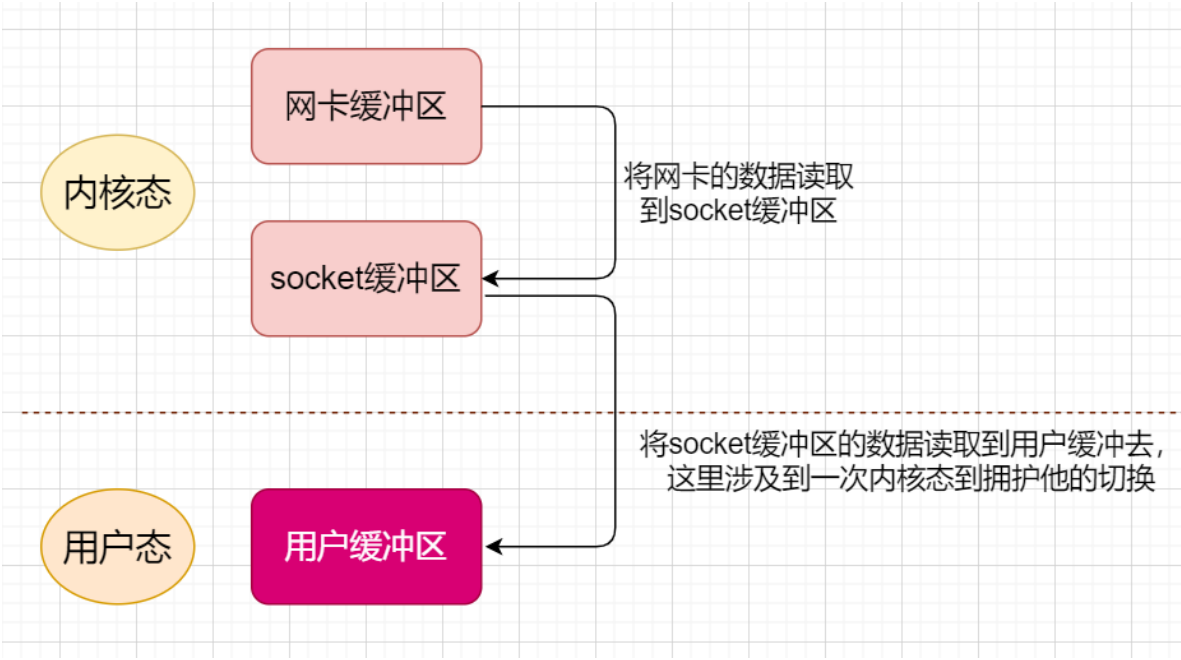
并发服务器

一、五种网络IO模型

1、数据传输过程

操作系统为了保护自己，设计了用户态、内核态两个状态。应用程序一般工作在用户态，当调用一些底层操作的时候（比如 IO 操作），就需要切换到内核态才可以进行。



服务器从网络接收的大致流程如下：

- 数据通过计算机网络传到网卡
- 把网卡的数据读取到socket缓冲区
- 把socket缓冲区读取到用户缓冲区，之后应用程序就可以使用

核心就是两次读取操作，五种网络IO 模型的不同之处也就在于这两个读取操作怎么交互

2、两组重要概念

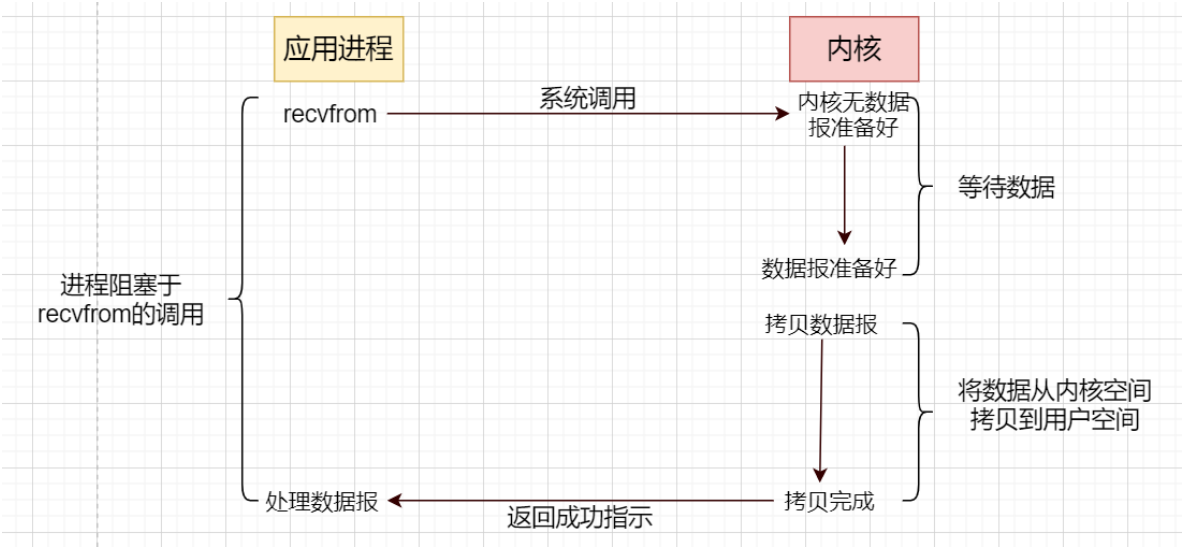
阻塞/非阻塞关注的是**用户态进程/线程的状态**，其要访问的数据是否就绪，进程/线程是否需要等待。当前接口数据还未准备就绪时，线程是否被阻塞挂起。何为阻塞挂起？就是当前线程还处于CPU时间片当中，调用了阻塞的方法，由于数据未准备就绪，则时间片还未到就让出CPU。而非阻塞就是当前接口数据还未准备就绪时，线程不会被阻塞挂起，可以不断轮询请求接口，看看数据是否已经准备就绪。

同步/异步关注的是**消息通信机制**。所谓同步，就是在发出一个调用时，自己需要参与等待结果的过程，则为同步。同步需要主动读写数据，在读写数据的过程中还是会阻塞。异步IO，则指出发出调用以后到数据准备完成，自己都未参与，则为异步。异步只需要关注IO操作完成的通知，并不主动读写数据，由**操作系统内核**完成数据的读写。

3、五种网络IO模型

3.1、阻塞式IO

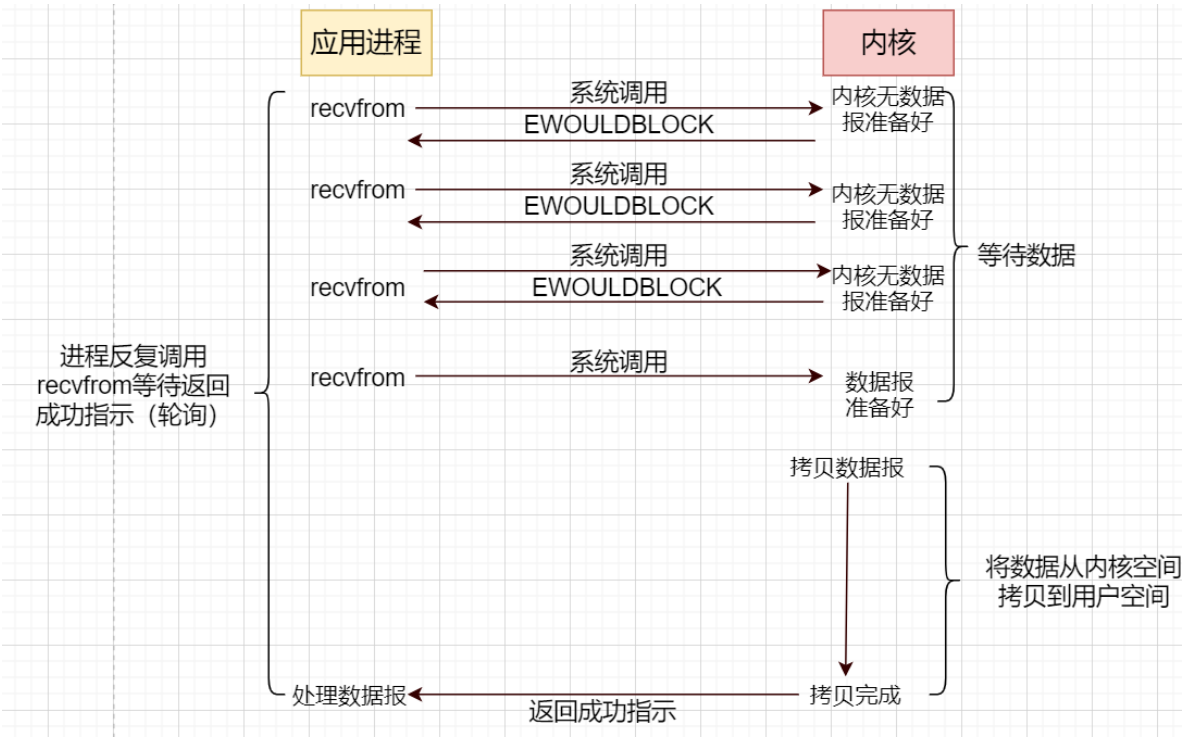
应用调用recvfrom读取数据时，其系统调用直到数据包到达且被复制到应用缓冲区中或者发送错误时才返回，在此期间一直会等待，进程从调用到返回这段时间内都是被阻塞的。在内核将数据准备好之前，系统调用会一直等待。所有的套接字, 默认都是阻塞方式。



- 应用进程向内核发起recfrom读取数据
- 内核进行准备数据报（此时应用进程阻塞）
- 内核将数据从内核复制到应用空间。
- 复制完成后，返回成功提示

3.2、非阻塞式IO

当应用进程发起读取数据申请时，如果内核数据没有准备好会即刻告诉应用进程，不会让应用进程在这里等待,如果内核还未将数据准备好, 系统调用仍然会直接返回, 并且返回EWOULDBLOCK错误码。非阻塞IO往往需要程序员循环的方式反复尝试读写文件描述符

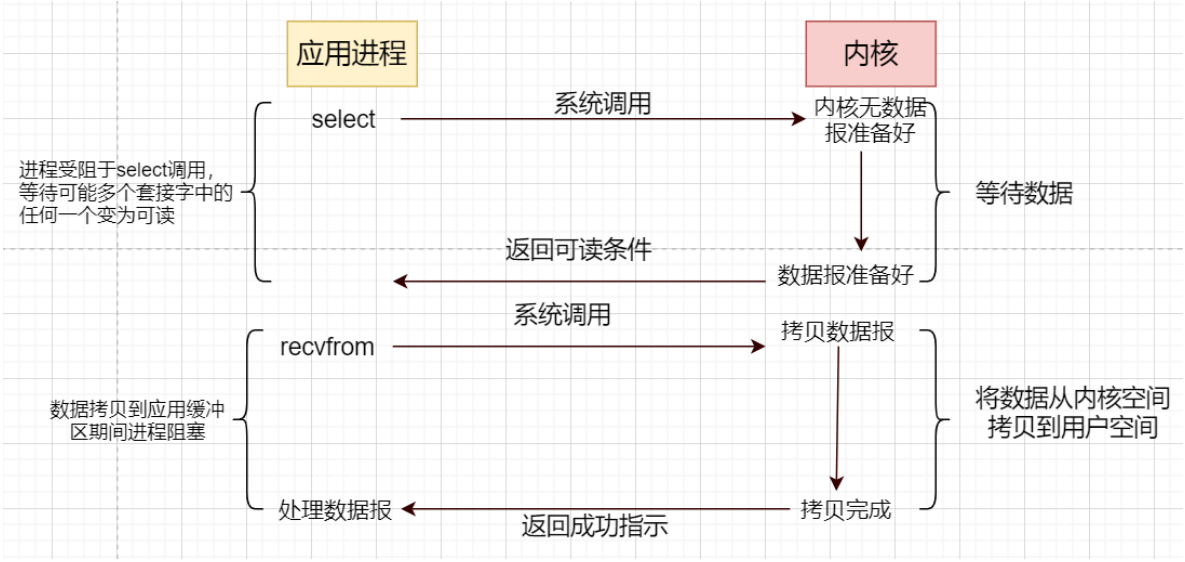


- 应用进程向内核发起recvfrom读取数据。
- 内核数据报没有准备好，即刻返回EWOULDBLOCK错误码。
- 应用进程再次向内核发起recvfrom读取数据。
- 内核如果已有数据包准备好就进行下一步骤，否则还是返回错误码
- 内核将数据拷贝到用户空间。

- 完成后，返回成功提示

3.3、IO多路复用

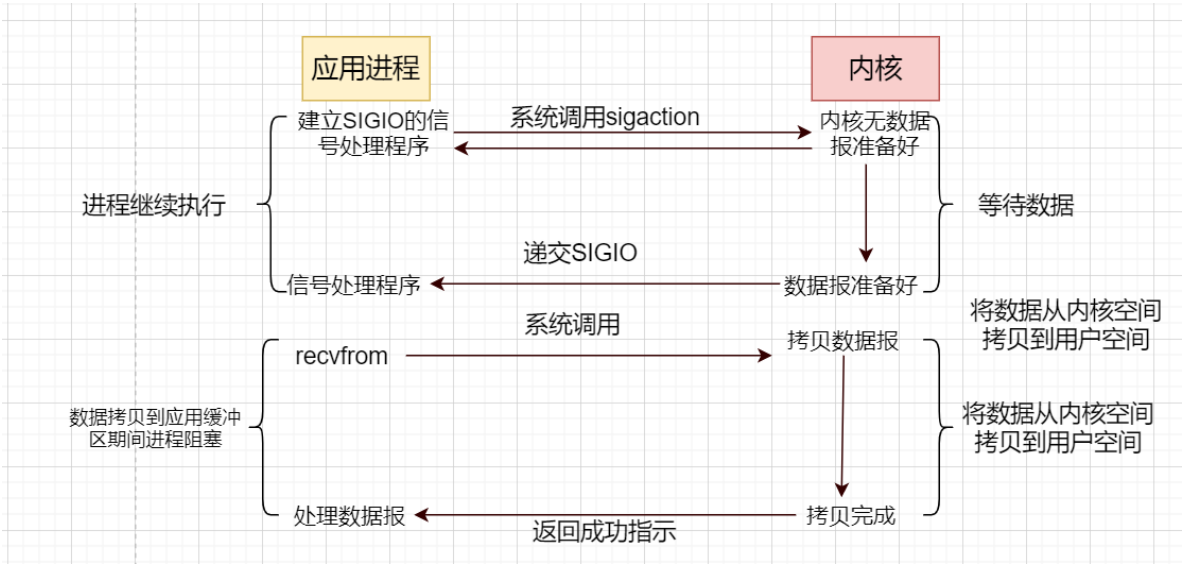
由一个线程监控多个网络请求（linux系统把所有网络请求以一个文件描述符来标识）,来完成数据状态询问的操作，当有数据准备就绪之后再分配对应的线程去读取数据。



- 应用进程向内核发起recvfrom读取数据
- 内核进行准备数据报（此时应用进程阻塞）
- 内核倘若已有数据包准备好则通知应用线程
- 内核将数据拷贝到用户空间
- 完成后，返回成功提示

3.4、信号驱动式IO

信号驱动IO是在调用sigaction时候建立一个SIGIO的信号联系，当内核准备好数据之后再通过SIGIO信号通知线程,此文件描述符准备就绪；当线程收到可读信号后，此时再向内核发起recvfrom读取数据的请求，因为信号驱动IO的模型下，应用线程在发出信号监控后即可返回，不会阻塞，所以一个应用线程也可以同时监控多个文件描述符。

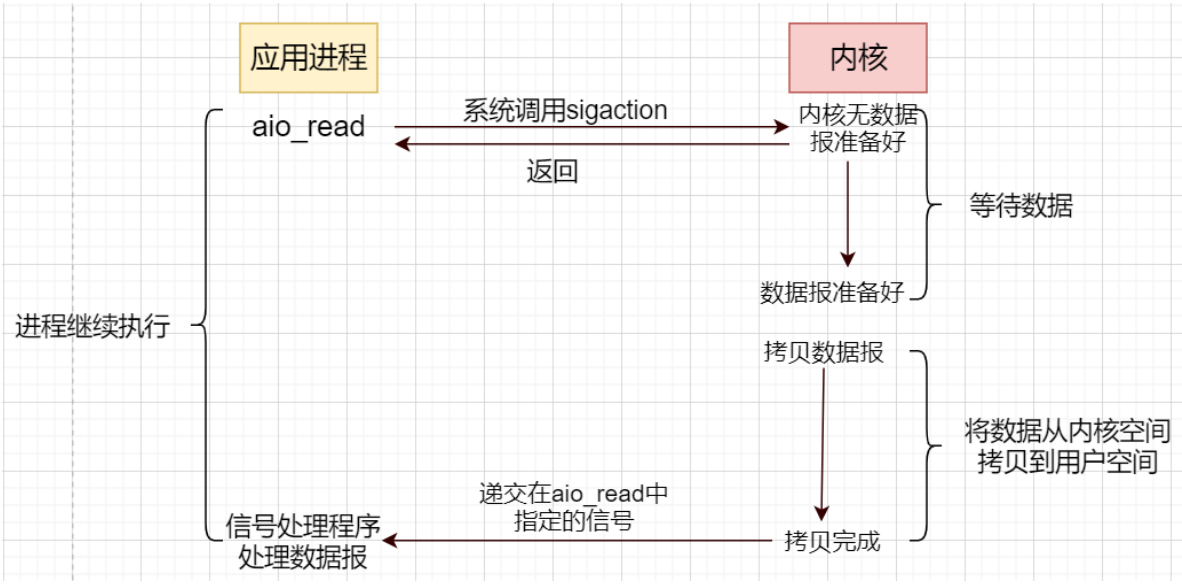


- 应用进程向内核发起recvfrom读取数据
- 内核进行准备数据报，即刻返回
- 内核倘若已有数据包准备好则通知应用线程
- 应用进程向内核发起recvfrom读取数据
- 内核将数据拷贝到用户空间

- 完成后，返回成功提示

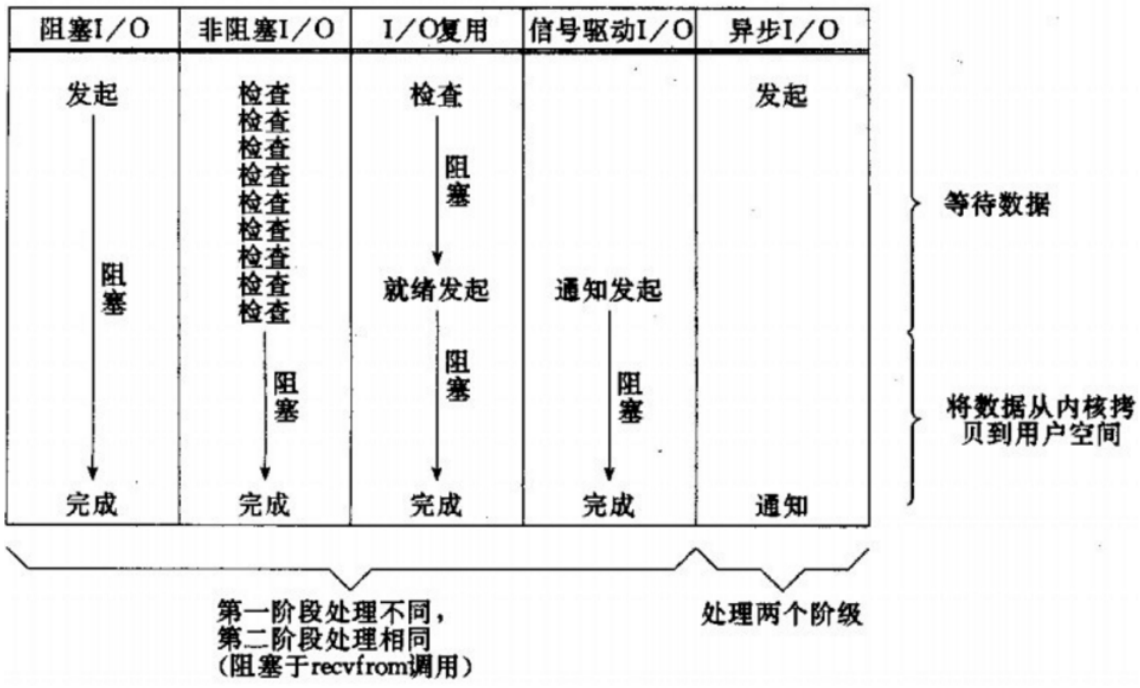
3.5、异步IO

应用进程只需要向内核发送一个读取请求，告诉内核它要读取数据后即刻返回；内核收到请求后会建立一个信号联系，当数据准备就绪，内核会主动把数据从内核空间复制到用户空间，等所有操作都完成之后，内核会发起一个通知告诉应用进程。



- 应用进程向内核发起recvfrom读取数据
- 内核进行准备数据报，即刻返回
- 内核收到后会建立一个信号联系，如果已有数据包准备好，内核将数据拷贝到用户空间
- 完成后，返回成功提示

4、五种网络IO模型对比



5、举例说明

场景：哩哩去新华书店买《王道机试指南》

- 如果新华书店没有，就一直等着书店有了书之后才走离开 (同步阻塞)

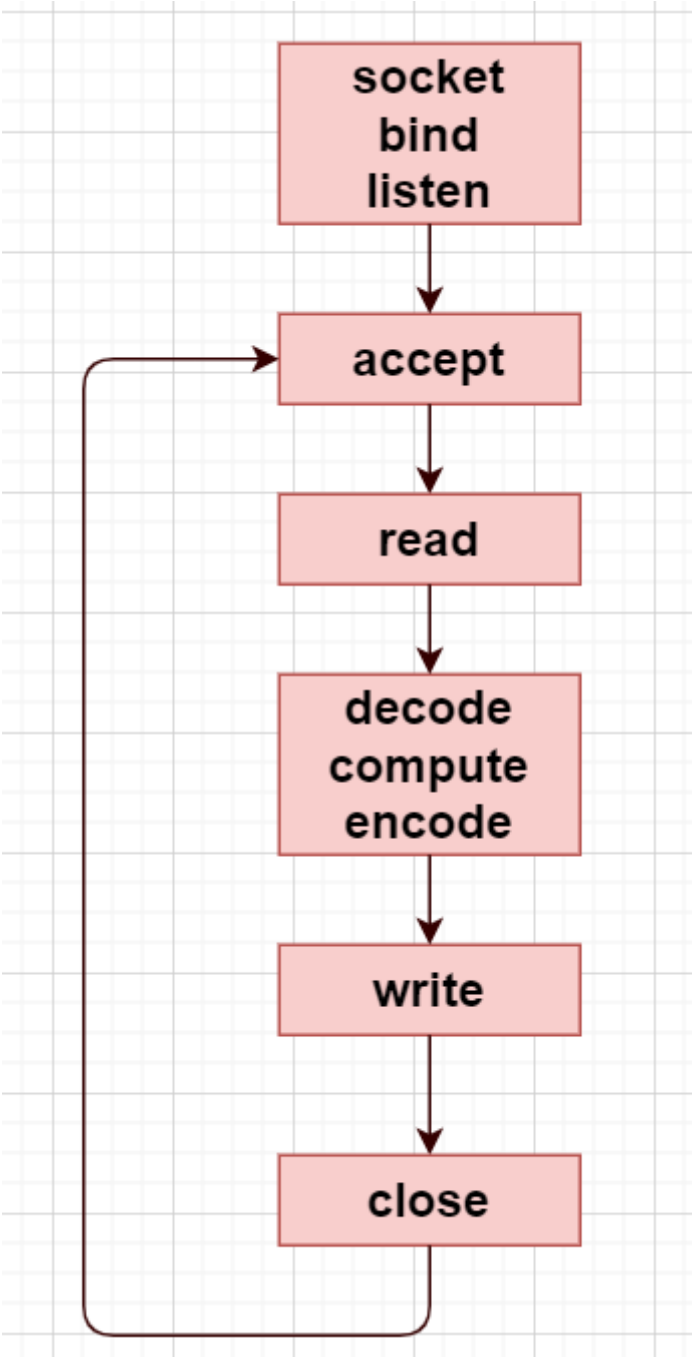
- 如果新华书店没有，先离开书店；然后每天都去书店逛一次，直到书店到货了，买了就走 。(同步非阻塞)
- 如果新华书店没有，留下电话号码；书店有货时，老板打电话通知他，他再去书店买书。(信号驱动IO同步非阻塞)
- 如果新华书店没有，留下地址；书店有货时，老板直接把书送到家(异步非阻塞)

  对应于程序：用户进程调用系统调用。**用户进程**对应哩哩，**内核**对应书店老板，**书**对应**数据**， 买书就是一个**系统调用**，而内核拷贝数据到进程这个过程近似于老板送书到哩哩手中。

二、并发服务器模型

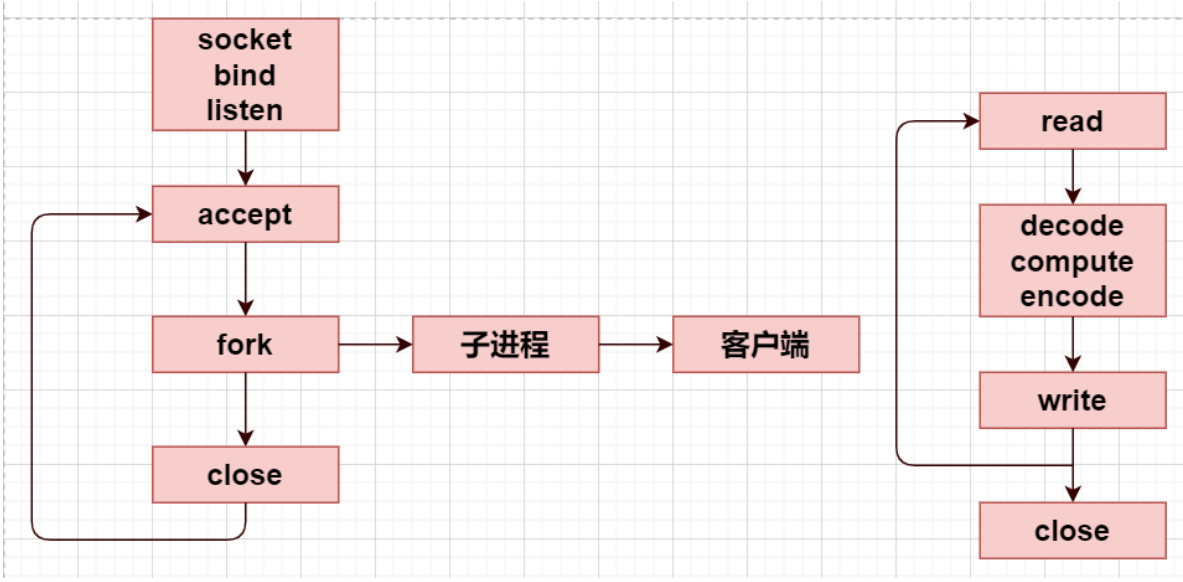
1、循环式迭代式模型

一种单线程的应用程序，它只能使用短连接而不能使用长连接，缺点是无法充分利用多核CPU，不适合执行时间较长的服务，即适用于短连接（这样可以处理多个客户端），如果是长连接则需要在read/write之间循环，那么只能服务一个客户端。所以循环式服务器只能使用短连接，而不能使用长链接，否则无法处理多个客户端的请求，因为整个程序是一个单线程的应用程序。



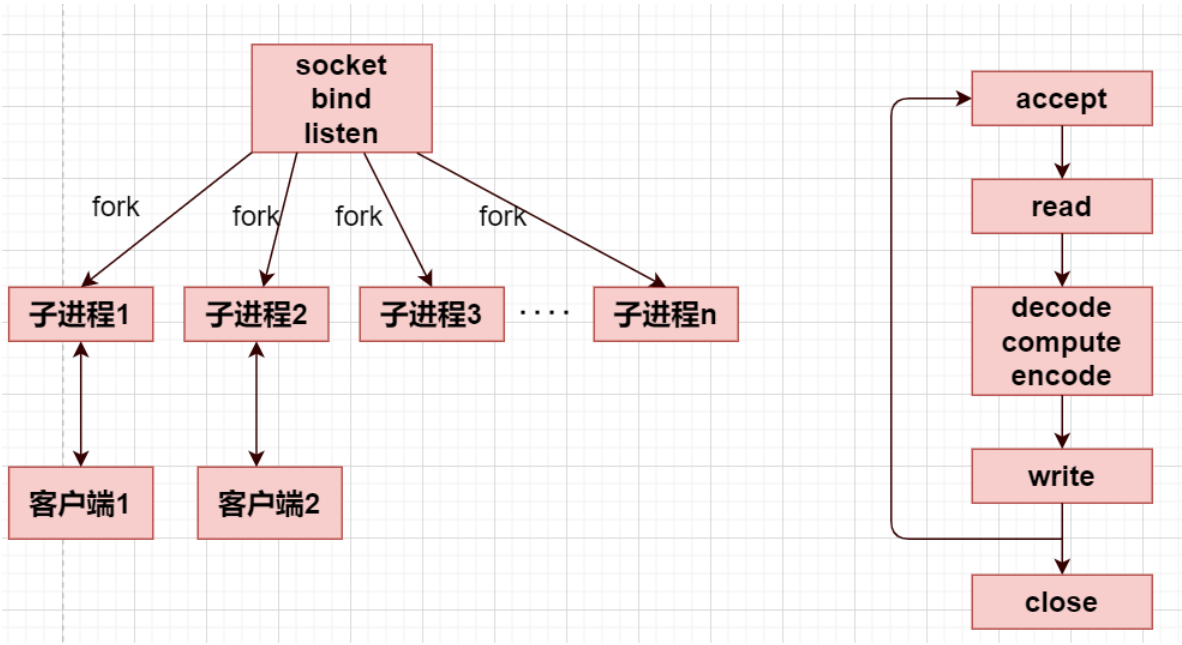
2、并发式服务器

适合执行时间比较长的服务。在父进程中要关闭创建连接的套接字，等待下一个客户端请求连接，所以可以并发式服务器处理多个客户端的请求，一个客户端一个进程；子进程在处理客户端的请求，子进程是长连接的，不断的处理请求，即使子进程中解包，计算，打包的过程时间过长，也不会影响父进程去连接其他客户端的请求。本模型也适用于线程，主线程每次accept 回来就创建一个子线程服务，由于线程共享文件描述符，故不用关闭，最后一个线程关闭监听套接字。



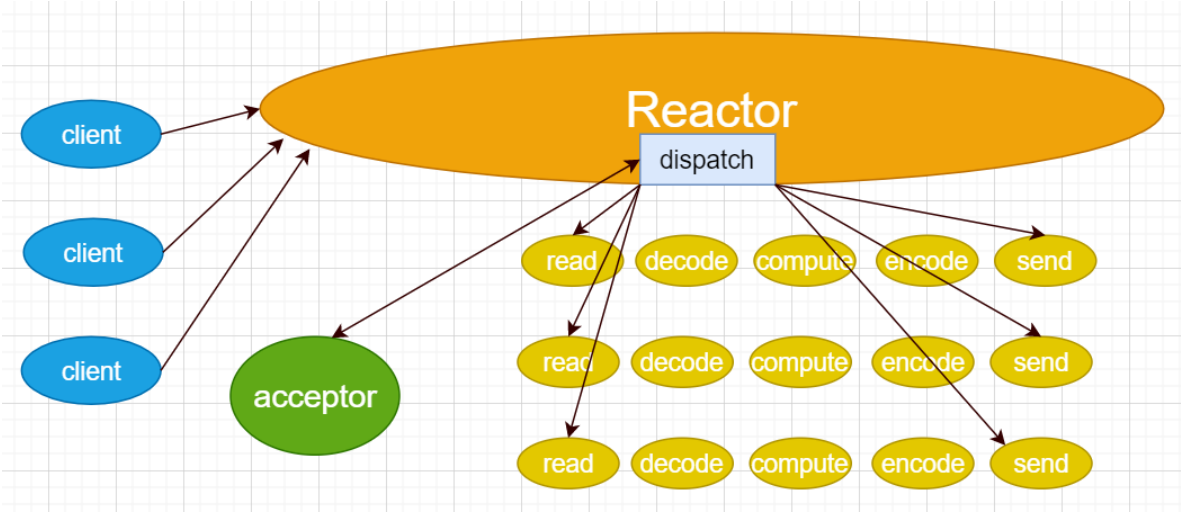
3、prefork服务器

它处理连接的进程/线程都是预先创建好的，因此可以减小创建进程/线程的开销，能够提高响应速速。进程预先fork了n个子进程（n的数目需要提前设定），每个子进程负责和客户端的通信，每个子进程执行的都是右图的流程。前面的创建套接字，绑定端口号，监听套接字这些步骤，每个预先创建的进程已经完成，他们分别调用accept并由内核置入睡眠状态。这种服务器的优点是：提高了响应速度，不需要引入父进程执行fork的开销，新客户就能得到处理。缺点在于：每次启动服务器，父进程必须预测到底需要产生多少子进程，还有一个就是如果不考虑再派生子进程，先前派生的子进程可能被客户请求占用完，以后新到的请求只能先完成三次握手，并且达到listen接口的最大并发连接数backlog，直到有子进程可用。服务器才调用accept，将这些已经完成的连接传递给accept。



4、反应式服务器

服务器可以并发处理多个请求。不过本质上这些请求还是在一个线程中完成的，即：单线程轮询多个客户端。也无法充分利用多核CPU，不适合执行时间比较长的服务，所以为了让客户感觉是在“并发”处理而不是“循环”处理，每个请求必须在相对较短时间内执行。当然如果这个请求不能在有限的时间内完成我们可以将这个请求拆分开来，使用有限状态机机制来完成。其中的Reactor可以使用IO多路复用select/poll/epoll。



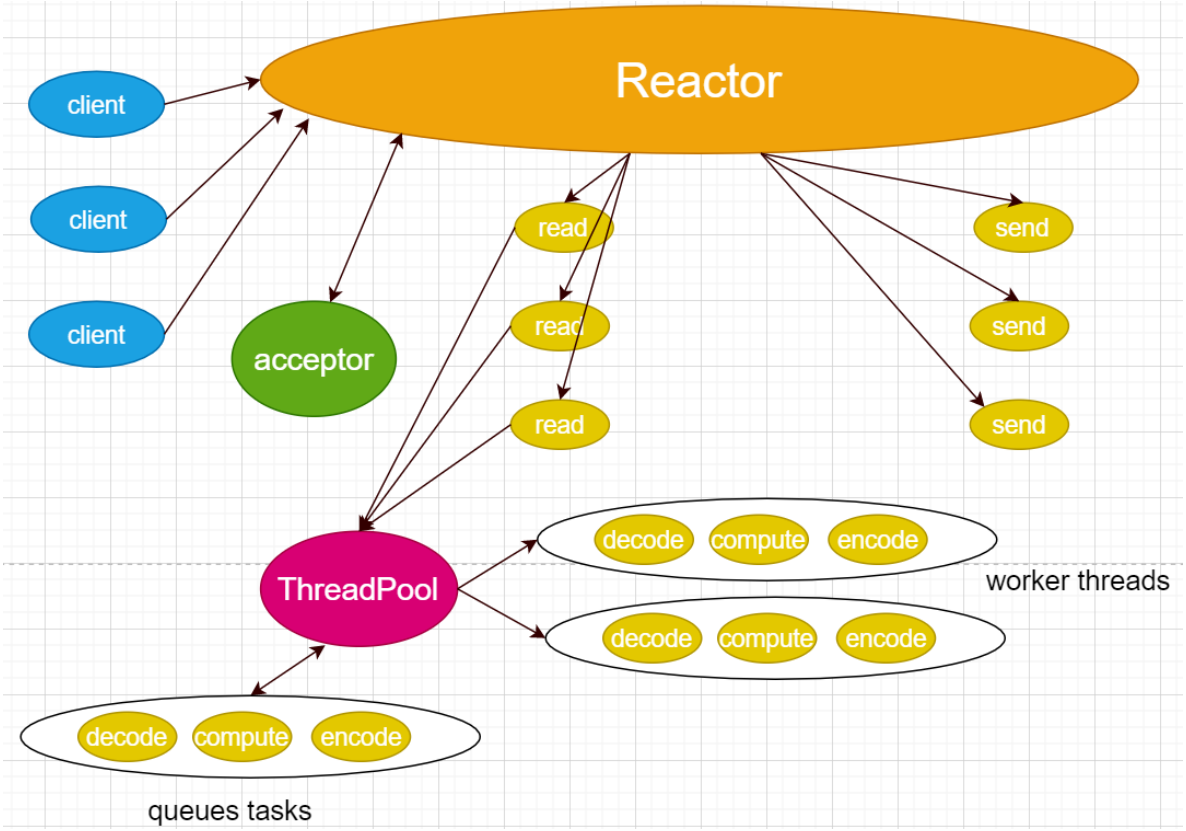
模型的过程解析：

- Reactor是一个线程对象，该线程会启动事件循环，并使用select/poll/epoll来实现IO多路复用。注册一个Acceptor事件处理器到Reactor中，Acceptor事件处理器所关注的事件是accept事件，这样Reactor会监听客户端向服务器端发起的连接请求事件。
- 客户端向服务器端发起一个连接请求，Reactor监听到了该accept事件的发生并将该accept事件派发给相应的Acceptor处理器来进行处理。Acceptor处理器通过accept方法得到与这个客户端对应的连接，然后将该连接所关注的读事件以及对应的read读事件处理器注册到Reactor中，这样Reactor就会监听该连接的read事件了。或者当你需要向客户端发送数据时，就向Reactor注册该连接的写事件和其处理器。
- 当Reactor监听到有读或者写事件发生时，将调用dispatch将相关的事件分发给对应的处理器进行处理。比如，读处理器会通过read方法读取数据，此时read操作可以直接读取到数据，而不会堵塞与等待可读的数据到来。
- 每当处理完所有就绪的I/O事件后，Reactor线程会再次执行IO多路复用的函数阻塞等待新的事件就绪并将其分派给对应处理器进行处理。

目前的单线程Reactor模式中，不仅I/O操作在该Reactor线程上，连非I/O的业务操作也在该线程上进行处理了，这可能会大大延迟I/O请求的响应。所以我们应该将非I/O的业务逻辑操作从Reactor线程上分离出去，以此来加速Reactor线程对I/O请求的响应。这种的服务器的并发量比并发式服务器多，因为并发式服务器能够创建的进程或者线程数目是有限的

5、反应式 + 线程池型服务器

与单线程Reactor模式不同的是，增加了线程池对象，并将业务逻辑的处理从Reactor线程中移出转交给工作的线程池来执行。这样能够提高Reactor线程的I/O响应，不至于因为一些耗时的业务逻辑而延迟对后面I/O请求的处理。

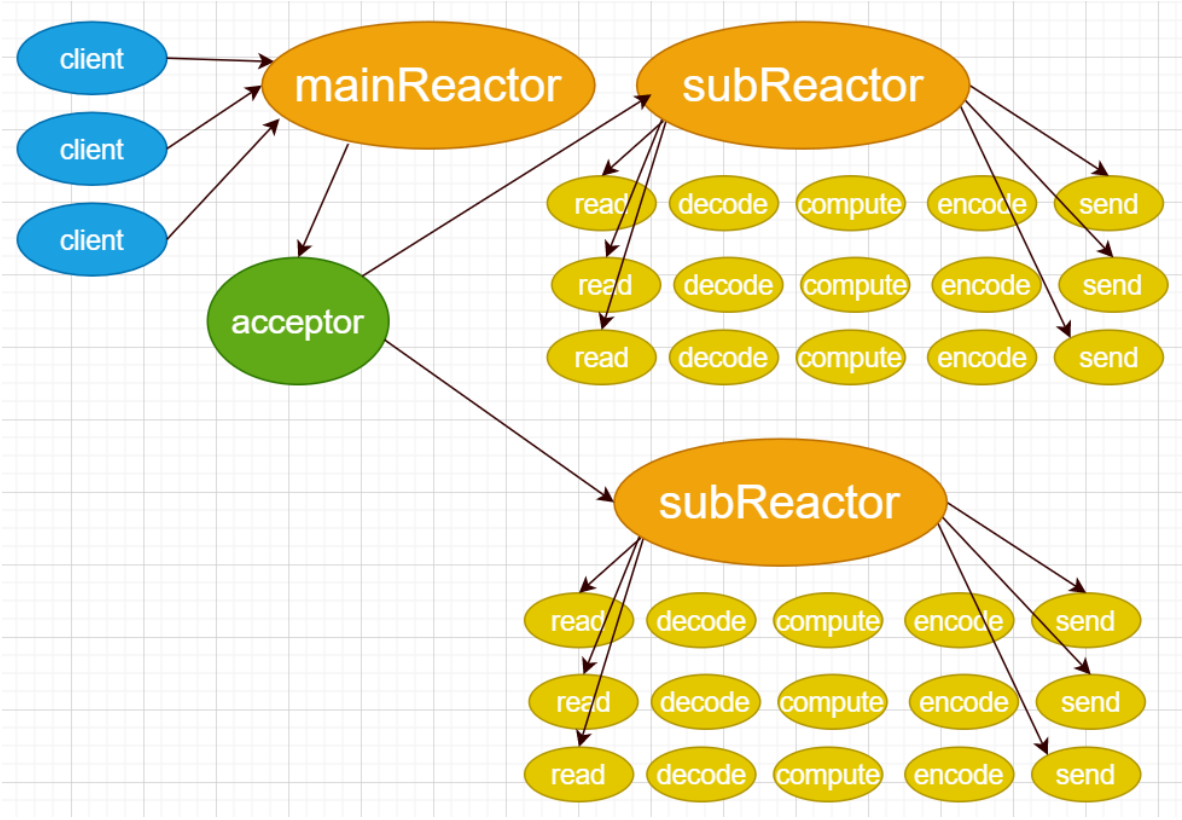


使用线程池带来的好处有如下几点：

- 线程池中的线程是提前创建好的，这样可以在处理多个请求时分摊在线程创建和销毁过程产生的巨大开销。
- 将IO操作与非IO操作分离，当请求到达时工作线程通常已经存在，因此不会由于等待创建线程而延迟任务的执行，从而提高了响应性。
- 可以进行职责分离，让IO线程做IO操作，线程池处理业务逻辑，处理大量计算，充分利用CPU的计算优势。

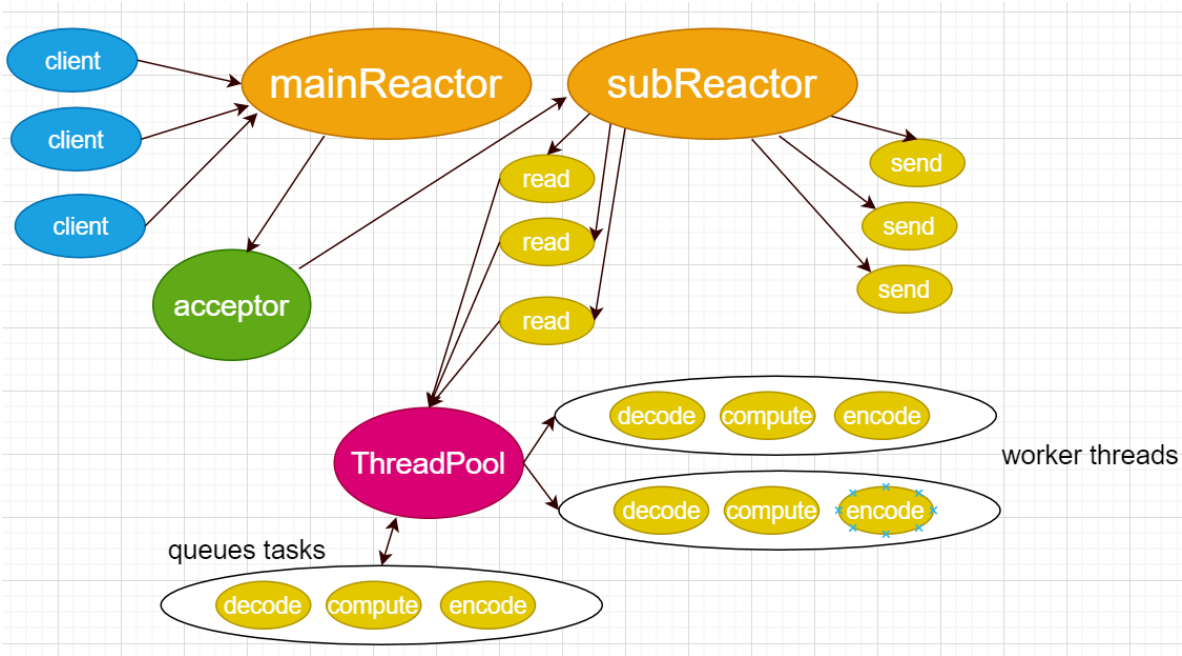
6、多反应式服务器

一个主线程，多个工作线程，主线程Reactor负责接收客户端连接，每个线程有各自的Reactor负责执行任务队列中的任务。



7、多反应式+ 线程池模型

多个Reactor的模式，mainReactor与subReactor都是一个线程，因为多进程之间无法共享计算线程池。这种模型能够适用IO频繁且计算密集的服务。

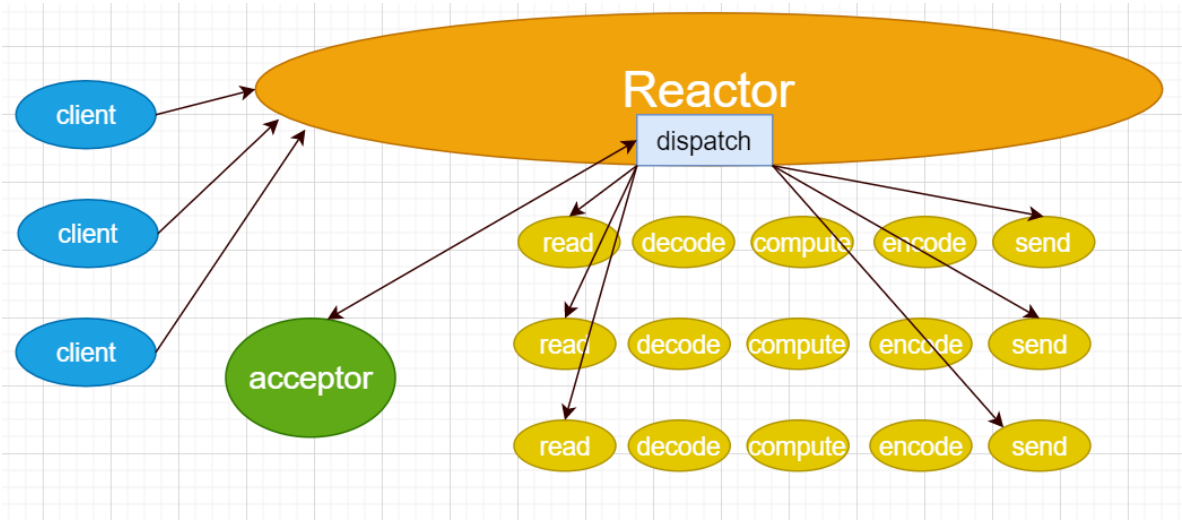


三、Reactor模型

1、Reactor的基本概述

上述的Reactor模型可以演变出很多种类型，但是这里我们主要研究两种Reactor模型，即：基础的Reactor以及Reactor与线程池结合的版本。那么我们先从基本的Reactor来研究，探究其中的原理，然后使用面向对象的设计思想将其实现出来。

先来回顾一下基本的Reactor模型的原理图



从图例中可以看到，多个客户端可以同时向Reactor服务器发起请求，而Reactor服务器是可以同时处理这些请求的。其中，Reactor使用IO多路复用技术监听多个客户端，但是为了能与多个客户端进行连接，所以注册了一个连接器Acceptor对象到Reactor中，进行连接事件的处理，也就是执行accept()函数，然后将连接交给Reactor对象，Reactor对象就对该连接进行对应的处理，读连接的数据，处理连接的数据，然后将处理好之后的数据发送给给个客户端，一条连接处理完毕之后继续处理下一条连接。其实这个过程，就是之前在Linux阶段，使用socket网络编程与IO多路复用（也就是epoll技术）实现的逻辑。接下来我们按照面向对象的思想进行重构。

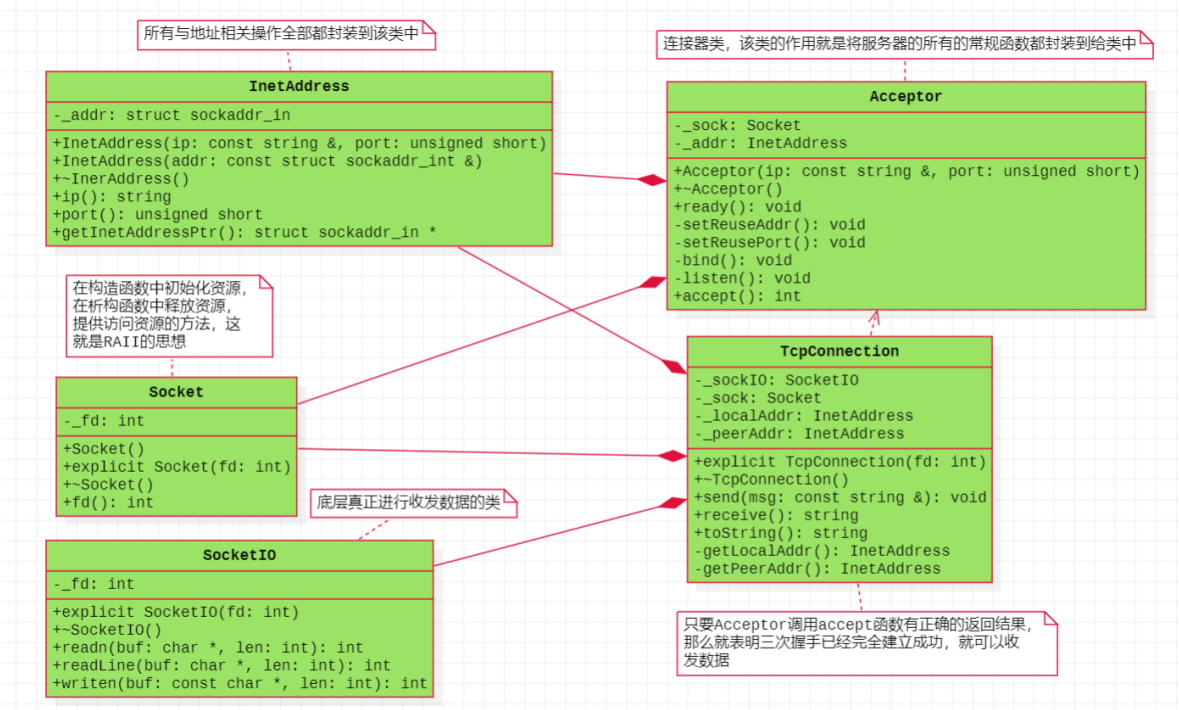
2、ReactorV1版本

2.1、类的设计

为了思维的过渡，我们先将只实现基本的socket编程，然后以此为基础再添加epoll的逻辑。那么在基本的socket编程中，会使用socket创建套接字、bind绑定服务器的ip与port、listen监听客户端的连接、accept接收客户端的连接、然后进行业务逻辑的处理、最后关闭对应的文件描述符。然后使用面向对象的设计，封装了下面五个类：

- Socket类：套接字类，将所有与套接字相关的操作都封装到该类中，包括：套接字的创建、套接字的关闭、套接字的获取。
- InetAddress类：地址类，将所有与地址相关的操作全部封装到该类中，包括：通过ip与端口号创建本对象、获取ip地址、获取port、获取struct sockarr_in的指针等等函数。
- Acceptor类：连接器类，将服务器的所有基本操作全部封装到该类中，包括：地址复用、端口复用、bind、listen、accept等。
- TcpConnection类：TCP连接类，如果Acceptor类的对象调用accept函数有正确的返回结果，就表明三次握手建立成功了，就可以创建出一条通信的TCP连接。以后就可以通过该连接的对象进行数据的收发：发送数据使用send函数、接收数据使用receive函数，具体的数据收发细节不在该类中，而是单独交给另外一个类SocketIO。注意：为了起到调试代码的作用，这边增加对应的获取服务器地址的函数getLocalAddr、获取客户端地址的函数getPeerAddr、以及打印服务器与客户端ip与端口号的函数toString。
- SocketIO类：读写数据类，将真正数据收发的细节全部封装到该类中，例如：每次读任意个字节的数据、每次写任意个字节的数据、每次读一行的数据等。

2.2、类图设计



3、ReactorV2版本

3.1、类的设计

这个版本就是在上一个版本的基础上，添加对epoll的处理，也就是新增了一个事件循环类EventLoop：本质上就是将epoll的三个函数（epoll_create、epoll_ctl、epoll_wait）以及对应的两个文件描述符的可读事件的处理（Socket类创建的listenfd、以及Acceptor中的accept函数的返回的标识连接创建的文件描述符connfd）。对应的数据成员设计如下：

- epoll_create函数的返回结果，需要将其设置为数据成员，因为该文件描述符需要被epoll的另外两个函数进行使用，所以要设置为数据成员`int _epfd`
- epoll_wait函数需要将就绪的文件描述符放在struct epoll_event结构体，也就是填充epoll_wait的第二个参数，后续还需要使用该结构体，也需要将其设置为数据成员，`vector<struct epoll_event> _evtList`
- 如果监听的文件描述符listenfd可读，就说明有新的连接，就需要调用accept函数进行接受，但是该函数被封装到Acceptor类中了，所以需要使用Acceptor类创建的对象，才能获取到accept函数，可以使用Acceptor的对象（对应的应用也可以、对象的指针也可以，目的都是为了获取其中的成员函数accept），这里可以选择Acceptor类型的引用，`Acceptor &_acceptor`；
- 对于每一个accept执行结束后，说明连接已经建立，所以需要有一个对应的连接对象，也就TcpConnection，但是每个对象会关联一个文件描述符connfd，所以有一个记录文件描述符到TcpConnection的数据结构，可以使用键值对进行存储，这样以后就可以通过文件描述符就可以找到对应的连接，同时文件描述符是不重复的，所以这里可以选择的数据结构有map或者unordered_map，我们这里选择map，即map<int, TcpConnection>，但是注意，TcpConnection是不能进行复制或者赋值的，并且创建出来之后还需要在后续使用该连接，所以最好可以设置为堆对象，否则有可能提前销毁了，所以为了更好的管理，可以使用智能指针shared_ptr管理，也就是`map<int, shared_ptr<TcpConnection>> _conns`；
- 为了标识整个事件循环是否运行，可以设置标志位`bool _isLooping`。

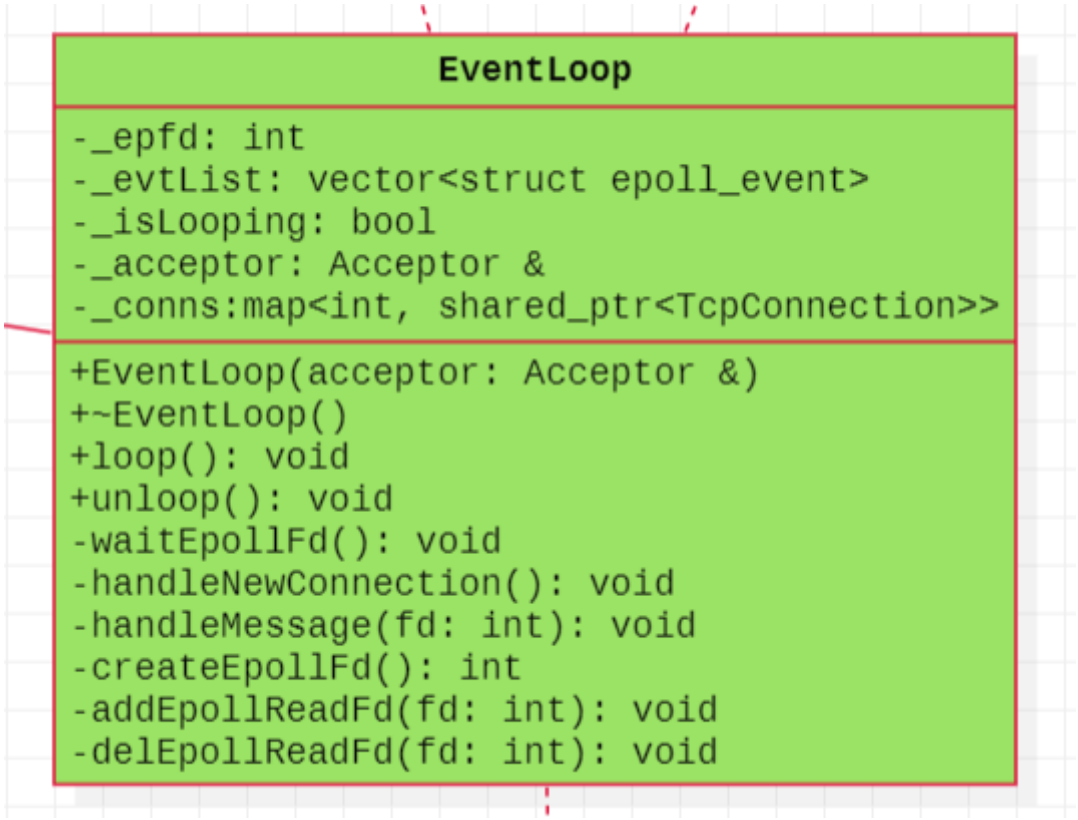
成员函数的设计如下：

- createEpollFd函数：作用就是为了封装epoll的epoll_create函数，创建文件描述符。
- addEpollReadFd函数：作用就是为了封装epoll的epoll_ctl函数，监听对应的文件描述符，例如：listenfd与connfd（这两个文件描述符上述都有描述）
- delEpollReadFd函数：作用也是为了封装epoll的epoll_ctl函数，不过是删除对文件描述符的监听。

- 构造函数：作用就是为了初始化数据成员，这里只有需要对引用数据成员需要额外进行初始化，所以使用EventLoop(Acceptor &acceptor)。
- 循环函数loop：对于IO多路复用epoll而言，核心就是大循环，里面封装epoll_wait函数。
- 非循环函数loop：这里可以让其不循环，直接跳出。
- waitEpollFd函数：里面封装epoll_wait函数，然后就是两大业务逻辑，文件描述符listenfd对应可读的新连接的建立、以及connfd文件描述符可读的读写操作。
- handleNewConnection函数：处理文件描述listenfd的可读事件，调用Acceptor中的accept函数获得文件描述符connfd以及使用文件描述符connfd创建连接TcpConnection的对象，然后监听文件描述符connfd，创建文件描述符connfd与连接TcpConnection的对象键值对，便于通过文件描述符可以找到对应的连接。
- handleMessage函数：通过文件描述符，映射出对应的连接，然后处理读写事件。

这就是对于epoll逻辑的分析而设计出来的数据成员与成员函数。

3.2、简易类图设计



3.3、TCP网络编程的三个半事件

再回顾一下对socket网络编程的认识，里面使用了TCP协议、TCP的三次握手建立连接、TCP的四次挥手断开连接、以及在连接建立好后与客户端进行数据的读写操作，这里面实际上包含着一些事件，也就是：

连接建立：包括服务器端被动接受连接（accept）和客户端主动发起连接（connect）。TCP连接一旦建立，客户端和服务端就是平等的，可以各自收发数据。
连接断开：包括主动断开（close、shutdown）和被动断开（read()返回0）。
消息到达：文件描述符可读。这是最为重要的一个事件，对它的处理方式决定了网络编程的风格（阻塞还是非阻塞，如何处理分包，应用层的缓冲如何设计等等）。
消息发送完毕：这算半个。对于低流量的服务，可不必关心这个事件；另外，这里的“发送完毕”是指数据写入操作系统缓冲区（内核缓冲区），将由TCP协议栈负责数据的发送与重传，不代表对方已经接收到数据。

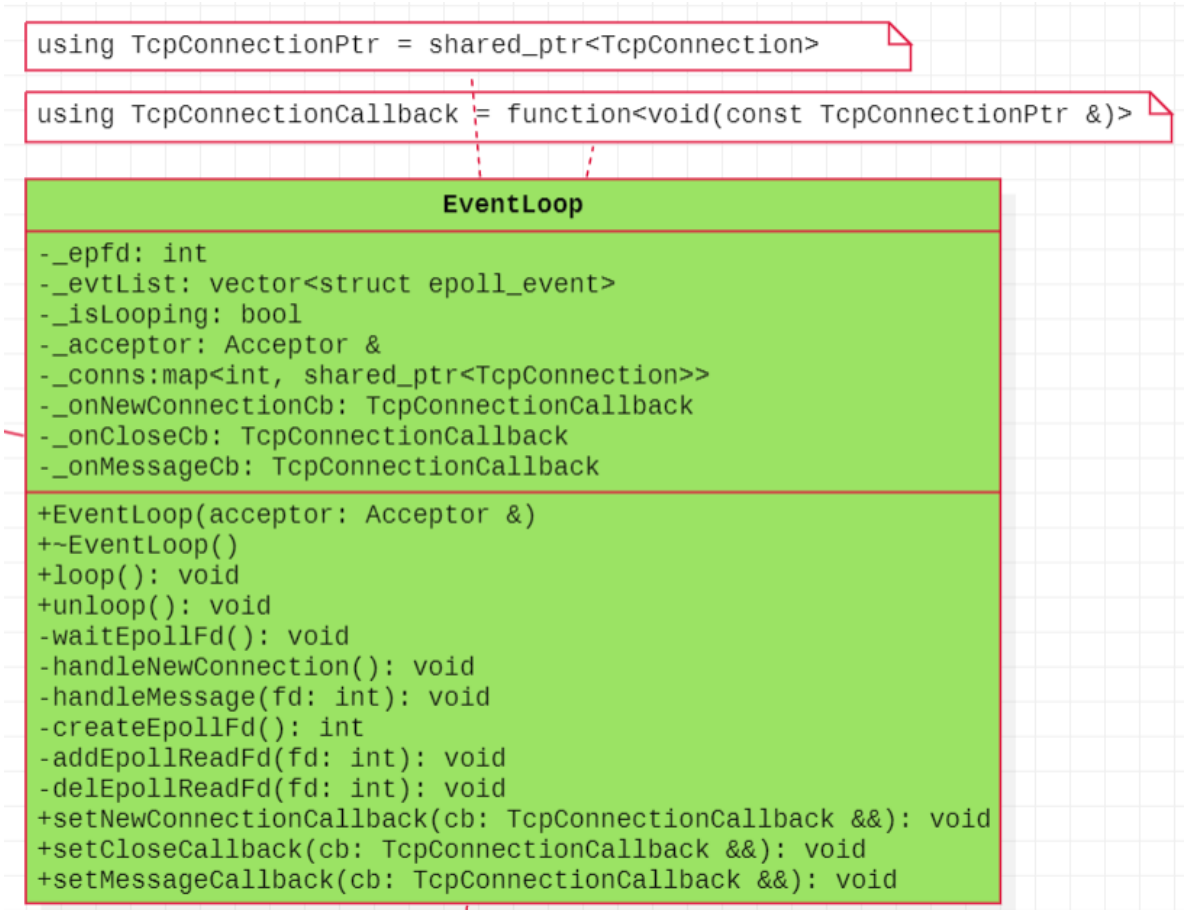
3.4、三个事件的添加

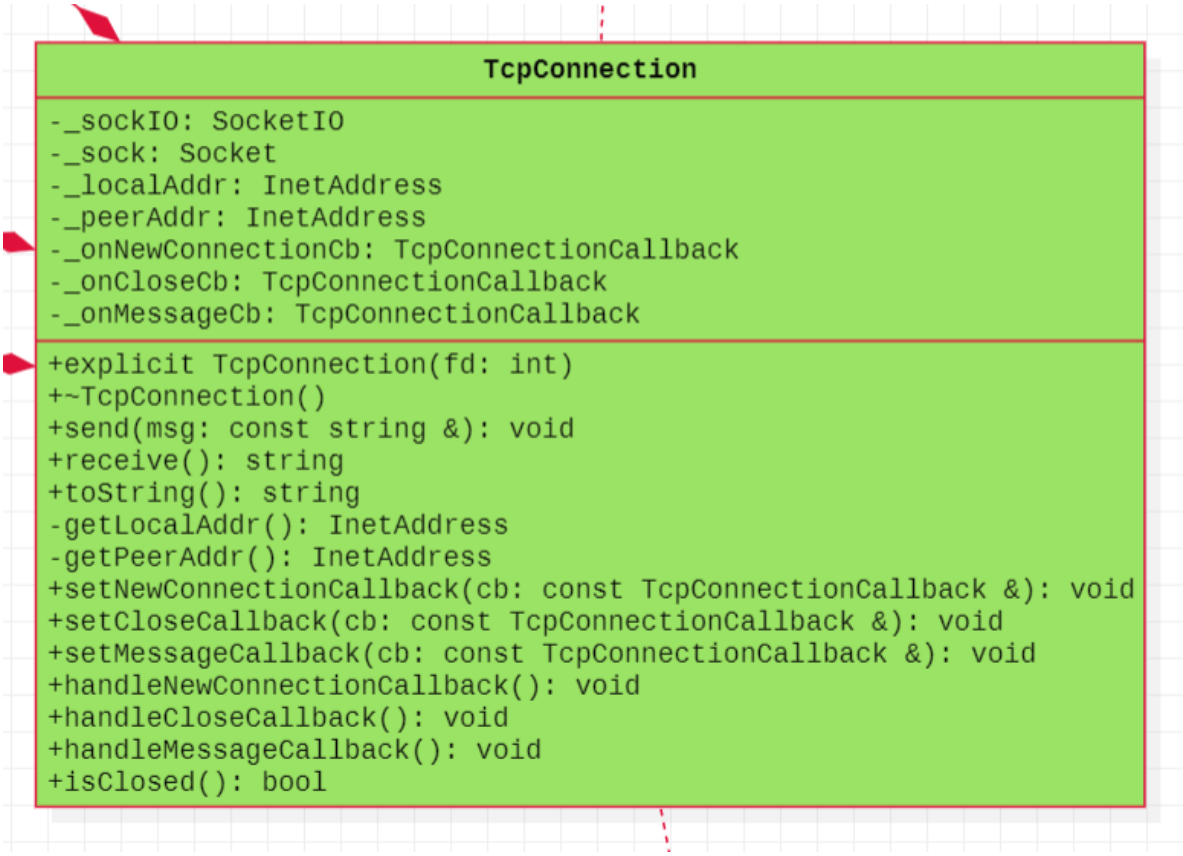
TCP网络编程中，确实存在主要的三个事件：连接的建立、连接的断开、消息到达（文件描述可读），那么这三个事件在发生的时候，我们的服务器到底要做什么，如何做这些事件呢，我们需要做处理。比如：连接建立的时候，是不是可以可以知道服务器自己的信息ip与port、客户端的信息ip与port呢，这不就是连接建立可以做的事件吗，当然连接建立之后还有可能做其他事件，那到底做什么事件我们不得而知，但是可以预先将要做的事件框架注册上来，等满足条件的时候再执行不就可以了吗，这不就是回调函数的思想，**注册回调函数与实现回调函数。**也就是可以使用手段std::function将函数类型设置出来，然后创建对象，最后回调函数。（类似之前的基于对象的实现方式）。三个事件对应三个回调函数，并且三个回调都是与连接TcpConnection相关的。但是连接TcpConnection对象本身都是在EventLoop中创建的，所以需要将三个回调先注册给EventLoop对象作为中转，然后交给TcpConnection对象进行注册与执行（**注意：这个思路非常重要，希望重点思考一下，想清楚这里面的原因**），既然是使用std::function，就需要封装函数类型，但是本次每个函数类型都要与连接TcpConnection对象相关，并且是需要以EventLoop作为中转最好交给TcpConnection对象的，所以类型是`function<void(const shared_ptr<TcpConnection> &)>`，因为在数据成员键值对map中存储的是TcpConnection的智能指针类型，所以后面使用的时候都是TcpConnection的智能指针类型（**注意：这里是代码层面上的使用，使用智能指针类型**）。既然需要先注册给EventLoop，那么就需要在其中有三个数据成员：

```
function<void(const shared_ptr<TcpConnection> &)> _onConnectionCb;
function<void(const shared_ptr<TcpConnection> &)> _onMessageCb;
function<void(const shared_ptr<TcpConnection> &)> _onCloseCb;
```

然后三个成员函数setNewConnectionCallback、setMessageCallback、setCloseCallback，三个函数的参数类型与数据成员的类型一致，但是对于EventLoop而言，只需要注册，不需要执行，最终的执行要交给TcpConnection的对象。那么TcpConnection类中也需要注册三个回调函数，并且好需要执行三个回调函数。**这就是三个回调函数的添加思想，希望大家多思考一下这个过程，有一定的难度。**

于是需要修改类图的设计。





思考题：这里为何在EventLoop中，三个回调函数的注册采用的是右值引用的方式，但是在TcpConnection中使用的是const左值引用的形式呢？在TcpConnection中能不能直接使用右值引用的形式呢？

3.5、核心代码

```
EventLoop::EventLoop(Acceptor &acceptor)
: _epfd(createEpollFd())
, _evtList(1024)
, _isLooping(false)
, _acceptor(acceptor)
{
    //将listenfd放在红黑树上进行监听
    int listenfd = _acceptor.fd();
    addEpollReadFd(listenfd);
}

EventLoop::~EventLoop()
{
    close(_epfd);
}

//事件循环与否
void EventLoop::loop()
{
    _isLooping = true;
    while(_isLooping)
    {
        waitEpollFd();
    }
}

void EventLoop::unloop()
```

```
{
    _isLooping = false;
}

//封装了epoll_wait函数
void EventLoop::waitEpollFd()
{
    int nready = 0;
    do
    {
        nready = epoll_wait(_epfd, &*_evtList.begin(), _evtList.size(),
3000);
    }while((-1 == nready && errno == EINTR));

    if(-1 == nready)
    {
        cerr << "-1 == nready" << endl;
        return;
    }
    else if(0 == nready)
    {
        cout << ">>epoll_wait timeout!!!" << endl;
    }
    else
    {
        //如果监听文件描述的个数超过设置的1024的，不能再进行扩容
        if(nready == (int)_evtList.size())
        {
            _evtList.resize(2 * nready);
        }

        for(int idx = 0; idx < nready; ++idx)
        {
            //连接是listenfd
            int fd = _evtList[idx].data.fd;
            int listenfd = _acceptor.fd();
            if(fd == listenfd)
            {
                if(_evtList[idx].events & EPOLLIN)
                {
                    //处理新的连接
                    handleNewConnection();
                }
            }
            else//处理老的连接
            {
                if(_evtList[idx].events & EPOLLIN)
                {
                    handleMessage(fd);
                }

            }
        }
    }
}
//处理新的连接请求
void EventLoop::handleNewConnection()
{
```

```
85      //如果connfd有正确返回结果，就表明三次握手建立成功，
86      //就可以创建连接
87      int connfd = _acceptor.accept();
88      if(connfd < 0)
89      {
90          perror("handleNewConnection accept");
91          return;
92      }
93
94      //将创建出来的文件描述符放在红黑树上进行监听
95      addEpollReadFd(connfd);
96
97      //创建新的连接
98      /* shared_ptr<TcpConnection> con(new TcpConnection(connfd)); */
99      TcpConnectionPtr con(new TcpConnection(connfd));
100
101     //将三个数据成员（回调函数）传递给连接TcpConnection
102     con->setNewConnectionCallback(_onNewConnectionCb);//连接建立
103     con->setMessageCallback(_onMessageCb);//消息到达
104     con->setCloseCallback(_onCloseCb);//连接断开
105
106     //将键值对存放在map中
107     /* _conns.insert({connfd, con}); */
108     _conns[connfd] = con;
109
110     //连接建立的时机到了，就可以进行回调执行
111     con->handleNewConnectionCallback();
112 }
113
114 //处理老的连接
115 void EventLoop::handleMessage(int fd)
116 {
117     auto it = _conns.find(fd);
118     if(it != _conns.end())
119     {
120         //如何判断连接是不是断开呢
121         bool flag = it->second->isClosed();
122         if(flag)
123         {
124             //连接断开了
125             it->second->handleCloseCallback();//处理连接断开的事件
126             delEpollReadFd(fd);//将文件描述符从红黑树上摘除掉
127             _conns.erase(it);//将文件描述符与连接的键值对从map中删除
128         }
129         else
130         {
131             //消息在正常的收发
132             it->second->handleMessageCallback();//消息到达（文件描述符可读）
133         }
134     }
135     else
136     {
137         cout << "该连接不存在" << endl;
138         return;
139     }
140 }
141
142 //创建epfd的函数
```

```cpp
int EventLoop::createEpollFd()
{
    int fd = epoll_create(10);
    if(fd < 0)
    {
        perror("createEpollFd");
        return fd;
    }

    return fd;
}

//监听文件描述符
void EventLoop::addEpollReadFd(int fd)
{
    struct epoll_event evt;
    evt.events = EPOLLIN;
    evt.data.fd = fd;

    int ret = epoll_ctl(_epfd, EPOLL_CTL_ADD, fd, &evt);
    if(ret < 0)
    {
        perror("addEpollReadFd");
        return;
    }
}

//删除监听文件描述符
void EventLoop::delEpollReadFd(int fd)
{
    struct epoll_event evt;
    evt.events = EPOLLIN;
    evt.data.fd = fd;

    int ret = epoll_ctl(_epfd, EPOLL_CTL_DEL, fd, &evt);
    if(ret < 0)
    {
        perror("delEpollReadFd");
        return;
    }
}

void EventLoop::setNewConnectionCallback(TcpConnectionCallback &&cb)
{
    _onNewConnectionCb = std::move(cb);
}

void EventLoop::setMessageCallback(TcpConnectionCallback &&cb)
{
    _onMessageCb = std::move(cb);
}

void EventLoop::setCloseCallback(TcpConnectionCallback &&cb)
{
    _onCloseCb = std::move(cb);
}
```

在TcpConnection类中新增的函数

```
//三个回调的注册
void TcpConnection::setNewConnectionCallback(const TcpConnectionCallback
&cb)
{
    _onNewConnectionCb = cb;
}

void TcpConnection::setMessageCallback(const TcpConnectionCallback &cb)
{
    _onMessageCb = cb;
}

void TcpConnection::setCloseCallback(const TcpConnectionCallback &cb)
{
    _onCloseCb = cb;
}

//三个回调的执行
void TcpConnection::handleNewConnectionCallback()
{
    if(_onNewConnectionCb)
    {
        /* _onNewConnectionCb(shared_ptr<TcpConnection>(this)); */
        _onNewConnectionCb(shared_from_this());
    }
    else
    {
        cout << "_onNewConnectionCb == nullptr" << endl;
    }
}

void TcpConnection::handleMessageCallback()
{
    if(_onMessageCb)
    {
        _onMessageCb(shared_from_this());
    }
    else
    {
        cout << "_onMessageCb == nullptr" << endl;
    }
}

void TcpConnection::handleCloseCallback()
{
    if(_onCloseCb)
    {
        _onCloseCb(shared_from_this());
    }
    else
    {
        cout << "_onCloseCb == nullptr" << endl;
    }
}

```

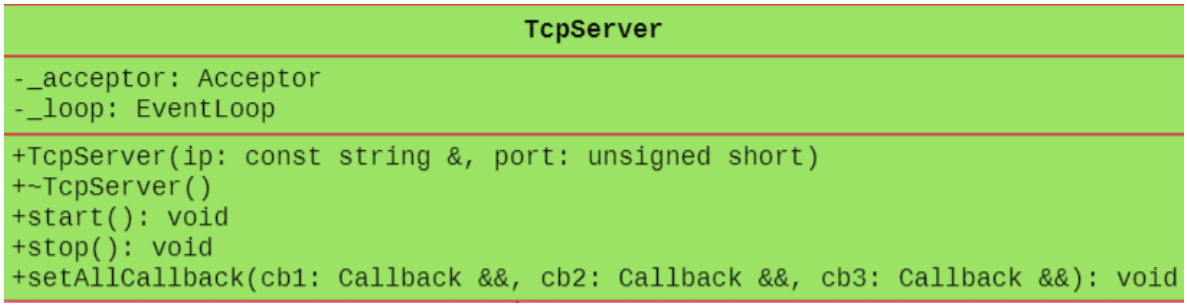
```cpp
bool TcpConnection::isClosed() const
{
    char buff[10] = {0};
    int ret = ::recv(_sock.fd(), buff, sizeof(buff), MSG_PEEK);

    return (0 == ret);
}
```

4、RectorV3版本

本版本其实没有做任何的改变，只是进一步做封装而已。

```cpp
Acceptor _acceptor;
EventLoop _loop;

TcpServer(const string &ip, unsigned short port);//初始化两个数据成员
void start();//执行Acceptor的ready函数,以及EventLoop的loop函数
void stop();//执行EventLoop中的unloop函数
void setAllCallbacks(function<void(const shared_ptr<TcpConnection> &)
&&onConnection,
                     function<void(const shared_ptr<TcpConnection> &)
&&onMessage,
                     function<void(const shared_ptr<TcpConnection> &)
&&onClose);
```

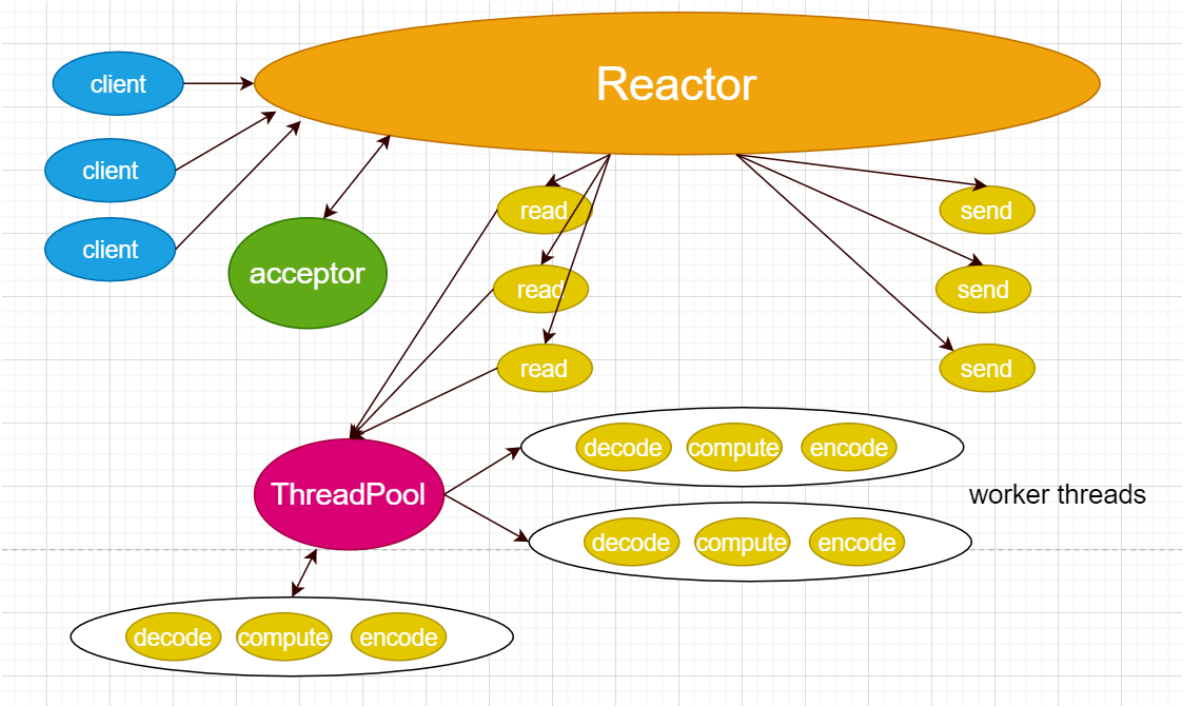
类图设计



到此为止，我们才将基本的Reactor版本进行了实现，但是版本会存在一个问题，IO操作与非IO操作都是一起处理的，那么一旦业务逻辑的处理，decode、compute、encode的处理很耗时，本版本就会存在性能瓶颈，就需要改为线程池版本。

5、ReactorV4版本

5.1、原理图



5.2、原理分析

本模型相对于基本的Reactor模型而言，可以将业务逻辑的处理交给线程池来进行处理，这样能够提高Reactor线程的I/O响应，不至于因为一些耗时的业务逻辑而延迟对后面I/O请求的处理，可以进一步提高性能。但是在本模型中，Reactor在获得连接后，需要采用TcpConnection对象读取数据，也就是执行Read操作（代码中是receive函数），然后将获取到的数据交给线程池的对象进行处理，其实也即是将数据当做任务交给线程池即可，这个比较简单，之前的线程池版本中就实现了添加任务与获取任务，然后让线程池中的线程处理任务，当线程池处理好业务逻辑的处理(decode、compute、encode)后，线程池中的线程（也就是线程池对象）需要通知Reactor对象，并将处理好之后的数据发送给客户端，这里就存在一个问题，线程池对象如何与Reactor线程之间进行通信？通信之后数据如何传递给Reactor？Reactor如何将数据发送给客户端？

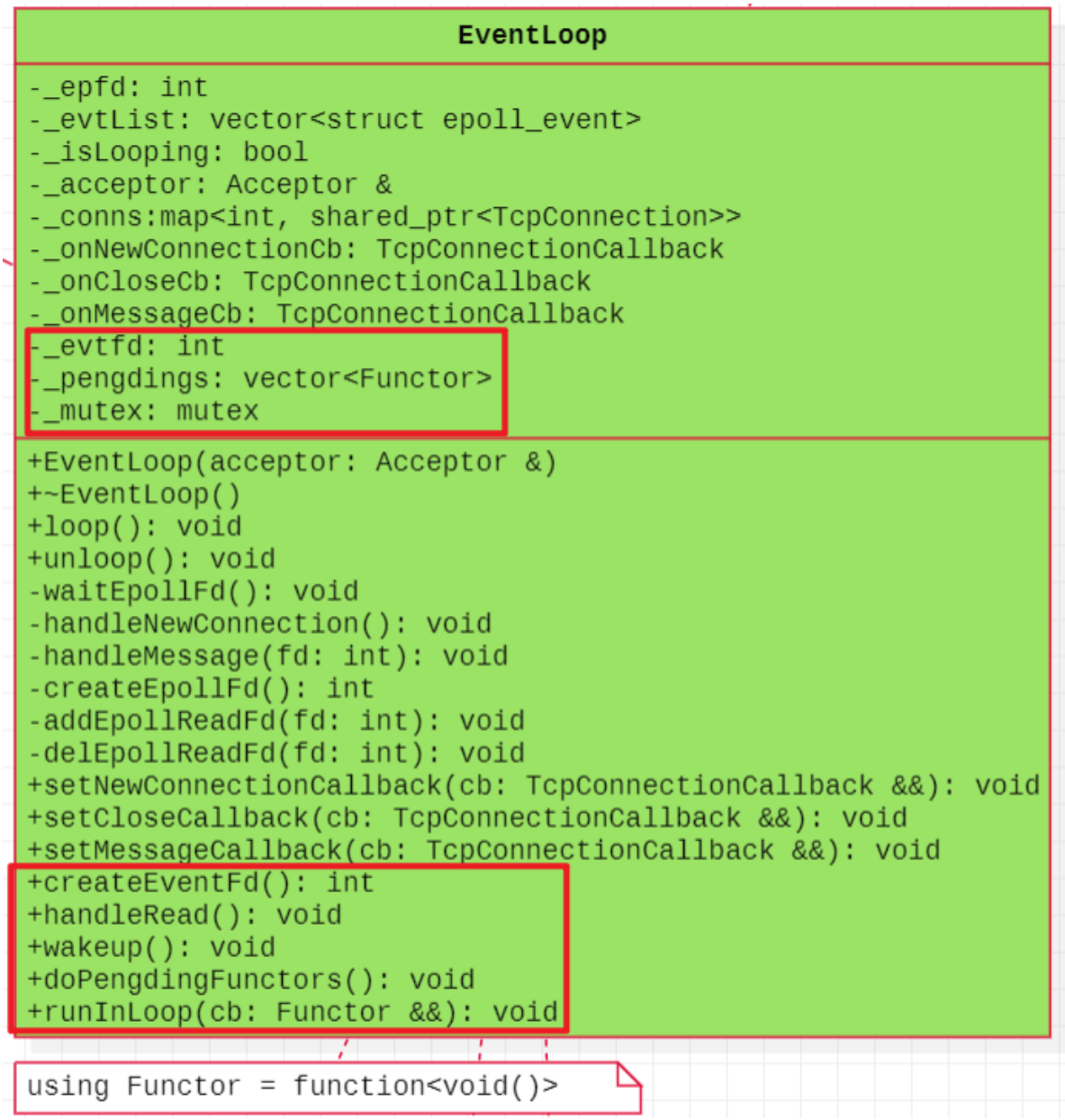
问题一：线程池与Reactor之间通信的方式有哪些？这里可以采用系统调用eventfd的方式（原理如何、怎么使用后面会讲解），学会了就可以解决本版本的问题,可以跳转到eventfd，学习其原理学习以及使用方式。

问题二：数据如何传递给Reactor？这里面会将数据通过相应的函数传递给Reactor，这个与具体的代码设计相关，因为在第三个版本中，线程池中会有TcpConnection的对象，而TcpConnection对象都是在EventLoop中创建的，所以很好传递到Reactor/EventLoop中。

问题三：Reactor如何将数据发送给客户端？很简单，借助TcpConnection对象的send函数发送即可。

5.3、类图设计

将用于通信的eventfd系统调用（也就是线程之间通信的代码）封装到ReactorV3版本的EventLoop类中，然后只要线程池中的线程在执行完业务逻辑的处理之后，调用wakeup唤醒Reactor线程（**注意：名为唤醒，其实Reactor线程并不会真正的阻塞，而是负责监听用于通信的文件描述符即可，只要就绪，就可以处理发送任务给客户端了，如果真的Reactor在睡眠，那么新的连接上来了，线程不就阻塞住了吗**），也就是通知Reactor线程即可，然后执行发送操作，将数据发送给客户端即可。



数据成员：

- 就是将线程之间进行通信的文件描述符`int _evtfd`
- 线程池处理完之后要通过Reactor发送给客户端的**任务**比较多，所以就用容器存起来`vector<Functor> _pengdings`
- 多个线程对容器vector的访问是互斥的，需要加锁`mutex _mutex`

成员函数：

- createEventFd函数：调用eventfd系统调用创建用于通信的文件描述符。
- handleRead函数：里面封装了read函数，该函数读取eventfd返回的文件描述符。
- wakeup函数：里面封装了write函数，该函数向eventfd返回的文件描述符中写数据，也就是唤醒阻塞的线程，从而达到通信的目的。
- doPengdingFunctors函数：就是将存放在vector中的**任务**进行执行，这里的**任务**比较多，所以需要遍历。
- runInLoop函数：vector中的任务是如何传递进来的？本函数就是将线程池处理好之后的要发送给客户端的数据，最终传递到vector中的，主要是做这个事情。

思考题：vector存放的任务，这里的任务到底是什么，难道只是线程池处理好要发送给客户端的数据吗?

5.4、核心函数

```cpp
EventLoop::EventLoop(Acceptor &acceptor)
: _epfd(createEpollFd())
, _evtList(1024)
, _isLooping(false)
, _acceptor(acceptor)
, _evtfd(createEventFd())
, _mutex()
{
    //将listenfd放在红黑树上进行监听
    int listenfd = _acceptor.fd();
    addEpollReadFd(listenfd);
    //将用于通信的文件描述符进行监听
    addEpollReadFd(_evtfd);
}

EventLoop::~EventLoop()
{
    close(_epfd);
    close(_evtfd);
}

//事件循环与否
void EventLoop::loop()
{
    _isLooping = true;
    while(_isLooping)
    {
        waitEpollFd();
    }
}

void EventLoop::unloop()
{
    _isLooping = false;
}

//封装了epoll_wait函数
void EventLoop::waitEpollFd()
{
    int nready = 0;
    do
    {
        nready = epoll_wait(_epfd, &*_evtList.begin(), _evtList.size(),
3000);
    }while((-1 == nready && errno == EINTR));

    if(-1 == nready)
    {
        cerr << "-1 == nready" << endl;
        return;
    }
    else if(0 == nready)
    {
        cout << ">>epoll_wait timeout!!!" << endl;
    }
    else
    {
```

```
        //如果监听文件描述的个数超过设置的1024的，不能再进行扩容
        if(nready == (int)_evtList.size())
        {
            _evtList.resize(2 * nready);
        }

        for(int idx = 0; idx < nready; ++idx)
        {
            //连接是listenfd
            int fd = _evtList[idx].data.fd;
            int listenfd = _acceptor.fd();
            if(fd == listenfd)
            {
                if(_evtList[idx].events & EPOLLIN)
                {
                    //处理新的连接
                    handleNewConnection();
                }
            }
            //监听的用于通信的文件描述符就绪了
            else if(fd == _evtfd)
            {
                if(_evtList[idx].events & EPOLLIN)
                {
                    handleRead();
                    //执行所有的"任务"
                    doPengdingFunctors();
                }
            }
            else//处理老的连接
            {
                if(_evtList[idx].events & EPOLLIN)
                {
                    handleMessage(fd);
                }

            }
        }
    }
}
//处理新的连接请求
void EventLoop::handleNewConnection()
{
    //如果connfd有正确返回结果，就表明三次握手建立成功，
    //就可以创建连接
    int connfd = _acceptor.accept();
    if(connfd < 0)
    {
        perror("handleNewConnection accept");
        return;
    }

    //将创建出来的文件描述符放在红黑树上进行监听
    addEpollReadFd(connfd);

    //创建新的连接
    /* shared_ptr<TcpConnection> con(new TcpConnection(connfd)); */
    TcpConnectionPtr con(new TcpConnection(connfd, this));
```

```cpp

    //将三个数据成员（回调函数）传递给连接TcpConnection
    con->setNewConnectionCallback(_onNewConnectionCb);//连接建立
    con->setMessageCallback(_onMessageCb);//消息到达
    con->setCloseCallback(_onCloseCb);//连接断开

    //将键值对存放在map中
    /* _conns.insert({connfd, con}); */
    _conns[connfd] = con;

    //连接建立的时机到了，就可以进行回调执行
    con->handleNewConnectionCallback();
}

//处理老的连接
void EventLoop::handleMessage(int fd)
{
    auto it = _conns.find(fd);
    if(it != _conns.end())
    {
        //如何判断连接是不是断开呢
        bool flag = it->second->isClosed();
        if(flag)
        {
            //连接断开了
            it->second->handleCloseCallback();//处理连接断开的事件
            delEpollReadFd(fd);//将文件描述符从红黑树上摘除掉
            _conns.erase(it);//将文件描述符与连接的键值对从map中删除
        }
        else
        {
            //消息在正常的收发
            it->second->handleMessageCallback();//消息到达（文件描述符可读）
        }
    }
    else
    {
        cout << "该连接不存在" << endl;
        return;
    }
}

//创建epfd的函数
int EventLoop::createEpollFd()
{
    int fd = epoll_create(10);
    if(fd < 0)
    {
        perror("createEpollFd");
        return fd;
    }

    return fd;
}

//监听文件描述符
void EventLoop::addEpollReadFd(int fd)
{
```

```cpp
    struct epoll_event evt;
    evt.events = EPOLLIN;
    evt.data.fd = fd;

    int ret = epoll_ctl(_epfd, EPOLL_CTL_ADD, fd, &evt);
    if(ret < 0)
    {
        perror("addEpollReadFd");
        return;
    }
}

//删除监听文件描述符
void EventLoop::delEpollReadFd(int fd)
{
    struct epoll_event evt;
    evt.events = EPOLLIN;
    evt.data.fd = fd;

    int ret = epoll_ctl(_epfd, EPOLL_CTL_DEL, fd, &evt);
    if(ret < 0)
    {
        perror("delEpollReadFd");
        return;
    }
}

void EventLoop::setNewConnectionCallback(TcpConnectionCallback &&cb)
{
    _onNewConnectionCb = std::move(cb);
}

void EventLoop::setMessageCallback(TcpConnectionCallback &&cb)
{
    _onMessageCb = std::move(cb);
}

void EventLoop::setCloseCallback(TcpConnectionCallback &&cb)
{
    _onCloseCb = std::move(cb);
}

int EventLoop::createEventFd()
{
    int fd = eventfd(10, 0);
    if(fd < 0)
    {
        perror("eventfd");
        return fd;
    }

    return fd;
}

void EventLoop::handleRead()
{
    uint64_t one = 1;
    ssize_t ret = read(_evtfd, &one, sizeof(uint64_t));
```

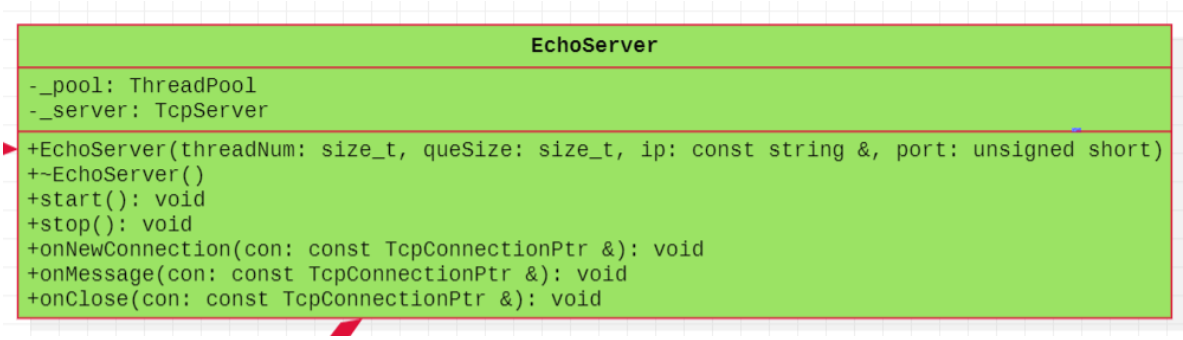
```
    if(ret != sizeof(uint64_t))
    {
        perror("read");
        return;
    }
}

void EventLoop::wakeup()
{
    uint64_t one = 1;
    ssize_t ret = write(_evtfd, &one, sizeof(uint64_t));
    if(ret != sizeof(uint64_t))
    {
        perror("wakeup");
        return;
    }
}

void EventLoop::doPengdingFunctors()
{
    vector<Functor> tmp;
    {
        lock_guard<mutex> lg(_mutex);
        tmp.swap(_pengdings);
    }

    //将所有的任务都进行执行
    for(auto &cb : tmp)
    {
        cb();//回调的执行
    }
}

void EventLoop::runInLoop(Functor &&cb)
{
    {
        lock_guard<mutex> lg(_mutex);
        _pengdings.push_back(std::move(cb));
    }

    //线程池就需要通知EventLoop执行“任务”
    wakeup();
}
```

6、ReactorV5版本

本版本只是在上一个版本的基础上，进一步的做封装，将线程池与TcpServer对象进一步封装。

```
ThreadPool _pool;
TcpServer _server;

EchoServer(size_t threadNum, size_t queSize, const string &ip, unsigned short port);
~EchoServer();

```

```
//让服务器启动与停止
void start();
void stop();

//三个回调函数
void onNewConnection(const TcpConnectionPtr &con);
void onMessage(const TcpConnectionPtr &con);
void onClose(const TcpConnectionPtr &con);
```



7、进线程通信方式eventfd

7.1、作用

从Linux 2.6.27版本开始，新增了不少系统调用，其中包括eventfd，它的主要是用于进程或者线程间通信(如通知/等待机制的实现)

7.2、函数接口

```
#include <sys/eventfd.h>

int eventfd(unsigned int initval, int flags);

initval：初始化计数器值，该值保存在内核。
flags:如果是2.6.26或之前版本的内核，flags 必须设置为0。
flags支持以下标志位：
EFD_NONBLOCK 类似于使用O_NONBLOCK标志设置文件描述符。
EFD_CLOEXEC  类似open以O_CLOEXEC标志打开， O_CLOEXEC 应该表示执行exec()时，之前通过open()打开的文件描述符会自动关闭。

返回值：函数返回一个文件描述符，与打开的其他文件一样，可以进行读写操作。
```

7.3、eventfd支持的操作

eventfd系统调用返回的是文件描述符，该文件描述符与以前学习的文件描述符一样，可以读、写、监听。

read函数：如果计数器A的值不为0时，读取成功，获得到该值；如果A的值为0，非阻塞模式时，会直接返回失败，并把error置为EINVAL；如果为阻塞模式，一直会阻塞到A为非0为止。
write函数：将缓冲区写入的8字节整形值加到内核计数器上，即会增加8字节的整数在计数器A上，如果其值达到0xfffffffffffffffe时，就会阻塞（在阻塞模式下），直到A的值被read。
select/poll/epoll：支持被io多路复用监听。当内核计数器的值发生变化时，就会触发事件。

通过对eventfd函数返回的文件描述符进行通信。一个进程或者线程A执行read操作，如果内核计数器的值为0，并且是阻塞模式，那么A就会阻塞；另外一个进程或者线程B执行write操作，就会向内核计数器写，那么阻塞的A发现内核计数器的值不为0，就会被触发，那么两个进程或者线程A与B就达到通信的目的了。

7.4、进程之间通信

在man手册中，是存在进程之间通信的方式的，代码如下

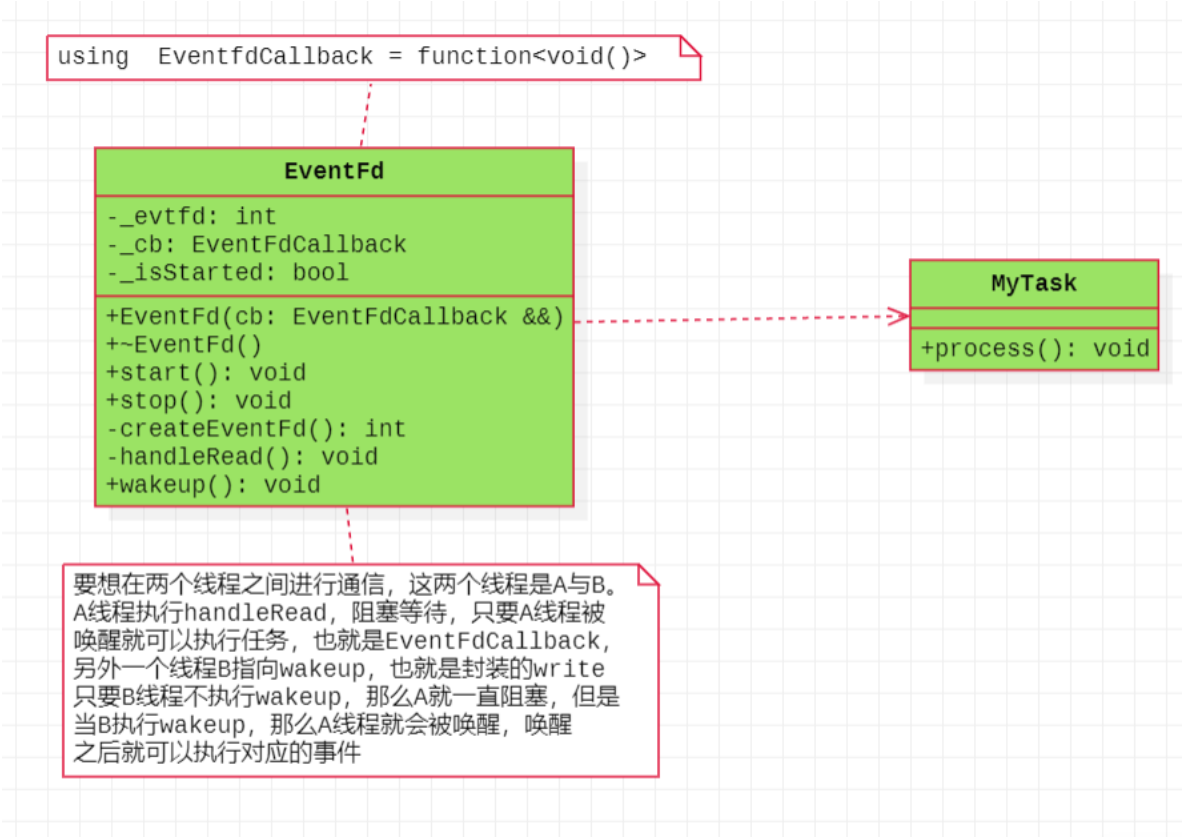
```
#include <sys/eventfd.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>    /* Definition of uint64_t */

#define handle_error(msg) \
    do { perror(msg); exit(EXIT_FAILURE); } while (0)

int main(int argc, char *argv[])
{
    int efd, j;
    uint64_t u;
    ssize_t s;

    if (argc < 2)
    {
        fprintf(stderr, "Usage: %s <num>...\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    efd = eventfd(10, 0);//eventfd的第一个参数代表的是内核上的计数器
    if (efd == -1)
    {
        handle_error("eventfd");
    }

    switch (fork())
    {
    case 0:
        //case 0部分是子线程的执行流
        for (j = 1; j < argc; j++)
        {
            //打印命令行参数的值
            printf("Child writing %s to efd\n", argv[j]);
            //将命令行参数从字符串转换为整型
            u = strtoull(argv[j], NULL, 0);/* strtoull() allows various
bases */

            //write可以写多次，每执行一次，就会执行一次加法
            s = write(efd, &u, sizeof(uint64_t));
            if (s != sizeof(uint64_t))
            {
                handle_error("write");
            }

            sleep(1);
        }
```

```
        printf("Child completed write loop\n");

        exit(EXIT_SUCCESS);

    default:
        //父线程的一个执行流
        sleep(2);

        for(int idx = 2; idx < argc; ++idx)
        {
            printf("Parent about to read\n");
            //read操作会将计数器的值清0
            s = read(efd, &u, sizeof(uint64_t));
            if (s != sizeof(uint64_t))
            {
                handle_error("read");
            }

            //将从内核计数器读到的值以不同进制的形式打印出来
            printf("Parent read %llu (0x%llx) from efd\n",
                    (unsigned long long) u, (unsigned long long) u);
            sleep(1);
        }
        exit(EXIT_SUCCESS);

    case -1:
        handle_error("fork");
    }
}
```

父进程通过eventfd返回的文件描述进行**读数据**，子进程通过eventfd返回的文件描述符进行**写数据**，如果父进程读到的内核计数器的值为0，就会阻塞，但是只要子进程写数据到内核计数器，那么父进程就会被唤醒达到通信的目的（这里父进程读到了内核计数器维护的值）

7.5、线程之间通信

与进程之间通信模式类似，我们可以在线程之间进行通信，一个线程A读取eventfd返回的文件描述符，如果读取到的内核计数器的值为0，那么就会阻塞；而另外一个线程B向eventfd返回的文件描述符进行写数据，就会唤醒A线程，从而达到B唤醒A，达到B线程（写线程）唤醒（通知）A线程的目的，这样两个线程之间就可以达到通信的目的，就是这个原理。在进程之间通信之后，父进程在读内核计数器的值，这里可以让A线程阻塞等待执行某种任务，只要不被B唤醒，A就一直阻塞，只有被唤醒就可以执行任务。那么使用面向对象封装，可以进行类图设计如下：



数据成员：

- 用于通信的文件描述符，也就是eventfd返回的文件描述符 `int _evtfd`
- 被唤醒的线程需要执行的任务 `EventFdCallback _cb`
- 标识EventFd运行标志的标志位 `bool _isStarted`

成员函数：

- start函数：该函数启动，并通过IO多路复用方式select/poll/epoll中的一种循环监视数据成员，用于通信的文件描述符_evtfd是不是就绪，如果就绪就可以让线程读该文件描述符并且执行被唤醒后需要执行的事件，也即是EventFdCallback类型的任务。
- stop函数：停止运行。
- handleRead函数：里面封装了read函数，该函数读取eventfd返回的文件描述符。
- wakeup函数：里面封装了write函数，该函数向eventfd返回的文件描述符中写数据，也就是唤醒阻塞的线程，从而达到通信的目的

核心函数：

```
EventFd::EventFd(EventFdCallback &&cb)
: _evtfd(createEventFd())
, _cb(std::move(cb))//注册
, _isStarted(false)
{
}

EventFd::~EventFd()
{
    close(_evtfd);
}

//运行与停止
void EventFd::start()
{
    struct pollfd pfd;
```

```
    pfd.fd = _evtfd;
    pfd.events = POLLIN;

    _isStarted = true;
    while(_isStarted)
    {
        int nready = poll(&pfd, 1, 3000);
        if(-1 == nready &&  errno == EINTR)
        {
            continue;
        }
        else if(-1 == nready)
        {
            cerr << "-1 == nready" << endl;
            return;
        }
        else if(0 == nready)
        {
            cout << ">>poll timeout!!!" << endl;
        }
        else
        {
            if(pfd.revents & POLLIN)
            {
                handleRead();//阻塞等待被唤醒
                if(_cb)
                {
                    _cb();//通信之后需要执行的任务
                }

            }
        }
    }
}

void EventFd::stop()
{
    _isStarted = false;
}

//创建用于通信的文件描述符
int EventFd::createEventFd()
{
    int ret = eventfd(10, 0);
    if(ret < 0)
    {
        perror("eventfd");
        return ret;
    }

    return ret;
}

//A线程需要执行的read的操作
void EventFd::handleRead()
{
    uint64_t one = 1;
    ssize_t ret = read(_evtfd, &one, sizeof(uint64_t));
```

```
    if(ret != sizeof(uint64_t))
    {
        perror("read");
        return;
    }
}
//用于唤醒线程的函数
void EventFd::wakeup()
{
    uint64_t one = 1;
    ssize_t ret = write(_evtfd, &one, sizeof(uint64_t));
    if(ret != sizeof(uint64_t))
    {
        perror("write");
        return;
    }
}
```

8、定时器

8.1、作用

timerfd是Linux提供的一个定时器接口。这个接口基于文件描述符，**通过文件描述符的可读事件进行超时通知**，所以能够被用于select/poll/epoll的应用场景。timerfd是linux内核2.6.25版本中加入的接口

8.2、函数接口

```
#include <sys/timerfd.h>
int timerfd_create(int clockid, int flags);
参数详解：
clockid:可设置为
CLOCK_REALTIME：相对时间，从1970.1.1到目前的时间。更改系统时间 会更改获取的值，它以系统时间为标。
CLOCK_MONOTONIC：绝对时间，获取的时间为系统重启到现在的时间，更改系统时间对齐没有影响。
flags： 可设置为
TFD_NONBLOCK（非阻塞）;TFD_CLOEXEC（同O_CLOEXEC）linux内核2.6.26版本以上都指定为0

返回值：该函数生成一个定时器对象，返回与之关联的文件描述符。
```

```
int timerfd_settime(int fd, int flags,
                    const struct itimerspec *new_value,
                    struct itimerspec *old_value);
参数详解：
fd: timerfd对应的文件描述符
flags: 0表示是相对定时器;TFD_TIMER_ABSTIME表示是绝对定时器
new_value:设置超时时间，如果为0则表示停止定时器。
old_value:一般设为NULL，不为NULL,则返回定时器这次设置之前的超时时间。

struct timespec
{
    time_t tv_sec;  //精确到秒数
    long   tv_nsec; //精确到纳秒数
};

struct itimerspec
{
```

```
    struct timespec it_interval;  //定时器周期时间，前后两次超时时间差
    struct timespec it_value; //定时器起始时间，比如：12:00:00开始，或者相对某个时间点开始技时
};

返回值：该函数能够启动和停止定时器。
```

8.3、支持的操作

定时器对象（也就是定时器创建出来的文件描述符），是可以被读以及监听的。

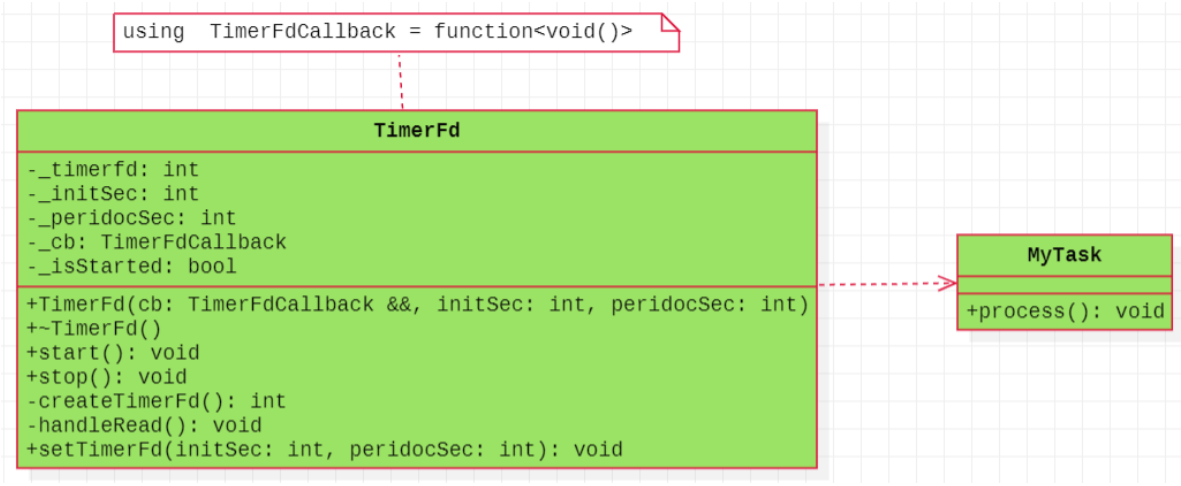
read函数：读取缓冲区中的数据，其占据的存储空间为sizeof(uint64_t)，表示超时次数。

select/poll/epoll：当定时器超时时，会触发定时器相对应的文件描述符上的读操作，IO多路复用操作会返回，然后再去对该读事件进行处理。

其实就是定时器对象在设置好定时操作后，当设置的超时时间到达后，定时器对象就就绪，就可以被读取了，然后当设置的超时时间到达后，定时器对象就就绪，就可以又被读取了，以此往复。

8.4、线程间通信

设置好定时器对象（就像设置的闹钟一样），当定时器超时后，会发出超时通知，如果线程之前在循环监视对应的文件描述符，那么文件描述符就会就绪（可读），就可以执行read函数，接下来就可以执行预先设置好的任务。当定时器后续继续超时后，监听的文件描述符会继续就绪，文件描述符继续可读，就可以继续执行任务了，所以可见，每间隔指定的时间，线程都会因为超时而被唤醒，也就达到通知的目的。那么使用面向对象封装，可以进行类图设计如下：



数据成员：

- 用于超时通知的文件描述符，也就是timerfd_create创建的文件描述符`int _timerfd`
- 定时器初始时间与超时时间`int _initSec  int _peridocSec`
- 被唤醒的线程需要执行的任务`TimerFdCallback _cb`
- 标识EventFd运行标志的标志位`bool _isStarted`

成员函数：

- start函数：该函数启动，并通过IO多路复用方式select/poll/epoll中的一种循环监视数据成员，用于超时通知的文件描述符_timerffd是不是就绪，如果就绪就可以让线程读该文件描述符并且执行被唤醒后需要执行的事件，也即是TimerFdCallback类型的任务。
- stop函数：停止运行。
- handleRead函数：里面封装了read函数，该函数读取timerfd_create返回的文件描述符。
- setTimerFd函数：里面封装了timerfd_settime函数，用于设定定时器的起始时间与超时时间，可以是启动定时器或者关闭定时器（起始时间与超时时间都为0）

核心函数：

```
TimerFd::TimerFd(TimerFdCallback &&cb, int initSec, int peridocSec)
: _tfd(createTimerFd())
, _initSec(initSec)
, _peridocSec(peridocSec)
, _cb(std::move(cb))//注册
, _isStarted(false)
{
}

TimerFd::~TimerFd()
{
    setTimerFd(0, 0);
    close(_tfd);
}

//运行与停止
void TimerFd::start()
{
    struct pollfd pfd;
    pfd.fd = _tfd;
    pfd.events = POLLIN;

    //设定了定时器
    setTimerFd(_initSec, _peridocSec);

    _isStarted = true;
    while(_isStarted)
    {
        int nready = poll(&pfd, 1, 3000);
        if(-1 == nready &&  errno == EINTR)
        {
            continue;
        }
        else if(-1 == nready)
        {
            cerr << "-1 == nready" << endl;
            return;
        }
        else if(0 == nready)
        {
            cout << ">>poll timeout!!!" << endl;
        }
        else
        {
            if(pfd.revents & POLLIN)
            {
                handleRead();//阻塞等待被唤醒
                if(_cb)
                {
                    _cb();//通信之后需要执行的任务
                }

            }
        }
    }
```

```cpp
}

void TimerFd::stop()
{
    if(_isStarted)
    {
        _isStarted = false;
        setTimerFd(0, 0);
    }
}

//创建用于通信的文件描述符
int TimerFd::createTimerFd()
{
    int ret = timerfd_create(CLOCK_REALTIME, 0);
    if(ret < 0)
    {
        perror("createTimerFd");
        return ret;
    }

    return ret;
}

//A线程需要执行的read的操作
void TimerFd::handleRead()
{
    uint64_t one = 1;
    ssize_t ret = read(_tfd, &one, sizeof(uint64_t));
    if(ret != sizeof(uint64_t))
    {
        perror("read");
        return;
    }
}

void TimerFd::setTimerFd(int initSec, int peridocSec)
{
    struct itimerspec newValue;
    newValue.it_value.tv_sec = initSec;//起始的秒数
    newValue.it_value.tv_nsec = 0;

    newValue.it_interval.tv_sec = peridocSec;//周期时间
    newValue.it_interval.tv_nsec = 0;

    int ret = timerfd_settime(_tfd, 0, &newValue, nullptr);
    if(ret)
    {
        perror("timerfd_settime");
        return;
    }
}
```