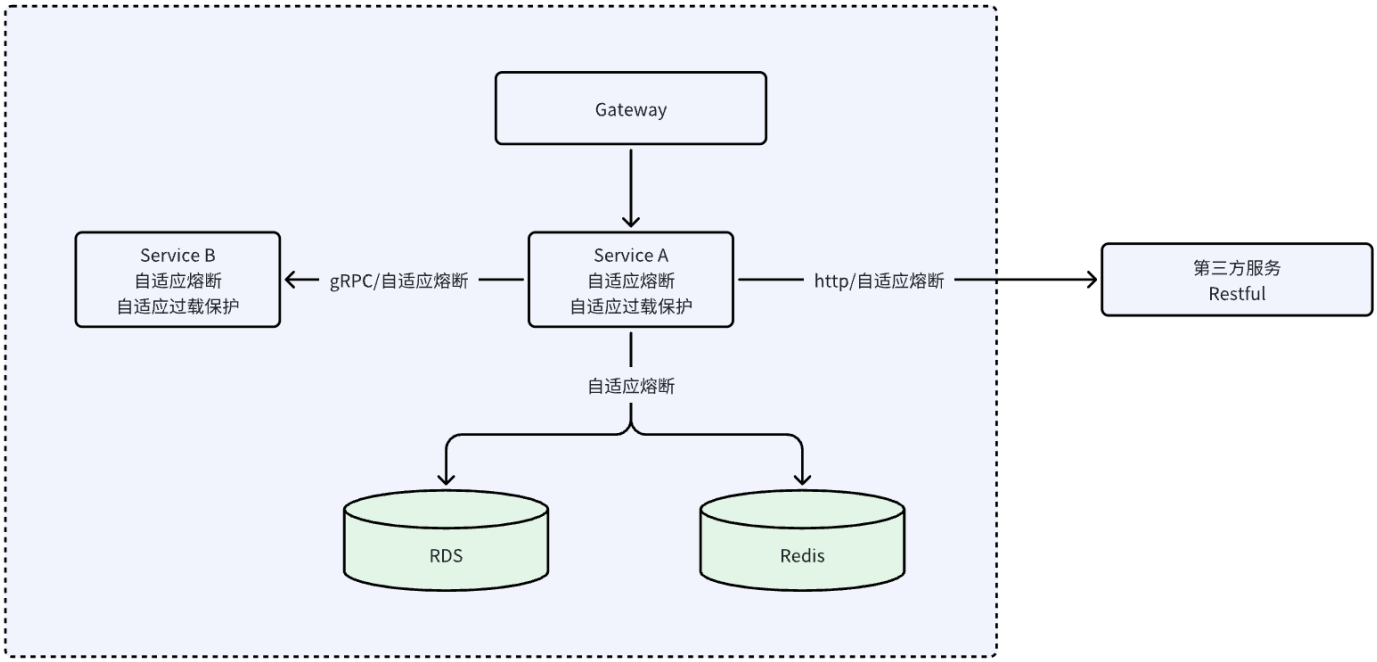
微服务容灾治理

1. go-zero 稳定性能力概览

经过这么多年大流量服务端架构设计的沉淀，go-zero 在保护服务的稳定性上下足了功夫，不管是 CPU 密集型还是 IO 密集型服务，go-zero 都能很好的保护服务在如下场景不被拖垮或卡死：

- 远超服务容量的突发大流量
- CPU 打满
- 上下游故障或者超时
- MySQL、MongoDB、Redis 等中间件故障或者超负载（典型的是 CPU 飙高）



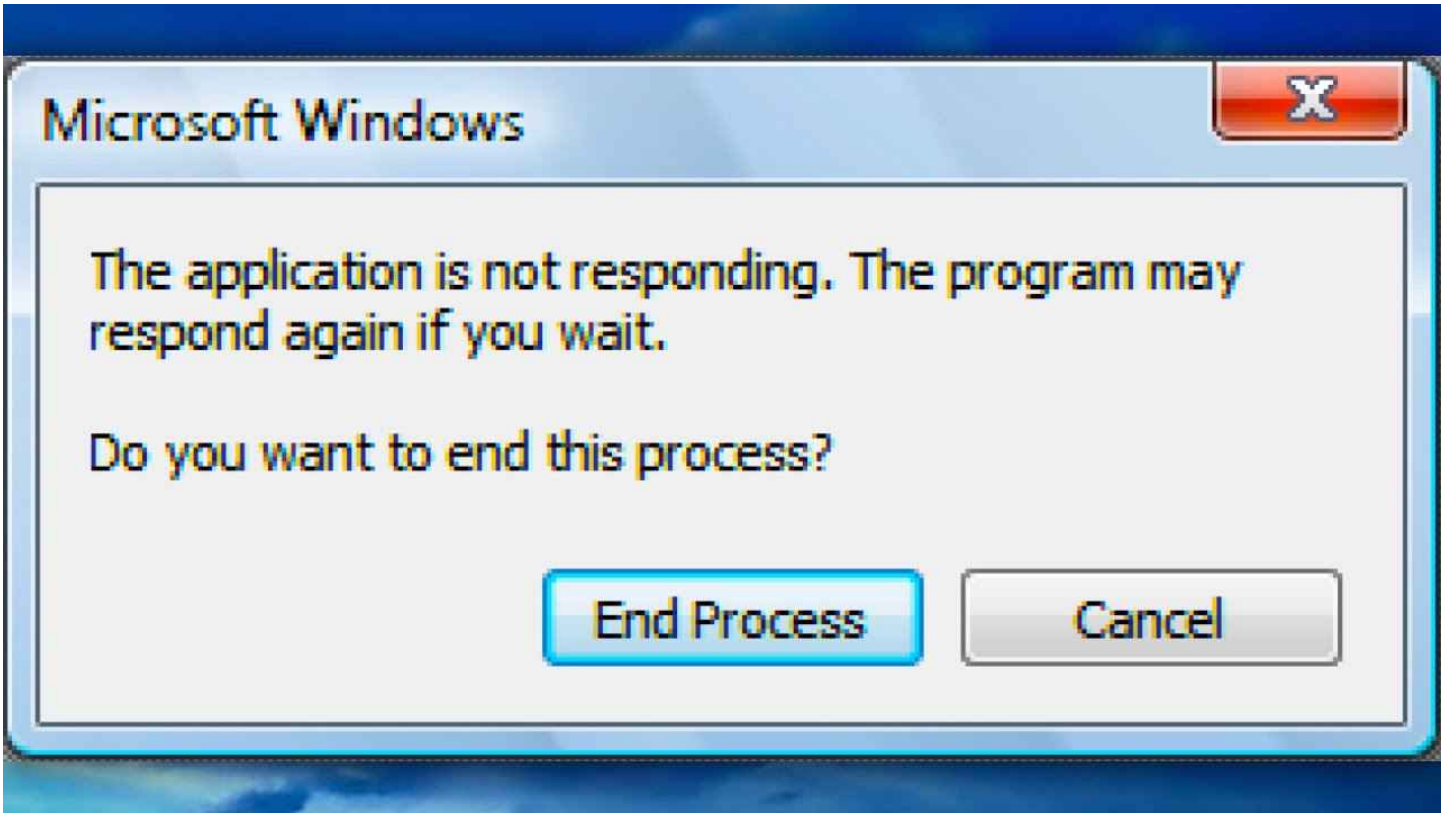
如图，我们从三个方面来保护系统的稳定性：

- 服务端自适应过载保护
- 服务端自适应熔断
- 客户端自适应熔断

当然，我们还有自动适配后端服务能力的负载均衡算法，对稳定性进一步保驾护航。本文主要讲解自适应过载保护的原理、场景和表现。

2. 自适应过载保护压测

用过 Windows 的同学对这个界面应该都不陌生，这就是典型 CPU 打满服务不可用的表现。此时，我们一般都是心里默默骂一句，然后点左边那个按钮，对吧？



那我们想想，如果我们的服务 CPU 被打满了，是不是后面所有的请求也都被卡住了？等服务处理完请求的时候，用户那里可能已经超时离开了，结果服务器很忙，但都是做的无用功。如果这里不能理解，停下来好好思考一番，如果还不懂的话，可以来 go-zero 群里讨论讨论。。。

2.1 模拟 CPU 密集型服务

有人可能会问 CPU 密集型服务怎么定义？你的服务 CPU 会打满吗？处理请求会包含复杂的计算逻辑吗？你经常需要通过 cpu profiling 来优化性能吗？可以理解为服务的 IO 比较快，或者比较少，瓶颈是在 CPU 消耗上。

你可以直接用 `goctl quickstart -t mono` 命令生成一个 HTTP 服务，然后在 logic 代码里加上模拟 CPU 负载的请求处理代码。

模拟 CPU 计算的代码：https://gist.github.com/kevwan/ccfaf45aa190ac44003d93c094a12c3f

```
benchmarkCPU-10        330    3600743 ns/op
```

从 `benchmark` 结果可以看出单个请求的逻辑处理需要 3.6ms CPU资源（不包括服务端中间件处理消耗）。对于两核的容器来说，qps 上限约为 550（2000/3.6）。但是我们是一个 HTTP server，肯定还有接受请求、解析请求、返回结果等开销，实际上是达不到 550qps 的。

这个模拟 CPU 的代码本身不重要，就不做介绍了。

2.2 压测场景

2.2.1 场景一（不开启过载保护）

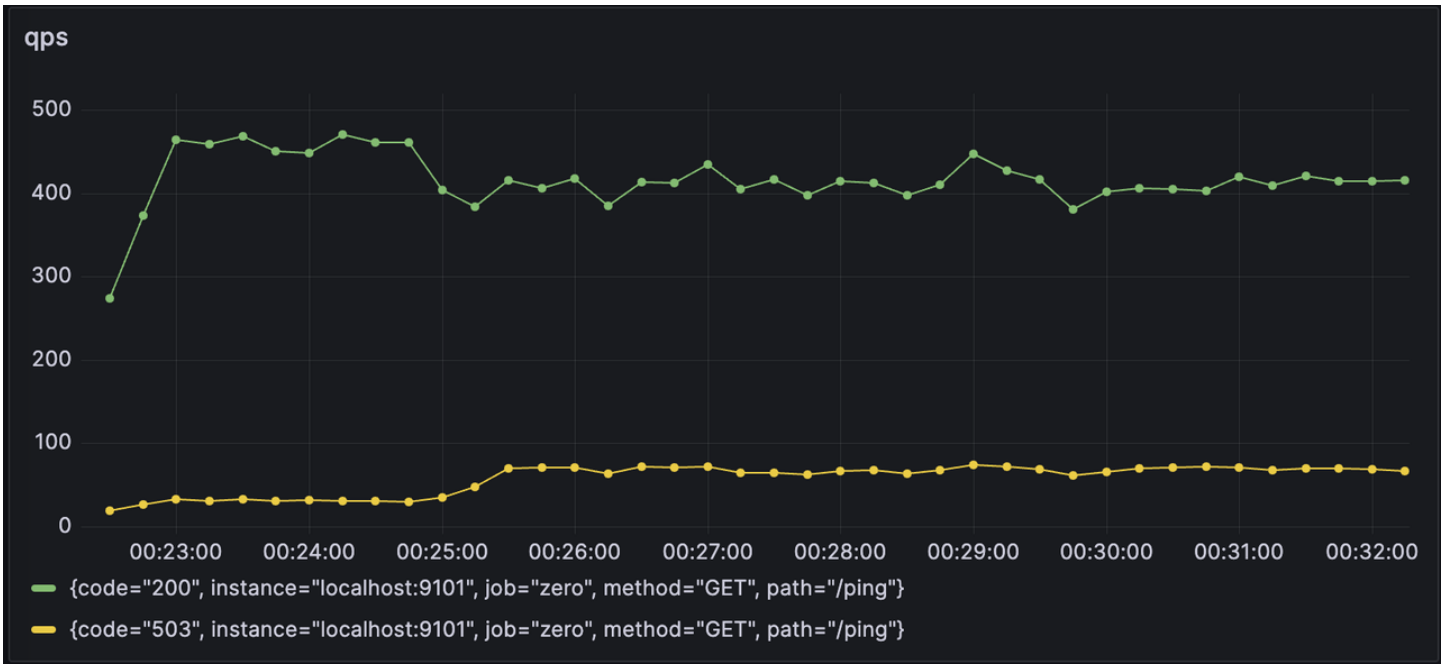
Timeout: 1000

Middlewares:

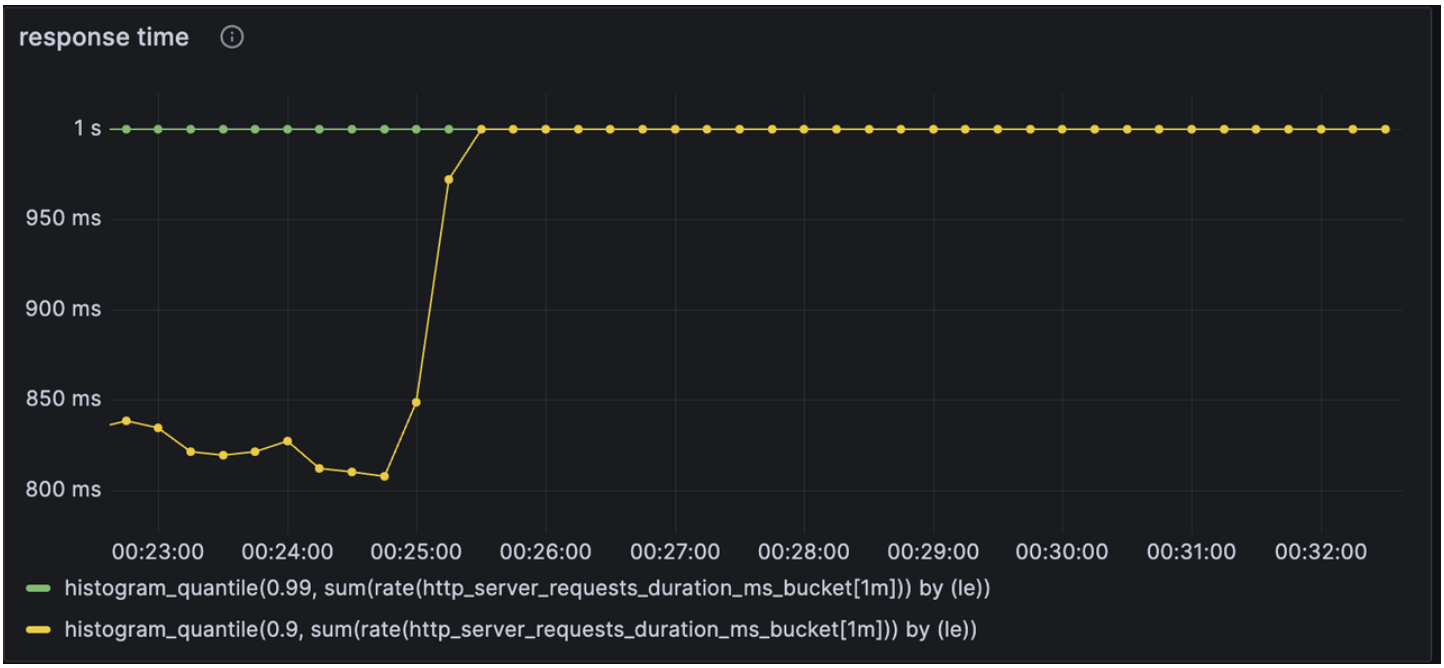
Breaker: false

Shedding: false

- 服务跑在两核的容器内
- 不开启过载保护
- 超时 1s
- `loops 2 hey -c 200 -z 60m "http://localhost:8888/ping"`
  - `loops` 是我的一个 `alias`：`loops='fs() {for i in {1..$1}; do ${@:2} & done; wait}; fs'`，用来并行执行给定命令指定的次数



- 可以看到系统总共只处理了大概 500qps 的请求，其中 400qps 多一点是成功的，近 100qps 是超时的（返回了 503 状态码）



- 随着请求的堆积，很快就会大量请求都超时了，并且p99，甚至p90都已经超过 1s 了
- 这里进一步解释一下，超时的请求意味着对系统资源的浪费，比如接受到一个请求，花了不少cpu时间处理完了，然后返回结果时，发现请求已经超时了，用户已经收到了类似“服务器繁忙，请稍后再试！”的提示。这里可能有用户会说，go不是有超时控制吗？这里有两点：
  - 超时不是在代码的每个指令处都能先判断是否超时再执行，比如我们有一个for循环，不会每次都先判断ctx是否超时，然后再执行下一次迭代，如果你真的这样写了，性能可能需要特别关注，得看你每次循环的计算和判断超时的开销对比
  - 即使能够比较好的判断超时，在侦测到超时之前也已经白费了一些系统资源处理请求了

2.2.2 场景二（开启过载保护）

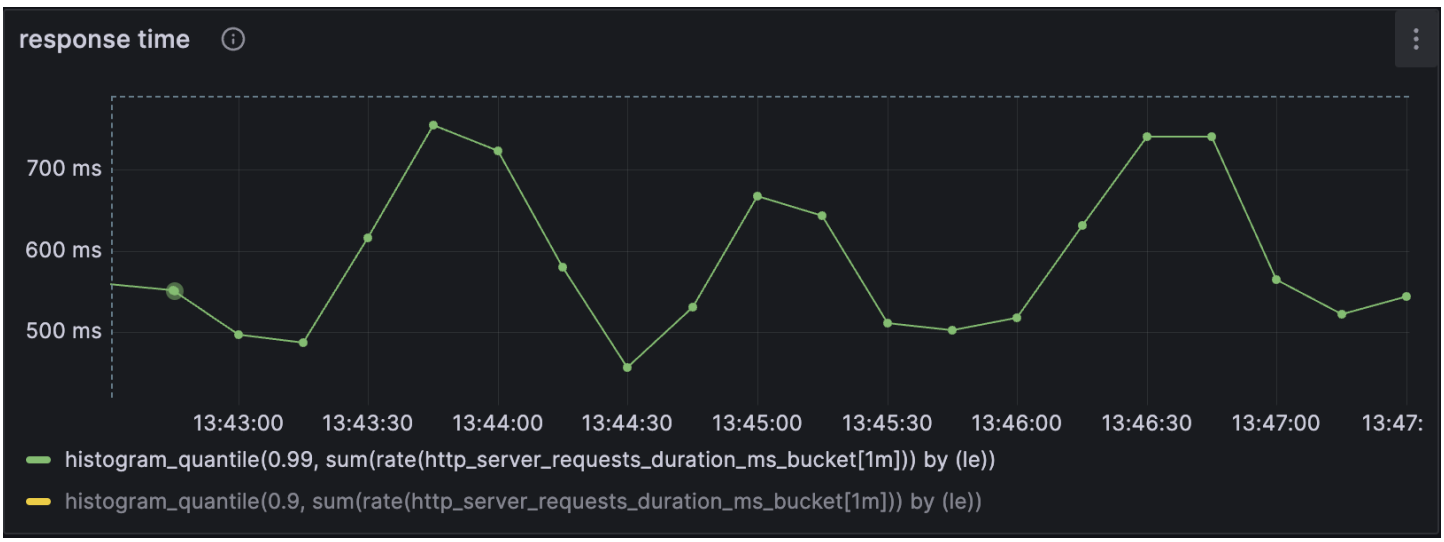
Timeout: 1000

Middlewares:

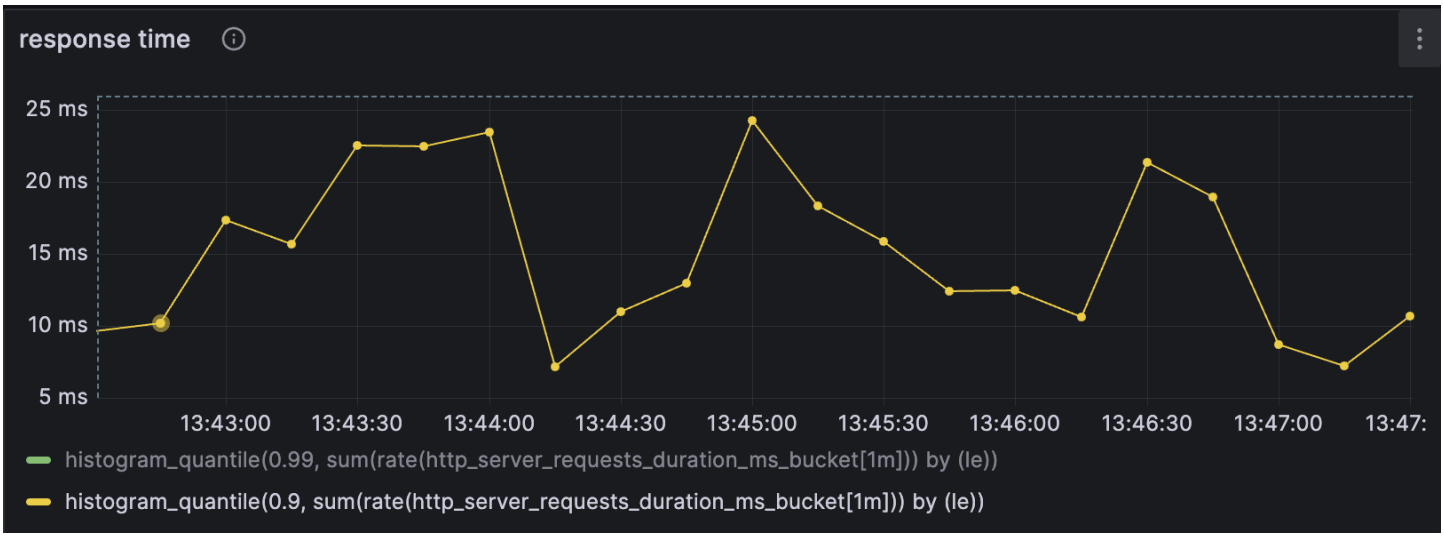
Breaker: false

- 开启过载保护（默认）
- 超时 1s
- `loops 2 hey -c 200 -z 60m "http://localhost:8888/ping"`

- 总 qps 大概在 10000 左右，流量大约是系统容量的 20 倍
- 拒绝了约 95% 的过载请求
- 成功处理请求在 360-400 qps，大概损失了 10% 的 qps，被拒绝的近 1000qps 请求也需要消耗少量系统资源（从接受请求到被拒绝）



- 处理时延 p99 不到 700ms



- 处理时延 p90 不到 25ms

2.3 压测结论：

- 流量未知的情况下，保障系统不卡死（无过载保护情况下，CPU 满载一般表现为大量请求超时），且保证了系统容量的 400 qps 没有大幅下降
- 自动拒绝了过量的请求，避免过量请求浪费系统资源（即使处理，系统最后返回给用户的也是不可用错误、超时错误等）

3. 自适应过载保护原理

先上一张总的设计图，从整体上说明了自适应过载保护的设计思路，也可以看完下面原理解析再回头来看这张图。

3.1 CPU 使用率

CPU 使用率计算是第一个难点，很难算准，依赖于系统繁忙程度，是否能够及时调度到读取 CPU 使用率相关文件的执行。

3.1.1 CPU 检测场景

过载保护的一个触发条件就是当系统的 CPU 高于一个阈值时，go-zero 会自动触发过载保护，那么我们怎么检测 CPU 使用率呢？

首先，我们要明确需要覆盖的场景，当前无外乎虚机和容器两大类了。而容器里又分为 `cgroup v1` 和 `cgroup v2`，所以总的有三类需要处理：

- 虚机（不同云厂商有不同的叫法，比如 ECS, EC2 等）
- 容器 cgroup v1
- 容器 cgroup v2

这里有个特别需要关注的点是：容器是否设置了 `cpu limit`，如果没设，就只能用可以调度的 cpu 个数来计算，比如 `cgroup v2` 里可以读取 `/sys/fs/cgroup/cpuset.cpus.effective` 文件。

这里详细读取 `cgroup` 和 `/proc` 下文件的方法我就不细述了，详见 go-zero 代码。

3.1.2 CPU 使用率计算方法（以 `cgroup v2` 为例）

3.1.2.1 实时 CPU 使用率的计算

- 方法一

$$cpu\ usage = 1000 \times \frac{\Delta usage \times cores}{\Delta system \times quota}$$

- 方法二

$$cpu\ usage = 1000 \times \frac{\Delta usage}{\Delta periods \times quota}$$

go-zero使用了方法一，因为我考虑到 `periods` 更新间隔是 100ms，而 go-zero 检测窗口期是 250ms，这样两次检测的 `Δperiods` 就有时是 2 有时是 3，对计算结果造成非常可见的影响。

3.1.2.2 防止 CPU 使用率毛刺

go-zero 里使用了滑动平均算法（Moving Average）来避免 CPU 的毛刺。比如我们看股票价格曲线时，都会有 MA 线，如图，MA 线在价格波动较大时能够反应较长时间段内的价格变化趋势。对于我们的场景来说，刚好可以反应 CPU 变化趋势的同时也消除了 CPU 毛刺，避免一次小的抖动触发了过载保护。简单的理解滑动平均就是取之前的 N 个值的平均值。

go-zero 里对于滑动平均的超参 `beta` 取值 0.95，相当于最后 20 个值（`1 / (1-0.95) = 20`）的均值，所以 CPU 达到 90% 后约 5 秒会触发过载保护。

3.2 系统容量计算

容量计算是第二个难点，怎么动态评估系统当前容量是算法的关键点之一。计算公式如下：

$$\frac{maxPass \times windows \times minRt}{1000}$$

先解释一下各个参数的含义：

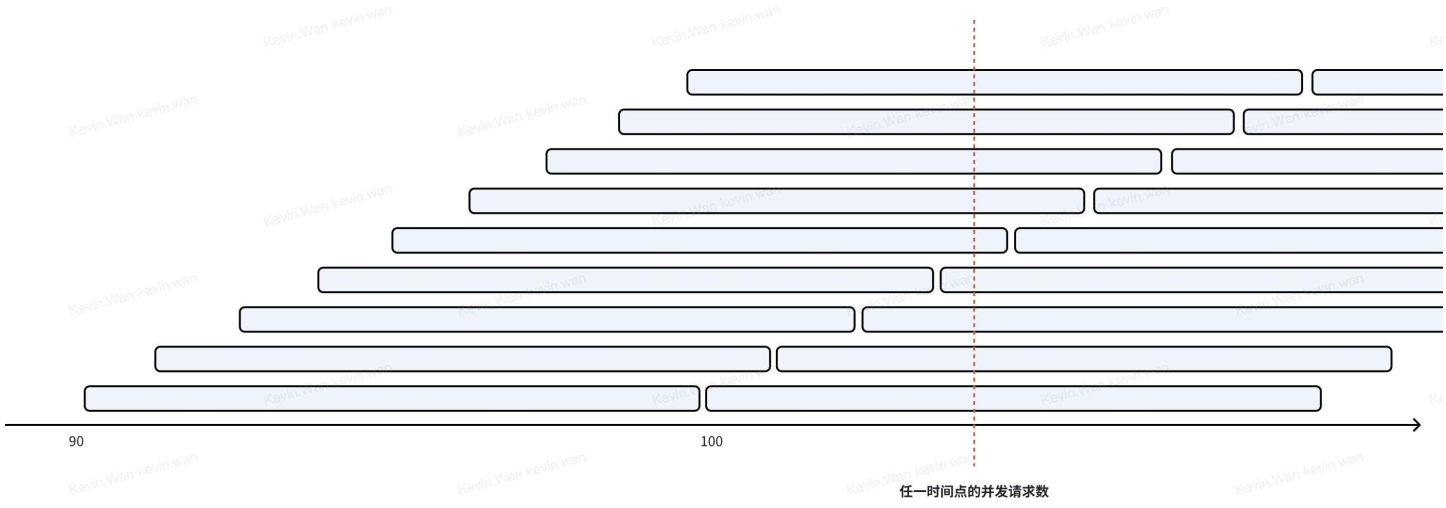
- `maxPass` 是当前记录的所有窗口里面最大的成功处理的请求数，go-zero 是 100ms 一个窗口，记录了 50 个窗口，这里就是算过去 50 个窗口里最多成功处理请求的一个窗口里的请求数
- `minRt` 是指以窗口为单位的最小平均请求耗时，单位毫秒，比如某个窗口里的 RT 是 50ms，且是所有窗口里最小的，那么这个 `minRt` 就是 50
- `windows` 是指每秒有多少个窗口

我来一步一步推导一下这个公式怎么来的。

- `maxPass * windows` 就是每秒系统能成功处理的请求数
- `minRt / 1000` 是个缩放系数，用来把每秒能处理的请求数缩放到 `minRt` 时间长度上系统能处理的请求数
- 一个请求的处理时长保守估算为 `minRt`，所以 `maxPass * windows * minRt / 1000` 代表着系统能处理的保守并发数。这里相对比较难理解，可以想想一个请求的时间跨度为 `minRt`，那么把每秒的请求数 `maxPass * windows` 平铺到 1s 的时间线上，是不是任意点的并发请求数可以估算为 `QPS * minRt / 1000`。从概率的角度可以理解为均匀分布的区间积分，即 1 秒内均匀分布的 n 个请求，在 minRt 区间上的请求数量。

如果当前并发请求数大于这里算出的系统容量，那么就会拒绝请求，所以这里估算系统容量是关键所在，也是整个算法最难理解的部分。

附一个图来说明怎么计算任一时间点的并发请求数的，假设 QPS 是 1000，每个请求的 `minRt` 是 10ms。



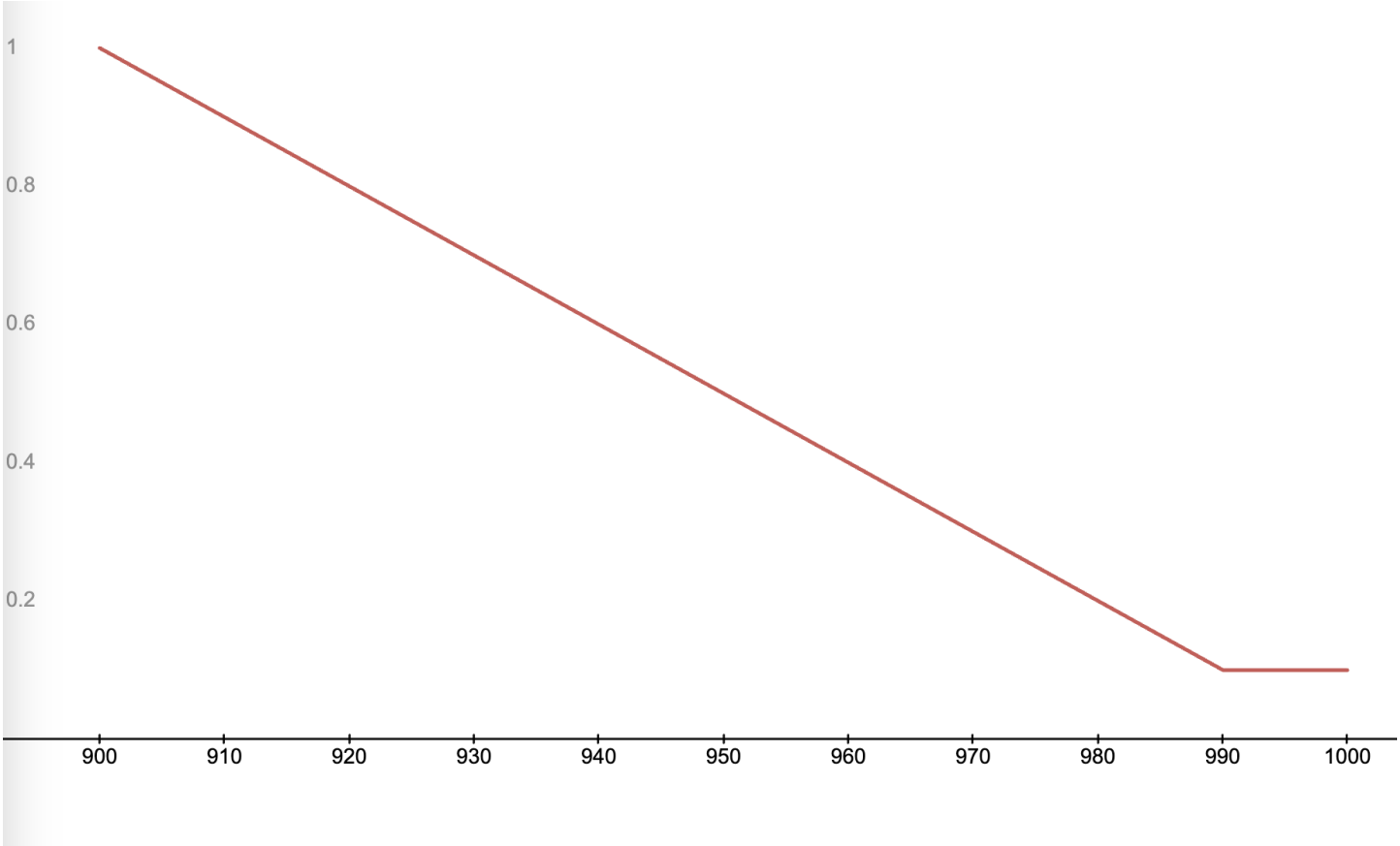
3.3 CPU 负载反馈因子

当 CPU 负载超过了设置的阈值时，我们期望 CPU 越高，对请求的拒绝比例越高，否则 CPU 依然有可能越来越靠近 100%。

反馈因子计算公式如下：

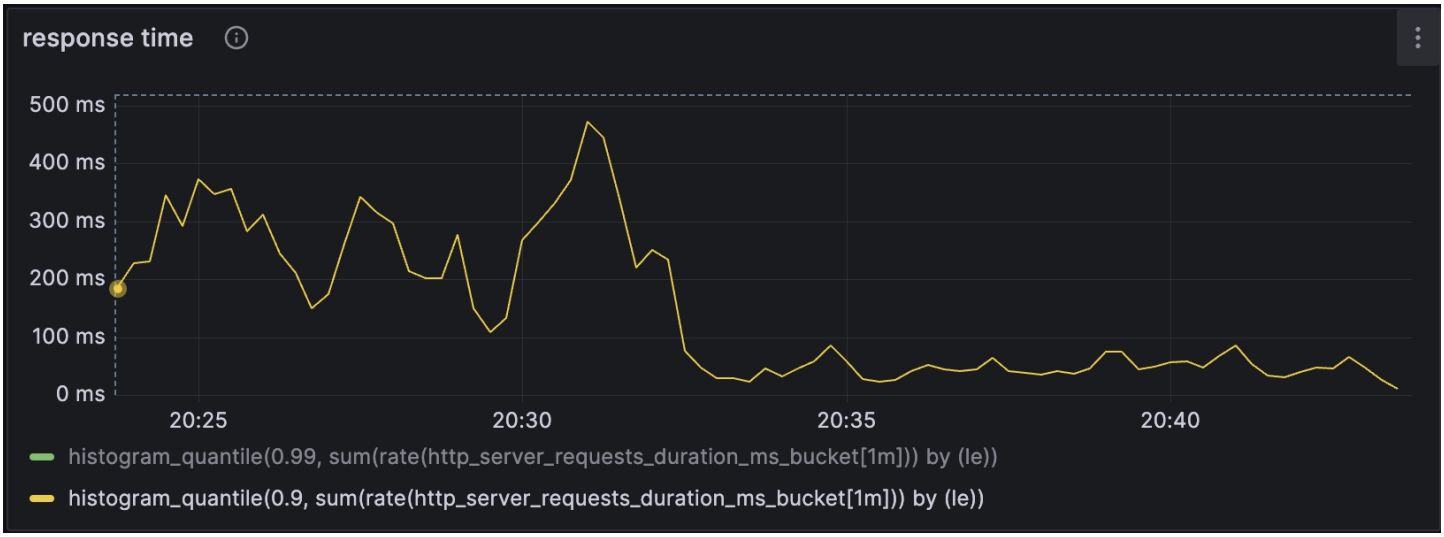
$$\max(0.1, \frac{1000 - usage}{1000 - threshold})$$

反馈因子的效果类似于神经网络中的 ReLU 激活函数。其中 0.1（兜底的经验值）是用来保证不管负载多高，至少放过估算出来的系统容量的 10% 的请求，否则整个服务就完全不可用了。CPU 负载反馈因子随着 CPU 负载的变化如下图：



对比有无 CPU 反馈因子的情况：

- 加了反馈因子后能接受更多请求，从不到 3000qps 上升到了 5000qps 左右，上图是拒绝的请求
- 成功处理的请求数有略微下降，因为进来的请求变多了，拒绝请求消耗了少量资源
- 加了反馈因子 P99 时延从 900ms 左右，降低到 700ms 左右



- 加了反馈因子 P90 从 250ms 降到了 50ms

由此可见，CPU 负载反馈因子的作用还是很明显的。

3.4 过载保护计算公式

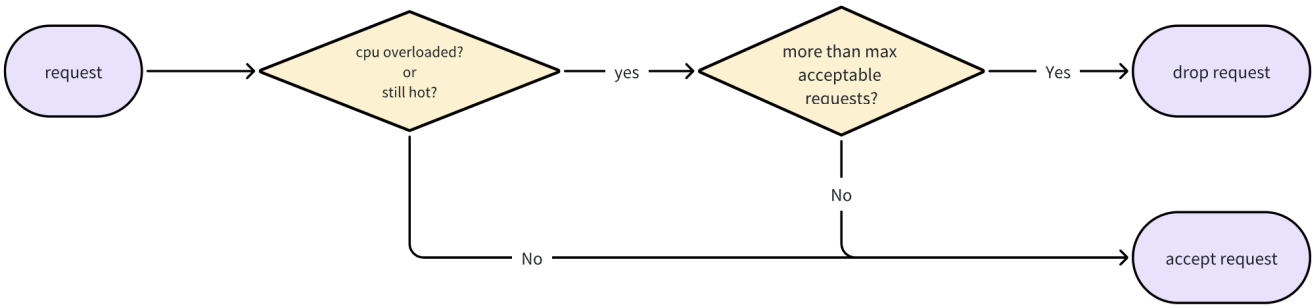
把上面所有计算逻辑归总到计算公式里，如下图：

$$\frac{maxPass \times windows \times minRt}{1000} \times max(0.1, \frac{1000 - usage}{1000 - threshold})$$

这就是计算当前系统允许的最大并发请求数，超过这个值就会拒绝请求。

3.5 如何判断是否拒绝请求

对于一个请求是否需要拒绝。判断逻辑如下：

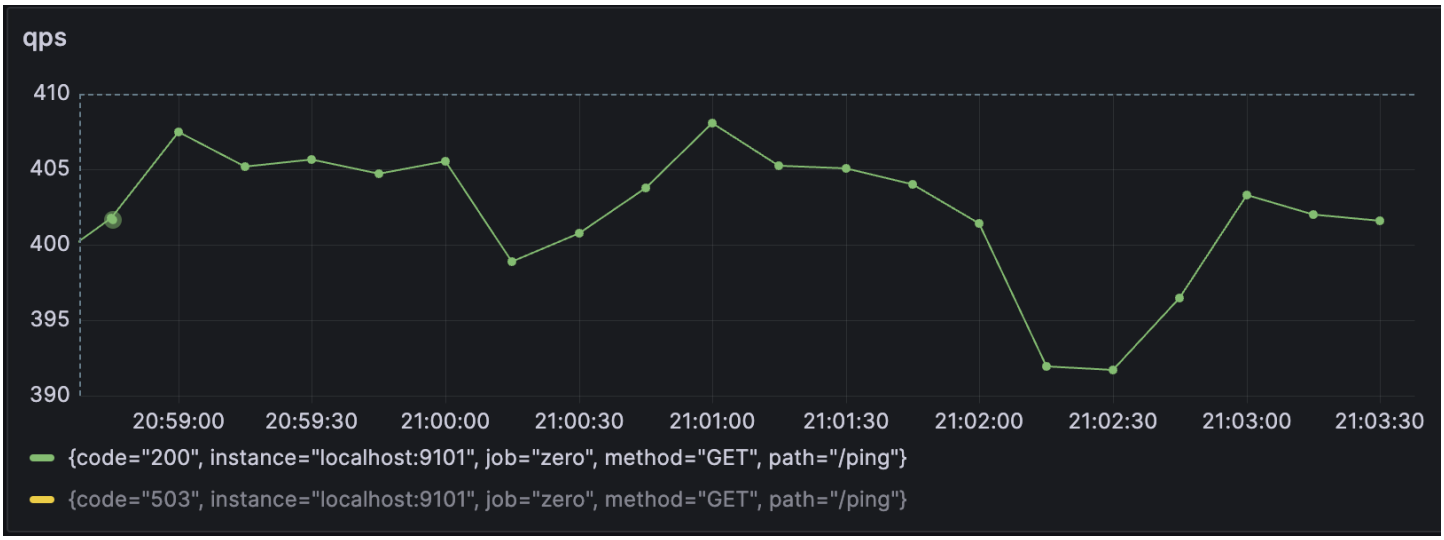


有以下情况之一，计算是否超过系统容量，如超过，则拒绝该请求

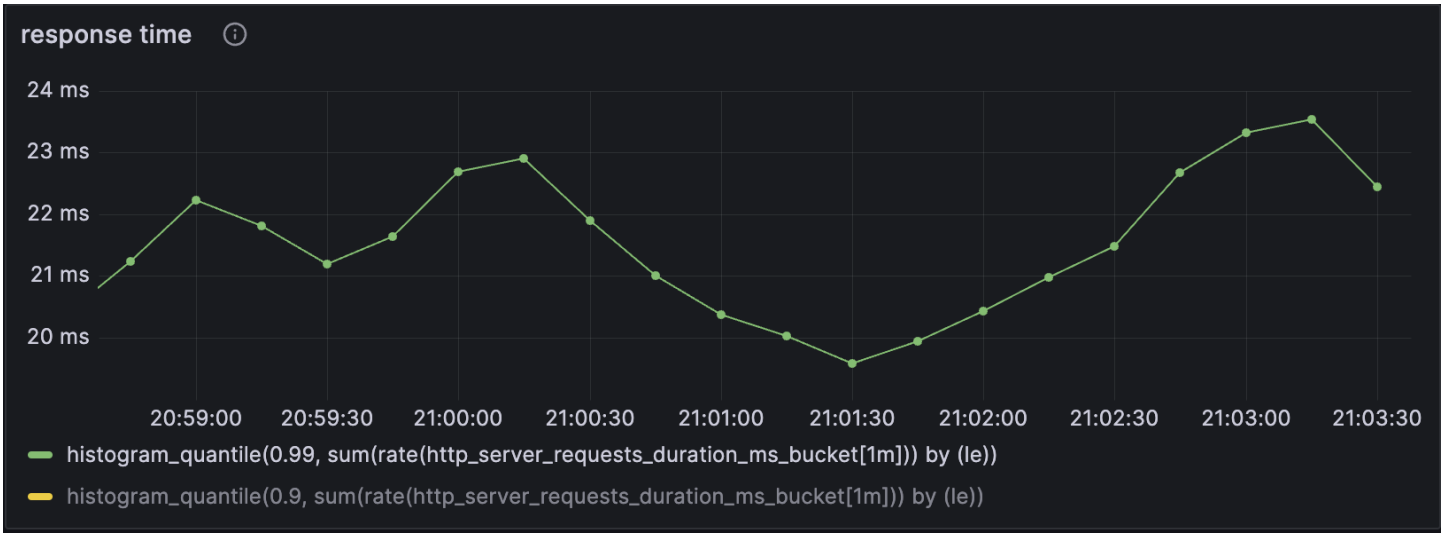
- 首先判断 CPU 是否超标（默认 90%）
- 再判断是否在冷却期（1s）
- 两者有其一，则计算是否拒绝
  - 计算当前系统容量 `maxPass * windows * minRt / 1000`
  - 并发请求数是否大于当前系统容量，如是则拒绝请求

至此，原理基本讲完了，还有一些实现细节和技巧你可以翻看 go-zero 源码。

我们再来看一下系统容量 10 倍流量的场景下整体表现：



- 成功处理的 qps 在 400 左右
- 拒绝了大概 90% 的请求



- P99 时延控制在 20-24ms 之间
- P90 时延在 5ms 以下
- CPU 峰值控制在 95% 以下

如文档开始的压测数据，如果不是过载保护，在不到 600 qps 的情况下，P99 甚至 P90 都已经到了 1s 的超时阈值了，服务基本已经开始不可用了。

3.6 跟 Kubernetes HPA 的协同

当我们在使用 Kubernetes 并设置了 HPA 根据 CPU 使用率自动伸缩的时候，Kubernetes 默认会在 4 个连续的 15 秒探测周期探测到 CPU 使用率超标（有 0.9 - 1.1 的容忍幅度）时，启动增加 pod 来应对系统容量不足，但这需要分钟级扩容，且当系统资源不够或者 pod 数达到最大设置时不生效。此时，过载保护可以在 Kubernetes 未来得及扩容或者集群容量不足时保护我们的系统不被打到卡死。

但这里有个点需要注意，go-zero 默认设置的过载保护触发的 CPU 阈值是 90%，Kubernetes HPA 自动扩容默认 CPU 阈值是 80%，当你在设置这两个参数的时候，不要让 HPA 的 CPU 阈值大于 go-zero 的过载保护 CPU 阈值，否则可能会抑制 HPA 的生效。

当然整个系统并不是链路上所有服务和中间件都可以自动或及时扩容的，这里就牵出另一个稳定性能力 - **自适应熔断**了。有了自适应过载保护和自适应熔断的双重加持，流量再大（上限是所有 CPU 都用在降载熔断等能力上），服务也不会挂。后续文章我会深度分析自适应熔断的场景压测和实现原理。

1. **总结**

自适应过载保护的算法有如下要点：

- **CPU 使用率检测**。要在各种不同环境尽可能保证检测结果准确。这里要防止检测周期受系统调度影响出现较大偏差，需要消除异常值，且通过滑动平均的方式来避免毛刺对算法的影响。
- **系统容量评估**。类似于 BBR 算法里检测**带宽**和 **RTT**，我们需要探测的是 **系统能承载的 QPS** 和 **minRT**（最小处理时间），不同于 BBR 的不能同时探测，我们是可以同时探测的，所以我们不需要有复杂的状态机，只需要通过滑动窗口来记录过去一段周期内的请求状况即可。
- **CPU 负载反馈因子**。类似于神经网络中的激活函数，基于 CPU 负载对系统容量做调节。

- **过载保护冷却时间**。用于避免过载保护在 CPU 临界值周边来回启停，有利于大流量下的请求抑制效果。
- **跟 HPA 的协同**。一定要避免 HPA 的 CPU 扩容配置低于过载保护设置的阈值，否则过载保护会抑制 HPA 的生效。但一般不太会发生，HPA 默认值是 80%，过载保护默认值是 90%，不建议把 HPA 的配置往上调。

go-zero 的三类服务都已经默认集成了自适应过载保护：

- **Restful service**
- **gRPC service**
- **Gateway service**

确保你在使用 go-zero 的时候，无需额外代码和配置，服务默认都是有自适应过载保护的。这也秉承了 go-zero 一贯追求最简原则，不给用户带来额外的心智负担。

这篇文章花了我春节不少时间，既要讲原理，又要写代码给出压测数据，但依然感觉不是那么容易懂，不过好在所有算法细节和知识点都已经集成在 go-zero 源码里了，如果想了解实现细节，可以阅读源码。