前言

基于 Docker 的容器技术是在2015年的时候开始接触的，两年多的时间，作为一名 Docker 的 DevOps，也见证了 Docker 的技术体系的快速发展。本文主要是结合在公司搭建的微服务架构的实践过程，做一个简单的总结。希望给在创业初期探索如何布局服务架构体系的 DevOps，或者想初步了解企业级架构的同学们一些参考。

Microservice 和 Docker

对于创业公司的技术布局，很多声音基本上是，创业公司就是要快速上线快速试错。用单应用或者前后台应用分离的方式快速集成，快速开发，快速发布。但其实这种结果造成的隐性成本会更高。当业务发展起来，开发人员多了之后，就会面临庞大系统的部署效率，开发协同效率问题。然后通过服务的拆分，数据的读写分离、分库分表等方式重新架构，而且这种方式如果要做的彻底，需要花费大量人力物力。

个人建议，DevOps 结合自己对于业务目前以及长期的发展判断，能够在项目初期使用微服务架构，多为后人谋福。

随着 Docker 周围开源社区的发展，让微服务架构的概念能有更好的一个落地实施的方案。并且在每一个微服务应用内部，都可以使用 DDD（Domain-Drive Design）的六边形架构来进行服务内的设计。关于 DDD 的一些概念也可以参考之前写的几篇文章：领域驱动设计整理——概念&架构、领域驱动设计整理——实体和值对象设计、领域服务、领域事件。

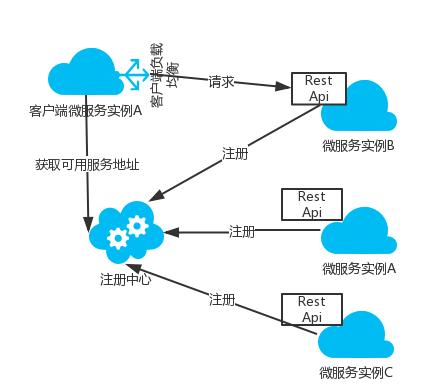
清晰的微服务的领域划分，服务内部有架构层次的优雅的实现，服务间通过 RPC 或者事件驱动完成必要的 IPC，使用 API gateway 进行所有微服务的请求转发，非阻塞的请求结果合并。本文下面会具体介绍，如何在分布式环境下，也可以快速搭建起来具有以上几点特征的，微服务架构 with Docker。

**服务发现模式**

如果使用 Docker 技术来架构微服务体系，服务发现就是一个必然的课题。目前主流的服务发现模式有两种：客户端发现模式，以及服务端发现模式。

**客户端发现模式**

客户端发现模式的架构图如下：



客户端发现模式的典型实现是Netflix体系技术。客户端从一个服务注册服务中心查询所有可用服务实例。客户端使用负载均衡算法从多个可用的服务实例中选择出一个，然后发出请求。比较典型的一个开源实现就是 Netflix 的 Eureka。

**Netflix-Eureka**

Eureka 的客户端是采用自注册的模式，客户端需要负责处理服务实例的注册和注销，发送心跳。

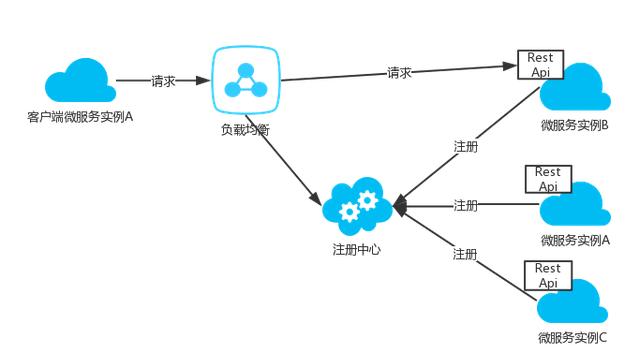
在使用 SpringBoot 集成一个微服务时，结合 SpringCloud 项目可以很方便得实现自动注册。在服务启动类上添加@EnableEurekaClient即可在服务实例启动时，向配置好的 Eureka 服务端注册服务，并且定时发送以心跳。客户端的负载均衡由 Netflix Ribbon 实现。服务网关使用 Netflix Zuul，熔断器使用 Netflix Hystrix。

除了服务发现的配套框架，SpringCloud 的 Netflix-Feign，提供了声明式的接口来处理服务的 Rest 请求。当然，除了使用 FeignClient，也可以使用 Spring RestTemplate。项目中如果使用@FeignClient可以使代码的可阅读性更好，Rest API 也一目了然。

服务实例的注册管理、查询，都是通过应用内调用 Eureka 提供的 REST API 接口（当然使用 SpringCloud-Eureka 不需要编写这部分代码）。由于服务注册、注销是通过客户端自身发出请求的，所以这种模式的一个主要问题是对于不同的编程语言会注册不同服务，需要为每种开发语言单独开发服务发现逻辑。另外，使用 Eureka 时需要显式配置健康检查支持。

**服务端发现模式**

服务端发现模式的架构图如下：



客户端向负载均衡器发出请求，负载均衡器向服务注册表发出请求，将请求转发到注册表中可用的服务实例。服务实例也是在注册表中注册，注销的。负载均衡可以使用可以使用 Haproxy 或者 Nginx。服务端发现模式目前基于 Docker 的主流方案主要是 Consul、Etcd 以及 Zookeeper。

**Consul**

Consul 提供了一个 API 允许客户端注册和发现服务。其一致性上基于RAFT算法。通过 WAN 的 Gossip 协议，管理成员和广播消息，以完成跨数据中心的同步，且支持 ACL 访问控制。Consul 还提供了健康检查机制，支持 kv 存储服务（Eureka 不支持）。Consul 的一些更详细的介绍可以参考之前写的一篇：Docker 容器部署 Consul 集群。

**Etcd**

Etcd 都是强一致的（满足 CAP 的 CP），高可用的。Etcd 也是基于 RAFT 算法实现强一致性的 KV 数据同步。Kubernetes 中使用 Etcd 的 KV 结构存储所有对象的生命周期。

关于 Etcd 的一些内部原理可以看下etcd v3原理分析

**Zookeeper**

ZK 最早应用于 Hadoop，其体系已经非常成熟，常被用于大公司。如果已经有自己的 ZK 集群，那么可以考虑用 ZK 来做自己的服务注册中心。

Zookeeper 同 Etcd 一样，强一致性，高可用性。一致性算法是基于 Paxos 的。对于微服务架构的初始阶段，没有必要用比较繁重的 ZK 来做服务发现。

**服务注册**

服务注册表是服务发现中的一个重要组件。除了 Kubernetes、Marathon 其服务发现是内置的模块之外。服务都是需要注册到注册表上。上文介绍的 Eureka、consul、etcd 以及 ZK 都是服务注册表的例子。

微服务如何注册到注册表也是有两种比较典型的注册方式：自注册模式，第三方注册模式。

**自注册模式 Self-registration pattern**

上文中的 Netflix-Eureka 客户端就是一个典型的自注册模式的例子。也即每个微服务的实例本身，需要负责注册以及注销服务。Eureka 还提供了心跳机制，来保证注册信息的准确，具体的心跳的发送间隔时间可以在微服务的 SpringBoot 中进行配置。

如下，就是使用 Eureka 做注册表时，在微服务（SpringBoot 应用）启动时会有一条服务注册的信息：

com.netflix.discovery.DiscoveryClient : DiscoveryClient\_SERVICE-USER/{your\_ip}:service-user:{port}:cc9f93c54a0820c7a845422f9ecc73fb: registering service...

同样，在应用停用时，服务实例需要主动注销本实例信息：

2018-01-04 20:41:37.290 INFO 49244 --- [ Thread-8] c.n.e.EurekaDiscoveryClientConfiguration : Unregistering application service-user with eureka with status DOWN

2018-01-04 20:41:37.340 INFO 49244 --- [ Thread-8] com.netflix.discovery.DiscoveryClient : Shutting down DiscoveryClient ...

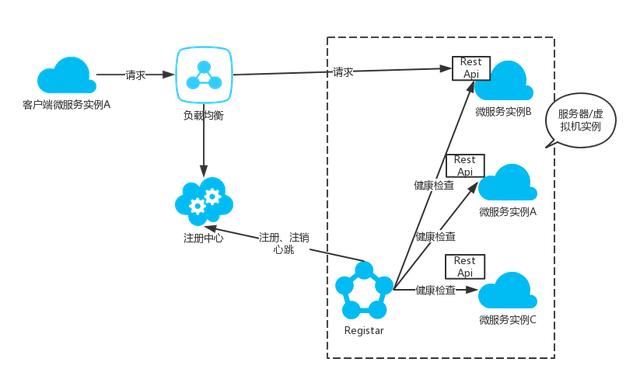
2018-01-04 20:41:37.381 INFO 49244 --- [ Thread-8] com.netflix.discovery.DiscoveryClient : Unregistering ...

2018-01-04 20:41:37.559 INFO 49244 --- [ Thread-8] com.netflix.discovery.DiscoveryClient : DiscoveryClient\_SERVICE-USER/{your\_ip}:service-user:{port}:cc9f93c54a0820c7a845422f9ecc73fb - deregister status: 200

自注册方式是比较简单的服务注册方式，不需要额外的设施或代理，由微服务实例本身来管理服务注册。但是缺点也很明显，比如 Eureka 目前只提供了 Java 客户端，所以不方便多语言的微服务扩展。因为需要微服务自己去管理服务注册逻辑，所以微服务实现也耦合了服务注册和心跳机制。跨语言性比较差。

第三方注册模式 Third party registration pattern

第三方注册，也即服务注册的管理（注册、注销服务）通过一个专门的服务管理器（Registar）来负责。Registrator 就是一个开源的服务管理器的实现。Registrator 提供了对于 Etcd 以及 Consul 的注册表服务支持。 Registrator 作为一个代理服务，需要部署、运行在微服务所在的服务器或者虚拟机中。比较简单的安装方式就是通过 Docker，以容器的方式来运行。三方注册模式的架构图如下：



通过添加一个服务管理器，微服务实例不再直接向注册中心注册，注销。由服务管理器（Registar）通过订阅服务，跟踪心跳，来发现可用的服务实例，并向注册中心（consul、etcd 等）注册，注销实例，以及发送心跳。这样就可以实现服务发现组件和微服务架构的解耦。

Registrator 配合 Consul，以及 Consul Template 搭建服务发现中心，可以参考： Scalable Architecture DR CoN: Docker, Registrator, Consul, Consul Template and Nginx 。此文示例了 Nginx 来做负载均衡，在具体的实施过程中也可以用 Haproxy 或其他方案进行替代。

小结

除了以上几种做服务发现的技术，Kubernetes 自带了服务发现模块，负责处理服务实例的注册和注销。Kubernetes 也在每个集群节点上运行代理，来实现服务端发现路由器的功能。如果编排技术使用的 k8n，可以用 k8n 的一整套 Docker 微服务方案，对 k8n 感兴趣的可以阅读下Kubernetes 架构设计与核心原理。

在实际的技术选型中，最主要还是要结合业务、系统的未来发展的特征进行合理判断。

* 在 CAP 理论中。Eureka 满足了 AP，Consul 是 CA，ZK 和 Etcd 是 CP。 在分布式场景下 Eureka 和 Consul 都能保证可用性。而搭建 Eureka 服务会相对更快速，因为不需要搭建额外的高可用服务注册中心，在小规模服务器实例时，使用 Eureka 可以节省一定成本。
* Eureka、Consul 都提供了可以查看服务注册数据的 WebUI 组件。Consul 还提供了 KV 存储，支持支持 http 和 dns 接口。对于创业公司最开始搭建微服务，比较推荐这两者。
* 在多数据中心方面，Consul 自带数据中心的 WAN 方案。ZK 和 Etcd 均不提供多数据中心功能的支持，需要额外的开发。
* 跨语言性上，Zookeeper 需要使用其提供的客户端 api，跨语言支持较弱。Etcd、Eureka 都支持 http，Etcd 还支持 grpc。Consul 除了 http 之外还提供了 DNS 的支持。
* 安全方面，Consul，Zookeeper 支持 ACL，另外 Consul、Etcd 支持安全通道 Https。
* SpringCloud 目前对于 Eureka、Consul、Etcd、ZK 都有相应的支持。
* Consul 和 Docker 一样，都是用 Go 语言实现，基于 Go 语言的微服务应用可以优先考虑用 Consul。

服务间的 IPC 机制

按照微服务的架构体系，解决了服务发现的问题之后。就需要选择合适的服务间通信的机制。如果是在 SpringBoot 应用中，使用基于 Http 协议的 REST API 是一种同步的解决方案。而且 Restful 风格的 API 可以使每个微服务应用更加趋于资源化，使用轻量级的协议也是微服务一直提倡的。

如果每个微服务是使用 DDD（Domain-Driven Design）思想的话，那么需要每个微服务尽量不使用同步的 RPC 机制。异步的基于消息的方式比如 AMQP 或者 STOMP，来松耦合微服务间的依赖会是很好的选择。目前基于消息的点对点的 pub/sub 的框架选择也比较多。下面具体介绍下两种 IPC 的一些方案。

**同步**

对于同步的请求/响应模式的通信方式。可以选择基于 Restful 风格的 Http 协议进行服务间通信，或者跨语言性很好的 Thrift 协议。如果是使用纯 Java 语言的微服务，也可以使用 Dubbo。如果是 SpringBoot 集成的微服务架构体系，建议选择跨语言性好、Spring 社区支持比较好的 RPC。

**Dubbo**

Dubbo是由阿里巴巴开发的开源的 Java 客户端的 RPC 框架。Dubbo 基于 TCP 协议的长连接进行数据传输。传输格式是使用 Hessian 二进制序列化。服务注册中心可以通过 Zookeeper 实现。

**ApacheThrift**

ApacheThrift 是由 Facebook 开发的 RPC 框架。其代码生成引擎可以在多种语言中，如 C++、 Java、Python、PHP、Ruby、Erlang、Perl 等创建高效的服务。传输数据采用二进制格式，其数据包要比使用 Json 或者 XML 格式的 HTTP 协议小。高并发，大数据场景下更有优势。

**Rest**

Rest 基于 HTTP 协议，HTTP 协议本身具有语义的丰富性。随着 Springboot 被广泛使用，越来越多的基于 Restful 风格的 API 流行起来。REST 是基于 HTTP 协议的，并且大多数开发者也是熟知 HTTP 的。

这里另外提一点，很多公司或者团队也是使用Springboot的，也在说自己是基于 Restful 风格的。但是事实其实往往是实施得并不到位。对于你的 Restful 是否是真的 Restful，可以参考这篇文章，对于 Restful 风格 API 的成熟度进行了四个层次的分析： Richardson Maturity Model steps toward the glory of REST。

如果使用Springboot的话，无论使用什么服务发现机制，都可以通过 Spring 的RestTemplate来做基础的Http请求封装。

如果使用的前文提到的Netflix-Eureka的话，可以使用Netflix-Feign。Feign是一个声明式 Web Service 客户端。客户端的负载均衡使用 Netflix-Ribbon。

**异步**

在微服务架构中，排除纯粹的“事件驱动架构”，使用消息队列的场景一般是为了进行微服务之间的解耦。服务之间不需要了解是由哪个服务实例来消费或者发布消息。只要处理好自己领域范围的逻辑，然后通过消息通道来发布，或者订阅自己关注的消息就可以。目前开源的消息队列技术也很多。比如 Apache Kafka，RabbitMQ，Apache ActiveMQ 以及阿里巴巴的 RocketMQ 目前已经成为 Apache 项目之一。消息队列的模型中，主要的三个组成就是：

* Producer：生产消息，将消息写入 channel。
* Message Broker：消息代理，将写入 channel 的消息按队列的结构进行管理。负责存储/转发消息。Broker 一般是需要单独搭建、配置的集群，而且必须是高可用的。
* Consumer：消息的消费者。目前大多数的消息队列都是保证消息至少被消费一次。所以根据使用的消息队列设施不同，消费者要做好幂等。

不同的消息队列的实现，消息模型不同。各个框架的特性也不同：

**RabbitMQ**

RabbitMQ 是基于 AMQP 协议的开源实现，由以高性能、可伸缩性出名的 Erlang 写成。目前客户端支持 Java、.Net/C# 和 Erlang。在 AMQP（Advanced Message Queuing Protocol）的组件中，Broker 中可以包含多个Exchange（交换机）组件。Exchange 可以绑定多个 Queue 以及其他 Exchange。消息会按照 Exchange 中设置的 Routing 规则，发送到相应的 Message Queue。在 Consumer 消费了这个消息之后，会跟 Broker 建立连接。发送消费消息的通知。则 Message Queue 才会将这个消息移除。

**Kafka**

Kafka 是一个高性能的基于发布/订阅的跨语言分布式消息系统。Kafka 的开发语言为 Scala。其比较重要的特性是：

* 以时间复杂度为O(1)的方式快速消息持久化；
* 高吞吐率；
* 支持服务间的消息分区，及分布式消费，同时保证消息顺序传输；
* 支持在线水平扩展,自带负载均衡；
* 支持只消费且仅消费一次（Exactly Once）模式等等。
* 说个缺点： 管理界面是个比较鸡肋了点，可以使用开源的kafka-manager

其高吞吐的特性，除了可以作为微服务之间的消息队列，也可以用于日志收集， 离线分析， 实时分析等。

Kafka 官方提供了 Java 版本的客户端 API，Kafka 社区目前也支持多种语言，包括 PHP、Python、Go、C/C++、Ruby、NodeJS 等。

**ActiveMQ**

ActiveMQ 是基于 JMS（Java Messaging Service）实现的 JMSProvider。JMS主要提供了两种类型的消息：点对点（Point-to-Point）以及发布/订阅（Publish/Subscribe）。目前客户端支持 Java、C、C++、 C#、Ruby、Perl、Python、PHP。而且 ActiveMQ 支持多种协议：Stomp、AMQP、MQTT 以及 OpenWire。

**RocketMQ/ONS**

RocketMQ 是由阿里巴巴研发开源的高可用分布式消息队列。ONS是提供商业版的高可用集群。ONS 支持 pull/push。可支持主动推送，百亿级别消息堆积。ONS 支持全局的顺序消息，以及有友好的管理页面，可以很好的监控消息队列的消费情况，并且支持手动触发消息多次重发。

小结

通过上篇的微服务的服务发现机制，加上 Restful API，可以解决微服务间的同步方式的进程间通信。当然，既然使用了微服务，就希望所有的微服务能有合理的限界上下文（系统边界）。微服务之间的同步通信应尽量避免，以防止服务间的领域模型互相侵入。为了避免这种情况，就可以在微服务的架构中使用一层API gateway（会在下文介绍）。所有的微服务通过API gateway进行统一的请求的转发，合并。并且API gateway也需要支持同步请求，以及NIO的异步的请求（可以提高请求合并的效率以及性能）。

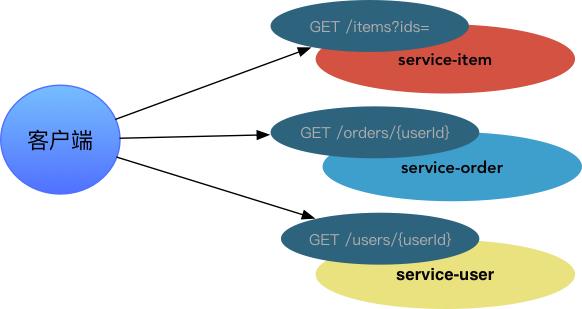
消息队列可以用于微服务间的解耦。在基于Docker的微服务的服务集群环境下，网络环境会比一般的分布式集群复杂。选择一种高可用的分布式消息队列实现即可。如果自己搭建诸如Kafka、RabbitMQ集群环境的话，那对于Broker设施的高可用性会要求很高。基于Springboot的微服务的话，比较推荐使用Kafka 或者ONS。虽然ONS是商用的，但是易于管理以及稳定性高，尤其对于必要场景才依赖于消息队列进行通信的微服务架构来说,会更适合。如果考虑到会存在日志收集，实时分析等场景，也可以搭建Kafka集群。目前阿里云也有了基于Kafka的商用集群设施。

使用 API Gateway 处理微服务请求转发、合并

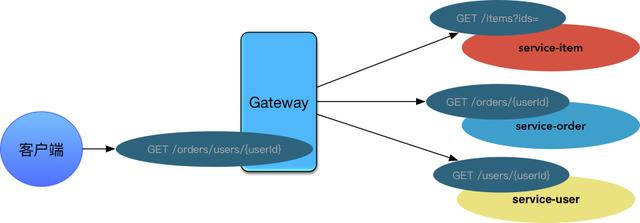
前面主要介绍了如何解决微服务的服务发现和通信问题。在微服务的架构体系中，使用DDD思想划分服务间的限界上下文的时候，会尽量减少微服务之间的调用。为了解耦微服务，便有了基于API Gateway方式的优化方案。

解耦微服务的调用

比如，下面一个常见的需求场景——“用户订单列表”的一个聚合页面。需要请求”用户服务“获取基础用户信息，以及”订单服务“获取订单信息，再通过请求“商品服务”获取订单列表中的商品图片、标题等信息。如下图所示的场景 ：



如果让客户端（比如H5、Android、iOS）发出多个请求来解决多个信息聚合，则会增加客户端的复杂度。比较合理的方式就是增加API Gateway层。API Gateway跟微服务一样，也可以部署、运行在Docker容器中，也是一个Springboot应用。如下，通过Gateway API进行转发后：



所有的请求的信息，由Gateway进行聚合，Gateway也是进入系统的唯一节点。并且Gateway和所有微服务，以及提供给客户端的也是Restful风格API。Gateway层的引入可以很好的解决信息的聚合问题。而且可以更好得适配不同的客户端的请求，比如H5的页面不需要展示用户信息，而iOS客户端需要展示用户信息，则只需要添加一个Gateway API请求资源即可，微服务层的资源不需要进行变更。

API Gateway 的特点

API gateway除了可以进行请求的合并、转发。还需要有其他的特点，才能成为一个完整的Gateway。

**响应式编程**

Gateway是所有客户端请求的入口。类似Facade模式。为了提高请求的性能，最好选择一套非阻塞I/O的框架。在一些需要请求多个微服务的场景下，对于每个微服务的请求不一定需要同步。前文举例的“用户订单列表”的例子中，获取用户信息，以及获取订单列表，就是两个独立请求。只有获取订单的商品信息，需要等订单信息返回之后，根据订单的商品id列表再去请求商品微服务。为了减少整个请求的响应时间，需要Gateway能够并发处理相互独立的请求。一种解决方案就是采用响应式编程。

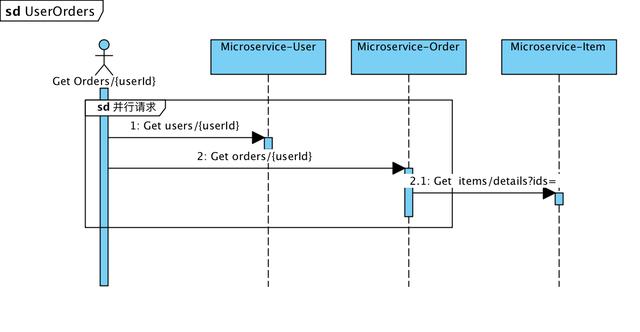
目前使用Java技术栈的响应式编程方式有，Java8的CompletableFuture，以及ReactiveX提供的基于JVM的实现-RxJava。

ReactiveX是一个使用可观察数据流进行异步编程的编程接口，ReactiveX结合了观察者模式、迭代器模式和函数式编程的精华。除了RxJava还有RxJS,RX.NET等多语言的实现。

对于Gateway来说，RxJava提供的Observable可以很好的解决并行的独立I/O请求，并且如果微服务项目中使用Java8，团队成员会对RxJava的函数学习吸收会更快。同样基于Lambda风格的响应式编程，可以使代码更加简洁。关于RxJava的详细介绍可以可以阅读RxJava文档和教程。

通过响应式编程的Observable模式，可以很简洁、方便得创建事件流、数据流，以及用简洁的函数进行数据的组合和转换，同时可以订阅任何可观察的数据流并执行操作。

通过使用RxJava,“用户订单列表”的资源请求时序图：



响应式编程可以更好的处理各种线程同步、并发请求，通过Observables和Schedulers提供了透明的数据流、事件流的线程处理。在敏捷开发模式下，响应式编程使代码更加简洁，更好维护。

**鉴权**

Gateway作为系统的唯一入口，基于微服务的所有鉴权，都可以围绕Gateway去做。在Springboot工程中，基础的授权可以使用spring-boot-starter-security以及Spring Security（Spring Security也可以集成在Spring MVC项目中）。

Spring Security主要使用AOP，对资源请求进行拦截，内部维护了一个角色的Filter Chain。因为微服务都是通过Gateway请求的，所以微服务的@Secured可以根据Gateway中不同的资源的角色级别进行设置。

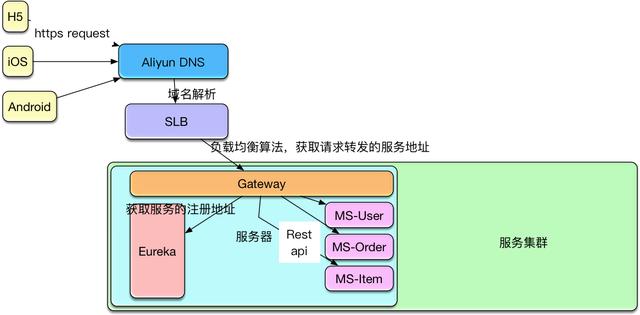
Spring Security提供了基础的角色的校验接口规范。但客户端请求的Token信息的加密、存储以及验证，需要应用自己完成。对于Token加密信息的存储可以使用Redis。这里再多提一点，为了保证一些加密信息的可变性，最好在一开始设计Token模块的时候就考虑到支持多个版本密钥，以防止万一内部密钥被泄露（之前听一个朋友说其公司的Token加密代码被员工公布出去）。至于加密算法，以及具体的实现在此就不再展开。 在Gateway鉴权通过之后，解析后的token信息可以直接传递给需要继续请求的微服务层。

如果应用需要授权（对资源请求需要管理不同的角色、权限），也只要在Gateway的Rest API基础上基于AOP思想来做即可。统一管理鉴权和授权，这也是使用类似Facade模式的Gateway API的好处之一。

**负载均衡**

API Gateway跟Microservice一样，作为Springboot应用，提供Rest api。所以同样运行在Docker容器中。Gateway和微服务之间的服务发现还是可以采用前文所述的客户端发现模式，或者服务端发现模式。

在集群环境下，API Gateway 可以暴露统一的端口，其实例会运行在不同IP的服务器上。因为我们是使用阿里云的ECS作为容器的基础设施，所以在集群环境的负载均衡也是使用阿里云的负载均衡SLB,域名解析也使用AliyunDNS。下图是一个简单的网络请求的示意：



在实践中，为了不暴露服务的端口和资源地址，也可以在服务集群中再部署Nginx服务作为反向代理，外部的负载均衡设施比如SLB可以将请求转发到Nginx服务器，请求通过Nginx再转发给Gateway端口。如果是自建机房的集群，就需要搭建高可用的负载均衡中心。为了应对跨机器请求，最好使用Consul,Consul（Consul Template）+Registor+Haproxy来做服务发现和负载均衡中心。

**缓存**

对于一些高QPS的请求，可以在API Gateway做多级缓存。分布式的缓存可以使用Redis，Memcached等。如果是一些对实时性要求不高的，变化频率不高但是高QPS的页面级请求，也可以在Gateway层做本地缓存。而且Gateway可以让缓存方案更灵活和通用。

**API Gateway的错误处理**

在Gateway的具体实现过程中，错误处理也是一个很重要的事情。对于Gateway的错误处理，可以使用Hystrix来处理请求的熔断。并且RxJava自带的onErrorReturn回调也可以方便得处理错误信息的返回。对于熔断机制，需要处理以下几个方面：

* 服务请求的容错处理

作为一个合理的Gateway，其应该只负责处理数据流、事件流，而不应该处理业务逻辑。在处理多个微服务的请求时，会出现微服务请求的超时、不可用的情况。在一些特定的场景下， 需要能够合理得处理部分失败。比如上例中的“用户订单列表”，当“User”微服务出现错误时，不应该影响“Order”数据的请求。最好的处理方式就是给当时错误的用户信息请求返回一个默认的数据，比如显示一个默认头像，默认用户昵称。然后对于请求正常的订单，以及商品信息给与正确的数据返回。如果是一个关键的微服务请求异常，比如当“Order”领域的微服务异常时，则应该给客户端一个错误码，以及合理的错误提示信息。这样的处理可以尽量在部分系统不可用时提升用户体验。使用RxJava时，具体的实现方式就是针对不同的客户端请求的情况，写好onErrorReturn，做好错误数据兼容即可。

* 异常的捕捉和记录

Gateway主要是做请求的转发、合并。为了能清楚得排查问题，定位到具体哪个服务、甚至是哪个Docker容器的问题，需要Gateway能对不同类型的异常、业务错误进行捕捉和记录。如果使用FeignClient来请求微服务资源，可以通过对ErrorDecoder接口的实现，来针对Response结果进行进一步的过滤处理，以及在日志中记录下所有请求信息。如果是使用Spring Rest Template,则可以通过定义一个定制化的RestTempate，并对返回的ResponseEntity进行解析。在返回序列化之后的结果对象之前，对错误信息进行日志记录。

* 超时机制

Gateway线程中大多是IO线程，为了防止因为某一微服务请求阻塞，导致Gateway过多的等待线程，耗尽线程池、队列等系统资源。需要Gateway中提供超时机制，对超时接口能进行优雅的服务降级。

在SpringCloud的Feign项目中集成了Hystrix。Hystrix提供了比较全面的超时处理的熔断机制。默认情况下，超时机制是开启的。除了可以配置超时相关的参数，Netflix还提供了基于Hytrix的实时监控Netflix -Dashboard,并且集群服务只需再附加部署Netflix-Turbine。通用的Hytrix的配置项可以参考Hystrix-Configuration。

如果是使用RxJava的Observable的响应式编程，想对不同的请求设置不同的超时时间，可以直接在Observable的timeout()方法的参数进行设置回调的方法以及超时时间等。

* 重试机制

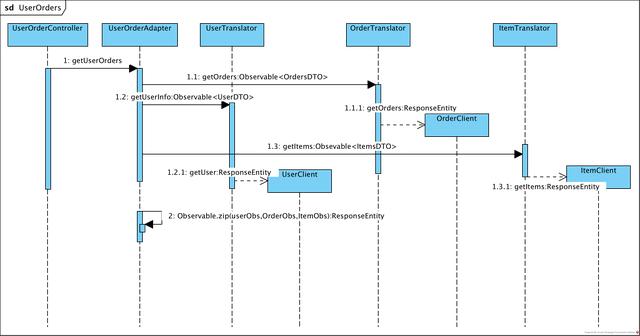
对于一些关键的业务，在请求超时时，为了保证正确的数据返回，需要Gateway能提供重试机制。如果使用SpringCloudFeign，则其内置的Ribbon，会提供的默认的重试配置，可以通过设置spring.cloud.loadbalancer.retry.enabled=false将其关闭。Ribbon提供的重试机制会在请求超时或者socket read timeout触发，除了设置重试，也可以定制重试的时间阀值以及重试次数等。

对于除了使用Feign，也使用Spring RestTemplate的应用，可以通过自定义的RestTemplate,对于返回的ResponseEntity对象进行结果解析，如果请求需要重试（比如某个固定格式的error-code的方式识别重试策略），则通过Interceptor进行请求拦截，以及回调的方式invoke多次请求。

小结

对于微服务的架构，通过一个独立的API Gateway,可以进行统一的请求转发、合并以及协议转换。可以更灵活得适配不同客户端的请求数据。而且对于不同客户端（比如H5和iOS的展示数据不同）、不同版本兼容的请求,可以很好地在Gateway进行屏蔽，让微服务更加纯粹。微服务只要关注内部的领域服务的设计，事件的处理。

API gateway还可以对微服务的请求进行一定的容错、服务降级。使用响应式编程来实现API gateway可以使线程同步、并发的代码更简洁，更易于维护。在对于微服务的请求可以统一通过FeignClint。代码也会很有层次。如下图，是一个示例的请求的类层次。



* Clint负责集成服务发现（对于使用Eureka自注册方式）、负载均衡以及发出请求，并获取ResponseEntity对象。
* Translator将ResponseEntity转换成Observable<XDTO>对象，以及对异常进行统一日志采集，类似于DDD中防腐层的概念。
* Adapter调用各个Translator，使用Observable函数，对请求的数据流进行合并。如果存在多个数据的组装，可以增加一层Assembler专门处理DTO对象到Model的转换。
* Controller,提供Restful资源的管理，每个Controller只请求唯一的一个Adapter方法。

微服务的持续集成部署

前文主要介绍了微服务的服务发现、服务通信以及API Gateway。整体的微服务架构的模型初见。在实际的开发、测试以及生产环境中。使用Docker实现微服务，集群的网络环境会更加复杂。微服务架构本身就意味着需要对若干个容器服务进行治理，每个微服务都应可以独立部署、扩容、监控。下面会继续介绍如何进行Docker微服务的持续集成部署（CI/CD）。

**镜像仓库**

用Docker来部署微服务,需要将微服务打包成Docker镜像，就如同部署在Web server打包成war文件一样。只不过Docker镜像运行在Docker容器中。

如果是Springboot服务，则会直接将包含Apache Tomcat server的Springboot，以及包含Java运行库的编译后的Java应用打包成Docker镜像。

为了能统一管理打包以及分发（pull/push）镜像。企业一般需要建立自己的镜像私库。实现方式也很简单。可以在服务器上直接部署Docker hub的镜像仓库的容器版Registry2。目前最新的版本是V2。

**代码仓库**

代码的提交、回滚等管理，也是项目持续集成的一环。一般也是需要建立企业的代码仓库的私库。可以使用SVN,GIT等代码版本管理工具。

目前公司使用的是Gitlab,通过Git的Docker镜像安装、部署操作也很便捷。具体步骤可以参考docker gitlab install。为了能快速构建、打包，也可将Git和Registry部署在同一台服务器上。

**项目构建**

在Springboot项目中，构建工具可以用Maven,或者Gradle。Gradle相比Maven更加灵活，而且Springboot应用本身去配置化的特点，用基于Groovy的Gradle会更加适合，DSL本身也比XML更加简洁高效。

因为Gradle支持自定义task。所以微服务的Dockerfile写好之后，就可以用Gradle的task脚本来进行构建打包成Docker Image。

目前也有一些开源的Gradle构建Docker镜像的工具，比如Transmode-Gradlew插件。其除了可以对子项目（单个微服务）进行构建Docker镜像，也可以支持同时上传镜像到远程镜像仓库。在生产环境中的build机器上，可以通过一个命令直接执行项目的build,Docker Image的打包,以及镜像的push。

**容器编排技术**

Docker镜像构建之后，因为每个容器运行着不同的微服务实例，容器之间也是隔离部署服务的。通过编排技术，可以使DevOps轻量化管理容器的部署以及监控，以提高容器管理的效率。

目前一些通用的编排工具比如Ansible、Chef、Puppet，也可以做容器的编排。但他们都不是专门针对容器的编排工具，所以使用时需要自己编写一些脚本，结合Docker的命令。比如Ansible，确实可以实现很便利的集群的容器的部署和管理。目前Ansible针对其团队自己研发的容器技术提供了集成方案:Ansible Container。

集群管理系统将主机作为资源池，根据每个容器对资源的需求，决定将容器调度到哪个主机上。

目前，围绕Docker容器的调度、编排，比较成熟的技术有Google的Kubernetes(下文会简写k8s)，Mesos结合Marathon管理Docker集群,以及在Docker 1.12.0版本以上官方提供的Docker Swarm。编排技术是容器技术的重点之一。选择一个适合自己团队的容器编排技术也可以使运维更高效、更自动化。

**Docker Compose**

Docker Compose是一个简单的Docker容器的编排工具，通过YAML文件配置需要运行的应用，然后通过compose up命令启动多个服务对应的容器实例。Docker中没有集成Compose,需要另外安装。

Compose可以用于微服务项目的持续集成，但其不适合大型集群的容器管理，大集群中，可以Compose结合Ansible做集群资源管理，以及服务治理。

对于集群中服务器不多的情况，可以使用Compose，其使用步骤主要是：

1. 结合微服务运行环境，定义好服务的Dockerfile
2. 根据服务镜像、端口、运行变量等编写docker-compose.yml文件，以使服务可以一起部署，运行
3. 运行docker-compose up 命令启动并且进入容器实例，如果需要使用后台进程方式运行，使用docker-compose up -d即可。

**Docker Swarm**

在16年，Docker的1.12版本出来之后,使用新版本的Docker,就自带Docker swarm mode了。不需要额外安装任何插件工具。可以看出去年开始Docker团队也开始重视服务编排技术，通过内置Swarm mode，也要抢占一部分服务编排市场。

如果团队开始使用新版本的Docker,可以选择Docker swarm mode来进行集群化的容器调度和管理。Swarm还支持滚动更新、节点间传输层安全加密、负载均衡等。

DockerSwarm的使用示例可以参考之前写的一篇：使用docker-swarm搭建持续集成集群服务。

**Kubernetes**

Kubernetes是Google开源的容器集群管理系统，使用Go语言实现，其提供应用部署、维护、 扩展机制等功能。目前可以在GCE、vShpere、CoreOS、OpenShift、Azure等平台使用k8s。国内目前Aliyun也提供了基于k8s的服务治理平台。如果是基于物理机、虚拟机搭建的Docker集群的话，也可以直接部署、运行k8s。在微服务的集群环境下，Kubernetes可以很方便管理跨机器的微服务容器实例。

目前k8s基本是公认的最强大开源服务治理技术之一。其主要提供以下功能：

1. 自动化对基于Docker对服务实例进行部署和复制
2. 以集群的方式运行，可以管理跨机器的容器，以及滚动升级、存储编排。
3. 内置了基于Docker的服务发现和负载均衡模块
4. K8s提供了强大的自我修复机制,会对崩溃的容器进行替换(对用户，甚至开发团队都无感知)，并可随时扩容、缩容。让容器管理更加弹性化。

k8s主要通过以下几个重要的组件完成弹性容器集群的管理的：

1. Pod是Kubernetes的最小的管理元素，一个或多个容器运行在pod中。pod的生命周期很短暂，会随着调度失败，节点崩溃，或者其他资源回收时消亡。
2. Label是key/value存储结构的，可以关联pod，主要用来标记pod，给服务分组。微服务之间通过label选择器（Selectors）来识别Pod。
3. Replication Controller是k8s Master节点的核心组件。用来确保任何时候Kubernetes集群中有指定数量的pod副本(replicas)运行。即提供了自我修复机制的功能，并且对缩容扩容、滚动升级也很有用。
4. Service是对一组Pod的策略的抽象。也是k8s管理的基本元素之一。Service通过Label识别一组Pod。创建时也会创建一个本地集群的DNS（存储Service对应的Pod的服务地址）。所以在客户端请求通过请求DNS来获取一组当前可用的Pods的ip地址。之后通过每个Node中运行的kube-proxy将请求转发给其中一个Pod。这层负载均衡是透明的，但是目前的k8s的负载均衡策略还不是很完善，默认是随机的方式。

小结

微服务架构体系中，一个合适的持续集成的工具，可以很好得提升团队的运维、开发效率。目前类似Jenkins也有针对Docker的持续集成的插件，但是还是存在很多不完善。所以建议还是选择专门应对Docker容器编排技术的Swarm,k8s,Mesos。或者多个技术结合起来，比如Jenkins做CI+k8s做CD。

Swarm,k8s,Mesos各有各的特性，他们对于容器的持续部署、管理以及监控都提供了支持。Mesos还支持数据中心的管理。Docker swarm mode扩展了现有的Docker API，通过Docker Remote API的调用和扩展，可以调度容器运行到指定的节点。Kubernetes则是目前市场规模最大的编排技术，目前很多大公司也都加入到了k8s家族，k8s应对集群应用的扩展、维护和管理更加灵活，但是负载均衡策略比较粗糙。而Mesos更专注于通用调度，提供了多种调度器。

对于服务编排，还是要选择最适合自己团队的，如果初期机器数量很少，集群环境不复杂也可以用Ansible+Docker Compose,再加上Gitlab CI来做持续集成。

**服务集群的解决方案**

企业在实践使用Docker部署、运行微服务应用的时候，无论是一开始就布局微服务架构，或者从传统的单应用架构进行微服务化迁移。都需要能够处理复杂的集群中的服务调度、编排、监控等问题。下面主要为介绍在分布式的服务集群下，如何更安全、高效得使用Docker,以及在架构设计上，需要考虑的方方面面。

**负载均衡**

这里说的是集群中的负载均衡，如果是纯服务端API的话就是指Gateway API的负载均衡，如果使用了Nginx的话，则是指Nginx的负载均衡。我们目前使用的是阿里云的负载均衡服务SLB。其中一个主要原因是可以跟DNS域名服务进行绑定。对于刚开始进行创业的公司来说，可以通过Web界面来设置负载均衡的权重，比较便于部分发布、测试验证,以及健康检查监控等等。从效率和节约运维成本上来说都是个比较适合的选择。

如果自己搭建七层负载均衡如使用Nginx或Haproxy的话，也需要保证负责负载均衡的集群也是高可用的,以及提供便捷的集群监控，蓝绿部署等功能。

**持久化及缓存**

**关系型数据库（RDBMS）**

对于微服务来说，使用的存储技术主要是根据企业的需要。为了节约成本的话，一般都是选用Mysql，在Mysql的引擎选择的话建议选择InnoDB引擎(5.5版本之前默认MyISAM)。InnoDB在处理并发时更高效，其查询性能的差距也可以通过缓存、搜索等方案进行弥补。InnoDB处理数据拷贝、备份的免费方案有binlog，mysqldump。不过要做到自动化的备份恢复、可监控的数据中心还是需要DBA或者运维团队。相对花费的成本也较高。如果初创企业，也可以考虑依托一些国内外比较大型的云计算平台提供的PaaS服务。

微服务一般按照业务领域进行边界划分，所以微服务最好是一开始就进行分库设计。是否需要进行分表需要根据每个微服务具体的业务领域的发展以及数据规模进行具体分析。但建议对于比较核心的领域的模型，比如“订单”提前做好分表字段的设计和预留。

**KV模型数据库（Key-Value-stores）**

Redis是开源的Key-Value结构的数据库。其基于内存，具有高效缓存的性能，同时也支持持久化。Redis主要有两种持久化方式。一种是RDB，通过指定时间间隔生成数据集的时间点快照，从内存写入磁盘进行持久化。RDB方式会引起一定程度的数据丢失，但性能好。另外一种是AOF,其写入机制，有点类似InnoDB的binlog，AOF的文件的命令都是以Redis协议格式保存。这两种持久化是可以同时存在的，在Redis重启时，AOF文件会被优先用于恢复数据。因为持久化是可选项，所以也可以禁用Redis持久化。

在实际的场景中，建议保留持久化。比如目前比较流行的解决短信验证码的验证，就可使用Redis。在微服务架构体系中，也可以用Redis处理一些KV数据结构的场景。轻量级的数据存储方案，也很适合本身强调轻量级方案的微服务思想。

我们在实践中，是对Redis进行了缓存、持久化，两个功能特征进行分库的。

在集成Springboot项目中会使用到spring-boot-starter-data-redis来进行Redis的数据库连接以及基础配置、以及spring-data-redis提供的丰富的数据APIOperations。

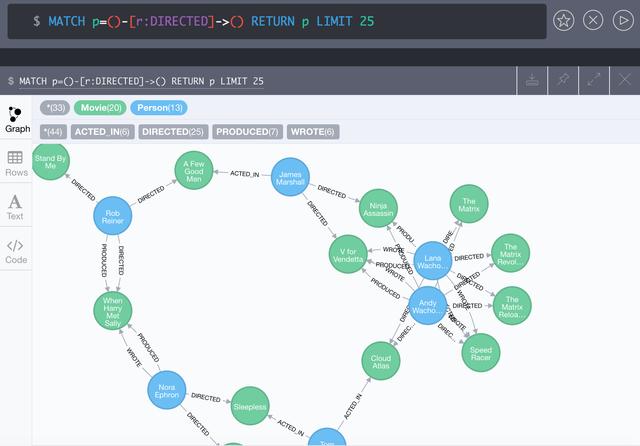
另外，如果是要求高吞吐量的应用，可以考虑用Memcached来专门做简单的KV数据结构的缓存。其比较适合大数据量的读取，但支持的数据结构类型比较单一。

**图形数据库（Graph Database）**

涉及到社交相关的模型数据的存储，图形数据库是一种相交关系型数据库更高效、更灵活的选择。图形数据库也是Nosql的一种。其和KV不同，存储的数据主要是数据节点（node）,具有指向性的关系（Relationship）以及节点和关系上的属性（Property）。

如果用Java作为微服务的主开发语言，最好选择Neo4j。Neo4j是一种基于Java实现的支持ACID的图形数据库。其提供了丰富的JavaAPI。在性能方面，图形数据库的局部性使遍历的速度非常快，尤其是大规模深度遍历。这个是关系型数据库的多表关联无法企及的。

下图是使用Neo4j的WebUI工具展示的一个官方Getting started数据模型示例。示例中的语句MATCH p=()-[r:DIRECTED]->() RETURN p LIMIT 25是Neo4j提供的查询语言——Cypher。



在项目使用时可以集成SpringData的项目Spring Data Neo4j。以及SpringBootStartersspring-boot-starter-data-neo4j

**文档数据库（Document database）**

目前应用的比较广泛的开源的面向文档的数据库可以用Mongodb。Mongo具有高可用、高可伸缩性以及灵活的数据结构存储，尤其是对于Json数据结构的存储。比较适合博客、评论等模型的存储。

**搜索技术**

在开发的过程中，有时候经常会看到有人写了很长很绕、很难维护的多表查询SQL，或者是各种多表关联的子查询语句。对于某一领域模型，当这种场景多的时候，就该考虑接入一套搜索方案了。不要什么都用SQL去解决，尤其是查询的场景。慢查询语句的问题有时候甚至会拖垮DB，如果DB的监控体系做的不到位，可能问题也很难排查。

Elasticsearch是一个基于Apache Lucene实现的开源的实时分布式搜索和分析引擎。Springboot的项目也提供了集成方式: spring-boot-starter-data-elasticsearch以及spring-data-elasticsearch。

对于搜索集群的搭建，可以使用Docker。具体搭建方法可以参考用Docker搭建Elasticsearch集群,对于Springboot项目的集成可以参考在Springboot微服务中集成搜索服务。至今，最新版本的SpringDataElasticsearch已经支持到了5.x版本的ES，可以解决很多2.x版本的痛点了。

如果是小规模的搜索集群，可以用三台低配置的机器，然后用ES的Docker进项进行搭建。也可以使用一些商业版的PaaS服务。如何选择还是要根据团队和业务的规模、场景来看。

目前除了ES,使用比较广泛的开源搜索引擎还有Solr,Solr也基于Lucene,且专注在文本搜索。而ES的文本搜索确实不如Solr,ES主要专注于对分布式的支持，并且内置了服务发现组件Zen来维护集群状态，相对Solr（需要借助类似Zookeeper实现分布式）部署也更加轻量级。ES除了分析查询，还可以集成日志收集以及做分析处理。

**消息队列**

消息队列如前篇所述，可以作为很好的微服务解耦通信方式。在分布式集群的场景下，对于分布式下的最终一致性也可以提供技术基础保障。并且消息队列也可以用来处理流量削锋。

消息队列的对比在此不再赘述。目前公司使用的是阿里云的ONS。因为使用消息队列还是考虑用在对高可用以及易于管理、监控上的要求，所以选择了安全可靠的消息队列平台。

**安全技术**

安全性是做架构需要考虑的基础。互联网的环境复杂，保护好服务的安全，也是对用户的基本承诺。安全技术涉及到的范围比较广，本文选几个常见问题以及常用方式来简单介绍下。

**服务实例安全**

分布式集群本身就是对于服务实例安全的一种保障。一台服务器或者某一个服务实例出现问题的时候，负载均衡可以将请求转发到其他可用的服务实例。但很多企业是自建机房，而且是单机房的，这种布局其实比较危险。因为服务器的备份容灾也得不到完整的保障。最怕的就是数据库也是在同一机房，主备全都在一起。不单是安全性得不到很高的保障，平常的运维花销也会比较大。而且需要注意配置防火墙安全策略。

如果可以，尽量使用一些高可用、高可伸缩的稳定性IaaS平台。

**网络安全**

**1. 预防网络攻击**

目前主要的网络攻击有一下几种：

* SQL注入：根据不同的持久层框架，应对策略不同。如果使用JPA，则只要遵循JPA的规范，基本不用担心。
* XSS攻击：做好参数的转义处理和校验。具体参考XSS预防
* CSRF攻击：做好Http的Header信息的Token、Refer验证。具体参考CSRF预防
* DDOS攻击：大流量的DDoS攻击，一般是采用高防IP。也可以接入一些云计算平台的高防IP。

以上只是列举了几种常见的攻击，想要深入了解的可以多看看REST安全防范表。在网络安全领域，一般很容易被初创企业忽视，如果没有一个运维安全团队，最好使用类似阿里云-云盾之类的产品。省心省成本。

**2. 使用安全协议**

这个不用多说，无论是对于使用Restful API的微服务通信，还是使用的CDN或者使用的DNS服务。涉及到Http协议的，建议都统一使用Https。无论是什么规模的应用，都要防范流量劫持，否则将会给用户带来很不好的使用体验。

**3. 鉴权**

关于微服务的鉴权前面API Gateway已经有介绍。除了微服务本身之外，我们使用的一些如Mysql，Redis,Elasticsearch,Eureka等服务，也需要设置好鉴权，并且尽量通过内网访问。不要对外暴露过多的端口。对于微服务的API Gateway,除了鉴权，最好前端通过Nginx反向代理来请求API层。

**日志采集、监控**

基于容器技术的微服务的监控体系面临着更复杂的网络、服务环境。日志采集、监控如何能对微服务减少侵入性、对开发者更透明，一直是很多微服务的DevOps在不断思考和实践的。

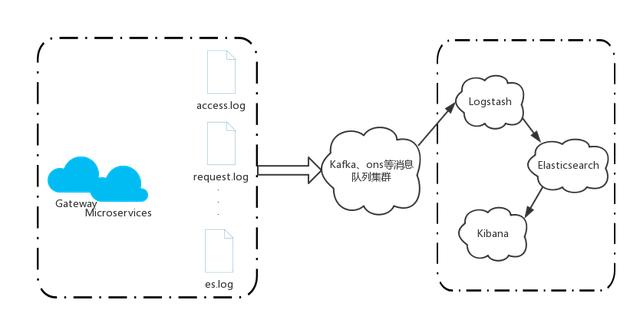
**1. 微服务日志的采集**

微服务的API层的监控，需要从API Gateway到每个微服务的调用路径的跟踪，采集以及分析。使用Rest API的话，为了对所有请求进行采集，可以使用Spring Web的OncePerRequestFilter对所有请求进行拦截，在采集日志的时候，也最好对请求的rt进行记录。

除了记录access，request等信息，还需要对API调用进行请求跟踪。如果单纯记录每个服务以及Gateway的日志，那么当Gateway Log出现异常的时候，就不知道其具体是微服务的哪个容器实例出现了问题。如果容器达到一定数量，也不可能排查所有容器以及服务实例的日志。比较简单的解决方式就是对log信息都append一段含有容器信息的、唯一可标识的Trace串。

日志采集之后，还需要对其进行分析。如果使用E.L.K的技术体系，就可以灵活运用Elasticsearch的实时分布式特性。Logstash可以进行日志进行收集、分析，并将数据同步到Elasticsearch。Kibana结合Logstash和ElasticSearch，提供良好的便于日志分析的WebUI，增强日志数据的可视化管理。

对于数据量大的日志的采集，为了提升采集性能，需要使用上文提到的消息队列。优化后的架构如下：



**2. 基础服务的调用日志采集**

通过对微服务的所有Rest API的日志采集、分析可以监控请求信息。

在服务内部，对于中间件、基础设施（包括Redis,Mysql,Elasticsearch等）调用的性能的日志采集和分析也是必要的。

对于中间件服务的日志采集，我们目前可以通过动态代理的方式，对于服务调用的如cache、repository（包括搜索和DB）的基础方法，进行拦截及回调日志记录方法。具体的实现方式可以采用字节码生成框架ASM,关于方法的逻辑注入，可以参考之前写的一篇ASM(四) 利用Method 组件动态注入方法逻辑，如果觉得ASM代码不太好维护，也可以使用相对API友好的Cglib。

架构五要素：

最后，结合架构核心的五要素来回顾下我们在搭建Docker微服务架构使用的技术体系：

1. 高性能

消息队列、RxJava异步并发、分布式缓存、本地缓存、Http的Etag缓存、使用Elasticsearch优化查询、CDN等等。

1. 可用性

容器服务集群、RxJava的熔断处理、服务降级、消息的幂等处理、超时机制、重试机制、分布式最终一致性等等。

1. 伸缩性

服务器集群的伸缩、容器编排Kubernetes、数据库分库分表、Nosql的线性伸缩、搜索集群的可伸缩等等。

1. 扩展性

基于Docker的微服务本身就是为了扩展性而生！

1. 安全性

JPA/Hibernate,SpringSecurity、高防IP、日志监控、Https、Nginx反向代理、HTTP/2.0等等。

小结

对于服务集群的解决方案，其实无论是微服务架构或者SOA架构，都是比较通用的。只是对于一些中间件集群的搭建，可以使用Docker。一句Docker ps就可以很方便查询运行的服务信息，并且升级基础服务也很方便。

对于优秀的集群架构设计的追求是永无止境的。在跟很多创业公司的技术朋友们接触下来，大家都是比较偏向于快速搭建以及开发、发布服务。然而一方面也顾虑微服务的架构会比较复杂，杀鸡用牛刀。但是微服务本身就是一种敏捷模式的优秀实践。这些朋友往往会在业务飞速发展的时候面临一个困扰，就是服务拆分，数据库的分库分表、通过消息去解耦像面条一样的同步代码，想要优化性能但是无从下手的尴尬。

相关文档

Apache Thrift

使用Mesos和Marathon管理Docker集群

基于docker-swarm搭建持续集成集群服务

Kubernetes中文文档

后记

本文主要是对于Docker的微服务实践进行技术方案选型以及介绍。不同的业务、团队可能会适合不通过的架构体系和技术方案。

作为架构师应该结合公司近期、长期的战略规划，进行长远的布局。最起码基础的架构也是需要能支撑3年发展，期间可以不断引入新的技术并进行服务升级和持续的代码层重构。

也许一个架构师从0开始搭建一整套体系并不需要花费多久时间，最需要其进行的就是不断在团队推行Domain-Driven Design。并且使团队一起遵循Clean Code，进行敏捷开发OvO。

PS:本文所有图均原创图，主要使用工具： Processon、OmniGraffle、VisualParadigm。

Thanks for your reading!