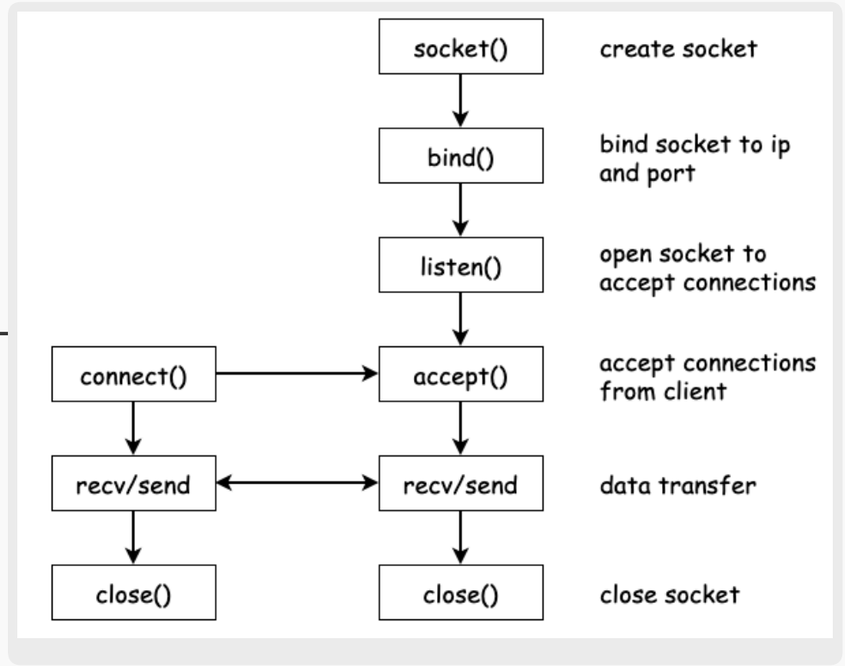
socket

分支主题 1

基本流程



recv

ssize_t recv(int sockfd, void *buffer, size_t length, int flags);

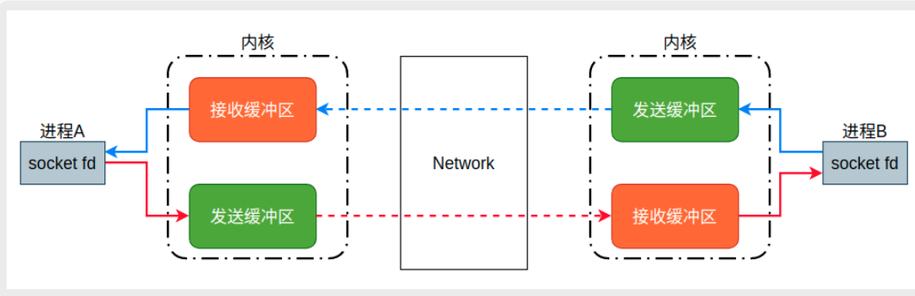
recv() 函数专用于 scoket 上的 read 操作，本质上是从接收缓冲区读取数据

该系统调用将返回实际读取的字节数，其值可能会小于传入的 length 参数

TCP 流式协议

首先，TCP 协议是可靠传输协议，也就是说，内核实现的 TCP 协议将保证将一个 TCP 包发送给另一端

拥塞控制也是为了实现可靠传输协议中的重要一环，因此，TCP 中的发送缓冲区和接收缓冲区就成了一种"必然"：TCP 需要这个东西

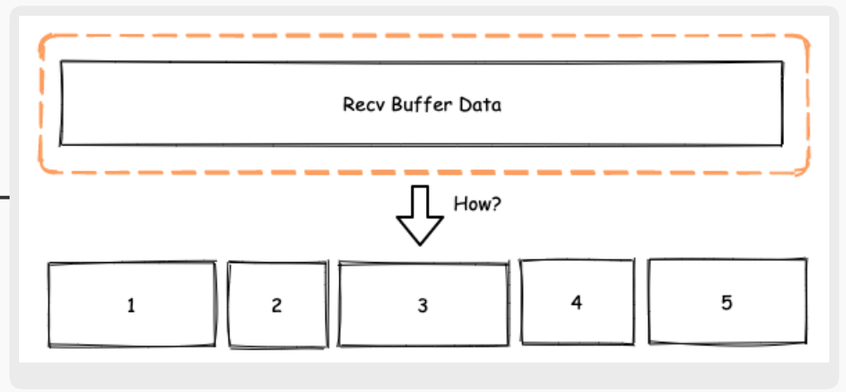


在一般的 Linux 操作系统下，发送缓冲区和接收缓冲区的的大小默认为208K

也就是说，假如进程 A 不从接收缓冲区读取任何数据，那么数据将会在接收缓冲区中持续堆积，知道缓冲区已满

数据在网络中是采用二进制的形式进行传输的，也就是说，接收缓冲区中的内容其实也是二进制数据。并且数据将会按照先后顺序堆在接收缓冲区中，就会出现所谓的 "粘包" 问题。"粘包" 这一词自个儿能理解就好，其实是一个伪概念。这就好像我们说从水龙头出来的水"粘"在一起一样，想必没人会这么描述

简单地来说，流式协议带来的一个问题就是，我们该如何区分接收缓冲区中的数据到底是属于哪一个 TCP 数据包？

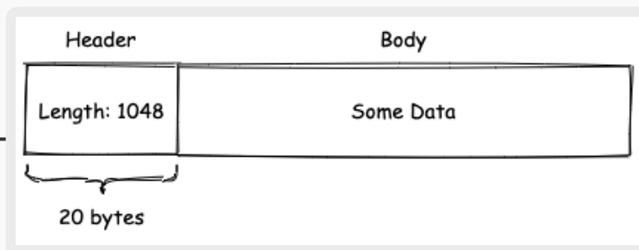


举一个简单的例子，客户端向服务端发送了 3 个 TCP 数据包，这 3 个数据包的内容分别为 QA、QB、QC，对应了 3 个不同的请求

那么 Recv Buffer 中的数据就可能是 QAQBQC，服务端需要将这一串儿数据拆解成 3 个数据包并分别进行处理

包头+包体

通常而言，我们都会使用包头+包体的方式来解决这个问题。其中包头长度固定，并且在包头中记录下此次数据的总大小，包体则是实际的数据传输



在读取 Recv Buffer 的内容时，首先读取包头，根据包头中的数据大小再读取包体

假设我们的包头大小固定为 20 字节，那么首先就去 Recv Buffer 中读取 20 个字节解析出 Header，并根据 Header.Length 决定再去缓冲区冲读取多少数据：1048 - 20 = 1028 bytes

一个需要注意的点是，Recv Buffer 中可能只包含某一次通讯的 Header，或者是 Header 的一部分，或者是 Header + Body 的一部分，因此，我们需要详细的记录下当前请求我们收到了哪些数据、还剩多少数据要收等信息

send

ssize_t send(int sockfd, const void *buffer, size_t length, int flags);

send() 函数专用于 scoket 上的 write 操作，本质上是向发送缓冲区中写入数据

内核在发送 TCP 数据时，通常会使用 Nagle 算法把多个小的数据包合并成一个发送给另一端，以提高效率