zhuanlan.zhihu.com

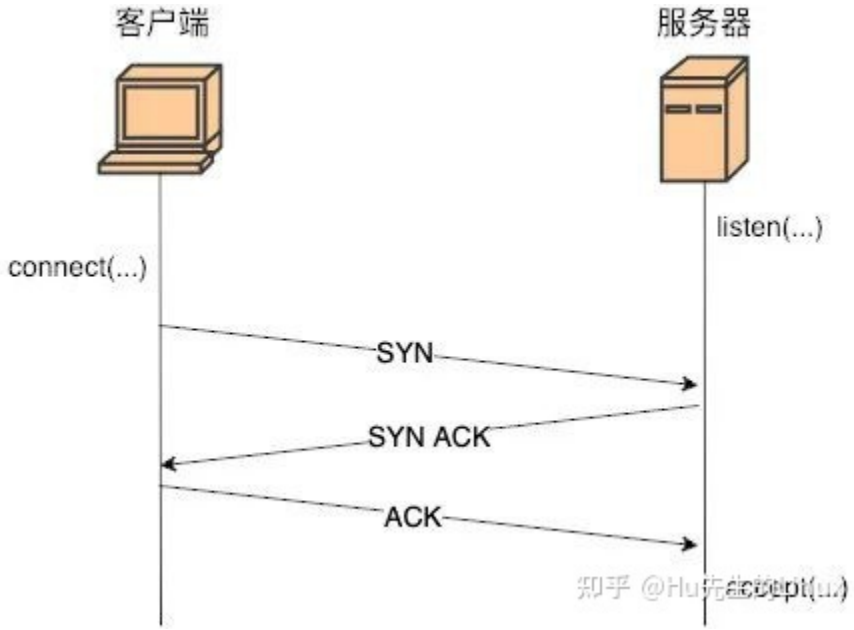
深入理解 Linux 的 TCP 三次握手

45-56 minutes

在后端相关岗位的入职面试中，三次握手的出场频率非常的高，甚至说它是必考题也不为过。一般的答案都是说客户端如何发起 SYN 握手进入 SYN_SENT 状态，服务器响应 SYN 并回复 SYNACK，然后进入 SYN_RECV 等诸如此类。但今天我想给出一份不一样的答案。

其实三次握手在内核的实现中，并不只是简单的状态的流转，还包括端口选择，半连接队列、syncookie、全连接队列、重传计时器等关键操作。如果能深刻理解这些，你对线上把握和理解将更进一步。如果有面试官问起你三次握手，相信这份答案一定能帮你在面试官面前赢得非常多的加分。

在基于 TCP 的服务开发中，三次握手的主要流程图如下：



服务器中的核心代码是创建 socket，绑定端口，listen 监听，最后 accept 接收客户端的请求。

```
//服务端核心代码
int main(int argc, char const *argv[])
{
 int fd = socket(AF_INET, SOCK_STREAM, 0);
 bind(fd, ...);
 listen(fd, 128);
 accept(fd, ...);
 ...
}
```

客户端的相关代码是创建 socket，然后调用 connect 连接 server。

```
//客户端核心代码
int main(){
 fd = socket(AF_INET,SOCK_STREAM, 0);
 connect(fd, ...);
 ...
}
```

看起来简单的几个系统调用，实际上却包含了非常复杂的内核底层操作。根据内核工作原理，我深度展开一下三次握手过程中的内部操作。

友情提示：本文中内核源码会比较多。如果你能理解的了更好，如果觉得理解起来有困难，那直接重点看本文中的描述性的文字，尤其是加粗部分的即可。另外文章最后有一张总结图归纳和整理了全文内容。

一、服务器的 listen

我们都知道，服务器在开始提供服务之前都需要先 listen 一下。但

listen 内部究竟干了啥，我们平时很少去琢磨。

今天就让我们详细来看看，直接上一段 listen 时执行到的内核代码。

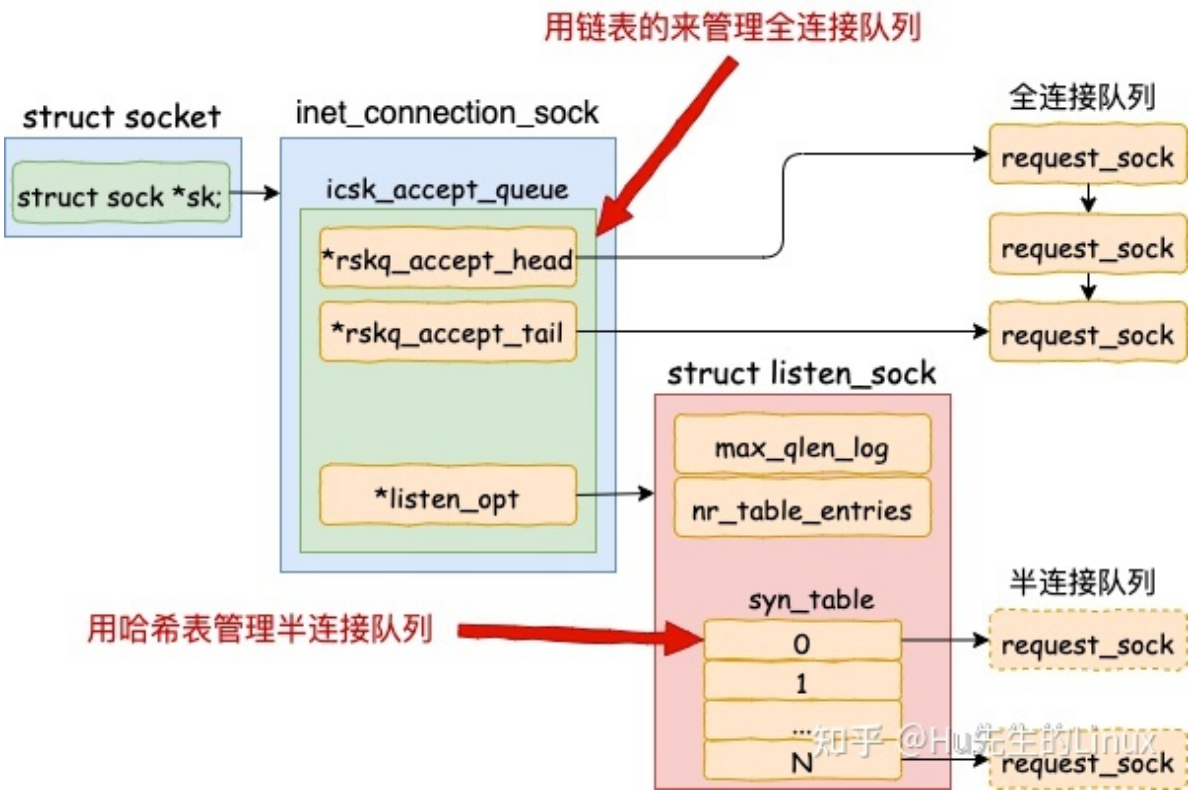
```
//file: net/core/request_sock.c
int reqsk_queue_alloc(struct request_sock_queue *queue,
     unsigned int nr_table_entries)
{
 size_t lopt_size = sizeof(struct listen_sock);
 struct listen_sock *lopt;

 //计算半连接队列的长度
 nr_table_entries = min_t(u32, nr_table_entries,
sysctl_max_syn_backlog);
 nr_table_entries = ......

 //为半连接队列申请内存
 lopt_size += nr_table_entries * sizeof(struct request_sock
*);
 if (lopt_size > PAGE_SIZE)
  lopt = vzalloc(lopt_size);
 else
  lopt = kzalloc(lopt_size, GFP_KERNEL);

 //全连接队列头初始化
 queue->rskq_accept_head = NULL;

 //半连接队列设置
 lopt->nr_table_entries = nr_table_entries;
 queue->listen_opt = lopt;
```

```
    ......
}
```

在这段代码里，内核计算了半连接队列的长度。然后据此算出半连接队列所需要的实际内存大小，开始申请用于管理半连接队列对象的内存（半连接队列需要快速查找，所以内核是用哈希表来管理半连接队列的，具体在 listen_sock 下的 syn_table 下）。最后将半连接队列挂到了接收队列 queue 上。

另外 queue->rskq_accept_head 代表的是全连接队列，它是一个链表的形式。在 listen 这里因为还没有连接，所以将全连接队列头 queue->rskq_accept_head 设置成 NULL。

当全连接队列和半连接队列中有元素的时候，他们在内核中的结构图大致如下。



在服务器 listen 的时候，主要是进行了全/半连接队列的长度限制计算，以及相关的内存申请和初始化。全/连接队列初始化了以后才可以相应来自客户端的握手请求。

二、客户端 connect

客户端通过调用 connect 来发起连接。在 connect 系统调用中会进

入到内核源码的 tcp_v4_connect。

```
//file: net/ipv4/tcp_ipv4.c
int tcp_v4_connect(struct sock *sk, struct sockaddr *uaddr,
int addr_len)
{
 //设置 socket 状态为 TCP_SYN_SENT
 tcp_set_state(sk, TCP_SYN_SENT);

 //动态选择一个端口
 err = inet_hash_connect(&tcp_death_row, sk);

 //函数用来根据 sk 中的信息，构建一个完成的 syn 报文，并将
它发送出去。
 err = tcp_connect(sk);
}
```

在这里将完成把 socket 状态设置为 TCP_SYN_SENT。再通过 inet_hash_connect 来动态地选择一个可用的端口后（端口选择详细过程参考前文**《TCP 连接中客户端的端口号是如何确定的？》**），进入到 tcp_connect 中。

```
//file:net/ipv4/tcp_output.c
int tcp_connect(struct sock *sk)
{
 tcp_connect_init(sk);

 //申请 skb 并构造为一个 SYN 包
 ......

 //添加到发送队列 sk_write_queue 上
```

```
    tcp_connect_queue_skb(sk, buff);

    //实际发出 syn
    err = tp->fastopen_req ? tcp_send_syn_data(sk, buff) :
       tcp_transmit_skb(sk, buff, 1, sk->sk_allocation);

    //启动重传定时器
    inet_csk_reset_xmit_timer(sk, ICSK_TIME_RETRANS,
         inet_csk(sk)->icsk_rto, TCP_RTO_MAX);
}
```

在 tcp_connect 申请和构造 SYN 包，然后将其发出。同时还启动了一个重传定时器，该定时器的作用是等到一定时间后收不到服务器的反馈的时候来开启重传。在 3.10 版本中首次超时时间是 1 s，一些老版本中是 3 s。

总结一下，客户端在 connect 的时候，把本地 socket 状态设置成了 TCP_SYN_SENT，选了一个可用的端口，接着发出 SYN 握手请求并启动重传定时器。

三、服务器响应 SYN

在服务器端，所有的 TCP 包（包括客户端发来的 SYN 握手请求）都经过网卡、软中断，进入到 tcp_v4_rcv。在该函数中根据网络包（skb）TCP 头信息中的目的 IP 信息查到当前在 listen 的 socket。然后继续进入 tcp_v4_do_rcv 处理握手过程。

```
//file: net/ipv4/tcp_ipv4.c
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
 ...
 //服务器收到第一步握手 SYN 或者第三步 ACK 都会走到这里
```

```
 if (sk->sk_state == TCP_LISTEN) {
  struct sock *nsk = tcp_v4_hnd_req(sk, skb);
 }

 if (tcp_rcv_state_process(sk, skb, tcp_hdr(skb),
skb->len)) {
  rsk = sk;
  goto reset;
 }
}
```

在 tcp_v4_do_rcv 中判断当前 socket 是 listen 状态后，首先会到 tcp_v4_hnd_req 去查看半连接队列。服务器第一次响应 SYN 的时候，半连接队列里必然是空空如也，所以相当于什么也没干就返回了。

```
//file:net/ipv4/tcp_ipv4.c
static struct sock *tcp_v4_hnd_req(struct sock *sk, struct
sk_buff *skb)
{
 // 查找 listen socket 的半连接队列
 struct request_sock *req = inet_csk_search_req(sk, &prev,
th->source,
          iph->saddr, iph->daddr);
 ...
 return sk;
}
```

在 tcp_rcv_state_process 里根据不同的 socket 状态进行不同的处理。

```
//file:net/ipv4/tcp_input.c
```

```
int tcp_rcv_state_process(struct sock *sk, struct sk_buff
*skb,
     const struct tcphdr *th, unsigned int len)
{
 switch (sk->sk_state) {
  //第一次握手
  case TCP_LISTEN:
   if (th->syn) { //判断是 SYN 握手包
    ...
    if (icsk->icsk_af_ops->conn_request(sk, skb) < 0)
     return 1;
 ......
}
```

其中 conn_request 是一个函数指针，指向 tcp_v4_conn_request。**服务器响应 SYN 的主要处理逻辑都在这个 tcp_v4_conn_request 里。**

```
//file: net/ipv4/tcp_ipv4.c
int tcp_v4_conn_request(struct sock *sk, struct sk_buff
*skb)
{
 //看看半连接队列是否满了
 if (inet_csk_reqsk_queue_is_full(sk) && !isn) {
  want_cookie = tcp_syn_flood_action(sk, skb, "TCP");
  if (!want_cookie)
   goto drop;
 }

 //在全连接队列满的情况下，如果有 young_ack，那么直接丢
 if (sk_acceptq_is_full(sk) &&
```

```
inet_csk_reqsk_queue_young(sk) > 1) {
  NET_INC_STATS_BH(sock_net(sk),
LINUX_MIB_LISTENOVERFLOWS);
  goto drop;
 }
 ...
 //分配 request_sock 内核对象
 req = inet_reqsk_alloc(&tcp_request_sock_ops);

 //构造 syn+ack 包
 skb_synack = tcp_make_synack(sk, dst, req,
  fastopen_cookie_present(&valid_foc) ? &valid_foc : NULL);

 if (likely(!do_fastopen)) {
  //发送 syn + ack 响应
  err = ip_build_and_send_pkt(skb_synack, sk,
ireq->loc_addr,
     ireq->rmt_addr, ireq->opt);

  //添加到半连接队列，并开启计时器
  inet_csk_reqsk_queue_hash_add(sk, req, TCP_TIMEOUT_INIT);
 }else ...
}
```

在这里首先判断半连接队列是否满了，如果满了的话进入 tcp_syn_flood_action 去判断是否开启了 tcp_syncookies 内核参数。**如果队列满，且未开启 tcp_syncookies，那么该握手包将直接被丢弃！！**

接着还要判断全连接队列是否满。因为全连接队列满也会导致握手异常的，那干脆就在第一次握手的时候也判断了。**如果全连接队列**

满了，且有 young_ack 的话，那么同样也是直接丢弃。

> young_ack 是半连接队列里保持着的一个计数器。记录的是刚有 SYN 到达，没有被 SYN_ACK 重传定时器重传过 SYN_ACK，同时也没有完成过三次握手的 sock 数量

接下来是构造 synack 包，然后通过 ip_build_and_send_pkt 把它发送出去。

最后把当前握手信息添加到半连接队列，并开启计时器。计时器的作用是如果某个时间之内还收不到客户端的第三次握手的话，服务器会重传 synack 包。

总结一下，服务器响应 ack 是主要工作是判断下接收队列是否满了，满的话可能会丢弃该请求，否则发出 synack。申请 request_sock 添加到半连接队列中，同时启动定时器。

四、客户端响应 SYNACK

客户端收到服务器端发来的 synack 包的时候，也会进入到 tcp_rcv_state_process 函数中来。不过由于自身 socket 的状态是 TCP_SYN_SENT，所以会进入到另一个不同的分支中去。

```
//file:net/ipv4/tcp_input.c
//除了 ESTABLISHED 和 TIME_WAIT，其他状态下的 TCP 处理都走这里
int tcp_rcv_state_process(struct sock *sk, struct sk_buff *skb,
     const struct tcphdr *th, unsigned int len)
{
 switch (sk->sk_state) {
  //服务器收到第一个ACK包
  case TCP_LISTEN:
   ...
```

```
  //客户端第二次握手处理
  case TCP_SYN_SENT:
   //处理 synack 包
   queued = tcp_rcv_synsent_state_process(sk, skb, th,
len);
   ...
   return 0;
}
```

tcp_rcv_synsent_state_process 是客户端响应 synack 的主要逻辑。

```
//file:net/ipv4/tcp_input.c
static int tcp_rcv_synsent_state_process(struct sock *sk,
struct sk_buff *skb,
      const struct tcphdr *th, unsigned int len)
{
 ...

 tcp_ack(sk, skb, FLAG_SLOWPATH);

 //连接建立完成
 tcp_finish_connect(sk, skb);

 if (sk->sk_write_pending ||
   icsk->icsk_accept_queue.rskq_defer_accept ||
   icsk->icsk_ack.pingpong)
  //延迟确认...
 else {
  tcp_send_ack(sk);
 }
```

```
}
```

tcp_ack()->tcp_clean_rtx_queue()

```
//file: net/ipv4/tcp_input.c
static int tcp_clean_rtx_queue(struct sock *sk, int
prior_fackets,
       u32 prior_snd_una)
{
 //删除发送队列
 ...

 //删除定时器
 tcp_rearm_rto(sk);
}

//file: net/ipv4/tcp_input.c
void tcp_finish_connect(struct sock *sk, struct sk_buff
*skb)
{
 //修改 socket 状态
 tcp_set_state(sk, TCP_ESTABLISHED);

 //初始化拥塞控制
 tcp_init_congestion_control(sk);
 ...

 //保活计时器打开
 if (sock_flag(sk, SOCK_KEEPOPEN))
  inet_csk_reset_keepalive_timer(sk,
keepalive_time_when(tp));
```

```
}
```

客户端修改自己的 socket 状态为 ESTABLISHED，接着打开 TCP 的保活计时器。

```
//file:net/ipv4/tcp_output.c
void tcp_send_ack(struct sock *sk)
{
 //申请和构造 ack 包
 buff = alloc_skb(MAX_TCP_HEADER, sk_gfp_atomic(sk,
GFP_ATOMIC));
 ...

 //发送出去
 tcp_transmit_skb(sk, buff, 0, sk_gfp_atomic(sk,
GFP_ATOMIC));
}
```

在 tcp_send_ack 中构造 ack 包，并把它发送了出去。

客户端响应来自服务器端的 synack 时清除了 connect 时设置的重传定时器，把当前 socket 状态设置为 ESTABLISHED，开启保活计时器后发出第三次握手的 ack 确认。

五、服务器响应 ACK

服务器响应第三次握手的 ack 时同样会进入到 tcp_v4_do_rcv

```
//file: net/ipv4/tcp_ipv4.c
int tcp_v4_do_rcv(struct sock *sk, struct sk_buff *skb)
{
 ...
 if (sk->sk_state == TCP_LISTEN) {
```

```
  struct sock *nsk = tcp_v4_hnd_req(sk, skb);

  if (nsk != sk) {
   if (tcp_child_process(sk, nsk, skb)) {
    ...
   }
   return 0;
  }
 }
 ...
}
```

不过由于这已经是第三次握手了，半连接队列里会存在上次第一次握手时留下的半连接信息。所以 tcp_v4_hnd_req 的执行逻辑会不太一样。

```
//file:net/ipv4/tcp_ipv4.c
static struct sock *tcp_v4_hnd_req(struct sock *sk, struct
sk_buff *skb)
{
 ...
 struct request_sock *req = inet_csk_search_req(sk, &prev,
th->source,
          iph->saddr, iph->daddr);
 if (req)
  return tcp_check_req(sk, skb, req, prev, false);
 ...
}
```

inet_csk_search_req 负责在半连接队列里进行查找，找到以后返回一个半连接 request_sock 对象。然后进入到 tcp_check_req 中。

```
//file: net/ipv4/tcp_minisocks.c
struct sock *tcp_check_req(struct sock *sk, struct sk_buff
*skb,
      struct request_sock *req,
      struct request_sock **prev,
      bool fastopen)
{
 ...
 //创建子 socket
 child = inet_csk(sk)->icsk_af_ops->syn_recv_sock(sk, skb,
req, NULL);
 ...

 //清理半连接队列
 inet_csk_reqsk_queue_unlink(sk, req, prev);
 inet_csk_reqsk_queue_removed(sk, req);

 //添加全连接队列
 inet_csk_reqsk_queue_add(sk, req, child);
 return child;
}
```

5.1 创建子 socket

icsk_af_ops->syn_recv_sock 对应的是 tcp_v4_syn_recv_sock 函数。

```
//file:net/ipv4/tcp_ipv4.c
const struct inet_connection_sock_af_ops ipv4_specific = {
 ......
 .conn_request      = tcp_v4_conn_request,
```

```
 .syn_recv_sock      = tcp_v4_syn_recv_sock,

//三次握手接近就算是完毕了，这里创建 sock 内核对象
struct sock *tcp_v4_syn_recv_sock(struct sock *sk, struct
sk_buff *skb,
      struct request_sock *req,
      struct dst_entry *dst)
{
 //判断接收队列是不是满了
 if (sk_acceptq_is_full(sk))
  goto exit_overflow;

 //创建 sock && 初始化
 newsk = tcp_create_openreq_child(sk, req, skb);
```

注意，在第三次握手的这里又继续判断一次全连接队列是否满了，如果满了修改一下计数器就丢弃了。如果队列不满，那么就申请创建新的 sock 对象。

5.2 删除半连接队列

把连接请求块从半连接队列中删除。

```
//file: include/net/inet_connection_sock.h
static inline void inet_csk_reqsk_queue_unlink(struct sock
*sk, struct request_sock *req,
 struct request_sock **prev)
{
 reqsk_queue_unlink(&inet_csk(sk)->icsk_accept_queue, req,
prev);
}
```

reqsk_queue_unlink 中把连接请求块从半连接队列中删除。

5.3 添加全连接队列

接着添加到全连接队列里边来。

```
//file:net/ipv4/syncookies.c
static inline void inet_csk_reqsk_queue_add(struct sock
*sk,
      struct request_sock *req,
      struct sock *child)
{
 reqsk_queue_add(&inet_csk(sk)->icsk_accept_queue, req, sk,
child);
}
```

在 reqsk_queue_add 中将握手成功的 request_sock 对象插入到全连接队列链表的尾部。

```
//file: include/net/request_sock.h
static inline void reqsk_queue_add(...)
{
 req->sk = child;
 sk_acceptq_added(parent);

 if (queue->rskq_accept_head == NULL)
  queue->rskq_accept_head = req;
 else
  queue->rskq_accept_tail->dl_next = req;

 queue->rskq_accept_tail = req;
 req->dl_next = NULL;
```

```
}
```

5.4 设置连接为 ESTABLISHED

tcp_v4_do_rcv => tcp_child_process => tcp_rcv_state_process

```
//file:net/ipv4/tcp_input.c
int tcp_rcv_state_process(struct sock *sk, struct sk_buff
*skb,
     const struct tcphdr *th, unsigned int len)
{
 ...
 switch (sk->sk_state) {

  //服务端第三次握手处理
  case TCP_SYN_RECV:

   //改变状态为连接
   tcp_set_state(sk, TCP_ESTABLISHED);
   ...
 }
}
```

将连接设置为 TCP_ESTABLISHED 状态。

服务器响应第三次握手 ack 所做的工作是把当前半连接对象删除，创建了新的 sock 后加入到全连接队列中，最后将新连接状态设置为 ESTABLISHED。

六、服务器 accept

最后 accept 一步咱们长话短说。

```
//file: net/ipv4/inet_connection_sock.c
struct sock *inet_csk_accept(struct sock *sk, int flags,
int *err)
{
 //从全连接队列中获取
 struct request_sock_queue *queue =
&icsk->icsk_accept_queue;
 req = reqsk_queue_remove(queue);

 newsk = req->sk;
 return newsk;
}
```

reqsk_queue_remove 这个操作很简单，就是从全连接队列的链表里获取出第一个元素返回就行了。

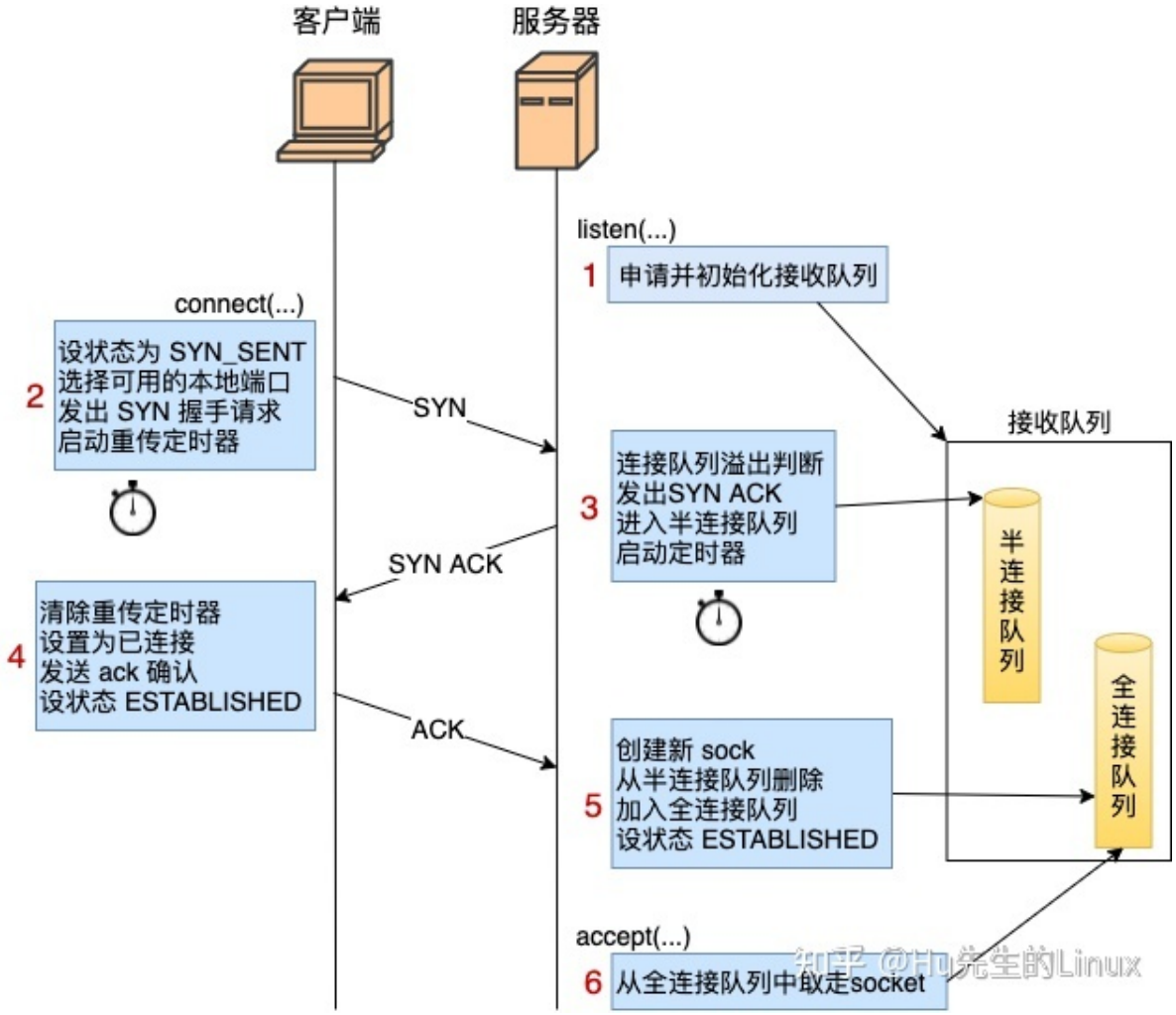
```
//file:include/net/request_sock.h
static inline struct request_sock
*reqsk_queue_remove(struct request_sock_queue *queue)
{
 struct request_sock *req = queue->rskq_accept_head;

 queue->rskq_accept_head = req->dl_next;
 if (queue->rskq_accept_head == NULL)
  queue->rskq_accept_tail = NULL;

 return req;
}
```

所以，accept 的重点工作就是从已经建立好的全连接队列中取出一个返回给用户进程。

本文总结

在后端相关岗位的入职面试中，三次握手的出场频率非常的高。其实在三次握手的过程中，不仅仅是一个握手包的发送 和 TCP 状态的流转。还包含了端口选择，连接队列创建与处理等很多关键技术点。通过今天一篇文章，我们深度去了解了三次握手过程中内核中的这些内部操作。

全文洋洋洒洒上万字字，其实可以用一幅图总结起来。



- 服务器 listen 时，计算了全/半连接队列的长度，还申请了相关内存并初始化。
- 客户端 connect 时，把本地 socket 状态设置成了 TCP_SYN_SENT，选则一个可用的端口，发出 SYN 握手请求并启动重传定时器。

- 服务器响应 ack 时，会判断下接收队列是否满了，满的话可能会丢弃该请求。否则发出 synack，申请 request_sock 添加到半连接队列中，同时启动定时器。
- 客户端响应 synack 时，清除了 connect 时设置的重传定时器，把当前 socket 状态设置为 ESTABLISHED，开启保活计时器后发出第三次握手的 ack 确认。
- 服务器响应 ack 时，把对应半连接对象删除，创建了新的 sock 后加入到全连接队列中，最后将新连接状态设置为 ESTABLISHED。
- accept 从已经建立好的全连接队列中取出一个返回给用户进程。

  另外要注意的是，如果握手过程中发生丢包（网络问题，或者是连接队列溢出），内核会等待定时器到期后重试，重试时间间隔在 3.10 版本里分别是 1s 2s 4s ...。在一些老版本里，比如 2.6 里，第一次重试时间是 3 秒。最大重试次数分别由 tcp_syn_retries 和 tcp_synack_retries 控制。

  如果你的线上接口正常都是几十毫秒内返回，但偶尔出现了 1 s、或者 3 s 等这种偶发的响应耗时变长的问题，那么你就要去定位一下看看是不是出现了握手包的超时重传了。

  以上就是三次握手中一些更详细的内部操作。深度理解这个握手过程对于你排查线上问题会有极大的帮助的。下一讲我们来介绍三次握手中常见的异常问题。

参考资料

推荐一个零声教育C/C++后台开发的免费公开课程，个人觉得老师

讲得不错，分享给大家：C/C++后台开发高级架构师，内容包括Linux，Nginx，ZeroMQ，MySQL，Redis，fastdfs，MongoDB，ZK，流媒体，CDN，P2P，K8S，Docker，TCP/IP，协程，DPDK等技术内容，立即学习

原文：深入理解 Linux 的 TCP 三次握手