# 微服务

## 什么是微服务？

微服务架构是一种架构模式，将单一的应用程序划分成一组小的服务，每个服务运行在其独立的自己的进程内，当用户发出请求，服务之间相互协调，相互配合，共同完成整个功能。

## 为什么要用微服务？

### 单一架构模式：

#### 优势：

* 开发效率高
* 易于部署
* 易于测试

#### 劣势：

* 部署成本高（无论是修改1行代码，还是10行代码，都要全量替换）
* 可伸缩性差，系统的扩容只能只对这个应用进行扩容，不能做到对某个功能点进行扩容
* 复杂度高，随着业务的不断迭代，项目的代码量会急剧的增多，项目模块也会随着而增加，模块与模块之间的关系就会变成的很复杂，整个项目就会变成的非常复杂

### 微服务架构

#### 优势

* 微服务是松耦合的，可以对某一修改的模块单独部署
* 可伸缩性比较好，能做到对某个功能点进行扩容
* 不同服务之间可以使用不同的技术

#### 劣势

* 服务间通信成本问题；
* 系统部署比较复杂
* 故障诊断比较难，不容易定位错误节点；

# SpringCloud Netflix

## Eureka（服务治理）

Eureka的服务治理体系有三个核心角色：服务注册中心、服务提供者，，服务消费者。

#### 服务注册中心：

指的是Eureka提供的服务端，主要负责提供服务注册与发现功能。

#### 服务提供者：

指的是可以提供服务的应用。

#### 服务消费者：

指的是消费者从服务注册中心获取服务列表。

服务提供者启动时，会通过发送rest请求的方式向server注册自己，并且携带自身的元数据信息（ip,端口，微服务名称等），保存在server的注册表中，然后服务提供者每隔30s会向server发送心跳，告诉server自己还存活，如果sever在默认的时间（90s）内没有收到来自客户端的心跳，那么server就会注销该服务实例。

sever中保存客户端数据的注册表是基于内存进行存储的，在内存里维护了一个CocurrentHashMap<String ,Map<String,Leasse<InstanceInfo>>>的数据结构。该结构是以双层map的形式来进行存储的数据的，最外层的key是客户端微服务的名称，value也是一个map结构，这个map的key是服务实例的id(host+ serviceId + port),value就是client实例信息（ip、端口号、等信息）。

服务消费者在启动的时候，向注册中心发送REST请求给服务注册中心，从服务注册中心获取服务列表清单。并将其缓存在本地，客户端会使用该信息查找其他服务，从而进行远程调用，同时也会每隔30s也会向server发送心跳，完成服务续约，server中的注册表信息定期（每30秒钟）更新一次。

实际上当客户端在拉取服务注册表的时候，其实客户端不是直接从 Eureka 中的服务注册表中获取数据的，因为Eureka 做了二级缓存，第一级叫做 ReadOnly（ReadOnlyCacheMap） 缓存，二级叫做 ReadWrite（ReadWriteCacheMap） 缓存。首先客户端会直接从ReadOnly 缓存中读取注册表信息。若没有，就找ReadWrite里缓存的注册表。如果还没有，就从内存中获取实际的注册表数据。

当服务在进行注册的时候，先向服务注册表中写入注册信息，服务注册表更新了，立马会同步一份数据到 ReadWrite 缓存中去，此时会有一个定时任务（每隔30s）会定时去检查 ReadWrite缓存是否跟  ReadOnly缓存不一致, 不一致就把数据同步到 ReadOnly缓存中去,如果注册表发生变更的时候，会在内存中更新变更的注册表数据，同时过期掉ReadWrite缓存。此过程不会影响ReadOnly缓存提供人家查询注册表。在一段时间内（默认是30s）,各服务拉取注册表会直接读ReadOnly缓存, 30s过后，后台线程发现ReadWrite缓存已经清空了，也会清空ReadOnly缓存中的缓存, 等到下次有服务拉取注册表，又会从内存中获取最新的数据了，同时填充各个缓存。

#### 多级缓存机制的优点是什么？

尽可能保证了内存注册表数据不会出现频繁的读写冲突问题。并且进一步保证对Eureka Server的大量请求，都是快速从纯内存走，性能极高。

#### 服务同步

当服务提供者发送请求到一个注册中心时，注册中心会将注册的信息转发给其它集群相连的注册中心，完成注册中心之间的服务同步。通过注册中心之间的服务同步，集群注册中心服务信息保持一致。

#### 服务下线：

当服务实例进行正常关闭操作时，它会触发一个服务下线的REST请求Eureka Server，注册中心收到请求时，服务端将该服务状态置为下线(DOWN)，同时把该下线事件传播下去。

#### 失效剔除：

服务有时候并不一定会正常下线，可能由于异常故障使服务运行不正常，但是，服务注册中心并未收到"服务下线"请求。注册中心为了将这些无法提供服务的实例剔除，Eureka Server在启动的时候会创建一个定时任务，默认每隔一段时间(默认为60秒)将当前清单中超时(默认为90秒)没有续约的服务从注册中心剔除。

#### 自我保护：

默认情况下，如果Eureka Server在一定时间内（默认90秒）没有接收到某个微服务实例的心跳，Eureka Server将会移除该实例。

但是当网络分区故障发生时，微服务与Eureka Server之间无法正常通信，而微服务本身是正常运行的，此时不应该移除这个微服务，所以引入了自我保护机制。

自我保护机制的工作机制是如果在15分钟内超过85%的客户端节点都没有正常的心跳，那么Eureka就认为客户端与注册中心出现了网络故障，Eureka Server自动进入自我保护机制，

此时会出现以下几种情况：

1）Eureka Server不再从注册列表中移除因为长时间没收到心跳而应该过期的服务。

2）Eureka Server仍然能够接受新服务的注册和查询请求，但是不会被同步到其它节点上，保证当前节点依然可用。

3）当网络稳定时，当前Eureka Server新的注册信息会被同步到其它节点中。

#### 注解：

@EnableEurekaServer：

@EnableDiscoveryClient

## ribbon（负载均衡）

#### 概要

是一个客户端的负载均衡工具，主要功能是提供客户端的软件负载均衡算法和服务调用，可以和RestTemplate搭配使用，使用@LoadBalanced注解完成服务间的负载均衡调用功能，并携带重试机制。

#### Ribbon 负载均衡策略：

RoundRobinRule: 轮询策略，Ribbon以轮询的方式选择服务器，这个是默认值。所以示例中所启动的两个服务会被循环访问;

WeightedResponseTimeRule: 带有加权的轮询策略，对各个服务器响应时间进行加权处理，响应速度越快的实例选择权重越大，越容易被选择。然后在采用轮询的方式来获取相应的服务器;

RandomRule: 随机选择，也就是说Ribbon会随机从服务器列表中选择一个进行访问;

RetryRule：重试规则，先按照RoundRobinRule的策略获取服务，如果获取服务失败则在指定时间内会进行重试，获取可用的服务。

BestAvailableRule: 最大可用策略，会先过滤掉由于多次访问故障而处于断路器跳闸状态的服务，然后选择一个并发量最小的服务;

AvailabilityFilteringRule: 先过滤掉故障实例，再选择并发较小的实例。

ZoneAvoidanceRule: 区域权衡策略，先使用主过滤条件（区域负载器，选择最优区域）对所有实例过滤并返回过滤后的实例清单，依次使用过滤条件列表中的过滤条件对主过滤条件的结果进行过滤，判断最小过滤数（默认1）和最小过滤百分比（默认0），最后对满足条件的服务器则使用RoundRobinRule(轮询方式)选择一个服务器实例。

#### 自定义负责均衡策略

还可以自定义负责均衡策略，

官方文档明确给出了警告：这个自定义配置类不能放在@ComponentScan所扫描的当前包下以及子包下，否则我们自定义的这个配置类就会被所有的Ribbon客户端所共享，达不到特殊化定制的目的了。

实现这个IRule接口，就能自定义负载均衡策略,在启动类配置@RibbonClient(name= "CLOUD-PAYMENT-SERVICE",configuration=MySelfRule.class)（MySelfRule是自定义策略配置类的名称）

#### 轮训负载均衡算法原理：

负载均衡算法：rest接口第几次请求数 % 服务器集群总数量 = 实际调用服务器位置下标  ，每次服务重启动后rest接口计数从1开始。

List<ServiceInstance> instances = discoveryClient.getInstances("CLOUD-PAYMENT-SERVICE");

如：   List [0] instances = 127.0.0.1:8002

　　　List [1] instances = 127.0.0.1:8001

8001+ 8002 组合成为集群，它们共计2台机器，集群总数为2， 按照轮询算法原理：

当总请求数为1时： 1 % 2 =1 对应下标位置为1 ，则获得服务地址为127.0.0.1:8001

当总请求数位2时： 2 % 2 =0 对应下标位置为0 ，则获得服务地址为127.0.0.1:8002

当总请求数位3时： 3 % 2 =1 对应下标位置为1 ，则获得服务地址为127.0.0.1:8001

当总请求数位4时： 4 % 2 =0 对应下标位置为0 ，则获得服务地址为127.0.0.1:8002

#### Nginx 和 Ribbon 的对比

##### 集中式负载均衡器

将所有请求都集中起来，然后再进行负载均衡（nginx就是属于集中式的负载均衡器）。

##### 进程内负载均衡器

负载均衡器集成到消费方，请求到达消费方，消费方再从服务注册中心获知有哪些地址可用，然后自己再从这些地址中选择出一个合适的服务器。

#### 注解

@LoadBalanced

## Feign（服务调用）

#### 概要:

Feign是一种声明式、模板化的HTTP客户端 ，再RestTemplate的基础上更进一步的封装了，以创建Java接口并使用注解配置的方式来完成对服务提供方的参数绑定，并且在Feign的内部集成了Ribbon负载均衡器，实现了客户端的负载均衡调用。

#### Feign的底层实现

Feign是基于面向接口动态代理的方式生成实现类，然后将请求调用委托到动态代理实现类中，根据默认的contract协议解析接口上的方法和注解，转换成内部的MethodHandler处理方式，根据传入的bean对象和注解信息，从中提取出数据，生成request对象，Feign最终会将请求转化成Http消息发送出去，传入的请求对象，会被解析成消息体。在请求转换的过程，可以自定义拦截器，对请求和响应进行装饰处理，并且在发送请求和响应请求的时候，定义了统一的日志门面来输出日志消息，Feign 真正发送HTTP请求是委托给 Feign.Client 来做的。Feign内部设置一个重试器，当http请求发生io异常时，会有一个最大重试次数。重新发出请求。

#### Feign的源码实现的过程：

* 首先通过@EnableFeignClients注解开启FeignClient
* 根据Feign的规则实现接口，并加@FeignClient注解
* 程序启动后，会进行包扫描，扫描所有的@FeignClient的注解的类，并将这些信息注入到ioc容器中。
* 当接口的方法被调用，通过jdk的代理，将Bean对象和注解信息生成具体的RequesTemplate，构造相对应的Request对象
* Request对象交给Client去处理，其中Client可以是HttpUrlConnection、HttpClient也可以是Okhttp
* 最后Client被封装到LoadBalanceClient类，这个类结合Ribbon做到了负载均衡。
* Feign 内置了一个重试器，当HTTP请求出现IO异常时，Feign会有一个最大尝试次数发送请求

#### Feig和Ribbion的区别：

##### 相同点

Ribbon 和 Feign 都是用于调用其他服务的，都实现了负载均衡和重试机制。

##### 不同点

* 启动类使用的注解不同，Ribbon 用的是@RibbonClient，Feign 用的是@EnableFeignClients。
* 服务的指定位置不同，Ribbon 是在启动类上使用@RibbonClient 注解上声明，Feign则是在定义抽象方法的接口中使用@FeignClient 声明。
* 调用方式不同，Ribbon 需要自己构建 http 请求，模拟 http 请求然后使用 RestTemplate 发送给其他服务，Feign 则是在 Ribbon 的基础上进行了一次改进，采用接口的方式，将需要调用的其他服务的方法定义成抽象方法即可，不需要自己构建 http 请求。不过要注意的是抽象方法的注解、方法签名要和提供服务的方法完全一致。

#### Feign和OpenFeign的区别：

* Feign本身不支持Spring MVC的注解，它有一套自己的注解。
* OpenFeign是Spring Cloud 在Feign的基础上支持了Spring MVC的注解，如@RequesMapping等等

#### 注解

@EnableFeignClients

@FeignClient

## Hystrix（断路器）

#### 概要

Hystrix是一个用于处理分布式系统的延迟和容错的开源库，分布式系统里，为了防止依赖服务之间调用失败，而导致的服务雪崩问题，对依赖服务调用之间出现的调用延迟和调用失败进行控制和容错保护。

#### 服务雪崩：

当微服务A,调用微服务B的接口时，由于微服务B发生异常，产生请求阻塞，用户无法响应，线程得不到释放，就会造成越来越多的请求阻塞，因为服务器的支持的并发数和线程有限，请求一直阻塞，服务器的资源耗尽，会导致其他的服务也不可用，造成雪崩现象的产生。

#### 解决方案：

服务限流、服务熔断、服务降级、超时监控、请求缓存（生产不建议使用）、请求合并（生产不建议使用）

#### 服务降级

服务降级是指，当请求超时、资源不足等情况发生时进行服务降级处理，不调用真实服务逻辑，而是使用快速失败（fallback）方式直接返回一个托底数据，保证服务链条的完整，避免服务雪崩。

##### 哪些情况会出发降级？

* 程序运行异常
* 超时
* 服务熔断触发服务降级
* 线程池/信号量打满也会导致服务降级

#### 服务熔断

当一定时间内，异常请求比例（请求超时、网络故障、服务异常等）达到阀值时，启动熔断器，熔断器一旦启动，则会停止调用具体服务逻辑，通过fallback快速返回托底数据，保证服务链的完整。

##### 熔断机制状态：

对于熔断机制的实现，Hystrix设计了三种状态，熔断关闭、熔断开启、半熔断状态。

###### 熔断关闭状态（Closed）

服务没有故障时，熔断器所处的状态，对调用方的调用不做任何限制。

###### 熔断开启状态（Open）

在时间滑动窗口中内（Hystrix默认是10秒），接口调用次数大于或等于20次，接口调用出错比率达到一个阈值（Hystrix默认为50%），会进入熔断开启状态。进入熔断状态后，后续对该服务接口的调用不再经过远程调用，直接执行本地的fallback方法。

###### 半熔断状态（Half-Open）

在进入熔断开启状态一段时间之后（Hystrix默认是5秒），熔断器会进入半熔断状态。会尝试让一条请求经过断路器，看能不能正常调用。如果调用成功了，那么就自动恢复，断路器转为 Close 状态。如果重试失败，熔断持续开启，并返回fallback数据。等待下一次休眠时间过后重新尝试。

#### 服务限流

服务端的请求承载量是有限的，当客户端请求时，Hystrix 通过判断线程池或者信号量是否已满，限制请求的并发数，当请求数量过大对服务端产生压力超出容量的请求，直接拒绝走降级，从而达到限流的作用。

##### 线程隔离

Hystrix会为每一个依赖服务分配一个小的线程池，用户的请求是通过线程池的空闲线程来调用服务，如果线程池已满或者请求超时，用户的请求将不再直接访问服务，调用将会被拒绝，通过fallback返回托底数据。

###### 优点：

当线程池出现问题时，线程池是完全隔离状态的，是独立的，不会影响到其他服务的正常执行，独立的线程池也提供了限制并发处理能力, 并且支持异步处理，支持超时处理，当请求超时，会自动中断，是默认的隔离机制。

###### 缺点：

线程池隔离机制，请求处理线程和服务调用线程，不是同一个线程，需要上下文切换，会导致服务硬件计算开销加大。

##### 信号量隔离

信号量隔离就是hystrix的限流功能之一。实际上它是通过信号量来实现的限流功能，而信号量说白了就是个计数器，设置一个并发处理的最大极值，计数器计算达到设定的阈值，多余的请求再来请求线程资源，就被拒绝了，直接就做异常处理，通过fallback返回托底数据，保证服务完整性。

###### 优点：

信号量隔离机制，也提供了限制并发处理能力，请求处理线程和服务调用线程，是同一个线程，无线程切换，开销比较小。

###### 缺点：

不支持超时处理，必须等待远程调用结束，才能判断是否超时。

##### 适用场景：

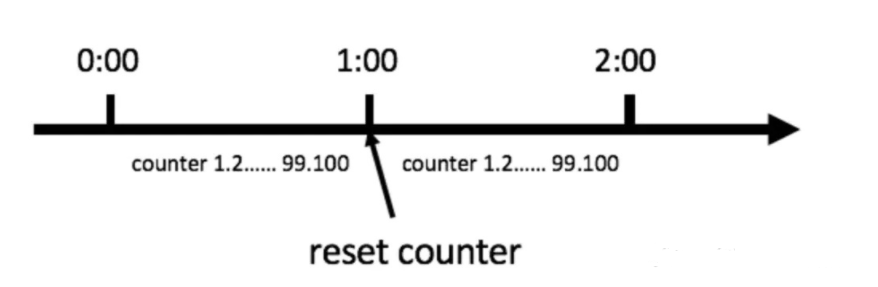
* 线程池技术，适合绝大多数场景，比如说我们对依赖服务的网络请求的调用和访问、需要对调用的 timeout 进行控制（捕捉 timeout 超时异常）。
* 信号量技术，适合说你的访问不是对外部依赖的访问，而是对内部的一些比较复杂的业务逻辑的访问，并且系统内部的代码，其实不涉及任何的网络请求，那么只要做信号量的普通限流就可以了，因为不需要去捕获 timeout 类似的问题。

##### 限流的算法

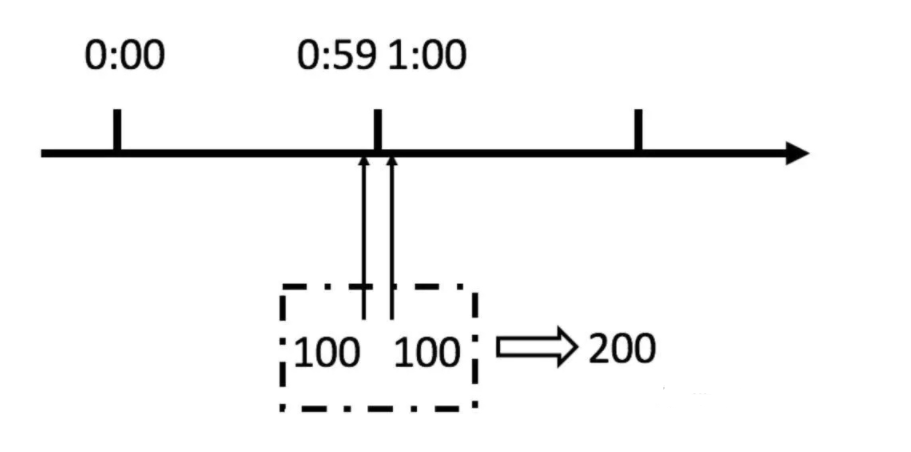
###### 固定窗口计数器算法

**规定我们单位时间处理的请求数量。**

比如我们规定我们的一个接口一分钟只能访问100次的话。使用固定窗口计数器算法的话可以这样实现：给定一个变量counter来记录处理的请求数量，当1分钟之内处理一个请求之后，counter+1，1分钟之内的如果counter=100的话，后续的请求就会被全部拒绝。等到 1分钟结束后，将counter回归成0，重新开始计数（ps：只要过了一个周期就讲counter回归成0）。



这个算法虽然简单，但是有一个十分致命的问题，那就是临界问题。

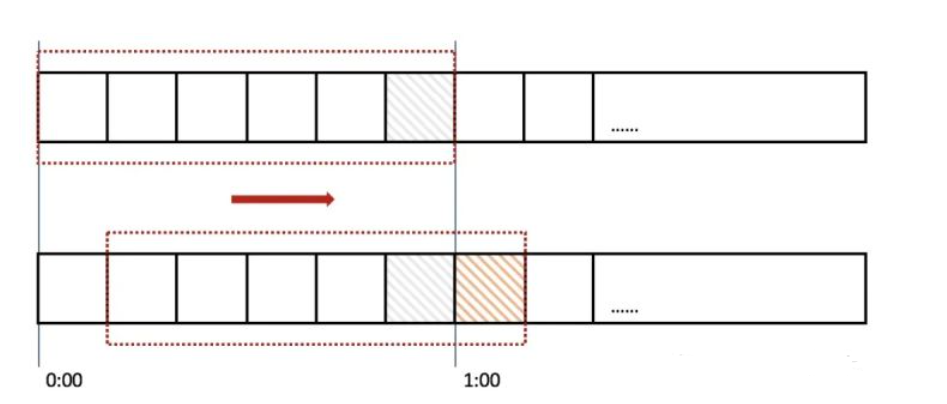


假设有一个恶意用户，他在0:59时，瞬间发送了100个请求，并且1:00又瞬间发送了100个请求，那么其实这个用户在 1秒里面，瞬间发送了200个请求。用户通过在时间窗口的重置节点处突发请求，可以瞬间超过我们的速率限制。用户有可能通过算法的这个漏洞，瞬间压垮我们的系统。

###### 滑动窗口计数器算法

算的上是固定窗口计数器算法的升级版。滑动窗口计数器算法相比于固定窗口计数器算法的优化在于：**它把时间以一定比例分片。**

例如我们的借口限流每分钟处理60个请求，我们可以把 1 分钟分为60个窗口。每隔1秒移动一次，每个窗口一秒只能处理 不大于 60(请求数)/60（窗口数） 的请求， 如果当前窗口的请求计数总和超过了限制的数量的话就不再处理其他请求。



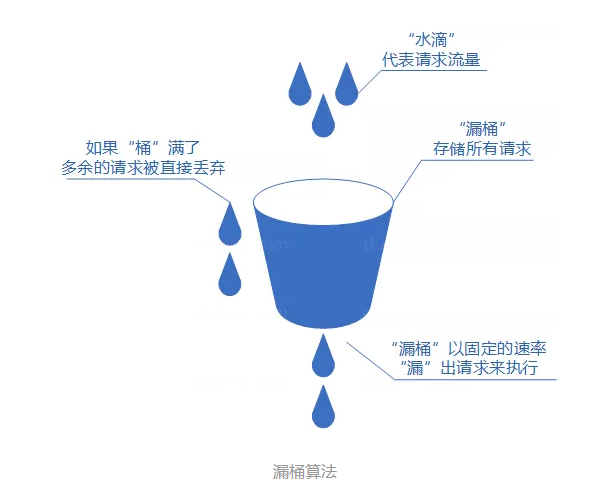
在上图中，整个红色的矩形框表示一个时间窗口，在我们的例子中，一个时间窗口就是一分钟。然后我们将时间窗口进行划分，比如图中，我们就将滑动窗口划成了6格，所以每格代表的是10秒钟。每过10秒钟，我们的时间窗口就会往右滑动一格。每一个格子都有自己独立的计数器counter，比如当一个请求 在0:35秒的时候到达，那么0:30~0:39对应的counter就会加1。

那么滑动窗口怎么解决之前的临界问题的呢？我们可以看上图，0:59到达的100个请求会落在灰色的格子中，而1:00到达的请求会落在橘黄色的格子中。当时间到达1:00时，我们的窗口会往右移动一格，那么此时时间窗口内的总请求数量一共是200个，超过了限定的100个，所以此时能够检测出来触发了限流。

很显然：当滑动窗口的格子划分的越多，滑动窗口的滚动就越平滑，限流的统计就会越精确。

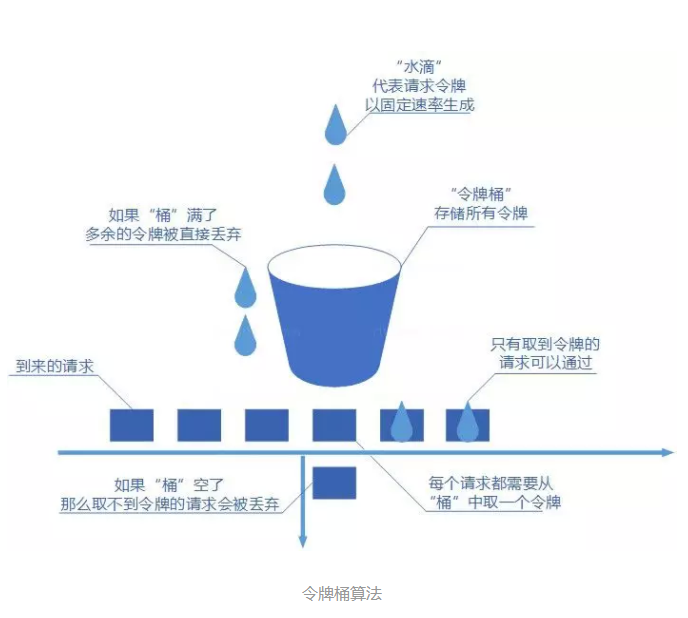
###### 漏桶算法

我们可以把发请求的动作比作成注水到桶中，我们处理请求的过程可以比喻为漏桶漏水。我们往桶中以任意速率流入水，以一定速率流出水。当水超过桶流量则丢弃，因为桶容量是不变的，保证了整体的速率。如果想要实现这个算法的话也很简单，准备一个队列用来保存请求，然后我们定期从队列中拿请求来执行就好了。



###### 令牌桶算法

令牌桶算法和漏桶算法一样,大小固定的令牌桶可自行以恒定的速率源源不断地产生令牌。如果令牌不被消耗，或者被消耗的速度小于产生的速度，令牌就会不断地增多，直到把桶填满，后面再产生的令牌就会从桶中溢出，请求在被处理之前需要拿到一个令牌，请求处理完毕之后将这个令牌丢弃（删除），最后桶中可以保存的最大令牌数永远不会超过桶的大小，所以可以通过控制往桶里添加令牌的速率，可以控制流量峰值的大小。



#### 请求缓存

请求缓存保证在一次请求中多次调用同一个服务提供者接口，在cacheKey不变的情况下，后续调用结果都是第一次的缓存结果，而不是多次请求服务提供者，从而降低服务提供者处理重复请求的压力。

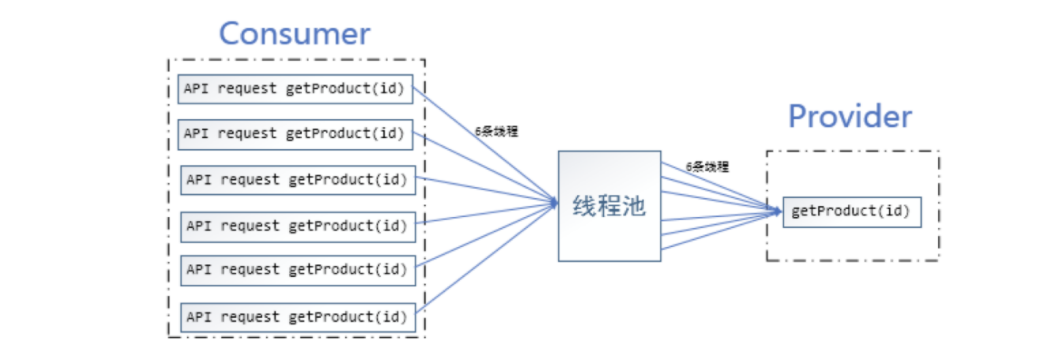
请求缓存不是只写入一次结果就不再变化的，而是每次请求到达Controller的时候，我们都需要为HystrixRequestContext进行初始化，之前的缓存也就是不存在了，我们是在同一个请求中保证结果相同，同一次请求中的第一次访问后对结果进行缓存，缓存的生命周期只有一次请求！

但是使用请求缓存会导致很多的隐患，如：缓存管理不当导致的数据不同步、问题排查困难等，索引解决服务雪崩效应不推荐使用请求缓存。

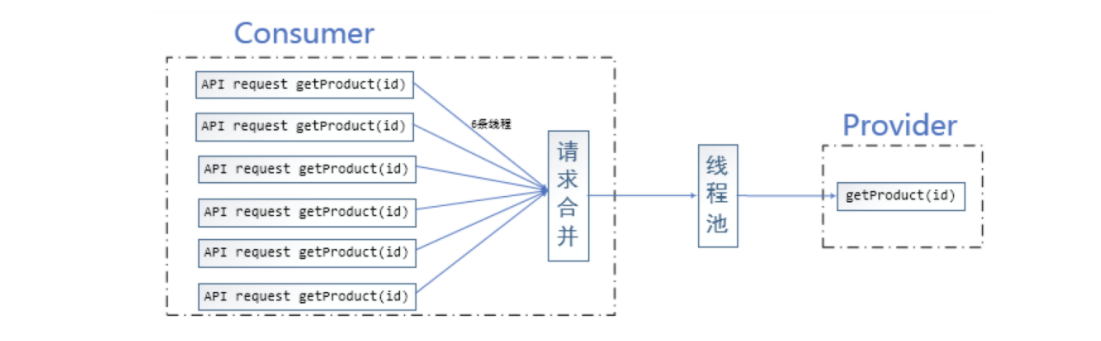
#### 请求合并

在一定时间内，收集一定量的同类型请求，合并请求需求后，一次性访问服务提供者，得到批量结果。这种方式可以减少服务消费者和服务提供者之间的通讯次数，提升应用执行效率。

未使用请求合并



使用请求合并后



##### 优点

当同类型请求的并发数很高，通过设置请求合并器的延迟时间，在一定时间内，收集同类型请求，合并请求需求后，一次性访问服务提供者，可以得到批量结果。减少服务消费者和服务提供者之间的通讯次数，提升应用执行效率。

##### 缺点

通过请求合并器设置延迟时间，这样在并发不高的接口上使用请求合并，会降低响应速度。

#### 注解

@EnableCircuitBreaker

@HystrixCommand

## Zuul（网关）

#### 概要

Zuul是一个微服务网关，会在Eureka注册中心中进行服务的注册和发现。Zuul网关的核心功能是过滤和路由，不管是来自于客户端（PC或移动端）的请求，还是服务内部调用，一切对服务的请求都会经过Zuul这个网关，然后再由网关来实现鉴权、动态路由等操作。

在Zuul的底层使用的是Hystrix+ribbon来实现请求路由，具有负载均衡和服务容错能力，但是Zuul的fallback容错处理逻辑，只针对timeout异常处理，当请求被Zuul路由后，只要服务有返回（包括异常），都不会触发Zuul的fallback容错逻辑。

Zuul中使用Hystrix的配置来设置路由转发时的执行超时时间，其默认的超时时长为1000毫秒。使用Ribbon的配置来设置路由转发时请求连接及处理的超时时间，ribbon底层默认超时时长为5000毫秒。如果Hystrix超时，直接返回超时异常。如果ribbon超时，同时Hystrix未超时，ribbon会自动进行服务集群轮询重试，直到Hystrix超时为止。如果Hystrix超时时长小于ribbon超时时长，ribbon不会进行服务集群轮询重试。并且Zuul网关组件也提供了限流保护。当请求并发达到阀值，自动触发限流保护，返回错误结果。

#### 路由

路由功能负责将外部请求转发到具体的服务实例上去，是实现统一访问入口的基础。

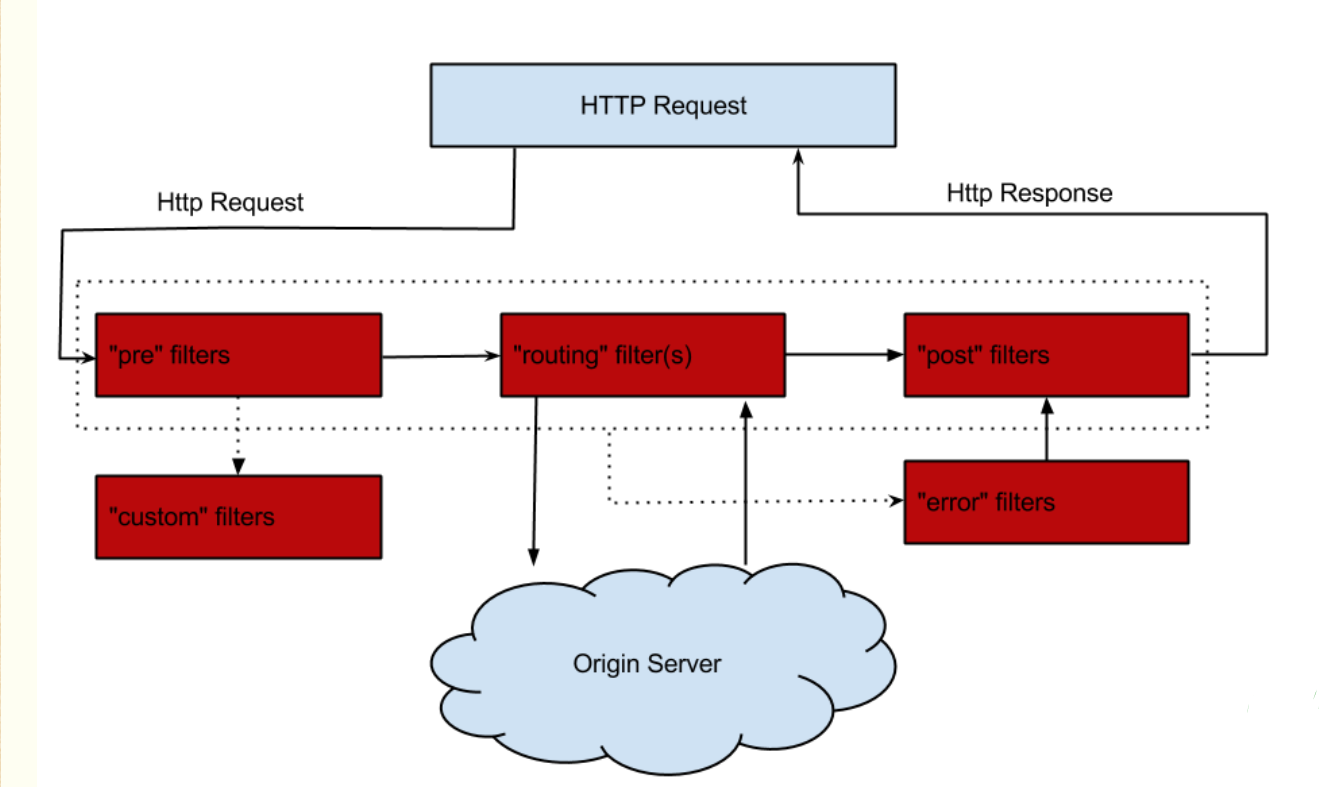
#### 过滤

过滤功能负责对请求过程进行额外的处理，是请求校验过滤及服务聚合的基础。

#### 过滤器类型：

* pre：在请求被路由到目标服务前执行，比如权限校验、打印日志等功能；
* routing：在请求被路由到目标服务时执行，这是使用HttpClient或Ribbon构建和发送原始HTTP请求的地方；
* post：在请求被路由到目标服务后执行，比如给目标服务的响应添加头信息，收集统计数据等功能；
* error：请求在其他阶段发生错误时执行。

#### 请求到过滤的执行流程



首先HTTP请求到达Zuul之后，最先来到pre过滤器（在请求到达路由前被调用），在这里会根据映射Url 找到目标地址，然后将请求与找到的地址交给route类型的过滤器进行求转发，请求服务实例获取响应，通过post类型过滤器对处理结果进行加工与转换等操作返回，在这整个请求过程中只要有异常才会触发，将异常结果交post类型过滤器加工返回。

#### 注解

@EnableZuulProxy

## Config（配置中心）以及Bus（消息总线）

#### 概要

SpringCloud Config一个基于http协议的远程配置实现方式, 通过统一的配置管理服务器进行配置管理，有两个重要的角色，服务端和客户端。

服务端也称为分布式配置中心，它是一个独立的微服务应用，用来连接配置配置仓库（服务器）并为客户端提供获取配置信息。

客户端是通过指定的配置中心来管理应用资源以及业务相关的配置内容，并启动时从配置中心获取和加载配置信息。配置服务器默认采用git来存储配置信息。

#### 配置手动刷新

使用controller层使用@RefreshScope，并需要运维人员手动向客户端发送Post请求，才能生效。

#### 配置自动刷新

Spring Cloud Bus 配合Spring Cloud Config 使用可以实现配置的动态刷新。

##### 自动刷新的原理

ConfigClient实例都监听MQ中同一个topic(默认是Spring Cloud Bus)。当一个服务刷新数据的时候，它会把这个信息放入到Topic中，这样其它监听同一Topic的服务就能得到通知，然后去更新自身的配置。

#### 消息总线

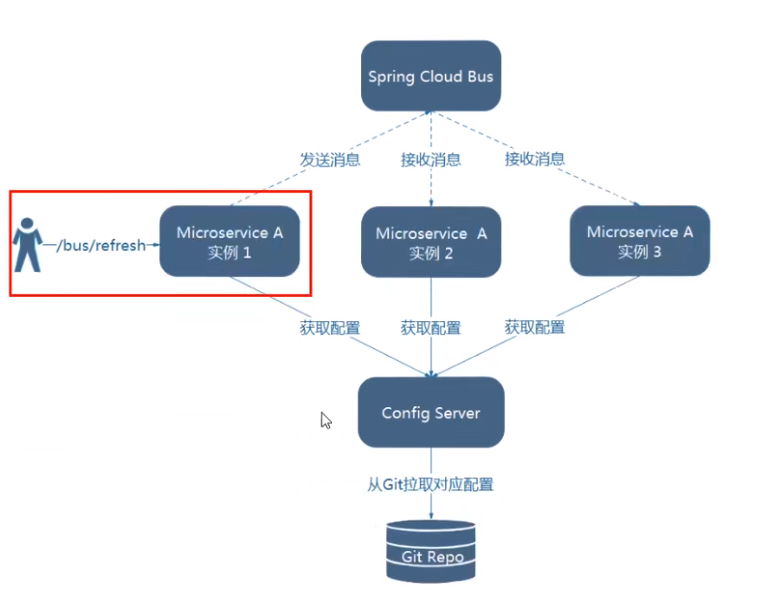
在微服务架构的系统中，会构建一个共用的消息主题，并让系统中所有微服务实例都连接上来，由于该主题中产生的消息会被所有实例监听和消费，所以称它为消息总线。

Spring Cloud Bus是用来将分布式系统的节点与消息系统链接起来的框架，它整合了Java的事件处理机制和消息中间件的功能。

#### Bus动态刷新全局广播的设计思想

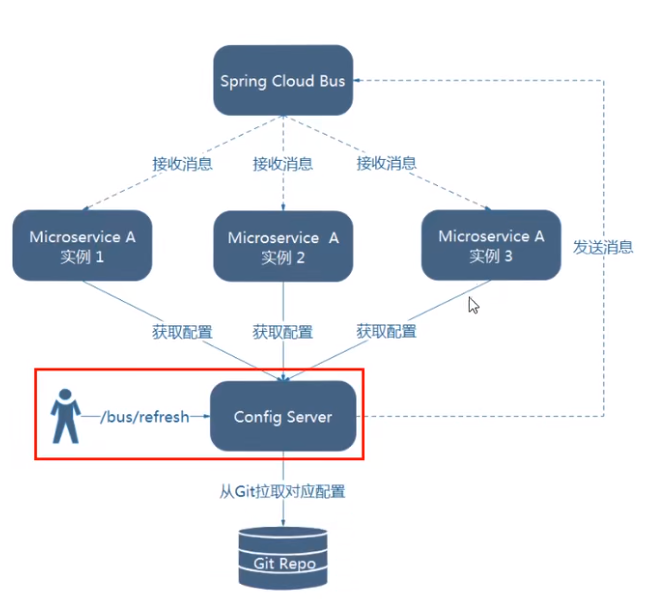
##### 刷新客户端

利用消息总线触发一个客户端/bus/refresh,而刷新所有客户端的配置。



##### 刷新配置中心

利用消息总线触发一个服务端ConfigServer的/bus/refresh端点，而刷新所有客户端的配置。



需要发送POST请求

curl -X POST "http://localhost:3344/actuator/bus-refresh"

—次发送，处处生效

#### Bus动态刷新定点通知

指定具体某一个实例生效而不是全部

公式：http://localhost:3344/actuator/bus-refresh/{destination}

/bus/refresh请求不再发送到具体的服务实例上，而是发给config server通过destination参数类指定需要更新配置的服务或实例

例如：

我们这里以刷新运行在3355端口上的config-client（配置文件中设定的应用名称）为例，只通知3355，不通知3366

curl -X POST "http://localhost:3344/actuator/bus-refresh/config-client:3355

#### 注解

@EnableConfigServer

## Sleuth（全链路追踪）

#### 概要

Spring Cloud Sleuth提供了一套完整的服务跟踪的解决方案，并且支持集成了zipkin。

#### 术语

（1）Trace：它是由一组有相同Trace ID的Span串联形成一个树状结构。为了实现请求跟踪，当请求到分布式系统的入口端点时，只需要服务跟踪框架为该请求创建一个唯一的跟踪标识（即前文提到的Trace ID），同时在分布式系统内部流转的时候，框架始终保持该传递的唯一标识，直到返回请求为止，我们通过它将所有请求过程中的日志关联起来；

（2）Span：它代表了一个基础的工作单元，例如服务调用。为了统计各处理单元的时间延迟，当前请求到达各个服务组件时，也通过一个唯一标识（即前文提到的Span ID）来标记它的开始、具体过程以及结束。通过span的开始和结束的时间戳，就能统计该span的时间延迟，除此之外，我们还可以获取如事件名称、请求信息等元数据。

（3）Annotation：它用于记录一段时间内的事件。

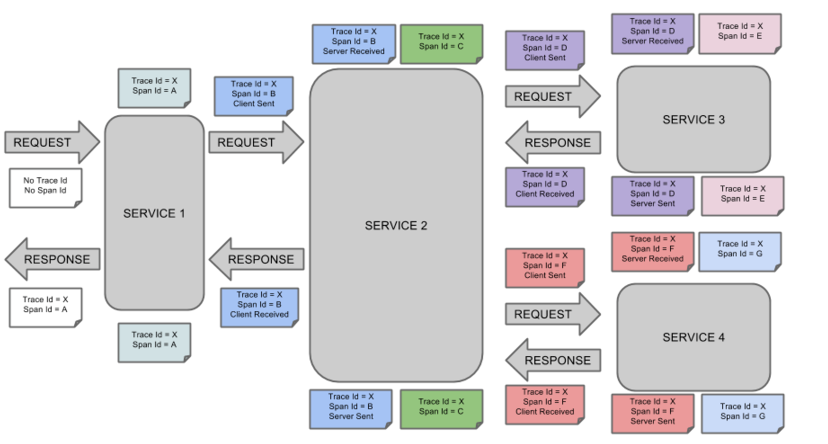
内部使用的最重要的注释是：

* cs - Client Sent - 客户端发送一个请求，这个注解描述了这个Span的开始。
* sr - Server Received - 服务端获得请求并准备开始处理它，其中（sr – cs） 时间戳便可得到网络传输的时间。
* ss - Server Sent （服务端发送响应）– 该注解表明请求处理的完成(当请求返回客户端)， （ss – sr）时间戳就可以得到服务器请求的时间。
* cr - Client Received （客户端接收响应）- 表明此时Span的结束，（cr – cs）时间戳便可以得到整个请求所消耗的时间。

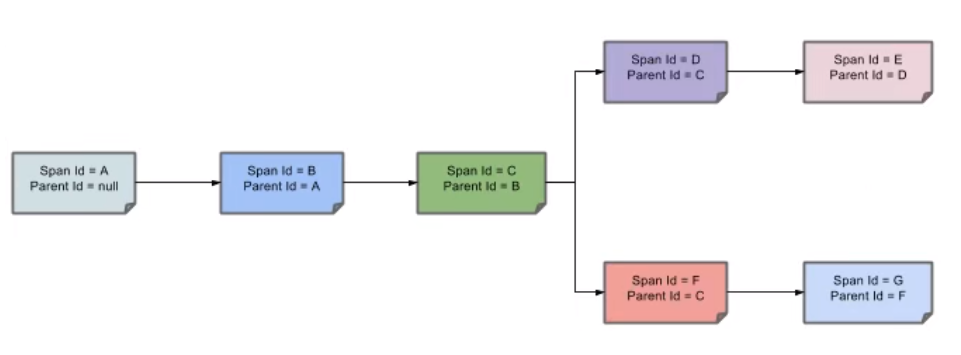
#### 追踪原理

为了实现请求跟踪，当请求发送到分布式系统的入口端点时，只需要服务跟踪框架为该请求创建一个唯一的跟踪标识，同时在分布式系统内部流转的时候，框架始终保持传递该唯一标识，直到返回给请求方为止，这个唯一标识就是前文中提到的Trace ID。通过Trace ID的记录，我们就能将所有请求过程日志关联起来。

为了统计各处理单元的时间延迟，当请求达到各个服务组件时，或是处理逻辑到达某个状态时，也通过一个唯一标识来标记它的开始、具体过程以及结束，该标识就是我们前文中提到的Span ID，对于每个Span来说，它必须有开始和结束两个节点，通过记录开始Span和结束Span的时间戳，就能统计出该Span的时间延迟，除了时间戳记录之外，它还可以包含一些其他元数据，比如：事件名称、请求信息等。



表示一条请求链路，一条链路通过TraceId作为唯一标识，Span标识发起的请求信息也具有唯一标识，叫做SpanId，各span通过parentId关联起来（后一个Span的parentId就是前一个Span的SpanId）。



# 2、Spring Cloud Alibaba

## 2.1、Nacos（服务治理和配置管理）

#### 概要

Nacos在微服务架构中的具有服务治理和配置中心的功能，其他服务的服务信息（ip，端口等信息）可以注册到Nacos服务端，同时Nacos又为客户端提供了服务发现的功能以及配置中心。

#### 服务治理

##### 服务注册

###### 客户端

当客户端启动的时候，客户端的服务信息注册到Nacos服务端，这个过程是基于SpringBoot的自动装配来完成的，监听Spring容器启动过程中的事件，向Nacos服务端，注册客户端实例，并且定时发送心跳（心跳周期是每5s发一次），默认心跳时间超过15s，该服务实例就处于不健康状态，默认超过30s，就会删除该客户端的实例，客户端的服务注册信息，默认保存到nacos服务端的注册表中，Nacos的注册表保存数据采用的是分级模型进行存储的，最外层是namespace命名空间，用来隔离环境，接着是Group分组，用来进行服务分组，接着就是Service服务，一个服务包含多个实例，可能处于不同的机房，因此一个Service下面有多个Cluster，Cluster下面才是具体的实例，(对应在java中的数据结构是一个双层Map结构（Map<String, Map<String, Service>>），map的最外层的key是namespace的Id,value也是一个map,这个map的key是group拼接（@@）serviceName，value是Service对象，而在service对象中，又包含了一个集群Map属性，map的key是集群的名称，value是集群对象Cluster，在Cluster内部又维护了一个set的实例集合属性)。

###### 服务端

当客户端服务注册的请求到达nacos服务端时，会异步完成本地注册表信息更新以及异步进行Nacos集群的数据操作（会将服务的唯一标识和要更新的实例列表写到缓存中dataStore，接着将服务的唯一标识和操作类型，放入到阻塞队列中，让线程池异步的去执行任务，更新实例是采用CopyOnWrite的思想，先将旧的实例列表拷贝一份，然后更新拷贝的实例列表，再用更新后的实例列表来覆盖旧的实例列表，防止发生并发冲突问题，集群数据同步也是如此，将服务的唯一标识和操作类型封装成一个任务，交给阻塞队列，由线程池异步去执行，如果异步执行失败，就会再次进入阻塞队列，重新执行操作）。

##### 服务发现

###### 定时拉取

当Nacos客户端启动的时候，服务之间的负载均衡调用是基于ribbon来完成的，所以，服务拉取的动作也是ribbon的动态服务列表负载均衡器 （DynamicServerListLoadBalancer）来完成的，拉取服务信息的时候，会先走客户端的本地缓存中拉取，如果本地缓存中没有，会向更新map（updateMap）中，添加一条更新的数据，然后再去向客户端发送http请求拉取数据，获取完数据，会存到本地缓存中，然后删除更新map中的这条数据。在拉取服务的时候，如果缓存中有对应的服务信息，且更新map中要修改的服务信息，则会延时等待5s，然后再去创建一个定时更新服务信息的任务，由线程池异步执行，在任务执行过程中，会拿当前服务的更新时间和上一次的参考时间做对比，如果当前服务中的时间小于或者等于上次的参考时间，会重新去调用服务端的服务列表，然后服务的最新信息存放到本地缓存中，如果当前服务中的时间大于上次的参考时间，则会重新记录时间。

###### 订阅推送

在Nacos客户端启动，去获取服务端的服务列表的信息时，会向服务端发送一个推送人的一个udp端口，同时会开启一个接受Udp数据数据报的阻塞线程，接收数据，解析数据报信息，并同时发送ack确认数据报信息。

在服务端接受到来自客户端发送的请求时，会判断推送服务中是否有当前服务的客户端存在，如果没有，则新增客户端，保存到推送服务中，如果存在，则刷新时间，当服务列表中的服务状态发生变更的时候，会发布服务改变事件的通知，当服务改变事件发布的时候，会异步向当前服务所有的订阅客户端发送含有最新服务列表的数据报信息，并保存缓存中去，如果服务端未接收到来自客户端的ack信息，会进行重发，重发的信息是从缓存中进行读取发送的。

#### 健康检测

另外Nacos的健康检测有两种模式：

##### 临时实例：

采用客户端心跳检测模式（客户端以http请求的方式向服务端发送心跳），心跳周期5秒，心跳间隔超过15秒则标记为不健康，心跳间隔超过30秒则从服务列表删除。

##### 永久实例：

服务端是自己与客户端建立socket通道，以主动请求的方式进行健康检测，周期为2000毫秒 + 5000毫秒内的随机数，检测异常只会标记为不健康，不会删除实例。

#### Nacos如何支撑阿里内部数十万服务注册压力？

Nacos内部接收到注册的请求时，不会立即写数据，而是将服务注册的任务放入一个阻塞队列就立即响应给客户端。然后利用线程池读取阻塞队列中的任务，异步来完成实例更新，从而提高并发写能力。

#### Nacos如何避免并发读写冲突问题？

##### 读写冲突

Nacos在更新实例列表时，会采用CopyOnWrite技术，首先将旧的实例列表拷贝一份，然后更新拷贝的实例列表，再用更新后的实例列表来覆盖旧的实例列表。这样在更新的过程中，就不会对读实例列表的请求产生影响，也不会出现脏读问题了。

##### 并发写

Nacos在更新实例列表时，会对当前的服务加锁，类似分段锁，保证当前的服务每次都只能有一个线程去更新实例，而更新的其他的服务则不受任何影响。

#### Nacos与Eureka的区别有哪些？

##### 相同点

从接口方式来说，Nacos与Eureka都对外暴露了Rest风格的API接口，用来实现服务注册、发现等功能，并且都支持服务提供者心跳方式做健康检测。

##### 不同点

* 实例类型：Nacos的实例有永久和临时实例之分；而Eureka只支持临时实例。
* 健康检测：Nacos对临时实例采用心跳模式检测，对永久实例采用主动请求来检测；Eureka只支持心跳模式。
* 服务发现：Nacos支持定时拉取和订阅推送两种模式；Eureka只支持定时拉取模式。
* Nacos集群默认采用AP方式，当集群中存在非临时实例时，采用CP模式；Eureka采用AP方式

#### 配置中心

https://blog.csdn.net/u010398838/article/details/108774579

https://blog.csdn.net/fenglibing/article/details/119895079

https://blog.csdn.net/qq\_35067322/article/details/114958085

##### 客户端启动初始化

当客户端启动的过程中，会加载NacosPropertySourceLocator的locate（）方法,在这个方法中会依次加载一些配置文件（加载顺序为，加载共享配置 -> 加载扩展配置 -> 加载当前应用配置），接着会调用loadNacosPropertySource方法，在方法里判断是否是否需要动态刷新，如果不需要，直接从缓存中获取返回，如果需要动态刷新，则会发送http的get请求，从远程服务器进行获取（在远程获取的过程中,先会优先使用本地配置，如果没有则从远程服务端获取，然后再写入到缓存中）。

##### 动态感知

当客户端启动的过程中，会自动加载NacosConfigService的构造方法，初始化一个客户端工作类（ClientWorker）,这个客户端工作类做了两件事，第一件事是初始化一个用于进行长轮询的线程池，另外一件事就是初始化核心线程个数为1的线程池，用来执行每隔10ms检查配置的任务，在检查配置的过程中，会先获取任务分片，根据任务分片和每个长轮询线程处理的任务(每个Long Polling线程任务处理的任务数的默认值为3000)进行向上取整，获取所需要的长轮询的线程数，如过当前的长轮询线程数量小于所需要的线程数量，会通过线程池来执行创建长轮询任务（创建多个线程，主要是为避免处理过量的数据），在创建长轮询任务的过程中，会取出与该 taskId 相关的 CacheData，然后对 CacheData 进行检查，包括本地配置检查和缓存数据的 md5 检查（本地检查主要是做一个故障容错，当服务端挂掉后，Nacos 客户端可以从本地的文件系统中获取相关的配置信息），然后向服务端发起长轮询（默认超时时间是45s）的请求检查当前任务下的分片配置在服务端是否发生了变化，返回发生了变化的key，遍历发生了变化的key，并根据key去服务端请求最新配置（读取超时时间3s），并更新到内存缓存中，如果请求失败，会延时2s，再次请求。

##### 服务端

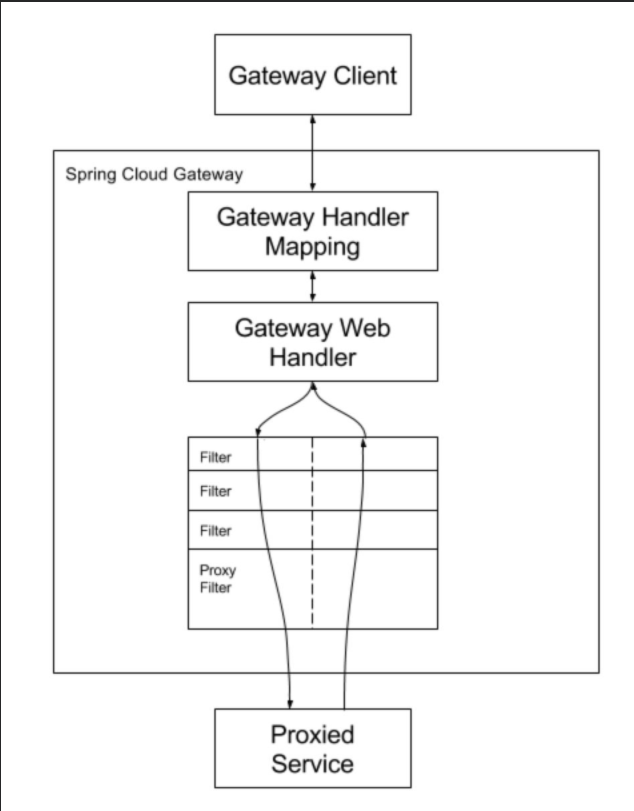
当服务端接收到请求后，会根据请求头里是否有长轮询超时时间判断是否是长轮询请求，如果不是长轮询，根据传过来的数据进行比较，返回变化的配置的key，如果是长轮询，在长轮询初始化的时候，会给通知中心，注册订阅者，并添加本地数据改变的监听事件，然后将这个请求添加到长轮询的队列中，当config配置信息发生修改的时候，就会触发该事件，执行一个事件数据改变的任务，在这任务中，会根据触发事件的配置找到对应的长连接，然后将数据返回。

## 2.2、Gateway（网关）

#### 概要

Gateway是一个服务网关，Zuul网关的核心功能是过滤、路由、断言，不管是来自于客户端（PC或移动端）的请求，还是服务内部调用，一切对服务的请求都会经过Gateway网关，通过网关实现日志记录、鉴权、动态路由等操作。

#### 工作机制



* Gateway 接收客户端请求。
* 客户端请求与路由信息进行匹配，匹配成功的才能够被发往相应的下游服务。
* 请求经过 Filter 过滤器链，执行 pre 处理逻辑，如修改请求头信息等。
* 请求被转发至下游服务并返回响应。
* 响应经过 Filter 过滤器链，执行 post 处理逻辑。
* 向客户端响应应答。

路由、断言、过滤器

#### 微服务网关Zuul和Gateway的区别？

##### 相同点：

二者都是微服务网关，处理的都是http请求，都具有路由和过滤的功能

##### 不同点

* Zuul采用的是Tomcat容器，使用的是传统的Servlet IO处理模型，是基于servlet之上的一个阻塞式处理模型，不支持长连接。
* Gateway 底层使用的通信框架是Netty，Webflux模式替换了旧的Servlet线程模型，非阻塞式处理模型，支持长连接。

## 2.3、Sentinel（熔 断）

#### 概要

Sentinel 是面向分布式服务架构的流量控制组件，主要以流量为切入点，从限流、流量整形、熔断降级、系统负载保护、热点防护等多个维度来帮助开发者保障微服务的稳定性。

#### 资源

Sentinel默认只会监控controller里面的端点

#### 解决雪崩问题的常见方式

##### 1、超时处理：

设定超时时间，请求超过一定时间没有响应就返回错误信息，不会无休止等待。

##### 2、舱壁模式：

限定每个业务服务能使用的线程数，避免一个业务服务宕机，耗尽整个服务器的资源。

##### 3、熔断降级：

由断路器统计业务执行的异常比例，如果超出阈值则会熔断该业务，拦截访问该业务的一切请求。

##### 4、流量控制：

限制业务访问的QPS，避免服务因流量的突增而故障。

#### 流控规则

##### 阈值类型

###### QPS：每秒的查询数（Query Per Second）

###### 线程数：设置的线程个数

##### 流控模式

###### 直接：统计当前资源的请求，触发阈值时对当前资源直接限流，也是默认的模式。



###### 关联：统计与当前资源相关的另一个资源，当另一个资源触发阈值时，对当前资源限流。

使用场景：比如用户支付时需要修改订单状态，同时用户要查询订单。查询和修改操作会争抢数据库锁，产生竞争。业务需求是有限支付和更新订单的业务，因此当修改订单业务触发阈值时，需要对查询订单业务限流。当/write资源访问量触发阈值时，就会对/read资源限流，避免影响/write资源。

适合使用关联模式的场景需要满足的条件：1、两个有竞争关系的资源。2、一个优先级较高，一个优先级较低。



###### 链路：统计从指定链路访问到本资源的请求，触发阈值时，对指定链路限流。

业务场景：有查询订单和创建订单业务，两者都需要查询商品。针对从查询订单进入到查询商品的请求统计，并设置限流。



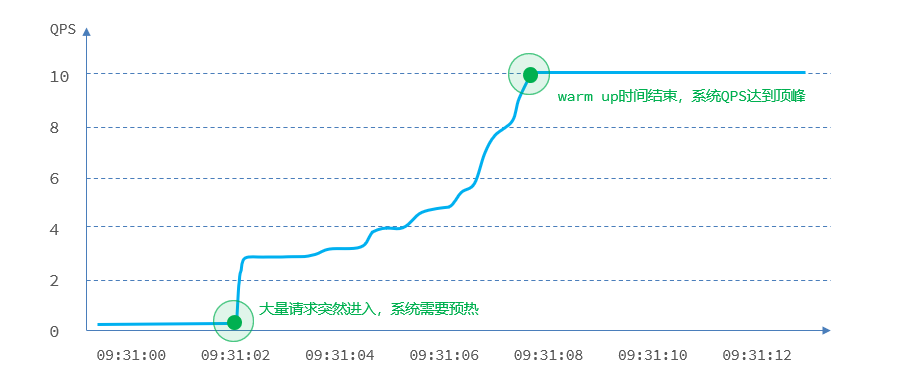
##### 流控效果

###### 快速失败：达到阈值后，新的请求会被立即拒绝并抛出FlowException异常。是默认的处理方式。



###### Warm Up ：预热模式，对超出阈值的请求同样是拒绝并抛出异常。但这种模式阈值会动态变化，从一个较小值逐渐增加到最大阈值。

warm up也叫预热模式，是应对服务冷启动的一种方案。请求阈值初始值是 threshold（最大阈值） / coldFactor（冷启动因子），持续指定时长后，逐渐提高到threshold值。而coldFactor的默认值是3. 例如，我设置QPS的threshold为10，预热时间为5秒，那么初始阈值就是 10 / 3 ，也就是3，然后在5秒后逐渐增长到10.

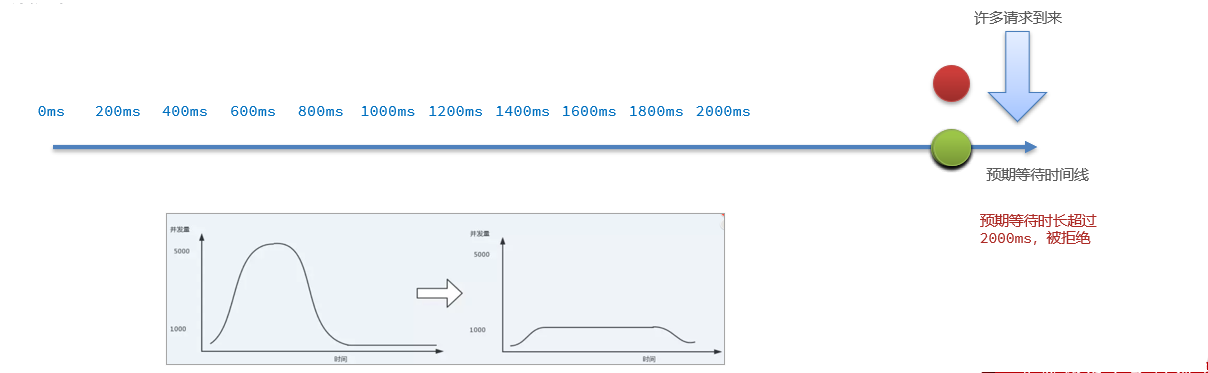




###### 排队等待：让所有的请求按照先后次序排队执行，两个请求的间隔不能小于指定时长。

当请求超过QPS阈值时，快速失败和warm up 会拒绝新的请求并抛出异常。而排队等待则是让所有请求进入一个队列中，然后按照阈值允许的时间间隔依次执行。后来的请求必须等待前面执行完成，如果请求预期的等待时间超出最大时长，则会被拒绝。

例如：QPS = 5，意味着每200ms处理一个队列中的请求；timeout = 2000，意味着预期等待超过2000ms的请求会被拒绝并抛出异常。



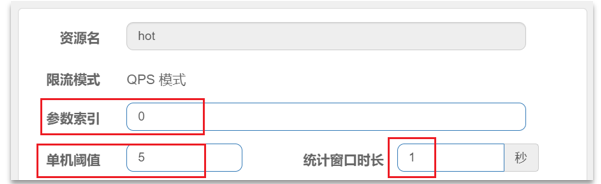


#### 热点规则

热点参数限流是分别统计参数值相同的请求，判断是否超过QPS阈值。



配置示例



代表的含义是：对hot这个资源的0号参数（第一个参数）做统计，每1秒相同参数值的请求数不能超过5

在热点参数限流的高级选项中，可以对部分参数设置例外配置：



结合上一个配置，这里的含义是对0号的long类型参数限流，每1秒相同参数的QPS不能超过5，有两个例外：

如果参数值是100，则每1秒允许的QPS为10

如果参数值是101，则每1秒允许的QPS为15

热点参数限流对默认的SpringMVC资源无效，Controller属于Springmvc资源，需要加@SentinelResource。

#### 降级规则

#### Sentinel和Hystrix的区别？



##### 熔断降级设计理念

Hystrix 默认使用线程池隔离的方式，来对依赖(在我们的概念中对应资源)进行了隔离。这样做的好处是资源和资源之间做到了最彻底的隔离。缺点是除了增加了线程切换的成本，还需要预先给各个资源做线程池大小的分配。

Sentinel 对这个问题采取了两种手段:

###### 1、通过并发线程数进行限制

和线程池隔离的方法不同，Sentinel 通过限制资源并发线程的数量，来减少不稳定资源对其它资源的影响。这样不但没有线程切换的损耗，也不需要预先分配线程池的大小。当某个资源出现不稳定的情况下，例如响应时间变长，对资源的直接影响就是会造成线程数的逐步堆积。当线程数在特定资源上堆积到一定的数量之后，对该资源的新请求就会被拒绝。堆积的线程完成任务后才开始继续接收请求。

###### 2、通过响应时间对资源进行降级

除了对并发线程数进行控制以外，Sentinel 还可以通过响应时间来快速降级不稳定的资源。当依赖的资源出现响应时间过长后，所有对该资源的访问都会被直接拒绝，直到过了指定的时间窗口之后才重新恢复。

# 3、分布式事务

## 分布式事务

分布式事务，就是指不是在单个服务或单个数据库架构下，产生的事务，是跨数据源的分布式事务，跨服务的分布式事务。

### CAP定理

* Consistency（一致性）：

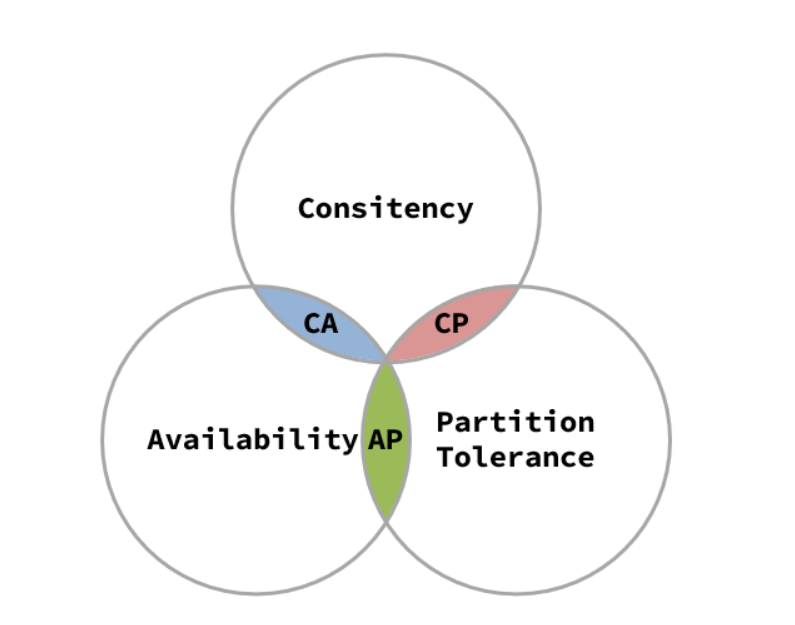
用户访问分布式系统中的任意节点，得到的数据必须一致。

* Availability（可用性）：

用户访问集群中的任意健康节点，必须能得到响应，而不是超时或拒绝。

* Partition tolerance （分区容错性）。

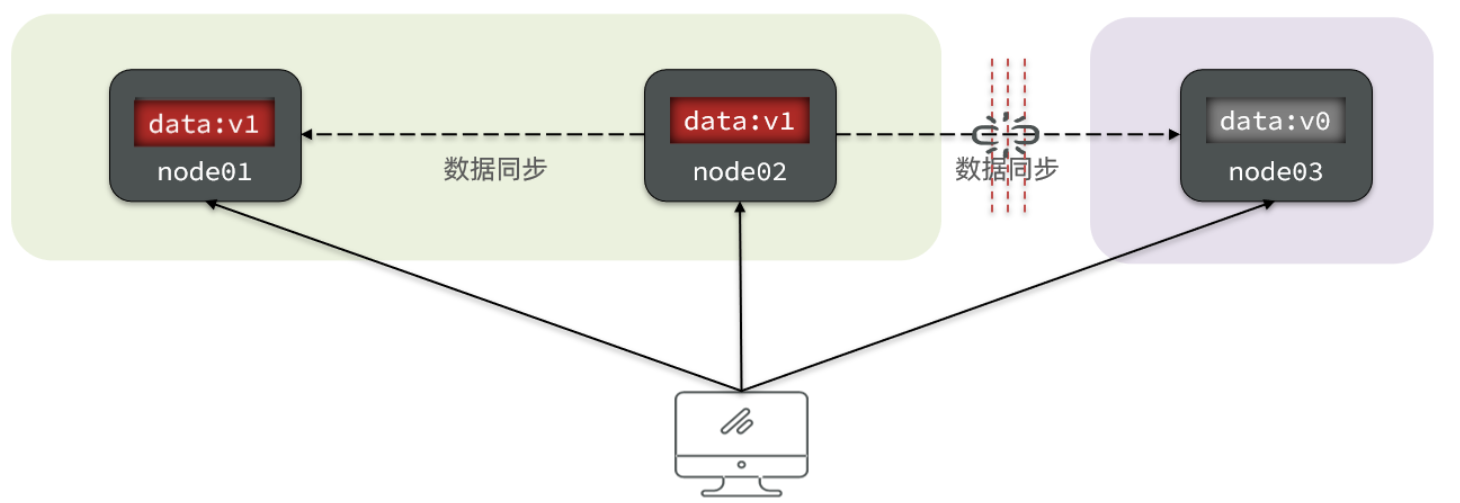
因为网络故障或其它原因导致分布式系统中的部分节点与其它节点失去连接，形成独立分区。



### 矛盾

在分布式系统中，系统间的网络不能100%保证健康，一定会有故障的时候，而服务有必须对外保证服务。因此Partition Tolerance不可避免。

当节点接收到新的数据变更时，就会出现问题了：



如果此时要保证一致性，就必须等待网络恢复，完成数据同步后，整个集群才对外提供服务，服务处于阻塞状态，不可用，即服务不可用。

如果此时要保证可用性，就不能等待网络恢复，那node01、node02与node03之间就会出现数据不一致，即数据不一致。

### BASE理论

BASE理论是对CAP的一种解决思路，包含三个思想：

* Basically Available （基本可用）：分布式系统在出现故障时，允许损失部分可用性，即保证核心可用。
* Soft State（软状态）：在一定时间内，允许出现中间状态，比如临时的不一致状态。
* Eventually Consistent（最终一致性）：虽然无法保证强一致性，但是在软状态结束后，最终达到数据一致。

### 解决分布式事务的思路

分布式事务最大的问题是各个子事务的一致性问题，因此可以借鉴CAP定理和BASE理论，有两种解决思路：

* AP模式：各子事务分别执行和提交，允许出现结果不一致，然后采用弥补措施恢复数据即可，实现最终一致。
* CP模式：各个子事务执行后互相等待，同时提交，同时回滚，达成强一致。但事务等待过程中，处于弱可用状态。

## Seata

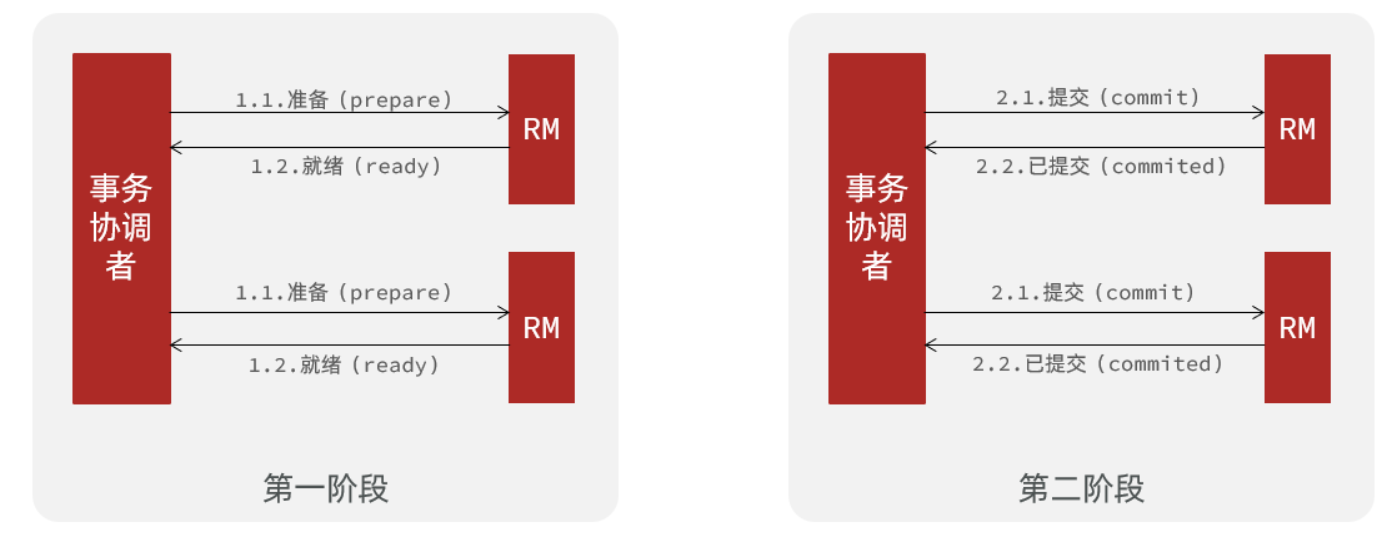
### XA模式

XA 规范 是 X/Open 组织定义的分布式事务处理（DTP，Distributed Transaction Processing）标准，XA 规范 描述了全局的TM与局部的RM之间的接口，几乎所有主流的数据库都对 XA 规范 提供了支持。

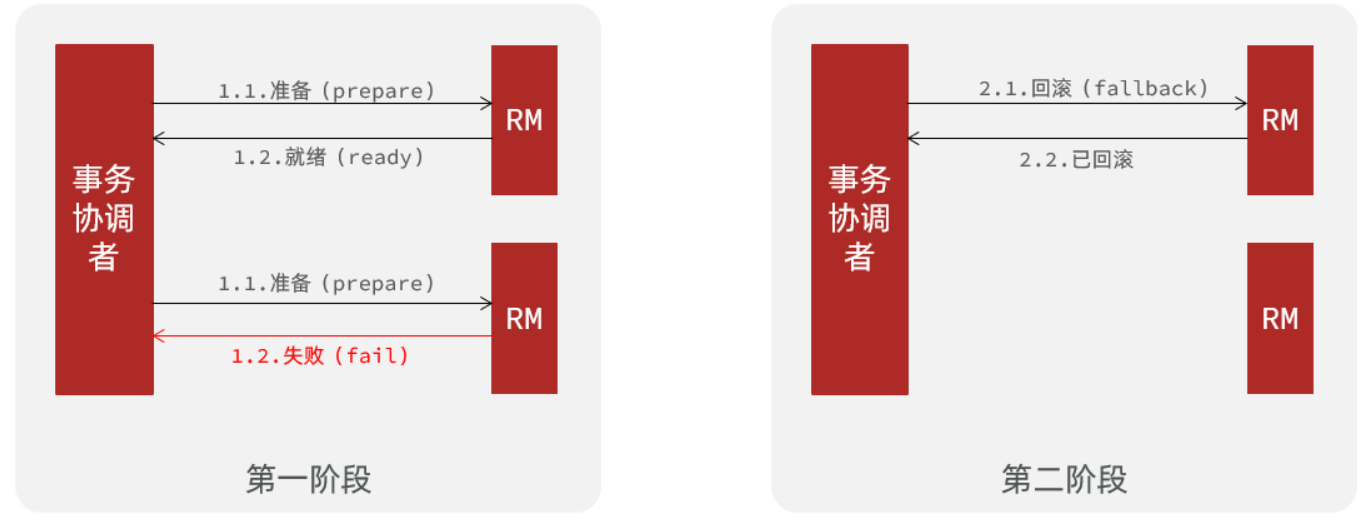
#### 两阶段提交

XA是规范，目前主流数据库都实现了这种规范，实现的原理都是基于两阶段提交。

正常情况：



异常情况：



一阶段：

* 事务协调者通知每个事物参与者执行本地事务
* 本地事务执行完成后报告事务执行状态给事务协调者，此时事务不提交，继续持有数据库锁

二阶段：

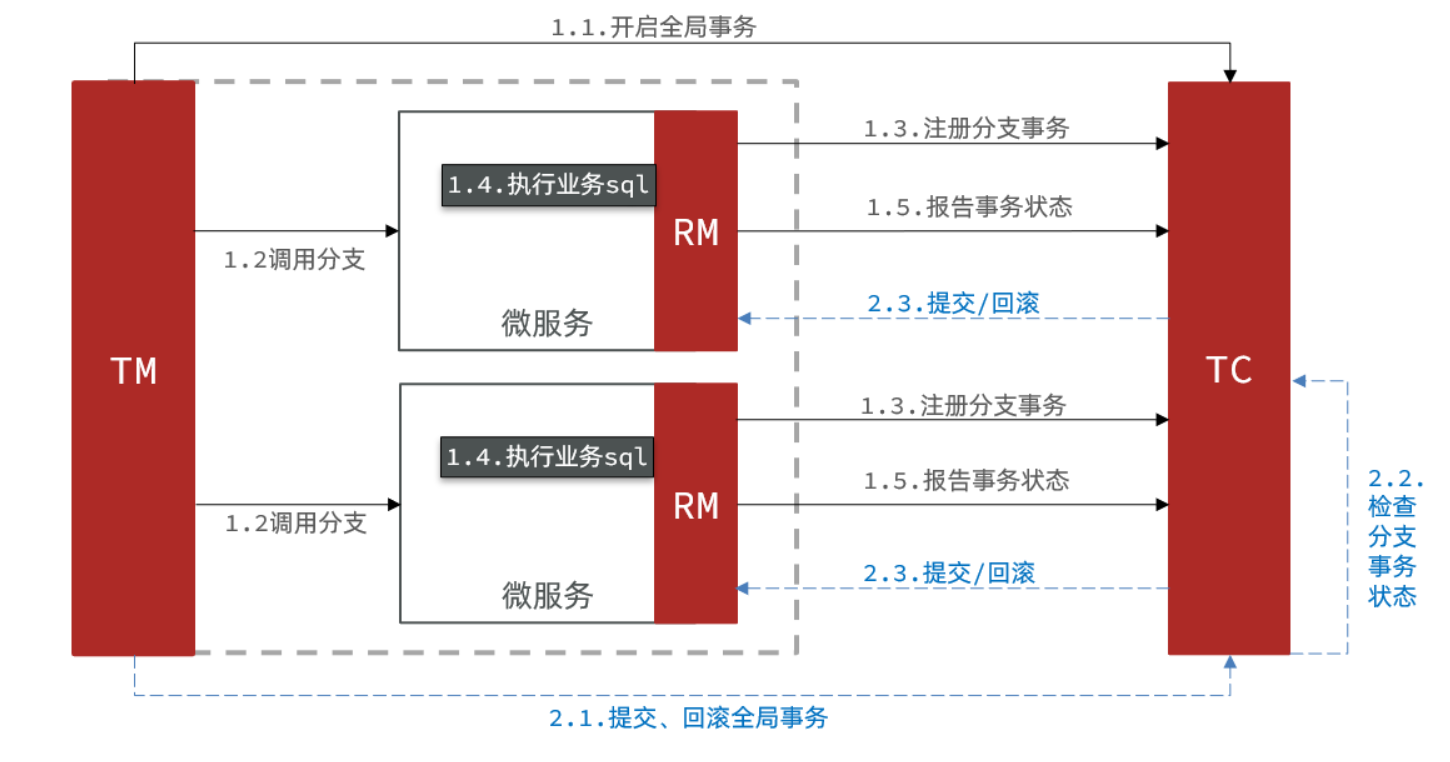
* 事务协调者基于一阶段的报告来判断下一步操作

如果一阶段都成功，则通知所有事务参与者，提交事务

如果一阶段任意一个参与者失败，则通知所有事务参与者回滚事务

#### Seata的XA模型

Seata对原始的XA模式做了简单的封装和改造，以适应自己的事务模型，基本架构如图：



一阶段，首先TM会向TC提交开启事务，TM会调用各个微服务，执行各个微服务的分支事务，RM会先向TC注册分支事务，然后再去执行分支事务，执行完毕之后不去提交，去向TC报告事务的状态。

二阶段，TM发现事务结束了，通知TC，然后TC会去检查分支事务的状态，如果各个分支事务都成功，则进行事务的提交，如果有一个失败，则进行事务的回滚。

RM一阶段的工作：

① 注册分支事务到TC

② 执行分支业务sql但不提交

③ 报告执行状态到TC

TC二阶段的工作：

* TC检测各分支事务执行状态

a.如果都成功，通知所有RM提交事务

b.如果有失败，通知所有RM回滚事务

RM二阶段的工作：

* 接收TC指令，提交或回滚事务

#### 优缺点

XA模式的优点是什么？

* 事务的强一致性，满足ACID原则。
* 常用数据库都支持，实现简单，并且没有代码侵入

XA模式的缺点是什么？

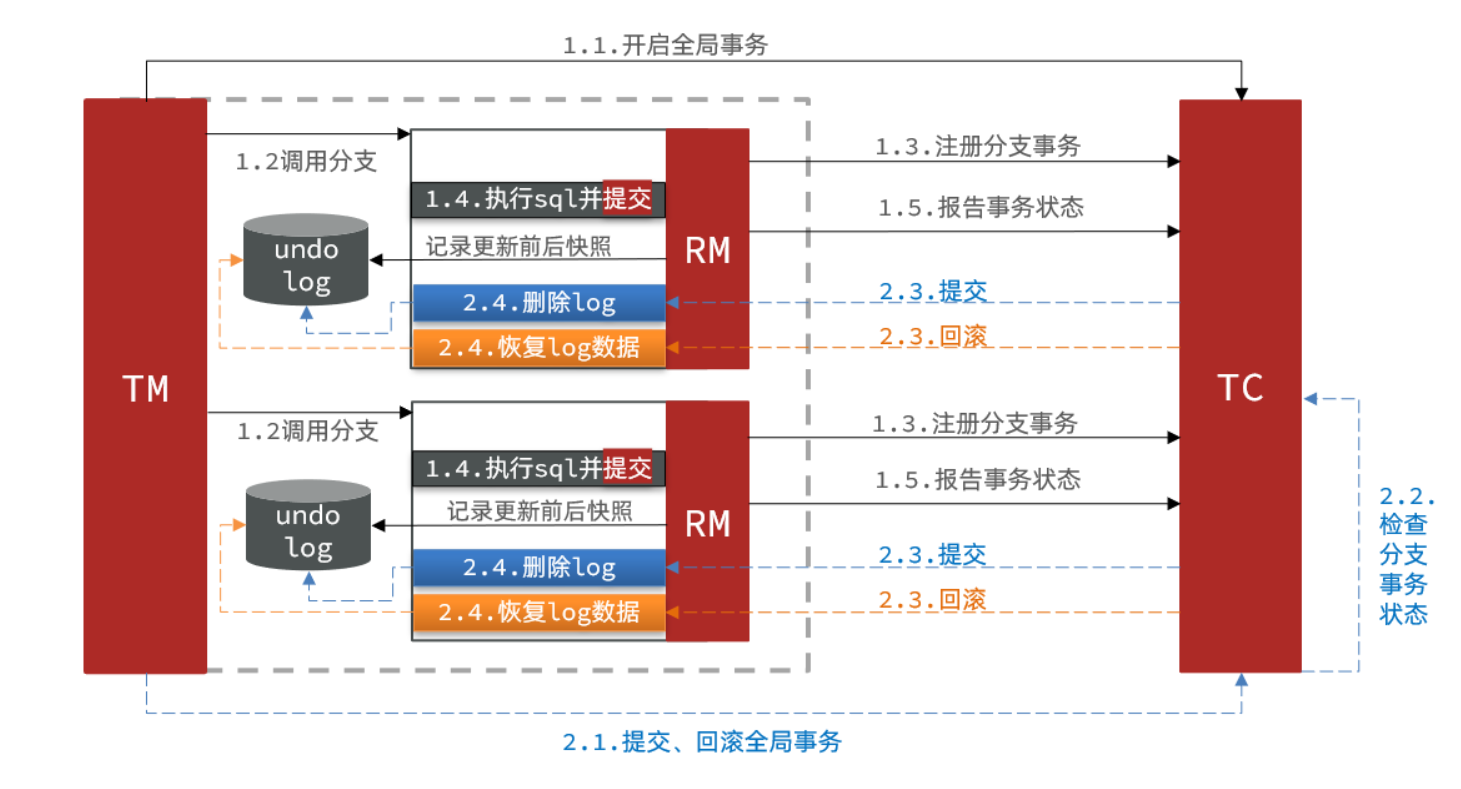
* 因为一阶段需要锁定数据库资源，等待二阶段结束才释放，性能较差
* 依赖关系型数据库实现事务

### AT模式

AT模式同样是分阶段提交的事务模型，不过缺弥补了XA模型中资源锁定周期过长的缺陷。

#### Seata的AT模型

基本流程图：



一阶段，首先TM会向TC提交开启事务，TM会调用各个微服务，执行各个微服务的分支事务，RM会先向TC注册分支事务，然后再去执行分支事务，执行完毕之后直接提交事务，但是在事务提交之前会进行一个快照备份，然后再去向TC报告事务的状态。TM发现事务结束了，通知TC，然后TC会去检查分支事务的状态，如果各个分支事务都成功，在二阶段，则异步进行快照备份的删除，如果有一个失败，则进行备份文件的恢复操作，然后异步的去删除快照备份文件。

阶段一RM的工作：

- 注册分支事务

- 记录undo-log（数据快照）

- 执行业务sql并提交

- 报告事务状态

阶段二提交时RM的工作：

- 删除undo-log即可

阶段二回滚时RM的工作：

- 根据undo-log恢复数据到更新前

#### AT与XA的区别

##### 简述AT模式与XA模式最大的区别是什么？

* XA模式一阶段不提交事务，锁定资源；AT模式一阶段直接提交，不锁定资源。
* XA模式依赖数据库机制实现回滚；AT模式利用数据快照实现数据回滚。
* XA模式强一致；AT模式最终一致

#### 脏写问题

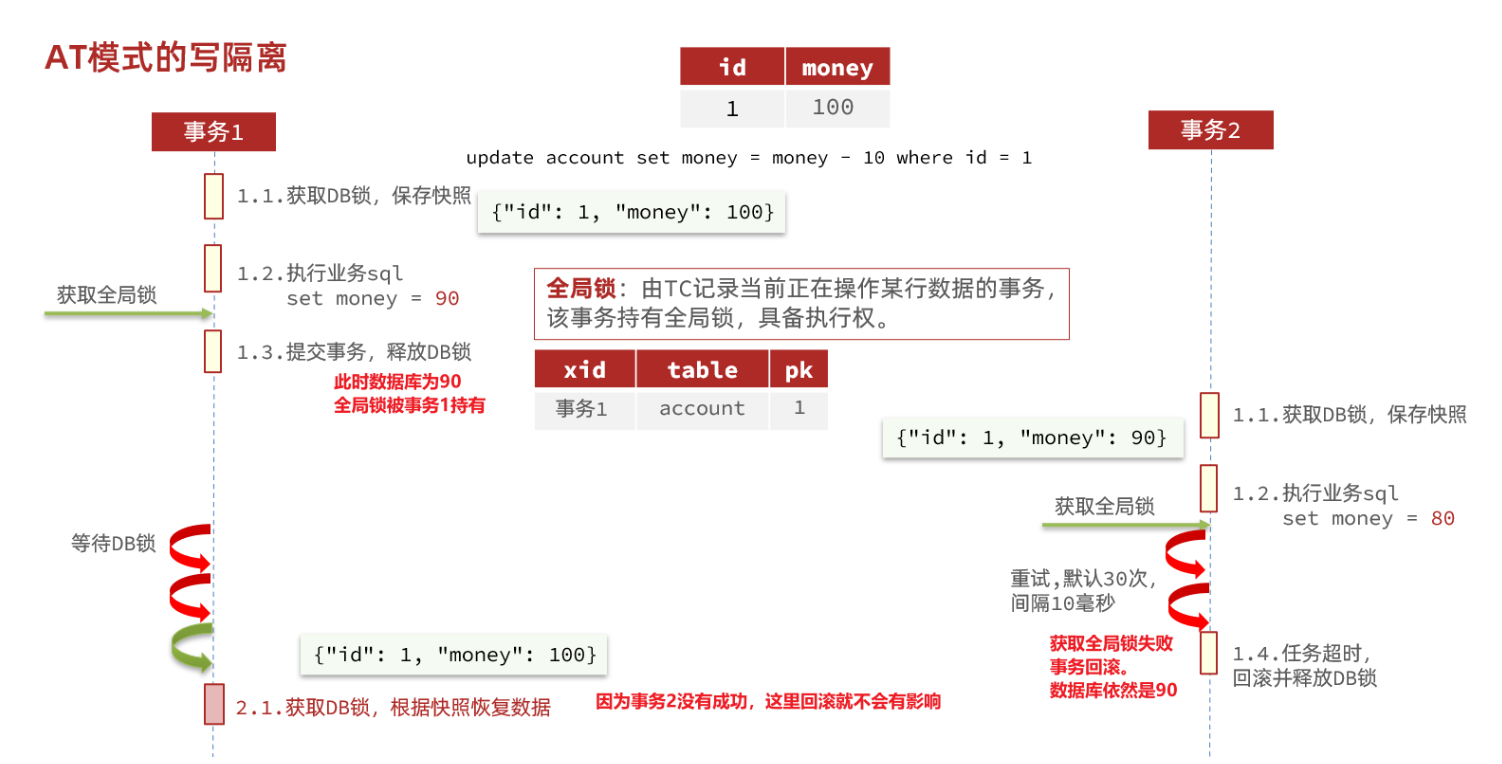
在多线程并发访问AT模式的分布式事务时，有可能出现脏写问题。



当事务1执行事务的时候，会先去获取到DB锁，保存快照，然后再去执行业务sql，提交完事务，会释放DB锁，接着应该通知TC，等待快照的恢复或者删除，但是因为整个过程的操作不是原子性的，所以在事务1释放DB锁之后，可能会被另一个事务2所抢占到，执行事务2的事务，在事务2执行完操作之后，更改完数据，提交事务之后，释放DB锁，再次被事务1所抢占到，经过TC判别，要进行快照的恢复，就导致事务2修改的数据被事务1回滚的事务进行覆盖，出现脏写的问题。

#### 解决脏写问题

解决思路就是引入了全局锁的概念。在释放DB锁之前，先拿到全局锁。避免同一时刻有另外一个事务来操作当前数据。



当事务1执行事务的时候，会先去获取到DB锁，保存快照，然后再去执行业务sql，sql执行完毕之后，会先去拿到全局锁，全局锁是由TC去记录的，然后再提交完事务，释放DB锁，接着应该通知TC，等待快照的恢复或者删除，但是因为整个过程的操作不是原子性的，所以在事务1释放DB锁之后，可能会被另一个事务2所抢占到，执行事务2的事务，在事务2进行获取DB锁，保存快照，执行sql，在执行sql之后，也会去获取全局锁，但是因为在TC中记录的全局锁，被事务1所获取，就会进行重试，默认是30次，每次间隔10毫秒，如果时间超时之后，便会进行事务的回滚，释放DB锁，之后被事务1获取到锁，根据TC的状态进行快照的恢复或者删除，利用全局锁，就保证了多线程环境下同时只有一条事务执行成功，不会出现脏写的问题。

如果事务1是被Seata管理的事务，事务2没有被Seata管理，那么事务1在保存快照的时候会保存两份快照一个是修改之前的，一个是修改之后的，当发生快照数据恢复的时候，会先比对数据库的数据与快照备份修改之后的数据是否一致，如果一致的话则照常恢复数据，不是的话则记录异常发送警告。

#### 优缺点

AT模式的优点：

* 一阶段完成直接提交事务，释放数据库资源，性能比较好
* 利用全局锁实现读写隔离
* 没有代码侵入，框架自动完成回滚和提交

AT模式的缺点：

* 两阶段之间属于软状态，属于最终一致
* 框架的快照功能会影响性能，但比XA模式要好很多

### TCC模式

TCC模式与AT模式非常相似，每阶段都是独立事务，不同的是TCC通过人工编码来实现数据恢复。需要实现三个方法：

* Try：资源的检测和预留；
* Confirm：完成资源操作业务；要求 Try 成功 Confirm 一定要能成功。
* Cancel：预留资源释放，可以理解为try的反向操作。

#### 流程分析

举例，一个扣减用户余额的业务。假设账户A原来余额是100，需要余额扣减30元。

* 阶段一（ Try ）：检查余额是否充足，如果充足则冻结金额增加30元，可用余额扣除30

初始余额：

余额充足，可以冻结：



此时，总金额 = 冻结金额 + 可用金额，数量依然是100不变。事务直接提交无需等待其它事务。

* 阶段二（Confirm)：假如要提交（Confirm），则冻结金额扣减30

确认可以提交，不过之前可用金额已经扣减过了，这里只要清除冻结金额就好了：



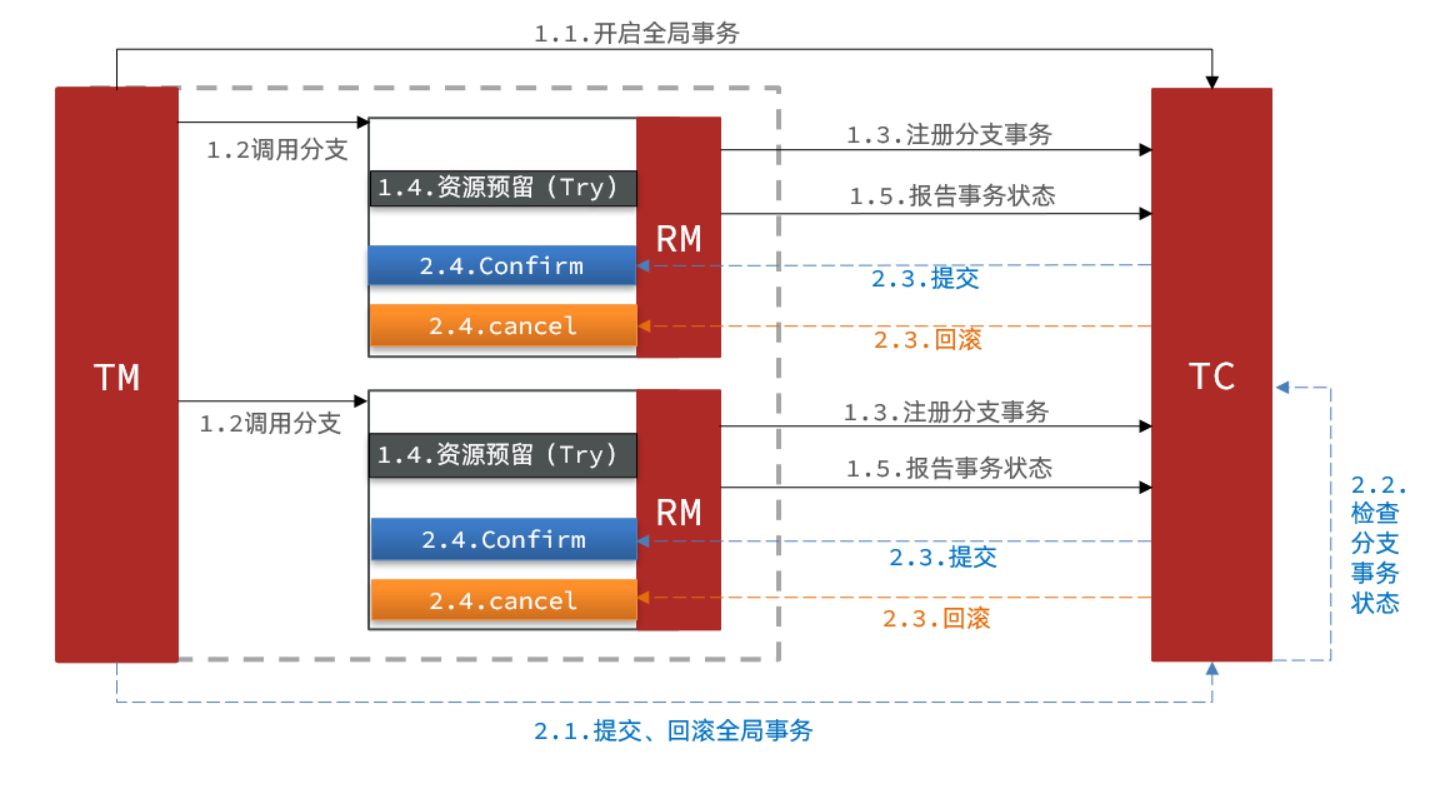
此时，总金额 = 冻结金额 + 可用金额 = 0 + 70 = 70元

* 阶段二(Canncel)：如果要回滚（Cancel），则冻结金额扣减30，可用余额增加30

需要回滚，那么就要释放冻结金额，恢复可用金额：



#### Seata的TCC模型



一阶段，首先TM会向TC提交开启事务，TM会调用各个微服务，执行各个微服务的分支事务，RM会先向TC注册分支事务，然后再去执行分支事务，进行资源预留，执行完毕之后提交事务，去向TC报告事务的状态。

二阶段，TM发现事务结束了，通知TC，然后TC会去检查分支事务的状态，如果各个分支事务都成功，则进行分支事务的confirm，会完成资源的操作，如果有一个失败，则进行分支事务的cancel，进行预留资源的释放。

#### 优缺点

TCC模式的每个阶段是做什么的？

* Try：资源检查和预留
* Confirm：业务执行和提交
* Cancel：预留资源的释放

TCC的优点是什么？

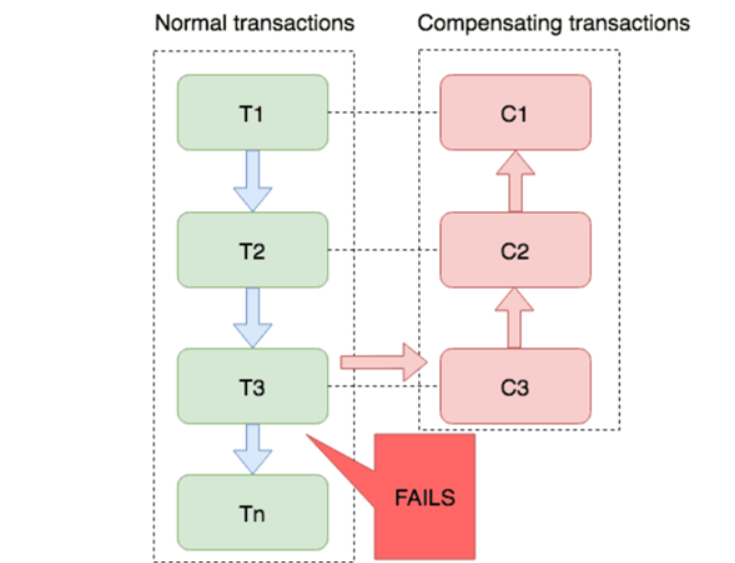
* 一阶段完成直接提交事务，释放数据库资源，性能好
* 相比AT模型，无需生成快照，无需使用全局锁，性能最强
* 不依赖数据库事务，而是依赖补偿操作，可以用于非事务型数据库

TCC的缺点是什么？

* 有代码侵入，需要人为编写try、Confirm和Cancel接口，太麻烦
* 软状态，事务是最终一致
* 需要考虑Confirm和Cancel的失败情况，做好幂等处理

### SAGA模式

在 Saga 模式下，分布式事务内有多个参与者，每一个参与者都是一个冲正补偿服务，需要用户根据业务场景实现其正向操作和逆向回滚操作。分布式事务执行过程中，依次执行各参与者的正向操作，如果所有正向操作均执行成功，那么分布式事务提交。如果任何一个正向操作执行失败，那么分布式事务会去退回去执行前面各参与者的逆向回滚操作，回滚已提交的参与者，使分布式事务回到初始状态。



Saga也分为两个阶段：

一阶段：直接提交本地事务

二阶段：成功则什么都不做；失败则通过编写补偿业务来回滚

#### 优缺点

优点：

* 事务参与者可以基于事件驱动实现异步调用，吞吐高
* 一阶段直接提交事务，无锁，性能好
* 不用编写TCC中的三个阶段，实现简单

缺点：

* 软状态持续时间不确定，时效性差
* 没有锁，没有事务隔离，会有脏写

### 四种模式对比

我们从以下几个方面来对比四种实现：

* 一致性：能否保证事务的一致性？强一致还是最终一致？
* 隔离性：事务之间的隔离性如何？
* 代码侵入：是否需要对业务代码改造？
* 性能：有无性能损耗？
* 场景：常见的业务场景

