

36 | 套路篇：怎么评估系统的网络性能？

2019-02-13 倪朋飞



朗读：冯永吉

时长16:04 大小14.73M



你好，我是倪朋飞。

上一节，我们回顾了经典的 C10K 和 C1000K 问题。简单回顾一下，C10K 是指如何单机同时处理 1 万个请求（并发连接 1 万）的问题，而 C1000K 则是单机支持处理 100 万个请求（并发连接 100 万）的问题。

I/O 模型的优化，是解决 C10K 问题的最佳良方。Linux 2.6 中引入的 epoll，完美解决了 C10K 的问题，并一直沿用至今。今天的很多高性能网络方案，仍都基于 epoll。

自然，随着互联网技术的普及，催生出更高的性能需求。从 C10K 到 C100K，我们只需要增加系统的物理资源，就可以满足要求；但从 C100K 到 C1000K，光增加物理资源就不够了。

这时，就要对系统的软硬件进行统一优化，从硬件的中断处理，到网络协议栈的文件描述符数量、连接状态跟踪、缓存队列，再到应用程序的工作模型等的整个网络链路，都需要深入优化。

再进一步，要实现 C10M，就不是增加物理资源、调优内核和应用程序可以解决的问题了。这时内核中冗长的网络协议栈就成了最大的负担。

需要用 XDP 方式，在内核协议栈之前，先处理网络包。

或基于 DPDK，直接跳过网络协议栈，在用户空间通过轮询的方式处理。

其中，DPDK 是目前最主流的高性能网络方案，不过，这需要能支持 DPDK 的网卡配合使用。

当然，实际上，在大多数场景中，我们并不需要单机并发 1000 万请求。通过调整系统架构，把请求分发到多台服务器中并行处理，才是更简单、扩展性更好的方案。

不过，这种情况下，就需要我们评估系统的网络性能，以便考察系统的处理能力，并为容量规划提供基准数据。

那么，到底该怎么评估网络的性能呢？今天，我就带你一起来看看这个问题。

性能指标回顾

在评估网络性能前，我们先来回顾一下，衡量网络性能的指标。在 Linux 网络基础篇中，我们曾经说到，带宽、吞吐量、延时、PPS 等，都是最常用的网络性能指标。还记得它们的具体含义吗？你可以先思考一下，再继续下面的内容。

首先，**带宽**，表示链路的最大传输速率，单位是 b/s（比特 / 秒）。在你为服务器选购网卡时，带宽就是最核心的参考指标。常用的带宽有 1000M、10G、40G、100G 等。

第二，**吞吐量**，表示没有丢包时的最大数据传输速率，单位通常为 b/s（比特 / 秒）或者 B/s（字节 / 秒）。吞吐量受带宽的限制，吞吐量 / 带宽也就是该网络链路的使用率。

第三，**延时**，表示从网络请求发出后，一直到收到远端响应，所需要的时间延迟。这个指标在不同场景中可能会有不同的含义。它可以表示建立连接需要的时间（比如 TCP 握手延

时)，或者一个数据包往返所需时间（比如 RTT）。

最后，**PPS**，是 Packet Per Second（包 / 秒）的缩写，表示以网络包为单位的传输速率。PPS 通常用来评估网络的转发能力，而基于 Linux 服务器的转发，很容易受到网络包大小的影响（交换机通常不会受到太大影响，即交换机可以线性转发）。

这四个指标中，带宽跟物理网卡配置是直接关联的。一般来说，网卡确定后，带宽也就确定了（当然，实际带宽会受限于整个网络链路中最小的那个模块）。

另外，你可能在很多地方听说过“网络带宽测试”，这里测试的实际上不是带宽，而是网络吞吐量。Linux 服务器的网络吞吐量一般会比带宽小，而对交换机等专门的网络设备来说，吞吐量一般会接近带宽。

最后的 PPS，则是以网络包为单位的网络传输速率，通常用在需要大量转发的场景中。而对 TCP 或者 Web 服务来说，更多会用并发连接数和每秒请求数（QPS，Query per Second）等指标，它们更能反应实际应用程序的性能。

网络基准测试

熟悉了网络的性能指标后，接下来，我们再来看看，如何通过性能测试来确定这些指标的基准值。

你可以先思考一个问题。我们已经知道，Linux 网络基于 TCP/IP 协议栈，而不同协议层的行为显然不同。那么，测试之前，你应该弄清楚，你要评估的网络性能，究竟属于协议栈的哪一层？换句话说，你的应用程序基于协议栈的哪一层呢？

根据前面学过的 TCP/IP 协议栈的原理，这个问题应该不难回答。比如：

基于 HTTP 或者 HTTPS 的 Web 应用程序，显然属于应用层，需要我们测试 HTTP/HTTPS 的性能；

而对大多数游戏服务器来说，为了支持更大的同时在线人数，通常会基于 TCP 或 UDP，与客户端进行交互，这时就需要我们测试 TCP/UDP 的性能；

当然，还有一些场景，是把 Linux 作为一个软交换机或者路由器来用的。这种情况下，你更关注网络包的处理能力（即 PPS），重点关注网络层的转发性能。

接下来，我就带你从下往上，了解不同协议层的网络性能测试方法。不过要注意，低层协议是其上的各层网络协议的基础。自然，低层协议的性能，也就决定了高层的网络性能。

注意，以下所有的测试方法，都需要两台 Linux 虚拟机。其中一台，可以当作待测试的目标机器；而另一台，则可以当作正在运行网络服务的客户端，用来运行测试工具。

各协议层的性能测试

转发性能

我们首先来看，网络接口层和网络层，它们主要负责网络包的封装、寻址、路由以及发送和接收。在这两个网络协议层中，每秒可处理的网络包数 PPS，就是最重要的性能指标。特别是 64B 小包的处理能力，值得我们特别关注。那么，如何来测试网络包的处理能力呢？

说到网络包相关的测试，你可能会觉得陌生。不过，其实在专栏开头的 CPU 性能篇中，我们就接触过一个相关工具，也就是软中断案例中的 hping3。

在那个案例中，hping3 作为一个 SYN 攻击的工具来使用。实际上，hping3 更多的用途，是作为一个测试网络包处理能力的性能工具。

今天我再来介绍另一个更常用的工具，Linux 内核自带的高性能网络测试工具 [pktgen](#)。pktgen 支持丰富的自定义选项，方便你根据实际需要构造所需网络包，从而更准确地测试出目标服务器的性能。

不过，在 Linux 系统中，你并不能直接找到 pktgen 命令。因为 pktgen 作为一个内核线程来运行，需要你加载 pktgen 内核模块后，再通过 /proc 文件系统来交互。下面就是 pktgen 启动的两个内核线程和 /proc 文件系统的交互文件：

 复制代码

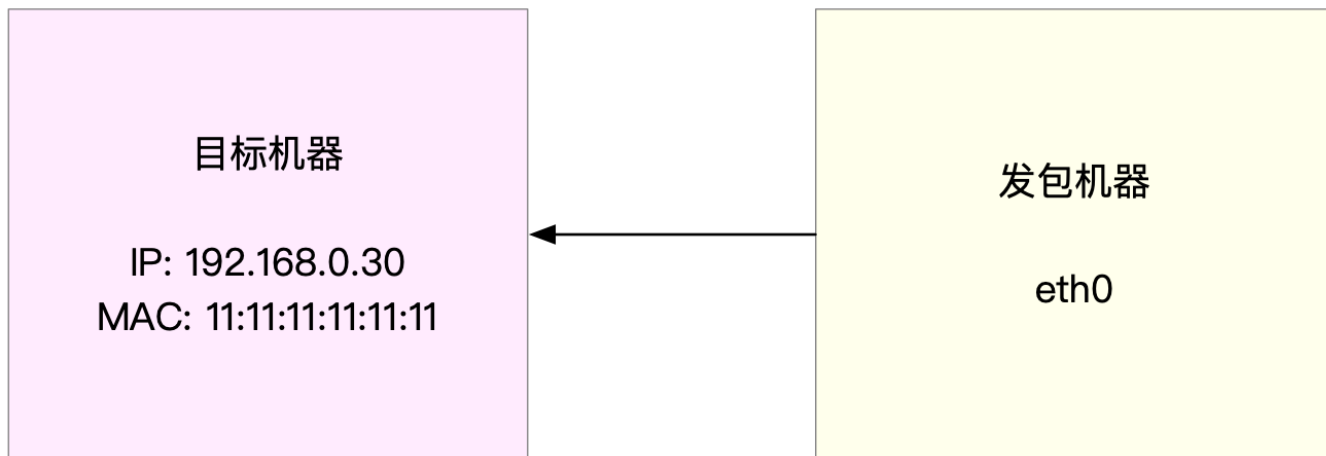
```
1 $ modprobe pktgen
2 $ ps -ef | grep pktgen | grep -v grep
3 root      26384      2  0 06:17 ?        00:00:00 [kpktgend_0]
4 root      26385      2  0 06:17 ?        00:00:00 [kpktgend_1]
5 $ ls /proc/net/pktgen/
6 kpktgend_0  kpktgend_1  pgctrl
```

pktgen 在每个 CPU 上启动一个内核线程，并可以通过 /proc/net/pktgen 下面的同名文件，跟这些线程交互；而 pgctrl 则主要用来控制这次测试的开启和停止。


如果 modprobe 命令执行失败，说明你的内核没有配置 CONFIG_NET_PKTGEN 选项。这就需要你配置 pktgen 内核模块（即 CONFIG_NET_PKTGEN=m）后，重新编译内核，才可以使用。

在使用 pktgen 测试网络性能时，需要先给每个内核线程 kpktgend_X 以及测试网卡，配置 pktgen 选项，然后再通过 pgctrl 启动测试。

以发包测试为例，假设发包机器使用的网卡是 eth0，而目标机器的 IP 地址为 192.168.0.30，MAC 地址为 11:11:11:11:11:11。



接下来，就是一个发包测试的示例。

 复制代码


```
1 # 定义一个工具函数，方便后面配置各种测试选项
2 function pgset() {
3     local result
4     echo $1 > $PGDEV
5
6     result=`cat $PGDEV | fgrep "Result: OK:"`
7     if [ "$result" = "" ]; then
8         cat $PGDEV | fgrep Result:
9     fi
10 }
11
12 # 为 0 号线程绑定 eth0 网卡
13 PGDEV=/proc/net/pktgen/kpktgend_0
14 pgset "rem_device_all" # 清空网卡绑定
15 pgset "add device eth0" # 添加 eth0 网卡
```

```

16
17 # 配置 eth0 网卡的测试选项
18 PGDEV=/proc/net/pktgen/eth0
19 pgset "count 1000000"      # 总发包数量
20 pgset "delay 5000"         # 不同包之间的发送延迟（单位纳秒）
21 pgset "clone_skb 0"        # SKB 包复制
22 pgset "pkt_size 64"         # 网络包大小
23 pgset "dst 192.168.0.30"    # 目的 IP
24 pgset "dst_mac 11:11:11:11:11:11" # 目的 MAC
25
26 # 启动测试
27 PGDEV=/proc/net/pktgen/pgctrl
28 pgset "start"

```

稍等一会儿，测试完成后，结果可以从 /proc 文件系统中获取。通过下面代码段中的内容，我们可以查看刚才的测试报告：

 复制代码

```

1 $ cat /proc/net/pktgen/eth0
2 Params: count 1000000 min_pkt_size: 64 max_pkt_size: 64
3     frags: 0 delay: 0 clone_skb: 0 ifname: eth0
4     flows: 0 flowlen: 0
5 ...
6 Current:
7     pkts-sofar: 1000000 errors: 0
8     started: 1534853256071us stopped: 1534861576098us idle: 70673us
9 ...
10 Result: OK: 8320027(c8249354+d70673) usec, 1000000 (64byte,0frags)
11     120191pps 61Mb/sec (61537792bps) errors: 0

```

你可以看到，测试报告主要分为三个部分：

第一部分的 Params 是测试选项；

第二部分的 Current 是测试进度，其中，packets so far (pkts-sofar) 表示已经发送了 100 万个包，也就表明测试已完成。

第三部分的 Result 是测试结果，包含测试所用时间、网络包数量和分片、PPS、吞吐量以及错误数。

根据上面的结果，我们发现，PPS 为 12 万，吞吐量为 61 Mb/s，没有发生错误。那么，12 万的 PPS 好不好呢？

作为对比，你可以计算一下千兆交换机的 PPS。交换机可以达到线速（满负载时，无差错转发），它的 PPS 就是 1000Mbit 除以以太网帧的大小，即 $1000\text{Mbps}/((64+20)*8\text{bit}) = 1.5\text{ Mpps}$ （其中 20B 为以太网帧的头部大小）。

你看，即使是千兆交换机的 PPS，也可以达到 150 万 PPS，比我们测试得到的 12 万大多了。所以，看到这个数值你并不担心，现在的多核服务器和万兆网卡已经很普遍了，稍做优化就可以达到数百万的 PPS。而且，如果你用了上节课讲到的 DPDK 或 XDP，还能达到千万数量级。


TCP/UDP 性能

掌握了 PPS 的测试方法，接下来，我们再来看 TCP 和 UDP 的性能测试方法。说到 TCP 和 UDP 的测试，我想你已经很熟悉了，甚至可能一下子就能想到相应的测试工具，比如 iperf 或者 netperf。

特别是现在的云计算时代，在你刚拿到一批虚拟机时，首先要做的，应该就是用 iperf，测试一下网络性能是否符合预期。

iperf 和 netperf 都是最常用的网络性能测试工具，测试 TCP 和 UDP 的吞吐量。它们都以客户端和服务端通信的方式，测试一段时间内的平均吞吐量。

接下来，我们就以 iperf 为例，看一下 TCP 性能的测试方法。目前，iperf 的最新版本为 iperf3，你可以运行下面的命令来安装：

 复制代码


```
1 # Ubuntu
2 apt-get install iperf3
3 # CentOS
4 yum install iperf3
```

然后，在目标机器上启动 iperf 服务端：

 复制代码


```
1 # -s 表示启动服务端，-i 表示汇报间隔，-p 表示监听端口
2 $ iperf3 -s -i 1 -p 10000
```

接着，在另一台机器上运行 iperf 客户端，运行测试：

 复制代码

```
1 # -c 表示启动客户端，192.168.0.30 为目标服务器的 IP
2 # -b 表示目标带宽（单位是 bits/s）
3 # -t 表示测试时间
4 # -P 表示并发数，-p 表示目标服务器监听端口
5 $ iperf3 -c 192.168.0.30 -b 1G -t 15 -P 2 -p 10000
```

稍等一会儿（15 秒）测试结束后，回到目标服务器，查看 iperf 的报告：

 复制代码

```
1 [ ID] Interval           Transfer     Bandwidth
2 ...
3 [SUM]   0.00-15.04  sec   0.00 Bytes   0.00 bits/sec             sender
4 [SUM]   0.00-15.04  sec  1.51 GBytes  860 Mbits/sec            receiver
```

最后的 SUM 行就是测试的汇总结果，包括测试时间、数据传输量以及带宽等。按照发送和接收，这一部分又分为了 sender 和 receiver 两行。


从测试结果你可以看到，这台机器 TCP 接收的带宽（吞吐量）为 860 Mb/s，跟目标的 1Gb/s 相比，还是有些差距的。

HTTP 性能

从传输层再往上，到了应用层。有的应用程序，会直接基于 TCP 或 UDP 构建服务。当然，也有大量的应用，基于应用层的协议来构建服务，HTTP 就是最常用的一个应用层协议。比如，常用的 Apache、Nginx 等各种 Web 服务，都是基于 HTTP。

要测试 HTTP 的性能，也有大量的工具可以使用，比如 ab、webbench 等，都是常用的 HTTP 压力测试工具。其中，ab 是 Apache 自带的 HTTP 压测工具，主要测试 HTTP 服务的每秒请求数、请求延迟、吞吐量以及请求延迟的分布情况等。

运行下面的命令，你就可以安装 ab 工具：


 复制代码

```
1 # Ubuntu
```




```
1 # Ubuntu
2 $ apt-get install -y apache2-utils
3 # CentOS
4 $ yum install -y httpd-tools
```

接下来，在目标机器上，使用 Docker 启动一个 Nginx 服务，然后用 ab 来测试它的性能。首先，在目标机器上运行下面的命令：

 复制代码

```
1 $ docker run -p 80:80 -itd nginx
```

而在另一台机器上，运行 ab 命令，测试 Nginx 的性能：

 复制代码

```
1 # -c 表示并发请求数为 1000，-n 表示总的请求数为 10000
2 $ ab -c 1000 -n 10000 http://192.168.0.30/
3 ...
4 Server Software:      nginx/1.15.8
5 Server Hostname:      192.168.0.30
6 Server Port:          80
7
8 ...
9
10 Requests per second:  1078.54 [#/sec] (mean)
11 Time per request:     927.183 [ms] (mean)
12 Time per request:     0.927 [ms] (mean, across all concurrent requests)
13 Transfer rate:        890.00 [Kbytes/sec] received
14
15 Connection Times (ms)
16      min  mean[+/-sd] median   max
17 Connect:    0   27 152.1      1   1038
18 Processing:  9   207 843.0     22   9242
19 Waiting:    8   207 843.0     22   9242
20 Total:      15   233 857.7     23   9268
21
22 Percentage of the requests served within a certain time (ms)
23   50%    23
24   66%    24
25   75%    24
26   80%    26
27   90%   274
28   95%  1195
29   98% 2335
30
31   99%  4663
32  100% 9268 (longest request)
```

可以看到，ab 的测试结果分为三个部分，分别是请求汇总、连接时间汇总还有请求延迟汇总。以上面的结果为例，我们具体来看。

在请求汇总部分，你可以看到：

Requests per second 为 1074;

每个请求的延迟 (Time per request) 分为两行，第一行的 927 ms 表示平均延迟，包括了线程运行的调度时间和网络请求响应时间，而下一行的 0.927ms，则表示实际请求的响应时间；

Transfer rate 表示吞吐量 (BPS) 为 890 KB/s。

连接时间汇总部分，则是分别展示了建立连接、请求、等待以及汇总等的各类时间，包括最小、最大、平均以及中值处理时间。

最后的请求延迟汇总部分，则给出了不同时间段内处理请求的百分比，比如，90% 的请求，都可以在 274ms 内完成。

应用负载性能


当你用 iperf 或者 ab 等测试工具，得到 TCP、HTTP 等的性能数据后，这些数据是否就能表示应用程序的实际性能呢？我想，你的答案应该是否定的。

比如，你的应用程序基于 HTTP 协议，为最终用户提供一个 Web 服务。这时，使用 ab 工具，可以得到某个页面的访问性能，但这个结果跟用户的实际请求，很可能不一致。因为用户请求往往会附带着各种各样的负载 (payload)，而这些负载会影响 Web 应用程序内部的处理逻辑，从而影响最终性能。

那么，为了得到应用程序的实际性能，就要求性能工具本身可以模拟用户的请求负载，而 iperf、ab 这类工具就无能为力了。幸运的是，我们还可以用 wrk、TCPCopy、Jmeter 或者 LoadRunner 等实现这个目标。


以 [wrk](#) 为例，它是一个 HTTP 性能测试工具，内置了 LuaJIT，方便你根据实际需求，生成所需的请求负载，或者自定义响应的处理方法。

wrk 工具本身不提供 yum 或 apt 的安装方法，需要通过源码编译来安装。比如，你可以运行下面的命令，来编译和安装 wrk：

 复制代码

```
1 $ https://github.com/wg/wrk
2 $ cd wrk
3 $ apt-get install build-essential -y
4 $ make
5 $ sudo cp wrk /usr/local/bin/
```

wrk 的命令行参数比较简单。比如，我们可以用 wrk，来重新测一下前面已经启动的 Nginx 的性能。

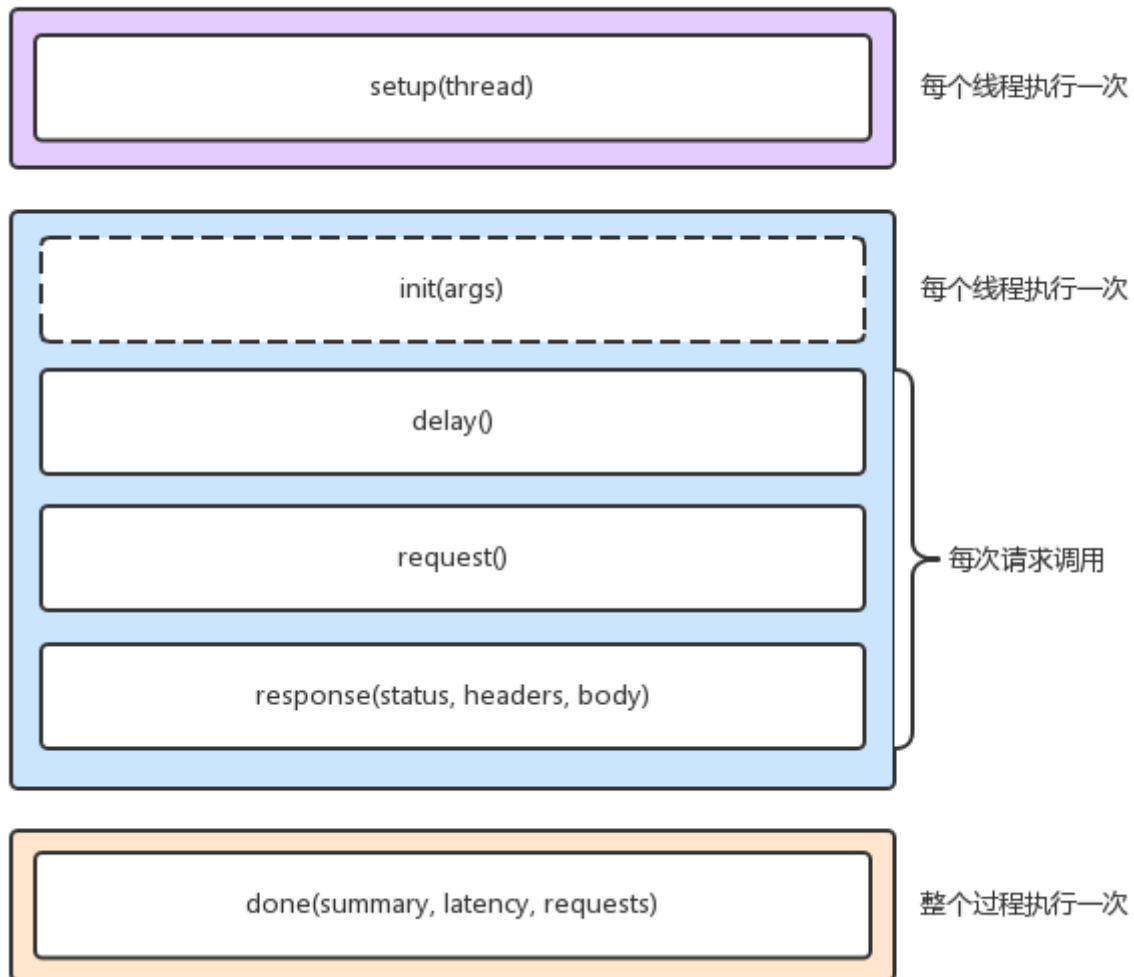
 复制代码

```
1 # -c 表示并发连接数 1000，-t 表示线程数为 2
2 $ wrk -c 1000 -t 2 http://192.168.0.30/
3 Running 10s test @ http://192.168.0.30/
4 2 threads and 1000 connections
5 Thread Stats   Avg      Stdev     Max   +/-  Stdev
6   Latency    65.83ms  174.06ms   1.99s   95.85%
7   Req/Sec    4.87k    628.73    6.78k   69.00%
8 96954 requests in 10.06s, 78.59MB read
9 Socket errors: connect 0, read 0, write 0, timeout 179
10 Requests/sec: 9641.31
11 Transfer/sec: 7.82MB
```

这里使用 2 个线程、并发 1000 连接，重新测试了 Nginx 的性能。你可以看到，每秒请求数为 9641，吞吐量为 7.82MB，平均延迟为 65ms，比前面 ab 的测试结果要好很多。

这也说明，性能工具本身的性能，对性能测试也是至关重要的。不合适的性能工具，并不能准确测出应用程序的最佳性能。

当然，wrk 最大的优势，是其内置的 LuaJIT，可以用来实现复杂场景的性能测试。wrk 在调用 Lua 脚本时，可以将 HTTP 请求分为三个阶段，即 setup、running、done，如下图所示：



(图片来自[网易云博客](#))


比如，你可以在 `setup` 阶段，为请求设置认证参数（来自于 wrk 官方[示例](#)）：

复制代码

```
1 -- example script that demonstrates response handling and
2 -- retrieving an authentication token to set on all future
3 -- requests
4
5 token = nil
6 path  = "/authenticate"
7
8 request = function()
9     return wrk.format("GET", path)
10 end
11
12 response = function(status, headers, body)
13     if not token and status == 200 then
14         token = headers["X-Token"]
15         path  = "/resource"
16         wrk.headers["X-Token"] = token
17     end
18 end
```

18 end

而在执行测试时，通过 `-s` 选项，执行脚本的路径：

 复制代码

```
1 $ wrk -c 1000 -t 2 -s auth.lua http://192.168.0.30/
```

`wrk` 需要你用 Lua 脚本，来构造请求负载。这对于大部分场景来说，可能已经足够了。不过，它的缺点也正是，所有东西都需要代码来构造，并且工具本身不提供 GUI 环境。

像 Jmeter 或者 LoadRunner（商业产品），则针对复杂场景提供了脚本录制、回放、GUI 等更丰富的功能，使用起来也更加方便。

小结

今天，我带你一起回顾了网络的性能指标，并学习了网络性能的评估方法。

性能评估是优化网络性能的前提，只有在你发现网络性能瓶颈时，才需要进行网络性能优化。根据 TCP/IP 协议栈的原理，不同协议层关注的性能重点不完全一样，也就对应不同的性能测试方法。比如，

在应用层，你可以使用 `wrk`、Jmeter 等模拟用户的负载，测试应用程序的每秒请求数、处理延迟、错误数等；

而在传输层，则可以使用 `iperf` 等工具，测试 TCP 的吞吐情况；

再向下，你还可以用 Linux 内核自带的 `pktgen`，测试服务器的 PPS。

由于低层协议是高层协议的基础。所以，一般情况下，我们需要从上到下，对每个协议层进行性能测试，然后根据性能测试的结果，结合 Linux 网络协议栈的原理，找出导致性能瓶颈的根源，进而优化网络性能。

思考

最后，我想请你来聊一聊。

你是如何评估网络性能的？

在评估网络性能时，你会从哪个协议层、选择哪些指标，作为性能测试最核心的目标？

你又会用哪些工具，测试并分析网络的性能呢？

你可以结合今天学到的网络知识，总结自己的思路。

欢迎在留言区和我讨论，也欢迎你把这篇文章分享给你的同事、朋友。我们一起在实战中演练，在交流中进步。



Linux 性能优化实战

10 分钟帮你找到系统瓶颈

倪朋飞

微软资深工程师
Kubernetes 项目维护者



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得转载

上一篇 35 | 基础篇：C10K 和 C1000K 回顾

下一篇 37 | 案例篇：DNS 解析时快时慢，我该怎么办？

精选留言 (5)

写留言



我来也

2019-02-13

1

[D36打卡]

想不到网络篇这么快就开始"套路"了 😊

由于现在客户端的网络环境复杂,经常会出现部分用户反馈卡顿的情况.

我这边能做的也有限.

只能在第一个网路出入口,记录每次收发消息的内容和具体的时间戳(精确到ms)....

展开 ▾

作者回复: 嗯, 网络抖动是很常见的现象。可以考虑更多的接入点、专线、CDN 等等都可以优化公网的链路延迟问题



Leon 📷

2019-02-15



老师。我前几天测试局域网三台机器高并发修改了参数
/etc/security/limits.conf

* soft nofile 102400

* hard nofile 102400

/etc/sysctl.conf...

展开 ▾



王崧霁

2019-02-14



做全链路监控



black_mirror

2019-02-14



倪老师

请问linux系统下怎么查看进程的每个线程占用多少内存那? top -Hp 看到线程内存与进程一样, 看起来它们是共享的



Enterprize

2019-02-13



有时会遇到偶然性的api请求响应慢, 这种问题排查的思路是怎样的呢, 怎么确定是网络抖动还是服务器配置就有问题?

展开 ▾

作者回复: 如果是现象还在的话，抓个包就可以看出来。不过，最好还是加上各层的延迟监控，这样可以看到历史的情况

