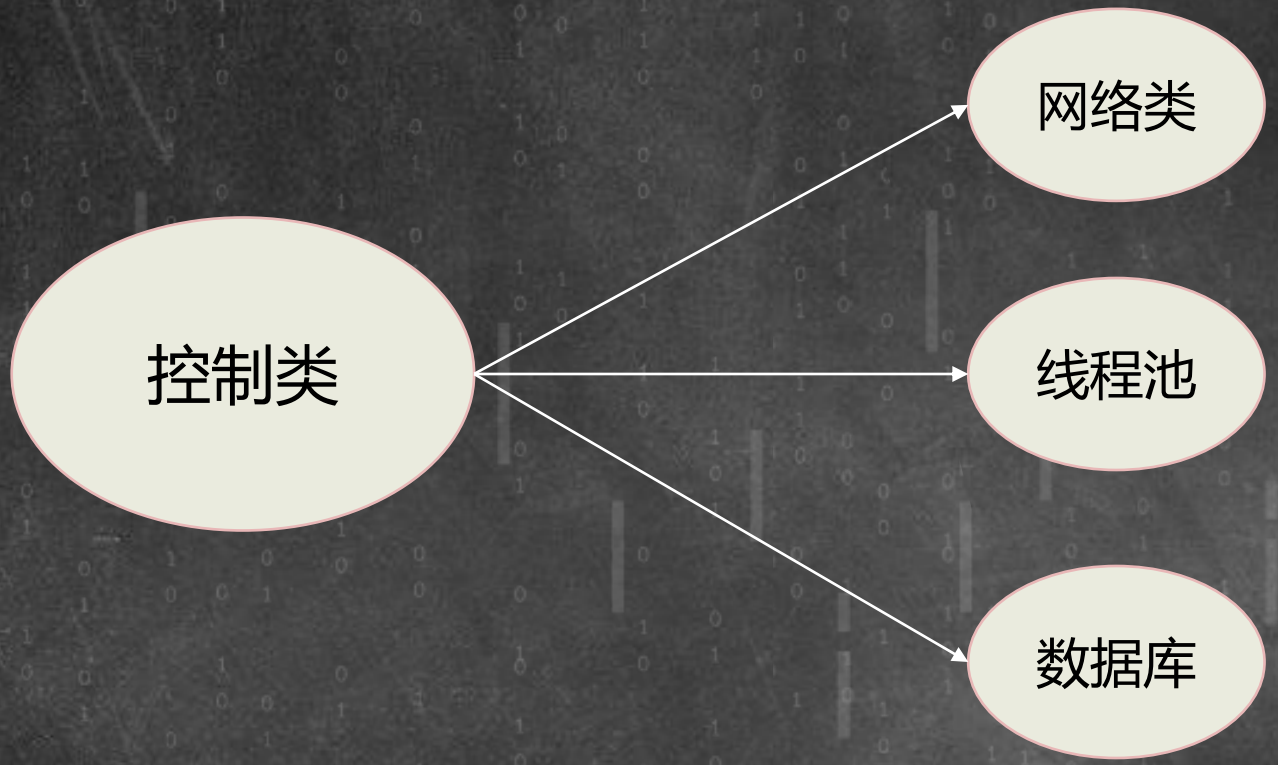
服务器架构



数据库

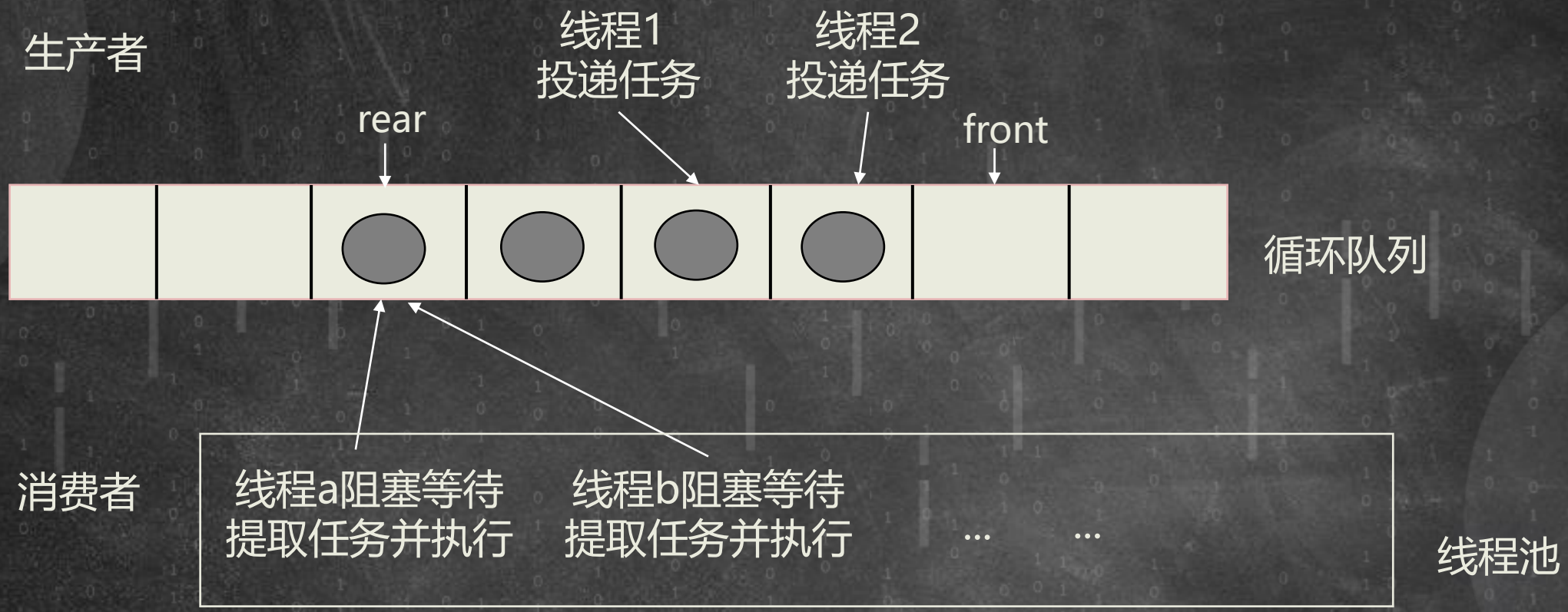
1.安装mysql 数据库服务器

2.安装mysql开发库 libmysqlclient

3.编译的时候添加 -lmysqlclient

注意中文编码的问题

线程池　采用生产者消费者模型



循环队列

成员

任务数组 arr[MAX_QUEUE_SIZE];

队列头 front;

队列尾 rear;

读写互斥锁 lock;　　//避免多线程并发

条件变量not_empty;　　//队列空, 消费者等待

条件变量not_full;　　//队列满, 生产者等待

线程池

0. 开始创建很多线程, 处于阻塞等待

1. 某线程中创建任务, 队列未满时投递任务, 同时发送条件变量信号;

队列满, 则等待.

2. 队列里有任务投递后, 一个空闲线程收到条件变量信号, 从队列尾取出任务并执行

3. 执行后回来继续等待条件变量信号

线程池管理

0. 创建一个线程用于管理线程池 , 每隔一段时间查看忙碌线程数/存活线程数

1. 在空闲线程占比不足 20%时, 创建一些线程

2. 在空闲线程占比超过 66%时, 销毁一些线程

网络类 采用epoll模型阻塞socket

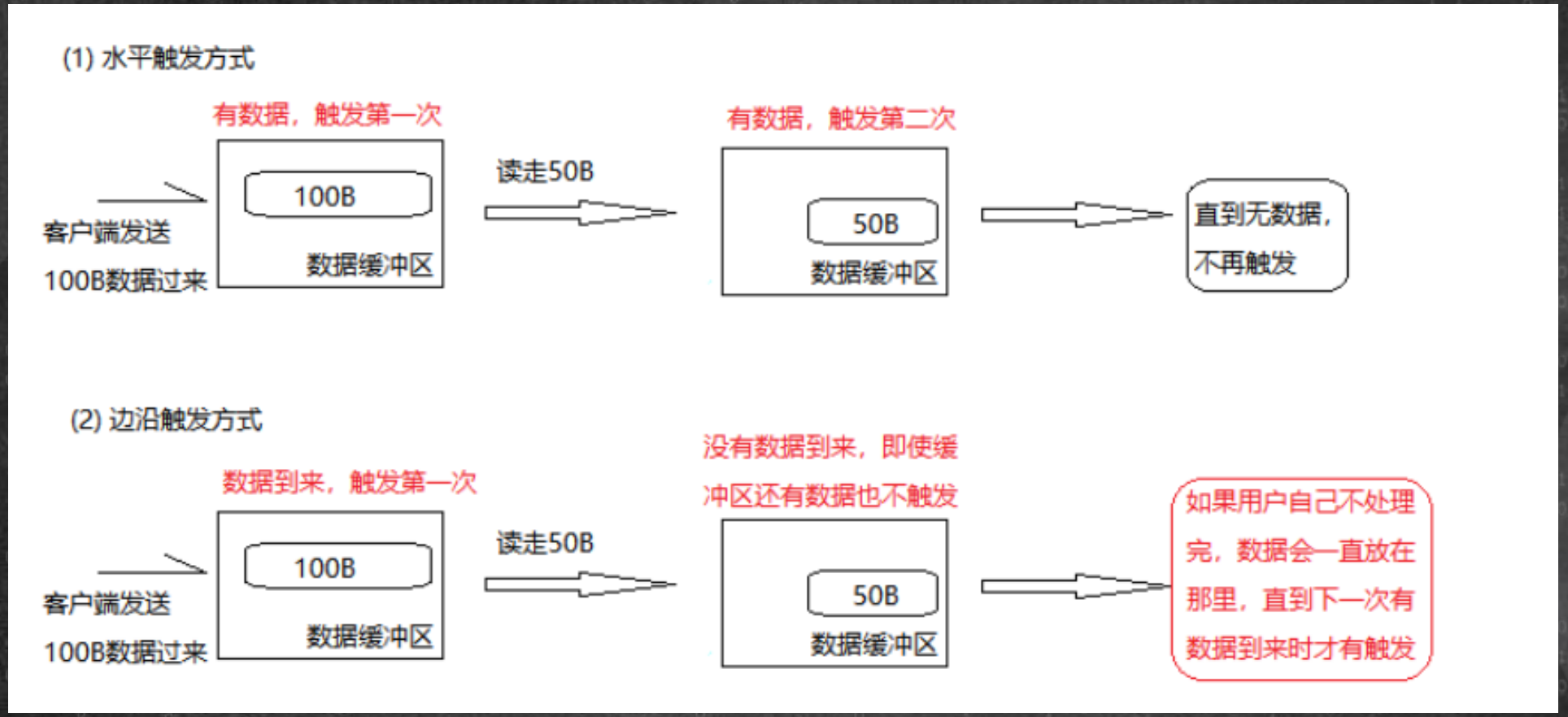
为什么是epoll?

- 同步阻塞多线程，具有编程简单，问题少等优点，但不足是无法处理过多客户端
- 使用多路复用IO模型中较优秀的epoll，可以解决上面的问题，并且处理网络请求效率高，被多数优秀服务器作为首选模型.

为什么使用阻塞socket?

- 这里阻塞socket，指accept客户端连接后，返回的套接字，这种阻塞套接字好处在于代码简单，出错少，更适合入口学习.
- 但也有不可避免的IO效率问题.

epoll ET or LT



epoll ET or LT

ET和LT 差别

- ET 边缘触发，一般是有到没有，没有到有，才会有触发
- 对于接收也就是一次数据来了，事件通知，如果你没读取完，不会再次事件通知
- 对于发送，发送缓冲区由满到不满，触发一次，不满到满，触发一次，而且都是仅一次
- LT 水平触发，只要有数据要读，有空间可以写，就会触发
- 对于接收，有数据没读完，就会通知
- 对于发送，发送缓冲区，有空间可写，就触发通知

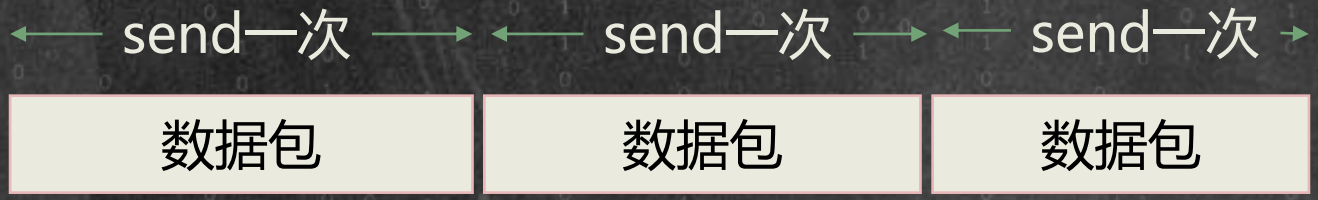
ET 与 LT 选择

- 对于主套接字accept，采用非阻塞LT模式，避免没有读取，不触发而丢失连接
- 对于客户端套接字，采用LT模式，更简单，避免数据没有读取

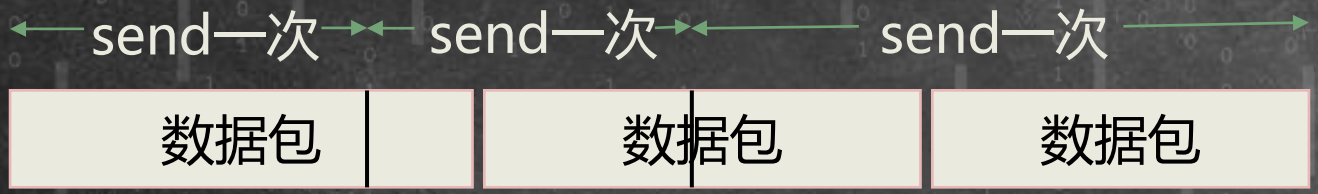
同步和异步IO的差别

- 异步IO就是系统完成数据从内核缓冲区到用户缓存区的拷贝, 并返回一个信号或通知
- 同步IO就是需要代码实现数据从内核缓冲区到用户缓存区的拷贝

阻塞send

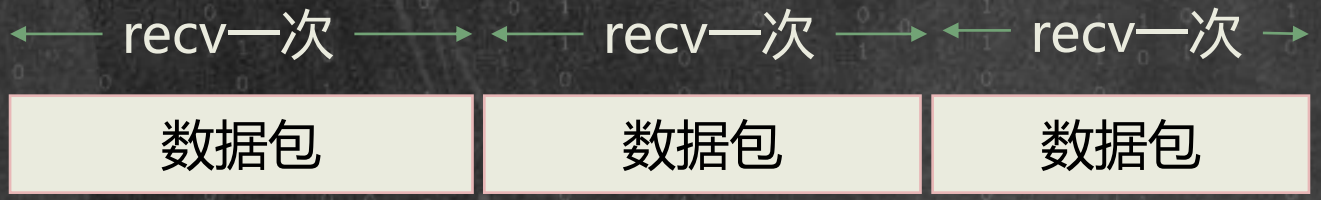


非阻塞send

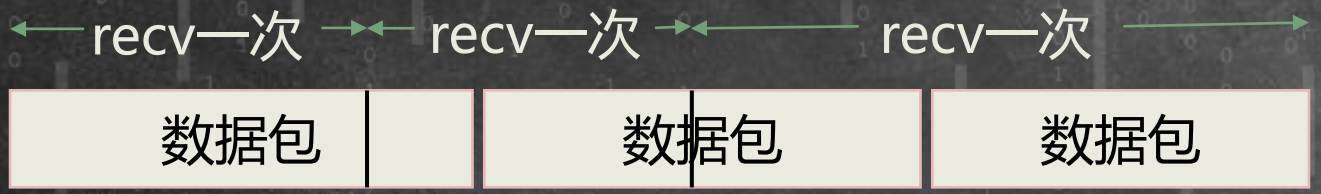


总结: 非阻塞send, 需要添加发送数据缓冲层, 发送缓冲区填满时 ,将没有发送完的数据缓存, 等待缓冲区有空间时(参考epoll EPOLLOUT 事件)再发送

阻塞recv



非阻塞recv



总结: 非阻塞recv, 需要添加接收数据缓冲层, 对于epoll ET模式,

一次事件, 都读取到接收数据缓冲,

直到(errno == EAGAIN) , 判断接收缓冲区里面的数据, 分解成一个个协议包

阻塞和非阻塞IO的差别

首先要清楚阻塞和非阻塞send的差别

- send是将用户缓冲区数据拷贝到发送缓冲区, 并不代表send就直接发送给对方用户
- 阻塞send, 等待发送缓冲区有足够空间将数据拷贝到发送缓冲区, 再拷贝返回
- 非阻塞send, 发送缓冲区有多少空间,用户数据就拷贝多少, 返回成功拷贝的字节数

  可能无法一次发送完数据, 需要循环send发送

然后要清楚阻塞和非阻塞recv的差别

- recv是将数据从接收缓冲区拷贝用户缓冲区
- 阻塞recv , 等待接收缓冲区有数据, 有数据会尽可能拷贝用户指定长度数据, 返回拷贝长度
- 非阻塞recv , 接收缓冲区有多少数据就拷贝多少数据到用户缓冲区,

  因而可能存在数据包数据不完整的情况 , 需要循环recv接收

反应堆 reactor 模型

三个重要组件

- 多路复用器

同时阻塞多个fd, 监听读写事件

- 事件分发器

就绪事件分发到处理器

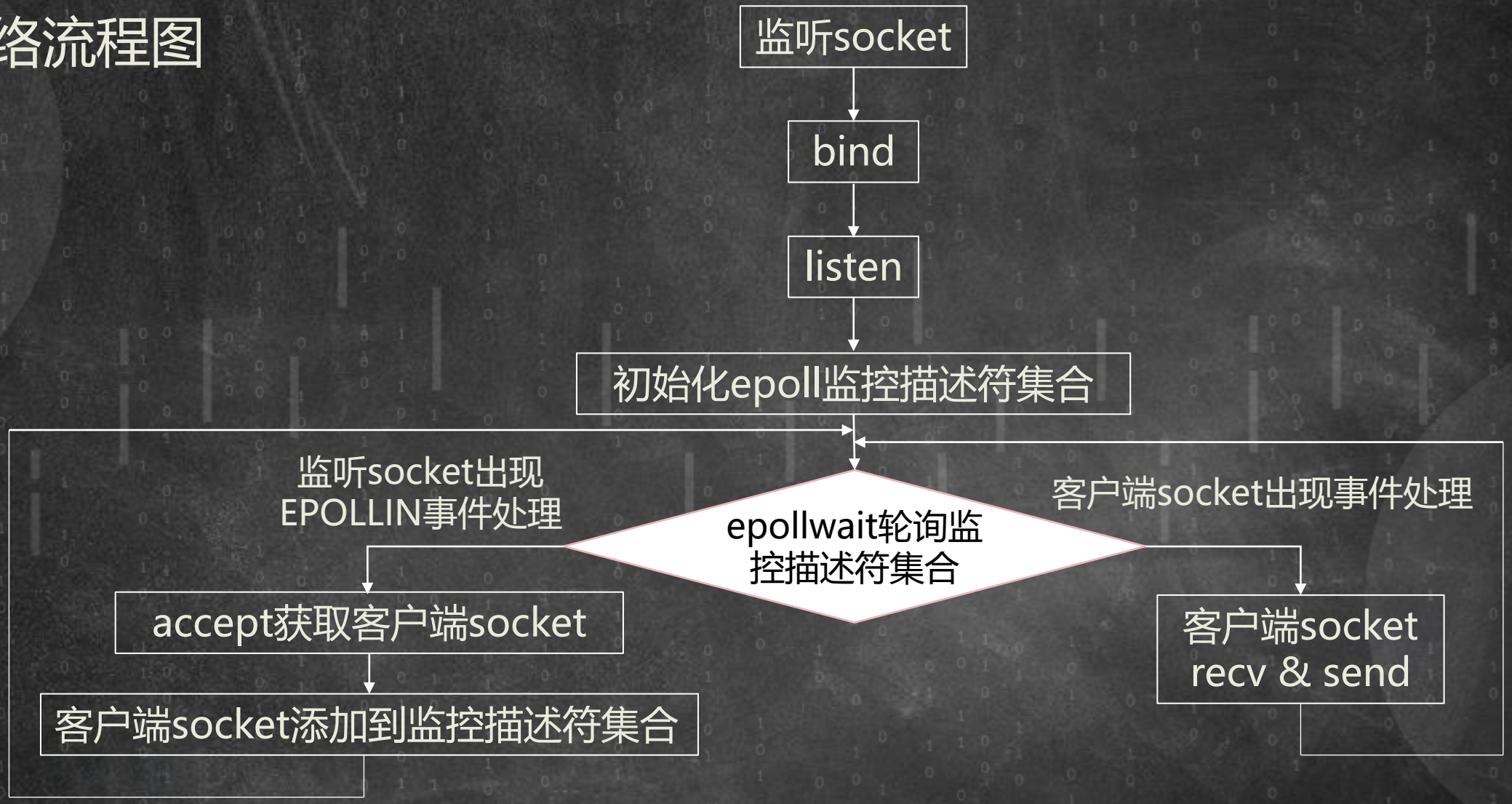
- 事件处理器

负责处理特定事件的处理函数

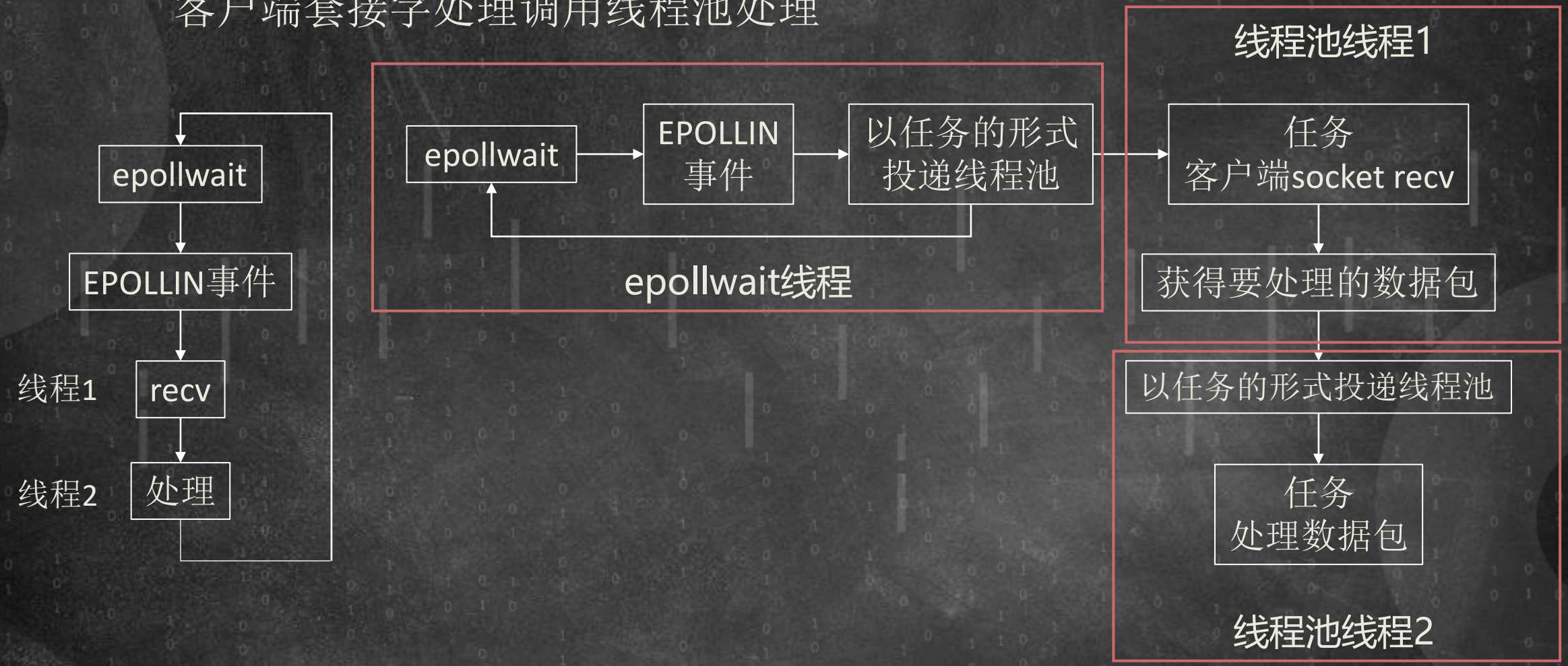
epoll_wait( ) 多路复用

```
if(  fd & EPOLLIN ){ //分发器
        read_cb( );  //处理器
}
if(  fd & EPOLLOUT ){ //分发器
        write_cb( );  //处理器
}
```

网络流程图


监听socket
bind
listen
初始化epoll监控描述符集合
监听socket出现
EPOLLIN事件处理
epollwait轮询监
控描述符集合
客户端socket出现事件处理
accept获取客户端socket
客户端socket添加到监控描述符集合
客户端socket
recv & send

客户端套接字处理调用线程池处理



EPOLLONESHOT事件

使用场合：

一个线程在读取完某个socket上的数据后开始处理这些数据，而数据的处理过程中该socket又有新数据可读，此时另外一个线程被唤醒来读取这些新的数据。

于是，就出现了两个线程同时操作一个socket的局面。可以使用epoll的EPOLLONESHOT事件实现一个socket连接在任一时刻都被一个线程处理。

作用：

对于注册了EPOLLONESHOT事件的文件描述符，操作系统最多触发其上注册的一个可读，可写或异常事件，且只能触发一次。同一时刻肯定只有一个线程在为它服务，避免了多线程可能的竞争.

使用：

注册了EPOLLONESHOT事件的socket一旦被某个线程处理完毕，该线程就应该立即重置这个socket上的EPOLLONESHOT事件，以确保这个socket下一次可读时，其EPOLLIN事件能被触发，进而让其他工作线程有机会继续处理这个sockt。

感谢大家的聆听

Colin 科林明伦