



下载APP



43 | Socket通信：遇上特大项目，要学会和其他公司合作

2019-07-05 刘超

趣谈Linux操作系统

[进入课程 >](#)



讲述：刘超

时长 10:20 大小 8.29M



上一篇预习文章说了这么多，现在我们终于可以来看一下，在应用层，我们应该如何使用 socket 的接口来进行通信。

如果你对 socket 相关的网络协议原理不是非常了解，建议你先去看一看上一篇的预习文章，再来看这一篇的内容，会比较轻松。

按照前一篇文章说的分层机制，我们可以想到，socket 接口大多数情况下操作的是传输层，更底层的协议不用它来操心，这就是分层的好处。

在传输层有两个主流的协议 TCP 和 UDP，所以我们的 socket 程序设计也是主要操作这两个协议。这两个协议的区别是什么呢？通常的答案是下面这样的。

TCP 是面向连接的，UDP 是面向无连接的。

TCP 提供可靠交付，无差错、不丢失、不重复、并且按序到达；UDP 不提供可靠交付，不保证不丢失，不保证按顺序到达。

TCP 是面向字节流的，发送时发的是一个流，没头没尾；UDP 是面向数据报的，一个一个的发送。

TCP 是可以提供流量控制和拥塞控制的，既防止对端被压垮，也防止网络被压垮。

这些答案没有问题，但是没有到达本质，也经常让人产生错觉。例如，下面这些问题，你看看你是否了解？

所谓的连接，容易让人误以为，使用 TCP 会使得两端之间的通路和使用 UDP 不一样，那我们会在沿途建立一条线表示这个连接吗？

我从中国访问美国网站，中间这么多环节，我怎么保证连接不断呢？

中间有个网络管理员拔了一根网线不就断了吗？我不能控制它，它也不会通知我，我一个个人电脑怎么能够保持连接呢？

还让我做流量控制和拥塞控制，我既管不了中间的链路，也管不了对端的服务器呀，我怎么能够做到？

按照网络分层，TCP 和 UDP 都是基于 IP 协议的，IP 都不能保证可靠，说丢就丢，TCP 怎么能够保证呢？

IP 层都是一个包一个包的发送，TCP 怎么就变成流了？

从本质上来讲，所谓的**建立连接**，其实是为了在客户端和服务端维护连接，而建立一定的数据结构来维护双方交互的状态，并用这样的数据结构来保证面向连接的特性。TCP 无法左右中间的任何通路，也没有什么虚拟的连接，中间的通路根本意识不到两端使用了 TCP 还是 UDP。


所谓的**连接**，就是两端数据结构状态的协同，两边的状态能够对上。符合 TCP 协议的规则，就认为连接存在；两面

状态对不上，连接就算断了。

流量控制和拥塞控制其实就是根据收到的对端的网络包，调整两端数据结构的状态。TCP 协议的设计理论上认为，这样调整了数据结构的状态，就能进行流量控制和拥塞控制了，其实在通路上是不是真的做到了，谁也管不着。

所谓的**可靠**，也是两端的数据结构做的事情。不丢失其实是数据结构在“点名”，顺序到达其实是数据结构在“排序”，面向数据流其实是数据结构将零散的包，按照顺序捏成一个流发给应用层。总而言之，“连接”两个字让人误以为功夫在通路，其实功夫在两端。

当然，无论是用 socket 操作 TCP，还是 UDP，我们首先都要调用 socket 函数。

 复制代码

```
1 int socket(int domain, int type, int protocol);
```

socket 函数用于创建一个 socket 的文件描述符，唯一标识一个 socket。我们把它叫作文件描述符，因为在内核中，我们会创建类似文件系统的数据结构，并且后续的操作都有用到它。

socket 函数有三个参数。

domain: 表示使用什么 IP 层协议。AF_INET 表示 IPv4, AF_INET6 表示 IPv6。

type: 表示 socket 类型。SOCK_STREAM, 顾名思义就是 TCP 面向流的, SOCK_DGRAM 就是 UDP 面向数据报的, SOCK_RAW 可以直接操作 IP 层, 或者非 TCP 和 UDP 的协议。例如 ICMP。

protocol 表示的协议, 包括 IPPROTO_TCP、IPPROTO_UDP。

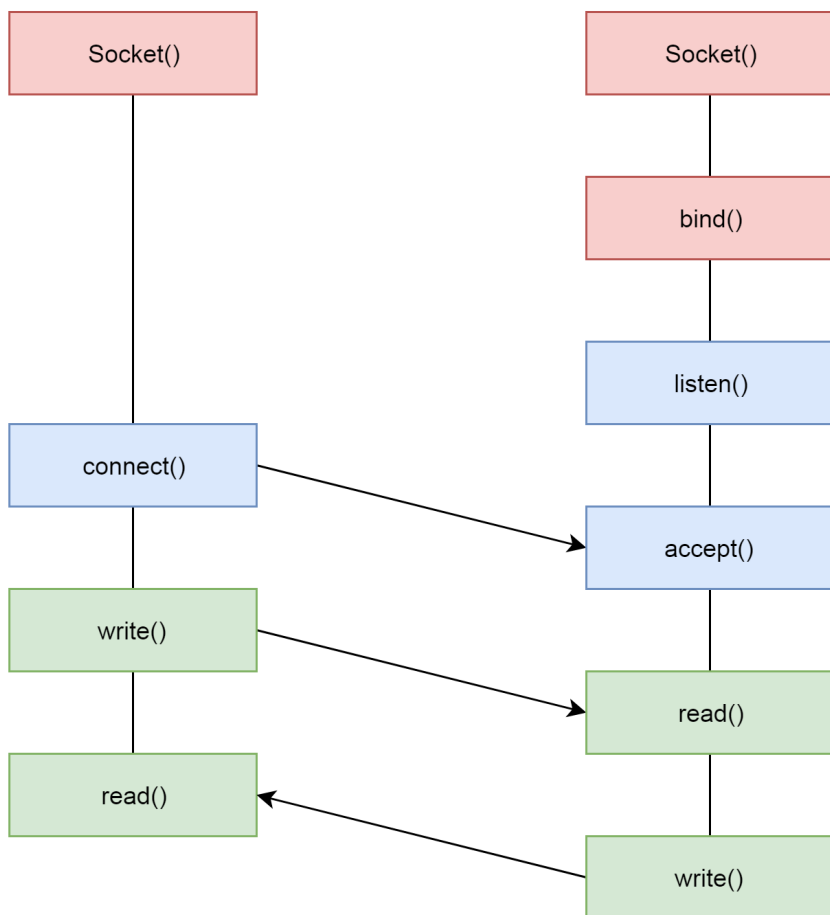
通信结束后, 我们还要像关闭文件一样, 关闭 socket。

针对 TCP 应该如何编程?

接下来我们来看, 针对 TCP, 我们应该如何编程。

客户端

服务端



TCP 的服务端要先监听一个端口，一般是先调用 `bind` 函数，给这个 `socket` 赋予一个端口和 IP 地址。

```

1 int bind(int sockfd, const struct sockaddr *addr, socklen_t
2
3 struct sockaddr_in {
4     __kernel_sa_family_t  sin_family;      /* Address family
5     __be16                 sin_port;        /* Port number
6     struct in_addr         sin_addr;        /* Internet address
7
8     /* Pad to size of `struct sockaddr'. */
9     unsigned char          __pad[__SOCK_SIZE__ - sizeof(st
10                                sizeof(unsigned short int) - si
11 };
12
13 struct in_addr {
14     __be32  s_addr;
15 };

```

其中，`sockfd` 是上面我们创建的 `socket` 文件描述符。在 `sockaddr_in` 结构中，`sin_family` 设置为 `AF_INET`，表示 IPv4；`sin_port` 是端口号；`sin_addr` 是 IP 地址。

服务端所在的服务器可能有多个网卡、多个地址，可以选择监听在一个地址，也可以监听 `0.0.0.0` 表示所有的地址都监听。服务端一般要监听在一个众所周知的端口上，例如，Nginx 一般是 80，Tomcat 一般是 8080。

客户端要访问服务端，肯定事先要知道服务端的端口。无论是电商，还是游戏，还是视频，如果你仔细观察，会发现都

有一个这样的端口。可能你会发现，客户端不需要 bind，因为浏览器嘛，随机分配一个端口就可以了，只有你主动去连接别人，别人不会主动连接你，没有人关心客户端监听到了哪里。

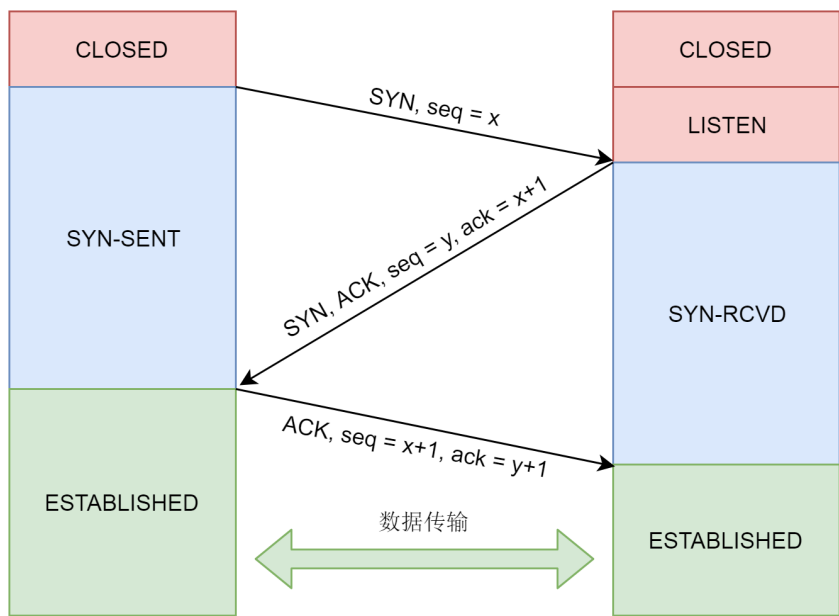
如果你看上面代码中的数据结构，里面的变量名称都有“be”两个字母，代表的意思是“big-endian”。如果在网络上传输超过 1 Byte 的类型，就要区分**大端**（Big Endian）和**小端**（Little Endian）。

假设，我们要在 32 位 4 Bytes 的一个空间存放整数 1，很显然只要 1 Byte 放 1，其他 3 Bytes 放 0 就可以了。那问题是，最后一个 Byte 放 1 呢，还是第一个 Byte 放 1 呢？或者说，1 作为最低位，应该放在 32 位的最后一个位置呢，还是放在第一个位置呢？


最低位放在最后一个位置，我们叫作小端，最低位放在第一个位置，叫作大端。TCP/IP 栈是按照大端来设计的，而 x86 机器多按照小端来设计，因而发出去时需要做一个转换。

接下来，就要建立 TCP 的连接了，也就是著名的三次握手，其实就是将客户端和服务端的状态通过三次网络交互，

达到初始状态是协同的状态。下图就是三次握手的序列图以及对应的状态转换。




接下来，服务端要调用 listen 进入 LISTEN 状态，等待客户端进行连接。

 复制代码

```
1 int listen(int sockfd, int backlog);
```


连接的建立过程，也即三次握手，是 TCP 层的动作，是在内核完成的，应用层不需要参与。

接着，服务端只需要调用 `accept`，等待内核完成了至少一个连接的建立，才返回。如果没有一个连接完成了三次握手，`accept` 就一直等待；如果有多个客户端发起连接，并且在内核里面完成了多个三次握手，建立了多个连接，这些连接会被放在一个队列里面。`accept` 会从队列里面取出一个来进行处理。如果想进一步处理其他连接，需要调用多次 `accept`，所以 `accept` 往往在一个循环里面。

 复制代码

```
1 int accept(int sockfd, struct sockaddr *addr, socklen_t
```

接下来，客户端可以通过 `connect` 函数发起连接。

 复制代码

```
1 int connect(int sockfd, const struct sockaddr *addr, sc
```

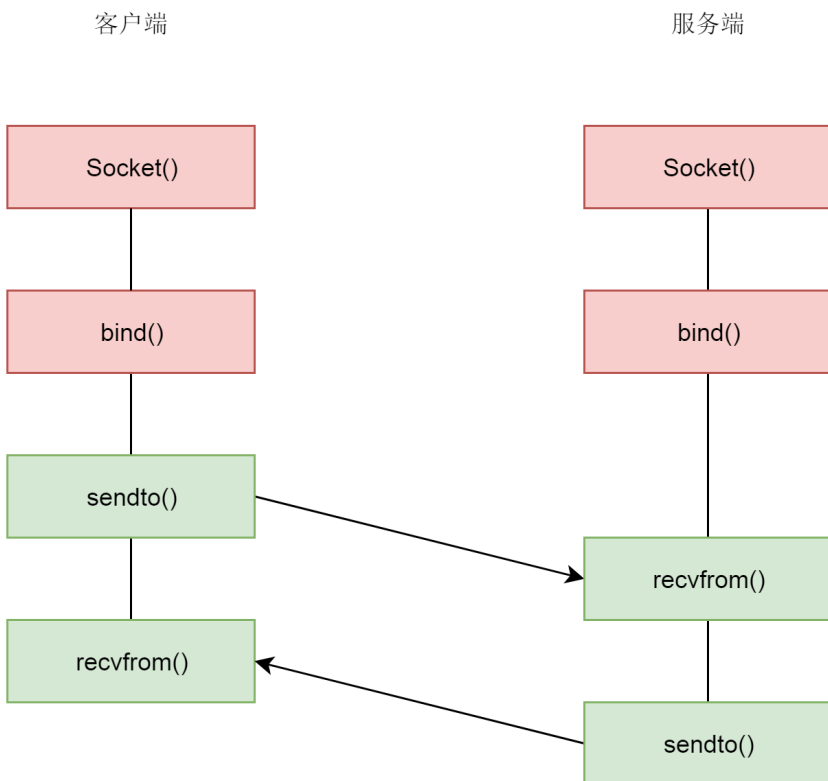
我们先在参数中指明要连接的 IP 地址和端口号，然后发起三次握手。内核会给客户端分配一个临时的端口。一旦握手

成功，服务端的 `accept` 就会返回另一个 `socket`。

这里需要注意的是，监听的 `socket` 和真正用来传送数据的 `socket`，是两个 `socket`，一个叫作**监听 socket**，一个叫作**已连接 socket**。成功连接建立之后，双方开始通过 `read` 和 `write` 函数来读写数据，就像往一个文件流里面写东西一样。

针对 UDP 应该如何编程？


接下来我们来看，针对 UDP 应该如何编程。



UDP 是没有连接的，所以不需要三次握手，也就不需要调用 `listen` 和 `connect`，但是 UDP 的交互仍然需要 IP 地址和端口号，因而也需要 `bind`。

对于 UDP 来讲，没有所谓的连接维护，也没有所谓的连接的发起方和接收方，甚至都不存在客户端和服务端的概念，大家就都是客户端，也同时都是服务端。只要有一个 `socket`，多台机器就可以任意通信，不存在哪两台机器是属

于一个连接的概念。因此，每一个 UDP 的 socket 都需要 bind。每次通信时，调用 sendto 和 recvfrom，都要传入 IP 地址和端口。

 复制代码

```
1 ssize_t sendto(int sockfd, const void *buf, size_t len,  
2  
3 ssize_t recvfrom(int sockfd, void *buf, size_t len, int
```

总结时刻

这一节我们讲了网络协议的基本原理和 socket 系统调用，这里请你重点关注 TCP 协议的系统调用。

通过学习，我们知道，socket 系统调用是用户态和内核态的接口，网络协议的四层以下都是在内核中的。很多的书籍会讲如何开发一个高性能的 socket 程序，但是这不是我们这门课的重点，所以我们主要看内核里面的机制就行了。

因此，你需要记住 TCP 协议的 socket 调用的过程。我们接下来就按照这个顺序，依次回忆一下这些系统调用到内核都做了什么：

服务端和客户端都调用 socket，得到文件描述符；

服务端调用 `listen`，进行监听；

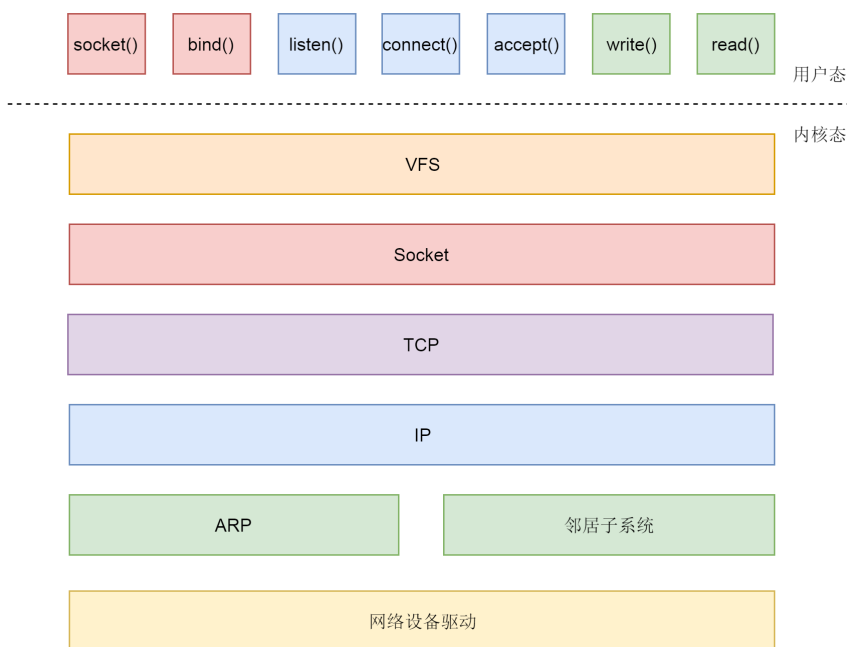
服务端调用 `accept`，等待客户端连接；

客户端调用 `connect`，连接服务端；

服务端 `accept` 返回用于传输的 `socket` 的文件描述符；

客户端调用 `write` 写入数据；

服务端调用 `read` 读取数据。



课堂练习

请你根据今天讲的 socket 系统调用，写一个简单的 socket 程序来传输一个字符串。

欢迎留言和我分享你的疑惑和见解，也欢迎可以收藏本节内容，反复研读。你也可以把今天的内容分享给你的朋友，和他一起学习和进步。

 极客时间

趣谈 Linux 操作系统

像故事一样的操作系统入门课

刘超

网易杭州研究院
云计算技术部首席架构师



新版升级：点击「 请朋友读」，10位好友免费读，邀请订阅更有**现金**奖励。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 43 预习 | Socket通信之网络协议基本原理

下一篇 44 | Socket内核数据结构：如何成立特大项目合作部？

精选留言 (3)

写留言



安排

2019-07-05

老师，可不可以在答疑篇，增加一个select,poll,epoll的内核机制分析？

展开 ∨



11



kdb_reboot

2019-07-05

老师厉害了，依然在更新；
最近我有时间学习这个专栏了，但是目前只跟到第十课，把专栏作为引子，每天的阅读量还是很大的
然后，我有个问题：专栏更新完老师还会答疑吗？因为进度原因，可能还没学到最后面，专栏已经更新完了

展开 ∨



1



bo

2019-07-05

老师好！udp中的connect背后做了什么工作？

展开 ∨



