# 一、微服务介绍

## 1. 什么是微服务

在介绍微服务时，首先得先理解什么是微服务，顾名思义，微服务得从两个方面去理解，什么是"微"、什么是"服务"， 微 狭义来讲就是体积小、著名的"2 pizza 团队"很好的诠释了这一解释（2 pizza 团队最早是亚马逊 CEO Bezos提出来的，意思是说单个服务的设计，所有参与人从设计、开发、测试务由来

微服务最早由Martin Fowler与James Lewis于2014年共同提出，微服务架构风格是一种使用一套小服务来开发单个应用的方式途径，每个服务运行在自己的进程中，并使用轻量级机制通信，通常是HTTP API，这些服务基于业务能力构建，并能够通过自动化部署机制来独立部署，这些服务使用不同的编程语言实现，以及不同数据存储技术，并保持最低限度的集中式管理。

## 3. 为什么需要微服务？

在传统的IT行业软件大多都是各种独立系统的堆砌，这些系统的问题总结来说就是扩展性差，可靠性不高，维护成本高。到后面引入了SOA服务化，但是，由于 SOA 早期均使用了总线模式，这种总线模式是与某种技术栈强绑定的，比如：J2EE。这导致很多企业的遗留系统很难对接，切换时间太长，成本太高，新系统稳定性的收敛也需要一些时间。最终 SOA 看起来很美，但却成为了企业级奢侈品，中小公司都望而生畏。

### 3.1 最期的单体架构带来的问题

单体架构在规模比较小的情况下工作情况良好，但是随着系统规模的扩大，它暴露出来的问题也越来越多，主要有以下几点：

**1.复杂性逐渐变高**

* 比如有的项目有几十万行代码，各个模块之间区别比较模糊，逻辑比较混乱，代码越多复杂性越高，越难解决遇到的问题。

**2.技术债务逐渐上升**

* 公司的人员流动是再正常不过的事情，有的员工在离职之前，疏于代码质量的自我管束，导致留下来很多坑，由于单体项目代码量庞大的惊人，留下的坑很难被发觉，这就给新来的员工带来很大的烦恼，人员流动越大所留下的坑越多，也就是所谓的技术债务越来越多。

**3.部署速度逐渐变慢**

* 这个就很好理解了，单体架构模块非常多，代码量非常庞大，导致部署项目所花费的时间越来越多，曾经有的项目启动就要一二十分钟，这是多么恐怖的事情啊，启动几次项目一天的时间就过去了，留给开发者开发的时间就非常少了。

**4.阻碍技术创新**

* 比如以前的某个项目使用struts2写的，由于各个模块之间有着千丝万缕的联系，代码量大，逻辑不够清楚，如果现在想用spring mvc来重构这个项目将是非常困难的，付出的成本将非常大，所以更多的时候公司不得不硬着头皮继续使用老的struts架构，这就阻碍了技术的创新。

**5.无法按需伸缩**

* 比如说电影模块是CPU密集型的模块，而订单模块是IO密集型的模块，假如我们要提升订单模块的性能，比如加大内存、增加硬盘，但是由于所有的模块都在一个架构下，因此我们在扩展订单模块的性能时不得不考虑其它模块的因素，因为我们不能因为扩展某个模块的性能而损害其它模块的性能，从而无法按需进行伸缩。

### 3.2 微服务与单体架构区别

1. 单体架构所有的模块全都耦合在一块，代码量大，维护困难，微服务每个模块就相当于一个单独的项目，代码量明显减少，遇到问题也相对来说比较好解决。
2. 单体架构所有的模块都共用一个数据库，存储方式比较单一，微服务每个模块都可以使用不同的存储方式（比如有的用redis，有的用mysql等），数据库也是单个模块对应自己的数据库。
3. 单体架构所有的模块开发所使用的技术一样，微服务每个模块都可以使用不同的开发技术，开发模式更灵活。

### 3.3 微服务与SOA区别

微服务，从本质意义上看，还是 SOA 架构。但内涵有所不同，微服务并不绑定某种特殊的技术，在一个微服务的系统中，可以有 Java 编写的服务，也可以有 Python编写的服务，他们是靠Restful架构风格统一成一个系统的。所以微服务本身与具体技术实现无关，扩展性强。

## 4. 微服务本质

1. 微服务，关键其实不仅仅是微服务本身，而是系统要提供一套基础的架构，这种架构使得微服务可以独立的部署、运行、升级，不仅如此，这个系统架构还让微服务与微服务之间在结构上“**松耦合**”，而在功能上则表现为一个统一的整体。这种所谓的“统一的整体”表现出来的是统一风格的界面，统一的权限管理，统一的安全策略，统一的上线过程，统一的日志和审计方法，统一的调度方式，统一的访问入口等等。
2. **微服务的目的是有效的拆分应用，实现敏捷开发和部署 。**
3. 微服务提倡的理念团队间应该是 **inter-operate, not integrate** 。inter-operate是定义好系统的边界和接口，在一个团队内全栈，让团队自治，原因就是因为如果团队按照这样的方式组建，将沟通的成本维持在系统内部，每个子系统就会更加内聚，彼此的依赖耦合能变弱，跨系统的沟通成本也就能降低。

## 5. 什么样的项目适合微服务

微服务可以按照业务功能本身的独立性来划分，如果系统提供的业务是非常底层的，如：操作系统内核、存储系统、网络系统、数据库系统等等，这类系统都偏底层，功能和功能之间有着紧密的配合关系，如果强制拆分为较小的服务单元，会让集成工作量急剧上升，并且这种人为的切割无法带来业务上的真正的隔离，所以无法做到独立部署和运行，也就不适合做成微服务了。

**能不能做成微服务，取决于四个要素：**

* 小：微服务体积小，2 pizza 团队。
* 独：能够独立的部署和运行。
* 轻：使用轻量级的通信机制和架构。
* 松：为服务之间是松耦合的。

## 6. 微服务折分与设计

1. 从单体式结构转向微服务架构中会持续碰到服务边界划分的问题：比如，我们有user 服务来提供用户的基础信息，那么用户的头像和图片等是应该单独划分为一个新的service更好还是应该合并到user服务里呢？如果服务的粒度划分的过粗，那就回到了单体式的老路；如果过细，那服务间调用的开销就变得不可忽视了，管理难度也会指数级增加。目前为止还没有一个可以称之为服务边界划分的标准，只能根据不同的业务系统加以调节
2. 拆分的大原则是当一块业务不依赖或极少依赖其它服务，有独立的业务语义，为超过2个的其他服务或客户端提供数据，那么它就应该被拆分成一个独立的服务模块。

## 6.1 微服务设计原则

**单一职责原则**

* 意思是每个微服务只需要实现自己的业务逻辑就可以了，比如订单管理模块，它只需要处理订单的业务逻辑就可以了，其它的不必考虑。

**服务自治原则**

* 意思是每个微服务从开发、测试、运维等都是独立的，包括存储的数据库也都是独立的，自己就有一套完整的流程，我们完全可以把它当成一个项目来对待。不必依赖于其它模块。

**轻量级通信原则**

* 首先是通信的语言非常的轻量，第二，该通信方式需要是跨语言、跨平台的，之所以要跨平台、跨语言就是为了让每个微服务都有足够的独立性，可以不受技术的钳制。

**接口明确原则**

* 由于微服务之间可能存在着调用关系，为了尽量避免以后由于某个微服务的接口变化而导致其它微服务都做调整，在设计之初就要考虑到所有情况，让接口尽量做的更通用，更灵活，从而尽量避免其它模块也做调整。

## 7. 微服务优势与缺点

### 7.1 特性

1. 每个微服务可独立运行在自己的进程里；
2. 一系列独立运行的微服务共同构建起了整个系统；
3. 每个服务为独立的业务开发，一个微服务一般完成某个特定的功能，比如：订单管理，用户管理等；
4. 微服务之间通过一些轻量级的通信机制进行通信，例如通过REST API或者RPC的方式进行调用。

### 7.2 特点

**易于开发和维护**

* 由于微服务单个模块就相当于一个项目，开发这个模块我们就只需关心这个模块的逻辑即可，代码量和逻辑复杂度都会降低，从而易于开发和维护。

**启动较快**

* 这是相对单个微服务来讲的，相比于启动单体架构的整个项目，启动某个模块的服务速度明显是要快很多的。

**局部修改容易部署**

* 在开发中发现了一个问题，如果是单体架构的话，我们就需要重新发布并启动整个项目，非常耗时间，但是微服务则不同，哪个模块出现了bug我们只需要解决那个模块的bug就可以了，解决完bug之后，我们只需要重启这个模块的服务即可，部署相对简单，不必重启整个项目从而大大节约时间。

**技术栈不受限**

* 比如订单微服务和电影微服务原来都是用java写的，现在我们想把电影微服务改成nodeJs技术，这是完全可以的，而且由于所关注的只是电影的逻辑而已，因此技术更换的成本也就会少很多。

**按需伸缩**

* 我们上面说了单体架构在想扩展某个模块的性能时不得不考虑到其它模块的性能会不会受影响，对于我们微服务来讲，完全不是问题，电影模块通过什么方式来提升性能不必考虑其它模块的情况。

### 7.3 缺点

**运维要求较高**

* 对于单体架构来讲，我们只需要维护好这一个项目就可以了，但是对于微服务架构来讲，由于项目是由多个微服务构成的，每个模块出现问题都会造成整个项目运行出现异常，想要知道是哪个模块造成的问题往往是不容易的，因为我们无法一步一步通过debug的方式来跟踪，这就对运维人员提出了很高的要求。

**分布式的复杂性**

* 对于单体架构来讲，我们可以不使用分布式，但是对于微服务架构来说，分布式几乎是必会用的技术，由于分布式本身的复杂性，导致微服务架构也变得复杂起来。

**接口调整成本高**

* 比如，用户微服务是要被订单微服务和电影微服务所调用的，一旦用户微服务的接口发生大的变动，那么所有依赖它的微服务都要做相应的调整，由于微服务可能非常多，那么调整接口所造成的成本将会明显提高。

**重复劳动**

* 对于单体架构来讲，如果某段业务被多个模块所共同使用，我们便可以抽象成一个工具类，被所有模块直接调用，但是微服务却无法这样做，因为这个微服务的工具类是不能被其它微服务所直接调用的，从而我们便不得不在每个微服务上都建这么一个工具类，从而导致代码的重复。

## 8. 微服务开发框架

**目前微服务的开发框架，最常用的有以下四个：**

1. Spring Cloud：http://projects.spring.io/spring-cloud（现在非常流行的微服务架构）
2. Dubbo：http：//dubbo.io
3. Dropwizard：http://www.dropwizard.io （关注单个微服务的开发）
4. Consul、etcd&etc.（微服务的模块）

## 9. Sprint cloud 和 Sprint boot区别

Spring Boot:

旨在简化创建产品级的Spring应用和服务，简化了配置文件，使用嵌入式web服务器，含有诸多开箱即用微服务功能，可以和spring cloud联合部署。

Spring Cloud：

微服务工具包，为开发者提供了在分布式系统的配置管理、服务发现、断路器、智能路由、微代理、控制总线等开发工具包。

# 二、微服务实践先知

## 1. 客户端如何访问这些服务？（API Gateway）

传统的开发方式，所有的服务都是本地的，UI可以直接调用，现在按功能拆分成独立的服务，跑在独立的一般都在独立的虚拟机上的 Java进程了。客户端UI如何访问他的？后台有N个服务，前台就需要记住管理N个服务，一个服务下线/更新/升级，前台就要重新部署，这明显不服务我们 拆分的理念，特别当前台是移动应用的时候，通常业务变化的节奏更快。另外，N个小服务的调用也是一个不小的网络开销。还有一般微服务在系统内部，通常是无状态的，用户登录信息和权限管理最好有一个统一的地方维护管理（OAuth）。

所以，一般在后台N个服务和UI之间一般会一个代理或者叫API Gateway，他的作用包括

* 提供统一服务入口，让微服务对前台透明
* 聚合后台的服务，节省流量，提升性能
* 提供安全，过滤，流控等API管理功能
* 我的理解其实这个API Gateway可以有很多广义的实现办法，可以是一个软硬一体的盒子，也可以是一个简单的MVC框架，甚至是一个Node.js的服务端。他们最重要的作用是为前台（通常是移动应用）提供后台服务的聚合，提供一个统一的服务出口，解除他们之间的耦合，不过API Gateway也有可能成为单点故障点或者性能的瓶颈。

## 2. 服务之间如何通信？（服务调用）

因为所有的微服务都是独立的Java进程跑在独立的虚拟机上，所以服务间的通信就是IPC（inter process communication），已经有很多成熟的方案。现在基本最通用的有两种方式。这几种方式，展开来讲都可以写本书，而且大家一般都比较熟悉细节了， 就不展开讲了。

* REST（JAX-RS，Spring Boot）
* RPC（Thrift, Dubbo）
* 异步消息调用(Kafka, Notify)

一般**同步调用**比较简单，一致性强，但是容易出调用问题，性能体验上也会差些，特别是调用层次多的时候。RESTful和RPC的比较也是一个很有意 思的话题。一般REST基于HTTP，更容易实现，更容易被接受，服务端实现技术也更灵活些，各个语言都能支持，同时能跨客户端，对客户端没有特殊的要 求，只要封装了HTTP的SDK就能调用，所以相对使用的广一些。RPC也有自己的优点，传输协议更高效，安全更可控，特别在一个公司内部，如果有统一个的开发规范和统一的服务框架时，他的开发效率优势更明显些。就看各自的技术积累实际条件，自己的选择了。

而**异步消息**的方式在分布式系统中有特别广泛的应用，他既能减低调用服务之间的耦合，又能成为调用之间的缓冲，确保消息积压不会冲垮被调用方，同时能 保证调用方的服务体验，继续干自己该干的活，不至于被后台性能拖慢。不过需要付出的代价是一致性的减弱，需要接受数据最终一致性；还有就是后台服务一般要 实现幂等性，因为消息发送出于性能的考虑一般会有重复（保证消息的被收到且仅收到一次对性能是很大的考验）；最后就是必须引入一个独立的broker，如 果公司内部没有技术积累，对broker分布式管理也是一个很大的挑战。

## 3. 这么多服务怎么查找？（服务发现）

在微服务架构中，一般每一个服务都是有多个拷贝，来做负载均衡。一个服务随时可能下线，也可能应对临时访问压力增加新的服务节点。服务之间如何相互 感知？服务如何管理？这就是服务发现的问题了。一般有两类做法，也各有优缺点。基本都是通过zookeeper等类似技术做服务注册信息的分布式管理。当 服务上线时，服务提供者将自己的服务信息注册到ZK（或类似框架），并通过心跳维持长链接，实时更新链接信息。服务调用者通过ZK寻址，根据可定制算法，找到一个服务，还可以将服务信息缓存在本地以提高性能。当服务下线时，ZK会发通知给服务客户端。

**客户端做：**优点是架构简单，扩展灵活，只对服务注册器依赖。缺点是客户端要维护所有调用服务的地址，有技术难度，一般大公司都有成熟的内部框架支持，比如Dubbo。

**服务端做：**优点是简单，所有服务对于前台调用方透明，一般在小公司在云服务上部署的应用采用的比较多。

## 4. 服务挂了怎么办？

分布式最大的特性就是网络是不可靠 的。通过微服务拆分能降低这个风险，不过如果没有特别的保障，结局肯定是噩梦。我们刚遇到一个线上故障就是一个很不起眼的SQL计数功能，在访问量上升 时，导致数据库load彪高，影响了所在应用的性能，从而影响所有调用这个应用服务的前台应用。所以当我们的系统是由一系列的服务调用链组成的时候，我们必须确保任一环节出问题都不至于影响整体链路。相应的手段有很多：

* 重试机制
* 限流
* 熔断机制
* 负载均衡
* 降级（本地缓存） 这些方法基本上都很明确通用，就不详细说明了。比如Netflix的Hystrix：https://github.com/Netflix/Hystrix

## 5. 微服务需要考虑的问题

这里有一个图非常好的总结微服务架构需要考虑的问题，包括

* API Gateway
* 服务间调用
* 服务发现
* 服务容错
* 服务部署
* 数据调用

# 三、微服务重要部件

## 1. 微服务基本能力

## 2. 服务注册中心

服务之间需要创建一种服务发现机制，用于帮助服务之间互相感知彼此的存在。服务启动时会将自身的服务信息注册到注册中心，**并订阅自己需要消费的服务**。

服务注册中心是服务发现的核心。它保存了各个可用服务实例的网络地址（IPAddress和Port）。服务注册中心必须要有高可用性和实时更新功能。上面提到的 Netflix Eureka 就是一个服务注册中心。它提供了服务注册和查询服务信息的REST API。服务通过使用POST请求注册自己的IPAddress和Port。每30秒发送一个PUT请求刷新注册信息。通过DELETE请求注销服务。客户端通过GET请求获取可用的服务实例信息。 Netflix的高可用（Netflix achieves high availability ）是通过在Amazon EC2运行多个实例来实现的,每一个Eureka服务都有一个弹性IP Address。当Eureka服务启动时，有DNS服务器动态的分配。Eureka客户端通过查询 DNS来获取Eureka的网络地址（IP Address和Port）。一般情况下，都是返回和客户端在同一个可用区Eureka服务器地址。 其他能够作为服务注册中心的有：

* etcd —– 高可用，分布式，强一致性的，key-value，Kubernetes和Cloud Foundry都是使用了etcd。
* consul —–一个用于discovering和configuring的工具。它提供了允许客户端注册和发现服务的API。Consul可以进行服务健康检查，以确定服务的可用性。
* zookeeper —— 在分布式应用中被广泛使用，高性能的协调服务。 Apache Zookeeper 最初为Hadoop的一个子项目，但现在是一个顶级项目。

### 2.1 zookeeper服务注册和发现

简单来讲，zookeeper可以充当一个服务注册表（Service Registry），让多个服务提供者形成一个集群，让服务消费者通过服务注册表获取具体的服务访问地址（ip+端口）去访问具体的服务提供者。如下图所示：

具体来说，zookeeper就是个分布式文件系统，每当一个服务提供者部署后都要将自己的服务注册到zookeeper的某一路径上: /{service}/{version}/{ip:port}, 比如我们的HelloWorldService部署到两台机器，那么zookeeper上就会创建两条目录：分别为/HelloWorldService/1.0.0/100.19.20.01:16888 /HelloWorldService/1.0.0/100.19.20.02:16888。

zookeeper提供了“心跳检测”功能，它会定时向各个服务提供者发送一个请求（实际上建立的是一个 socket 长连接），如果长期没有响应，服务中心就认为该服务提供者已经“挂了”，并将其剔除，比如100.19.20.02这台机器如果宕机了，那么zookeeper上的路径就会只剩/HelloWorldService/1.0.0/100.19.20.01:16888。

服务消费者会去监听相应路径（/HelloWorldService/1.0.0），一旦路径上的数据有任务变化（增加或减少），zookeeper都会通知服务消费方服务提供者地址列表已经发生改变，从而进行更新。

更为重要的是zookeeper 与生俱来的容错容灾能力（比如leader选举），可以确保服务注册表的高可用性。

## 3. 负载均衡

服务高可用的保证手段，为了保证高可用，每一个微服务都需要部署多个服务实例来提供服务。此时客户端进行服务的负载均衡。

### 3.1 负载均衡的常见策略

#### 3.1.1 随机

把来自网络的请求随机分配给内部中的多个服务器。

#### 3.1.2 轮询

每一个来自网络中的请求，轮流分配给内部的服务器，从1到N然后重新开始。此种负载均衡算法适合服务器组内部的服务器都具有相同的配置并且平均服务请求相对均衡的情况。

#### 3.1.3 加权轮询

根据服务器的不同处理能力，给每个服务器分配不同的权值，使其能够接受相应权值数的服务请求。例如：服务器A的权值被设计成1，B的权值是3，C的权值是6，则服务器A、B、C将分别接受到10%、30％、60％的服务请求。此种均衡算法能确保高性能的服务器得到更多的使用率，避免低性能的服务器负载过重。

#### 3.1.4 IP Hash

这种方式通过生成请求源IP的哈希值，并通过这个哈希值来找到正确的真实服务器。这意味着对于同一主机来说他对应的服务器总是相同。使用这种方式，你不需要保存任何源IP。但是需要注意，这种方式可能导致服务器负载不平衡。

#### 3.1.5 最少连接数

客户端的每一次请求服务在服务器停留的时间可能会有较大的差异，随着工作时间加长，如果采用简单的轮循或随机均衡算法，每一台服务器上的连接进程可能会产生极大的不同，并没有达到真正的负载均衡。最少连接数均衡算法对内部中需负载的每一台服务器都有一个数据记录，记录当前该服务器正在处理的连接数量，当有新的服务连接请求时，将把当前请求分配给连接数最少的服务器，使均衡更加符合实际情况，负载更加均衡。此种均衡算法适合长时处理的请求服务，如FTP。

## 4. 容错

容错，这个词的理解，直面意思就是可以容下错误，不让错误再次扩张，让这个错误产生的影响在一个固定的边界之内，“千里之堤毁于蚁穴”我们用容错的方式就是让这种蚁穴不要变大。那么我们常见的降级，限流，熔断器，超时重试等等都是容错的方法。

在调用服务集群时，如果一个微服务调用异常，如超时，连接异常，网络异常等，则根据容错策略进行服务容错。目前支持的服务容错策略有快速失败，失效切换。如果连续失败多次则直接熔断，不再发起调用。这样可以避免一个服务异常拖垮所有依赖于他的服务。

### 4.1 容错策略

#### 4.1.1 快速失败

服务只发起一次待用，失败立即报错。通常用于非幂等下性的写操作

#### 4.1.2 失效切换

服务发起调用，当出现失败后，重试其他服务器。通常用于读操作，但重试会带来更长时间的延迟。重试的次数通常是可以设置的

#### 4.1.3 失败安全

失败安全， 当服务调用出现异常时，直接忽略。通常用于写入日志等操作。

#### 4.1.4 失败自动恢复

当服务调用出现异常时，记录失败请求，定时重发。通常用于消息通知。

#### 4.1.5 forking Cluster

并行调用多个服务器，只要有一个成功，即返回。通常用于实时性较高的读操作。可以通过forks=n来设置最大并行数。

#### 4.1.6 广播调用

广播调用所有提供者，逐个调用，任何一台失败则失败。通常用于通知所有提供者更新缓存或日志等本地资源信息。

## 5. 熔断

熔断技术可以说是一种“智能化的容错”，当调用满足失败次数，失败比例就会触发熔断器打开，有程序自动切断当前的RPC调用,来防止错误进一步扩大。实现一个熔断器主要是考虑三种模式，关闭，打开，半开。各个状态的转换如下图。

我们在处理异常的时候，要根据具体的业务情况来决定处理方式，比如我们调用商品接口，对方只是临时做了降级处理，那么作为网关调用就要切到可替换的服务上来执行或者获取托底数据，给用户友好提示。还有要区分异常的类型，比如依赖的服务崩溃了，这个可能需要花费比较久的时间来解决。也可能是由于服务器负载临时过高导致超时。作为熔断器应该能够甄别这种异常类型，从而根据具体的错误类型调整熔断策略。增加手动设置，在失败的服务恢复时间不确定的情况下，管理员可以手动强制切换熔断状态。最后，熔断器的使用场景是调用可能失败的远程服务程序或者共享资源。如果是本地缓存本地私有资源，使用熔断器则会增加系统的额外开销。还要注意，熔断器不能作为应用程序中业务逻辑的异常处理替代品。

有一些异常比较顽固，突然发生，无法预测，而且很难恢复，并且还会导致级联失败（举个例子，假设一个服务集群的负载非常高，如果这时候集群的一部分挂掉了，还占了很大一部分资源，整个集群都有可能遭殃）。如果我们这时还是不断进行重试的话，结果大多都是失败的。因此，此时我们的应用需要立即进入失败状态(fast-fail)，并采取合适的方法进行恢复。

我们可以用状态机来实现CircuitBreaker，它有以下三种状态：

* 关闭( Closed )：默认情况下Circuit Breaker是关闭的，此时允许操作执行。CircuitBreaker内部记录着最近失败的次数，如果对应的操作执行失败，次数就会续一次。如果在某个时间段内，失败次数（或者失败比率）达到阈值，CircuitBreaker会转换到开启( Open )状态。在开启状态中，Circuit Breaker会启用一个超时计时器，设这个计时器的目的是给集群相应的时间来恢复故障。当计时器时间到的时候，CircuitBreaker会转换到半开启( Half-Open )状态。
* 开启( Open )：在此状态下，执行对应的操作将会立即失败并且立即抛出异常。
* 半开启( Half-Open )：在此状态下，Circuit Breaker会允许执行一定数量的操作。如果所有操作全部成功，CircuitBreaker就会假定故障已经恢复，它就会转换到关闭状态，并且重置失败次数。如果其中 任意一次 操作失败了，Circuit Breaker就会认为故障仍然存在，所以它会转换到开启状态并再次开启计时器（再给系统一些时间使其从失败中恢复）

## 6. 限流和降级

保证核心服务的稳定性。为了保证核心服务的稳定性，随着访问量的不断增加，需要为系统能够处理的服务数量设置一个极限阀值，超过这个阀值的请求则直接拒绝。同时，为了保证核心服务的可用，可以对某些非核心服务进行降级，通过限制服务的最大访问量进行限流，通过管理控制台对单个微服务进行人工降级

## 7. SLA

SLA：Service-LevelAgreement的缩写，意思是服务等级协议。 是关于网络服务供应商和客户间的一份合同，其中定义了服务类型、服务质量和客户付款等术语。 典型的SLA包括以下项目：

* 分配给客户的最小带宽；
* 客户带宽极限；
* 能同时服务的客户数目；
* 在可能影响用户行为的网络变化之前的通知安排；
* 拨入访问可用性；
* 运用统计学；
* 服务供应商支持的最小网络利用性能，如99.9%有效工作时间或每天最多为1分钟的停机时间；
* 各类客户的流量优先权；
* 客户技术支持和服务；
* 惩罚规定，为服务供应商不能满足 SLA需求所指定。

## 8. API网关

这里说的网关是指API网关，直面意思是将所有API调用统一接入到API网关层，有网关层统一接入和输出。一个网关的基本功能有：统一接入、安全防护、协议适配、流量管控、长短链接支持、容错能力。有了网关之后，各个API服务提供团队可以专注于自己的的业务逻辑处理，而API网关更专注于安全、流量、路由等问题。

## 9. 多级缓存

最简单的缓存就是查一次数据库然后将数据写入缓存比如redis中并设置过期时间。因为有过期失效因此我们要关注下缓存的穿透率，这个穿透率的计算公式，比如查询方法queryOrder(调用次数1000/1s)里面嵌套查询DB方法queryProductFromDb(调用次数300/s)，那么redis的穿透率就是300/1000,在这种使用缓存的方式下，是要重视穿透率的，穿透率大了说明缓存的效果不好。还有一种使用缓存的方式就是将缓存持久化，也就是不设置过期时间，这个就会面临一个数据更新的问题。一般有两种办法，一个是利用时间戳，查询默认以redis为主，每次设置数据的时候放入一个时间戳，每次读取数据的时候用系统当前时间和上次设置的这个时间戳做对比，比如超过5分钟，那么就再查一次数据库。这样可以保证redis里面永远有数据，一般是对DB的一种容错方法。还有一个就是真正的让redis做为DB使用。就是图里面画的通过订阅数据库的binlog通过数据异构系统将数据推送给缓存，同时将将缓存设置为多级。可以通过使用jvmcache作为应用内的一级缓存，一般是体积小，访问频率大的更适合这种jvmcache方式，将一套redis作为二级remote缓存，另外最外层三级redis作为持久化缓存。

## 10. 超时和重试

超时与重试机制也是容错的一种方法，凡是发生RPC调用的地方，比如读取redis，db，mq等，因为网络故障或者是所依赖的服务故障，长时间不能返回结果，就会导致线程增加，加大cpu负载，甚至导致雪崩。所以对每一个RPC调用都要设置超时时间。对于强依赖RPC调用资源的情况，还要有重试机制，但是重试的次数建议1-2次，另外如果有重试，那么超时时间就要相应的调小，比如重试1次，那么一共是发生2次调用。如果超时时间配置的是2s，那么客户端就要等待4s才能返回。因此重试+超时的方式，超时时间要调小。这里也再谈一下一次PRC调用的时间都消耗在哪些环节，一次正常的调用统计的耗时主要包括： ①调用端RPC框架执行时间 + ②网络发送时间 + ③服务端RPC框架执行时间 + ④服务端业务代码时间。调用方和服务方都有各自的性能监控，比如调用方tp99是500ms，服务方tp99是100ms，找了网络组的同事确认网络没有问题。那么时间都花在什么地方了呢，两种原因，客户端调用方，还有一个原因是网络发生TCP重传。所以要注意这两点。

# 11. 线程池隔离

在抗量这个环节，Servlet3异步的时候，有提到过线程隔离。线程隔离的之间优势就是防止级联故障，甚至是雪崩。当网关调用N多个接口服务的时候，我们要对每个接口进行线程隔离。比如，我们有调用订单、商品、用户。那么订单的业务不能够影响到商品和用户的请求处理。如果不做线程隔离，当访问订单服务出现网络故障导致延时，线程积压最终导致整个服务CPU负载满。就是我们说的服务全部不可用了，有多少机器都会被此刻的请求塞满。那么有了线程隔离就会使得我们的网关能保证局部问题不会影响全局。

# 12. 降级和限流

关于降级限流的方法业界都已经有很成熟的方法了，比如FAILBACK机制，限流的方法令牌桶，漏桶，信号量等。这里谈一下我们的一些经验，降级一般都是由统一配置中心的降级开关来实现的，那么当有很多个接口来自同一个提供方，这个提供方的系统或这机器所在机房网络出现了问题，我们就要有一个统一的降级开关，不然就要一个接口一个接口的来降级。也就是要对业务类型有一个大闸刀。还有就是 降级切记暴力降级，什么是暴力降级的，比如把论坛功能降调，结果用户显示一个大白板，我们要实现缓存住一些数据，也就是有托底数据。限流一般分为分布式限流和单机限流，如果实现分布式限流的话就要一个公共的后端存储服务比如redis，在大nginx节点上利用lua读取redis配置信息。我们现在的限流都是单机限流，并没有实施分布式限流。

# 13. 网关监控和统计

API网关是一个串行的调用，那么每一步发生的异常要记录下来，统一存储到一个地方比如elasticserach中，便于后续对调用异常的分析。鉴于公司docker申请都是统一分配，而且分配之前docker上已经存在3个agent了，不再允许增加。我们自己实现了一个agent程序，来负责采集服务器上面的日志输出，然后发送到kafka集群，再消费到elasticserach中，通过web查询。现在做的追踪功能还比较简单，这块还需要继续丰富。