**微服务**

## 一、微服务架构风格

**1 微服务是一种架构风格**，一个大型复杂软件应用由一个或多个微服务组成。**系统中的各个微服务可被独立部署，各个微服务之间是松耦合的**。每个微服务仅关注于完成一件任务并很好地完成该任务。在所有情况下，每个任务代表着一个小的业务能力。尽管“微服务”这种架构风格没有精确的定义，但其具有一些共同的特性，如围绕业务能力组织服务、自动化部署、智能端点、对语言及数据的“去集中化”控制等等。

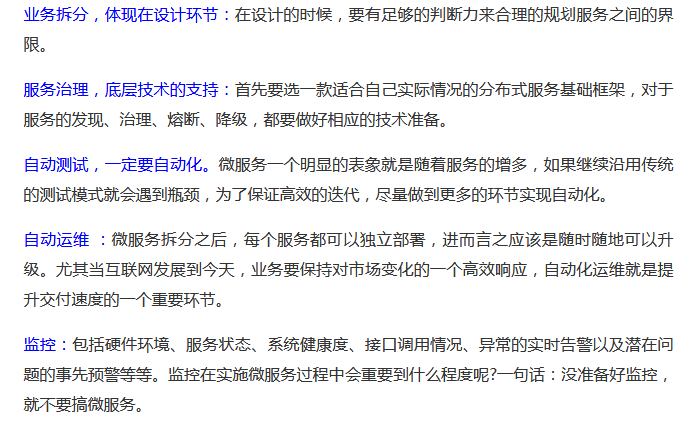
**2 微服务最早由Martin Fowler与James Lewis于2014年共同提出**，微服务架构风格是一种使用一套小服务来开发单个应用的方式途径，每个服务运行在自己的进程中，并使用轻量级机制通信，通常是HTTP API，这些服务基于业务能力构建，并能够通过自动化部署机制来独立部署，这些服务使用**不同的编程语言实现，以及不同数据存储技术，并保持最低限度的集中式管理**。

**3 每个微服务可独立运行在自己的进程里**

**一系列独立运行的微服务共同构建起了整个系统；**

**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；**

**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。**



**4 单体架构问题：**

*@.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。

*@.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。

*@.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。

*@.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。

@.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。

**5 微服务与单体架构区别**

@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。

@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。

@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

**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。

**6 微服务四个要素：**

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。

**7 微服务折分与设计**

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。

**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。**

**8 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。

**9 微服务优点**

**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。

**10 微服务缺点**

**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。

**11 微服务开发框架**

**Spring Cloud：**http://projects.spring.io/spring-cloud（现在非常流行的微服务架构）

**Dubbo**：http：//dubbo.io

**Dropwizard**：http://www.dropwizard.io （关注单个微服务的开发）

**Consul、etcd&etc.**（微服务的模块）

**12 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

**13 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

**@REST（JAX-RS，Spring Boot）**

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（Thrift, Dubbo）**

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。

**@异步消息调用(Kafka, Notify)**

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.

**@Webservice**

**@Httpclient**

**14 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。

**15 服务挂了**

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）

**16 服务注册**

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。

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。

**17 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。

**18 负载均衡策略**

**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。

**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 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。

**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。

**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。

**19 容错机制**

**（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

**（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

**（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）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。

**（5）并行集群，**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。

**（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。

**20 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。**

我们可以用状态机来实现熔断，它有以下三种状态：

**@ 关闭( Closed )：**默认情况下CircuitBreaker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，CircuitBreaker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）

**21 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，可以对否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。

**22 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。

## 二、微服务架构精华

**dubbo网关**，身份认证，拦截，不用修改所有的app，handlerInterceptor

webmvcConfigureAdapter

**dubbo负载均衡**，loadBalance接口自己实现，优先级～admin界面配>reference>service。随机，最少活跃，轮训，一致性hash

**服务降级**，通过mock，远程失败时直接返回指定的逻辑。

**服务熔断，**指定时间失败次数大于阈值，进去断开状态，熔断

机制是应对雪崩效应的一种微服务链路保护机制。某个微服务

不可用或者响应时间太长时，会进行服务的降级，进而熔断该

节点微服务的调用，快速返回错误的响应信息。当定时检测到

该节点微服务调用响应正常后，恢复调用链路。

**服务限流**，根据机器硬件配置设置最大访问量。一旦达到限定

速率拒绝服务，定向到错误页面。基于令牌桶，redis实战限流。

**服务容错**，dubbo提供快速失败，安全失败等

**ribbon**负载均衡基于http和tcp客户端的，rule，配置bean返回IRule，包括最大可用，加权，轮训，随机

**Zuul**

enableZuulProxy，配置path拦截路径，继承ZuulFilter，run方法，和handler拦截器原理一样的。

**Hystrix**

从框架级别解决送错和降级，mock从功能级别。enableCircuitBreak，在消费端配置hystrixCommand，fallbackmethod，在application.pro配置超时时间，错误比率阈值等参数

**spring cloud admin**

服务监控，应用监控，断路器监控

**spring cloud config**

git - config server - 模块abcd和消息总线实战热部署。

**spring data mysql es redis**

redisTemplete 实战分布式锁，根据key值判断redis是否存在？不存在true，存在并发问题，可以放在阻塞队列，还要定时获取结果字段，实现崩溃转移机制，把任务转交其他实例。

还有一种方案是quartz分布式定时任务，内置分布式锁，自带崩溃转移功能。

**表单验证**

spring boot validation

**分布式会话**

http无状态，发起http，生成sid存续cookie，tomcat将sid存在内存，分布式下无法打通，spring session redis重写getsesion方法，交给redis存储，统一分布式会话

**消息队列**

springboot amqp封装了rabbitMQ，开箱急用，@enableRabbit，消费端用ribbatListener（quenes指定topic）注解接受消息，生产端使用rabbitTemplate.send发送

**分布式异常处理**

依赖json序列化依赖，阿里的fastjson，exceptionEntity有code，message，errer，时间，新建自定义异常，basicException继承runtime，business继承basic，新建error.ftl模板，全局异常控制类，@controllAdvice作用类表示处理全局异常。@exceptionHandler（value）作用方法上，对指定异常处理，bindExecption是hibernate表单验证异常。

**分布式安全认证**

dubbo中使用RpcFilter实战白名单机制。app，web，第三方，网关拦截处理有些简单，使用spring security Oauth2，1.0版本是允许第三方移动访问不需要传密码，2.0同时为web，桌面，手机认证。资源所有者，资源服务器，客户端，授权服务器。dubbo使用

**oauth2，**

springcloud使用oauth2，zuul只是转发了请求不是资源实际拥有者，授权中心负责生成和验证token。Token令牌，服务端生成的一串字符串，用户第一次登录后，服务器生成一个token并将此token返回给客户端，以后客户端只需带上这个token前来请求数据即可，无需再次带上用户名和密码。简单Token的组成：uid(用户唯一的身份标识)、time(当前时间的时间戳)、sign（token的前几位以哈希算法压缩成的一定长度的十六进制字符串。为防止token泄露）

**分布式日志管理**

springboot推荐使用logback，新一代日志框架取代lig4j，适用于分布式，效率高。但是还是slf4j的门面设计模式。单机只记录自己的日志，分布式日志用elk极大的支持，模块abcde...

logstash... es ...kibana

可以在logstash配置收集，也可以springboot配置logstash路径直接输出。

**热部署**

springboot基于maven引入devtools监控，在app.pro中配置排除的目录，和是否开启目录

**接口文档管理**

dubbo和springcloud都使用swagger2，分成json和ui的形式展现

**分布式架构图～**

web 手机 第三方

！

网关模块

！

授权中心

！

注册中心ha --监控治理熔断

！ ！ ！

模块a 模块b 模块c

消息 缓存 私服 构建 事务 日志 配置 文档 异常