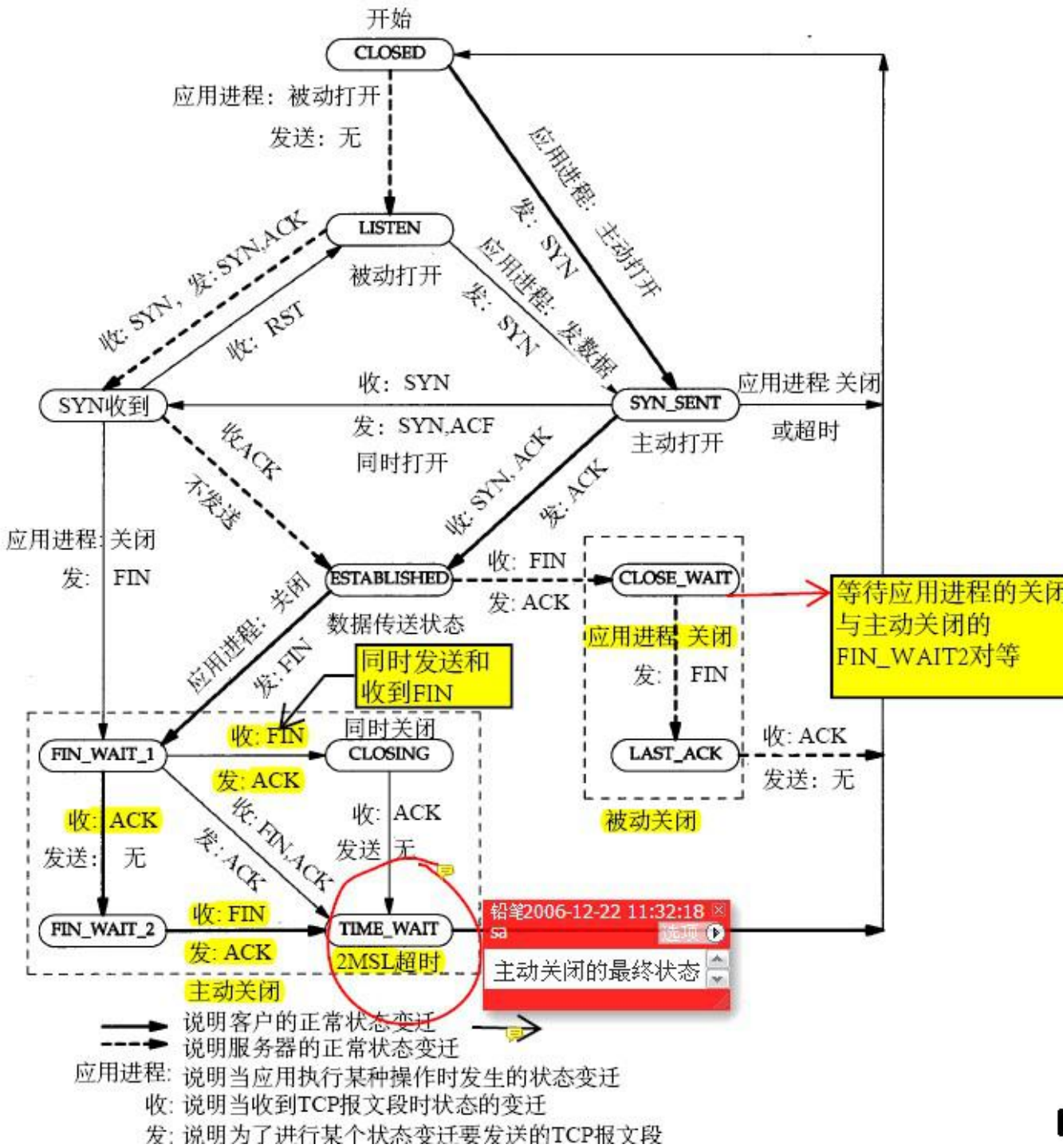
TCP/IP 连接状态变迁图 CLOSE_WAIT

终止一个连接要经过 4 次握手。这由 TCP 的半关闭（half-close）造成的。既然一个 TCP 连接是全双工（即数据在两个方向上能同时传递，可理解为两个方向相反的独立通道），因此每个方向必须单独地进行关闭。这原则就是当一方完成它的数据发送任务后就能发送一个 FIN 来终止这个方向连接。当一端收到一个 FIN，内核让 read 返回 0 来通知应用层另一端已经终止了向本端的数据传送。发送 FIN 通常是应用层对 socket 进行关闭的结果。

例如：TCP 客户端发送一个 FIN，用来关闭从客户到服务器的数据传送。

半关闭对服务器究竟有什么影响呢？先看看下面的 TCP 状态转化图



tcp 状态装换图

首先对上面这个图示进行解释：

CLOSED：表示初始状态。对服务端和 C 客户端双方都一样。

LISTEN：表示监听状态。服务端调用了 listen 函数，可以开始 accept 连接了。

SYN_SENT：表示客户端已经发送了 SYN 报文。当客户端调用 connect 函数发起连接时，首先发 SYN 给服务端，然后自己进入 SYN_SENT 状态，并等待服务端发送 ACK+SYN。

SYN_RCVD：表示服务端收到客户端发送 SYN 报文。服务端收到这个报文后，进入 SYN_RCVD 状态，然后发送 ACK+SYN 给客户端。

ESTABLISHED：表示连接已经建立成功了。服务端发送完 ACK+SYN 后进入该状态，客户端收到 ACK 后也进入该状态。

FIN_WAIT_1：表示主动关闭连接。无论哪方调用 close 函数发送 FIN 报文都会进入这个这个状态。

FIN_WAIT_2：表示被动关闭方同意关闭连接。主动关闭连接方收到被动关闭方返回的 ACK 后，会进入该状态。

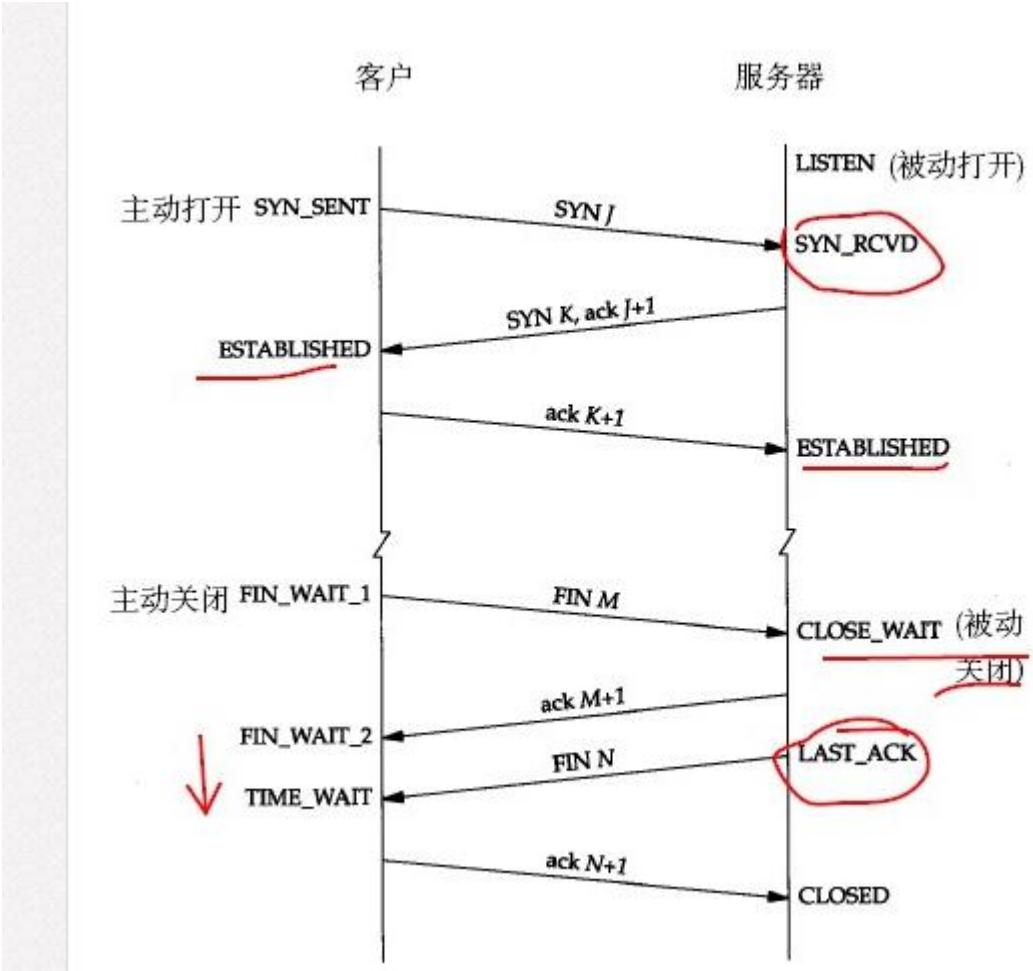
TIME_WAIT：表示收到对方的 FIN 报文并发送了 ACK 报文，就等 2MSL 后即可回到 CLOSED 状态了。如果 FIN_WAIT_1 状态下，收到对方同时带 FIN 标志和 ACK 标志的报文时，可以直接进入 TIME_WAIT 状态，而无须经过 FIN_WAIT_2 状态。

CLOSING：表示双方同时关闭连接。如果双方几乎同时调用 close 函数，那么会出现双方同时发送 FIN 报文的情况，此时就会出现 CLOSING 状态，表示双方都在关闭连接。

CLOSE_WAIT：表示被动关闭方等待关闭。当收到对方调用 close 函数发送的 FIN 报文时，回应对方 ACK 报文，此时进入 CLOSE_WAIT 状态。

LAST_ACK：表示被动关闭方发送 FIN 报文后，等待对方的 ACK 报文状态，当收到 ACK 后进入 CLOSED 状态。

三次握手和四次握手所对应的上述状态：



客户端主动关闭时，发出 FIN 包，收到服务器的 ACK，客户端停留在 FIN_WAIT2 状态。而服务端收到 FIN，发出 ACK 后，停留在 COLSE_WAIT 状态。

这个 CLOSE_WAIT 状态非常讨厌，它持续的时间非常长，服务器端如果积攒大量的 COLSE_WAIT 状

态的 socket，有可能将服务器资源耗尽，进而无法提供服务。

那么，服务器上是怎么产生大量的失去控制的 COLSE_WAIT 状态的 socket 呢？我们来追踪一下。

一个很浅显的原因是，服务器没有继续发 FIN 包给客户端。

服务器为什么不发 FIN，可能是业务实现上的需要，现在不是发送 FIN 的时机，因为服务器还有数据要发往客户端，发送完了自然就要通过系统调用发 FIN 了，这个场景并不是上面我们提到的持续的 COLSE_WAIT 状态，这个在受控范围之内。

那么究竟是什么原因呢，咱们引入两个系统调用 close(sockfd)和 shutdown(sockfd,how)接着往下分析。

在这儿，需要明确的一个概念---- 一个进程打开一个 socket，然后此进程再派生子进程的时候，此 socket 的 sockfd 会被继承。socket 是系统级的对象，现在的结果是，此 socket 被两个进程打开，此 socket 的引用计数会变成 2。

继续说上述两个系统调用对 socket 的关闭情况。

调用 close(sockfd)时，内核检查此 fd 对应的 socket 上的引用计数。如果引用计数大于 1，那么将这个引用计数减 1，然后返回。如果引用计数等于 1，那么内核会真正通过发 FIN 来关闭 TCP 连接。

调用 shutdown(sockfd，SHUT_RDWR)时,内核不会检查此 fd 对应的 socket 上的引用计数，直接通过发 FIN 来关闭 TCP 连接。

现在应该真相大白了，可能是服务器的实现有点问题，父进程打开了 socket，然后用派生子进程来处理业务，父进程继续对网络请求进行监听，永远不会终止。客户端发 FIN 过来的时候，处理业务的子进程的 read 返回 0，子进程发现对端已经关闭了，直接调用 close()对本端进行关闭。实际上，仅仅使 socket 的引用计数减 1，socket 并没关闭。从而导致系统中又多了一个 CLOSE_WAIT 的 socket。。。

如何避免这样的情况发生？

子进程的关闭处理应该是这样的：

```
shutdown(sockfd,SHUT_RDWR);

close(sockfd);
```

这样处理，服务器的 FIN 会被发出，socket 进入 LAST_ACK 状态，等待最后的 ACK 到来，就能进入初始状态 CLOSED。

补充一下 shutdown()的函数说明

linux 系统下使用 shutdown 系统调用来控制 socket 的关闭方式

```
int shutdown(intsockfd,int how);
```

参数 how 允许为 shutdown 操作选择以下几种方式：

SHUT_RD：关闭连接的读端。也就是该套接字不再接受数据，任何当前在套接字接受缓冲区的数据将被丢弃。进程将不能对该套接字发出任何读操作。对 TCP 套接字该调用之后接受到的任何数据将被确认然后被丢弃。

SHUT_WR:关闭连接的写端。

SHUT_RDWR:相当于调用 shutdown 两次：首先是以 SHUT_RD,然后以 SHUT_WR

注意:

在多进程中如果一个进程中 shutdown(sfd, SHUT_RDWR)后其它的进程将无法进行通信.如果一个进程 close(sfd)将不会影响到其它进程.