

## 12 | 连接无效：使用Keep-Alive还是应用心跳来检测？

2019-08-28 盛延敏

网络编程实战

[进入课程 >](#)



讲述：冯永吉

时长 12:09 大小 11.14M



你好，我是盛延敏，这里是网络编程实战第 12 讲，欢迎回来。

上一篇文章中，我们讲到了如何使用 `close` 和 `shutdown` 来完成连接的关闭，在大多数情况下，我们会优选 `shutdown` 来完成对连接一个方向的关闭，待对端处理完之后，再完成另外一个方向的关闭。

在很多情况下，连接的一端需要一直感知连接的状态，如果连接无效了，应用程序可能需要报错，或者重新发起连接等。

在这一篇文章中，我将带你体验一下对连接状态的检测，并提供检测连接状态的最佳实践。

### 从一个例子开始

让我们用一个例子开始今天的话题。

我之前做过一个基于 NATS 消息系统的项目，多个消息的提供者（pub）和订阅者（sub）都连到 NATS 消息系统，通过这个系统来完成消息的投递和订阅处理。

突然有一天，线上报了一个故障，一个流程不能正常处理。经排查，发现消息正确地投递到了 NATS 服务端，但是消息订阅者没有收到该消息，也没能做出处理，导致流程没能进行下去。

通过观察消息订阅者后发现，消息订阅者到 NATS 服务端的连接虽然显示是“正常”的，但实际上，这个连接已经是无效的了。为什么呢？这是因为 NATS 服务器崩溃过，NATS 服务器和消息订阅者之间的连接中断 FIN 包，由于异常情况，没能够正常到达消息订阅者，这样造成的结果就是消息订阅者一直维护着一个“过时的”连接，不会收到 NATS 服务器发送来的消息。

这个故障的根本原因在于，作为 NATS 服务器的客户端，消息订阅者没有及时对连接的有效性进行检测，这样就造成了问题。

保持对连接有效性的检测，是我们在实战中必须要注意的一个点。

## TCP Keep-Alive 选项

很多刚接触 TCP 编程的人会惊讶地发现，在没有数据读写的“静默”的连接上，是没有办法发现 TCP 连接是有效还是无效的。比如客户端突然崩溃，服务器端可能在几天内都维护着一个无用的 TCP 连接。前面提到的例子就是这样的场景。

那么有没有办法开启类似的“轮询”机制，让 TCP 告诉我们，连接是不是“活着”的呢？

这就是 TCP 保持活跃机制所要解决的问题。实际上，TCP 有一个保持活跃的机制叫做 Keep-Alive。

这个机制的原理是这样的：

定义一个时间段，在这个时间段内，如果没有任何连接相关的活动，TCP 保活机制会开始作用，每隔一个时间间隔，发送一个探测报文，该探测报文包含的数据非常少，如果连续几

个探测报文都没有得到响应，则认为当前的 TCP 连接已经死亡，系统内核将错误信息通知给上层应用程序。

上述的可定义变量，分别被称为保活时间、保活时间间隔和保活探测次数。在 Linux 系统中，这些变量分别对应 `sysctl` 变量 `net.ipv4.tcp_keepalive_time`、`net.ipv4.tcp_keepalive_intvl`、`net.ipv4.tcp_keepalive_probes`，默认设置是 7200 秒（2 小时）、75 秒和 9 次探测。

如果开启了 TCP 保活，需要考虑以下几种情况：

第一种，对端程序是正常工作的。当 TCP 保活的探测报文发送给对端，对端会正常响应，这样 TCP 保活时间会被重置，等待下一个 TCP 保活时间的到来。

第二种，对端程序崩溃并重启。当 TCP 保活的探测报文发送给对端后，对端是可以响应的，但由于没有该连接的有效信息，会产生一个 RST 报文，这样很快就会发现 TCP 连接已经被重置。

第三种，是对端程序崩溃，或对端由于其他原因导致报文不可达。当 TCP 保活的探测报文发送给对端后，石沉大海，没有响应，连续几次，达到保活探测次数后，TCP 会报告该 TCP 连接已经死亡。

TCP 保活机制默认是关闭的，当我们选择打开时，可以分别在连接的两个方向上开启，也可以单独在一个方向上开启。如果开启服务器端到客户端的检测，就可以在客户端非正常断连的情况下清除在服务器端保留的“脏数据”；而开启客户端到服务器端的检测，就可以在服务器无响应的情况下，重新发起连接。

为什么 TCP 不提供一个频率很好的保活机制呢？我的理解是早期的网络带宽非常有限，如果提供一个频率很高的保活机制，对有限的带宽是一个比较严重的浪费。

## 应用层保活

如果使用 TCP 自身的 keep-Alive 机制，在 Linux 系统中，最少需要经过 2 小时 11 分 15 秒才可以发现一个“死亡”连接。这个时间是怎么计算出来的呢？其实是通过 2 小时，加上 75 秒乘以 9 的总和。实际上，对很多对时延要求敏感的系统，这个时间间隔是不可接受的。

所以，必须在应用程序这一层来寻找更好的解决方案。

我们可以通过在应用程序中模拟 TCP Keep-Alive 机制，来完成在应用层的连接保活。

我们可以设计一个 PING-PONG 的机制，需要保活的一方，比如客户端，在保活时间达到后，发起对连接的 PING 操作，如果服务器端对 PING 操作有回应，则重新设置保活时间，否则对探测次数进行计数，如果最终探测次数达到了保活探测次数预先设置的值之后，则认为连接已经无效。


这里有两个比较关键的点：

第一个是需要使用定时器，这可以通过使用 I/O 复用自身的机制来实现；第二个是需要设计一个 PING-PONG 的协议。

下面我们尝试来完成这样的一个设计。

## 消息格式设计

我们的程序是客户端来发起保活，为此定义了一个消息对象。你可以在文稿中看到这个消息对象，这个消息对象是一个结构体，前 4 个字节标识了消息类型，为了简单，这里设计了 MSG\_PING、MSG\_PONG、MSG\_TYPE 1 和 MSG\_TYPE 2 四种消息类型。

 复制代码


```
1 typedef struct {
2     u_int32_t type;
3     char data[1024];
4 } messageObject;
5
6 #define MSG_PING          1
7 #define MSG_PONG          2
8 #define MSG_TYPE1        11
9 #define MSG_TYPE2        21
```

## 客户端程序设计

客户端完全模拟 TCP Keep-Alive 的机制，在保活时间达到后，保活次数增加 1，同时向服务器端发送 PING 格式的消息，此后以预设的保活时间间隔，不断地向服务器端发送 PING

格式的消息。如果能收到服务器端的应答，则结束保活，将保活时间置为 0。

这里我们使用 select I/O 复用函数自带的定时器，select 函数将在后面详细介绍。

 复制代码

```
1 #include "lib/common.h"
2 #include "message_object.h"
3
4 #define    MAXLINE    4096
5 #define    KEEP_ALIVE_TIME    10
6 #define    KEEP_ALIVE_INTERVAL    3
7 #define    KEEP_ALIVE_PROBETIMES    3
8
9
10 int main(int argc, char **argv) {
11     if (argc != 2) {
12         error(1, 0, "usage: tcpclient <IPaddress>");
13     }
14
15     int socket_fd;
16     socket_fd = socket(AF_INET, SOCK_STREAM, 0);
17
18     struct sockaddr_in server_addr;
19     bzero(&server_addr, sizeof(server_addr));
20     server_addr.sin_family = AF_INET;
21     server_addr.sin_port = htons(SERV_PORT);
22     inet_pton(AF_INET, argv[1], &server_addr.sin_addr);
23
24     socklen_t server_len = sizeof(server_addr);
25     int connect_rt = connect(socket_fd, (struct sockaddr *) &server_addr, server_len);
26     if (connect_rt < 0) {
27         error(1, errno, "connect failed ");
28     }
29
30     char recv_line[MAXLINE + 1];
31     int n;
32
33     fd_set readmask;
34     fd_set allreads;
35
36     struct timeval tv;
37     int heartbeats = 0;
38
39     tv.tv_sec = KEEP_ALIVE_TIME;
40     tv.tv_usec = 0;
41
42     messageObject messageObject;
43
44     FD_ZERO(&allreads);
```

```

45     FD_SET(socket_fd, &allreads);
46     for (;;) {
47         readmask = allreads;
48         int rc = select(socket_fd + 1, &readmask, NULL, NULL, &tv);
49         if (rc < 0) {
50             error(1, errno, "select failed");
51         }
52         if (rc == 0) {
53             if (++heartbeats > KEEP_ALIVE_PROBETIMES) {
54                 error(1, 0, "connection dead\n");
55             }
56             printf("sending heartbeat #%d\n", heartbeats);
57             messageObject.type = htonl(MSG_PING);
58             rc = send(socket_fd, (char *) &messageObject, sizeof(messageObject), 0);
59             if (rc < 0) {
60                 error(1, errno, "send failure");
61             }
62             tv.tv_sec = KEEP_ALIVE_INTERVAL;
63             continue;
64         }
65         if (FD_ISSET(socket_fd, &readmask)) {
66             n = read(socket_fd, recv_line, MAXLINE);
67             if (n < 0) {
68                 error(1, errno, "read error");
69             } else if (n == 0) {
70                 error(1, 0, "server terminated \n");
71             }
72             printf("received heartbeat, make heartbeats to 0 \n");
73             heartbeats = 0;
74             tv.tv_sec = KEEP_ALIVE_TIME;
75         }
76     }
77 }

```

这个程序主要分成三大部分：

第一部分为套接字的创建和连接建立：

15-16 行，创建了 TCP 套接字；

18-22 行，创建了 IPv4 目标地址，其实就是服务器端地址，注意这里使用的是传入参数作为服务器地址；

24-28 行，向服务器端发起连接。

第二部分为 select 定时器准备：

39-40 行，设置了超时时间为 KEEP\_ALIVE\_TIME，这相当于保活时间；

44-45 行，初始化 select 函数的套接字。

最重要的为第三部分，这一部分需要处理心跳报文：


48 行调用 select 函数，感知 I/O 事件。这里的 I/O 事件，除了套接字上的读操作之外，还有在 39-40 行设置的超时事件。当 KEEP\_ALIVE\_TIME 这段时间到达之后，select 函数会返回 0，于是进入 53-63 行的处理；

在 53-63 行，客户端已经在 KEEP\_ALIVE\_TIME 这段时间内没有收到任何对当前连接的反馈，于是发起 PING 消息，尝试问服务器端：“喂，你还活着吗？”这里我们通过传送一个类型为 MSG\_PING 的消息对象来完成 PING 操作，之后我们会看到服务器端程序如何响应这个 PING 操作；

第 65-74 行是客户端在接收到服务器端程序之后的处理。为了简单，这里就没有再进行报文格式的转换和分析。在实际的工作中，这里其实是需要对报文进行解析后处理的，只有是 PONG 类型的回应，我们才认为是 PING 探活的结果。这里认为既然收到服务器端的报文，那么连接就是正常的，所以会对探活计数器和探活时间都置零，等待下一次探活时间的来临。

## 服务器端程序设计

服务器端的程序接受一个参数，这个参数设置的比较大，可以模拟连接没有响应的情况。服务器端程序在接收到客户端发送来的各种消息后，进行处理，其中如果发现是 PING 类型的消息，在休眠一段时间后回复一个 PONG 消息，告诉客户端：“嗯，我还活着。”当然，如果这个休眠时间很长的话，那么客户端就无法快速知道服务器端是否存活，这是我们模拟连接无响应的一个手段而已，实际情况下，应该是系统崩溃，或者网络异常。

 复制代码

```
1 #include "lib/common.h"
2 #include "message_object.h"
3
4 static int count;
5
6 int main(int argc, char **argv) {
7     if (argc != 2) {
8         error(1, 0, "usage: tcpsever <sleepingtime>");
```



```

9     }
10
11     int sleepingTime = atoi(argv[1]);
12
13     int listenfd;
14     listenfd = socket(AF_INET, SOCK_STREAM, 0);
15
16     struct sockaddr_in server_addr;
17     bzero(&server_addr, sizeof(server_addr));
18     server_addr.sin_family = AF_INET;
19     server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
20     server_addr.sin_port = htons(SERV_PORT);
21
22     int rt1 = bind(listenfd, (struct sockaddr *) &server_addr, sizeof(server_addr));
23     if (rt1 < 0) {
24         error(1, errno, "bind failed ");
25     }
26
27     int rt2 = listen(listenfd, LISTENQ);
28     if (rt2 < 0) {
29         error(1, errno, "listen failed ");
30     }
31
32     int connfd;
33     struct sockaddr_in client_addr;
34     socklen_t client_len = sizeof(client_addr);
35
36     if ((connfd = accept(listenfd, (struct sockaddr *) &client_addr, &client_len)) < 0)
37         error(1, errno, "bind failed ");
38     }
39
40     messageObject message;
41     count = 0;
42
43     for (;;) {
44         int n = read(connfd, (char *) &message, sizeof(messageObject));
45         if (n < 0) {
46             error(1, errno, "error read");
47         } else if (n == 0) {
48             error(1, 0, "client closed \n");
49         }
50
51         printf("received %d bytes\n", n);
52         count++;
53
54         switch (ntohl(message.type)) {
55             case MSG_TYPE1 :
56                 printf("process  MSG_TYPE1 \n");
57                 break;
58
59             case MSG_TYPE2 :
60                 printf("process  MSG_TYPE2 \n");

```



```

61         break;
62
63     case MSG_PING: {
64         messageObject pong_message;
65         pong_message.type = MSG_PONG;
66         sleep(sleepingTime);
67         ssize_t rc = send(connfd, (char *) &pong_message, sizeof(pong_message),
68             if (rc < 0)
69                 error(1, errno, "send failure");
70         break;
71     }
72
73     default :
74         error(1, 0, "unknown message type (%d)\n", ntohs(message.type));
75 }
76
77 }
78
79 }

```

服务器端程序主要分为两个部分。

第一部分为监听过程的建立，包括 7-38 行；第 13-14 行先创建一个本地 TCP 监听套接字；16-20 行绑定该套接字到本地端口和 ANY 地址上；第 27-38 行分别调用 listen 和 accept 完成被动套接字转换和监听。

第二部分为 43 行到 77 行，从建立的连接套接字上读取数据，解析报文，根据消息类型进行不同的处理。

55-57 行为处理 MSG\_TYPE1 的消息；

59-61 行为处理 MSG\_TYPE2 的消息；

重点是 64-72 行处理 MSG\_PING 类型的消息。通过休眠来模拟响应是否及时，然后调用 send 函数发送一个 PONG 报文，向客户端表示“还活着”的意思；


74 行为异常处理，因为消息格式不认识，所以程序出错退出。

## 实验


基于上面的程序设计，让我们分别做两个不同的实验：

第一次实验，服务器端休眠时间为 60 秒。

我们看到，客户端在发送了三次心跳检测报文 PING 报文后，判断出连接无效，直接退出了。之所以造成这样的结果，是因为在这段时间内没有接收到来自服务器端的任何 PONG 报文。当然，实际工作的程序，可能需要不一样的处理，比如重新发起连接。

 复制代码


```
1 ./pingclient 127.0.0.1
2 sending heartbeat #1
3 sending heartbeat #2
4 sending heartbeat #3
5 connection dead
```

 复制代码

```
1 ./pingserver 60
2 received 1028 bytes
3 received 1028 bytes
```

第二次实验，我们让服务器端休眠时间为 5 秒。

我们看到，由于这一次服务器端在心跳检测过程中，及时地进行了响应，客户端一直都会认为连接是正常的。

 复制代码

```
1 ./pingclient 127.0.0.1
2 sending heartbeat #1
3 sending heartbeat #2
4 received heartbeat, make heartbeats to 0
5 received heartbeat, make heartbeats to 0
6 sending heartbeat #1
7 sending heartbeat #2
8 received heartbeat, make heartbeats to 0
9 received heartbeat, make heartbeats to 0
```

 复制代码

```
1 $./pingserver 5
2 received 1028 bytes
3 received 1028 bytes
4 received 1028 bytes
5 received 1028 bytes
```

## 总结

通过今天的文章，我们能看到虽然 TCP 没有提供系统的保活能力，让应用程序可以方便地感知连接的存活，但是，我们可以在应用程序里灵活地建立这种机制。一般来说，这种机制的建立依赖于系统定时器，以及恰当的应用层报文协议。

## 思考题

和往常一样，我留两道思考题给大家：

你可以看到今天的内容主要是针对 TCP 的探活，那么你觉得这样的方法是否同样适用于 UDP 呢？

第二道题是，有人说额外的探活报文占用了有限的带宽，对此你是怎么想的呢？而且，为什么需要多次探活才能决定一个 TCP 连接是否已经死亡呢？

欢迎你在评论区写下你的思考，我会和你一起交流。也欢迎把这篇文章分享给你的朋友或者同事，与他们一起讨论一下这两个问题吧。

# 网络编程实战

从底层到实战，深度解析网络编程

盛延敏

前大众点评云平台首席架构师



新版升级：点击「 请朋友读」，20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，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 11 | 优雅地关闭还是粗暴地关闭？

下一篇 13 | 小数据包应对之策：理解TCP协议中的动态数据传输

## 精选留言 (15)

写留言



传说中的成大大 置顶

2019-08-28

思考题

1. udp不需要连接 所以没有必要心跳包
2. 我觉得还是很有必要判定存活 像以前网吧打游戏 朋友的电脑突然蓝屏死机 朋友的角色还残留于游戏中,所以服务器为了判定他是否真的存活还是需要一个心跳包 隔了一段时间过后把朋友角色踢下线

展开 ∨

作者回复: 2是一个很好的例子。





fjpcode  
2019-08-28

- 1.UDP里面各方并不会维护一个socket上下文状态是无连接的，如果为了连接而保活是不必要的，如果为了探测对端是否正常工作而做ping-pong也是可行的。
- 2.额外的探活报文是会占用一些带宽资源，可根据实际业务场景，适当增加保活时间，降低探活频率，简化ping-pong协议。
- 3.多次探活是为了防止误伤，避免ping包在网络中丢失掉了，而误认为对端死亡。

展开 ▾

作者回复: 📬

💬 4



苦行僧  
2019-09-02

使用心跳包不就是为了保持keep alive吗？文章内容到最后也没有点题？

💬 📬



沉淀的梦想  
2019-09-01

老师在pingclient的实现中，为什么只要该一下结构体的字段值就能重置探活时间了？

```
if (FD_ISSET(socket_fd, &readmask)) {  
    //.....  
    // 重置探活时间...
```

展开 ▾

💬 📬



徐凯  
2019-08-29

我看到大家对于第二个问题的答案都提到了为了避免探活包丢包 所以要发多个探活包。但是我觉得只要发了探活包对方就一定能收到，就算丢了 发送端也会重传。而大家说的发送多个探活包的原因 是因为重传需要等到计时器超时才传，而如果网络堵塞的话可能会出现频繁丢包 那么服务端可能需要很久才能知道对方已经离开 发多个探活包的话 就减少了这个等待的时间 这样理解对么

展开 ▾

作者回复: 探活包的多次发送，还是为了减少误判的概率，你说的是一种情况，但我觉得多此探活肯定会拉长对一个"无效"连接的判断时间的。这个是一个tradeoff，所以大多数程序都把这个作为选项让使用者自己配置。

< >

💬 👍



**石将从**

2019-08-29

为啥这句套接字要加1呢? `int rc = select(socket_fd + 1, &readmask, NULL, NULL, &t  
v);`

作者回复: 这是因为, 嗯, 跟select实现有关, 我再后面讲select时详细剖析, 现在先记住吧。

< >

💬 1 👍



**张立华**

2019-08-28

请问下作者, 你提到的nats客户端的问题, 是nats客户端的那个版本的bug

作者回复: 很老的一个版本, 大概在2014-1015年的java-nats client版本吧。

< >

💬 👍



**传说中的成大大**

2019-08-28

其实我也在想比如客户端崩溃重启过后 然后重新建立连接 创建socket 那之前那个连接是  
怎么处理的? 因为之前那个服务器那个连接发往的还是我 所以就想知道这个时候是怎么处  
理的

作者回复: 那是一个过时的连接, 显然应该被服务器端检测到, 并摘掉这个过时的占用系统资源的  
连接。

< >

💬 👍



**QQ怪**

2019-08-28

- 1.udp本来无连接, 不需要探活
- 2.多次探活的原因可能会发生探活丢包

💬 👍



**许童童**

2019-08-28

思考题:

1.适用也不适用，具体看应用场景，UDP本身无连接，根本没有必要保活，如果是基于UDP做一些其它协议如HTTP/3，就是在UDP层做的保活。

2.保活报文很小，但如果每台机器，每个应用，每个连接都大量存在保活，那整个网络环境就都是保活报文了，带宽确实会被大量占用。为什么要多次保活，因为保活报文也可...

展开 ▾



锦

2019-08-28

1,我觉得同样适合udp协议，虽然udp是无连接的，但是为了更好的利用网络资源，还是需要保活机制

2,保活机制设置合理的情况下对带宽影响不大，资源消耗不多，到作用很大，性价比很高需要多次保活是为了double check，可能有些回包还没找到回家的路，也有可能对端正在故障恢复，需要一些时间。

展开 ▾



webmin

2019-08-28

长连接的情况下，最好要做双向探测。

UDP也适用，UDP下的探测可以知道是否网络发生故障。

单次看比较小，连接数的量级很大的话，加起来探测占用的带宽就会很大。

一次通信在网络中有可能要经过多个设备，其中有设备可能出现瞬时故障，也可能会多条通路，其中一条路有问题。

展开 ▾



LDxy

2019-08-28

TCP本身的keep-alive的时间是可以自己设置的吗？如果是可以自己设置的，为何还需要自己实现这个机制？

作者回复: 可以设置的，但是有时候应用层需要感知处理这样的异常



杨领well

2019-08-28



能多次探测，说明对端极有可能还存在，只是因为各种原因导致没收到回复的包。因为如果对端不存在了，应该会收到 RST，导致链接关闭。

展开 ∨



**xindoo**

2019-08-28

第一个问题，我觉得不适合udp，udp本身是无连接的，所以就没必要探活来保持连接。  
第二个问题，以现今网络传输数据量看，探活报文只会占有所有传输数据量极小部分，甚至可以忽略不计，所以占有限贷款一说就是伪命题。至于需要多次探活，还是为了避免因为网络丢包的问题导致的误判。

展开 ∨

