Nginx 从入门到企业实战（二）

Nginx 调优与背后原理

1. Linux 网络 IO 流程

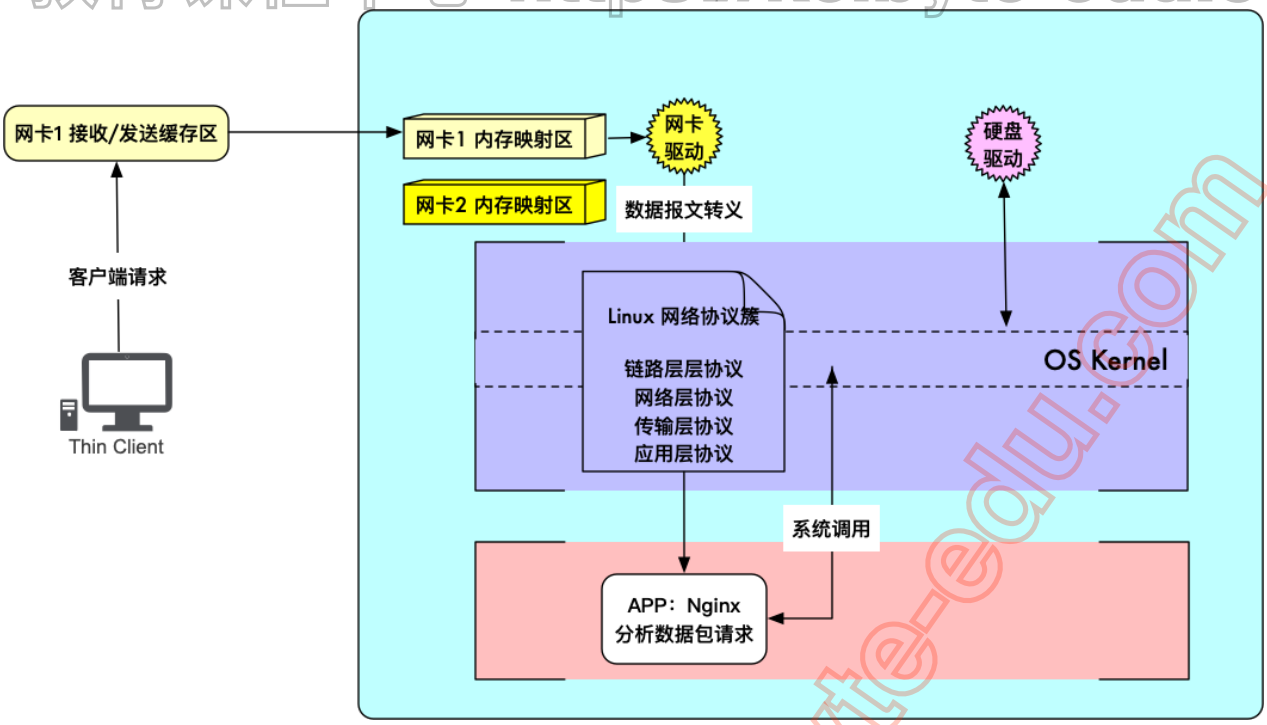
我们访问一个网站，从我们在浏览器上打开输入网址到网页被打开，中间经历的哪些流程？如果网络比较好的情况下，通常在几秒之内就可以打开一个内容非常丰富的网站，但是背后的执行过程是非常复杂的，我们简单对其中的某些过程进行描述。而我们要描述的地方就是从服务器网卡接收到用户请求数据开始，如何处理这些请求并将报文响应给用户。

这个过程我们省略了从用户发起请求一直到请求到达服务端的过程，这中间还涉及到 `DNS` 解析、路由查找等技术细节。

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，并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。
3. 网卡通过硬件中断（ `IRQ` ）告知 `CPU` 有数据来了。
4. `CPU` 根据中断表，调用已经注册的中断函数，这个中断函数会调动驱动程序中相应的函数对数据进行处理。
5. 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。
6. 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格式，数据包会被先存入到 `CPU` 的 `input_pkt_queue` 队列中，如果队列满了数据会被丢弃（注意：我们可以通过调整系统参数 `net.core.netdev_max_backlog` 来调整队列大小）。
7. 此时 `CPU` 从队列中获取数据并 `Linux` 网络协议栈相应的函数，将数据包交给协议栈处理。
8. 通过协议分析将数据包传给链路层，此时放在该协议栈中的 `netfilter` 钩子函数（也就是咱们通过 `iptables` ） 设置的一些规则会对数据包进行处理，可以进行修改，也可能会进行丢弃。如果没有丢弃，则会继续交由网络层协议。
9. 网络层协议对报文进行处理，查看 `IP` 地址是否为本机，为本机则接收。如果不为本机，要看是否开启了 `ip_forward` 即 `IP` 转发功能，如果开启则进行网络报文转发，如果未开启则丢弃报文。假设报文的目的地址是本机，则会交由传输层协议进行处理，比如 `TCP` 或者 `UDP` 等。
10. 传输层协议会根据数据包中的 `IP` 和 端口来查找相应的 `socket` ，如果没有找到则丢弃该报文，如果找到则将数据放到 `socket` 的队列中，然后由操作系统内核通知相关的应用来进行数据读取。
11. 应用层将请求报文打开后，假如请求读取某个文件，则应用会通过系统调用来读取该文件。
12. 当内核完成数据的处理后，会将数据复制到应用内存中供应用读取。
13. 应用读取到数据，构建响应包，通过网络协议簇将报文进行封装，通过链路层发送到网卡缓冲区队列中。
14. 网卡再次通过中断函数调用网卡驱动完成数据的发送。

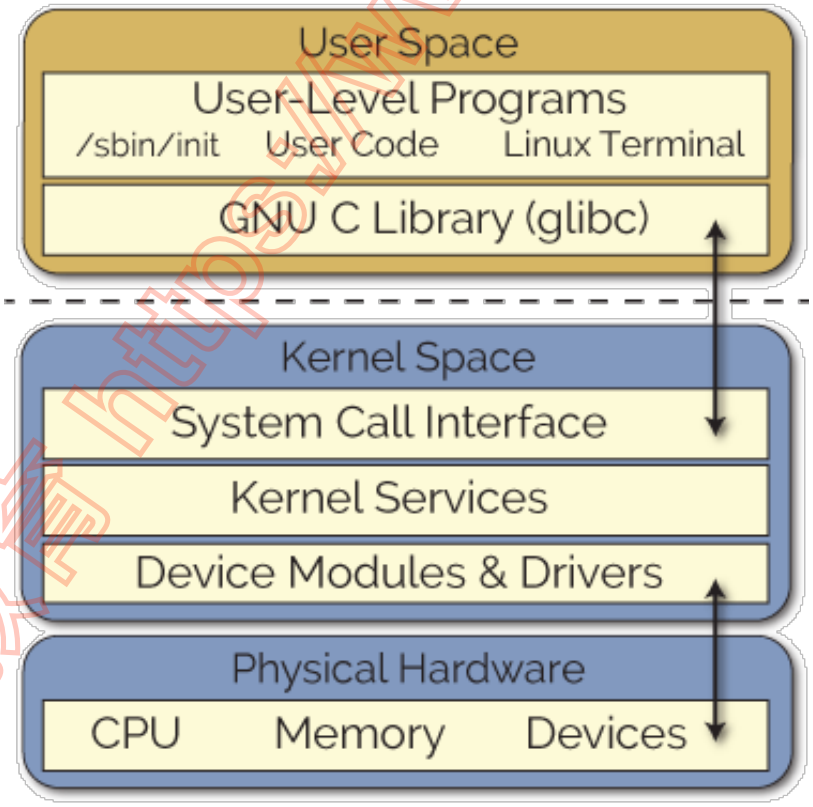
注意：上述步骤知识进行了简单描述，很多细节均没有提及，同时也省略了很多和协议以及硬件有关的描述。

详细流程可参考 linux网络之数据包的接收过程



2. Linux 网络 IO 模型

上述内容即为 `Linux` 最简单的网络流程描述，或许你觉得已经很复杂了，但是下面我们介绍的内容只是整个流程中的很小很小一部分，也是咱们图中一笔带过的 `nginx` 对 `OS Kernel` 使用系统调用的部分。即：当应用接收到用户请求时，如何对请求进行处理，这里就设计到 `Linux` 网络 `IO` 模型。先来简单回一下用户空间和内核空间的关系：



即当用户空间的应用程序需要读写硬盘上的数据时，会通过系统调用的方式让内核去处理，此时进入内核空间，内核去将数据从硬盘读取到自己的内存缓冲区中，然后进行数据的读写处理。将数据准备好之后，会把数据拷贝到应用程序的内存缓冲区，此时进入用户空间由应用程序再对数据进行处理，比如数据封包等。在这里会涉及到两个阶段：

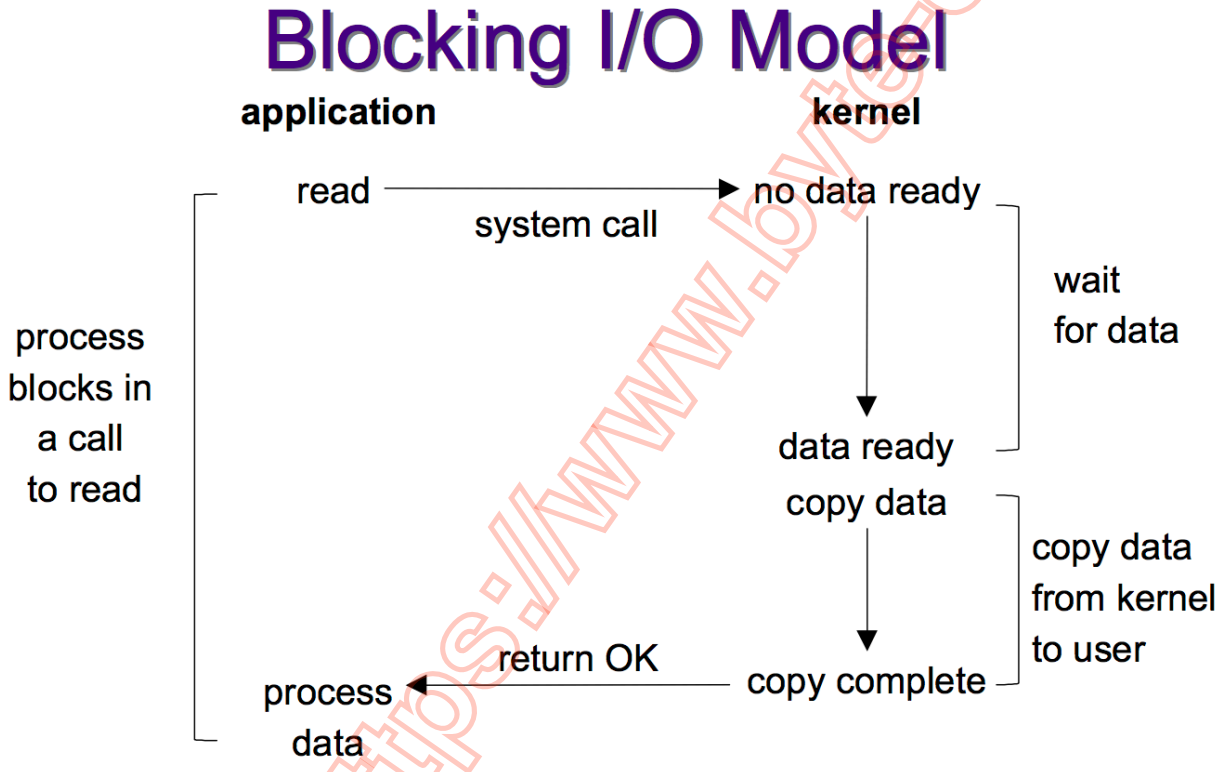
- **等待数据准备就绪 (Waiting for the data to be ready)**
- **将数据从内核缓冲区拷贝到进程缓冲区中 (Copying the data from the kernel to the process)**

正式因为这两个阶段，linux系统产生了下面五种网络模式的方案：

- **阻塞式 IO 模型 (blocking IO model)**
- **非阻塞式 IO 模型 (noblocking IO model)**
- **IO 复用式 IO 模型 (IO multiplexing model)**
- **信号驱动式 IO 模型 (signal-driven IO model)**
- **异步 IO 式 IO 模型 (asynchronous IO model)**

2.1 阻塞式IO模型 (blocking IO model)

在 `linux` 中，默认情况下所有的 IO 操作都是 `blocking`，一个典型的读操作流程大概是这样：



当用户进程调用了 `recvfrom` 这个系统调用，`kernel` 就开始了 `IO` 的第一个阶段：准备数据（对于网络IO来说，很多时候数据在一开始还没有到达。比如，还没有收到一个完整的 `UDP` 包。这个时候 `kernel` 就要等待足够的数据到来），而数据被拷贝到操作系统内核的缓冲区中是需要一个过程的，这个过程需要等待。此时用户进程会被阻塞（当然，是进程自己选择的阻塞）。当 `kernel` 一直等到数据准备好了，它就会将数据从 `kernel` 中拷贝到用户空间的缓冲区以后，然后 `kernel` 返回结果，用户进程才解除 `block` 的状态，重新运行起来。

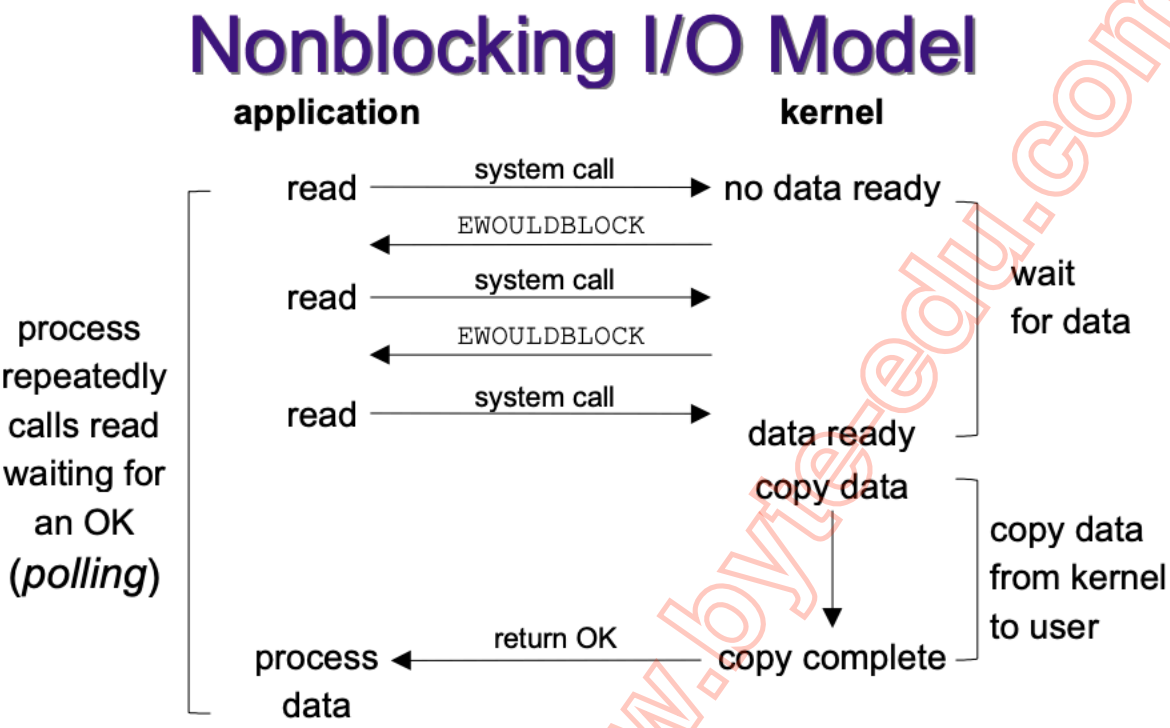
所以：**`blocking IO` 的特点就是在 IO 执行的下两个阶段的时候都被 block 了。**

- **等待数据准备就绪 (Waiting for the data to be ready)** 「阻塞」
- **将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)** 「阻塞」

2.2 非阻塞 I/O（nonblocking IO）

在 `linux` 中，可以通过设置 `socket` 使其变为 `non-blocking`。

`Socket` 设置为 `NONBLOCK`（非阻塞）就是告诉内核，当所请求的 `I/O` 操作无法完成时，不要将进程睡眠，而是返回一个错误码（`EWOULDBLOCK`），这样请求就不会阻塞。当对一个 `non-blocking socket` 执行读操作时，流程是这个样子：



当用户进程调用了 `recvfrom` 这个系统调用，如果 `kernel` 中的数据还没有准备好，那么它并不会 `block` 用户进程，而是立刻返回一个 `EWOULDBLOCK` 错误代码。从用户进程角度讲，它发起一个 `read` 操作后，并不需要等待，而是马上就得到了一个结果。用户进程判断结果是一个 `EWOULDBLOCK` 时，它就知道数据还没有准备好，此时用户进程可以进行其他操作。一旦 `kernel` 中的数据准备好了，并且又再次收到了用户进程的 `system call`，那么它马上就将数据拷贝到了用户空间缓冲区，然后返回。

可以看到，`I/O` 操作函数将不断的测试数据是否已经准备好，如果没有准备好，继续轮询，直到数据准备好为止。整个 `I/O` 请求的过程中，虽然用户线程每次发起 `I/O` 请求后可以立即返回，但是为了等到数据，仍需要不断地轮询、重复请求，消耗了大量的 `CPU` 的资源。

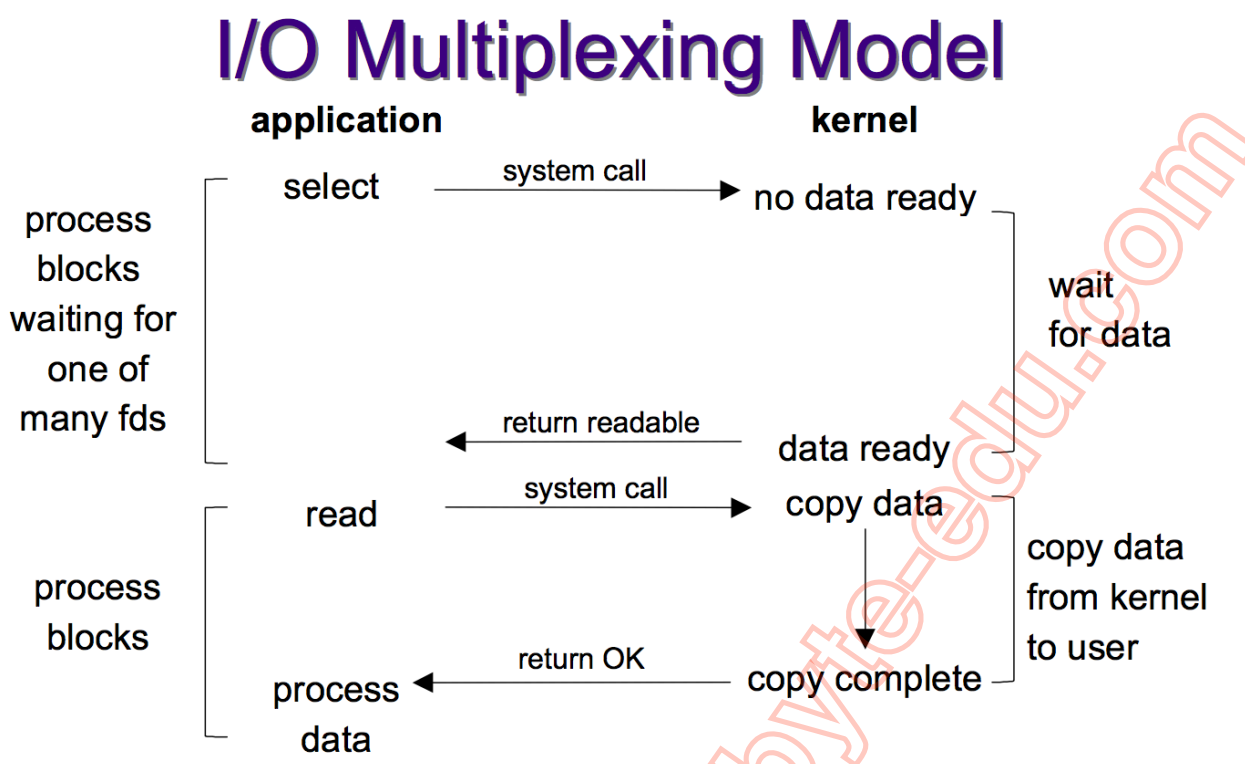
所以，**non blocking IO 的特点是用户进程需要不断的主动询问 kernel 数据好了没有**：

- **等待数据准备就绪 (Waiting for the data to be ready)** 「非阻塞」
- **将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)** 「阻塞」

一般很少直接使用这种模型，而是在其他 `I/O` 模型中使用非阻塞 `I/O` 这一特性。这种方式对单个 `I/O` 请求意义不大,但给 `I/O` 多路复用铺平了道路.

2.3 I/O 多路复用（IO multiplexing）

`IO multiplexing` 在我们使用中基于这种模型的代表就是常见的 `select, poll, epoll`，有些地方也称这种 `IO` 方式为事件驱动模型 `event driven IO`。`select/poll/epoll` 的好处就在于单个系统 `process` 就可以同时处理多个网络连接的 `IO`。它的基本原理就是 `select, poll, epoll` 这些个 `function` 会不断的轮询所负责的所有 `socket`，当某个 `socket` 有数据到达了，就通知用户进程。



当用户进程调用了 `select` ，那么整个进程会被 `block` ，而同时， `kernel` 会“监视”所有 `select` 负责的 `socket` ，当任何一个 `socket` 中的数据准备好了， `select` 就会返回。这个时候用户进程再调用 `read` 操作，将数据从 `kernel` 拷贝到用户进程。

所以，**I/O 多路复用的特点是通过一种机制一个进程能同时等待多个文件描述符，而这些文件描述符（套接字描述符）其中的任意一个进入读就绪状态，`select()` 函数就可以返回。**

这个图和 `blocking IO` 的图其实并没有太大的不同，事实上因为 `IO` 多路复用多了添加监视 `socket` ，以及调用 `select` 函数的额外操作，效率更差。还更差一些。因为这里需要使用两个 `system call` (`select` 和 `recvfrom`)，而 `blocking IO` 只调用了一个 `system call` (`recvfrom`)。但是，但是，使用 `select` 以后最大的优势是用户可以在一个线程内同时处理多个 `socket` 的 `I/O` 请求。用户可以注册多个 `socket` ，然后不断地调用 `select` 读取被激活的 `socket` ，即可达到在同一个线程内同时处理多个 `I/O` 请求的目的。而在同步阻塞模型中，必须通过多线程的方式才能达到这个目的。

所以，如果处理的连接数不是很高的话，使用 `select/epoll` 的web server 并没有性能优势，可能延迟还更大。 `select/epoll` 的优势并不是对于单个连接能处理得更快，而是在于能处理更多的连接。）

在 `IO multiplexing Model` 中，实际中，对于每一个 `socket` ，一般都设置成为 `non-blocking` ，但是，如上图所示，整个用户的 `process` 其实是一直被 `block` 的。只不过 `process` 是被 `select` 这个函数 `block` ，而不是被 `socket IO` 给 `block` 。

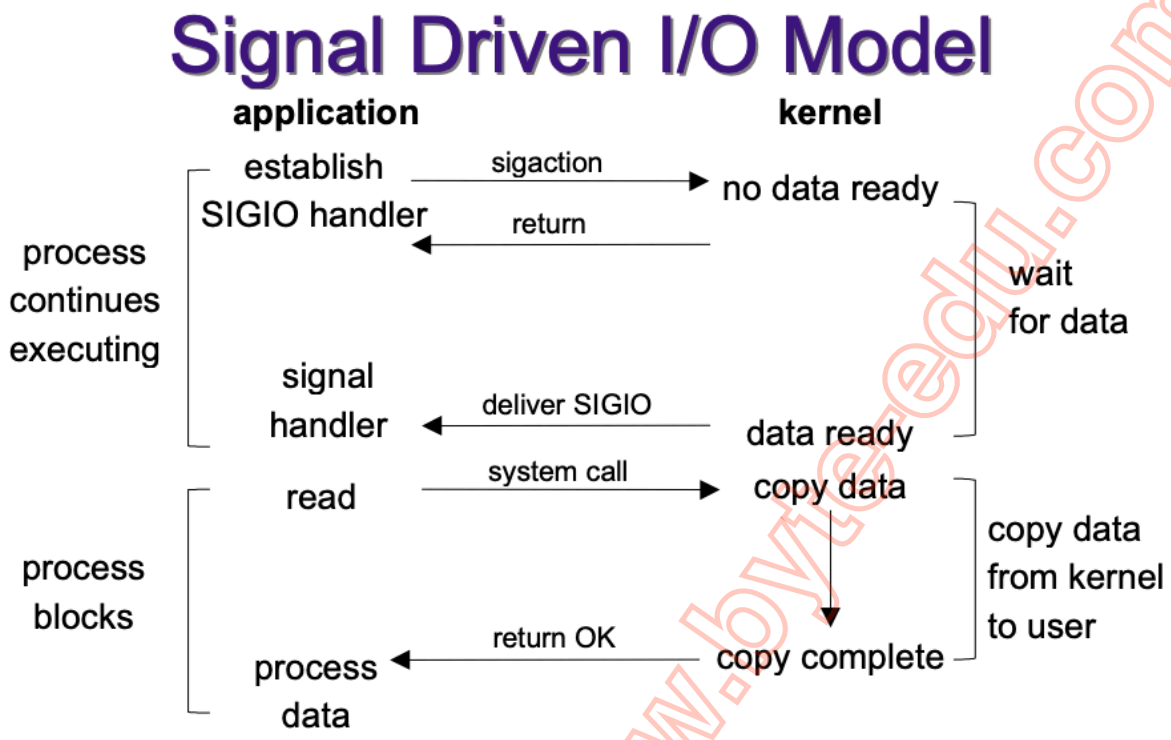
因此对于 `IO` 多路复用模型来说：

- **等待数据准备就绪 (Waiting for the data to be ready)** 「阻塞」
- **将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)** 「阻塞」

> 对于 `select/poll/epoll` 可以根据调用时指定的参数不同，而决定是阻塞或者非阻塞。

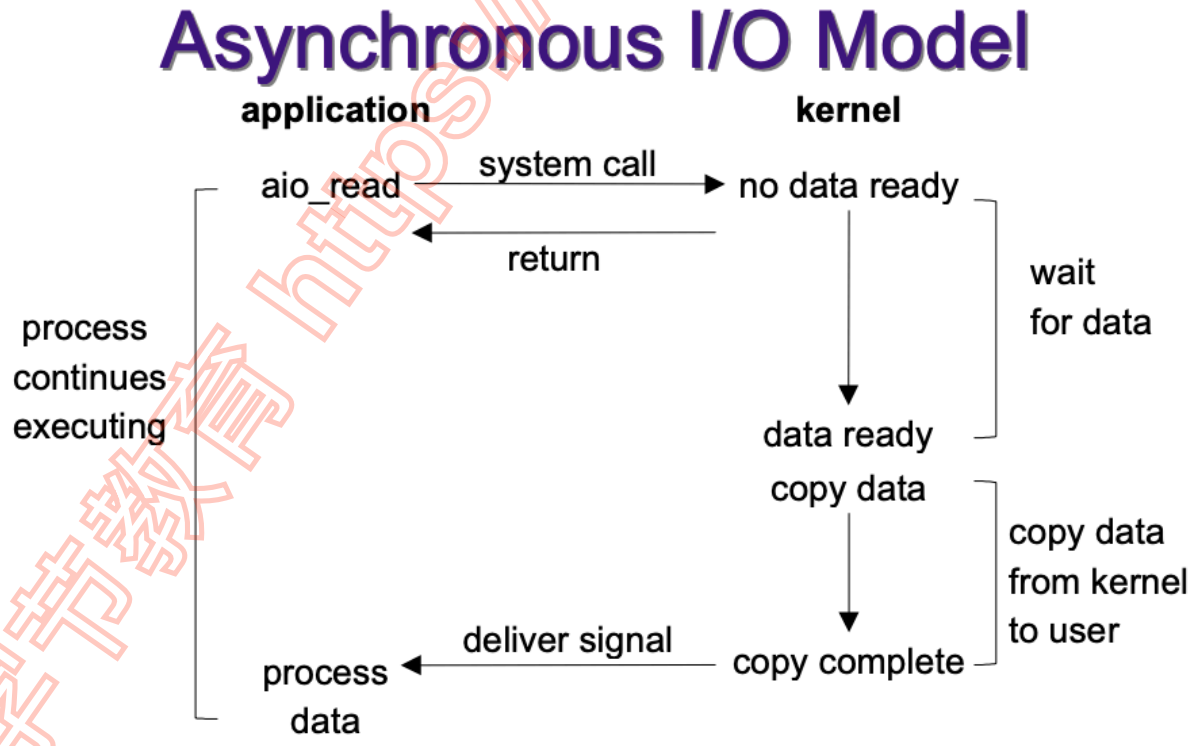
2.4 信号驱动式 IO 模型 (signal-driven IO model)

首先我们允许 `socket` 进行信号驱动 `I/O`,并安装一个信号处理函数，进程继续运行并不阻塞。当数据准备好时，进程会收到一个 `SIGIO` 信号，可以在信号处理函数中调用 `I/O` 操作函数处理数据。



2.5 异步 I/O（asynchronous IO）

接下来我们看看 `linux` 下的 `asynchronous IO` 的流程：



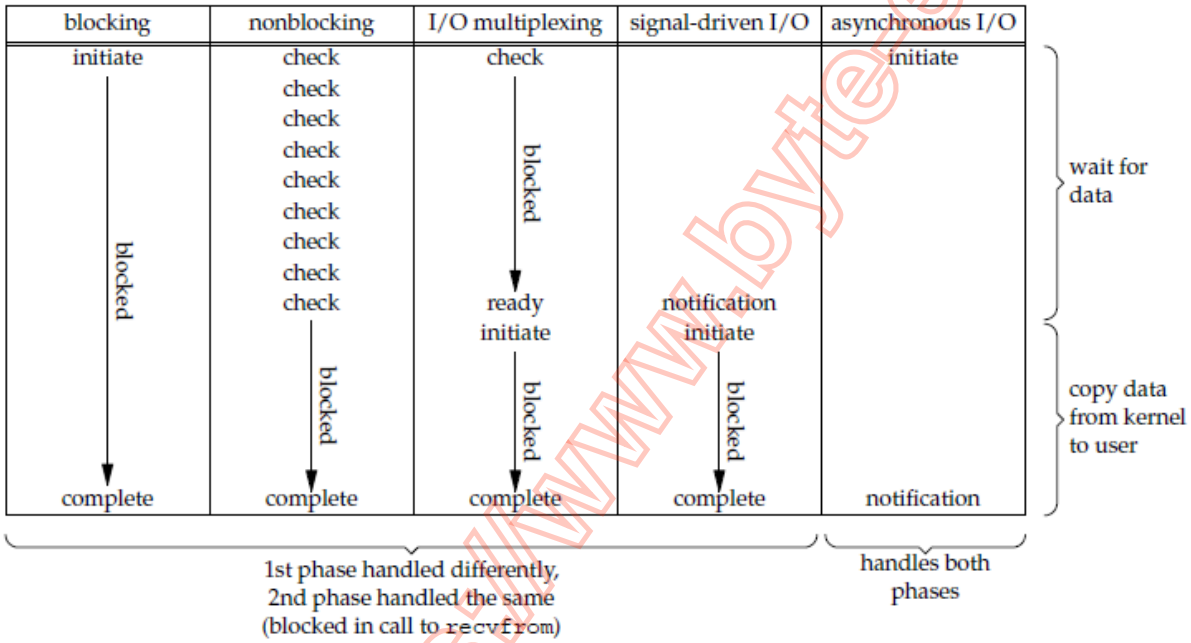
用户进程发起 `aio_read` 调用之后，立刻就可以开始去做其它的事。而另一方面，从 `kernel` 的角度，当它发现一个 `asynchronous read` 之后，首先它会立刻返回，所以不会对用户进程产生任何 `block`。然后，`kernel` 会等待数据准备完成，然后将数据拷贝到用户内存，当这一切都完成之后，`kernel` 会给用户进程发送一个 `signal`，告诉它 `read` 操作完成了。

异步 `I/O` 模型使用了 `Proactor` 设计模式实现了这一机制。

因此对异步 `IO` 模型来说：

- **等待数据准备就绪 (Waiting for the data to be ready)** 「非阻塞」
- **将数据从内核拷贝到进程中 (Copying the data from the kernel to the process)** 「非阻塞」

2.6 各个IO Model的比较



前四种模型的区别是阶段1不相同，阶段2基本相同（都是将数据从内核拷贝到调用者的缓冲区）。而异步 `I/O` 的两个阶段都不同于前四个模型。同步 `I/O` 操作引起请求进程阻塞，直到 `I/O` 操作完成。异步 `I/O` 操作不引起请求进程阻塞。

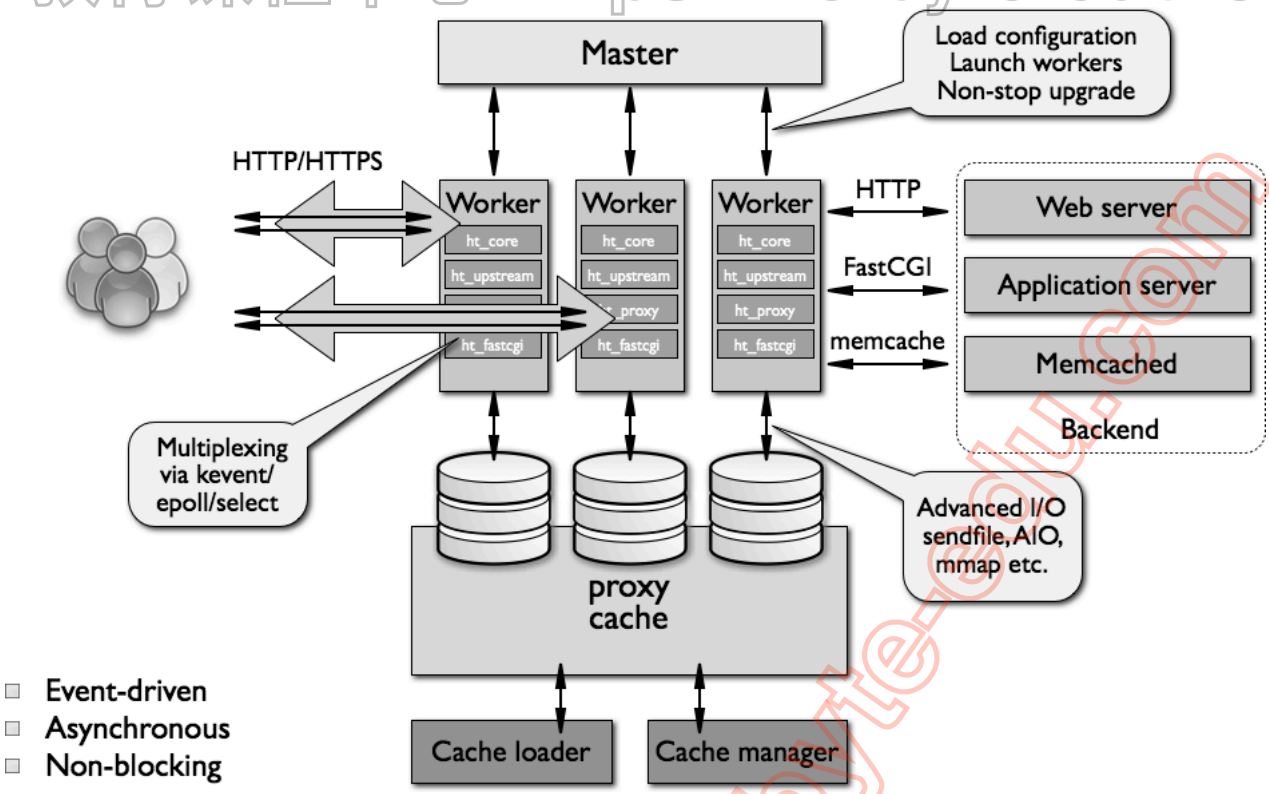
同时通过上面的图片，可以发现 `non-blocking IO` 和 `asynchronous IO` 的区别还是很明显的。在 `non-blocking IO` 中，虽然进程大部分时间都不会被 `block`，但是它仍然要求进程去主动的 `check`，并且当数据准备完成以后，也需要进程主动的再次调用 `recvfrom` 来将数据拷贝到用户内存。而 `asynchronous IO` 则完全不同。它就像是用户进程将整个 `IO` 操作交给了他人（`kernel`）完成，然后他人做完后发信号通知。在此期间，用户进程不需要去检查 `IO` 操作的状态，也不需要主动的去拷贝数据

有兴趣的朋友，可以再去读一下

Linux下的I/O复用与epoll详解

Linux Network IO Model

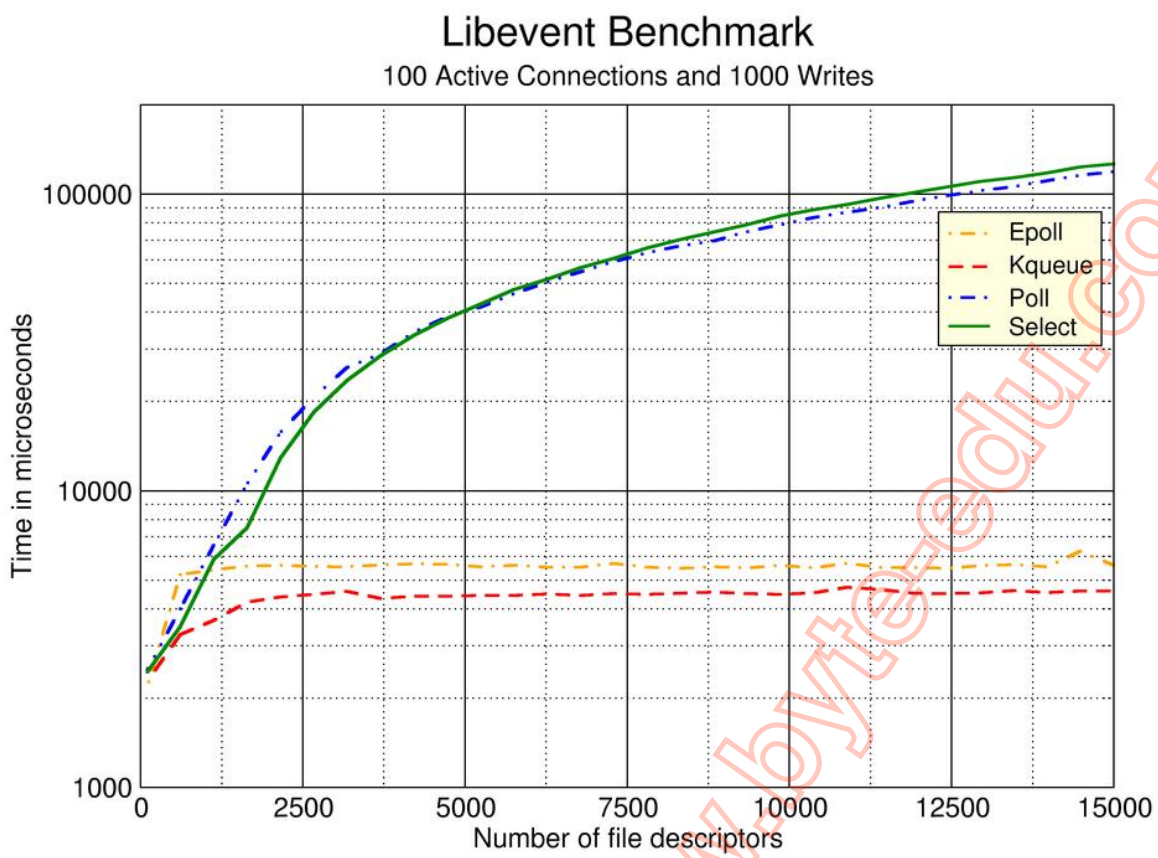
大话 Select、Poll、Epoll



下面三种 `IO` 多路复用的对比，可以看到 `epoll` 模型的性能比 `select/poll` 要高很多。

系统调用	select	poll	epoll
操作方式	遍历	遍历	回调
底层实现	数组 `bitmap`	链表	哈希表
查找就绪 `fd` 时间复杂度	O(n)	O(n)	O(1)
最大支持文件描述符数	一般有最大值限制	无上限	65535
工作模式	LT	LT	支持 ET 高效模式
`fd` 拷贝	每次调用 `select` 都需要把 `fd` 集合从用户态拷贝到内核态	每次调用 `poll` 都需要把 `fd` 集合从用户态拷贝到内核态	采用回调方式检测就绪事件，时间复杂度：O(1)

下面是不同网络模型的性能对比：



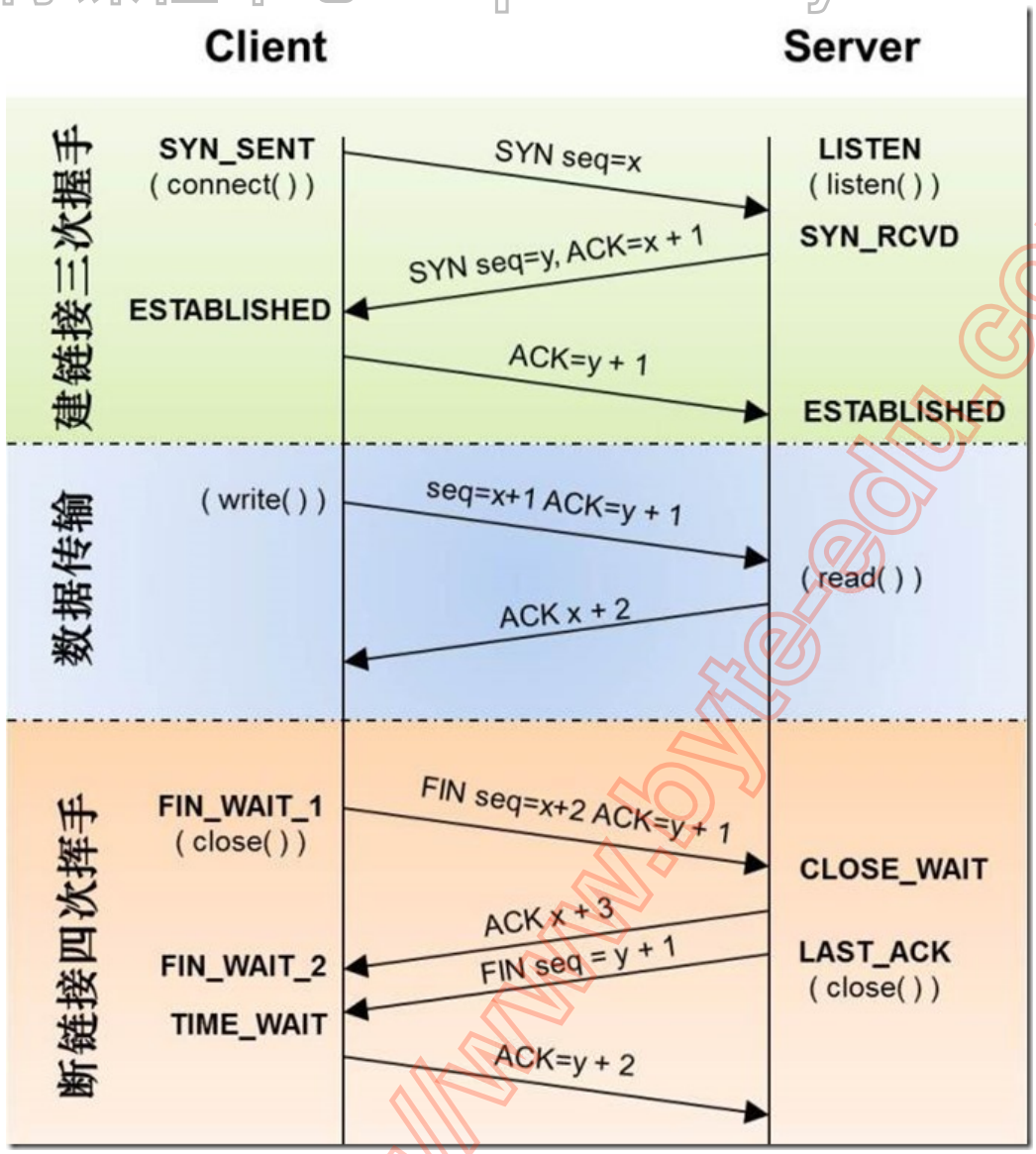
> 其中 `Kqueue` 是在 `FreeBSD` 平台上实现的 `I/O` 多路复用模型，跟 `Linux` 上的 `Epoll` 非常类似。

---

3. TCP/IP 协议简介

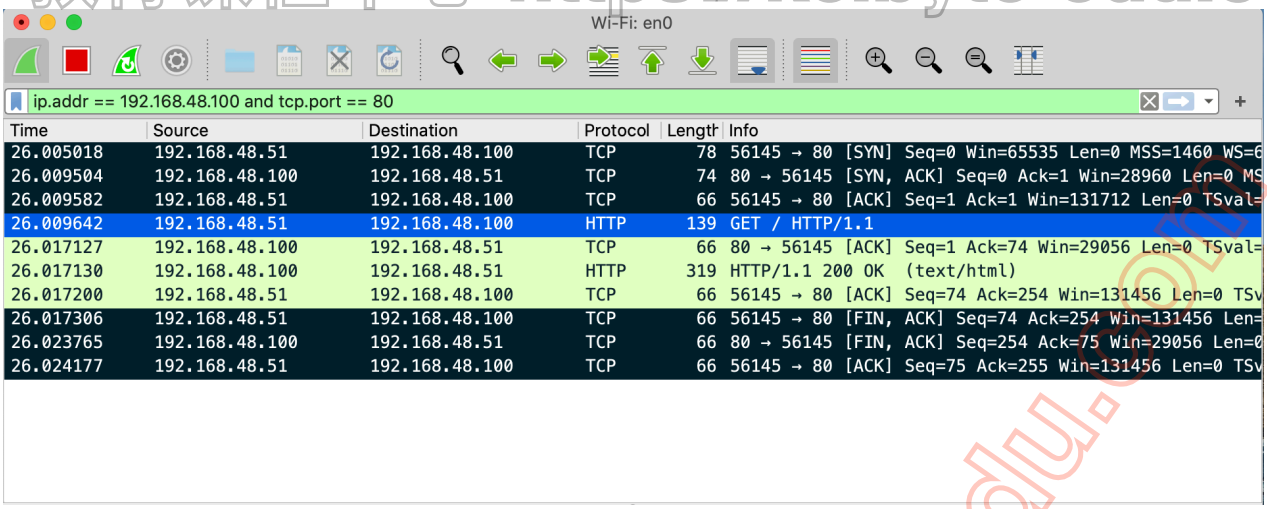
3.1 TCP 三次握手四次断开简介

简言之 `TCP/IP` 是指协议簇，也就是一组协议，而非单指 `TCP` 协议或者 `IP` 协议。我们今天要介绍的是这组协议中的 `TCP` 协议，主要会介绍跟 `Nginx` 调优相关的一些技术。首先，我们要介绍下著名的 "**`TCP` 协议的三次握手与四次挥手**"，如下图：

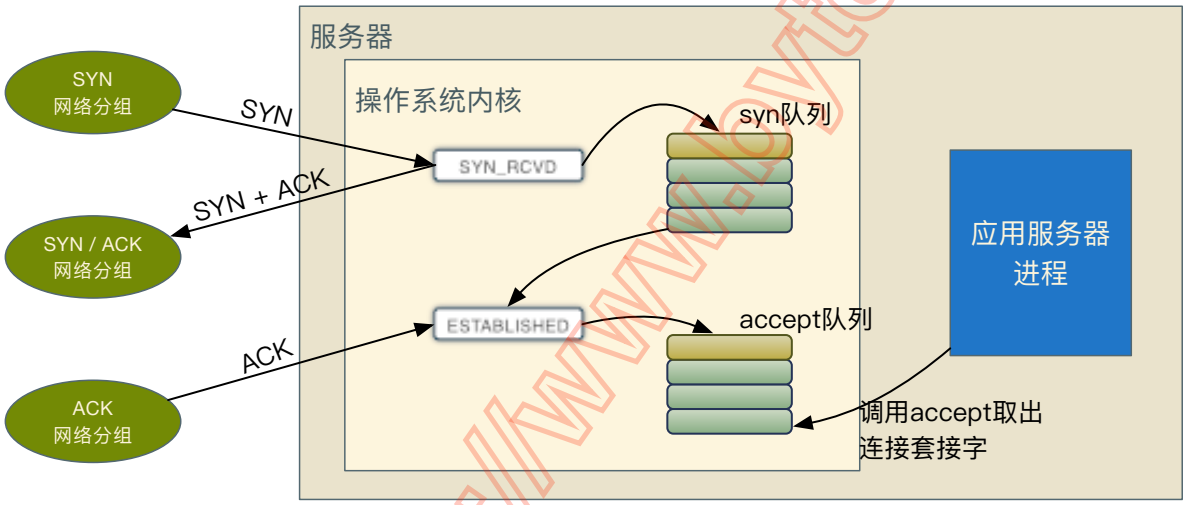


在咱们之前建立的 `Nginx` 服务器上，我们也可以通过抓包来查看这三次握手和四次挥手的过程，分别如下图所示的两个黑色选中部分，第一部分三个报文是建立 `TCP` 的三次握手，握手成功后开始建立 `http` 请求，与下面黑色部分之间则是数据报文的传递，客户端请求、服务端进行响应。后面的三行则是进行 `TCP` 的 "四次挥手"（但是为什么只有三个报文，是因为我们请求的数据服务端已经完成响应，没有额外需要发送的数据，所以客户端发送 `FIN` 标志位之后，服务端就直接回复 `ACK` 并且调用 `close()` 方法，也想客户端发送了 `FIN` 标志，客户端接收到之后发送 `ACK` 确认信息，等待两个 `MSL` 周期之后关闭连接）。

> **注意：两个最大段生命周期：`2MSL, 2 Maximum Segment Lifetime`**



我们的优化部分会设计到这 "**三次握手和四次挥手**"，当然这跟系统级的优化也有关系。我们先来看下从网卡接收到数据包到交由 `Linux` 内核协议簇处理，再到建立 `TCP` 三次握手的简单过程，如下：



如上图所示：对于 `Client` 端的一个请求，流程是这样的：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格式，数据包会被先存入到 `CPU` 的 `input_pkt_queue` 队列中（通过调整系统参数 `net.core.netdev_max_backlog` 来调整队列大小），然后内核将 `Client` 首先发送的 `SYN` 连接信息放到 `syn` 队列中（该队列由内核参数 `net.ipv4.tcp_max_syn_backlog` 决定），这个时候队列中都处于半连接状态，同时返回一个 `SYN+ACK` 包给客户端，这里涉及到一个调优参数 **net.ipv4.tcp_synack_retries**：即服务端向客户端返回 `SYN+ACK` 的尝试测试，在 `CentOS7` 上默认是 5 次（`1s + 2s + 4s + 8s + 16s = 31s` 即会尝试 5 次，大概 31s 时间）。之后 `Client` 再次发送 `ACK` 包给服务端，内核会把连接从 `syn` 队列中取出，再把这个连接放到 `accept` 队列中，此时已经是 `ESTABLISHED` 状态。最后应用服务器调用 `accept()` 系统从 `accept` 队列中获取已经建立成功的连接套接字。这里我们就涉及到以下内核调优参数，其中包含三个队列：

net.core.netdev_max_backlog：接收自网卡，但未被内核协议栈处理的报文队列长度

net.ipv4.tcp_max_syn_backlog：`SYN_RCVD` 状态（即半连接）队列长度

backlog：全连接队列也就是我们图中标注的 `accept 队列`，该队列大小由系统参数和应用参数共同决定，即：**全连接队列的大小取决于：`min(backlog, somaxconn)`，其中 `backlog` 是由应用程序传入，`somaxconn` 是一个 os 级别的系统参数，通过设置 `net.core.somaxconn` 来调整。** 在 `Nginx` 中，`backlog` 参数在 `Listen` 参数后面指定，在 `CentOS` 上，默认值为 `511`。

3.2 由三次握手引发的一些问题

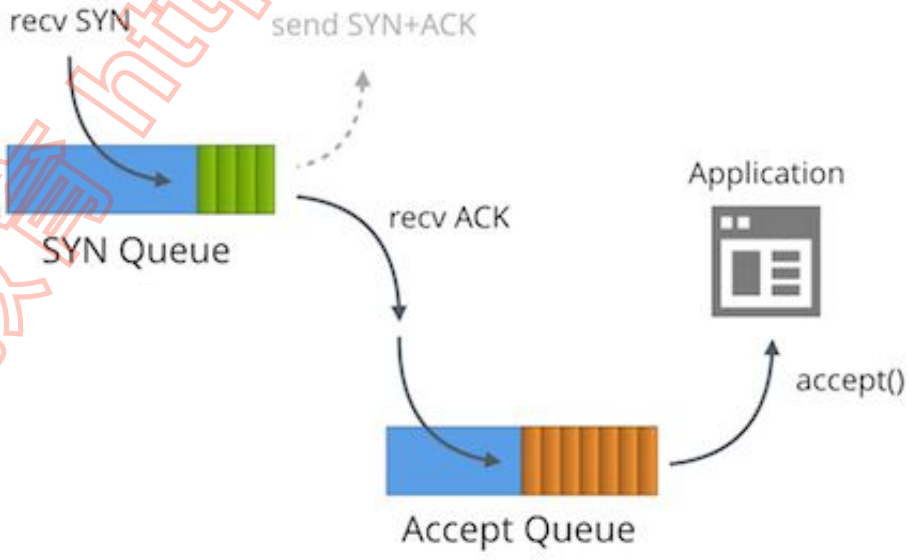
如果是 `syn 队列` 满了，不开启 `syncookies`（由参数 **net.ipv4.tcp_syncookies** 决定）的时候，服务端会丢弃新来的 `SYN` 包，而 `Client` 在多次重发 `SYN`（由参数 **net.ipv4.tcp_syn_retries** 决定，默认为 6 次，大概是 63s 时间）包得不到响应而返回 `connection timeout` 错误。

> **注：`Client` 端在多次重发 `SYN` 包得不到响应而返回 `connection time out` 错误，通过命令行查看会有如下信息：**
>
> ```
> c
> 4375 SYNs to LISTEN sockets dropped
> ```
>
> **即会显示 `XXX SYNs to LISTEN sockets dropped`，也就是 `SYN` 包被丢弃。**

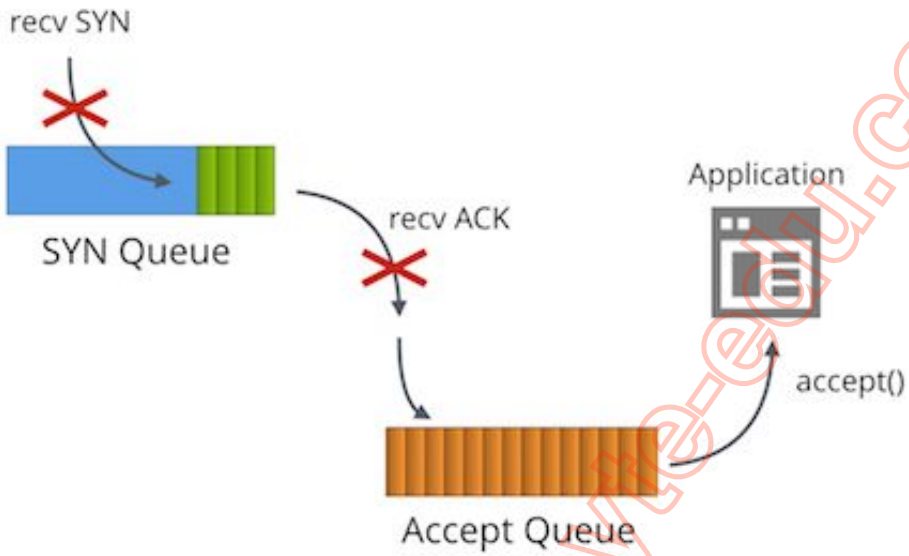
这里还涉及到另外一个内核参数 **net.ipv4.ip_local_port_range**，默认值为 `32768~60999` 表示主动连接端可以建立的随机端口号范围。但是，当 `Server` 端开启了 `net.ipv4.tcp_syncookies=1`，那么 `SYN` 半连接队列就没有逻辑上的最大值了。

如果 `accept 队列` 即**"全连接队列"**满了之后，即使 `Client` 继续向 `server` 发送 `ACK` 的包，服务端会通过 **net.ipv4.tcp_abort_on_overflow** 来决定如何响应，0 表示直接丢弃该 `ACK`，服务端过一段时间再次发送 `SYN+ACK` 给 `Client`（也就是重新走握手的第二步），如果 `Client` 超时等待比较短，就会返回 `read timeout`；1 表示发送 `RST` 通知 `Client`，`Client` 端会返回 `connection reset by peer`。

我们来看下这个过程：



这是我们正常的建立三次握手流程，其中半连接队列（`SYN Queue`）正常，全连接队列（`Accept Queue`） 正常。如果半连接队列满了，但是全连接队列正常（这种情况比较少），则请求发起方会在重复发送多次 `SYN` 后得不到响应而受到 `connection timeout` 错误。但是如果全连接队列满了，半连接队列未满，此时同样会影响半连接的建立，应用程序读取正常连接速度较慢，如下：



即当全连接队列满时，会发生下面四个行为：

- `SYN` 队列的入站 `SYN` 数据包将被丢弃
- `SYN` 队列的入站 `ACK` 数据包将被丢弃
- `TcpExtListenOverflows` / `LINUX_MIB_LISTENOVERFLOWS` 计数增加
- `TcpExtListenDrops` / `LINUX_MIB_LISTENDROPS` 计数增加

下面两个是 `Linux` 底层代码的逻辑数字大家不用关注，我们只需要知道在全连接队列满了的情况下，半连接队列也是不会接收入栈数据的，至于为什么，这是因为系统的设计者希望让应用能够通过读取读取全连接队列中的数据而慢慢的恢复。所以这也是为什么正常情况下，`sockets overflowed` 会大于等于 `sockets ignored`，并且 `overflowed` 增加时 `ignored` 同步增加。

> **注：`sockets overflowed` 表示全连接队列溢出次数，`sockets ignored` 表示半连接队列溢出次数，如：**

```
[root@server ~]# netstat -s | egrep "listen|LISTEN"
1641906 times the listen queue of a socket overflowed
1641906 SYNs to LISTEN sockets ignored
```

我们可以通过 `ss` 命令来查看某个 `IP:PORT` 设置的全连接队列，以及当前正在使用的队列大小，如下：

```
[root@web-nginx ~]# ss -tnl sport = :80
State      Recv-Q Send-Q Local Address:Port               Peer Address:Port
LISTEN     0      511          *:80                        *:*
LISTEN     0      511       [::]:80                     [::]:*
```

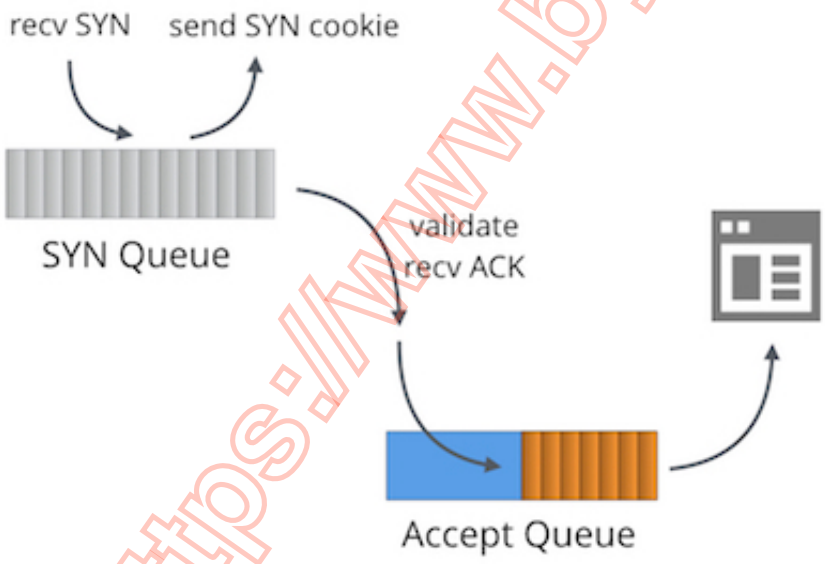
其中 `Recv-Q` 字段表示全连接队列中正在使用的队列大小，`Send-Q` 表示 `backlog` 设置的大小，注意该值的大小由 `min(backlog, somaxconn)` 决定。所以在生产环境中，我们要适当的调整这两个队列参数的大小，以应对更大的数据流量。说到这里，我们就不得不引入一个常见的问题，叫做 **`SYN` 洪水攻击**。

3.3 SYN 洪水攻击

在三次握手过程中，服务器发送 `SYN-ACK` 之后，收到客户端的 `ACK` 之前的 `TCP` 连接称为半连接(`half-open connect`)。此时服务器处于 `SYN_RCVD` 状态。当收到 `ACK` 后，服务器才能转入 `ESTABLISHED` 状态.

`SYN` 攻击指的是，攻击客户端在短时间内伪造大量不存在的 `IP` 地址，向服务器不断地发送 `SYN` 包，服务器回复确认包，并等待客户的确认。由于源地址是不存在的，服务器需要不断的重发直至超时，这些伪造的 `SYN` 包将长时间占用半连接队列，导致正常的 `SYN` 请求被丢弃，导致目标系统运行缓慢，严重者会引起网络堵塞甚至系统瘫痪。也就是说只要填满 `SYN` 队列，就可以用很少的带宽成功地拒绝几乎所有 `TCP` 服务器的服务。

在 `Linux` 中我们可以通过设置参数 `net.ipv4.tcp_syncookies = 1` 来达到减少这种攻击的影响，如下图：



当设置 `net.ipv4.tcp_syncookies = 1` 时，当 `SYN` 队列满了之后，新的 `SYN` 不进入队列，由服务端计算出 `cookie` 后再以 `SYN + ACK` 的方式返回给客户端，正常客户端发送报文时，服务器根据报文中携带的 `cookie` 信息重新恢复连接。

通过这个参数可以应对较小的 `SYN` 洪水攻击，如果攻击数量包太多，可能需要使用专业的防火墙。需要注意的是，如果启用了 `tcp_syncookies` ，则 `TCP_FAST_OPEN` 功能无法使用。

3.4 关于 TFO 介绍

这里跟大家再介绍一个参数 **net.ipv4.tcp_fastopen** ，我们来看下 `TCP FastOpen` 简称 **"TFO"** 过程，如下：

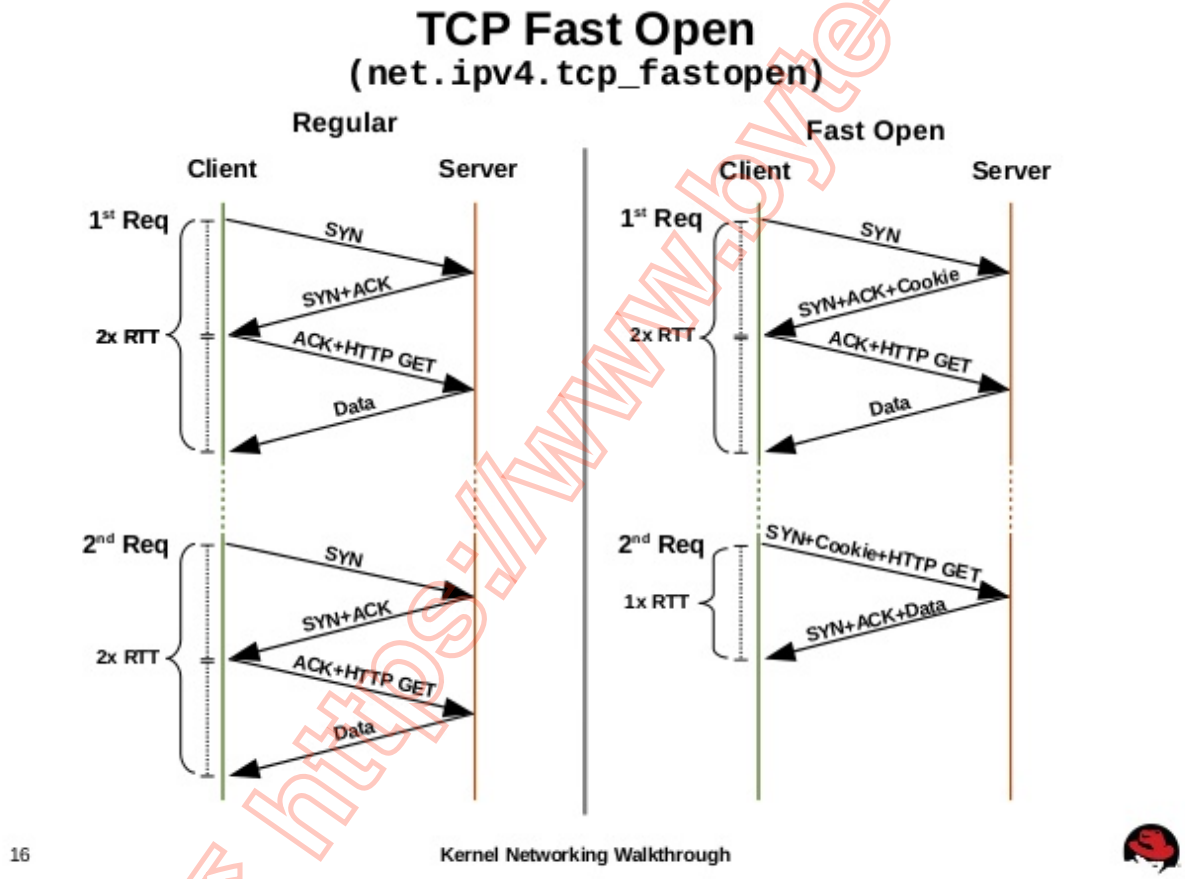
在使用 **TFO** 之前， `Client` 首先需要通过一个普通的三次握手连接获取 `FOC(Fast Open Cookie)`

- `Client` 发送一个带有 `Fast Open` 选项的 `SYN` 包，同时携带一个空的 `Cookie` 域来请求一个 `Cookie`

- `Server` 产生一个 `Cookie` ，然后通过 `SYN+ACK` 包的 `Fast Open` 选项来返回给 `Client`
- `Client` 缓存这个 `Cookie` 以备将来使用 `TFO` 连接的时候使用

这是第一次建立 `TCP` 连接时的过程，主要目的是获取 `Cookie` ，之后的连接需要通过该 `Cookie` ，如下：

- `Client` 发送一个带有请求数据的 `SYN` 包，同时在 `Fast Open` 选项中携带之前通过正常连接获取的 `Cookie`
- `Server` 验证这个 `Cookie` ，如果这个 `Cookie` 是有效的， `Server` 会返回 `SYN+ACK` 报文，同时会对请求的报文直接进行数据响应。如果这个 `Cookie` 是无效的， `Server` 会丢掉 `SYN` 包中的数据，同时返回一个 `SYN+ACK` 包来确认 `SYN` 包中的系列号进行普通握手。
- 剩下的连接处理就类似正常的 `TCP` 连接了， `Client` 一旦获取到 `FOC` ，可以重复 `Fast Open` 直到 `Cookie` 过期。



`TFO` 在 `Linux` 内核 `2.6.34` 中开始引入， `Nginx` 也在版本 `Release-1.5.8` 支持了该功能，同样在 `listen` 字段中进行指定，下面我们就来介绍下这个参数：

net.ipv4.tcp_fastopen ：有四个可用值，分别为 **"0、1、2、3"**，请含义如下：

0：禁用 TFO 功能；

1：作为客户端时可以使用 TFO 功能；

2：作为服务端时可以使用 TFO 功能；

3：无论是作为客户端还是服务端都可以使用 TFO 功能；

另外，如果在 `Nginx` 服务中使用该功能，为了防止带有数据的 `SYN` 攻击，一般会限制 TFO 队列的连接长度，比如：

```
listen 127.0.0.1:8000 fastopen=1024;
```

在实际使用中需要客户端和服务端同时支持 `TFO` 功能才能启用（目前浏览器好像只有 `chrome` 才支持），所以真正应用的场景并不太多。

3.5 TCP 缓冲区优化设置

net.ipv4.tcp_rmem = 4096 87380 6291456 ：读缓冲区 **最小值、默认值、最大值**，单位为字节，其默认值与最大值会覆盖内核参数：`net.core.rmem_default` 与 `net.core.rmem_max`；

net.ipv4.tcp_wmem = 4096 16384 4194304 ：写缓冲区 **最小值、默认值、最大值**，单位为字节，其默认值与最大值会覆盖内核参数：`net.core.wmem_default` 与 `net.core.wmem_max`；

如果指定了 `tcp_wmem`，则 `net.core.wmem_default` 被 `tcp_wmem` 的覆盖。`send Buffer` 在 `tcp_wmem` 的最小值和最大值之间自动调整。发送缓冲区的大小计算公式可以根据 **BDP 带宽时延积** 来得出，比如客户端与服务端之间的带宽为 `100Mbps`，延时为 `20ms`，那么 **BDP = 0.02s * (100Mb/8) = 256 Kbytes**，此时我们可以将 `net.ipv4.tcp_wmem` 的默认值设置为 `262144`，需要注意的是，我们的系统默认是可以自动调整发送、接收缓冲区大小的，所以一般程序中不要设置该参数。我们只需要根据带宽和延迟给出比较合理的范围即可。

一般生产环境上会适当调大这两个参数。这里还有两个参数做个简单介绍：

net.ipv4.tcp_mem = 1541646 2055528 3083292 ：分别表示 **系统无内存压力、启动压力模式阈值、最大值**，单位为页的数量；

net.ipv4.tcp_moderate_rcvbuf = 1 ：表示开启自动调整缓存模式；

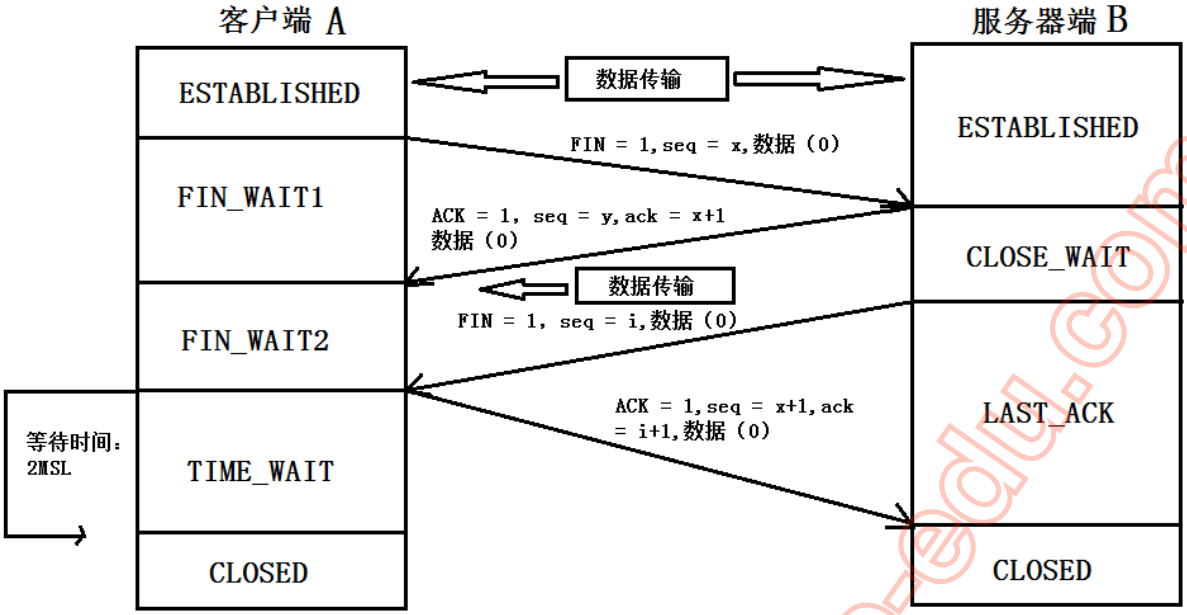
在 `Nginx` 中，我们同样可以在 `listen` 参数中对该参数进行设置，但是 **这种设置会破坏系统的自动调节能力，所以一般情况下我们不加指定**。

```
listen address[:port] [rcvbuf=size] [sndbuf=size]
```

> **关于 TCP 连接内存的使用，大家可以看下这篇文章** 高性能网络编程7--tcp连接的内存使用

3.6 TCP 连接断开优化

我们来看下四次挥手过程中主动关闭一方和被动关闭一方两者不同时期的状态变化，如下图：



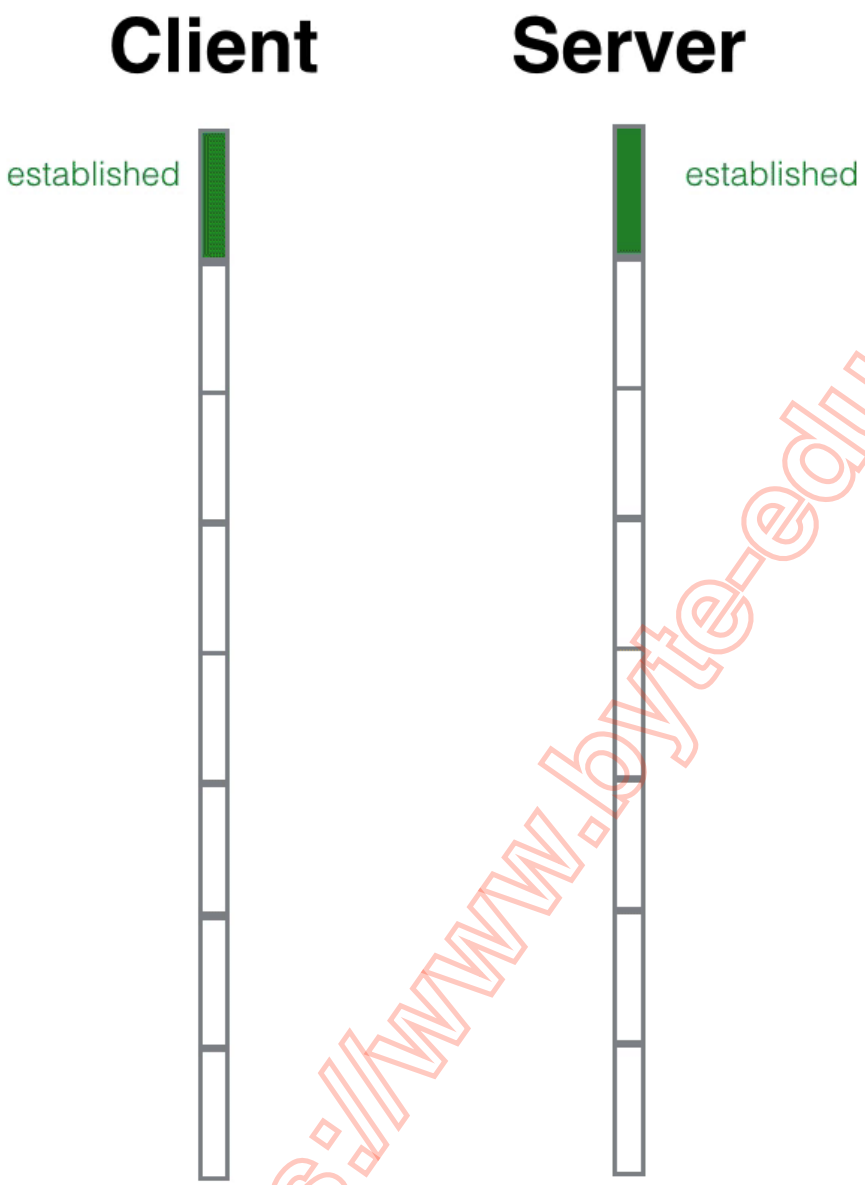
我们可以看下当一方主动断开时，主动断开方和被动断开方之间的状态变化：

我们来对上面的过程（江湖人称“四次断开”）做个简单的流程说明。首先需要跟大家明确的一点是，我们图中标注的 "**客户端A**" 不一定是指咱们的浏览器或者某个 `app` 应用，也可以是服务器。比如当我们使用 `Nginx` 做网络代理时，其实此时的 `Nginx` 服务器就作为用户的服务端，同时也作为后端真正应用服务器的客户端。为了方便说明问题，我们将主动断开的一方此处称为 "**客户端A**"（或者下图的 `Client`），将被动断开的一方此处称为 "**服务器端B**"（或者下图的 `Server`）。

- 客户端如果希望断开连接会像服务端发送带有 `FIN` 标志位的报文，此时客户端连接由 `ESTABLISHED` 状态变成 `FIN_WAIT1`；
- 服务器端在收到客户端发送的 `FIN` 请求报文后，此时服务器端可能还有报文需要发送，会带着未发送完的数据和（客户端 `FIN` 报文的确认信息） `ACK` 信息返回给客户端，同时连接也会由 `ESTABLISHED` 状态变成 `CLOSE_WAIT`；
- 当客户端收到服务器端发来的 `ACK` 确认报文后，就变成 `FIN_WAIT2` 状态，等待服务器端的断开；
- 如果服务器端数据已经发送完了，将向客户端发送 `FIN` 报文，申请关闭连接，此时服务器端由 `CLOSE_WAIT` 状态变成 `LAST_ACK` 状态，等待客户端的确认信息；
- 客户端在收到服务器端的 `FIN` 报文后，会向服务器端发送 `ACK` 确认信息，同时从 `FIN_WAIT2` 状态变成 `TIME_WAIT` 状态；
- 服务器端再收到客户端的 `ACK` 确认信息之后，会立即关闭连接（`CLOSED`）；
- 客户端再等待 `2MSL` 之后，也会关闭己方维护的连接，至此客户端与服务器端完成 `TCP` 的四次断开；

> **注意：有时候断开连接可能只需要三次，即当被动断开方在收到主动断开方的 `FIN` 报文后，也无额外数据响应，此时就会连同 `ACK` 确认号以及 `FIN` 标志一起发送给主动断开方，同时进入 `LAST_ACK` 阶段。我们上面的抓包示例图其实就是三次断开。**

这里有个四次断开的动态示意图，大家看一下。



在 `TCP` 四次断开的过程中，我们可以根据服务器上状态的异常情况来发现一些网络问题。

3.6.1 被动断开方异常

- 被动断开一方（多为 `RealServer`）上有大量的 `CLOSE_WAIT` 状态连接，一定是我们的应用程序有 `Bug`，也就是没有判断数据读取为 0 或者说数据发送完之后，没有调用 `close()` 方法，导致一直处于 `CLOSE_WAIT` 阶段；
- 被动断开一方（多为 `RealServer`）上有大量的 `LAST_ACK` 状态连接，则一般是无法收到主动断开方的 `ACK` 报文，一般这种情况出现时多是网络故障，和系统与应用层面的关系不大；

3.6.2 主动断开方异常

这里我们说的主动断开方，更多的场景是在 `Nginx` 作为反向代理时，一般情况下服务器上我们很少会看到 `FIN_WAIT1` 和 `FIN_WAIT2` 状态（除非网络层面故障），关于这两种状态我们有两个内核参数可以进行优化，如下：

net.ipv4.tcp_orphan_retries = 0：用来表示发送 `FIN` 报文的重试次数，默认为 8 次，0 也表示 8次，如果服务器上有大量的 `FIN_WAIT1` 连接，肯定是网络层面出现问题，即使我们调下这个值，也之后减少该状态值出现的时间，而不能解决根本问题，即治标不治本。但是适当的调小此时有利于在网络层面出现问题的情况下直接暴露出来，而不是长时间的尝试等待。一般在 `Web` 服务器上，我们会把值设置为 2 或者 3 即可。

```
# 下面这段代码诠释了为什么值为 0 的时候也相当于 8;
{
        int retries = sysctl_tcp_orphan_retries; /* May be zero. */

        /* We know from an ICMP that something is wrong. */
        if (sk->sk_err_soft && !alive)
                retries = 0;

        /* However, if socket sent something recently, select some safe
         * number of retries. 8 corresponds to >100 seconds with minimal
         * RTO of 200msec. */
        if (retries == 0 && alive)
                retries = 8;
        return retries;
}
```

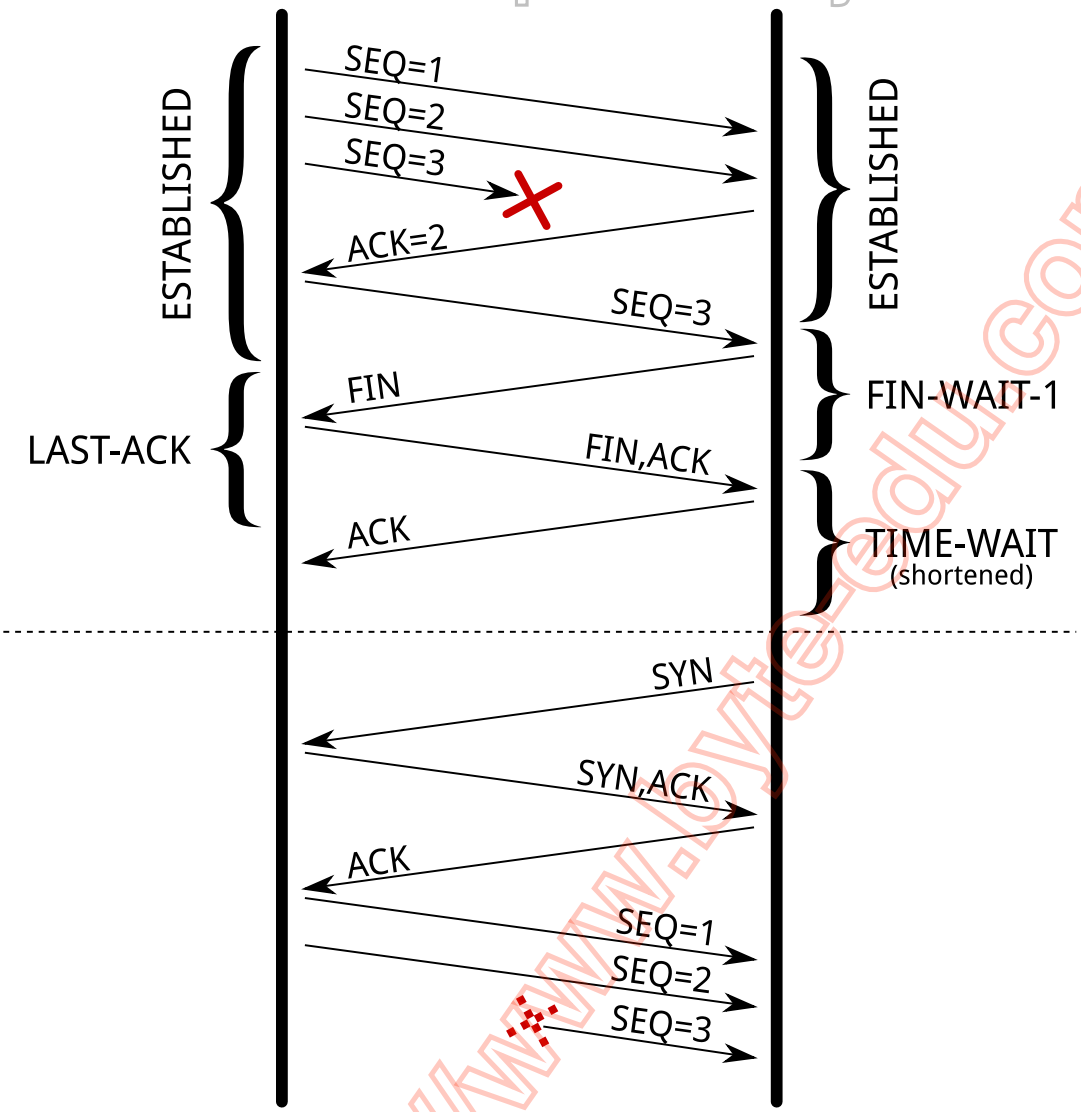
net.ipv4.tcp_fin_timeout = 60：表示主动断开连接端保持在 `FIN_WAIT2` 状态的超时时间，这种情况的出现跟被动断开端的 `CLOSE_WAIT` 大量出现问题相似，也有可能是被动断开端应用未调用 `close()` 方法发送 `FIN` 报文导致，当然也有可能是网络层面原因。在 `Web` 服务的生产环境中，一般会适当调下该值，比如我们可以设置 10 或者 15（单位秒）。

在主动关闭一方，`FIN_WAIT1` 与 `FIN_WAIT2` 这种状态在正常情况下很少会看到，但是 `TIME_WAIT` 状态则经常会遇到，下面我们来讨论下 `TIME_WAIT` 状态存在的含义，这个值如果设置的太小或者不存在会有什么影响以及该如何优化等问题。

首先我们来讨论下 `TIME_WAIT` 时间过短或者不存在会发生什么问题?

- **数据报文重复** 造成的数据混乱问题

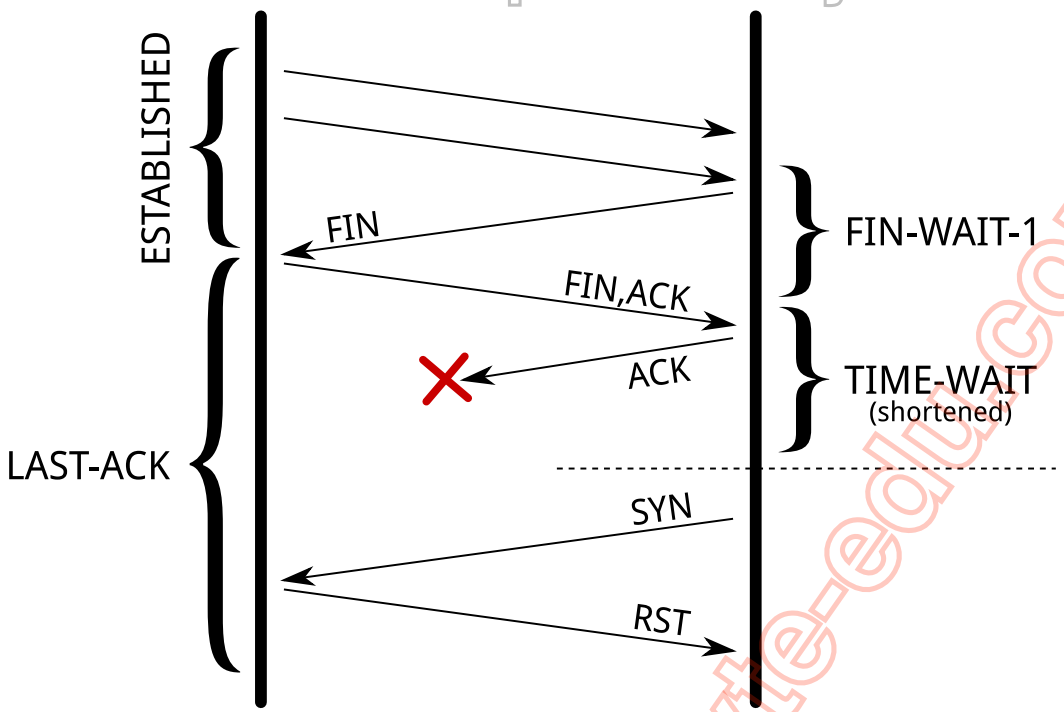
如下图：



即客户端（主动断开方）在发送 `FIN` 报文之前，服务端给我们发送了三个报文，但是由于网络原因，客户端只收到了前两个报文，于是服务器又重新发送了第三个报文。客户端在接收到这个报文后发起了断开连接请求，也就是我们前面介绍的四次挥手。但是这时如果我们 `TIME_WAIT` 时间设置过短，直接关闭了客户端连接。此时又与服务器端建立了新连接，并且凑巧的建立连接的网络四元组（**Client IP，Client PORT & Server IP，Server PORT**）恰好一致，同时服务器端也发送了三号报文，这时上次连接的三号报文恰好被服务器端接收，就会造成数据报文的重复混乱。这种概率很小，但是网络数据报文数据量非常大，所以这个问题会给我们造成非常大的困扰。

- **ACK 报文丢失** 造成的连接被重置问题

如下图：



同样，如果 `TIME_WAIT` 时间过短或者不存在，导致客户端向服务器端发送最后一个 `ACK` 报文丢失而没能重发（因为 `TIME_WAIT` 时间太短，客户端已经自己关闭了这个连接），此时客户端连接已经关闭，但是服务器端仍处于 `LAST-ACK` 阶段，等待最后一个 `ACK` 报文的到来。如果这个时候客户端期望建立新的连接，向服务器端发送了 `SYN` 报文，此时服务器端会直接回复 `RST` 报文而重置此连接，这时客户端就会产生异常。

以上两个问题就是 `TIME_WAIT` 为什么要存在，且存在周期那么长的原因。同样，因为存在时间过长，所以可能导致大量的资源被浪费，包括端口资源，协议栈队列，所以大量的 `TIME_WAIT` 会影响 `socket` 建立新连接，这点特别在高性能的 `Web` 服务器中很讲究，如何进行相关的优化呢？

首先我们可以考虑增大服务器允许的 `TIME_WAIT` 数量值，通过修改内核参数 **`net.ipv4.tcp_max_tw_buckets`** 来调整，默认值是 `16384` ，比如我们可以调大至 `262144` ，超过该数量后直接关闭连接。但是这只能解决 `TIME_WAIT` 过多导致的无法建立新连接的问题，而不能解决资源浪费的问题。当然，如果你的服务器资源并不富裕，那么你可以调小这个值，让 `TIME_WAIT` 始终处于一个低值（比如 `5000` ），以避免造成资源浪费。所以，大家根据自己需求调整吧。

> **注意：`net.ipv4.tcp_max_tw_buckets` 表示系统同时保持 `TIME_WAIT` 的最大数量，如果超过这个量，`TIME_WAIT` 将打印警告信息：**
>
> ```
> TCP: time wait bucket table overflow
> TCP: time wait bucket table overflow
> TCP: time wait bucket table overflow
> TCP: time wait bucket table overflow
> TCP: time wait bucket table overflow
> ```
>
> **超限的时候后面产生的 `TIME_WAIT` 直接不处理，释放资源。注意：是新的连接直接释放资源，老的连接还是处于 `TIME_WAIT` 状态。**

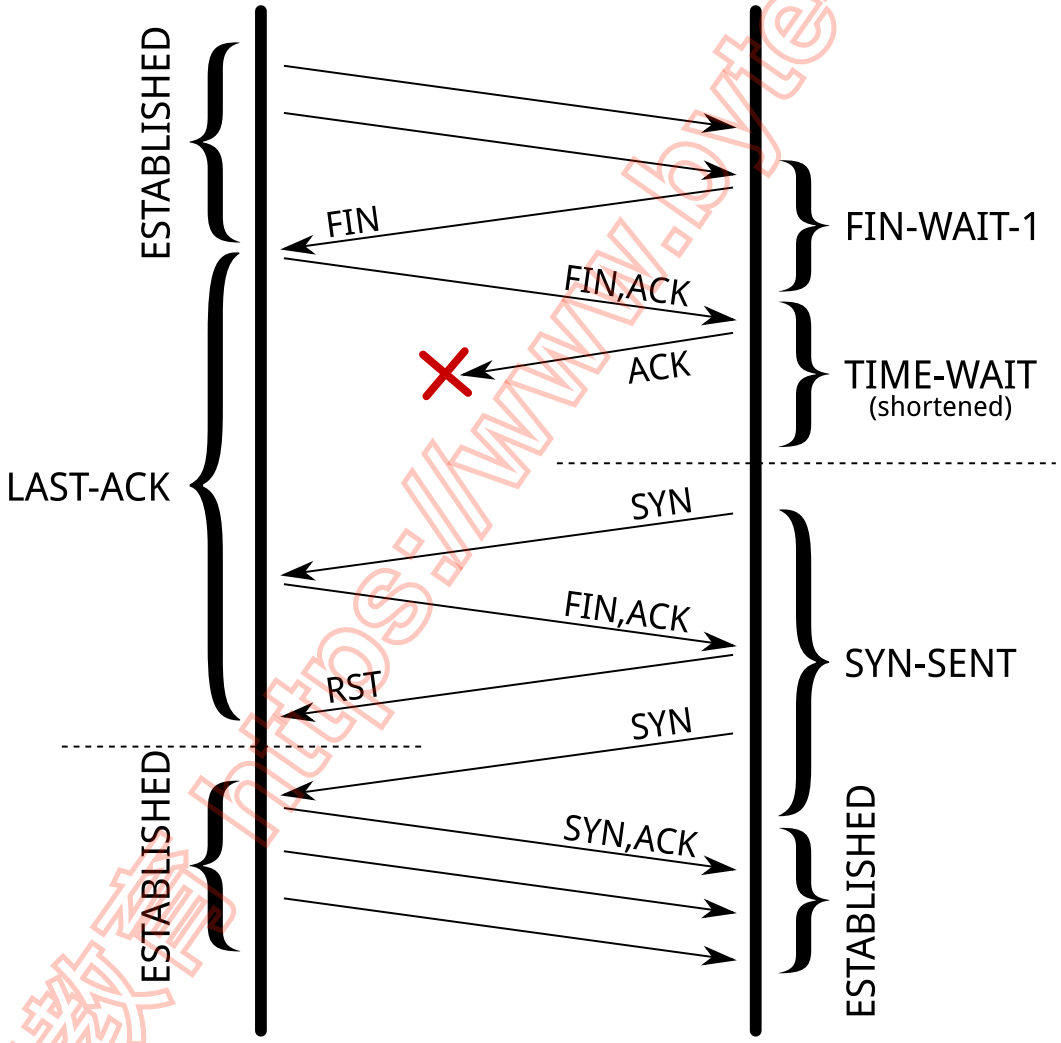
我们可以考虑使用这个内核参数：

net.ipv4.tcp_tw_reuse = 1：开启后，我们在作为客户端时可以使用仍然处于 `TIME_WAIT` 状态的端口，与服务端使用之前维持的连接，但是同样需要解决上面的两个问题，即 **数据报文重复** 造成的数据混乱问题以及 **ACK 报文丢失** 造成的连接被重置问题。

对于 **数据报文重复** 问题，引入了 `timestamp` 即时间戳，让操作系统可以拒绝迟到的报文。因此如果想要启动 `net.ipv4.tcp_tw_reuse` 必须先设置 **net.ipv4.tcp_timestamps = 1**（默认即为启用）

对于 **ACK 报文丢失** 造成的连接被重置问题，在我们复用了客户端 `TIME_WAIT` 连接之后，`Socket` 五元组并没有变化，因此客户端使用该连接的地址和端口想服务端发送 `SYN` 报文，不会被服务端 `RST`，但是因为未收到客户端的 `ACK` 报文，所以会再次发送 `FIN+ACK` 报文，这时客户端看到此报文后认为异常，会重置 `RST` 服务器端连接，此时客户端再次发送 `SYN` 请求握手，服务器端返回 `SYN+ACK`，然后重新建立新的连接。

其原理如下：



有的同学可能会有点懵，如果客户端最后一个 ACK 没有丢失怎么办？没有丢失的话当然更好了，因为此时服务器端已经正常端口连接了，我们在客户端复用 TIME_WAIT 端口重新与服务端建立新连接即可。

跟 `TIME_WAIT` 端口复用的还有一个参数，也是基于 `timestamp` 才能启用的，即：

net.ipv4.tcp_tw_recycle：启动该参数后，同时作为客户端和服务器都可以使用 `TIME_WAIT` 状态的端口，这会带来很多不安全问题，同时经常会因为报文延迟、重复给新连接造成混乱（虽然也可以通过 `timestamp` 来避免），但是另外一个重要的弊端则无法解决。即：当作为服务端时，用户经过 `NAT` 网络访问时，其源 `IP` 地址都会转换成相同的出口地址，当来自同一个 `IP` 地址（任意源端口号）后来的数据包中 `TCP` 选项字段如果有 `timestamp` 且比前面的数据包中的 `timestamp` 小，则 `server` 不做 `ACK` 响应。目前在高版本 `Linux 4.12` 已经移除了该参数，所以大家尽量不要启动该功能，即设置 **net.ipv4.tcp_tw_recycle = 0** 。

3.7 其他内核参数优化

下面介绍两个 **关于 TCP 丢包重传** 的参数：

net.ipv4.tcp_retries1 = 3：默认值为 3，表示到达上限后，更新路由缓存；

net.ipv4.tcp_retries2 = 15：默认值为 15，表示到达上限后，关闭 `TCP` 连接；

需要注意的是，这两个值在生产上使用不合适，实际情况下可能还没达到 `retries` 次数就已经达到时间上限，所以生产上我们一般会调整两个参数的值，比如：

```
net.ipv4.tcp_retries1 = 2
net.ipv4.tcp_retries2 = 3
```

下面在介绍下 **关于 TCP KeepAlive** 的参数：

我们知道在实际使用中，为了减少 `TCP` 建立连接的次数， `TCP` 连接保持一段时间之后再断开是一个非常好的优化手段，但是如果一个 `TCP` 连接如果长时间打开而不使用，则会很大程度上浪费服务器资源，因此我们会适当调整系统级与应用级参数，以最大限度的提升服务器性能。

net.ipv4.tcp_keepalive_time = 7200：表示多久对一个 `TCP` 空闲连接发送一次心跳探测报文，单位是秒，所以默认是 2 小时；

net.ipv4.tcp_keepalive_intvl = 75：探测包每次发送的时间间隔；

net.ipv4.tcp_keepalive_probes = 9：探测包发送次数；

因为系统级的配置会影响所有应用，所以这几个参数都设置的非常“宽松”，因此我们应该根据自己的业务去适当调整，其实最好去调整某个具体应用。这里我们给出一个相对较为妥善的参数值：

```
net.ipv4.tcp_keepalive_time = 600
net.ipv4.tcp_keepalive_intvl = 2
net.ipv4.tcp_keepalive_probes = 3
```

> **注意：一般生产情况下，最好尽量去设置某个应用的连接超时时间，因为修改系统值会影响到所有的应用。不过我们设置为 10-30 分钟作用也是一个对各个应用来说差不多都可以接受的值。**

在 `Nginx` 中，我们也可以针对 `Nginx` 应用做单独设置，配置方式如下：

```
listen  so_keepalive=on|off|[keepidle]:[keepintvl]:[keepcnt]
```

比如：

```
listen so_keepalive=5m::10
```

如果某个参数没有指定，则会使用系统的默认参数，比如我们没有指定探测包发送时间间隔，则会根据系统内核参数 `net.ipv4.tcp_keepalive_intvl` 来进行设置。

> **注意：该参数是针对 TCP keepalive 行为，并不是 http 的 keep-alive 即长连接功能，两者无论是设计作用还是功能都不一样。关于 TCP keepalive 和 HTTP keep-alive 的区别大家可以参考这篇文章：HTTP Keep-Alive 和 TCP Keepalive原理与对比**

除此之外，还有几个和文件句柄相关的内核参数，我们知道在 `Linux` 中除了进程外，一切皆文件，`socket` 当然也是文件，所以我们能建立多少 `socket` 并维持多少连接，和我们能打开的文件句柄数有直接联系。

【系统级全局限制】

fs.file-max = 398458 ：表示操作系统可以使用的最大句柄数，该值默认值太小，我们需要调整为 `fs.file-max = 40000500`

fs.file-nr = 768 0 398458 ：这个参数不用设置，是用来查看当前已经分配、正使用、最大值（与 `fs.file-max` 一致）的文件句柄数；

fs.nr_open = 1048576 ：表示单个进程可以打开的文件句柄说，默认值已经很大了，但是如果是分布式图片服务器可能还是不足，大家可以适当调整。

除了系统级全局限制，还有针对用户的限制，我们可以通过创建文件 `/etc/security/limits.d/nofile.conf` ：

```
[root@web-nginx ~]# vim /etc/security/limits.d/nofile.conf
*          soft    nofile     65535
*          hard    nofile     65535
root       soft    nofile     65535
root       hard    nofile     65535
```

除此之外，还有应用层面的限制，比如在 `Nginx` 中，我们可以这样设置：

```
Syntax: worker_rlimit_nofile number;
Default: —
Context: main
```

> **Changes the limit on the maximum number of open files (RLIMIT_NOFILE) for worker processes.**

至此，系统级相关的参数功能以及优化设置我们就整理的差不多了，现在我们可以总结一下。

---

4. Linux 内核优化

首先，你需要修改 `/etc/sysctl.conf` 来更改内核参数。

```
net.ipv4.tcp_synack_retries = 2

net.core.netdev_max_backlog = 250000

net.ipv4.tcp_max_syn_backlog = 10240

net.core.somaxconn = 10240

net.ipv4.tcp_syncookies = 1

net.ipv4.tcp_syn_retries = 2

net.ipv4.ip_local_port_range = 1024 65000

net.ipv4.tcp_abort_on_overflow = 1

net.ipv4.tcp_rmem = 16384 1048576 12582912

net.ipv4.tcp_wmem = 16384 1048576 12582912

net.ipv4.tcp_mem = 1541646 2055528 3083292

net.ipv4.tcp_moderate_rcvbuf = 1

net.ipv4.tcp_orphan_retries = 3

net.ipv4.tcp_fin_timeout = 15

net.ipv4.tcp_max_tw_buckets = 5000

net.ipv4.tcp_tw_reuse = 1

net.ipv4.tcp_tw_recycle =  0

net.ipv4.tcp_timestamps = 1

net.ipv4.tcp_tw_recycle =  0

net.ipv4.tcp_retries1 = 2

net.ipv4.tcp_retries2 = 3

net.ipv4.tcp_keepalive_time = 600

net.ipv4.tcp_keepalive_intvl = 2
```

```
net.ipv4.tcp_keepalive_probes = 3

fs.file-max = 500000000

fs.nr_open = 10000000
```

5. Nginx 调优

5.1 主配置段优化

在介绍完关于系统部分的优化内容后，`Nginx` 自身也提供了一些优化参数，我们一起来看下。

【设置进程静态优先级】

我们从宏观层面上看一颗 `CPU` 好像可以同时执行很多进程，但其实从微观意义上来看，这些进程其实是串行执行的，也就是 `CPU` 把进程的运行时间分成一个个的时间片，然后 `CPU` 在通过系统调度某个进程，然后最多执行该进程时间片的时间。所以，如果进程时间片较长，那么他的执行优先级就更高，在 `Linux` 中我们可以通过调整 `NICE` 即静态优先级 `-20 ~ 19` 的大小来实现。数值越小，优先级越高，因此我们可以将 `Nginx` 优先级设置为 `-20`，设置参数如下：

```
Syntax: worker_priority number;
Default:  worker_priority 0;
Context:  main
```

> **注意：在 `Nginx` 官方文档中提到可以将该值设置为 `-20 ~ 20`，其实从 `Linux` 系统层面来说超过 `-20 ~ 19` 都是无意义的。**

设置完成后，需要重启 `Nginx` 才能生效，如下：

```
[root@web-nginx ~]# top -u nginx
top - 12:03:27 up 5 min,  1 user,  load average: 0.00, 0.00, 0.00
Tasks: 168 total,   1 running, 167 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.0 us,  0.1 sy,  0.0 ni, 99.8 id,  0.0 wa,  0.0 hi,  0.1 si,  0.0 st
KiB Mem :  4091196 total,  3681496 free,   287180 used,   122520 buff/cache
KiB Swap:  4063228 total,  4063228 free,        0 used.  3617484 avail Mem

  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
 1410 nginx     0 -20  121456   6856   4284 S   0.0  0.2   0:00.00 nginx
 1411 nginx     0 -20  121456   6856   4284 S   0.0  0.2   0:00.00 nginx
 1412 nginx     0 -20  121456   6856   4284 S   0.0  0.2   0:00.00 nginx
 1413 nginx     0 -20  121456   6856   4284 S   0.0  0.2   0:00.00 nginx
```

【设置 worker 进程数】

现在我们的处理器一般都是多核 `CPU`，进程的数量一般设置要大于 `CPU` 核心数，但是在 `Nginx` 的体系中，`worker` 进程是用来处理用户请求的进程非常繁忙，所以最好设置成与 `CPU` 核心数一样，以免多个 `worker` 进程争抢一颗 `CPU`，同时最好将 `worker` 进程与 `CPU` 核心进行绑定，一遍更好的利用 `CPU` 的多级缓存。目前关于 `worker` 进程数的设置语法如下：

```
Syntax: worker_processes number | auto;
Default:  worker_processes 1;
Context:  main
```

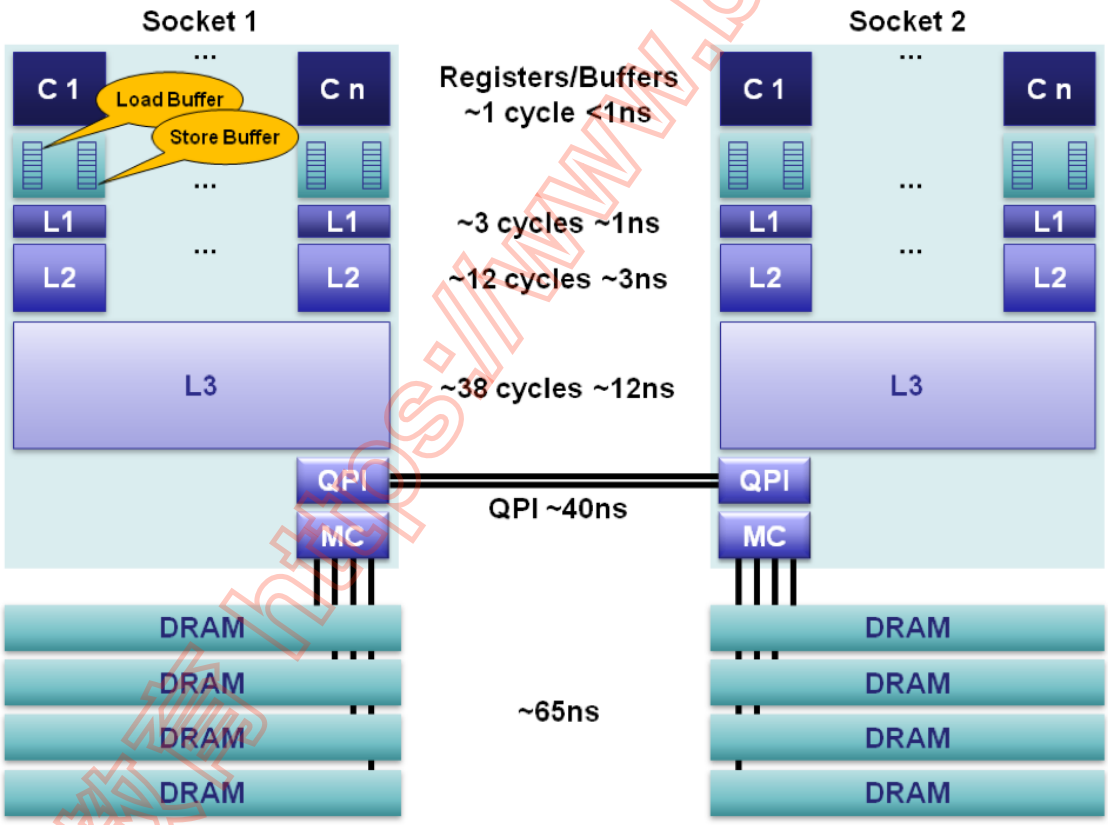
如果设置成 `auto` 则会根据 `CPU` 核心数自动创建相同个数的 `worker` 进程，在 `CentOS` 中可以通过如下命令查看：

```
[root@web-nginx ~]# lscpu | grep CPU\(s\)
CPU(s):                4
On-line CPU(s) list:   0-3
```

比如，我的服务器适合将 `worker_processes` 设置成 4 。

【CPU 绑定提供缓存命中率】

为了适应 `CPU` 的高速取值，现在的 `CPU` 一般会有三级缓存，其中一级 `L1` 缓存速度最好，二级 `L2` 次之，三级 `L3` 最慢，其中 `L1` 与 `L2` 为单插槽上单核 `CPU` 独享，`L3` 则是单插槽上多核 `CPU` 共享，如下：



因此，为了让 `worker` 能够更好的使用 `CPU` 的高速缓存，我们建议将 `worker` 与 `CPU` 核心进行一对一绑定，基本语法如下：

```
Syntax: worker_cpu_affinity cpumask ...;
        worker_cpu_affinity auto [cpumask];
Default:  —
Context:  main
```

示例：对于四颗核心的 `CPU` 应该这样绑定：

```
worker_processes    4;
worker_cpu_affinity 0001 0010 0100 1000;
```

假如我只想启动 2 个 `worker` 进程，并且指定仅使用 `CPU0` 与 `CPU3`，可以这样设置：

```
worker_processes    2;
worker_cpu_affinity 0001 1000;
```

还有一种写法比较特殊，如下：

```
worker_processes    2;
worker_cpu_affinity 0101 1010;
```

这表示第一个 `worker` 进程被绑定到 `CPU0/CPU2`，第二个 `worker` 进程被绑定到 `CPU1/CPU3`。这种使用方式较少见。

我们可以查看自己节点上 `CPU` 的三级缓存大小，如下：

```
# 1级缓存大小
[root@web-nginx ~]# cat /sys/devices/system/cpu/cpu0/cache/index1/size
32K

# 2级缓存大小
[root@web-nginx ~]# cat /sys/devices/system/cpu/cpu0/cache/index2/size
256K

# 3级缓存大小
[root@web-nginx ~]# cat /sys/devices/system/cpu/cpu0/cache/index3/size
25600K
```

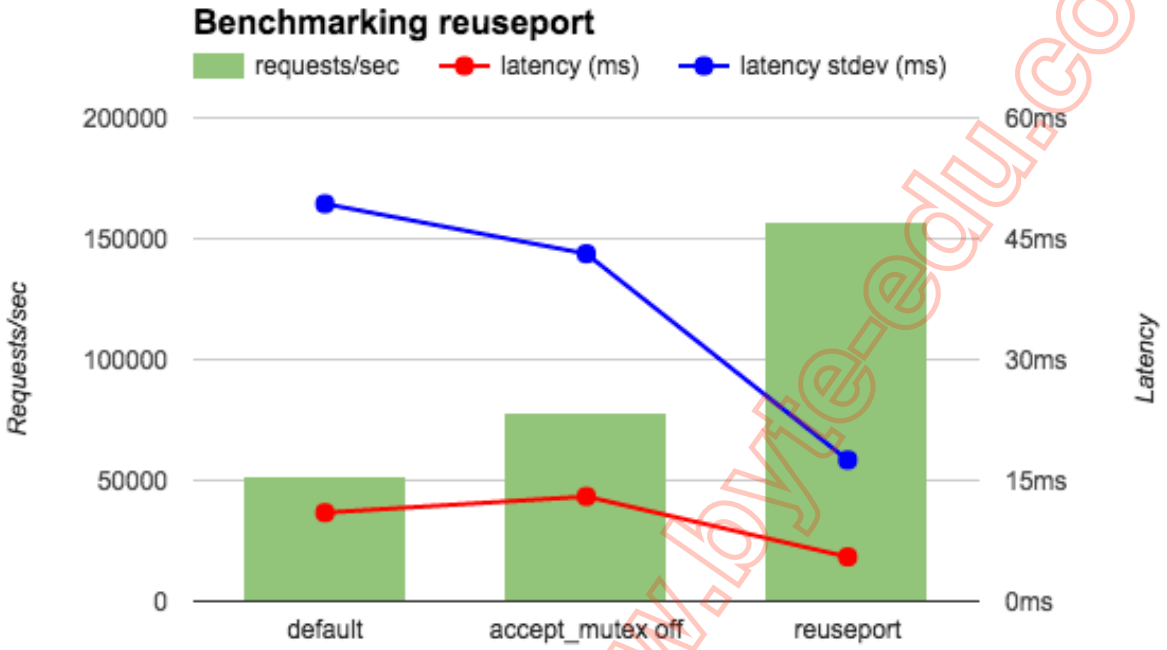
我们知道 3 级缓存是一个 `CPU` 插槽上所有核心共享的，可以看下哪几个核心在一个插槽上：

```
[root@web-nginx ~]# cat
/sys/devices/system/cpu/cpu0/cache/index3/shared_cpu_list
0-1
[root@web-nginx ~]# cat
/sys/devices/system/cpu/cpu2/cache/index3/shared_cpu_list
2-3
```

> 可以看到在我们的机器上，`CPU0/CPU1` 在一个插槽上，`CPU2/CPU3` 在另外一个插槽上。

【worker 进程间的负载均衡】

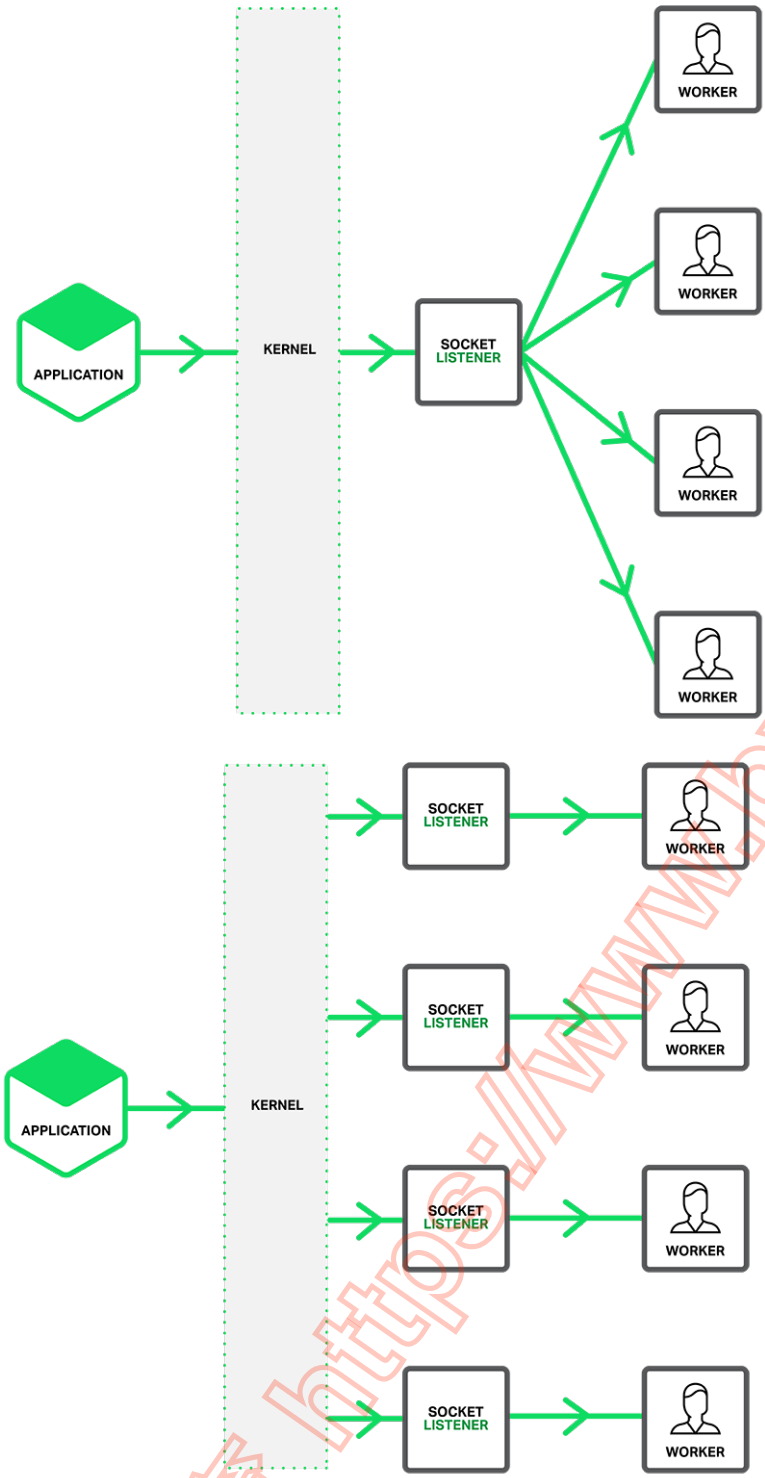
如何让多个 `worker` 进程间的响应处理达到比较合理的均衡值，`Nginx` 引入了 `accept_mutex` 参数，如果启用该参数表示多个 `worker` 进程间顺序接收新连接，关闭的话则会所有 `worker` 进程会同时被唤醒，但是只有一个 `worker` 进程能接收处理该连接。在 `Nginx 1.11.3` 版本以前，该参数默认是启用的。但是这种处理方式在处理并发连接较多时性能特别差，下面是三种机制的性能测试对比[数据出处]：



这里还有一组实验数据：

	Latency (ms)	Latency stdev (ms)	CPU Load
`Default`	15.65	26.59	0.3
`accept_mutex off`	15.59	26.48	10
`reuseport`	12.35	3.15	0.3

可以看下未启用 `reuseport` 和启用 `reuseport` 时，`worker` 处理进程的差异方式：



未启用 `reuseport` 启用 `reuseport`

需要注意的是 `reuseport` 需要 `Linux` 内核支持，即必须 `Linux 3.9+` 以上才可以使用，这种方式从内核层实现了 `worker` 进程间的负责均衡调度，效果非常好。启用语法：

```
Syntax: listen address[:port] [reuseport]
```

需要注意的是，在使用 `reuseport` 时，请设置 `accept_mutex off;`

> **There is no need to enable accept_mutex on systems that support the EPOLLEXCLUSIVE flag (1.11.3) or when using reuseport.**

【单个 worker 进程允许同时打开的连接数】

我们可以通过配置 **worker_connections** 来修改单个 worker 进程可以同时打开的连接数，默认是 512，需要调大该值，但是不能超过 worker_rlimit_nofile 。

```
Syntax: worker_connections number;
Default:  worker_connections 512;
Context:  events
```

> **Sets the maximum number of simultaneous connections that can be opened by a worker process.**

【所有 worker 进程允许同时打开的文件数】

一般 **worker_rlimit_nofile** 表示所有 worker 进程可以打开的文件句柄数，该值默认值未设置，将会受到系统级、用户级限制（注意：Nginx worker 进程的默认用户是 nginx ）。

> **maximum number of open files (RLIMIT_NOFILE) for worker processes.**

配置语法：

```
Syntax: worker_rlimit_nofile number;
Default:  —
Context:  main
```

【使用 epoll 事件模型】

对于这个参数我们不用过多赘述了，在 RHEL/CentOS 系统中 Nginx 编译后默认就是使用的 epoll 模型，基本语法如下：

```
Syntax: use method;
Default:  —
Context:  events
```

5.2 HTTP 配置段优化

我们这里简单介绍一些关于 http 配置段的一些比较通用的优化参数，另外还有一些会结合具体业务在后面配置应用课程中再做介绍。

同样的，在 HTTP 配置段中的 "**优化**" 我们也从 **"文件IO"** 与 **"网络IO"** 以及 **"网络连接"** 等三个核心层面进行介绍，不过说到底其实这三者之间也是紧密联系的，比如我们可以通过将文件压缩以实现更快的网络发送与接收。

5.2.1 文件 IO 优化方向

- 优化读取速度
- 减少磁盘写入

【优化读取速度 之 sendfile】

当然，升级高效的 **SSD** 固态硬盘效果最明显，这是属于硬件层面升级，我们来看下应用层面该如何优化。相信很多接触过 `Nginx` 的都会听说过这么一个参数 **sendfile** ，基本语法如下：

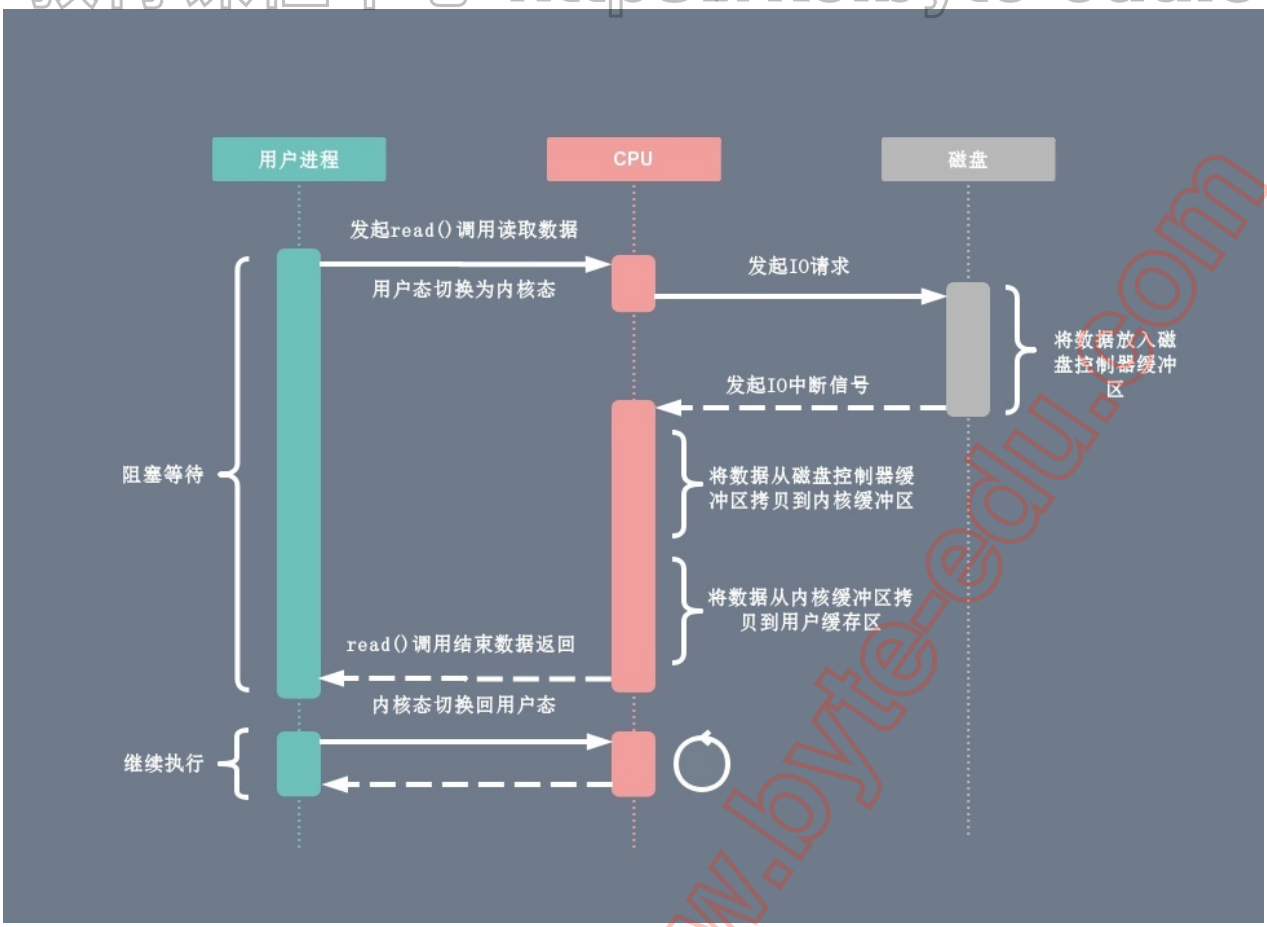
```
Syntax: sendfile on | off;
Default:  sendfile off;
Context:  http, server, location, if in location
```

需要注意的是这个参数其实并不是适用于任何场景，我们来解析下这个参数。在正式介绍 `sendfile` 这个参数时，我们首先介绍下 `CPU` 是如何处理磁盘或者网卡中断以及关于 `DMA` 的相关知识。

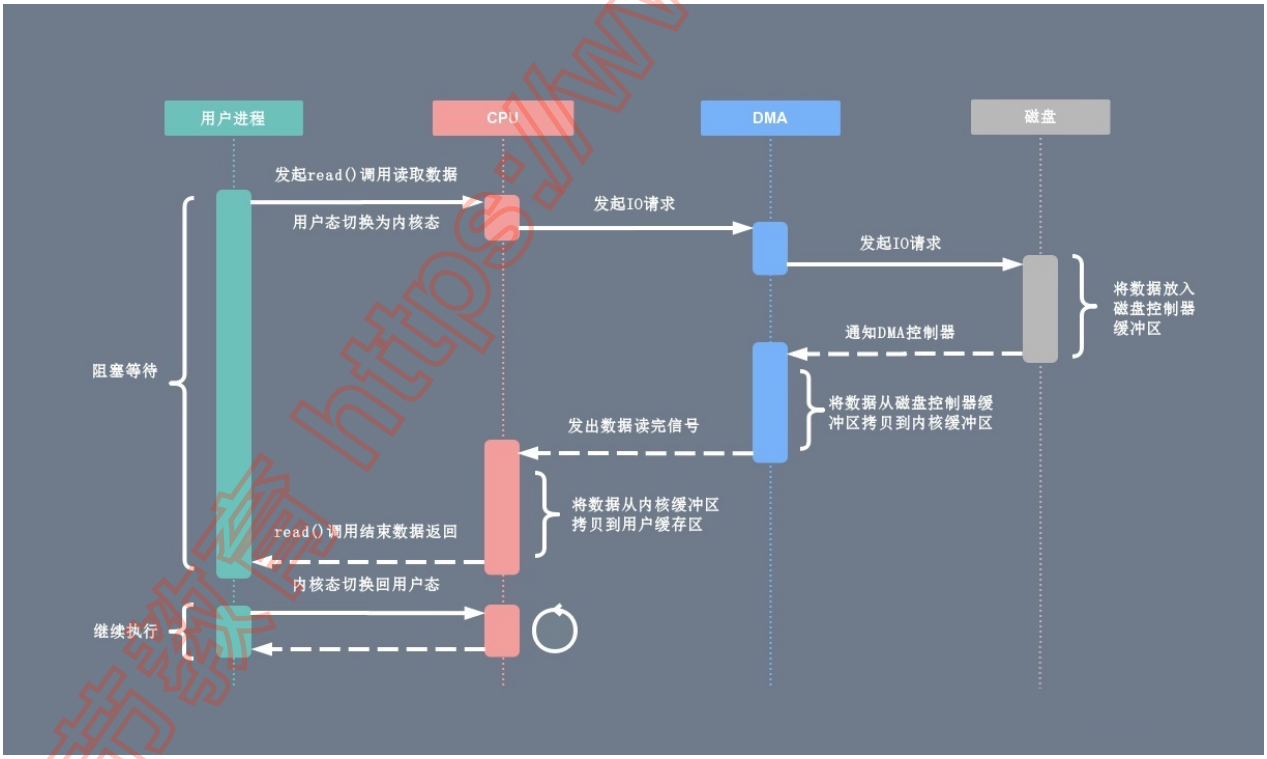
关于 DMA 小知识：

> **`DMA` 的全称叫直接内存存取（`Direct Memory Access`），是一种允许外围设备（硬件子系统）直接访问系统主内存的机制。也就是说，基于 `DMA` 访问方式，系统主内存于硬盘或网卡之间的数据传输可以绕开 `CPU` 的全程调度。目前大多数的硬件设备，包括磁盘控制器、网卡、显卡以及声卡等都支持 `DMA` 技术。**
>
> **整个数据传输操作在一个 `DMA` 控制器的控制下进行的。`CPU` 除了在数据传输开始和结束时做一点处理外（开始和结束时候要做中断处理），在传输过程中 `CPU` 可以继续进行其他的工作。这样在大部分时间里，`CPU` 计算和 `I/O` 操作都处于并行操作，使整个计算机系统的效率大大提高。**

下面即为硬件设备 **不支持/支持 DMA** 时 **CPU** 的工作流程：

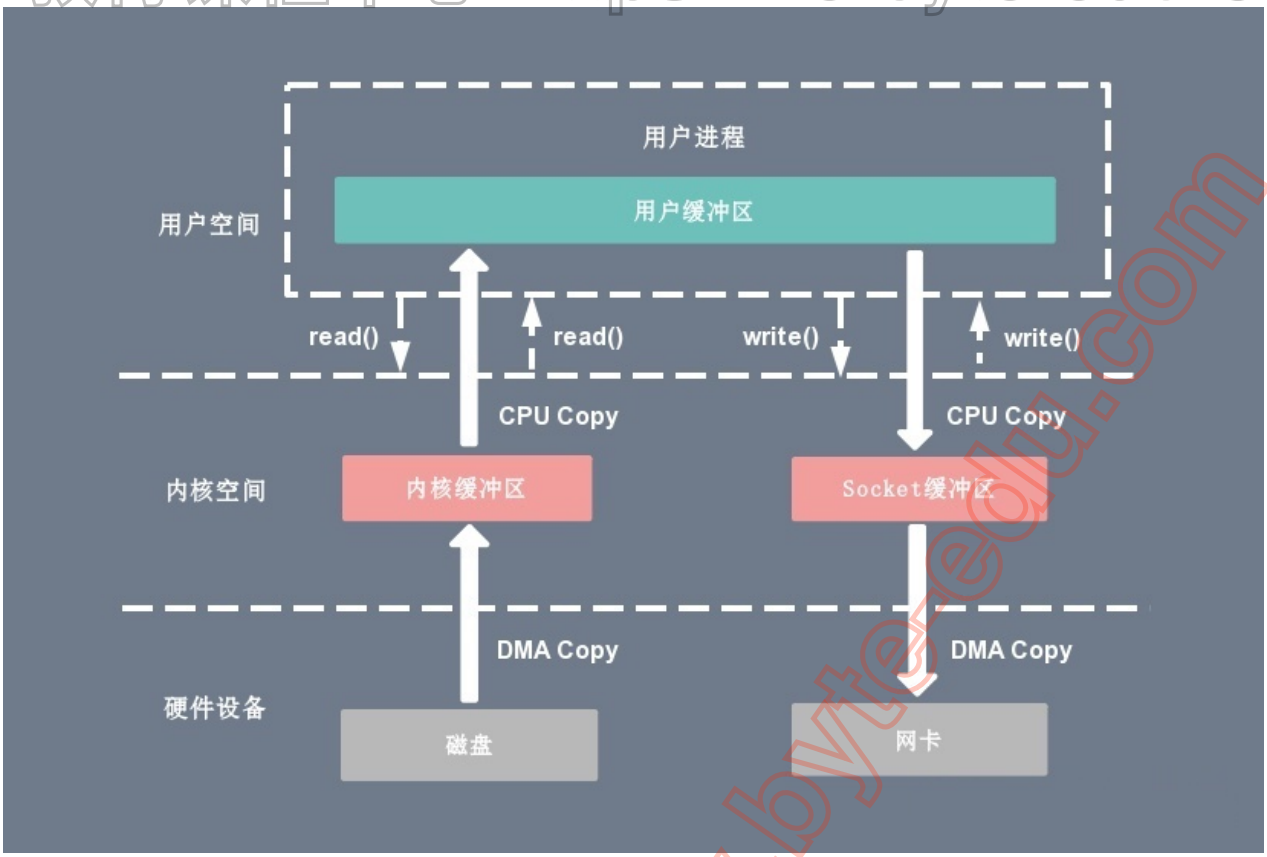


不支持 DMA 的硬件 CPU 工作流程



支持 DMA 的硬件 CPU 工作流程

首先，我们来看下传统硬件支持 `DMA` 控制器的 `I/O` 方式，如果要完成一次数据的接收与发送需要经历哪些过程：

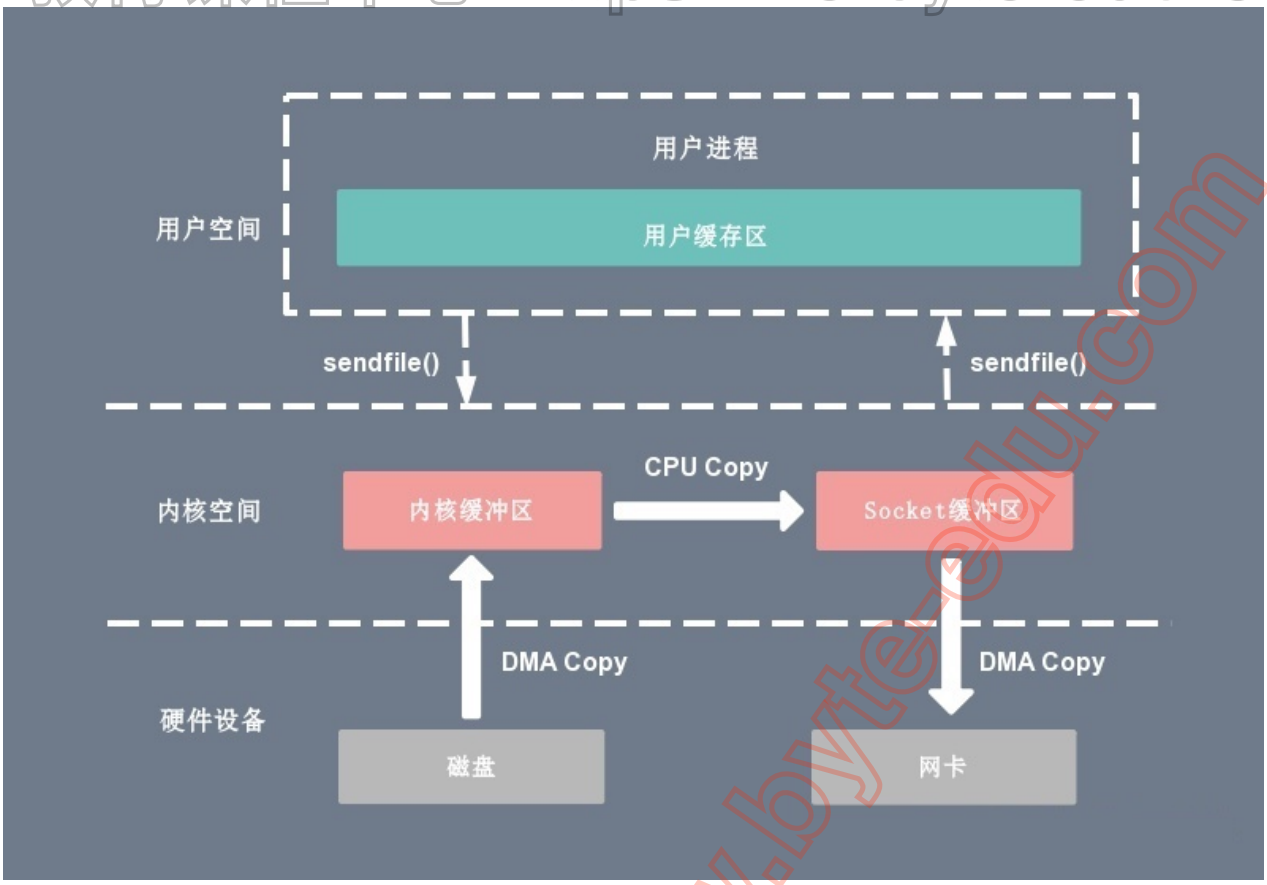


这个步骤流程如下：

- 用户进程向内核发起一次 `read()` 调用，上下文从用户态切换为内核态。（第一次上下文切换）
- `CPU` 通知 `DMA` 控制器从外部存储设备拷贝数据到内核缓冲区。（`DMA Copy`）
- `CPU` 将内核缓冲区的数据拷贝到用户缓冲区，上下文从内核态切换到用户态（`CPU Copy`，第二次上下文切换）
- 用户进程向内核发起一次 `write()` 调用，上下文从用户态切换到内核态。（第三次上下文切换）
- `CPU` 从用户缓冲区拷贝数据到内核空间的 `Socket` 缓冲区。（`CPU Copy`）
- `CPU` 通知 `DMA` 控制器将 `Socket` 缓冲区的数据拷贝到网卡进行传输，上下文从内核态切换为用户态并返回。（`DMA Copy`，第四次上下文切换）

传统的 I/O 方式在进行一次读写的时候共涉及了 4 次上下文切换，2 次 DMA 拷贝以及 2 次 CPU 拷贝。

为了减少这种频繁的上下文切换以及数据拷贝，`Linux kernel-2.1` 引入了多种实现方案，我们这里主要介绍与 `Nginx` 相关的，其中一种方案就是 `sendfile` 技术，原理如下：



用户程序通过 `sendfile` 系统调用，数据可以直接在内核空间内部进行 `I/O` 传输，从而省去了数据在用户空间和内核空间之间的来回拷贝。

基于 "sendfile" 系统调用的零拷贝方式，整个过程会发生 2 次上下文切换，1 次 CPU 拷贝和 2 次 DMA 拷贝

`Sendfile` 系统调用方式下，用户程序读写数据的流程如下：

- 用户进程通过 `sendfile()` 函数向内核（`kernel`）发起系统调用，上下文从用户态（`user space`）切换为内核态（`kernel space`）。
- `CPU` 利用 `DMA` 控制器将数据从主存或硬盘拷贝到内核空间（`kernel space`）的读缓冲区（`read buffer`）。
- `CPU` 将读缓冲区（`read buffer`）中的数据拷贝到的网络缓冲区（`socket buffer`）。
- `CPU` 利用 `DMA` 控制器将数据从网络缓冲区（`socket buffer`）拷贝到网卡进行数据传输。
- 上下文从内核态（`kernel space`）切换回用户态（`user space`），`sendfile` 系统调用执行返回。

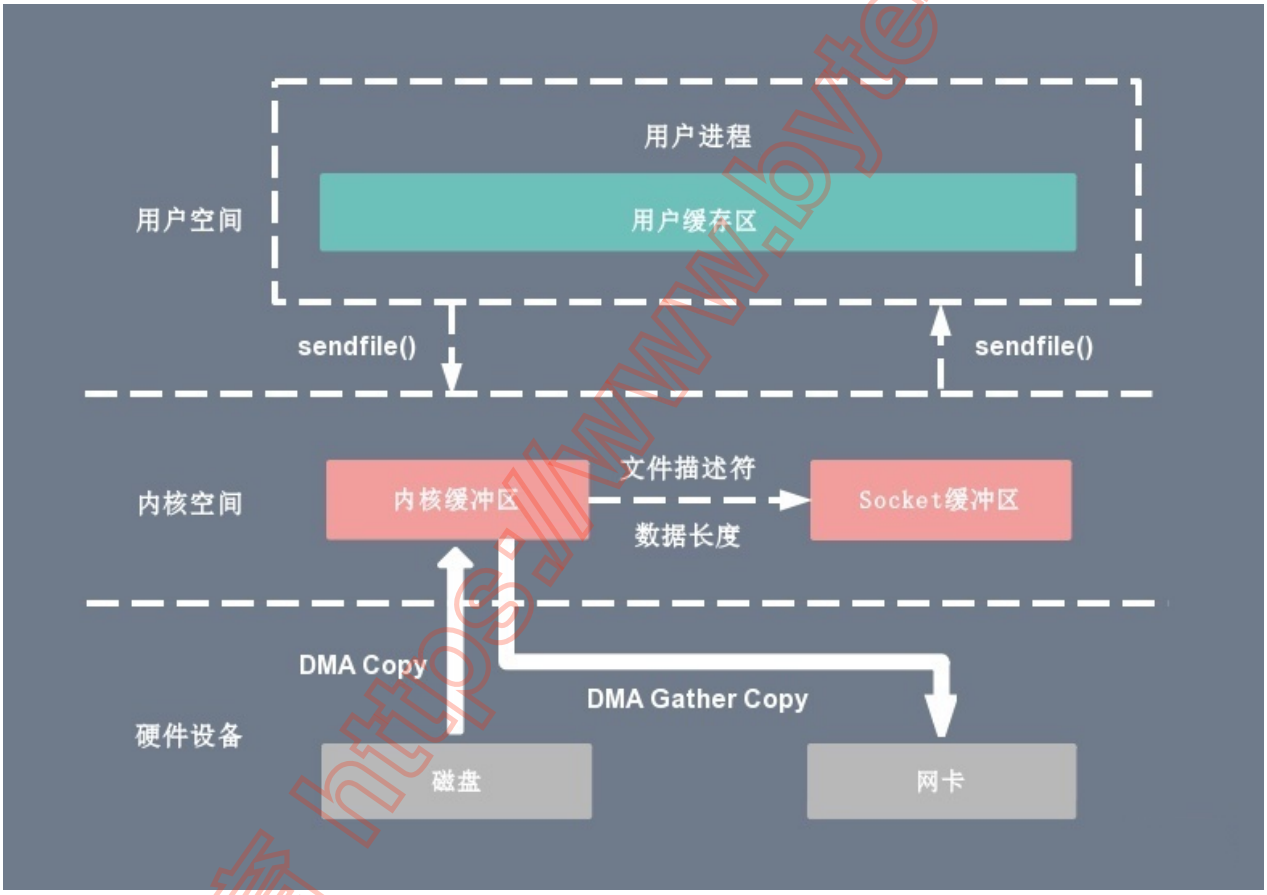
相比较于传统 I/O 方式，`sendfile` 减少了上下文切换和数据拷贝次数，但是 `sendfile` 存在的问题是用户程序不能对数据进行修改（比如 `Nginx` 通过对文件进行压缩以实现网络高速传输就不能使用 `sendfile`），而只是单纯地完成了一次数据传输过程。

> **Nginx 中打开 sendfile 配置选项即可享受 sendfile 系统调用所带来的优化，但仍需注意该优化仅针对静态资源处理有效，对于反向代理并不起作用，这是因为 sendfile 中数据源句柄只能是文件句柄，而反向代理的双端都是 socket 句柄，导致不能使用 sendfile。**

> 由于 sendfile 系统调用导致数据不经过 user space ，因此与 nginx 中 output-filter 存在冲突，比如在同一个上下文中同时打开 gzip 与 sendfile 会导致 sendfile 失效。

另外，在 sendfile 实现方案中，如果硬件支持 DMA Gather 操作（ Linux 2.4 版本的内核对 sendfile 系统调用进行修改，为 DMA 拷贝引入了 gather 操作），sendfile 拷贝方式不再从内核缓冲区的数据拷贝到 socket 缓冲区，取而代之的仅仅是缓冲区文件描述符和数据长度的拷贝，这样 DMA 引擎直接利用 gather 操作将页缓存中数据打包发送到网络中即可.

> DMA gather 将内核空间（ kernel space ）的读缓冲区（ read buffer ）中对应的数据描述信息（内存地址、地址偏移量）记录到相应的网络缓冲区（ socket buffer ）中，由 DMA 根据内存地址、地址偏移量将数据批量地从读缓冲区（ read buffer ）拷贝到网卡设备中，这样就省去了内核空间中仅剩的 1 次 CPU 拷贝操作，如下图：



基于 sendfile + DMA gather copy 系统调用的零拷贝方式，整个拷贝过程会发生 2 次上下文切换、0 次 CPU 拷贝以及 2 次 DMA 拷贝。

用户程序读写数据的流程如下：

- 用户进程通过 sendfile() 函数向内核（ kernel ）发起系统调用，上下文从用户态（ user space ）切换为内核态（ kernel space ）。
- CPU 利用 DMA 控制器将数据从主存或硬盘拷贝到内核空间（ kernel space ）的读缓冲区（ read buffer ）。
- CPU 把读缓冲区（ read buffe r）的文件描述符（ file descriptor ）和数据长度拷贝到网络缓冲区（ socket buffer ）。

- 基于已拷贝的文件描述符（`file descriptor`）和数据长度，`CPU` 利用 `DMA` 控制器的 `gather/scatter` 操作直接批量地将数据从内核的读缓冲区（`read buffer`）拷贝到网卡进行数据传输。
- 上下文从内核态（`kernel space`）切换回用户态（`user space`），`sendfile()` 系统调用执行返回。

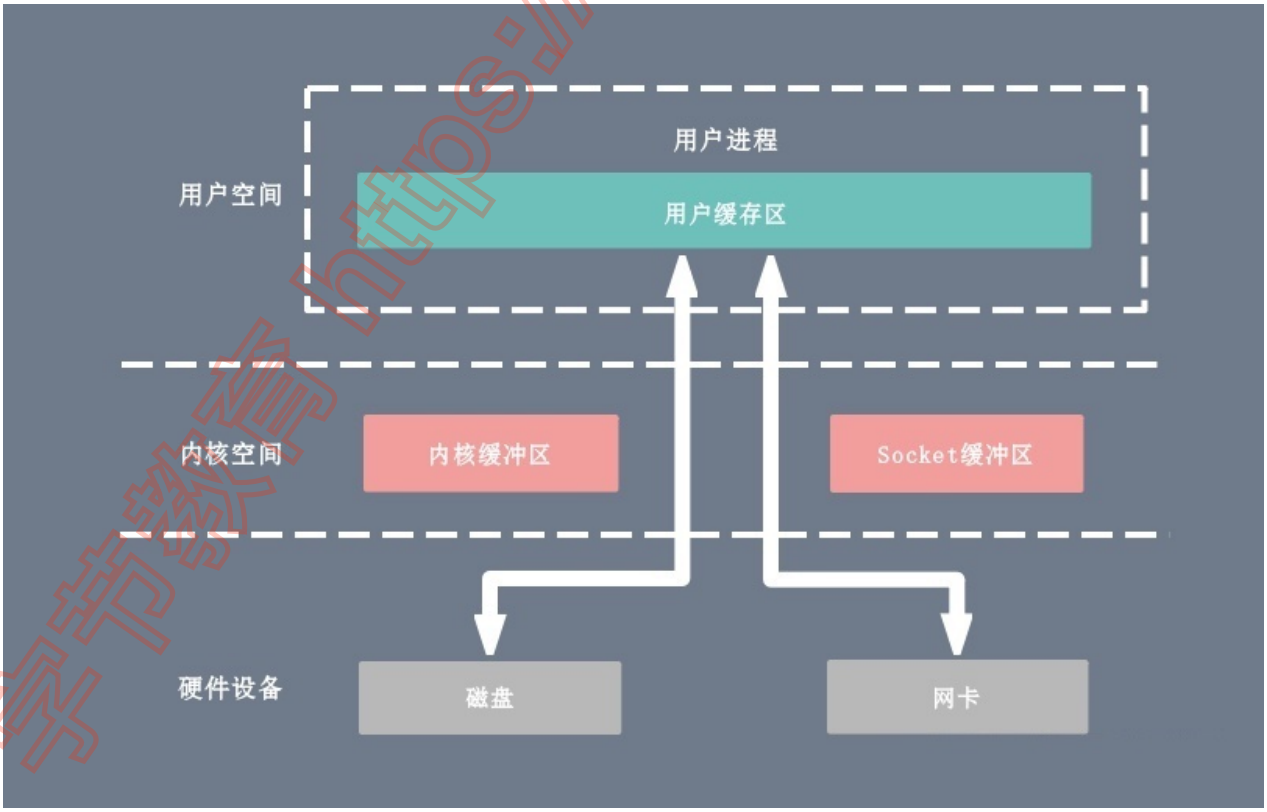
`sendfile + DMA gather copy` 拷贝方式同样存在用户程序不能对数据进行修改的问题，而且本身需要硬件的支持。

【优化读写速度 之 direct I/O】

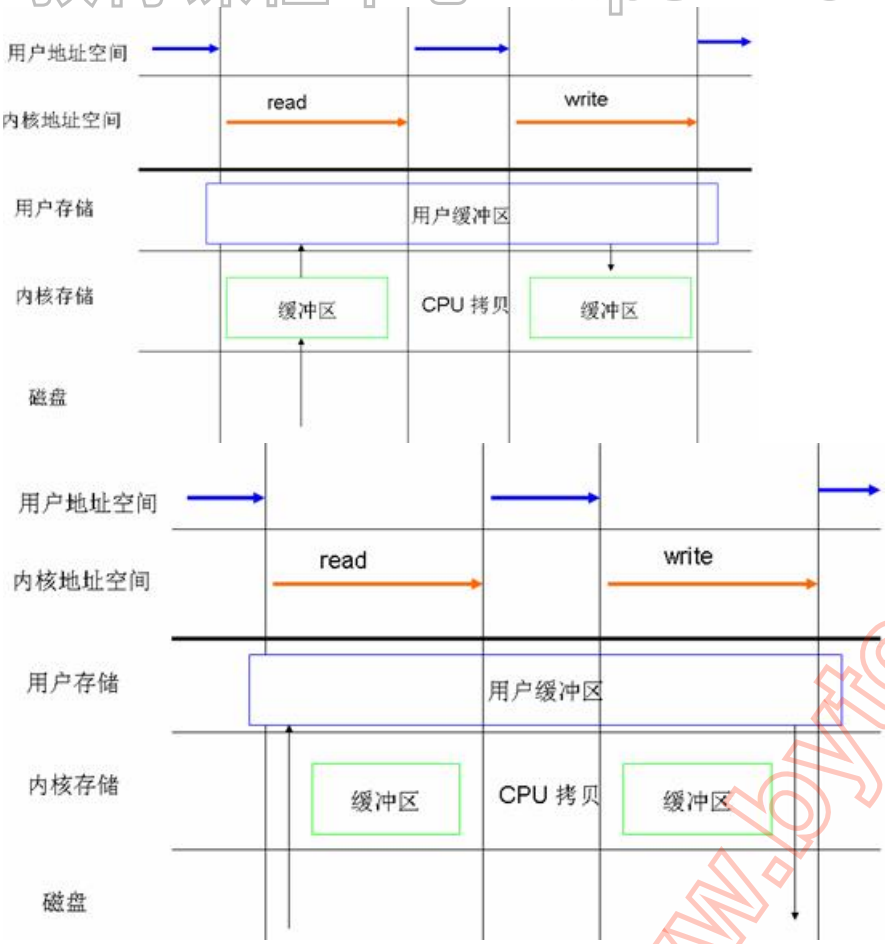
我们知道一般数据 **I/O** 时，一定会经过内核缓冲区、应用缓冲区，这是因为缓冲区的速度往往较 **IO** 设备高很多，可以极大程度提升数据的读写效率。但是有些场景、也有些应用不适合使用。在 `Nginx` 中也是支持 `Direct I/O` 模式，相关的参数为 **directio**，应用的常见主要是读取大文件时。基本语法如下：

```
Syntax: directio size | off;
Default:  directio off;
Context:  http, server, location
This directive appeared in version 0.7.7.
```

凡是通过直接 **I/O** 方式进行数据传输，数据均直接在用户地址空间的缓冲区和磁盘之间直接进行传输，完全不需要页缓存的支持。操作系统层提供的缓存往往会使应用程序在读写数据的时候获得更好的性能，但是对于某些特殊的应用程序，比如说数据库管理系统这类应用，他们更倾向于选择他们自己的缓存机制，因为数据库管理系统往往比操作系统更了解数据库中存放的数据，数据库管理系统可以提供一种更加有效的缓存机制来提高数据库中数据的存取性能。



下图是 **标准访问文件方式** 和 **Direct I/O 的访问方式** 流程图对比：



标准访问文件方式 Direct I/O 的访问方式

> **`Linux 2.6` 内核中直接 `I/O` 的设计与实现**
>
> **在块设备或者网络设备中执行直接 `I/O` 完全不用担心实现直接 `I/O` 的问题，`Linux 2.6` 操作系统内核中高层代码已经设置和使用了直接 `I/O`，驱动程序级别的代码甚至不需要知道已经执行了直接 `I/O`；但是对于字符设备来说，执行直接 `I/O` 是不可行的，`Linux 2.6` 提供了函数 `get_user_pages()` 用于实现直接 `I/O`。**

`Direct I/O` 的特点就是 **数据传输不经过操作系统内核缓冲区**。

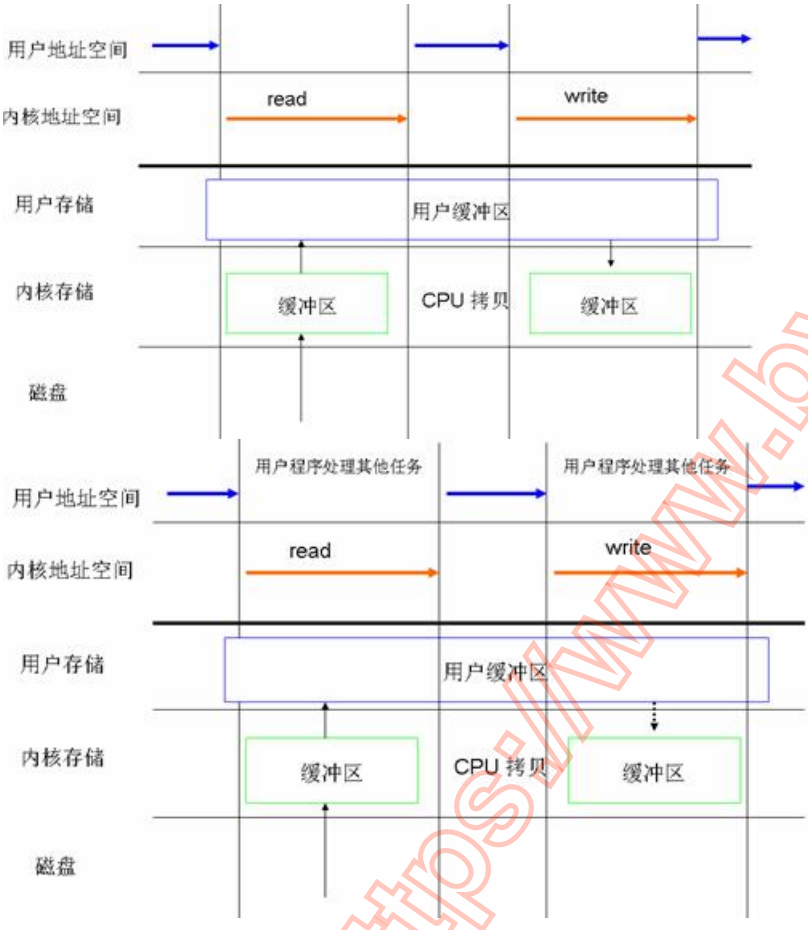
用户态 `Direct I/O` 使得应用进程或运行在用户态（`user space`）下的库函数直接访问硬件设备，数据直接跨过内核进行传输，内核在数据传输过程除了进行必要的虚拟存储配置工作之外，不参与任何其他工作，这种方式能够直接绕过内核，极大提高了性能。用户态 `Direct I/O` 只能适用于不需要内核缓冲区处理的应用程序，这些应用程序通常在进程地址空间有自己的数据缓存机制，称为自缓存应用程序，如数据库管理系统就是一个代表。其次，这种零拷贝机制会直接操作磁盘 `I/O`，由于 `CPU` 和磁盘 `I/O` 之间的执行时间差距，会造成大量资源的浪费，解决方案是配合异步 `I/O` 使用。

【优化读写速度 之 异步 I/O】

`Linux` 异步 `I/O` 是 `Linux 2.6.22` 中的一个标准特性，`CPU` 处理其他任务和 `I/O` 操作可以重叠执行，在 `Nginx` 中启用异步 `I/O` 的参数为 **`aio`**。基本语法如下：

```
Syntax: aio on | off | threads[=pool];
Default:  aio off;
Context:  http, server, location
This directive appeared in version 0.8.11.
```

其本质思想就是进程发出数据传输请求之后，进程不会被阻塞，也不用等待任何操作完成，进程可以在数据传输的时候继续执行其他的操作。相对于同步访问文件的方式来说，异步访问文件的方式可以提高应用程序的效率，并且提高系统资源利用率。直接 `I/O` 经常会和异步访问文件的方式结合在一起使用。



标准访问文件方式 异步 I/O 的访问方式

小结：

在 `Nginx` 中我们会把 `directio` 与 异步 `I/O` 即 `aio` 一起使用，以免造成数据读取阻塞。

在 `Nginx` 中我们同样会将 `sendfile` 与 异步 `I/O` 一起使用， `aio` 会为 `sendfile` 提前预加载数据。

在 `Nginx` 中在使用 `directio` 时会自动禁用 `sendfile` 。

使用示例：

```
location /video/ {
    sendfile    on;
    aio         on;
    directio    8m;
}
```

表示当文件大小超过 `8M` 时，启动 `AIO` 与 `Direct I/O`，此时的 `sendfile` 会自动禁用；当文件小于 `8M` 时，`AIO` 与 `sendfile` 一起使用，此时 `Direct I/O` 不生效。

关于 `Linux` 直接 `I/O` 相关内容介绍，可以参考 文档，以及 `Linux` 零拷贝技术 相关文档

【静态内容缓存优化】

在 `Nginx` 中我们可以使用 `open_file_cache` 进一步提高性能，`Nginx` 缓存将最近使用的文件描述符和相关元数据（如修改时间，大小等）、目录结构信息等存储在缓存中。需要注意的是缓存不会存储所请求文件的内容。

> **可以缓存的信息有：**
>
> ```
> open file descriptors, their sizes and modification times;
> information on existence of directories;
> file lookup errors, such as “file not found”, “no read permission”, and so on
> ```

如果要设置文件缓存，需要设置该参数 **open_file_cache**，基本语法：

```
Syntax: open_file_cache off;
        open_file_cache max=N [inactive=time];
Default:  open_file_cache off;
Context:  http, server, location
```

该参数默认没有启用，如果要启用的话可以设置 `open_file_cache max=N [inactive=time]`，其中：

`max`：用来设置缓存项的上限，当达到上限后将采用 `LRU` 算法进行管理；

`inactive`：缓存项的非活动时长，在指定的时长内未被命中的即为非活动项，将被删除；另外，下面还有一个参数 **open_file_cache_min_uses** 与该处的非活动时长有关，该参数表示在设置的非活动时长内，命中的次数少于 `open_file_cache_min_uses` 指定的次数，也会被删除；

open_file_cache_min_uses：不用解释了吧，默认值为 1，基本语法：

```
Syntax: open_file_cache_min_uses number;
Default:  open_file_cache_min_uses 1;
Context:  http, server, location
```

我们知道缓存的内容可以是查找文件时的一些错误信息，但是这需要我们手动设置，该参数为：

open_file_cache_errors，基本语法：

```
Syntax: open_file_cache_errors on | off;
Default:  open_file_cache_errors off;
Context:  http, server, location
```

另外，**open_file_cache_valid** 用来指定缓存有效性的检查频率，默认为 `60s`，基本语法如下：

```
Syntax: open_file_cache_valid time;
Default:  open_file_cache_valid 60s;
Context:  http, server, location
```

综上，结合上面的参数内容结束，生产上我们可以这样配置：

```
open_file_cache          max=10240 inactive=20s;
open_file_cache_valid    30s;
open_file_cache_min_uses 2;
open_file_cache_errors   on;
```

> **需要根据自己的实际业务做适当调整。**

【客户端请求缓存优化】

还有一些参数，是客户端对服务端发起请求时，服务端同样会把请求的头部以及 `body` 信息先放到缓存区中，相关的一些配置参数有：

client_header_buffer_size：设置客户端请求报文首部缓冲区大小，默认值为 `1K`。有时客户端请求首部带有 `cookie` 很大的信息，会造成 `400` 错误，强烈建议增大大小。配置语法：

```
Syntax: client_header_buffer_size size;
Default:  client_header_buffer_size 1k;
Context:  http, server
```

我们一般将该缓冲区大小设置为系统分页大小，可以使用 `getconf PAGESIZE` 来进行查看：

```
[root@web-nginx conf.d]# getconf PAGESIZE
4096
```

所以我们可以这样设置 `client_header_buffer_size 4k;`

需要注意的是，如果我们设置的 `client_header_buffer_size` 大小不够用户存储客户端请求头，那么会使用：

large_client_header_buffers：该参数在 `client_header_buffer_size` 不足时使用，动态按需分配。默认配置：

```
Syntax: large_client_header_buffers number size;
Default:  large_client_header_buffers 4 8k;
Context:  http, server
```

一般使用默认配置即可。

client_body_buffer_size：用来缓存用户请求体的大小，在 `32bit` 系统上默认大小为 `8k`，在 `64bit` 系统上，默认大小为 `16k`，一般我们使用默认值即可；

```
Syntax: client_body_buffer_size size;
Default:  client_body_buffer_size 8k|16k;
Context:  http, server, location
```

当用户请求体 `body` 太大时，则将其暂存在磁盘上由 `client_body_temp_path` 指定的位置，所以为了更好的性能，我们建议使用高性能磁盘来作为存储位置。

client_body_temp_path：基本用法如下：

```
Syntax: client_body_temp_path path [level1 [level2 [level3]]];
Default:  client_body_temp_path client_body_temp;
Context:  http, server, location
```

用来指定存储路径和目录结构，注意目录结构最多为三级结构，每级结构最多支持两位 `16进制` 数量的目录，`1` 表示一个 `16进制`，`2` 表示两个 `16进制`，比如可以这样设定 `1`、`1 2` 或 `1 2 2`，其中：`1 2 2` 表示的含义如下：

1：`1` 级目录占 `1` 位 `16` 进制，可以创建 `16^1` 即 `16` 个目录，目录名称为 `0 ~ f`；

2：`2` 级目录占 `2` 位 `16` 进制，可以创建 `16^2` 即 `256` 个目录，目录名称为 `00 ~ ff`；

2：`3` 级目录占 `2` 位 `16` 进制，可以创建 `16^2` 即 `256` 个目录，目录名称为 `00 ~ ff`；

假设我们设置为 `1 2` 两层目录结构，如下：

```
client_body_temp_path /spool/nginx/client_temp 1 2;
```

那么存储的文件路径可能是这样的：

```
/spool/nginx/client_temp/7/45/00000123457
```

注意看该文件 `hash` 值的后面 **3** 位。如果我们设置为 `3` 级结构，那么该文件存储的位置一定是：

```
/spool/nginx/client_temp/7/45/23/00000123457
```

client_max_body_size：用来设置客户端请求体大小，该大小可以从 `Content-Length` 看到，默认大小是 `1m`，基本上能够满足常见的请求体，如果请求体超过该参数，则会报 `413 (Request Entity Too Large)` 错误。如果 `Nginx` 作为上传文件服务器，那么这个值可能就不够了。需要注意的是：**上传会受限与该参数，但是文件下载跟这个参数没有关系。**

5.2.2 网络 IO 优化方向

如上所述，除了在文件层面进行优化外，我们还可以从网络层面来实现网络 `IO` 性能优化。

【全连接队列长度】

前面的课程中我们介绍了在 `Linux` 中，客户端与服务端建立连接涉及到三个重要的队列，即：

net.core.netdev_max_backlog：接收自网卡，但未被内核协议栈处理的报文队列长度

net.ipv4.tcp_max_syn_backlog：`SYN_RCVD` 状态（即半连接）队列长度

backlog：全连接队列也就是我们图中标注的 `accept 队列`，该队列大小由系统参数和应用参数共同决定，即：**全连接队列的大小取决于：`min(backlog, somaxconn)`，其中 `backlog` 是由应用程序传入，`somaxconn` 是一个 os 级别的系统参数，通过设置 `net.core.somaxconn` 来调整。** 在 `Nginx` 中，`backlog` 参数在 `Listen` 参数后面指定，在 `CentOS` 上，默认值为 `511`，如果并发连接比较多，我们可以适当调大此队列长度，配置语法如下：

```
Syntax: listen address[:port] [backlog=number]
Default:  listen *:80 | *:8000;
Context:  server
```

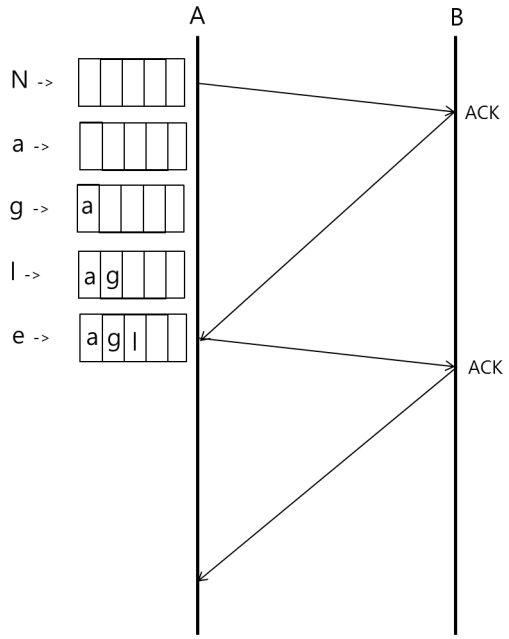
示例：

```
listen    127.0.0.1:80 backlog=4096;
```

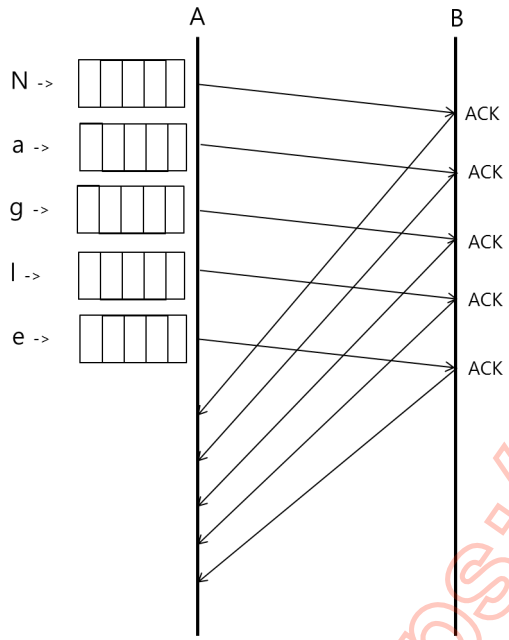
【tcp_nodelay】

该配置参数其实是关于是否启用 `Nagle` 算法（延迟算法），所谓 `Nagle` 算法特性可以简单描述如下：

- **避免同一个连接上同时存在大量小报文**
  - 合并多个小报文一起发送
  - 最多只存在一个小报文（即连接上的最后一个小报文，无其他报文与之合并）
- **提高带宽利用率**



Nagle



Non Nagle (TCP_NODELAY)

启用 `Nagle` 算法 `tcp_nodelay off` 禁用 `Nagle` 算法 `tcp_nodelay on`

所以，是否启用该参数，决定与你的网站业务类型或者你期望侧重于网站的哪些特性，比如：

- **吞吐量优先**：启用 `Nagle` 算法，设置 `tcp_nodelay off`；
- **低时延优先**：禁用 `Nagle` 算法，设置 `tcp_nodelay on`；

在 `Nginx` 的配置模块 `http` 与 `stream` 配置段都有此参数，需要注意的是，在 `http` 配置段中，`tcp_nodelay` 仅对 `HTTP Keep-Alive` 连接有效。

> **Enables or disables the use of the `TCP_NODELAY` option. The option is enabled when a connection is transitioned into the keep-alive state.**

`http` 模块中 `tcp_nodelay` 配置语法：

```
Syntax: tcp_nodelay on | off;
Default:  tcp_nodelay on;
Context:  http, server, location
```

`stream` 模块中 `tcp_delay` 配置语法：

```
Syntax: tcp_nodelay on | off;
Default:  tcp_nodelay on;
Context:  stream, server
This directive appeared in version 1.9.4.
```

另外，在 `Nginx` 的 `http` 配置段，我们也可以通过设置 **postpone_output** 的大小来避免发送小报文，在 `Nginx` 中该参数默认是启用的，设置的报文发送阈值为 `1460` 字节，这个参数和 `Nagle` 类似，也允许带有 `FIN` 标志位的小报文发送。配置语法如下：

```
Syntax: postpone_output size;
Default:  postpone_output 1460;
Context:  http, server, location
```

【tcp_nopush】

该配置参数是用来设置是否启用 `CORK` 算法，`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。启用该算法后，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。

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）。

`CORK` 算法可以理解成一种加强的 `Nagle` 算法，都是累计数据然后发送。但它没有 `Nagle` 中一个小报文的限制，它是可以实现完全禁止所有小报文的发送。

在 `Nginx` 中我们可以通过设置 `tcp_nopush` 参数来决定是否启用 `CORK` 算法，配置语法如下：

```
Syntax: tcp_nopush on | off;
Default:  tcp_nopush off;
Context:  http, server, location
```

需要注意的是，`tcp_nopush` 参数只有在 `sendfile on` 时才有效。

> **The options are enabled only when sendfile is used.**

5.2.3 网络连接层面优化

【HTTP 长连接】

在 `http` 配置段中，默认是启用 `http` 长连接的，我们知道 `TCP` 在建立连接时需要进行三次握手，断开连接需要进行四次挥手，是非常消耗时间的。因此我们希望在 `http` 请求中可以重复使用建立的连接通道，这就是 `http` 长连接，这在连接请求比较多时是非常高效且能够有效降低网络拥塞的一种方案，配置方式如下：

```
Syntax: keepalive_timeout timeout [header_timeout];
Default:  keepalive_timeout 75s;
Context:  http, server, location
```

长连接超时时间默认为 `75s`，如果服务端比较繁忙，可以适当的调小此值，比如设置成 `60s` 或者 `30s` 。不过 `MSIE` 会在 `60s` 左右时主动端口长连接请求。

另外还有一个参数需要注意，就是在一个长连接上允许发送的请求个数，默认是 `100` 个，配置语法如下：

```
Syntax: keepalive_requests number;
Default:  keepalive_requests 100;
Context:  http, server, location
This directive appeared in version 0.8.0.
```

【以 RST 代替四次挥手】

我们知道正常的 `TCP` 连接端口需要经历四次报文的传递，这个过程还是比较耗费时间的，不过为了安全的端口连接，这也是可以接受的。但是有些时候，可能由于客户端的一些错误导致连接关闭，这个时候如果再等待四次挥手就比较耗费时长，而且毫无意义。所以，在 `Nginx` 中有这样一个参数 **reset_timedout_connection** ，可以在连接断开的响应码为 `444` 时，直接以 `RST` 代替正常的四次挥手，快速释放内存。配置语法如下：

```
Syntax: reset_timedout_connection on | off;
Default:  reset_timedout_connection off;
Context:  http, server, location
```

我们只需要设置 `reset_timedout_connection on;` 即可。

【TLS/SSL 优化握手】

我们现在绝大多数的 `Web` 应用都可以使用 `TLS/SSL` 进行加密，但是建立 `TLS/SSL` 连接的速度非常慢（比如密钥协商、密钥交换等），所以如果我们能为客户将加密会话缓存起来，这样就可以极大的提升 `TLS/SSL` 的请求连接速度了。

`Session` 复用，是指将握手时算出来的对称密钥存起来，后续请求中直接使用。在 `TLS/SSL` 会话建立时，最消耗资源的就是通过非对称加密加算出对称密钥来。

在 `Nginx` 中，我们可以使用这个参数 **ssl_session_cache** ，该参数的使用方式如下：

```
Syntax: ssl_session_cache off | none | [builtin[:size]] [shared:name:size];
Default:  ssl_session_cache none;
Context:  http, server
```

可以看到 `ssl_session_cache` 有四个可以使用的值，每个值的含义如下：

`off`：不使用 `session` 缓存，且 `Nginx` 在协议中明确告诉客户端不使用缓存；

`none`：不使用 `session` 缓存，但是 `Nginx` 在协议中告诉客户单可能会使用缓存，但是自身根本不存储会话缓存；

`builtin`：使用 `openssl` 的缓存，由于在内存中使用，所以仅当同一个客户端在两次连接都命中到同一个 `worker` 进程时，`session` 缓存才会生效；

`shared`：定义共享缓存，为所有的 `worker` 进程提供 `session` 缓存服务，官方文档上给出的统计值为 `1MB` 内存大概可以缓存 `4000 session` 会话；所以生产环境单节点中共享缓存的方案使用最多，一般设置大小 `10M - 20M` 足够使用了；比如，我们可以这样定义：

```
ssl_session_cache shared:SSL:10M
```

还有一个参数 **ssl_session_timeout** 用来设置会话复用的超时时间，注意，不是会话的超时时间，而是复用会话的超时时间，默认为 `5m`；生产上可以根据业务不同可以适当的调整此值，比如一个供用户上传下载的网站：

```
ssl_session_timeout 60m;
```

而一个简单的浏览性的网站，可以设置：

```
ssl_session_timeout 10m;
```

需要注意的是，这种方式可以极大的提高 `TLS/SSL` 连接建立速度，但是只能使用与单个节点，如果是集群环境，则需要借助其他的技术手段。比如，我们可以引入 `redis` 来存储会话，`lua-nginx-redis` 可以让我们把 `session id` 的内容放到 `redis` 中，这样在整个 `Nginx` 集群之间就可以复用会话连接。还有一种方式，可以使用另外一个参数：**ssl_session_tickets**，使用语法如下：

```
Syntax: ssl_session_tickets on | off;
Default:  ssl_session_tickets on;
Context:  http, server
This directive appeared in version 1.5.9.
```

启用 `ssl_session_tickets` 会让 `Nginx` 将会话 `session` 中的信息作为 `tickets` 加密发给客户端，当客户端下次发起 `TLS` 连接时带上 `tickets`，再由 `Nginx` 进行解密验证，通过后则恢复用该会话 `session`。会话票证一般会在 `Nginx` 集群中使用，但是破坏了 `TLS/SSL` 的安全机制，有安全风险，必须频繁更换 `tickets` 密钥。加密 `tickets` 的密钥文件配置参数如下：

```
Syntax: ssl_session_ticket_key file;
Default:  —
Context:  http, server
This directive appeared in version 1.5.7.
```

> **注意：** `file` **必须包含 80 或者 48 位的随机数据，我们可以使用这个命令创建：**
>
> ```
> openssl rand 80 > ticket.key
> ```

注意：`ssl_session_cache` 与 `ssl_session_tickets` 两种解决方案最好不要同时使用。

5.2.4 模块层面优化

除了上面提到的优化方案，我们还可以通过引入一些模块，让 `Nginx` 具备更好的性能，我们介绍几个比较常用的模块。

【ngx_http_gzip_module】

该模块在 `Nginx` 中默认已经被编译进去，可以直接使用，如果不想使用，也可以在编译的时候通过 `--without-http_gzip_module` 选项来禁止加载该模块。`ngx_http_gzip_module` 模块通过设置参数 `gzip on` 启用，用来实时压缩 `http` 包体，以提升网络传输效率。看下在 `Nginx` 该如何配置 `gzip` 以及相关参数，如下：

```
Syntax: gzip on | off;
Default:  gzip off;
Context:  http, server, location, if in location
```

除此之外，还有一些与 `gzip` 相关且较常用的参数，比如：

gzip_min_length ：用来指定当报文长度达到多少字节时才启用压缩，默认是 20 字节，这个值我们一般需要调大，比如 `gzip_min_length 1024`；基本用法：

```
Syntax: gzip_min_length length;
Default:  gzip_min_length 20;
Context:  http, server, location
```

gzip_types ：指定对何种响应报文进行压缩，默认为 `text/html`，不用显式指定，否则可能会报错，基本语法如下：

```
Syntax: gzip_types mime-type ...;
Default:  gzip_types text/html;
Context:  http, server, location
```

一般生产上，我们会调整压缩类型，比如：

```
gzip_types text/plain text/css text/xml application/javascript
application/json application/xml+rss;
```

> **注意1：支持的压缩资源我们可以通过 `/etc/nginx/mime.types` （通过 `yum` 方式安装）查找。**
>
> **注意2：我们不要去压缩图片、视频等静态资源，因为这些资源已经被压缩过了，再压缩一遍也不会变小，白白浪费 `CPU` 资源。**

gzip_comp_level：指定压缩级别，可以支持 `1-9` 压缩级别，级别越高（数字越大）压缩程度越高，同时耗时越长，相应的消耗的 `CPU` 资源越高。基本语法如下：

```
Syntax: gzip_comp_level level;
Default:  gzip_comp_level 1;
Context:  http, server, location
```

一般我们不建议将压缩级别设置太高，会消耗 `CPU` 资源，如下：

```
gzip_comp_level 2;
```

gzip_disable：设置对于哪些客户端不进行压缩，判断依据是根据字段 `User-Agent` 进行正则匹配，基本语法如下：

```
Syntax: gzip_disable regex ...;
Default:  —
Context:  http, server, location
This directive appeared in version 0.6.23.
```

比如，我们可以这样设置：

```
gzip_disable "MSIE[1-6]\.";
```

禁止对客户端为 `IE1-IE6` 进行报文压缩。

gzip_vary：用来设置当浏览器启用 `gzip`、`gzip_static`、`gunzip` 模块时，是否在响应报文中显示 `Vary: Accept-Encoding`；生产上如果不想显示，就设置 `gzip_vary off;` 即可，基本语法如下：

```
Syntax: gzip_vary on | off;
Default:  gzip_vary off;
Context:  http, server, location
```

除此之外还有 **gzip_buffers** 用来设置压缩报文时的缓冲区数量和大小，一般大小设置为1个内存页大小，使用默认值即可：

```
Syntax: gzip_buffers number size;
Default:  gzip_buffers 32 4k|16 8k;
Context:  http, server, location
```

gzip_http_version：设置最小支持压缩的 `http` 协议版本号，默认为 `http 1.1`，基本语法：

```
Syntax: gzip_http_version 1.0 | 1.1;
Default:  gzip_http_version 1.1;
Context:  http, server, location
```

这个值一般不用配置，使用默认即可，但是下面还有一个非常重要的参数，即：

gzip_proxied：设置对上游服务器的响应进行压缩处理，基本语法如下：

```
Syntax: gzip_proxied off | expired | no-cache | no-store | private |
no_last_modified | no_etag | auth | any ...;
Default:  gzip_proxied off;
Context:  http, server, location
```

可以使用的值如下：

`off`：不对上游服务器的响应进行压缩；

`expired`：如果上游服务器响应中含有 `Expires` 字段，且其值中间的时间与系统时间对比确定不会缓存，则压缩响应；

`no-cache`、`no-store`、`private`：如果上游服务器响应报文含有 `Cache-Control` 首部，并且其值含有相应的 `no-cache`、`no-store`、`private` 值，则压缩响应；

`no_last_modified`：如果上游服务器响应中没有 `Last-Modified` 头部，则压缩响应；

`no_etag`：如果上游服务器响应中没有 `ETag` 头部，则压缩响应；

`auth`：如果 **客户端请求报文** 中含有 `Authorization` 头部，则压缩响应；

`any`：压缩所有来自上游的响应；

注意：一般情况下，如果上游服务器是我们自己配置的，最好在上游服务器配置中觉得是否进行压缩；如果是代理加速别人的上游节点，则最好不要压缩响应，因为有些客户端是不支持压缩的，我们在代理层压缩之后很可能导致客户端访问源站报错。

虽然，我们使用 `ngx_http_gzip_module` 模块将数据压缩之后进行响应可以提高网络响应速度，但是由于对数据进行压缩导致我们调用 `sendfile` 方法不再生效，所以一个比较好的方式是先将文件进行压缩好，当客户端请求时直接用这个压缩好的文件进行响应。这里就需要使用另外一个模块 `ngx_http_gzip_static_module`，这个模块不是 `Nginx` 默认模块，但是在 `EPEL` 源中的版本已经帮我们把他编译进来了，如果是自己编译的话，只需要加上 `--with-http_gzip_static_module` 即可。

【ngx_http_gzip_static_module】

通过参数 **gzip_static** 设置该模块，基础语法如下：

```
Syntax: gzip_static on | off | always;
Default:  gzip_static off;
Context:  http, server, location
```

可以使用的值有：

`on` 或者 `off`：用来决定检查是否存在预压缩文件（以文件名 + `.gz` 结尾，如果存在则发送），设置成 `on` 还会根据客户端是否支持 `gzip` 解压缩来确定是否发送压缩文件；

`always`：无论客户端是否支持 `gzip` 解压缩，均会检查 `.gz` 文件是否存在，存在就发送。这种情况在磁盘上没有未压缩的文件时，结合 `ngx_http_gunzip_module` 模块一起使用。

> **注意1：`always` 需要结合 `ngx_http_gunzip_module` 模块一起使用！**
>
> **注意2：如果我们想要使用 `gzip_static` 模块，需要使用 `gzip` 提前压缩好相应的文件。**
>
> **The files can be compressed using the `gzip` command, or any other compatible one. It is recommended that the modification date and time of original and compressed files be the same.**

【ngx_http_gunzip_module】

该模块需要我们自己编译，使用 `--with-http_gunzip_module`，该模块的主要功能是当客户端不支持 `gzip` 解压时，并且磁盘上仅有压缩文件，则将文件实时解压并发送给客户端。该模块只有两个相关的配置参数：

gunzip：用来指定是否启用该模块，基础语法如下：

```
Syntax: gunzip on | off;
Default:  gunzip off;
Context:  http, server, location
```

还有一个是配置 `gunzip` 使用的缓存区数量与大小，该参数与系统平台相关，即 **gunzip_buffers**，我们保持默认即可，基本语法如下：

```
Syntax: gunzip_buffers number size;
Default:  gunzip_buffers 32 4k|16 8k;
Context:  http, server, location
```

> **注意：`gunzip` 模块会在客户端不支持 `gzip` 压缩并且服务端依然返回 `.gz` 文件时才会启用，这就要求我们在 `gzip_static` 必须设置为 `always`，否则一旦检测到客户端不支持 `gzip` 压缩，则不会发送压缩文件（因为 `gzip_static` 处理阶段在 `gunzip` 之前），此时如果服务端仅有压缩文件，就会报 `404` 错误，需要特别注意！**

另外还有一些优化参数，我们需要结合应用场景以及自身业务去调整，等介绍到相关模块后再做细述，所以，综上所述，在 `Nginx` 的 `http` 模块配置段，上述内容即为比较通用的优化参数，我们可以在自己的网站中进行调整后应用。下面的课程我们将重点介绍 `Nginx` 的相关应用、配置等内容，其中也会涉及到一些相关的调优参数。