Linux kernel tcp 连接建立详解

——由 listen 系统调用的 backlog 参数引发的长篇大论

目录

第 0 章 本文的目的

一直以来，对 tcp 的连接的建立过程只停留在三次握手的层面，阅读代码时发现好多逻辑流程不是很清楚。还有就是对系统调用 int listen(int sockfd, int backlog)的第二个参数 backlog 似懂非懂。它到底是限制 server 端能同时处理的连接请求数量，还是限制同时建立的连接数量，还是限制等待建立连接队列的长度？

代码面前，了无秘密，关键是得探求。如果你也有同样的困惑，不妨读读本文。

我分析的是 v3.12-rc4 的代码，commit 0e7a3ed04f0c 。本文偏重于连接建立的函数调用过程、sock 状态变化，以及 backlog 参数，其他的也很重要的内容，如 tcp 序号、拥塞控制等，但是本文从略，否则可以写本书了。并且本文只关注 tcp 层，IP 层及以下也忽略。

第 1 章是简单的 client/server 程序，如果你对第 1 章中的程序行为全部理解，那么恭喜你，完全可以略过本文。

第 1 章 listen()系统调用的困惑

先编译附录 1 和附录 2 的 client 和 server 程序，然后准备一打终端，我们开始实验。

1.1 前提条件

cat /proc/sys/net/ipv4/tcp_abort_on_overflow，确保其值为 0。在我的 fedora 14 上这是默认值。

1.2 分析 client 程序

client 程序很简单，主动向 127.0.0.1 的 20000 端口建立 tcp 连接，如果连接建立成功，会打印“Connected to server: IP 127.0.0.1 PORT 20000”字样。如果不成功，会打印“CONNECT FAILED !!!!!!”，然后跟上详细的错误原因。

1.3 分析 server 程序

server 程序的大体结构与普通的服务期端程序很类似，有两点需要注意：

1) LENGTH_OF_LISTEN_QUEUE 的值设为 1，即 listen()系统调用的 backlog 参数设置为 1，我们看看这个 backlog 究竟会有什么样的作用；
2) 2 TEST_LISTEN_BACKLOG 的值设为 1，意味着 listen()之后始终不调用 accept()。server 端会在 for 循环中一直运行，不会调用下面的 accept()系统调用了。这样就会把 client 的连接请求阻塞在这个 for 循环中，从而便于分析 backlog 是如何限制连接数量的。

1.4 第一次观察程序行为

1 在第一个终端运行./server

发现屏幕每隔 1 秒钟打印一行“server is listening! ”

2 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 10.66.17.97:33577       10.66.127.11:631        ESTABLISHED
```

其中红色的行表示 server 端正在 20000 端口等待连接。

3 在第三个终端运行./client

屏幕显示“connected to server: IP 127.0.0.1 PORT 20000”，表示连接成功。

4 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 10.66.17.97:33577       10.66.127.11:631        ESTABLISHED
tcp       13      0 127.0.0.1:20000         127.0.0.1:45484         ESTABLISHED
tcp        0      0 127.0.0.1:45484         127.0.0.1:20000         ESTABLISHED
```

绿色的两行，表示一对新的 tcp 连接已经建立了。

5 在第四个终端运行./client

屏幕显示“connected to server: IP 127.0.0.1 PORT 20000”，表示连接成功。

6 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
```

```
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp       13      0 127.0.0.1:20000         127.0.0.1:45485         ESTABLISHED
tcp        0      0 10.66.17.97:33577       10.66.127.11:631        ESTABLISHED
tcp       13      0 127.0.0.1:20000         127.0.0.1:45484         ESTABLISHED
tcp        0      0 127.0.0.1:45485         127.0.0.1:20000         ESTABLISHED
tcp        0      0 127.0.0.1:45484         127.0.0.1:20000         ESTABLISHED
```

同样，绿色的两行表示一对新建立的 tcp 连接。

等等，listen()系统调用的参数 backlog 不是 1 么，怎么能在第三个终端和第四个终端运行./client 都显示连接成功呢？由此推断，即使 backlog 的值为 1，也允许 client 端与 server 端先后建立两个连接。

7 在第五个终端运行./client

屏幕显示“connected to server: IP 127.0.0.1 PORT 20000”，表示连接成功。

嗯？backlog 到底能限制什么呢？

8 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:20000         127.0.0.1:33769         SYN_RECV
tcp        0     13 127.0.0.1:33769         127.0.0.1:20000         ESTABLISHED
tcp       13      0 127.0.0.1:20000         127.0.0.1:45485         ESTABLISHED
tcp        0      0 10.66.17.97:33577       10.66.127.11:631        ESTABLISHED
tcp       13      0 127.0.0.1:20000         127.0.0.1:45484         ESTABLISHED
tcp        0      0 127.0.0.1:45485         127.0.0.1:20000         ESTABLISHED
tcp        0      0 127.0.0.1:45484         127.0.0.1:20000         ESTABLISHED
```

有变化了！红色的两行是新的。什么意思呢？它表示对于 client 端来说，状态为 ESTABLISHED，即 client 端认为连接已经建立了。

但是对于 server 端来说，连接仍处于 SYN_RECV 状态，表示 server 端认为连接还未完成呢？哎

呀呀，这是什么情况？

9 在第二个终端运行 tcpdump -i lo -p tcp -nn

15:08:09.923809 IP 127.0.0.1.33769 > 127.0.0.1.20000: Flags [S], seq 495577845, win 32792, options [mss 16396,sackOK,TS val 795517246 ecr 0,nop,wscale 6], length 0

15:08:09.923842 IP 127.0.0.1.20000 > 127.0.0.1.33769: Flags [S.], seq 501188914, ack 495577846, win 32768, options [mss 16396,sackOK,TS val 795517246 ecr 795517246,nop,wscale 6], length 0

15:08:09.923861 IP 127.0.0.1.33769 > 127.0.0.1.20000: Flags [.], ack 1, win 513, options [nop,nop,TS val 795517246 ecr 795517246], length 0

15:08:11.924175 IP 127.0.0.1.33769 > 127.0.0.1.20000: Flags [P.], seq 1:14, ack 1, win 513, options [nop,nop,TS val 795519246 ecr 795517246], length 13

15:08:12.128861 IP 127.0.0.1.33769 > 127.0.0.1.20000: Flags [P.], seq 1:14, ack 1, win 513, options [nop,nop,TS val 795519451 ecr 795517246], length 13

15:08:12.539864 IP 127.0.0.1.33769 > 127.0.0.1.20000: Flags [P.], seq 1:14, ack 1, win 513, options [nop,nop,TS val 795519862 ecr 795517246], length 13

15:08:13.361860 IP 127.0.0.1.33769 > 127.0.0.1.20000: Flags [P.], seq 1:14, ack 1, win 513, options [nop,nop,TS val 795520684 ecr 795517246], length 13

15:08:14.333869 IP 127.0.0.1.20000 > 127.0.0.1.33769: Flags [S.], seq 501188914, ack 495577846, win 32768, options [mss 16396,sackOK,TS val 795521656 ecr 795520684,nop,wscale 6], length 0

前三行没有任何异样，三次握手的包一个不少。第 4567 行，既然 client 认为连接已经完成了，那么就连续向 server 端发送四次数据，每次 13 个字节。但是第 8 行最奇怪了，server 端竟然向 client 端发送了一个 SYN/ACK 包，请求建立连接。这个肯定是 backlog 为 1 在作怪！

10 继续在新的终端运行./client，结果与第 7 步的类似。看来超过两个连接之后的行为都是类似的。

1.5 第二次观察程序行为

我们先把所有 client 端终止，再把 server 端终止，然后把 server.c 中的 TEST_LISTEN_BACKLOG 设置为 0，重新编译。这样 server 端不会一直在 for 循环中运行了，会调用 accept()系统调用处理连接请求。

1 在第一个终端运行./server

显示“server is listening!”

2 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
```

```
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 10.66.17.97:33577       10.66.127.11:631        ESTABLISHED
```

红色部分表示 server 端正在等待连接。

3 在第三个终端运行./client

connected to server: IP 127.0.0.1 PORT 20000

length=41

recv from server: hello, world! : Thu Oct 17 15:25:37 2013

每隔 2 秒会有新的输出。

同时 server 端也会有新的输出：

a client connected: IP 127.0.0.1 PORT 59645

hello, world! : Thu Oct 17 15:25:37 2013

这表示 client 与 server 正在正常通信呢。

4 在第四个终端运行./client

connected to server: IP 127.0.0.1 PORT 20000

length=41

recv from server: hello, world! : Thu Oct 17 15:30:31 2013

5 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:59645         127.0.0.1:20000         ESTABLISHED
tcp      255      0 127.0.0.1:44525         127.0.0.1:20000         ESTABLISHED
tcp        0      0 127.0.0.1:20000         127.0.0.1:59645         ESTABLISHED
tcp        0      0 127.0.0.1:20000         127.0.0.1:44525         ESTABLISHED
tcp        0      0 10.66.17.97:56372       10.66.127.11:631        ESTABLISHED
```

绿色的四行表示第三第四个终端与 server 端的连接。

6 在第五个终端运行./client

connected to server: IP 127.0.0.1 PORT 20000

length=41
recv from server: hello, world! : Thu Oct 17 15:35:28 2013
哎呀呀，也能与 server 端正常通信啊！太奇怪了，backlog 不是 1 么？
7 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:59645         127.0.0.1:20000         ESTABLISHED
tcp        0      0 127.0.0.1:44525         127.0.0.1:20000         ESTABLISHED
tcp        0      0 127.0.0.1:20000         127.0.0.1:59645         ESTABLISHED
tcp        0      0 127.0.0.1:20000         127.0.0.1:34693         ESTABLISHED
tcp        0      0 127.0.0.1:20000         127.0.0.1:44525         ESTABLISHED
tcp      473      0 127.0.0.1:34693         127.0.0.1:20000         ESTABLISHED
tcp        0      0 10.66.17.97:56372       10.66.127.11:631        ESTABLISHED
```

其中绿色的两行表示第五个终端与 server 端建立的连接，很正常。
8 在新的终端继续运行./client，结果与第 6 步的类似

1.6 第三次观察程序行为

在 1 .1 节的前提条件中，我们看到/proc/sys/net/ipv4/tcp_abort_on_overflow 的值为 0。

下面我们看看如果设置为 1 会怎么样。我们先把所有 client 端终止，再把 server 端终止，然后把 server.c 中的 TEST_LISTEN_BACKLOG 设置为 1，重新编译。这样连接请求仍会被阻塞在 for 循环中，不会被 accept()系统调用处理。

1 在第一个终端运行./server

2 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
```

```
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 10.66.17.97:56372       10.66.127.11:631        ESTABLISHED
```

其中绿色的行表示 server 端正在 20000 端口监听。

3 在第三个终端运行./client

4 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:57404         127.0.0.1:20000         ESTABLISHED
tcp        0      0 10.66.17.97:56372       10.66.127.11:631        ESTABLISHED
tcp       13      0 127.0.0.1:20000         127.0.0.1:57404         ESTABLISHED
```

其中绿色的两行，表示新建立的连接。

5 在第四个终端运行./client

6 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

```
tcp        0      0 0.0.0.0:111             0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:631           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:44695           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:25            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23676           0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:20000           0.0.0.0:*               LISTEN
tcp       13      0 127.0.0.1:20000         127.0.0.1:58301         ESTABLISHED
tcp        0      0 127.0.0.1:58301         127.0.0.1:20000         ESTABLISHED
tcp        0      0 127.0.0.1:57404         127.0.0.1:20000         ESTABLISHED
tcp        0      0 10.66.17.97:56372       10.66.127.11:631        ESTABLISHED
```

tcp 13 0 127.0.0.1:20000 127.0.0.1:57404 ESTABLISHED

其中绿色的两行表示新建立的连接。

7 在第五个终端运行./client

屏幕打印“CONNECT FAILED !!!!!!: Connection reset by peer”，:-)，这个有点意思，有点符合我们的预期了。就是 server 端限制一定的连接数量，超过了这个限制，新的 client 端的连接我们就通知它“出错啦，快断开连接吧”！

8 我们看看 g 步骤的 tcpdump 输出

tcpdump: verbose output suppressed, use -v or -vv for full protocol decode

listening on lo, link-type EN10MB (Ethernet), capture size 65535 bytes

11:07:20.474855 IP 127.0.0.1.58302 > 127.0.0.1.20000: Flags [S], seq 3470574313, win 32792, options [mss 16396,sackOK,TS val 867467796 ecr 0,nop,wscale 6], length 0

11:07:20.474877 IP 127.0.0.1.20000 > 127.0.0.1.58302: Flags [S.], seq 3469333181, ack 3470574314, win 32768, options [mss 16396,sackOK,TS val 867467797 ecr 867467796,nop,wscale 6], length 0

11:07:20.474894 IP 127.0.0.1.58302 > 127.0.0.1.20000: Flags [.], ack 1, win 513, options [nop,nop,TS val 867467797 ecr 867467797], length 0

11:07:20.474905 IP 127.0.0.1.20000 > 127.0.0.1.58302: Flags [R], seq 3469333182, win 0, length 0

我们看到，在正常的三次握手的三个包之后，server 端向 client 端发送了一个 RST 包，从而将连接断掉了。

9 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：

tcp 0 0 0.0.0.0:111 0.0.0.0:* LISTEN

tcp 0 0 0.0.0.0:22 0.0.0.0:* LISTEN

tcp 0 0 127.0.0.1:631 0.0.0.0:* LISTEN

tcp 0 0 0.0.0.0:44695 0.0.0.0:* LISTEN

tcp 0 0 127.0.0.1:25 0.0.0.0:* LISTEN

tcp 0 0 0.0.0.0:23676 0.0.0.0:* LISTEN

tcp 0 0 0.0.0.0:20000 0.0.0.0:* LISTEN

tcp 13 0 127.0.0.1:20000 127.0.0.1:58301 ESTABLISHED

tcp 0 0 127.0.0.1:58301 127.0.0.1:20000 ESTABLISHED

tcp 0 0 127.0.0.1:57404 127.0.0.1:20000 ESTABLISHED

tcp 0 0 10.66.17.97:56372 10.66.127.11:631 ESTABLISHED

tcp 13 0 127.0.0.1:20000 127.0.0.1:57404 ESTABLISHED

这里与步骤 6 的输出一样，还是只有两个连接。

这样 server 端就可以限制 client 的连接数量了。

1.7 略作猜测与分析

1）server.c 中的 TEST_LISTEN_BACKLOG 宏

如果 TEST_LISTEN_BACKLOG 设置为 1，server 端会在 for 循环中一直运行，不会调用下面的 accept()系统调用了。这样就会把 client 的连接请求阻塞在这个 for 循环中，从而便于分析 backlog 是如何限制连接数量的。

2）tcp_abort_on_overflow 的含义

从字面意思看，是如果发生溢出，就终止连接。这就是我们看到“CONNECT FAILED !!!!!!: Connection reset by peer”的原因。

关键是这个溢出是哪个队列溢出了？肯定是与连接请求的数量有关。但细节也不甚了了。欲求其甚解，待我细细道来。

第 2 章 tcp 三次握手代码分析

所谓 tcp 三次握手，这个大家比较熟悉了，来张图。

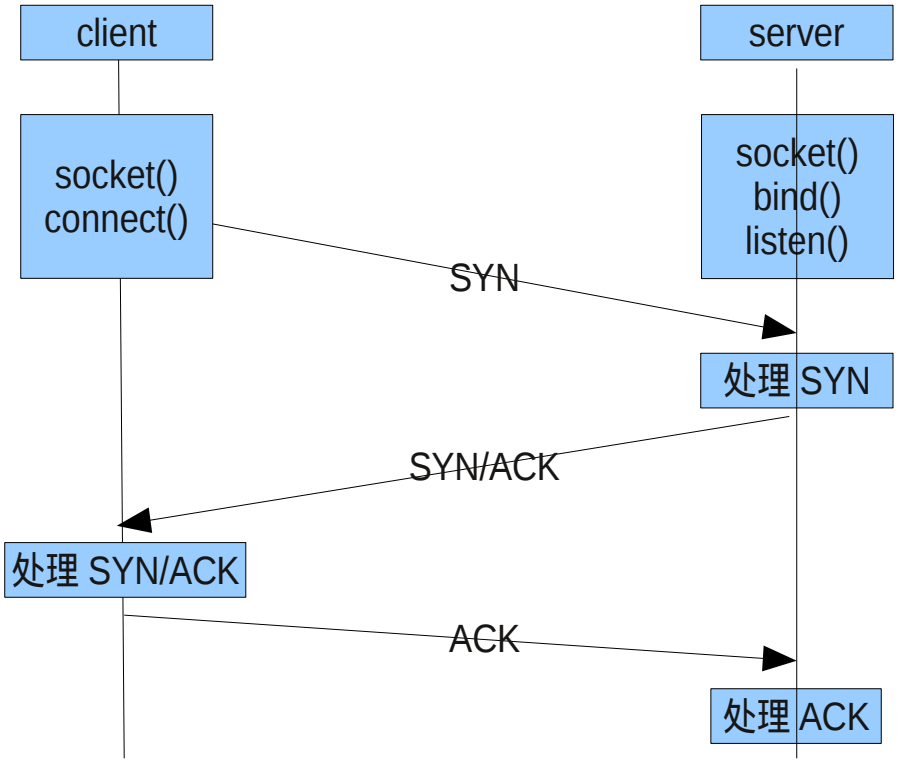

图 1 三次握手

socket 我们也比较熟悉，是用户程序与内核之间的一个接口。而内核中，还有一个通用的 sock 结构，用于隐藏具体的实现细节。sock 与 socket 是一一对应的。还有与具体协议相关的结构，如 tcp_sock，这是与具体协议的实现相关的。在三次握手的不同阶段，sock 会处于不同的状态，并根据不同的事件进行状态转移。

include/net/tcp_states.h

```
16 enum {
17      TCP_ESTABLISHED = 1,    /*表示已经连接*/
18      TCP_SYN_SENT,           /*表示已经发送一个 SYN 包，请求建立连接*/
19      TCP_SYN_RECV,           /*表示已经接收到一个 SYN 包*/
20      TCP_FIN_WAIT1,
21      TCP_FIN_WAIT2,
22      TCP_TIME_WAIT,
```

```
23      TCP_CLOSE,
24      TCP_CLOSE_WAIT,
25      TCP_LAST_ACK,
26      TCP_LISTEN,                 /*表示已经处于监听状态，可以接收连接请求*/
27      TCP_CLOSING,    /* Now a valid state */
28
29      TCP_MAX_STATES  /* Leave at the end! */
30 };
```

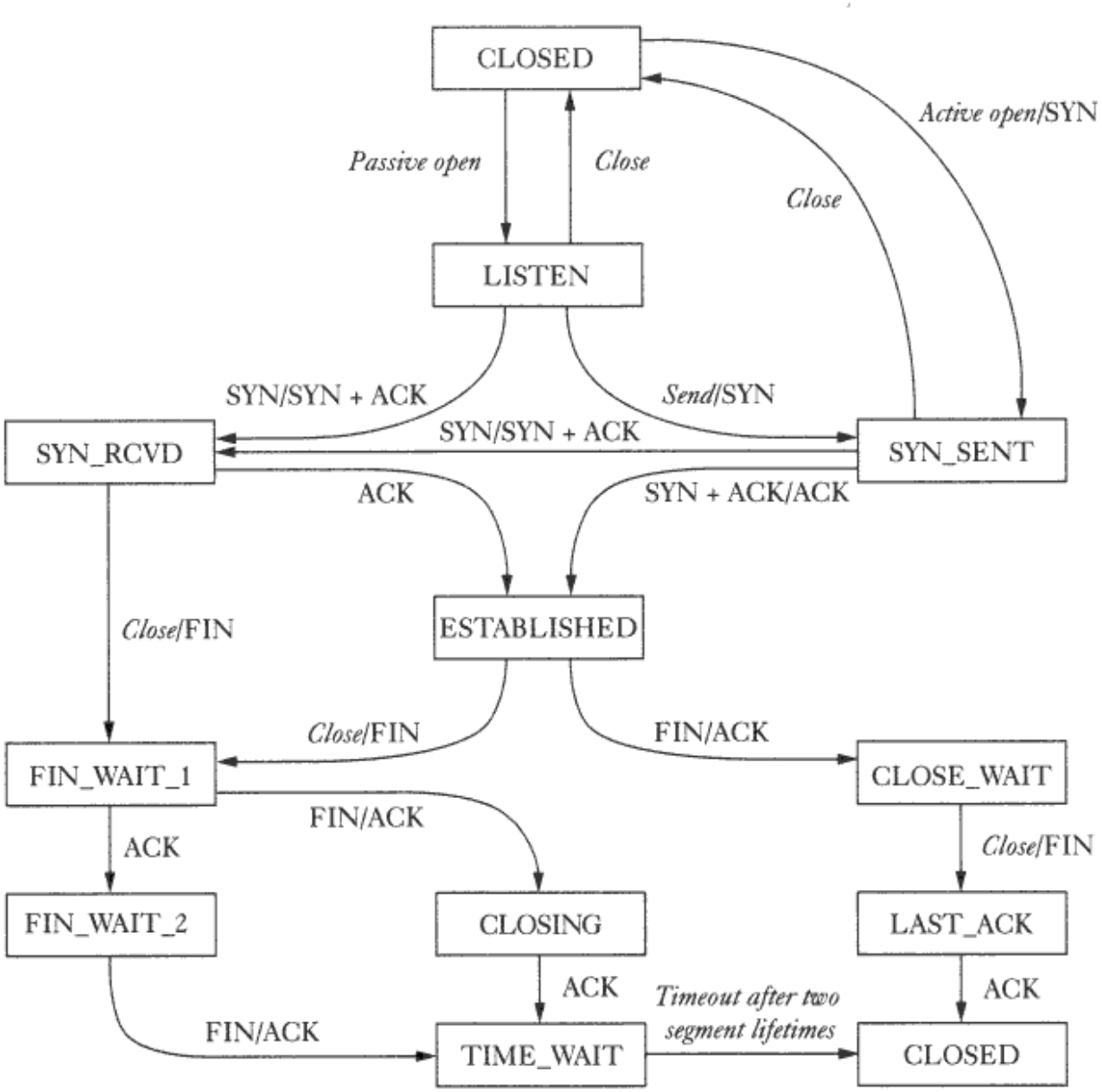

图 2 sock 状态转移图

下面将详细分析 listen()、处理 SYN、处理 SYN/ACK、处理 ACK 这几个逻辑，以及状态转移。

2.1 client/server 程序结构

这个大家也很熟悉，参见附录 1 和附录 2 的代码。

2.2 server 端 listen()系统调用

server 端调用 listen()系统调用的目的，是向系统宣告：我，即某个 socket，正在某个端口提供 tcp 服务，欢迎 client 端的连接请求。

看看 listen()系统调用的定义：int listen(int sockfd, int backlog);

我们再看看关于 backlog 的解释：

The backlog argument defines the maximum length to which the queue of pending connections for sockfd may grow. If a connection request arrives when the queue is full, the client may receive an error with an indication of ECONNREFUSED or, if the underlying protocol supports retransmission, the request may be ignored so that a later reattempt at connection succeeds.

不管你看懂了没有，反正我是没看懂。什么 queue？ECONNREFUSED 是怎么回事？还可以 reattempt？等看完本文，我们再来看看这个解释，看看理解会不会跟清晰些。

Listen 系统调用的定义：

```
net/socket.c
1534 SYSCALL_DEFINE2(listen, int, fd, int, backlog)
1535 {
                ......
1547            if (!err)
1548                    err = sock->ops->listen(sock, backlog);
                ......
1552        return err;
1553 }
```

这里的 ops 指针指向 inet_stream_ops，所以 listen 函数指针指向 inet_listen()。

```
net/ipv4/af_inet.c
195 int inet_listen(struct socket *sock, int backlog)
196 {
        ......
 214    if (old_state != TCP_LISTEN) {
                ......
 235            err = inet_csk_listen_start(sk, backlog);
 238    }
 239    sk->sk_max_ack_backlog = backlog;
        ......
}
```

inet_listen()函数的主要作用是判断当前 socket 如果不是处于 TCP_LISTEN 状态，就调用

inet_csk_listen_start()来创建 request queue，这个队列以后我们称之为请求连接队列。还有一个关键点是设置 sk_max_ack_backlog 的值。该值会作为 sk_ack_backlog 的上限值，并会被 sk_acceptq_is_full()函数使用。

include/net/sock.h

```
736 static inline bool sk_acceptq_is_full(const struct sock *sk)
737 {
738         return sk->sk_ack_backlog > sk->sk_max_ack_backlog;
739 }
```

也就是说，如果 sk_ack_backlog 的值比 sk_max_ack_backlog 大则说明 accept queue 满了。这个队列以后我们称之为等待连接队列。等待连接队列与请求连接队列肯定是不同的，那么区别和联系是什么？（关子 0，这个是本文的核心，呵呵）

还有，sk_ack_backlog 代表什么呢？在什么条件下改变它呢？别急，慢慢往下看。（关子 1）

net/ipv4/inet_connection_sock.c

```
751 int inet_csk_listen_start(struct sock *sk, const int nr_table_entries)
752 {
753         struct inet_sock *inet = inet_sk(sk);
754         struct inet_connection_sock *icsk = inet_csk(sk);
755         int rc = reqsk_queue_alloc(&icsk->icsk_accept_queue, nr_table_entries);
756
757         if (rc != 0)
758                 return rc;
759
760         sk->sk_max_ack_backlog = 0;
761         sk->sk_ack_backlog = 0;
762         inet_csk_delack_init(sk);
763
764         /* There is race window here: we announce ourselves listening,
765          * but this transition is still not validated by get_port().
766          * It is OK, because this socket enters to hash table only
767          * after validation is complete.
768          */
769         sk->sk_state = TCP_LISTEN;
770         if (!sk->sk_prot->get_port(sk, inet->inet_num)) {
771                 inet->inet_sport = htons(inet->inet_num);
772
773                 sk_dst_reset(sk);
774                 sk->sk_prot->hash(sk);
775
776                 return 0;
777         }
778
```

```
779         sk->sk_state = TCP_CLOSE;
780         __reqsk_queue_destroy(&icsk->icsk_accept_queue);
781         return -EADDRINUSE;
782 }
```

这个函数的主要作用是 Line755，根据 nr_table_entries（也就是 backlog 的值）调用 reqsk_queue_alloc()函数创建请求连接队列。

Line761 初始化 sk->sk_ack_backlog 的值为 0，这里有一点与关子 1 相关了。

Line769 设置 sk->sk_state 值为 TCP_LISTEN，表示当前 socket 处于 TCP_LISTEN 状态了。

Line770 其实也很关键，它为 socket 设置或者选择端口。本文不太涉及端口问题，故这里从略。

你猜猜这个函数如果正常返回，哪里是出口？对了，是 Line766，返回 0。否则是选择或者设置端口出错了，从 Line781 返回。

在分析 reqsk_queue_alloc()这个函数前，我们先看一下 listen_sock 结构体的定义：

```
include/net/request_sock.h
94 struct listen_sock {
95         u8                      max_qlen_log;
96         u8                      synflood_warned;
97         /* 2 bytes hole, try to use */
98         int                     qlen;
99         int                     qlen_young;
100        int                     clock_hand;
101        u32                     hash_rnd;
102        u32                     nr_table_entries;
103        struct request_sock     *syn_table[0];
104 };
```

listen_sock 结构用于管理一个 sock 接收到的所有的连接请求，其中的 syn_table[]是个变长数组，这个就是所谓的 request queue，即请求连接队列，顾名思义，是 server 端用于存放 client 发过来的连接请求的。qlen 表示请求连接队列的长度，qlen_young 表示请求连接队列中“年轻的”请求的数量。所谓“年轻的”，是指在请求连接队列中等待的时间不长，没有发生过一次超时。第一次超时时间为 1 秒钟。

请求连接队列管理的是元素是 request_sock 结构体，看看它的定义：

```
include/net/request_sock.h
48 /* struct request_sock - mini sock to represent a connection request
    * 表示一个连接请求的最小的 sock
49  */
50 struct request_sock {
51         struct request_sock             *dl_next;
52         u16                             mss;
53         u8                              num_retrans; /* number of retransmits */
54         u8                              cookie_ts:1; /* syncookie: encode tcpopts in timestamp */
55         u8                              num_timeout:7; /* number of timeouts */
```

```
56      /* The following two fields can be easily recomputed I think -AK */
57      u32                     window_clamp; /* window clamp at creation time */
58      u32                     rcv_wnd;        /* rcv_wnd offered first time */
59      u32                     ts_recent;
60      unsigned long               expires;
61      const struct request_sock_ops   *rsk_ops;
62      struct sock                 *sk;
63      u32                     secid;
64      u32                     peer_secid;
65 };
```

request_sock 代表一个连接请求，它与 sock 是有直接关系的。server 端在接收到一个连接请求（SYN 包）后，并不是直接创建一个 sock 结构，而是创建一个 request_sock 结构，因为 sock 结构非常大，会占用非常多的内存，而 request_sock 结构就非常轻量。等到 server 端接收到最后一个 ACK 包时，才根据 request_sock 结构创建相应的 sock 结构。这样做的好处主要是节省内存。

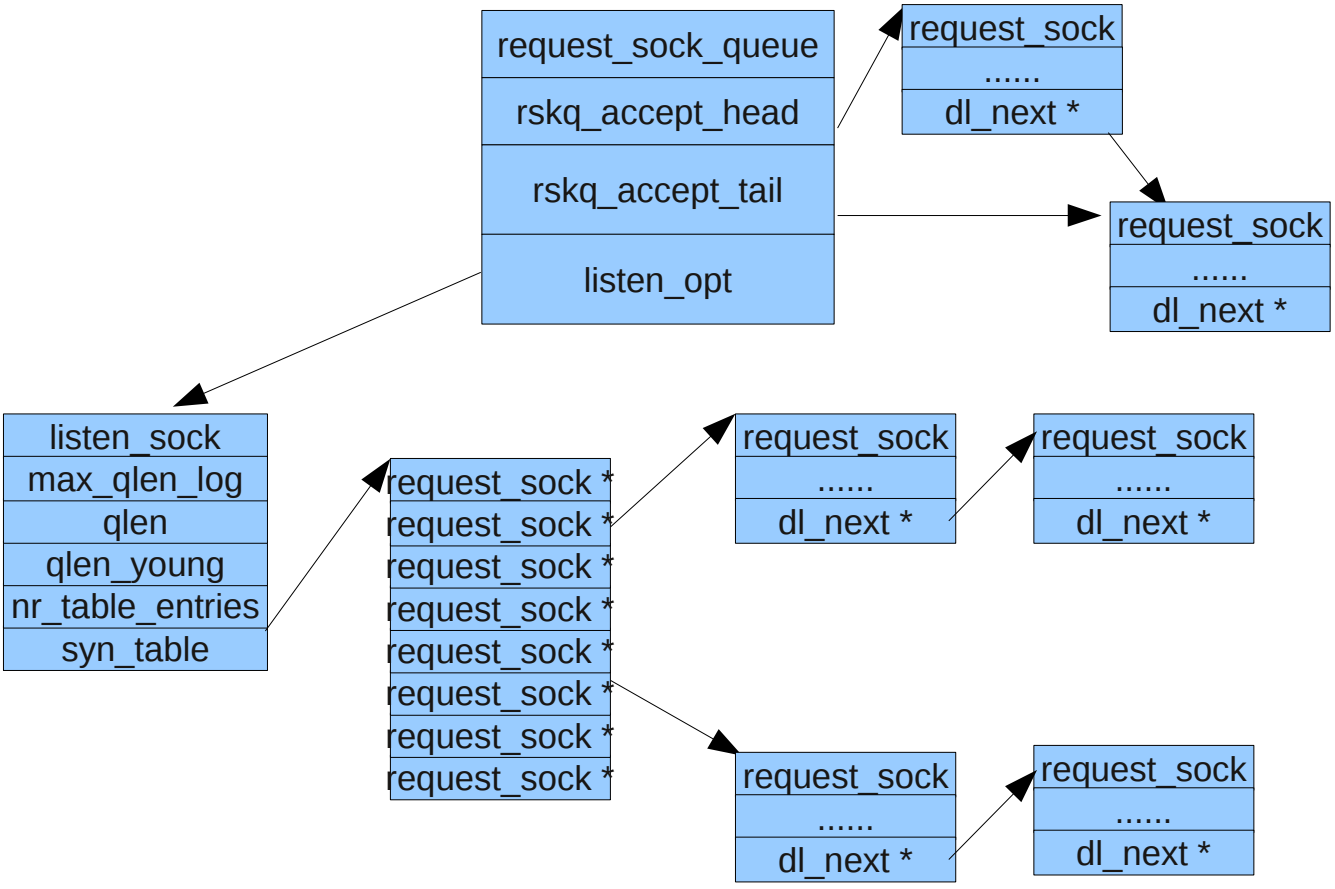

图 3 等待连接队列与请求连接队列

include/net/request_sock.h

```
150 struct request_sock_queue {
151         struct request_sock     *rskq_accept_head;
152         struct request_sock     *rskq_accept_tail;
            ......
156         struct listen_sock      *listen_opt;
            ......
163 };
```

request_sock_queue 是请求连接队列和等待连接队列的组合。（解释关子 0）

reqsk_queue_alloc()的作用，就是根据 nr_table_entries 的值，也就是 backlog 的值，创建一个大小适中的连接请求对列，即哈希表。

net/core/request_sock.c

```
37 int sysctl_max_syn_backlog = 256;

40 int reqsk_queue_alloc(struct request_sock_queue *queue,
41                         unsigned int nr_table_entries)
42 {
43         size_t lopt_size = sizeof(struct listen_sock);
44         struct listen_sock *lopt;
45
46         nr_table_entries = min_t(u32, nr_table_entries, sysctl_max_syn_backlog);
47         nr_table_entries = max_t(u32, nr_table_entries, 8);
48         nr_table_entries = roundup_pow_of_two(nr_table_entries + 1);
49         lopt_size += nr_table_entries * sizeof(struct request_sock *);
50         if (lopt_size > PAGE_SIZE)
51                 lopt = vzalloc(lopt_size);
52         else
53                 lopt = kzalloc(lopt_size, GFP_KERNEL);
54         if (lopt == NULL)
55                 return -ENOMEM;
56
57         for (lopt->max_qlen_log = 3;
58              (1 << lopt->max_qlen_log) < nr_table_entries;
59              lopt->max_qlen_log++);
60
61         get_random_bytes(&lopt->hash_rnd, sizeof(lopt->hash_rnd));
62         rwlock_init(&queue->syn_wait_lock);
63         queue->rskq_accept_head = NULL;
64         lopt->nr_table_entries = nr_table_entries;
65
66         write_lock_bh(&queue->syn_wait_lock);
67         queue->listen_opt = lopt;
68         write_unlock_bh(&queue->syn_wait_lock);
69
```

```
70        return 0;
71 }
```

Line46、Line47、Line48 三条语句决定了哈希表的项数，即哈希关键字的个数。Line46 说明，这个数不能大于 256；Line47 说明，这个数不能小于 8；Line48 说明这个数必须是 2 的幂次方。总结起来，就是下面的公式：

8<= N <=256 并且 N=2 ^ max_qlen_log

lopt->max_qlen_log 最小值为 3，最大为 8。

好了，现在假设我们 listen()系统调用传递进来的 backlog 参数是 1（参见附件 2 server.c 程序），那么，nr_table_entries 的值在函数开始前是 1，经过 Line46、 Line47、 Line48 三条语句，nr_table_entries 的值变为 8。

别着急，这个 8 是说 syn_table[]哈希表的关键字的个数为 8，并不是限制哈希表里面可以存放的 request_sock 的数量为 8，因为哈希冲突这里是用单向链表解决的，故理论上，可存放的 request_sock 的数量可以很大，只受限于系统内存！

但是，不要高兴太早，request_sock 的数量还是需要限制的。

include/net/request_sock.h

```
257 static inline int reqsk_queue_is_full(const struct request_sock_queue *queue)
258 {
259        return queue->listen_opt->qlen >> queue->listen_opt->max_qlen_log;
260 }
```

这个函数限制了 qlen 的大小不能超过 2 ^ max_qlen_log。从这里看，即使 backlog 的值为 1，经过 Line57、Line58、Line59 三行代码，max_qlen_log 的值是 3，所以 qlen 只要小于 7，reqsk_queue_is_full()就会返回 0，即表示请求连接队列还没有满。这个函数的用法，下面 connect()系统调用时会讲到。（关子 2）

小结一下 backlog 与请求连接队列的关系。backlog 会决定哈希表的项数，以及 max_qlen_log 的值，进而决定请求连接队列在什么条件下会被认为满了。但是 backlog 的值与哈希表的项数不是一定相等的，分三种情况：

1) backlog > 256, nr_table_entries = 256, max_qlen_log = 8；
2) backlog < 8, nr_table_entries = 8, max_qlen_log = 3；
3) 8 < backlog < 256, nr_table_entries = backlog, 2 ^ max_qlen_log = backlog。

总结一下，backlog 参数的作用，有三点：

1) 决定请求连接队列，即哈希表的关键字的个数；
2) 决定 max_qlen_log 的值，用于限定请求连接队列的长度；
3) 决定 sk_max_ack_backlog 的值，用于限定等待连接队列的长度。

等等等等，什么玩意儿？又是请求连接队列，又是等待连接队列，乱七八糟！

再次澄清一下（解释关子 0）：

所谓请求连接队列，即 request queue，是个哈希表，由 request_sock 结构体组成，用 syn_table[]表示。

所谓等待连接队列，即 accept queue，是个单向链表，也是由 request_sock 结构体组成，用 rskq_accept_head 指针指向链表头，rskq_accept_tail 指向链表尾。

两者的联系和区别，下文会详述。

好了，listen()系统调用讲完了，很简单吧？总结起来两点：

1) 根据 backlog 的值创建大小合适的请求连接队列（哈希表）；
2) 绑定端口，设置 TCP_LISTEN 状态。

2.3 client 端 connect()系统调用

client 端调用 connect()系统调用的作用，是向 server 端发送 SYN 包，请求建立连接。

```
net/socket.c
1666 SYSCALL_DEFINE3(connect, int, fd, struct sockaddr __user *, uservaddr,
1667                 int, addrlen)
1668 {
              ......
1685         err = sock->ops->connect(sock, (struct sockaddr *)&address, addrlen,
1686                                  sock->file->f_flags);
              ......
1690         return err;
1691 }
```

这里的 connect 函数指针指向 inet_stream_connect()：

```
net/ipv4/af_inet.c
684 int inet_stream_connect(struct socket *sock, struct sockaddr *uaddr,
685                        int addr_len, int flags)
686 {
687         int err;
688
689         lock_sock(sock->sk);
690         err = __inet_stream_connect(sock, uaddr, addr_len, flags);
691         release_sock(sock->sk);
692         return err;
693 }

net/ipv4/af_inet.c
597 int __inet_stream_connect(struct socket *sock, struct sockaddr *uaddr,
```

```
598                     int addr_len, int flags)
599 {
          ......
613       switch (sock->state) {
          ......
624       case SS_UNCONNECTED:
          ......
629             err = sk->sk_prot->connect(sk, uaddr, addr_len);
          ......
633             sock->state = SS_CONNECTING;
641       }
642
643       timeo = sock_sndtimeo(sk, flags & O_NONBLOCK);
644
645       if ((1 << sk->sk_state) & (TCPF_SYN_SENT | TCPF_SYN_RECV)) {
646             int writebias = (sk->sk_protocol == IPPROTO_TCP) &&
647                             tcp_sk(sk)->fastopen_req &&
648                             tcp_sk(sk)->fastopen_req->data ? 1 : 0;
649
650             /* Error code is set above */
651             if (!timeo || !inet_wait_for_connect(sk, timeo, writebias))
652                   goto out;
                  ......
657       }

670       sock->state = SS_CONNECTED;
671       err = 0;
672 out:
673       return err;
          ......
681 }
```

Line629 这里的 connect()函数指针指向 tcp_v4_connect()。

__inet_stream_connect()主要逻辑：

1) 如果 sock->state 是 SS_UNCONNECTED，则调用 tcp_v4_connect()构造并发送 SYN 包（Line629）；
2) 设置 sock->state 为 SS_CONNECTING，表示正在与 server 端进行连接（Line633）；
3) 如果 socket 是阻塞的（不是 O_NONBLOCK），则进行等待，直到连接建立（Line651）；
4) 如果 inet_wait_for_connect()成功返回，则设置 sock->state 为 SS_CONNECTED，表示连接已经建立好了。

net/ipv4/tcp_ipv4.c
```
143 int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr, int addr_len)
144 {
        ......
222     tcp_set_state(sk, TCP_SYN_SENT);
238     if (!tp->write_seq && likely(!tp->repair))
239         tp->write_seq = secure_tcp_sequence_number(inet->inet_saddr,
240                                                    inet->inet_daddr,
241                                                    inet->inet_sport,
242                                                    usin->sin_port);
        ......
246     err = tcp_connect(sk);
252     return 0;
        ......
264 }
```

Line222 将 sock 的状态设置为 TCP_SYN_SENT，表示 SYN 包已经发出去了。（其实 SYN 包还未发出去呢，tcp_connect()才会将它发送出去。）

Line238 至 242 的作用是为这个 tcp 连接选定一个开始的序号，即 tcp 包的 seq，并保存至 tp->write_seq 中。

net/ipv4/tcp_output.c
```
2939 int tcp_connect(struct sock *sk)
2940 {
        ......
2959    tcp_init_nondata_skb(buff, tp->write_seq++, TCPHDR_SYN);
2964    /* Send off SYN; include data in Fast Open. */
2965    err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
2966        tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);
2978    inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
2979                              inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
2980    return 0;
2981 }
```

tcp_connect()会构造一个 SYN 包，并调用 tcp_transmit_skb()发送出去。

net/ipv4/tcp_output.c
```
364 static void tcp_init_nondata_skb(struct sk_buff *skb, u32 seq, u8 flags)
365 {
366     skb->ip_summed = CHECKSUM_PARTIAL;
367     skb->csum = 0;
368
369     TCP_SKB_CB(skb)->tcp_flags = flags;
370     TCP_SKB_CB(skb)->sacked = 0;
```

```
371
372         skb_shinfo(skb)->gso_segs = 1;
373         skb_shinfo(skb)->gso_size = 0;
374         skb_shinfo(skb)->gso_type = 0;
375
376         TCP_SKB_CB(skb)->seq = seq;
377         if (flags & (TCPHDR_SYN | TCPHDR_FIN))
378                 seq++;
379         TCP_SKB_CB(skb)->end_seq = seq;
380 }
```

tcp_init_nondata_skb()会根据 tp->write_seq 的值构建一个 SYN 包，SYN 的标志是通过 TCPHDR_SYN 传递进来的。Line376 和 Line379 会设置 SYN 包的开始和结束序号。当然，这些序号，都是暂时设置在 TCP_SKB_CB 控制块中的，还没有设置到真正的 tcp 包头上。tcp_transmit_skb()会构建一个真正要发送到网络上的 tcp 包。

```
net/ipv4/tcp_output.c
836 static int tcp_transmit_skb(struct sock *sk, struct sk_buff *skb, int clone_it,
837                              gfp_t gfp_mask)
838 {
            ......
902         th = tcp_hdr(skb);
903         th->source              = inet->inet_sport;
904         th->dest                = inet->inet_dport;
905         th->seq                 = htonl(tcb->seq);
906         th->ack_seq             = htonl(tp->rcv_nxt);
907         *(((__be16 *)th) + 6)   = htons(((tcp_header_size >> 2) << 12) |
908                                         tcb->tcp_flags);
            ......
957         err = icsk->icsk_af_ops->queue_xmit(skb, &inet->cork.fl);
958         if (likely(err <= 0))
959                 return err;
            ......
964 }
```

tcp_transmit_skb()会构建一个真正要发送到网络上的 tcp 包，设置 tcp 头的相关域，包括源端口号、目的端口号、开始序号、确认序号、标志位等等，然后调用 ip_queue_xmit()将 SYN 包发送到网络上。ip_queue_xmit()会构造 ip 包。tcp 会选择合适的时机，将该 SYN 包发送到网络上。

总之，client 端 connect()系统调用的逻辑还是比较简单的，主要是构造一个 SYN 包，然后发送给 server 端。

2.4 server 端处理 SYN 包

2.3 节构造的 SYN 包，经过中间一系列的交换机和路由器，最终会到达 server 端。这个路由的过程，本文从略。

我们从 server 端接到一个 tcp 包开始（处理 ip 包的过程省略），往下看：

```
net/ipv4/tcp_ipv4.c
1935 int tcp_v4_rcv(struct sk_buff *skb)
1936 {
            ......
1999        bh_lock_sock_nested(sk);
2000        ret = 0;
2001        if (!sock_owned_by_user(sk)) {
2002 #ifdef CONFIG_NET_DMA
2003                struct tcp_sock *tp = tcp_sk(sk);
2004                if (!tp->ucopy.dma_chan && tp->ucopy.pinned_list)
2005                    tp->ucopy.dma_chan = net_dma_find_channel();
2006                if (tp->ucopy.dma_chan)
2007                    ret = tcp_v4_do_rcv(sk, skb);
2008                else
2009 #endif
2010                {
2011                    if (!tcp_prequeue(sk, skb))
2012                        ret = tcp_v4_do_rcv(sk, skb);
2013                }
2014        } else if (unlikely(sk_add_backlog(sk, skb,
2015                                           sk->sk_rcvbuf + sk->sk_sndbuf))) {
2016                bh_unlock_sock(sk);
2017                NET_INC_STATS_BH(net, LINUX_MIB_TCPBACKLOGDROP);
2018                goto discard_and_relse;
2019        }
2020        bh_unlock_sock(sk);
2021
2022        sock_put(sk);
2023
2024        return ret;
             ......
2085 }
```

Line1999 到 Line2020 这一段比较复杂，涉及到 sock 接收数据的三个队列，即 backlog 队列（与 listen()的 backlog 参数没有任何关系）、prequeue 队列以及 receive_queue 队列，还有一个“实时拷贝”的 ucopy 机制，还涉及到软中断上下文和进程上下文，逻辑关系相当复杂。但是对于本文提到的 SYN、SYN/ACK 和 ACK 三个包，他们不管之前在什么上下文中被排入哪个

队列，最终都会被 tcp_v4_do_rcv()函数处理。

net/ipv4/tcp_ipv4.c
```
1777 int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
1778 {
          ......
1791      if (sk->sk_state == TCP_ESTABLISHED) { /* Fast path */
               ......
1802           tcp_rcv_established(sk, skb, tcp_hdr(skb), skb->len);
1803           return 0;
1804      }
          ......
1809      if (sk->sk_state == TCP_LISTEN) {
1810           struct sock *nsk = tcp_v4_hnd_req(sk, skb);
1811           if (!nsk)
1812                goto discard;
1813
1814           if (nsk != sk) {
1815                sock_rps_save_rxhash(nsk, skb);
1816                if (tcp_child_process(sk, nsk, skb)) {
1817                     rsk = nsk;
1818                     goto reset;
1819                }
1820                return 0;
1821           }
1822      } else
1823           sock_rps_save_rxhash(sk, skb);
1824
1825      if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
1826           rsk = sk;
1827           goto reset;
1828      }
1829      return 0;
          ......
1846 }
```

tcp_v4_do_rcv()函数会判断当前 sk 的状态，如果是已经连接，即 TCP_ESTABLISHED，就调用 tcp_rcv_established()进行处理（Line1802）。否则如果当前 sk 是 TCP_LISTEN 状态，则在 Line1809 至 Line1828 中进行处理。

现在回想一下 2.2 节，当 server 端调用 listen()系统调用后，就是处于 TCP_LISTEN 状态的。Line1810 调用 tcp_v4_hnd_req()，

net/ipv4/tcp_ipv4.c
```
1716 static struct sock *tcp_v4_hnd_req(struct sock *sk, struct sk_buff *skb)
1717 {
1718      struct tcphdr *th = tcp_hdr(skb);
```

```
1719        const struct iphdr *iph = ip_hdr(skb);
1720        struct sock *nsk;
1721        struct request_sock **prev;
1722        /* Find possible connection requests. */
1723        struct request_sock *req = inet_csk_search_req(sk, &prev, th->source,
1724                                                       iph->saddr, iph->daddr);
1725        if (req)
1726              return tcp_check_req(sk, skb, req, prev, false);
            ......
1744        return sk;
1745 }
```

tcp_v4_hnd_req()是个很比较特殊的函数，倒不是很复杂，问题是它会在三次握手中的两次都会被调用。而因为每次的情况是不一样的，所以处理的逻辑也是不一样的。

Line1723 会调用 inet_csk_search_req()函数，根据源端口号、源 IP 地址和目的 IP 地址在请求连接队列（request queue）中进行查找，即查找 syn_table 哈希表。

net/ipv4/inet_connection_sock.c

```
492 struct request_sock *inet_csk_search_req(const struct sock *sk,
493                                         struct request_sock ***prevp,
494                                         const __be16 rport, const __be32 raddr,
495                                         const __be32 laddr)
496 {
497        const struct inet_connection_sock *icsk = inet_csk(sk);
498        struct listen_sock *lopt = icsk->icsk_accept_queue.listen_opt;
499        struct request_sock *req, **prev;
500
501        for (prev = &lopt->syn_table[inet_synq_hash(raddr, rport, lopt->hash_rnd,
502                                                    lopt->nr_table_entries)];
503             (req = *prev) != NULL;
504             prev = &req->dl_next) {
505             const struct inet_request_sock *ireq = inet_rsk(req);
506
507             if (ireq->rmt_port == rport &&
508                 ireq->rmt_addr == raddr &&
509                 ireq->loc_addr == laddr &&
510                 AF_INET_FAMILY(req->rsk_ops->family)) {
511                   WARN_ON(req->sk);
512                   *prevp = prev;
513                   break;
514             }
515        }
516
517        return req;
518 }
```

先看这第一次，此时，server 端刚刚收到 SYN 包，还没有为之创建 request_sock 结构，故

inet_csk_search_req()会返回 NULL。

回到 tcp_v4_hnd_req()Line1723，因为 inet_csk_search_req()返回 NULL，所以 Line1725 行的条件为 FALSE，于是 tcp_v4_hnd_req()会从 Line1744 行退出，返回值是调用 tcp_v4_hnd_req()时的参数之一 sk。

于是回到 tcp_v4_do_rcv()Line1810。 tcp_v4_hnd_req()返回调用时的 sk，Line1811 和 Line1814 的条件都为 FALSE，于是控制转到 Line1825，调用 tcp_rcv_state_process()函数继续处理这个 SYN 包。

有人可能会奇怪了，这第一次调用 tcp_v4_hnd_req()没啥作用了，在请求连接队列中逛了一圈，又出来了，没有任何实际性的操作。

确实如此，但是第二次调用 tcp_v4_hnd_req()可就不一样了。作者为了逻辑一致和代码简化，所以才这样设计的。（关子 3，看看第二次吧）

下面开始动真格的了。

net/ipv4/tcp_input.c

```
5579 int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,
5580                          const struct tcphdr *th, unsigned int len)
5581 {
              ......
5590          switch (sk->sk_state) {
              ......
5594          case TCP_LISTEN:
5595                if (th->ack)
5596                      return 1;
5597
5598                if (th->rst)
5599                      goto discard;
5600
5601                if (th->syn) {
5602                      if (th->fin)
5603                            goto discard;
5604                      if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
5605                            return 1;
5606
                       ......
5624                      kfree_skb(skb);
5625                      return 0;
5626                }
              ......
5840          return 0;
5841 }
```

Line5594，因为此时 server 端的 sock 仍处于 TCP_LISTEN 状态，故控制会转到 Line5594 至 Line5625 进行处理。

Line5595，判断 tcp 包中是否设置了 ACK 标志。如果是，返回 1，表示出错推出。因为我们知道，第一个 SYN 包中是不能包含 ACK 标志的。
Line5598，判断 tcp 包中是否设置了 RST 标志。如果是，则丢弃当前的 tcp 包。
Line5601 终于走到正题，判断 tcp 包中是否设置了 SYN 标志，如果是，继续进行处理。
Line5602，判断 tcp 包中是否设置了 FIN 标志。SYN 与 FIN 是相反的两个标志，一个是建立连接，一个是断开连接，所以不能同时设置。
Line5604，这里的 conn_request 函数指针指向 tcp_v4_conn_request()函数，这才是 server 端处理 SYN 包的重头戏。

net/ipv4/tcp_ipv4.c

```
1443 int tcp_v4_conn_request(struct sock *sk, struct sk_buff *skb)
1444 {
            ......
1468        if ((sysctl_tcp_syncookies == 2 ||
1469             inet_csk_reqsk_queue_is_full(sk)) && !isn) {
1470                want_cookie = tcp_syn_flood_action(sk, skb, "TCP");
1471                if (!want_cookie)
1472                        goto drop;
1473        }
            ......
1480        if (sk_acceptq_is_full(sk) && inet_csk_reqsk_queue_young(sk) > 1) {
1481                NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
1482                goto drop;
1483        }
1485        req = inet_reqsk_alloc(&tcp_request_sock_ops);
            ......
1502        tcp_openreq_init(req, &tmp_opt, skb);
            ......
1577        skb_synack = tcp_make_synack(sk, dst, req,
1578            fastopen_cookie_present(&valid_foc) ? &valid_foc : NULL);
                ......
1588            err = ip_build_and_send_pkt(skb_synack, sk, ireq->loc_addr,
1589                ireq->rmt_addr, ireq->opt);
                ......
1596            /* Add the request_sock to the SYN table */
1597            inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
            ......
1604        return 0;
            ......
1613 }
```

先看两个内联函数，这两个函数对于本文意义重大。

include/net/inet_connection_sock.h
```
297 static inline int inet_csk_reqsk_queue_is_full(const struct sock *sk)
298 {
299         return reqsk_queue_is_full(&inet_csk(sk)->icsk_accept_queue);
300 }
```

```
257 static inline int reqsk_queue_is_full(const struct request_sock_queue *queue)
258 {
259         return queue->listen_opt->qlen >> queue->listen_opt->max_qlen_log;
260 }
```

inet_csk_reqsk_queue_is_full()函数判断请求连接队列（request queue）是否满了（解释关子 2）。从 reqsk_queue_is_full()函数看，参与比较的是 qlen 和 max_qlen_log。max_qlen_log 我们在 2.2 节介绍过，是请求连接队列的上限值。如果 qlen 大于这个值，则表示请求连接队列满了。关于 qlen 的修改操作，下文详述。

include/net/sock.h
```
736 static inline bool sk_acceptq_is_full(const struct sock *sk)
737 {
738         return sk->sk_ack_backlog > sk->sk_max_ack_backlog;
739 }
```

sk_acceptq_is_full()函数判断等待连接队列是否满了。2.2 节我们看到，sk_max_ack_backlog 的值就是 listen()系统调用的参数 backlog。而 sk_ack_backlog 的修改操作，下文详述。

当第一个 SYN 包来到 server 端时，因为 request_sock_queue->listen_opt->qlen 为 0，所以 Line1468 至 Line1473 中的代码不会执行。

同样，因为 server 端刚收到一个 SYN 包，所以等待连接队列是空的，于是 Line1480 至 Line1483 中的代码也不会执行到这里。

但是，这里应该注意，在 server 端处理 SYN 包之前，要分别检测请求连接队列和等待连接队列的长度。如果超出上限值，则会丢弃这个 SYN 包，从而忽略这个连接请求。反过来说，只有请求连接队列和等待连接队列中的数据没有达到上限值时，才会允许处理新的连接请求。

下面分析一下 Line1480。

include/net/inet_connection_sock.h
```
292 static inline int inet_csk_reqsk_queue_young(const struct sock *sk)
293 {
294         return reqsk_queue_len_young(&inet_csk(sk)->icsk_accept_queue);
295 }
```

```
252 static inline int reqsk_queue_len_young(const struct request_sock_queue *queue)
253 {
254         return queue->listen_opt->qlen_young;
255 }
```

关于 qlen_young，先看一段注释：

net/ipv4/inet_connection_sock.c
```
586
587         /* Normally all the openreqs are young and become mature
588          * (i.e. converted to established socket) for first timeout.
589          * If synack was not acknowledged for 1 second, it means
590          * one of the following things: synack was lost, ack was lost,
591          * rtt is high or nobody planned to ack (i.e. synflood).
592          * When server is a bit loaded, queue is populated with old
593          * open requests, reducing effective size of queue.
594          * When server is well loaded, queue size reduces to zero
595          * after several minutes of work. It is not synflood,
596          * it is normal operation. The solution is pruning
597          * too old entries overriding normal timeout, when
598          * situation becomes dangerous.
599          *
600          * Essentially, we reserve half of room for young
601          * embrions; and abort old ones without pity, if old
602          * ones are about to clog our table.
603          */
```

qlen_young 是用于实现 request_sock 的老化机制的。这里简单解释一下。所谓老化机制，就是如果 tcp 发现一个 request_sock 在请求连接队列里待的时间太长了，就把他的年龄变老。如果它变得太老了，超过的上限值，则直接把它从请求连接队列里删除。这样做的目的，是避免有的 request_sock 长期占用请求连接队列，这样如果数量多了，会占用请求连接队列太多空间，导致后续的正常连接请求也不能够正常处理了。详见 net/ipv4/inet_connection_sock.c 中的 inet_csk_reqsk_queue_prune()函数，这里不展开了。

现在回到 tcp_v4_conn_request() Line1480，此时因为是服务器端刚刚接收到第一个 SYN 包，sk_acceptq_is_full(sk)一定为 0，(inet_csk_reqsk_queue_young(sk) 的值也为 0)，于是转到 Line1485 行继续处理。

inet_reqsk_alloc()函数是本节的重头戏。

include/net/inet_sock.h
```
230 static inline struct request_sock *inet_reqsk_alloc(struct request_sock_ops *ops)
231 {
232         struct request_sock *req = reqsk_alloc(ops);
233         struct inet_request_sock *ireq = inet_rsk(req);
234
235         if (req != NULL) {
236                 kmemcheck_annotate_bitfield(ireq, flags);
237                 ireq->opt = NULL;
238         }
239
```

```
240         return req;
241 }
```

inet_reqsk_alloc()函数的作用，是调用 reqsk_alloc()函数分配一个新的 request_sock 结构，用于代表一个连接请求。

include/net/request_sock.h

```
 67 static inline struct request_sock *reqsk_alloc(const struct request_sock_ops *ops)
 68 {
 69         struct request_sock *req = kmem_cache_alloc(ops->slab, GFP_ATOMIC);
 70
 71         if (req != NULL)
 72                 req->rsk_ops = ops;
 73
 74         return req;
 75 }
```

reqsk_alloc()也比较简单，就是从 slab 中分配一个 request_sock 结构体。

Line1485 行运行结束后，我们就得到一个代表本次连接请求的 request_sock 结构体了。

Line1502 会调用 tcp_openreq_init()函数对其进行初始化，这个比较简单，从略。

当 server 端接收到 SYN 包后，会向 client 端发送 SYN/ACK 包，表示“我已经收到你发给我的连接请求了，现在我也向你发送我的连接请求，请予以妥善处理为盼”。这是通过 Line1577 和 Line1588 实现的。tcp_make_synack()函数会生成一个 SYN/ACK 包，ip_build_and_send_pkt()会将其发送出去。

然后进到 Line1597，inet_csk_reqsk_queue_hash_add()，这个函数对于请求连接队列至关重要。它会把刚刚生成的 request_sock 结构体加入到请求连接队列中。

net/ipv4/inet_connection_sock.c

```
521 void inet_csk_reqsk_queue_hash_add(struct sock *sk, struct request_sock *req,
522                                     unsigned long timeout)
523 {
524         struct inet_connection_sock *icsk = inet_csk(sk);
525         struct listen_sock *lopt = icsk->icsk_accept_queue.listen_opt;
526         const u32 h = inet_synq_hash(inet_rsk(req)->rmt_addr, inet_rsk(req)->rmt_port,
527                                      lopt->hash_rnd, lopt->nr_table_entries);
528
529         reqsk_queue_hash_req(&icsk->icsk_accept_queue, h, req, timeout);
530         inet_csk_reqsk_queue_added(sk, timeout);
531 }
```

include/net/request_sock.h

```
262 static inline void reqsk_queue_hash_req(struct request_sock_queue *queue,
263                                         u32 hash, struct request_sock *req,
264                                         unsigned long timeout)
```

```
265 {
266         struct listen_sock *lopt = queue->listen_opt;
267
268         req->expires = jiffies + timeout;
269         req->num_retrans = 0;
270         req->num_timeout = 0;
271         req->sk = NULL;
272         req->dl_next = lopt->syn_table[hash];
273
274         write_lock(&queue->syn_wait_lock);
275         lopt->syn_table[hash] = req;
276         write_unlock(&queue->syn_wait_lock);
277 }
```

reqsk_queue_hash_req()的作用是将新创建的 req 加到请求连接队列中，即 lopt->syn_table[]，可以看到这里采用的是“头插法”。

include/net/inet_connection_sock.h

```
280 static inline void inet_csk_reqsk_queue_added(struct sock *sk,
281                                               const unsigned long timeout)
282 {
283         if (reqsk_queue_added(&inet_csk(sk)->icsk_accept_queue) == 0)
284                 inet_csk_reset_keepalive_timer(sk, timeout);
285 }
```

inet_csk_reset_keepalive_timer()与实现 request_sock 的老化机制的有关，从略。

include/net/request_sock.h

```
237 static inline int reqsk_queue_added(struct request_sock_queue *queue)
238 {
239         struct listen_sock *lopt = queue->listen_opt;
240         const int prev_qlen = lopt->qlen;
241
242         lopt->qlen_young++;
243         lopt->qlen++;
244         return prev_qlen;
245 }
```

reqsk_queue_added()函数的作用是增加计数，Line243 把请求连接队列的长度加 1，表示请求队列中又多了一个请求。而 Line242 行将 qlen_young 的长度加 1，表示刚加入的这个请求是“年轻的”。这个值之后还会讨论到。

好了，讨论到这里，server 端处理 SYN 包的流程基本完毕，总结一下挺简单的，就是 tcp_rcv_state_process()调用 tcp_v4_conn_request()，创建一个代表连接请求的 request_sock 结构体，并加入到请求连接队列中，然后向 client 端发送 SYN/ACK 包。

另外，反过来想一想，如果 inet_csk_reqsk_queue_is_full()为 TRUE，意味着什么？意味着在一个很小的时间段内 server 端收到了太多的连接请求，请求连接队列已经满了。这时，

server 端会区分，这是正常的情况呢？还是不正常的情况。这里所谓正常，是指在这一很小的时间段，确实有多个合法的 client 端发过来的连接请求超出限制了。所谓不正常，是指其实并没有太多的 client 端发过来请求，而是由某个 client 端伪造了太多的连接请求，从而导致 server 端的连接请求队列塞满了，不能处理了。这种不正常的情况，称之为 SYN flood，即 SYN 泛洪，是一种常见的网络攻击手段。魔高一尺，道高一丈，内核实现了 SYN cookie 机制来检测 SYN flood。

2.5 client 端处理 SYN/ACK 包

有了 2.4 节的基础，再分析 client 端处理 SYN/ACK 包的流程就相对简单些了。前边的逻辑都是一样的。

tcp_v4_rcv()->tcp_v4_do_rcv()

因为 client 在发出 SYN 包后，处于 TCP_SYN_SENT 状态，所以在 tcp_v4_do_rcv()函数中直接转到 Line1825，调用 tcp_rcv_state_process()处理 SYN/ACK 包。而在 tcp_rcv_state_process()函数中，转到 Line5630 行调用 tcp_rcv_synsent_state_process()函数处理 SYN/ACK 包。

```
net/ipv4/tcp_input.c
5352 static int tcp_rcv_synsent_state_process(struct sock *sk, struct sk_buff *skb,
5353                                           const struct tcphdr *th, unsigned int len)
5354 {
            ......
5364        if (th->ack) {
                ......
5404            if (!th->syn)
5405                goto discard_and_undo;
                ......
5459            tcp_finish_connect(sk, skb);
                ......
5475            inet_csk_schedule_ack(sk);
                ......
5488        }
            ......
5570 }
```

因为 server 端发回来的是 SYN/ACK 包，所以 Line5364 的条件是满足的。然后在 Line5404 行判断是否设置了 SYN 标志，如果没有设置，出错返回。如果设置了，调用 tcp_finish_connect()函数进行 client 端建立连接的收尾操作。然后通过 Line5475 行调度一个 ACK 发给 server 端，用于通知 server 端，连接建立在 client 端已经完成了。

这里只是调度了一个 ACK，那么什么时间发送呢？这要回到 tcp_rcv_state_process()函数的 Line5637，通过 tcp_data_snd_check()函数发送 ACK。

下面看看 tcp_finish_connect ()函数。

net/ipv4/tcp_input.c

```
5270 void tcp_finish_connect(struct sock *sk, struct sk_buff *skb)
5271 {
5272        struct tcp_sock *tp = tcp_sk(sk);
5273        struct inet_connection_sock *icsk = inet_csk(sk);
5274
5275        tcp_set_state(sk, TCP_ESTABLISHED);
5276
5277        if (skb != NULL) {
5278              icsk->icsk_af_ops->sk_rx_dst_set(sk, skb);
5279              security_inet_conn_established(sk, skb);
5280        }
5281
5282        /* Make sure socket is routed, for correct metrics.  */
5283        icsk->icsk_af_ops->rebuild_header(sk);
5284
5285        tcp_init_metrics(sk);
5286
5287        tcp_init_congestion_control(sk);
5288
5289        /* Prevent spurious tcp_cwnd_restart() on first data
5290         * packet.
5291         */
5292        tp->lsndtime = tcp_time_stamp;
5293
5294        tcp_init_buffer_space(sk);
5295
5296        if (sock_flag(sk, SOCK_KEEPOPEN))
5297              inet_csk_reset_keepalive_timer(sk, keepalive_time_when(tp));
5298
5299        if (!tp->rx_opt.snd_wscale)
5300              __tcp_fast_path_on(tp, tp->snd_wnd);
5301        else
5302              tp->pred_flags = 0;
5303
5304        if (!sock_flag(sk, SOCK_DEAD)) {
5305              sk->sk_state_change(sk);
5306              sk_wake_async(sk, SOCK_WAKE_IO, POLL_OUT);
5307        }
5308 }
```

其中 Line5275 行，设置 client 端的 sock 为 TCP_ESTABLISHED 状态，表示连接已经建立。对于 client 端来说，大功告成！

2.6 server 端处理 ACK 包

2.5 节说到 client 端最后会发送一个 ACK 给 server 端，用于表示 client 端的建立连接工作已经完成，请 server 继续处理建立连接的工作，这样双方就可以进行数据的传输了。

server 端处理这个 ACK 包，就比较复杂了。

前边的逻辑还是一样的，tcp_v4_rcv()->tcp_v4_do_rcv()。因为 server 端在处理 client 端的 SYN 包时，状态并没有改变，一直是 TCP_LISTEN（实际上除非要断开连接，否则 server 端一直处于 TCP_LISTEN 状态等待 client 的连接）。所以此时在 tcp_v4_do_rcv()函数中又会执行到 Line1809。为了方便查看代码，这里把 tcp_v4_do_rcv()的相关代码再贴一遍。

net/ipv4/tcp_ipv4.c

```
1777 int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
1778 {
            ......
1809        if (sk->sk_state == TCP_LISTEN) {
1810            struct sock *nsk = tcp_v4_hnd_req(sk, skb);
1811            if (!nsk)
1812                goto discard;
1813
1814            if (nsk != sk) {
1815                sock_rps_save_rxhash(nsk, skb);
1816                if (tcp_child_process(sk, nsk, skb)) {
1817                    rsk = nsk;
1818                    goto reset;
1819                }
1820                return 0;
1821            }
1822        } else
1823            sock_rps_save_rxhash(sk, skb);
1824
1825        if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb), skb->len)) {
1826            rsk = sk;
1827            goto reset;
1828        }1829        return 0;
            ......
1846 }
```

然后进入 Line1810，调用 tcp_v4_hnd_req()函数。

net/ipv4/tcp_ipv4.c

```
1716 static struct sock *tcp_v4_hnd_req(struct sock *sk, struct sk_buff *skb)
1717 {
1718        struct tcphdr *th = tcp_hdr(skb);
1719        const struct iphdr *iph = ip_hdr(skb);
1720        struct sock *nsk;
```

```
1721        struct request_sock **prev;
1722        /* Find possible connection requests. */
1723        struct request_sock *req = inet_csk_search_req(sk, &prev, th->source,
1724                                                       iph->saddr, iph->daddr);
1725        if (req)
1726                return tcp_check_req(sk, skb, req, prev, false);
             ......
1745 }
```

此时，虽然与 server 端第一次处理 SYN 包一样是调用 tcp_v4_hnd_req()函数，但是事过境迁，物是人非了（解释关子 3）。

在 tcp_v4_hnd_req()函数的 Line1723，再一次调用 inet_csk_search_req()函数在请求连接队列中查找对应的 request_sock。2.4 节费了半天劲，就是创建这个 request_sock，此时应该找到了。于是转到 Line1726 行，调用 tcp_check_req()函数。

tcp_check_req()函数的作用，是进行一些必要的检查，然后创建一个代表 server 与 client 端连接的 sock 结构 。此后 client 与 server 端的通信，都通过该 sock 结构完成。这个 sock 结构，是从 server 端原来的 sock 结构 clone 出来的，具有相同的 IP 地址、端口等属性。也就是说，server 端原来有一个 sock 结构，是应用程序通过 socket()系统调用创建的，这个 sock 结构一直处于 TCP_LISTEN 状态。当每个客户端与之建立连接时，它便 clone 一个自己，与客户端进行通信。只有这样，一个 sever 端才能同时与多个客户端建立多个连接。

net/ipv4/tcp_minisocks.c

```
503 struct sock *tcp_check_req(struct sock *sk, struct sk_buff *skb,
504                           struct request_sock *req,
505                           struct request_sock **prev,
506                           bool fastopen)
507 {
             ......
691         child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb, req, NULL);
692         if (child == NULL)
693                 goto listen_overflow;
694
695         inet_csk_reqsk_queue_unlink(sk, req, prev);
696         inet_csk_reqsk_queue_removed(sk, req);
697
698         inet_csk_reqsk_queue_add(sk, req, child);
699         return child;
700
701 listen_overflow:
702         if (!sysctl_tcp_abort_on_overflow) {
703                 inet_rsk(req)->acked = 1;
704                 return NULL;
705         }
706
```

```
707 embryonic_reset:
708         if (!(flg & TCP_FLAG_RST)) {
709                 /* Received a bad SYN pkt - for TFO We try not to reset
710                  * the local connection unless it's really necessary to
711                  * avoid becoming vulnerable to outside attack aiming at
712                  * resetting legit local connections.
713                  */
714                 req->rsk_ops->send_reset(sk, skb);
715         } else if (fastopen) { /* received a valid RST pkt */
716                 reqsk_fastopen_remove(sk, req, true);
717                 tcp_reset(sk);
718         }
719         if (!fastopen) {
720                 inet_csk_reqsk_queue_drop(sk, req, prev);
721                 NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_EMBRYONICRSTS);
722         }
723         return NULL;
724 }
```

Line691 的 syn_recv_sock 函数指针指向 tcp_v4_syn_recv_sock()。

```
net/ipv4/tcp_ipv4.c
1621 struct sock *tcp_v4_syn_recv_sock(struct sock *sk, struct sk_buff *skb,
1622                                    struct request_sock *req,
1623                                    struct dst_entry *dst)
1624 {
              ......
1634         if (sk_acceptq_is_full(sk))
1635                 goto exit_overflow;
1636
1637         newsk = tcp_create_openreq_child(sk, req, skb);
               ......
1699
1700         return newsk;
1701
1702 exit_overflow:
1703         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENOVERFLOWS);
1704 exit_nonewsk:
1705         dst_release(dst);
1706 exit:
1707         NET_INC_STATS_BH(sock_net(sk), LINUX_MIB_LISTENDROPS);
1708         return NULL;
1709 put_and_exit:
1710         inet_csk_prepare_forced_close(newsk);
1711         tcp_done(newsk);
1712         goto exit;
1713 }
```

我们先看看 Line1634。sk_acceptq_is_full()函数前面我们看到过定义了，它的作用是判断等待连接队列是否已满。这里因为假设 server 端只收到一个 连接请求，是不会满的。注意看看如果等待连接队列满了会怎样处理。Line1635 调到标号 Line1702，然后会递增 LINUX_MIB_LISTENOVERFLOWS 和 LINUX_MIB_LISTENDROPS 计数器，并返回 NULL 表示连接建立失败。这两个计数器后边进行程序分析时会看到。

```
net/ipv4/tcp_minisocks.c
379 struct sock *tcp_create_openreq_child(struct sock *sk, struct request_sock *req, struct sk_buff
*skb)
380 {
381         struct sock *newsk = inet_csk_clone_lock(sk, req, GFP_ATOMIC);
            ......
488         return newsk;
489 }
```

tcp_create_openreq_child ()函数调用 inet_csk_clone_lock()函数，创建一个新的 sock 结构体。这里的 clone 的意思上文提到过了，就是根据 server 端原有的 sk，新创建一个 newsk。

```
net/ipv4/inet_connection_sock.c
674 struct sock *inet_csk_clone_lock(const struct sock *sk,
675                                    const struct request_sock *req,
676                                    const gfp_t priority)
677 {
678         struct sock *newsk = sk_clone_lock(sk, priority);
            ......
683         newsk->sk_state = TCP_SYN_RECV;
700         return newsk;
701 }
```

inet_csk_clone_lock()函数根据原来的 sk，创建一个新的 newsk，并将状态设置为 TCP_SYN_RECV。这里大家可能比较奇怪，为什么不直接设置成 TCP_ESTABLISHED ？不是说连接建立已经完成了么？（关子 4）

```
net/core/sock.c
1433 struct sock *sk_clone_lock(const struct sock *sk, const gfp_t priority)
1434 {
1435         struct sock *newsk;
1436
1437         newsk = sk_prot_alloc(sk->sk_prot, priority, sk->sk_family);
              ......
1525         return newsk;
1526 }

net/core/sock.c
1247 static struct sock *sk_prot_alloc(struct proto *prot, gfp_t priority,
1248                 int family)
```

```
1249 {
1250        struct sock *sk;
1251        struct kmem_cache *slab;
1252
1253        slab = prot->slab;
1254        if (slab != NULL) {
1255              sk = kmem_cache_alloc(slab, priority & ~__GFP_ZERO);
1278        return sk;
             ......
1288 }
```

sk_clone_lock()和 sk_prot_alloc()函数比较简单，就是从 tcp 协议的 slab 中分配一个新的 sock 结构体。

tcp_create_openreq_child()、inet_csk_clone_lock()和 sk_clone_lock()函数对新分配的 sock（tcp_sock）进行了一些设置，这里从略了，直接回到 tcp_v4_syn_recv_sock()函数。tcp_v4_syn_recv_sock()会返回新生成的 sock，然后逻辑回到 tcp_check_req()函数 Line691。我们先看看如果 tcp_v4_syn_recv()函数的返回值为 NULL 时怎么处理。返回 NULL，表示等待连接队列已经满了，此时，如果 sysctl_tcp_abort_on_overflow 为 0，则 inet_rsk(req)->acked = 1，然后返回 NULL。这样依次回到 tcp_v4_hnd_req()函数和 tcp_v4_do_rcv ()函数，丢弃这个 client 发过来的 ACK。虽然是丢弃了这个 ACK，但是 req 中记录了 inet_rsk(req)->acked = 1，这个表明 client 端曾经发送过 ACK，但是因为等待连接队列的长度的限制，我们把它丢弃了。如果 sysctl_tcp_abort_on_overflow 为 1，我们通过 Line714 的 req->rsk_ops->send_reset(sk, skb)向 client 端发送一个 RST 包，断开连接。这是我们在 1.6 节看到“CONNECT FAILED !!!!!!: Connection reset by peer”的原因。

反之，如果创建新 sock 成功，child 不会为 NULL，于是 Line695 调用 inet_csk_reqsk_queue_unlink()将代表连接请求的 req 从请求连接队列中删除，并调用 inet_csk_reqsk_queue_removed()函数更新计数器，然后再调用 inet_csk_reqsk_queue_add()函数将代表连接请求的 req 加入到等待连接队列。我们来看看这三个函数。

include/net/inet_connection_sock.h

```
302 static inline void inet_csk_reqsk_queue_unlink(struct sock *sk,
303                                                 struct request_sock *req,
304                                                 struct request_sock **prev)
305 {
306       reqsk_queue_unlink(&inet_csk(sk)->icsk_accept_queue, req, prev);
307 }

187 static inline void reqsk_queue_unlink(struct request_sock_queue *queue,
188                                       struct request_sock *req,
189                                       struct request_sock **prev_req)
190 {
191       write_lock(&queue->syn_wait_lock);
```

```
192         *prev_req = req->dl_next;
193         write_unlock(&queue->syn_wait_lock);
194 }
```

reqsk_queue_unlink()函数将 req 从单向链表中删除。回想一下 syn_table[]哈希表是用单向链表来解决冲突的。

include/net/inet_connection_sock.h

```
273 static inline void inet_csk_reqsk_queue_removed(struct sock *sk,
274                                                 struct request_sock *req)
275 {
276         if (reqsk_queue_removed(&inet_csk(sk)->icsk_accept_queue, req) == 0)
277                 inet_csk_delete_keepalive_timer(sk);
278 }
```

include/net/request_sock.h

```
226 static inline int reqsk_queue_removed(struct request_sock_queue *queue,
227                                       struct request_sock *req)
228 {
229         struct listen_sock *lopt = queue->listen_opt;
230
231         if (req->num_timeout == 0)
232                 --lopt->qlen_young;
233
234         return --lopt->qlen;
235 }
```

reqsk_queue_removed()递减了两个计数器。如果该 req 在请求连接队列中待的时间很短，就不会有超时，所以 num_timeout 会是 0，所以 Line232 行递减 qlen_young，表示在请求连接队列中“年轻的”req 又少了一个。

Line234 递减 qlen，表示请求连接队列中的 req 又少了一个。

include/net/inet_connection_sock.h

```
261
262 static inline void inet_csk_reqsk_queue_add(struct sock *sk,
263                                             struct request_sock *req,
264                                             struct sock *child)
265 {
266         reqsk_queue_add(&inet_csk(sk)->icsk_accept_queue, req, sk, child);
267 }
```

include/net/request_sock.h

```
196 static inline void reqsk_queue_add(struct request_sock_queue *queue,
197                                    struct request_sock *req,
198                                    struct sock *parent,
199                                    struct sock *child)
200 {
```

```
201         req->sk = child;
202         sk_acceptq_added(parent);
203
204         if (queue->rskq_accept_head == NULL)
205                 queue->rskq_accept_head = req;
206         else
207                 queue->rskq_accept_tail->dl_next = req;
208
209         queue->rskq_accept_tail = req;
210         req->dl_next = NULL;
211 }
```

reqsk_queue_add()会把 req 与新生成 sock（即 child）相关连，然后加入到等待连接队列。这个等待连接队列用 rskq_accept_head 指向队列头，用 rskq_accept_tail 指向队列尾。

include/net/sock.h

```
 731 static inline void sk_acceptq_added(struct sock *sk)
 732 {
 733         sk->sk_ack_backlog++;
 734 }
```

终于看到 sk_ack_backlog 的修改函数了（解释关子 1）。 sk_acceptq_added()函数的作用是将等待连接的数量增加 1，表示在等待连接队列中又有一个新的项。

至此。tcp_check_req()的工作完成了，逻辑返回到 tcp_v4_hnd_req()进而返回到 tcp_v4_do_rcv()。

Line1814，nsk 指向新生成的 sock，而 sk 指向原来的 sk，nsk 是 sk 的 clone，两者这次不同了，于是在 Line1816 调用 tcp_child_process()函数对新产生的 nsk 进行进一步处理。

net/ipv4/tcp_minisocks.c

```
739 int tcp_child_process(struct sock *parent, struct sock *child,
740                         struct sk_buff *skb)
741 {
742         int ret = 0;
743         int state = child->sk_state;
744
745         if (!sock_owned_by_user(child)) {
746                 ret = tcp_rcv_state_process(child, skb, tcp_hdr(skb),
747                                             skb->len);
748                 /* Wakeup parent, send SIGIO */
749                 if (state == TCP_SYN_RECV && child->sk_state != state)
750                         parent->sk_data_ready(parent, 0);
751         } else {
752                 /* Alas, it is possible again, because we do lookup
753                  * in main socket hash table and lock on listening
754                  * socket does not protect us more.
755                  */
```

```
756             __sk_add_backlog(child, skb);
757         }
758
759         bh_unlock_sock(child);
760         sock_put(child);
761         return ret;
762 }
```

如果 Line745 行的判断为 TRUE，表示该 child 没有被锁住，于是 Line746 调用 tcp_rcv_state_process()进一步进行状态处理；否则，通过 Line756 行的__sk_add_backlog()函数将 skb 加入到 clild 的 backlog 队列，以后处理。
等等，什么叫“以后什么时间处理，在哪处理”。看看 net/core/sock.c 的 release_sock()函数就明白了。（提示一下，tcp_v4_do_rcv()这次会直接调到 Line1825 调用 tcp_rcv_state_process()进行处理了。）

总之殊途同归，不管是现在，还是等一会，都会调用 tcp_rcv_state_process()进行处理。在进入该函数之前，我们想想这个新的 sock 处于什么状态？对了， TCP_SYN_RECV。 如果忘了，回去看看 inet_csk_clone_lock()再接着往下看（解释关子 4）。

net/ipv4/tcp_input.c

```
5579 int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,
5580                           const struct tcphdr *th, unsigned int len)
5581 {
              ......
5656         /* step 5: check the ACK field */
5657         acceptable = tcp_ack(sk, skb, FLAG_SLOWPATH |
5658                                       FLAG_UPDATE_TS_RECENT) > 0;
5659
5660         switch (sk->sk_state) {
5661         case TCP_SYN_RECV:
5662                 if (!acceptable)
5663                         return 1;
                      ......
5681                 tcp_set_state(sk, TCP_ESTABLISHED);
                      ......
5717                 break;
                      ......
5781         }
              ......
5803         switch (sk->sk_state) {
              ......
5824         case TCP_ESTABLISHED:
5825                 tcp_data_queue(sk, skb);
5826                 queued = 1;
5827                 break;
```

```
5828        }
             ......
5840        return 0;
5841 }
```

Line5657 行，调用 tcp_ack()函数，如果返回 TRUE，表示接收的 ACK 是合法有效的。否则，是无效的。无效的 ACK 会导致逻辑从 Line5663 返回。否则，ACK 为有效的，会在 Line5681，将新生成的 sock 设置为 TCP_ESTABLISHED，表示 server 端建立连接的过程已经完了。至此，client 端和 server 端通过三次握手，tcp 连接终于建立。

Line5803 到 Line5828 的逻辑是处理数据的，即 client 端如果在返回第三个 ACK 时，如果顺便发送了数据，这里会进行处理。

好，原路返回，从 tcp_v4_do_rcv()函数 Line1820 成功退出。返回 tcp_v4_rcv()函数，从 Line2024 成功退出。Enjoy！

第 3 章 accept()系统调用分析

在 1.4 节我们看到，当 server.c 中的 TEST_LISTEN_BACKLOG 设置为 1，server 程序会一直在 for 循环中等待连接，这样就可以把 client 端的连接请求阻塞在等待连接队列中。反之，在 1.5 节， TEST_LISTEN_BACKLOG 设置为 0，server 程序会继续调用 accept()系统调用处理连接，这样 client 端的连接请求就不会阻塞在等待连接队列中，两端可以正常通信。

我们看看 accept()系统调用的实现。

3.1 accept()系统调用的分析

net/socket.c

```
1567 SYSCALL_DEFINE4(accept4, int, fd, struct sockaddr __user *, upeer_sockaddr,
1568             int __user *, upeer_addrlen, int, flags)
1569 {
1570        struct socket *sock, *newsock;
1571        struct file *newfile;
             ......
1581        sock = sockfd_lookup_light(fd, &err, &fput_needed);
1582        if (!sock)
1583             goto out;
1584
1585        err = -ENFILE;
1586        newsock = sock_alloc();
1587        if (!newsock)
1588             goto out_put;
1589
1590        newsock->type = sock->type;
1591        newsock->ops = sock->ops;
             ......
```

```
1599        newfd = get_unused_fd_flags(flags);
1600        if (unlikely(newfd < 0)) {
1601                err = newfd;
1602                sock_release(newsock);
1603                goto out_put;
1604        }
1605        newfile = sock_alloc_file(newsock, flags, sock->sk->sk_prot_creator->name);
1606        if (unlikely(IS_ERR(newfile))) {
1607                err = PTR_ERR(newfile);
1608                put_unused_fd(newfd);
1609                sock_release(newsock);
1610                goto out_put;
1611        }
                ......
1617        err = sock->ops->accept(sock, newsock, sock->file->f_flags);
1618        if (err < 0)
1619                goto out_fd;
                ......
1635        fd_install(newfd, newfile);
1636        err = newfd;
1637
1638 out_put:
1639        fput_light(sock->file, fput_needed);
1640 out:
1641        return err;
1642 out_fd:
1643        fput(newfile);
1644        put_unused_fd(newfd);
1645        goto out_put;
1646 }
```

从 line1570 和 Line1571 的变量定义我们看到，accept()要生成新的 socket 和新的 file，这样，2.6 节创建的新的 sock 就可以与新的 socket 相关连，然后再把新 socket 和新 file 相关连，这样 server 端就可以通过与文件系统的接口与 client 端通信了。

Line1581 查找 server 端原来的 socket 结构。

Line1586 调用 sock_alloc()函数分配一个新的 socket。

Line1590 和 Line1591 根据 server 端原来的 socket 结构设置新分配的 socket。

Line1599 分配一个新的未使用过的 fd。

Line1605 分配一个新的 file 结构，并与 newsock 相关联。

Line1617 中的 accept 函数指针指向 inet_accept()，这个我们需要详细分析一下。

net/ipv4/af_inet.c

```
700 int inet_accept(struct socket *sock, struct socket *newsock, int flags)
701 {
702        struct sock *sk1 = sock->sk;
```

```
703         int err = -EINVAL;
704         struct sock *sk2 = sk1->sk_prot->accept(sk1, flags, &err);
705
706         if (!sk2)
707                 goto do_err;
708
709         lock_sock(sk2);
710
711         sock_rps_record_flow(sk2);
712         WARN_ON(!((1 << sk2->sk_state) &
713                   (TCPF_ESTABLISHED | TCPF_SYN_RECV |
714                   TCPF_CLOSE_WAIT | TCPF_CLOSE)));
715
716         sock_graft(sk2, newsock);
717
718         newsock->state = SS_CONNECTED;
719         err = 0;
720         release_sock(sk2);
721 do_err:
722         return err;
723 }
```

Line704 的 accept 函数指针指向 inet_csk_accept()函数。

net/ipv4/inet_connection_sock.c

```
304 struct sock *inet_csk_accept(struct sock *sk, int flags, int *err)
305 {
306         struct inet_connection_sock *icsk = inet_csk(sk);
307         struct request_sock_queue *queue = &icsk->icsk_accept_queue;
308         struct sock *newsk;
309         struct request_sock *req;
310         int error;
311
312         lock_sock(sk);
313
314         /* We need to make sure that this socket is listening,
315          * and that it has something pending.
316          */
317         error = -EINVAL;
318         if (sk->sk_state != TCP_LISTEN)
319                 goto out_err;
320
321         /* Find already established connection */
322         if (reqsk_queue_empty(queue)) {
323                 long timeo = sock_rcvtimeo(sk, flags & O_NONBLOCK);
324
325                 /* If this is a non blocking socket don't sleep */
```

```
326             error = -EAGAIN;
327             if (!timeo)
328                     goto out_err;
329
330             error = inet_csk_wait_for_connect(sk, timeo);
331             if (error)
332                     goto out_err;
333     }
334     req = reqsk_queue_remove(queue);
335     newsk = req->sk;
336
337     sk_acceptq_removed(sk);
338     if (sk->sk_protocol == IPPROTO_TCP && queue->fastopenq != NULL) {
339             spin_lock_bh(&queue->fastopenq->lock);
340             if (tcp_rsk(req)->listener) {
341                     /* We are still waiting for the final ACK from 3WHS
342                      * so can't free req now. Instead, we set req->sk to
343                      * NULL to signify that the child socket is taken
344                      * so reqsk_fastopen_remove() will free the req
345                      * when 3WHS finishes (or is aborted).
346                      */
347                     req->sk = NULL;
348                     req = NULL;
349             }
350             spin_unlock_bh(&queue->fastopenq->lock);
351     }
352 out:
353     release_sock(sk);
354     if (req)
354     if (req)
355             __reqsk_free(req);
356     return newsk;
357 out_err:
358     newsk = NULL;
359     req = NULL;
360     *err = error;
361     goto out;
362 }
```

从 line318 可以再次印证，server 端一开始建立的 sock 始终是处于 TCP_LISTEN 状态的。Line322 判断等待连接队列是否为空，如果为空，则表示 server 端还没有收到顺利经过三次握手而建立的连接请求，那么会调用 inet_csk_wait_for_connect()函数进行等待。

否则，如果等待连接队列不为空，说明已经有经过三次握手的建立的连接请求了。Line334 调用 reqsk_queue_remove()函数将连接请求从等待连接队列中删除。

include/net/request_sock.h

```
213 static inline struct request_sock *reqsk_queue_remove(struct request_sock_queue *queue)
214 {
215         struct request_sock *req = queue->rskq_accept_head;
216
217         WARN_ON(req == NULL);
218
219         queue->rskq_accept_head = req->dl_next;
220         if (queue->rskq_accept_head == NULL)
221                 queue->rskq_accept_tail = NULL;
222
223         return req;
224 }
```

可以看到这个 reqsk_queue_remove()函数实现的就是从单向链表中删除第一项的工作。

然后回到 inet_csk_accept()函数的 Line335，将 newsk 指向 2.6 节为 req 创建的 sock，将来作为 inet_csk_accept()函数的返回值返回给 inet_accept()函数。

inet_csk_accept()函数的 Line337 调用 sk_acceptq_removed()函数，递减 sk_ack_backlog 计数，表明等待连接队列中又少了一项。（解释关子 1）

如果一切正常，inet_csk_accept()函数从 Line356 返回到 inet_accept()函数的 Line704。inet_accept()函数 Line716，调用 sock_graft()函数将 sock 与 socket 相关联。

include/net/sock.h

```
1722 static inline void sock_graft(struct sock *sk, struct socket *parent)
1723 {
1724         write_lock_bh(&sk->sk_callback_lock);
1725         sk->sk_wq = parent->wq;
1726         parent->sk = sk;
1727         sk_set_socket(sk, parent);
1728         security_sock_graft(sk, parent);
1729         write_unlock_bh(&sk->sk_callback_lock);
1730 }
```

```
1695 static inline void sk_set_socket(struct sock *sk, struct socket *sock)
1696 {
1697         sk_tx_queue_clear(sk);
1698         sk->sk_socket = sock;
1699 }
```

最后，inet_accept()函数返回到 SYSCALL_DEFINE4(accept4）Line1617。继续向下运行，Line1635 调用的 fd_install()函数，是个对用户程序非常重要的函数。

fs/file.c

```
579 void fd_install(unsigned int fd, struct file *file)
580 {
581         __fd_install(current->files, fd, file);
```

```
582 }

568 void __fd_install(struct files_struct *files, unsigned int fd,
569                 struct file *file)
570 {
571         struct fdtable *fdt;
572         spin_lock(&files->file_lock);
573         fdt = files_fdtable(files);
574         BUG_ON(fdt->fd[fd] != NULL);
575         rcu_assign_pointer(fdt->fd[fd], file);
576         spin_unlock(&files->file_lock);
577 }
```

__fd_install()函数 Line575，会将新分配的 newfd 与新分配的 file 结构相关联，此 newfd 会最为 accetp()系统调用的返回值返回给应用程序使用，这样 server 端应用程序就可以通过 newfd 与 client 端进行通信了。

accept()系统调用比较简单，总结一下：

1) 分配新的 fd、file 和 socket；
2) 调用 inet_csk_accept()函数，将连接请求从等待连接队列中删除，并把 2.6 节创建的 sock 结构返回；
3) 调用 inet_accept()函数，将 inet_csk_accept()函数返回的 sock 与新分配的 socket 相关联；
4) 将 fd 与 file、socket 相关联，这样应用程序就可以通过 fd 与 client 端通信了。

3.2 连接请求的生命周期

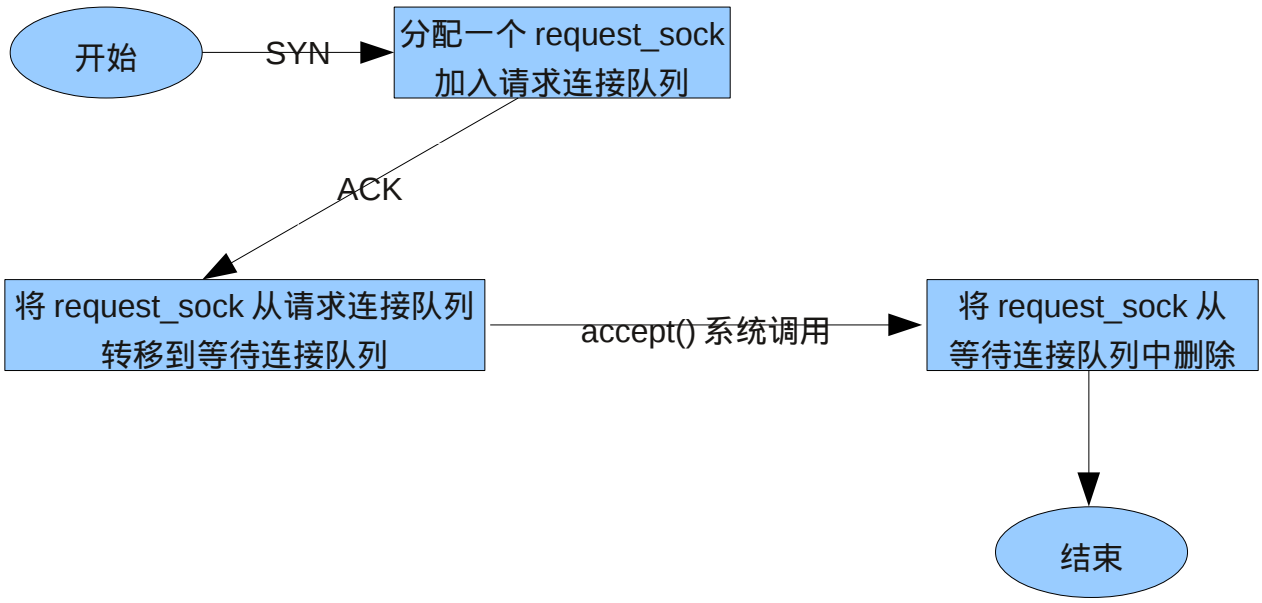

图 4 连接请求的生命周期

2.4 节我们看到，当 server 端接收到一个 SYN 包后，会为之分配一个 request_sock 结构体，代表一个连接请求。该 request_sock 一开始是放在请求连接队列中，即 syn_table[]哈希表中。请求连接队列中的 request_sock 数量是有限制的，inet_csk_reqsk_queue_is_full()用于测试其是否满了。

2.6 节我们看到，当 server 端就收到 ACK 包后，会将 request_sock 从请求连接对列中删除，然后加入到等待连接队列中。（这里，等待的意思，3.1 节介绍的很清楚了，就是等待 accept()系统调用的意思）。等待连接队列中的 request_sock 数量，也是有限制的，sk_acceptq_is_full()用于测试其是否满了。

3.1 节，应用程序调用 accept()系统调用，会将 request_sock 从等待连接队列中删除，然后将 fd、socket、sock 分别相关联，并将用于通信的 fd 返回给应用程序。

3.2 节也是解释关子 0 的。

第 4 章 listen()系统调用的参数 backlog 详解

本章将详细分析一下 backlog 参数的作用，并详细分析第一章中的程序的行为。

4.1 listen()系统调用的参数 backlog 的作用

作用 1

从 2.2 和 2.4 节我们看到，backlog 决定了 max_qlen_log 的值，从而决定了请求连接队列中，当 request_sock 的数量达到多少时，可以认为该队列已经满了。如果该队列已经满了，当遇到一个新的连接请求时，会在 tcp_v4_conn_request()函数 Line1472 丢弃该请求。

这样，就可以通过 backlog 的值限制在“很小的一段时间内”，server 端能处理的连接请求的数量。

那么这个“很小的时间段”是什么？其实就是 server 端已经接收到 SYN 包，但是尚未接收到 ACK 包这个时间段。

作用 2

从 2.2 和 2.4 节我们还可以看到，backlog 决定了 sk_max_ack_backlog 的值，从而决定了等待连接队列中，当 request_sock 的数量达到多少时，可以认为该队列已经满了。如果该队列已经满了，当遇到一个新的连接请求时，会在 tcp_v4_conn_request()函数 Line1482 丢弃该请求。（当然丢弃请求时，inet_csk_reqsk_queue_young()必须大于 1，表示等待连接队列中“年轻的”连接请求数量大于 1）

这样，就可以通过 backlog 的值限制等待连接队列的长度。也就是说，等待连接队列的长度是受限制的，它限制了当三次握手已经完成，但是 server 端尚未调用 accept()系统调前等待连接队列中 request_sock 的数量。

从 2.6 节我们可以看到，当 server 端接收了 client 端的 ACK 包后，如果发现等待连接队列已经满了，那么它会递增 LINUX_MIB_LISTENOVERFLOWS 和 LINUX_MIB_LISTENDROPS 计数，然后丢弃该 ACK。

4.2 分析 1.4 节中程序的行为

1 在第一个终端运行./server
第 1 步的作用是启动 server 端，使其在 20000 端口监听 tcp 连接请求。并且设置 listen ()系统调用的 backlog 参数为 1，这样 max_qlen_log 为 3，即请求连接队列中超过 7 个 request_sock，则表示请求连接队列满了，不能接收新的请求了。同时，因为 backlog 为 1，则 sk_max_ack_backlog 与之相等，也为 1，这样如果等待连接中 request_sock 的数量大于 1 个，就表示等待连接队列满了。
2 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'，结果类似下面：
第 1 步的作用是观察 server 端是否在监听 tcp 连接请求
3 在第三个终端运行./client
第 3 步，client 端向 server 端第一次发出连接请求，因为 server 端以前没有接收到过连接请求，故请求连接对列和等待连接队列都为空，这样经过三次握手以后，client 可以顺利地与 server 端建立连接。但是要注意，server 端此时始终在 for 循环中，并没有调用 accept()系统调用，所以两者之间是不能进行通信的。所以即使 client 端向 server 端发送了数据，server 端也不会向 client 端发送数据的。这里我们在 server 端利用 for 循环，将 client 的连接请求阻塞在等待连接队列中。
第 3 步 client 与 server 建立连接后，listen_sock->qlen 为 0（注意，这个值在收到 SYN 包后先变为 1，在收到 ACK 包后变为 0），listen_sock->qlen_young 也为 0，sock->sk_ack_backlog 为 1。
4 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'
绿色的两行表示（127.0.0.1，45484，127.0.0.1，20000）四元组确定的唯一一个 tcp 连接。
5 在第四个终端运行./client
逻辑类似于第 3 步。因为此时 listen_sock->qlen 为 0，sock->sk_ack_backlog 为 1，所以 tcp_v4_conn_request()和 tcp_v4_syn_recv_sock()函数中的 inet_csk_reqsk_queue_is_full()以及 sk_acceptq_is_full()检测都不会阻止 server 端处理该连接请求，所以 client 可以顺利地与 server 端建立了第二个 tcp 连接。
此时 listen_sock->qlen 仍为 0，listen_sock->qlen_young 也仍为 0，但是 sock->sk_ack_backlog 变为 2。
6 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'
绿色的两行表示（127.0.0.1，45485，127.0.0.1，20000）四元组确定的唯一一个 tcp 连接。
7 在第五个终端运行./client

屏幕显示“connected to server: IP 127.0.0.1 PORT 20000”，表示连接成功。
这是为什么呢？
从 server 端的角度来看，在 tcp_v4_conn_request()中，inet_csk_reqsk_queue_is_full()返回 FALSE，虽然 sk_acceptq_is_full()返回 TRUE，但是 Line1480 还有另一个条件，inet_csk_reqsk_queue_young(sk) > 1 为 FALSE，所以 tcp_v4_conn_request()不会丢弃这个 SYN 包，继续处理连接请求。然后 server 端会顺利地向 client 端发送 SYN/ACK 包。
当该 SYN/ACK 包到达 client 端，client 端会认为与 server 端的连接已经建立，随之向 server 端发送 ACK，然后开始向 server 端发送数据。
但是，在 server 端接收到这个 ACK 后，会调用 tcp_v4_syn_recv_sock()函数处理，Line1634 sk_acceptq_is_full()返回 TRUE，于是丢弃该 ACK 包。所以对于 server 端来说，连接并未真正建立。而且 request_sock 仍会保存在请求连接队列中，并不会转移到等待连接队列中。
然后 server 端会递增 LINUX_MIB_LISTENOVERFLOWS 和 LINUX_MIB_LISTENDROPS 计数，反应到 netstat -s 中的输出上，“times the listen queue of a socket overflowed”以及“SYNs to LISTEN sockets ignored”都会增加。
8 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'
我们看到，对于 client 端来说，它认为连接已经建立，于是(127.0.0.1, 33769, 127.0.0.1, 20000)这个四元组表示的连接状态为 ESTABLISHED。但是对于 server 端来说，它认为连接尚未建立起来， 于是(127.0.0.1, 20000, 127.0.0.1, 33769)这个四元组表示的连接状态为 SYN_RECV。
这里就有一个小问题，为什么显示 SYN_RECV 呢？实际上在 server 端只为 SYN 包创建了一个 request_sock 结构，并未创建 sock 结构啊。实际上，这个是 netstat 显示的问题。netstat 是从/proc/net/tcp 获取数据进行显示的，而 tcp 在处理/proc/net/tcp 的输出时，会将所有在请求连接队列中排队的 request_sock 显示为 TCP_SYN_RECV 状态，表示 SYN 包已经收到了，虽然连接尚未完全建立。
如果你有兴趣研究这个显示的过程，提示一下：

net/ipv4/tcp_ipv4.c

```
2600 static void get_openreq4(const struct sock *sk, const struct request_sock *req,
2601                           struct seq_file *f, int i, kuid_t uid, int *len)
2602 {
2603         const struct inet_request_sock *ireq = inet_rsk(req);
2604         long delta = req->expires - jiffies;
2605
2606         seq_printf(f, "%4d: %08X:%04X %08X:%04X"
2607                 " %02X %08X:%08X %02X:%08lX %08X %5u %8d %u %d %pK%n",
2608                 i,
2609                 ireq->loc_addr,
2610                 ntohs(inet_sk(sk)->inet_sport),
2611                 ireq->rmt_addr,
2612                 ntohs(ireq->rmt_port),
```

```
2613            TCP_SYN_RECV,
2614            0, 0, /* could print option size, but that is af dependent. */
2615            1,    /* timers active (only the expire timer) */
2616            jiffies_delta_to_clock_t(delta),
2617            req->num_timeout,
2618            from_kuid_munged(seq_user_ns(f), uid),
2619            0,  /* non standard timer */
2620            0, /* open_requests have no inode */
2621            atomic_read(&sk->sk_refcnt),
2622            req,
2623            len);
2624 }
```

9 在第二个终端运行 tcpdump -i lo -p tcp -nn

红色的一行，表示 server 端认为与 client 的连接尚未建立，于是再一次发送 SYN/ACK 包给 client 端，请求继续建立连接。注意红色行与第一次的绿色行是一样的。那么这第二个 SYN/ACK 是由什么机制处理的？

是由 2.4 节提到的 request_sock 的老化机制负责发送的。inet_csk_reqsk_queue_prune()函数会定期检查请求连接队列，然后替尚未建立连接的 request_sock 发送 SYN/ACK 包给 client 端。参见：

net/ipv4/inet_connection_sock.c

```
560 int inet_rtx_syn_ack(struct sock *parent, struct request_sock *req)
561 {
562         int err = req->rsk_ops->rtx_syn_ack(parent, req);
563
564         if (!err)
565                 req->num_retrans++;
566         return err;
567 }
```

这个 rtx_syn_ack 函数指针指向 tcp_v4_rtx_synack()函数。

net/ipv4/tcp_ipv4.c

```
852 static int tcp_v4_rtx_synack(struct sock *sk, struct request_sock *req)
853 {
854         int res = tcp_v4_send_synack(sk, NULL, req, 0);
855
856         if (!res)
857                 TCP_INC_STATS_BH(sock_net(sk), TCP_MIB_RETRANSSEGS);
858         return res;
859 }
```

10 继续在新的终端运行./client，结果与第 7 步的类似。看来超过两个连接之后行为都是类似的。这是因为 listen_sock->qlen 仍为 0，listen_sock->qlen_young 也仍为 0，而 sock->sk_ack_backlog 仍为 2，sk_acceptq_is_full()函数会与第 7 步一样，阻止

tcp_v4_syn_recv_sock()继续建立连接。

4.3 分析 1.5 节中程序的行为

1 在第一个终端运行./server
这次把 server.c 程序的 TEST_LISTEN_BACKLOG 设置为 0，这样 for 循环就不会阻止 accept()系统调用了。
2 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'
红色部分表示 server 端正在本地的 20000 端口等待 tcp 连接。
3 在第三个终端运行./client
可以看到 client 端与 server 端可以正常通信了。
因为此时 server 端调用 accept()系统调用处理连接请求了，故双方可以正常通信了。因为 server 端接收到 SYN 包，会把连接请求先排入请求连接队列。然后 server 端接收到 ACK 包，会将连接请求排入等待连接队列。接着 server 端调用 accept()系统调用，会将连接请求从等待连接队列中删除。故此时 listen_sock->qlen 为 0，listen_sock->qlen_young 也为 0，而 sock->sk_ack_backlog 也为 0。
4 在第四个终端运行./client
结果和过程都与第 3 步类似，因为不管是 listen_sock->qlen 还是 sock->sk_ack_backlog 都不会阻止新的连接请求。
5 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'
可以看到两对连接已经建立起来。
6 在第五个终端运行./client
结果和过程都与第 3 步类似，因为不管是 listen_sock->qlen 还是 sock->sk_ack_backlog 都不会阻止新的连接请求。
7 在第二个终端运行 netstat -a -t -n |grep -E '127|0.0.0.0'
可以看到 client 端与 server 端已经建立了三对 tcp 连接了，并且都能正常通信。
8 在新的终端继续运行./client，结果与第 6 步的类似。
虽然这第二次实验 backlog 参数为 1，但是它不能限制 server 端的连接数量。这要与第一次实验区分开来。在第一次实验中，server 端并未调用 accept()系统调用处理连接请求，故连接请求一直阻塞在等待连接队列中。而 backlog 可以直接限制等待连接队列的长度，所以限制了超过 backlog+1（这里是 2）个的连接请求。
但是在第二次实验中，server 端会调用 accept()系统调用处理连接请求，故连接请求基本不会在请求连接队列和等待连接队列中等待太长的时间，就被从队列中删除了，所以 listen_sock->qlen 一直为 0，sock->sk_ack_backlog 也一直为 0，并不会限制 server 端接收新的连接请求。
那么，是不是说 backlog 参数在第二次实验中就没有用了呢？
不是的。请注意前面一段有两个红色的词组，我用的词都很暧昧。“基本不会”，就说明还是

有可能会的。“等待太长”，那么多长算是太长？
对于 server 端来说，有两个很重要的时间段。
第一个是接收到 SYN 包后到接收到 ACK 包之前的这个时间段。因为在这个时间段中，连接请求会一直停留在请求连接队列中。此时新的连接请求会受到 listen_sock->qlen 的限制的。如果 server 端在此时间段中接收到太多的连接请求，可以用 backlog 来控制。
第二个是接收到 ACK 包到 server 端调用 accept()系统调用这个时间段。这个时间段中，连接请求会一直停留在等待连接队列中，此时新的连接请求会受到 sock->sk_ack_backlog 的限制。这个时间段可以分成两部分（进程调度和上下文切换的时间忽略），一部分是 kernel 处理 accept()的时间，这个时间基本上可以认为是固定的，比较短的。另一部分是应用程序前后两次调用 accept()的时间间隔，这个时间就不是固定的了。所以如果 server 想处理尽量多的连接请求，就需要尽可能缩短两次调用 accept()系统调用的时间间隔。如果 server 在期间做大量耗时的操作的话，会影响到处理新的连接请求。

4.4 分析 1.6 节中程序的行为

第三次实验与/proc/sys/net/ipv4/tcp_abort_on_overflow 相关，它的意思是当请求连接队列或者等待连接队列溢出时，server 端会向 client 端发送一个 RST 包，断开连接，即告诉 client 端：我这很忙呢，别烦我了，哪凉快哪待着去吧。这个过程在 2.6 节 tcp_check_req()函数 Line714。
1.6 节的实验结果都很直观，这里不做过多解释了。

4.5 backlog 参数的用法

从 kernel 代码的角度来说，如果 server 端想限制请求连接队列和等待连接队列的长度，可以使用 backlog 参数。从应用程序的角度来说，如果 server 端想限制一个很小的时间段内能够处理的新的连接请求的数量，可以使用 backlog 参数。并且如果 server 端很强势，超出限制后希望断开连接，那么设置/proc/sys/net/ipv4/tcp_abort_on_overflow 为 1。

附录 1 client.c

```
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <time.h>
#include <arpa/inet.h>


#define SERVER_PORT 20000 // define the defualt connect port id
#define CLIENT_PORT ((20001+rand()) % 65536) // define the defualt client port as a random port
#define BUFFER_SIZE 256

int main(int argc, char **argv)
{
    char buf[BUFFER_SIZE];
    int client_fd, length = 0;
    struct sockaddr_in server_addr,client_addr;
    socklen_t socklen = sizeof(server_addr);

    client_fd = socket(AF_INET, SOCK_STREAM, 0);

    srand(time(NULL));//initialize random generator
    bzero(&client_addr, sizeof(client_addr));
    client_addr.sin_family = AF_INET;
    client_addr.sin_port = htons(CLIENT_PORT);
    client_addr.sin_addr.s_addr = htons(INADDR_ANY);

    bzero(&server_addr, sizeof(server_addr));
    server_addr.sin_family = AF_INET;
    inet_aton("127.0.0.1", &server_addr.sin_addr);
    server_addr.sin_port = htons(SERVER_PORT);

    errno = 0;
    if (connect(client_fd, (struct sockaddr*)&server_addr, socklen)){
        perror("CONNECT FAILED !!!!!!");
        close(client_fd);
        exit(1);
    }

    printf("connected to server: IP %s PORT %d\n",
```

```
            inet_ntoa(server_addr.sin_addr),
            ntohs(server_addr.sin_port));

    while (1)
    {
        sleep(2);
        memset(buf, '\0', BUFFER_SIZE);
        strncat(buf, "hello, world!", BUFFER_SIZE);
        if (send(client_fd, buf, strlen(buf), 0) < 0) {
            fprintf(stderr, "send message error\n");
            break;
        }
        memset(buf, '\0', BUFFER_SIZE);
        length = recv(client_fd, buf, BUFFER_SIZE, 0);
        printf("length=%d\n", length);
        if (length > 0) {
            buf[length-1] = '\0';
            fprintf(stderr, "recv from server: %s \n", buf);
        } else
            break;
    }

    close(client_fd);

    return 0;
}
```

附录 2 server.c

编译 gcc -o server server.c -lpthread

```
#include <stdio.h>
#include <sys/socket.h>
#include <unistd.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>
#include <time.h>
#include <arpa/inet.h>

#define TEST_LISTEN_BACKLOG 1

#define SERVER_PORT 20000           // define the defualt connect port id
#define LENGTH_OF_LISTEN_QUEUE 1      // length of listen queue in server
```

```
#define BUFFER_SIZE 255

void *thread(void *client_fd)
{
    char buf[BUFFER_SIZE] = {'\0'};
    long timestamp;

    while (1) {
        memset(buf, '\0', BUFFER_SIZE);

        if (recv((*(int *)client_fd), buf, BUFFER_SIZE, 0) < 0) {
            fprintf(stderr, "get message from client error\n");
            close(*(int *)client_fd);
            break;
        }

        sleep(1);
        strcat(buf, " : ");
        timestamp = time(NULL);
        strcat(buf, ctime(&timestamp));
        puts(buf);
        send((*(int *)client_fd), buf, strlen(buf), 0);
    }

    return NULL;
}

int main(int argc, char **argv)
{
    pthread_t thread_id;
    int server_fd,client_fd;
    struct sockaddr_in servaddr;
    struct sockaddr_in client_addr;

    server_fd = socket(AF_INET, SOCK_STREAM, 0);

    bzero(&servaddr, sizeof(servaddr));
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(SERVER_PORT);
    servaddr.sin_addr.s_addr = htons(INADDR_ANY);

    bind(server_fd, (struct sockaddr*)&servaddr, sizeof(servaddr));
    listen(server_fd, LENGTH_OF_LISTEN_QUEUE);

    #if TEST_LISTEN_BACKLOG
    /* wait the client to connect */
    for (;;) {
```

```
        printf("server is listening! \n");
        sleep(1);
    }
    #else
        printf("server is listening! \n");
    #endif

    while (1) {
        socklen_t length = sizeof(client_addr);

        client_fd = accept(server_fd, (struct sockaddr*)&client_addr, &length);
        printf("a client connected: IP %s PORT %d\n",
                inet_ntoa(client_addr.sin_addr),
                ntohs(client_addr.sin_port));

        if (pthread_create(&thread_id, NULL, thread, (void *)&client_fd) < 0) {
            fprintf(stderr, "create thread error\n");
            exit(1);
        }
    }

    close(server_fd);

    return 0;
}
```