详解压测中出现的 nginx 502 no live upstreams 错误

xnathan.com/2019/05/07/nginx-502

2019年5月7日

在一次压测过程中，发现随着并发用户量的增加，压测客户端收到错误请求越来越多，Nginx 返回大量 `502 Bad Gateway` 错误。

以此次压测为契机，让我们有机会探讨高并发环境下可能出现的问题，本文借助 nginx 和 Linux 内核源码，分析产生 502 错误码的原因，并提出相应解决办法，为今后解决类似问题提供思路和参考。

背景和现象

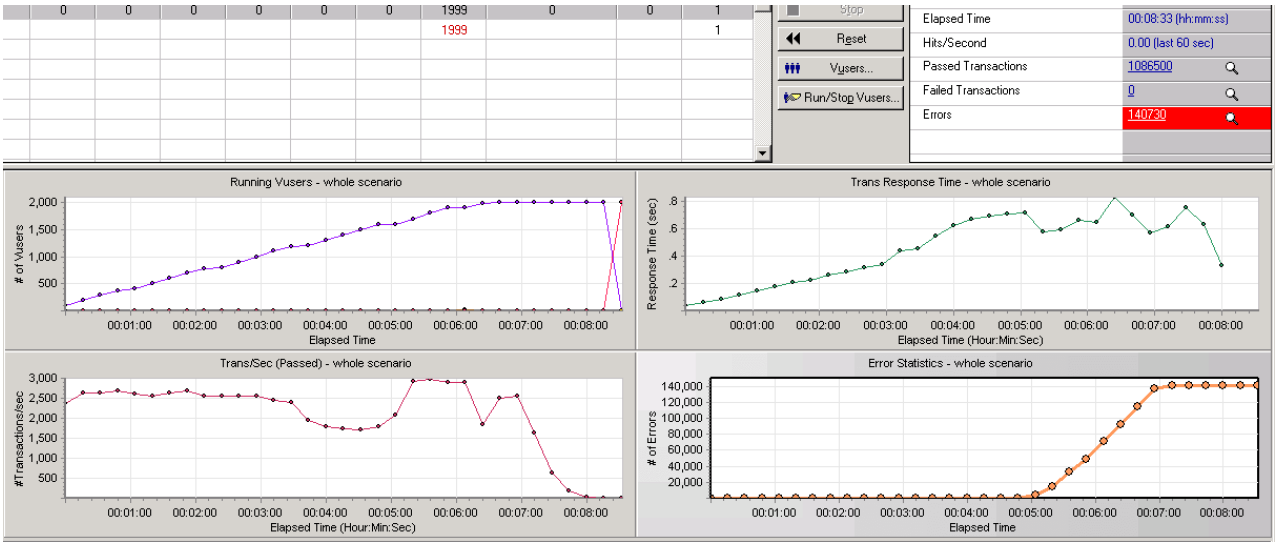
项目部署在三台腾讯云服务器上，其中两台部署了 web 服务，运行在 docker 里，另一台在宿主机上部署了 Nginx，用来反代两台应用服务器。

机器配置：

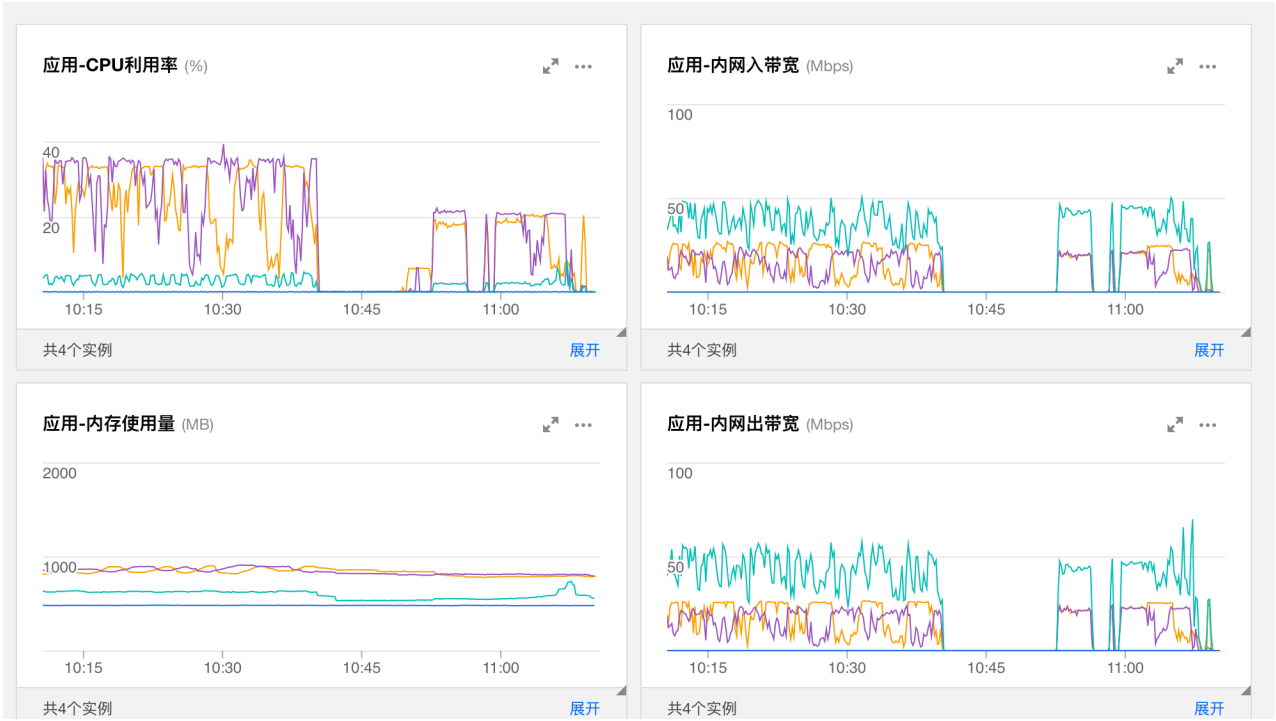
> 系统： CentOS 7.5.1804
>
> CPU: 2x8 Core 2.4 GHz
>
> 内存：32G
>
> 硬盘：50G + 500G

压测机在同一网段的另一台机器上，压测启动时，只压部署 Nginx 的机器。

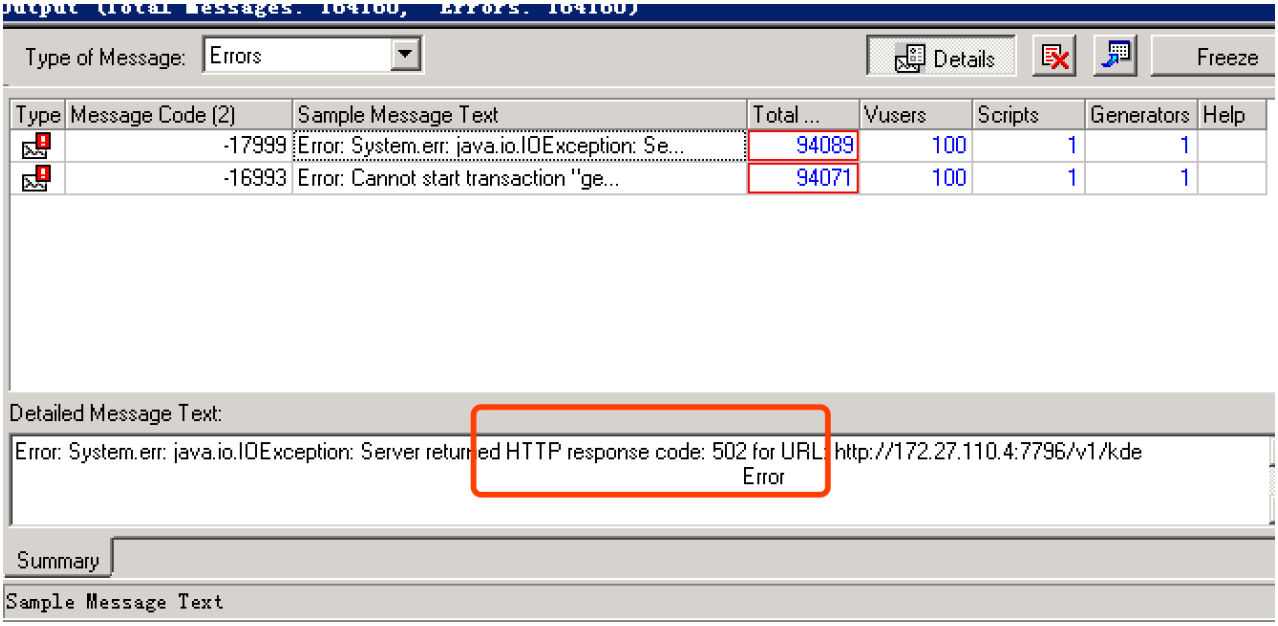
从后台日志看，预测服务的响应时间大概在 2ms 左右，平稳的 TPS 在 2600 左右，压测一段时间后，错误率开始上升，TPS 发生抖动。



观察服务器后台监控，发现应用服务器的 CPU 和网络带宽在这一段时间内也有较大波动。



查看压测客户端收到的错误，基本都是 502。



本地复现

由于线上服务器还需要提供服务，也没有安装调试工具，只能想办法在本地搭一个类似的环境。

经过实验，发现在本地用 docker 模拟运行环境，jmeter 作为压测工具，也可以复现 502 错误。

程序代码已经放到了 nginx_502_debug 项目 中。

1. Web 应用

应用服务器依然选用 tornado，运行一个 `hello world` 程序：

```
# webserver/app.py

import time
import tornado.gen
import tornado.ioloop
import tornado.web
import tornado.httpserver

class
MainHandler(tornado.web.RequestHandler):
    def get(self):
        # Simulate a slow web server
        time.sleep(0.02)
        self.write("Hello, world")

def make_app():
    return tornado.web.Application([
        (r"/", MainHandler),
    ])

if __name__ == "__main__":
    app = make_app()
    server =
tornado.httpserver.HTTPServer(app)
    server.bind(8888)
    server.start()

tornado.ioloop.IOLoop.current().start()
```

2. Nginx 配置

Nginx 的关键配置保持和线上压测使用的一致：

```
# nginx.conf

user  nginx;
worker_processes  8;

error_log  /var/log/nginx/error.log;
pid        /var/run/nginx.pid;

events {
    worker_connections  30000;
}


http {
    include       /etc/nginx/mime.types;
    default_type  application/octet-stream;

    log_format  main  '$remote_addr - $remote_user [$time_local]
"$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile        on;
    tcp_nopush     on;
    tcp_nodelay on;

    keepalive_timeout  65;

        upstream s1 {
                server app1:8888;
                server app2:8888;
        }

        server {
                listen 8888;
                location / {
                        proxy_pass http://s1;
                }
        }

    #gzip  on;

    include /etc/nginx/conf.d/*.conf;
}
```

设置 `worker_connections` 为一个比较大的数，是为了让单个 nginx worker 能处理更多的连接。

在 `http` 段中增加 server 监听 8888 端口，并且反代 app1 和 app2 两个 host 8888 端口。

3. docker compose 配置

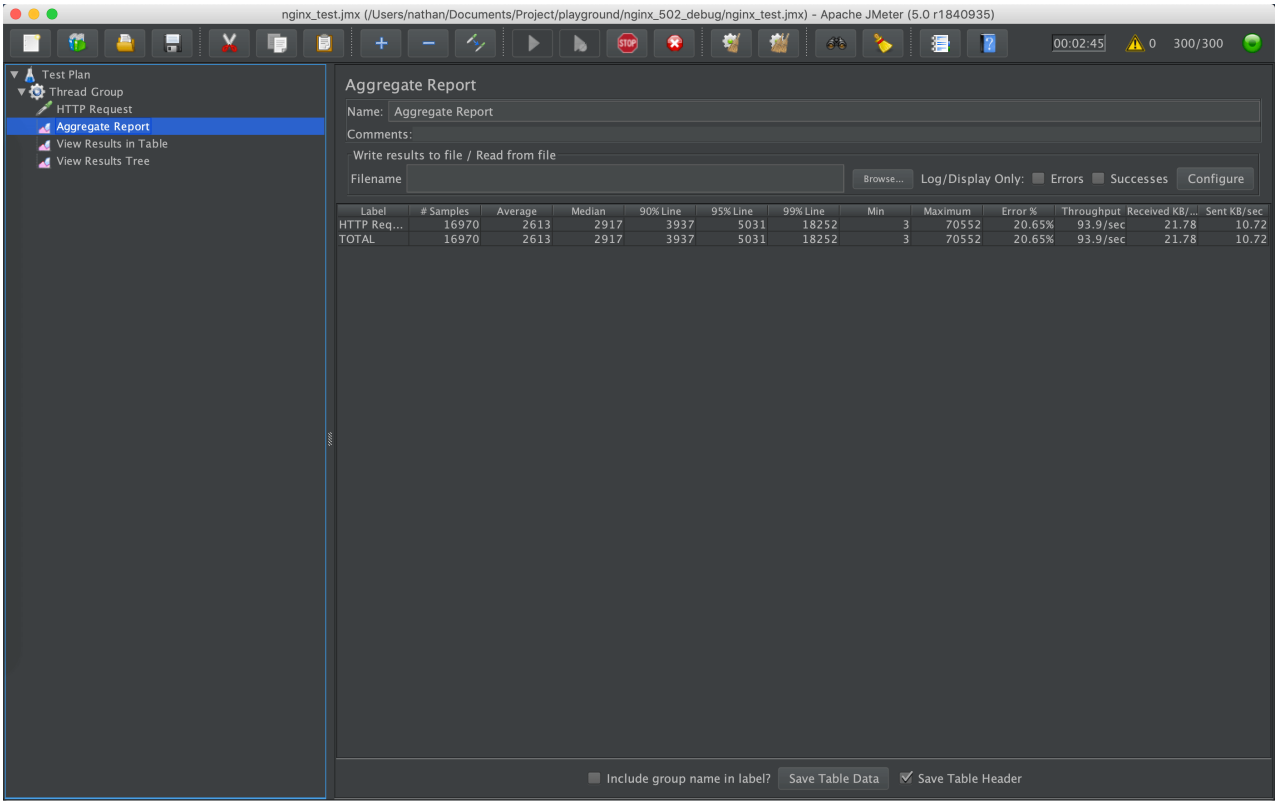
相应的 docker compose 文件：

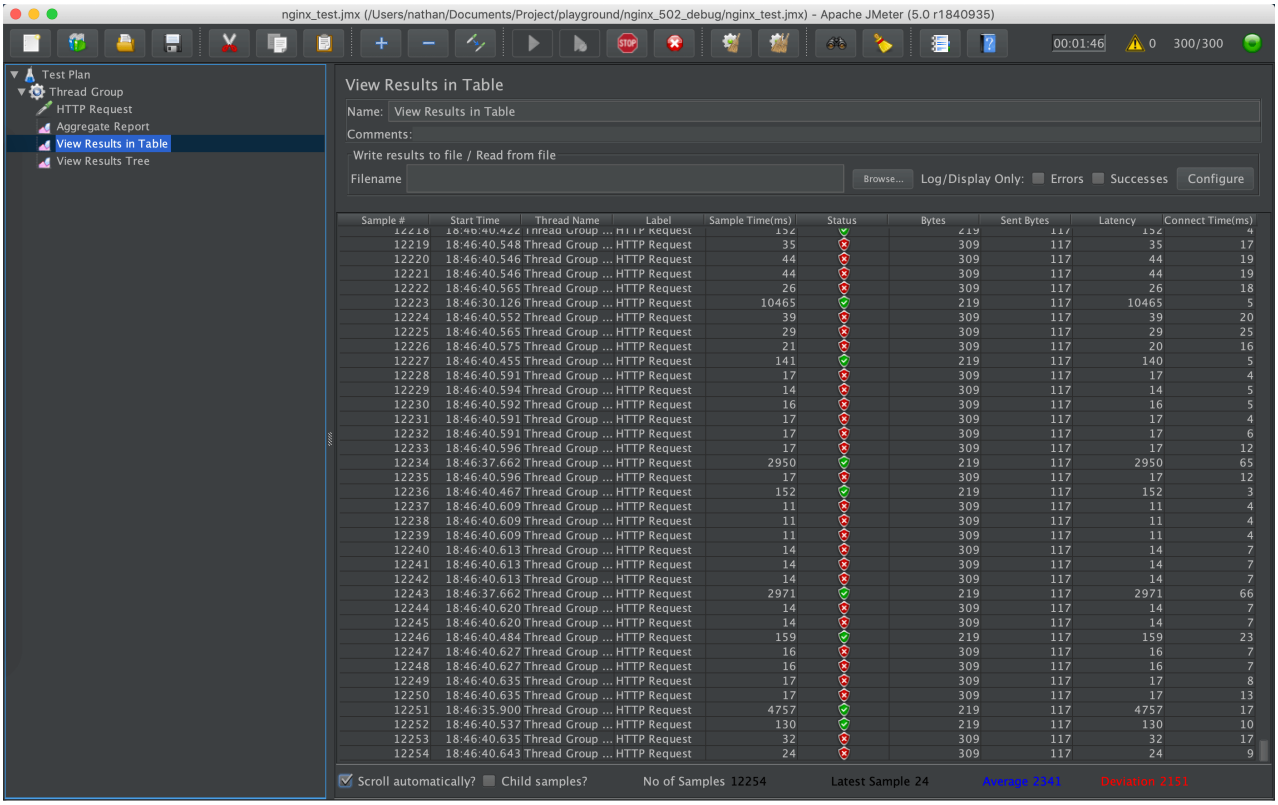
```
version: "3"

services:
  app1:
    build:
      context: ./webserver

  app2:
    build:
      context: ./webserver

  nginx:
    image: nginx
    volumes:
      -
./nginx.conf:/etc/nginx/nginx.conf
    ports:
      - 8888:8888
    links:
      - app1
      - app2
```

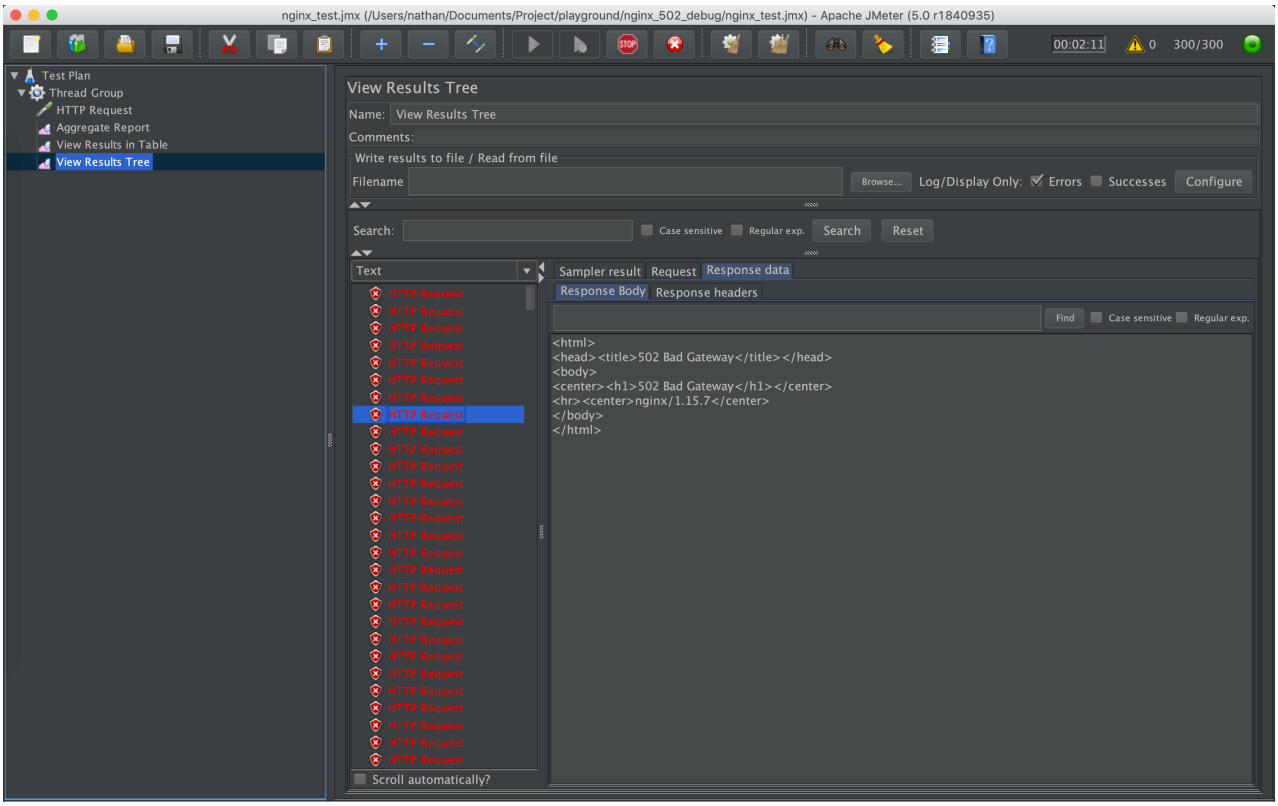
运行 `docker-compose up` 启动服务。

打开 jmeter 软件，打开 `nginx_test.jmx` 压测脚本，点击开始按钮。

几分钟后，`View Result Tree` 中出现错误响应，返回码是 502，同时 Nginx 日志中出现大量 `[error] 2960#0: *2536622 no live upstreams while connecting to upstream` 错误。观察应用程序，CPU 正常，依然有请求，应用程序也没有停止运行，和线上压测的现象一致。







多运行一会儿，Nginx 日志也可能出现以下两个错误：

```
[error] 7#7: *22642 recv() failed (104: Connection reset by peer) while
reading response header from upstream
[error] 7#7: *98207 upstream timed out (110: Connection timed out) while
reading response header from upstream
```

至此，已经模拟出当时压测的情况，并且得到了与线上一致的运行结果。

原因分析

根据报错判断主要原因是 Nginx 没有从上游服务器得到有效的返回结果，可能的原因有以下几种：

1. 后端服务挂了
2. 后端应用性能达到瓶颈，比如 CPU、内存跑满，带宽跑满等
3. 后端应用响应超时
4. 防火墙配置问题
5. HTTP 头部过大
6. DNS 配置问题

根据压测的现象，判断问题出在后端应用上，但是后端应用并没挂，CPU 内存没满，也没有任何报错信息。

只能先从 Nginx 源码入手，搜索 `no live upstream` 错误。

Nginx 源码分析

线上 nginx 版本是 1.12.2，下载对应版本源码。

```
$ wget https://github.com/nginx/nginx/archive/release-1.12.2.tar.gz
$ tar xzf release-1.12.2.tar.gz
$ cd nginx-release-1.12.2

# 用 rg (或者 grep、ack、ag，推荐 rg) 搜索 `no live upstream`
$ rg "no live upstream"
# src/stream/ngx_stream_proxy_module.c
# 696:        ngx_log_error(NGX_LOG_ERR, c->log, 0, "no live upstreams");
#
# src/http/ngx_http_upstream.c
# 1499:        ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "no live
upstreams");
#
# docs/xml/nginx/changes.xml
# 7254:answer from cache if there were no live upstreams.
```

出错代码主要和 http 相关，打开 `src/http/ngx_http_upstream.c` 文件，1499 行左右代码如下：

```
if (rc == NGX_BUSY) {
    ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, "no live
upstreams");
    ngx_http_upstream_next(r, u, NGX_HTTP_UPSTREAM_FT_NOLIVE);
    return;
}
```

往上翻找到 `rc` 定义：

```
// src/http/ngx_http_upstream.c, Line:
1485
rc = ngx_event_connect_peer(&u->peer);
```

可以看出，只要 `rc` 返回 `NGX_BUSY` 状态，就会记录一条 `no live upstreams` 信息。

`ngx_event_connect_peer` 函数在 `src/event/ngx_event_connect.c` 中实现，搜索整个文件没有发现 `NGX_BUSY` ，只能是以下这段代码返回的结果。

```
// src/event/ngx_event_connect.c, Line
34-37
rc = pc->get(pc, pc->data);
if (rc != NGX_OK) {
    return rc;
}
```

根据轮询规则的不同， `pc->get` 实际指向的函数也不一样。默认为 round robin 方式， `pc-get` 指向 `ngx_http_upstream_get_round_robin_peer` 函数。

```
// src/http/ngx_http_upstream_round_robin.c, Line 435-
461
if (peers->single) {
    peer = peers->peer;

    if (peer->down) {
        goto failed;
    }

    if (peer->max_conns && peer->conns >= peer-
>max_conns) {
        goto failed;
    }

    rrp->current = peer;

} else {

    /* there are several peers */

    peer = ngx_http_upstream_get_peer(rrp);

    if (peer == NULL) {
        goto failed;
    }

    ngx_log_debug2(NGX_LOG_DEBUG_HTTP, pc->log, 0,
                   "get rr peer, current: %p %i",
                   peer, peer->current_weight);
}
```

如果设置的 upstream peers 只有一个，那么如果这个 peer down 掉或者超过最大连接数，那么直接进入 `failed`。 如果有多个 upstream，先会通过 `ngx_http_upstream_get_peer` 找到最合适的 peer，这个函数具体实现代码就不贴了，主要过程就是判断 peer 是否存活，是否在重试次数限制内，根据 weight 找到最好的 peer 返回，如果没有满足条件的 peer，就返回 NULL。

所以，如果判断没有满足条件的 peer，也会进入 `failed`。

`failed` 相关源码：

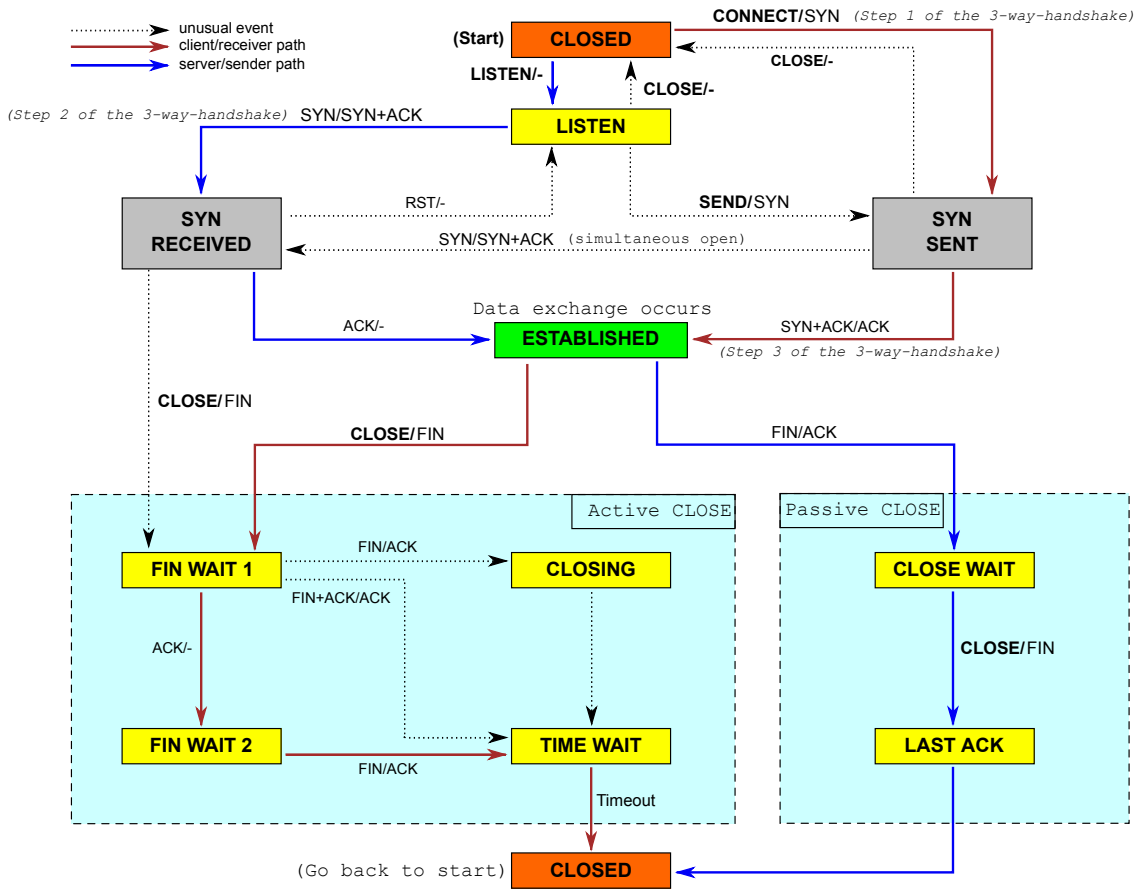
```
// src/http/ngx_http_upstream_round_robin.c, Line: 473-503
failed:

    if (peers->next) {

        ngx_log_debug0(NGX_LOG_DEBUG_HTTP, pc->log, 0, "backup
servers");

        rrp->peers = peers->next;

        n = (rrp->peers->number + (8 * sizeof(uintptr_t) - 1))
                / (8 * sizeof(uintptr_t));

        for (i = 0; i < n; i++) {
            rrp->tried[i] = 0;
        }

        ngx_http_upstream_rr_peers_unlock(peers);

        rc = ngx_http_upstream_get_round_robin_peer(pc, rrp);

        if (rc != NGX_BUSY) {
            return rc;
        }

        ngx_http_upstream_rr_peers_wlock(peers);
    }

    ngx_http_upstream_rr_peers_unlock(peers);

    pc->name = peers->name;

    return NGX_BUSY;
```

在这终于找到了 `NGX_BUSY`。进入 `failed` 后，先判断有没有 backup server，如果有的话先尝试连接 backup server，连接成功后返回，连接不成功则执行到最后，返回 `NGX_BUSY`。

Nginx 源码分析结论： Nginx 先尝试连接 upstream 中的 peer，如果 peer 挂了，或者一段时间内的失败次数超过限制，那么没有 peer 可以连接，记录一条 `no live upstreams` 日志，接着调用 `ngx_http_upstream_next` 重试连接。

可是什么情况会让 nginx 认为没有可用的 upstream 服务器了呢？从 Linux 内核源码和 TCP/IP 实现原理中找到了答案。

Linux TCP 原理分析



后端应用启动后，进入 `LISTEN` 状态，创建 socket 监听端口。

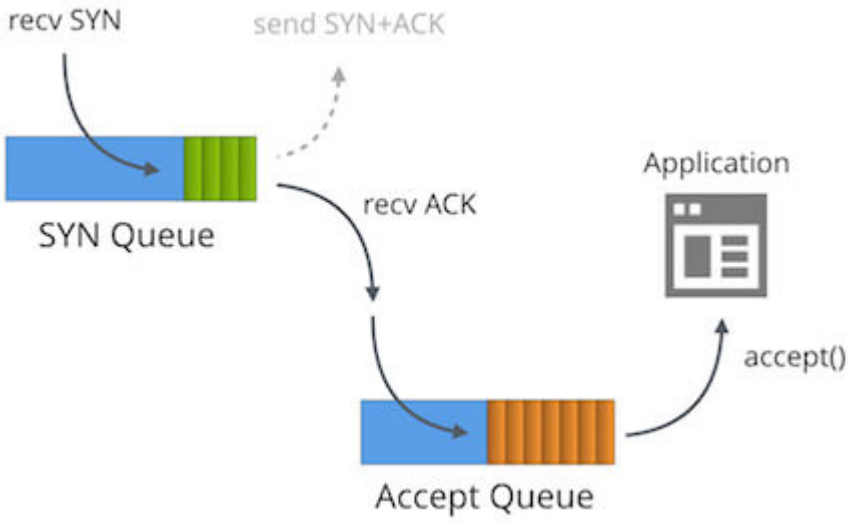
当客户端开始连接服务器，先进行 TCP 三次握手

1. 客户端发送 SYN 报文
2. 服务端收到 SYN 后响应 SYN/ACK，确认收到的 SYN，连接进入 `SYN RECEIVED` 状态
3. 客户端收到响应的 SYN/ACK，发送 ACK 报文，服务器收到之后，连接进入 `ESTABLISHED` 状态

经过三次握手，成功建立 TCP 连接，开始传输数据。

在 Linux 内核实现中，对于 `LISTEN` 状态的程序，会维护两个队列，一个叫半连接队列（incomplete connection queue），保存的是 `SYN RECEIVED` 状态的连接，另一个连接叫全连接队列（completed queue），保存的是完成三次握手，状态已经是 `ESTABLISHED` 的连接。

收到 SYN 报文后，先返回 SYN/ACK，把连接放到半连接队列中；如果收到客户端发来的 ACK 报文，再把连接从半连接队列中移到全连接队列中，等待应用程序通过 `accept(2)` 系统调用，把连接从队列中取出，开始数据传输。



既然是由队列保存连接，那么就会有队列长度限制，而且会存在队列满的情况。

全连接队列的长度大小由程序调用系统的 `listen(2)` 函数指定。

函数定义： `int listen(int sockfd, int backlog);` ， `backlog` 参数指定的就是这个程序的全连接队列大小。

实际的队列大小由传入的 `backlog` 以及内核 `net.core.somaxconn` 中较小者所确定， `net.core.somaxconn` 默认值为 128。

一般而言，全连接队列大部分时候为空，因为完成三次握手后的连接很快会被应用程序通过 `accept(2)` 函数取走。但是如果应用程序处理较慢，或者请求量太大，可能会造成全连接队列满的情况。

如果全连接队列已经满了，这时收到半连接队列中一个客户端发来的 ACK 报文，TCP 会把这个 ACK 忽略，或者返回 RST 报文，由 `net.ipv4.tcp_abort_on_overflow` 参数控制，默认为 0，设置为 1 则会返回 RST。

相关源码在 `net/ipv4/tcp_minisocks.c` 文件中的 `tcp_check_req` 函数：

> 以下源码以 Linux 4.19.12 为例

```
// net/ipv4/tcp_minisocks.c, Line: 786-789
child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb, req,
NULL,
                          req, &own_req);
if (!child)
    goto listen_overflow;
```

实际调用的函数是 `net/ipv4/tcp_ipv4.c` 中的 `tcp_v4_syn_recv_sock` ：

```
// net/ipv4/tcp_ipv4.c, Line: 1413-1418
if (sk_acceptq_is_full(sk))
        goto exit_overflow;

// include/net/sock.h, Line: 857-860
static inline bool sk_acceptq_is_full(const struct sock
*sk)
{
        return sk->sk_ack_backlog > sk-
>sk_max_ack_backlog;
}
```

先判断连接队列是否已满，如果满了，就增加 `/proc/net/netstat` 文件中的 `ListenOverflows` 和 `ListenDrops` 计数，返回 `NULL` 。

回到 `tcp_check_req` 函数，child 为 `NULL` ，跳转到 `listen_overflow` 中。

```
// net/ipv4/tcp_minisocks.c, Line 796-800
listen_overflow:
        if (!sock_net(sk)->ipv4.sysctl_tcp_abort_on_overflow) {
                inet_rsk(req)->acked = 1;
                return NULL;
        }

embryonic_reset:
        if (!(flg & TCP_FLAG_RST)) {
                /* Received a bad SYN pkt - for TFO We try not to
reset
                 * the local connection unless it's really
necessary to
                 * avoid becoming vulnerable to outside attack
aiming at
                 * resetting legit local connections.
                 */
                req->rsk_ops->send_reset(sk, skb);
        } else if (fastopen) { /* received a valid RST pkt */
                reqsk_fastopen_remove(sk, req, true);
                tcp_reset(sk);
        }
        if (!fastopen) {
                inet_csk_reqsk_queue_drop(sk, req);
```

```
                __NET_INC_STATS(sock_net(sk),
LINUX_MIB_EMBRYONICRSTS);
        }
        return NULL;
}
```

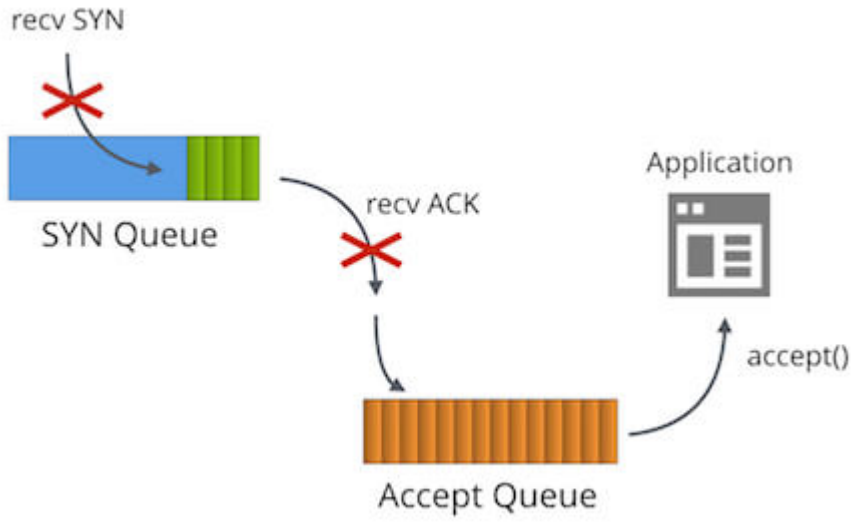
如果没设置 `net.ipv4.tcp_abort_on_overflow` ，那么只是把 acked 标志设为 1，直接返回 NULL，相当于忽略了客户端最后一个 ACK 报文；如果设置值为 1，代码会顺序执行下去，调用 `send_reset()` 给客户端返回 RST，客户端会出现 `Connection reset by peer` 之类的错误。

ACK 被忽略后，服务端会用二进制指数退避算法重传第二次握手时的 SYN/ACK，在客户端看来，ACK 包丢失，收到重发的 SYN/ACK 后也会重发 ACK，服务端重传 SYN/ACK 的次数由 `net.ipv4.tcp_synack_retries` 参数指定，默认为 5。

利用 TCP 的重传机制，可以让客户端等待，直到服务端程序有空闲资源接收并处理连接，或者最终超时。

以上暂只讨论了全连接队列满的情况，半连接队列也会满。

半连接队列长度由系统 `net.ipv4.tcp_max_syn_backlog` 参数控制，在没启用 `syncookies` 的情况下，超过限制的报文会被丢弃，如果启用了 `syncookies` ，理论上半连接队列没有长度限制。



如果连接队列和半连接队列都满了，系统会丢弃发来的 SYN 和 ACK 包，避免过度拥堵。

Linux 源码分析总结：

1. Linux 使用两个队列保存不同状态的 TCP 连接
2. 半连接队列长度由系统参数 `net.ipv4.tcp_max_syn_backlog` 控制，但如果启用了 `syncookies` 则不存在队列长度上限
3. 全连接队列由程序调用 `listen(2)` 时传入，最大可取 `net.core.somaxconn` 设置的值，内核为每个监听的端口维护不同的全连接队列
4. 如果连接队列满，系统会丢弃 ACK 报文，无法将连接从半连接队列中转移到连接队列，会发生重传，直到超时或者被应用程序 `accept()`

解决方法

经过以上的分析，找到了 nginx 发生 `no live upstreams` 错误的具体情况，以及 Linux 内核维护 TCP 连接时采用的队列机制和相关配置参数。接下来只要针对性地调整参数就能把错误率降低。

内核调优

修改 `/etc/sysctl.conf` ，调整 Nginx 宿主机的内核配置：

```
fs.file-max=1024000
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 0
net.ipv4.tcp_fin_timeout = 30
```

```
# max SYN backlog
net.ipv4.tcp_max_syn_backlog = 4096
# max timewait sockets held by system
simultaneously
net.ipv4.tcp_max_tw_buckets = 5000
# TCP receive buffer
net.ipv4.tcp_rmem = 4096 87380 67108864
# TCP write buffer
net.ipv4.tcp_wmem = 4096 65536 67108864
# turn on path MTU discovery
net.ipv4.tcp_mtu_probing = 1
# max read buffer
net.core.rmem_max = 67108864
# max write buffer
net.core.wmem_max = 67108864
# default read buffer
net.core.rmem_default = 65536
# default write buffer
net.core.wmem_default = 65536
# max processor input queue
net.core.netdev_max_backlog = 4096
# max backlog
net.core.somaxconn = 65535
```

运行应用程序 docker 的宿主上不用设置内核参数，因为针对宿主机的内核配置并不会对容器产生影响，它们属于不同的命名空间，网络栈也是隔离的，并且容器内部已经对某些内核参数做了优化，比如 `fs.file-max`、`net.ipv4.tcp_max_syn_backblog` 等，但是 `net.core.somaxconn` 还是默认值 `128`。

`net.core.somaxconn` 值变大不会对我们的 tornado 程序产生实际的影响，因为 tornado 启动时默认传的 `backlog` 参数就是 128。为使内核参数修改产生实际作用，需要在 tornado 的监听函数传入新的 `backlog` 值，以增加全连接队列长度。

如果需要设置容器使用的内核参数，可以在启动时加上 `--sysctl` 参数，例如 `docker run --sysctl net.core.somaxconn=4096 ...`。 需要注意的是目前只有一部分内核参数是支持修改的，具体参考 Docker 文档。

Nginx 配置

调试过程中发现，nginx 会和后端应用建立大量连接，导致系统中存在大量 `TIME_WAIT` 状态的连接，消耗服务器的端口资源。

参考 nignx 调优文档，发现让 nginx 和后端应用使用 keepalive 保持长连接即可，可以利用 nginx 连接池中已经创建好的连接，不用每个请求都创建新连接。

配置方法：

```
upstream s1 {
    keepalive 1000;

    server app1:8888;
    server app2:8888;
}

server {
    listen 8888;
    location / {
        proxy_pass http://s1;
        proxy_http_version 1.1;
        proxy_set_header
Connection "";
    }
}
```

让 nginx 和后端使用 http 1.1 协议，开启 keepalive 功能。

`proxy_http_version` 设置协议版本为 1.1，默认支持 `keepalive` ；

`proxy_set_header` 把 `Connection` 头清空，以免客户端发来的头部包含 `Connection: close` ，而意外地把长连接关闭；

`keepalive` 参数设置的是每个 nginx worker 可缓存的最大空闲连接数，不代表每个 worker 能创建的连接数。

比如说，如果一个 nginx worker 收到一个客户端请求，同时它又没有空闲的连接可用，就会和 upstream 创建一个新的长连接，数据传输完，这个连接变成空闲状态，等待下一个客户端请求，但是如果此时空闲状态的连接超过 `keepalive` 参数设置的值，最早创建的连

接就会被 nginx 主动关闭。

如果 `keepalive` 参数设置得过小，比如 16，高并发的时候就会同时创建许多长连接，同时又有很多连接传输结束，变为空闲状态，空闲状态的连接很容易就超过 16，使这些连接被关闭，导致 nginx 服务器出现大量 `TIME_WAIT` 状态的连接，消耗端口资源；

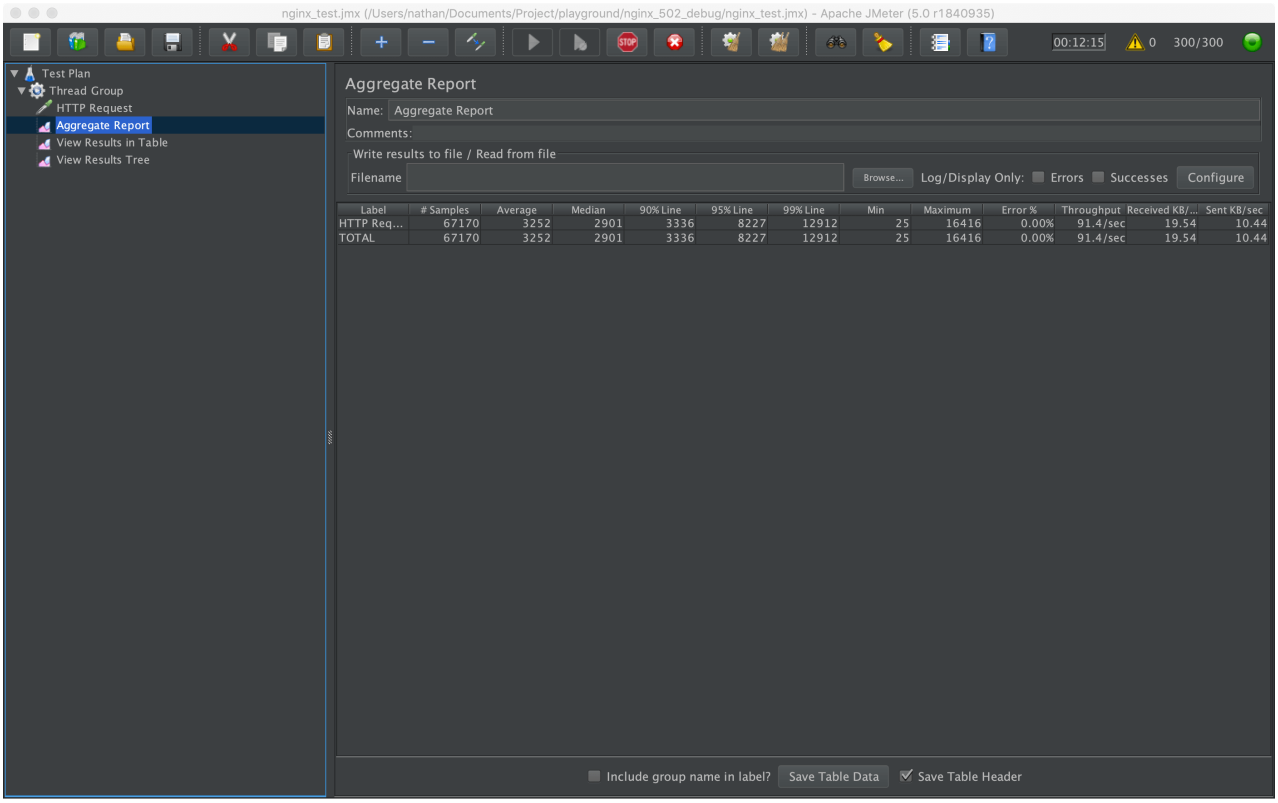
但是如果 `keepalive` 参数设置得过大，后端应用和 nginx 就会保持许多长连接而不释放，后端应用可能没有可用端口，无法再接收新的请求。 所以 `keepalive` 参数要通过并发量和部署架构合理设置，此处演示设置为 1000，暂时没有碰到什么问题。

重新测试

使用新的 nginx 配置重新启动测试服务：

```
$ docker-compose -f docker-
compose.stable.yml up
```

启动 jmeter 压测脚本，长时间运行后也没有报 502 错误。



查看后端应用的连接情况，已经没有了 timewait 状态的连接。

```
$ sudo nsenter -t $(docker inspect -f '{{.State.Pid}}' nginx_502_debug_app2_1)
-n ss -s
# Total: 198
# TCP:   1185 (estab 191, closed 991, orphaned 2, timewait 0)

# Transport Total   IP        IPv6
# RAW     0         0         0
# UDP     1         1         0
# TCP     194       193       1
# INET    195       194       1
# FRAG    0         0         0
```

其他技巧

1. 查看容器内创建的连接数

由于容器与宿主机有隔离，无法在宿主机上看到容器内的 TCP 连接数，一个方法是 attach 到运行的容器里，安装 `iproute2` 或者 `netstat` 之类的工具，但是还有一种更方便的方法，只要宿主机上安装了相应的软件即可。

```
$ sudo nsenter -t $(docker inspect -f '{{.State.Pid}}' nginx_502_debug_app2_1)
-n ss -s
```

`docker inspect` 找到容器进程对应的宿主机进程 pid，`nsenter -t` 命令进入对应进程的命名空间，`-n ss -s` 代表在命名空间中执行相应的命令 `ss -s`。

通过这个方法，可以看到容器内建立的连接数，或者执行其他需要在命名空间内执行的命令。

2. 查看 TCP 队列溢出数

```
$ netstat -s | grep -i listen
# 674915 times the listen queue of a socket
overflowed
# 674915 SYNs to LISTEN sockets dropped

# 或者

$ nstat -a | grep Listen
# TcpExtListenOverflows           674915
0.0
# TcpExtListenDrops               674915
0.0
```

3. 查看不同状态的连接总数

可以使用 `netstat` 或 `ss` 命令查看，现在比较推荐 `ss` 命令，它速度更快，查询功能更简单

以查找 `TIME_WAIT` 状态为例：

```
$ netstat -ant | grep WAIT | wc -l
$ ss -ant -4 | grep WAIT | wc -l
$ ss -ant -4 state time-wait | wc -l  # 需要减1，减去header
这一行
```

4. 查看全连接队列长度和已用数

`ss -lnt` 命令只查看TCP监听端口

```
$ ss -lnt
# State   Recv-Q  Send-Q  Local Address:Port  Peer
Address:Port
# LISTEN  0       32      192.168.122.1:53    0.0.0.0:*
# LISTEN  0       128     0.0.0.0:22          0.0.0.0:*
# LISTEN  0       20480   127.0.0.1:8384      0.0.0.0:*
# LISTEN  0       1024    127.0.0.1:9001      0.0.0.0:*
# LISTEN  8       128     127.0.0.1:5900      0.0.0.0:*
```

`Recv-Q` 代表全连接队列中已有的连接数量，也就是已经完成三次握手，但还没被应用程序 `accept()` 的连接数； `Send-Q` 代表当前进程全连接队列的最大长度。

总结

nginx 出现 `no live upstreams` 是由于连不上后端服务器，因为后端服务器连接队列满了，丢弃 nginx 发来的请求，导致 nginx 认为没有可用的 upstream 服务器，nginx 在短时间内重试所有 upstream 后还失败，就会报 502 错误。

一段时间内 nginx 都会认为后端服务全 down 掉，直接给客户端返回 502，不给后端转发流量，所以出现监控图中的 CPU、网络波动情况。

对于后端服务器来说，丢弃连接是 Linux 内核控制的，后端应用调用 `accept()` 只会从已经成功建立连接的队列中取，所以内核丢弃的连接对应用程序来说是无法感知的，应用程序就不会有报错信息。

等待一段时间后，后端应用处理完队列中的连接，nginx 也认为后端已经恢复，流量又重新分发给后端应用，短时间内大量请求又填满了后端的连接队列，重新出现错误，如此反复。

解决方法是以下两点：

1. 调整 nginx 主机内核参数，增加全连接队列、半连接队列长度，提高 nginx 并发能力
2. 配置 nginx 和后端使用长连接，使用合理的 `keepalive` 参数，节省 TCP 连接创建的开销，节省后端服务器端口资源

分析完这次错误后，在此把整个排查思路和使用的工具记录下来，供大家共同讨论，如果文中有表述不清晰之处，还请多多指出。

参考资料