三次握手背后的秘密——全连接队列、半连接队列

搞懂下面这几个问题

- backlog、半连接队列、全连接队列是什么
- linux 内核是如何计算半连接队列、全连接队列的
- 为什么只修改系统的 somaxconn 和 tcp*max*syn_backlog 对最终的队列大小不起作用
- iprouter 库中的 ss 工具的原理是什么
- 如何快速模拟半连接队列溢出，全连接队列溢出

listen 函数的定义

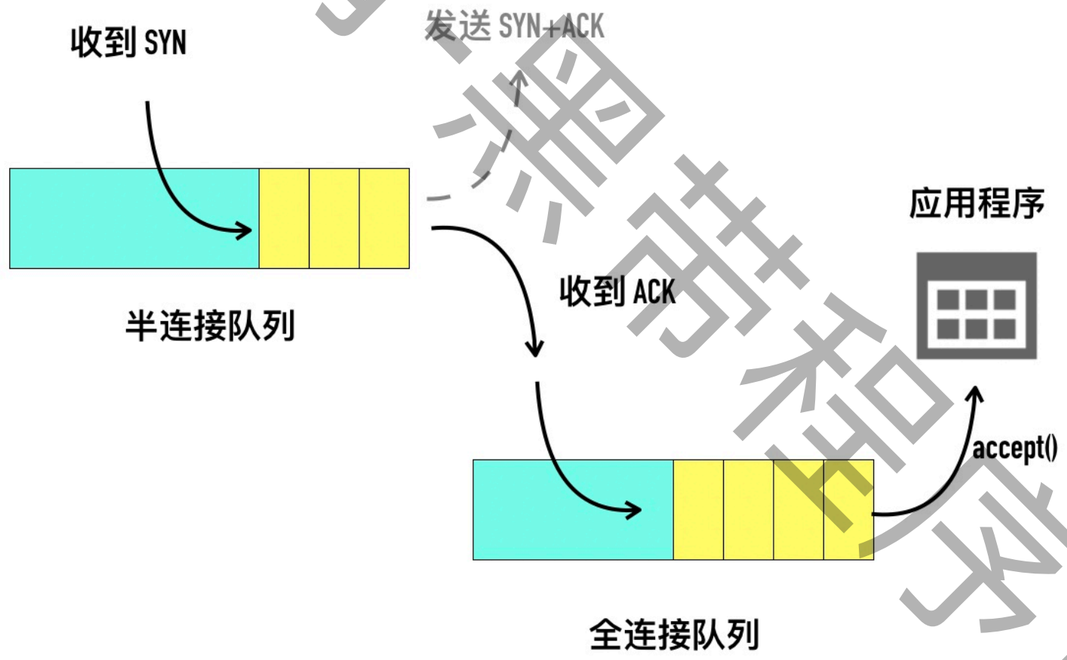
```
int listen(int sockfd, int backlog);
```

何为 backlog

backlog：n.积压的工作

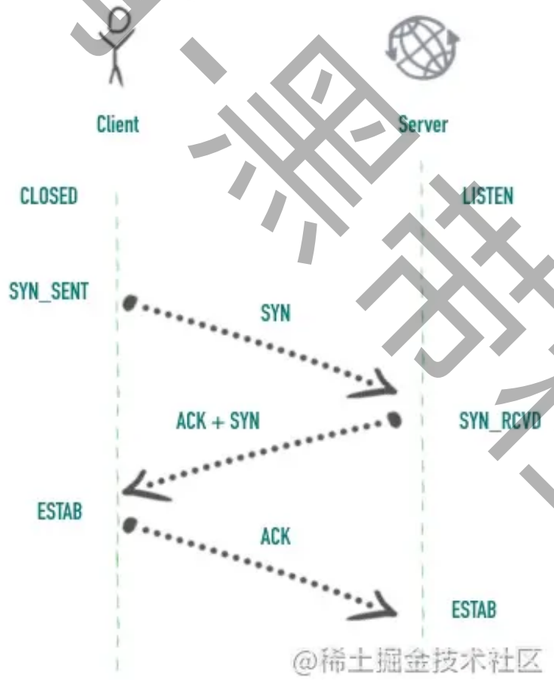
本次分享的主角

- 半连接队列（Incomplete connection queue），又称 SYN 队列
- 全连接队列（Completed connection queue），又称 Accept 队列

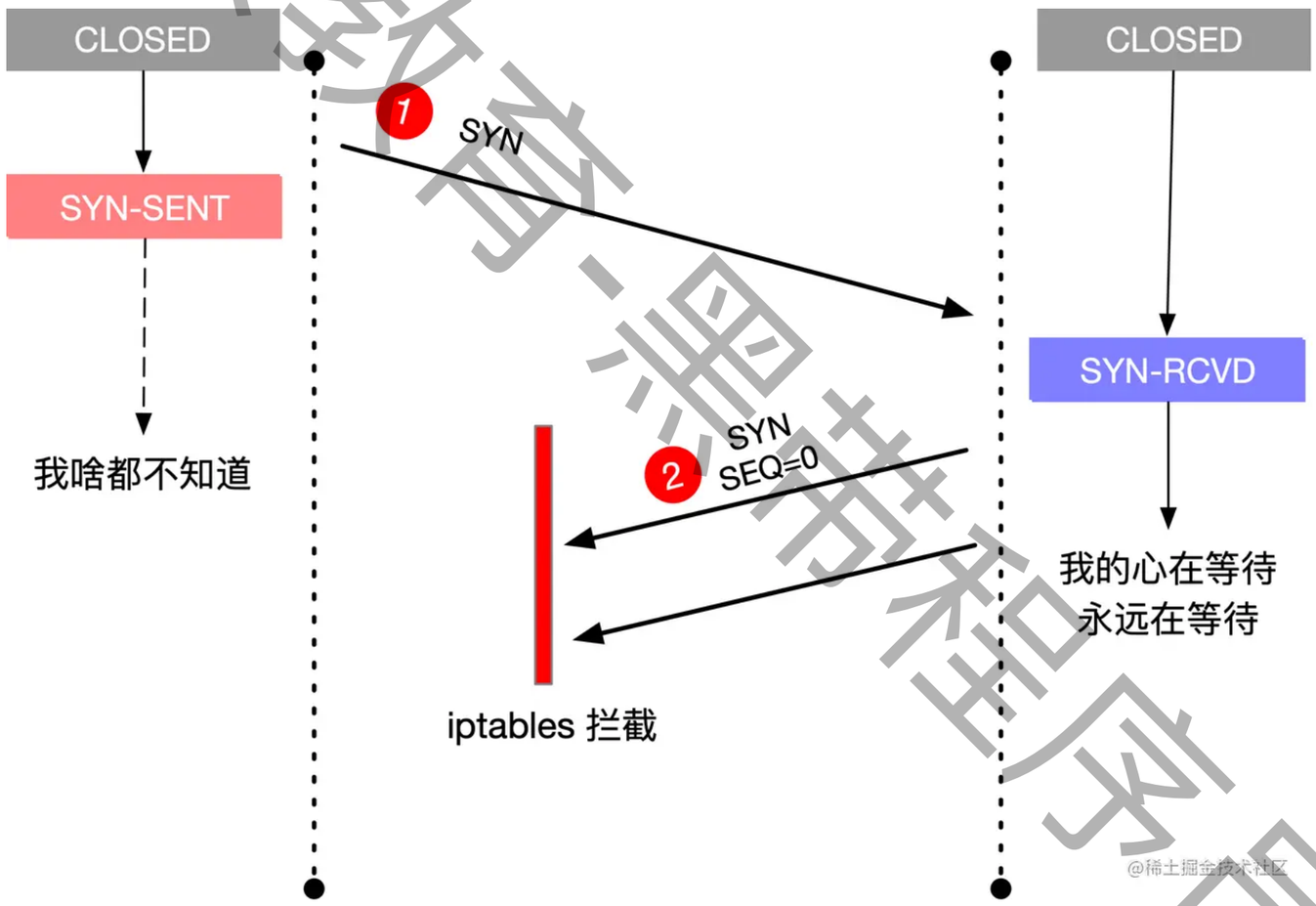


半连接队列（SYN Queue）

当客户端发起 SYN 到服务端，服务端收到以后会回 ACK 和自己的 SYN。这时服务端这边的 TCP 从 listen 状态变为 SYN_RCVD (SYN Received)，此时会将这个连接信息放入「半连接队列」，半连接队列也被称为 SYN Queue，存储的是 "inbound SYN packets"。



模拟半连接队列满的情况



防火墙 drop 三次握手第二步的 SYN

```
sudo iptables --append INPUT  --match tcp --protocol tcp --src 10.211.55.15 --sport 9090 --tcp-flags SYN SYN --jump DROP
```

执行这个 go 程序，在服务端使用 netstat 查看当前 9090 端口的连接状态

```
package main

import (
    "fmt"
    "net"
    "time"
)

func main() {
    for i := 0; i < 2000; i++ {
        go connect()
    }
    time.Sleep(time.Minute * 10)
}
func connect() {
    _, err := net.Dial("tcp4", "10.211.55.3:9090")
    if err != nil {
        fmt.Println(err)
    }
}
```

somaxconn、max_syn_backlog、backlog 三者之间不同组合的最终半连接队列大小值

somaxconn	max_syn_backlog	listen backlog	半连接队列大小
128	128	5	16
128	128	10	16
128	128	50	64
128	128	128	256
128	128	1000	256
128	128	5000	256
1024	128	128	256
1024	1024	128	256
4096	4096	128	256
4096	4096	4096	8192

半连接队列源码剖析

如果用户传入的 backlog 值大于系统变量 net.core.somaxconn 的值，用户设置的 backlog 不会生效，使用系统变量值，默认为 128。

```
SYSCALL_DEFINE2(listen, int, fd, int, backlog)
{
    // sysctl_somaxconn 是系统变量 net.core.somaxconn 的值
    int somaxconn = sysctl_somaxconn;
    if ((unsigned int)backlog > somaxconn)
        backlog = somaxconn;
    sock->ops->listen(sock, backlog);
}
```

接下来这个 backlog 值会被依次传递给 inet_listen()->inet_csk_listen_start()->reqsk_queue_alloc() 方法，在 reqsk_queue_alloc 方法中进行了最终的计算

```
int reqsk_queue_alloc(struct request_sock_queue *queue,
                      unsigned int nr_table_entries)
{
    // min(50, 128) = 50
    nr_table_entries = min_t(u32, nr_table_entries, sysctl_max_syn_backlog);
    // max(50, 8) = 50
    nr_table_entries = max_t(u32, nr_table_entries, 8);
    // roundup_pow_of_two(51) = 64
    nr_table_entries = roundup_pow_of_two(nr_table_entries + 1);

    // max_qlen_log 最小值为 2^3 = 8
    for (lopt->max_qlen_log = 3;
         (1 << lopt->max_qlen_log) < nr_table_entries;
         lopt->max_qlen_log++);
    // 经过 for 循环 max_qlen_log = 2^6 = 64
}
```

一些结论

- 在系统参数不修改的情形，盲目调大 listen 的 backlog 对最终半连接队列的大小不会有影响。
- 在 listen 的 backlog 不变的情况下，盲目调大 somaxconn 和 max_syn_backlog 对最终半连接队列的大小不会有影响

全连接队列（Accept Queue）

「全连接队列」包含了服务端所有完成了三次握手，但是还未被应用取走的连接队列。此时的 socket 处于 ESTABLISHED 状态。每次应用调用 accept() 函数会移除队列头的连接。如果队列为空，accept() 通常会阻塞。全连接队列也被称为 Accept 队列。

生产者、消费者模型

- 内核是一个负责三次握手的生产者，握手完的连接会放入一个队列。
- 我们的应用程序是一个消费者，accept 取走队列中的连接进行下一步的处理。

这种生产者消费者的模式，在生产过快、消费过慢的情况下就会出现队列积压。

队列大小限制

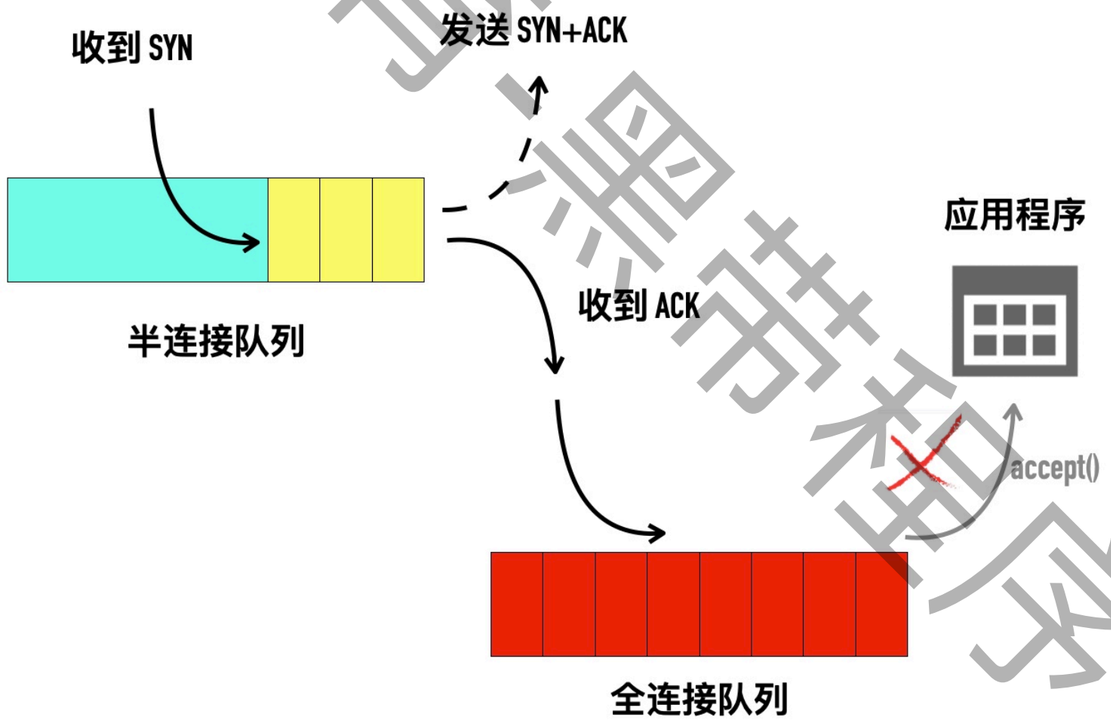
两个队列都不是无限大小的，listen 函数的第二个参数 backlog 用来设置全连接队列大小，但不一定就会选用这一个 backlog 值，还受限于 somaxconn。

```
int listen(int sockfd, int backlog)
```

如果全连接队列满，server 会舍弃掉 client 发过来的 ack（server 会认为此时连接还未完全建立）

模拟一下全连接队列满的情况

因为只有 accept 才会移除全连接的队列，所以如果我们只 listen，不调用 accept，那么很快全连接就可以被占满。



实验环节

全连接队列的大小

全连接队列的大小是 listen 传入的 backlog 和 somaxconn 中的较小值。全连接队列大小判断是否满的函数是 /include/net/sock.h 中 的 sk_acceptq_is_full 方法。

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

这里本身没有什么毛病，只是真正全连接队列大小是 backlog + 1。当你指定 backlog 值为 1 时，能容纳的连接个数会是 2。

ss 命令

ss 命令可以查看全连接队列的大小和当前等待 accept 的连接个数，执行 ss -lnt 即可，比如上面的 accept 队列满的例子中，执行 ss 命令的输出结果如下。

```
ss -lnt | grep :9090
State      Recv-Q Send-Q Local Address:Port               Peer Address:Port
LISTEN     51     50           *:9090                     *:*
```

对于 LISTEN 状态的套接字，Recv-Q 表示 accept 队列排队的连接个数，Send-Q 表示全连接队列（也就是 accept 队列）的总大小。

源码分析

```
static void tcp_diag_get_info(struct sock *sk, struct inet_diag_msg *r,
                              void *_info)
{
    struct tcp_info *info = _info;

   // LISTEN 状态的 socket
    if (inet_sk_state_load(sk) == TCP_LISTEN) {
       // 对应 Recv-Q
        r->idiag_rqueue = READ_ONCE(sk->sk_ack_backlog);
        // 对应 Send-Q
        r->idiag_wqueue = READ_ONCE(sk->sk_max_ack_backlog);    }
    // 非 LISTEN 状态的 socket
    else if (sk->sk_type == SOCK_STREAM) {
        const struct tcp_sock *tp = tcp_sk(sk);
        // receive queue 大小
        r->idiag_rqueue = max_t(int, READ_ONCE(tp->rcv_nxt) -
                                READ_ONCE(tp->copied_seq), 0);
        // send queue 大小
        r->idiag_wqueue = READ_ONCE(tp->write_seq) - tp->snd_una;
    }
}
```

多大的 backlog 是合适的

- 你如果的接口处理连接的速度要求非常高，或者在做压力测试，很有必要调高这个值
- 如果业务接口本身性能不好，accept 取走已建连的速度较慢，那么把 backlog 调的再大也没有用，只会增加连接失败的可能性。

典型的 backlog 值：

- Nginx 和 Redis 默认的 backlog 值等于 511
- Linux 默认的 backlog 为 128
- Java 默认的 backlog 等于 50

tcp_abort_on_overflow 参数

默认情况下，全连接队列满以后，服务端会忽略客户端的 ACK，随后会重传SYN+ACK，也可以修改这种行为，这个值由/proc/sys/net/ipv4/tcp_abort_on_overflow决定。

tcp_abort_on_overflow 参数值的含义

- tcp_abort_on_overflow 为 0 表示三次握手最后一步全连接队列满以后 server 会丢掉 client 发过来的 ACK，服务端随后会进行重传 SYN+ACK。
- tcp_abort_on_overflow 为 1 表示全连接队列满以后服务端直接发送 RST 给客户端。

但是回给客户端 RST 包会带来另外一个问题，客户端不知道服务端响应的 RST 包到底是因为「该端口没有进程监听」，还是「该端口有进程监听，只是它的队列满了」。

小结

- 半连接队列：服务端收到客户端的 SYN 包，回复 SYN+ACK 但是还没有收到客户端 ACK 情况下，会将连接信息放入半连接队列。半连接队列又被称为 SYN 队列。
- 全连接队列：服务端完成了三次握手，但是还未被 accept 取走的连接队列。全连接队列又被称为 Accept 队列。
- 半连接队列的大小与用户 listen 传入的 `backlog`、`somaxconn`、`max_syn_backlog` 都有关系，准确的计算规则见上面的源码分析
- 全连接队列的大小是用户 listen 传入的 backlog 与 net.core.somaxconn 的较小值