**openresty性能优化小结**

[2016年10月12日](http://xbsura.com/2016/10/12/openresty%e6%80%a7%e8%83%bd%e4%bc%98%e5%8c%96%e5%b0%8f%e7%bb%93/)  [xbsura](http://xbsura.com/author/xbsura/)  [没有意思的技术](http://xbsura.com/category/code/) 阅读: 785



点击播放或暂停

00:00/00:00

1无论如何- 游学志

歌曲是<<千与千寻>>的插曲, 很好听

**前言**

性能优化的主体是一个 http 代理服务, 关于http 代理服务的开发选型, 我们有几种调研方案:

测试的机器性能: 单核nginx 处理 echo, 1w2 qps, ★★★★★为满分

1. nginx c 扩展(upstream 模块): 性能很好, 转发请求有1w2/2=6k qps; 由于需要识别服务, 做策略(记状态), 略复杂的转发重试逻辑, 开发不是很方便  
   性能: ★★★★★  
   效率: ★★
2. golang: 语言本身性能没问题, http 模块性能较差, 转发模式下只有3k qps, 是 c 扩展的一半; 丰富的扩展和简单的语法使得开发非常容易  
   性能: ★★  
   效率: ★★★★★
3. openresty: 即 nginx+lua 方式, 在简单的转发测试中性能很好, 与 c 模块无差别, 在增加业务逻辑后有一定性能下降; 扩展很多, 配合 nginx, 开发比较方便  
   性能: ★★★★  
   效率: ★★★★

经过对比, 最终选择使用 **openresty** 开发, 服务具体的功能有:

1. 全局唯一的动态拉取的配置
2. 确定服务唯一名称
3. 根据配置的策略确定后端, 并代理请求
4. 后端失效屏蔽, 多次重试等逻辑

在增加所有业务逻辑后, 实际测试的性能由6k降低到3k, 损失一半

为了能够进一步降低资源占用效率, 试图对当前的逻辑进行优化, 效果是qps 由3k 增加到4k

**分析工具**

分析openresty的性能问题可以使用一个叫做火焰图的工具, 因为要安装内核 debug 包, 不推荐内核较低的开发机上折腾, 比较方便的过程如下:

1. 安装虚拟机, mac 有 virtualbox/pd(四百多快一年)
2. 安装一个开源 linux 操作系统最新版, 推荐 fedora
3. 安装stap(如果没有附带), 执行stap-pre(自动安装所需要的包)
4. 下载 <https://github.com/openresty/nginx-systemtap-toolkit>, 如果使用了 lua 代码, 将所需(所有)文件拷贝到nginx 目录, 与 conf 同级(或将所需 lua 目录拷贝到工具所在目录)
5. 启动 nginx 服务, 起压力, 查看 worker 的 pid
6. cd 到4步骤所在的目录, 依次执行
7. ./ngx-sample-lua-bt -p pid --luajit20 -t 10 > tmp.bt
8. ./fix-lua-bt tmp.bt > flame.bt
9. perl ./stackcollapse-stap.pl flame.bt > flame.cbt
10. perl ./flamegraph.pl flame.cbt > flame.svg

最后生成的 svg 用浏览器打开即可, 打开后, 火焰图呈现的状态为: 横向为某过程占用的 cpu比例, 纵向为调用深度

如果出现横向很长的单个调用, 可能存在 cpu 热点

**优化点**

**减少全局变量**

在同一个请求中, 不同阶段之间, 不同调用之前可以通过 ngx.ctx 与 ngx.var 传递全局变量, 这两个调用非常消耗性能, 在 echo 测试中, 一条 ctx 赋值就能将整体 qps 下降接近1000(12000 -> 11000), 大概相当于3000个local变量赋值操作

原因可能与全局变量需要加锁, 元表方法调用等相关, **不太明白根本原因**

总结一下, var 与 ctx 变量尽量少用, 在大部分场景下, 重新计算消耗的 cpu 都要比这样存起来更少

**减少每个阶段的钩子**

具体来说, 每个\*\_by\_lua都会造成一定的性能下降, 数量级在百左右, 转发模块本来有四个\*\_by\_lua的处理, 优化后, 变成两个

有两个阶段, header\_filter\_by\_lua 与 body\_filter\_by\_lua只有在出错时(请求没有发送给后端, 直接返回)才会起作用, 不应该在每个请求中都使用

总结一下, 尽量减少\*\_by\_lua的阶段

**减少函数调用**

语法比函数调用更省 cpu, 比如连接两个字符串, string.format("%s%s", x, y) 就要比 x .. y慢, 在大量使用字符串连接之后, 这种慢会影响到 qps

所以, 尽可能使用语法而不是函数调用, 对性能有正优化

**合理使用jit**

编译 openresty 时, 使用--with-luajit打开 jit 功能在大多数情况下有利于性能提升

一般情况下, lua 代码总是被 lua 解释器解释执行的, jit 是一个优化执行的项目, 包括解释执行和编译执行

即使是 luajit, 所有的代码一开始也是被解释执行的, 只有一部分能被编译成机器码的方法执行频率超过某个阈值时才会被编译成机器码以提高执行效率, 所以, 能通过 jit 加速的条件有两个, 一个是一部分方法, 另一个是执行频率

这要求我们在热代码中尽可能不使用不能被编译成机器码的方法(简称为 NYI), 这些方法在 <http://wiki.luajit.org/NYI>(<https://moonbingbing.gitbooks.io/openresty-best-practices/content/lua/what_jit.html>)中有记录, 如果开启了jit 但是大量使用NYI 语法, 反而会降低代码效率, 对于耗时较多的NYI 方法, 可以考虑适当使用缓存进行保存

**使用openresty 提供的方法**

具体使用上, 尽可能少使用非 openresty 提供的方法, 除了以上说的 jit 的原因外, 还因为阻塞问题

比如 sleep, 使用 os.execute("sleep 1")与 ngx.sleep(1), 前者是阻塞的, 后者可以让出 cpu, 对性能的影响很大

openresty 提供的所有方法都是非阻塞的, 在 io等待时都可以让出 cpu(比如 resty-开头的数据库驱动, ngx.方法等等), 推荐使用, 而lua 自己提供的方法一般不是(如 os 库, time 库等等), 不推荐使用

**使用开源库**

在”转发请求”这种场景下, 使用 openresty 的 balancer 库性能要比使用自己编写的 http 库(即使使用了 ngx.tcp 非阻塞 socket)要快很多(字符串操作在 c 中完成), 而且代码简洁

**小结**

openresty 非常适合做 http 代理服务器的开发, 与 go/php 相比有自己鲜明的特色(性能/效率), 配合合理的开发习惯, 可以在满足需求的情况下, 较大提高开发效率