/test</imageName>  <dockerDirectory>${project.basedir}/src/main/docker</doc kerDirectory>  <!-- 解决Connect to localhost:2375的问题的其中一种方式，注意要跟docker-machine env相一致 -->  <dockerHost>https://192.168.99.100:2376</dockerHost>  <dockerCertPath>C:\Users\Administrator\.docker\machine\m achines\default</dockerCertPath>  <resources>  <resource>  <targetPath>/</targetPath>  <directory>${project.build.directory}</directory>  <include>${project.build.finalName}.jar</include>  </resource>  </resources>  <!-- 以下两行是为了docker push到DockerHub使用的。 -->  <serverId>docker-hub</serverId>  <registryUrl>https://index.docker.io/v1/</registryUrl>  </configuration>  </plugin> | |
|  |  |

参考：<https://github.com/spotify/docker-maven-plugin/issues/116>

**TIPS**

* + 1. imageName必须符合正则[a-z0-9-\_.]，否则将会构建失败
    2. 插件默认使用localhost:2375去连接Docker，如果你的Docker端口不是2375， 需要配置环境变量 DOCKER\_HOST=tcp://<host>:2375

### 代码地址（任选其一）

[https://git.oschina.net/itmuch/spring-cloud- study/tree/master/docker/microservice-discovery-eureka](https://git.oschina.net/itmuch/spring-cloud-study/tree/master/docker/microservice-discovery-eureka) [https://github.com/eacdy/spring-cloud-study/tree/master/docker/microservice- discovery-eureka](https://github.com/eacdy/spring-cloud-study/tree/master/docker/microservice-discovery-eureka)

参考文档

[http://developer.51cto.com/art/201404/434879.htm https://linux.cn/article- 6131-rss.html](https://linux.cn/article-6131-rss.html)

* 1. **Docker Compose**

前文提到的Dockerfile 可以让用户管理一个单独的容器，那么如果我要管理多个容器呢，例如：我需要管理一个Web应用的同时还要加上其后端的数据库服务容器呢？Compose就是这样的一个工具。让我们看下官网对Compose的定义：

Compose 是一个用于定义和运行多容器的Docker应用的工具。使用Compose，你可以在一个配置文件（yaml格式）中配置你应用的服务，然后使用一个命令，即可 创建并启动配置中引用的所有服务。下面我们进入Compose的实战吧。

我们使用最新的Docker Compose 1.8.0进行讲解。

* + 1. **Docker Compose**的安装

安装**Compose**

Compose的安装有多种方式，例如通过shell安装、通过pip安装、以及将compose 作为容器安装等等。本文讲解通过shell安装的方式。其他安装方式如有兴趣，可以 查看Docker的官方文档：<https://docs.docker.com/compose/install/>

下载 docker-compose ，并放到 /usr/local/bin/

curl -L https://github.com/docker/compose/releases/download/1.8. 0/docker-compose-`uname -s`-`uname -m` > /usr/local/bin/docker-c ompose

为Docker Compose脚本添加执行权限

chmod +x /usr/local/bin/docker-compose

安装完成，测试：

docker-compose --version

结果显示：

docker-compose version 1.8.0, build f3628c7

说明Compose已经成功安装完成了。

安装**Compose**命令补全工具

按照上文讲解，我们已经成功地安装完Docker Compose。但是，我们输

入 docker-compose 命令，按下TAB键，发现此时Compose并没有给我们该命令的提示，那么如何让命令给我们提示呢？我们需要安装Compose命令补全工具。

Compose命令补全在Bash和Zsh下的安装方式不同，由于笔者是使用CentOS 7进行讲解的，而CentOS 7默认使用Bash，故而本文只讲解命令补全在Bash下的安装，其他Shell以及其他系统上的安装，请查看Docker的官方文

档：<https://docs.docker.com/compose/completion/>

curl -L https://raw.githubusercontent.com/docker/compose/$(docke r-compose version --short)/contrib/completion/bash/docker-compos e > /etc/bash\_completion.d/docker-compose

这样，在重新登录后，输入 docker-compose 命令后，按下TAB键盘，效果如下：

r

pull

port

p

ps

pause

kill up

logs

version

build config down exec ush rm scale stop bundle create events help

estart run start unpause

[root@localhost ~]# docker-compose

发现已经可以自动提示了。

* + 1. **Docker Compose**入门示例

Compose的使用非常简单，只需要编写一个 docker-compose.yml ，然后使用 docker-compose 命令操作即可。 docker-compose.yml 描述了容器的配

置，而 docker-compose 命令描述了对容器的操作。我们首先通过一个示例快速入门：

还记得前文，我们使用Dockerfile为项目 microservice-discovery-eureka 构建

Docker镜像吗？我们还以此项目为例，在node0（192.168.11.143）这台机器上测 试。

我们在 microservice-discovery-eureka-0.0.1-SNAPSHOT.jar 所在目录的上一级目录，创建 docker-compose.yml 文件。 目录树结构：

├── docker-compose.yml

└── eureka

├── Dockerfile

└── microservice-discovery-eureka-0.0.1-SNAPSHOT.jar

然后在 docker-compose.yml 中添加内容如下：

eureka:

build: ./eureka ports:

- "8761:8761"

expose:

- 8761

在 docker-compose.yml 所在路径执行：

docker-compose up

发现打印日志：

] s.b.c.e.t.TomcatEmbeddedServletContainer : Tomcat started on p ort(s): 8761 (http)

eureka\_1 | 2016-09-23 02:23:46.164 INFO 1 --- [ main

] c.n.e.EurekaDiscoveryClientConfiguration : Updating port to 87 61

eureka\_1 | 2016-09-23 02:23:46.167 INFO 1 --- [ main

] c.itmuch.cloud.study.EurekaApplication : Started EurekaAppli cation in 8.791 seconds (JVM running for 9.939)

eureka\_1 | 2016-09-23 02:24:46.016 INFO 1 --- [a-EvictionTimer

] c.n.e.registry.AbstractInstanceRegistry : Running the evict t ask with compensationTime 0ms

main

eureka\_1 | 2016-09-23 02:23:46.163 INFO 1 --- [

访问： http://宿主机IP:8761/ ，本文

为： http://192.168.11.143:8761/ ，发现可以正常启动。

# docker-compose.yml常用命令

###### image

指定镜像名称或者镜像id，如果该镜像在本地不存在，Compose会尝试pull下来。 示例：

image: java

**build**

指定Dockerfile文件的路径。可以是一个路径，例如：

build: ./dir

也可以是一个对象，用以指定Dockerfile和参数，例如：

build:

context: ./dir

dockerfile: Dockerfile-alternate args:

buildno: 1

**command**

覆盖容器启动后默认执行的命令。示例：

command: bundle exec thin -p 3000

也可以是一个list，类似于Dockerfile总的CMD指令，格式如下：

command: [bundle, exec, thin, -p, 3000]

**links**

链接到其他服务中的容器。可以指定服务名称和链接的别名使用 SERVICE:ALIAS

的形式，或者只指定服务名称，示例：

web:

links:

* db
* db:database
* redis

**external\_links**

表示链接到docker-compose.yml外部的容器，甚至并非Compose管理的容器，特 别是对于那些提供共享容器或共同服务。格式跟links类似，示例：

external\_links:

* redis\_1
* project\_db\_1:mysql
* project\_db\_1:postgresql

**ports**

暴露端口信息。使用宿主端口:容器端口的格式，或者仅仅指定容器的端口（此时宿 主机将会随机指定端口），类似于 docker run -p ，示例：

ports:

- "3000"

- "3000-3005"

- "8000:8000"

- "9090-9091:8080-8081"

- "49100:22"

- "127.0.0.1:8001:8001"

- "127.0.0.1:5000-5010:5000-5010"

**expose**

暴露端口，只将端口暴露给连接的服务，而不暴露给宿主机，示例：

expose:

- "3000"

- "8000"

**volumes**

卷挂载路径设置。可以设置宿主机路径 （ HOST:CONTAINER ） 或加上访问模式

（ HOST:CONTAINER:ro ）。示例：

volumes:

# Just specify a path and let the Engine create a volume

* /var/lib/mysql

# Specify an absolute path mapping

* /opt/data:/var/lib/mysql

# Path on the host, relative to the Compose file

* ./cache:/tmp/cache

# User-relative path

* ~/configs:/etc/configs/:ro

# Named volume

* datavolume:/var/lib/mysql

###### volumes\_from

从另一个服务或者容器挂载卷。可以指定只读或者可读写，如果访问模式没有指 定，则默认是可读写。示例：

volumes\_from:

* service\_name
* service\_name:ro
* container:container\_name
* container:container\_name:rw

**environment**

设置环境变量。可以使用数组或者字典两种方式。只有一个key的环境变量可以在运行Compose的机器上找到对应的值，这有助于加密的或者特殊主机的值。示例：

environment:

RACK\_ENV: development SHOW: 'true' SESSION\_SECRET:

environment:

* RACK\_ENV=development
* SHOW=true
* SESSION\_SECRET

**env\_file**

从文件中获取环境变量，可以为单独的文件路径或列表。如果通过 docker- compose -f FILE 指定了模板文件，则 env\_file 中路径会基于模板文件路径。如果有变量名称与 environment 指令冲突，则以 envirment 为准。示例：

env\_file: .env

env\_file:

* ./common.env
* ./apps/web.env
* /opt/secrets.env

###### extends

继承另一个服务，基于已有的服务进行扩展。

**net**

设置网络模式。示例：

net: "bridge" net: "host" net: "none"

net: "container:[service name or container name/id]"

**dns**

配置dns服务器。可以是一个值，也可以是一个列表。示例：

dns: 8.8.8.8

dns:

- 8.8.8.8

- 9.9.9.9

**dns\_search**

配置DNS的搜索域，可以是一个值，也可以是一个列表，示例：

dns\_search: example.com dns\_search:

* dc1.example.com
* dc2.example.com

其他

docker-compose.yml 还有很多其他命令，本文仅挑选常用命令进行讲解，其他不不作赘述。如果感兴趣的，可以参考docker-compose.yml文件官方文

档：<https://docs.docker.com/compose/compose-file/>

参考文档

Docker Compose安装及使用： <http://www.tuicool.com/articles/AnIVJn>

Docker Compose使用全解：<http://blog.csdn.net/zhiaini06/article/details/45287663> Docker Compose命令详

解：<http://blog.csdn.net/wanghailong041/article/details/52162293>

* + 1. **docker-compose**常用命令

TODO

参考：

Docker官方文档：<https://docs.docker.com/compose/overview/>

[Dokcer教程：http://wiki.jikexueyuan.com/project/docker-technology-and- combat/install.html](http://wiki.jikexueyuan.com/project/docker-technology-and-combat/install.html)