Nginx多进程高并发、低时延、高可靠机制缓存代理中的应用

# 开发背景

     现有开源缓存代理中间件有twemproxy、codis等，其中twemproxy为单进程单线程模型，只支持memcache单机版和redis单机版，都不支持集群版功能。

     由于twemproxy无法利用多核特性，因此性能低下，短连接QPS大约为3W，长连接QPS大约为13W，同时某些场景时延抖动厉害。

      为了适应公有云平台上业务方的高并发需求，因此决定借助于twemproxy来做二次开发，把nginx的高性能、高可靠、高并发机制引入到twemproxy中，通过master+多worker进程来实现七层转发功能。

# Twemproxy

## Twemproxy简介

Twemproxy 是一个快速的单线程代理程序，支持 [Memcached](http://www.oschina.net/p/memcached) ASCII协议和更新的[Redis](http://www.oschina.net/p/redis)协议。它全部用C写成，使用Apache 2.0 License授权。支持以下特性：

i)速度快

ii)轻量级

iii)维护持久的服务器连接

iiii)启用请求和响应的管道

iiiii)支持代理到多个后端缓存服务器

iiiii)同时支持多个服务器池

iiiiii)多个服务器自动分享数据

iiiiiii)可同时连接后端多个缓存集群

iiiiiiii)实现了完整的 [memcached ascii](https://github.com/twitter/twemproxy/blob/master/notes/memcache.txt) 和 [redis](https://github.com/twitter/twemproxy/blob/master/notes/redis.md) 协议.

iiiiiiiii)服务器池配置简单，通过一个 YAML 文件即可

iiiiiiiiii)一致性hash

iiiiiiiiii)详细的监控统计信息

iiiiiiiiiii)支持 Linux, \*BSD, OS X and Solaris (SmartOS)

iiiiiiiiiiii)支持设置HashTag

iiiiiiiiiiiiiii)连接复用，内存复用，提高效率

## 滴云memcache缓存集群拓扑结构

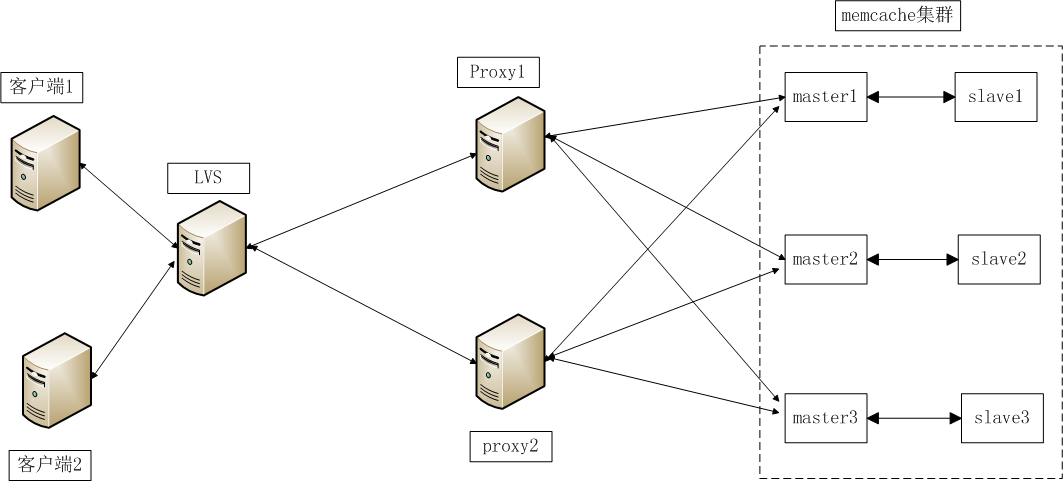


图1 twemproxy缓存集群拓扑图

如上图所示，实际应用中业务程序通过轮询不同的twemproxy来提高qps，同时实现负载均衡。

## 推特原生twemproxy瓶颈

如今twemproxy凭借其高性能的优势, 在很多互联网公司得到了广泛的应用，已经占据了其不可动摇的地位, 然而在实际的生产环境中, 存在以下缺陷，如下：

i)单进程单线程, 无法充分发挥服务器多核cpu的性能

ii)当twemproxy qps短连接达到8000后，消耗cpu超过70%，时延陡增。

iii)大流量下造成IO阻塞，无法处理更多请求，qps上不去，业务时延飙升

iiii)维护成本高，如果想要充分发挥服务器的所有资源包括cpu、 网络io等，就必须建立多个twemproxy实例，维护成本高

iiiii)扩容、升级不便

原生twemproxy进程呈现了下图现象：一个人干活，多个人围观。多核服务器只有一个cpu在工作，资源没有得到充分利用。



# Nginx

nginx是俄罗斯软件工程师Igor Sysoev开发的免费开源web服务器软件，聚焦于高性能，高并发和低内存消耗问题，因此成为业界公认的高性能服务器，并逐渐成为业内主流的web服务器。主要特点有：

i)完全借助epoll机制实现异步操作，避免阻塞。

ii)重复利用现有服务器的多核资源。

iii)充分利用CPU 亲和性（affinity），把每个进程与固定CPU绑定在一起，给定的 CPU 上尽量长时间地运行而不被迁移到其他处理器的倾向性，减少进程调度开销。

iiii)请求响应快

iiiii)支持模块化开发，扩展性好

iiiii)Master+多worker进程方式，确保worker进程可靠工作。当worker进程出错时，可以快速拉起新的worker子进程来提供服务。

iiiiii)内存池、连接池等细节设计保障低内存消耗。

iiiiii)热部署支持，master与worker进程分离设计模式，使其具有热部署功能。

iiiiiii)升级方便，升级过程不会对业务造成任何伤害。

Nginx多进程提供服务过程如下图所示：



# Nginx master+worker多进程机制在twemproxy中的应用

## 为什么选择nginx多进程机制做为参考？

Twemproxy和nginx都属于网络io密集型应用，都属于七层转发应用，时延要求较高，应用场景基本相同。

Nginx充分利用了多核cpu资源，性能好，时延低。

## Master-worker多进程机制原理

Master-worker进程机制采用一个master进程来管理多个worker进程。每一个worker进程都是繁忙的，它们在真正地提供服务，master进程则很“清闲”，只负责监控管理worker进程, 包含：接收来自外界的信号，向各worker进程发送信号，监控worker进程的运行状态，当worker进程退出后(异常情况下)，会自动重新启动新的worker进程。

worker进程负责处理客户端的网络请求，多个worker进程同时处理来自客户端的不同请求，worker进程数可配置。

## 多进程关键性能问题点

master-worker多进程模式需要解决的问题主要有：

i)linux内核低版本(2.6以下版本), “惊群”问题

ii) linux内核低版本(2.6以下版本),负载均衡问题

iii)linux内核高版本(3.9以上版本)新特性如何利用

iii)如何确保进程见高可靠通信

iiii)如何减少worker进程在不同cpu切换的开销

iiiii)master进程如何汇总各个工作进程的监控数据

iiiiii)worker进程异常，如何快速恢复

### linux内核低版本关键技术问题

由于linux低内核版本缺陷，因此存在”惊群”、负载不均问题，解决办法完全依赖应用层代码保障。

### 3.3.1.1如何解决“惊群”问题

当客户端发起连接后，由于所有的worker子进程都监听着同一个端口，内核协议栈在检测到客户端连接后，会激活所有休眠的worker子进程，最终只会有一个子进程成功建立新连接，其他子进程都会accept失败。

Accept失败的子进程是不应该被内核唤醒的，因为它们被唤醒的操作是多余的，占用本不应该被占用的系统资源，引起不必要的进程上下文切换，增加了系统开销，同时也影响了客户端连接的时延。

“惊群”问题是多个子进程同时监听同一个端口引起的，因此解决的方法是同一时刻只让一个子进程监听服务器端口，这样新连接事件只会唤醒唯一正在监听端口的子进程。因此

“惊群”问题通过非阻塞的accept锁来实现进程互斥accept()，其原理是：在worker进程主循环中非阻塞trylock获取accept锁，如果trylock成功，则此进程把监听端口对应的fd通过epoll\_ctl()加入到本进程自由的epoll事件集；如果trylock失败，则把监听fd从本进程对应的epoll事件集中清除。

Nginx实现了两套互斥锁：基于原子操作和信号量实现的互斥锁、基于文件锁封装的互斥锁。考虑到锁的平台可移植性和通用性，改造twemproxy选择时，选择文件锁实现。

如果获取accept锁成功的进程占用锁时间过长，那么其他空闲进程在这段时间内无法获取到锁，从而无法接受新的连接。最终造成客户端连接相应时间变长，qps低，同时引起负载严重不均衡。为了解决该问题，选择通过post事件队列方式来提高性能，trylock获取到accept锁成功的进程，其工作流程如下：

1.trylock获取accept锁成功

2.通过epoll\_wait获取所有的事件信息，把监听到的所有accept事件信息加入accept\_post列表，把已有连接触发的读写事件信息加入read\_write\_post列表。

3.执行accept\_post列表中的所有事件

4.Unlock锁

5.执行read\_write\_post列表中的事件。

Worker进程主循环工作流程图如下：



从上图可以看出，worker进程借助epoll来实现网络异步收发，客户端连接twemproxy的时候，worker进程循环检测客户端的各种网络事件和后端memcached的网络事件，并进行相应的处理。

twemproxy各个进程整体网络i/o处理过程图如下：



### 如何解决“负载均衡“问题

在多个子进程争抢处理同一个新连接事件时，一定只有一个worker子进程最终会成功建立连接，随后，它会一直处理这个连接直到连接关闭。这样，如果有的子进程“运气”很好，它们抢着建立并处理了大部分连接，其他子进程就只能处理少量连接，这对多核cpu架构下的应用很不利。理想情况下，每个子进程应该是平等的，每个worker子进程应该大致平均的处理客户端连接请求。如果worker子进程负载不均衡，必然影响整体服务的性能。

nginx通过连接阈值机制来实现负载均衡，其原理如下：每个进程都有各自的最大连接数阈值max\_threshold和当前连接阈值数local\_threshold，和当前连接数阈值，进程每接收一个新的连接，local\_threshold增一，连接断开后，local\_threashold减一。如果local\_threshold超过max\_threshold，则不去获取accept锁，把accept机会留给其他进程，同时把local\_threshold减1，这样下次就有机会获取accept锁，接收客户端连接了。

在实际业务应用中，有的业务采用长连接和twemproxy建立连接，连接数最大可能就几百连接，如果设置max\_threshold阈值过大，多个连接如果同时压到twemproxy，则很容易引起所有连接被同一个进程获取从而造成不均衡。

为了尽量减少负载不均衡，在实际应用中，新增了epoll\_wait超时时间配置选项，把该超时时间设短，这样减少空闲进程在epoll\_wait上的等待事件，从而可以更快相应客户端连接，并有效避免负载不均衡。

### Linux内核高版本TCP REUSEPORT特性如何利用

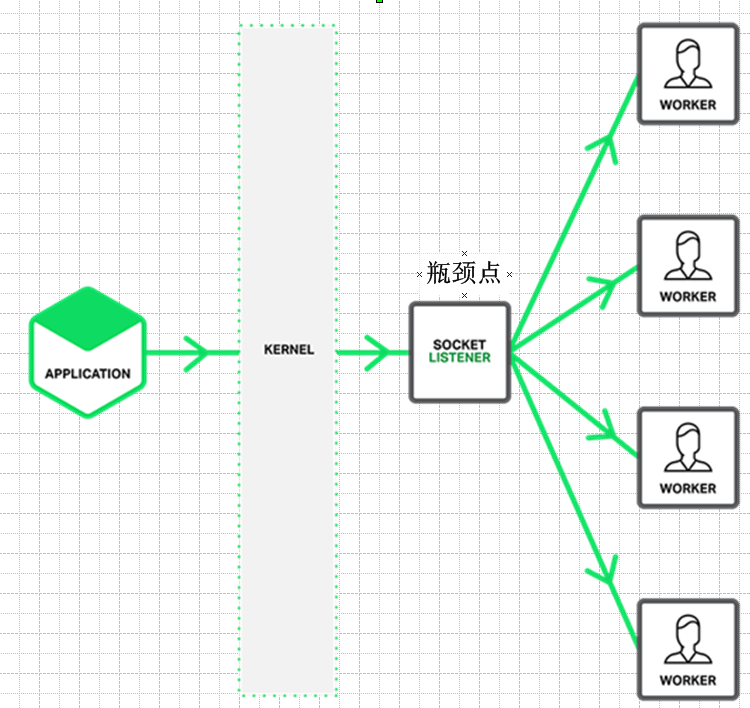
3.3.2.1什么是reuseport？

reuseport是一种套接字复用机制，它允许你将多个套接字bind在同一个IP地址/端口对上，这样一来，就可以建立多个服务来接受到同一个端口的连接。

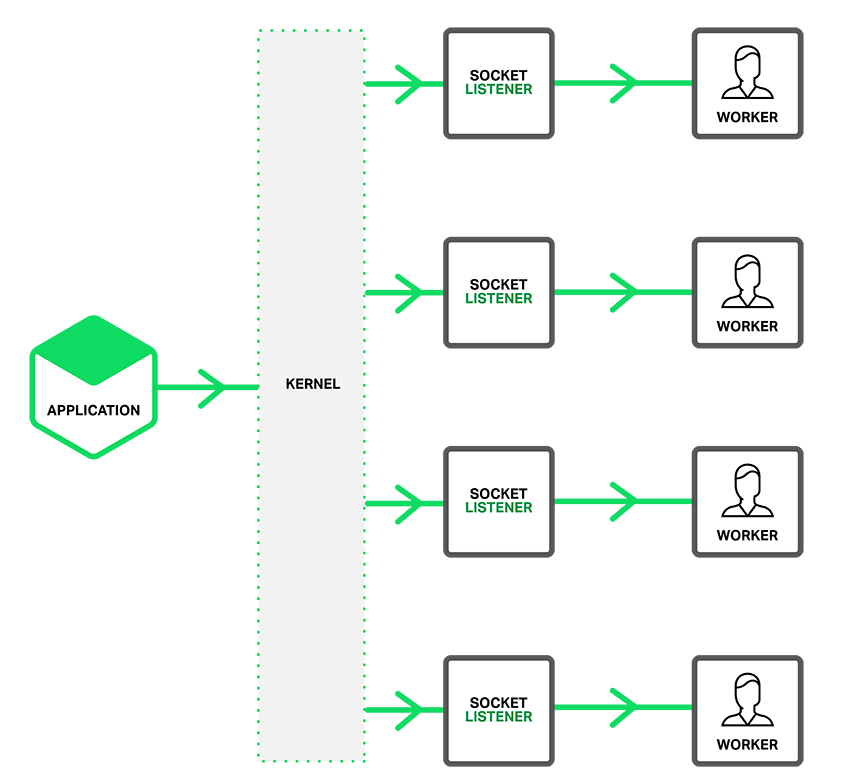
* + - 1. 支持reuseport和不支持reuseport的区别

如果linux内核版本小于3.9，则不支持reuseport(注:部分centos发行版在低版本中已经打了reuseport patch,所以部分linux低版本发行版本也支持该特性)。

不支持该特性的内核，一个ip+port组合，只能被监听bind一次。这样在多核环境下，往往只能有一个线程（或者进程）是listener，也就是同一时刻只能由一个进程或者线程做accept处理，在高并发情况下，往往这就是性能瓶颈。其网络模型如下:



在Linux kernel 3.9带来了reuseport特性，它可以解决上面(单进程listen，多工作进程accept() )的问题，其网络模型如下:



reuseport是支持多个进程或者线程绑定到同一端口，提高服务器程序的吞吐性能，其优点体现在如下几个方面:

i)允许多个套接字 bind()/listen() 同一个TCP/UDP端口

ii)每一个线程拥有自己的服务器套接字

iii)在服务器套接字上没有了锁的竞争，因为每个进程一个服务器套接字

iiii)内核层面实现负载均衡

iiiii)安全层面，监听同一个端口的套接字只能位于同一个用户下面

### Master进程和worker进程如何通信？

由于master进程需要实时获取worker进程的工作状态，并实时汇总worker进程的各种统计信息，所以选择一种可靠的进程间通信方式必不可少。

在twemproxy改造过程中，直接参考nginx的信号量机制和channel机制(依靠socketpair)来实现父子进程见通信。Master进程通过信号量机制来检测子进程是否异常，从而快速直接的反应出来；此外，借助socketpair，封装出channel接口来完成父子进程见异步通信，master进程依靠该机制来统计子进程的各种统计信息并汇总，通过获取来自master的汇总信息来判断整个twemproxy中间件的稳定性、可靠性。

配置下发过程：主进程接收实时配置信息，然后通过channel机制发送给所有的worker进程，各个worker进程收到配置信息后应答给工作进程。流程如下:



获取监控信息流程和配置下发流程基本相同，master进程收到各个工作进程的应答后，由master进程做统一汇总，然后发送给客户端。

### 如何减少worker进程在不同cpu切换的开销

CPU 亲和性（affinity） 就是进程要在某个给定的 CPU 上尽量长时间地运行而不被迁移到其他处理器的倾向性。

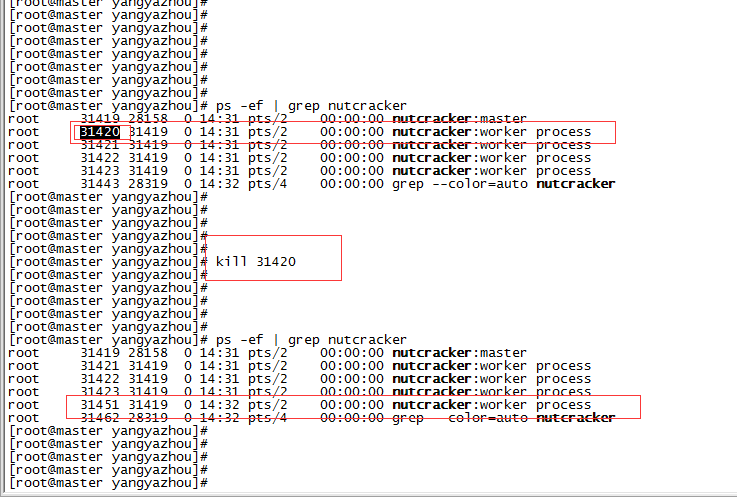
Linux 内核进程调度器天生就具有被称为 软 CPU 亲和性（affinity） 的特性，这意味着进程通常不会在处理器之间频繁迁移。这种状态正是我们希望的，因为进程迁移的频率小就意味着产生的负载小。具体参考sched\_setaffinity函数。

### worker进程异常如何减少对业务的影响？

在实际线上环境中，经常出现这样的情况:某个多线程服务跑几个月后，因为未知原因进程挂了，最终造成整个服务都会不可用。

这时候，master+多worker的多进程模型就体现了它的优势，如果代码有隐藏的并且不容易触发的bug，某个时候如果某个请求触发了这个bug，则处理这个请求的worker进程会段错误退出。但是其他worker进程不会收到任何的影响，也就是说如果一个改造后的twemproxy起了20个worker进程，某个时候一个隐藏bug被某个请求触发，则只有处理该请求的进程段错误异常，其他19个进程不会受到任何影响，该隐藏bug触发后影响面仅为5%。如果是多线程模型，则影响面会是100%。

如果某个worker进程挂了，master父进程会感知到这个信号，然后重新拉起一个worker进程，实现瞬间无感知”拉起”恢复。以下为模拟触发异常段错误流程：



如上图所示，杀掉31420 worker进程后，master进程会立马在拉起一个31451工作进程，实现了快速恢复。

多进程异常，自动”拉起”功能源码，可以参考如下demo：

https://github.com/y123456yz/reading-code-of-nginx-1.9.2/blob/master/nginx-1.9.2/src/demo.c

# 网络优化

## 网卡多队列

在实际上线后，发现软中断过高，几乎大部分都集中在一个或者几个CPU上，严重影响客户端连接和数据转发，qps上不去，时延抖动厉害。

RSS（Receive Side Scaling）是网卡的硬件特性，实现了多队列，可以将不同的流分发到不同的CPU上。支持RSS的网卡，通过多队列技术，每个队列对应一个中断号，通过对每个中断的绑定，可以实现网卡中断在cpu多核上的分配，最终达到负载均衡的作用。

## 可怕的40ms

原生twemproxy在线上跑得过程中，发现时延波动很大，抓包发现其中部分数据包应答出现了40ms左右的时延，拉高了整体时延抓包如下：



解决办法如下：在recv系统调用后，调用一次setsockopt函数，设置TCP\_QUICKACK。代码修改如下：

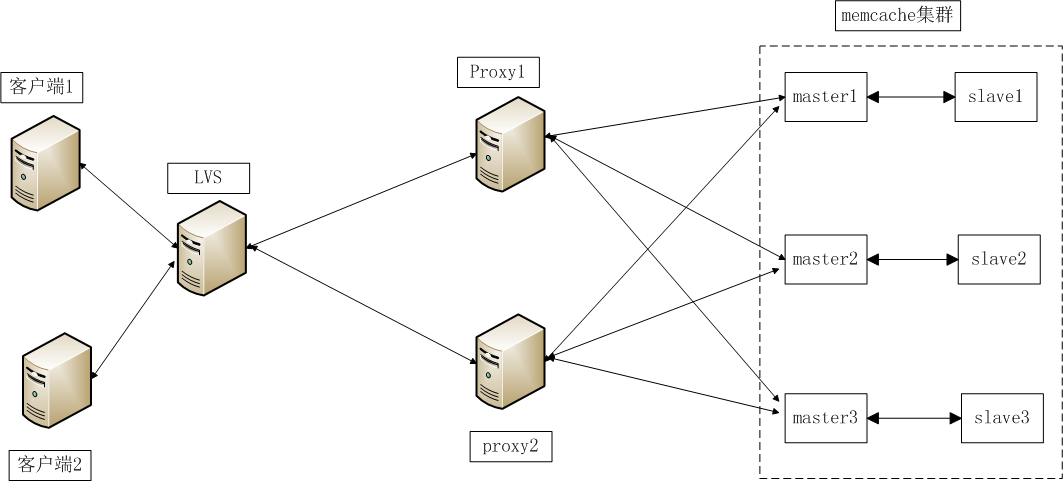


# Twemproxy改造前后性能对比(时延、qps对比)

## 线上真实流量时延对比

* + 1. 改造前线上twemproxy集群时延

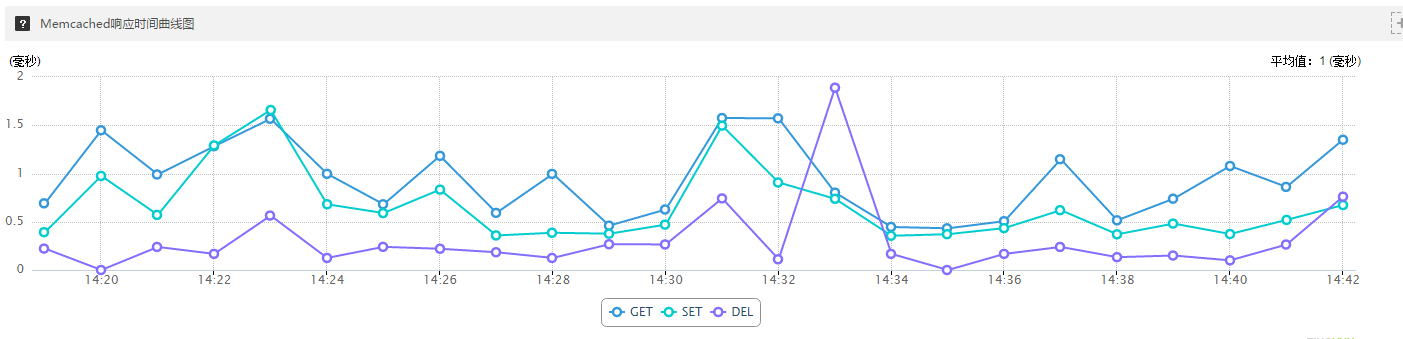
线上集群完全采用开源twemproxy做代理，架构如下：

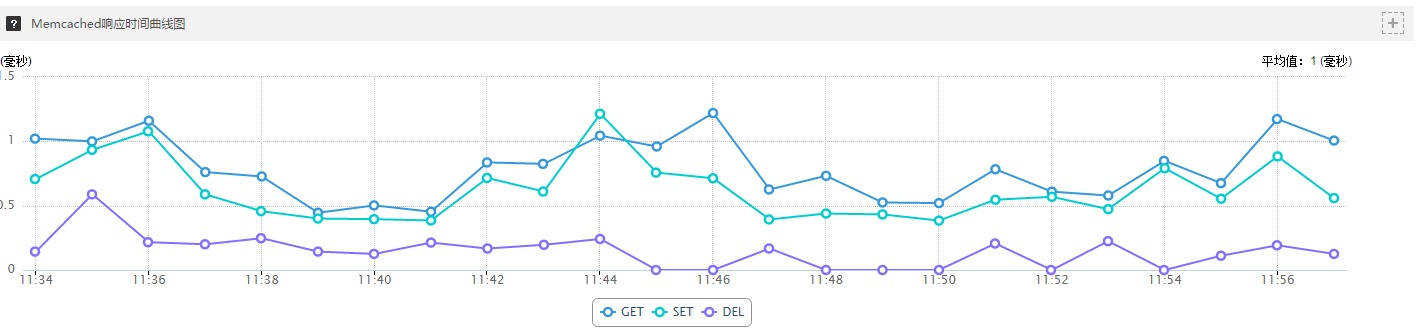


***原生twemproxy***

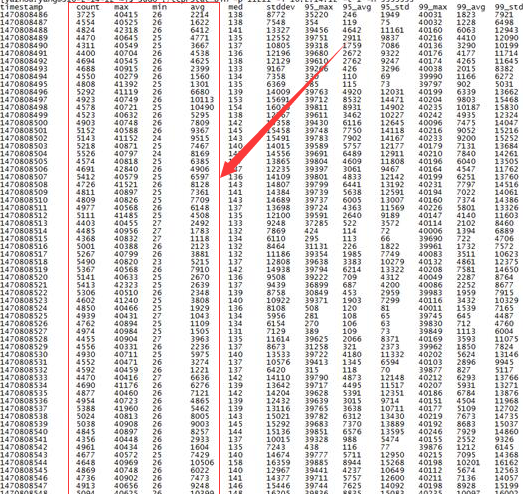
***原生twemproxy***

未改造前twemproxy集群，qps=5000~6000，长连接，客户端时延分布如下图所示：





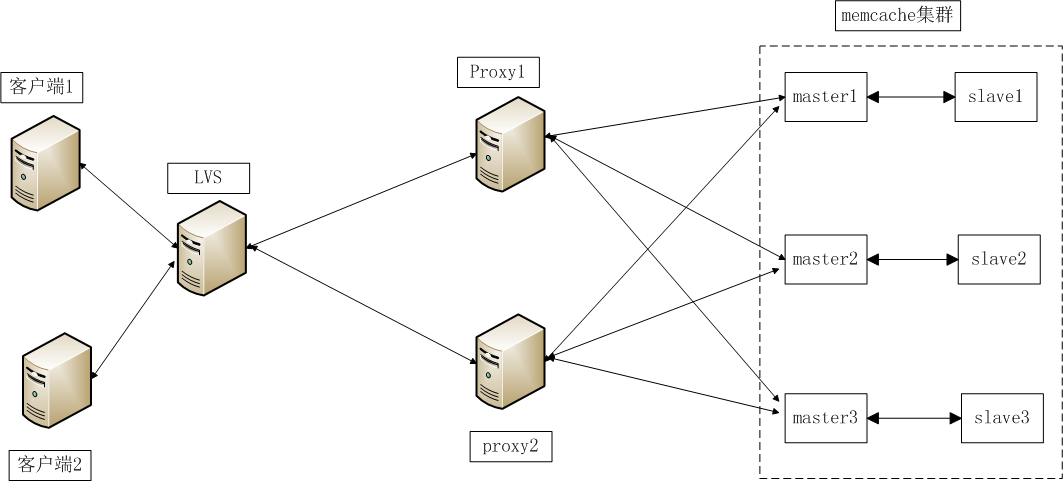
在twemproxy机器上使用tcprstat监控到的网卡时延如下:



从上面两个图可以看出，采用原生twemproxy,时延抖动厉害。

* + 1. 参照nginx改造后的twemproxy时延

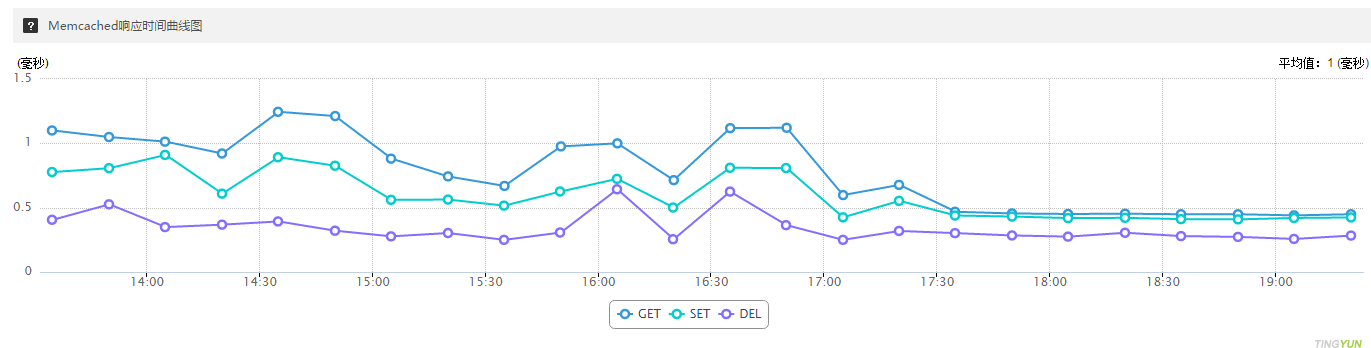
线上集群一个twemproxy采用官方原生twemproxy，另一个为改造后的twemproxy，其中改造后的twemproxy配置worker进程数为1，保持和原生开源twemproxy进程数一致，架构如下：

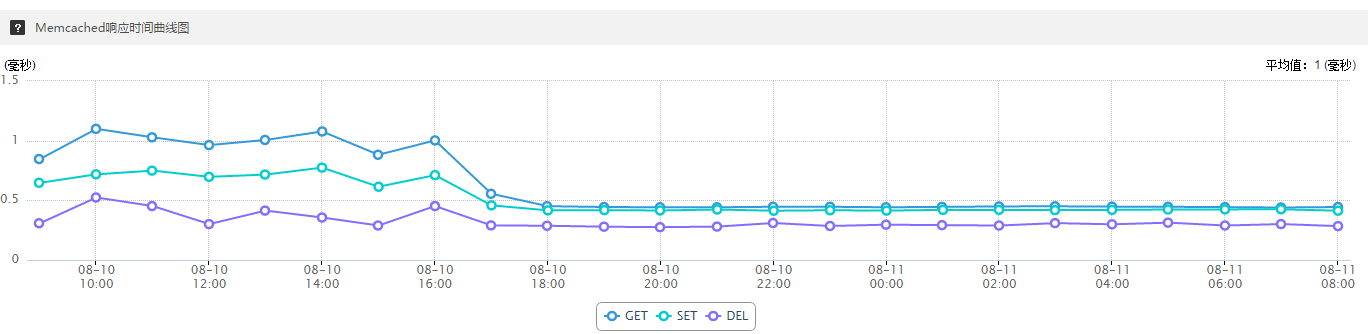


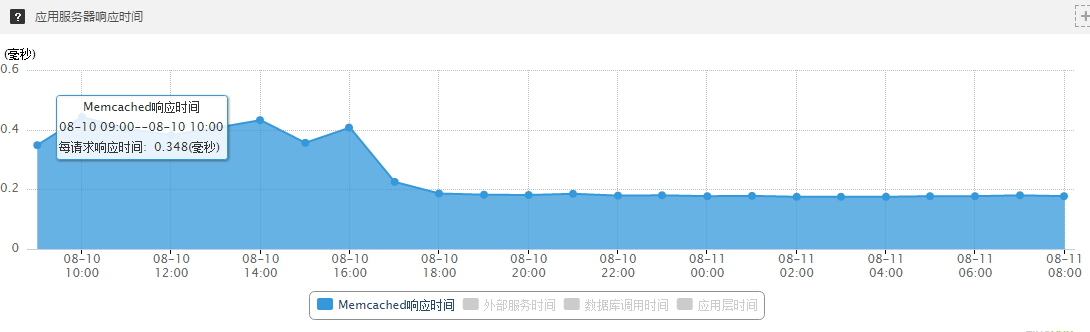
***多进程改造后twemproxy***

***原生twemproxy***

替换线上集群两个代理中的一个后(影响50%流量)，qps=5000~6000，客户端埋点监控时延分布如下:

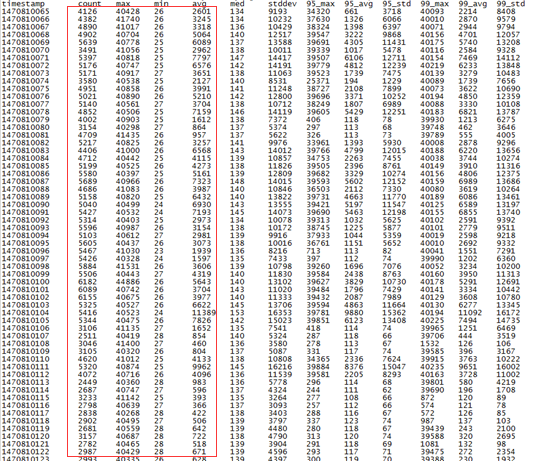




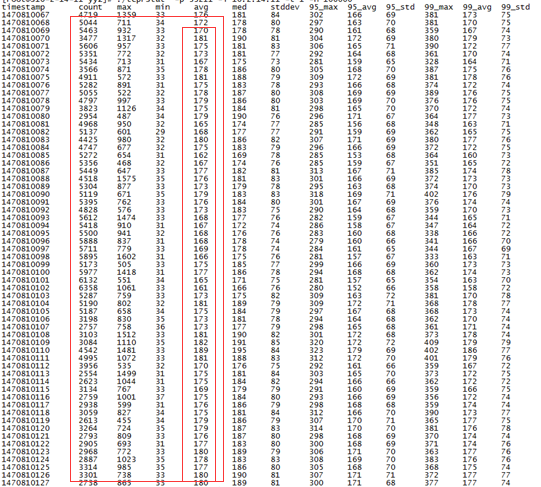


替换两个proxy中的一个后，使用tcprstat在代理集群上面查看两个代理的时延分布如下:

原生twemproxy节点机器上的时延分布:



另一个改造后的twemproxy节点机器上的时延分布：



总结：替换线上两个proxy中的一个后，客户端时间降低了一倍，如果线上集群两个代理都替换为改造后的twemproxy，客户端监控时延预计会再降低一倍，总体时延降低3倍左右。

此外，从监控可以看出，改造后的twemproxy时延更加稳定，无任何波动。

## 参考nginx多进程改造后的twemproxy线下压测结果(开启reuseport功能)

监听同一个端口，数据长度100字节，压测结果如下：

linux内核版本:linux-3.10

物理机机型： M10(48 cpu)

|  |  |  |
| --- | --- | --- |
| worker进程数 | 短链接最大tps | 长链接最大tps |
| 1 | 3.7万 | 11.5万 |
| 5 | 11万 | 41万 |
| 10 | 25万 | 85万 |
| 15 | 30万 | 112万 |
| 20 | 35万 | 133万 |
| 25 | 38万 | 137万 |
| 30 | 42万 | 141万 |
| 35 | 42.5万 | 146万 |
| 40 | 46万 | 152万 |

多进程监听同一个端口，数据长度150字节，压测结果如下：

 linux内核版本:linux-3.10

 物理机机型： TS60 (24 cpu)

|  |  |  |
| --- | --- | --- |
| worker进程数 | 短链接最大tps | 长链接最大tps |
| 1 | 4.8万 | 15万 |
| 2 | 8万 | 26万 |
| 3 | 11万 | 35万 |
| 4 | 13万 | 43万 |
| 5 | 15万 | 48万 |
| 6 | 17万 | 56万 |
| 7 | 19万 | 62万 |
| 8 | 20.5万 | 66万 |
| 9 | 21.3万 | 74万 |
| 10 | 23万 | 75万 |
| 12 | 25万 | 85万 |
| 14 | 26.5万 | 89万 |
| 16 | 28.5万 | 91万 |
| 18 | 29.2万 | 93万 |
| 20 | 30.5万 | 95万 |

# 总结

## 多进程、多线程机制选择

    选择参照nginx多进程机制，而不选择多线程实现原因主要有:

i) 多进程机制无锁操作，实现更容易

ii) 多进程的代理，整个worker进程无任何锁操作，性能更好

ii) 如果是多线程方式，如果代码出现bug段错误，则整个进程挂掉，整个服务不可用。而如果是多进程方式，因为bug触发某个worker进程段错误异常，其他工作进程不会受到如何影响，20个worker进程，如果触发异常，同一时刻只有有1/20的流量受到影响。而如果是多线程模式，则100%的流量会受到影响。

iii) worker进程异常退出后，master进程立马感知拉起一个新进程提供服务，可靠性更高。

iiii)  配置热加载、程序热升级功能实现更加容易

## 参照nginx改造后的twemproxy特性

 支持nginx几乎所有的优秀特性，同时也根据自己实际情况新增加了自有特性:

i)  master+多worker进程机制

ii) 适配所有linux内核版本，内核低版本惊群问题避免支持

iii) quic\_ack支持

iiii) reuser\_port适配支持

iiii) worker进程异常，master进程自动拉起功能支持

iiiiii) 90%、95%、98%、100%平均时延统计功能支持

iiiiiii) memcache单机版、集群版支持

 iiiiiiii) redis单机版、集群版支持

iiiiiiiii) redis、memcache集群在线扩容、缩容、数据迁移支持，扩缩容、数据迁移过程对业务无任何影响。

iiiiiiiiii) 多租户支持，一个代理可以接多个memcache、redis集群，并支持混部。

iiiiiiiiiii) mget、gets、sets等批量处理命令优化处理

iiiiiiiiiiii) 慢响应日志记录功能支持

iiiiiiiiiiiii) 内存参数实时修改支持

iiiiiiiiiiiiii) 详细的集群监控统计功能

iiiiiiiiiiiiiii) CPU亲缘性自添加

iiiiiiiiiiiiiiii)内存配置动态实时修改

## 6.3后期计划

   添加如下功能:

     i) 配置文件热加载支持。

     ii) 代码热升级功能支持。

## 长远规划展望

抽象出一款类似nginx的高性能代理软件，nginx支持http协议，我们的支持tcp协议代理，覆盖nginx所有功能，包括前面提到的所有功能，同时支持模块化开发。这样，很多的tcp协议代理就无需关心网络架构底层实现，只想要关心自己的协议和一些自己关心的统计、审计等功能，降低开发成本。现有开源的中间件，很大一部分都是tcp的，有自己的私有tcp协议，把这个抽象出来，开发成本会更低。

对nginx有兴趣的可以源码分析参考：

<https://github.com/y123456yz/reading-code-of-nginx-1.9.2>

内核网卡时延分析工具:

<https://github.com/y123456yz/tcprstat>

twemproxy源码分析:

<https://github.com/y123456yz/Reading-and-comprehense-twemproxy0.4.1>

内核协议栈延迟确认机制：

https://github.com/y123456yz/Reading-and-comprehense-linux-Kernel-network-protocol-stack