注意：此文档描述的内容正在建设中或处于功能早期阶段，请持续关注文档更新！

自适应负载均衡与限流

本文所说的柔性服务主要是指**consumer端的负载均衡**和**provider端的限流**两个功能。在之前的dubbo版本中，负载均衡部分更多的考虑的是公平性原则，即consumer端尽可能平等的从provider中作出选择，在某些情况下表现并不够理想。而限流部分只提供了静态的限流方案，需要用户对provider端设置静态的最大并发值，然而该值的合理选取对用户来讲并不容易。我们针对这些存在的问题进行了改进。

整体介绍

本文所说的柔性服务主要是指**consumer端的负载均衡**和**provider端的限流**两个功能。在之前的dubbo版本中，

- 负载均衡部分更多的考虑的是公平性原则，即consumer端尽可能平等的从provider中作出选择，在某些情况下表现并不够理想。
- 限流部分只提供了静态的限流方案，需要用户对provider端设置静态的最大并发值，然而该值的合理选取对用户来讲并不容易。

我们针对这些存在的问题进行了改进。

负载均衡

使用介绍

他的几种方案主要是考虑选择时的公平性和稳定性。

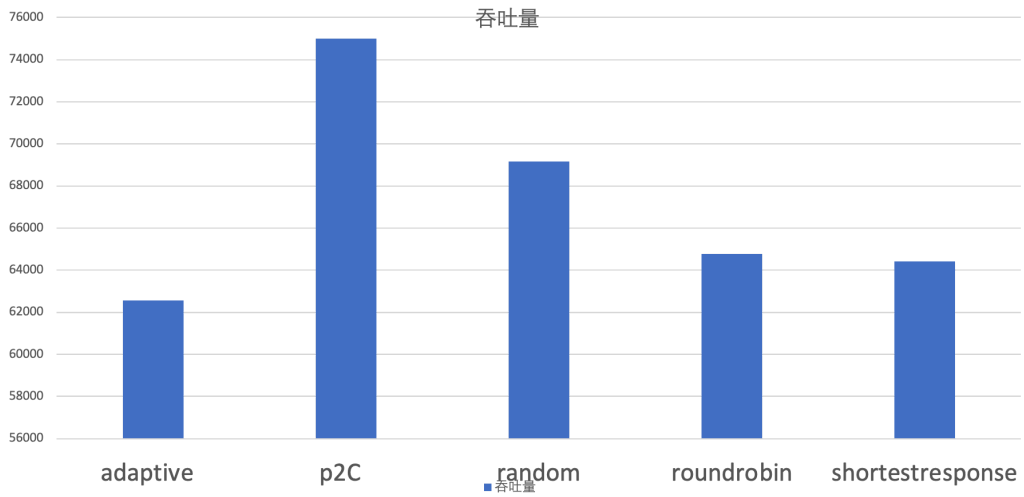
对于 `ShortestResponse` 来说，其设计目的是从所有备选的 provider 中选择 response 时间最短的以提高系统整体的吞吐量。然而存在两个问题：

1. 在大多数的场景下，不同provider的response时长没有非常明显的区别，此时该算法会退化为随机选择。
2. response的时间长短有时也并不能代表机器的吞吐能力。对于 `LeastActive` 来说，其认为应该将流量尽可能分配到当前并发处理任务较少的机器上。但是其同样存在和 `ShortestResponse` 类似的问题，即这并不能单独代表机器的吞吐能力。

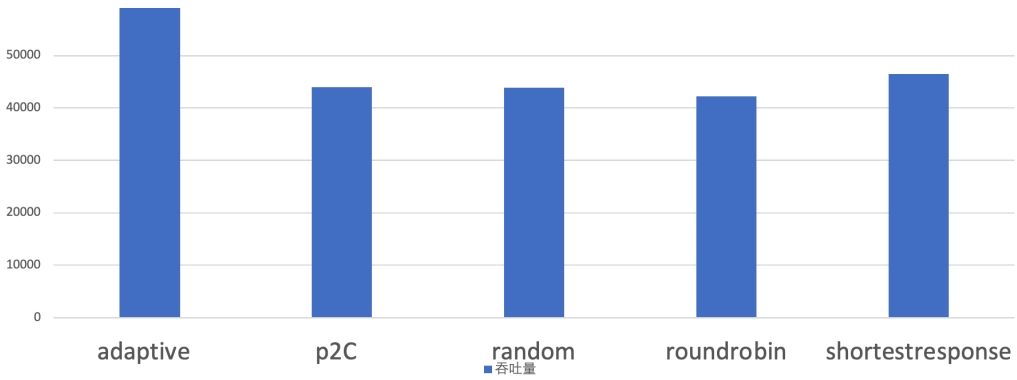
基于以上分析，我们提出了两种新的负载均衡算法。一种是同样基于公平性考虑的单纯 `P2C` 算法，另一种是基于自适应的方法 `adaptive` ，其试图自适应的衡量 provider 端机器的吞吐能力，然后将流量尽可能分配到吞吐能力高的机器上，以提高系统整体的性能。

总体效果

对于负载均衡部分的有效性实验在两个不同的情况下进行的，分别是提供端机器配置比较均衡和提供端机器配置差距较大的情况。



EN



使用方法

Dubbo Java 实现的使用方法 与原本的负载均衡方法相同。只需要在consumer端将"loadbalance"设置为"p2c"或者"adaptive"即可。

代码结构

负载均衡部分的算法实现只需要在原本负载均衡框架内继承LoadBalance接口即可。

原理介绍

P2C算法

Power of Two Choice 算法简单但是经典，主要思路如下：

1. 对于每次调用，从可用的provider列表中做两次随机选择，选出两个节点providerA和providerB。
2. 比较providerA和providerB两个节点，选择其“当前正在处理的连接数”较小的那个节点。

adaptive算法

代码的github地址

相关指标

1. cpuLoad $cpuLoad = cpu$一分钟平均负载 $* 100 /$可用cpu数量。该指标在provider端机器获得，并通过invocation的attachment传递给consumer端。
2. rt rt为一次rpc调用所用的时间，单位为毫秒。

5. currentProviderTime provider端在计算cpuLoad时的时间，单位是毫秒

6. currentTime currentTime为最后一次计算load时的时间，初始化为currentProviderTime，单位是毫秒。

7. multiple
$$multiple = (\text{当前时间} - currentTime)/timeout + 1$$

8. lastLatency
$$lastLatency = \begin{cases} 2 * timeout, & currentTime == currentProviderTime \\ lastLatency >> multiple, & otherwise \end{cases}$$

9. beta 平滑参数，默认为0.5

10. ewma lastLatency的平滑值
$$lastLatency = beta * lastLatency + (1 - beta) * lastLatency$$

11. inflight inflight为consumer端还未返回的请求的数量。
$$inflight = consumerReq - consumerSuccess - errorReq$$

12. load 对于备选后端机器x来说，若距离上次被调用的时间大于2*timeout，则其load值为0。 否则,

$$load = CpuLoad * (sqrt(ewma) + 1) * (inflight + 1)/(((consumerSuccess/(consumerReq + 1)) * weight) + 1)$$

算法实现

依然是基于P2C算法。

1. 从备选列表中做两次随机选择，得到providerA和providerB
2. 比较providerA和providerB的load值，选择较小的那个。

自适应限流

与负载均衡运行在consumer端不同的是，限流功能运行在provider端。其作用是限制provider端处理并发任务时的最大数量。从理论上讲，服务端机器的处理能力是存在上限的，对于一台服务端机器，当短时间内出现大量的请求调用时，会导致处理不及时的请求积压，使机器过载。在这种情况下可能导致两个问题：

1. 由于请求积压，最终所有的请求都必须等待较长时间才能被处理，从而使整个服务瘫痪。
2. 服务端机器长时间的过载可能有宕机的风险。

下，该值难以通过计算静态设置。

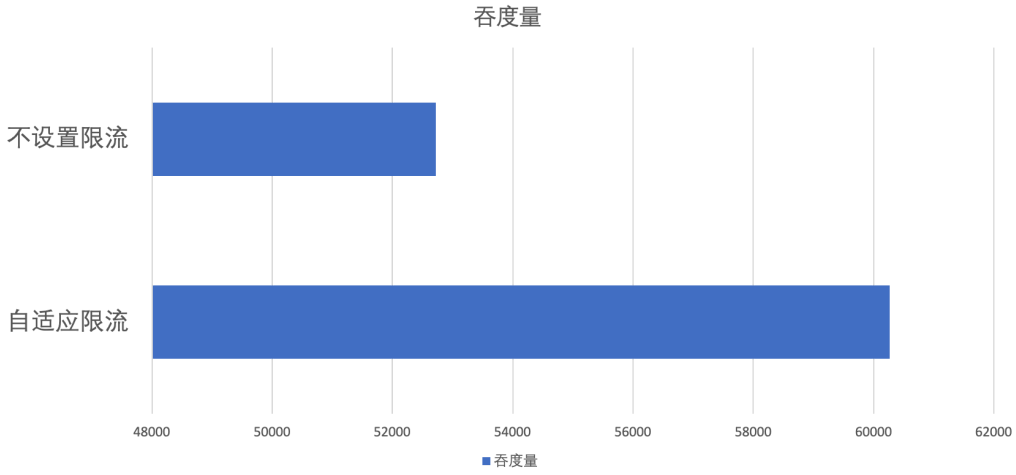
基于以上原因，我们需要一种自适应的算法，其可以动态调整服务端机器的最大并发值，使其可以在保证机器不过载的前提下，尽可能多的处理接收到的请求。因此，我们参考相关理论与算法实践基础上，在 Dubbo 框架内实现了两种自适应限流算法，分别是基于启发式平滑的 `HeuristicSmoothingFlowControl` 和基于窗口的 `AutoConcurrencyLimier` 。

代码的github地址

使用介绍

总体效果

自适应限流部分的有效性实验我们在提供端机器配置尽可能大的情况下进行，并且为了凸显效果，在实验中我们将单次请求的复杂度提高，将超时时间尽可能设置的大，并且开启消费端的重试功能。



使用方法

要确保服务端存在多个节点，并且消费端开启重试策略的前提下，限流功能才能更好的发挥作用。

Dubbo Java 实现的自适应限流开启方法 与静态的最大并发值设置类似，只需在provider端将"flowcontrol"设置为"autoConcurrencyLimier"或者"heuristicSmoothingFlowControl"即可。

代码结构

1. FlowControlFilter：在provider端的filter负责根据限流算法的结果来对provider端进行限流功能。

3. CpuUsage：周期性获取cpu的相关指标
4. HardwareMetricsCollector：获取硬件指标的相关方法
5. ServerMetricsCollector：基于滑动窗口的获取限流需要的指标的相关方法。比如qps等。
6. AutoConcurrencyLimier：自适应限流的具体实现算法。
7. HeuristicSmoothingFlowControl：自适应限流的具体实现方法。

原理介绍

HeuristicSmoothingFlowControl

相关指标

1. alpha alpha为可接受的延时的上升幅度，默认为0.3
2. minLatency 在一个时间窗口内的最小的Latency值。
3. noLoadLatency noLoadLatency是单纯处理任务的延时，不包括排队时间。这是服务端机器的固有属性，但是并不是一成不变的。在HeuristicSmoothingFlowControl算法中，我们根据机器CPU的使用率来确定机器当前的noLoadLatency。当机器的CPU使用率较低时，我们认为minLatency便是noLoadLatency。当CPU使用率适中时，我们平滑的用minLatency来更新noLoadLatency的值。当CPU使用率较高时，noLoadLatency的值不再改变。
4. maxQPS 一个时间窗口周期内的QPS的最大值。
5. avgLatency 一个时间窗口周期内的Latency的平均值，单位为毫秒。
6. maxConcurrency 计算得到的当前服务提供端的最大并发值。
   $$maxConcurrency = ceil(maxQPS * ((2 + alpha) * noLoadLatency - avgLatency))$$

算法实现

当服务端收到一个请求时，首先判断CPU的使用率是否超过50%。如果没有超过50%，则接受这个请求进行处理。如果超过50%，说明当前的负载较高，便从HeuristicSmoothingFlowControl算法中获得当前的maxConcurrency值。如果当前正在处理的请求数量超过了maxConcurrency，则拒绝该请求。

AutoConcurrencyLimier

2. MinExploreRatio 默认设置为0.06
3. SampleWindowSizeMs 采样窗口的时长。默认为1000毫秒。
4. MinSampleCount 采样窗口的最小请求数量。默认为40。
5. MaxSampleCount 采样窗口的最大请求数量。默认为500。
6. emaFactor 平滑处理参数。默认为0.1。
7. exploreRatio 探索率。初始设置为MaxExploreRatio。 若 avgLatency<=noLoadLatency*(1.0 + MinExploreRatio)或者 qps>=maxQPS*(1.0 + MinExploreRatio) 则 exploreRatio=min(MaxExploreRatio,exploreRatio+0.02) 否则 exploreRatio=max(MinExploreRatio,exploreRatio-0.02)
8. maxQPS 窗口周期内QPS的最大值。
$$maxQPS = \begin{cases} qps, & qps > maxQPS \\ qps * emaFactor + maxQPS * (1 - emaFactor), & otherwise \end{cases}$$
9. noLoadLatency
$$noLoadLatency = \begin{cases} avgLatency, & noLoadLatency <= 0 \\ avgLatency * emaFactor + noLoadLatency * (1 - emaFactor), & otherwise \end{cases}$$
10. halfSampleIntervalMs 半采样区间。默认为25000毫秒。
11. resetLatencyUs 下一次重置所有值的时间戳，这里的重置包括窗口内值和noLoadLatency。单位是微秒。初始为0.
$$resetLatencyUs = samplingTimeUs + 2 * avgLatency, if(remeasureStartUs <= samplingTimeUs)$$
12. remeasureStartUs 下一次重置窗口的开始时间。
$$remeasureStartUs = samplingTimeUs + (halfSampleIntervalMS + \text{随机值}) * 1000$$
13. startSampleTimeUs 开始采样的时间。单位为微秒。
14. sampleCount 当前采样窗口内请求的数量。
15. totalSampleUs 采样窗口内所有请求的latency的和。单位为微秒。
16. totalReqCount 采样窗口时间内所有请求的数量和。注意区别 sampleCount。
17. samplingTimeUs 采样当前请求的时间戳。单位为微秒。
18. latency 当前请求的latency。
19. qps 在该时间窗口内的qps值。
$$qps = totalReqCount * 1000000/(samplingTimeUs - startSampleTimeUs)$$

21. maxConcurrency 上一个窗口计算得到当前周期的最大并发值。
22. nextMaxConcurrency 当前窗口计算出的下一个周期的最大并发值。

$$nextMaxConcurrency = \begin{cases} ceil(maxQPS * noLoadLatency * 0.9/1000000), remeasureStartUs <= samplingTimeUs \\ ceil(noLoadLatency * maxQPS * (1 + exploreRatio)/1000000), otherwise \end{cases}$$

Little's Law

- 当服务处于稳定状态时：concurrency=latency*qps。这是自适应限流理论的基础。
- 当请求没有导致机器超载时，latency基本稳定，qps和concurrency处于线性关系。
- 当短时间内请求数量过多，导致服务超载的时候，concurrency会和latency一起上升，qps则会趋于稳定。

算法实现

AutoConcurrencyLimier的算法使用过程和HeuristicSmoothingFlowControl类似。与HeuristicSmoothingFlowControl的最大区别是:

AutoConcurrencyLimier是基于窗口的。每当窗口内积累了一定量的采样数据时，才利用窗口内的数据来更新得到maxConcurrency。 其次，利用exploreRatio来对剩余的容量进行探索。

另外，每隔一段时间都会自动缩小max_concurrency并持续一段时间，以处理noLoadLatency上涨的情况。因为估计noLoadLatency时必须先让服务处于低负载的状态，因此对maxConcurrency的缩小是难以避免的。

由于 max_concurrency < concurrency 时，服务会拒绝掉所有的请求，限流算法将 “排空所有的经历过排队的等待请求的时间” 设置为 2*latency，以确保 minLatency 的样本绝大部分时没有经过排队等待的。

反馈

此页是否对您有帮助?

是 否

关注我们

请通过以下任一或多个渠道关注社区动态，与社区开发者保持密切沟通。

微信 **钉钉** GITHUB

文档 资源

概览 社区

快速开始

开发者指南