



guisu，程序人生。逆水行舟，不进则退。

能干的人解决问题。智慧的人避开问题(A clever person solves a problem. A wise person avoids it)

目录视图 摘要视图 [RSS](#) 订阅

个人资料



真实的归宿

访问: 4466127次
积分: 27165
等级: [BLOG](#)
排名: 第153名

原创: 216篇 转载: 2篇
译文: 0篇 评论: 1293条

文章分类

- 操作系统 (5)
- Linux (23)
- MySQL (12)
- PHP (41)
- 架构 (5)
- PHP内核 (11)
- 技术人生 (8)
- 数据结构与算法 (29)
- 云计算hadoop (25)
- 网络知识 (8)
- c/c++ (23)
- memcache (5)
- HipHop (1)
- 计算机原理 (4)
- Java (11)
- socket网络编程 (8)
- 设计模式 (26)
- AOP (2)
- 重构 (11)
- 重构与模式 (1)
- 搜索引擎Search Engine (15)
- 大数据处理 (12)
- HTML5 (1)
- Android (1)
- webserver (3)
- NOSQL (7)
- NOSQL Mongo (0)
- 分布式 (1)
- 数据结构与算法 xi (0)
- 协议 (1)
- 信息论的熵 (0)

[【1024程序员节】获奖结果公布](#) [【观点】有了深度学习，你还学传统机器学习算法么？](#) [【资源库】火爆了的React Native都在研究什么](#)

TCP连接的状态详解以及故障排查

2014-08-20 07:06 46630人阅读 评论(7) 收藏 举报

分类: [c/c++ \(22\)](#) [socket网络编程 \(7\)](#)

版权声明：本文为博主原创文章，未经博主允许不得转载。

目录(?) [\[+\]](#)

我们通过了解TCP各个状态，可以排除和定位网络或系统故障时大有帮助。（总结网络上的内容）

1、TCP状态

linux查看tcp的状态命令：

- 1)、netstat -nat 查看TCP各个状态的数量
- 2)、lsof -i:port 可以检测到打开套接字的状况
- 3)、sar -n SOCK 查看tcp创建的连接数
- 4)、tcpdump -iany tcp port 9000 对tcp端口为9000的进行抓包

LISTENING：侦听来自远方的**TCP**端口的连接请求。

首先服务端需要打开一个[socket](#)进行监听，状态为LISTEN。

有提供某种服务才会处于LISTENING状态，**TCP状态变化就是某个端口的状态变化**，提供一个服务就打开一个端口，例如：提供www服务默认开的是80端口，提供ftp服务默认的端口为21，当提供的服务没有被连接时就处于LISTENING状态。FTP服务启动后首先处于侦听（LISTENING）状态。处于侦听LISTENING状态时，该端口是开放的，等待连接，但还没有被连接。就像你房子的门已经敞开的，但还没有人进来。

看LISTENING状态最主要的是看本机开了哪些端口，这些端口都是哪个程序开的，关闭不必要的端口是保证安全的一个非常重要的方面，服务端口都对应一个服务（应用程序），停止该服务就关闭了该端口，例如要关闭21端口只要停止IIS服务中的FTP服务即可。关于这方面的知识请参阅其它文章。

如果你不幸中了服务端口的木马，木马也开个端口处于LISTENING状态。

SYN-SENT：客户端SYN_SENT状态：

再发送连接请求后等待匹配的连接请求:客户端通过应用程序调用connect进行active open.于是客户端tcp发送一个SYN以请求建立一个连接.之后状态置为SYN_SENT. /*The socket is actively attempting to establish a connection. 在发送连接请求后等待匹配的连接请求 */

当请求连接时客户端首先要发送同步信号给要访问的机器，此时状态为SYN_SENT，如果连接成功了就变为ESTABLISHED，正常情况下SYN_SENT状态非常短暂。例如要访问网站http://www.baidu.com,如果是正常连接的话，用TCPView观察[IEXPLORE](#).EXE（IE）建立的连接会发现很快从SYN_SENT变为ESTABLISHED，表示连接成功。SYN_SENT状态快的也许看不到。

如果发现有SYN_SENT出现，那一般有这么几种情况，一是你要访问的网站不存在或线路不好，二是用扫描软件扫描一个网段的机器，也会出出现很多SYN_SENT，另外就是可能中了病毒了，例如中了“冲击波”，病毒发作时会扫描其它机器，这样会有很多SYN_SENT出现。

关于php的libevent扩展的应用 (0)

libevent简单介绍 (0)

SOA (0)

文章存档

2016年08月 (4)

2015年12月 (2)

2015年10月 (4)

2015年05月 (2)

2015年04月 (1)

展开

阅读排行

八大排序算法 (432733)

Mysql 多表联合查询效率 (170622)

深入理解java异常处理机 (165173)

socket阻塞与非阻塞，同 (128809)

Linux的SOCKET编程详解 (125829)

设计模式 (十八) 策略模 (122345)

Hadoop Hive sql语法详解 (113191)

高性能Mysql主从架构的 (106425)

hbase安装配置（整合到 (101816)

Java输入输出流 (94867)

评论排行

八大排序算法 (126)

深入理解java异常处理机 (109)

socket阻塞与非阻塞，同 (56)

硬盘的读写原理 (47)

设计模式 (十八) 策略模 (41)

设计模式 (一) 工厂模式 (37)

Redis应用场景 (31)

Linux的SOCKET编程详解 (27)

海量数据处理算法—Bit-M (27)

设计模式 (十七) 状态模 (25)

推荐文章

* 2016 年最受欢迎的编程语言是什么？

* Chromium扩展（Extension）的页面（Page）加载过程分析

* Android Studio 2.2 来啦

* 手把手教你做音乐播放器（二）技术原理与框架设计

* JVM 性能调优实战之：使用阿里开源工具 TProfiler 在海量业务代码中精确定位性能代码

最新评论

Linux的SOCKET编程详解 Sherlock_Holmes: 确实整理得不错，感谢。

设计模式（四）原型模式Prototy 12期-张婷: 博主的分享很好，看了之后明白了很多

SYN-RECEIVED: 服务器端状态SYN_RCVD

再收到和发送一个连接请求后等待对方连接请求的确认

当服务器收到客户端发送的同步信号时，将标志位ACK和SYN置1发送给客户端，此时服务器端处于SYN_RCVD状态，如果连接成功了就变为ESTABLISHED，正常情况下SYN_RCVD状态非常短暂。

如果发现有**很多SYN_RCVD状态，那你的机器有可能被SYN Flood的DoS(拒绝服务攻击)攻击了**。

SYN Flood的攻击原理是：

在进行三次握手时，攻击软件向被攻击的服务器发送SYN连接请求（握手的第一步），但是这个地址是伪造的，如攻击软件随机伪造了51.133.163.104、65.158.99.152等等地址。服务器在收到连接请求时将标志位ACK和SYN置1发送给客户端（握手的第二步），但是这些客户端的IP地址都是伪造的，服务器根本找不到客户机，也就是说握手的第三步不可能完成。

这种情况下服务器端一般会重试（再次发送SYN+ACK给客户端）并等待一段时间后丢弃这个未完成的连接，这段时间的长度我们称为SYN Timeout，一般来说这个时间是分钟的数量级（大约为30秒-2分钟）；一个用户出现异常导致服务器的一个线程等待1分钟并不是什么很大的问题，但如果有一个恶意的攻击者大量模拟这种情况，服务器端将为了维护一个非常大的半连接列表而消耗非常多的资源——数以万计的半连接，即使是简单的保存并遍历也会消耗非常多的CPU时间和内存，何况还要不断对这个列表中的IP进行SYN+ACK的重试。此时从正常客户的角度来看，服务器失去响应，这种情况我们称做：**服务器端受到了SYN Flood攻击（SYN洪水攻击）**

ESTABLISHED: 代表一个打开的连接。

ESTABLISHED状态是表示两台机器正在传输数据，观察这个状态最主要的就是看哪个程序正处于ESTABLISHED状态。

服务器出现很多ESTABLISHED状态：`netstat -nat |grep 9502`或者使用`lsof -i:9502`可以检测到。

当客户端未主动close的时候就断开连接：即客户端发送的FIN丢失或未发送。

这时候若客户端断开的时候发送了FIN包，则服务端将会处于CLOSE_WAIT状态；

这时候若客户端断开的时候未发送FIN包，则服务端处还是显示ESTABLISHED状态；

结果客户端重新连接服务器。

而新连接上来的客户端（也就是刚才断掉的重新连上来了）在服务端肯定是ESTABLISHED;如果客户端重复的上演这种情况，那么服务端将会出现大量的假的ESTABLISHED连接和CLOSE_WAIT连接。

最终结果就是新的其他客户端无法连接上来，但是利用netstat还是能看到一条连接已经建立，并显示ESTABLISHED，但始终无法进入程序代码。

FIN-WAIT-1: 等待远程TCP连接中断请求，或先前的连接中断请求的确认

主动关闭(active close)端应用程序调用close，于是其TCP发出FIN请求主动关闭连接，之后进入FIN_WAIT1状态。`/* The socket is closed, and the connection is shutting down.` 等待远程TCP的连接中断请求，或先前的连接中断请求的确认 */

如果服务器出现shutdown再重启，使用netstat -nat查看，就会看到很多**FIN-WAIT-1**的状态。就是因为服务器当前有很多客户端连接，直接关闭服务器后，无法接收到客户端的ACK。

FIN-WAIT-2: 从远程TCP等待连接中断请求

主动关闭端接到ACK后，就进入了FIN-WAIT-2。`/* Connection is closed, and the socket is waiting for a shutdown from the remote end.` 从远程TCP等待连接中断请求 */

这就是著名的半关闭的状态了，这是在关闭连接时，客户端和服务端两次握手之后的状态。在这个状态下，应用程序还有接受数据的能力，但是已经无法发送数据，但是也有一种可能是，客户端一直处于FIN_WAIT_2状态，而服务器则一直处于WAIT_CLOSE状态，而直到应用层来决定关闭这个状态。

CLOSE-WAIT: 等待从本地用户发来的连接中断请求

被动关闭(passive close)端TCP接到FIN后，就发出ACK以回应FIN请求(它的接收也作为文件结束符传递给上层应用程序),并进入CLOSE_WAIT。`/* The remote end has shut down, waiting for the socket to close.` 等待从本地用户发来的连接中断请求 */

设计模式（一）工厂模式Factory Rany来吧_程序猿: 很好

设计模式（一）工厂模式Factory Rany来吧_程序猿: 看着楼主的代码，跟着你的代码敲了一遍，完全看懂了，不过就是还是不是很清楚，在项目中应该怎么使用，只能...

深入理解java异常处理机制 张亚军: 通篇也没告诉这个例子的原理

网络互联参考模型（详解）hello_stranger: 邮局那个太形象了

Linux系统结构 详解 真实的归宿: @jhsunnyshine: 是的，linux最新支持的文件系统是ext4

高性能Mysql主从架构的复制原理 李秀才: 用这些配置了从服务器: log_bin = mysql-binserver_id ...

高性能Mysql主从架构的复制原理 李秀才: 我在主服务器里配置了: server-id=1log-bin=mysql-bin可是为什么用SHOW ...

Linux的SOCKET编程详解 eastvenuslee: 一级棒 赞！！！！

友情链接

图灵机器人: 聊天api的最佳选择

CLOSING: 等待远程TCP对连接中断的确认

比较少见。/* Both sockets are shut down but we still don't have all our data sent. 等待远程TCP对连接中断的确认 */

LAST-ACK: 等待原来的发向远程TCP的连接中断请求的确认

被动关闭端一段时间后，接收到文件结束符的应用程序将调用CLOSE关闭连接。这导致它的TCP也发送一个 FIN。/* The remote end has shut down, and the socket is closed. Waiting for acknowledgement. 等待原来发向远程的ACK。*/

使用并发压力测试的时候，突然断开压力测试客户端，服务器会看到很多LAST-ACK。

TIME-WAIT: 等待足够的时间以确保远程TCP接收到连接中断请求的确认

在主动关闭端接收到FIN后，TCP就发送ACK包，并进入TIME-WAIT状态。/* The socket is waiting after close to handle

packets still in the network.等待足够的时间以确保远程TCP接收到连接中断请求的确认 */

TIME_WAIT等待状态，这个状态又叫做2MSL状态，说的是在TIME_WAIT2发送了最后一个ACK数据报以后，要进入TIME_WAIT状态，这个状态是防止最后一次握手的数据报没有传送到对方那里而准备的（注意这不是四次握手，这是第四次握手的保险状态）。这个状态在很大程度上保证了双方都可以正常结束，但是，问题也来了。由于插口的2MSL状态（插口是IP和端口对的意思，socket），使得应用程序在2MSL时间内是无法再次使用同一个插口的，对于客户程序还好一些，但是对于服务程序，例如httpd，它总是要使用同一个端口来进行服务，而在2MSL时间内，启动httpd就会出现错误（插口被使用）。为了避免这个错误，服务器给出了一个平静时间的概念，这是说在2MSL时间内，虽然可以重新启动服务器，但是这个服务器还是要平静的等待2MSL时间的过去才能进行下一次连接。

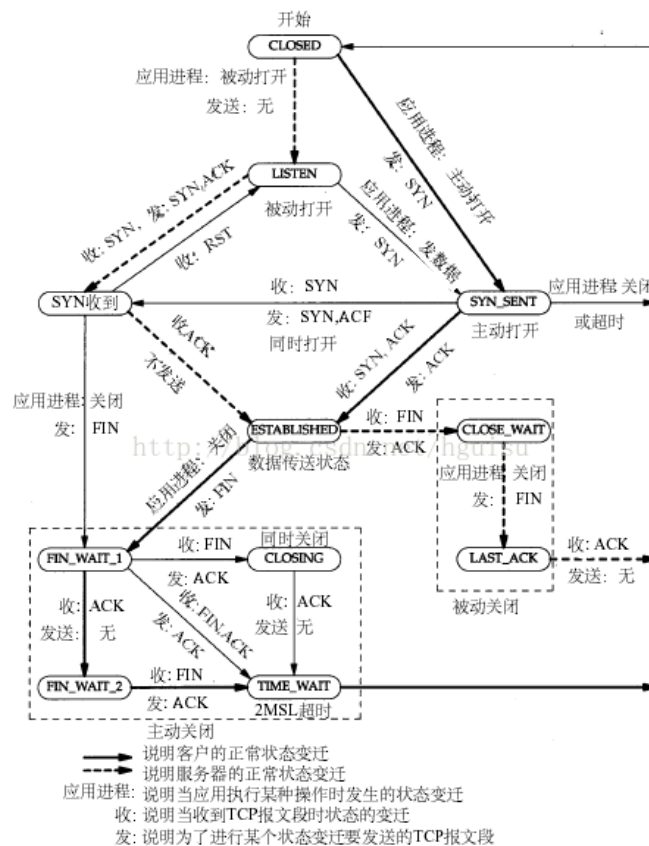
详情请看: TIME_WAIT引起Cannot assign requested address报错

CLOSED: 没有任何连接状态

被动关闭端在接受到ACK包后，就进入了closed的状态。连接结束。/* The socket is not being used. 没有任何连接。*/

2、TCP状态迁移路线图

client/server两条路线讲述TCP状态迁移路线图:



这是一个看起来比较复杂的状态迁移图，因为它包含了两个部分——服务器的状态迁移和客户端的状态迁移，如果从某一个角度出发来看这个图，就会清晰许多，这里面的服务器和客户端都不是绝对的，发送数据的就是客户端，接受数据的就是服务器。

客户端应用程序的状态迁移图

客户端的状态可以用如下的流程来表示:

CLOSED->SYN SENT->ESTABLISHED->FIN WAIT 1->FIN WAIT 2->TIME WAIT->CLOSED

以上流程是在程序正常的情况下应该有的流程，从书中的图中可以看到，在建立连接时，当客户端收到SYN报文的ACK以后，客户端就打开了数据交互地连接。而结束连接则通常是客户端主动结束的，客户端结束应用程序以后，需要经历FIN_WAIT_1，FIN_WAIT_2等状态，这些状态的迁移就是前面提到的结束连接的四次握手。

服务器的状态迁移图

服务器的状态可以用如下的流程来表示:

CLOSED->LISTEN->SYN收到->ESTABLISHED->CLOSE WAIT->LAST ACK->CLOSED

在建立连接的时候，服务器端是在第三次握手之后才进入数据交互状态，而关闭连接则是在关闭连接的第二次握手以后（注意不是第四次）。而关闭以后还要等待客户端给出最后的ACK包才能进入初始的状态。

其他状态迁移

还有一些其他的状态迁移, 这些状态迁移针对服务器和客户端两方面的总结如下

LISTEN->SYN SENT, 对于这个解释就很简单了, 服务器有时候也要打开连接的嘛。

SYN_SENT->SYN收到，服务器和客户端在SYN_SENT状态下如果收到SYN数据报，则都需要发送SYN的ACK数据报并把自己的状态调整到SYN收到状态，准备进入ESTABLISHED

SYN SENT->CLOSED, 在发送超时的情况下, 会返回到CLOSED状态。

SYN 收到->LISTEN, 如果受到RST包, 会返回到LISTEN状态。

SYN 收到->FIN WAIT 1, 这个迁移是说, 可以不用到ESTABLISHED状态, 而可以直接跳转到FIN WAIT 1状态并等待关闭。

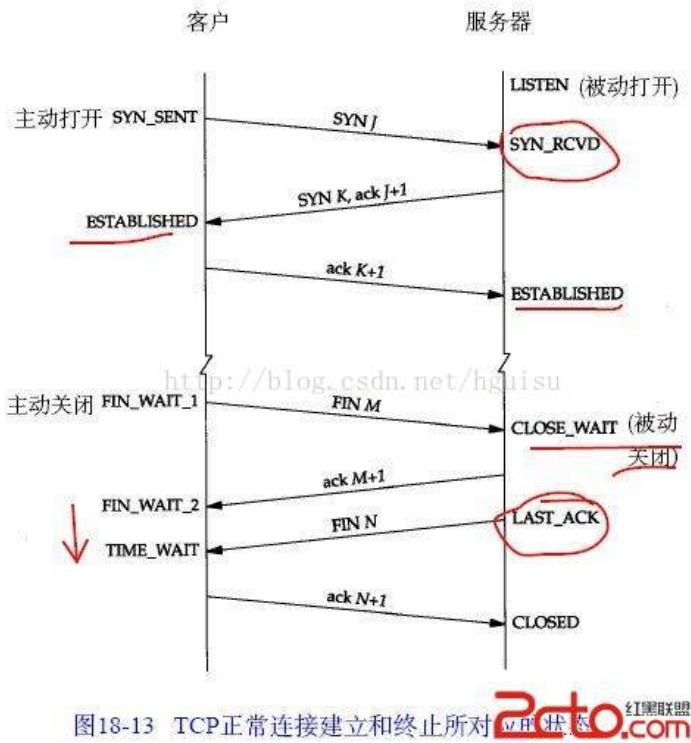


图18-13 TCP正常连接建立和终止所对

怎样牢牢地将这张图刻在脑中呢？那么你就一定要对这张图的每一个状态，及转换的过程有深刻的认识，不能只停留在一知半解之中。下面对这张图的11种状态详细解析一下，以便加强记忆！不过在这之前，先回顾一下TCP建立连接的三次握手过程，以及关闭连接的四次握手过程。

3、TCP连接建立三次握手

TCP是一个面向连接的协议，所以在连接双方发送数据之前，都需要首先建立一条连接。

Client连接Server:

当Client端调用socket函数调用时，相当于Client端产生了一个处于Closed状态的套接字。

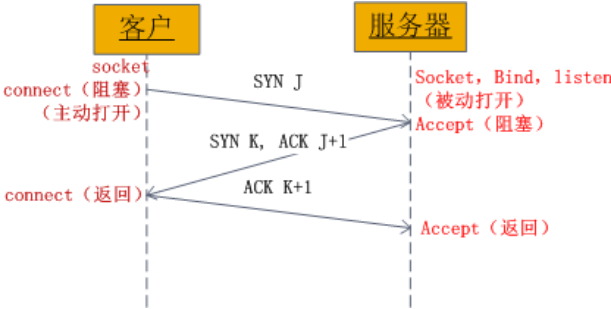
(1) 第一次握手：Client端又调用connect函数调用，系统为Client随机分配一个端口，连同传入connect中的参数(Server的IP和端口)，这就形成了一个连接四元组，客户端发送一个带SYN标志的TCP报文到服务器。这是三次握手过程中的报文1。connect调用让Client端的socket处于SYN_SENT状态，等待服务器确认；SYN：同步序列编号(Synchronize Sequence Numbers)。

(2) 第二次握手：服务器收到syn包，必须确认客户的SYN (ack=j+1)，同时自己也发送一个SYN包 (syn=k)，即SYN+ACK包，此时服务器进入SYN_RECV状态；

(3) 第三次握手：客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1)，此包发送完毕，客户端和服务器进入ESTABLISHED状态，完成三次握手。连接已经可以进行读写操作。

一个完整的三次握手也就是： 请求---应答---再次确认。

TCP协议通过三个报文段完成连接的建立，这个过程称为三次握手(three-way handshake)，过程如下图所示。
对应的函数接口：



2)Server

当Server端调用socket函数调用时，相当于Server端产生了一个处于Closed状态的监听套接字
Server端调用bind操作，将监听套接字与指定的地址和端口关联，然后又调用listen函数，系统会为其分配未完成队列和完成队列，此时的监听套接字可以接受Client的连接，监听套接字状态处于LISTEN状态。
当Server端调用accept操作时，会从完成队列中取出一个已经完成的client连接，同时在server这段会产生一个会话套接字，用于和client端套接字的通信，这个会话套接字的状态是ESTABLISH。

从图中可以看出，当客户端调用connect时，触发了连接请求，向服务器发送了SYN J包，这时connect进入阻塞状态；服务器监听到连接请求，即收到SYN J包，调用accept函数接收请求向客户端发送SYN K，ACK J+1，这时accept进入阻塞状态；客户端收到服务器的SYN K，ACK J+1之后，这时connect返回，并对SYN K进行确认；服务器收到ACK K+1时，accept返回，至此三次握手完毕，连接建立。

我们可以通过网络抓包的查看具体的流程：
比如我们服务器开启9502的端口。使用tcpdump来抓包：

```
tcpdump -iany tcp port 9502

然后我们使用telnet 127.0.0.1 9502开连接.:

telnet 127.0.0.1 9502

14:12:45.104687 IP localhost.39870 > localhost.9502: Flags [S], seq 2927179378, win 32792, options [mss 16396,sackOK,TS val 255474104 ecr 0,nop,wscale 3], length 0 (1)

14:12:45.104701 IP localhost.9502 > localhost.39870: Flags [S.], seq 1721825043, ack 2927179379, win 32768, options [mss 16396,sackOK,TS val 255474104 ecr 255474104,nop,wscale 3], length 0 (2)

14:12:45.104711 IP localhost.39870 > localhost.9502: Flags [.], ack 1, win 4099, options [nop,nop,TS val 255474104 ecr 255474104], length 0 (3)

14:13:01.415407 IP localhost.39870 > localhost.9502: Flags [P.], seq 1:8, ack 1, win 4099, options [nop,nop,TS val 255478182 ecr 255474104], length 7

14:13:01.415432 IP localhost.9502 > localhost.39870: Flags [.], ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 0

14:13:01.415747 IP localhost.9502 > localhost.39870: Flags [P.], seq 1:19, ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 18

14:13:01.415757 IP localhost.39870 > localhost.9502: Flags [.], ack 19, win 4097, options [nop,nop,TS val 255478182 ecr 255478182], length 0
```

我们看到 (1) (2) (3) 三步是建立tcp:

第一次握手:

14:12:45.104687 IP localhost.39870 > localhost.9502: Flags [S], seq 2927179378

客户端IP localhost.39870 (客户端的端口一般是自动分配的) 向服务器localhost.9502 发送syn包(syn=j)到服务器》syn的seq= 2927179378

第二次握手:

14:12:45.104701 IP localhost.9502 > localhost.39870: Flags [S.], seq 1721825043, ack 2927179379,

服务器收到syn包，必须确认客户的SYN (ack=j+1)，同时自己也发送一个SYN包 (syn=k)，即SYN+ACK包

SYN (ack=j+1) =ack 2927179379 服务器主机SYN包 (syn=seq 1721825043)

第三次握手:

14:12:45.104711 IP localhost.39870 > localhost.9502: Flags [.] ack 1,
客户端收到服务器的SYN+ACK包，向服务器发送确认包ACK(ack=k+1)
客户端和服务端进入ESTABLISHED状态后，可以进行通信数据交互。此时和accept接口没有关系，即使没有accepte，也进行3次握手完成。

连接出现连接不上问题，一般是网路出现问题或者网卡超负荷或者是连接数已经满啦。

紫色背景的部分：

IP localhost.39870 > localhost.9502: Flags [P.], seq 1:8, ack 1, win 4099, options [nop,nop,TS val 255478182 ecr 255474104], length 7

客户端向服务器发送长度为7个字节的数据，

IP localhost.9502 > localhost.39870: Flags [.] ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 0

服务器向客户确认已经收到数据

IP localhost.9502 > localhost.39870: Flags [P.], seq 1:19, ack 8, win 4096, options [nop,nop,TS val 255478182 ecr 255478182], length 18

然后服务器同时向客户端写入数据。

IP localhost.39870 > localhost.9502: Flags [.] ack 19, win 4097, options [nop,nop,TS val 255478182 ecr 255478182], length 0

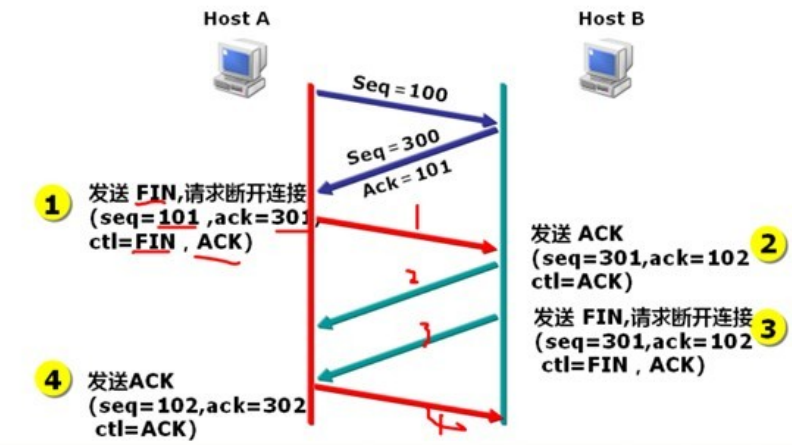
客户端向服务器确认已经收到数据

这个就是tcp可靠的连接，每次通信都需要对方来确认。

4. TCP连接的终止（四次握手释放）

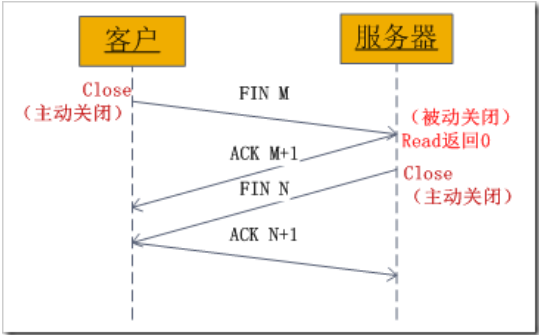
由于TCP连接是全双工的，因此每个方向都必须单独进行关闭。这原则是当一方完成它的数据发送任务后就能发送一个FIN来终止这个方向的连接。收到一个FIN只意味着这一方向上没有数据流动，一个TCP连接在收到一个FIN后仍能发送数据。首先进行关闭的一方将执行主动关闭，而另一方执行被动关闭。

建立一个连接需要三次握手，而终止一个连接要经过四次握手，这是由TCP的半关闭(half-close)造成的，如图：



- (1) 客户端A发送一个FIN，用来关闭客户端A到服务器B的数据传送（报文段4）。
- (2) 服务器B收到这个FIN，它发回一个ACK，确认序号为收到的序号加1（报文段5）。和SYN一样，一个FIN将占用一个序号。
- (3) 服务器B关闭与客户端A的连接，发送一个FIN给客户端A（报文段6）。
- (4) 客户端A发回ACK报文确认，并将确认序号设置为收到序号加1（报文段7）。

对应函数接口如图：



调用过程如下：

- 1) 当client想要关闭它与server之间的连接。client（某个应用进程）首先调用close主动关闭连接，这时TCP发送一个FIN M；client端处于FIN_WAIT1状态。
- 2) 当server端接收到FIN M之后，执行被动关闭。对这个FIN进行确认，返回给client ACK。当server端返回给client ACK后，client处于FIN_WAIT2状态，server处于CLOSE_WAIT状态。它的接收也作为文件结束符传递给应用进程，因为FIN的接收意味着应用进程在相应的连接上再也接收不到额外数据；
- 3) 一段时间之后，当server端检测到client端的关闭操作(read返回为0)。接收到文件结束符的server端调用close关闭它的socket。这导致server端的TCP也发送一个FIN N；此时server的状态为LAST_ACK。
- 4) 当client收到来自server的FIN后。client端的套接字处于TIME_WAIT状态，它会向server端再发送一个ack确认，此时server端收到ack确认后，此套接字处于CLOSED状态。

这样每个方向上都有一个FIN和ACK。

1. 为什么建立连接协议是三次握手，而关闭连接却是四次握手呢？

这是因为服务端的LISTEN状态下的SOCKET当收到SYN报文的建连请求后，它可以把ACK和SYN（ACK起应答作用，而SYN起同步作用）放在一个报文里来发送。但关闭连接时，当收到对方的FIN报文通知时，它仅仅表示对方没有数据发送给你了；但未必你所有的数据都全部发送给对方了，所以你可以未必会马上会关闭SOCKET,也即你可能还需要发送一些数据给对方之后，再发送FIN报文给对方来表示你同意现在可以关闭连接了，所以它这里的ACK报文和FIN报文多数情况下都是分开发送的。

2. 为什么TIME_WAIT状态还需要等2MSL后才能返回到CLOSED状态？

这是因为虽然双方都同意关闭连接了，而且握手的4个报文也都协调和发送完毕，按理可以直接回到CLOSED状态（就好比从SYN_SEND状态到ESTABLISH状态那样）：

一方面是可靠的实现TCP全双工连接的终止，也就是当最后的ACK丢失后，被动关闭端会重发FIN，因此主动关闭端需要维持状态信息，以允许它重新发送最终的ACK。

另一方面，但是因为我们必须要假想网络是不可靠的，你无法保证你最后发送的ACK报文会一定被对方收到，因此对方处于LAST_ACK状态下的SOCKET可能会因为超时未收到ACK报文，而重发FIN报文，所以这个TIME_WAIT状态的作用就是用来重发可能丢失的ACK报文。

TCP在2MSL等待期间，定义这个连接(4元组)不能再使用，任何迟到的报文都会丢弃。设想如果没有2MSL的限制，恰好新到的连接正好满足原先的4元组，这时候连接就可能接收到网络上的延迟报文就可能干扰最新建立的连接。

5、同时打开

两个应用程序同时执行主动打开的情况是可能的，虽然发生的可能性较低。每一端都发送一个SYN,并传递给对方，且每一端都使用对端所知的端口作为本地端口。例如：

主机a中一应用程序使用7777作为本地端口，并连接到主机b 8888端口做主动打开。

主机b中一应用程序使用8888作为本地端口，并连接到主机a 7777端口做主动打开。

tcp协议在遇到这种情况时，只会打开一条连接。

这个连接的建立过程需要4次数据交换，而一个典型的连接建立只需要3次交换（即3次握手）

但多数伯克利版的tcp/ip实现并不支持同时打开。

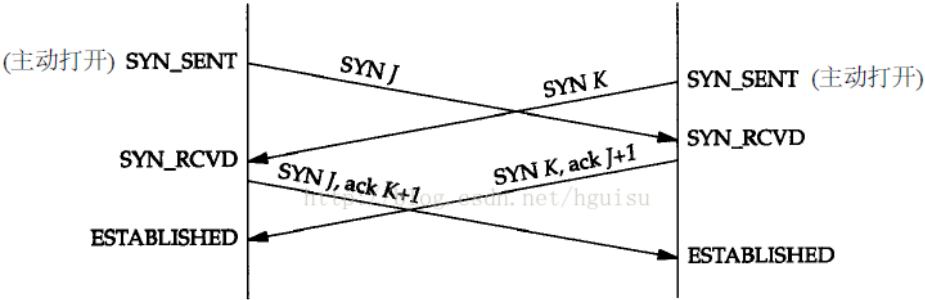


图18-17 同时打开期间报文段的交换

6、同时关闭

如果应用程序同时发送FIN，则在发送后会首先进入FIN_WAIT_1状态。在收到对端的FIN后，回复一个ACK，会进入CLOSING状态。在收到对端的ACK后，进入TIME_WAIT状态。这种情况称为同时关闭。

同时关闭也需要有4次报文交换，与典型的关闭相同。

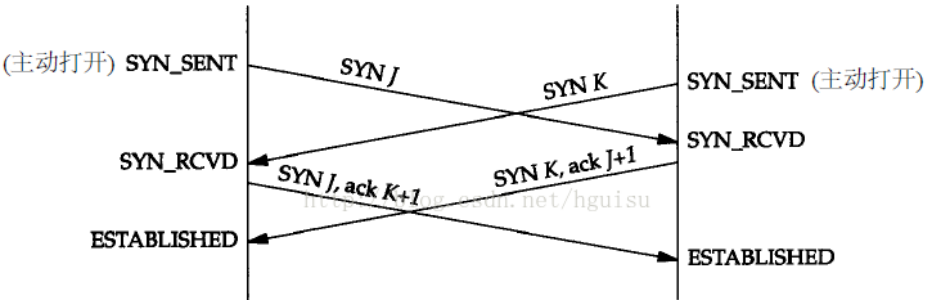


图18-17 同时打开期间报文段的交换

7. TCP通信中服务器处理客户端意外断开

引用地址：<http://blog.csdn.net/klkklkxiaofei/article/details/12966407>

如果TCP连接被对方正常关闭，也就是说，对方是正确地调用了closesocket(s)或者shutdown(s)的话，那么上面的Recv或Send调用就能马上返回，并且报错。这是由于close socket(s)或者shutdown(s)有个正常的关闭过程，会告诉对方“TCP连接已经关闭，你不需要再发送或者接受消息了”。

但是，如果意外断开，客户端（3g的移动设备）并没有正常关闭socket。双方并未按照协议上的四次挥手去断开连接。

那么这时候正在执行Recv或Send操作的一方就会因为没有任何连接中断的通知而一直等待下去，也就是会被长时间卡住。

像这种如果一方已经关闭或异常终止连接，而另一方却不知道，我们将这样的TCP连接称为半打开的。

解决意外中断办法都是利用保活机制。而保活机制分又可以让底层实现也可自己实现。

1、自己编写心跳包程序

简单的说也就是在自己的程序中加入一条线程，定时向对端发送数据包，查看是否有ACK，如果有则连接正常，没有的话则连接断开

2、启动TCP编程里的keepAlive机制

一、双方拟定心跳（自实现）

一般由客户端发送心跳包，服务端并不回应心跳，只是定时轮询判断一下与上次的时间间隔是否超时（超时时间自己设定）。服务器并不主动发送是不想增添服务器的通信量，减少压力。

但这会出现三种情况：

情况1.

客户端由于某种网络延迟等原因很久后才发送心跳（它并没有断），这时服务器若利用自身设定的超时判断其已经断开，而后去关闭socket。若客户端有重连机制，则客户端会重新连接。若不确定这种方式是否关闭了原本正常的客户端，则在ShutDown的时候一定要选择send,表示关闭发送通道，服务器还可以接收一下，万一客户端正在发送比较重要的数据呢，是不？

情况2.

客户端很久没传心跳，确实是自身断掉了。在其重启之前，服务端已经判断出其超时，并主动close，则四次挥手成功交互。

情况3.

客户端很久没传心跳，确实是自身断掉了。在其重启之前，服务端的轮询还未判断出其超时，在未主动close的时候该客户端已经重新连接。

这时候若客户端断开的时候发送了FIN包，则服务端将会处于CLOSE_WAIT状态；

这时候若客户端断开的时候未发送FIN包，则服务端处还是显示ESTABLISHED状态；

而新连接上来的客户端（也就是刚才断掉的重新连上来了）在服务端肯定是ESTABLISHED;这时候就有个问题，若利用轮询还未检测出上条旧连接已经超时（这很正常，timer总有个间隔吧），而在这时，客户端又重复的上演情况3，那么服务端将会出现大量的假的ESTABLISHED连接和CLOSE_WAIT连接。

最终结果就是新的其他客户端无法连接上来，但是利用netstat还是能看到一条连接已经建立，并显示ESTABLISHED，但始终无法进入程序代码。个人最初感觉导致这种情况是因为假的ESTABLISHED连接和CLOSE_WAIT连接会占用较大的系统资源，程序无法再次创建连接（因为每次我发现这个问题的时候我只连了10个左右客户端却已经有40多条无效连接）。而最近几天测试却发现有一次程序内只连接了2, 3个设备，但是有8条左右的虚连接，此时已经连接不了新客户端了。这时候我就觉得我想错了，不可能这几条连接就占用了大量连接把，如果说几十条还有可能。但是能肯定的是，这个问题的产生绝对是设备在不停的重启，而服务器这边又是简单的轮询，并不能及时处理，暂时还未能解决。

二、利用KeepAlive

其实keepalive的原理就是TCP内嵌的一个心跳包，

以服务器端为例，如果当前server端检测到超过一定时间（默认是 7,200,000 milliseconds，也就是2个小时）没有数据传输，那么会向client端发送一个keep-alive packet（该keep-alive packet就是ACK和当前TCP序列号减一的组合），此时client端应该为以下三种情况之一：

1. client端仍然存在，网络连接状况良好。此时client端会返回一个ACK。server端接收到ACK后重置计时器（复位存活定时器），在2小时后再发送探测。如果2小时内连接上有数据传输，那么在该时间基础上向后推延2个小时。

2. 客户端异常关闭，或是网络断开。在这两种情况下，client端都不会响应。服务器没有收到对其发出探测的响应，并且在一定时间（系统默认为1000 ms）后重复发送keep-alive packet，并且重复发送一定次数（2000 XP 2003 系统默认为5次，Vista后的系统默认为10次）。

3. 客户端曾经崩溃，但已经重启。这种情况下，服务器将会收到对其存活探测的响应，但该响应是一个复位，从而引起服务器对连接的终止。

对于应用程序来说，2小时的空闲时间太长。因此，我们需要手工开启Keepalive功能并设置合理的Keepalive参数。

全局设置可更改/etc/sysctl.conf, 加上：

```
net.ipv4.tcp_keepalive_intvl = 20
net.ipv4.tcp_keepalive_probes = 3
net.ipv4.tcp_keepalive_time = 60
```

在程序中设置如下:

```
[cpp] view plain copy print ?
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <sys/types.h>
#include <netinet/tcp.h>

int keepAlive = 1; // 开启keepalive属性
int keepIdle = 60; // 如该连接在60秒内没有任何数据往来,则进行探测
int keepInterval = 5; // 探测时发包的时间间隔为5 秒
int keepCount = 3; // 探测尝试的次数.如果第1次探测包就收到响应了,则后2次的不再发.

setsockopt(rs, SOL_SOCKET, SO_KEEPALIVE, (void *)&keepAlive, sizeof(keepAlive));
setsockopt(rs, SOL_TCP, TCP_KEEPIRL, (void *)&keepIdle, sizeof(keepIdle));
setsockopt(rs, SOL_TCP, TCP_KEEPIRL, (void *)&keepInterval, sizeof(keepInterval));
setsockopt(rs, SOL_TCP, TCP_KEEPCNT, (void *)&keepCount, sizeof(keepCount));
```

在程序中表现为,当tcp检测到对端socket不再可用时(不能发出探测包,或探测包没有收到ACK的响应包),select会返回socket可读,并且在recv时返回-1,同时置上errno为ETIMEDOUT.

8. Linux错误信息(errno)列表

经常出现的错误:

22: 参数错误,比如ip地址不合法,没有目标端口等

101: 网络不可达,比如不能ping通

111: 链接被拒绝,比如目标关闭链接等

115: 当链接设置为非阻塞时,目标没有及时应答,返回此错误,socket可以继续使用。比如socket连接

附录: Linux的错误码表(errno table)

```
_ 124 EMEDIUMTYPE_ Wrong medium type
_ 123 ENOMEDIUM__ No medium found
_ 122 EDQUOT___ Disk quota exceeded
_ 121 EREMOTEIO__ Remote I/O error
_ 120 EISNAM___ Is a named type file
_ 119 ENAVAIL___ No XENIX semaphores available
_ 118 ENOTNAM___ Not a XENIX named type file
_ 117 EUCLEAN___ Structure needs cleaning
_ 116 ESTALE___ Stale NFS file handle
_ 115 EINPROGRESS +Operation now in progress
```

操作正在进行中。一个阻塞的操作正在执行。

```
_ 114 EALREADY___ Operation already in progress
_ 113 EHOSTUNREACH No route to host
_ 112 EHOSTDOWN___ Host is down
_ 111 ECONNREFUSED Connection refused
```

1、拒绝连接。一般发生在连接建立时。

按服务器端网线测试,客户端设置keep alive时,recv较快返回0,先收到ECONNREFUSED (Connection refused)错误码,其后都是ETIMEDOUT。

2、an error returned from connect(), so it can only occur in a client (if a client is defined as the party that initiates the connection)

- _ 110 ETIMEDOUT_ +Connection timed out
- _ 109 ETOOMANYREFS Too many references: cannot splice
- _ 108 ESHUTDOWN_ Cannot send after transport endpoint shutdown
- _ 107 ENOTCONN_ Transport endpoint is not connected

在一个没有建立连接的socket上，进行read，write操作会返回这个错误。出错的原因是socket没有标识地址。Setsoc也可能出错。

还有一种情况就是收到对方发送过来的RST包，系统已经确认连接被断开了。

_ 106 EISCONN_ Transport endpoint is already connected

一般是socket客户端已经连接了，但是调用connect，会引起这个错误。



buffer space available
Connection reset by peer

以下几种原因：远程主机停止服务，重新启动;当在执行某些操作时遇到失败，因为设置连接被关闭，一般与ENETRESET一起出现。

中，客户端异常退出，并没有回收关闭相关的资源，服务器端会先收到ECONNRESET错误，然后收到EISCONN错误。

- 2、连接被远程主机关闭。有以下几种原因：远程主机停止服务，重新启动;当在执行某些操作时遇到失败，因为设置了“keep alive”选项，连接被关闭，一般与ENETRESET一起出现。
- 3、远程端执行了一个“hard”或者“abortive”的关闭。应用程序应该关闭socket，因为它不再可用。当执行在一个UDP socket上时，这个错误表明前一个send操作返回一个ICMP “port unreachable”信息。
- 4、如果client关闭连接，server端的select并不出错(不返回-1，使用select对唯一一个socket进行non- blocking检测)，但是写该socket就会出错。错误号:ECONNRESET。读(recv) socket并没有返回错误。
- 5、该错误被描述为“connection reset by peer”，即“对方复位连接”，这种情况一般发生在服务进程较客户进程提前终止。当服务进程提前终止，客户 TCP 发送 FIN 分节，客户 TCP 回应 ACK，服务 TCP 将转入 FIN_WAIT2 状态。此时如果客户进程没有处理该 FIN （如阻塞在其它调用上而没有关闭 Socket 时），则客户 TCP 将处于 CLOSE_WAIT 状态。当客户进程再次向 FIN_WAIT2 状态的服务 TCP 发送数据时，则服务 TCP 将立刻响应 RST。一般来说，这种情况还可以会引发另外的应用程序异常，客户进程在发送完数据后，往往会等待从网络IO接收数据，很典型的如 read 或 readline 调用，此时由于执行时序的原因，如果该调用发生在 RST 分节收到前执行的话，那么结果是客户进程会得到一个非预期的 EOF 错误。此时一般会输出“server terminated prematurely” — “服务器过早终止”错误。

_ 103 ECONNABORTED Software caused connection abort

- 1、软件导致的连接取消。一个已经建立的连接被host方的软件取消，原因可能是数据传输超时或者是协议错误。
- 2、该错误被描述为“software caused connection abort”，即“软件引起的连接中止”。原因在于当服务和客户进程在完成用于 TCP 连接的“三次握手”后，客户 TCP 却发送了一个 RST （复位）分节，在服务进程看来，就在该连接已由 TCP 排队，等着服务进程调用 accept 的时候 RST 却到达了。POSIX 规定此时的 errno 值必须 ECONNABORTED。源自 Berkeley 的实现完全在内核中处理中止的连接，服务进程将永远不知道该中止的发生。服务器进程一般可以忽略该错误，直接再次调用accept。

当TCP协议接收到RST数据段，表示连接出现了某种错误，函数read将以错误返回，错误类型为ECONNRESET。并且以后所有在这个套接字上的读操作均返回错误。错误返回时返回值小于0。

_ 102 ENETRESET_ Network dropped connection on reset

网络重置时丢失连接。

由于设置了“keep-alive”选项，探测到一个错误，连接被中断。在一个已经失败的连接上试图使用setsockopt操作，也会返回这个错误。

_ 101 ENETUNREACH_ Network is unreachable

网络不可达。Socket试图操作一个不可达的网络。这意味着local的软件知道没有路由到达远程的host。

- _ 100 ENETDOWN_ Network is down
- _ 99 EADDRNOTAVAIL Cannot assign requested address
- _ 98 EADDRINUSE_ Address already in use

- _ 97 EAFNOSUPPORT Address family not supported by protocol
- _ 96 EPFNOSUPPORT Protocol family not supported
- _ 95 EOPNOTSUPP_ Operation not supported
- _ 94 ESOCKTNOSUPPORT Socket type not supported

Socket类型不支持。指定的socket类型在其address family中不支持。如可选项SOCK_RAW，但实现并不支持SOCK_RAW sockets。

- _ 93 EPROTONOSUPPORT Protocol not supported

不支持的协议。系统中没有安装标识的协议，或者没有实现。如函数需要SOCK_DGRAM socket，但是标识了stream protocol。

- _ 92 ENOPROTOPT_ Protocol not available

该错误不是一个 Socket 连接相关的错误。errno 给出该值可能由于，通过 getsockopt 系统调用来获得一个套接字的当前选项状态时，如果发现了系统不支持的选项参数就会引发该错误。

- _ 91 EPROTOTYPE_ Protocol wrong type for socket

协议类型错误。标识了协议的Socket函数在不支持的socket上进行操作。如ARPA Internet UDP协议不能被标识为SOCK_STREAM socket类型。

- _ 90 EMSGSIZE_ +Message too long

消息体太长。

发送到socket上的一个数据包大小比内部的消息缓冲区大，或者超过别的网络限制，或是用来接收数据包的缓冲区比数据包本身小。

- _ 89 EDESTADDRREQ Destination address required

需要提供目的地址。

在一个socket上的操作需要提供地址。如往一个ADDR_ANY 地址上进行sendto操作会返回这个错误。

- _ 88 ENOTSOCK_ Socket operation on non-socket

在非socket上执行socket操作。

- _ 87 EUSERS_ Too many users
- _ 86 ESTRPIPE_ Streams pipe error
- _ 85 ERESTART_ Interrupted system call should be restarted
- _ 84 EILSEQ_ Invalid or incomplete multibyte or wide character
- _ 83 ELIBEXEC_ Cannot exec a shared library directly
- _ 82 ELIBMAX_ Attempting to link in too many shared libraries
- _ 81 ELIBSCN_ .lib section in a.out corrupted
- _ 80 ELIBBAD_ Accessing a corrupted shared library
- _ 79 ELIBACC_ Can not access a needed shared library
- _ 78 EREMCHG_ Remote address changed
- _ 77 EBADFD_ File descriptor in bad state
- _ 76 ENOTUNIQ_ Name not unique on network
- _ 75 EOVERFLOW_ Value too large for defined data type
- _ 74 EBADMSG_ +Bad message
- _ 73 EDOTDOT_ RFS specific error
- _ 72 EMULTIHOP_ Multihop attempted
- _ 71 EPROTO_ Protocol error
- _ 70 ECOMM_ Communication error on send
- _ 69 ESRMNT_ Srmount error
- _ 68 EADV_ Advertise error
- _ 67 ENOLINK_ Link has been severed
- _ 66 EREMOTE_ Object is remote

_ 65 ENOPKG__ Package not installed
_ 64 ENONET__ Machine is not on the network
_ 63 ENOSR__ Out of streams resources
_ 62 ETIME__ Timer expired
_ 61 ENODATA__ No data available
_ 60 ENOSTR__ Device not a stream
_ 59 EBFONT__ Bad font file format
_ 57 EBADSLT__ Invalid slot
_ 56 EBADRQC__ Invalid request code
_ 55 ENOANO__ No anode
_ 54 EXFULL__ Exchange full
_ 53 EBADR__ Invalid request descriptor
_ 52 EBADE__ Invalid exchange
_ 51 EL2HLT__ Level 2 halted
_ 50 ENOCSI__ No CSI structure available
_ 49 EUNATCH__ Protocol driver not attached
_ 48 ELNRNG__ Link number out of range
_ 47 EL3RST__ Level 3 reset
_ 46 EL3HLT__ Level 3 halted
_ 45 EL2NSYNC__ Level 2 not synchronized
_ 44 ECHRNG__ Channel number out of range
_ 43 EIDRM__ Identifier removed
_ 42 ENOMSG__ No message of desired type
_ 40 ELOOP__ Too many levels of symbolic links
_ 39 ENOTEMPTY_ +Directory not empty
_ 38 ENOSYS__ +Function not implemented
_ 37 ENOLCK__ +No locks available
_ 36 ENAMETOOLONG +File name too long
_ 35 EDEADLK__ +Resource deadlock avoided
_ 34 ERANGE__ +Numerical result out of range
_ 33 EDOM__ +Numerical argument out of domain
_ 32 EPIPE__ +Broken pipe

接收端关闭(缓冲中没有多余的数据),但是发送端还在write:

1、Socket 关闭,但是socket号并没有置-1。继续在此socket上进行send和recv,就会返回这种错误。这个错误会引发SIGPIPE信号,系统会将产生此EPIPE错误的进程杀死。所以,一般在网络程序中,首先屏蔽此消息,以免发生不及时设置socket进程被杀死的情况。

2、write(..) on a socket that has been closed at the other end will cause a SIGPIPE.

3、错误被描述为“broken pipe”,即“管道破裂”,这种情况一般发生在客户进程不理睬(或未及时处理)Socket 错误,继续向服务 TCP 写入更多数据时,内核将向客户进程发送 SIGPIPE 信号,该信号默认会使进程终止(此时该前台进程未进行 core dump)。结合上边的 ECONNRESET 错误可知,向一个 FIN_WAIT2 状态的服务 TCP(已 ACK 响应 FIN 分节)写入数据不成问题,但是写一个已接收了 RST 的 Socket 则是一个错误。

_ 31 EMLINK__ +Too many links
_ 30 EROFS__ +Read-only file system
_ 29 ESPIPE__ +Illegal seek
_ 28 ENOSPC__ +No space left on device
_ 27 EFBIG__ +File too large
_ 26 ETXTBSY__ Text file busy
_ 25 ENOTTY__ +Inappropriate ioctl for device
_ 24 EMFILE__ +Too many open files

打开了太多的socket。对进程或者线程而言,每种实现方法都有一个最大的可用socket数目处理,或者是全局的,或者是局部的。

_ 23 ENFILE__ +Too many open files in system
_ 22 EINVAL__ +Invalid argument

无效参数。提供的参数非法。有时也会与socket的当前状态相关,如一个socket并没有进入listening状态,此时调

用accept，就会产生EINVAL错误。

- _ 21 EISDIR__ +Is a directory
- _ 20 ENOTDIR__ +Not a directory
- _ 19 ENODEV__ +No such device
- _ 18 EXDEV__ +Invalid cross-device link
- _ 17 EEXIST__ +File exists
- _ 16 EBUSY__ +Device or resource busy
- _ 15 ENOTBLK__ Block device required
- _ 14 EFAULT__ +Bad address地址错误
- _ 13 EACCES__ +Permission denied
- _ 12 ENOMEM__ +Cannot allocate memory
- _ 11 EAGAIN__ +Resource temporarily unavailable

在读数据的时候,没有数据在底层缓冲的时候会遇到,一般的处理是循环进行读操作,异步模式还会等待读事件的发生再读

- 1、Send返回值小于要发送的数据数目，会返回EAGAIN和EINTR。
- 2、recv 返回值小于请求的长度时说明缓冲区已经没有可读数据，但再读不一定会触发EAGAIN，有可能返回0表示TCP连接已被关闭。
- 3、当socket是非阻塞时,如返回此错误,表示写缓冲队列已满,可以做延时后再重试.
- 4、在Linux进行非阻塞的socket接收数据时经常出现Resource temporarily unavailable, errno代码为11 (EAGAIN)，表明在非阻塞模式下调用了阻塞操作，在该操作没有完成就返回这个错误，这个错误不会破坏socket的同步，不用管它，下次循环接着recv就可以。对非阻塞socket而言，EAGAIN不是一种错误。

- _ 10 ECHILD__ +No child processes
- _ 9 EBADF__ +Bad file descriptor
- _ 8 ENOEXEC__ +Exec format error
- _ 7 E2BIG__ +Argument list too long
- _ 6 ENXIO__ +No such device or address
- _ 5 EIO__ +Input/output error
- _ 4 EINTR__ +Interrupted system call

阻塞的操作被取消阻塞的调用打断。如设置了发送接收超时，就会遇到这种错误。

只能针对阻塞模式的socket。读，写阻塞的socket时，-1返回，错误号为INTR。另外，如果出现EINTR即errno为4，错误描述Interrupted system call，操作也应该继续。如果recv的返回值为0，那表明连接已经断开，接收操作也应该结束。

- _ 3 ESRCH__ +No such process
- _ 2 ENOENT__ +No such file or directory
- _ 1 EPERM__ +Operation not permitted

顶 4 踩 1

上一篇 专属个人的聊天机器人的实现——图灵机器人

下一篇 使用 libevent 和 libev 提高网络应用性能——I/O模型演进变化史

我的同类文章

c/c++（22）	socket网络编程（7）

• 位运算符及其应用	2012-08-21	阅读 7052	• 海量数据处理算法—Bit-Map	2012-08-21	阅读 28064
• B-树和B+树的应用：数据搜...	2012-07-29	阅读 42576	• 查找 -数据结构	2012-07-24	阅读 7373
• 八大排序算法	2012-07-23	阅读 432630	• 图的应用详解-数据结构	2012-07-15	阅读 14215
• 图的遍历 - 数据结构	2012-07-04	阅读 12429	• 回溯法 -数据结构与算法	2012-07-02	阅读 10109
• 二叉树的应用详解 - 数据结构	2012-06-25	阅读 12895	• 二叉树遍历 - 数据结构	2012-06-21	阅读 2872
更多文章					

参考知识库



.NET 知识库
1189 关注 | 798 收录



Linux 知识库
5215 关注 | 3227 收录

猜你在找

windows server 2008 r2服务器操作系统	TCP连接的状态详解以及故障排查
全网服务器数据备份解决方案案例实践	TCP连接的状态详解以及故障排查
Windows Server 2012 DHCP Server 管理	TCP连接的状态详解以及故障排查
Qt网络编程实战之HTTP服务器	TCP连接的状态详解以及故障排查
1. 16. ARM裸机第十六部分-shell原理和问答机制引入	TCP连接的状态详解以及故障排查



摊儿货 尖儿货

应有尽有

11月1-11日

 **¥11.11** 每日等你秒不停

查看评论

7楼 我爱默小兜 4天前 16:50发表



总结得非常好

6楼 newjueqi 2016-04-14 18:09发表



写得太好了，排查故障时特别有用。
请教博主一个问题：
一台机器只要内存足够的话，支持百万级连接都不是问题。现在linux可以根据不同的远端 ip:port, 决定是否重用某个本地端口，这个重用策略是否在某个tcp状态才生效？

Re: 真实的归宿 2016-04-21 11:13发表



回复newjueqi: 现在linux可以根据不同的远端 ip:port, 决定是否重用某个本地端口，这个重用策略是否在某个tcp状态才生效？
这个描述我不太明白. 可以举例吗？

5楼 曹学亮 2016-02-29 17:28发表



网络知识很重要啊

4楼 荒唐的timeFly 2015-11-06 11:10发表



很好，很详细

3楼 Jack_Simple 2015-04-01 16:18发表



写的很详细，比书中介绍写的好

2楼 Jack_Simple 2015-04-01 16:18发表



写的很详细，比书中介绍写的好

1楼 呆雁 2014-12-30 15:58发表




挺详细的，有用。

发表评论

用户名: chenyongsuda

评论内容:



提交

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

- [全部主题](#)[Hadoop](#)[AWS](#)[移动游戏](#)[Java](#)[Android](#)[iOS](#)[Swift](#)[智能硬件](#)[Docker](#)[OpenStack](#)[VPN](#)[Spark](#)[ERP](#)[IE10](#)[Eclipse](#)[CRM](#)[JavaScript](#)[数据库](#)[Ubuntu](#)[NFC](#)[WAP](#)[jQuery](#)[BI](#)[HTML5](#)[Spring](#)[Apache](#)[.NET](#)[API](#)[HTML](#)[SDK](#)[IIS](#)[Fedora](#)[XML](#)[LBS](#)[Unity](#)[Splashtop](#)[UML](#)[components](#)[Windows Mobile](#)[Rails](#)[QEMU](#)[KDE](#)[Cassandra](#)[CloudStack](#)[FTC](#)[coremail](#)[OPhone](#)[CouchBase](#)[云计算](#)[iOS6](#)[Rackspace](#)[Web App](#)[SpringSide](#)[Maemo](#)[Compuware](#)[大数据](#)[aptech](#)[Perl](#)[Tornado](#)[Ruby](#)[Hibernate](#)[ThinkPHP](#)[HBase](#)[Pure](#)[Solr](#)[Angular](#)[Cloud Foundry](#)[Redis](#)[Scala](#)[Django](#)[Bootstrap](#)