网络编程

TCP与UDP的区别

- TCP：传输控制协议,提供的是面向连接、可靠的字节流服务。当客户和服务器彼此交换数据前，必须先在双方之间建立一个TCP连接，之后才能传输数据。TCP提供超时重发，丢弃重复数据，检验数据，流量控制等功能，保证数据能从一端传到另一端。
- UDP：用户数据报协议，是一个简单的面向数据报的运输层协议。UDP不提供可靠性，它只是把应用程序传给IP层的数据报发送出去，但是并不能保证它们能到达目的地。由于UDP在传输数据报前不用在客户和服务器之间建立一个连接，且没有超时重发等机制，故而传输速度很快
- TCP与UDP选择：当数据传输的性能必须让位于数据传输的完整性、可控制性和可靠性时，TCP协议是当然的选择。当强调传输性能而不是传输的完整性时，如：音频和多媒体应用，UDP是最好的选择。在数据传输时间很短，以至于此前的连接过程成为整个流量主体的情况下，UDP也是一个好的选择，如：DNS交换。

TCP保证可靠性

- 确认与重传：接收方收到报文就会确认，发送方发送一段时间后没有收到确认就会重传。
- 数据校验：TCP报文头有校验和，用于校验报文是否损坏
- 数据合理分片与排序：TCP会按最大传输单元(MTU)合理分片，接收方会缓存未按序到达的数据，重新排序后交给应用层。
- 流量控制：当接收方来不及处理发送方的数据，能通过滑动窗口，提示发送方降低发送的速率，防止包丢失。
- 拥塞控制：当网络拥塞时，通过拥塞窗口，减少数据的发送，防止包丢失。

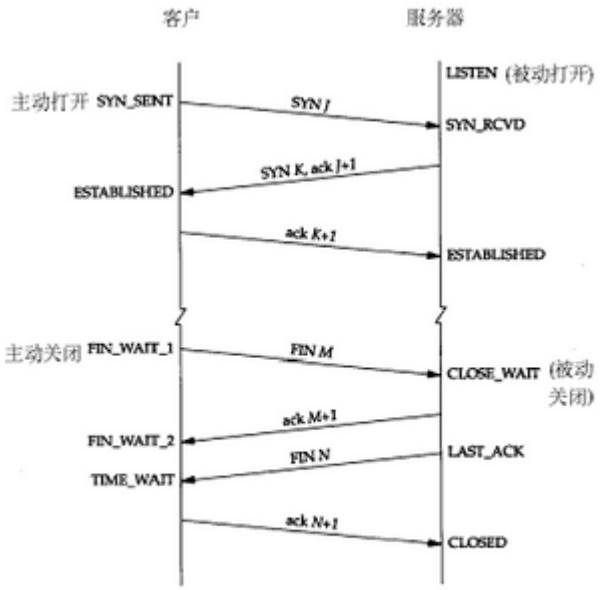
UPD实现可靠性

最简单的方式是在应用层模仿传输层TCP的可靠性传输，简单设计：

- 添加seq/ack机制，确保数据发送到对端。
- 添加发送与接收缓冲区，主要是用户超时重传。
- 添加超时重传机制

目前开源程序利用UDP实现了可靠传输分别是RUDP（实时音频传输协议）、RTP（实时传输协议）、UDT（支持高速广域网上的海量数据传输）

TCP 3次握手与4次挥手



- 三次握手的原因：

  A 发送同步信号**SYN** + **A's Initial sequence number**

  B 确认收到A的同步信号，并记录 A's ISN 到本地，命名 **B's ACK sequence number**

  B发送同步信号**SYN** + **B's Initial sequence number**

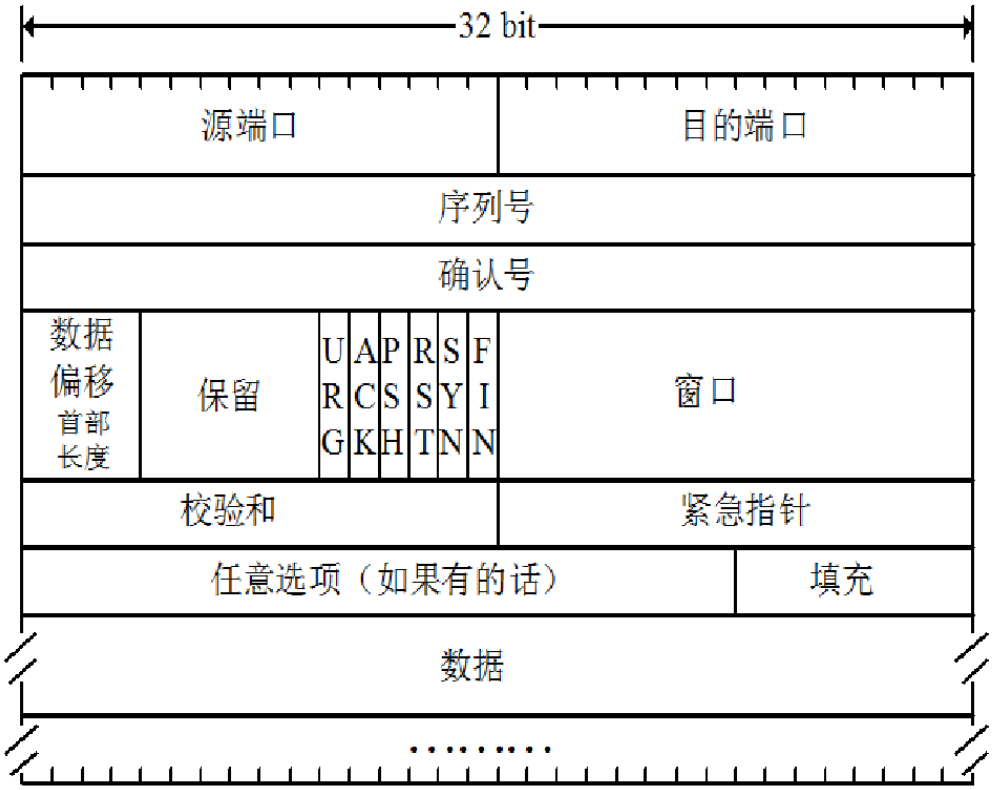
  A确认收到B的同步信号，并记录 B's ISN 到本地，命名 **A's ACK sequence number**

  显然步骤2和3 可以合并，**只需要三次握手，**可以提高连接的速度与效率。

  A 发送同步信号**SYN** + **A's Initial sequence number**

  B发送同步信号**SYN** + **B's Initial sequence number** + **B's ACK sequence number**

  问题：**B**无法知道**A**是否已经接收到自己的同步信号，如果丢失，**A**与**B**的初始序列号无法达成一致。
- 四次挥手 ： TCP全双工通信
  - 第一次挥手：当主动方发送断开连接的请求（即FIN报文）给被动方时，仅仅代表主动方不会再发送数据报文了，但主动方仍可以接收数据报文。
  - 第二次挥手：被动方此时有可能还有相应的数据报文需要发送，因此需要先发送ACK报文，告知主动方“我知道你想断开连接的请求了”。这样主动方便不会因为没有收到应答而继续发送断开连接的请求（即FIN报文）。
  - 第三次挥手：被动方在处理完数据报文后，便发送给主动方FIN报文；这样可以保证数据通信正常可靠地完成。发送完FIN报文后，被动方进入LAST_ACK阶段（超时等待）。
  - 第四挥手：如果主动方及时发送ACK报文进行连接中断的确认，这时被动方就直接释放连接，进入可用状态。
- TIME_WAIT状态需要经过2MSL(最大报文段生存时间)才能返回到CLOSE状态。
  - 可靠地实现TCP全双工连接的终止，服务端超时重传FIN，客户端能够重传ACK
  - 允许老的重复分节在网络中消逝。防止重新建立起相同的IP地址和端口之间的TCP对上一个连接分组的误解。



MTU、MSS

- **TTL**：IP协议包中的一个值，它告诉网络,数据包在网络中的时间是否太长而应被丢弃
- **MTU**： Maximum Transmit Unit，最大传输单元，即物理接口（数据链路层）提供给其上层（通常是IP层）最大一次传输数据的大小；以普遍使用的以太网接口为例，缺省MTU=1500 Byte，这是以太网接口对IP层的约束，如果IP层有<=1500 byte 需要发送，只需要一个IP包就可以完成发送任务；如果IP层有> 1500 byte 数据需要发送，需要分片才能完成发送，这些分片有一个共同点，即IP Header ID相同。
- **MSS**：Maximum Segment Size ，TCP提交给IP层最大分段大小，不包含TCP Header和 TCP Option，只包含TCP Payload ，MSS是TCP用来限制application层最大的发送字节数，是tcp能发送的分组的最大长度。 MSS是系统默认的，就是系统TCP/IP栈所能允许的最大包。在建立连接时，这个值已经被确定了，这个值并不是客观的值，而是由tcp/ip的实现确定的。

  如果底层物理接口MTU= 1500 byte，则 MSS = 1500- 20(IP Header) -20 (TCP Header) = 1460 byte，如果application 有2000 byte发送，需要两个segment才可以完成发送，第一个TCP segment = 1460，第二个TCP segment = 540。

  Window Size 占两个byte，最大值为65535

滑动窗口

- TCP使用滑动窗口来实现流量控制，该协议允许发送方在停止并等待确认前可以连续发送多个分组，避免了每发送一个分组就停下来等待确认，因此可以加速数据传输。发送方的窗口由接收方决定。滑动窗口存在死锁的可能，通常在发送端接收到0窗口值时，其会启动持续计时器，若超时则会发送零窗口探测报文段，而对方收到后则会回复现有的窗口值，若窗口值不为零，则死锁解除。
- TCP_CORK

所谓的CORK就是塞子的意思,形象地理解就是用CORK将连接塞住,使得数据先不发出去,等到拔去塞子后再发出去.设置该选项后,内核会尽力把小数据包拼接成一个大的数据包(一个MTU)再发送出去,当然若一定时间后(一般为200ms,该值尚待确认),内核仍然没有组合成一个MTU时也必须发送现有的数据(不可能让数据一直等待吧).

然而,TCP_CORK的实现可能并不像你想象的那么完美,CORK并不会将连接完全塞住.内核其实并不知道应用层到底什么时候会发送第二批数据用于和第一批数据拼接以达到MTU的大小,因此内核会给出一个时间限制,在该时间内没有拼接成一个大包(努力接近MTU)的话,内核就会无条件发送.也就是说若应用层程序发送小包数据的间隔不够短时,TCP_CORK就没有一点作用,反而失去了数据的实时性(每个小包数据都会延时一定时间再发送).

Nagle算法和CORK算法非常类似,但是它们的着眼点不一样,Nagle算法主要避免网络因为太多的小包(协议头的比例非常之大)而拥塞,而**CORK算法则是为了提高网络的利用率**,使得总体上协议头占用的比例尽可能的小.如此看来这**二者在避免发送小包上是一致的**,在用户控制的层面上,Nagle算法完全不受用户socket的控制,你只能简单的设置TCP_NODELAY而禁用它,CORK算法同样也是通过设置或者清除TCP_CORK使能或者禁用之,然而**Nagle算法关心的是网络拥塞问题**,只要所有的ACK回来则发包,而CORK算法却只关心内容,在前后数据包发送间隔很短的前提下(很重要,否则内核会帮你将分散的包发出),即使你是分散发送多个小数据包,你也可以通过使能CORK算法将这些内容拼接在一个包内,如果此时用Nagle算法的话,则可能做不到这一点.

- Nagle算法：由于TCP协议无论发送多少数据都需要加上头部协议，也需要ACK表示确认，为了尽可能的利用网络带宽，TCP总希望发送足够大的数据（尽量以MSS发送），Nagle算法就是为了尽可能发送大块数据，避免网络中充斥着许多小数据块。Nagle算法的基本定义是任意时刻,最多只能有一个未被确认的小段. 所谓"小段",指的是小于MSS尺寸的数据块,所谓"未被确认",是指一个数据块发送出去后,没有收到对方发送的ACK确认该数据已收到。Nagle算法的规则
  - 如果包长度达到MSS,则允许发送;
  - 如果该包含有FIN,则允许发送;
  - 设置了TCP_NODELAY选项,则允许发送;
  - 未设置TCP_CORK选项时,若所有发出去的小数据包(包长度小于MSS)均被确认,则允许发送;
  - 上述条件都未满足,但发生了超时(一般为200ms),则立即发送.

  Nagle算法只允许一个未被ACK的包存在于网络，它并不管包的大小，因此他就是一个扩展的停-等待协议，只不过它是基于包停-等的，这会带来一个问题，如果ACK回复过快，Nagle事实上不会拼接太多的数据包，虽然避免了网络拥塞，网络总体利用率依然很低。

  Nagle算法是silly window syndrome(SWS)预防算法的一个半集.SWS算法预防发送少量的数据，Nagle算法是其在发送方的实现,而接收方要做的是不要通告缓冲空间的很小增长,不通知小窗口,除非缓冲区空间有显著的增长.这里显著的增长定义为完全大小的段(MSS)或增长到大于最大窗口的一半.

TCP流量控制与拥塞控制

- **流量控制**：流量控制往往指的是点对点通信量的控制，是个端到端的问题。流量控制所要做的就是控制发送端发送数据的速率，以便使接收端来得及接受。

  措施： 利用滑动窗口机制可以很方便的在TCP连接上实现对发送方的流量控制。TCP的窗口单位是字节，不是报文段，发送方的发送窗口不能超过接收方给出的接收窗口的数值。

- **拥塞控制**：在某段时间，若对网络中某资源的需求超过了该资源所能提供的可用部分，网络的性能就要变坏。

  措施:

  - **慢开始**：在主机刚刚开始发送报文段时可先将拥塞窗口 cwnd 设置为一个最大报文段 MSS 的数值。在每收到一个对新的报文段的确认后，将拥塞窗口增加至多一个 MSS 的数值。用这样的方法逐步增大发送端的拥塞窗口 cwnd，可以使分组注入到网络的速率更加合理。每经过一个传输轮回，拥塞窗口(发送端)就加倍。
  - **拥塞避免**：让拥塞窗口缓慢增大，每经过一个往返时间就加1，而不是加倍，按线性规律缓慢增长。拥塞窗口大于慢开始门限，就执行拥塞避免算法。"乘法减小"：指不论在慢开始还是拥塞避免阶段，只要出现

超时重传就把慢开始门限值减半。"加分增大":指执行拥塞避免算法后，使拥塞窗口缓慢增大，以防止网络过早出现拥塞。合起来叫AIMD算法。

- **快重传算法**：发送方只要一连收到三个重复确认就应当重传对方尚未收到的报文。而不必等到该分组的重传计时器到期。
- **快恢复算法**：(1)当发送端收到连续三个重复的确认时，就执行"乘法减小"算法，把慢开始门限 ssthresh 减半。但接下去不执行慢开始算法。(2)由于发送方现在认为网络很可能没有发生拥塞，因此现在不执行慢开始算法，即拥塞窗口 cwnd 现在不设置为 1，而是设置为慢开始门限 ssthresh 减半后的数值，然后开始执行拥塞避免算法（"加法增大"），使拥塞窗口缓慢地线性增大.

epoll、select与poll原理

- epoll
  - 原理：在linux，一切皆文件．所以当调用epoll_create时，内核给这个epoll分配一个file，但是这个不是普通的文件，而是只服务于epoll．当内核初始化epoll时，会开辟一块内核高速cache区，用于安置我们监听的socket，这些socket会以红黑树的形式保存在内核的cache里，以支持快速的查找，插入，删除．同时，建立了一盒list链表，用于存储准备就绪的事件．所以调用epoll_wait时，在timeout时间内，只是简单的观察这个list链表是否有数据，如果没有，则睡眠至超时时间到返回；如果有数据，则在超时时间到，拷贝至用户态events数组中．当执行epoll_ctl时，除了把socket放到epoll文件系统里file对象对应的红黑树上之外，还会给内核中断处理程序注册一个回调函数，告诉内核，如果这个句柄的中断到了，就把它放到准备就绪list链表里。
  - ET与LT：epoll有两种模式LT(水平触发)和ET(边缘触发)，LT模式下，主要缓冲区数据一次没有处理完，那么下次epoll_wait返回时，还会返回这个句柄；而ET模式下，缓冲区数据一次没处理结束，那么下次是不会再通知了，只在第一次返回．所以在ET模式下，一般是通过while循环，一次性读完全部数据．epoll默认使用的是
- ET与LT

```
            if (epi->event.events & EPOLLONESHOT)
                epi->event.events &= EP_PRIVATE_BITS;
            else if (!(epi->event.events & EPOLLET)) {
                /*
                 * If this file has been added with Level
                 * Trigger mode, we need to insert back inside
                 * the ready list, so that the next call to
                 * epoll_wait() will check again the events
                 * availability. At this point, no one can insert
                 * into ep->rdllist besides us. The epoll_ctl()
                 * callers are locked out by
                 * ep_scan_ready_list() holding "mtx" and the
                 * poll callback will queue them in ep->ovflist.
                 */
                list_add_tail(&epi->rdllink, &ep->rdllist);
                ep_pm_stay_awake(epi);
            }
```

对于LT模式，上一次的就绪链表，还会添加到这次的就绪链表中，进行poll检验。

- epoll与select/poll对比
  - 首先select/poll（poll只是采用链表的方式突破select文件描述符的限制）监听的文件描述符个数受限．select的文件描述符默认为2048，而现在的服务器连接数在轻轻松松就超过2048个；epoll支持的fd个数不

受限制，它支持的fd上限是最大可以打开文件的数目，一般远大于2048，1G内存的机器上是大约10万左右。
    - select和poll需要循环检测所有fd是否就绪，当fd数量百万或者更多时，这是很耗时的，根据前面原理分析可知，epoll只处理就绪的fd，而一般一次epoll_wait返回时，就绪的fd是不多的，所以处理起来不是很耗时。
    - epoll使用共享内存的方式来复制文件描述符，避免每次都需要将文件描述符从用户态拷贝到内核态，而select/poll需要每次拷贝；
    - epoll支持内核微调

ICMP、Ping

- ICMP网络控制消息协议，工作在网络层，用于在IP主机、路由器之间传递控制消息。控制消息是指网络通不通、主机是否可达、路由是否可用等网络本身的消息。ICMP靠IP协议来完成任务，所以ICMP报文中要封装IP头部。一个ICMP报文包括IP报头（至少20字节）、ICMP报头（至少八字节）和ICMP报文，常见ICMP的报文类型：询问类型和差错报告报文。
- Ping：ICMP的一个典型应用是Ping。Ping是检测网络连通性的常用工具，同时也能够收集其他相关信息。用户可以在Ping命令中指定不同参数，如ICMP报文长度、发送的ICMP报文个数、等待回复响应的超时时间等，设备根据配置的参数来构造并发送ICMP报文，进行Ping测试。
- Traceroute:ICMP的另一个典型应用是Traceroute。Traceroute基于报文头中的TTL值来逐跳跟踪报文的转发路径。为了跟踪到达某特定目的地址的路径，源端首先将报文的TTL值设置为1。该报文到达第一个节点后，TTL超时，于是该节点向源端发送TTL超时消息，消息中携带时间戳。然后源端将报文的TTL值设置为2，报文到达第二个节点后超时，该节点同样返回TTL超时消息，以此类推，直到报文到达目的地。这样，源端根据返回的报文中的信息可以跟踪到报文经过的每一个节点，并根据时间戳信息计算往返时间。Traceroute是检测网络丢包及时延的有效手段，同时可以帮助管理员发现网络中的路由环路。

长链接与短连接

- TCP长链接：每次通信完毕后，不会关闭连接，这样可以做到连接复用，省去了创建连接的耗时。
- TCP短连接：每次通信完毕后就关闭连接。
- TCP keepAlive：在一定时间内（一般时间为7200s，参数tcp_keepalive_time）在链路上没有数据传送的情况下，TCP层将发送相应的KeepAlive探针以确定连接可用性，探测失败后重试10（参数tcp_keepalive_probes）次，每次间隔时间75s（参数tcp_keepalive_intvl），所有探测失败后，才认为当前连接已经不可用。这些参数是机器级别，可以调整。
- 应用层心跳包：TCP keepAlive的保活机制是机器级别的，调整起来不太方便，维护成本过高，其次KeepAlive的保活机制只在链路空闲的情况下才会起到作用，若数据发送时，物理链路已经不通，自然会走TCP重传机制，但要知道TCP超时重传，指数退避算法是一个相当长的过程，因此一个可靠的系统，长链接的保活肯定需要依赖应用层的心跳来保证。
- 应用层设计：1、无心跳设计比较简单，但是处理异常与重连比较迟钝；2、必须由发起连接方来进行心跳检测，因为只有发起连接的一端检测心跳，知道链路有问题，这时才会去断开连接，进行重连，或者重连到另一台服务器。
- 总结：三种使用 KeepAlive 的实践方案
    - 默认情况下使用 KeepAlive 周期为 2 个小时，如不选择更改，属于误用范畴，造成资源浪费：内核会为每一个连接都打开一个保活计时器，N 个连接会打开 N 个保活计时器。 优势很明显：
        - TCP 协议层面保活探测机制，系统内核完全替上层应用自动给做好了
        - 内核层面计时器相比上层应用，更为高效
        - 上层应用只需要处理数据收发、连接异常通知即可
        - 数据包将更为紧凑

- 关闭 TCP 的 KeepAlive，完全使用应用层心跳保活机制。由应用掌管心跳，更灵活可控，比如可以在应用级别设置心跳周期，适配私有协议。
- 业务心跳 + TCP KeepAlive 一起使用，互相作为补充，但 TCP 保活探测周期和应用的心跳周期要协调，以互补方可，不能够差距过大，否则将达不到设想的效果。

在浏览器中输入URL后执行的全部过程

- 域名解析
  - 查询浏览器DNS缓存
  - 查询操作系统DNS缓存
  - 操作系统将域名发送至本地域名服务器，由本地域名服务器进行递归或者迭代DNS查询
- TCP连接建立
- HTTP请求
- 服务器收到请求并响应HTTP请求
- 浏览器解析HTML代码，并请求HTML代码中的资源（如js、css图片）
- 断开TCP连接
- 浏览器对页面进行渲染呈现给用户

TCP粘包与分包

- 原因：
  - 要发送的数据大于TCP发送缓冲区剩余空间大小，将会发生拆包。
  - 待发送数据大于MSS（最大报文长度），TCP在传输前将进行拆包。
  - 要发送的数据小于TCP发送缓冲区的大小，TCP将多次写入缓冲区的数据一次发送出去，将会发生粘包。
  - 接收数据端的应用层没有及时读取接收缓冲区中的数据，将发生粘包。
- 解决方法：
  - 发送端给每个数据包添加包首部，首部中应该至少包含数据包的长度
  - 发送端将每个数据包封装为固定长度
  - 可以在数据包之间设置边界

TCP延时关闭SO_LINGER

```
struct linger {
     int l_onoff;
     int l_linger;
};
```

- 设置 **l_onoff**为0，则该选项关闭，**l_linger**的值被忽略，等于内核缺省情况，**close**调用会立即返回给调用者，如果可能将会传输任何未发送的数据；
- 设置*** l_onoff**为非0，**l_linger**为0，则套接口关闭时TCP夭折连接，TCP将丢弃保留在套接口发送缓冲区中的任何数据并发送一个RST给对方，而不是通常的四分组终止序列，这避免了TIME_WAIT状态；
- 设置**l_onoff** 为非0，**l_linger**为非0，当套接口关闭时内核将拖延一段时间（由**l_linger**决定）。如果套接口缓冲区中仍残留数据，进程将处于睡眠状态，直 到（a）所有数据发送完且被对方确认，之后进行正常的终止序列（描述字访问计数为0）或（b）延迟时间到。此种情况下，应用程序检查close的返回值是非常重要的，如果在数据发送完并被确认前时间到，close将返回EWOULDBLOCK错误且套接口发送缓冲区中的任何数据都丢失。close的成功返回仅告诉我们发送的数据（和FIN）已由对方TCP确认，它并不能告诉我们对方应用进程是否已读了数据。如果套接口设为非阻塞的，它将不等待close完成。

IO模型（前4为同步模型）

- 阻塞IO
- 非阻塞IO
- 多路复用IO
- 信号驱动式IO
- 异步IO

阻塞与非阻塞表示函数调用的方式，而同步与异步表示代码过程与系统的交互方式。

端口数与连接数

TCP端口最大值为65535，但是标识一个TCP连接是由IP与Port这个四元组来决定的，所以TCP最大的连接数是由linux系统最大可打开文件描述符限制，当然最大可打开文件数可以通过修改系统配置文件修改，所以最终最大连接数的限制来自于系统的资源限制。

数据从网卡到内核

https://blog.csdn.net/lishanmin11/article/details/77162070

Linux网络 - 数据包的接收过程

网卡到内存

1) 数据包从外面的网络进入物理网卡。如果目的地址不是该网卡，且该网卡没有开启混杂模式，该包会被网卡丢弃。

2) 网卡将数据包通过DMA的方式写入到指定的内存地址，该地址由网卡驱动分配并初始化。注： 老的网卡可能不支持DMA，不过新的网卡一般都支持。

3) 网卡通过硬件中断（IRQ）通知CPU，告诉它有数据来了

4) CPU根据中断表，调用已经注册的中断函数，这个中断函数会调到驱动程序（NIC Driver）中相应的函数

5) 驱动先禁用网卡的中断，表示驱动程序已经知道内存中有数据了，告诉网卡下次再收到数据包直接写内存就可以了，不要再通知CPU了，这样可以提高效率，避免CPU不停的被中断。

6) 启动软中断。这步结束后，硬件中断处理函数就结束返回了。由于硬中断处理程序执行的过程中不能被中断，所以如果它执行时间过长，会导致CPU没法响应其它硬件的中断，于是内核引入软中断，这样可以将硬中断处理函数中耗时的部分移到软中断处理函数里面来慢慢处理。

内核的网络模块

1) 内核中的ksoftirqd进程专门负责软中断的处理，当它收到软中断后，就会调用相应软中断所对应的处理函数，对于上面第6步中是网卡驱动模块抛出的软中断，ksoftirqd会调用网络模块的net_rx_action函数。

2) net_rx_action调用网卡驱动里的poll函数来一个一个的处理数据包

3) 在pool函数中，驱动会一个接一个的读取网卡写到内存中的数据包，内存中数据包的格式只有驱动知道

4) 驱动程序将内存中的数据包转换成内核网络模块能识别的skb格式，然后调用napi_gro_receive函数

5) napi_gro_receive会处理GRO相关的内容，也就是将可以合并的数据包进行合并，这样就只需要调用一次协议栈。然后判断是否开启了RPS，如果开启了，将会调用enqueue_to_backlog

6) 在enqueue_to_backlog函数中，会将数据包放入CPU的softnet_data结构体的input_pkt_queue中，然后返回，如果input_pkt_queue满了的话，该数据包将会被丢弃，queue的大小可以通过net.core.netdev_max_backlog来配置

7) CPU会接着在自己的软中断上下文中处理自己input_pkt_queue里的网络数据（调用__netif_receive_skb_core）

8) 如果没开启RPS，napi_gro_receive会直接调用__netif_receive_skb_core

9) 看是不是有AF_PACKET类型的socket（也就是我们常说的原始套接字），如果有的话，拷贝一份数据给它。tcpdump抓包就是抓的这里的包。

10) 调用协议栈相应的函数，将数据包交给协议栈处理。

11) 待内存中的所有数据包被处理完成后（即poll函数执行完成），启用网卡的硬中断，这样下次网卡再收到数据的时候就会通知CPU