限流、降级、熔断实践分享

> 上集总结简单回顾：如何保障xx稳定性 v1.0 ，5min

1.【稳定性策略介绍】

保障好高并发系统稳定性的三把斧，分别是：**缓存、降级和限流**

- **缓存**：降低系统处理单请求资源占用，提高系统吞吐。
- **降级**：合理分配有限的系统资源，用户体验有损感知下降到最低。
- **限流**：保障系统不出现过载，保命。

可能大家还有一个疑问？那么“**熔断**”是什么？其实简单理解为“**熔断**”是被动的限流。

2.【稳定性策略优先级】

> 稳定性问题就如程序中的bug，我们没办法在有限的时间做到面面俱到，因此优先级的区分是做好在有限资源取得最优解的基础。
>
> 上集实际上少说了一个问题？常见的生产环境问题的原因是什么？常见程度从高到低：
>
> 1. 变更（代码变更、配置变更、服务配置变更）->95%+
>   a. 变更导致=>不负责任预估95%+
> 2. 基础依赖组件（14/257）
>   a. 变更导致=>86%
>   b. 性能问题=>14%

个人认为稳定性策略优先级如下：限流>缓存>降级>熔断

- 限流：零开发配置，成本低。
  - 接入服务>核心服务>存储>其他
- 缓存：组件方式开发接入，成本中等。
- 降级：定制开发，成本高。
- 熔断：与限流配置合适的效果等同且无特殊定制重试逻辑，真实场景使用较少，注意这里不是说没有作用，只是优先级没那么高。

监控、报警偏向于发现问题，资源隔离偏向控制爆炸范围，此处暂不讨论。

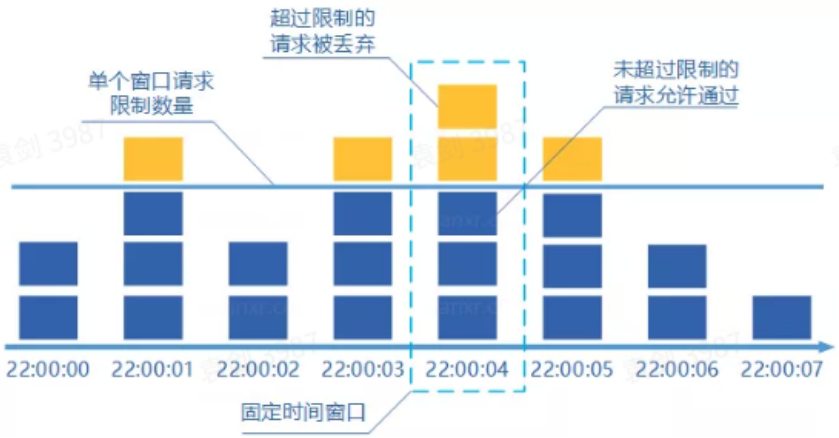
3.【稳定性策略详细介绍】

3.1. 限流策略

3.1.1. 限流策略原理

3.1.1.1. 计数器算法

固定窗口计数器



算法描述：

- 固定时间窗口内对请求进行计数，与阀值进行比较判断是否需要限流，一旦到了时间临界点，将计数器清零。

算法优点：

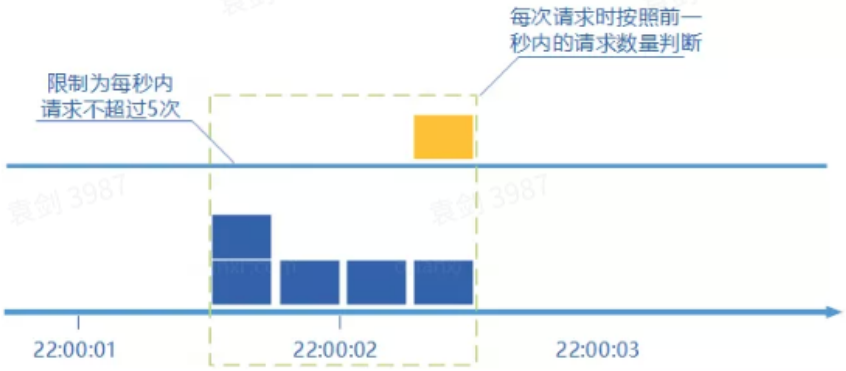
- 代码实现成本低。

算法缺点：

- 跨时间窗口会出现限流不准确，如图所示：

> 假设限制每秒通过 5 个请求，时间窗口的大小为 1 秒，当前时间窗口周期内的后半秒正常通过了 5 个请求，下一个时间窗口周期内的前半秒正常通过了 5 个请求，在这两个窗口内都没有超过限制。但是在这两个窗口的中间那一秒实际上通过了 10 个请求，显然不满足每秒 5 个请求的限制。

滑动窗口计数器



算法描述：

- 滑动窗口算法将一个大的时间窗口分成多个小窗口（本质上把大时间窗口细化了），每次大窗口向后滑动一个小窗口，并保证大的窗口内流量不会超出最大值。

算法优点：

- 实现比固定窗口的流量曲线更加平滑。

算法缺点：

- 没有真正解决固定窗口算法的临界突发流量问题。

3.1.1.2. 漏桶算法

(a) A leaky bucket with water. (b) a leaky bucket with packets.

算法描述：

- 将进来的请求流量视为水滴先放入桶内。
- 水从桶的底部以固定的速率匀速流出，相当于在匀速处理请求。
- 当漏桶内的水满时(超过了限流阈值)则拒绝服务。

算法优点：

- 限制数据的传输速率。

算法缺点：

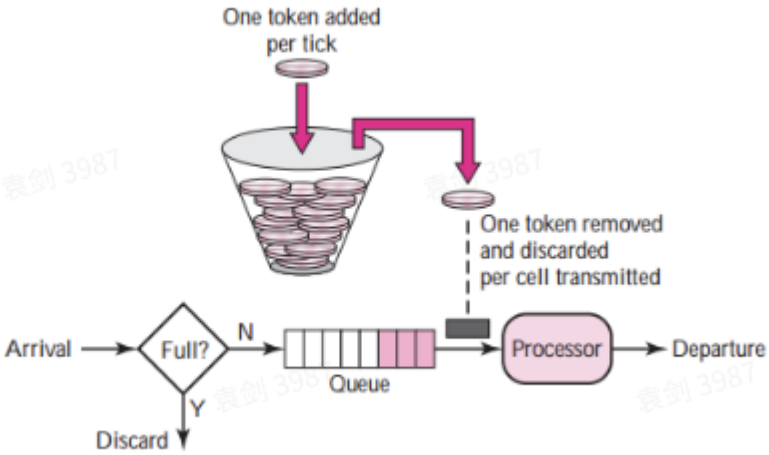
- 突发流量丢弃请求较多。

算法代码描述，参考：https://zhuanlan.zhihu.com/p/338368004

Go

```go
// 漏桶
// 一个固定大小的桶，请求按照固定的速率流出
// 如果桶是空的，不需要流出请求
// 请求数大于桶的容量，则抛弃多余请求

type LeakyBucket struct {
    rate       float64    // 每秒固定流出速率
    capacity   float64    // 桶的容量
    water      float64    // 当前桶中请求量
    lastLeakMs int64      // 桶上次漏水微秒数
    lock       sync.Mutex // 锁
}

func (leaky *LeakyBucket) Allow() bool {
    leaky.lock.Lock()
    defer leaky.lock.Unlock()

    now := time.Now().UnixNano() / 1e6
    // 计算剩余水量,两次执行时间中需要漏掉的水
    leakyWater := leaky.water - (float64(now-leaky.lastLeakMs) * leaky.rate / 1000)
    leaky.water = math.Max(0, leakyWater)
    leaky.lastLeakMs = now
    if leaky.water+1 <= leaky.capacity {
        leaky.water++
        return true
    } else {
        return false
    }
}

func (leaky *LeakyBucket) Set(rate, capacity float64) {
    leaky.rate = rate
    leaky.capacity = capacity
    leaky.water = 0
    leaky.lastLeakMs = time.Now().UnixNano() / 1e6
}
```

3.1.1.3. 令牌桶算法



算法描述：

- 按照一定的速率生产令牌并放入令牌桶中。
- 如果桶中令牌已满，则丢弃令牌。
- 请求过来时先到桶中拿令牌，拿到令牌则放行通过，否则拒绝请求。

算法优点：

- 限制数据的平均传输速率的同时还允许某种程度的突发传输。
- 突发流量丢弃请求少。

算法缺点：

- 代码实现复杂。

算法代码描述，参考：https://zhuanlan.zhihu.com/p/338368004

Go

```go
type TokenBucket struct {
    rate         int64 //固定的token放入速率, r/s
    capacity     int64 //桶的容量
    tokens       int64 //桶中当前token数量
    lastTokenSec int64 //上次向桶中放令牌的时间的时间戳，单位为秒

    lock sync.Mutex
}

func (bucket *TokenBucket) Take() bool {
    bucket.lock.Lock()
    defer bucket.lock.Unlock()

    now := time.Now().Unix()
    bucket.tokens = bucket.tokens + (now-bucket.lastTokenSec)*bucket.rate // 先添加令牌
    if bucket.tokens > bucket.capacity {
        bucket.tokens = bucket.capacity
    }
    bucket.lastTokenSec = now
    if bucket.tokens > 0 {
        // 还有令牌，领取令牌
        bucket.tokens--
        return true
    } else {
        // 没有令牌,则拒绝
        return false
    }
}

func (bucket *TokenBucket) Init(rate, cap int64) {
    bucket.rate = rate
    bucket.capacity = cap
    bucket.tokens = 0
    bucket.lastTokenSec = time.Now().Unix()
}
```

3.1.2. 限流策略实现

> 使用场景：
>
> - 计数器算法：就一个优点是简单。
> - 漏桶算法：严格控制传输速率场景
> - 令牌桶算法：其余常用场景。

3.1.2.1. 单机限流方案

方案分析：

1. 优点是无需任何存储依赖，基于内存高性能。
2. 缺点是重启数据丢失，导致数据无法持久化；基于集群的配置修改麻烦，集群扩容缩容操作都需要调整单机的令牌桶配置，需要换算成每个实例的qps。

3.1.2.2. 分布式限流方案

分布式存储

方案描述：

- 如何实现分布式限流，最简单直观的方案是基于分布式存储实现分流组件。

方案优点：

- 限流精度较高。
- 低限流场景支持。

方案缺点：

- 附带较高的网络成本和存储成本。
- 实现方案复杂，体现如下：
  - 高请求QPS带来的系统架构设计问题，如热点访问（解决方案：远端限流Quota批量获取，本地缓存和限流Quota扣减）
  - 分布式系统天然问题，如分布式一致性（解决方案：Redis+lua、Abase+cas、Bytekv、分布式事务）

多单机限流

> https://codebase.byted.org/repo/amazing/utilsv2/-/blob/limiter/limit_control.go

方案描述：

- 现在要限制某服务方法调用100wQPS，那么如果我共有100个机器实例，那么每个实例限制1wQPS。

方案优点：

- 无需额外网络成本和存储成本。
- 适用于高限流场景。

方案缺点：

- 限流精度较低，集群限流与机器数量强相关，当集群机器数量出现波动时，会出现误限、漏限甚至失效。
- 服务扩容缩容需要手动调节单机限流值（解决方案：基于服务发现能力，做到机器数量调整的同时，自动调整单机限流）

3.1.2.3. 公司限流解决方案

TLB 接口限流方案

可参考：Tlb集群限流使用

支持单机限流和集群（机房）限流（多单机限流）。

通用流量治理 - 限流解决方案

> AGW、Janus限流同方案：ApiGateWay中心式限流设计方案 Janus 分布式限流接入指南

解决方案思路：

- 低QPS=>高精度中心化限流：https://site.bytedance.net/docs/5402/neptune/distributed/#b-wildcard-%E9%80%9A%E9%85%8D%E7%AC%A6
- 高QPS=>低精度简单集群限流：https://site.bytedance.net/docs/5402/neptune/simple_cluster/
  - 在部分场景不准，比如请求流量不均匀，限流阈值比较低，每个实例分配到的quota比较少

3.2. 降级策略

> 降级的目标：
>
> 1. 资源优先级重新分配。
> 2. 非核心依赖的事故隔离。

3.2.1. 降级原理

> 好的降级策略离不开对业务的理解，当然同时避免不了定制开发成本。

3.2.1.1. 服务端降级手段

常见的服务端降级手段：

- 拒绝处理请求：详情页CDN回源接口，服务端拒绝请求返回304，CDN缓存将无法被更新。
- 代码不走非必要逻辑：信息流goapi、service和loader非核心依赖调用有降级方案，极端情况确保核心题材可用。
- 请求频率降级：奥运会体育直播间内比分轮询接口，轮询间隔可控。
- 业务兜底逻辑降级：比如抽奖活动时出现的谢谢回顾。
- 负反馈降级：负反馈调节在服务稳定性的应用 分享，越接近降级阀值，降级的力度越狠。

3.2.1.2. 客户端/前端降级手段

常见的展示降级手段：

- 展示兜底：春节红包雨活动期间，服务端异常返回，客户端默认展示“很抱歉未抢到”；春节主会场部分数据服务端不反悔，客户端主动隐藏相关数据展示。
- 请求降级：春节期间地铁模式关闭，客户端请求优先级丢弃。

- 默认配置：L0通参白名单、频道、底tab都有客户端默认配置，即便服务端下发失败，也有兜底策略。

3.2.2. 业务降级方案举例

信息流业务降级+容灾方案：信息流稳定性方案

3.3. 缓存策略

3.3.1. 服务端缓存手段

常见服务端缓存手段：

- 中心式缓存
  - 常用方案举例：Redis、Abase、Memcache
  - 优点：缓存容量相比分布式缓存要大。
  - 缺点：网络IO开销。
  - 常见问题：
    - 用户数据，需要注意多机房同步时延导致的数据短暂不一致。
    - 全局数据，需要考虑大热key处理，大key告警与大key查询
- 分布式缓存
  - 常用方案举例：Localcache
  - 优点：性能好。
  - 缺点：跨机器数据天然不互通。
  - 常见问题：
    - 用户数据，此时用户的数据是有状态的，用户请求路由应该是固定。
- 中心式缓存+分布式缓存多级缓存方案
- Cookie或者类Cookie机制，有大小限制且容易被篡改，不安全。

3.3.2. 客户端/前端缓存手段（不安全）

常见客户端/前端缓存手段，建议配合对称加密和校验和配合使用：

- 文件系统：容易被篡改。
- 本地内存：进程重启会丢失，容易被篡改。

3.4. 熔断策略

> 微服务架构中，服务的依赖和调用关系变得错综复杂，带来灵活性的同时，对服务稳定性也带来了新的隐患。如下图所示，当 “服务C” 出现问题时，可能是宕机，上线出 bug，流量过大或缓存穿透数据库压垮服务：

- 这时“服务 C”响应就会出问题，而“服务 B”由于拿不到响应结果又会不断重试进一步压垮“服务 C”，
- 同时“服务 B”同步调用也会有大量等待线程，出现资源耗尽，导致“服务 B”变得不可用，进而影响到“服务 A”，形成雪崩效应。

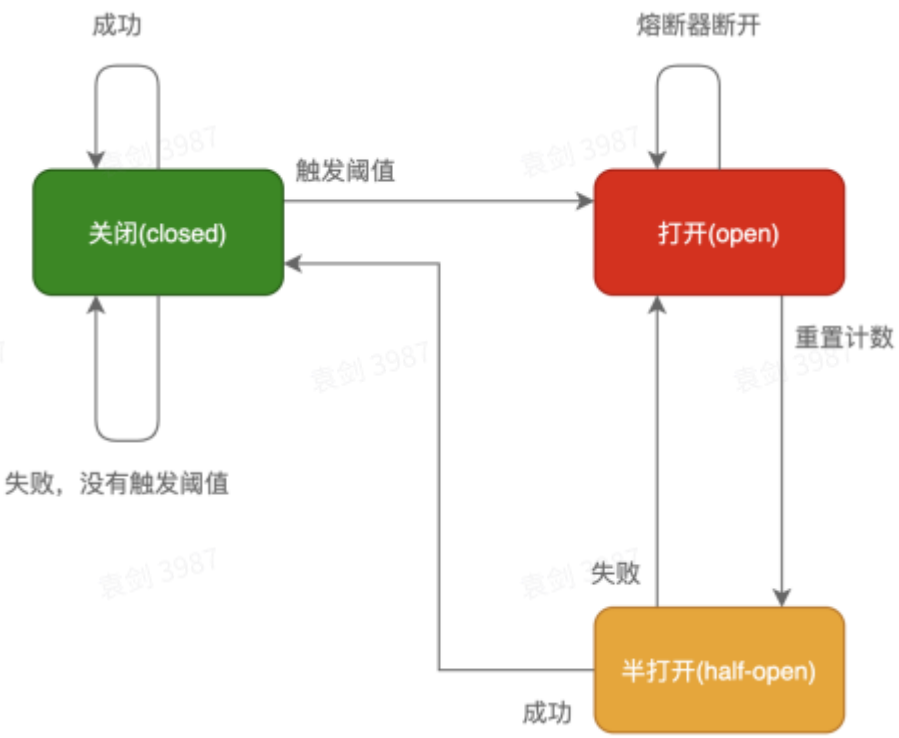
3.4.1. 熔断原理

熔断机制参考了我们日常生活中的保险丝的保护机制，当电路超负荷运行时，保险丝会自动的断开，从而保证电路中的电器不受损害。

服务治理中的熔断机制，指的是在发起服务调用的时候，如果被调用方返回的错误率超过一定的阈值，那么后续的请求将不会真正发起请求，而是在调用方直接返回错误

在这种模式下，服务调用方为每一个调用服务 (调用路径) 维护一个状态机，在这个状态机中有三个状态：

- 关闭 (Closed)：在这种状态下，我们需要一个计数器来记录调用失败的次数和总的请求次数，如果在某个时间窗口内，失败的失败率达到预设的阈值，则切换到打开状态，此时开启一个超时时间，当到达该时间则切换到半打开状态，该超时时间是给了系统一次机会来修正导致调用失败的错误，以回到正常的工作状态。在关闭状态下，调用错误是基于时间的，在特定的时间间隔内会重置，这能够防止偶然错误导致熔断器进去断开状态
- 打开 (Open)：在该状态下，发起请求时会立即返回错误，一般会启动一个超时计时器，当计时器超时后，状态切换到半打开状态，也可以设置一个定时器，定期的探测服务是否恢复
- 半打开 (Half-Open)：在该状态下，允许应用程序一定数量的请求发往被调用服务，如果这些调用正常，那么可以认为被调用服务已经恢复正常，此时熔断器切换到关闭状态，同时需要重置计数。如果这部分仍有调用失败的情况，则认为被调用方仍然没有恢复，熔断器会切换到打开状态，然后重置计数器，半打开状态能够有效防止正在恢复中的服务被突然大量请求再次打垮。



3.4.2. 熔断如何计算到达阈值?

> 举例常见熔断器的开源实现方案：Sentinel
>
> https://sentinelguard.io/zh-cn/blog/sentinel-vs-hystrix.html

可参考Golang版本的Sentinel：https://github.com/alibaba/sentinel-golang/blob/master/core/circuitbreaker/circuit_breaker.go，底层基于LeapArray这种数据结构来实现滑动窗口，可参考https://github.com/alibaba/sentinel-golang/blob/194d4be01dfe65559fd95b7ca4896eb6bda0c531/core/stat/base/leap_array.go#L176。

数组和时间区间进行映射，每个数组元素代表一个窗口，在获取指定下标对应的窗口时，要分情况进行处理：

- 如果对应下标窗口为null，那么就是第一次进入，创建新窗口并使用cas设置。如果非空走下面的逻辑。
- 如果获取到的窗口开始时间等于当前时间计算出来的对应窗口开始时间，那么就拿到了当前时间需要的窗口，直接返回。
- 如果获取到的窗口开始时间小于当前时间计算出来的对应窗口开始时间，那么就说明这个窗口已经过期了，所以加锁重置，然后重复使用。
- 当前时间小于旧的窗口的开始时间，理论上来说是不应该出现这种情况的，如果存在这种情况，那么返回一个无效的空窗口。

4.【FAQ】

Q1. 公司已经有通用的限流和降级策略了？为什么信息流/业务还需要自己搞?

A1：主要成本和效果的问题，通用的流量治理策略粒度大效果一般但成本低，业务的细粒度降级效果好但成本高。

比如信息流降级，普通接口降级和限流都是一刀切，但是在实际上可以做得更好，比如优先保障推荐和重要频道，比如保障核心题材等。

5.【参考文档】

头条主端&Lite春节活动 - 稳定性保障预案

信息流稳定性方案