

图 10-4 trace 列表

6. 单击该 trace，可看到如图 10-5 所示界面，该图详细展示了请求的细节。

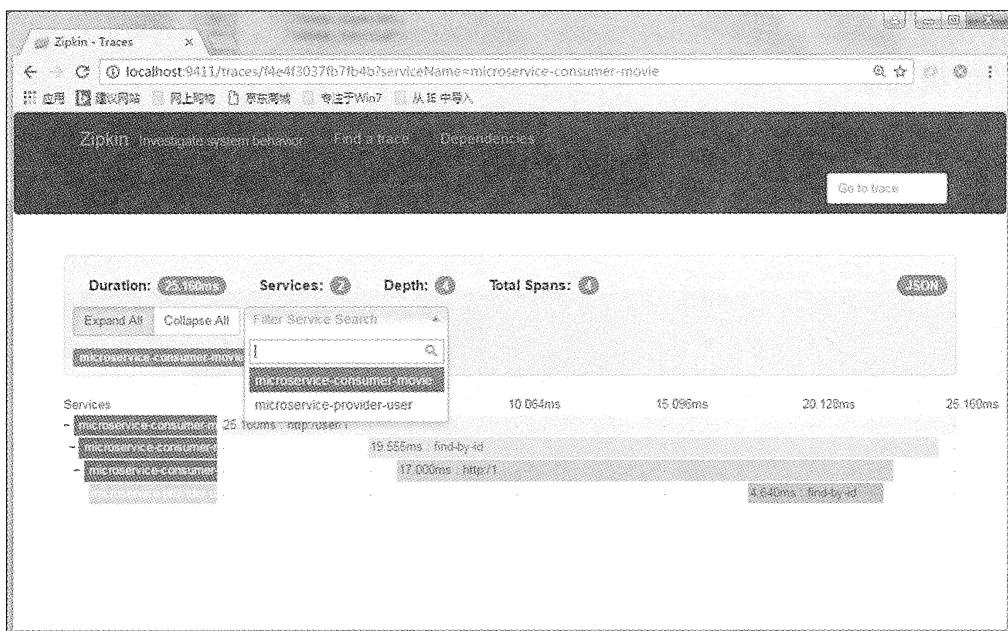


图 10-5 trace 详情

7. 单击图 10-5 中的 span，即可获得 span 的详细信息，如图 10-6 所示。

microservice-consumer-movie.http://1: 11.000ms			
AKA: microservice-consumer-movie,microservice-provider-user			
Date	Time	Relative Time	Annotation
2017/1/18 上午 10:06:06		4.000ms	Client Send
2017/1/18 上午 10:06:06		6.000ms	Server Receive
2017/1/18 上午 10:06:06		14.000ms	Server Send
2017/1/18 上午 10:06:06		15.000ms	Client Receive

Key	Value
http.host	localhost
http.method	GET
http.path	/1
http.url	http://localhost:8000/1
Server Address	169.254.159.28:8010 (microservice-consumer-movie)

图 10-6 span 详情

8. Zipkin 还有助于分析微服务间的依赖关系。单击导航栏上的 Dependencies 按钮，即可看到如图 10-7 所示界面。当然，也可选择起止时间，让 Zipkin 分析微服务间的依赖关系。

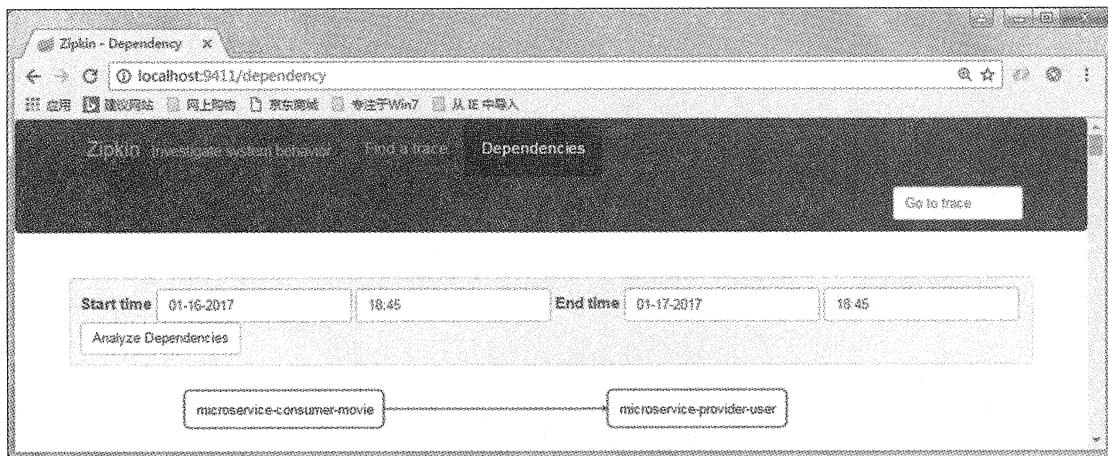


图 10-7 微服务依赖关系



Sleuth 支持多种采样器，例如 AlwaysSampler、NeverSampler、PercentageBased-Sampler 等。使用非常简单，例如：

```
@Bean  
public Sampler defaultSampler() {  
    return new AlwaysSampler();  
}
```



很多初学者在测试时，会感觉 Sleuth 没有正常工作，这是因为默认采样百分比是 10%，Sleuth 会忽略掉大量 span。因此，建议在开发、测试时，将属性 spring.sleuth.sampler.percentage 设置得大一点，例如 1.0 (100%)。

10.5.4 使用消息中间件收集数据

前文是使用 HTTP 直接收集跟踪数据的，本节来讨论如何使用消息中间件收集追踪数据。相比 HTTP 的方式来说，使用消息中间件有以下优点：

- 微服务与 Zipkin Server 解耦，微服务无须知道 Zipkin Server 的网络地址。
- 一些场景下，Zipkin Server 与微服务网络可能不通，使用 HTTP 直接收集的方式无法工作，此时可借助消息中间件实现数据收集。

笔者以 RabbitMQ 作为消息中间件进行演示，RabbitMQ 的安装详见 7.5.3.1 节。

10.5.4.1 改造 Zipkin Server

先来改造 Zipkin Server。

1. 复制项目 microservice-trace-zipkin-server，将 ArtifactId 修改为 microservice-trace-zipkin-server-stream。
2. 将 pom.xml 的依赖修改为以下内容。

```
<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>
```

3. 修改启动类，将注解 `@EnableZipkinServer` 修改为 `@EnableZipkinStreamServer`。
4. 将配置文件 `application.yml` 修改为如下内容。

```
server:
  port: 9411
spring:
  rabbitmq:
    host: localhost
    port: 5672
    username: guest
    password: guest
```

这样，Zipkin Server 就改造完成了。

10.5.4.2 改造微服务

改造完 Zipkin Server 后，接下来改造前文编写的微服务。

1. 复制项目 `microservice-simple-provider-user-trace-zipkin`，将 `ArtifactId` 修改为 `microservice-simple-provider-user-trace-zipkin-stream`。
2. 修改 `pom.xml`，添加以下依赖。

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-sleuth-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>
```

3. 修改配置文件 application.yml，删除其中的：

```
spring:
zipkin:
  base-url: http://localhost:9411
```

添加如下内容：

```
spring:
rabbitmq:
  host: localhost
  port: 5672
  username: guest
  password: guest
```

这样，微服务就改造完成了。同理，改造电影微服务，详见本书配套代码中的项目 microservice-simple-consumer-movie-trace-zipkin-stream。

依次启动 microservice-trace-zipkin-server-stream、microservice-simple-provider-user-trace-zipkin-stream、microservice-simple-consumer-movie-trace-zipkin-stream 后，按照前文的步骤测试，会发现依然可以正常跟踪微服务的调用。

10.5.5 存储跟踪数据

前文的示例中，Zipkin Server 是将数据存储在内存中的。这种方式一般不适用于生产环境，因为一旦 Zipkin Server 重启或发生崩溃，就会导致历史数据的丢失。

Zipkin Server 支持多种后端存储，例如 MySQL、Elasticsearch、Cassandra 等。本节将讨论如何将数据存储在 Elasticsearch 5.1.2 中。

用项目 microservice-trace-zipkin-server-stream 进行改造，让其使用 RabbitMQ 收集跟踪数据并使用 Elasticsearch 5.1.2 作为后端存储。

1. 复制项目 microservice-trace-zipkin-server-stream，将 ArtifactId 修改为 microservice -trace-zipkin-server-stream-elasticsearch。
2. 将 pom.xml 的依赖修改为以下内容。

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

```
<artifactId>spring-cloud-sleuth-zipkin-stream</artifactId>
</dependency>
<dependency>
    <groupId>io.zipkin.java</groupId>
    <artifactId>zipkin-autoconfigure-ui</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-storage-elasticsearch-http</artifactId>
    <version>1.16.2</version>
</dependency>
```

3. 修改配置文件 application.yml，添加如下内容。

```
zipkin:
  storage:
    type: elasticsearch
    elasticsearch:
      cluster: elasticsearch
      hosts: http://localhost:9200
      index: zipkin
      index-shards: 5
      index-replicas: 1
```

这样，代码就改造完成了。

测试

1. 启动 Elasticsearch 5.1.2。
2. 启动项目 microservice-trace-zipkin-server-stream-elasticsearch。
3. 启动项目 microservice-simple-provider-user-trace-zipkin-stream。
4. 启动项目 microservice-simple-consumer-movie-trace-zipkin-stream。
5. 按照前文讲解的方式测试，可获得预期结果。
6. 访问 Zipkin 首页，可正常显示跟踪数据。
7. 访问 http://localhost:9200/_search，可看到类似如下的结果。

```
{  
    "took": 16,  
    "timed_out": false,  
    "_shards": {  
        "total": 5,  
        "successful": 5,  
        "failed": 0  
    },  
    "hits": {  
        "total": 12,  
        "max_score": 1,  
        "hits": [  
            {  
                "_index": "zipkin-2017-01-17",  
                ...  
            }  
        ]  
    }  
}
```

说明能够正常将数据存储在 Elasticsearch 5.1.2 中。

8. 重启 Zipkin Server，在 Zipkin Server 首页输入条件查询，仍可查询到历史数据，说明可以正常从 Elasticsearch 5.1.2 中读取数据。

11 Spring Cloud 常见问题与总结

在使用 Spring Cloud 的过程中，可能会遇到一些问题。事实上，不少问题已在前面的章节中以 WARNING 的形式标出。

本章来对 Spring Cloud 的常见问题做一些总结。

11.1 Eureka 常见问题

本节将总结 Eureka 使用中常会遇到的一些问题。

11.1.1 Eureka 注册服务慢

默认情况下，服务注册到 Eureka Server 的过程较慢。在开发或测试时，常常希望能够加速这一过程，从而提升工作效率。

Spring Cloud 官方文档详细描述了该问题的原因并提供了解决方案：

Why is it so Slow to Register a Service?

Being an instance also involves a periodic heartbeat to the registry (via the client's serviceUrl) with default duration 30 seconds. A service is not available for discovery by clients until the instance, the server and the client all have the same metadata in

their local cache (so it could take 3 heartbeats). You can change the period using eureka.instance.leaseRenewalIntervalInSeconds and this will speed up the process of getting clients connected to other services. In production it's probably better to stick with the default because there are some computations internally in the server that make assumptions about the lease renewal period.

简单翻译一下：服务的注册涉及到周期性心跳，默认 30 秒一次（通过客户端配置的 serviceUrl）。只有当实例、服务器端和客户端的本地缓存中的元数据都相同时，服务才能被其他客户端发现（所以可能需要 3 次心跳）。可以使用参数 eureka.instance.leaseRenewalIntervalInSeconds 修改时间间隔，从而加快客户端连接到其他服务的过程。在生产环境中最好坚持使用默认值，因为在服务器内部有一些计算，它们会对续约做出假设。

综上，要想解决服务注册慢的问题，只须将 eureka.instance.leaseRenewalIntervalInSeconds 设成一个更小的值。该配置用于设置 Eureka Client 向 Eureka Server 发送心跳的时间间隔，默认是 30，单位是秒。在生产环境中，建议坚持使用默认值。



原文来自：http://cloud.spring.io/spring-cloud-static/Camden.SR1/#_why_is_it_so_slow_to_register_a_service。

11.1.2 已停止的微服务节点注销慢或不注销

在开发环境下，常常希望 Eureka Server 能迅速有效地注销已停止的微服务实例。然而，由于 Eureka Server 清理无效节点周期长（默认 90 秒），以及自我保护模式等原因，可能会遇到微服务注销慢甚至不注销的问题。解决方案如下：

- Eureka Server 端：

配置关闭自我保护，并按需配置 Eureka Server 清理无效节点的时间间隔。

```
eureka.server.enable-self-preservation
# 设为false，关闭自我保护，从而保证会注销微服务
eureka.server.eviction-interval-timer-in-ms
# 清理间隔（单位毫秒，默认是60*1000）
```

- Eureka Client 端：

配置开启健康检查，并按需配置续约更新时间和到期时间。

```
eureka.client.healthcheck.enabled  
# 设为true，开启健康检查（需要spring-boot-starter-actuator依赖）  
eureka.instance.lease-renewal-interval-in-seconds  
# 续约更新时间间隔（默认30秒）  
eureka.instance.lease-expiration-duration-in-seconds  
# 续约到期时间（默认90秒）
```

值得注意的是，这些配置仅建议在开发或测试时使用，生产环境建议坚持使用默认值。

示例

- Eureka Server 配置：

```
eureka:  
  server:  
    enable-self-preservation: false  
    eviction-interval-timer-in-ms: 4000
```

- Eureka Client 配置：

```
eureka:  
  client:  
    healthcheck:  
      enabled: true  
  instance:  
    lease-expiration-duration-in-seconds: 30  
    lease-renewal-interval-in-seconds: 10
```



修改 Eureka 的续约频率可能会打破 Eureka 的自我保护特性，详见：<https://github.com/spring-cloud/spring-cloud-netflix/issues/373>。这意味着在生产环境中，如果想要使用 Eureka 的自我保护特性，应坚持使用默认配置。

11.1.3 如何自定义微服务的 Instance ID

本节来探讨如何自定义微服务的 Intance ID。Instance ID 用于唯一标识注册到 Eureka Server 上的微服务实例。

在 Eureka Server 的首页可以直观地看到各个微服务的 Instance ID。例如，图 11-1 中的itmuch:microservice-provider-user:8000 就是 Intance ID。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - itmuch:microservice-provider-user:8000

图 11-1 Eureka Server 上的微服务列表

在 Spring Cloud 中，服务的 Instance ID 的默认值是\${spring.cloud.client.hostname}:\${spring.application.name}:\${spring.application.instance_id:\${server.port}}。如果想要自定义这部分的内容，只须在微服务中配置eureka.instance.instance-id 属性即可，例如：

```
spring:  
  application:  
    name: microservice-provider-user  
eureka:  
  instance:  
    # 将Instance ID设置成IP:端口的形式  
    instance-id: ${spring.cloud.client.ipAddress}:${server.port}
```

这样，就可将微服务microservice-provider-user 的 Instance ID 设为 IP: 端口的形式。这样设置后，效果如图 11-2 所示。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
MICROSERVICE-PROVIDER-USER	n/a (1)	(1)	UP (1) - 192.168.0.59:8000

图 11-2 Eureka Server 上的微服务列表



Spring Cloud 初始化 Instance ID 的相关代码：

- org.springframework.cloud.netflix.eureka.EurekaClientAutoConfiguration
- org.springframework.cloud.commons.util.IdUtils.getDefaultInstanceId(PropertyResolver)
- org.springframework.cloud.netflix.eureka.EurekaInstanceConfigBean.getInstanceId()

11.1.4 Eureka 的 UNKNOWN 问题总结与解决

注册信息 UNKNOWN，是新手常会遇到的问题。如图 11-3，有两种 UNKNOWN 的情况，一种是应用名称 UNKNOWN，另一种是应用状态 UNKNOWN。下面分别讨论这两种情况。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
MICROSERVICE-CONFIG-SERVER-EUREKA	n/a (1)	{1}	UP (1) - kmuch/microservice-config-server-eureka:8080
MICROSERVICE-FOO	n/a (1)	{1}	UNKNOWN (1) - kmuch/microservice-foo:8081
UNKNOWN	n/a (1)	{1}	UP (1) - kmuch:8000

图 11-3 Eureka Server 上的微服务列表

应用名称 UNKNOWN

应用名称 UNKNOWN 显然不合适，首先是微服务的名称不够语义化，无法直观看出这是哪个微服务；更重要的是，我们常常使用应用名称消费对应微服务的接口。

一般来说，有两种情况会导致该问题的发生：

- 未配置 `spring.application.name` 或者 `eureka.instance.appname` 属性。如果这两个属性均不配置，就会导致应用名称 UNKNOWN 的问题。
- 某些版本的 SpringFox 会导致该问题，例如 SpringFox 2.6.0。建议使用 SpringFox 2.6.1 或更新版本。

微服务实例状态 UNKNOWN

微服务实例的状态 UNKNOWN 同样很麻烦。一般来讲，只会请求状态是 UP 的微服务。该问题一般由健康检查导致。

`eureka.client.healthcheck.enabled=true` 必须设置在 `application.yml` 中，而不能设置在 `bootstrap.yml` 中，否则一些场景下会导致应用状态 UNKNOWN 的问题。



- SpringFox 是一款基于 Spring 和 Swagger 的开源的 API 文档框架，前身是 `swagger-springmvc`。官网网站：<http://springfox.io/>。
- Swagger 是一款非常流行的 API 文档框架，它可帮助我们设计、构建、测试 RESTful 接口，也可生成 RESTful 接口文档。官方网站：<http://swagger.io/>。

11.2 Hystrix/Feign 整合 Hystrix 后首次请求失败

某些场景下，Feign 或 Ribbon 整合 Hystrix 后，会出现首次调用失败的问题。本节将对该问题作一些总结。

11.2.1 原因分析

Hystrix 默认的超时时间是 1 秒，如果在 1 秒内得不到响应，就会进入 fallback 逻辑。由于 Spring 的懒加载机制，首次请求往往比较慢，因此在某些机器（特别是配置低的机器）上，首次请求需要的时间可能就会大于 1 秒。

了解原因后，来总结如何解决该问题。

11.2.2 解决方案

有很多方式解决该问题，以下列举几种比较简单的方案。

- 方法一，延长 Hystrix 的超时时间，示例：

```
hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds: 5000
```

该配置让 Hystrix 的超时时间改为 5 秒。

- 方法二，禁用 Hystrix 的超时，示例：

```
hystrix.command.default.execution.timeout.enabled: false
```

- 方法三，对于 Feign，还可为 Feign 禁用 Hystrix，示例：

```
feign.hystrix.enabled: false
```

这样即可为 Feign 全局禁用 Hystrix 支持。该方式比较极端，一般不建议使用。

11.3 Turbine 聚合的数据不完整

在某些版本的 Spring Cloud（例如 Brixton SR5）中，Turbine 会发生该问题。该问题的直观体现是：使用 Turbine 聚合了多个微服务，但在 Hystrix Dashboard 上只能看到部分微服务的监控数据。

例如 Turbine 配置如下：

```
turbine:  
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-hystrix-  
    fallback-stream  
  clusterNameExpression: "'default'"
```

Turbine 理应聚合 microservice-consumer-movie 和 microservice-consumer-movie-feign-hystrix-fallback-stream 这两个微服务的监控数据，然而打开 Hystrix Dashboard 时，会发现 Dashboard 上只显示部分微服务的监控数据，如图 11-4 所示。

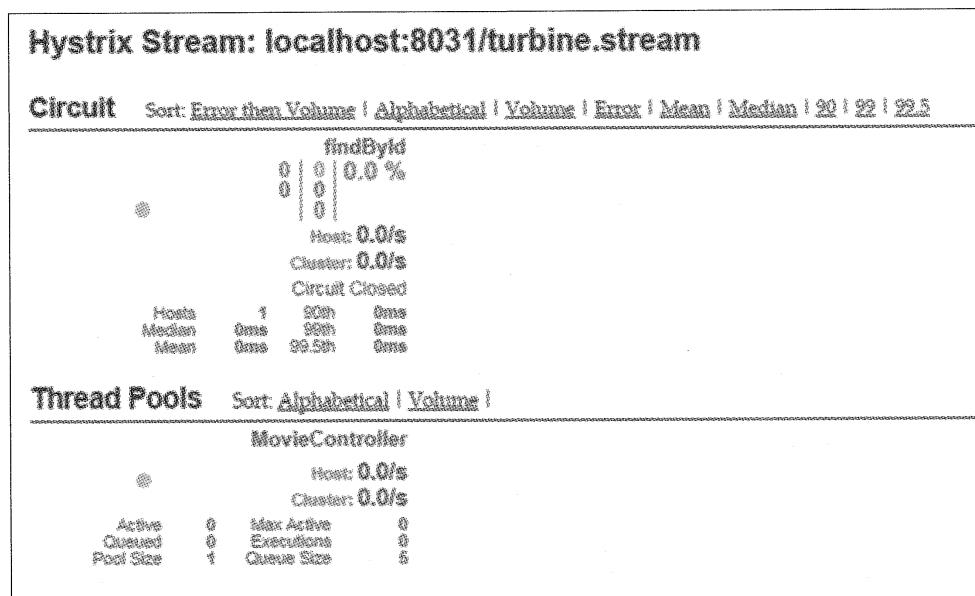


图 11-4 Hystrix Dashboard 监控页面

这显然不正常，那么如何解决这个问题呢？

解决方案

当 Turbine 聚合的微服务部署在同一台主机上时，就会出现该问题。

解决方案如下：

- 方法一：为各个微服务配置不同的 hostname，并将 preferIpAddress 设为 false 或者不设置。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
  instance:
    hostname: ribbon # 配置hostname
```

- 方法二：设置 turbine.combine-host-port = true。

```
turbine:  
  appConfig: microservice-consumer-movie,microservice-consumer-movie-feign-  
    hystrix-fallback-stream  
  clusterNameExpression: "'default'"  
  combine-host-port: true
```

- 方法三：升级 Spring Cloud 到 Camden 或更新版本。当然，也可单独升级 Spring Cloud Netflix 到 1.2.0 或更新版本（一般不建议单独升级 Spring Cloud Netflix，因为可能会跟 Spring Cloud 其他组件冲突）。

这是因为老版本中的 `turbine.combine-host-port` 默认值是 `false`。Spring Cloud 已经意识到该问题，所以在新的版本中将该属性的默认值修改为 `true`。该解决方案和方法二本质上是一致的。



- 相关代码：

```
org.springframework.cloud.netflix.turbine.TurbineProperties.combine-  
HostPort  
org.springframework.cloud.netflix.turbine.CommonsInstanceDiscovery.  
getInstance(String, String, String, Boolean)
```

- 相关 Issue：<https://github.com/spring-cloud/spring-cloud-netflix/issues/1087>。



本书所使用的 Spring Cloud Camden SR4 不存在该问题。

11.4 Spring Cloud 各组件配置属性

经过本书讲解，相信大家已经发现，Spring Cloud 中的大部分问题都可使用配置属性来解决。本节会将相关组件的配置的地址罗列出来，方便读者查阅与检索。

11.4.1 Spring Cloud 的配置

Spring Cloud 的所有组件配置都在其官方文档的附录，地址如下：

http://cloud.spring.io/spring-cloud-static/Camden.SR4/#_appendix_compendium_of_configuration_properties

11.4.2 原生配置

Spring Cloud 整合了很多类库，例如 Eureka、Ribbon、Feign 等。这些组件自身也有一些配置属性，如下：

- Eureka 的配置：<https://github.com/Netflix/eureka/wiki/Configuring-Eureka>。
- Ribbon 的配置：<https://github.com/Netflix/ribbon/wiki/Programmers-Guide>。
- Hystrix 的配置：<https://github.com/Netflix/Hystrix/wiki/Configuration>。
- Turbine 的配置：[https://github.com/Netflix/Turbine/wiki/Configuration-\(1.x\)](https://github.com/Netflix/Turbine/wiki/Configuration-(1.x))。

11.5 Spring Cloud 定位问题思路总结

本节对如何定位 Spring Cloud 问题做一些总结。

根据笔者观察，Spring Cloud 进入 Camden 时代后，已经比较稳定。一般来说，问题都不是 Spring Cloud 本身的 Bug 导致。因此，读者排查问题的思路不妨按照以下步骤展开。

1. 排查配置问题

排查配置有无问题，举几个简单的例子。

• YAML 缩进是否正确

曾经有朋友发现 Spring Cloud 应用程序无法正常启动，或配置无法正常加载。经笔者协助，发现仅仅是 YAML 配置文件缩进不正确。

类似问题应在编码的过程中严格规避。

• 配置属性是否正确

配置的属性写错，也是一个非常常见的问题。尽管该问题很低级，但从笔者的观察来看，不少初学者都会遇到这类问题。

很多场景下，这类问题可借助 IDE 的提示功能来排查——当 IDE 不自动提示或给出警告时，应格外注意。

• 配置属性的位置是否正确

配置属性位置不正确可能会导致应用的不正常。举几个常见的例子：

- 应当配置在 Eureka Client 项目上的属性，配置在了 Eureka Server 项目上。
- 应当写在 bootstrap.yml 中的属性，写在了 application.yml 中，例如：

```
spring:  
  cloud:  
    config:
```

```
uri: http://localhost:8080/
```

该属性应当存放在 bootstrap.yml 中。

- 应当写在 application.yml 的属性，写在了 bootstrap.yml 中，例如：

```
eureka.client.healthcheck.enabled=true
```

2. 排查环境问题

如确认配置无误，即可考虑运行环境是否存在问题。举几个例子：

- **环境变量**

例如 Java 环境变量、Maven 环境变量以及 Docker 容器环境变量等。当应用无法正常工作时，应该确保环境变量配置正确。

- **依赖下载是否完整**

曾经有朋友遇到应用无法正常启动的问题，最终发现仅仅是依赖没有下载完整所致。因此，建议在启动应用前，使用以下命令打包，从而确认依赖的完整性。

```
mvn clean package
```

- **网络问题**

微服务之间通过网络保持通信，因此，网络常常是排查问题的关键。当问题发生时，可优先排查网络问题。

3. 排查代码问题

如经过以上步骤，依然没有定位到 Spring Cloud 的问题，那么可能是编写的代码出了问题。很多时候，常常因为少了某个注解，或是依赖缺失，而导致了各种异常。

许多场景下，设置合理的日志级别，会对问题的定位有奇效。

4. 排查 Spring Cloud 自身的问题

如果确定不是自身代码问题，就可 Debug 一下 Spring Cloud 的代码了。同时，可在 GitHub 等平台给 Spring Cloud 项目组提交 Issue，然后参考官方回复，尝试规避相应问题。如问题无法规避，就需要 Spring Cloud 进行扩展，或者修复 Spring Cloud 的 Bug，从而满足需求。此时，请不要忘记在 Spring Cloud 的 Github 上 Pull Request，协助官方改进 Spring Cloud，让 Spring Cloud 更加完善、稳定。



可供参考的资源：

- 各项目自身的 GitHub，例如 Eureka 的 GitHub：<https://github.com/Netflix/eureka>。

- Spring Cloud 对应项目的 GitHub，例如 Eureka 项目在 Spring Cloud Netflix 中：[*https://github.com/spring-cloud/spring-cloud-netflix*](https://github.com/spring-cloud/spring-cloud-netflix)。
- Spring Cloud 的 StackOverflow：[*http://stackoverflow.com/questions/tagged/spring-cloud*](http://stackoverflow.com/questions/tagged/spring-cloud)。
- Spring Cloud 的 Gitter：[*https://gitter.im/spring-cloud/spring-cloud*](https://gitter.im/spring-cloud/spring-cloud)。
- Spring Cloud 中国社区：[*http://springcloud.cn*](http://springcloud.cn)。

在这些地方，均有官方人员参与，可帮助我们迅速解决问题。

12

Docker 入门

12.1 Docker 简介

Docker 是一个开源的容器引擎，它有助于更快地交付应用。Docker 可将应用程序和基础设施层隔离，并且能将基础设施当作程序一样进行管理。使用 Docker，可更快地打包、测试以及部署应用程序，并可以缩短从编写到部署运行代码的周期。



- Docker 的官方网站：<https://www.docker.com/>。
- Docker 的 GitHub：<https://github.com/docker/docker>。

12.2 Docker 的架构

看一下来自 Docker 官方文档的架构图，如图 12-1 所示。

接下来讲解一下图中包含的组件。

- Docker daemon (Docker 守护进程)

Docker daemon 是一个运行在宿主机 (DOCKER_HOST) 的后台进程。可通过 Docker 客户端与之通信。

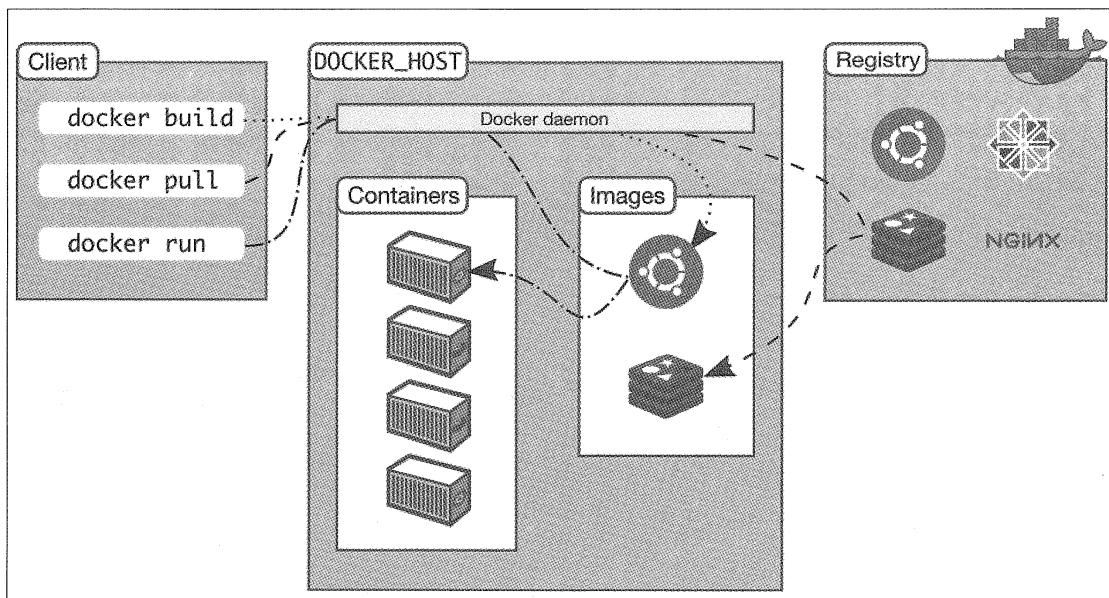


图 12-1 Docker 架构图

- Client (Docker 客户端)

Docker 客户端是 Docker 的用户界面，它可以接受用户命令和配置标识，并与 Docker daemon 通信。图中，`docker build` 等都是 Docker 的相关命令。

- Images (Docker 镜像)

Docker 镜像是一个只读模板，它包含创建 Docker 容器的说明。它和系统安装光盘有点像——使用系统安装光盘可以安装系统，同理，使用 Docker 镜像可以运行 Docker 镜像中的程序。

- Container (容器)

容器是镜像的可运行实例。镜像和容器的关系有点类似于面向对象中，类和对象的关系。可通过 Docker API 或者 CLI 命令来启停、移动、删除容器。

- Registry

Docker Registry 是一个集中存储与分发镜像的服务。构建完 Docker 镜像后，就可在当前宿主机上运行。但如果想要在其他机器上运行这个镜像，就需要手动复制。此时可借助 Docker Registry 来避免镜像的手动复制。

一个 Docker Registry 可包含多个 Docker 仓库，每个仓库可包含多个镜像标签，每个标签对应一个 Docker 镜像。这跟 Maven 的仓库有点类似，如果把 Docker Registry 比作 Maven 仓库的话，那么 Docker 仓库就可理解为某 jar 包的路径，而镜像标签则可理解为 jar 包的版本号。

Docker Registry 可分为公有 Docker Registry 和私有 Docker Registry。最常用的 Docker Registry 莫过于官方的 Docker Hub，这也是默认的 Docker Registry。Docker Hub 上存放着大量优秀的镜像，可使用 Docker 命令下载并使用。

12.3 安装 Docker

Docker 官方建议将 Docker 运行在 Linux 操作系统上。当然，Docker 也可运行在其他的平台，例如 Windows、Mac OS 等。

本节将演示如何在 CentOS 上安装 Docker，其他操作系统上的安装可参考官方文档：[https://docs.docker.com/engine/installation/。](https://docs.docker.com/engine/installation/)

12.3.1 系统要求

- Docker 运行在 CentOS 7.X 之上。
- Docker 需要安装在 64 位平台。

12.3.2 移除非官方软件包

Red Hat 操作系统包含了一个旧版本的 Docker 软件包，该旧版本软件包的名称是“docker”（新版是“docker-engine”）。因此，如已安装该软件包，那么需要执行以下命令移除。

```
sudo yum -y remove docker
```

执行该命令只会移除旧版本的 Docker，/var/lib/docker 目录中的内容不会被删除，因此，旧版本 Docker 所创建的镜像、容器、卷等都会保留下来。

12.3.3 设置 Yum 源

Docker 有多种安装方式，例如 Yum 安装、RPM 包安装、Shell 安装等。本节以 Yum 为例进行讲解。

1. 安装 yum-utils，这样就能使用 yum-config-manager 工具设置 Yum 源。

```
sudo yum install -y yum-utils
```

2. 执行以下命令，添加 Docker 的 Yum 源。

```
sudo yum-config-manager \
    --add-repo \
    https://docs.docker.com/engine/installation/linux/repo_files/centos/docker.
repo
```

3. [可选] 启用测试仓库。测试仓库包含在 docker.repo 文件中，但默认情况下是禁用的。如需启用测试仓库，可使用以下命令：

```
sudo yum-config-manager --enable docker-testing
```

想要禁用测试仓库，可执行以下命令：

```
sudo yum-config-manager --disable docker-testing
```

12.3.4 安装 Docker

1. 更新 Yum 包的索引。

```
sudo yum makecache fast
```

2. 安装最新版本的 Docker。

```
sudo yum -y install docker-engine
```

这样，经过一段时间的等待后，Docker 就安装完成了。

3. 在生产系统中，可能需要安装指定版本的 Docker，而并不总是安装最新版本。执行以下命令，即可列出可用的 Docker 版本。

```
yum list docker-engine.x86_64 --showduplicates | sort -r
```

其中，sort -r 命令表示对结果由高到低排序。执行后，可看到类似于如下的表格：

docker-engine.x86_64	1.13.0-1.el7.centos	docker-main
docker-engine.x86_64	1.12.6-1.el7.centos	docker-main
docker-engine.x86_64	1.12.5-1.el7.centos	docker-main
...		

该表格有三列，第一列是软件包名称，第二列是版本字符串，第三列是仓库名称，表示软件包存储的位置，例如 docker-main、docker-testing 等。列出 Docker 版本后，可使用以下命令安装指定版本的 Docker。

```
sudo yum -y install docker-engine-<VERSION_STRING>
```

例如：

```
sudo yum -y install docker-engine-1.13.0
```

4. 启动 Docker。

```
sudo systemctl start docker
```

5. 执行以下命令，验证安装是否正确。

```
sudo docker run hello-world
```

如看到类似于如下的结果，则说明安装正确。

```
Unable to find image 'hello-world:latest' locally
...
Hello from Docker!
This message shows that your installation appears to be working correctly.
...
```

6. 查看 Docker 版本。

```
docker version
```

可看到类似于如下的结果：

```
Client:
Version: 1.13.0
API version: 1.25
Go version: go1.7.3
Git commit: 49bf474
Built: Tue Jan 17 09:55:28 2017
OS/Arch: linux/amd64

Server:
Version: 1.13.0
API version: 1.25 (minimum version 1.12)
Go version: go1.7.3
Git commit: 49bf474
Built: Tue Jan 17 09:55:28 2017
OS/Arch: linux/amd64
Experimental: false
```

由结果可知当前 Docker 版本、API 版本、Go 语言版本等信息。

12.3.5 卸载 Docker

1. 卸载 Docker 软件包。

```
sudo yum -y remove docker-engine
```

2. 如需删除镜像、容器、卷以及自定义的配置文件，可执行以下命令：

```
sudo rm -rf /var/lib/docker
```

12.4 配置镜像加速器

国内访问 Docker Hub 的速度很不稳定，有时甚至出现连接不上的情况。本节来为 Docker 配置镜像加速器，从而解决这个问题。目前国内很多云服务商都提供了镜像加速的服务。

常用的镜像加速器有：阿里云加速器、DaoCloud 加速器等。各厂商镜像加速器的使用方式大致类似，本节以阿里云加速器为例进行讲解。

1. 注册阿里云账号后，即可在阿里云控制台（<https://cr.console.aliyun.com/#/accelerator>）看到如图 12-2 的页面。



图 12-2 阿里云管理控制台

2. 按照图 12-2 的说明，即可配置镜像加速器。

12.5 Docker 常用命令

Docker 有很多命令，这些命令有助于控制 Docker 的行为。本节将详细探讨 Docker 常用命令。

12.5.1 Docker 镜像常用命令

首先来讨论 Docker 镜像的常用命令。

- **搜索镜像**

可使用`docker search`命令搜索存放在 Docker Hub 中的镜像。例如：

```
docker search java
```

执行该命令后，Docker 就会在 Docker Hub 中搜索含有 `java` 这个关键词的镜像仓库。

执行该命令后，可看到类似于如下的表格：

NAME	DESCRIPTION	STARS	OFFICIAL	AUTOMATED
java	Java is a concurrent, ...	1281	[OK]	
anapsix/alpine-java	Oracle Java 8 (and 7) ...	190		[OK]
isuper/java-oracle	This repository conta ...	48		[OK]
lwieske/java-8	Oracle Java 8 Contain ...	32		[OK]
nimmis/java-centos	This is docker images ...	23		[OK]
...				

该表格包含五列，含义如下。

- NAME：镜像仓库名称。
- DESCRIPTION：镜像仓库描述。
- STARS：镜像仓库收藏数，表示该镜像仓库的受欢迎程度，类似于 GitHub 的 Stars。
- OFFICIAL：表示是否为官方仓库，该列标记为 [OK] 的镜像均由各软件的官方项目组创建和维护。由结果可知，`java` 这个镜像仓库是官方仓库，而其他的仓库都不是镜像仓库。
- AUTOMATED：表示是否是自动构建的镜像仓库。

- **下载镜像**

使用命令`docker pull`命令即可从 Docker Registry 上下载镜像，例如：

```
docker pull java
```

执行该命令后，Docker 会从 Docker Hub 中的 `java` 仓库下载最新版本的 Java 镜像。若镜像下载缓慢，可配置镜像加速器，详见 12.4 一节。

该命令还可指定想要下载的镜像标签以及 Docker Registry 地址，例如：

```
docker pull reg.itmuch.com/java:7
```

这样就可以从指定的 Docker Registry 中下载标签为 7 的 Java 镜像。

- 列出镜像

使用`docker images`命令即可列出已下载的镜像。

执行该命令后，将会看到类似于如下的表格：

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
java	latest	861e95c114d6	4 weeks ago	643.1 MB
hello-world	latest	c54a2cc56cbb	5 months ago	1.848 kB

该表格包含了 5 列，含义如下。

- REPOSITORY：镜像所属仓库名称。
- TAG：镜像标签。默认是 latest，表示最新。
- IMAGE ID：镜像 ID，表示镜像唯一标识。
- CREATED：镜像创建时间。
- SIZE：镜像大小。

- 删除本地镜像

使用`docker rmi`命令即可删除指定镜像。

例 1：删除指定名称的镜像。

```
docker rmi hello-world
```

表示删除 hello-world 这个镜像。

例 2：删除所有镜像。

```
docker rmi -f $(docker images)
```

-f 参数表示强制删除。



Docker 的命令：<https://docs.docker.com/engine/reference/commandline/>。

12.5.2 Docker 容器常用命令

本节来讨论 Docker 容器的常用命令。

1. 新建并启动容器

使用以下`docker run`命令即可新建并启动一个容器。

该命令是最常用的命令，它有很多选项，下面将列举一些常用的选项。

- -d 选项：表示后台运行
- -P 选项：随机端口映射
- -p 选项：指定端口映射，有以下四种格式。
 - ip:hostPort:containerPort
 - ip::containerPort
 - hostPort:containerPort
 - containerPort
- --network 选项：指定网络模式，该选项有以下可选参数：
 - --network=bridge：默认选项，表示连接到默认的网桥。
 - --network=host：容器使用宿主机的网络。
 - --network=container:NAME_or_ID：告诉 Docker 让新建的容器使用已有容器的网络配置。
 - --network=none：不配置该容器的网络，用户可自定义网络配置。

示例 1：

```
docker run java /bin/echo 'Hello World'
```

这样终端会打印 Hello World 的字样，跟在本地直接执行`/bin/echo 'Hello World'`一样。

示例 2：

```
docker run -d -p 91:80 nginx
```

这样就能启动一个 Nginx 容器。在本例中，为 docker run 添加了两个参数，含义如下：

`-d` # 后台运行
`-p` 宿主机端口:容器端口 # 开放容器端口到宿主机端口

访问 `http://Docker` 宿主机 IP:91/，将会看到如图 12-3 的界面：

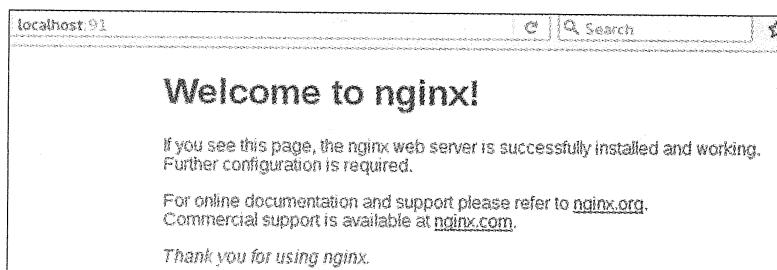


图 12-3 Nginx 首页



需要注意的是，使用 `docker run` 命令创建容器时，会先检查本地是否存在指定镜像。如果本地不存在该名称的镜像，Docker 就会自动从 Docker Hub 下载镜像并启动一个 Docker 容器。

2. 列出容器

使用 `docker ps` 命令即可列出运行中的容器。执行该命令后，可看到类似于如下的表格。

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS
NAMES					
784fd3b294d7	nginx	"nginx -g 'daemon off'"	20 minutes ago	Up 2 seconds	443/tcp, 0.0.0.0:91->80/tcp backstabbing_archimedes

如需列出所有容器（包括已停止的容器），可使用 `-a` 参数。

该表格包含了 7 列，含义如下。

- `CONTAINER_ID`: 表示容器 ID。
- `IMAGE`: 表示镜像名称。
- `COMMAND`: 表示启动容器时运行的命令。
- `CREATED`: 表示容器的创建时间。
- `STATUS`: 表示容器运行的状态。Up 表示运行中，Exited 表示已停止。
- `PORTS`: 表示容器对外的端口号。
- `NAMES`: 表示容器名称。该名称默认由 Docker 自动生成，也可使用 `docker run` 命令的 `--name` 选项自行指定。

3. 停止容器

使用 `docker stop` 命令，即可停止容器。例如：

```
docker stop 784fd3b294d7
```

其中 `784fd3b294d7` 是容器 ID，当然也可使用 `docker stop` 容器名称来停止指定容器。

4. 强制停止容器

可使用 `docker kill` 命令发送 SIGKILL 信号来强制停止容器。例如：

```
docker kill 784fd3b294d7
```

5. 启动已停止的容器

使用 `docker run` 命令，即可新建并启动一个容器。对于已停止的容器，可使用 `docker start` 命令来启动。例如：

```
docker start 784fd3b294d7
```

6. 重启容器

可使用`docker restart`命令来重启容器。该命令实际上是先执行了`docker stop`命令，然后执行了`docker start`命令。

7. 进入容器

某场景下，可能需要进入运行中的容器。

- 使用`docker attach`命令进入容器。例如：

```
docker attach 784fd3b294d7
```

很多场景下，使用`docker attach`命令并不方便。当多个窗口同时 attach 到同一个容器时，所有窗口都会同步显示。同理，如果某个窗口发生阻塞，其他窗口也无法执行操作。

- 使用`nsenter`进入容器。

`nsenter`工具包含在 util-linux 2.23 或更高版本中。为了连接到容器，需要找到容器第一个进程的 PID，可通过以下命令获取：

```
docker inspect --format "{{.State.Pid}}" $CONTAINER_ID
```

获得 PID 后，就可使用`nsenter`命令进入容器了：

```
nsenter --target "$PID" --mount --uts --ipc --net --pid
```

下面给出一个完整的例子：

```
[root@localhost ~]# docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
NAMES
784fd3b294d7 nginx "nginx -g 'daemon off'" 55 minutes ago Up 3 minutes 443/tcp, 0.0.0.0:91->80/tcp backstabbing_archimedes
[root@localhost ~]# docker inspect --format "{{.State.Pid}}" 784fd3b294d7
95492
[root@localhost ~]# nsenter --target 95492 --mount --uts --ipc --net --pid
root@784fd3b294d7:/#
```

读者也可将以上两条命令封装成一个 Shell，从而简化进入容器的过程。

8. 删除容器

使用`docker rm`命令即可删除指定容器。

例 1：删除指定容器。

```
docker rm 784fd3b294d7
```

该命令只能删除已停止的容器，如需删除正在运行的容器，可使用`-f`参数。

例 2：删除所有的容器。

```
docker rm -f $(docker ps -a -q)
```



- Docker 的网络：<https://docs.docker.com/engine/userguide/networking/>。
- Docker 命令：<https://docs.docker.com/engine/reference/commandline/>。

13 将微服务运行在 Docker 上

13.1 使用 Dockerfile 构建 Docker 镜像

本节将讨论如何使用 Dockerfile 构建 Docker 镜像。Dockerfile 是一个文本文件，其中包含了若干条指令，指令描述了构建镜像的细节。

先来编写一个最简单的 Dockerfile。以前文下载的 Nginx 镜像为例（详见 12.5.2 节），来编写一个 Dockerfile 修改该镜像的首页。

1. 例如：

```
FROM nginx
RUN echo '<h1>Spring Cloud与Docker微服务实战</h1>' > /usr/share/nginx/html/index.html
```

该 Dockerfile 非常简单，其中的 FORM、RUN 都是 Dockerfile 的指令。

FROM 指令用于指定基础镜像，RUN 指令用于执行命令。

2. 在 Dockerfile 所在路径执行以下命令构建镜像：

```
docker build -t nginx:my .
```

其中，命令最后的点（.）用于路径参数传递，表示当前路径。

3. 执行以下命令，即可使用该镜像启动一个 Docker 容器。

```
docker run -d -p 92:80 nginx:my
```

4. 访问 <http://Docker宿主机IP:92/>，可看到如图 13-1 的界面。

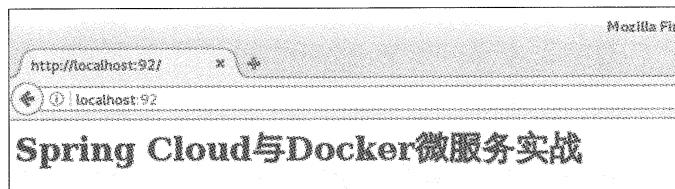


图 13-1 Nginx 首页

从本例不难看出 Dockerfile 的强大。仅仅编写了两行代码，就修改了原始镜像的行为。不仅如此，通过 Dockerfile，还可直观地看到修改镜像的具体过程。



除了使用 Dockerfile 构建镜像，也可手工制作 Docker 镜像，但这种方式烦琐、效率低，一般不适合生产，本书就不再赘述。

13.1.1 Dockerfile 常用指令

在前面的例子中，提到了 FORM、RUN 指令。事实上，Dockerfile 有十多个指令。本节将系统讲解这些指令，指令的一般格式为：指令名称 参数。

1. ADD 复制文件

ADD 指令用于复制文件，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：

- src 必须在构建的上下文内，不能使用例如：ADD .. /something /something 这样的命令，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。
- 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 对应内容文件将被下载到 dest。
- 如果 src 是一个 URL，同时 dest 以斜杠结尾，dest 将被视为目录，src 对应内容将被下载到 dest 目录。

- 如果 src 是一个目录，那么整个目录下的内容将会被复制，包括文件系统元数据。
- 如果文件是可识别的压缩包格式，则 docker 会自动解压。

示例：

```
ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
```

2. ARG 设置构建参数

ARG 指令用于设置构建参数，类似于 ENV。和 ARG 不同的是，ARG 设置的是构建时的环境变量，在容器运行时是不会存在这些变量的。

格式为：ARG <name>[=<default value>]。

示例：

```
ARG user1=someuser
```

3. CMD 容器启动命令

CMD 指令用于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，那么只有最后一条会被执行，如果启动容器时指定了运行的命令，则会覆盖掉 CMD 指定的命令。

支持 3 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）

示例：

```
CMD echo "This is a test." | wc -
```

4. COPY 复制文件

复制文件，格式为：

- COPY <src>... <dest>
- COPY [<src>, ... <dest>]

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

5. ENTRYPOINT 入口点

格式为：

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

ENTRYPOINT 和 CMD 指令的目的是一样的，都是指定 Docker 容器启动时执行的命令，可多次设置，但只有最后一个有效。

6. ENV 设置环境变量

ENV 指令用于设置环境变量，格式为：

- ENV <key> <value>
- ENV <key>=<value> ...

示例：

```
ENV JAVA_HOME /path/to/java
```

7. EXPOSE 声明暴露的端口

EXPOSE 指令用于声明在运行时容器提供服务的端口，格式为：EXPOSE <port> [<port>...]。

需要注意的是，这只是一个声明，运行时并不会因为该声明就打开相应端口。该指令的作用主要是帮助镜像使用者理解该镜像服务的守护端口；其次是当运行时使用随机映射时，会自动映射 EXPOSE 的端口。示例：

```
# 声明暴露一个端口示例
EXPOSE port1
# 相应的运行容器使用的命令
docker run -p port1 image
# 也可使用-P选项启动
docker run -P image

# 声明暴露多个端口示例
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
# 也可指定需要映射到宿主机器上的端口号
docker run -p host_port1:port1 -p host_port2:port2 -p host_port3:port3 image
```

8. FROM 指定基础镜像

使用 FORM 指令指定基础镜像，FORM 指令有点像 Java 里面的 extends 关键字。需要注意的是，FROM 指令必须指定且需要写在其他指令之前。FORM 指令后的所有指令都依赖于该指令所指定的镜像。

支持 3 种格式：

- FROM <image>
- FROM <image>:<tag>
- FROM <image>@<digest>

9. LABEL 为镜像添加元数据

LABEL 指令用于为镜像添加元数据。

格式为：LABEL <key>=<value> <key>=<value> <key>=<value> ...。

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."
```

10. MAINTAINER 指定维护者的信息

MAINTAINER 指令用于指定维护者的信息，用于为 Dockerfile 署名。

格式为：MAINTAINER <name>。

示例：

```
MAINTAINER 周立<eacdy0000@126.com>
```

11. RUN 执行命令

该指令支持两种格式：

- RUN <command>
- RUN ["executable", "param1", "param2"]

RUN <command>在 shell 终端中运行，在 Linux 中默认是/bin/sh -c，在 Windows 中是cmd /s /c，使用这种格式，就像直接在命令行中输入命令一样。RUN ["executable", "param1", "param2"] 使用 exec 执行，这种方式类似于函数调用。指定其他终端可以通过该方式操作，例如：RUN ["/bin/bash", "-c", "echo hello"]，该方式必须使用双引号"而不能使用单引号'，因为该方式会被转换成一个 JSON 数组。

12. USER 设置用户

该指令用于设置启动镜像时的用户或者 UID，写在该指令后的 RUN、CMD 以及 ENTRYPOINT 指令都将使用该用户执行命令。

格式为：USER 用户名。

示例：

```
USER daemon
```

13. VOLUME 指定挂载点

该指令使容器中的一个目录具有持久化存储的功能，该目录可被容器本身使用，也可共享给其他容器。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。格式为：VOLUME ["/data"]。

示例：

```
VOLUME /data
```

14. WORKDIR 指定工作目录

格式为：WORKDIR /path/to/workdir。

切换目录指令，类似于 cd 命令，写在该指令后的 RUN, CMD 以及 ENTRYPOINT 指令都将该目录作为当前目录，并执行相应的命令。

15. 其他

Dockerfile 还有一些其他的指令，例如 STOPSINGAL、HEALTHCHECK、SHELL 等。由于并不是十分常用，本书不再赘述。有兴趣的读者可前往 <https://docs.docker.com/engine/reference/builder/> 进行扩展阅读。



- Dockerfile 官方文档：<https://docs.docker.com/engine/reference/builder/#dockfile-reference>。
- Dockerfile 最佳实践：https://docs.docker.com/engine/userguide/eng-image/dockerfile_best-practices/#build-cache。

13.1.2 使用 Dockerfile 构建镜像

前文详细讲解了 Dockerfile 的常用指令，本节把前文编写的 Spring Cloud 微服务构建成 Docker 镜像。

准备工作

以项目 microservice-discovery-eureka 为例，首先执行以下命令，将项目构建成 jar 包：microservice-discovery-eureka-0.0.1-SNAPSHOT.jar。

```
mvn clean package      # 使用 Maven 打包项目
```

使用 Dockerfile 构建 Docker 镜像

1. 在 jar 包所在目录，创建名为 Dockerfile 的文件。

```
touch Dockerfile
```

2. 在 Dockerfile 中添加以下内容。

```
# 基于哪个镜像
FROM java:8

# 将本地文件夹挂载到当前容器
VOLUME ./tmp

# 复制文件到容器，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'

# 声明需要暴露的端口
EXPOSE 8761

# 配置容器启动后执行的命令
ENTRYPOINT ["java", "-Djava.security.egd=file:/dev/.urandom", "-jar", "/app.jar"]
```

3. 使用 docker build 命令构建镜像。

```
docker build -t itmuch/microservice-discovery-eureka:0.0.1 .
```

```
# 格式： docker build -t 仓库名称/镜像名称(:标签) Dockerfile 的相对位置
```

在这里，使用-t 选项指定了镜像的标签。执行该命令后，终端将会输出如下的内容。

```
Sending build context to Docker daemon 71.89 MB
Step 1/6 : FROM java:8
--> d23bdf5b1b1b
Step 2/6 : VOLUME /tmp
--> Running in da87c91b2ff7
--> ac325d3d36f0
Removing intermediate container da87c91b2ff7
Step 3/6 : ADD microservice-discovery-eureka-0.0.1-SNAPSHOT.jar app.jar
--> 11d1aeefaa05
Removing intermediate container 37eacdcd1a9a
Step 4/6 : RUN bash -c 'touch /app.jar'
--> Running in 140f4cff216f
```

```
--> 1487e388fc84
Removing intermediate container 140f4cff216f
Step 5/6 : EXPOSE 8761
--> Running in db272ee4f5db
--> c39508599bf1
Removing intermediate container db272ee4f5db
Step 6/6 : ENTRYPOINT java -Djava.security.egd=file:/dev/./urandom -jar /app.jar
--> Running in cd2798828a4f
--> d00aad554d2b
Removing intermediate container cd2798828a4f
Successfully built d00aad554d2b
```

由上，可看到镜像构建的详细过程与结果。



测试

1. 启动镜像

```
docker run -d -p 8761:8761 itmuch/microservice-discovery-eureka:0.0.1
```

2. 访问<http://Docker宿主机IP:8761/>，可正常显示 Eureka Server 首页。

可使用相同的方式，将其他微服务也构建成 Docker 镜像。

13.2 使用 Docker Registry 管理 Docker 镜像

至此，已经构建了 Docker 镜像，并将微服务运行在 Docker 之上。但是，一个完整的应用系统可能包含上百个微服务，那就可能对应着上百个镜像，如果考虑各个微服务的版本，那么可能会构建更多的镜像。这些镜像该如何管理呢？

13.2.1 使用 Docker Hub 管理镜像

Docker Hub 是 Docker 官方维护的 Docker Registry，上面存放着很多优秀的镜像。不仅如此，Docker Hub 还提供认证、工作组结构、工作流工具、构建触发器等工具来简化工作。

前文已经讲过，可使用`docker search`命令搜索存放在 Docker Hub 中的镜像。本节将详细探讨 Docker Hub 的使用。

注册与登录

Docker Hub 的使用非常简单，只须注册一个 Docker Hub 账号，就可正常使用了。登录后，可看到 Docker Hub 的主页，如图 13-2 所示。

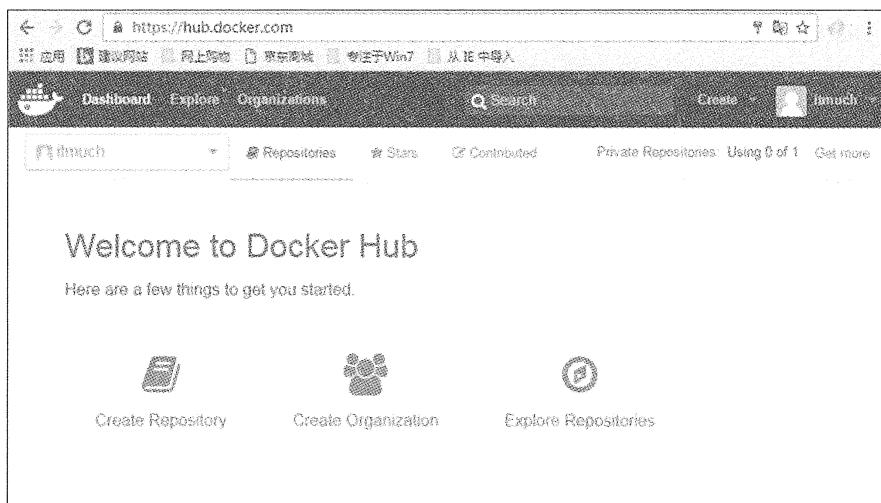


图 13-2 Docker Hub 主页

也可使用 `docker login` 命令登录 Docker Hub。输入该命令并按照提示输入账号和密码，即可完成登录。例如：

```
$ docker login
Login with your Docker ID to push and pull images from Docker Hub. If you don't
have a Docker ID, head over to https://hub.docker.com to create one.
Username: itmuch
Password:
Login Succeeded
```

创建仓库

点击 Docker Hub 主页上的 Create Repository 按钮，按照提示填入信息即可创建一个仓库。如图 13-3 所示，只须填入相关信息，并单击 Create 按钮，就可创建一个名为 `microservice-discovery-eureka` 的公共仓库。

推送镜像

下面来将前文构建的镜像推送到 Docker Hub。使用以下命令即可，例如：

```
docker push itmuch/microservice-discovery-eureka:0.0.1
```

经过一段时间的等待，就可推送成功。这样，就可在 Docker Hub 查看已推送的镜像。



图 13-3 创建仓库界面

13.2.2 使用私有仓库管理镜像

很多场景下，需使用私有仓库管理 Docker 镜像。相比 Docker Hub，私有仓库有以下优势：

- 节省带宽，对于私有仓库中已有的镜像，无须从 Docker Hub 下载，只须从私有仓库中下载即可。
- 更加安全。
- 便于内部镜像的统一管理。

本节来探讨如何搭建、使用私有仓库。可使用 docker-registry 项目或者 Docker Registry 2.0 来搭建私有仓库，但 docker-registry 已被官方标记为过时，并且已有 2 年不维护了，不建议使用。

先用 Docker Registry 2.0 搭建一个私有仓库，然后将 Docker 镜像推送到私有仓库。

搭建私有仓库

Docker Registry 2.0 的搭建非常简单，只须执行以下命令即可新建并启动一个 Docker Registry 2.0。

```
docker run -d -p 5000:5000 --restart=always --name registry2 registry:2
```

将镜像推送到私有仓库

前文使用了 docker push 命令将镜像推送到 Docker Hub，现在将前文构建的 itmuch/microservice-discovery-eureka:0.0.1 推送到私有仓库。

只须指定私有仓库的地址，即可将镜像推送到私有仓库。

```
docker push localhost:5000/itmuch/microservice-discovery-eureka:0.0.1
```

执行以上命令，发现推送并没有成功，且提示以下内容：

```
The push refers to a repository [localhost:5000/itmuch/microservice-discovery-eureka]
```

```
An image does not exist locally with the tag: localhost:5000/itmuch/microservice-discovery-eureka
```

Docker Hub 是默认的 Docker Registry，所以，itmuch/microservice-discovery-eureka:0.0.1 相当于 docker.io/itmuch/microservice-discovery-eureka:0.0.1。因此，要想将镜像推送到私有仓库，需要修改镜像标签，命令如下：

```
docker tag itmuch/microservice-discovery-eureka:0.0.1 localhost:5000/itmuch/microservice-discovery-eureka:0.0.1
```

修改镜像标签后，再次执行以下命令，即可将镜像推送到私有仓库。

```
docker push localhost:5000/itmuch/microservice-discovery-eureka:0.0.1
```



- docker-registry 的 GitHub： <https://github.com/docker/docker-registry>。
- Docker Registry 2.0 的 GitHub： <https://github.com/docker/distribution>。
- 本节中“私有仓库”表示私有 Docker Registry，并非 Docker 中仓库的概念。
- Docker Registry 2.0 需要 Docker 版本高于 1.6.0。
- 还可为私有仓库配置域名、SSL 登录、认证等。限于篇幅，本书不再赘述。有兴趣的读者可参考笔者的开源书：<http://git.oschina.net/itmuch/spring-cloud-book>。
- Docker Registry 2.0 能够满足大部分场景下的需求，但它不包含界面、用户管理、权限控制等功能。如果想要使用这些功能，可使用 Docker Trusted Registry。

13.3 使用 Maven 插件构建 Docker 镜像

Maven 是一个强大的项目管理与构建工具。如果可以使用 Maven 构建 Docker 镜像，工作就能得到进一步的简化。

经过调研，以下几款 Maven 的 Docker 插件进入笔者视野，如表 13-1 所示。

表 13-1 Maven 的 Docker 插件列表

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

笔者从各项目的功能性、文档易用性、更新频率、社区活跃度、Stars 等几个纬度考虑，选用了第一款。这是一款由 Spotify 公司开发的 Maven 插件。

下面来详细探讨如何使用 Maven 插件构建 Docker 镜像。

13.3.1 快速入门

以项目 microservice-discovery-eureka 为例。

1. 在 pom.xml 中添加 Maven 的 Docker 插件。

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <configuration>
    <imageName>itmuch/microservice-discovery-eureka:0.0.1</imageName>
    <baseImage>java</baseImage>
    <entryPoint>["java", "-jar", "${project.build.finalName}.jar"]</entryPoint>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
  </configuration>
</plugin>
```

```
</configuration>
</plugin>
```

简要说明一下插件的配置。

- `imageName`: 用于指定镜像名称, 其中 `itmuch` 是仓库名称, `microservice-discovery-eureka` 是镜像名称, `0.0.1` 是标签名称。
- `baseImage`: 用于指定基础镜像, 类似于 Dockerfile 中的 `FROM` 指令。
- `entrypoint`: 类似于 Dockerfile 的 `ENTRYPOINT` 指令。
- `resources.resource.directory`: 用于指定需要复制的根目录, `${project.build.directory}` 表示 `target` 目录。
- `resources.resource.include`: 用于指定需要复制的文件。 `${project.build.finalName}.jar` 指的是打包后的 jar 包文件。

2. 执行以下命令, 构建 Docker 镜像。

```
mvn clean package docker:build
```

发现终端输出类似于如下的内容:

```
[INFO] Building image itmuch/microservice-discovery-eureka:0.0.1
Step 1 : FROM java
--> 861e95c114d6
Step 2 : ADD /microservice-discovery-eureka-0.0.1-SNAPSHOT.jar //
--> 035a03f5b389
Removing intermediate container 2b0e70056f1d
Step 3 : ENTRYPOINT java -jar /microservice-discovery-eureka-0.0.1-SNAPSHOT.jar
--> Running in a0149704b949
--> eb96ca1402aa
Removing intermediate container a0149704b949
Successfully built eb96ca1402aa
```

由以上日志可知, 已成功构建了一个 Docker 镜像。

3. 执行 `docker images` 命令, 即可查看刚刚构建的镜像。

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
itmuch/microservice-discovery-eureka	0.0.1	eb96ca1402aa	39 seconds ago	685 MB

4. 启动以下镜像:

```
docker run -d -p 8761:8761 itmuch/microservice-discovery-eureka:0.0.1
```

发现该 Docker 镜像会很快地启动。

5. 访问测试

访问 <http://Docker> 宿主机 IP:8761，能够看到 Eureka Server 的首页。

13.3.2 插件读取 Dockerfile 进行构建

之前的示例直接在 pom.xml 中设置了一些构建的参数。很多场景下希望使用 Dockerfile 更精确、有可读性地构建镜像。

1. 首先在/microservice-discovery-eureka/src/main/docker 目录下，新建一个 Dockerfile 文件，例如：

```
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","-jar","/app.jar"]
```

2. 修改 pom.xml：

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <configuration>
    <imageName>itmuch/microservice-discovery-eureka:0.0.2</imageName>
    <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>
```

可以看到，不再指定 baseImage 和 entrypoint，而是使用 dockerDirectory 指定 Dockerfile 所在的路径。这样，就可以使用 Dockerfile 构建 Docker 镜像了。

13.3.3 将插件绑定在某个 phase 执行

很多场景下，有这样的需求，执行例如mvn clean package时，插件就自动为构建 Docker 镜像。要想实现这点，只须将插件的 goal 绑定在某个 phase 即可。

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。示例：

```
<plugin>
  <groupId>com.spotify</groupId>
  <artifactId>docker-maven-plugin</artifactId>
  <version>0.4.13</version>
  <executions>
    <execution>
      <id>build-image</id>
      <phase>package</phase>
      <goals>
        <goal>build</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <imageName>itmuch/microservice-discovery-eureka:0.0.3</imageName>
    <baseImage>java</baseImage>
    <entryPoint>["java", "-jar", "/${project.build.finalName}.jar"]</entryPoint>
    <resources>
      <resource>
        <targetPath>/</targetPath>
        <directory>${project.build.directory}</directory>
        <include>${project.build.finalName}.jar</include>
      </resource>
    </resources>
  </configuration>
</plugin>
```

由配置可知，只须添加如下配置：

```
<executions>
  <execution>
    <id>build-image</id>
    <phase>package</phase>
```

```

<goals>
    <goal>build</goal>
</goals>
</execution>
</executions>

```

就可将插件绑定在 package 这个 phase 上。也就是说，用户只须执行 mvn package，就会自动执行 mvn docker:build。当然，读者也可按照需求，将插件绑定到其他的 phase。

13.3.4 推送镜像

前文使用 docker push 命令实现了镜像的推送，也可使用 Maven 插件推送镜像。不妨使用 Maven 插件推送一个 Docker 镜像到 Docker Hub。

1. 修改 Maven 的全局配置文件 settings.xml，在其中添加以下内容，配置 Docker Hub 的用户信息。

```

<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>

```

2. 修改 pom.xml，示例：

```

<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.13</version>
    <configuration>
        <imageName>itmuch/microservice-discovery-eureka:0.0.4</imageName>
        <baseImage>java</baseImage>
        <entryPoint>["java", "-jar", "${project.build.finalName}.jar"]</entryPoint>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>

```

```
</resources>

<!-- 与maven配置文件settings.xml中配置的server.id一致，用于推送镜像 -->
<serverId>docker-hub</serverId>
</configuration>
</plugin>
```

如上，添加 serverId 段落，并引用 settings.xml 中的 server 的 id 即可。

3. 执行以下命令，添加 pushImage 的标识，表示推送镜像。

```
mvn clean package docker:build -DpushImage
```

经过一段时间的等待，会发现 Docker 镜像已经被 push 到 Docker Hub 了。同理，也可推送镜像到私有仓库，只需要将 imageName 指定成类似于如下的形式即可：

```
<imageName>localhost:5000/itmuch/microservice-discovery-eureka:0.0.4</imageName>
```



- 以上示例中，是通过 imageName 指定镜像名称和标签的，例如：

```
<imageName>itmuch/microservice-discovery-eureka:0.0.4</imageName>
```

也可借助 imageTags 元素更为灵活地指定镜像名称和标签，例如：

```
<configuration>
  <imageName>itmuch/microservice-discovery-eureka</imageName>
  <imageTags>
    <imageTag>0.0.5</imageTag>
    <imageTag>latest</imageTag>
  </imageTags>
  ...
<configuration>
```

这样就可为同一个镜像指定两个标签。

- 也可在执行构建命令时，使用 dockerImageTags 参数指定标签名称，例如：

```
mvn clean package docker:build -DpushImageTags -DdockerImageTags=
latest -DdockerImageTags=another-tag
```

- 如需重复构建相同标签名称的镜像，可将 forceTags 设为 true，这样就会覆盖构建相同标签的镜像。

```
<configuration>
    <!-- optionally overwrite tags every time image is built with
        docker:build -->
    <forceTags>true</forceTags>
<configuration>
```



Spotify 是全球最大的正版流媒体音乐服务平台。

13.4 常见问题与总结

Docker 官方说明文档非常完备，其中对 Docker 的常见问题进行了详细的总结。详见：[https://docs.docker.com/engine/faq/。](https://docs.docker.com/engine/faq/)

14

使用 Docker Compose 编排微服务

经过前文讲解，可使用 Dockerfile（或 Maven）构建镜像，然后使用 docker 命令操作容器，例如 docker run、docker kill 等。然而，使用微服务架构的应用系统一般包含若干个微服务，每个微服务一般都会部署多个实例。如果每个微服务都要手动启停，那么效率之低、维护量之大可想而知。

本章将讨论如何使用 Docker Compose 来轻松、高效地管理容器。为了简单起见，本章将 Docker Compose 简称为 Compose。

14.1 Docker Compose 简介

Compose 是一个用于定义和运行多容器 Docker 应用程序的工具，前身是 Fig。它非常适合用在开发、测试、构建 CI 工作流等场景。本书所使用的 Compose 版本是 1.10.0。



Compose 的 GitHub： <https://github.com/docker/compose>。

14.2 安装 Docker Compose

本节来讨论如何安装 Compose。

14.2.1 安装 Compose

Compose 有多种安装方式，例如通过 Shell、pip 以及将 Compose 作为容器安装等。本书讲解通过 Shell 来安装的方式，其他安装方式可详见官方文档：<https://docs.docker.com/compose/install/>。

1. 通过以下命令自动下载并安装适应系统版本的 Compose：

```
curl -L "https://github.com/docker/compose/releases/download/1.10.0/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

2. 为安装脚本添加执行权限：

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

这样，Compose 就安装完成了。

可使用以下命令测试安装结果：

```
docker-compose --version
```

可输出类似于如下的内容：

```
docker-compose version 1.10.0, build 4bd6f1a
```

说明 Compose 已成功安装。

14.2.2 安装 Compose 命令补全工具

现在已成功安装 Compose，然而，当输入 docker-compose 并按下 Tab 键时，Compose 并没有补全命令。要想使用 Compose 的命令补全，需要安装命令补全工具。

命令补全工具在 Bash 和 Zsh 下的安装方式不同，本书演示的是 Bash 下的安装。其他 Shell 以及其他操作系统上的安装，可详见 Docker 的官方文档：<https://docs.docker.com/compose/completion/>，笔者不再赘述。

执行以下命令，即可安装命令补全工具：

```
curl -L https://raw.githubusercontent.com/docker/compose/$(docker-compose version --short)/contrib/completion/bash/docker-compose -o /etc/bash_completion.d/docker-compose
```

这样，在重新登录后，输入 docker-compose 并按下 Tab 键，Compose 就可自动补全命令了。

14.3 Docker Compose 快速入门

本节将探讨 Compose 使用的基本步骤，并编写一个简单示例快速入门。

14.3.1 基本步骤

使用 Compose 大致有 3 个步骤：

- 使用 Dockerfile（或其他方式）定义应用程序环境，以便在任何地方重现该环境。
- 在 docker-compose.yml 文件中定义组成应用程序的服务，以便各个服务在一个隔离的环境中一起运行。
- 运行 docker-compose up 命令，启动并运行整个应用程序。

14.3.2 入门示例

下面以 microservice-discovery-eureka 为例讲解 Compose 的基本步骤。

1. 使用 mvn clean package 命令打包项目，获得 jar 包 microservice-discovery-eureka-0.0.1-SNAPSHOT.jar。
2. 在 microservice-discovery-eureka-0.0.1-SNAPSHOT.jar 所在路径（默认是项目的 target 目录）创建 Dockerfile 文件，并在其中添加如下内容。

```
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", "-jar", "/app.jar"]
```

3. 在 microservice-discovery-eureka-0.0.1-SNAPSHOT.jar 所在路径创建文件 docker-compose.yml，在其中添加如下内容。

```
version: '2'          # 表示该 docker-compose.yml 文件使用的是 Version 2 file
format
services:
  eureka:           # 指定服务名称
    build: .        # 指定 Dockerfile 所在路径
    ports:
      - "8761:8761" # 指定端口映射，类似 docker run 的 -p 选项，注意使用字符串形式
```

4. 在 docker-compose.yml 所在路径执行以下命令：

```
docker-compose up
```

Compose 就会自动构建镜像并使用镜像启动容器。也可使用 docker-compose up -d 后台启动并运行这些容器。

5. 访问 <http://宿主机IP:8761/>，即可访问 Eureka Server 首页。

14.3.3 工程、服务、容器

Docker Compose 将所管理的容器分为三层，分别是工程（ project ），服务（ service ）以及容器（ container ）。Docker Compose 运行目录下的所有文件（ docker-compose.yml 、 extends 文件或环境变量文件等）组成一个工程（默认为 docker-compose.yml 所在目录的目录名称）。一个工程可包含多个服务，每个服务中定义了容器运行的镜像、参数和依赖，一个服务可包括多个容器实例。

对应 14.3.2 节，工程名称是 docker-compose.yml 所在的目录名。该工程包含了 1 个服务，服务名称是 eureka 。执行 docker-compose up 时，启动了 eureka 服务的 1 个容器实例。

14.4 docker-compose.yml 常用命令

docker-compose.yml 是 Compose 的默认模板文件。该文件有多种写法，例如 Version 1 file format 、 Version 2 file format 、 Version 2.1 file format 、 Version 3 file format 等。其中， Version 1 file format 将逐步被弃用， Version 2.x 及 Version 3.x 基本兼容，是未来的趋势。考虑到目前业界的使用情况，本节只讨论 Version 2 file format 下的常用命令。

- build

配置构建时的选项，Compose 会利用它自动构建镜像。 build 的值可以是一个路径，例如：

```
build: ./dir
```

也可以是一个对象，用于指定 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 中的 CMD 指令，格式如下：

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

- 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
```

- environment

环境变量设置，可使用数组或字典两种方式。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，可指定一个文件路径或路径列表。如果通过 docker-compose -f FILE 指定了 Compose 文件，那么 env_file 中的路径是 Compose 文件所在目录的相对路径。使用 environment 指定的环境变量会覆盖 env_file 指定的环境变量。示例：

```
env_file: .env

env_file:
  - ./common.env
  - ./apps/web.env
  - /opt/secrets.env
```

- **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"
```

- **external_links**

连接到 docker-compose.yml 外部的容器，甚至并非 Compose 管理的容器，特别是提供共享或公共服务的容器。格式跟 links 类似，例如：

```
external_links:
  - redis_1
  - project_db_1:mysql
  - project_db_1:postgresql
```

- **image**

指定镜像名称或镜像 ID，如果本地不存在该镜像，Compose 会尝试下载该镜像。

示例：

```
image: java
```

- **links**

连接到其他服务的容器。可以指定服务名称和服务别名（SERVICE:ALIAS），也可只指定服务名称。例如：

```
web:
  links:
    - db
    - db:database
    - redis
```

- **networks**

详见本书 14.6 节。

- network_mode

设置网络模式。示例：

```
network_mode: "bridge"
network_mode: "host"
network_mode: "none"
network_mode: "service:[service name]"
network_mode: "container:[container name/id]"
```

- ports

暴露端口信息，可使用HOST:CONTAINER的格式，也可只指定容器端口（此时宿主机将会随机选择端口），类似于docker run -p。

需要注意的是，当使用HOST:CONTAINER格式映射端口时，容器端口小于 60 将会得到错误的接口，因为 yaml 会把xx:yy的数字解析为 60 进制。因此，建议使用字符串的形式。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"
```

- volumes

卷挂载路径设置。可以设置宿主机路径（HOST:CONTAINER），也可指定访问模式（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`

从另一个服务或容器挂载卷。可指定只读（`ro`）或读写（`rw`），默认是读写（`rw`）。

示例：

```
volumes_from:  
  - service_name  
  - service_name:ro  
  - container:container_name  
  - container:container_name:rw
```



docker-compose.yml 还有很多其他命令，比如 `depends_on`、`pid`、`devices` 等。限于篇幅，笔者仅挑选常用的命令进行讲解，其他命令不再赘述。感兴趣的读者们可参考官方文档：<https://docs.docker.com/compose/compose-file/>。

14.5 docker-compose 常用命令

和 docker 命令一样，docker-compose 命令也有很多选项。下面来详细探讨 docker-compose 的常用命令。

- `build`

构建或重新构建服务。服务被构建后将会以 `project_service` 的形式标记，例如：`compo-setest_db`。

- `help`

查看指定命令的帮助文档，该命令非常实用。docker-compose 所有命令的帮助文档都可通过该命令查看。

```
docker-compose help COMMAND
```

示例：

```
docker-compose help build      # 查看docker-compose build的帮助
```

- `kill`

通过发送 SIGKILL 信号停止指定服务的容器。示例：

```
docker-compose kill eureka
```

该命令也支持通过参数来指定发送的信号，例如：

```
docker-compose kill -s SIGINT
```

- logs

查看服务的日志输出。

- port

打印绑定的公共端口。示例：

```
docker-compose port eureka 8761
```

这样就可输出 eureka 服务 8761 端口所绑定的公共端口。

- ps

列出所有容器。示例：

```
docker-compose ps
```

也可列出指定服务的容器，示例：

```
docker-compose ps eureka
```

- pull

下载服务镜像。

- rm

删除指定服务的容器。示例：

```
docker-compose rm eureka
```

- run

在一个服务上执行一个命令。示例：

```
docker-compose run web bash
```

这样即可启动一个 web 服务，同时执行 bash 命令。

- scale

设置指定服务运行容器的个数，以 service=num 的形式指定。示例：

```
docker-compose scale user=3 movie=3
```

- start

启动指定服务已存在的容器。示例：

```
docker-compose start eureka
```

- stop

停止已运行的容器。示例：

```
docker-compose stop eureka
```

停止后，可使用 docker-compose start 再次启动这些容器。

- up

构建、创建、重新创建、启动，连接服务的相关容器。所有连接的服务都会启动，除非它们已经运行。

docker-compose up 命令会聚合所有容器的输出，当命令退出时，所有容器都会停止。
使用 docker-compose up -d 可在后台启动并运行所有容器。



本节仅讨论常用的 docker-compose 命令，其他命令可详见 Docker 官方文档：
<https://docs.docker.com/compose/reference/overview/>。

14.6 Docker Compose 网络设置

本节将详细探讨 Compose 的网络设置。本节介绍的网络特性仅适用于 Version 2 file format，Version 1 file format 不支持该特性。

14.6.1 基本概念

默认情况下，Compose 会为应用创建一个网络，服务的每个容器都会加入该网络中。这样，容器就可被该网络中的其他容器访问，不仅如此，该容器还能以服务名称作为 hostname 被其他容器访问。

默认情况下，应用程序的网络名称基于 Compose 的工程名称，而项目名称基于 docker-compose.yml 所在目录的名称。如需修改工程名称，可使用 --project-name 标识或 COMPOSE_PROJECT_NAME 环境变量。

举个例子，假如一个应用程序在名为 myapp 的目录中，并且 docker-compose.yml 如下所示：

```
version: '2'
services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
    image: postgres
```

当运行 docker-compose up 时，将会执行以下几步：

1. 创建一个名为 myapp_default 的网络。
2. 使用 web 服务的配置创建容器，它以“web”这个名称加入网络 myapp_default。
3. 使用 db 服务的配置创建容器，它以“db”这个名称加入网络 myapp_default。

容器间可使用服务名称（web 或 db）作为 hostname 相互访问。例如，web 这个服务可使用 postgres://db:5432 访问 db 容器。

14.6.2 更新容器

当服务的配置发生更改时，可使用 docker-compose up 命令更新配置。

此时，Compose 会删除旧容器并创建新容器。新容器会以不同的 IP 地址加入网络，名称保持不变。任何指向旧容器的连接都会被关闭，容器会重新找到新容器并连接上去。

14.6.3 links

前文讲过，默认情况下，服务之间可使用服务名称相互访问。links 允许定义一个别名，从而使用该别名访问其他服务。举个例子：

```
version: '2'  
services:  
  web:  
    build: .  
    links:  
      - "db:database"  
  db:  
    image: postgres
```

这样 Web 服务就可使用 db 或 database 作为 hostname 访问 db 服务了。

14.6.4 指定自定义网络

一些场景下，默认的网络配置满足不了我们的需求，此时可使用 networks 命令自定义网络。networks 命令允许创建更加复杂的网络拓扑并指定自定义网络驱动和选项。不仅如此，还可使用 networks 将服务连接到不是由 Compose 管理的、外部创建的网络。

如下，在其中定义了两个自定义网络。

```
version: '2'  
  
services:
```

```
proxy:
  build: ./proxy
  networks:
    - front

app:
  build: ./app
  networks:
    - front
    - back

db:
  image: postgres
  networks:
    - back

networks:
  front:
    # Use a custom driver
    driver: custom-driver-1
  back:
    # Use a custom driver which takes special options
    driver: custom-driver-2
    driver_opts:
      foo: "1"
      bar: "2"
```

其中， proxy 服务与 db 服务隔离，两者分别使用自己的网络， app 服务可与两者通信。

由本例不难发现，使用 networks 命令，即可方便实现服务间的网络隔离与连接。

14.6.5 配置默认网络

除自定义网络外，也可为默认网络自定义配置。

```
version: '2'

services:
  web:
    build: .
    ports:
      - "8000:8000"
  db:
```

```
image: postgres

networks:
  default:
    # Use a custom driver
    driver: custom-driver-1
```

这样，就可为该应用指定自定义的网络驱动。

14.6.6 使用已存在的网络

一些场景下，并不需要创建新的网络，而只须加入已存在的网络，此时可使用 external 选项。示例：

```
networks:
  default:
    external:
      name: my-pre-existing-network
```

14.7 综合实战：使用 Docker Compose 编排 Spring Cloud 微服务

本节将使用 Compose 编排前文编写的 Spring Cloud 微服务。

14.7.1 编排 Spring Cloud 微服务

本节来编排前文编写的 Spring Cloud 微服务。表 14-1 是本节编排时用到的微服务项目。

表 14-1 本节编排的微服务列表

微服务项目名称	项目微服务中的角色
microservice-discovery-eureka	服务发现组件
microservice-provider-user	服务提供者
microservice-consumer-movie-ribbon-hystrix	服务消费者
microservice-gateway-zuul	API Gateway
microservice-hystrix-turbine	Hystrix 聚合监控工具
microservice-hystrix-dashboard	Hystrix 监控界面

编写代码

- 本节使用 Maven 插件构建 Docker 镜像，在各个项目的 pom.xml 中添加以下内容。

```
<!-- 添加docker-maven插件 -->
<plugin>
    <groupId>com.spotify</groupId>
    <artifactId>docker-maven-plugin</artifactId>
    <version>0.4.13</version>
    <configuration>
        <imageName>itmuch/${project.artifactId}:${project.version}</imageName>
        <forceTags>true</forceTags>
        <baseImage>java</baseImage>
        <entryPoint>["java", "-jar", "${project.build.finalName}.jar"]</entryPoint>
        <resources>
            <resource>
                <targetPath>/</targetPath>
                <directory>${project.build.directory}</directory>
                <include>${project.build.finalName}.jar</include>
            </resource>
        </resources>
    </configuration>
</plugin>
```

由配置可知，构建出来的镜像名称是itmuch/各个微服务的artifactId:各个微服务的版本，例如：microservice-discovery-eureka:0.0.1-SNAPSHOT。

- 前文中为各个项目配置的eureka.client.serviceUrl.defaultZone的值是`http://localhost:8761/eureka/`。由于 Docker 默认的网络模式是 bridge，各个容器的 IP 都不相同，因此使用`http://localhost:8761/eureka/`满足不了需求。可为 Eureka Server 所在容器配置一个主机名（例如 discover），并让各个微服务使用主机名访问 Eureka Server。

将所有微服务eureka.client.serviceUrl.defaultZone修改为如下内容。

```
eureka:
  client:
    serviceUrl:
      defaultZone: http://discovery:8761/eureka/
```

- 在每个项目的根目录执行以下命令，构建 Docker 镜像。

```
mvn clean package docker:build
```

- 编写 docker-compose.yml。

```
# 表示该docker-compose.yml文件使用的是Version 2 file format
version: '2'
# Version 2 file format的固定写法，为project定义服务
services:
    # 指定服务名称
    microservice-discovery-eureka:
        # 指定服务所使用的镜像
        image: itmuch/microservice-discovery-eureka:0.0.1-SNAPSHOT
        # 暴露端口信息
        ports:
            - "8761:8761"
    microservice-provider-user:
        image: itmuch/microservice-provider-user:0.0.1-SNAPSHOT
        # 连接到microservice-discovery-eureka，这边使用的是SERVICE:ALIAS的形式
        links:
            - microservice-discovery-eureka:discovery
    microservice-consumer-movie-ribbon-hystrix:
        image: itmuch/microservice-consumer-movie-ribbon-hystrix:0.0.1-SNAPSHOT
        links:
            - microservice-discovery-eureka:discovery
    microservice-gateway-zuul:
        image: itmuch/microservice-gateway-zuul:0.0.1-SNAPSHOT
        links:
            - microservice-discovery-eureka:discovery
    microservice-hystrix-dashboard:
        image: itmuch/microservice-hystrix-dashboard:0.0.1-SNAPSHOT
        links:
            - microservice-discovery-eureka:discovery
    microservice-hystrix-turbine:
        image: itmuch/microservice-hystrix-turbine:0.0.1-SNAPSHOT
        links:
            - microservice-discovery-eureka:discovery
```

启动与测试

1. 执行以下命令启动项目。

```
docker-compose up
```

效果如图 14-1 所示。

Instances currently registered with Eureka			
Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a (1)	{1}	UP [1] - 6af4aaef38722.microservice-consumer-movie:8010
MICROSERVICE-GATEWAY-ZUUL	n/a (1)	{1}	UP [1] - 7a29925086cc.microservice-gateway-zuul:8040
MICROSERVICE-HYSTRIX-TURBINE	n/a (1)	{1}	UP [1] - c5ced2ed3686.microservice-hystrix-turbine:8031
MICROSERVICE-PROVIDER-USER	n/a (1)	{1}	UP [1] - 3233f11a77.microservice-provider-user:8001

图 14-1 Eureka Server 上的微服务列表

2. 按照前文章节中的描述，测试各微服务是否能够正常运行。

简化 Compose 编写

前文讲过，在 Version 2 file format 的 docker-compose.yml 中，同一个 Compose 工程中的所有服务共享一个隔离网络，可使用服务名称作为主机名来发现其他服务。因此，本节中的 docker-compose.yml 也可简化成如下形式。

```
version: '2'
services:
  discovery:
    image: itmuch/microservice-discovery-eureka:0.0.1-SNAPSHOT
    ports:
      - "8761:8761"
  microservice-provider-user:
    image: itmuch/microservice-provider-user:0.0.1-SNAPSHOT
  microservice-consumer-movie-ribbon-hystrix:
    image: itmuch/microservice-consumer-movie-ribbon-hystrix:0.0.1-SNAPSHOT
  microservice-gateway-zuul:
    image: itmuch/microservice-gateway-zuul:0.0.1-SNAPSHOT
  microservice-hystrix-dashboard:
    image: itmuch/microservice-hystrix-dashboard:0.0.1-SNAPSHOT
  microservice-hystrix-turbine:
    image: itmuch/microservice-hystrix-turbine:0.0.1-SNAPSHOT
```



Version 2 file format 与 Version 1 file format 的区别造成的问题：<https://github.com/docker/compose/issues/3926>。

14.7.2 编排高可用的 Eureka Server

在 14.7.2 节中，构建了一个双节点的 Eureka Server 集群，本节将使用 Compose 编排该 Eureka Server 集群。

1. 执行以下命令构建 Docker 镜像。

```
mvn clean package docker:build
```

2. 编写 docker-compose.yml，如下：

```
version: "2"          # 表示使用docker-compose.yml的Version 2 file format编写
services:
  microservice-discovery-eureka-ha1:
    hostname: peer1      # 指定hostname
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    links:
      - microservice-discovery-eureka-ha2
    ports:
      - "8761:8761"
    environment:
      - spring.profiles.active=peer1
  microservice-discovery-eureka-ha2:
    hostname: peer2
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    links:
      - microservice-discovery-eureka-ha1
    ports:
      - "8762:8762"
    environment:
      - spring.profiles.active=peer2
```

由文件内容可知，我们定义了两个服务：microservice-discovery-eureka-ha1 和 microservice-discovery-eureka-ha2。他们的 hostname 分别是 peer1 和 peer2，通过 links 标签互相连接。

3. 执行以下命令启动项目。

```
docker-compose up
```

然而，终端会输出类似以下的异常：

```
ERROR: Circular dependency between microservice-discovery-eureka-ha1 and
microservice-discovery-eureka-ha2
```

从异常可知，该写法存在循环依赖。也就是说，links 无法实现双向连接。如何解决这个问题呢？

解决循环依赖

该问题有很多解决方案，例如使用 ambassador pattern，使用外部 DNS 容器等。本节采用最简单的方式，配置如下。

```
version: "2"
services:
  # 默认情况下，其他服务可使用服务名称连接到该服务。对于peer2节点，它需连接
  # http://peer1:8761/eureka/，因此，可配置该服务的名称为peer1
  peer1:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    ports:
      - "8761:8761"
    environment:
      - spring.profiles.active=peer1
  peer2:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    hostname: peer2
    ports:
      - "8762:8762"
    environment:
      - spring.profiles.active=peer2
```



- 解决循环依赖的总结：<http://www.dockone.io/article/929>。
- ambassador pattern 官方介绍：https://docs.docker.com/engine/admin/ambassador_pattern_linking/。
- StackOverflow 上对该问题的深入探讨：<http://stackoverflow.com/questions/29307645/how-to-link-docker-container-to-each-other-with-docker-compose>。
- Github 上的相关 Issue：<https://github.com/docker/compose/issues/666>。

14.7.3 编排高可用 Spring Cloud 微服务集群及动态伸缩

本节再来写一个 Compose 编排 Spring Cloud 微服务的示例。在这个示例中，所有微服务节点最终都是高可用的。表 14-2 是本节编排时使用到的微服务项目。

表 14-2 本节编排的微服务列表

微服务项目名称	项目微服务中的角色
microservice-discovery-eureka-ha	服务发现组件
microservice-provider-user	服务提供者
microservice-consumer-movie-ribbon-hystrix	服务消费者
microservice-gateway-zuul	API Gateway
microservice-hystrix-turbine	Hystrix 聚合监控工具

编写代码

- 由于使用了 microservice-discovery-eureka-ha，需要将所有微服务的 eureka.client.serviceUrl.defaultZone 属性修改为如下内容。

```
eureka:
  client:
    service-url:
      defaultZone: http://peer1:8761/eureka/,http://peer2:8762/eureka/
```

- 在每个项目的根目录，执行以下命令构建 Docker 镜像。

```
mvn clean package docker:build
```

- 编写 docker-compose.yml。

```
version: "2"
services:
  peer1:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    ports:
      - "8761:8761"
    environment:
      - spring.profiles.active=peer1
  peer2:
    image: itmuch/microservice-discovery-eureka-ha:0.0.1-SNAPSHOT
    hostname: peer2
    ports:
      - "8762:8762"
    environment:
      - spring.profiles.active=peer2
```

```

microservice-provider-user:
  image: itmuch/microservice-provider-user:0.0.1-SNAPSHOT
microservice-consumer-movie-ribbon-hystrix:
  image: itmuch/microservice-consumer-movie-ribbon-hystrix:0.0.1-SNAPSHOT
microservice-gateway-zuul:
  image: itmuch/microservice-gateway-zuul:0.0.1-SNAPSHOT
microservice-hystrix-turbine:
  image: itmuch/microservice-hystrix-turbine:0.0.1-SNAPSHOT

```

启动与测试

- 执行以下命令启动项目

```
docker-compose up
```

启动后的效果如图 14-2 所示。

Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a {1}	{1}	UP {1} - 9d5a2bb10cad:microservice-consumer-movie:8010
MICROSERVICE-DISCOVERY-EUREKA-HA	n/a {2}	{2}	UP {2} - f33628b79bbb:microservice-discovery-eureka-ha:8761, peer2:microservice-discovery-eureka-ha:8762
MICROSERVICE-GATEWAY-ZUUL	n/a {1}	{1}	UP {1} - ec48c421f935:microservice-gateway-zuul:8040
MICROSERVICE-HYSTRIX-TURBINE	n/a {1}	{1}	UP {1} - 933c78c01717:microservice-hystrix-turbine:8031
MICROSERVICE-PROVIDER-USER	n/a {1}	{1}	UP {1} - 31ded80e2423:microservice-provider-user:8000

图 14-2 Eureka Server 上的微服务列表

- 按照前文各章节的讲解，测试各微服务能否正常运行。
- 执行以下命令，为各个微服务动态扩容。让除 Eureka Server 以外的所有微服务都启动 3 个实例。

```
docker-compose scale microservice-provider-user=3 microservice-consumer-movie-ribbon-hystrix=3 microservice-gateway-zuul=3 microservice-hystrix-turbine=3
```

效果如图 14-3 所示。

由图可知，除 Eureka Server 外，每个微服务都启动了 3 个节点，说明已实现微服务的动态扩容。同理，也可实现动态缩容。

Application	AMIs	Availability Zones	Status
MICROSERVICE-CONSUMER-MOVIE	n/a (3)	(3)	UP [3] - 9d5a2bb10ced/microservice-consumer-movie:8010 , a3f37fed913e/microservice-consumer-movie:8010 , 926ee40063cc/microservice-consumer-movie:8010
MICROSERVICE-DISCOVERY-EUREKA-HA	n/a (2)	(2)	UP [2] - f33628b79bbb/microservice-discovery-eureka-ha:8761 , peer2/microservice-discovery-eureka-ha:8762
MICROSERVICE-GATEWAY-ZUUL	n/a (3)	(3)	UP [3] - ec40ed21f515/microservice-gateway-zuul:8040 , 2bb8a9e9f683/microservice-gateway-zuul:8040 , 4c2236aba1fa/microservice-gateway-zuul:8040
MICROSERVICE-HYSTRIX-TURBINE	n/a (3)	(3)	UP [3] - 49b464268018/microservice-hystrix-turbine:8031 , 776a612fad04/microservice-hystrix-turbine:8031 , 933c76c01717/microservice-hystrix-turbine:8031
MICROSERVICE-PROVIDER-USER	n/a (3)	(3)	UP [3] - 95948fd68e80/microservice-provider-user:8000 , 39423c388898/microservice-provider-user:8000 , 31cd80a2421/microservice-provider-user:8000

图 14-3 Eureka Server 上的微服务列表

14.8 常见问题与总结

Compose 官方说明文档非常完备，其中对 Docker 的常见问题进行了详细的总结。详见：
[https://docs.docker.com/compose/faq/。](https://docs.docker.com/compose/faq/)

后记

至此，Spring Cloud 与 Docker 微服务架构实战的探索之旅已经结束。

由于篇幅有限，笔者无法为大家讲述微服务中的方方面面。微服务是一个非常宏观的话题，要想切实落地微服务架构，光靠一两本书是远远不够的。微服务粒度、持续集成、自动化机制、组织机构的建设乃至如何从传统架构向微服务架构迁移，都是值得我们深思的问题。

所幸，Pivotal 工程师 Matt Stine 的著作 *Migrating to Cloud-Native Application Architectures*（迁移到云原生应用架构）对本书未提到的许多话题有非常精辟的阐述，相信大家阅读后会对微服务有更深的理解与体会。

最后，希望大家能够享受这段 Spring Cloud 与 Docker 实战微服务架构的旅行，也希望本书可以切实地帮助大家实现微服务架构的落地。



Migrating to Cloud-Native Application Architectures（迁移到云原生应用架构）阅读地址：<https://pivotal.io/platform-as-a-service/migrating-to-cloud-native-application-architectures-ebook>，也可使用搜索引擎搜索中文翻译版辅助阅读。