Go进阶训练营
第9课
Go 语言实践- 网络编程答疑

大明

知识梳理

- epoll 和 select 的实现
- 网络协议设计

知识梳理 —— epoll 和 select 的实现

核心就在于，怎么从一大堆创建好的文件描述符里面挑出来符合条件的文件描述符？

从直觉出发：

- 每次我都遍历所有创建好的文件描述符，挨个检查是否符合条件。把可读的连接返回；

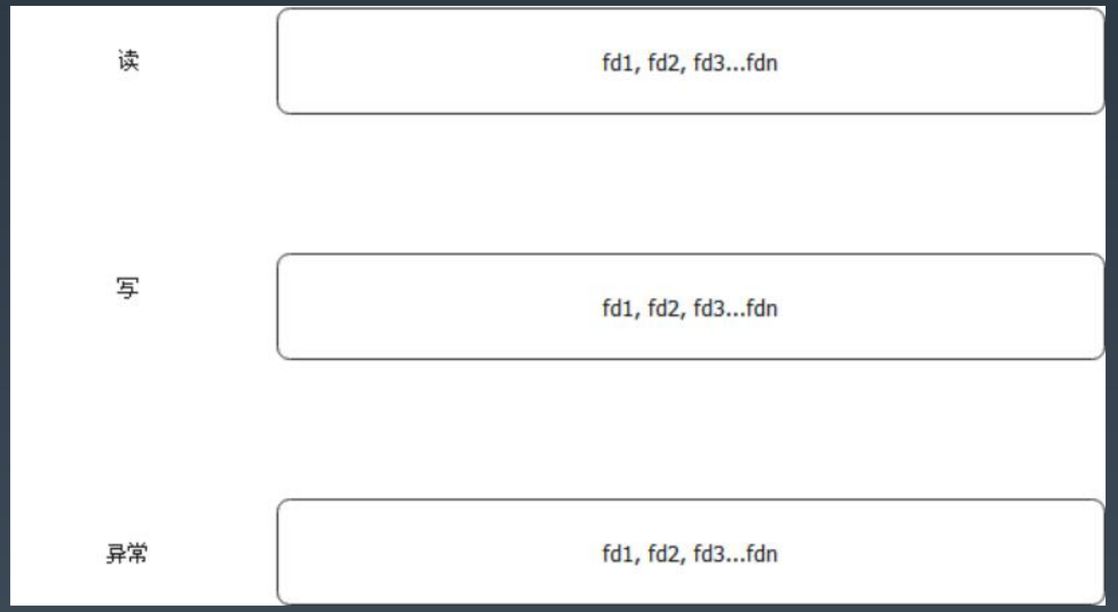
- 我不想遍历所有的文件描述符，因为很耗时。我尝试在文件描述符满足条件的时候，将它们挪到一个队列里面，如果用户询问我，我就直接返回这个队里的数据；

知识梳理 —— epoll 和 select 的实现

select 的实现：每次我都遍历所有创建好的连接，挨个检查是否可读。把可读的连接返回

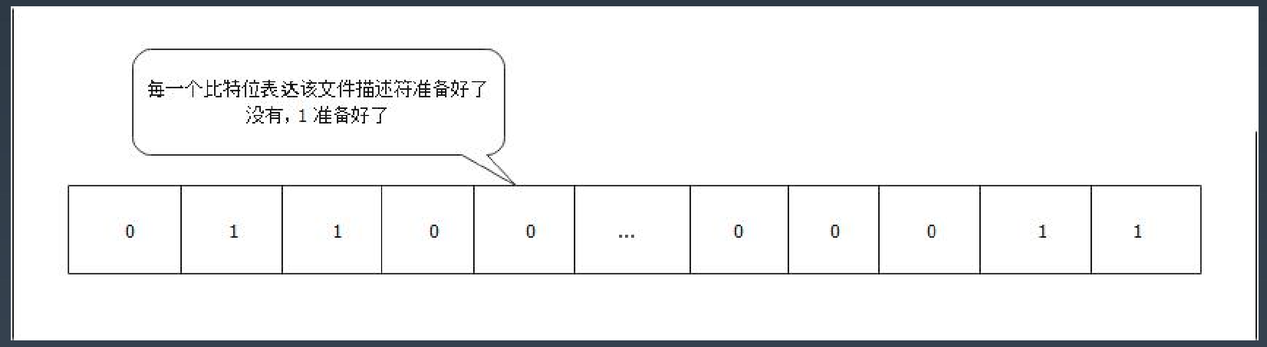
细节：

1. select 接收三个文件描述符集合：可读、可写和异常文件描述符集合，作为它监听的对象（遍历的对象）



知识梳理 —— epoll 和 select 的实现

select 的实现：每次我都遍历所有创建好的连接，挨个检查是否可读。把可读的连接返回



细节：

1. select 接收三个文件描述符集合：可读、可写和异常文件描述符集合，作为它监听的对象（遍历的对象）

2. 文件描述符集合是一个 bitset，每一个比特位表达该文件描述符的状态，默认容量是 1024

知识梳理 —— epoll 和 sele

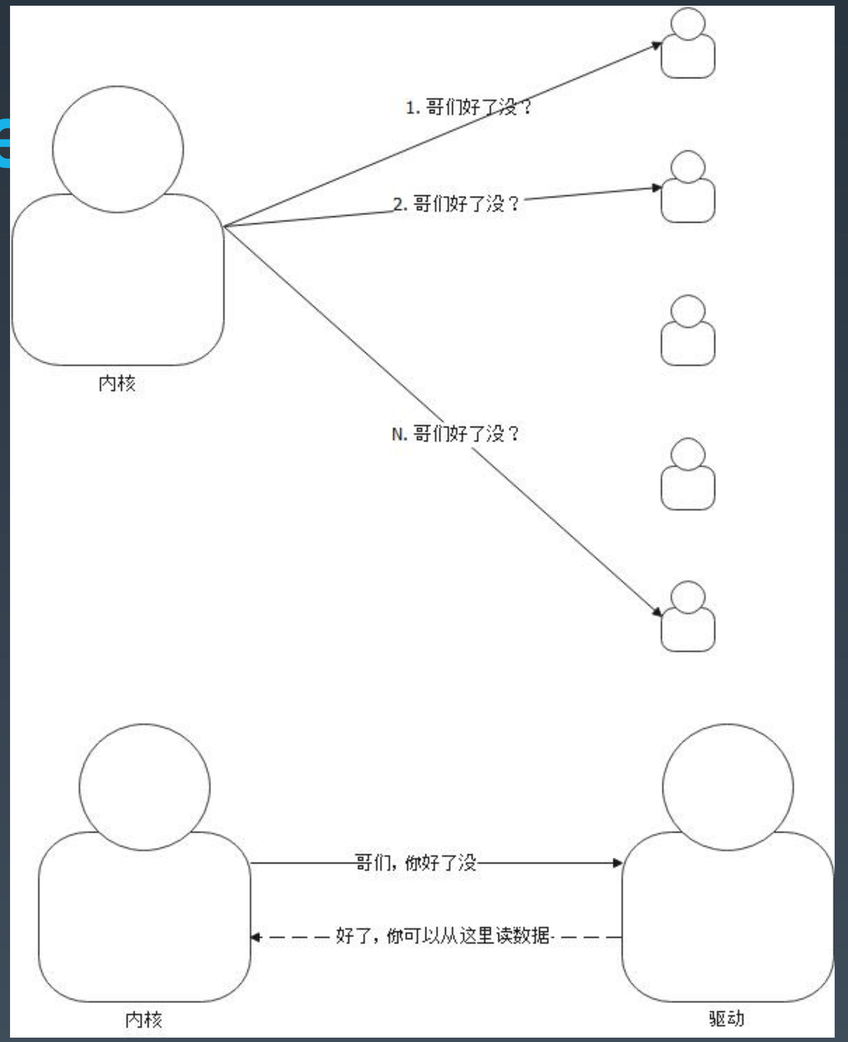
select 的实现：每次我都遍历所有创建好的连接，挨个检查是否可读。把可读的连接返回

细节：

1. select 接收三个文件描述符集合：可读、可写和异常文件描述符集合，作为它监听的对象（遍历的对象）

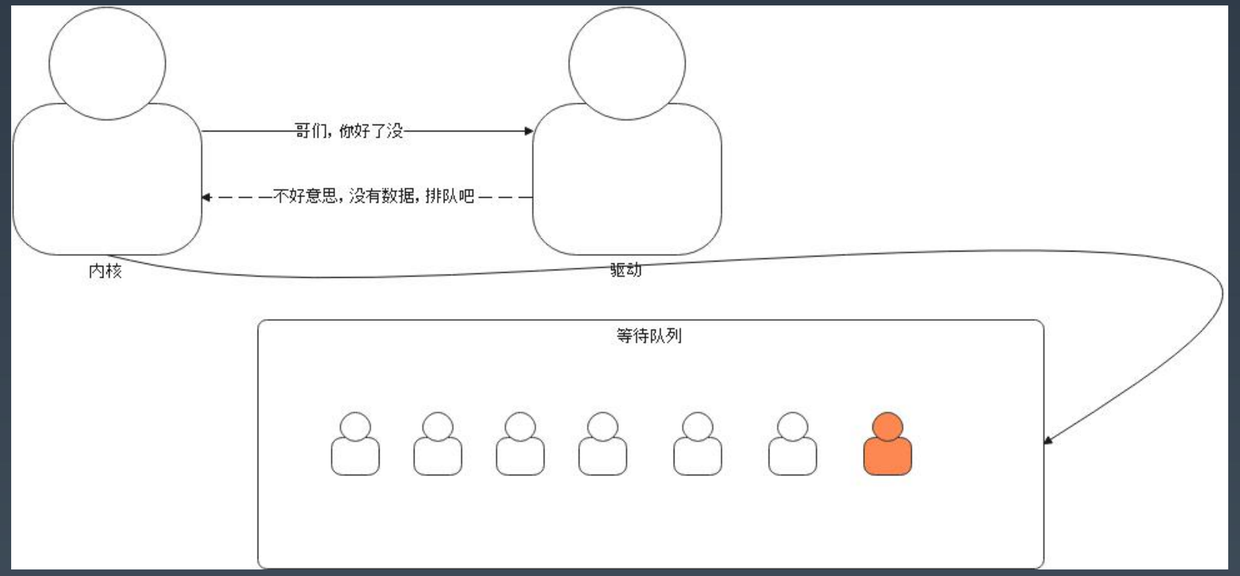
2. 文件描述符集合是一个 bitset，每一个比特位表达该文件描述符的状态，默认容量是 1024

3. 发起 select 调用，则需要传入我们希望 select 监听的文件描述符集合，select 遍历这些文件描述符，根据文件描述符去找驱动，驱动会回答这些问题



知识梳理 —— epoll 和 select 的实现

select 的实现：每次我都遍历所有创建好的连接，挨个检查是否可读。把可读的连接返回



细节：

1. select 接收三个文件描述符集合：可读、可写和异常文件描述符集合，作为它监听的对象（遍历的对象）

2. 文件描述符集合是一个 bitset，每一个比特位表达该文件描述符的状态，默认容量是 1024

3. 发起 select 调用，则需要传入我们希望 select 监听的文件描述符集合，select 遍历这些文件描述符

4. 如果没有数据，并且设置了超时，那么会进入等待队列

知识梳理 —— epoll 和 select 的实现

细节：

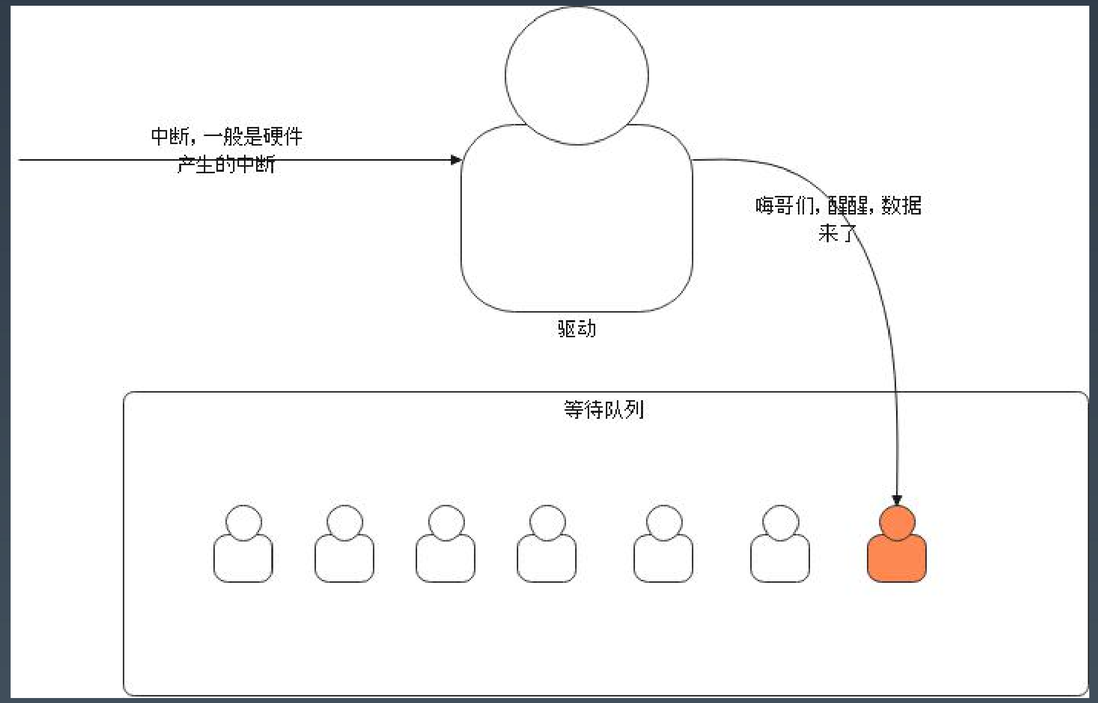
1. select 接收三个文件描述符集合：可读、可写和异常文件描述符集合，作为它监听的对象（遍历的对象）

2. 文件描述符集合是一个 bitset，每一个比特位表达该文件描述符的状态，默认容量是 1024

3. 发起 select 调用，则需要传入我们希望 select 监听的文件描述符集合，select 遍历这些文件描述符

4. 如果没有数据，并且设置了超时，那么会进入等待队列

5. 如果超时之前等到了，驱动会唤醒内核线程



知识梳理 —— epoll 和 select 的实现

细节：

1. select 接收三个文件描述符集合：可读、可写和异常文件描述符集合，作为它监听的对象（遍历的对象）

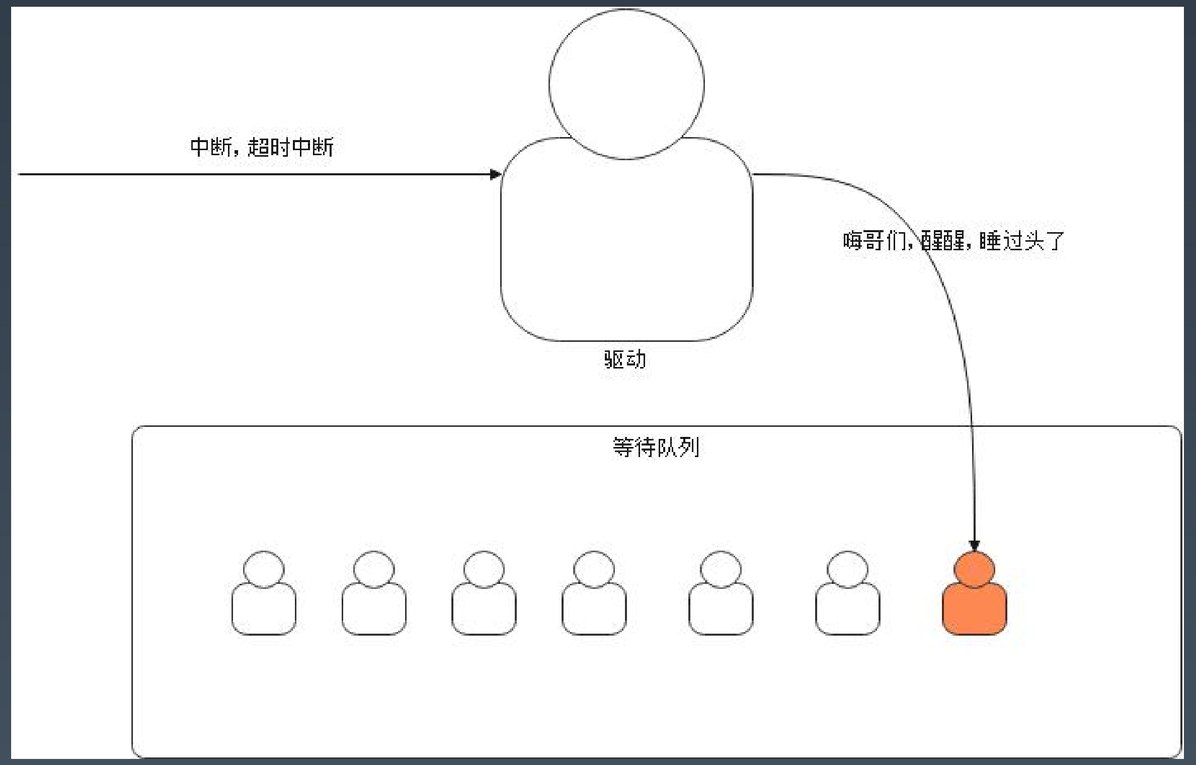
2. 文件描述符集合是一个 bitset，每一个比特位表达该文件描述符的状态，默认容量是 1024

3. 发起 select 调用，则需要传入我们希望 select 监听的文件描述符集合，select 遍历这些文件描述符

4. 如果没有数据，并且设置了超时，那么会进入等待队列

5. 如果超时之前等到了，驱动会唤醒内核线程

6. 如果超时了，那么



知识梳理 —— epoll 和 select 的实现

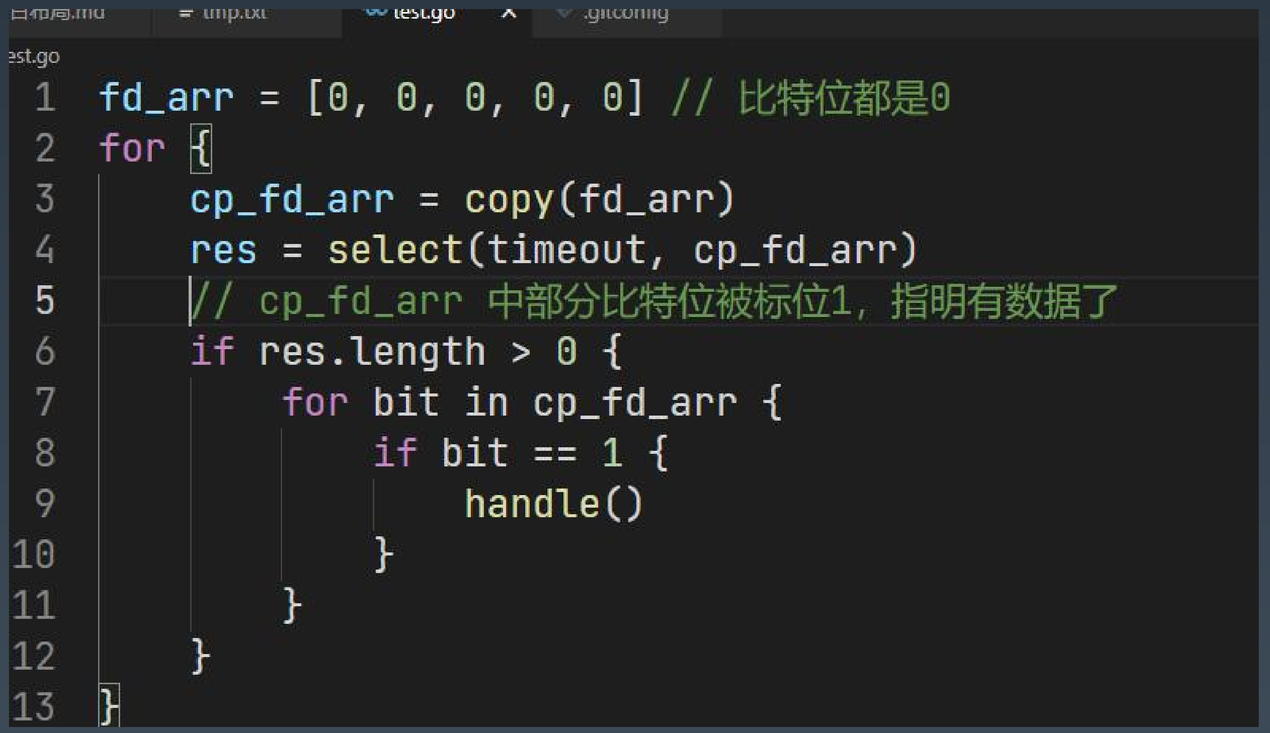
用户使用方法：

1. 准备需要 select 的文件描述符集合 fd_arr

2. 复制文件描述符集合，作为参数传递给 select 系统调用

3. 检查每一个比特位，确认有没有就绪的文件描述符

4. 处理就绪的文件描述符



知识梳理 —— epoll 和 select 的实现

epoll： 我不想遍历所有的文件描述符，因为很耗时。我尝试在文件描述符满足条件的时候，将它们挪到一个队列里面，如果用户询问我，我就直接返回这个队里的数据

细节：

1. 核心在于三个方法：epoll_create, epoll_ctl, epoll_wait

epoll_create: 创建一个 epoll 对象，它本身也是一个文件描述符

epoll_ctl：控制 epoll 对象。例如将文件描述符加到 epoll 里面，或者从 epoll 里面挪走一个文件描述符

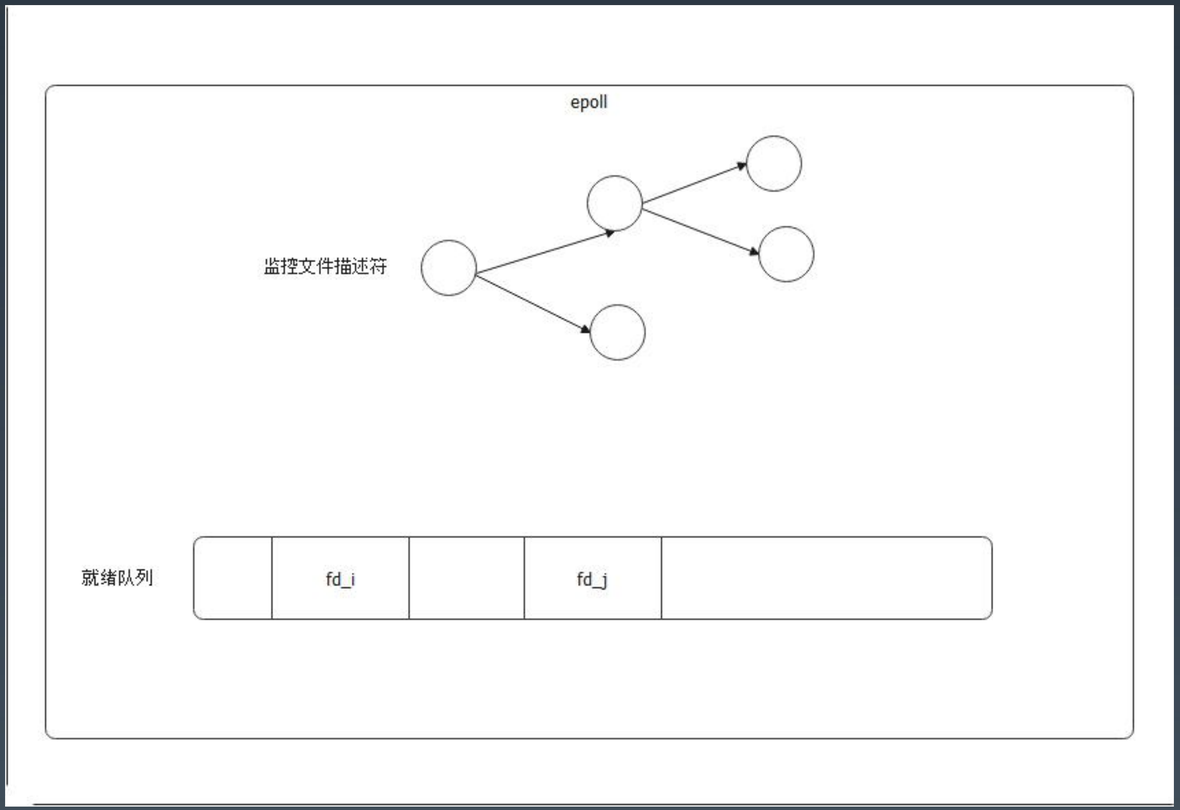
epoll_wait：在超时时间内，如果监控的文件描述符上发生了对应的事情，就返回给给用户线程

知识梳理 —— epoll 和 select 的实现

细节：

1. 核心在于三个方法：epoll_create, epoll_ctl, epoll_wait

2. 一个 epoll 对象主要有两个结构，一个是用红黑树来存储被监控文件描述符，一个是就绪队列，存储就绪文件描述符



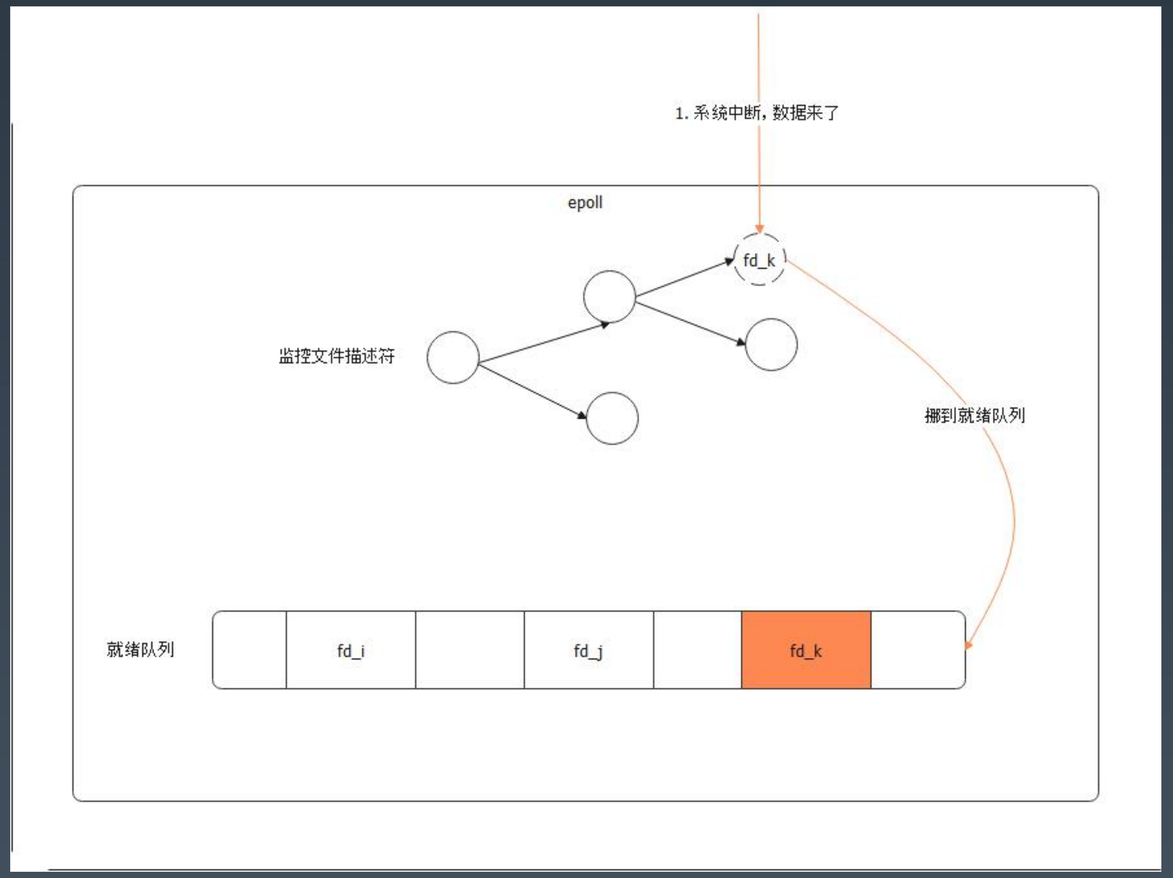
知识梳理 —— epoll 和 select 的实现

细节：

1. 核心在于三个方法：epoll_create, epoll_ctl, epoll_wait

2. 一个 epoll 对象主要有两个结构，一个是用红黑树来存储被监控文件描述符，一个是就绪队列，存储就绪文件描述符

3. epoll 会监听系统中断，而后将文件描述符挪到就绪队列



知识梳理 —— epoll 和 select 的实现

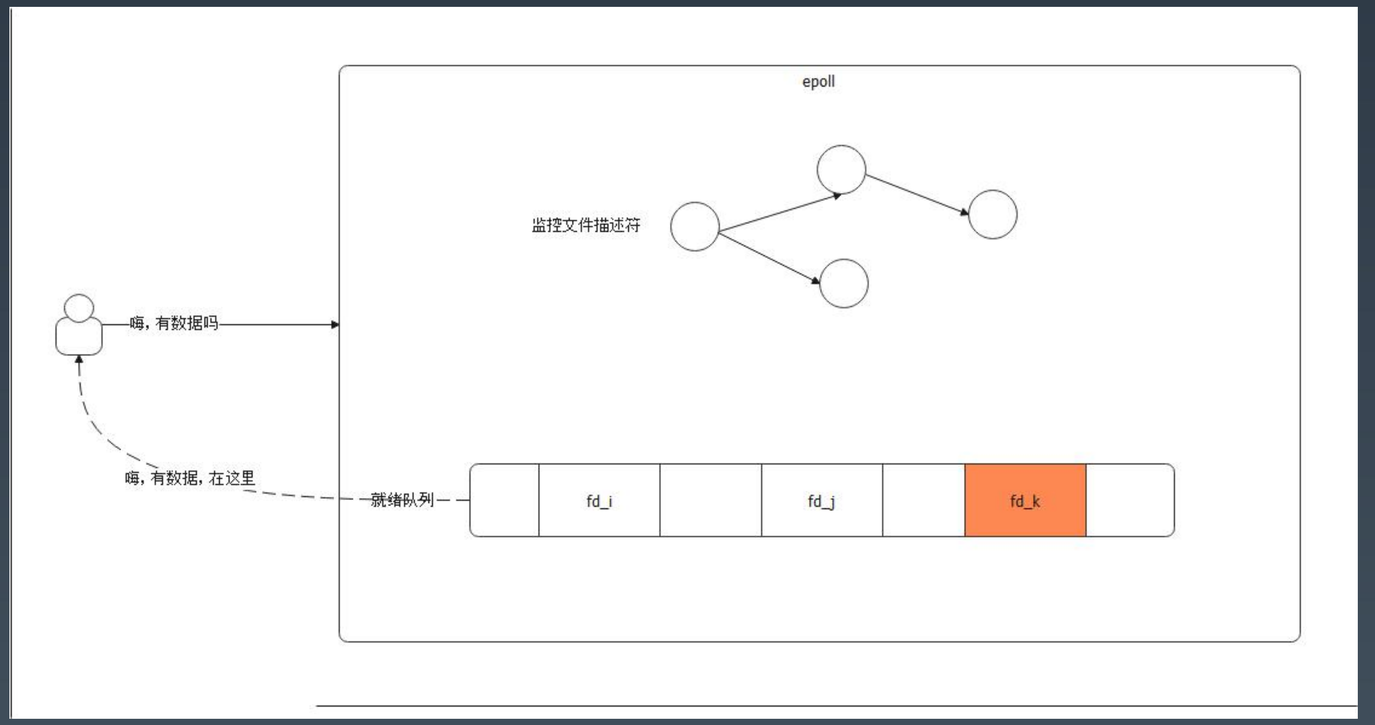
细节：

1. 核心在于三个方法：epoll_create, epoll_ctl, epoll_wait

2. 一个 epoll 对象主要有两个结构，一个是用红黑树来存储被监控文件描述符，一个是就绪队列，存储就绪文件描述符

3. epoll 会监听系统中断，而后将文件描述符挪到就绪队列

4. 用户查询的时候，直接返回就绪队列



知识梳理 —— epoll 和 select 的实现

用户使用代码：

1. 创建 epoll
2. 往 epoll 里面添加文件描述符
3. 不断从 epoll 里面找数据

```
15
16  epoll_fd = epoll_create(100) // 最多一百个文件描述符
17  epoll_ctl(epoll_fd, OP_ADD, fd1)
18  epoll_ctl(epoll_fd, OP_ADD, fd2)
19  // ... 把文件描述符都加进去
20  epoll_ctl(epoll_fd, OP_ADD, fdN)
21
22  for {
23      list = epoll_wait(epoll_fd, READ, timeout)
24      for fd in list {
25          handle(fd)
26      }
27  }
```

知识梳理 —— epoll 和 select 的实现

面试要点：

1. select 和 epoll 的实现方式
2. select 和 epoll 对比
3. epoll 为什么比 select 高效
4. epoll 怎么把文件描述符挪到就绪队列？——关键字：中断
5. epoll 和 select 中超时的效果——有超时就等，没超时就直接返回。利用时钟中断来控制超时；

知识梳理

- epoll 和 select 的实现
- 网络协议设计

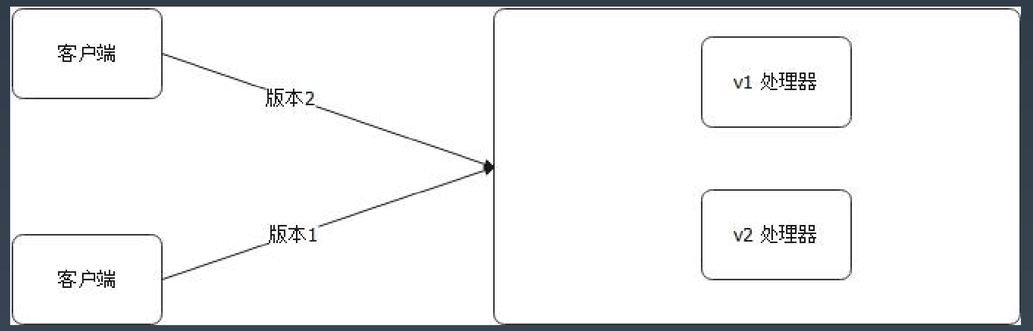
知识梳理——网络协议设计

- 一般分成两个部分：头部(header)和身体(body)
  - header：一般存放和协议相关的元数据，例如魔数、协议版本、数据长度
  - body：请求内容

知识梳理——常见头部

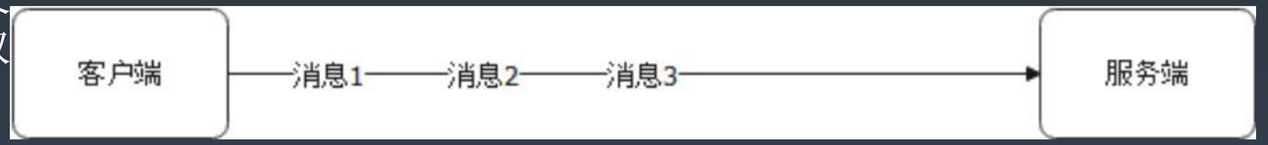
常见头部字段：

- 魔数：没啥实际用，一般就是用来标记某个协议，用作报文快速检测
- 版本：用于版本控制，使得服务端和客户端可以做到前后兼容
- 长度：标记消息有多长
- message id：用于标记请求，一般只有在特殊的时候才会启用



知识梳理——为什么要有长度字段？

它是为了解决：如何分割消息？即客户端源源不断发送数据，就好比流水。而服务器读取数据，应该读取多长，才会刚好读完一个完整的消息？



举个例子：好比全年级参加运动会，人按照班级的组织，由老师（头部）带着挨个进入体育场，你需要把相应班级的人放到相应班级所在的位置。

所以，你站在门口，遇到的第一个人，你知道是老师，问他你们班有多少人，他说五十个人。于是接下来的五十个人你知道就是这位老师班级的人，然后你告诉这些人去A区域。

接着你数下一个，你知道他又是老师，然后他告诉你，他们班有七十个人，于是接下来七十个人你引导过去B区域

...

知识梳理——粘包

那么粘包是什么？粘包就是这个问题，我怎么分割消息。除了头部字段指明之外，还有两种

定长分割：我每个班级固定就是五十个人，不多不少，加上老师也就是五十一个人，分毫不差；

特殊分隔符：每个班级人数不固定，但是除了最后一个人会举着旗子，其他学生都不会拿旗子；

总结：

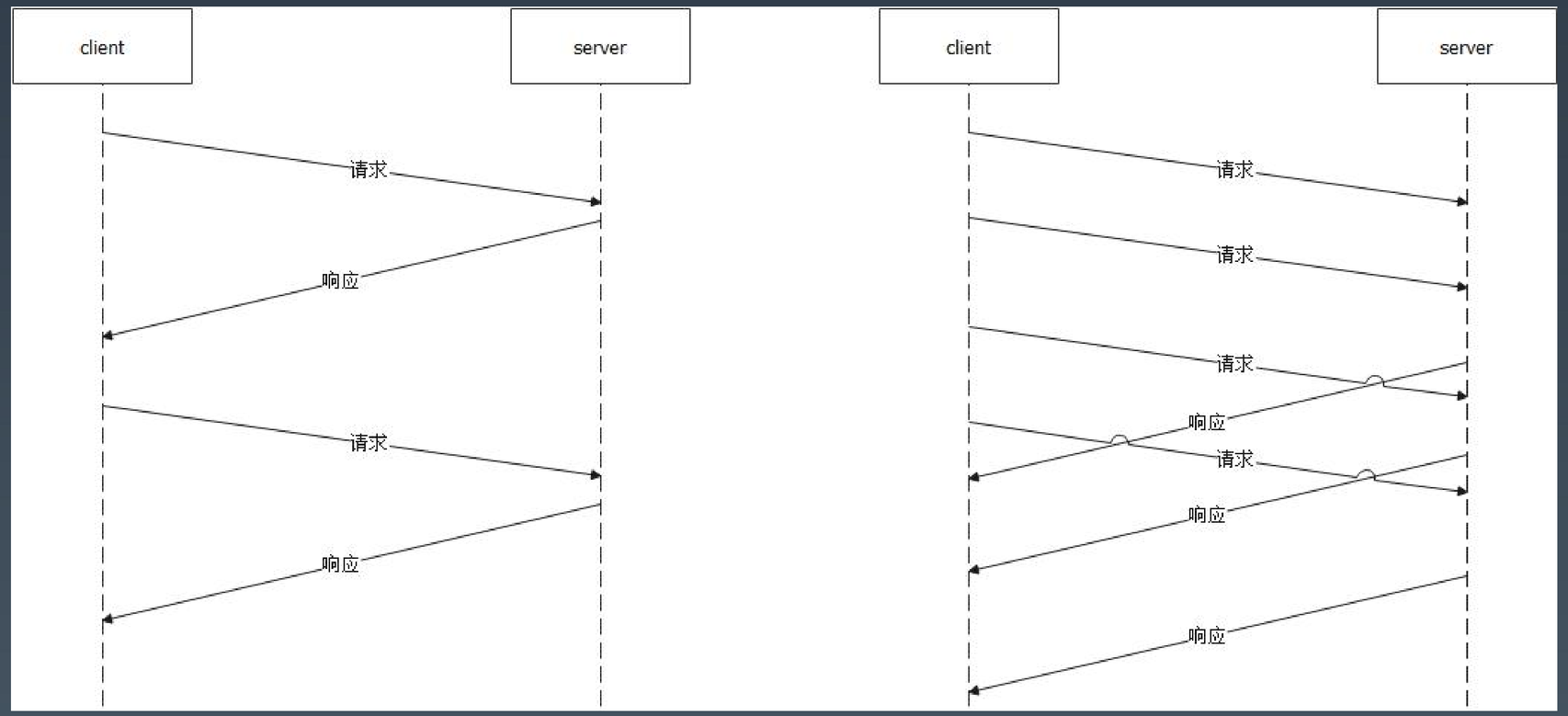
1. 定长分割适用于简单的网络协议，或者很底层的通讯协议，例如 MTU（数据链路层最大传输单元）
2. 特殊分隔符要结合转义符来使用，例如 HTTP 是按照特殊分隔符分割
3. 定长头部 + 不定长 body：如 Dubbo 协议
4. 不定长头部 + 不定长body： 如 GoIM 协议。不定长头部意味着需要一个额外的字段指明头部有多长

可以自己去看看各种协议，TCP，UDP，IP这种底层协议，以及不同的 RPC 框架的 RPC 协议，或者其它自定义应用层协议的框架

知识梳理—— message id

如果我们默认一个请求发出去之后。TCP 连接被 goroutine 一直持有，那么 message id 是不需要的。

但是如果允许 TCP 上请求并发发送，那么就需要。



知识梳理——其它奇奇怪怪的字段

其它奇奇怪怪的字段：

- 状态位：标记通信的效果，例如 Dubbo 协议用了状态位来标记 RPC 请求的处理结果
- 标记位：比较一些特殊信息，Dubbo 协议用了标记位标记是否是事件请求，TCP 用了标记位来标记各种报文，如 sync 标记位
- 超时：有些通讯协议会把超时字段放在头部
- 序列化协议：标记 body 是怎么被编码的
- 压缩算法：标记 body 是怎么被压缩的。有些时候头部也可以分成两部分，一部分被压缩，一部分吧不被压缩

知识梳理——读写数

写入和读取的过程类似镜像：

- 写入就是一个个字段拼，拼好了发送下去

- 读取就是先读各种定长字段，获得消息长度
后一口气读完整条消息，然后逐个部分解析

```
28
29  读取：
30
31  // 读取 package length，只读取 4 字节
32  con.Read(pkgLengthBytes)
33  // 大端解码
34  pkgLength := binary.BigEndian.Uint32(pkgLengthBytes)
35  // 将整个消息读出来，因为我们已经读了四个字节，所以要减去4
36  msg := make([]byte, 0, pkgLength - 4)
37  con.Read(msg)
38  // 读取头部长度
39  headerLengthBytes = msg[0:2]
40  headerLength := binary.BigEndian.Uint32(headerLengthBytes)
41
42  // 整个头部，需要除去长度字段，因为我们已经处理过了
43  header := msg[0:headerLength-4]
44  // 遍历 header 的所有字段
45
46  // 处理 version
47  version := msg[2:4]
48
49  // 处理 operation
50  op := msg[4:8]
51
52  // 处理 sequence id
53  sid := msg[8:12]
54
55  // 头部剩余的数据
56  extraHeader := msg[12:headerLength - 4]
57
58  // body 长度
59  body := msg[headerLength-4:]
60  // 处理消息
61  handle(version, op, sid, extraHeader, body)
```