构建微服务

Building Microservices

译者：RickieChina@hotmail.com

目录

一、微服务操作模型

这里并不是介绍微服务概念，如需要了解微服务，可以阅读 Fowler-Microservices 文章。本博客假定我们已开始使用微服务解耦单体应用，用来提升可部署性和可扩展性。

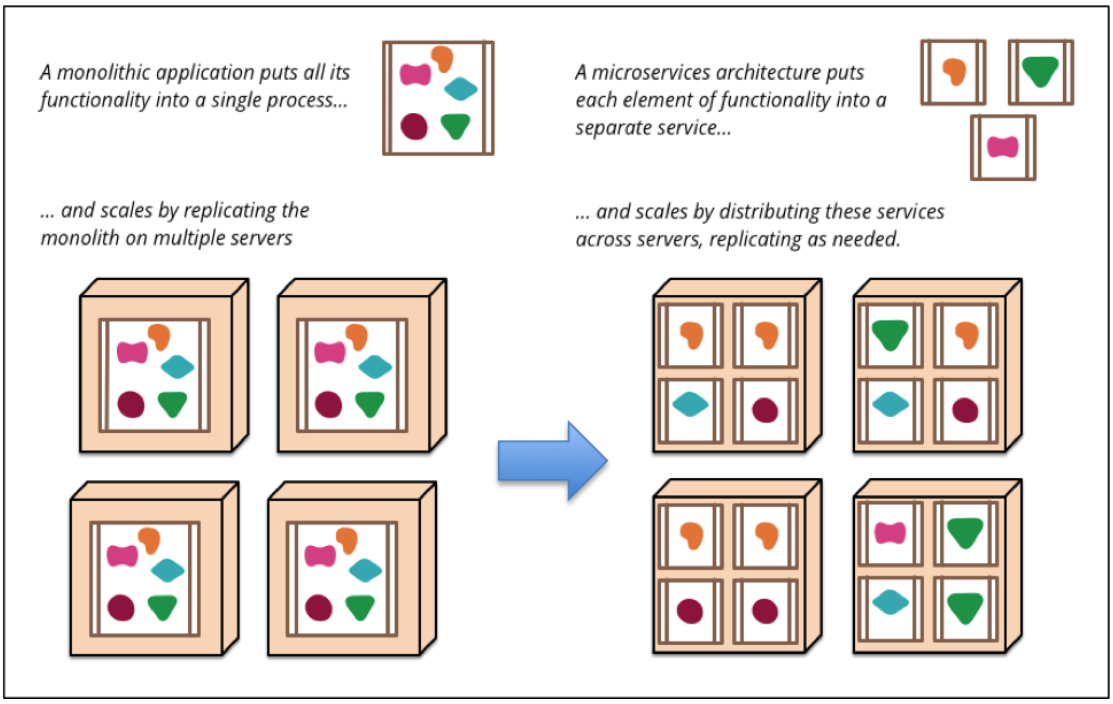
当我们在系统范围内部署大量的微服务时，一个新的挑战产生了，单体应用部署时不会发生。这篇文章将针对这些新的挑战，在系统范围内部署大量微服务时定义一套操作模型（operations model）。

这篇文章分为如下几个部分：

1. 前提条件；
2. 扩展；
3. 问题；
4. 需要的组件；
5. 参考模型；
6. 下一步；

1. 前提条件

当在系统范围内需要部署大量微服务时，需要什么条件呢？
根据 Flower 的文章，如下是我们想要得到的：



(**Source:**http://martinfowler.com/articles/microservices.html)

然而，在开始发布大量微服务替换单体应用之前，我们需要实现如下这些前置条件：

- 目标架构；
- 持续交付工具；
- 合适的组件结构；

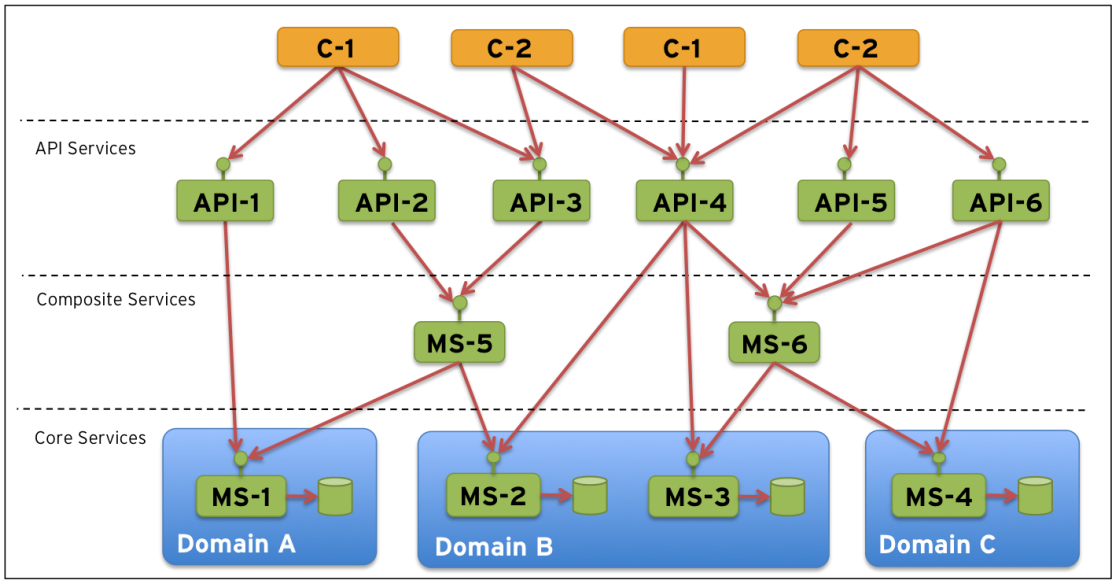
下面简要描述每一个前置条件。

1.1 目标架构

首先，我们需要分区微服务。例如，我们可以垂直分解微服务。

- 核心服务（Core services）-处理业务数据的持久化和实施业务逻辑和其他规则；
- 组合服务（Composite services）-组合服务指编排一组核心服务实现一个特定任务，或者从一组核心服务中聚合信息；
- API services – 向外暴露功能，例如允许第三方使用底层功能创建新的应用；

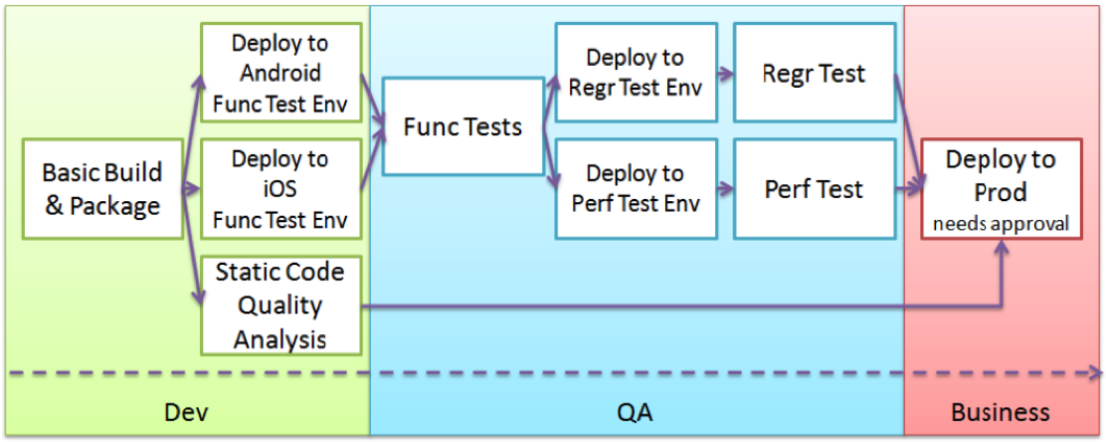
也可以水平上应用领域驱动分解。如下是一个目标架构：



备注：这仅仅是一个范例目标架构，你可以使用完全不同的架构。核心时在开始部署微服务之前，需要有简历一个目标架构。否则，你最终的架构有可能像一团面条一样，甚至比现有的单体应用更加糟糕。

1.2 持续交付

我们假定已经有了一套可持续交付的发布工具，以便我们可以高效地反复发布微服务。

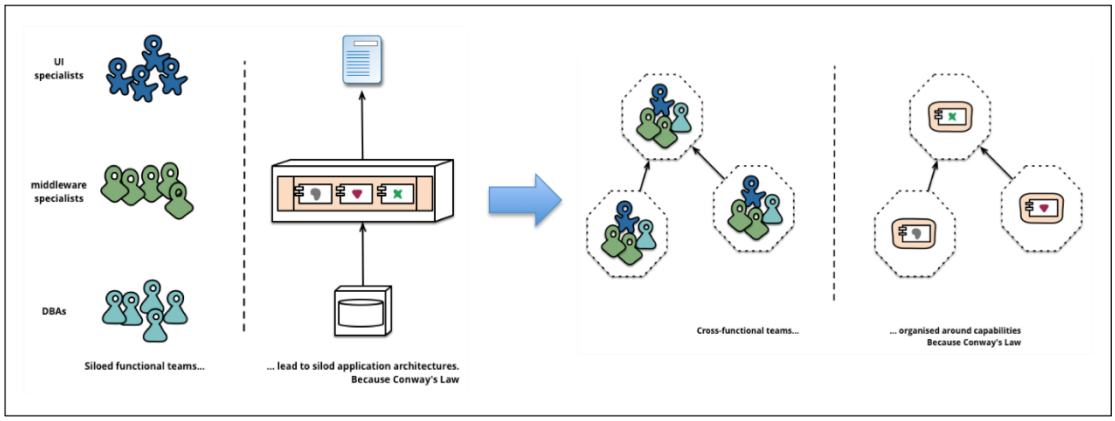


(Source: http://www.infoq.com/minibooks/emag-devops-toolchain)

1.3 合适的组织

最后，我们需要采用合适的组织结构，避免和 Conway 法则相冲突。Conway 法则如下：

Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.



(Source: http://martinfowler.com/articles/microservices.html)

2. 扩展

接下来是本文关注的要点：

当我们分解单体应用，并使用大量的微服务替换时，在系统范围内会发生什么呢？

(1) 大量的部署单元

将产生需要的小的微服务，而不是之前的一个大的单体应用，这将需要管理和跟踪大量的部署单元。

(2) 微服务将同时暴露和调用服务

在系统范围内，大部分的微服务彼此是互相连接的。

(3) 一些微服务暴露外部 API

这些微服务将负责包含其他的微服务不允许外部访问。

(4) 系统根据动态
新的微服务部署，替换老的服务。现有的微服务新增实例，满足增长的负荷。这意味着微服务将比以前更高的频率增加和下线。
(5) 平均故障时间(MTBF)将下降
在系统范围内，故障发生频率将更频繁。软件组件将不时发生问题。大量部署的微服务将比之前部署的单体应用出现问题的可能更高。

3. 问题

微服务模型将导致一些重要的运行时相关的问题：

(1) 微服务如何配置以及是否正确？
针对少量的应用程序，处理配置不是主要的问题，如使用配置文件或者在数据库中的配置表。在大量的微服务部署到大量的服务器上时，这一配置访问将变得非常复杂。这将导致大量的小的配置文件或者数据配置表遍布整个系统，使得难以高效可靠地维护。

(2) 什么微服务部署了，部署在哪里了？
在只有少量服务部署时，管理部署的主机和端口还是比较简单的。但是当有大量的微服务部署时，这些服务或多或少需要持续变更，如手工维护将变得非常麻烦。

(3) 如何维护路由信息？
在动态系统范围中，服务消费方也是一个挑战。尤其是路由表或者消费配置文件，需要手工更新。在微服务不断新增新的主机/端口时，将没有时间手工维护。交付时间将会延长，并且手工维护出错的风险也会增加。

(4) 如何防止失败链？
由于微服务之间是相互连接的，需要关注系统范围内的失败链。例如，一个被其他众多微服务依赖的微服务失败了，其他依赖的微服务也可能开始失败。如果没有合适处理，大量微服务将受到这个单一失败的微服务所影响，导致一个脆弱的系统。

(5) 如何验证所有的微服务已上线且在运行中？
跟踪少量应用的运行状态是比较简单的，但是如何验证所有的微服务是健康的，且准备好接收请求？

(6) 如何跟踪服务之间的消息？
如何应对组织开始接到关于一些流程执行失败？什么微服务到导致这一问题的根本原因？例如，订单 12345 卡住了，我们如何知道是因为微服务 A 无法访问，还是因为微服务 B 在发送一个订单确认消息之前，需要手工批准。

(7) 如何确保仅仅 API 服务暴露给外部？

例如我们如何避免外部未授权的请求，对内部微服务的访问？

(8) 如何保证 API 服务的安全？
这不是针对微服务的特定问题，但是保护对外暴露的微服务仍然是非常重要的。

4. 需要的组件

为了解决上述的一些问题，新的操作和管理功能是必须的。针对上述问题，建议的解决方案包括如下组件：

(1) 中心配置服务 Central Configuration Server
我们需要一个中心配置管理，而不是针对每一个部署单元（微服务）有一个本地配置。此外，我们还需一个配置 API，微服务用来获取配置信息。

(2) 服务发现服务 Service Discovery Server
我们需要服务发现功能，微服务在启动时，通过 API 自己注册，而不是手工跟踪微服务部署的主机和端口。

(3) 动态路由和负载均衡 Dynamic Routing and Load Balancer
基于服务发现功能，路由组件使用 discovery API 查询请求的微服务部署在哪里；在被请求的服务部署了多个实例的情况下，负载均衡组件可以决定路由请求到特定的实例。

(4) 电路断路器 Circuit Breaker
为了避免失败链问题，我们需要营养 Circuit Breaker 模式，详细信息可以参考 Fowler-Circuit Breaker 的文章。

(5) 监控 Monitoring
基于电路断路器，我们可以监控微服务的状态，同时收集运行时统计数据，获知服务的健康状态和当前使用率。这些信息可以收集并显示在 dashboard 上，并针对配置阈值设置自动报警。

(6) 中心日志分析 Centralized Log Analysis
为了跟踪消息，并检测微服务何时故障，我们需要一个中心日志分析功能，可以访问服务器并收集每一个微服务的 log 文件。日志分析功能保存 log 信息在中心数据库中，并提供了查询和 dashboard 功能。备注：为了查找相关的消息，所有微服务需要在 log 消息中使用相关的 id，这点很重要。

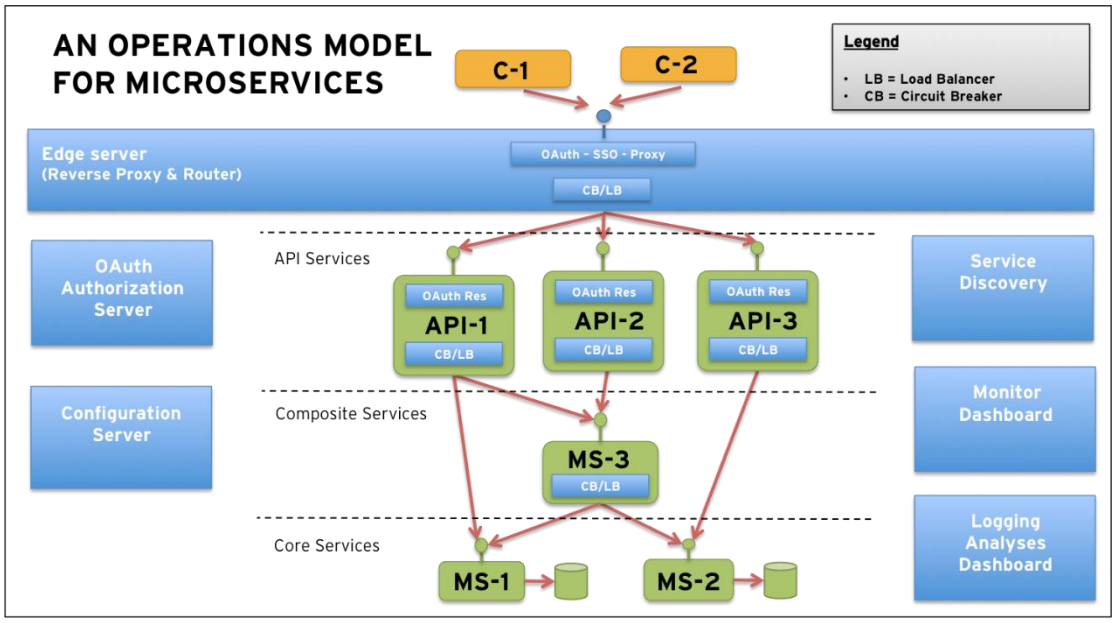
(7) 边缘服务 Edge Server
为了对外部暴露 API 服务，并阻止对内部微服务的未授权访问，我们需要一个边缘服务（Edge Server），所有外部的访问都经过边缘服务器。基于前面的服务发现组件，边缘服务器可以重用动态路由和负载均衡功能。边缘服务器作为一个动态和有效的反向代理，在内部系统更新时，不必手动更新。

(8) OAuth 2.0 保护的 API
建议 OAuth 2.0 标准保护暴露的 API 服务。应用 OAuth 2.0 有如下效果：
1/一个新组建作为 OAuth Authorization Server；
2/API 服务作为 OAuth Resource Server；
3/外部 API 消费方作为 OAuth Clients；

4/边缘服务器（Edge Server）作为 OAuth Token Relay；
（4.1）作为 OAuth Resource Server；
（4.2）将外部请求中的 OAuth Access Tokens 传递给 API 服务；
备注：OAuth 2.0 标准可能通过 OpenID Connect 标准来补充完善，提供更好的授权功能。

5. 参考模型

总而言之，微服务需要一套包含上述支持服务的基础设施，微服务使用它们的 API 来交互。下图描绘了这一基础设施：



备注：为了减少上图中交互的复杂度，微服务和支持服务的交互并没有画出来。

6. 下一步

在接下来的文章中，我们将描述和演示如何实现上述建议的参考模型。

二、基于 Spring Cloud 和 Netflix OSS 构建微服务，Part 1

前一篇文章中，我们定义了微服务使用的操作模型。这篇文章中，我们将开始使用 Spring Cloud 和 Netflix OSS 实现这一模型，包含核心部分：服务发现（Service Discovery）、动态路由（Dynamic Routing）、负载均衡（Load Balancing），和边缘服务器（Edge Server），其他部分在后面的文章中介绍。
我们将使用来自 Spring Cloud 和 Netflix OSS 的一些核心组件，实现在已部署的微服务交互，不必手动管理配置，如每一个微服务的端口或者手工配置路由规则等等。为了避免端口冲突，我们的微服务在启动时，将从端口段中动态获取可用的端口。为了方便访问微服务，我们将使用 Edge Server 提供一个微服务的访问入口点。
在简要介绍 Spring Cloud 和 Netflix OSS 组件之后，我们将描述本系列文章使用的系统，以及如何访问源代码，并编译。同时，也会简要指出源代码中的最重要部分。最后，我们将运行一些访问服务的测试代码，也会演示如何简单地创建一个新的服务实例，获取并使用负载均衡，所有这一些都不必手工配置。

1. Spring Cloud 和 Netflix OSS

Spring Cloud 是 spring.io 家庭的一个新项目，包含一系列组件，可用来实现我们的操作模型。很大程度上而言，Spring Cloud 1.0 是基于 Netflix OSS 组件。在 Spring 环境中，Spring Cloud 非常友好地集成了 Netflix 组件，使用了和 Spring Boot 相似的自动配置和惯例优于配置。
下表映射了操作模式中介绍的组件和我们将要使用的实际组件：

Operations Component	Netflix, Spring, ELK
Service Discovery server	Netflix Eureka
Dynamic Routing and Load Balancer	Netflix Ribbon
Circuit Breaker	Netflix Hystrix
Monitoring	Netflix Hystrix dashboard and Turbine
Edge Server	Netflix Zuul
Central Configuration server	Spring Cloud Config Server
OAuth 2.0 protected API's	Spring Cloud + Spring Security OAuth2
Centralised log analyses	Logstash, Elasticsearch, Kibana (ELK)

本文将包含 Eureka、Ribbon 和 Zuul：

1/Netflix Eureka – Service Discover Server 服务发现

Netflix Eureka 允许微服务在运行时自我注册

2/Netflix Ribbon-Dynamic Routing and Load Balancer 动态路由和负载均衡

Netflix Ribbon 可以在服务消费方运行时查询微服务。Ribbon 使用 Eureka 中的信息定位合适的服务实例。如果发现了多个服务实例，Ribbon 将应用负载均衡来转发请求到可用的微服务实例。**Ribbon 不作为一个单独的服务运行，而是嵌入在每一个服务消费方中**。

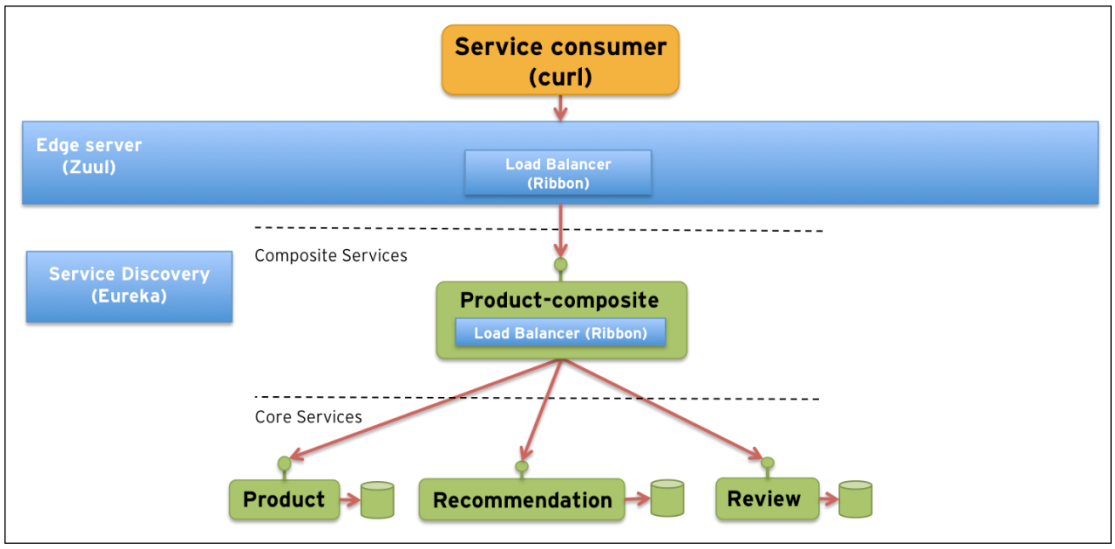
3/Netflix Zuul – Edge Server 边缘服务器

Zuul 是我们对外部世界的守门员，禁止任一未授权的外部请求进入。Zuul 在系统内部也提供了方便的进入入口点。通过使用动态分配的端口，可以避免端口冲突，以及最小化管理成本，但是也导致服务消费方更难接入。Zuul 使用 Ribbon 来查询可用的服务，并路由外部的请求到合适的服务实例。在本文中，我们将仅仅使用 Zuul 提供了便利的访问入口点，安全部分在下一篇文章中讨论。

备注：通过边缘服务器（Edge Server），可被外部访问的微服务，在系统中可称为 API。

2. 系统架构

为了测试这些组件，我们需要一个可实施的业务系统。本文的目标是开发实现如下系统：



上图包含 4 个业务服务（绿色文本框）：
1/ 三个核心服务负责处理信息：产品、推荐和评论；
2/ 一个组合服务 product-composite，用来聚合 3 个核心服务的信息，组合包含评论和推荐的产品信息视图；

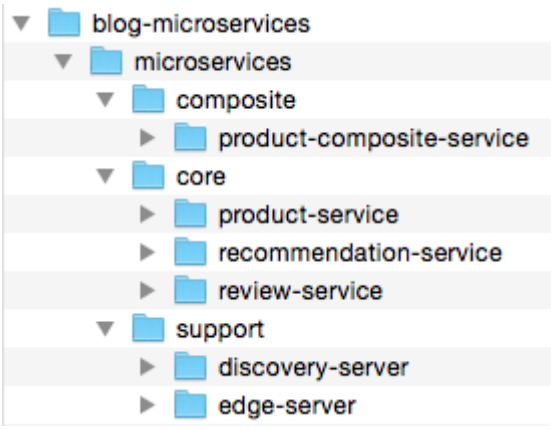
为了支持业务服务，我们使用了如下基础设施服务和组件（蓝色文本框）：
1/ 服务发现服务器（Service Discovery Server – Netflix Eureka）
2/ 动态路由和负载均衡（Netflix Ribbon）
3/ 边缘服务器（Edge Server – Netflix Zuul）

为了强调微服务和单体应用的差异，我们将每一个服务运行在单独的微服务进程中。在一个大系统中，如此细粒度的微服务可能并不方便。相应地，一组相关的微服务可能合并为一组，保持微服务的数量在可管理的水平，但这并不是退回到巨大的单体应用。

3. 获取源代码并编译

获取源代码，并进行测试，需要已安装 Java SE8 和 Git，接着执行如下操作：
$ git clone https://github.com/callistaenterprise/blog-microservices.git
$ cd blog-microservices
$ git checkout -b B1 M1.1

将生成如下的目录结构：



每一个组件独立编译（记住我们不再编译单体应用），因此每一个组件都有自己的 build 文件。我们使用 Gradle 编译系统，如果你没有安装 Gradle，build 文件将自动下载。为了简化编译过程，我们提供了一个小的 shell 脚本，可用来编译组件：
$ ./build-all.sh

如果在 Windows 环境下，你可以执行相应的 bat 文件 build-all.bat。
将显示 6 个 log 消息，并显示：BUILD SUCCESSFUL

4. 阅读源代码

快速看看关键的源代码，每一个微服务开发为一个独立的 Spring Boot 应用，并使用 Undertow（一个轻量级的 Servlet 3.1 容器）作为 web server。Spring MVC 用来实现 REST-based 服务，Spring RestTemplate 用来执行外部调用。如果你想更多地了解这些核心技术，你可以查看相关的文章。

这里，我们关注如何使用 Spring Cloud 和 Netflix OSS 功能。
备注：为了让源码易于理解，我们特意让实现尽量简单。

Gradle 依赖

本着 Spring Boot 的精髓，Spring Cloud 定义了一组 starter 依赖，便于引入需要的特定依赖。为了在微服务中使用 Eureka 和 Ribbon，以及方便调用其他微服务，在 build 文件中添加如下：

```
compile("org.springframework.cloud:spring-cloud-starter-eureka:1.0.0.RELEASE")
```

可以查看 product-service/build.gradle 获取完整的例子。

为了搭建 Eureka 服务器，添加如下依赖：

```
compile('org.springframework.cloud:spring-cloud-starter-eureka-server:1.0.0.RELEASE')
```

完整的例子，可以查看 discovery-server/build.gradle。

4.2 基础设施服务器

基于 Spring Cloud 和 Netflix OSS 搭建基础设施服务器相当方便。例如，在一个标准的 Spring Boot 应用中，添加@EnableEurekaServer 标注来搭建 Eureka 服务器。

```
@SpringBootApplication
@EnableEurekaServer
public class EurekaApplication {

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

完整的实例，可以查看 EurekaApplication.java 代码。

搭建 Zuul 服务器，可以添加@EnableZuulProxy 标注。完整的实例，可以查看 ZuulApplication.java 代码。

通过这些简单的标注，可以搭建一个默认的服务器配置。根据需要，也可以通过特定的设置覆盖默认配置。例如，我们可以通过覆盖默认的配置，限制边缘服务器允许路由调用的微服务。默认情况下，Zuul 搭建了 Eureka 中可以发现的每一个微服务的路由。通过如下的 application.yml 配置，限制了只允许访问组合服务-product service 的路由。

```
zuul:
  ignoredServices: "*"
  routes:
    productcomposite:
      path: /productcomposite/**
```

查看 edge-server/application.yml 获取完整的例子。

4.3 业务服务

通过在 Spring Boot 应用中，添加@EnableDiscoveryClient 标注，自动注册微服务到 Eureka Server 中。

```
@SpringBootApplication
@EnableDiscoveryClient
public class ProductServiceApplication {

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

完整的例子，可以查看 ProductServiceApplication.java。

为了查询和调用微服务实例，可以使用 Ribbon 和 Spring RestTemplate，如下所示：

```
@Autowired
    private LoadBalancerClient loadBalancer;
    ...
    public ResponseEntity<List<Recommendation>> getReviews(int productId) {

            ServiceInstance instance = loadBalancer.choose("review");
            URI uri = instance.getUri();
        ...
            response = restTemplate.getForEntity(url, String.class);
```

服务消费方只需要知道服务的名字，如上述例子中的 review，Ribbon（LoadBalancerClient 类）将发现服务实例，并返回 URI 给服务消费方。

5. 启动系统

在本文中，我们将在本地开发环境中作为一个 java 进程来启动微服务。在接下来的文章中，我们将描述如何部署微服务到云环境和 Docker 容器中。

为了运行下面的一些命令，需要安装 curl 和 jq 工具。
每一个微服务使用命令 ./gradlew bootRun 来启动。

首先，启动微服务基础设施，如：

```
$ cd .../blog-microservices/microservices

$ cd support/discovery-server;    ./gradlew bootRun
$ cd support/edge-server;          ./gradlew bootRun
```

一旦启动了上述基础设施微服务，接着启动业务微服务：

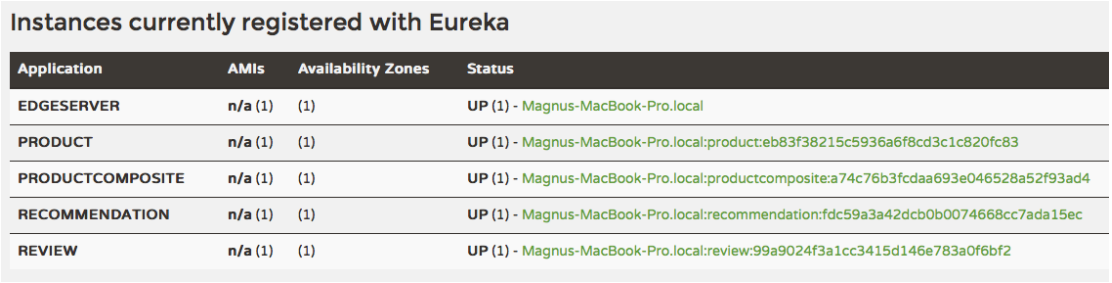
```
$ cd core/product-service;                    ./gradlew bootRun
$ cd core/recommendation-service;          ./gradlew bootRun
$ cd core/review-service;                    ./gradlew bootRun
$ cd composite/product-composite-service; ./gradlew bootRun
```

如果在 Windows 环境下，可以运行相应的 bat 文件，start-all.bat 文件。
一旦微服务启动了，将自注册到服务发现服务器（Service Discovery Server - Eureka）中去，并输出如下日志：

```
DiscoveryClient ... - registration status: 204
```

在服务发现 web 应用中，可以看到如下 4 个业务服务，和边缘服务器（Edge Server）

(http://localhost:8761)：



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EDGESERVER	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local
PRODUCT	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:product:eb83f38215c5936a6f8cd3c1c820fc83
PRODUCTCOMPOSITE	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:productcomposite:a74c76b3fcdaa693e046528a52f93ad4
RECOMMENDATION	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:recommendation:fdc59a3a42dcb0b0074668cc7ada15ec
REVIEW	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:review:99a9024f3a1cc3415d146e783a0f6bf2

为了了解上述服务的更多信息，如使用的 ip 地址和端口，可使用 Eureka REST API：

```
$ curl -s -H "Accept: application/json" http://localhost:8761/eureka/apps | jq '.applications.application[] | {service: .name, ip: .instance.ipAddr, port: .instance.port."$"}'
{
  "service": "PRODUCT",
  "ip": "192.168.0.116",
  "port": "59745"
}
{
  "service": "REVIEW",
  "ip": "192.168.0.116",
  "port": "59178"
}
{
  "service": "RECOMMENDATION",
  "ip": "192.168.0.116",
  "port": "48014"
}
{
  "service": "PRODUCTCOMPOSITE",
  "ip": "192.168.0.116",
  "port": "51658"
}
{
  "service": "EDGESERVER",
  "ip": "192.168.0.116",
  "port": "8765"
}
```

现在，我们已经准备好进行测试了。首先，验证可以到达我们的微服务，接着，我们创建一个新的微服务实例，并通过 Ribbon 在多个服务实例上实施负载均衡。

备注：在接下来的文章中，我们也会尝试失败的场景，演示电路断路器（Circuit Breaker）是如何工作的。

5.1 开始测试

通过边缘服务器来调用组合服务，边缘服务器在端口 8765（查看 application.yml 文件）。我们通过边缘服务器，以及路径/productcomposite/** 可到达 productcomposite 服务。返回的组合响应如下：

```
$ curl -s localhost:8765/productcomposite/product/1 | jq .
{
    "name": "name",
    "productId": 1,
    "recommendations": [
        {
            "author": "Author 1",
            "rate": 1,
            "recommendationId": 1
        },
        ...
    ],
    "reviews": [
        {
            "author": "Author 1",
            "reviewId": 1,
            "subject": "Subject 1"
        },
        ...
    ],
    "weight": 123
}
```

如果在微服务内部，我们实际上可以直接调用微服务，不必通过边缘服务器。当然，问题是我们不知道服务运行在什么端口，因为服务是动态分配的。但是，我们可以查看调用 Eureka REST API 的输出，就知道服务监听的端口了。我们可以使用如下的命令调用 3 个核心服务（端口号采用 Eureka REST API 输出的端口信息）：

```
$ curl -s localhost:51658/product/1 | jq .
$ curl -s localhost:59745/product/1 | jq .
$ curl -s localhost:59178/review?productId=1 | jq .
$ curl -s localhost:48014/recommendation?productId=1 | jq .
```

在自己的环境中，使用相应的端口号。

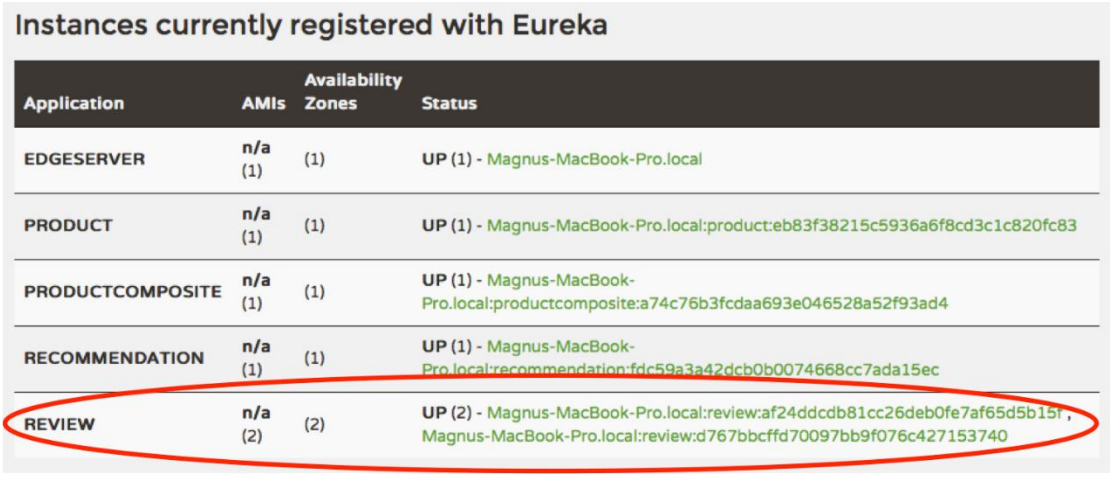
5.2 动态负载均衡

为了避免服务故障或者临时的网络问题，通常需要多个服务实例，通过负载均衡分发请求。因为我们使用动态分配的端口和服务发现 Server，可以非常容易添加新的服务实例。例如，

可以简单启动一个新的 review 服务，动态分配一个新的端口，并自我注册到服务发现服务器（Service Discovery Server）中。

```
$ cd .../blog-microservices/microservices/core/review-service
$ ./gradlew bootRun
```

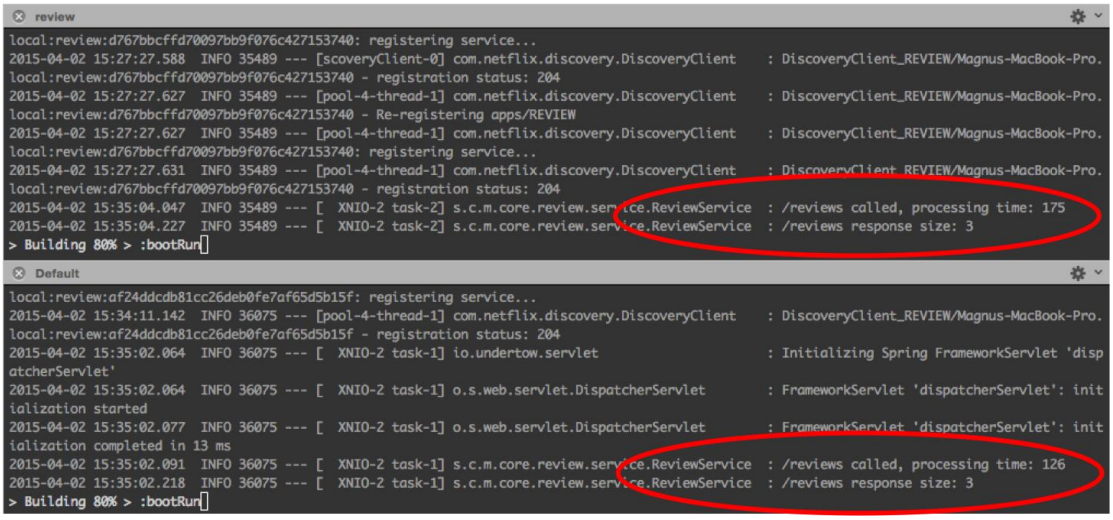
稍等片刻，第二个服务实例出现在服务发现 web 应用中（http://localhost:8761）：



Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EDGESERVER	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local
PRODUCT	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:product:eb83f38215c5936a6f8cd3c1c820fc83
PRODUCTCOMPOSITE	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:productcomposite:a74c76b3fcdaa693e046528a52f93ad4
RECOMMENDATION	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:recommendation:fdc59a3a42dcb0b0074668cc7ada15ec
REVIEW	n/a (2)	(2)	UP (2) - Magnus-MacBook-Pro.local:review:af24ddcdb81cc26deb0fe7af65d5b15f, Magnus-MacBook-Pro.local:review:d767bbcffd70097bb9f076c427153740

如果你运行之前的 curl 命令多次（curl -s localhost:8765/productcomposite/product/1 | jq .），查看 2 个 review 实例的 log 日志，可以发现负载均衡在 2 个实例之间自动处理调用请求，不必手工配置。



```
review
local:review:d767bbcffd70097bb9f076c427153740: registering service...
2015-04-02 15:27:27.588  INFO 35489 --- [scoveryClient-0] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740 - registration status: 204
2015-04-02 15:27:27.627  INFO 35489 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740 - Re-registering apps/REVIEW
2015-04-02 15:27:27.627  INFO 35489 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740: registering service...
2015-04-02 15:27:27.631  INFO 35489 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:d767bbcffd70097bb9f076c427153740 - registration status: 204
2015-04-02 15:35:04.047  INFO 35489 --- [  XNIO-2 task-2] s.c.m.core.review.service.ReviewService  : /reviews called, processing time: 175
2015-04-02 15:35:04.227  INFO 35489 --- [  XNIO-2 task-2] s.c.m.core.review.service.ReviewService  : /reviews response size: 3
> Building 80% > :bootRun
Default
local:review:af24ddcdb81cc26deb0fe7af65d5b15f: registering service...
2015-04-02 15:34:11.142  INFO 36075 --- [pool-4-thread-1] com.netflix.discovery.DiscoveryClient    : DiscoveryClient_REVIEW/Magnus-MacBook-Pro.
local:review:af24ddcdb81cc26deb0fe7af65d5b15f - registration status: 204
2015-04-02 15:35:02.064  INFO 36075 --- [  XNIO-2 task-1] io.undertow.servlet                       : Initializing Spring FrameworkServlet 'disp
atcherServlet'
2015-04-02 15:35:02.064  INFO 36075 --- [  XNIO-2 task-1] o.s.web.servlet.DispatcherServlet         : FrameworkServlet 'dispatcherServlet': init
ialization started
2015-04-02 15:35:02.077  INFO 36075 --- [  XNIO-2 task-1] o.s.web.servlet.DispatcherServlet         : FrameworkServlet 'dispatcherServlet': init
ialization completed in 13 ms
2015-04-02 15:35:02.091  INFO 36075 --- [  XNIO-2 task-1] s.c.m.core.review.service.ReviewService  : /reviews called, processing time: 126
2015-04-02 15:35:02.218  INFO 36075 --- [  XNIO-2 task-1] s.c.m.core.review.service.ReviewService  : /reviews response size: 3
> Building 80% > :bootRun
```

6. 总结

我们已经了解到 Spring Cloud 和 Netflix OSS 组件是如何用来简化独立部署微服务协同工作的，不必人工管理每一个微服务的端口，或者人工配置路由规则。当新的实例启动之后，它们会自动被服务发现 Server 监测到，并通过负载均衡来接收请求。通过使用边缘服务器

（Edge Server），我们可以控制什么微服务暴露给外部消费方，建立系统的 API。

7. 下一步

OK，完成测试之后。接下来还有一些问题没有回答，例如：
1/ 发生故障将如何处理，如出现一个失败的微服务；
2/ 如何阻止对 API 的未授权访问；
3/ 如何获知微服务内部的运行图，例如为什么订单#123456 没有交付？

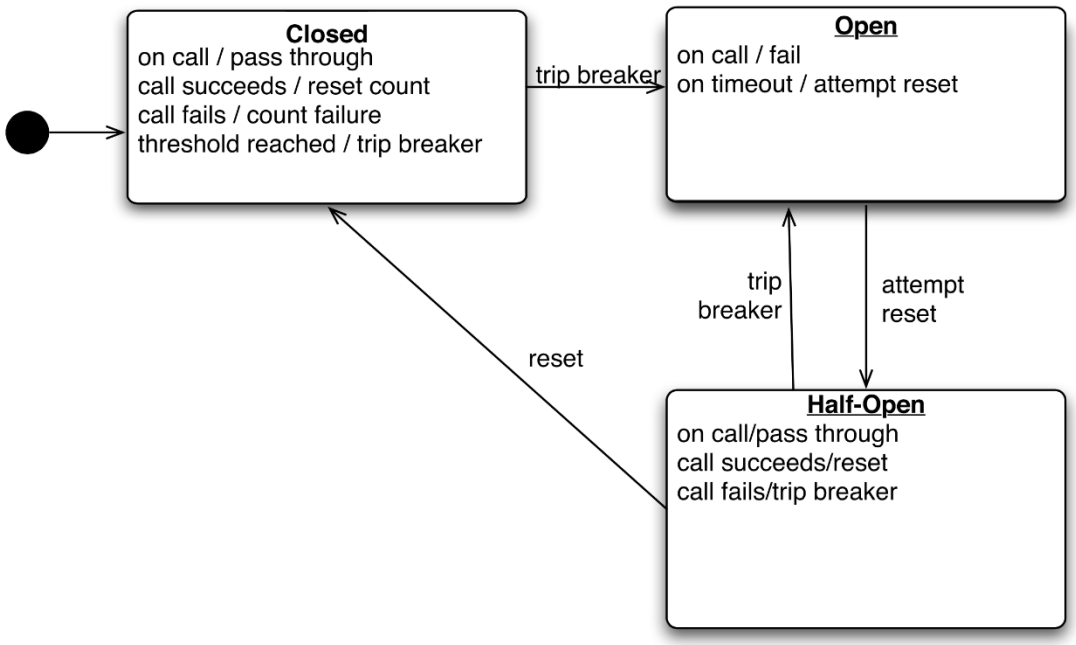
在接下来的文章中，我们将了解如何使用电路断路器（Circuit Breaker）来提升服务弹性，使用 OAuth 2 限制外部访问等等。也将了解如何使用 ELK 技术栈来收集所有微服务的日志，并呈现日志信息。

三、基于 Spring Cloud 和 Netflix OSS 构建微服务，Part 2

在上一篇文章中，我们已使用 Spring Cloud 和 Netflix OSS 中的核心组件，如 Eureka、Ribbon 和 Zuul，部分实现了操作模型（operations model），允许单独部署的微服务相互通信。在本文中，我们继续关注微服务环境中的故障处理，通过 Hystrix（Netflix Circuit Breaker）提升服务弹性。

现在我们建立的系统开始出现故障，组合服务（composite service）依赖的部分核心服务突然没有反应，如果故障没有正确处理，将进一步损害组合服务。

通常，我们将这一类问题称为失败链（a chain of failures），一个组件中的错误将导致依赖于错误组件中的其他组件也产生错误。在基于微服务的系统中，尤其是大量独立部署的微服务相互通信，这一情况需要特别关注。针对这一问题的通用解决方案是应用电路断路器模式（circuit breaker pattern），详细信息可以查阅其他文档，或者阅读 Fowler-Circuit Breaker 的文章。一个典型电路断路器应用如下状态转换图：



(Source: Release It!) https://pragprog.com/book/mnee/release-it

1. Spring Cloud 和 Netflix OSS

如下表所示，本文将包含：Hystrix、Hystrix dashboard 和 Turbine。

Operations Component	Netflix, Spring, ELK
Service Discovery server	Netflix Eureka
Dynamic Routing and Load Balancer	Netflix Ribbon
Circuit Breaker	Netflix Hystrix
Monitoring	Netflix Hystrix dashboard and Turbine
Edge Server	Netflix Zuul
Central Configuration server	Spring Cloud Config Server
OAuth 2.0 protected API's	Spring Cloud + Spring Security OAuth2
Centralised log analyses	Logstash, Elasticsearch, Kibana (ELK)

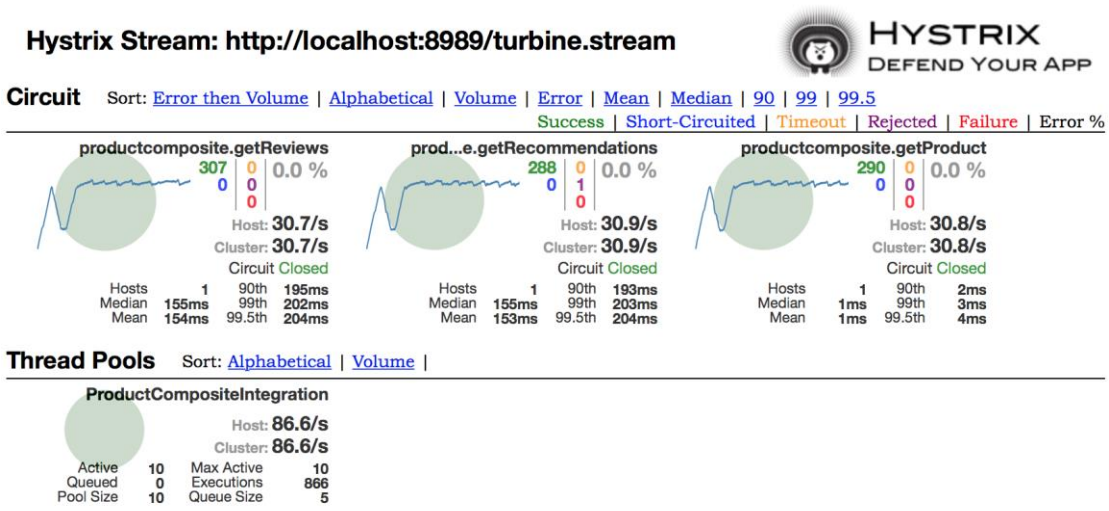
1/ Netflix Hystrix – 电路断路器（Circuit Breaker）

Netflix Hystrix 对微服务消费方提供了电路断路器功能。如果一个服务没有响应（如超时或者网络连接故障），Hystrix 可以在服务消费方中重定向请求到回退方法（fallback method）。如果服务重复失败，Hystrix 会打开电路，并快速失败（如直接调用内部的回退方法，不再尝试调用服务），直到服务重新恢复正常。

为了验证是否服务再次恢复正常，即使在电路打开的情况下，Hystrix 也会允许一部分请求再次调用微服务。Hystrix 是嵌入在服务调用方内部执行的。

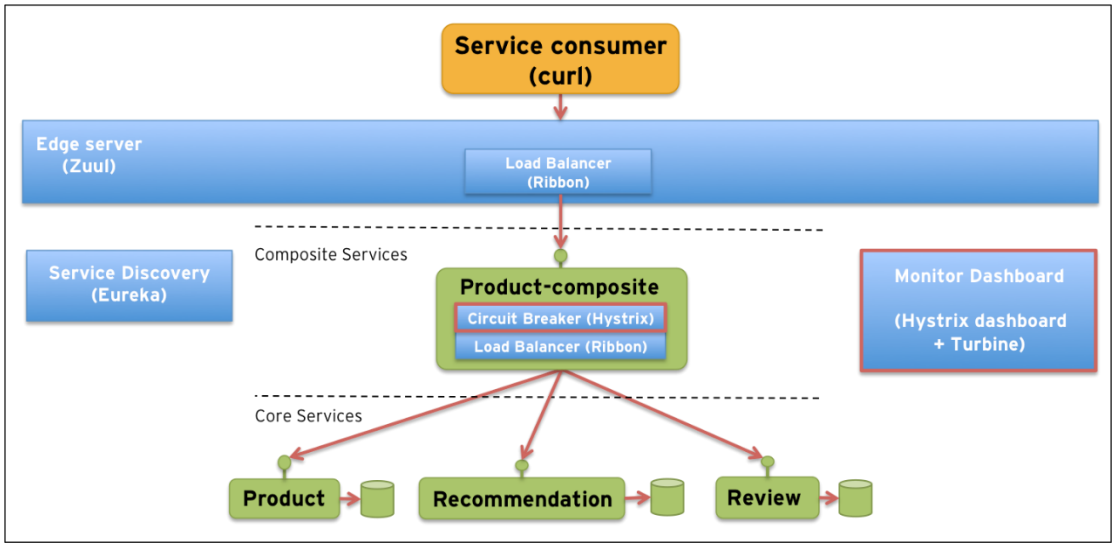
2/ Netflix Hystrix dashboard 和 Netflix Turbine –监控仪表盘（Monitor Dashboard）

Hystrix 仪表盘用来提供电路断路器的图形化视图；Turbine 基于 Eureka 服务器的信息，获取系统中所有电路断路器的信息，提供给仪表盘。下图是 Hystrix 仪表盘和 Turbine 工作视图：



2. 系统全貌

将前一部分 Part 1 实现的微服务系统，进一步添加支持性的基础服务-Hystrix Dashboard 和 Turbine。另外，微服务 product-composite 也增强了基于 Hystrix 的电路断路器。新增的 2 个组件标识为红色外框，如下图所示：



在 Part 1 中，我们重点强调了微服务和单体应用的差异，将每一个微服务独立部署运行（独立进程）

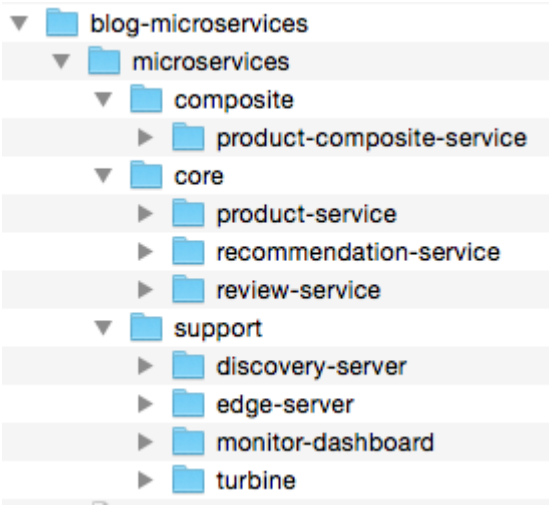
3. 构建源代码

在 Part 1 中，我们使用 Java SE 8，Git 和 Gradle 工具。接下来访问源代码，并进行编译：

```
$ git clone https://github.com/callistaenterprise/blog-microservices.git
$ cd blog-microservices
$ git checkout -b B2 M2.1
$ ./build-all.sh
```

如果运行在 windows 平台，请执行相应的 bat 文件 – build-all.bat。

在 Part 1 源码的基础上，新增了 2 个源码组件-monitor-dashboard 和 turbine：



编译输出 8 条 log 日志：
BUILD SUCCESSFUL

4. 阅读源代码

和 Part 1 源代码进行比较，本文在微服务 product-composite 中新增了 Hystrix 电路断路器的使用。因此，我们将关注电路断路器部分新增的额外代码。

4.1 Gradle 依赖

现在，我们在 build 文件中加入了几个 Hystrix 相关的 starter 依赖。因为 Hystrix 使用 RabbitMQ 消息中间件在电路断路器和仪表盘（dashboard）之间通信，因此我们也需要添加相应的依赖。

对于服务消费方，如需要使用 Hystrix 作为电路断路器，则需要添加如下依赖配置：

```
compile("org.springframework.cloud:spring-cloud-starter-hystrix:1.0.0.RELEASE")
compile("org.springframework.cloud:spring-cloud-starter-bus-amqp:1.0.0.RELEASE")
compile("org.springframework.cloud:spring-cloud-netflix-hystrix-amqp:1.0.0.RELEASE")
```

更完整的示例，可以查看 product-composite-service/build.gradle 文件。

为了搭建 Turbine 服务器，需要添加如下依赖：

```
compile('org.springframework.cloud:spring-cloud-starter-turbine-amqp:1.0.0.RELEASE')
```

更完整的示例，可以查看 turbine/build.gradle 文件。

4.2 基础设施服务器

在标准的 Spring Boot 应用中，添加@EnableTurbineAmqp 标注，就可以搭建 Turbine 服务器了。

```
@SpringBootApplication
@EnableTurbineAmqp
@EnableDiscoveryClient
public class TurbineApplication {

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

}
```

完整的示例，可以查看 TurbineApplication.java 文件。

搭建 Hystrix 仪表盘，则需要添加@EnableHystrixDashboard 标注。完整的示例，可以查看 HystrixDashboardApplication.java 文件。

通过上述简单的标注，就可以获得默认服务器配置了。可根据需要使用特定的配置，覆盖默认的配置。

4.3 业务服务

为了启用 Hystrix, 需要在 Spring Boot 应用中添加@EnableCircuitBreaker 标注。为了让 Hystrix 真实地生效，还需要在 Hystrix 监控的方法上标注@HystrixCommand，在这个标注中，还可以指定回退方法（fallback method），如下所示：

```
@HystrixCommand(fallbackMethod = "defaultReviews")
public ResponseEntity<List<Review>> getReviews(int productId) {
    ...
}

public ResponseEntity<List<Review>> defaultReviews(int productId) {
    ...
}
```

在发生错误的时候（如调用服务失败或者超时），Hystrix 会调用回退方法；或者在电路打开的时候，进行快速失败处理。完整的示例，可以查看 ProductCompositeIntegration.java 文件。

5. 启动系统

如前所述，Hystrix 通过 RabbitMQ 消息中间件进行内部通信，因此我们在启动微服务系统之前，需要先安装并运行 RabbitMQ。可以访问如下链接，了解 RabbitMQ 安装教程：
https://www.rabbitmq.com/download.html
安装完成之后，接着启动 RabbitMQ，通过运行 RabbitMQ 安装目录下的 sbin 子目录中的 rabbitmq-server 程序进行启动。

$ ~/Applications/rabbitmq_server-3.4.3/sbin/rabbitmq-server

```
                RabbitMQ 3.4.3. Copyright (C) 2007-2014 GoPivotal, Inc.
  ##  ##        Licensed under the MPL.  See http://www.rabbitmq.com/
  ##  ##
  ##########                Logs:      /Users/magnus/Applications/rabbitmq_server-
3.4.3/sbin/../var/log/rabbitmq/rabbit@Magnus-MacBook-Pro.log
  ######  ##                          /Users/magnus/Applications/rabbitmq_server-
3.4.3/sbin/../var/log/rabbitmq/rabbit@Magnus-MacBook-Pro-sasl.log
  ##########
                Starting broker... completed with 6 plugins.
```

如在 windows 系统中，确保 RabbitMQ 服务已经启动。

现在我们准备好启动系统了。使用./gradlew 命令启动每一个微服务。

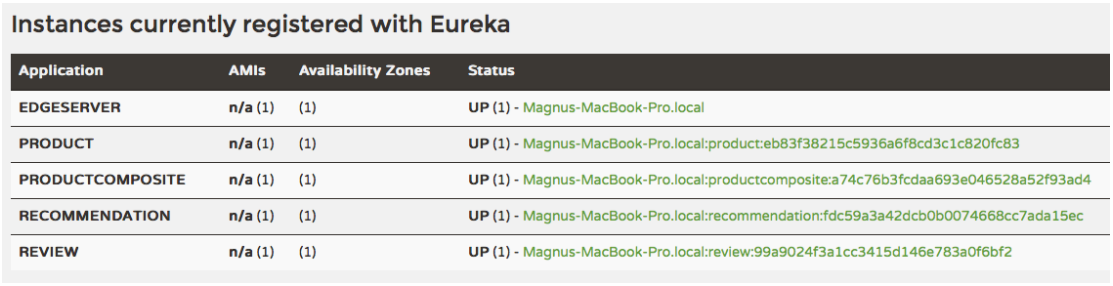
首先启动基础设施微服务：
$ cd support/discovery-server; ./gradlew bootRun
$ cd support/edge-server; ./gradlew bootRun
$ cd support/monitor-dashboard; ./gradlew bootRun
$ cd support/turbine; ./gradlew bootRun

一旦上述服务启动完成之后，接着启动业务微服务：
$ cd core/product-service; ./gradlew bootRun
$ cd core/recommendation-service; ./gradlew bootRun
$ cd core/review-service; ./gradlew bootRun
$ cd composite/product-composite-service; ./gradlew bootRun

如在 windows 平台，可以执行相应的 bat 文件 – start-all.bat。

一旦微服务启动完成，并注册到服务发现服务器（Service Discovery Server），将同时输出如下日志：
DiscoveryClient ... - registration status: 204

和 Part 1 一样，我们可以在服务发现 Web 应用中看到如下 4 个业务服务和一个 edge-server，如下所示(http://localhost:8761)：



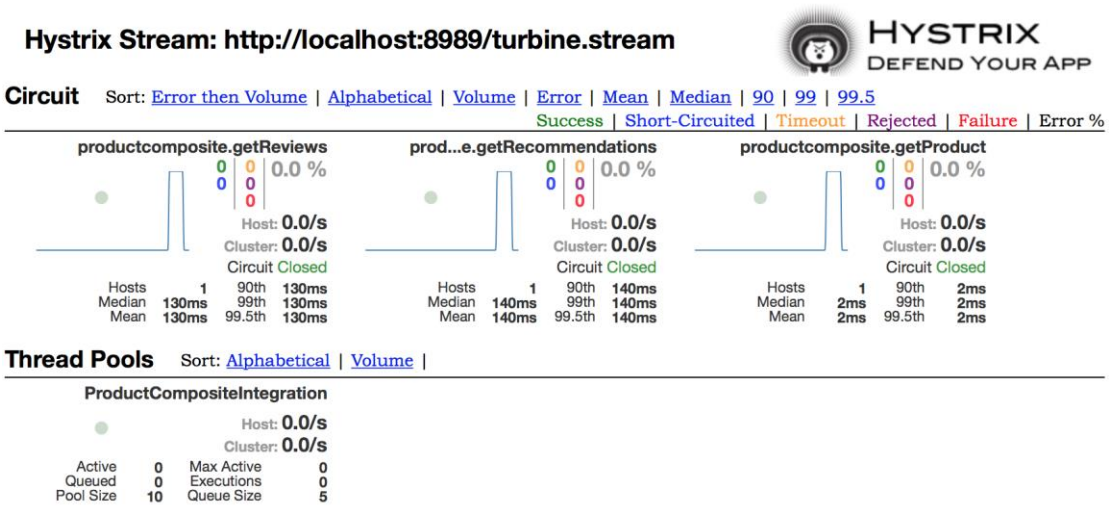
Instances currently registered with Eureka

Application	AMIs	Availability Zones	Status
EDGESERVER	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local
PRODUCT	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:product:eb83f38215c5936a6f8cd3c1c820fc83
PRODUCTCOMPOSITE	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:productcomposite:a74c76b3fcdaa693e046528a52f93ad4
RECOMMENDATION	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:recommendation:fdc59a3a42dcb0b0074668cc7ada15ec
REVIEW	n/a (1)	(1)	UP (1) - Magnus-MacBook-Pro.local:review:99a9024f3a1cc3415d146e783a0f6bf2

最后，验证电路断路器工作正常。在处于 closed 状态时，通过 edge-server 访问组合服务（composite service），输出响应结果如下：

```
$ curl -s localhost:8765/productcomposite/product/1 | jq .
{
    "name": "name",
    "productId": 1,
    "recommendations": [
        {
            "author": "Author 1",
            "rate": 1,
            "recommendationId": 0
        },
        ...
    ],
    "reviews": [
        {
            "author": "Author 1",
            "reviewId": 1,
            "subject": "Subject 1"
        },
        ...
    ],
    "weight": 123
}
```

在浏览器中首先访问 http://localhost:7979 地址（Hystrix Dashboard），接着在文本框中输入 http://localhost:8989/turbine.stream，并点击 Monitor Stream 按钮：



我们看到组合服务有 3 个电路断路器正在运行中，分别是 3 个依赖的核心服务。目前都工作正常。接着，我们准备尝试故障测试，验证电路断路器发挥作用。

6. 发生故障

停止 review 微服务，再次尝试之前的命令：

```
$ curl -s localhost:8765/productcomposite/product/1 | jq .
{
    "name": "name",
    "productId": 1,
    "recommendations": [
        {
            "author": "Author 1",
            "rate": 1,
            "recommendationId": 0
        },
        ...
    ],
    "reviews": null,
    "weight": 123
}
```

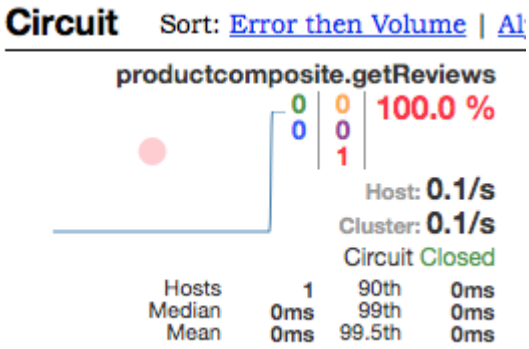
返回的响应报文中 review 部分是空的，但是其余部分报文保持不变。查看 product-composite 服务日志，可以发现如下警告信息：

```
2015-04-02 15:13:36.344 INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration : GetRecommendations...
```

```
2015-04-02 15:13:36.497 INFO 29901 --- [telntegration-2]
s.c.m.c.p.s.ProductCompositeIntegration : GetReviews...
2015-04-02 15:13:36.498 WARN 29901 --- [telntegration-2]
s.c.m.composite.product.service.Util : Failed to resolve serviceId 'review'. Fallback to URL
'http://localhost:8081/review'.
2015-04-02 15:13:36.500 WARN 29901 --- [telntegration-2]
s.c.m.c.p.s.ProductCompositeIntegration : Using fallback method for review-service
```

电路断路器检测到 review 服务发生了故障，将请求路由到服务消费方的回退方法（fallback method）。在本示例中，我们只是简单地返回一个 null，但我们也可以返回一个本地缓存数据，以便在 review 服务发生故障时，提供更好的效果。

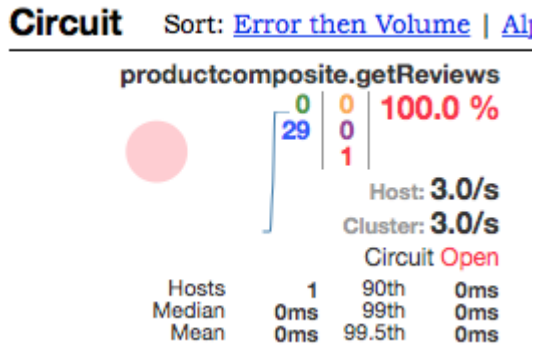
因为此时故障发生频率并不高，因此电路仍然是闭合状态（closed）：



我们接下来提高故障频率，并超出 Hystrix 打开电路的限制，开始快速失败。这里，我们使用 Apache HTTP server benchmarking tool 是实现这一目的：

```
ab -n 30 -c 5 localhost:8765/productcomposite/product/1
```

现在电路打开了：



随后的请求将快速失败，也就是说，电路断路器将直接转发请求到回退方法，不再调用 review 服务。此时，log 日志中将不再有 GetReviews 相关日志。

```
2015-04-02 15:14:03.930 INFO 29901 --- [telntegration-5]
s.c.m.c.p.s.ProductCompositeIntegration : GetRecommendations...
2015-04-02 15:14:03.984 WARN 29901 --- [ XNIO-2 task-62]
```

```
s.c.m.c.p.s.ProductCompositeIntegration   : Using fallback method for review-service
```

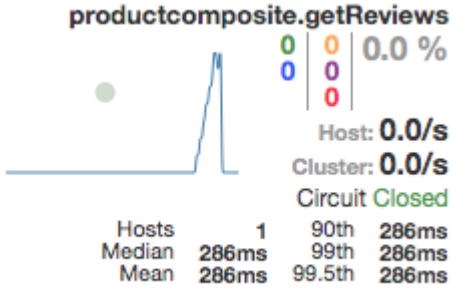
然而，Hystrix 不时地让一部分请求通过电路，查看是否可以调用成功，也就是检查 review 服务是否再次恢复正常。我们可以多次重复执行 curl 调用，查看 product-composite 服务的输出日志：

```
2015-04-02 15:17:33.587  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetRecommendations...
2015-04-02 15:17:33.769  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetReviews...
2015-04-02 15:17:33.769  WARN 29901 --- [eIntegration-10] s.c.m.composite.product.service.Util      : Failed to resolve serviceId 'review'. Fallback to URL 'http://localhost:8081/review'.
2015-04-02 15:17:33.770  WARN 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : Using fallback method for review-service
2015-04-02 15:17:34.431  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetRecommendations...
2015-04-02 15:17:34.569  WARN 29901 --- [ XNIO-2 task-18] s.c.m.c.p.s.ProductCompositeIntegration   : Using fallback method for review-service
2015-04-02 15:17:35.209  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetRecommendations...
2015-04-02 15:17:35.402  WARN 29901 --- [ XNIO-2 task-20] s.c.m.c.p.s.ProductCompositeIntegration   : Using fallback method for review-service
2015-04-02 15:17:36.043  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetRecommendations...
2015-04-02 15:17:36.192  WARN 29901 --- [ XNIO-2 task-21] s.c.m.c.p.s.ProductCompositeIntegration   : Using fallback method for review-service
2015-04-02 15:17:36.874  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetRecommendations...
2015-04-02 15:17:37.031  WARN 29901 --- [ XNIO-2 task-22] s.c.m.c.p.s.ProductCompositeIntegration   : Using fallback method for review-service
2015-04-02 15:17:41.148  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetRecommendations...
2015-04-02 15:17:41.340  INFO 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : GetReviews...
2015-04-02 15:17:41.340  WARN 29901 --- [eIntegration-10] s.c.m.composite.product.service.Util      : Failed to resolve serviceId 'review'. Fallback to URL 'http://localhost:8081/review'.
2015-04-02 15:17:41.341  WARN 29901 --- [eIntegration-10] s.c.m.c.p.s.ProductCompositeIntegration   : Using fallback method for review-service
```

从 log 日志的输出中，我们发现每 5 个调用，允许一次尝试调用 review 服务（仍然没有成功调用）。

现在，我们再次启动 review 服务，继续尝试调用组合服务 product-composite。

备注：此时，你可能需要一点耐心（最多 1 分钟）。在调用成功之前，需要服务发现服务器（Eureka）和动态路由（Ribbon）必须感知到 review 服务实例再次恢复可用。
现在，我们看到返回结果正常了，review 节点也恢复到返回报文中，电路也再次闭合（closed）：



7. 总结

我们已经看到了 Netflix Hystrix 如何用作电路断路器（Circuit Breaker）有效地处理失败链的问题。失败链是指：当单个微服务故障时，由于故障的扩散，会导致系统中大范围微服务故障事故。幸亏 Spring Cloud 框架提供的简单标注和 starter 依赖，可以非常容易在 Spring 环境中启用 Hystrix。最后，Hystrix dashboard 和 Turbine 提供的仪表盘（Dashboard）功能，使得监控系统范围内的大量电路断路器变得切实可行。

8. 接下来

在构建微服务的下一篇文章中，我们将学习如何使用 OAuth 2.0 来限制对暴露为外部 API 的微服务进行访问。

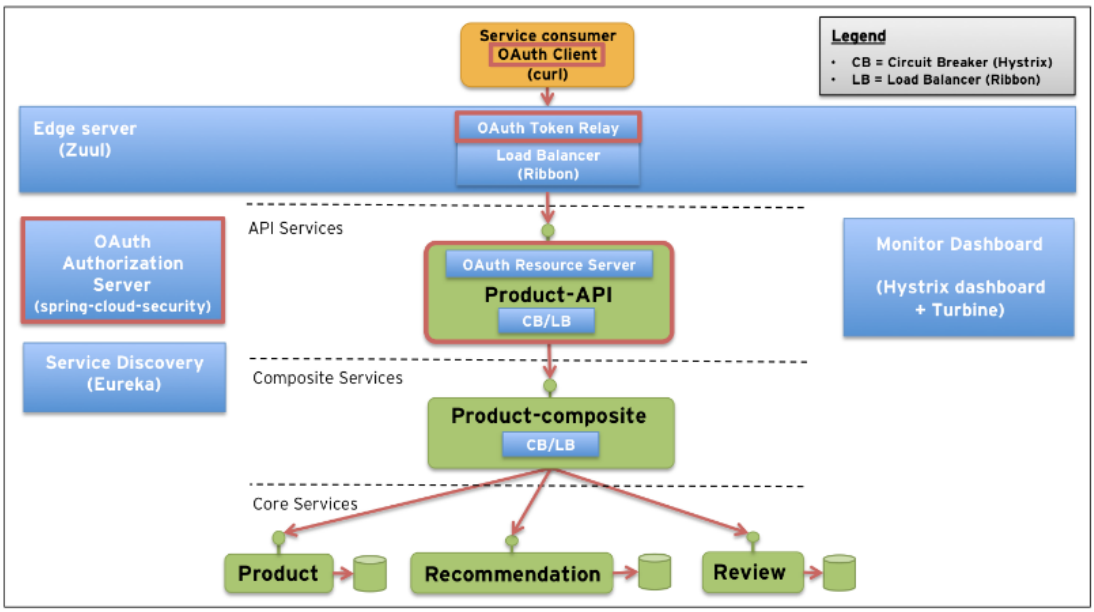
四、使用 OAuth 2.0 保护 API 接口

在本文中，我们将使用 OAuth 2.0，创建一个的安全 API，可供外部访问 Part 1 和 Part 2 完成的微服务。

关于 OAuth 2.0 的更多信息，可以访问介绍文档：Parecki - OAuth 2 Simplified 和 Jenkov - OAuth 2.0 Tutorial，或者规范文档 IETF RFC 6749。

我们将创建一个新的微服务，命名为 product-api，作为一个外部 API（OAuth 术语为资源服务器-Resource Server），并通过之前介绍过的 Edge Server 暴露为微服务，作为 Token Relay，也就是转发 Client 端的 OAuth 访问令牌到资源服务器（Resource Server）。另外添加 OAuth Authorization Server 和一个 OAuth Client，也就是服务消费方。

继续完善 Part 2 的系统全貌图，添加新的 OAuth 组件（标识为红色框）：



我们将演示 Client 端如何使用 4 种标准的授权流程，从授权服务器（Authorization Server）获取访问令牌（Access Token），接着使用访问令牌对资源服务器发起安全访问，如 API。

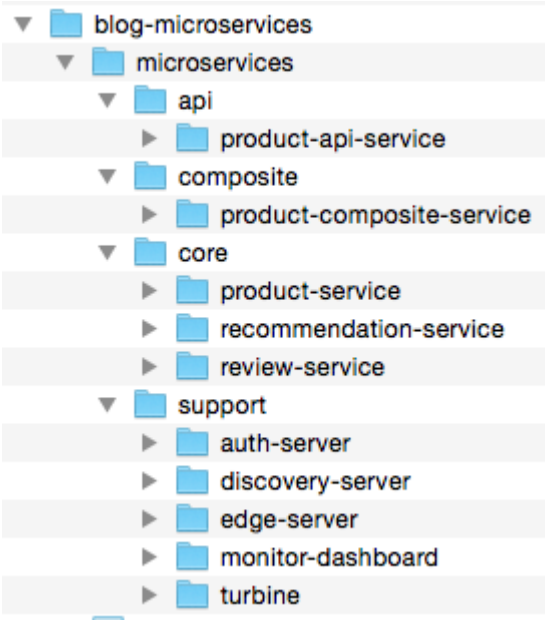
备注：
1/ 保护外部 API 并不是微服务的特殊需求，因此本文适用于任何使用 OAuth 2.0 保护外部 API 的架构；
2/ 我们使用的轻量级 OAuth 授权系统仅适用于开发和测试环境。在实际应用中，需要替换为一个 API 平台，或者委托给社交网络 Facebook 或 Twitter 的登录、授权流程。

3/ 为了降低复杂度，我们特意采用了 HTTP 协议。在实际的应用中，OAuth 通信需要使用 TLS，如 HTTPS 保护通信数据。
4/ 在前面的文章中，我们为了强调微服务和单体应用的差异性，每一个微服务单独运行在独立的进程中。

1. 编译源码

和在 Part 2 中一样，我们使用 Java SE 8、Git 和 Gradle 访问源代码，并进行编译：
git clone https://github.com/callistaenterprise/blog-microservices.git
cd blog-microservices
git checkout -b B3 M3.1
./build-all.sh
如果运行在 Windows 平台，则执行相应的 bat 文件-build-all.bat。

在 Part 2 的基础中，新增了 2 个组件源码，分别为 OAuth Authorization Server，项目名为 auth-server；另一个为 OAuth Resource Server，项目名为 product-api-service。



编译输出 10 条 log 消息：
BUILD SUCCESSFUL

2. 分析源代码

查看 2 个新组件是如何实现的，以及 Edge Server 是如何更新并支持传递 OAuth 访问令牌

的。我们也会修改 API 的 URL，以便于使用。

2.1 Gradle 依赖

为了使用 OAuth 2.0, 我们将引入开源项目 spring-cloud-security 和 spring-security-oauth2, 添加如下依赖。
auth-server 项目：

```
compile("org.springframework.boot:spring-boot-starter-security")
compile("org.springframework.security.oauth:spring-security-oauth2:2.0.6.RELEASE")
```

完整代码，可查看 auth-server/build.gradle 文件。

product-api-service 项目：

```
compile("org.springframework.cloud:spring-cloud-starter-security:1.0.0.RELEASE")
compile("org.springframework.security.oauth:spring-security-oauth2:2.0.6.RELEASE")
```

完整代码，可以查看 product-api-service/build.gradle 文件。

2.2 AUTH-SERVER

授权服务器（Authorization Server）的实现比较简单直接。可直接使用 @EnableAuthorizationServer 标注。接着使用一个配置类注册已批准的 Client 端应用，指定 client-id、client-secret、以及允许的授予流程和范围：

```
@EnableAuthorizationServer
protected static class OAuth2Config extends AuthorizationServerConfigurerAdapter {

    @Override
    public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
        clients.inMemory()
            .withClient("acme")
            .secret("acmesecret")
            .authorizedGrantTypes("authorization_code", "refresh_token", "implicit", "password",
"client_credentials")
            .scopes("webshop");
    }
}
```

显然这一方法仅适用于开发和测试场景模拟 Client 端应用的注册流程，实际应用中采用 OAuth Authorization Server，如 LinkedIn 或 GitHub。
完整的代码，可以查看 AuthserverApplication.java。
模拟真实环境中 Identity Provider 的用户注册（OAuth 术语称为 Resource Owner），通过在文件 application.properties 中，为每一个用户添加一行文本，如：

```
security.user.password=password
```

完整代码，可以查看 application.properties 文件。

实现代码也提供了 2 个简单的 web 用户界面，用于用户认证和用户准许，详细可以查看源代码：
https://github.com/callistaenterprise/blog-microservices/tree/B3/microservices/support/auth-server/src/main/resources/templates

2.3 PRODUCT-API-SERVICE

为了让 API 代码实现 OAuth Resource Server 的功能，我们只需要在 main 方法上添加 @EnableOAuth2Resource 标注：

```
@EnableOAuth2Resource
public class ProductApiServiceApplication {
```

完整代码，可以查看 ProductApiServiceApplication.java。

API 服务代码的实现和 Part 2 中的组合服务代码的实现很相似。为了验证 OAuth 工作正常，我们添加了 user-id 和 access token 的日志输出：

```
@RequestMapping("/{productId}")
    @HystrixCommand(fallbackMethod = "defaultProductComposite")
    public ResponseEntity<String> getProductComposite(
        @PathVariable int productId,
        @RequestHeader(value="Authorization") String authorizationHeader,
        Principal currentUser) {

        LOG.info("ProductApi: User={}, Auth={}, called with productId={}",
          currentUser.getName(), authorizationHeader, productId);
        ...
```

备注：
1/ Spring MVC 将自动填充额外的参数，如 current user 和 authorization header。
2/ 为了 URL 更简洁，我们从@RequestMapping 中移除了/product。当使用 Edge Server 时，它会自动添加一个/product 前缀，并将请求路由到正确的服务。
3/ 在实际的应用中，不建议在 log 中输出访问令牌（access token）。

2.4 更新 Edge Server

最后，我们需要让 Edge Server 转发 OAuth 访问令牌到 API 服务。非常幸运的是，这是默认的行为，我们不必做任何事情。
为了让 URL 更简洁，我们修改了 Part 2 中的路由配置：

```
zuul:
  ignoredServices: "*"
  prefix: /api
  routes:
    productapi: /product/**
```

这样，可以使用 URL：http://localhost:8765/api/product/123，而不必像前面使用的 URL：

http://localhost:8765/productapi/product/123。
我们也替换了到 composite-service 的路由为到 api-service 的路由。
完整的代码，可以查看 application.yml 文件。

3. 启动系统

首先启动 RabbitMQ：
$ ~/Applications/rabbitmq_server-3.4.3/sbin/rabbitmq-server
如在 Windows 平台，需要确认 RabbitMQ 服务已经启动。

接着启动基础设施微服务：
$ cd support/auth-server; ./gradlew bootRun
$ cd support/discovery-server; ./gradlew bootRun
$ cd support/edge-server; ./gradlew bootRun
$ cd support/monitor-dashboard; ./gradlew bootRun
$ cd support/turbine; ./gradlew bootRun

最后，启动业务微服务：
$ cd core/product-service; ./gradlew bootRun
$ cd core/recommendation-service; ./gradlew bootRun
$ cd core/review-service; ./gradlew bootRun
$ cd composite/product-composite-service; ./gradlew bootRun
$ cd api/product-api-service; ./gradlew bootRun
如在 Windows 平台，可以执行相应的 bat 文件-start-all.bat。

一旦微服务启动完成，并注册到服务发现服务器（Service Discovery Server），会输出如下日志：
DiscoveryClient ... - registration status: 204

现在已经准备好尝试获取访问令牌，并使用它安全地调用 API 接口。

4. 尝试 4 种 OAuth 授权流程

OAuth 2.0 规范定义了 4 种授予方式，获取访问令牌：

Grant Flow	Typical use case
Authorization Code	Used by a secure client like a web server
Implicit	Used by an client that can't protect a client secret or a refresh token, such as a mobile app or a HTML5 single page app
Resource Owner Password Credentials	Used when neither of the flows above works, e.g. if the user don't have access to a web browser
Client Credentials	Used if the client application does not need user consent to access a resource

更详细信息，可查看 Jenkov - OAuth 2.0 Authorization。

备注：Authorization Code 和 Implicit 是最常用的 2 种方式。如前面 2 种方式不使用，其他 2 种适用于一个特殊场景。
接下来看看每一个授予流程是如何获取访问令牌的。

4.1 授权代码许可（Authorization Code Grant）

首先，我们通过浏览器获取一个代码许可：
http://localhost:9999/uaa/oauth/authorize? response_type=code& client_id=acme& redirect_uri=http://example.com& scope=webshop& state=97536

先登录（user/password），接着重定向到类似如下 URL：
http://example.com/?
code=lyJh4Y&
state=97536
备注：在请求中 state 参数设置为一个随机值，在响应中进行检查，避免 cross-site request forgery 攻击。

从重定向的 URL 中获取 code 参数，并保存在环境变量中：
CODE=lyJh4Y

现在作为一个安全的 web 服务器，使用 code grant 获取访问令牌：

```
curl acme:acmesecret@localhost:9999/uaa/oauth/token \
  -d grant_type=authorization_code \
  -d client_id=acme \
  -d redirect_uri=http://example.com \
  -d code=$CODE -s | jq .
{
   "access_token": "eba6a974-3c33-48fb-9c2e-5978217ae727",
   "token_type": "bearer",
   "refresh_token": "0eebc878-145d-4df5-a1bc-69a7ef5a0bc3",
```

```
  "expires_in": 43105,
  "scope": "webshop"
}
```

在环境变量中保存访问令牌，为随后访问 API 时使用：

```
TOKEN=eba6a974-3c33-48fb-9c2e-5978217ae727
```

再次尝试使用相同的代码获取访问令牌，应该会失败。因为 code 实际上是一次性密码的工作方式。

```
curl acme:acmesecret@localhost:9999/uaa/oauth/token \
 -d grant_type=authorization_code \
 -d client_id=acme \
 -d redirect_uri=http://example.com \
 -d code=$CODE -s | jq .
{
  "error": "invalid_grant",
  "error_description": "Invalid authorization code: lyJh4Y"
}
```

4.2 隐式许可（Implicit Grant）

通过 Implicit Grant，可以跳过前面的 Code Grant。可通过浏览器直接请求访问令牌。在浏览器中使用如下 URL 地址：

```
http://localhost:9999/uaa/oauth/authorize? response_type=token& client_id=acme& redirect_uri=http://example.com& scope=webshop& state=48532
```

登录（user/password）并验证通过，浏览器重定向到类似如下 URL：

```
http://example.com/#
 access_token=00d182dc-9f41-41cd-b37e-59de8f882703&
 token_type=bearer&
 state=48532&
 expires_in=42704
```

备注：在请求中 state 参数应该设置为一个随机，以便在响应中检查，避免 cross-site request forgery 攻击。

在环境变量中保存访问令牌，以便随后访问 API 时使用：

```
TOKEN=00d182dc-9f41-41cd-b37e-59de8f882703
```

4.3 资源所有者密码凭证许可（Resource Owner Password Credentials Grant）

在这一场景下，用户不必访问 web 浏览器，用户在 Client 端应用中输入凭证，通过该凭证获取访问令牌（从安全角度而言，如果你不信任 Client 端应用，这不是一个好的办法）：

```
curl -s acme:acmesecret@localhost:9999/uaa/oauth/token  \
 -d grant_type=password \
 -d client_id=acme \
 -d scope=webshop \
 -d username=user \
```

```
  -d password=password | jq .
{
   "access_token": "62ca1eb0-b2a1-4f66-bcf4-2c0171bbb593",
   "token_type": "bearer",
   "refresh_token": "920fd8e6-1407-41cd-87ad-e7a07bd6337a",
   "expires_in": 43173,
   "scope": "webshop"
}
```

在环境变量中保存访问令牌，以便在随后访问 API 时使用：

```
TOKEN=62ca1eb0-b2a1-4f66-bcf4-2c0171bbb593
```

4.4 Client 端凭证许可（Client Credentials Grant）

在最后一种情况下，我们假定用户不必准许就可以访问 API。在这种情况下，Client 端应用进行验证自己的授权服务器，并获取访问令牌：

```
curl -s acme:acmesecret@localhost:9999/uaa/oauth/token   \
  -d grant_type=client_credentials \
  -d scope=webshop | jq .
{
   "access_token": "8265eee1-1309-4481-a734-24a2a4f19299",
   "token_type": "bearer",
   "expires_in": 43189,
   "scope": "webshop"
}
```

在环境变量中保存访问令牌，以便在随后访问 API 时使用：

```
TOKEN=8265eee1-1309-4481-a734-24a2a4f19299
```

5. 访问 API

现在，我们已经获取到了访问令牌，可以开始访问实际的 API 了。

首先在没有获取到访问令牌时，尝试访问 API，将会失败：

```
curl 'http://localhost:8765/api/product/123' -s | jq .
{
   "error": "unauthorized",
   "error_description": "Full authentication is required to access this resource"
}
```

OK，这符合我们的预期。

接着，我们尝试使用一个无效的访问令牌，仍然会失败：

```
curl 'http://localhost:8765/api/product/123' \
  -H   "Authorization: Bearer invalid-access-token" -s | jq .
```

```
{
   "error": "access_denied",
   "error_description": "Unable to obtain a new access token for resource 'null'. The provider manager is not configured to support it."
}
```

再一次如期地拒绝了访问请求。

现在，我们尝试使用许可流程返回的访问令牌，执行正确的请求：

```
curl 'http://localhost:8765/api/product/123' \
  -H   "Authorization: Bearer $TOKEN" -s | jq .
{
   "productId": 123,
   "name": "name",
   "weight": 123,
   "recommendations": [...],
   "reviews": [... ]
}
```

OK，这次工作正常了！

可以查看一下 api-service（product-api-service）输出的日志记录。

```
2015-04-23 18:39:59.014 INFO 79321 --- [ XNIO-2 task-20] o.s.c.s.o.r.UserInfoTokenServices : Getting user info from: http://localhost:9999/uaa/user
2015-04-23 18:39:59.030 INFO 79321 --- [ctApiService-10] s.c.m.a.p.service.ProductApiService : ProductApi: User=user, Auth=Bearer a0f91d9e-00a6-4b61-a59f-9a084936e474, called with productId=123
2015-04-23 18:39:59.381 INFO 79321 --- [ctApiService-10] s.c.m.a.p.service.ProductApiService : GetProductComposite http-status: 200
```

我们看到 API 联系 Authorization Server，获取用户信息，并在 log 中打印出用户名和访问令牌。

最后，我们尝试使访问令牌失效，模拟它过期了。可以通过重启 auth-server（仅在内存中存储了该信息）来进行模拟，接着再次执行前面的请求：

```
curl 'http://localhost:8765/api/product/123' \
  -H   "Authorization: Bearer $TOKEN" -s | jq .
{
   "error": "access_denied",
   "error_description": "Unable to obtain a new access token for resource 'null'. The provider manager is not configured to support it."
}
```

如我们的预期一样，之前可以接受的访问令牌现在被拒绝了。

6. 总结

多谢开源项目 spring-cloud-security 和 spring-security-auth，我们可以基于 OAuth 2.0 轻松设置安全 API。然后，请记住我们使用的 Authorization Server 仅适用于开发和测试环境。

7. 下一步

在随后的文章中，将使用 ELK 技术栈（Elasticsearch、LogStash 和 Kibana）实现集中的 log 管理。

英文原文链接：

构建微服务（Blog Series - Building Microservices）
http://callistaenterprise.se/blogg/teknik/2015/05/20/blog-series-building-microservices/

译者：RickieChina@hotmail.com
欢迎分享交流。不妥之处，欢迎指正。
THANKS.