微服务架构的核心要点和实现原理

目录

[1. 微服务架构中职能团队的划分 1](#_Toc32449)

[1.1. 微服务的去中心化治理 2](#_Toc11952)

[2. 微服务的交互模式 2](#_Toc14287)

[2.1. 1. 读者容错模式 3](#_Toc25881)

[2.2. 2. 消费者驱动契约模式 3](#_Toc2299)

[2.3. 3. 去数据共享模式 4](#_Toc18695)

[3. 组合微服务呢？ 5](#_Toc14586)

[3.1. 1. 服务代理模式 5](#_Toc32589)

[3.2. 2. 服务聚合模式 6](#_Toc13954)

[3.3. 3. 服务串联模式 7](#_Toc19529)

[3.4. 4. 服务分支模式 8](#_Toc2799)

[3.5. 5. 服务异步消息模式 10](#_Toc30842)

[4. 微服务的容错模式 11](#_Toc30481)

[4.1. 1. 舱壁隔离模式 11](#_Toc6802)

[4.1.1. 1）微服务容器分组 11](#_Toc13164)

[4.1.2. 2）线程池隔离 12](#_Toc31547)

[4.2. 2. 熔断模式 12](#_Toc22380)

[4.3. 3. 限流模式 13](#_Toc10220)

[4.3.1. 1）计数器 13](#_Toc18276)

[4.3.2. 2）令牌筒 14](#_Toc13697)

[4.3.3. 3）信号量 14](#_Toc18254)

[4.4. 4. 失效转移模式 16](#_Toc30832)

# 微服务架构中职能团队的划分

传统单体架构将系统分成具有不同职责的层次，对应的项目管理也倾向于将大的团队分成不同的职能团队，主要包括：用户交互UI团队、后台业务逻辑处理团队与数据存取ORM团队、DBA团队等。每个团队只对自己分层的职责负责，并对使用方提供组件服务质量保证。如果其中一个模块化组件需要升级、更新，那么这个变更会涉及不同的分层团队，即使升级和变更的改变很小，也需要进行跨团队沟通：需求阶段需要跨团队沟通产品功能，设计阶段需要跨团队沟通设计方案，开发阶段需要跨团队沟通具体的接口定义，测试阶段需要沟通业务回归等事宜，甚至上线都需要跨团队沟通应用的上线顺序。可见在传统的整体架构下，后期的维护成本很高，出现事故的风险很大。

根据康威定律，团队的交流机制应该与架构设计机制相对应。因此，在微服务架构下，职能团队的划分方法是我们首先要考虑的一个核心要素。

微服务时代的团队沟通方式如图1-9所示。

图1-9

微服务架构按照业务的功能进行划分，每个单一的业务功能叫作一个服务，每个服务对应一个独立的职能团队，团队里包含用户交互UI设计师、后台服务开发人员、DBA、运营和运维人员。

在传统的整体架构中，软件是有生命周期的，经历需求分析、开发和测试，然后被交付给运维团队，这时开发团队将会解散，这是对软件的一个“放手”。而在微服务架构中，提倡运维人员也是服务项目团队的一员，倡导谁开发、谁维护，实施终生维护制度。

在业务服务的内部实现需要升级或者变更时，团队内的各角色成员进行沟通即可，而不需要进行跨团队沟通，这大大提高了沟通效率。只有服务之间的接口需要变更时才需要跨部门沟通，如果前期在服务之间的交互上定义了良好的接口，则接口变更的概率并不大，即使接口模式有变更，也可以通过一定的设计模式和规范来解决，可参考1.3.3节。

## 微服务的去中心化治理

笔者曾经在一个互联网平台上工作，平台的决策者倡导建设API网关，所有外部服务和内部服务都由统一的API网关进行管理。在项目初期，中心化的API网关统一了所有API的入口，这看起来很规范，但从技术角度来看限制了API的多样化。随着业务的发展，API网关开始暴露问题，每个用户请求经过机房时只要有服务之间的交互，则都会从API网关进行路由，服务上量以后，由于内部服务之间的交互都会叠加在API网关的调用上，所以在很大程度上放大了API网关的调用TPS，API网关很快就遇到了性能瓶颈。

这个案例是典型的微服务的反模式，微服务倡导去中心化的治理，不推荐每个微服务都使用相同的标准和技术来开发和使用服务。在微服务架构下可以使用C++开发一个服务，来对接Java开发的另外一个服务，对于异构系统之间的交互标准，通常可以使用工具来补偿。开发者可以开发共用的工具，并分享给异构系统的开发者使用，来解决异构系统不一致的问题，例如：Thrift远程调用框架使用中间语言（IDL）来定义接口，中间语言是独立于任何语言的，并提供了工具来生成中间语言，以及在中间语言与具体语言之间的代码转换。

微服务架构倡导去中心化的服务管理和治理，尽量不设置中心化的管理服务，最差也需要在中心化的管理服务宕机时有替代方案和设计。在笔者工作的支付平台服务化建设中，第1层SOA服务化采用Dubbo框架进行定制化，如果Dubbo服务化出现了大面积的崩溃，则服务化体系会切换到点对点的hessian远程调用，这被称为服务化降级，降级后点对点的hessian远程调用时没有中心化节点，整体上符合微服务的原理。

# 微服务的交互模式

本节介绍微服务之间交互的通用设计模式，这些设计模式对微服务之间的交互定义契约，服务的生产者和调用者都需要遵守这些契约，才能保证微服务不出问题。

## 1. 读者容错模式

读者容错模式（Tolerant Reader）指微服务化中服务提供者和消费者之间如何对接口的改变进行容错。从字面上来讲，消费者需要对提供者提供的功能进行兼容性设计，尤其对服务提供者返回的内容进行兼容，或者解决在服务提供者改变接口或者数据的格式的情况下，如何让服务消费者正常运行。

任何一个产品在设计时都无法预见将来可能增加的所有需求，服务的开发者通常通过迭代及时地增加新功能，或者让服务提供的API自然地演进。不过，服务提供者对外提供的接口的数据格式的改变、增加和删除，都会导致服务的消费者不能正常工作。

因此，在服务消费者处理服务提供者返回的消息的过程中，需要对服务返回的消息进行过滤，只提取自己需要的内容，对多余或者未知的内容采取抛弃的策略，而不是硬生生地抛错处理。

在实现过程中不推荐使用严格的校验策略，而是推荐使用宽松的校验策略，即使服务消费者拿到的消息报文发生了改变，程序也只需尽最大努力提取需要的数据，同时忽略不可识别的数据。只有在服务消费者完全不能识别接收到的消息，或者无法通过识别的信息继续处理流程时，才能抛出异常。

服务的消费者的容错模式忽略了新的消息项、可选的消息项、未知的数据值及服务消费者不需要的数据项。

笔者当前在某个支付公司工作，公司里几乎每个业务都需要使用枚举类型，在微服务平台下，笔者在研发流程规范中定义了一条枚举值使用规范：

在服务接口的定义中，参数可以使用枚举值，在返回值的DTO中禁止使用枚举值。

这条规范就是读者容错模式在实践中的一个实例，之所以在参数中允许使用枚举值，是因为如果服务的提供者升级了接口，增加了枚举值，若服务的消费者并没有感知，则服务的消费者得知新的枚举值就可以传递新的枚举值了；但是如果接口的返回DTO中使用了枚举值，并且因为某种原因增加了枚举值，则服务消费者在反序列化枚举时就会报错，因此在返回值中我们应该使用字符串等互相认可的类型，来做到双方的互相兼容，并实现读者容错模式。

## 2. 消费者驱动契约模式

消费者驱动契约模式用来定义服务化中服务之间交互接口改变的最佳规则。

服务契约分为：提供者契约、消费者契约及消费者驱动的契约，它从期望与约束的角度描述了服务提供者与服务消费者之间的联动关系。

提供者契约：是我们最常用的一种服务契约，顾名思义，提供者契约是以提供者为中心的，提供者提供了什么功能和消息格式，各消费者都会无条件地遵守这些约定，不论消费者实际需要多少功能，消费者接受了提供者契约时，都会根据服务提供者的规则来使用服务。

消费者契约：是对某个消费者的需求进行更为精确的描述，在一次具体的服务交互场景下，代表消费者需要提供者提供功能中的哪部分数据。消费者契约可以被用来标识现有的提供者契约，也可以用来发现一个尚未明确的提供者契约。

消费者驱动的契约：代表服务提供者向其所有当前消费者承诺遵守的约束。一旦各消费者把自己的具体期望告知提供者，则提供者无论在什么时间和场景下，都不应该打破契约。

在现实的服务交互设计中，上面这三种契约是同时存在的，笔者所在的支付平台里，交易系统在完成一笔支付后，需要到账务系统为商户入账，在这个过程中，服务契约表现如下。

生产者契约：账务系统提供Dubbo服务化接口，参数为商户账户ID、入账订单号和入账金额。

消费者契约：账务系统返回DTO，包含商户账户ID、入账订单号、入账金额、入账时间、账务流水号、入账状态等，而交易系统只需使用其中的入账订单号和入账状态。

消费者驱动的契约：为了保证资金安全，交易系统作为入账的发起者向账务提出要求，需要账务做幂等和滤重处理，对重复的入账请求进行拦截；账务系统在接受这个契约后，即使将来有任何改变，也不能打破这个限制，否则就会造成资金的损失，这在金融系统中是最严重的问题。

服务之间的交互需要使用的三种服务契约如图1-10所示。

图1-10

从图1-10可以看到，服务提供者契约是服务提供者单方面定下的规则，而一个消费者契约会成为提供者契约的一部分，多个服务消费者可以对服务提供者提出约束，服务提供者需要在将来遵守服务消费者提出的契约，这就是消费者驱动的契约。

## 3. 去数据共享模式

与SOA服务化对比，微服务是去ESB总线、去中心化及分布式的；而SOA还是以ESB为核心实现遗留系统的集成，以及基于Web Service为标准实现的通用的面向服务的架构。在微服务领域，微服务之间的交互通过定义良好的接口来实现，不允许使用共享数据来实现。

在实践的过程中，有些方案的设计使用缓存或者数据库作为两个微服务之间的纽带，在业务流程的处理过程中，为了处理简单，前一个服务将中间结果存入数据库或者缓存，下一个服务从缓存或者数据库中拿出数据继续处理。处理流程如图1-11所示。

图1-11

这种交互流程的缺点如下。

使得微服务之间的交互除了接口契约，还存在数据存储契约。

上游的数据格式发生变化时，可能导致下游的处理逻辑出现问题。

多个服务共享一个资源服务，对资源服务的运维难以划清职责和界限。

在做双机房独立部署时，需要考虑服务和资源的路由情况，跨机房的服务调用不能使用独立的资源部署模式，因此难以实现服务自治。

因此，在设计微服务架构时，一定不要共享缓存和数据库等资源，也不要使用总线模式，服务之间的通信和交互只能依赖定义良好的接口，通常使用RESTful样式的API或者透明的RPC调用框架。

微服务的分解和组合模式

使用微服务架构划分服务和团队是微服务架构实施的重要一步，良好的划分和拆分使系统达到松耦合和高内聚的效果，然后通过微服务的灵活组装可以满足上层的各种各样的业务处理需求。

在微服务架构的需求分析和架构设计过程中，通常是用领域的动词和名词来划分微服务的，例如，对于一个电商后台系统，可以分解为订单、商品、商品目录、库存、购物车、交易、支付、发票、物流等子系统，每个名词和动词都可以是一个微服务，将这几个微服务组合在一起，就实现了电商平台用户购买商品的整个业务流。

这样拆分以后，系统具有敏捷性、灵活性、可伸缩性等，拆分后有多个高度自治的微服务，那么以什么方式

# 组合微服务呢？

## 1. 服务代理模式

服务代理模式是最简单的服务组合模式，它根据业务的需求选择调用后端的某个服务。在返回给使用端之前，代理可以对后端服务的输出进行加工，也可以直接把后端服务的返回结果返回给使用端。

服务代理模式的架构如图1-12所示。

图1-12

在笔者工作的微服务化架构平台下，经常会使用这种模式，典型的案例是做平滑的系统迁移，通常经历如下4个阶段。

在新老系统上双写。

迁移双写之前的历史遗留数据。

将读请求切换到新系统。

下调双写逻辑，只写新系统。

服务代理模式常常应用到第3步，一般会对读请求切换设计一个开关，开关打开时查询新系统，开关关闭时查询老系统。

迁移案例中开关的逻辑如图1-13所示。

图1-13

## 2. 服务聚合模式

服务聚合模式是最常用的服务组合模式，它根据业务流程处理的需要，以一定的顺序调用依赖的多个微服务，对依赖的微服务返回的数据进行组合、加工和转换，最后以一定的形式返回给使用方。

这里，每个被依赖的微服务都有自己的缓存和数据库，聚合服务本身可以有自己的数据存储，包括缓存和数据库等，也可以是简单的聚合，不需要持久化任何数据。

服务聚合模式的架构如图1-14所示。

图1-14

这里体现了DRY（Don’t Repeat Yourself）原则的设计理念，在设计或者构造应用时，最大限度地重用了现有的实现。假如一块业务逻辑由三个独立的逻辑块组成，每个独立的逻辑块可能有多个使用方，则DRY原则推荐将三个独立的逻辑块封装成三个独立运行的微服务，然后使用本节的服务聚合模式开发聚合服务，将三个独立的逻辑块聚合在一起提供给上层组合服务。这样的设计原则有如下好处。

三个独立的子服务可以各自独立开发、敏捷变更和部署。

聚合服务封装下层的业务处理服务，由三个独立的子服务完成数据持久化等工作，项目结构清晰明了。

三个独立的子服务对于其他使用方仍然可以重用。

考虑到本节开头的例子，在对微服务进行拆分时，将电商后台系统大致拆分成订单、商品、商品目录、库存、购物车、交易、支付、发票、物流等微服务，那么电商平台的前端应用就是后端各个微服务的一个最大的聚合服务，前端应用通过调用商品和商品目录显示商品列表，提供给用户选择商品的功能，用户选择商品后增加商品到购物车，在用户从购物车结算时，调用交易系统完成交易和支付等。

电商前台的聚合模式的案例架构如图1-15所示。

图1-15

另外，聚合服务也可以是一个纯后台服务，通过聚合对使用方输出组合的服务，例如在上面的电商系统案例中，在用户选择结算后，系统调用交易，交易系统会调用库存系统锁库存，然后创建交易订单，引导用户去支付，支付成功后扣减库存，最后通过发票服务开具电子发票。

电商后台交易服务的聚合模式架构如图1-16所示。

图1-16

## 3. 服务串联模式

服务串联模式类似于一个工作流，最前面的服务1负责接收请求和响应使用方，串联服务后再与服务1交互，随后服务1与服务2交互，最后，从服务2产生的结果经过服务1和串联服务逐个处理后返回给使用方，如图1-17所示。

图1-17

服务串联模式之间的调用通常使用同步的RESTful风格的远程调用实现，注意，这种模式采用的是同步调用方式，在串联服务没有完成并返回之前，所有服务都会阻塞和等待，一个请求会占用一个线程来处理，因此在这种模式下不建议服务的层级太多，如果能用服务聚合模式代替，则优先使用服务聚合模式，而不是使用这种服务串联模式。

相对于服务聚合模式，服务串联模式有一个优点，即串联链路上再增加一个节点时，只要不是在串联服务的正后面增加，那么串联服务是无感知的。

在串联服务中调用链的最后端增加服务无感知的架构如图1-18所示。

图1-18

在上面提及的电商案例中，UI前端应用调用交易，交易调用商品库存系统锁定库存和扣减库存，使用的就是服务串联模式。

服务串联模式案例的架构如图1-19所示。

图1-19

## 4. 服务分支模式

服务分支模式是服务代理模式、服务聚合模式和服务串联模式相结合的产物。

分支服务可以拥有自己的数据库存储，调用多个后端的服务或者服务串联链，然后将结果进行组合处理再返回给客户端。分支服务也可以使用代理模式，简单地调用后端的某个服务或者服务链，然后将返回的数据直接返回给使用方。

服务分支模式的架构如图1-20所示。

图1-20

在实际的业务平台建设中，由于业务的复杂性，抽象的微服务可能有多层的依赖关系，依赖关系并不会太简单，经常呈现树形的分支结构。

以电商平台的支付服务架构为例，如图1-21所示。

图1-21

支付服务对接两个外部的支付网关，都要经过各自的支付渠道网关，同时支持账户余额支付，这个支付服务其实就是一个分支模式，在实际项目中这种服务分支模式很多。

笔者在构建支付平台时，由于大量地使用了服务分支模式，所以发现了一个比较有趣的现象，如下所述。

假设有一个基础服务，在服务分支模式的多个层次中对基础服务都有依赖，那么当基础服务的一台机器宕机时，假设基础服务有8台机器，则最后受影响的流量并不是1/8。假设基础服务6共有8台机器，服务1、服务3和服务5组成某服务的一个调用链，则调用链过程中会多次调用基础服务6。

具体服务的调用链示意图如图1-22所示。

图1-22

某天，基础服务6的8台机器中的1台宕机，按照常理，大家都认为只影响其中1/8的流量，而统计结果显示影响的业务结果竟然大于1/8。

仔细思考，造成这个结果的原因是调用链上有多个层次重复调用了基础服务，导致基础服务挂掉时影响的流量有累加效果，具体计算如下。

假设进入系统的流量为n，调用链从服务3开始调用服务6，服务3有1/8的流量失败，这时剩下的成功的流量为7/8 ×n，剩下的成功的流量继续走到服务5，服务5再次调用服务6，又有1/8的流量失败，剩下7/8 × 7/8× n。

假设基础服务资源池中的机器个数为i，一次挂掉的机器个数为j，一个调用链中调用x次基础服务，那么正确处理的流量的计算公式为：

假设允许的可用性波动率为a，求出底层服务一次宕机1台时最少应该配置的机器数为：

对公式进行转换：

由于一次只允许一台机器宕机：

所以得出需要设置的机器数量i为：

对于上面的案例，每次最多允许基础服务6宕机1台，在这种情况下需要保持可用性的波动率小于25%，一共有两层服务依赖基础服务6，通过上述公式计算得出：

i > 7.5

结果，至少为服务6部署9台机器，这样在1台机器宕机时，对可用性的波动性影响控制在25%以内。

由于分支模式放大了服务的依赖关系，因此在现实的微服务设计中尽量保持服务调用级别的简单，在使用服务组合和服务代理模式时，不要使用服务串联模式和服务分支模式，以保持服务依赖关系的清晰明了，这也减少了日后维护的工作量。

## 5. 服务异步消息模式

前面的所有服务组合模式都使用同步的RESTful风格的同步调用来实现，同步调用模式在调用的过程中会阻塞线程，如果服务提供方迟迟没有返回，则服务消费方会一直阻塞，在严重情况下会撑满服务的线程池，出现雪崩效应。

因此，在构建微服务架构系统时，通常会梳理核心系统的最小化服务集合，这些核心的系统服务使用同步调用，而其他核心链路以外的服务可以使用异步消息队列进行异步化。

服务异步消息模式的架构如图1-23所示。

图1-23

在图1-23中，聚合服务同步调用服务1和服务2，而服务2通过消息队列将异步消息传递给服务3和服务4。

典型的案例就是在电商系统中，交易完成后向物流系统发起消息通知，通知物流系统发货，如图1-24所示。

图1-24

6. 服务共享数据模式

服务共享数据模式其实是反模式，在1.3.3节中提出了去数据共享模式，由于去掉了数据共享，所以仅仅通过服务之间良好定义的接口进行交互和通信，使得每个服务都是自治的，服务本身和服务的团队包含全角色栈的技术和运营人员，这些人都是专业的人做专业的事，使沟通在团队内部解决，因此可以使效率最大化。

服务共享数据模式的架构如图1-25所示。

图1-25

然而，在下面两种场景下，我们仍然需要数据共享模式。

单元化架构

一些平台由于对性能有较高的要求，所以采用微服务化将服务进行拆分，通过网络服务进行通信，尽管网络通信的带宽已经很宽，但是还会有性能方面的损耗，在这种场景下，可以让不同的微服务共享一些资源，例如：缓存、数据库等，甚至可以将缓存和数据在物理拓扑上与微服务部署在一个物理机中，最大限度地减少网络通信带来的性能损耗，我们将这种方法称为“单元化架构”。

单元化架构的示意图如图1-26所示。

图1-26

遗留的整体服务

对于历史遗留的传统单体服务，我们在重构微服务的过程中，发现单体服务依赖的数据库表耦合在一起，对其拆分需要进行反规范化的处理，可能会造成数据一致性问题，在没有对其完全理解和有把握的前提下，会选择保持现状，让不同的微服务暂时共享数据存储。

除了上面提到的两个场景，任何场景都不能使用服务数据共享模式。

# 微服务的容错模式

在使用了微服务架构以后，整体的业务流程被拆分成小的微服务，并组合在一起对外提供服务，微服务之间使用轻量级的网络协议通信，通常是RESTful风格的远程调用。由于服务与服务的调用不再是进程内的调用，而是通过网络进行的远程调用，众所周知，网络通信是不稳定、不可靠的，一个服务依赖的服务可能出错、超时或者宕机，如果没有及时发现和隔离问题，或者在设计中没有考虑如何应对这样的问题，那么很可能在短时间内服务的线程池中的线程被用满、资源耗尽，导致出现雪崩效应。本节针对微服务架构中可能遇到的这些问题，讲解应该采取哪些措施和方案来解决。

## 1. 舱壁隔离模式

这里用航船的设计比喻舱壁隔离模式，若一艘航船遇到了意外事故，其中一个船舱进了水，则我们希望这个船舱和其他船舱是隔离的，希望其他船舱可以不进水，不受影响。在微服务架构中，这主要体现在如下两个方面。

### 1）微服务容器分组

笔者所在的支付平台应用了微服务，将微服务的每个节点的服务池分为三组：准生产环境、灰度环境和生产环境。准生产环境供内侧使用；灰度环境会跑一些普通商户的流量；大部分生产流量和VIP商户的流量则跑在生产环境中。这样，在一次比较大的重构过程中，我们就可以充分利用灰度环境的隔离性进行预验证，用普通商户的流量验证重构没有问题后，再上生产环境。

另外一个案例是一些社交平台将名人的自媒体流量全部路由到服务的核心池子中，而将普通用户的流量路由到另外一个服务池子中，有效隔离了普通用户和重要用户的负载。

其服务分组如图1-27所示。

图1-27

### 2）线程池隔离

在微服务架构实施的过程中，我们不一定将每个服务拆分到微小的力度，这取决于职能团队和财务的状况，我们一般会将同一类功能划分在一个微服务中，尽量避免微服务过细而导致成本增加，适可而止。

这样就会导致多个功能混合部署在一个微服务实例中，这些微服务的不同功能通常使用同一个线程池，导致一个功能流量增加时耗尽线程池的线程，而阻塞其他功能的服务。

线程池隔离如图1-28所示。

图1-28

## 2. 熔断模式

可以用家里的电路保险开关来比喻熔断模式，如果家里的用电量过大，则电路保险开关就会自动跳闸，这时需要人工找到用电量过大的电器来解决问题，然后打开电路保险开关。在这个过程中，电路保险开关起到保护整个家庭电路系统的作用。

对于微服务系统也一样，当服务的输入负载迅速增加时，如果没有有效的措施对负载进行熔断，则会使服务迅速被压垮，服务被压垮会导致依赖的服务都被压垮，出现雪崩效应，因此，可通过模拟家庭的电路保险开关，在微服务架构中实现熔断模式。

微服务化的熔断模式的状态流转如图1-29所示。

图1-29

## 3. 限流模式

服务的容量和性能是有限的，在第3章中会介绍如何在架构设计过程中评估服务的最大性能和容量，然而，即使我们在设计阶段考虑到了性能压力的问题，并从设计和部署上解决了这些问题，但是业务量是随着时间的推移而增长的，突然上量对于一个飞速发展的平台来说是很常见的事情。

针对服务突然上量，我们必须有限流机制，限流机制一般会控制访问的并发量，例如每秒允许处理的并发用户数及查询量、请求量等。

有如下几种主流的方法实现限流。

### 1）计数器

通过原子变量计算单位时间内的访问次数，如果超出某个阈值，则拒绝后续的请求，等到下一个单位时间再重新计数。

在计数器的实现方法中通常定义了一个循环数组（见图1-30），例如：定义5个元素的环形数组，计数周期为1s，可以记录4s内的访问量，其中有1个元素为当前时间点的标志，通常来说每秒程序都会将前面3s的访问量打印到日志，供统计分析。

图1-30

我们将时间的秒数除以数组元素的个数5，然后取模，映射到环形数组里的数据元素，假如当前时间是1 000 000 002s，那么对应当前时间的环形数组里的第3个元素，下标为2。

此时的数组元素的数据如图1-31所示。

图1-31

在图1-31中，当前时间为1 000 000 002s，对应的计数器在第3个元素，下标为2，当前请求是在这个时间周期内的第1个访问请求，程序首先需要对后一个元素即第4个元素，也就是下标为3的元素清零；在1 000 000 002s内，任何一个请求如果发现下标为3的元素不为0，则都会将原子变量3清零，并记录清零的时间。

这时程序可以对第3个元素即下标为2的元素，进行累加并判断是否达到阈值，如果达到阈值，则拒绝请求，否则请求通过；同时，打印本次及之前3秒的数据访问量，打印结果如下。

当前：1次，前1s：302次，前2s：201次，前3s：518次

然而，如果当前秒一直没有请求量，下一秒的计数器始终不能清零，则下一秒的请求到达后要首先清零再使用，并更新清零时间。

在下一秒的请求到达后，若检查到当前秒对应的原子变量计数器不为0，而且最后的清零时间不是上一秒，则先对当前秒的计数器清零，再进行累加操作，这避免发生上一秒无请求的场景，或者上一秒的请求由于线程调度延迟而没有清零下一秒的场景，后面这种场景发生的概率较小。

另外一种实现计数器的简单方法是单独启动一个线程，每隔一定的时间间隔执行对下一秒的原子变量计数器清零操作，这个时间间隔必须小于计数时间间隔。

### 2）令牌筒

令牌筒是一个流行的实现限流的技术方案，它通过一个线程在单位时间内生产固定数量的令牌，然后把令牌放入队列，每次请求调用需要从桶中拿取一个令牌，拿到令牌后才有资格执行请求调用，否则只能等待拿到令牌再执行，或者直接丢弃。

令牌筒的结构如图1-32所示。

图1-32

### 3）信号量

限流类似于生活中的漏洞，无论倒入多少油，下面有漏管的流量是有限的，实际上我们在应用层使用的信号量也可以实现限流。

使用信号量的示例如下：

public class SemaphoreExample {

private ExecutorService exec = Executors.newCachedThreadPool();

public static void main(String[] args) {

final Semaphore sem = new Semaphore(5);

for (int index = 0; index < 20; index++) {

Runnable run = new Runnable() {

public void run() {

try {

// 获得许可

sem.acquire();

// 同时只有5个请求可以到达这里

Thread.sleep((long) (Math.random()));

// 释放许可

sem.release();

System.out.println("剩余许可：" + sem.availablePermits());

} catch (InterruptedException e) {

e.printStackTrace();

}

}

};

exec.execute(run);

}

exec.shutdown();

}

}

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

## 4. 失效转移模式

若微服务架构中发生了熔断和限流，则该如何处理被拒绝的请求呢？解决这个问题的模式叫作失效转移模式，通常分为下面几种。

采用快速失败的策略，直接返回使用方错误，让使用方知道发生了问题并自行决定后续处理。

是否有备份服务，如果有备份服务，则迅速切换到备份服务。

失败的服务有可能是某台机器有问题，而不是所有机器有问题，例如OOM问题，在这种情况下适合使用failover策略，采用重试的方法来解决，但是这种方法要求服务提供者的服务实现了幂等性。

微服务的粒度

在服务化系统或者微服务架构中，我们如何拆分服务才是最合理的？服务拆分到什么样的粒度最合适？

按照微服务的初衷，服务要按照业务的功能进行拆分，直到每个服务的功能和职责单一，甚至不可再拆分为止，以至于每个服务都能独立部署，扩容和缩容方便，能够有效地提高利用率。拆得越细，服务的耦合度越小，内聚性越好，越适合敏捷发布和上线。

然而，拆得太细会导致系统的服务数量较多，相互依赖的关系较复杂，更重要的是根据康威定律，团队要响应系统的架构，每个微服务都要有相应的独立、自治的团队来维护，这也是一个不切实际的想法。

因此，这里倡导对微服务的拆分适可而止，原则是拆分到可以让使用方自由地编排底层的子服务来获得相应的组合服务即可，同时要考虑团队的建设及人员的数量和分配等。

有的公司把每个接口包装成一个工程，或者把每一次JDBC调用包装成一个工程，然后号称是“微服务”，最后有成百上千的微服务项目，这是不合理的。当然，有的公司把一套接口完成的一个粗粒度的流程耦合在一个项目中，导致上层服务想要使用这套接口中某个单独的服务时，由于这个服务与其他逻辑耦合在一起，所以需要在流程中做定制化才能实现使用方使用部分服务的需求，这也是不合理的，原因是服务粒度太粗。

总之，拆分的粒度太细和太粗都是不合理的，根据业务需要，能够满足上层服务对底层服务自由编排并获得更多的业务功能即可，并需要适合团队的建设和布局。

---------------------

作者：钱曙光

来源：CSDN

原文：https://blog.csdn.net/qiansg123/article/details/80131044

版权声明：本文为博主原创文章，转载请附上博文链接！