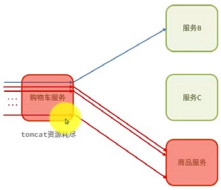
微服务保护

1、雪崩问题

微服务调用链路中的**某个服务故障**，**引起**整个链路中**所有微服务都不可用**。

【原因】：**微服务相互调用**，***服务提供者***出现**故障或阻塞**，***服务调用者***没有做好异常处理，大量去向该故障服务的**请求堆积**，**服务资源有限**，导致**自己故障**；甚至**调用链中**所有服务**级联失败**，导致整个集群故障。



2、服务保护方案

1）请求限流：限制访问微服务的**请求并发量**，避免服务**因流量激增而故障**；

2）线程隔离：通过**限定每个业务**能**使用的线程数量**，而**将故障业务隔离**；

3）服务熔断：由**断路器**统计请求的**异常比例**或**慢调用比例**（业务响应时间大于指定时长的请求），如果**超出阈值**则会**熔断该业务**，**拦截该接口**的所有请求，**熔断期间**，所有**请求快速失败**，全都走 fallback 逻辑；

4）失败处理：定义 fallback 逻辑，让业务失败时不抛出异常，而是返回友好提示。

（**超时处理**：发送的请求 1s 调用不到就返回错误信息，而不是无休止等待。但是请求输入多余请求输出时便无能为力）

3、sentinel

Sentinel 是一款**微服务流量控制组件**。

簇点链路：是**一次请求进入**服务后经过的每一个**被 sentinel 监控的资源链**。默认 sentinel 会监控 **springMVC 的每一个 http 接口**。

限流、熔断等都是对簇点链路中的**资源设置**；而**资源名**就是**接口中的请求路径**。

【注意】Restful 风格的 API 请求路径一般都相同，需要**修改配置**，将**请求方式+请求路径**作为**簇点资源名称**。

http-method-specify:true #开启请求方式前缀

4、请求限流

给某个资源设置流控规则，**QPS 的阈值**；

【3 种流控模式】：

1）直接（默认）：**统计当前资源**的请求，触发阈值时**对*当前资源*直接限流**。

2）关联：**统计**与当前资源相关的**另一个资源**，触发阈值时，**对*当前资源*限流**；

例：查询和修改操作会争抢数据库锁，若修改操作优先，则当修改操作触发阈值，限流查询。

3）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**。

例：服务 A 和服务 B 都调用服务 C，服务 A 并发量过大，就统计服务 A 访问服务 C 的请求，触发阈值，服务 C 对服务 A 的请求限流。

【3 种流控效果】指达到阈值时采取措施；

1）快速失败（默认）：达到阈值后，新的请求会被**立即拒绝**，并**抛 FlowException 异常**；

2）warm up：预热模式，对超出阈值的请求同样拒绝并抛异常，但**阈值会动态变化**，从**最大 QPS/3** 逐渐加到**最大 QPS**；（coldFactor=3）

3）排队等待：让**所有请求**按**先后次序排队执行**，**两请求间隔**不小于指定时长，否则被拒绝。

【热点参数限流】更加细粒度到某个请求分别统计***参数值相同***的请求，判断是否超或 QPS 阈值。对于**热点资源 QPS 设置高点**。

5、熔断降级步骤（服务状态机有 3 个状态）

1）closed 状态：**正常**状态；所有的请求都可以进入，此时，***断路器***统计**调用异常比**；

2）open 状态：**停止**状态；如果调用异常比**超过阈值**，则会**快速失败，熔断服务**，此时会有**熔断时间**；

3）half-open 状态：**放行一次请求**，根据该请求**结果调整状态**；若成功则 closed，反之 open。

【熔断策略】*慢调用*比例；***异常***比例；***异常数***。

服务限流详解

6、高可用

高可用是指系统即使在**发生硬件故障、服务故障或系统升级**的时候，**服务仍然是可用**的。

可用性的判定标准：1）该系统**所有运行时间**中可**用时间占比**多少个 9（99.99999%）；

2）某功能**失败次数**与**总请求次数**之比来衡量。

7、提高系统高可用性的方法

1）使用**集群**；2）**请求限流**；3）**超时重试**；4）**熔断**机制；5）**异步调用**；6）**冗余设计**。

8、冗余设计

是保证**系统**和**数据*高可用***的常用手段。

冗余思想就是**相同的服务/数据部署多份**，如果使用的**服务挂了**，可以**快速切换到备份上**。

1）高可用集群：**同一份服务部署多份**；当前服务**挂了可以快速切换**；（一整个集群可以部署在同一个机房）

2）同城灾备：相同服务部署**在同一城市的不同机房中**，**备用服务*不处理请求***；（可以避免机房意外情况停电、火灾）

3）异地灾备：相同服务部署**在异地不同机房**；

4）同城多活：类似同城灾备；（**同时提供服务**）

5）异地多活：类似异地灾备；（**同时提供服务**）

【注意】光有冗余并不够，**需要故障转移**；实现**不可用服务**快速且***自动地切换***到**可用服务**，整个过程不需要人为干涉。比如 **sentinel 检测到节点有问题**，帮助我们实现故障转移。

9、线程隔离（线程池隔离/信号量隔离）

1）线程池隔离：（hystix）利用**线程池本身设置的线程上限**，起到**服务间隔离**和**限流**效果。隔离性非常好，**每个服务有独立线程池**；但是，线程过多会有**额外的 cpu 开销**。

2）信号量隔离：（sentinel）直接**使用 Tomcat 上申请到的线程**，使用**信号量**，对用户请求的使用的线程**数量进行记录**，达到**信号量上限**则**拒绝后续请求**。（**没有额外**的资源消耗）

10、固定窗口计数器算法

即***限制*单位时间处理的请求数量**。

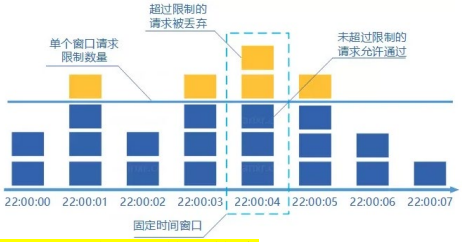
1）将**时间**划分为多个**固定大小的窗口**；

2）**每个窗口**分别计数统计，**每有一次请求将计数器加 1**；

3）如果**计数器超过限流阈值**，**拒绝**后续请求。

【缺点】1）**限流不够平滑**；例如单位时间内某一段时间集中处理，部分时间空闲不处理；

2）**无法保证限流速率，无法应对激增的流量**；例如单位时间允许的请求量在某一秒一起来，可能超过该接口的 QPS，**系统过载**。



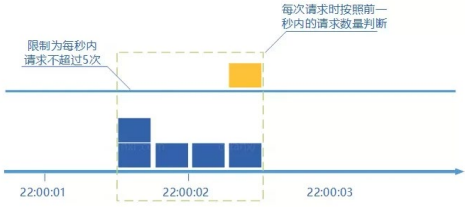
11、滑动窗口计数器算法

一个窗口内部划分为更小的区间。

窗口会根据当前请求所在时间移动，窗口范围从**当前时间-窗口大小**的**后一个时区**开始，到**当前所在时区**。

【优点】**可应对突然激增**的流量；**颗粒度更小**；

【缺点】**限流不够平滑**；实现麻烦。

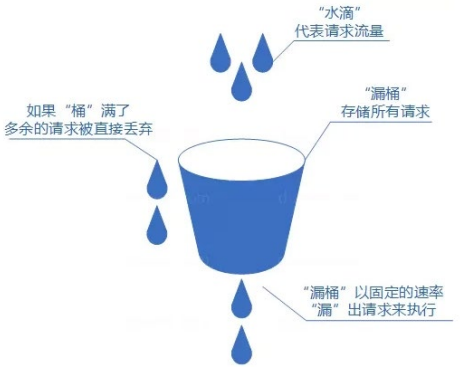


12、漏桶算法（排队等待）

把**发送请求的动作**比喻成**注水到桶中**，**处理请求**比喻成**漏桶漏水**。以***任意速率***注水，以***一定速率***出水，当**桶满**时，**拒绝**后续请求。

【优点】**可控制限流速率**；实现简单；

【缺点】**无法应对突然激增**的流量；如果注水速率一直大于漏水速率，**部分新请求会被拒**。

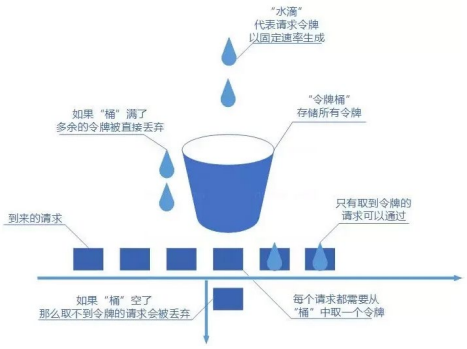


13、令牌桶算法（热点参数限流）

和漏桶算法一样，不过桶里装的令牌，**请求在被处理之前拿到一个令牌**，处理完后令牌丢弃；以***一定速率***向桶里***添加令牌***。

【优点】**可限制限流速率**；**动态调整令牌生成**；

【缺点】实现复杂。



【注意 sentinel 限流和 gateway 限流区别】

1）**gateway** 是基于**redis 实现的令牌桶算法**；

2）**Sentinel** 限流有 3 种实现（滑动窗口/漏桶/令牌漏桶），**默认限流**模式是**滑动窗口**，另外**断路器**也是**滑动窗口**；

3）**限流后**可以**快速失败**和**排队等待（漏桶）**；

4）**热点参数限流**基于令牌漏桶算法。

【sentinel 工作主流程】

在 sentinel 中，所有资源对应一个资源名称和一个 entry，每个 entry 创建的时候同时创建一系列功能插槽。

1）NodeSelectorSlot 调用链路构建：负责手机资源路径，并将这些资源的调用路径，以树结构存储起来，用于根据调用路径来限流降级；

2）ClusterBuilderSlot 统计簇点构建：用于存储资源的统计信息以及调用者信息，作为多维度限流降级依据；

3）StatisticSlot 监控统计：用于统计运行时指标信息；

（下面进行相应的操作）

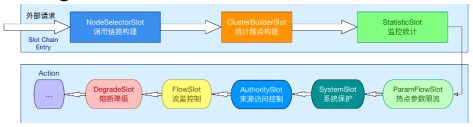
4）ParamFlowSlot：热点参数限流；

5）SystemSlot：根据系统状态控制总入口流量；

6）AuthoritySlot：根据黑白名单信息做控制；

7）FlowSlot：根据前面限流状态进行流量控制；

8）DegradeSlot：根据前面状态进行熔断降级；



分布式事务

1、分布式事务

在分布式系统中，如果一个业务**需要多个服务合作完成**，而且**每个服务都有事务**，**多个事务必须同时成功或失败**。

分支事务：每个服务的事务；

全局事务：整个业务。

2、seata 事务管理中有三个重要角色

1）TC-事务协调者：维护**全局和分支事务的*状态***，协调**全局事务提交或回滚**；

2）TM-事务管理器：定义**全局事务的*范围***，**开启**全局事务、**提交**或**回滚**全局事务；

3）RM-资源管理器：**管理分支事务**，**与 TC 传递信息**，以**注册分支事务**和**报告分支事务的状态**。

3、XA 模式（两个阶段）

阶段 1：

1）开启全局事务；

2）TM 调用分支事务；

3）各微服务中 RM 会**拦截**对数据库的操作，然后在 TC 中**注册**分支事务；

4）各微服务中 **RM 放行**，执行业务的 sql 语句，但**执行完不会提交**；（此时事务锁不释放）

5)各微服务中 RM 向 TC 报告分支事务状态；

阶段 2：

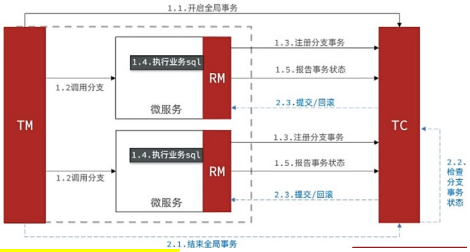
1）TM 向 TC 发送结束全局事务的信号；

2）TC **检查分支事务状态**；

3)RM 接收 TC 指令，**提交或回滚**；（释放锁）

【优点】事务**强一致性**，满足 ACID 原则；

【缺点】数据库资源**锁定一段时间**，性能较差。



4、实现 XA 模式

1)修改 yml 配置文件（参与事务的微服务都）

data-source-proxy-mode：XA

2）全局事务方法加@GlobalTransactional 注解

5、AT 模式（主推）

阶段 1：

1）开启全局事务；

2）TM 调用分支事务；

3）各微服务中 RM 会**拦截**对数据库的操作，然后在 TC 中注册分支事务；

4）各微服务中 RM 放行，注意**要记录更新前后的数据快照**，执行业务的 sql 语句，**并提交**；

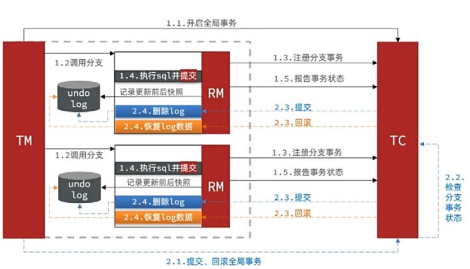
5)各微服务中 RM 向 TC 报告分支事务状态；

阶段 2：

1）TM 向 TC 发送结束全局事务的信号；

2）TC 检查分支事务状态；

3）都**成功提交**，则**删除数据快照**；**有失败**则**回滚**事务，根据数据快照**恢复到更新前状态**。



6、XA 模式和 AT 模式区别

1）XA：一阶段**不提交**事务，**锁定资源**；

AT：一阶段**直接提交**，**不锁定资源**；

2）XA：**依赖数据库机制**实现**回滚**；

AT：利用**数据快照**实现**数据回滚**；

3）XA：**强一致性**；（CP）

AT：**最终一致性**；（数据恢复有毫秒级不一致）

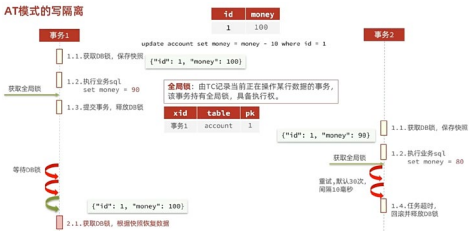
7、AT 模式的脏写问题

AT 模式是分为两个阶段执行的，如果**事务之间没有做好隔离，阶段 1 和阶段 2 之间有时间间隔**，可能导致有**其他同时事务见缝插针**，成功处理该数据，最后**阶段 2 如果要回滚**，则数据快照**覆盖**了更新的值，**产生了脏写**问题。

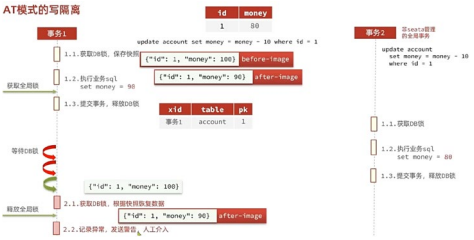
【解决方案】AT 模式的写隔离

全局锁：由 **TC 记录当前正在操作某行数据**的事务，**该事物持有全局锁**，具备执行权。

（**用一张表**记录全局锁，某个**事务**正在操作某个表的某行数据<主键>）



【注意】**全局锁**只能解决 **seata 管理的事务**之间的隔离问题；对于**非 seata 管理的事务不需要全局锁**就**可以操作数据库**，还是可能会见缝插针，产生脏数据。【解决】同时保存**执行完业务 sql *后*数据快照**，回滚的时候**比对**，有其他事物修改就记录异常，人工处理。



8、TCC 模式（每个阶段都是**独立的事务**）

1）try：资源的**检测和预留**；

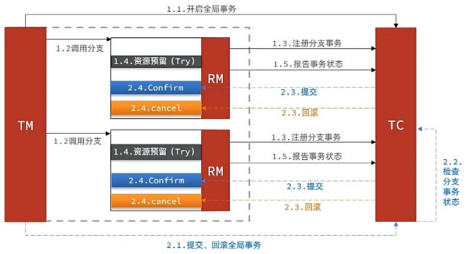
2）confirm：**完成**资源操作业务；

（try 成功 confirm 一定要能成功）

3）cancel：预留资源**释放**；

【优点】1）**无需生成快照**，也**无需使用全局锁**，性能最强；2）**不依赖数据库事务**，而是**依赖补偿**操作。

【缺点】1）需要人工编写代码；2）**事务是最终一致**；3）失败要做好幂等处理。



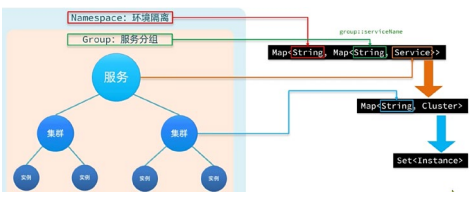
9、最大努力通知（最终一致性）

也就是失败重试，最大努力通知成功。

【微服务相关问题补充】

1、nacos 注册表结构—分级模型

环境隔离 Namespace→服务分组 group→服务→集群→实例（一个一个 map，最后实例 set）



2、nacos 如何应对高并发的注册压力？

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**放入到一个阻塞队列**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**，***异步完成*实例更新**，从而提高并发写能力。

3、nacos 如何解决并发读写冲突问题？

Nacos 在**更新实例列表**时，采用 **CopyOnWrite 技术**，首先将**旧实例列表拷贝**一份，然后**更新拷贝的**实例列表，再用更新后的实例列表**覆盖旧实例列表**。即**在更新过程**中不会对读服务列表操作产生影响，**不会脏读**。

4、Spring Cloud Balancer 内 2 种负载均衡

1）轮询负载均衡策略（默认）

RoundRobinLoadBalancer:getInstanceResponse 方法中利用 **incrementAndGet()方法**做**原子加操作**，并利用& Integer.MAX_VALUE 位运算保证不会产生负数，最后对实例数取余。

2）随机负载均衡策略

return 一个 RandomLoadBalancer 对象；

【注意】

1）设置局部负载均衡

service 接口上加**@LoadBalancerClients 注解**；

2）设置全局负载均衡

在**启动类**上加@LoadBalancerClients 注解。