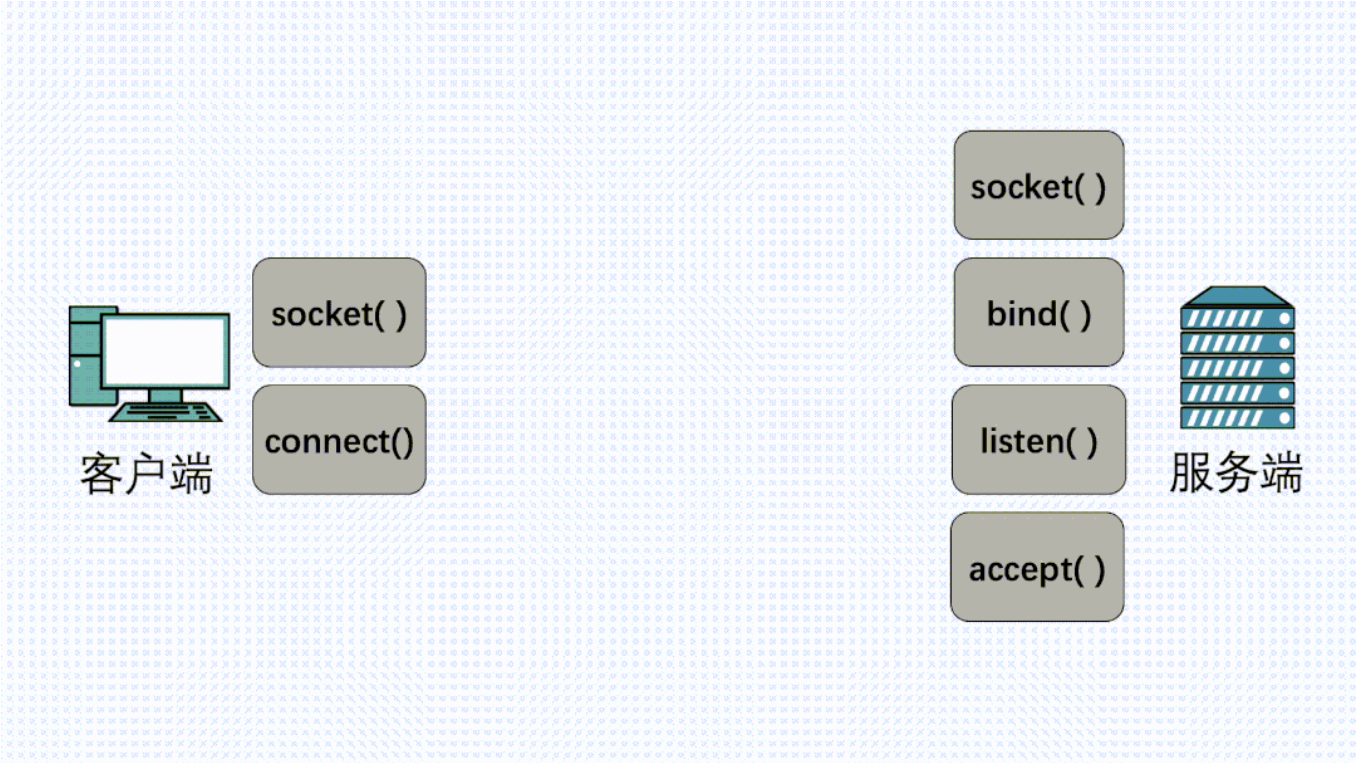
动图图解！没有accept，能建立TCP连接吗？

Original 小白 小白debug 2021-09-23 17:40

收录于合集

#面试 16 #tcp 7 #图解网络 17 #后端 18 #计算机基础 11



握手建立连接流程

上面这个动图，是我们平时客户端和服务端建立连接时的代码流程。

对应的是下面一段简化过的服务端伪代码。

```
int main()
{
    /*Step 1: 创建服务器端监听socket描述符listen_fd*/
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);

    /*Step 2: bind绑定服务器端的IP和端口，所有客户端都向这个IP和端口发送和请求数据*/
    bind(listen_fd, xxx);

    /*Step 3: 服务端开启监听*/
    listen(listen_fd, 128);

    /*Step 4: 服务器等待客户端的链接，返回值cfd为客户端的socket描述符*/
```

```
    cfd = accept(listen_fd, xxx);

      /*Step 5: 读取客户端发来的数据*/
      n = read(cfd, buf, sizeof(buf));
}
```

估计大家也是老熟悉这段伪代码了。

需要注意的是，在执行 `listen()` 方法之后还会执行一个 `accept()` 方法。

一般情况下，如果启动服务器，会发现最后程序会**阻塞在** `accept()` 里。

此时服务端就算ok了，就等客户端了。

那么，再看下简化过的客户端伪代码。

```
int main()
{
    /*Step 1: 创建客户端端socket描述符cfd*/
    cfd = socket(AF_INET, SOCK_STREAM, 0);

    /*Step 2: connect方法,对服务器端的IP和端口号发起连接*/
    ret = connect(cfd, xxxx);

    /*Step 4: 向服务器端写数据*/
    write(cfd, buf, strlen(buf));
}
```

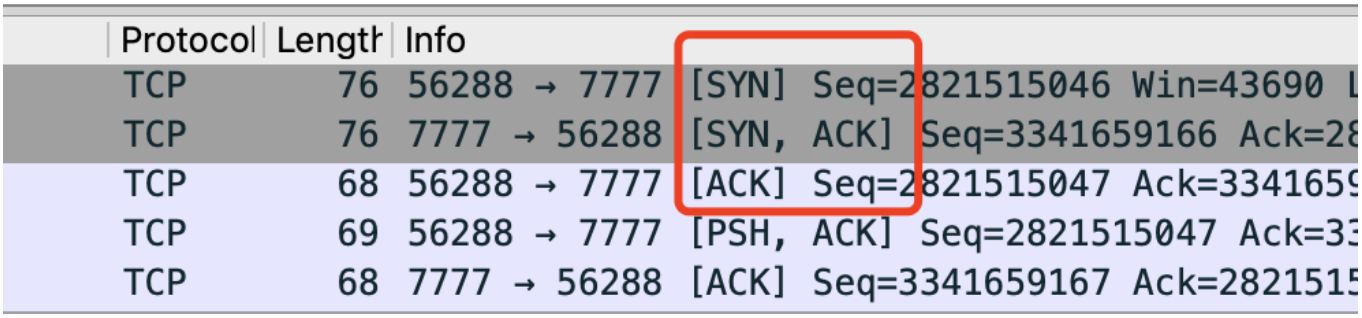
客户端比较简单，创建好 `socket` 之后，直接就发起 `connect` 方法。

此时回到服务端，会发现**之前一直阻塞的accept方法，返回结果了**。

这就算两端成功建立好了一条连接。之后就可以愉快的进行读写操作了。

那么，我们今天的问题是，**如果没有这个accept方法，TCP连接还能建立起来吗？**

其实只要在执行 `accept()` 之前执行一个 `sleep(20)` ，然后立刻执行客户端相关的方法，同时抓个包，就能得出结论。



不执行accept时抓包结果

从抓包结果看来，**就算不执行accept()方法，三次握手照常进行，并顺利建立连接。**

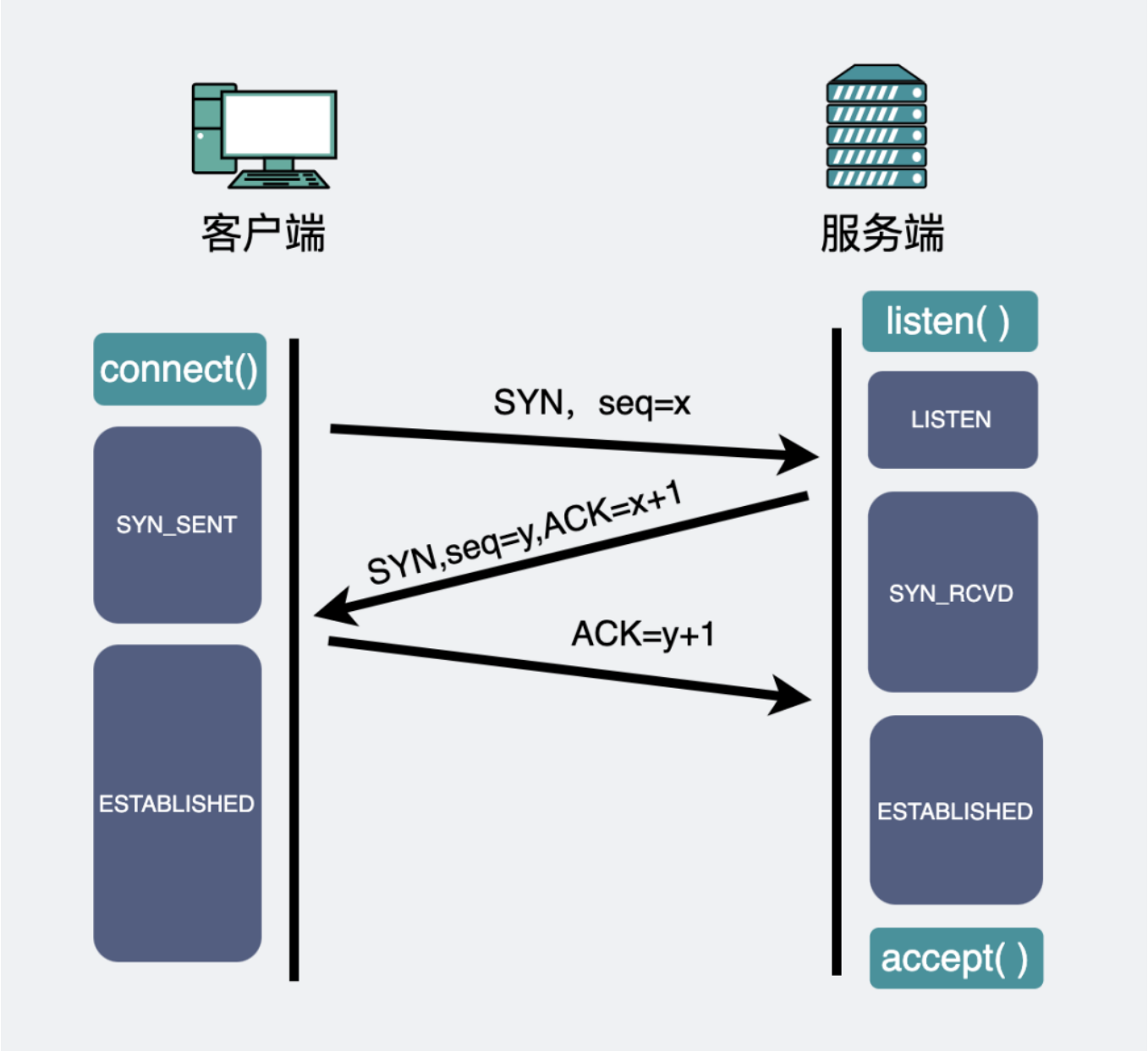
更骚气的是，**在服务端执行accept()前，如果客户端发送消息给服务端，服务端是能够正常回复ack确认包的。**

并且， `sleep(20)` 结束后，服务端正常执行 `accept()` ，客户端前面发送的消息，还是能正常收到的。

通过这个现象，我们可以多想想为什么。顺便好好了解下三次握手的细节。

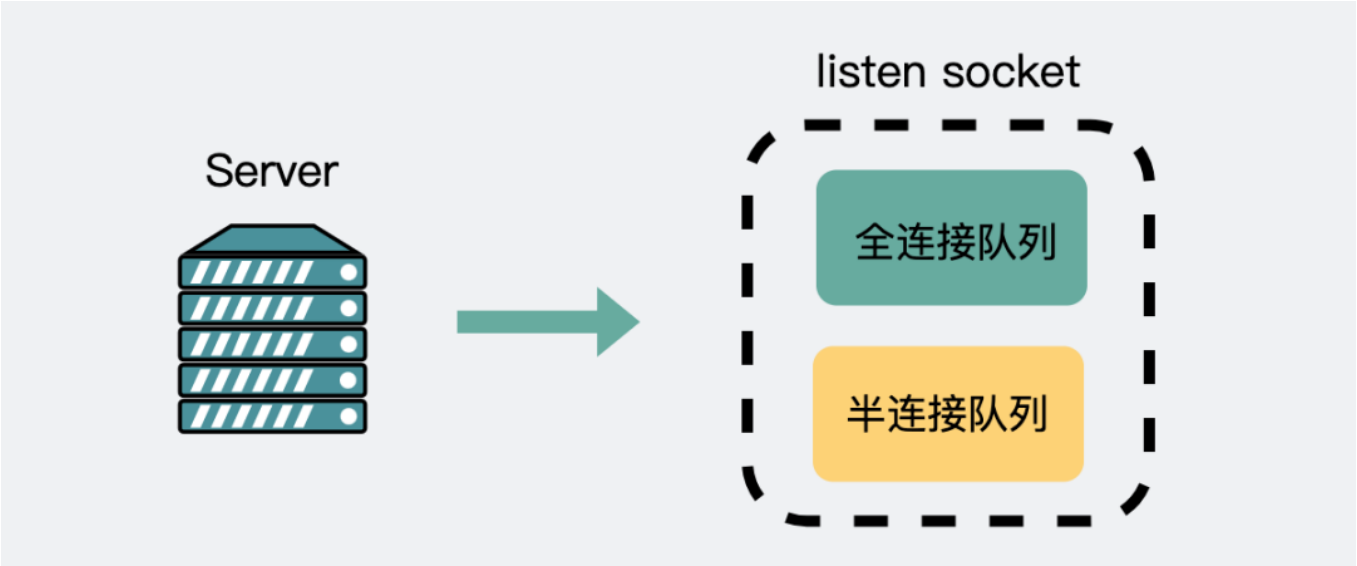
三次握手的细节分析

我们先看面试八股文的老股，三次握手。



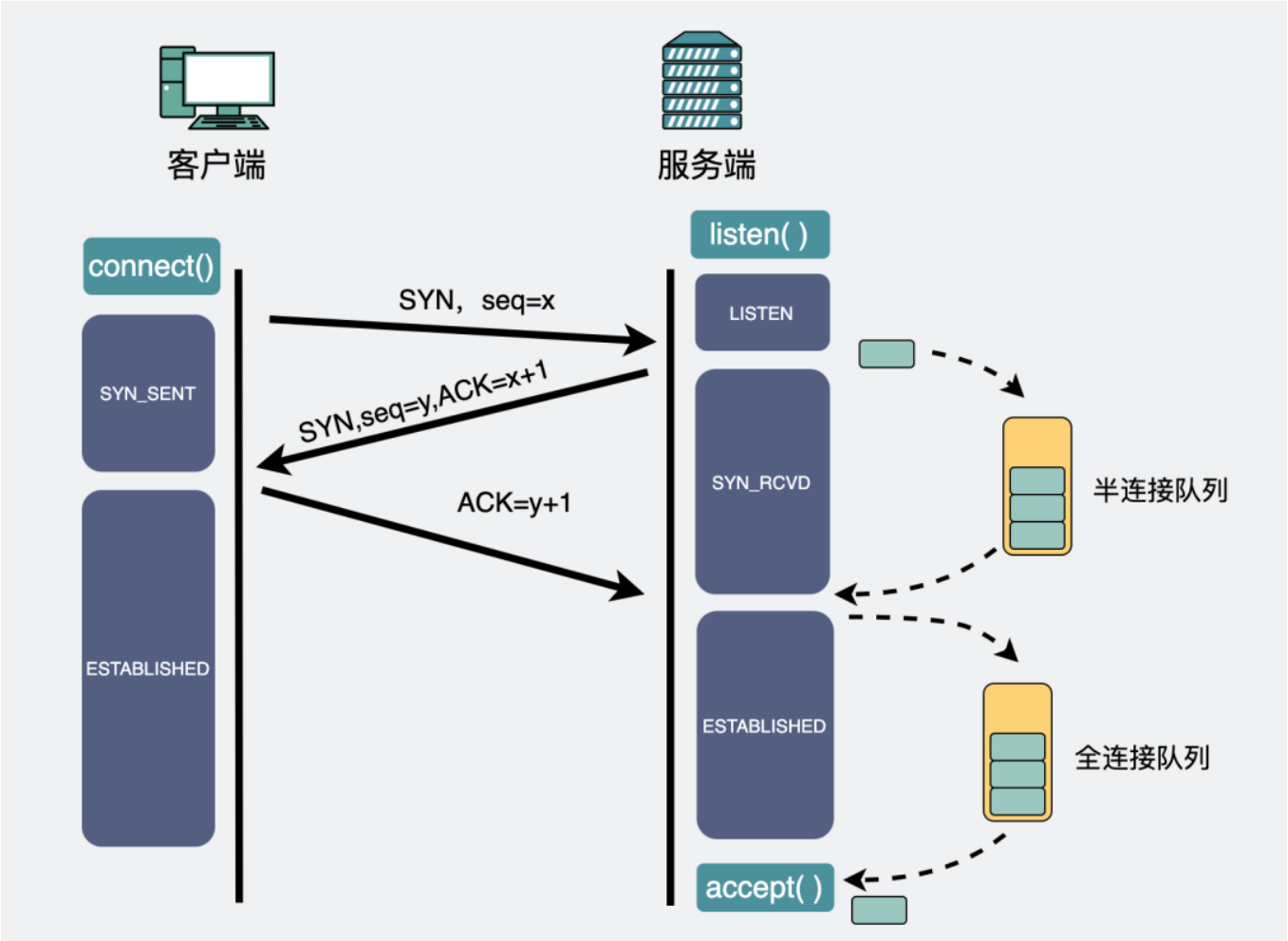
TCP三次握手

服务端代码，对socket执行bind方法可以绑定监听端口，然后执行 `listen方法` 后，就会进入监听（ `LISTEN` ）状态。内核会为每一个处于 `LISTEN` 状态的 `socket` 分配两个队列，分别叫**半连接队列和全连接队列**。



每个listen Socket都有一个全连接和半连接队列

半连接队列、全连接队列是什么



半连接队列和全连接队列

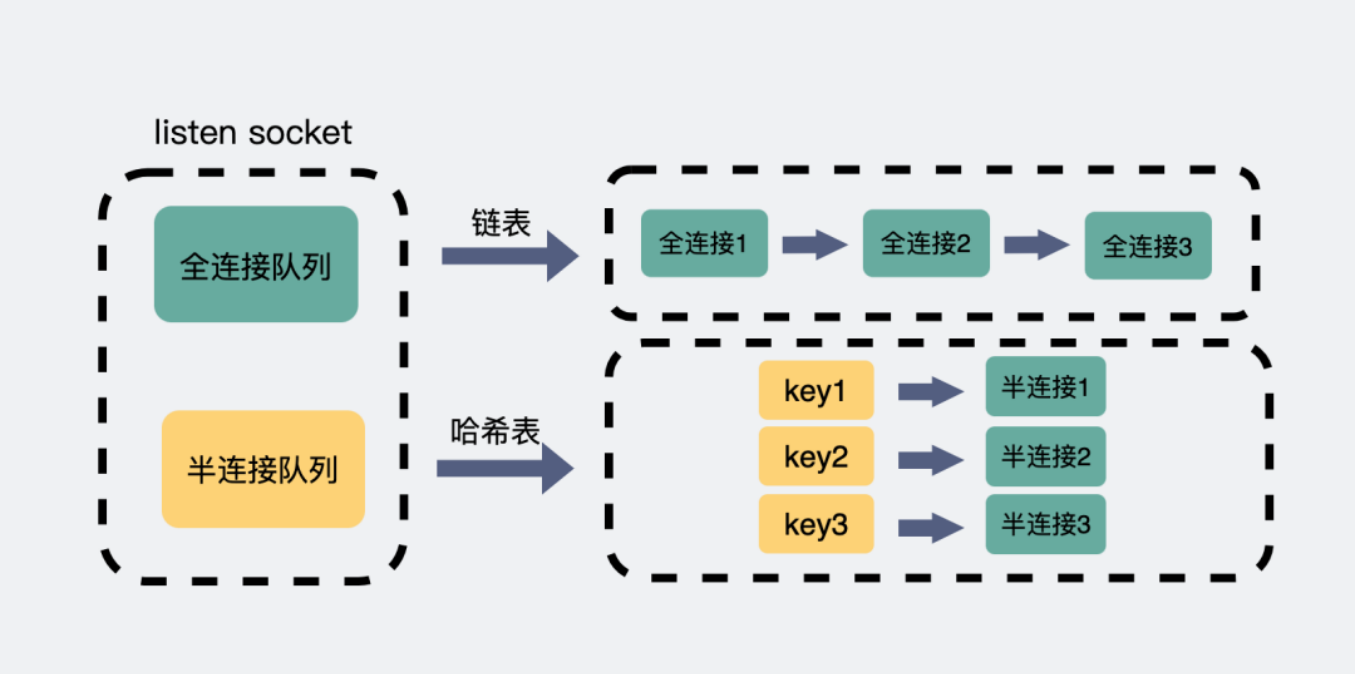
- **半连接队列（SYN队列）**，服务端收到**第一次握手**后，会将 `sock` 加入到这个队列中，队列内的 `sock` 都处于 `SYN_RECV` 状态。

- **全连接队列（ACCEPT队列）**，在服务端收到**第三次握手**后，会将半连接队列的 `sock` 取出，放到全连接队列中。队列里的 `sock` 都处于 `ESTABLISHED` 状态。这里面的连接，就**等着服务端执行accept()后被取出了。**

看到这里，文章开头的问题就有了答案，建立连接的过程中根本不需要 `accept()` 参与，**执行accept()只是为了从全连接队列里取出一条连接。**

我们把话题再重新回到这两个队列上。

虽然都叫**队列**，但其实**全连接队列（icsk_accept_queue）是个链表**，而**半连接队列（syn_table）是个哈希表**。



半连接全连接队列的内部结构

为什么半连接队列要设计成哈希表

先对比下**全连接里队列**，他本质是个链表，因为也是线性结构，说它是个队列也没毛病。它里面放的都是已经建立完成的连接，这些连接正等待被取走。而服务端取走连接的过程中，并不关心具体是哪个连接，只要是个连接就行，所以直接从队列头取就行了。这个过程算法复杂度为 `O(1)` 。

而**半连接队列**却不太一样，因为队列里的都是不完整的连接，嗷嗷等待着第三次握手的到来。那么现在有一个第三次握手来了，则需要从队列里把相应IP端口的连接取出，**如果半连接队列还是个链表，那我们就需要依次遍历，才能拿到我们想要的那个连接，算法复杂度就是O(n)。**

而如果将半连接队列设计成哈希表，那么查找半连接的算法复杂度就回到 `O(1)` 了。

因此出于效率考虑，全连接队列被设计成链表，而半连接队列被设计为哈希表。

怎么观察两个队列的大小

查看全连接队列

```
# ss -lnt
State      Recv-Q Send-Q     Local Address:Port           Peer Address:Port
LISTEN     0      128        127.0.0.1:46269              *:*
```

通过 `ss -lnt` 命令，可以看到全连接队列的大小，其中 `Send-Q` 是指全连接队列的最大值，可以看到我这上面的最大值是 `128` ； `Recv-Q` 是指当前的全连接队列的使用值，我这边用了 `0` 个，也就是全连接队列里为空，连接都被取出来了。

当上面 `Send-Q` 和 `Recv-Q` 数值很接近的时候，那么全连接队列可能已经满了。可以通过下面的命令查看是否发生过队列**溢出**。

```
# netstat -s | grep overflowed
    4343 times the listen queue of a socket overflowed
```

上面说明发生过 `4343次` 全连接队列溢出的情况。这个查看到的是**历史发生过的次数**。

如果配合使用 `watch -d` 命令，可以自动每 `2s` 间隔执行相同命令，还能高亮显示变化的数字部分，如果溢出的数字不断变多，说明**正在发生**溢出的行为。

```
# watch -d 'netstat -s | grep overflowed'
Every 2.0s: netstat -s | grep overflowed                        Fri Sep 1

    4343 times the listen queue of a socket overflowed
```

查看半连接队列

半连接队列没有命令可以直接查看到，但因为半连接队列里，放的都是 `SYN_RECV` 状态的连接，那可以通过统计处于这个状态的连接的数量，间接获得半连接队列的长度。

```
# netstat -nt | grep -i '127.0.0.1:8080' | grep -i 'SYN_RECV' | wc -l
0
```

注意半连接队列和全连接队列都是挂在某个 `Listen socket` 上的，我这里用的是 `127.0.0.1:8080` ，大家可以替换成自己想要查看的**IP端口**。

可以看到我的机器上的半连接队列长度为 `0` ，这个很正常，**正经连接谁会没事老待在半连接队列里。**

当队列里的半连接不断增多，最终也是会发生溢出，可以通过下面的命令查看。

```
# netstat -s | grep -i "SYNs to LISTEN sockets dropped"
    26395 SYNs to LISTEN sockets dropped
```

可以看到，我的机器上一共发生了 `26395` 次半连接队列溢出。同样建议配合 `watch -d` 命令使用。

```
# watch -d 'netstat -s | grep -i "SYNs to LISTEN sockets dropped"'
Every 2.0s: netstat -s | grep -i "SYNs to LISTEN sockets dropped"       Fri Sep 1

    26395 SYNs to LISTEN sockets dropped
```

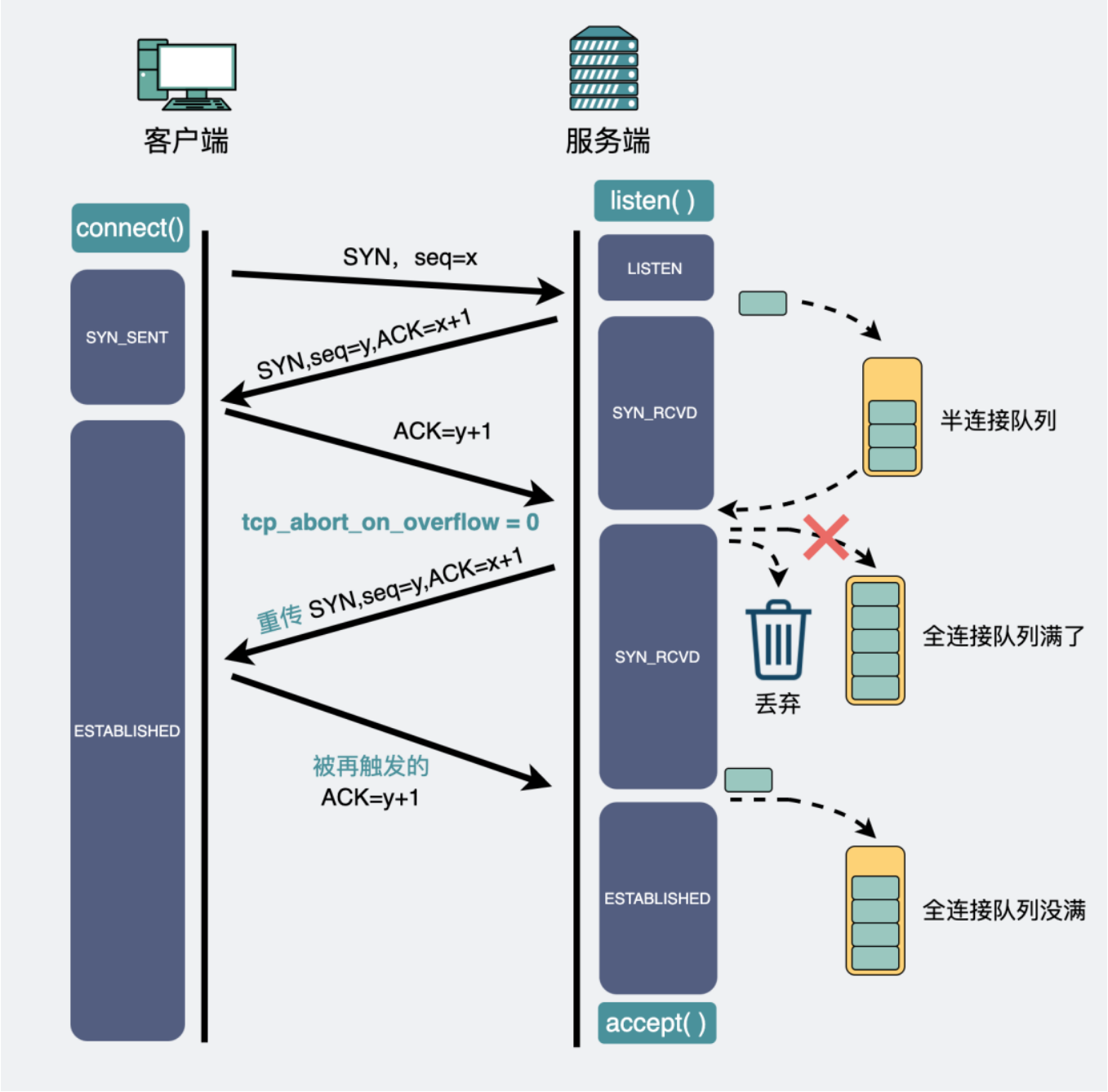
全连接队列满了会怎么样？

如果队列满了，服务端还收到客户端的第三次握手ACK，默认当然会丢弃这个ACK。

但除了丢弃之外，还有一些附带行为，这会受 `tcp_abort_on_overflow` 参数的影响。

```
# cat /proc/sys/net/ipv4/tcp_abort_on_overflow
0
```

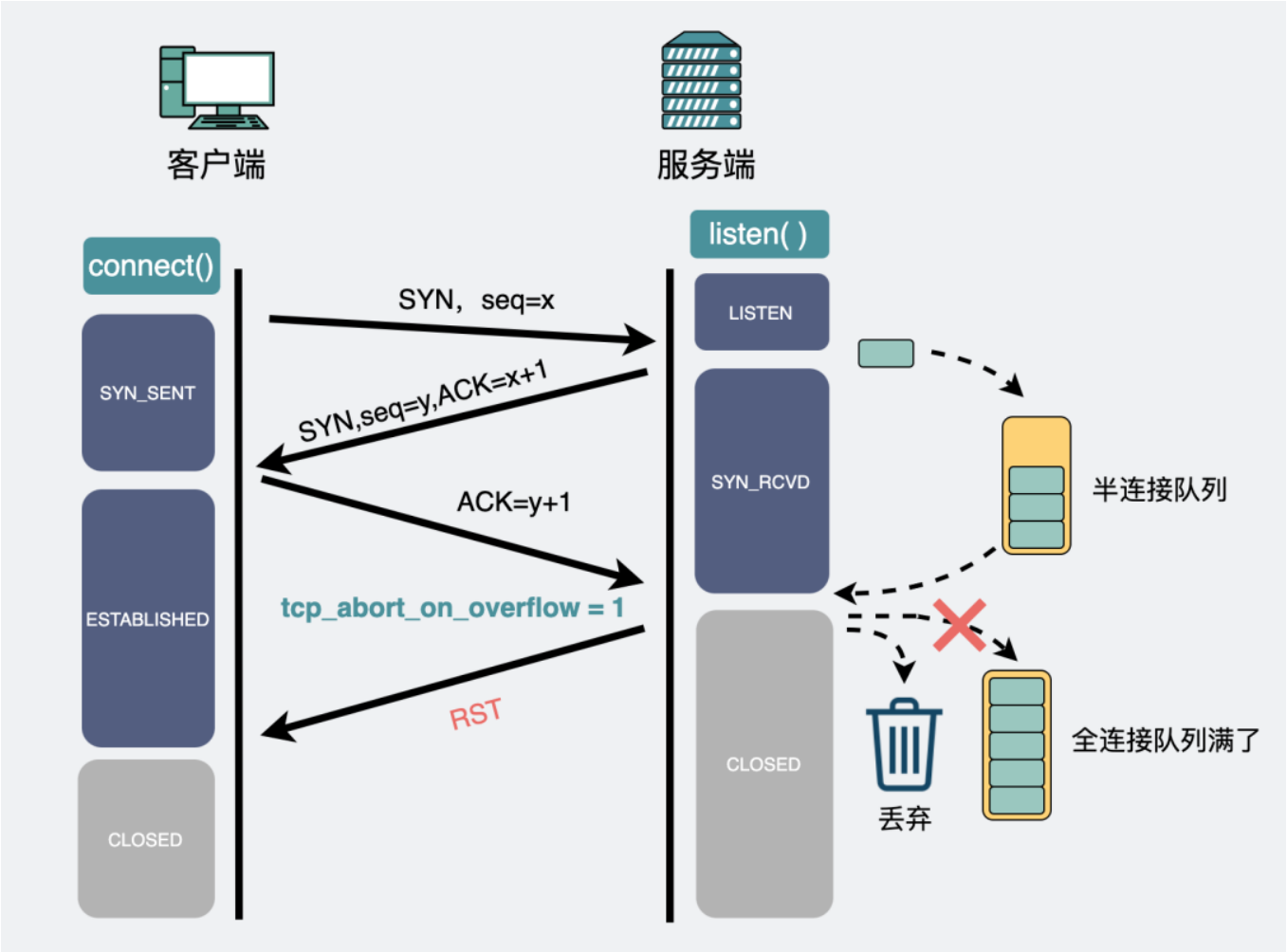
- `tcp_abort_on_overflow` 设置为 0，全连接队列满了之后，会丢弃这个第三次握手ACK包，并且开启定时器，重传第二次握手的SYN+ACK，如果重传超过一定限制次数，还会把对应的**半连接队列里的连接**给删掉。



tcp_abort_on_overflow为0

- `tcp_abort_on_overflow` 设置为 1，全连接队列满了之后，就直接发RST给客户端，效果上看就是连接断了。

这个现象是不是很熟悉，服务端**端口未监听**时，客户端尝试去连接，服务端也会回一个RST。这两个情况长一样，所以客户端这时候收到RST之后，其实无法区分到底是**端口未监听**，还是**全连接队列满了**。



tcp_abort_on_overflow为1

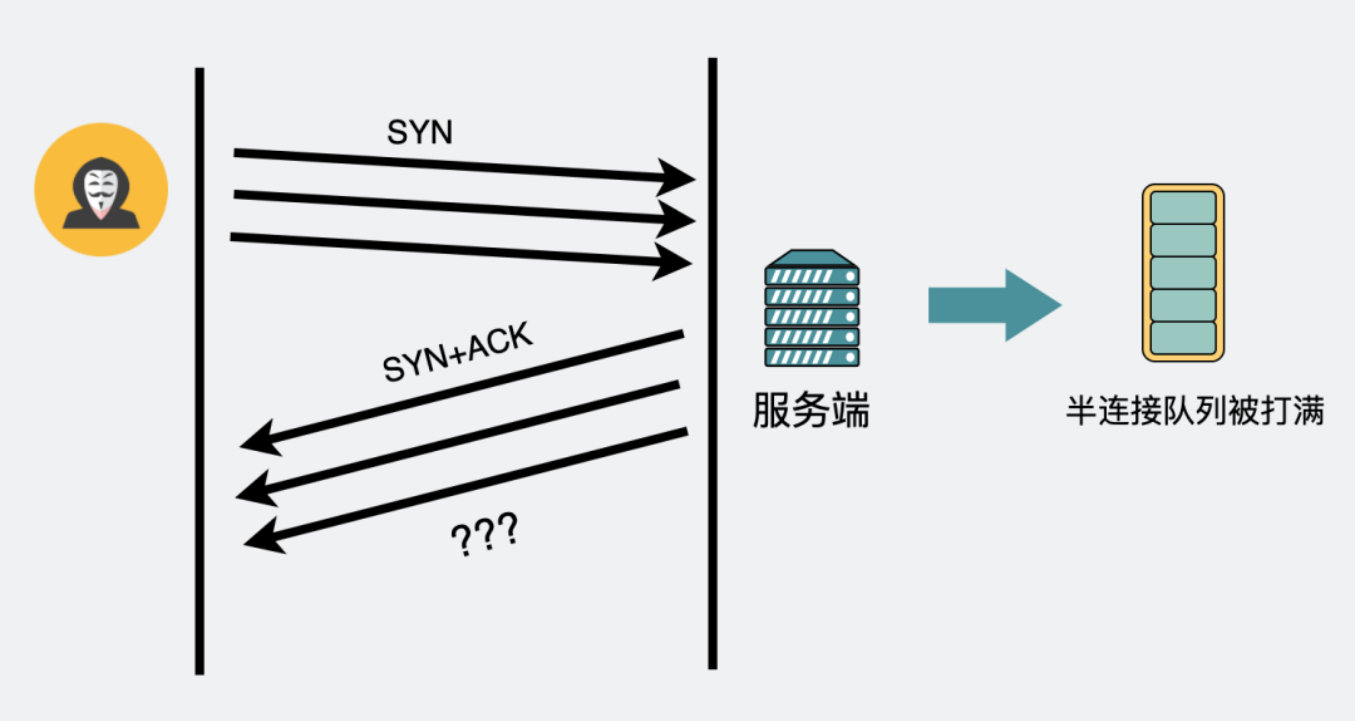
半连接队列要是满了会怎么样

一般是丢弃，但这个行为可以通过 `tcp_syncookies` 参数去控制。但比起这个，更重要的是先了解下半连接队列为什么会被打满。

首先我们需要明白，一般情况下，半连接的"生存"时间其实很短，只有在第一次和第三次握手间，如果半连接都满了，说明服务端疯狂收到第一次握手请求，如果是线上游戏

应用，能有这么多请求进来，那说明你可能要富了。但现实往往比较骨感，你可能遇到了**SYN Flood攻击**。

所谓**SYN Flood攻击**，可以简单理解为，攻击方模拟客户端疯狂发第一次握手请求过来，在服务端憨憨地回复第二次握手过去之后，客户端死活不发第三次握手过来，这样做，可以把服务端半连接队列打满，从而导致正常连接不能正常进来。



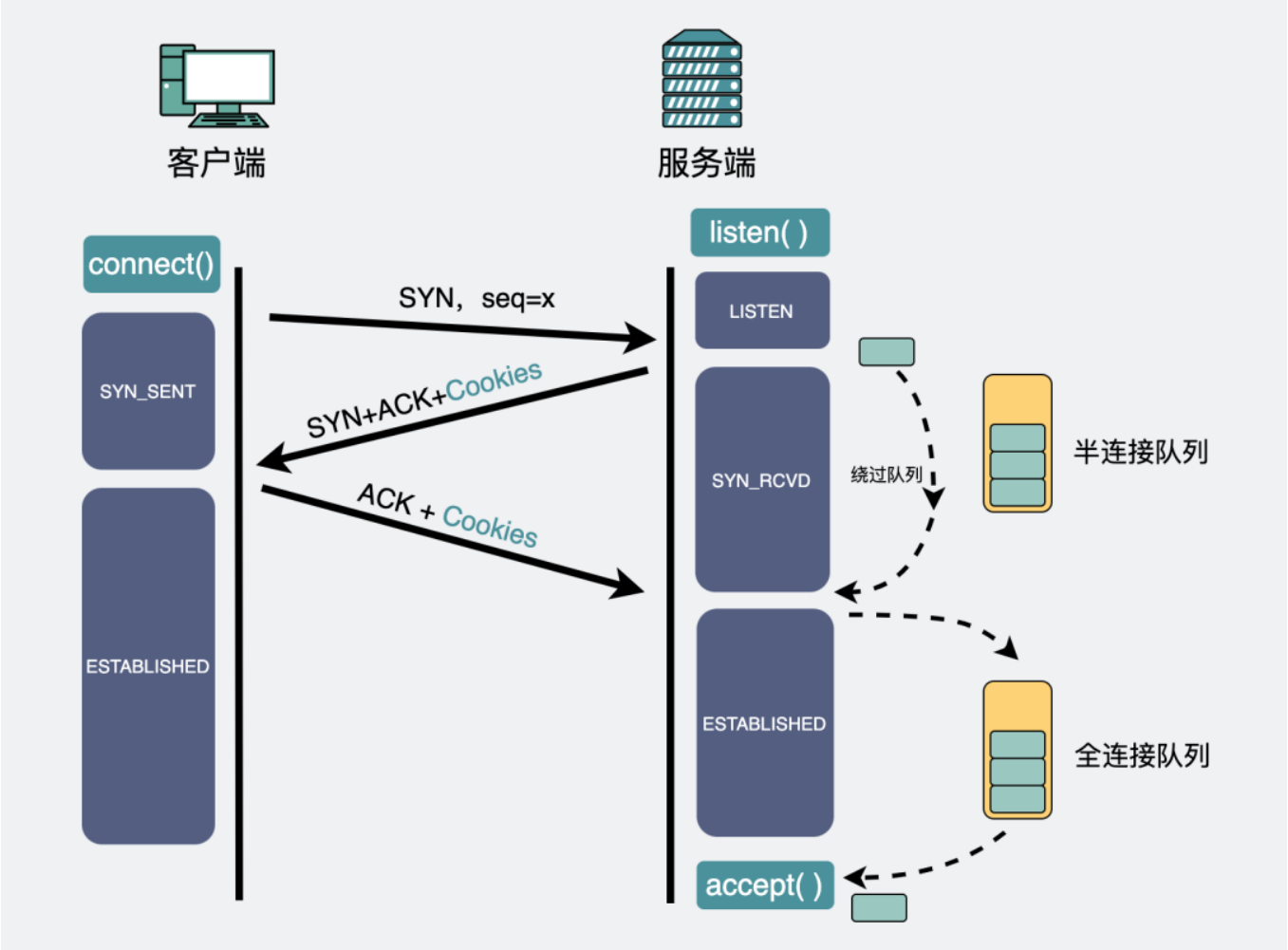
syn攻击

那这种情况怎么处理？有没有一种方法可以**绕过半连接队列**？

有，上面提到的 `tcp_syncookies` 派上用场了。

```
# cat /proc/sys/net/ipv4/tcp_syncookies
1
```

当它被设置为1的时候，客户端发来**第一次握手**SYN时，服务端**不会将其放入半连接队列中**，而是直接生成一个 `cookies` ，这个 `cookies` 会跟着**第二次握手**，发回客户端。客户端在发**第三次握手**的时候带上这个 `cookies` ，服务端验证到它就是当初发出去的那个，就会建立连接并放入到全连接队列中。可以看出整个过程不再需要半连接队列的参与。



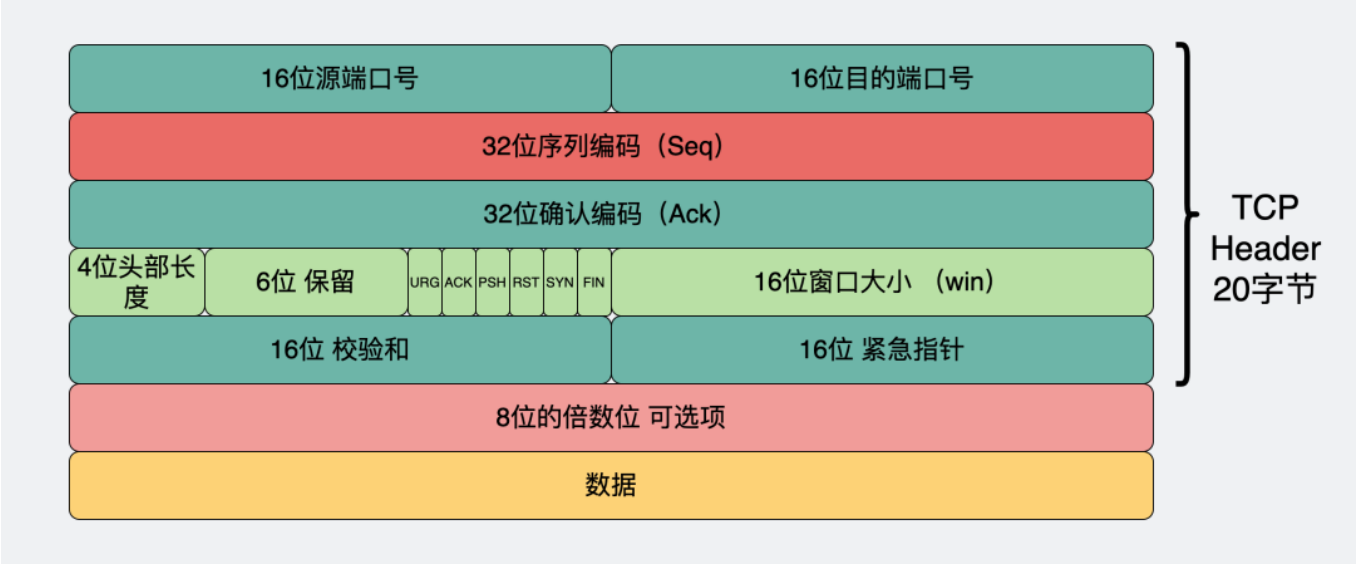
tcp_syncookies=1

会有一个cookies队列吗

生成是 `cookies` ，保存在哪呢？**是不是会有一个队列保存这些cookies?**

我们可以反过来想一下，如果有 `cookies` 队列，那它会跟半连接队列一样，到头来，还是会被**SYN Flood 攻击**打满。

实际上 `cookies` 并不会有一个专门的队列保存，它是通过**通信双方的IP地址端口、时间戳、MSS**等信息进行**实时计算**的，保存在**TCP报头**的 `seq` 里。



tcp报头_seq的位置

当服务端收到客户端发来的第三次握手包时，会通过seq还原出**通信双方的IP地址端口、时间戳、MSS**，验证通过则建立连接。

cookies方案为什么不直接取代半连接队列？

目前看下来 `syn cookies` 方案省下了半连接队列所需要的队列内存，还能解决 **SYN Flood攻击**，那为什么不直接取代半连接队列？

凡事皆有利弊， `cookies` 方案虽然能防 **SYN Flood攻击**，但是也有一些问题。因为服务端并不会保存连接信息，所以如果传输过程中数据包丢了，也不会重发第二次握手的信息。

另外，编码解码 `cookies` ，都是比较**耗CPU**的，利用这一点，如果此时攻击者构造大量的**第三次握手包（ACK包）**，同时带上各种瞎编的 `cookies` 信息，服务端收到 `ACK包` 后**以为是正经cookies**，憨憨地跑去解码（**耗CPU**），最后发现不是正经数据包后才丢弃。

这种通过构造大量 `ACK包` 去消耗服务端资源的攻击，叫**ACK攻击**，受到攻击的服务器可能会因为**CPU资源耗尽**导致没能响应正经请求。

ack攻击

没有listen，为什么还能建立连接

那既然没有 `accept` 方法能建立连接，那是不是没有 `listen` 方法，也能建立连接？是的，之前写的一篇文章提到过客户端是可以自己连自己的形成连接（**TCP自连接**），也可以两个客户端同时向对方发出请求建立连接（**TCP同时打开**），这两个情况都有个共同点，就是**没有服务端参与，也就是没有listen，就能建立连接。**

当时文章最后也留了个疑问，**没有listen，为什么还能建立连接？**

我们知道执行 `listen` 方法时，会创建半连接队列和全连接队列。

三次握手的过程中会在这两个队列中暂存连接信息。

所以形成连接，前提是你得**有个地方存放着**，方便握手的时候能根据IP端口等信息找到socket信息。

那么客户端会有半连接队列吗？

显然没有，因为客户端没有执行 `listen` ，因为半连接队列和全连接队列都是在执行 `listen` 方法时，内核自动创建的。

但内核还有个**全局hash表**，可以用于存放 `sock` 连接的信息。这个全局 `hash` 表其实还细分为 `ehash，bhash和listen_hash` 等，但因为过于细节，大家理解成有一个**全局hash**就够了，

在TCP自连接的情况中，客户端在 `connect` 方法时，最后会将自己的连接信息放入到这个**全局hash表**中，然后将信息发出，消息在经过回环地址重新回到TCP传输层的时候，就会根据IP端口信息，再一次从这个**全局hash**中取出信息。于是握手包一来一回，最后成功建立连接。

TCP同时打开的情况也类似，只不过从一个客户端变成了两个客户端而已。

总结

- **每一个** `socket` 执行 `listen` 时，内核都会自动创建一个半连接队列和全连接队列。
- 第三次握手前，TCP连接会放在半连接队列中，直到第三次握手到来，才会被放到全连接队列中。
- `accept方法` 只是为了从全连接队列中拿出一条连接，本身跟三次握手几乎**毫无关系**。
- 出于效率考虑，虽然都叫队列，但半连接队列其实被设计成了**哈希表**，而全连接队列本质是链表。
- 全连接队列满了，再来第三次握手也会丢弃，此时如果 `tcp_abort_on_overflow=1` ，还会直接发 `RST` 给客户端。
- 半连接队列满了，可能是因为受到了 `SYN Flood` 攻击，可以设置 `tcp_syncookies` ，绕开半连接队列。

- 客户端没有半连接队列和全连接队列，但有一个**全局hash**，可以通过它实现自连接或TCP同时打开。

参考资料

小林图解网络 -- 推荐大家关注《小林coding》

小林coding

专注图解计算机基础，让天下没有难懂的八股文！刷题网站：xiaolincodin...

261篇原创内容

公众号

如果文章对你有帮助，欢迎.....

算了。

兄弟们都是自家人，点不**点赞**，在不**在看**什么的，没关系的，大家看开心了就好。

在看，点赞什么的，我不是特别在意，真的，真的，别不信啊。

不三连也真的没关系的。

兄弟们不要在意啊。

我是**虚伪**的小白，我们下期见！

别说了，一起在知识的海洋里呛水吧

点击下方名片，关注公众号:【小白debug】

小白debug

答应我，关注之后，好好学技术，别只是收藏我的表情包。。