中山大學
SUN YAT-SEN UNIVERSITY

SSE-208 计算机网络实验

网络编程

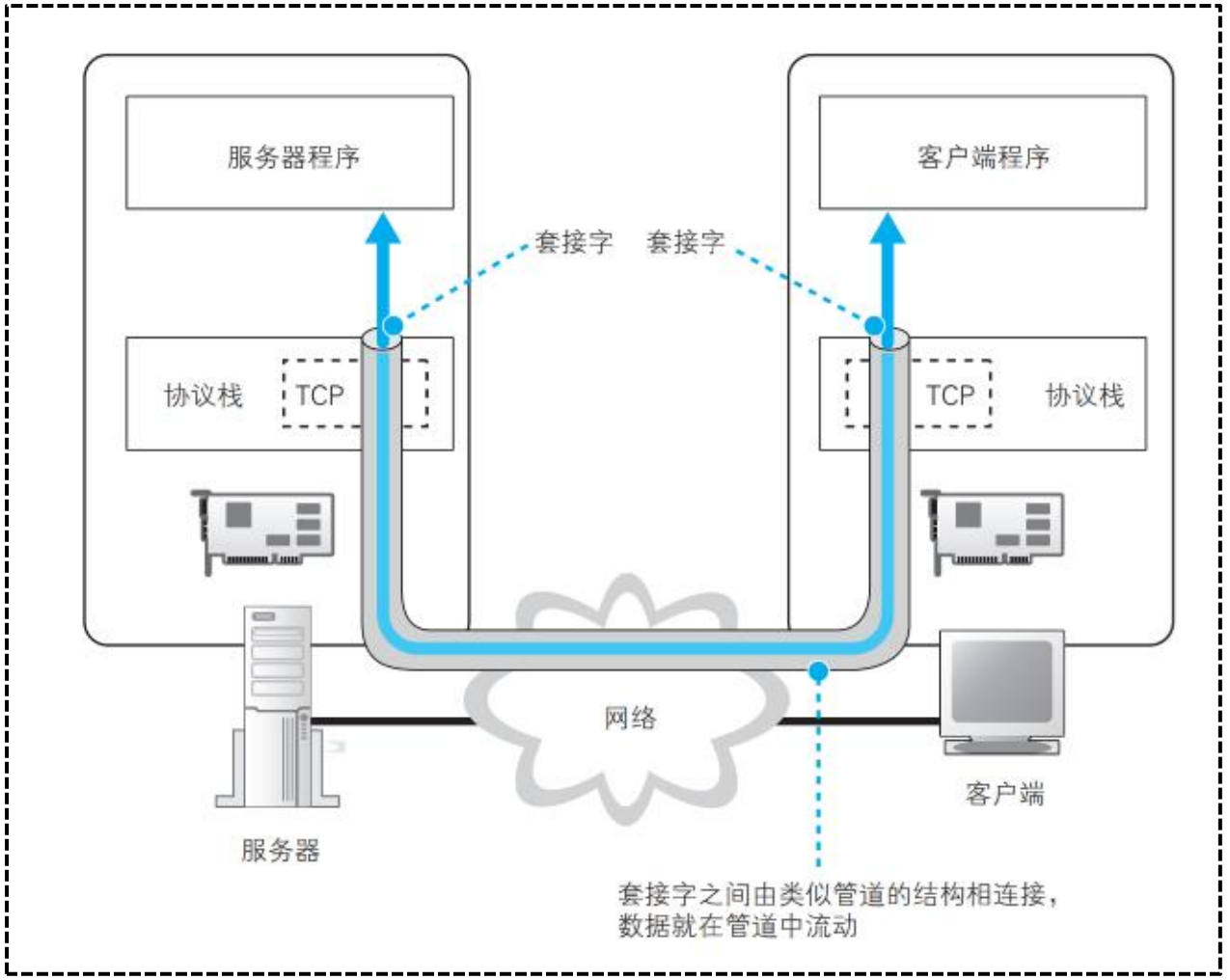
(UDP & TCP & Web 服务器 & UDP ping 实现)

2025.05

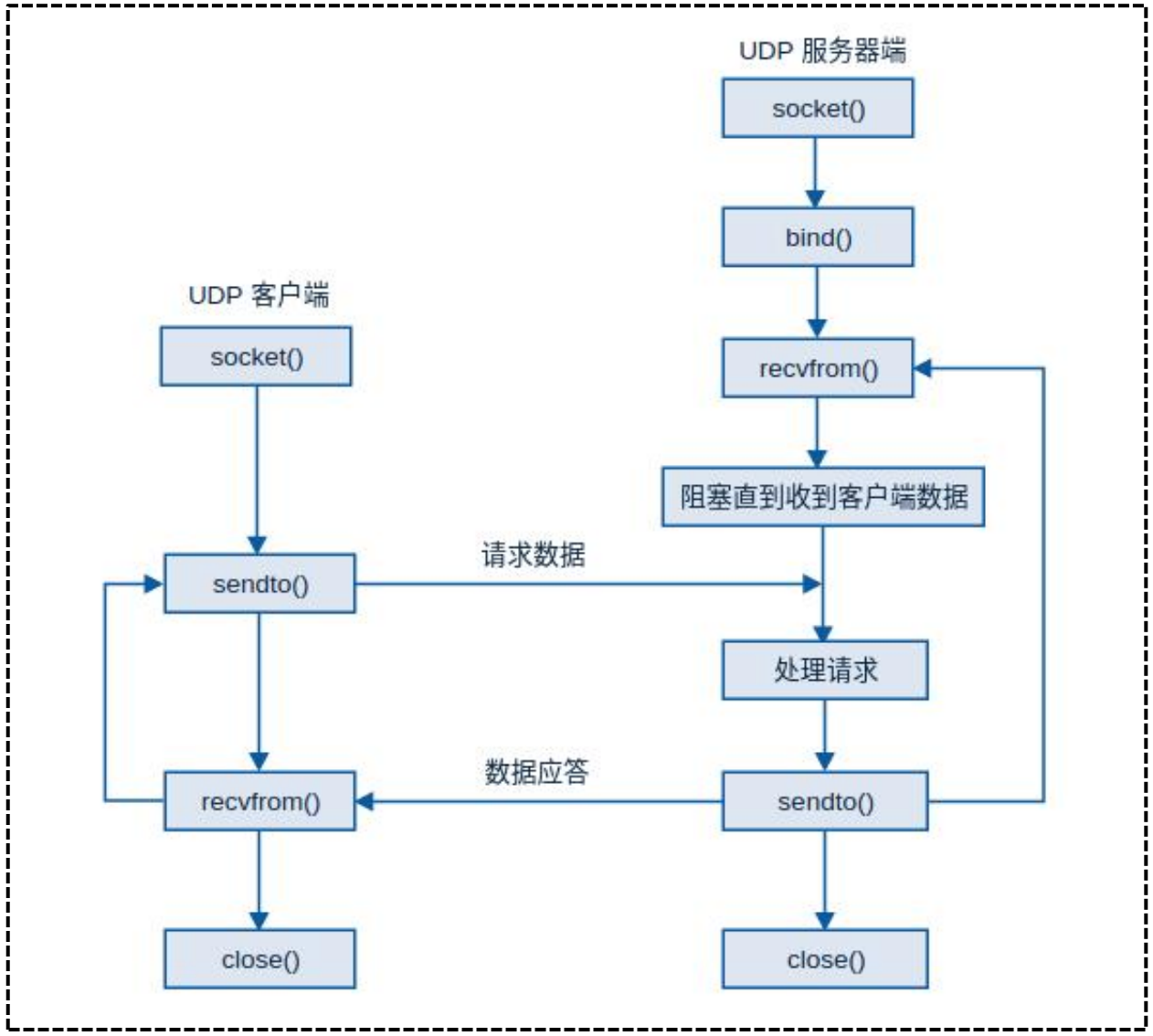
socket 介绍

socket

- 把数据通道想象成一条管道，将数据从一端送入管道，数据就会到达管道的另一端然后被取出。
- 不过，这并不是说现实中真的有这么一条管道，只是为了帮助大家理解



使用 socket 实现网络通信（UDP）



- 面向无连接
- 资源消耗小
- 基于数据报（datagram）
- 处理速度快

使用 socket 实现网络通信（TCP）

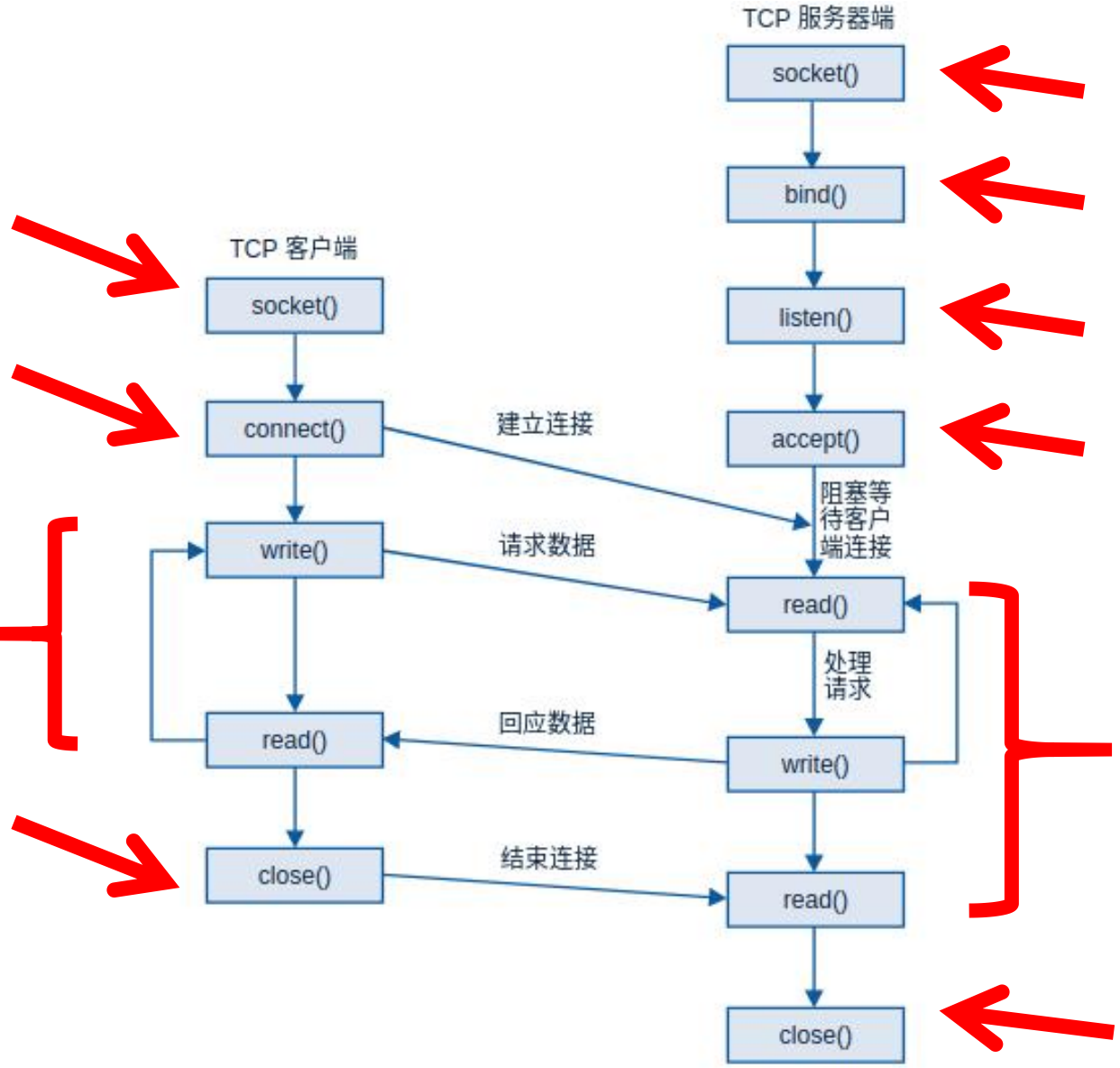
客户端

1. 创建socket

2. 连接**服务器端**的 socket

3. 收发数据

4. 断开连接，关闭 socket



服务器端

1. 创建socket

2-1. 绑定端口

2-2. 监听端口，等待连接

2-3. 接受连接

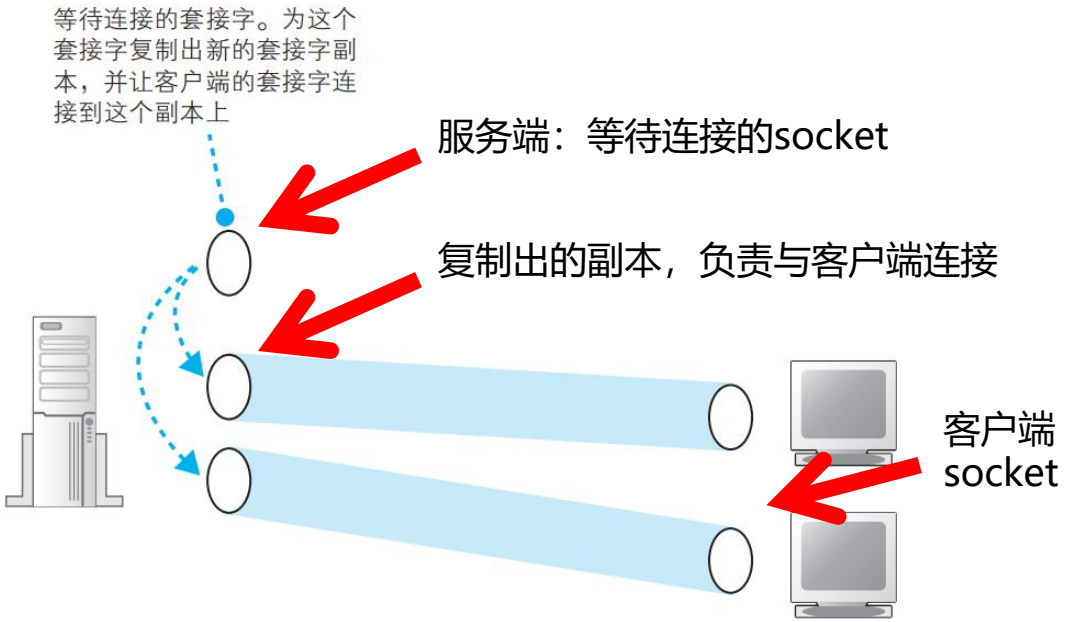
3. 收发数据

4. 断开连接，关闭 socket

使用 socket 实现网络通信（TCP）

socket 连接过程

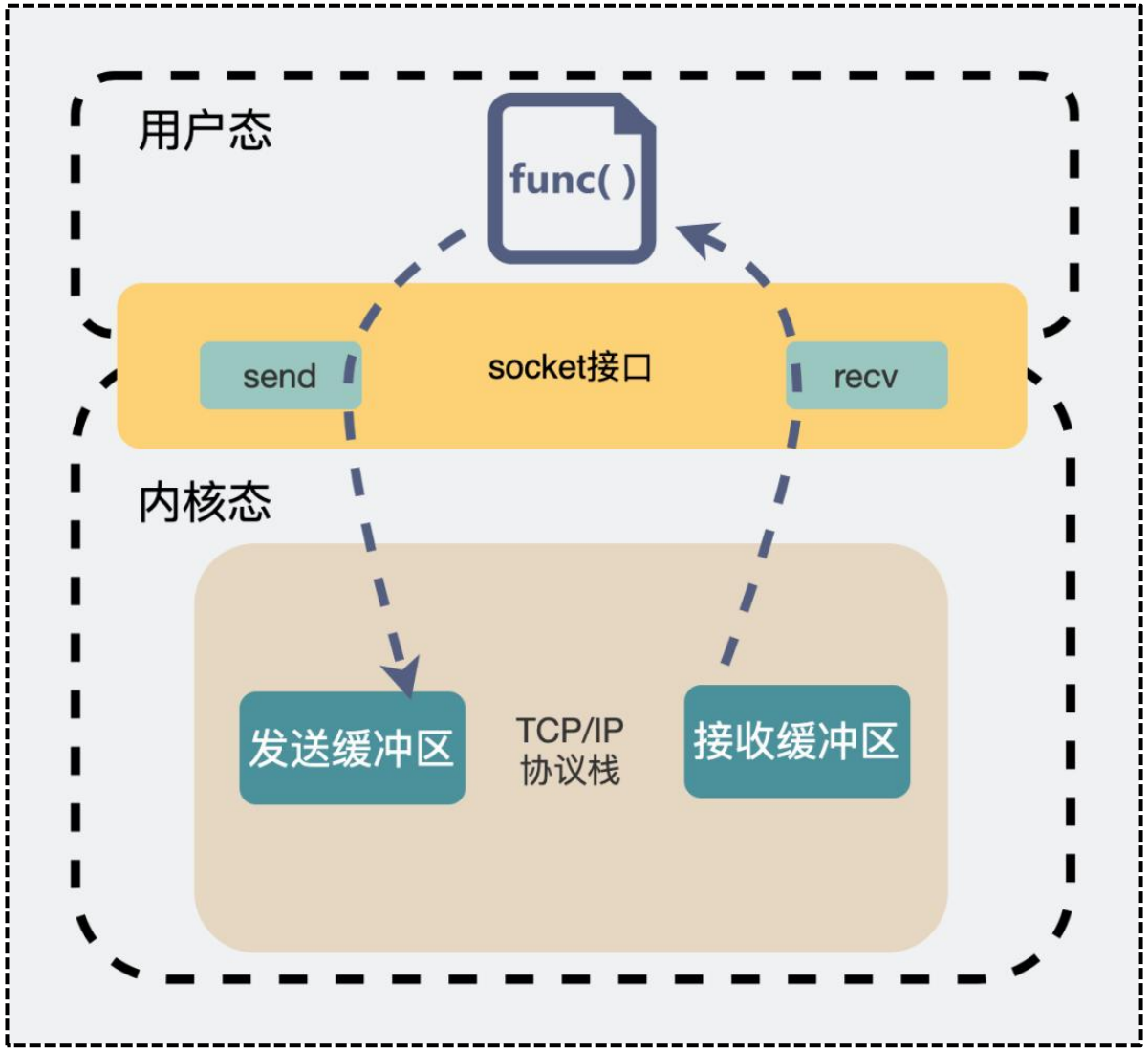
- 等待连接的 socket 会以继续监听，等待连接
- 复制新的 socket 副本，该 socket 负责与客户端 socket 连接
- 如果不创建新副本，而是直接让客户端连接到等待连接的 socket 上，那么就没有 socket 在继续等待新的连接了



socket 缓冲区

socket 数据收发过程

- 每个 socket 被创建后，无论使用的是 TCP 还是 UDP 协议，都会创建自己的输入缓冲区和输出缓冲区



socket 缓冲区

socket 数据收发过程 -- send（阻塞模式下）

- 发送缓冲区可用长度 vs. 待发送数据长度
  - 发送缓冲区可用长度 > 待发送数据长度，则数据将全部被拷贝到发送缓冲区
  - 发送缓冲区可用长度 < 待发送数据长度，则数据能拷贝多少就先拷贝多少（分批拷贝），一直等待，直到数据可以全部被拷贝到发送缓冲区为止，才调用返回。
- send() 并不能保证数据已经发送到对端，仅仅是把应用层 buffer 的数据拷贝进 socket 发送缓冲区中
- 至于缓冲区的数据什么时候发，发多少，取决于发送窗口、拥塞窗口以及当前发送缓冲区的大小等条件

socket 缓冲区

socket 数据收发过程 -- recv（阻塞模式下）

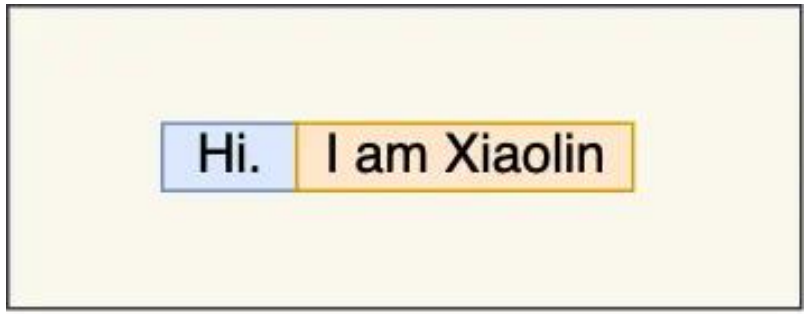
- 当接收缓冲区没数据时，程序就会一直阻塞等待，直到有数据可读为止
- socket.recv(bufsize[, flags])，bufsize 指定一次接收的最大数据量
  - 如果对方发送了超过bufsize的数据，在这种情况下要调用几次recv函数才能把套接字接收缓冲区中的数据拷贝完
  - recv() 所做的工作，仅仅是把socket接收缓冲区的数据拷贝到应用层 buffer 里面
- TCP 协议会保证数据的有序完整的传输，但是如何去正确完整的处理每一条信息，是开发人员的事情。

使用 socket 实现网络通信（TCP）：粘包问题

粘包示例

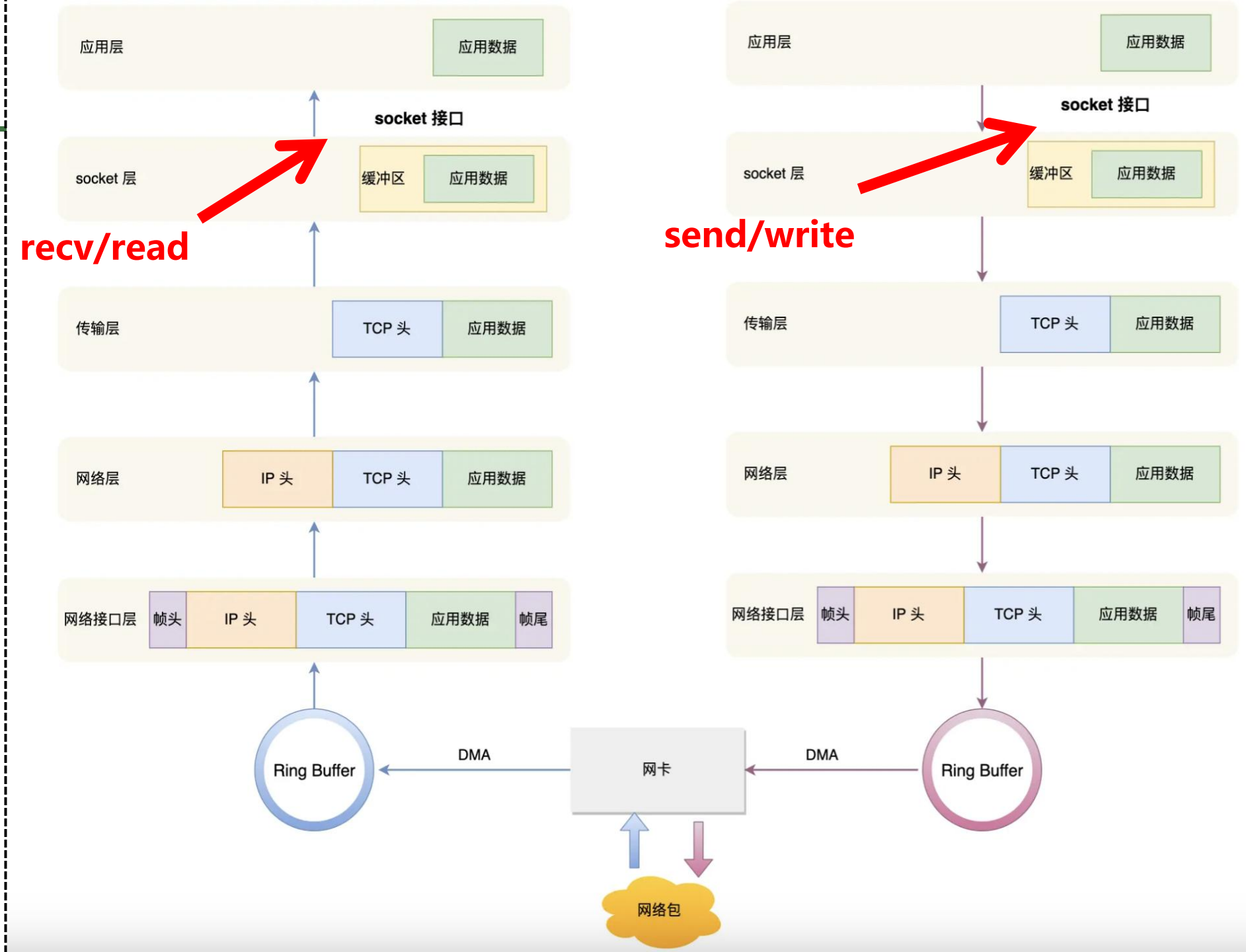
- 假设客户端调用 send 函数发送 “Hi”和”I am Xiao lin”两个报文
- 可能的结果：两个很短的报文积累在缓冲区，一次性发送
- 粘包的本质：TCP是面向连接的、可靠的、面向**字节流**的传输协议，没有消息边界，

  需要应用层自行解决从二进制字节流中提取数据、定义数据的边界的问题



同一个TCP 报文

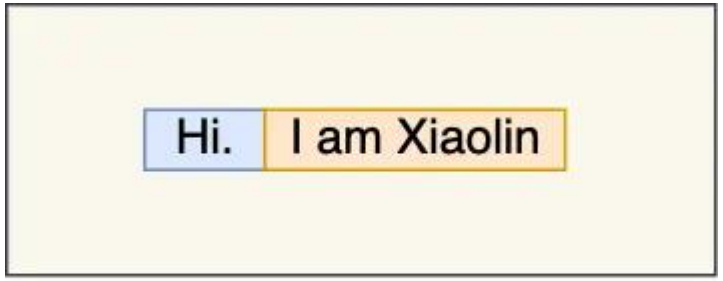
系统结构


应用层
应用数据
socket 接口
socket 层
缓冲区
应用数据
recv/read
传输层
TCP 头
应用数据
网络层
IP 头
TCP 头
应用数据
网络接口层
帧头
IP 头
TCP 头
应用数据
帧尾
Ring Buffer
DMA
网卡
网络包
应用层
应用数据
socket 接口
socket 层
缓冲区
应用数据
send/write
传输层
TCP 头
应用数据
网络层
IP 头
TCP 头
应用数据
网络接口层
帧头
IP 头
TCP 头
应用数据
帧尾
DMA
Ring Buffer

使用 socket 实现网络通信（TCP）：粘包问题

粘包原因

- 粘包：如果一次请求发送的数据量比较小，没达到缓冲区大小，则多个请求会被合并为同一个请求进行发送
- 原因
  - TCP是面向字节流，没有消息边界，由应用层负责处理消息边界的问题
  - 操作系统在发送 TCP 数据时，会通过缓冲区来进行优化



同一个TCP 报文

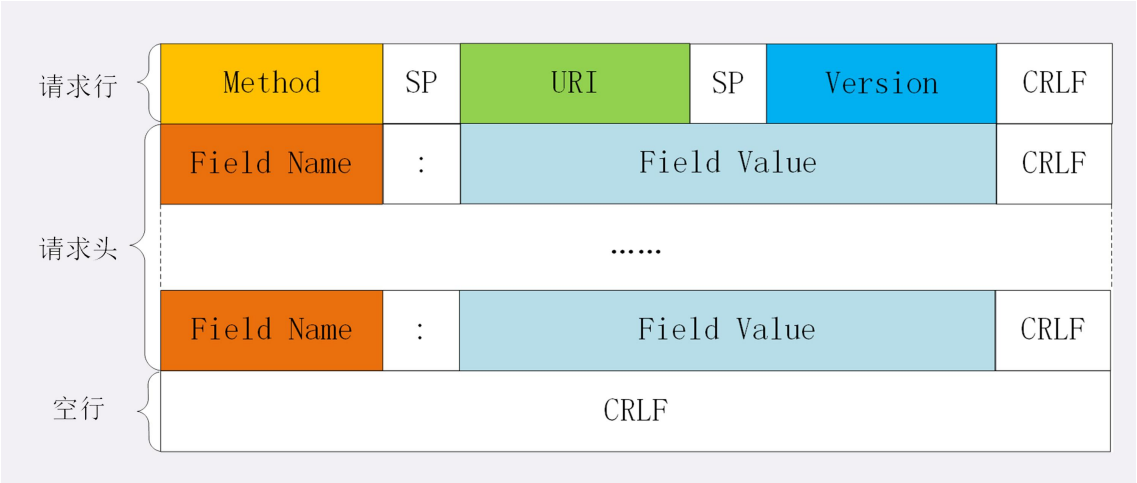
使用 socket 实现网络通信（TCP）：粘包问题

粘包解决方案

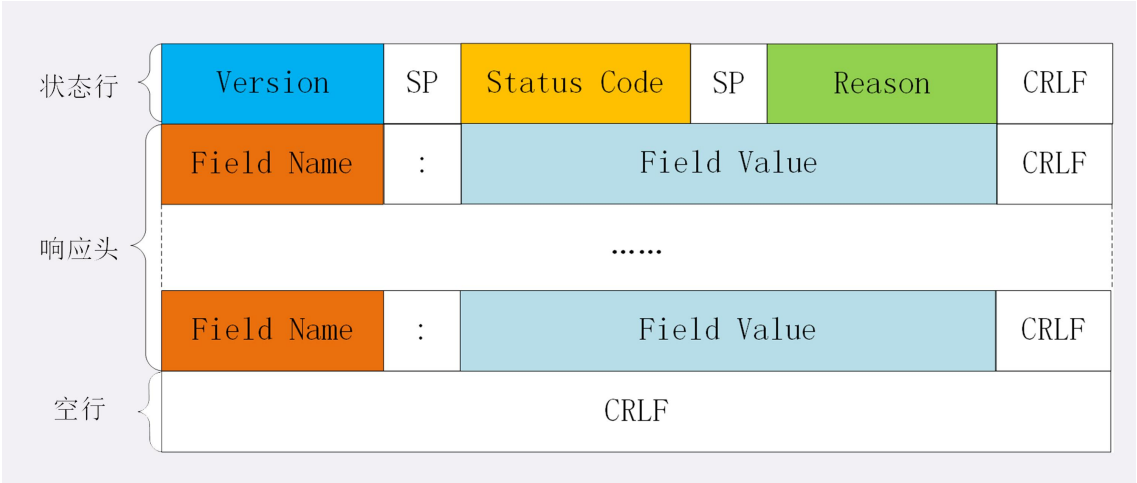
- 方案1：发送端将每个包都封装成固定的长度，比如100字节大小。如果不足100字节可通过补0或空等进行填充到指定长度；
- 方案2：发送端在每个包的末尾使用固定的分隔符，例如 \r\n
- 方案3：自定义一个消息结构，由头部和数据组成，其中头部是定长的，头部里有一个字段说明数据大小
- 方案4：自定义一个消息结构，由头部和数据组成，其中头部是变长的，头部有一个字段说明数据大小且头部有分隔符告知头部结束（参考HTTP协议实现）

HTTP 报文

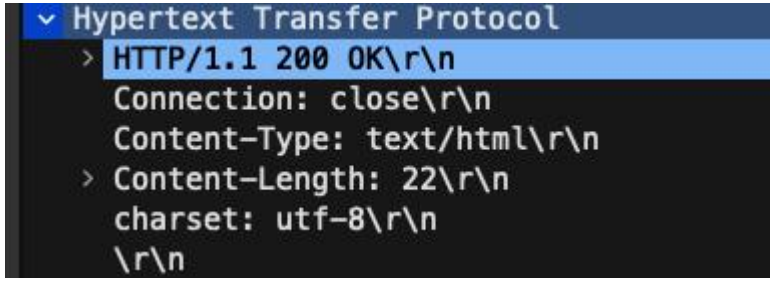
http 报文格式



空行后面紧跟着请求数据(可选)



空行后面紧跟着响应数据(可选)



CRLF: \r\n

SP:空格

socket 编程注意事项

编码问题

- recv并不是取完对方发送的数据，而是取一次。
- bufsize

socket.**recv**(*bufsize*[, *flags*])

从套接字接收数据。返回值是一个字节对象，表示接收到的数据。*bufsize* 指定一次接收的最大数据量。可选参数 *flags* 的含义请参阅 Unix 手册页 *recv(2)*，它默认为零。

备注: 为了最佳匹配硬件和网络的实际情况，*bufsize* 的值应为 2 的相对较小的幂，如 4096。

- 如果对方发送了超过bufsize的数据，recv需要多次调用，用户自己组装数据，自己写算法确保数据完整性。
- 即使对方发送的数据没有超过bufsize，也有情况需要多次调用recv，因为发送的数据可能超过了当时的tcp/ip的承载MTU最大传输量。

python 官方文档：socket 接口

socket 编程注意事项

recv 相关问题

- send 和 sendall 传送的数据类型是字节型，不是字符串
- 发送前需要 encode（例如‘utf-8’）
- recv后需要 decode（例如‘utf-8’）

send 和 sendall 的区别

- send 并不一定会把数据全部发完，而是只发一次
- 一般情况下，我们使用更多的是 sendall，使用 send 就需要自己处理未发送的数据。

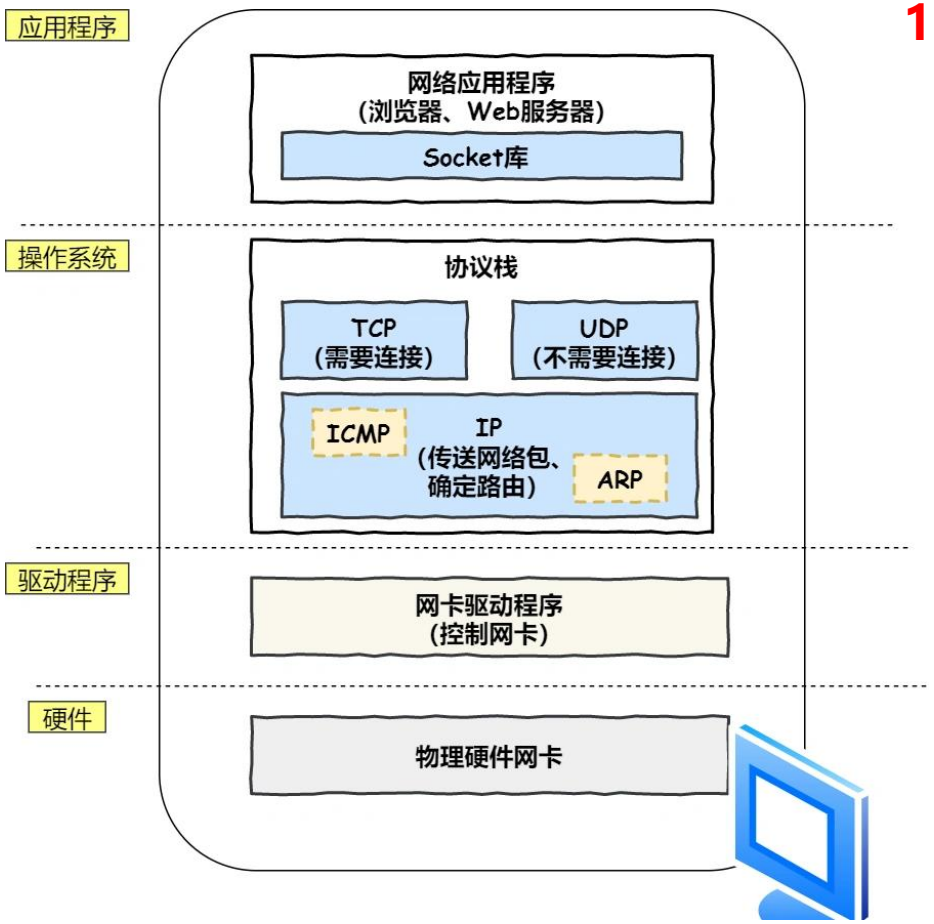
python 官方文档：socket 接口

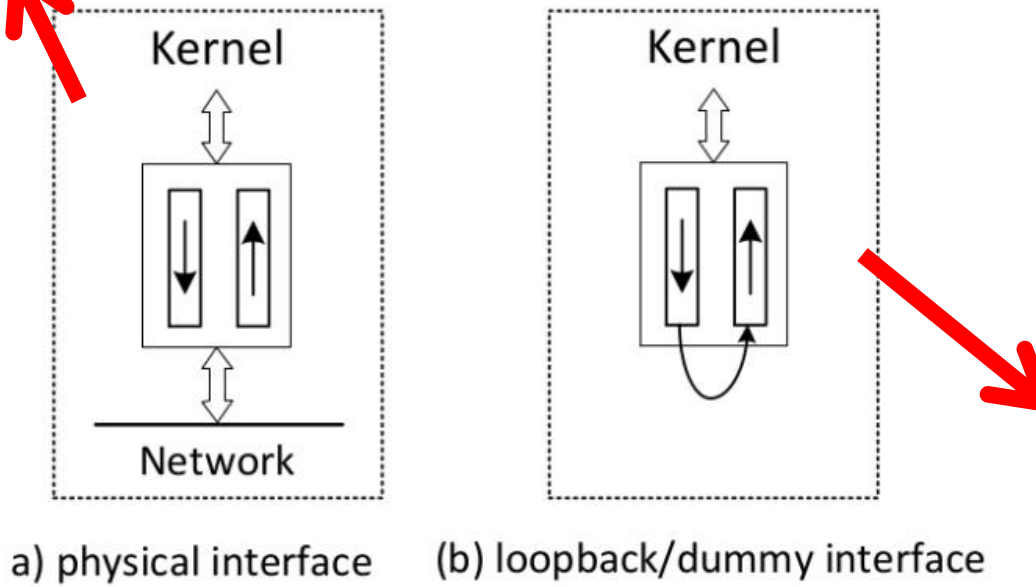
Web 服务器

Loopback

```
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=400<CHANNEL_IO>
        ether c8:89:f3:bc:75:e6
        inet6 fe80::8b2:a845:d370:7b38%en0 prefixlen 64 secured scopeid 0xe
        inet 172.27.34.19 netmask 0xffff8000 broadcast 172.27.127.255
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
```

172.27.34.19





a) physical interface (b) loopback/dummy interface

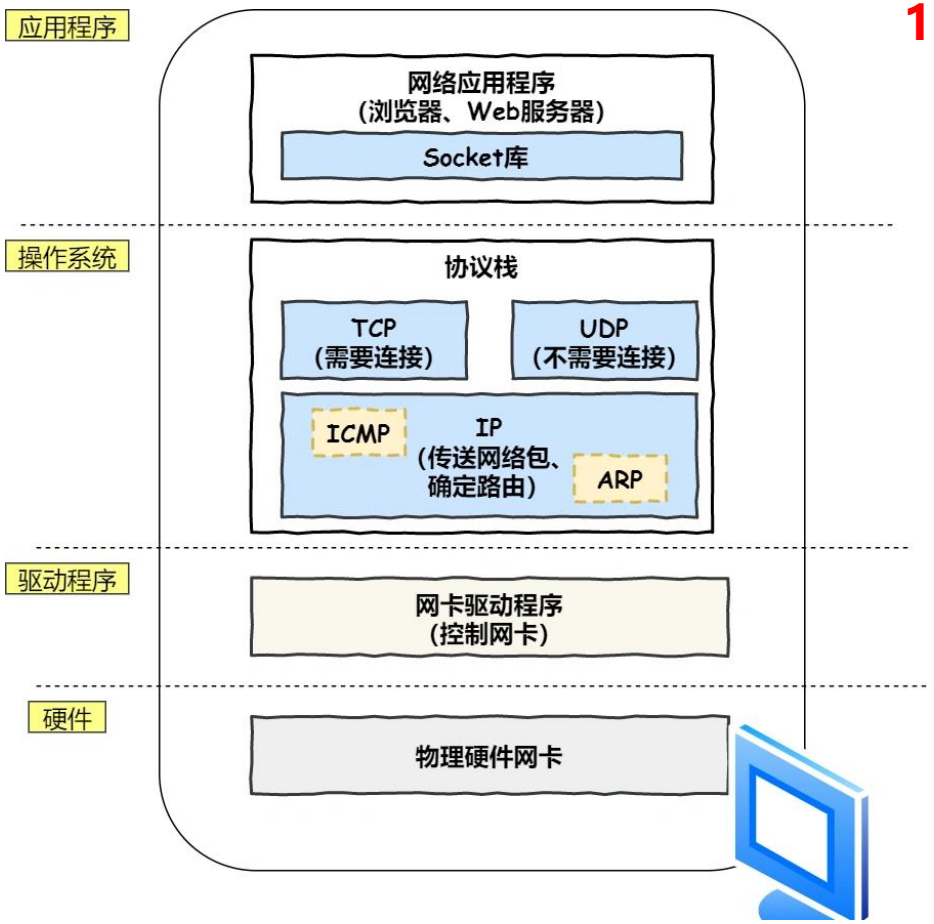
```
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=201<PERFORMNUD,DAD>
```

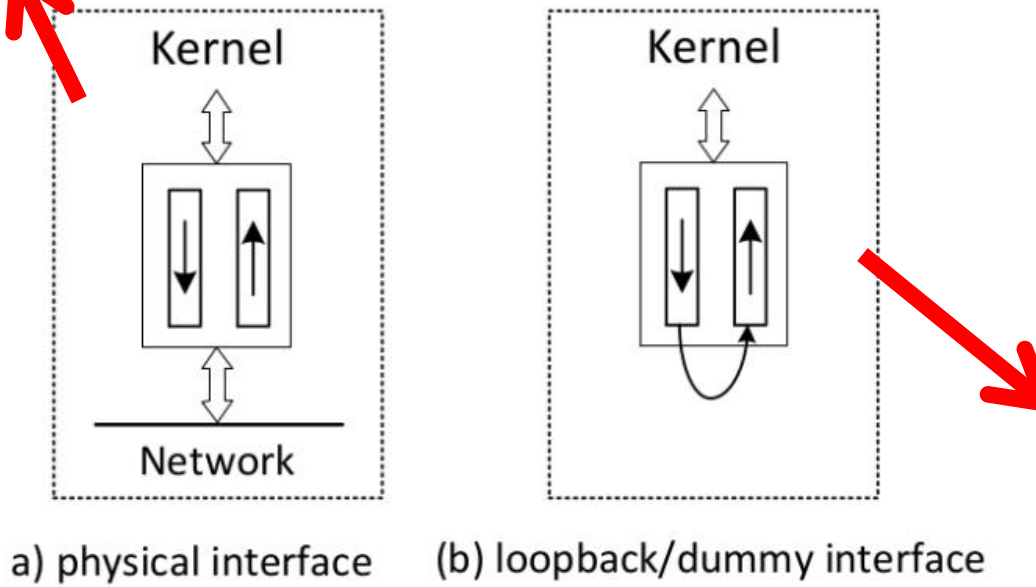
Web 服务器

Loopback

```
en0: flags=8863<UP,BROADCAST,SMART,RUNNING,SIMPLEX,MULTICAST> mtu 1500
        options=400<CHANNEL_IO>
        ether c8:89:f3:bc:75:e6
        inet6 fe80::8b2:a845:d370:7b38%en0 prefixlen 64 secured scopeid 0xe
        inet 172.27.34.19 netmask 0xffff8000 broadcast 172.27.127.255
        nd6 options=201<PERFORMNUD,DAD>
        media: autoselect
```

172.27.34.19





a) physical interface (b) loopback/dummy interface

```
lo0: flags=8049<UP,LOOPBACK,RUNNING,MULTICAST> mtu 16384
        options=1203<RXCSUM,TXCSUM,TXSTATUS,SW_TIMESTAMP>
        inet 127.0.0.1 netmask 0xff000000
        inet6 ::1 prefixlen 128
        inet6 fe80::1%lo0 prefixlen 64 scopeid 0x1
        nd6 options=201<PERFORMNUD,DAD>
```

Web 服务器

Web服务器-实验要求

- 你的 Web 服务器应该：接受并解析 HTTP 请求， 然后从服务器的文件系统中获取所请求的 html 文件，创建一个由响应文件组成的 HTTP 响应消息发送给客户端。
- 如果请求的文件不存在于服务器，则服务器应该向客户端发送 404 状态码响应消息

位于服务器（也就是你的PC）的html文件

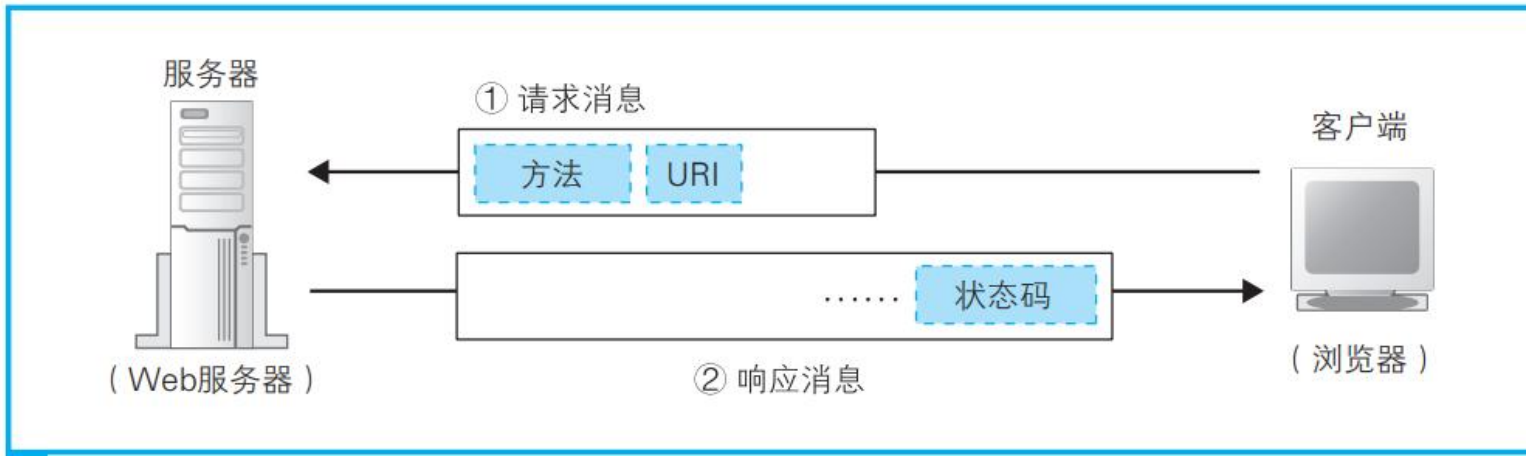
200 ok

404 Page not Found

Web 服务器

回顾：浏览网页的大致流程

- 1. 客户端**浏览器**解析 URL，确定 Web **服务器**和文件名
- 2. 浏览器根据这些信息来生成 HTTP 请求消息
- 3. 浏览器向 DNS 服务器查询 Web 服务器的 IP 地址

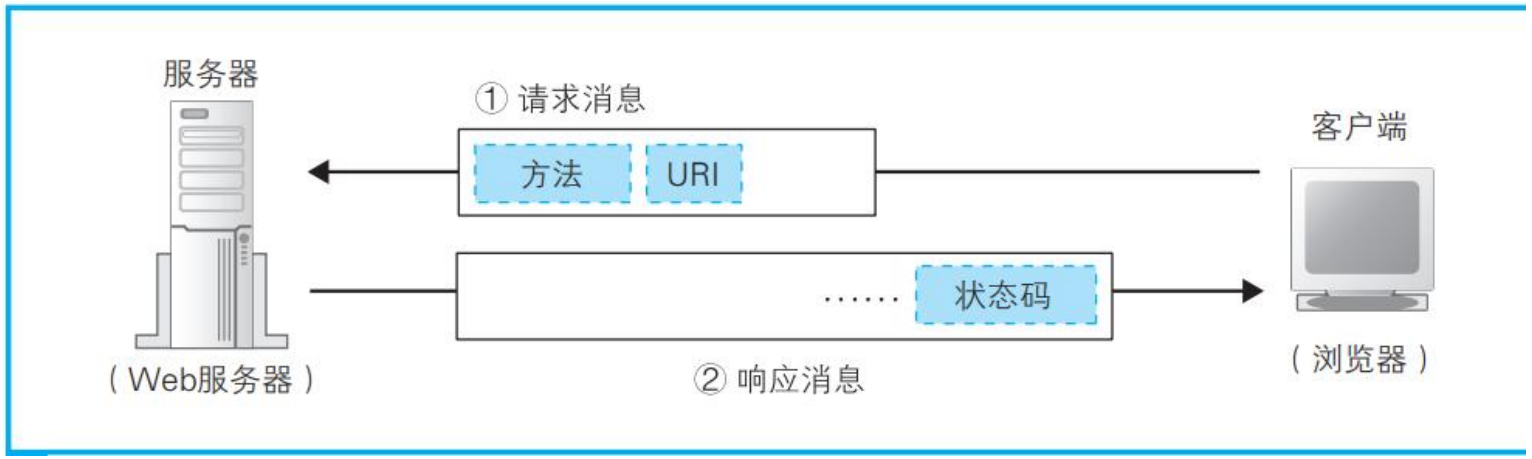


从浏览器中输入网址开始

Web 服务器

回顾：浏览网页的大致流程

- **4. 客户端浏览器向 Web 服务器发起 http 请求**
- 5. Web 服务器返回响应消息
- 6. 客户端浏览器将数据提取出来并显示在屏幕上



从浏览器中输入网址开始

Web 服务器

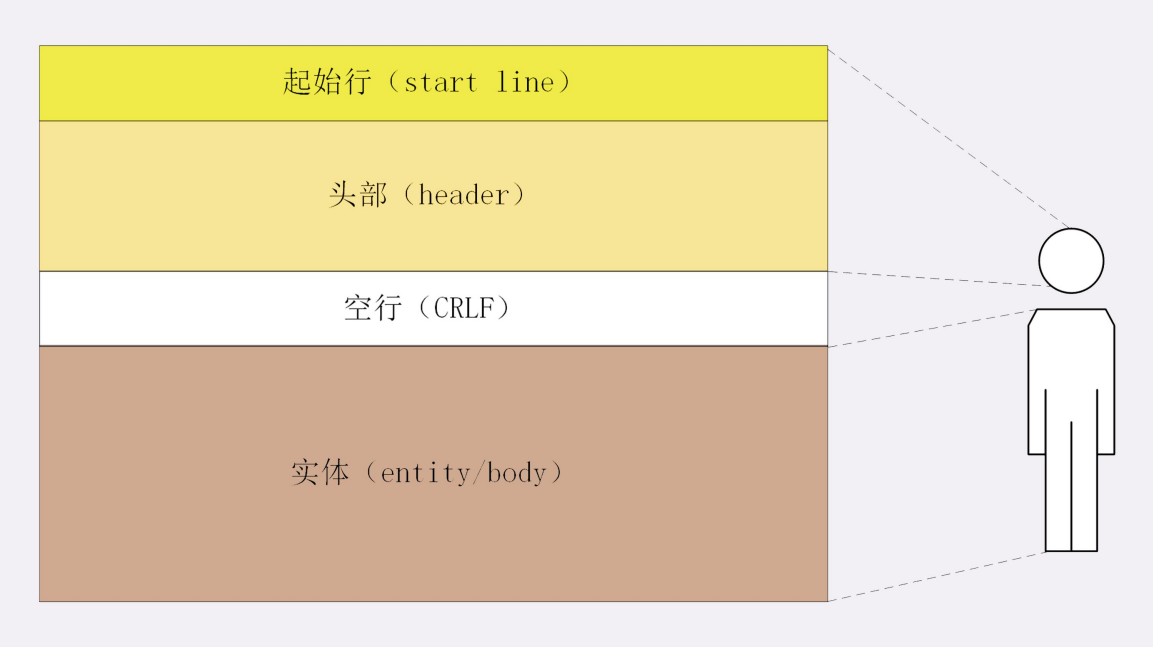
Web服务器 - 具体思路

- 使用 socket 编程，服务器端监听(127.0.0.1:6789), 等待客户端连接
- 服务端一旦收到消息，则解析 http 报文，得知客户端请求的文件
- 服务端查找本地，如果客户端请求的文件存在，则读取文件内容；如果不存在，读取 404.html 的内容。
- 服务端生成响应消息（状态行+响应头+空行+响应体）并返回给客户端

Web 服务器

Web服务器 - http报文

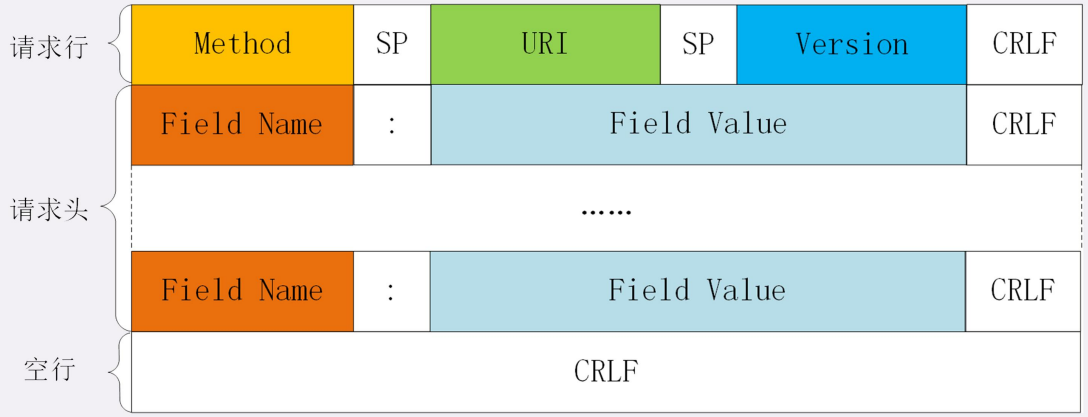
- 起始行（start line）：描述请求或响应的基本信息；
- 头部字段集合（header）：使用 key-value 形式更详细地说明报文；
- 消息正文（entity）：实际传输的数据，它不一定是纯文本，可以是图片、视频等二进制数据。



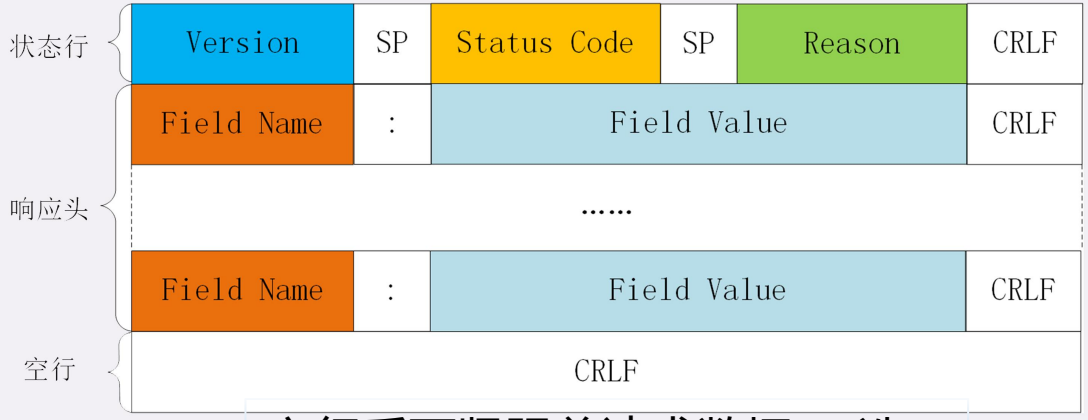
CRLF: \r\n

Web 服务器

Web服务器 - http报文



空行后面紧跟着请求数据(可选)



空行后面紧跟着请求数据(可选)

CRLF: \r\n　　**SP:空格**

```
Hypertext Transfer Protocol
  GET /Hello_SSE.html HTTP/1.1\r\n
  Host: 127.0.0.1:6789\r\n
  Connection: keep-alive\r\n
  Cache-Control: max-age=0\r\n
  sec-ch-ua: "Microsoft Edge";v="111", '
  sec-ch-ua-mobile: ?0\r\n
  sec-ch-ua-platform: "macOS"\r\n
  Upgrade-Insecure-Requests: 1\r\n
```

```
Hypertext Transfer Protocol
  HTTP/1.1 200 OK\r\n
  Connection: close\r\n
  Content-Type: text/html\r\n
  Content-Length: 22\r\n
  charset: utf-8\r\n
  \r\n
```

UDP ping

UDP ping

- ping 程序位于网络层，基于 ICMP 协议，其允许客户端机器发送一个数据包到远程机器，并使远程机器响应客户（称为回显）。另外，ping 程序还允许主机计算它到其他机器的往返时间。
- 我们已经介绍了 UDP 编程，但考虑到 ICMP 的复杂性，本次实验，请你基于 UDP 简单复现一个 ping 应用程序。

```
PS [redacted] ping 23.106.147.46

正在 Ping 23.106.147.46 具有 32 字节的数据:
来自 23.106.147.46 的回复: 字节=32 时间=161ms
来自 23.106.147.46 的回复: 字节=32 时间=162ms
来自 23.106.147.46 的回复: 字节=32 时间=162ms
来自 23.106.147.46 的回复: 字节=32 时间=161ms

23.106.147.46 的 Ping 统计信息:
    数据包: 已发送 = 4, 已接收 = 4, 丢失 = 0 (0% 丢失),
往返行程的估计时间(以毫秒为单位):
    最短 = 161ms, 最长 = 162ms, 平均 = 161ms
```

UDP ping

UDP ping - 实验要求

- 需要你实现一个 udp ping 客户端，向服务器发送 10 次 ping，并按照标准 ping 程序的格式进行输出。
  - 因为 UDP 是不可靠的协议，所以从客户端发送到服务器的数据包可能在网络中丢失。因此，客户端不能无限期地等待 ping 消息的回复。客户端一般会设置一个超时时间（ **clientSocket.settimeout(5)** ），如果在超时时间内没有收到回复， 客户端程序会认为数据包在网络传输期间丢失。
  - 本地网络环境下测试，通常不会丢包。服务器代码中已通过随机数的方式，模拟了一定概率的网络丢包（服务器可能不发送响应），服务器代码已提供，无需修改。