收到RST，就一定会断开TCP连接吗?

1. 什么是RST

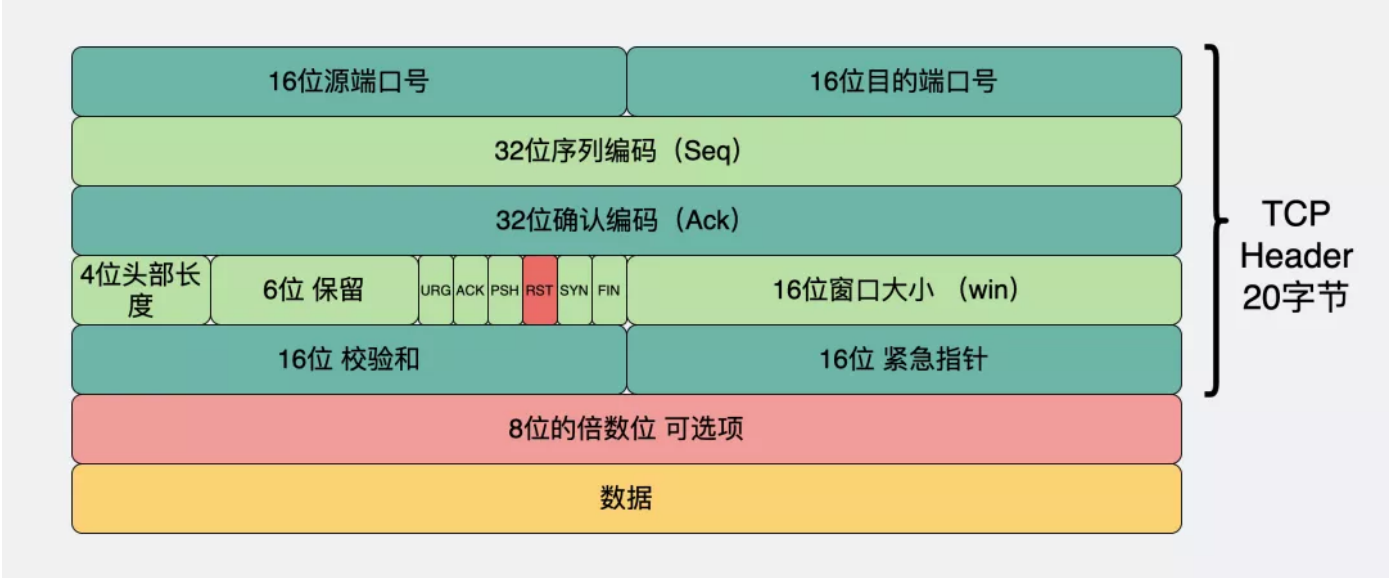
我们都知道TCP正常情况下断开连接是用四次挥手，那是**正常时候**的优雅做法。

但**异常情况**下，收发双方都不一定正常，连挥手这件事本身都可能做不到，所以就需要一个机制去强行关闭连接。

RST 就是用于这种情况，一般用来**异常地**关闭一个连接。它是一个TCP包头中的**标志位**。

正常情况下，不管是**发出**，还是**收到**置了这个标志位的数据包，相应的内存、端口等连接资源都会被释放。从效果上来看就是TCP连接被关闭了。

而接收到 RST的一方，一般会看到一个 `connection reset` 或 `connection refused` 的报错。



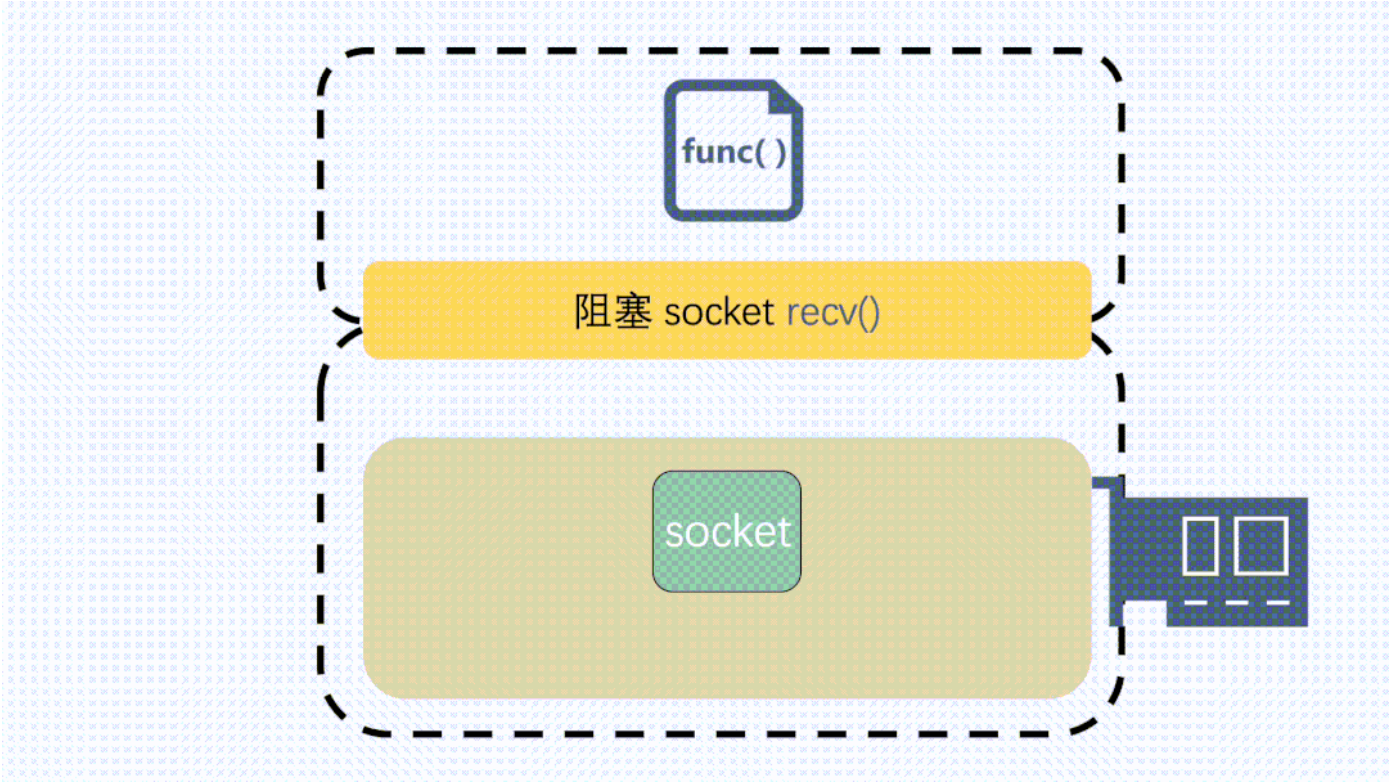
TCP报头RST位

2. 怎么知道收到RST了

我们知道**内核**跟**应用层**是分开的两层，网络通信功能在内核，我们的客户端或服务端属于应用层。应用层**只能**通过 `send/recv` 与内核交互，才能感知到内核是不是收到了 `RST`。

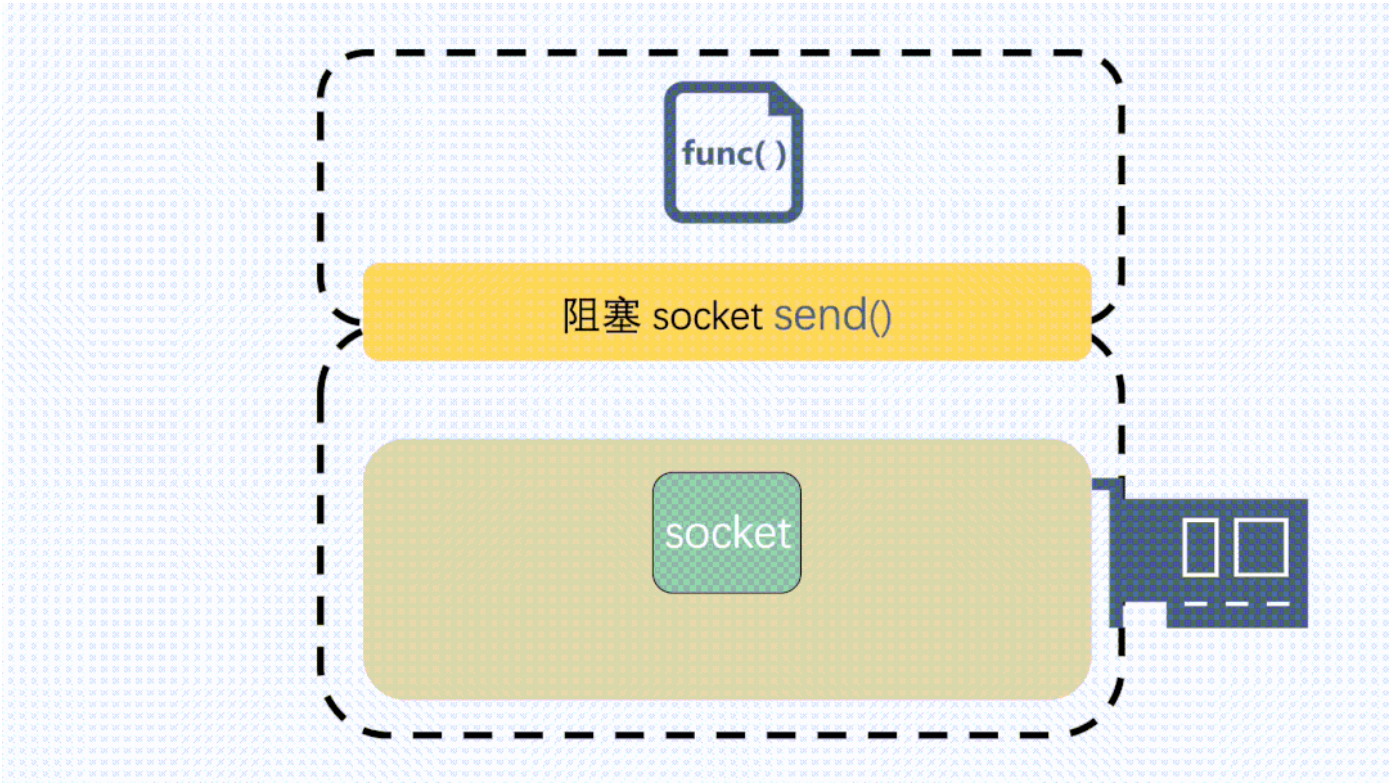
当本端收到远端发来的 `RST` 后，**内核**已经认为此链接已经关闭。

此时如果本端**应用层**尝试去执行**读数据**操作，比如 `recv`，应用层就会收到 **Connection reset by peer** 的报错，意思是**远端已经关闭连接**。



ResetByPeer

如果本端**应用层**尝试去执行**写数据**操作，比如`send`，那么应用层就会收到 **Broken pipe** 的报错，意思是发送通道已经坏了。



BrokenPipe

这两个是开发过程中很经常遇到的报错，感觉大家可以**把这篇文章放进收藏夹吃灰**了，等遇到这个问题了，再打开来擦擦灰，说不定对你会有帮助。

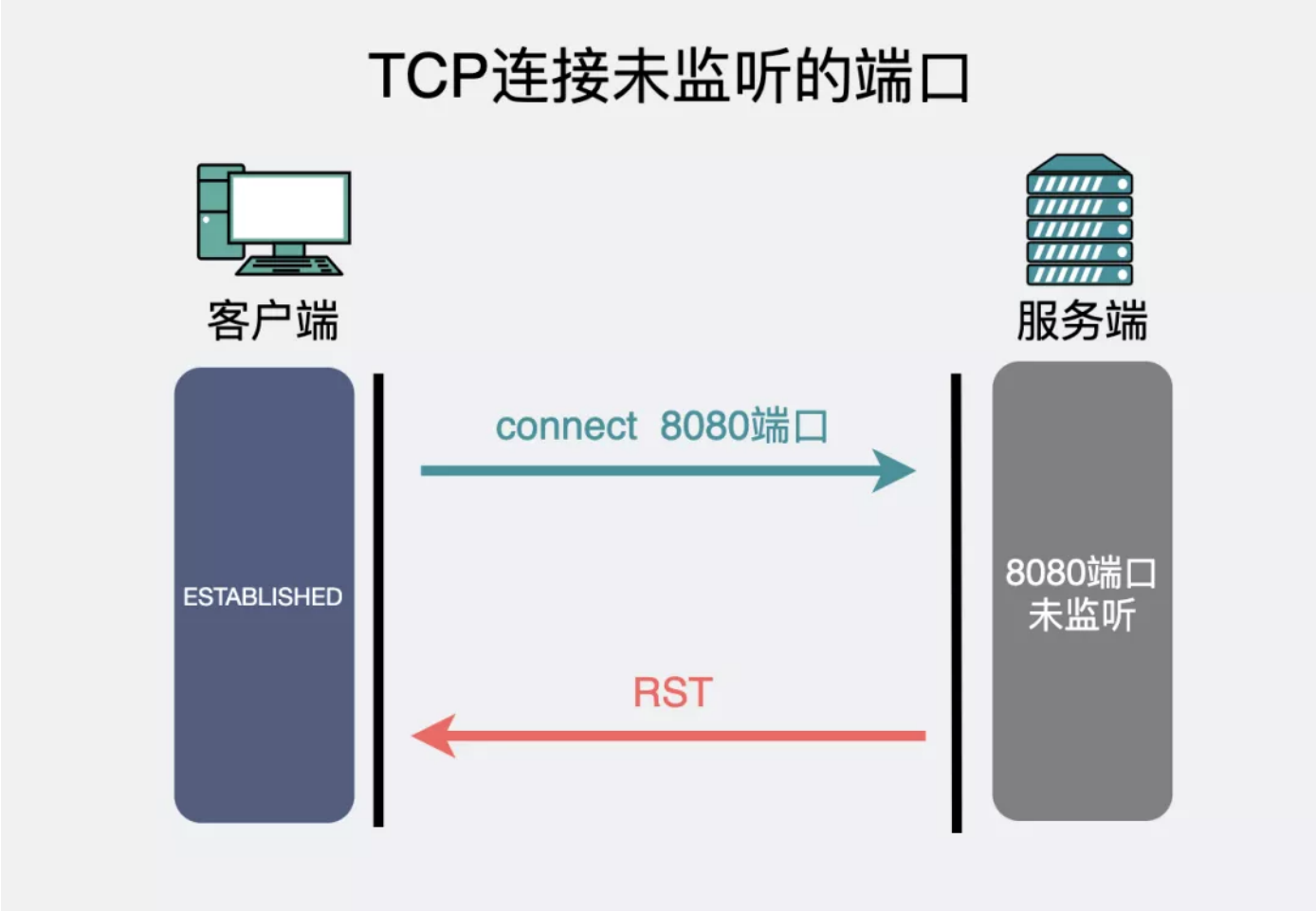
3. 出现RST的场景有哪些

RST一般出现于异常情况，归类为 **对端的端口不可用** 和 **socket提前关闭**。

3.1 端口不可用

端口不可用分为两种情况。要么是这个端口从来就没有"可用"过，比如根本就没监听（**listen**）过；要么就是曾经"可用"，但现在"不可用"了，比如服务**突然崩**了。

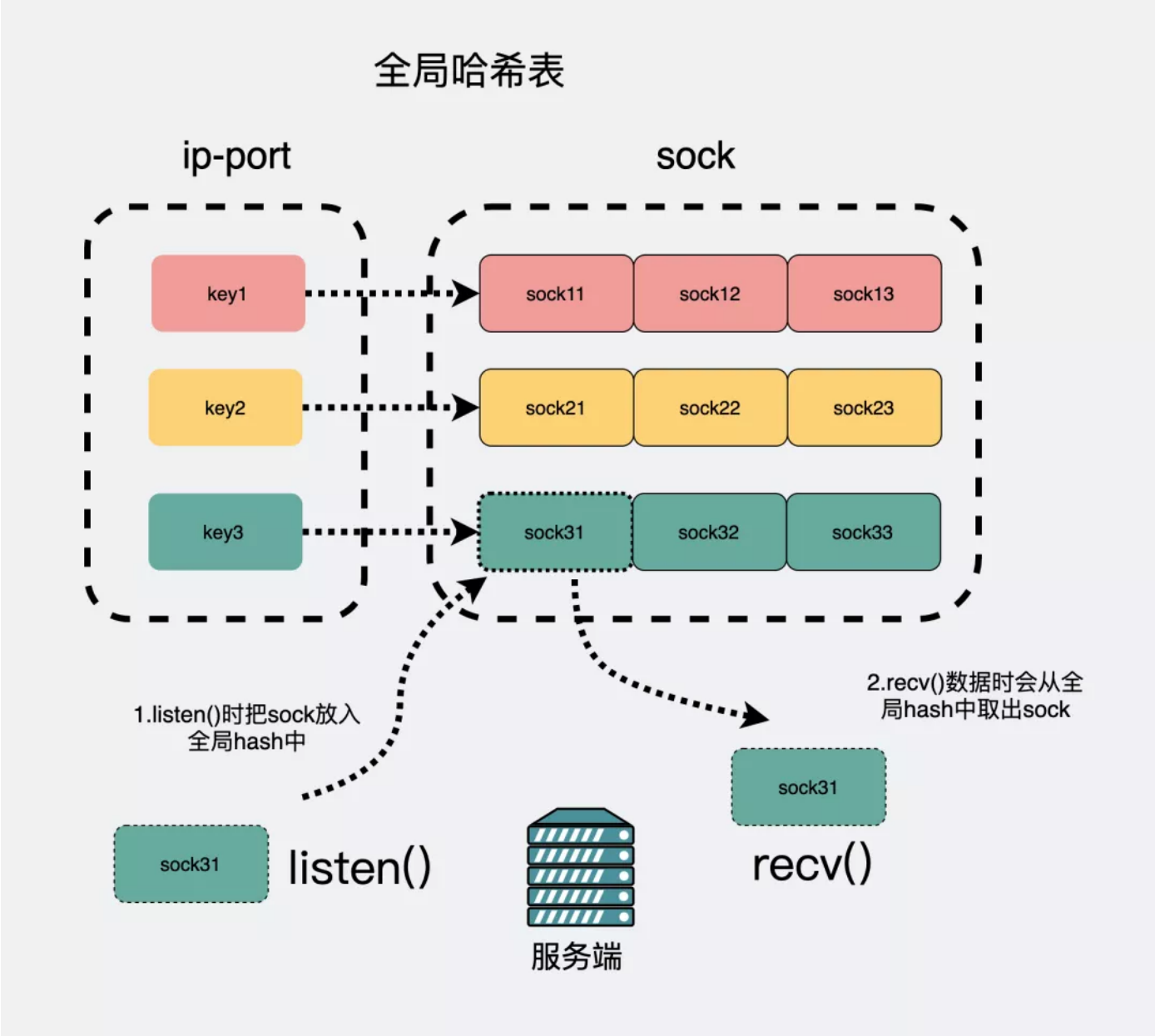
3.1.1 端口未监听



TCP连接未监听的端口

服务端`listen`方法会创建一个`sock`放入到全局的`哈希表`中。

此时客户端发起一个`connect`请求到服务端。服务端在收到数据包之后，第一时间会根据IP和端口从哈希表里去获取`sock`。



全局hash表

如果服务端执行过`listen`，就能从`全局哈希表`里拿到`sock`。

但如果服务端没有执行过`listen`，那`哈希表`里也就不会有对应的`sock`，结果当然是拿不到。此时，**正常情况下**服务端会发`RST`给客户端。

端口未监听就一定会发RST吗?

不一定。上面提到，发RST的前提是**正常情况下**，我们看下源码。

```
// net/ipv4/tcp_ipv4.c
// 代码经过删减
int tcp_v4_rcv(struct sk_buff *skb)
{
    // 根据ip、端口等信息 获取sock。
    sk = __inet_lookup_skb(&tcp_hashinfo, skb, th->source, th->dest);
    if (!sk)
```

```
        goto no_tcp_socket;

no_tcp_socket:
    // 检查数据包有没有出错
    if (skb->len < (th->doff << 2) || tcp_checksum_complete(skb)) {
        // 错误记录
    } else {
        // 发送RST
        tcp_v4_send_reset(NULL, skb);
    }
}
```

内核在收到数据后会从物理层、数据链路层、网络层、传输层、应用层，一层一层往上传递。到传输层的时候，根据当前数据包的协议是**TCP还是UDP**走不一样的函数方法。可以简单认为，**TCP**数据包都会走到`tcp_v4_rcv()`。这个方法会从全局哈希表里获取`sock`，如果此时服务端没有`listen()`过，那肯定获取不了`sock`，会跳转到`no_tcp_socket`的逻辑。

注意这里会先走一个`tcp_checksum_complete()`，目的是看看数据包的**校验和(Checksum)**是否合法。

> **校验和**可以验证数据从端到端的传输中是否出现异常。由发送端计算，然后由接收端验证。计算范围覆盖数据包里的TCP首部和TCP数据。

如果在发送端到接收端传输过程中，数据发生**任何改动**，比如被第三方篡改，那么接收方能检测到校验和有差错，此时TCP段会被直接丢弃。如果校验和没问题，那才会发RST。

所以，**只有在数据包没问题的情况下，比如校验和没问题，才会发RST包给对端。**

为什么数据包异常的情况下，不发RST?

一个数据包连校验都不能通过，那这个包，**多半有问题**。

有可能是在发送的过程中被篡改了，又或者，可能只是一个**胡乱伪造**的数据包。

五层网络，不管是哪一层，只要遇到了这种数据，**推荐的做法都是默默扔掉，而不是**去回复一个消息告诉对方数据有问题。

如果对方用的是TCP，是可靠传输协议，发现很久没有`ACK`响应，自己就会重传。

如果对方用的是UDP，说明发送端已经接受了“不可靠会丢包”的事实，那丢了就丢了。

因此，数据包异常的情况下，默默扔掉，不发`RST`，非常合理。

还是不能理解？那我**再举个例子**。

正常人喷你，他说话**条理清晰，主谓宾分明**。此时你喷回去，那你是个充满热情，正直，富有判断力的好人。

而此时一个憨憨也想喷你，但他**思维混乱，连话都说不清楚，一直阿巴阿巴**的，你虽然听不懂，但**大受震撼**，此时你会?

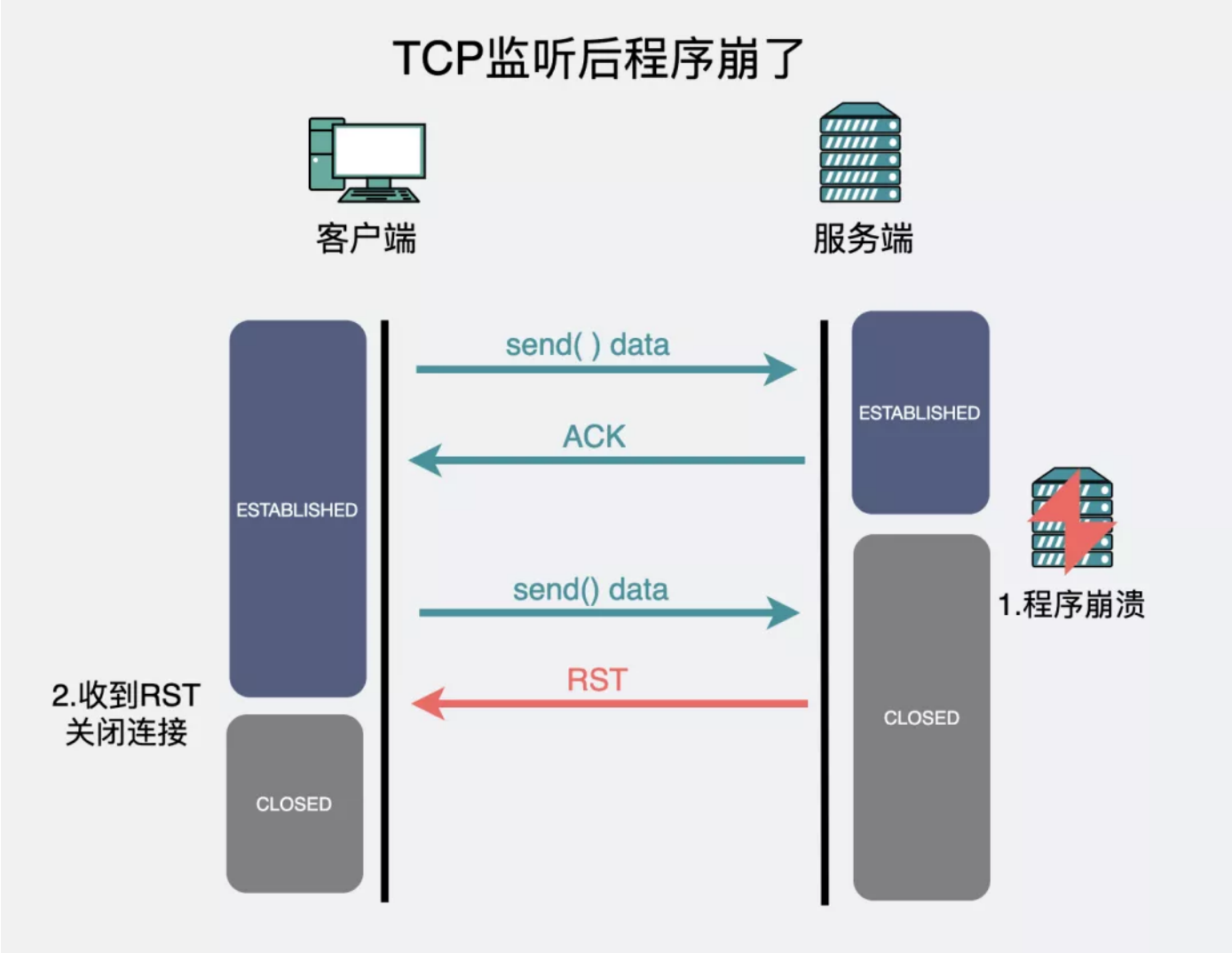
- A：跟他激情互喷
- B：不跟他一般见识，就当没听过

一般来说**最优选择是B**，毕竟你理他，他反而来劲。这下，应该就懂了。

3.1.2 程序启动了但是崩了

端口不可用的场景里，除了端口未监听以外，还有可能是从前监听了，但服务端机器上做监听操作的**应用程序突然崩了**，此时客户端还像往常一样正常发送消息，服务器内核协议栈收到消息后，则会**回一个RST**。在开发过程中，**这种情况是最常见的**。

比如你的服务端应用程序里，弄了个**空指针**，或者**数组越界**啥的，程序立马就崩了。



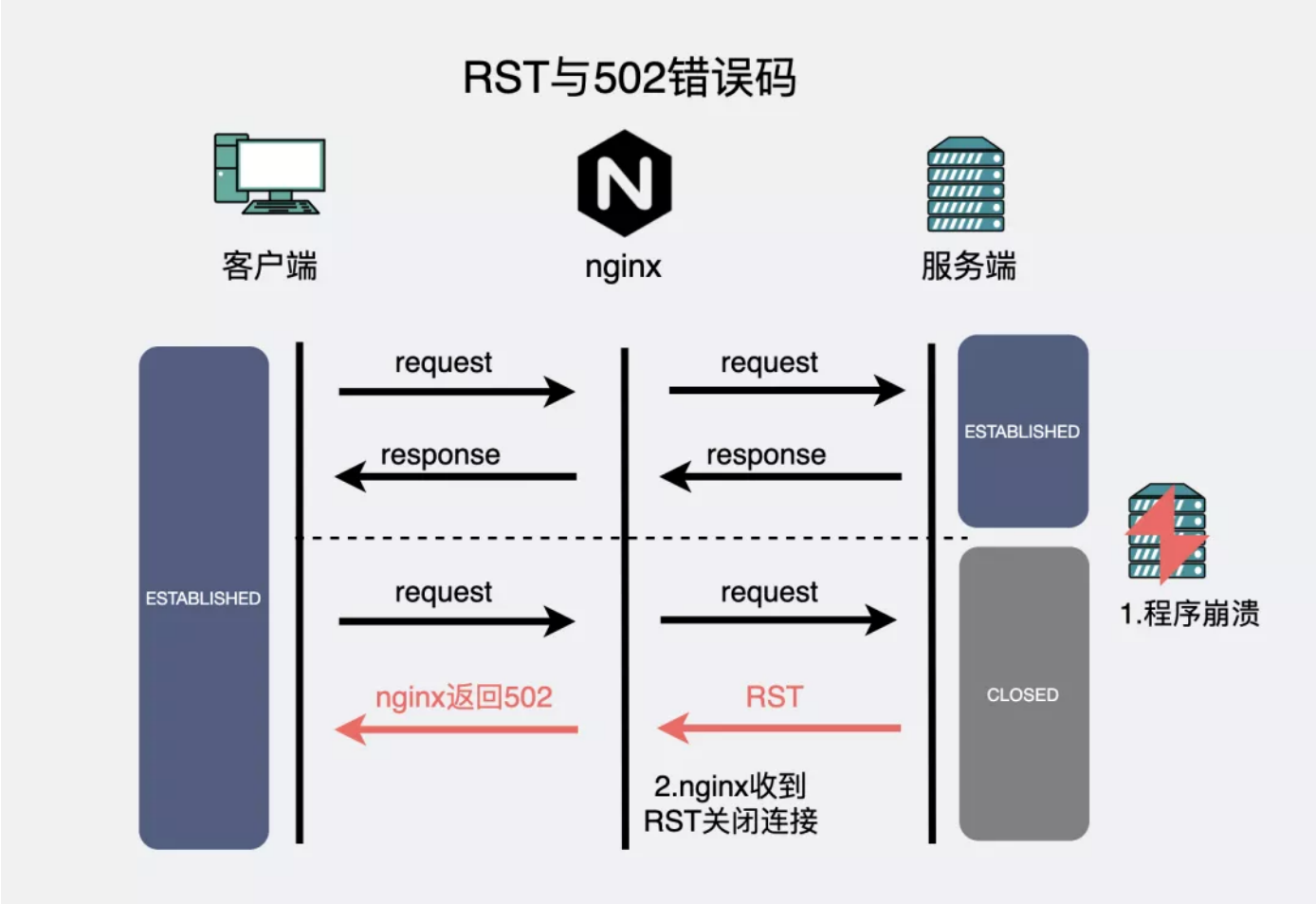
TCP监听了但崩了

这种情况跟**端口未监听**本质上类似，在服务端的应用程序**崩溃后**，原来监听的端口资源就被释放了，从效果上来看，类似于处于`CLOSED`状态。

此时服务端又收到了客户端发来的消息，内核协议栈会根据**IP端口**，从全局哈希表里查找`sock`，结果当然是拿不到对应的`sock`数据，于是走了跟上面**"端口未监听"**时一样的逻辑，回了个`RST`。客户端在收到RST后也**释放了sock资源**，从效果上来看，就是**连接断了**。

RST和502的关系

上面这张图，服务端程序崩溃后，如果客户端再有数据发送，会出现`RST`。但如果在客户端和服务端中间再加一个`nginx`，就像下图一样。



RST与502

`nginx`会作为客户端和服务端之间的"中间人角色"，负责**转发**请求和响应结果。但当服务端程序**崩溃**，比如出现**野指针或者OOM**的问题，那转发到服务器的请求，必然得不到响应，后端服务端还会返回一个`RST`给`nginx`。`nginx`在收到这个`RST`后会断开与服务端的连接，同时返回客户端一个`502`错误码。

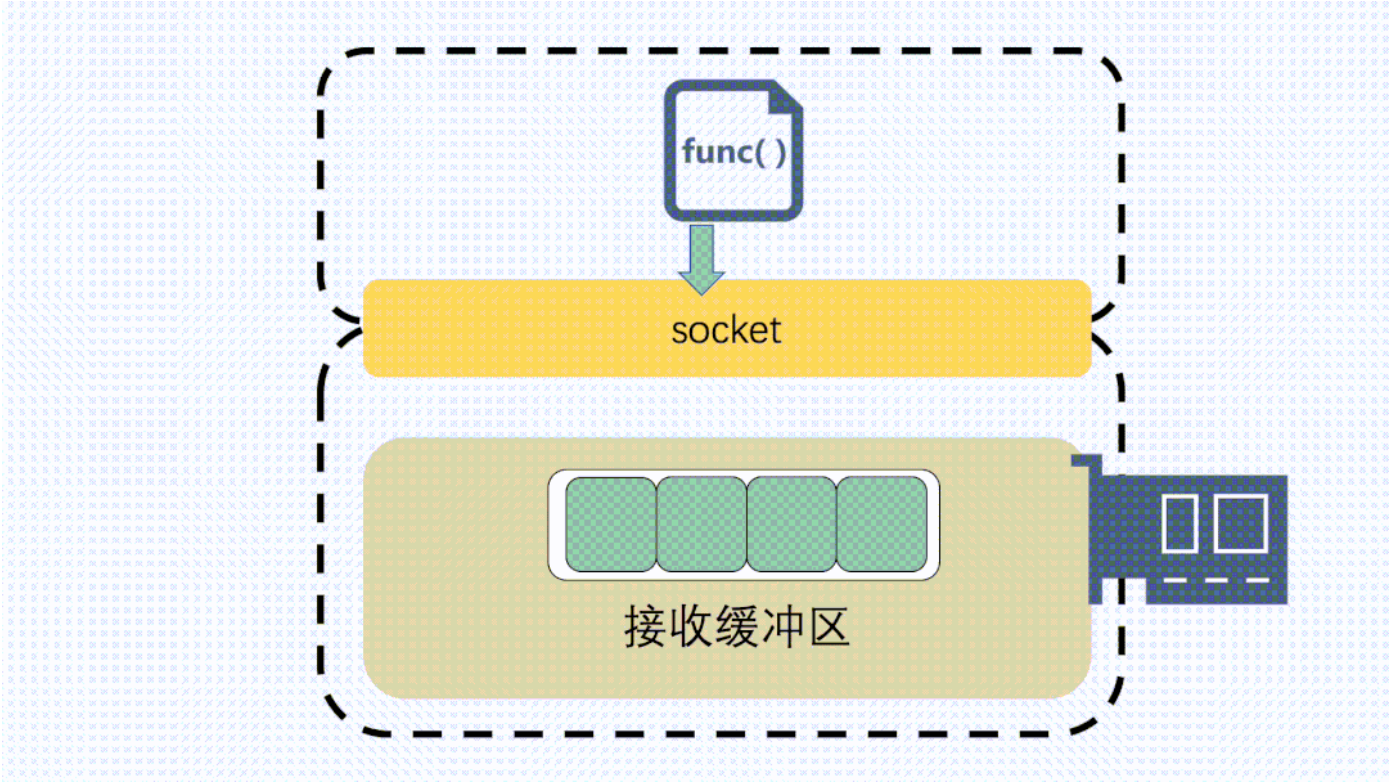
所以，出现502问题，一般情况下都是因为后端程序崩了，基于这一点假设，去看看监控是不是发生了OOM或者日志是否有空指针等报错信息。

3.2 socket提前关闭

这种情况分为**本端**提前关闭，和**远端**提前关闭。

3.2.1 本端提前关闭

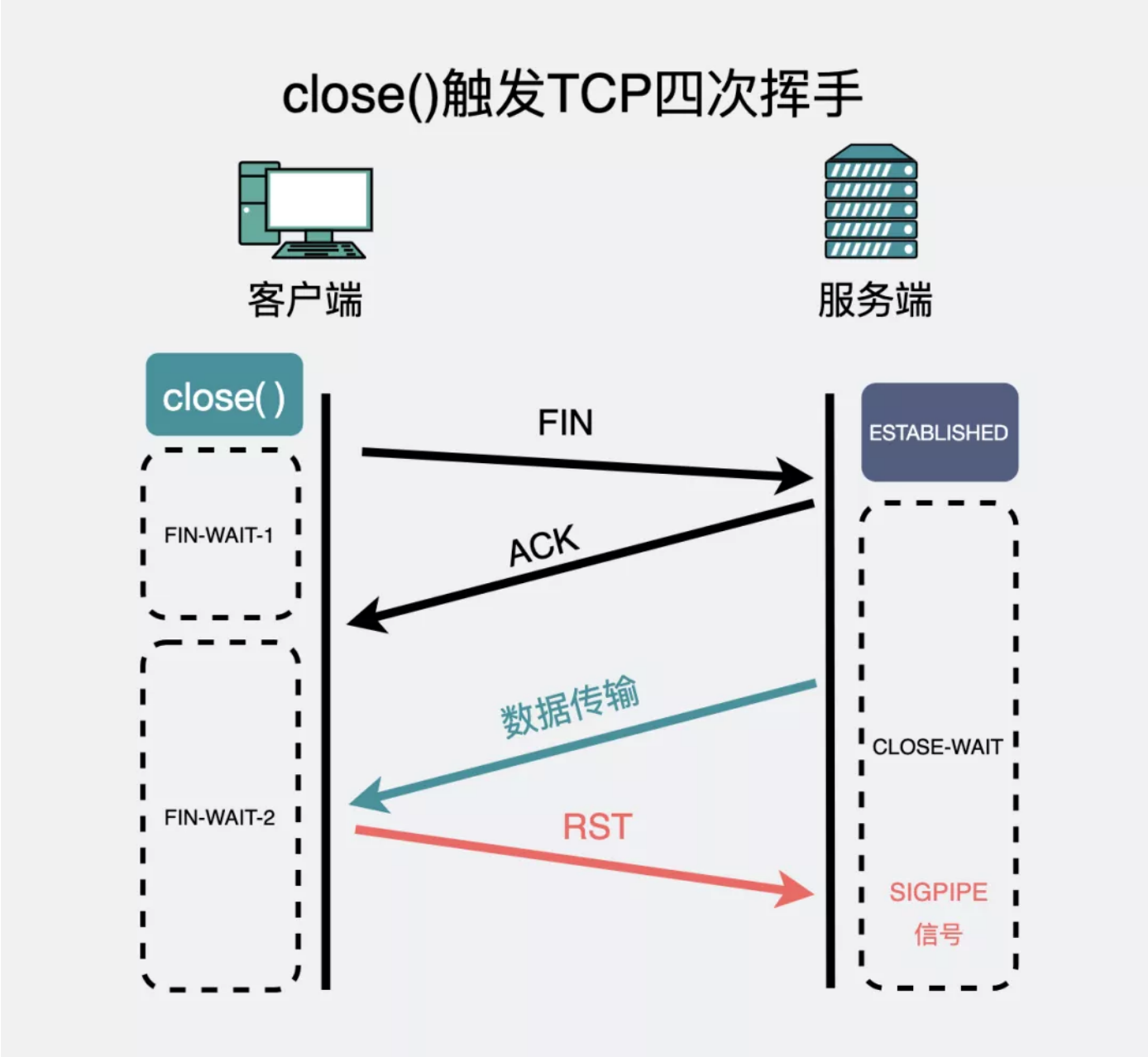
如果本端`socket`接收缓冲区**还有数据未读**，此时**提前**`close()` **socket**。那么本端会先把接收缓冲区的数据清空，然后给远端发一个RST。



recvbuf非空

3.2.2 远端提前关闭

远端已经`close()`了`socket`，此时本端还尝试发数据给远端。那么远端就会回一个RST。



close()触发TCP四次挥手

大家知道，TCP是**全双工通信**，意思是发送数据的同时，还可以接收数据。

`Close()`的含义是，此时要同时**关闭发送和接收**消息的功能。

客户端执行`close()`， 正常情况下，会发出**第一次**挥手FIN，然后服务端回**第二次**挥手ACK。如果在**第二次和第三次挥手之间**，如果服务方还尝试传数据给客户端，那么客户端不仅不收这个消息，还会发一个RST消息到服务端。直接结束掉这次连接。

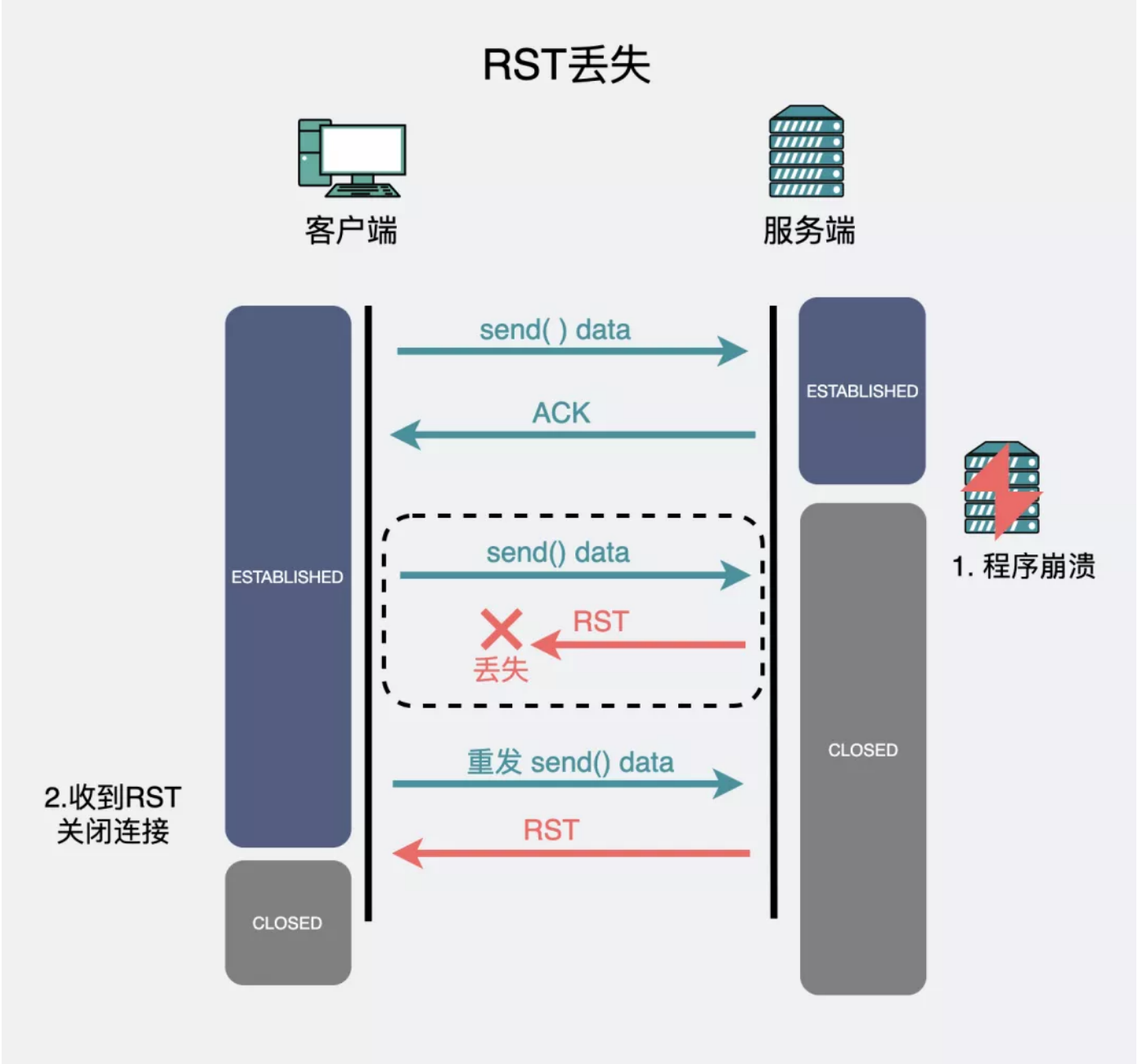
4. 对方没收到RST，会怎么样?

我们知道TCP是可靠传输，意味着本端发一个数据，远端在收到这个数据后就会回一个`ACK`，意思是"我收到这个包了"。

而RST，不需要ACK确认包。

因为`RST`本来就是设计来处理异常情况的，既然都已经在异常情况下了，还指望对方能正常回你一个`ACK`吗？**可以幻想，不要妄想。**

但**问题又来了**，网络环境这么复杂，丢包也是分分钟的事情，既然RST包不需要ACK来确认，那万一对方就是没收到RST，会怎么样？

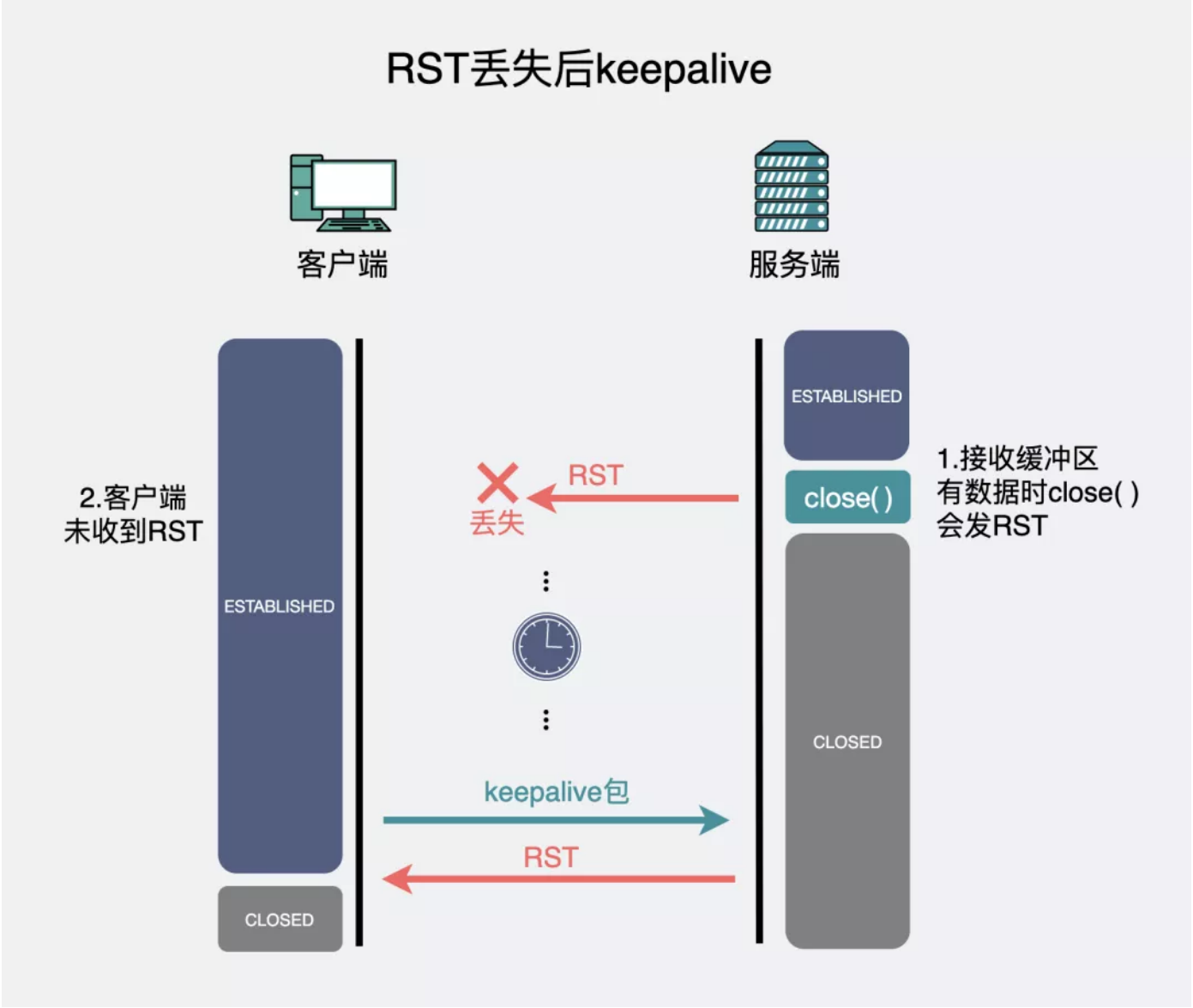


RST丢失

RST丢了，问题不大。比方说上图服务端，发了RST之后，服务端就认为连接不可用了。

如果客户端之前**发送了数据**，一直没等到这个数据的确认ACK，就会重发，重发的时候，自然就会触发一个新的RST包。

而如果客户端之前**没有发数据**，但服务端的RST丢了，TCP有个keepalive机制，会定期发送探活包，这种数据包到了服务端，也会重新触发一个RST。



RST丢失后keepalive

5. 收到RST就一定会断开连接吗?

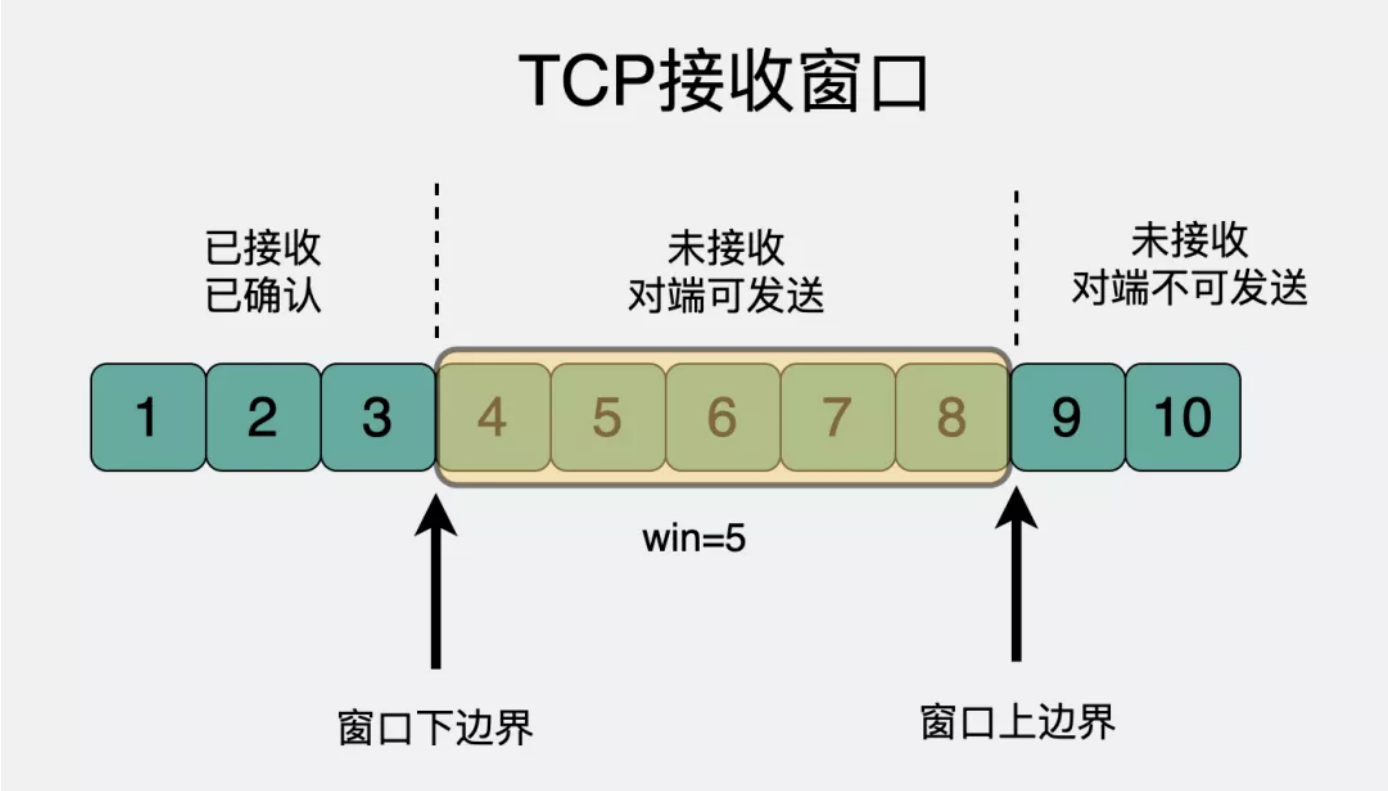
先说结论，**不一定会断开**。我们看下源码。

```
// net/ipv4/tcp_input.c
static bool tcp_validate_incoming()
{
    // 获取sock
    struct tcp_sock *tp = tcp_sk(sk);

    // step 1: 先判断seq是否合法（是否在合法接收窗口范围内）
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
        goto discard;
    }

    // step 2: 执行收到 RST 后该干的事情
```

```
13      if (th->rst) {
14          if (TCP_SKB_CB(skb)->seq == tp->rcv_nxt)
15              tcp_reset(sk);
16          else
17              tcp_send_challenge_ack(sk);
18          goto discard;
19      }
20  }
```

收到RST包，第一步会通过`tcp_sequence`先看下这个seq是否合法，其实主要是看下这个seq是否在合法**接收窗口**范围内。**如果不在范围内，这个RST包就会被丢弃。**

至于接收窗口是个啥，我们先看下面这个图。

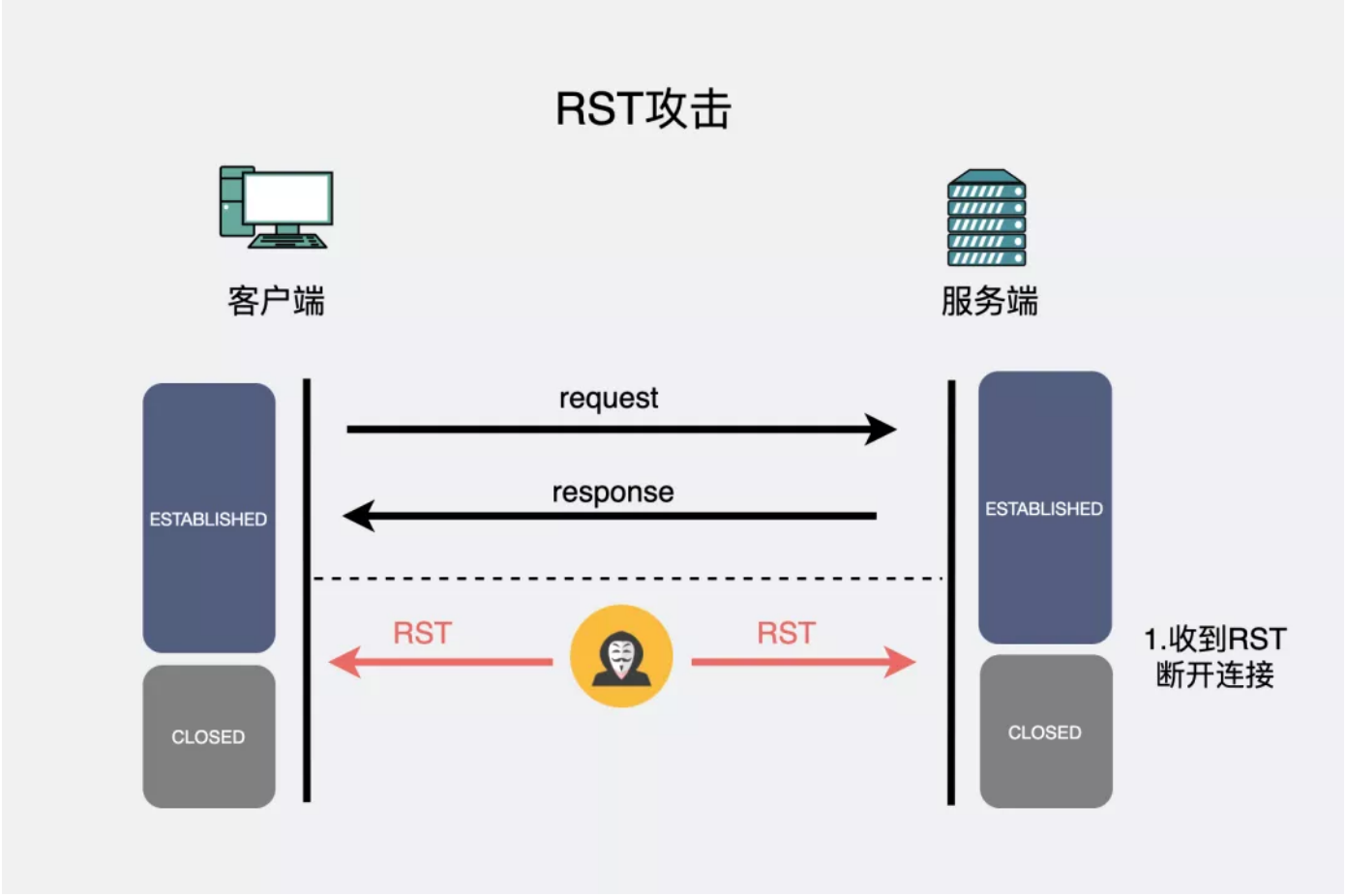


接收窗口

这里**黄色的部分**，就是指接收窗口，只要RST包的seq不在这个窗口范围内，那就会被丢弃。

5.1 为什么要校验是否在窗口范围内

正常情况下客户端服务端双方可以通过RST来断开连接。假设不做seq校验，如果这时候有不怀好意的第三方介入，构造了一个RST包，且在TCP和IP等报头都填上客户端的信息，发到服务端，那么服务端就会断开这个连接。同理也可以伪造服务端的包发给客户端。这就叫**RST攻击**。



RST攻击

受到RST攻击时，从现象上看，客户端老感觉服务端崩了，这非常影响用户体验。

如果这是个游戏，我相信多崩几次，第二天大家就不来玩了。

实际消息发送过程中，接收窗口是不断移动的，seq也是在飞快的变动中，此时第三方是**比较难**构造出合法seq的RST包的，那么通过这个seq校验，就可以拦下了很多不合法的消息。

5.2 加了窗口校验就不能用RST攻击了吗

不是，只是增加了攻击的成本。但如果想搞，还是可搞的。

以下是**面向监狱编程**的环节。希望大家只**了解原理**就好了，**不建议使用**。相信大家都不喜欢穿着蓝白条纹的衣服，拍**纯狱风**的照片。

从上面可以知道，不是每一个RST包都会导致连接重置的，要求是这个RST包的seq要在窗口范围内，所以，问题就变成了，**我们怎么样才能构造出合法的seq**。

盲猜seq

窗口数值seq本质上只是个uint32类型。

```
struct tcp_skb_cb {
    __u32       seq;        /* Starting sequence number */
}
```

如果在这个范围内疯狂猜测seq数值，并构造对应的包，发到目的机器，虽然概率低，但是总是能被试出来，从而实现**RST攻击**。这种乱棍打死老师傅的方式，就是所谓的**合法窗口盲打（blind in-window attacks）**。

觉得这种方式比较**笨**？那有没有聪明点的方式，还真有，但是在这之前需要先看下面的这个问题。

已连接状态下收到第一次握手包会怎么样?

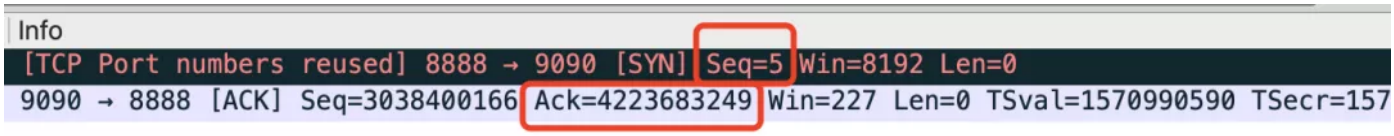
我们需要了解一个问题，比如服务端在已连接（`ESTABLISHED`）状态下，如果收到客户端发来的第一次握手包（`SYN`），会怎么样?

以前我以为**服务单会认为客户端憨憨了，直接RST连接。**

但实际，并不是。

```
static bool tcp_validate_incoming()
{
    struct tcp_sock *tp = tcp_sk(sk);

    /* 判断seq是否在合法窗口内 */
    if (!tcp_sequence(tp, TCP_SKB_CB(skb)->seq, TCP_SKB_CB(skb)->end_seq)) {
        if (!th->rst) {
            // 收到一个不在合法窗口内的SYN包
            if (th->syn)
                goto syn_challenge;
        }
    }

    /*
     * RFC 5691 4.2 : 发送 challenge ack
     */
    if (th->syn) {
syn_challenge:
        tcp_send_challenge_ack(sk);
    }
}
```

当客户端发出一个不在合法窗口内的SYN包的时候，服务端会发一个带有正确的seq数据ACK包出来，这个ACK包叫`challenge ack`。



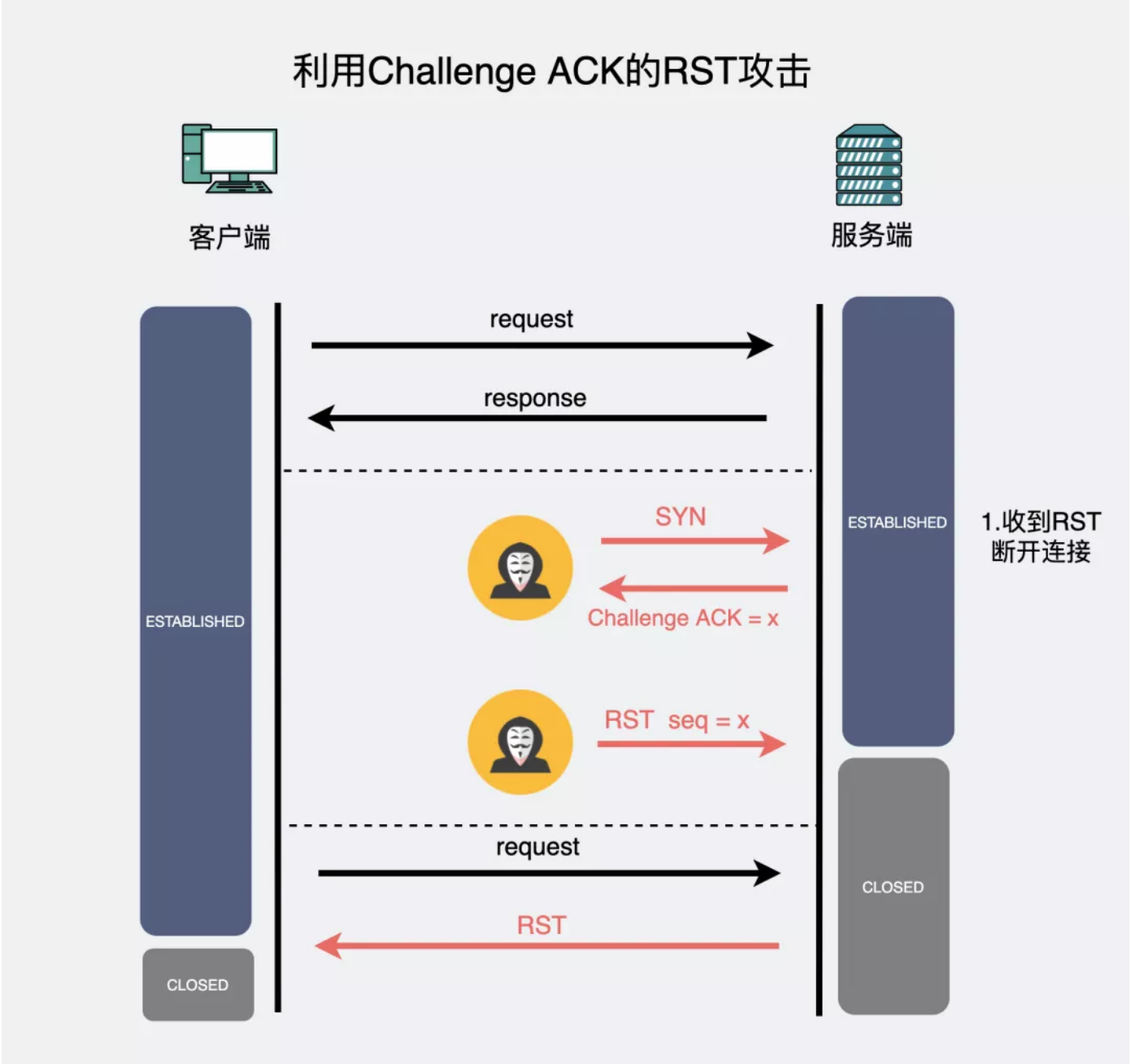
challenge ack抓包

上图是抓包的结果，用`scapy`随便伪造一个`seq=5`的包发到服务端（`端口9090`），服务端回复一个带有正确seq值的`challenge ack`包给客户端（`端口8888`）。

利用challenge ack获取seq

上面提到的**这个challenge ack，仿佛为盲猜seq的老哥们打开了一个新世界。**

在获得这个`challenge ack`后，攻击程序就可以以ack值为基础，在一定范围内设置seq，这样造成RST攻击的几率就大大增加了。



利用ChallengeACK的RST攻击

总结

- RST其实是TCP包头里的一个标志位，目的是为了在**异常情况**下关闭连接。
- 内核收到RST后，应用层只能通过调用读/写操作来感知，此时会对应获得 **Connection reset by peer** 和 **Broken pipe** 报错。
- 发出RST后不需要得到对方的ACK确认包，因此RST丢失后对方不能立刻感知，但是通过下一次**重传**数据或keepalive**心跳包**可以导致RST重传。
- **收到RST包，不一定会断开连接，seq不在合法窗口范围内的数据包会被默默丢弃。**通过构造合法窗口范围内seq，可以造成RST攻击，**这一点大家了解就好，千万别学！**

参考资料

TCP旁路攻击分析与重现 - https://www.cxyzjd.com/article/qq_27446553/52416369