爱油科技基于SpringCloud的微服务实践

个人简介

刘思贤（微博@starlight36），爱油科技架构师、PMP。主要负责业务平台架构设计，DevOps实施和研发过程持续改进等，关注领域驱动设计与微服务、建设高效团队和工程师文化培养。

摘要

本次分享主要介绍了爱油科技基于Docker和Spring Cloud将整体业务微服务化的一些实践经验，主要包括：

- 微服务架构的分层和框架选型
- 服务发现和配置管理
- 服务集成和服务质量保证
- 基于领域驱动设计
- 实施DevOps

从单体应用到微服务

单体应用

优点

- 小而美，结构简单易于开发实现
- 部署门槛低，单个Jar包或者网站打包即可部署
- 可快速实现多实例部署

缺点

- 随着业务发展更多的需求被塞进系统，体系结构逐渐被侵蚀反应堆林立
- 被技术绑架，难以为特定业务选择平台或框架，尽管可能有更适宜的技术做这件事
- 协作困难，不同业务的团队在一个系统上进行开发相互冲突
- 难以扩展，为了热点业务而不得不同时扩容全部业务，或者难以继续扩容

架构拆分

拆分：按行分层，按列分业务

在我们的微服务体系中，所有的服务被划分为了三个层次：

1. 基础设施层：为所有业务提供基础设施，包括服务注册、数据库和NoSQL、对象存储、消息队列等基础设施服务，这一层通常是由成熟组件、第三方服务组成。
2. 业务服务层：业务微服务，根据业务领域每个子域单独一个微服务，分而治之。
3. 接入层：直接对外提供服务，例如网站、API接口等。接入层不包含复杂的业务逻辑，只做呈现和转换。

项目中我们主要关注业务服务层和接入层，对于没有足够运维力量的我们，基础设施使用云服务是省事省力的选择。

业务服务层我们给他起名叫作Epic，接入层我们起名Rune，建立之初便订立了如下原则：

1. 业务逻辑层内所有服务完全对等，可相互调用
2. 业务逻辑层所有服务必须是无状态的
3. 接入层所有服务可调用业务逻辑层所有服务，但接入层内部同层服务之间不可调用
4. 接入层不能包含业务逻辑代码
5. 所有微服务必须运行在Docker容器里

业务逻辑层我们主要使用使用Java，接入层我们主要使用PHP或Node。后来随着团队的成长，逐步将接入层全部迁移至Node。

框架选型

爱油科技作为一家成品油行业的初创型公司，需要面对非常复杂的业务场景，而且随着业务的发展，变化的可能性非常高。所以在微服务架构设计之初，我们就期望我们的微服务体系能：

- 不绑定到特定的框架、语言
- 服务最好是Restful风格
- 足够简单，容易落地，将来能扩展
- 和Docker相容性好

目前常见的微服务相关框架：

- Dubbo、DubboX
- Spring Cloud
- Motan
- Thrift、gRPC

这些常见的框架中，Dubbo几乎是唯一能被称作全栈微服务框架的“框架”，它包含了微服务所需的几乎所有内容，而DubboX作为它的增强，增加了REST支持。

它优点很多，例如：

- 全栈，服务治理的所有问题几乎都有现成答案
- 可靠，经过阿里实践检验的产品
- 实践多，社区有许多成功应用Dubbo的经验

不过遗憾的是：

- 已经停止维护
- 不利于裁剪使用
- “过于Java”，与其他语言相容性一般

Motan是微博平台微服务框架，承载了微博平台千亿次调用业务。

优点是：

- 性能好，源自于微博对高并发和实时性的要求
- 模块化，结构简单，易于使用
- 与其他语言相容性好

不过：

- 为“短平快”业务而生，即业务简单，追求高性能高并发。

Apache Thrift、gRPC等虽然优秀，并不能算作微服务框架，自身并不包括服务发现等必要特性。

如果说微服务少不了Java，那么一定少不了Spring，如果说少不了Spring，那么微服务“官配”Spring Cloud当然是值得斟酌的选择。

优点：

- “不做生产者，只做搬运工”
- 简单方便，几乎零配置
- 模块化，松散耦合，按需取用
- 社区背靠Spring大树

不足：

- 轻量并非全栈
- 没解决RPC的问题
- 实践案例少

根据我们的目标，我们最终选择了Spring Cloud作为我们的微服务框架，原因有4点：

1. 虽然Dubbo基础设施更加完善，但结构复杂，我们很难吃得下，容易出坑
2. 基于`Apache Thrift`和`gRPC`自研，投入产出比很差
3. 不想过早引入RPC以防滥用，Restful风格本身就是一种约束。
4. 做选择时，`Motan`还没有发布

Spring Cloud

Spring Cloud是一个集成框架，将开源社区中的框架集成到Spring体系下，几个重要的家族项目：

- `spring-boot`，一改Java应用程序运行难、部署难，甚至无需Web容器，只依赖JRE即可
- `spring-cloud-netflix`，集成Netflix优秀的组件Eureka、Hystrix、Ribbon、Zuul，提供服务发现、限流、客户端负载均衡和API网关等特性支持
- `spring-cloud-config`，微服务配置管理
- `spring-cloud-consul`，集成Consul支持

服务发现和配置管理

Spring Cloud Netflix提供了Eureka服务注册的集成支持，不过没选它是因为：

- 更适合纯Java平台的服务注册和发现
- 仍然需要其他分布式KV服务做后端，没解决我们的核心问题

Docker作为支撑平台的重要技术之一，Consul几乎也是我们的必选服务。因此我们觉得一事不烦二主，理所应当的Consul成为我们的服务注册中心。

Consul的优势：

- 使用Raft一致性算法，能保证分布式集群内各节点状态一致
- 提供服务注册、服务发现、服务状态检查
- 支持HTTP、DNS等协议
- 提供分布式一致性KV存储

也就是说，Consul可以一次性解决我们对服务注册发现、配置管理的需求，而且长期来看也更适合跟不同平台的系统，包括和Docker调度系统进行整合。

最初打算自己开发一个Consul和Spring Cloud整合的组件，不过幸运的是，我们做出这个决定的时候，`spring-cloud-consul`刚刚发布了，我们可以拿来即用，这节约了很多的工作量。

因此借助Consul和`spring-cloud-consul`，我们实现了

- 服务注册，引用了`srping-cloud-consul`的项目可以自动注册服务，也可以通过HTTP接口手动注册，Docker容器也可以自动注册
- 服务健康状态检查，Consul可以自动维护健康的服务列表
- 异构系统可以直接通过Consul的HTTP接口拉取并监视服务列表，或者直接使用DNS解析服务
- 通过分布式一致性KV存储进行微服务的配置下发
- 为一些业务提供选主和分布式锁服务

当然也踩到了一些坑：

`spring-cloud-consul`服务注册时不能正确选判本地ip地址。对于我们的环境来说，无论是在服务器上，还是Docker容器里，都有多个网络接口同时存在，而`spring-cloud-consul`在注册服务时，需要先选判本地服务的IP地址，判断逻辑是以第一个非本地地址为准，常常错判。因此在容器中我们利用entrypoint脚本获取再通过环境变量强制指定。

```
#!/usr/bin/env bash
set -e

# If service runs as Rancher service, auto set advertise ip address
# from Rancher metadata service.
if [ -n "$RUN_IN_RANCHER" ]; then
    echo "Waiting for ip address..."
    # Waiting for ip address
    sleep 5

    RANCHER_MS_BASE=http://rancher-metadata/2015-12-19
    PRIMARY_IP=`curl -sSL $RANCHER_MS_BASE/self/container/primary_ip`
    SERVICE_INDEX=`curl -sSL $RANCHER_MS_BASE/self/container/service_index`

    if [ -n "$PRIMARY_IP" ]; then
        export SPRING_CLOUD_CONSUL_DISCOVERY_HOSTNAME=$PRIMARY_IP
    fi

    echo "Starting service #${SERVICE_INDEX-1} at $PRIMARY_IP."
fi

exec "$@"
```

我们的容器运行在Rancher中，所以可以利用Rancher的metadata服务来获取容器的IP地址，再通过`SPRING_CLOUD_CONSUL_DISCOVERY_HOSTNAME`环境变量来设置服务发现的注册地址。基于其他容器调度平台也会很相似。

另外一些服务中内置了定时调度任务等，多实例启动时需要单节点运行调度任务。通过Consul的分布式锁服务，我们可以让获取到锁的节点启用调度任务，没获取到的节点等待获取锁。

服务集成

为了方便开发人员使用，微服务框架应当简单容易使用。对于很多微服务框架和RPC框架来说，都提供了很好的机制。在Spring Cloud中通过`OpenFeign`实现微服务之间的快速集成：

服务方声明一个Restful的服务接口，和普通的Spring MVC控制器几乎别无二致：

```
@RestController
@RequestMapping("/users")
public class UserResource {
    @RequestMapping(value = "{id}", method = RequestMethod.GET, produces = "application/json")
    public UserRepresentation findOne(@PathVariable("id") String id) {
        User user = this.userRepository.findByUserId(new UserId(id));

        if (user == null || user.getDeleted()) {
            throw new NotFoundException("指定ID的用户不存在或者已被删除。");
        }

        return new UserRepresentation(user);
    }
}
```

客户方使用一个微服务接口，只需要定义一个接口：

```
@FeignClient("epic-member-microservice")
public interface UserClient {

    @Override
    @RequestMapping(value = "/users/{id}", method = RequestMethod.GET, produces =
"application/json")
    User findOne(@PathVariable("id") String id);
}
```

在需要使用`UserClient`的Bean中，直接注入`UserClient`类型即可。事实上，`UserClient`和相关VO类，可以直接作为公共接口封装在公共项目中，供任意需要使用的微服务引用，服务方Restful Controller直接实现这一接口即可。

`OpenFeign`提供了这种简单的方式来使用Restful服务，这大大降低了进行接口调用的复杂程度。

对于错误的处理，我们使用HTTP状态码作为错误标识，并做了如下规定：

- 4xx用来表示由于客户方参数错误、状态不正确、没有权限、操作冲突等种种原因导致的业务错误。
- 5xx用来表示由于服务方系统异常、无法服务等原因服务不可用的错误。

对于服务器端，只需要在一个异常类上添加注解，即可指定该异常的HTTP响应状态码，例如：

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class NotFoundException extends RuntimeException {

    public NotFoundException() {
        super("查找的资源不存在或者已被删除。");
    }

    public NotFoundException(String message) {
        super(message);
    }

    public NotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

对于客户端我们实现了自己的`FeignClientExceptionErrorDecoder`来将请求异常转换为对于的异常类，示例如下：

```
@Component
public class FeignClientExceptionErrorDecoder implements ErrorDecoder {

    private final ErrorDecoder delegate = new ErrorDecoder.Default();

    @Override
    public Exception decode(String methodKey, Response response) {
        // Only decode 4xx errors.
        if (response.status() >= 500) {
            return delegate.decode(methodKey, response);
        }

        // Response content type must be json
        if (response.headers().getOrDefault("Content-Type", Lists.newArrayList()).stream()
            .filter(s -> s.toLowerCase().contains("json")).count() > 0) {
            try {
                String body = Util.toString(response.body().asReader());
                // 转换并返回异常对象
                ...
            } catch (IOException ex) {
                throw new RuntimeException("Failed to process response body.", ex);
            }
        }

        return delegate.decode(methodKey, response);
    }
}
```

需要注意的是，`decode`方法返回的4xx状态码异常应当是`HystrixBadRequestException`的子类对象，原因在于，我们把4xx异常视作业务异常，而不是由于故障导致的异常，所以不应当被Hystrix计算为失败请求，并引发断路器动作，这一点非常重要。

在`UserClient.findOne`方法的调用代码中，即可直接捕获相应的异常了：

```
try {
    User user = this.userClient.findOne(new UserId(id));
} catch(NotFoundException ex) {
    ...
}
```

通过`OpenFeign`，我们大大降低了Restful接口进行服务集成的难度，几乎做到了无额外工作量的服务集成。

服务质量保证

微服务架构下，由于调用需要跨系统进行远程操作，各微服务独立运维，所以在设计架构时还必须考虑伸缩性和容错性，具体地说主要包括以下几点要求：

- 服务实例可以平滑地加入、移除
- 流量可以均匀地分布在不同的实例上
- 接口应当资源隔离，防止因为个别接口调用时间过长导致线程池被占满而导致整个服务不可用
- 能支持接口降级并隔离故障节点，防止集群雪崩
- 服务能进行平滑升级

Spring Cloud中内置的`spring-cloud-netflix`的其他组件为我们提供了很好的解决方案：

- Hystrix - 实现了断路器模式，帮助控流和降级，防止集群雪崩，就像汽车的避震器
- Ribbon - 提供了客户端负载均衡器
- Zuul - API网关模式，帮助实现接口的路由、认证等

下面主要介绍一下，各个组件在进行服务质量保证中是如何发挥作用的。

Consul

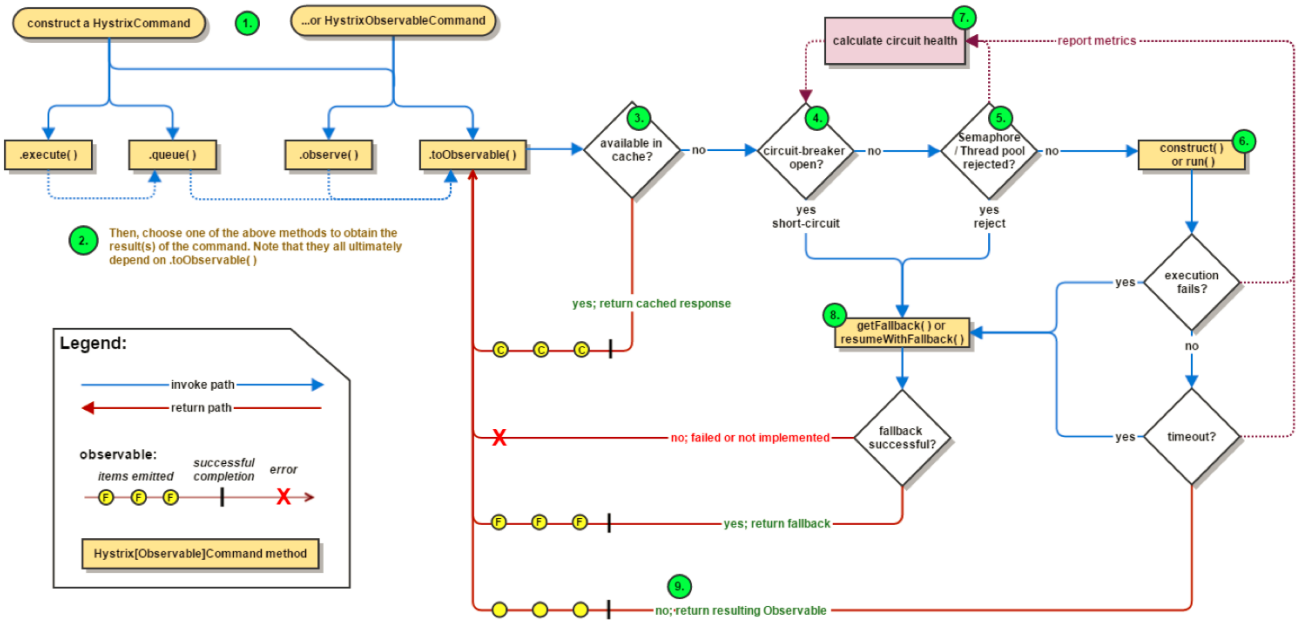
Consul中注册了一致性的可用的服务列表，并通过健康检查保证这些实例都是存活的，服务注册和检查的过程如下：

- 服务启动完成，服务端口开始监听时，`spring-cloud-consul`通过Consul接口发起服务注册，将服务的`/health`作为健康检查端点；
- Consul每隔5秒访问`/health`，检查当前微服务是否为`UP`状态；
- `/health`将会收集微服务内各个仪表收集上来的状态数据，主要包括数据库、消息队列是否连通等；
- 如果为UP状态，则微服务实例被标记为健康可用，否则被标记成失败；
- 当服务关闭时，先从Consul中取消服务注册，再优雅停机。

这样能够保证Consul中列出的所有微服务状态都是健康可用的，各个微服务会监视微服务实例列表，自动同步更新他们。

Hystrix

Hystrix提供了断路器模式的实现，主要在三个方面可以说明：



图片来自*Hystrix*项目文档

首先Hystrix提供了降级方法，断路器开启时，操作请求会快速失败不再向后投递，直接调用fallback方法来返回操作；当操作失败、被拒或者超时后，也会直接调用fallback方法返回操作。这可以保证在系统过载时，能有后备方案来返回一个操作，或者优雅的提示错误信息。断路器的存在能让故障业务被隔离，防止过载的流量涌入打死后端数据库等。

然后是基于请求数据统计的断路开关，在Hystrix中维护一个请求统计了列表（默认最多10条），列表中的每一项是一个桶。每个桶记录了在这个桶的时间范围内（默认是1秒），请求的成功数、失败数、超时数、被拒数。其中当失败请求的比例高于某一值时，将会触发断路器工作。

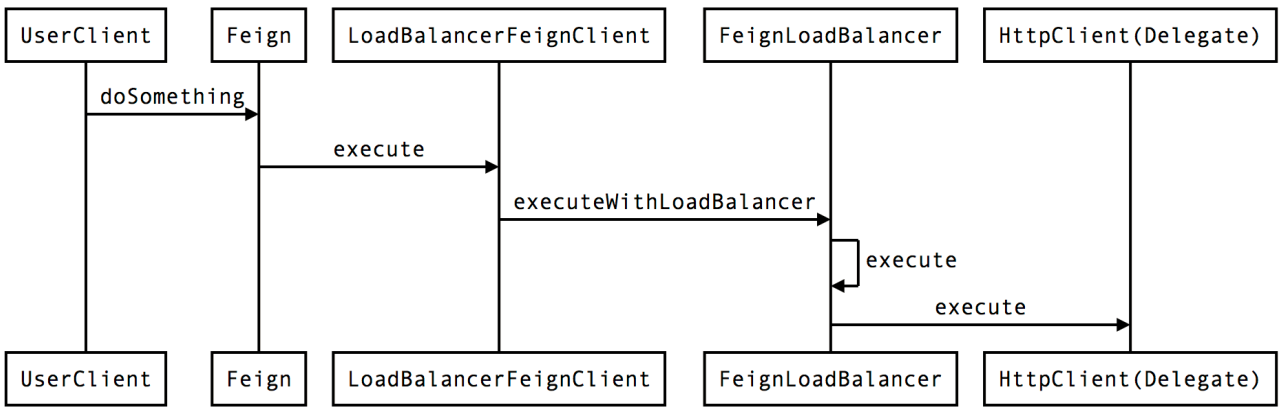
最后是不同的请求命令（`HystrixCommand`）可以使用彼此隔离的资源池，不会发生相互的挤占。在Hystrix中提供了两种隔离机制，包括线程池和信号量。线程池模式下，通过线程池的大小来限制同时占用资源的请求命令数目；信号量模式下通过控制进入临界区的操作数目来达到限流的目的。

这里包括了Hystrix的一些重要参数的配置项：

参数	说明
circuitBreaker.requestVolumeThreshold	至少在一个统计窗口内有多少个请求后，才执行断路器的开关，默认20
circuitBreaker.sleepWindowInMilliseconds	断路器触发后多久后才进行下一次判定，默认5000毫秒
circuitBreaker.errorThresholdPercentage	一个统计窗口内百分之多少的请求失败才触发熔断，默认是50%
execution.isolation.strategy	运行隔离策略，支持`Thread`，`Semaphore`，前者通过线程池来控制同时运行的命令，后者通过信号来控制，默认是`Thread`
execution.isolation.thread.interruptOnTimeout	命令执行的超时时间，默认1000毫秒
coreSize	线程池大小，默认10
keepAliveTimeMinutes	线程存活时间，默认为1分钟
maxQueueSize	最大队列长度，-1使用SynchronousQueue，默认-1。
queueSizeRejectionThreshold	允许队列堆积的最大数量

Ribbon

Ribbon使用Consul提供的服务实例列表，可以通过服务名选取一个后端服务实例连接，并保证后端流量均匀分布。`spring-cloud-netflix`整合了OpenFeign、Hystrix和Ribbon的负载均衡器，整个调用过程如下（返回值路径已经省略）：



在这个过程中，各个组件扮演的角色如下：

- Feign作为客户端工厂，负责生成客户端对象，请求和应答的编解码
- Hystrix提供限流、断路器、降级、数据统计
- Ribbon提供负载均衡器

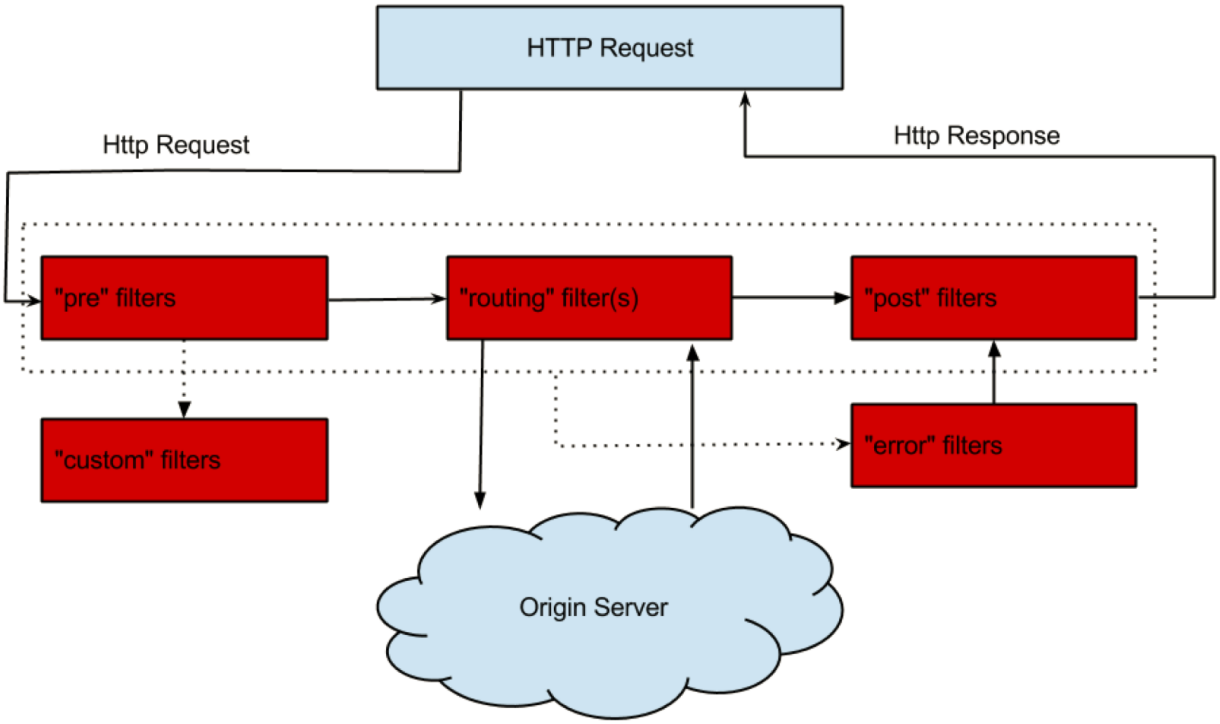
Feign负责提供客户端接口收调用，把发起请求操作（包括编码、解码和请求数据）封装成一个Hystrix命令，这个命令包裹的请求对象，会被Ribbon的负载均衡器处理，按照负载均衡策略选择一个主机，然后交给请求对象绑定的HTTP客户端对象发请求，响应成功或者不成功的结果，返回给Hystrix。

`spring-cloud-netflix`中默认使用了Ribbon的`ZoneAwareLoadBalancer`负载均衡器，它的负载均衡策略的核心指标是平均活跃请求数（Average Active Requests）。`ZoneAwareLoadBalancer`会拉取所有当前可用的服务器列表，然后将目前由于种种原因（比如网络异常）响应过慢的实例暂时从可用服务实例列表中移除，这样的机制可以保证故障实例被隔离，以免继续向其发送流量导致集群状态进一步恶化。不过由于目前`spring-cloud-consul`还不支持通过consul来指定服务实例的所在区，我们正在努力将这一功能完善。除了选区策略外，Ribbon中还提供了其他的负载均衡器，也可以自定义合适的负载均衡器。

总的来看，`spring-cloud-netflix`和Ribbon中提供了基本的负载均衡策略，对于我们来说已经足够用了。但实践中，如果需要进行灰度发布或者需要进行流量压测，目前来看还很难直接实现。而这些特性在Dubbo则开箱即用。

Zuul

Zuul为使用Java语言的接入层服务提供API网关服务，既可以根据配置反向代理指定的接口，也可以根据服务发现自动配置。Zuul提供了类似于iptables的处理机制，来帮助我们实现验证权鉴、日志等，请求工作流如下所示：

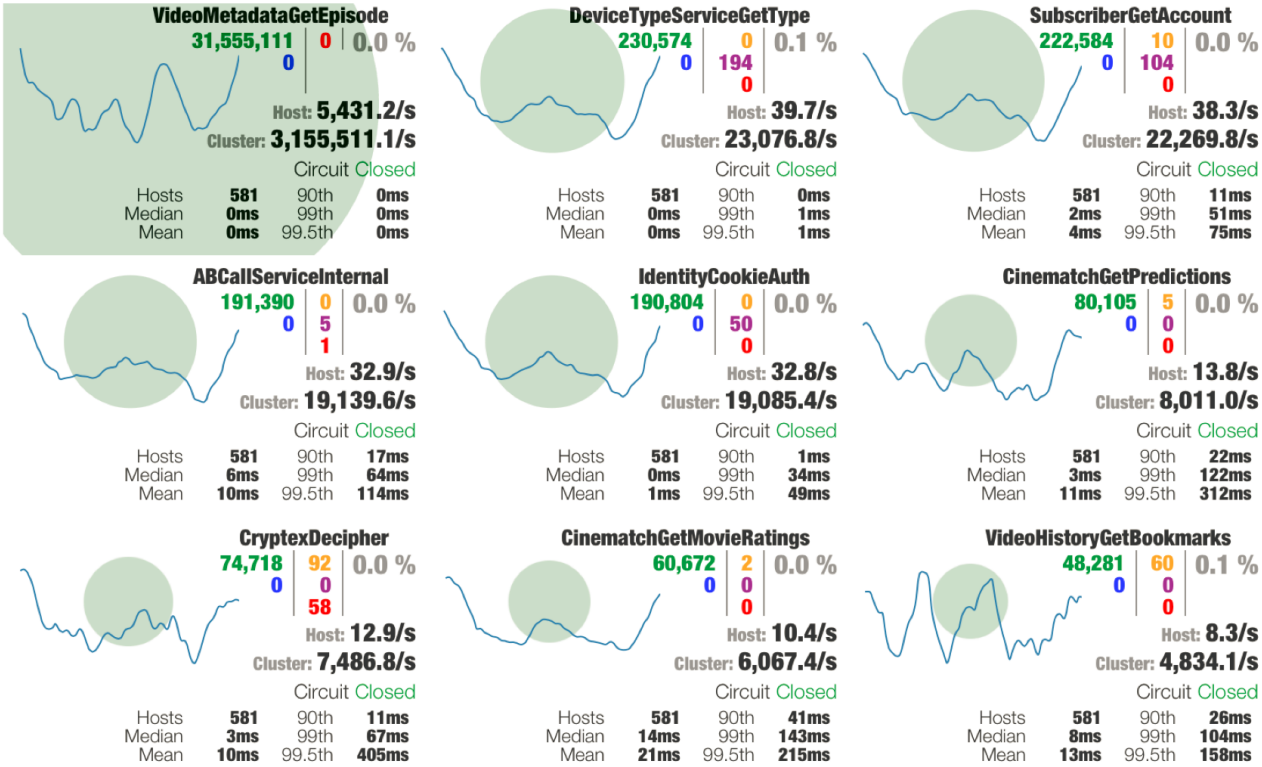


图片来自Zuul官方文档。

使用Zuul进行反向代理时，同样会走与OpenFeign类似的请求过程，确保API的调用过程也能通过Hystrix、Ribbon提供的降级、控流机制。

Hystrix Dashboard

Hystrix会统计每个请求操作的情况来帮助控制断路器，这些数据是可以暴露出来供监控系统热点。Hystrix Dashboard可以将当前接口调用的情况以图形形式展示出来：



图片来自Hystrix Dashboard官方示例

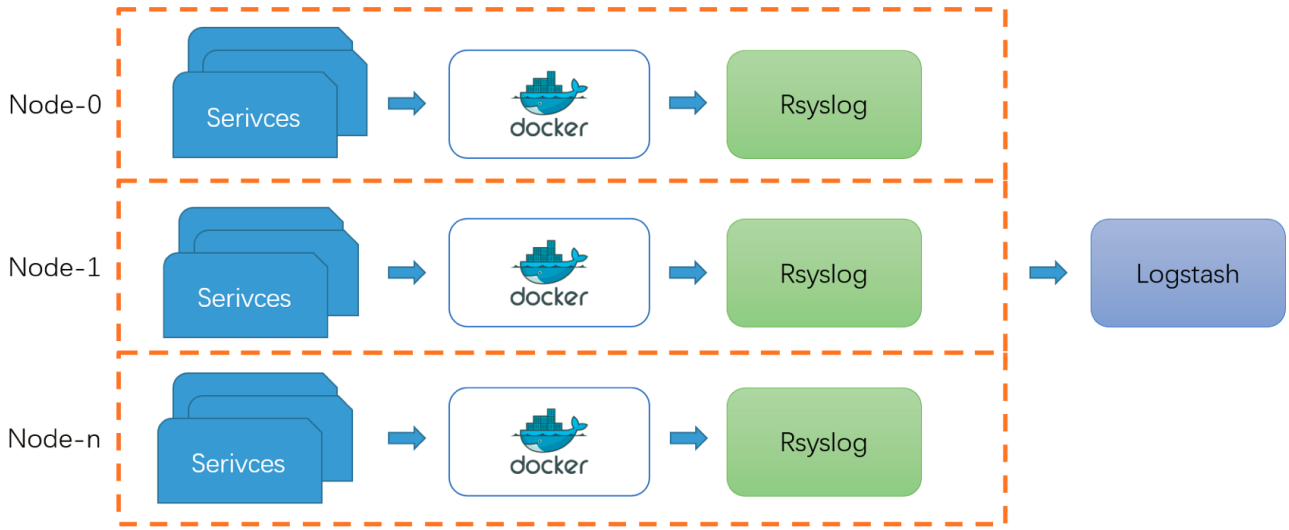
Hystrix Dashboard既可以集成在其他项目中，也可以独立运行。我们直接使用Docker启动一个Hystrix Dashboard服务即可：

```
docker run --rm -ti -p 7979:7979 kennedyoliveira/hystrix-dashboard
```

为了实现能对整个微服务集群的接口调用情况汇总，可以使用`spring-cloud-netflix-turbine`来将整个集群的调用情况汇集起来，供Hystrix Dashboard展示。

日志监控

微服务的日志直接输出到标准输出/标准错误中，再由Docker通过syslog日志驱动将日志写入至节点机器机的rsyslog中。rsyslog在本地暂存并转发至日志中心节点的Logstash中，既归档存储，又通过ElasticSearch进行索引，日志可以通过Kibana展示报表。



在rsyslog的日志收集时，需要将容器信息和镜像信息加入到tag中，通过Docker启动参数来进行配置：

```
--log-driver syslog --log-opt tag="{{.ImageName}}/{{.Name}}/{{.ID}}"
```

不过rsyslog默认只允许tag不超过32个字符，这显然是不够用的，所以我们自定义了日志模板：

```
template (name="LongTagForwardFormat" type="string" string="<%PRI%>%TIMESTAMP:::date-rfc3339%
%HOSTNAME% %syslogtag%%msg:::sp-if-no-1st-sp%%msg%")
```

在实际的使用过程中发现，当主机内存负载比较高时，rsyslog会发生日志无法收集的情况，报日志数据文件损坏。后来在Redhat官方找到了相关的问题，确认是rsyslog中的一个Bug导致的，当开启日志压缩时会出现这个问题，我们选择暂时把它禁用掉。

领域驱动设计

我们使用领域驱动设计（DDD）的方法来构建微服务，因为微服务架构和DDD有一种天然的契合。把所有业务划分成若干个子领域，有强内在关联关系的领域（界限上下文）应当被放在一起作为一个微服务。最后形成了界限上下文-工作团队-微服务一一对应的关系：

- 身份与访问 - 团队A - 成员微服务
- 商品与促销 - 团队B - 商品微服务
- 订单交易 - 团队C - 交易微服务
- ...

微服务设计

在设计单个微服务（Epic层的微服务）时，我们这样做：

- 使用OOD方法对业务进行领域建模，领域模型应当是充血模型
- 领域服务帮助完成多个领域对象协作
- 事件驱动，提供领域事件，供内部或者其他微服务使用
- 依赖倒置，在适配器接口中实现和框架、组件、SDK的整合

这给我们带来了显著的好处：

- 服务开发时关注于业务，边界合理清晰

- 容易直接对领域模型进行单元测试
- 不依赖特定组件或者平台

事务问题

从单体应用迁移到微服务架构时，不得不面临的问题之一就是事务。在单体应用时代，所有业务共享同一个数据库，一次请求操作可放置在同一个数据库事务中；在微服务架构下，这件事变得非常困难。然而事务问题不可避免，非常关键。

解决事务问题时，最先想到的解决方法通常是分布式事务。分布式事务在传统系统中应用的比较广泛，主要基于两阶段提交的方式实现。然而分布式事务在微服务架构中可行性并不高，主要基于这些考虑：

- 分布式事务需要事务管理器，对于不同语言平台来说，几乎没有有一致的实现来进行事务管理；
- 并非所有的持久化基施都提供完整ACID的事务，比如现在广泛使用的NoSQL；
- 分布式事务存在性能问题。

根据CAP理论，分布式系统不可兼得一致性、可用性、分区容错性（可靠性）三者，对于微服务架构来讲，我们通常会保证可用性、容错性，牺牲一部分一致性，追求最终一致性。所以对于微服务架构来说，使用分布式事务来解决事务问题无论是从成本还是收益上来看，都不划算。

对微服务系统来说解决事务问题，CQRS+Event Sourcing是更好的选择。

CQRS是命令和查询职责分离的缩写。CQRS的核心观点是，把操作分为修改状态的命令（Command），和返回数据的查询（Query），前者对应于“写”的操作，不能返回数据，后者对应于“读”的操作，不造成任何影响，由此领域模型被一分为二，分而治之。

Event Sourcing通常被翻译成事件溯源，简单的来说就是某一对象的当前状态，是由一系列的事件叠加后产生的，存储这些事件即可通过重放获得对象在任一时间节点上的状态。

通过CQRS+Event Sourcing，我们很容易获得最终一致性，例如对于一个跨系统的交易过程而言：

- 用户在交易微服务提交下单命令，产生领域事件`PlaceOrderEvent`，订单状态`PENDING`；
- 支付微服务收到领域事件进行扣款，扣款成功产生领域事件`PaidEvent`；
- 交易微服务收到领域事件`PaidEvent`，将订单标记为`CREATED`；
- 若支付微服务发现额度不足扣款失败，产生领域事件`InsufficientEvent`，交易微服务消费将订单标记为`CANCELED`。

我们只要保证领域事件能被持久化，那么即使出现网络延迟或部分系统失效，我们也能保证最终一致性。

实践上，我们利用Spring从4.2版本开始支持的自定义应用事件机制将本地事务和事件投递结合起来进行：

- 领域内业务过程会产生领域事件，通过Spring的应用事件机制进行应用内投递；
- 监听相应的领域事件，在事务提交前投递至消息队列；
- 以上全都没有异常发生，则本地事务提交，如果出现异常，本地事务回滚。

一些小经验

- 使用Spring Configured实现非Spring Bean的依赖注入（自己new的对象也可以注入了，对充血模型非常有用）
- 使用Swagger UI实现自文档的微服务，写好接口即有文档，即可调试

DevOps

到目前为止我们已经有数十个微服务运行于线上了，微服务数目甚至多过了团队人数。如果没有DevOps支持，运维这些微服务将是一场灾难。
我们使用Docker镜像作为微服务交付的标准件：

- Gitlab管理团队项目代码
- Gitlab-CI提供构建打包，大家提交的项目都要构建并跑通测试
- 使用Rancher作为Docker调度平台，Merge后RC分支自动部署
- 测试通过后统一上线发布

由于时间所限，这里就不展开赘述了。

永不完美

基于`spring-cloud-consul`的配置管理仍然需要完善，对于大规模应用的环境中，配置的版本控制、灰度、回滚等非常重要。SpringCloud提供了一个核，但是具体的使用还要结合场景、需求和环境等，再做一些工作。

对于非JVM语言的微服务和基于SpringCloud的微服务如何协同治理，这一问题仍然值得探索。包括像与Docker编排平台，特别是与Mesos协同进行伸缩的服务治理，还需要更多的实践来支持。

总结

- 是否选用微服务架构，应当根据业务实际情况进行判断，切勿跟风为了微服务而微服务；
- 目前来看还没有微服务全栈框架，Spring Cloud也未必是最优方案，技术选型还是应当务实；
- 微服务架构下，对于业务的理解拆分、领域建模等提出了更高的要求，相比框架，它们才是微服务架构的基石；
- DevOps是微服务实践中的重要一环，不容小视。