聊聊TCP连接耗时的那些事儿

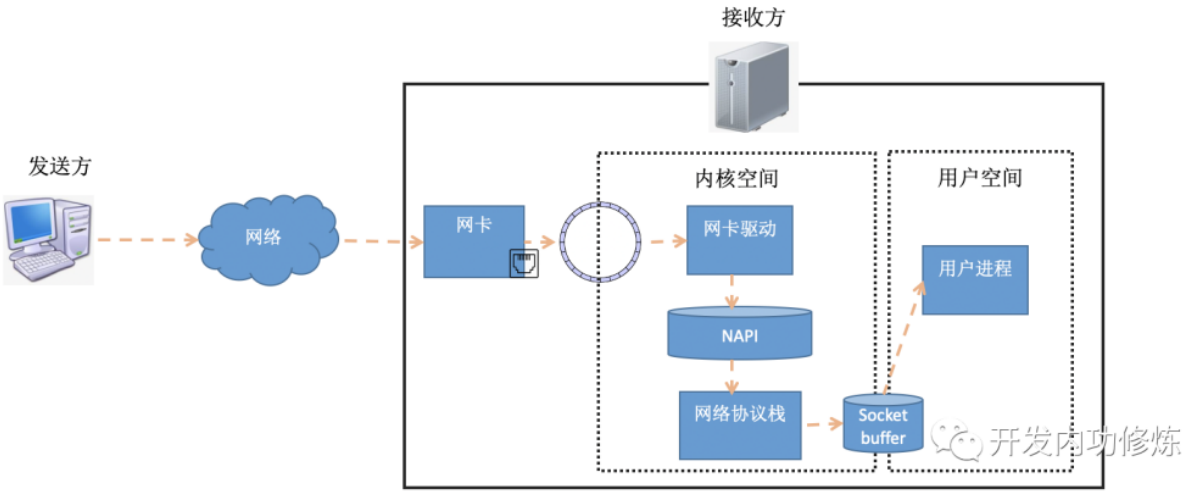
原创 张彦飞allen 开发内功修炼 2020-11-08 08:00

在互联网后端日常开发接口的时候中，不管你使用的是C、Java、PHP还是Golang，都避免不了需要调用mysql、redis等组件来获取数据，可能还需要执行一些rpc远程调用，或者再调用一些其它restful api。 在这些调用的底层，基本都是在使用TCP协议进行传输。这是因为在传输层协议中，TCP协议具备可靠的连接，错误重传，拥塞控制等优点，所以目前应用比UDP更广泛一些。

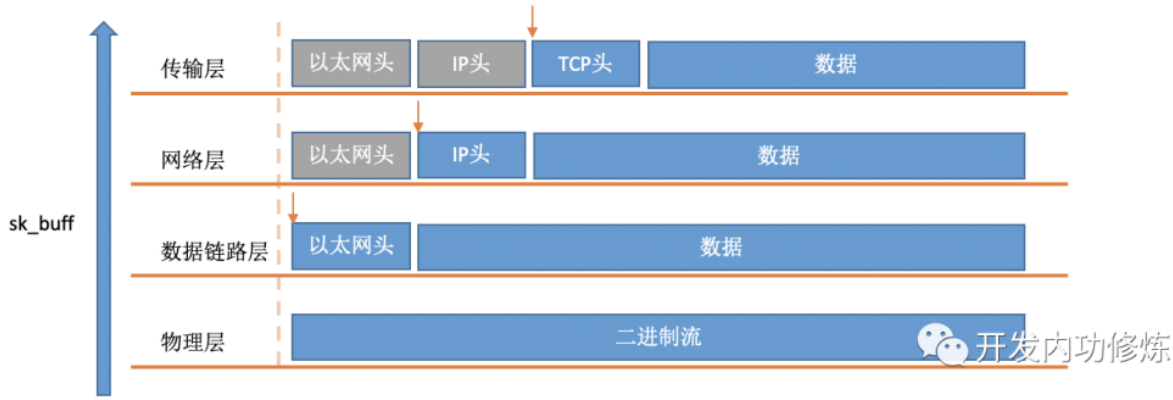
相信你也一定听闻过TCP也存在一些缺点，那就是老生常谈的开销要略大。但是各路技术博客里都在单单说开销大、或者开销小，而少见不给出具体的量化分析。不客气一点，这都是营养不大的废话。经过日常工作的思考之后，我更想弄明白的是，开销到底多大。一条TCP连接的建立需要耗时延迟多少，是多少毫秒，还是多少微秒？能不能有一个哪怕是粗略的量化估计？当然影响TCP耗时的因素有很多，比如网络丢包等等。我今天只分享我在工作实践中遇到的比较高发的各种情况。

一 正常TCP连接建立过程

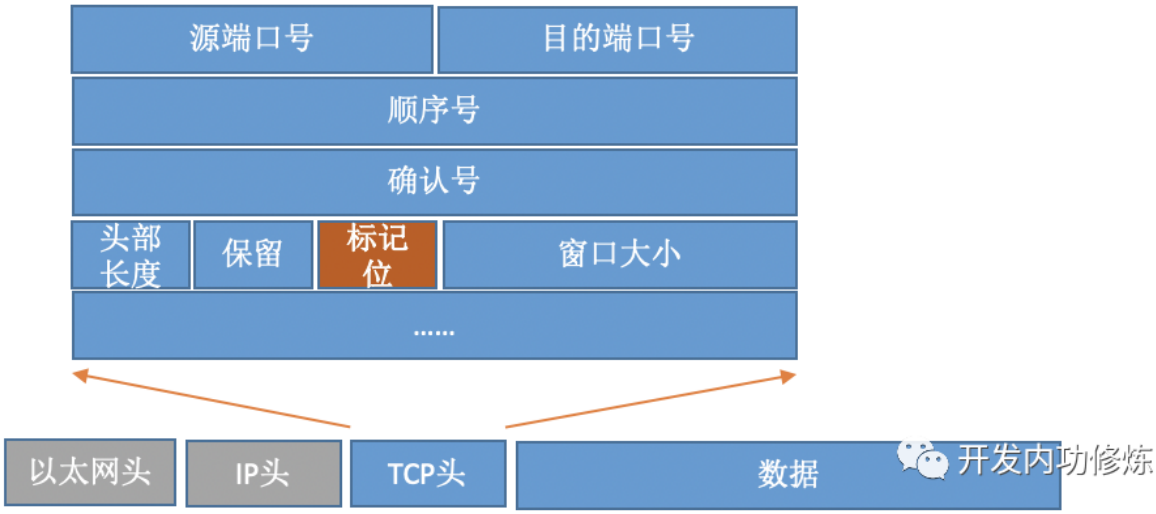
要想搞清楚TCP连接的建立耗时，我们需要详细了解连接的建立过程。在前文《图解Linux网络包接收过程》中我们介绍了数据包在接收端是怎么被接收的。数据包从发送方出来，经过网络到达接收方的网卡。在接收方网卡将数据包DMA到RingBuffer后，内核经过硬中断、软中断等机制来处理（如果发送的是用户数据的话，最后会发送到socket的接收队列中，并唤醒用户进程）。



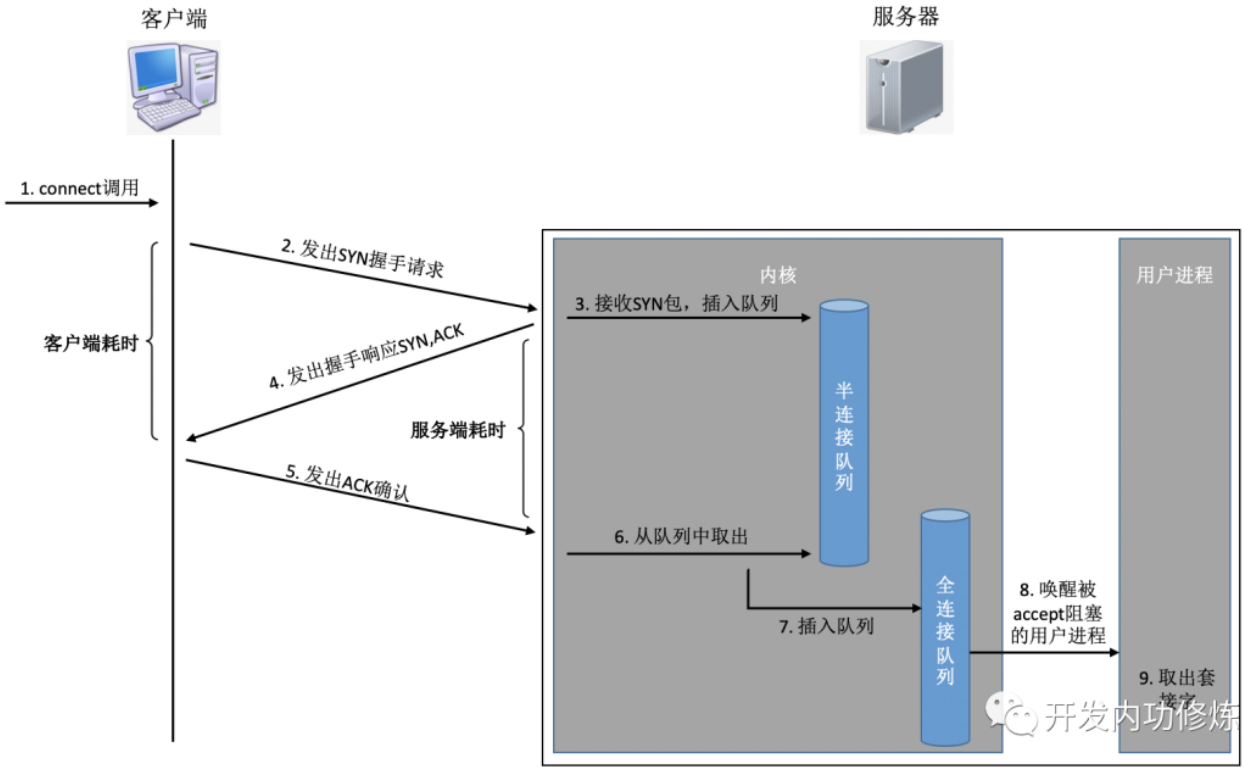
在软中断中，当一个包被内核从RingBuffer中摘下来的时候，在内核中是用 struct sk_buff 结构体来表示的(参见内核代码 include/linux/skbuff.h)。其中的data成员是接收到的数据，在协议栈逐层被处理的时候，通过修改指针指向data的不同位置，来找到每一层协议关心的数据。



对于TCP协议包来说，它的Header中有一个重要的字段-flags。如下图：



通过设置不同的标记为，将TCP包分成SYNC、FIN、ACK、RST等类型。客户端通过connect系统调用命令内核发出SYNC、ACK等包来实现和服务器TCP连接的建立。在服务器端，可能会接收许许多多的连接请求，内核还需要借助一些辅助数据结构-半连接队列和全连接队列。我们来看一下整个连接过程：



在这个连接过程中，我们来简单分析一下每一步的耗时

- 客户端发出SYNC包：客户端一般是通过connect系统调用来发出SYN的，这里牵涉到本机的系统调用和软中断的CPU耗时开销
- **SYN传到服务器**：SYN从客户端网卡被发出，开始“跨过山和大海，也穿过人山人海……”，这是一次长途远距离的网络传输
- 服务器处理SYN包：内核通过软中断来收包，然后放到半连接队列中，然后再发出SYN/ACK响应。又是CPU耗时开销
- **SYC/ACK传到客户端**：SYC/ACK从服务器端被发出后，同样跨过很多山、可能很多大海来到客户端。又一次长途网络跋涉
- 客户端处理SYN/ACK：客户端内核收包并处理SYN后，经过几us的CPU处理，接着发出ACK。同样是软中断处理开销
- **ACK传到服务器**：和SYN包，一样，再经过几乎同样远的路，传输一遍。 又一次长途网络跋涉
- 服务端收到ACK：服务器端内核收到并处理ACK，然后把对应的连接从半连接队列中取出来，然后放到全连接队列中。一次软中断CPU开销
- 服务器端用户进程唤醒：正在被accpet系统调用阻塞的用户进程被唤醒，然后从全连接队列中取出来已经建立好的连接。一次上下文切换的CPU开销

以上几步操作，可以简单划分为两类：

- 第一类是内核消耗CPU进行接收、发送或者是处理，包括系统调用、软中断和上下文切换。它们的耗时基本都是几个us左右。具体的分析过程可以参见《一次系统调用开销到底有多大？》、《软中断会吃掉你多少CPU？》、《进程/线程切换会用掉你多少CPU？》这三篇文章。
- 第二类是网络传输，当包被从一台机器上发出以后，中间要经过各式各样的网线、各种交换机路由器。所以网络传输的耗时相比本机的CPU处理，就要高的多了。根据网

络远近一般在几ms~到几百ms不等。。

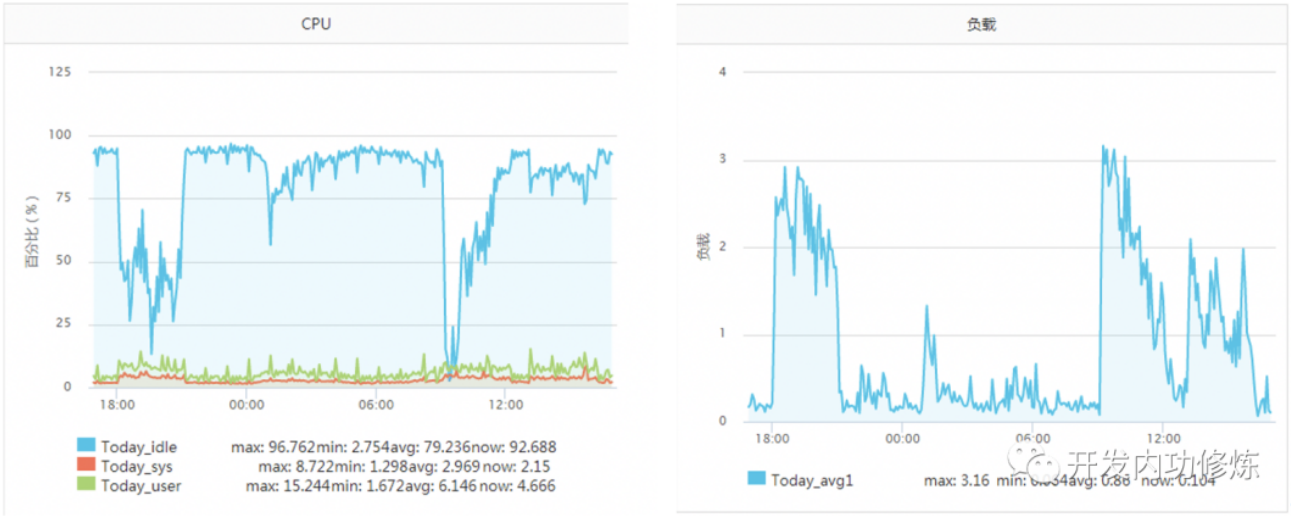
1ms就等于1000us，因此网络传输耗时比双端的CPU开销要高1000倍左右，甚至更高可能还到100000倍。所以，在正常的TCP连接的建立过程中，一般可以考虑网络延时即可。一个RTT指的是包从一台服务器到另外一台服务器的一个来回的延迟时间。所以从全局来看，TCP连接建立的网络耗时大约需要三次传输，再加上少许的双方CPU开销，总共大约比1.5倍RTT大一点点。不过从客户端视角来看，只要ACK包发出了，内核就认为连接是建立成功了。所以如果在客户端打点统计TCP连接建立耗时的话，只需要两次传输耗时-既1个RTT多一点的时间。（对于服务器端视角来看同理，从SYN包收到开始算，到收到ACK，中间也是一次RTT耗时）

二 TCP连接建立时的异常情况

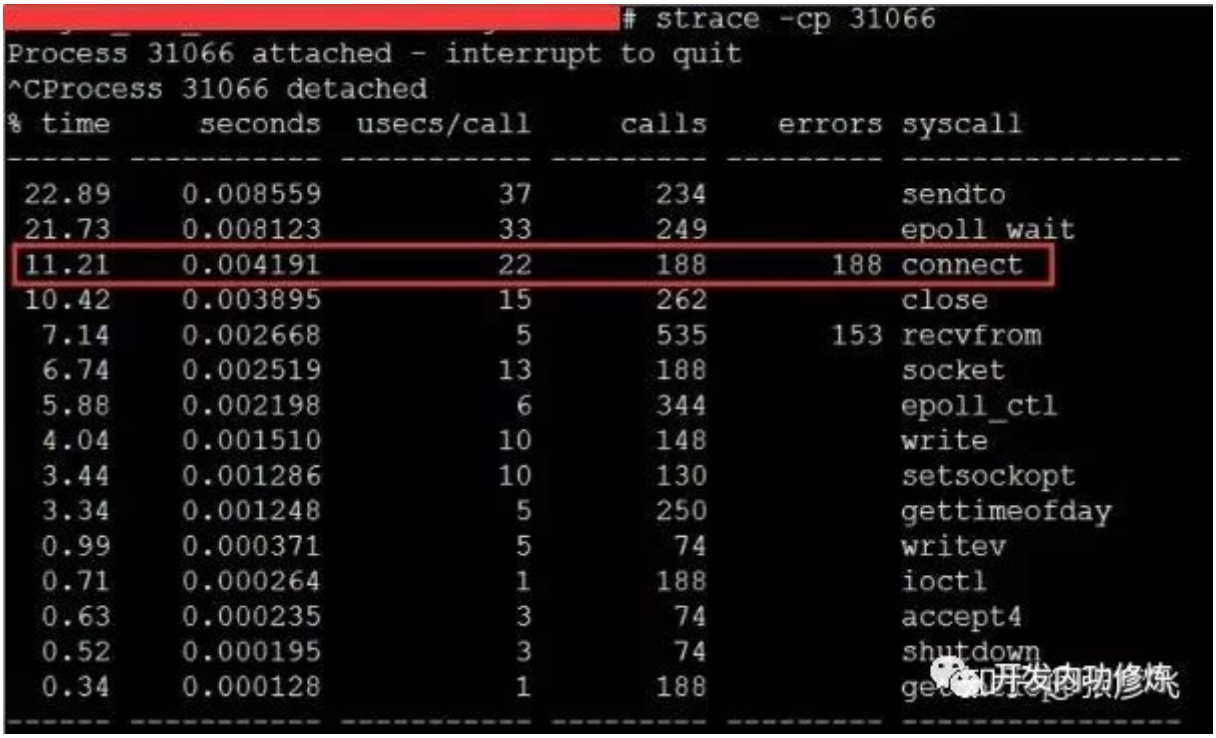
上一节可以看到在客户端视角，在正常情况下一次TCP连接总的耗时也就就大约是一次网络RTT的耗时。如果所有的事情都这么简单，我想我的这次分享也就没有必要了。事情不一定总是这么美好，总会有意外发生。在某些情况下，可能会导致连接时的网络传输耗时上涨、CPU处理开销增加、甚至是连接失败。现在我们说一下我在线上遇到过的各种沟沟坎坎。

1）客户端connect系统调用耗时失控

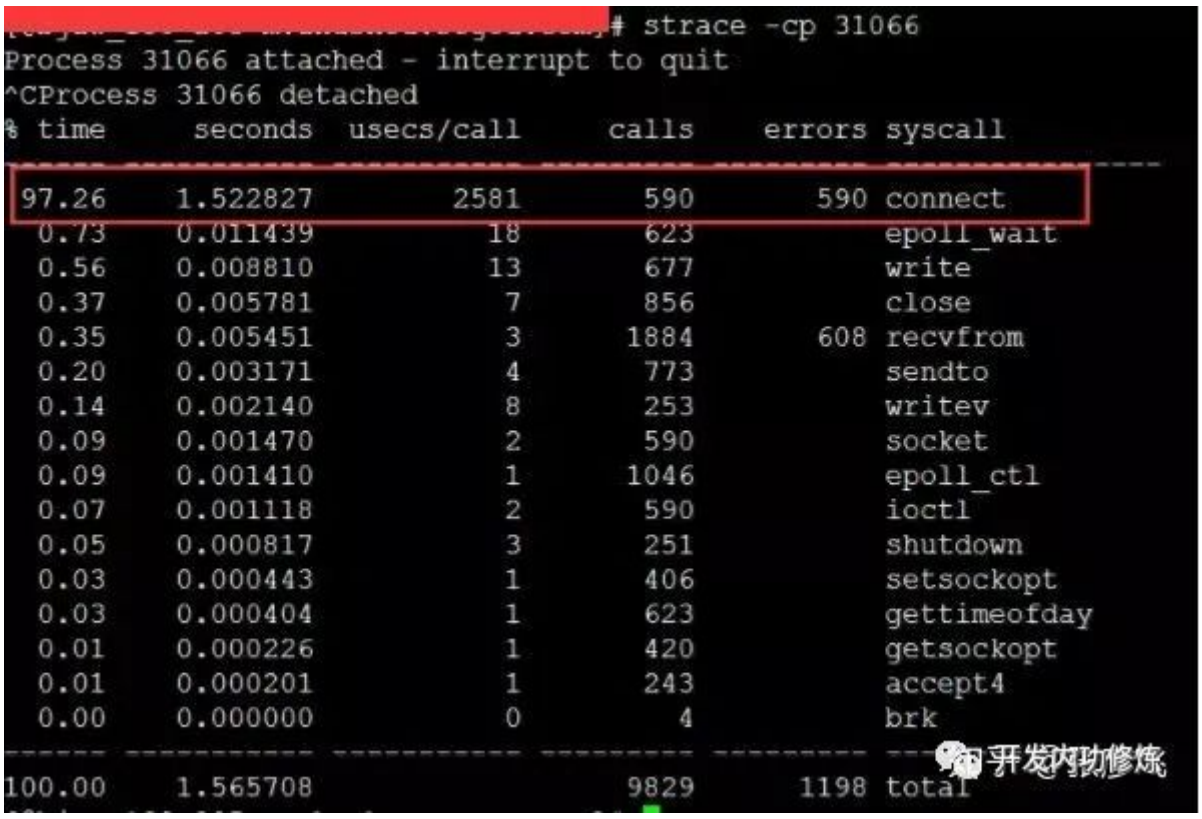
正常一个系统调用的耗时也就是几个us（微秒）左右。但是在《追踪将服务器CPU耗光的凶手!》一文中笔者的一台服务器当时遇到一个状况，某次运维同学转达过来说该服务CPU不够用了，需要扩容。当时的服务器监控如下图：



该服务之前一直每秒抗2000左右的qps，CPU的idel一直有70%+。怎么突然就CPU一下就不够用了呢。而且更奇怪的是CPU被打到谷底的那一段时间，负载却并不高（服务器为4核机器，负载3-4是比较正常的）。 后来经过排查以后发现当TCP客户端TIME_WAIT有30000左右，导致可用端口不是特别充足的时候，connect系统调用的CPU开销直接上涨了100多倍，每次耗时达到了2500us（微秒），达到了毫秒级别。



```
[redacted]# strace -cp 31066
Process 31066 attached - interrupt to quit
^CProcess 31066 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 22.89    0.008559          37       234           sendto
 21.73    0.008123          33       249           epoll_wait
 11.21    0.004191          22       188       188 connect
 10.42    0.003895          15       262           close
  7.14    0.002668           5       535       153 recvfrom
  6.74    0.002519          13       188           socket
  5.88    0.002198           6       344           epoll_ctl
  4.04    0.001510          10       148           write
  3.44    0.001286          10       130           setsockopt
  3.34    0.001248           5       250           gettimeofday
  0.99    0.000371           5        74           writev
  0.71    0.000264           1       188           ioctl
  0.63    0.000235           3        74           accept4
  0.52    0.000195           3        74           shutdown
  0.34    0.000128           1       188           ge[redacted]
------ ----------- ----------- --------- --------- ----------------
```

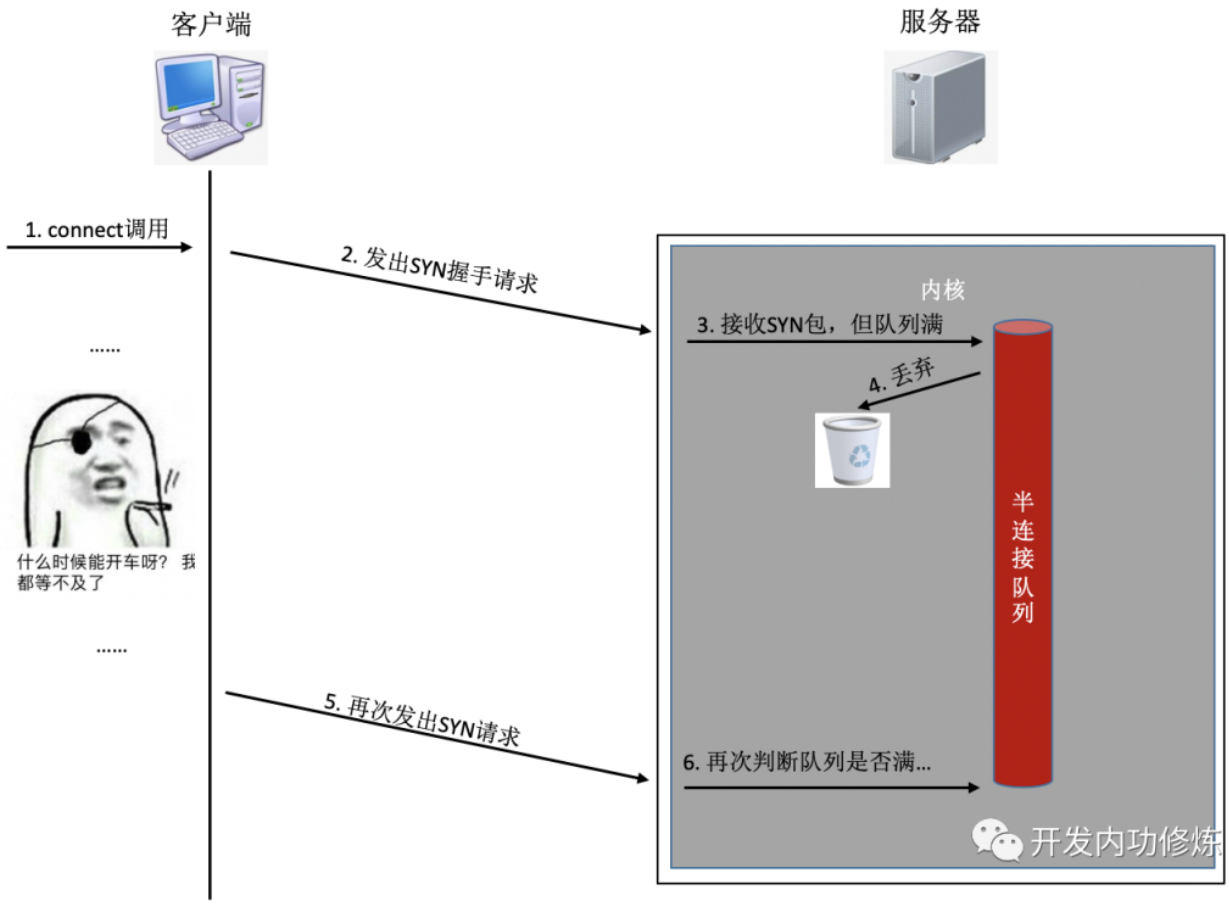



```
[redacted]# strace -cp 31066
Process 31066 attached - interrupt to quit
^CProcess 31066 detached
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
 97.26    1.522827        2581       590       590 connect
  0.73    0.011439          18       623           epoll_wait
  0.56    0.008810          13       677           write
  0.37    0.005781           7       856           close
  0.35    0.005451           3      1884       608 recvfrom
  0.20    0.003171           4       773           sendto
  0.14    0.002140           8       253           writev
  0.09    0.001470           2       590           socket
  0.09    0.001410           1      1046           epoll_ctl
  0.07    0.001118           2       590           ioctl
  0.05    0.000817           3       251           shutdown
  0.03    0.000443           1       406           setsockopt
  0.03    0.000404           1       623           gettimeofday
  0.01    0.000226           1       420           getsockopt
  0.01    0.000201           1       243           accept4
  0.00    0.000000           0         4           brk
------ ----------- ----------- --------- --------- ----------------
100.00    1.565708                  9829      1198 total
```


当遇到这种问题的时候，虽然TCP连接建立耗时只增加了2ms左右，整体TCP连接耗时看起来还可接受。但是这里的问题在于这2ms多都是在消耗CPU的周期，所以问题不小。 解决起来也非常简单，办法很多：修改内核参数net.ipv4.ip_local_port_range多预留一些端口号、改用长连接都可以。

2）半/全连接队列满

如果连接建立的过程中，任意一个队列满了，那么客户端发送过来的syn或者ack就会被丢弃。客户端等待很长一段时间无果后，然后会发出TCP Retransmission重传。拿半连接队列举例：



要知道的是上面TCP握手超时重传的时间是秒级别的。也就是说一旦server端的连接队列导致连接建立不成功，那么光建立连接就至少需要秒级以上。而正常的在同机房的情况下只是不到1毫秒的事情，整整高了1000倍左右。尤其是对于给用户提供实时服务的程序来说，用户体验将会受到较大影响。如果连重传也没有握手成功的话，很可能等不及二次重试，这个用户访问直接就超时了。

还有另外一个更坏的情况是，它还有可能会影响其它的用户。假如你使用的是进程/线程池这种模型提供服务，比如php-fpm。我们知道fpm进程是阻塞的，当它响应一个用户请求的时候，该进程是没有办法再响应其它请求的。假如你开了100个进程/线程，而某一段时间内有50个进程/线程卡在和redis或者mysql服务器的握手连接上了（**注意：这个时候你的服务器是TCP连接的客户端一方**）。这一段时间内相当于你可以用的正常工作的进程/线程只有50个了。而这个50个worker可能根本处理不过来，这时候你的服务可能就会产生拥堵。再持续稍微时间长一点的话，可能就产生雪崩了，整个服务都有可能会受影响。

既然后果有可能这么严重，那么我们如何查看我们手头的服务是否有因为半/全连接队列满的情况发生呢？在客户端，可以抓包查看是否有SYN的TCP Retransmission。如果有偶发的TCP Retransmission，那就说明对应的服务端连接队列可能有问题了。

在服务端的话，查看起来就更方便一些了。`netstat -s` 可查看到当前系统半连接队列满导致的丢包统计，但该数字记录的是总丢包数。你需要再借助 `watch` 命令动态监控。如果下面的数字在你监控的过程中变了，那说明当前服务器有因为半连接队列满而产生的丢包。你可能需要加大你的半连接队列的长度了。

```
$ watch 'netstat -s | grep LISTEN'
    8 SYNs to LISTEN sockets ignored
```

对于全连接队列来说呢，查看方法也类似。

```
$ watch 'netstat -s | grep overflowed'
   160 times the listen queue of a socket overflowed
```

如果你的服务因为队列满产生丢包，其中一个做法就是加大半/全连接队列的长度。 半连接队列长度Linux内核中，主要受tcp_max_syn_backlog影响 加大它到一个合适的值就可以。

```
# cat /proc/sys/net/ipv4/tcp_max_syn_backlog
1024
# echo "2048" > /proc/sys/net/ipv4/tcp_max_syn_backlog
```

全连接队列长度是应用程序调用listen时传入的backlog以及内核参数net.core.somaxconn二者之中较小的那个。你可能需要同时调整你的应用程序和该内核参数。

```
# cat /proc/sys/net/core/somaxconn
128
# echo "256" > /proc/sys/net/core/somaxconn
```

改完之后我们可以通过ss命令输出的 `Send-Q` 确认最终生效长度：

```
$ ss -nlt
Recv-Q Send-Q Local Address:Port Address:Port
0      128    *:80              *:*
```

> `Recv-Q` 告诉了我们当前该进程的全连接队列使用长度情况。如果 `Recv-Q` 已经逼近了 `Send-Q` ,那么可能不需要等到丢包也应该准备加大你的全连接队列了。

如果加大队列后仍然有非常偶发的队列溢出的话，我们可以暂且容忍。如果仍然有较长时间处理不过来怎么办？另外一个做法就是直接报错，不要让客户端超时等待。例如将Redis、Mysql等后端接口的内核参数tcp_abort_on_overflow为1。如果队列满了，直接发reset给client。告诉后端进程/线程不要痴情地傻等。这时候client会收到错误“connection reset by peer”。**牺牲一个用户的访问请求，要比把整个站都搞崩了还是要强的。**

三 TCP连接耗时实测

我写了一段非常简单的代码，用来在客户端统计每创建一个TCP连接需要消耗多长时间。

```
<?php
$ip = {服务器ip};
$port = {服务器端口};
$count = 50000;
function buildConnect($ip,$port,$num){
    for($i=0;$i<$num;$i++){
        $socket = socket_create(AF_INET,SOCK_STREAM,SOL_TCP);
        if($socket ==false) {
            echo "$ip $port socket_create() 失败的原因是:".socket_strerror(socket_last_error($socket))."\n";
            sleep(5);
            continue;
        }

        if(false == socket_connect($socket, $ip, $port)){
            echo "$ip $port socket_connect() 失败的原因是:".socket_strerror(socket_last_error($socket))."\n";
            sleep(5);
            continue;
        }
        socket_close($socket);
    }
}

$t1 = microtime(true);
buildConnect($ip, $port, $count);
echo (($t2-$t1)*1000).'ms';
```

在测试之前，我们需要本机linux可用的端口数充足，如果不够50000个，最好调整充足。

```
# echo "5000   65000" /proc/sys/net/ipv4/ip_local_port_range
```

1）正常情况

注意：无论是客户端还是服务器端都不要选择有线上服务在跑的机器，否则你的测试可能会影响正常用户访问

首先我的客户端位于河北怀来的IDC机房内，服务器选择的是公司广东机房的某台机器。执行ping命令得到的延迟大约是37ms，使用上述脚本建立50000次连接后，得到的连接平均耗时也是37ms。这是因为前面我们说过的，对于客户端来看，第三次的握手只要包发送出去，就认为是握手成功了，所以只需要一次RTT、两次传输耗时。虽然这中间还会有客户端和服务端的系统调用开销、软中断开销，但由于它们的开销正常情况下只有几个us(微秒)，所以对总的连接建立延时影响不大。

接下来我换了一台目标服务器，该服务器所在机房位于北京。离怀来有一些距离，但是和广东比起来可要近多了。这一次ping出来的RTT是1.6~1.7ms左右，在客户端统计建立50000次连接后算出每条连接耗时是

1.64ms。

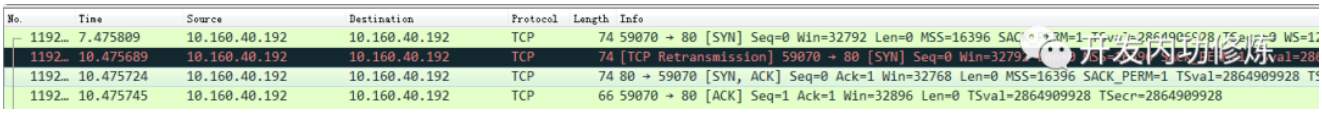
再做一次实验，这次选中实验的服务器和客户端直接位于同一个机房内，ping延迟在0.2ms~0.3ms左右。跑了以上脚本以后，实验结果是50000 TCP连接总共消耗了11605ms，平均每次需要0.23ms。

> **线上架构提示**：这里看到同机房延迟只有零点几ms，但是跨个距离不远的机房，光TCP握手耗时就涨了4倍。如果再要是跨地区到广东，那就是百倍的耗时差距了。线上部署时，理想的方案是将自己服务依赖的各种mysql、redis等服务和自己部署在同一个地区、同一个机房（再变态一点，甚至可以是甚至是同一个机架）。因为这样包括TCP链接建立啥的各种网络包传输都要快很多。要尽可能避免长途跨地区机房的调用情况出现。

2）连接队列溢出

测试完了跨地区、跨机房和跨机器。这次为了快，直接和本机建立连接结果会咋样呢？Ping本机ip或127.0.0.1的延迟大概是0.02ms，本机ip比其它机器RTT肯定要短。我觉得肯定连接会非常快，嗯实验一下。连续建立5W TCP连接，总时间消耗27154ms，平均每次需要0.54ms左右。嗯！？怎么比跨机器还长很多？ 有了前面的理论基础，我们应该想到了，由于本机RTT太短，所以瞬间连接建立请求量很大，就会导致全连接队列或者半连接队列被打满的情况。一旦发生队列满，当时撞上的那个连接请求就得需要3秒+的连接建立延时。所以上面的实验结果中，平均耗时看起来比RTT高很多。

在实验的过程中，我使用tcpdump抓包看到了下面的一幕。原来有少部分握手耗时3s+，原因是半连接队列满了导致客户端等待超时后进行了SYN的重传。



我们又重新改成每500个连接，sleep 1秒。嗯好，终于没有卡的了（或者也可以加大连接队列长度）。结论是本机50000次TCP连接在客户端统计总耗时102399 ms，减去sleep的100秒后，平均每个TCP连接消耗0.048ms。比ping延迟略高一些。这是因为当RTT变的足够小的时候，内核CPU耗时开销就会显现出来了，另外TCP连接要比ping的icmp协议更复杂一些，所以比ping延迟略高0.02ms左右比较正常。

四 结论

TCP连接建立异常情况下，可能需要好几秒，一个坏处就是会影响用户体验，甚至导致当前用户访问超时都有可能。另外一个坏处是可能会诱发雪崩。所以当你的服务器使用短连接的方式访问数据的时候，一定要学会要监控你的服务器的连接建立是否有异常状态发生。如果有，学会优化掉它。当然你也可以采用本机内存缓存，或者使用连接池来保持长连接，通过这两种方式直接避免掉TCP握手挥手的各种开销也可以。

再说正常情况下，TCP建立的延时大约就是两台机器之间的一个RTT耗时，这是避免不了的。但是你可以控制两台机器之间的物理距离来降低这个RTT，比如把你要访问的redis尽可能地部署的离后端接口机器近一点，这样RTT也能从几十ms削减到最低可能零点几ms。

最后我们再思考一下，如果我们把服务器部署在北京，给纽约的用户访问可行吗？ 前面的我们同机房也好，跨机房也好，电信号传输的耗时基本可以忽略（因为物理距离很近），网络延迟基本上是转发设备占用的耗时。但是如果是跨越了半个地球的话，电信号的传输耗时我们可得算一算了。 北京到纽约的球面距离大概是15000公里，那么抛开设备转发延迟，仅仅光速传播一个来回（RTT是Rround trip time，要跑两次），需要时间 = 15,000,000 *2 / 光速 = 100ms。实际的延迟可能比这个还要大一些，一般都得200ms以上。建立在这个延迟上，要想提供用户能访问的秒级服务就很困难了。所以对于海外用户，最好都要在当地建机房或者购买海外的服务器。

相关阅读：

- 1.图解Linux网络包接收过程
- 2.Linux网络包接收过程的监控与调优
- 3.进程/线程切换究竟需要多少开销?
- 4.软中断会吃掉你多少CPU?
- 5.一次系统调用开销到底有多大?
- 6.追踪将服务器CPU耗光的凶手

收录于合集 #开发内功修炼之网络篇 42

< 上一篇
Linux网络包接收过程的监控与调优

下一篇 >
漫画 | 一台Linux服务器最多能支撑多少个TCP连接?

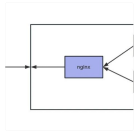
喜欢此内容的人还喜欢

第二本书交稿了

开发内功修炼

结合 Docker，快速掌握 Nginx 2 大核心用法

神光的编程秘籍



一样练！别人7天出腹肌，我咋1斤都不掉？

咕噜健身厨房