简述

服务器程序可以简单归纳为「收到数据，算一算，再发出去」，网络编程是不可或缺的部分。

> *甚至同一台物理机内进程间通信通过网络 IO 也是最方便的。*

```
// socket 创建
listen_fd = socket(AF_INET,SOCK_STREAM,0)
// 绑定
bind(listen_fd,(struct sockaddr *)&listen_addr,sizeof(struct sockaddr)
// 监听
listen(listen_fd,SERVER_BACKLOG)
while (true) {
    // 阻塞等待
    client_fd = accept(listen_fd,(struct sockaddr *)&remote_addr,&sin_size)
    // 接受data
    recv(client_fd,buff,BUFFER_SIZE,0)
    // 发送data
    send(client_fd,buff,n,0)
}
```

```
client_fd = socket(AF_INET,SOCK_STREAM,0)
// 主动连接
connect(client_fd,(struct sockaddr *)&remote_addr,sizeof(remote_addr)
// 发送数据
send(client_fd,data,strlen(data),0)
// 接收数据
recv(client_fd,buff,BUFSIZE,0)
```

在没有线程概念的时代，计算机处理 IO 并发是通过 fork 进程来处理请求，one connection one processor。
在操作系统引入线程之后，最简单的并发处理是为每一个链接创建一个线程。

(0) 迭代服务器（无进程控制，用作测量基准）；
(1) 并发服务器，每个客户请求fork一个子进程；
(2) 预先派生子进程，每个子进程无保护地调用accept；
(3) 预先派生子进程，使用文件上锁保护accept；
(4) 预先派生子进程，使用线程互斥锁上锁保护accept；
(5) 预先派生子进程，父进程向子进程传递套接字描述符；
(6) 并发服务器，每个客户请求创建一个线程；
(7) 预先创建线程服务器，使用互斥锁上锁保护accept；
(8) 预先创建线程服务器，由主线程调用accept。

「UNIX网络编程卷1 第30章 客户/服务器程序设计范式」

non-blocking IO + IO multiplexing

用线程并发处理连接很容易达到瓶颈，高并发下线程调度耗时 「C10K问题」。

操作系统带来的IO多路复用技术，poll/epll机制，配合非阻塞 read/write，用一个IO loop 线程就能轻轻松松管理百万级链接。

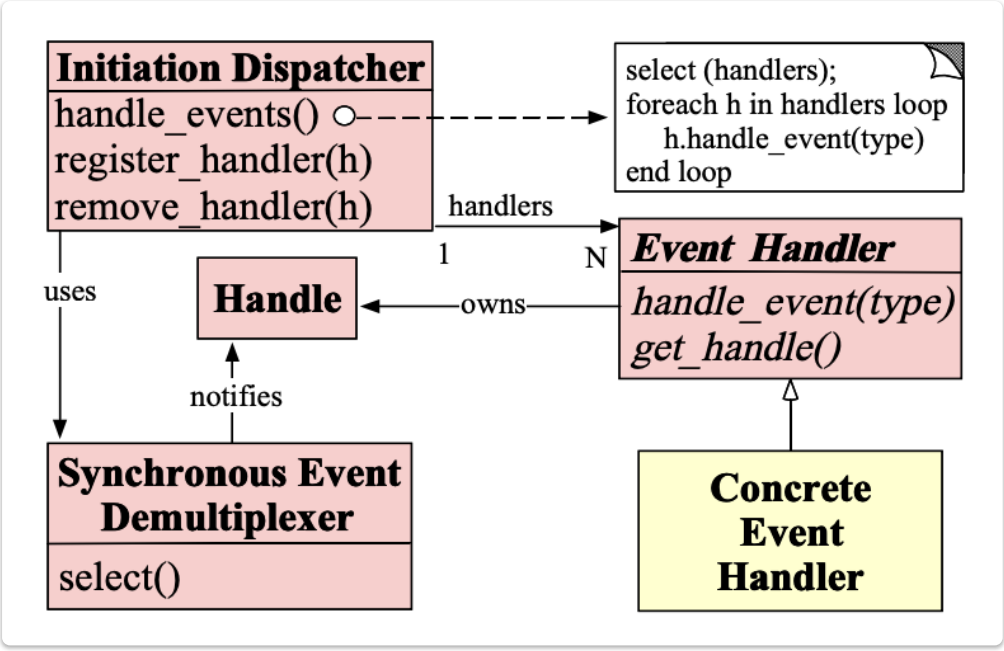
```
//创建一个描述符
epollfd = epoll_create(FDSIZE);
//添加监听描述符事件
add_event(epollfd,listenfd,EPOLLIN);
//循环等待
for (;;) {
        //该函数返回已经准备好的描述符事件数目
        ret = epoll_wait(epollfd,events,EPOLLEVENTS,-1);

        // 处理接收到的连接
        handle_events(epollfd,events,ret,listenfd,buf);
}

//事件处理函数
static void handle_events(int epollfd,struct epoll_event *events,int num,int
listenfd,char *buf) {
        //进行遍历;这里只要遍历已经准备好的io事件。num并不是当初epoll_create时的FDSIZE。
        for (i = 0;i < num;i++) {
                fd = events[i].data.fd;
                //根据描述符的类型和事件类型进行处理
                if ((fd == listenfd) &&(events[i].events & EPOLLIN))
                        handle_accpet(epollfd,listenfd);
                else if (events[i].events & EPOLLIN)
                        do_read(epollfd,fd,buf);
                else if (events[i].events & EPOLLOUT)
                        do_write(epollfd,fd,buf);
        }
}
```

reactor

reactor 模式是对多路复用+非阻塞 IO 基于 C++ 面向对象编程的封装。

- Handles 系统资源句柄，对应 Linux 中的 fd
- Synchronous Event Demultiplexer 事件分离器，封装了 poll/ epoll 系统调用细节
- Initiation Dispatcher reactor模式封装对象，事件处理类的增删，调用 poll
- Event Handler 事件处理类，每个 fd 对应一个事件处理对象
    - Concrete Event Handler 子类，覆写 handle_event 回调函数

https://www.dre.vanderbilt.edu/~schmidt/PDF/reactor-siemens.pdf



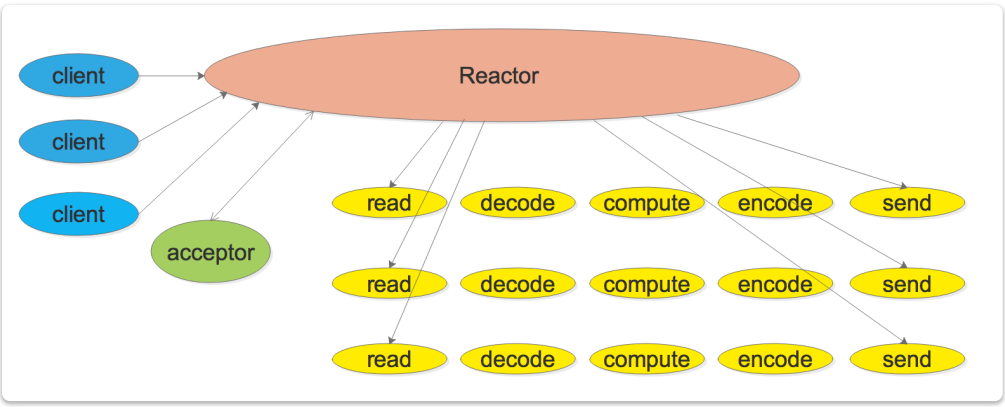
缺点

- 非抢占式，执行依赖事件发生顺序，容易导致超时
- 代码中会写很多回调，回想一下首页的 CompleteFuture 写法。

网络服务器结构

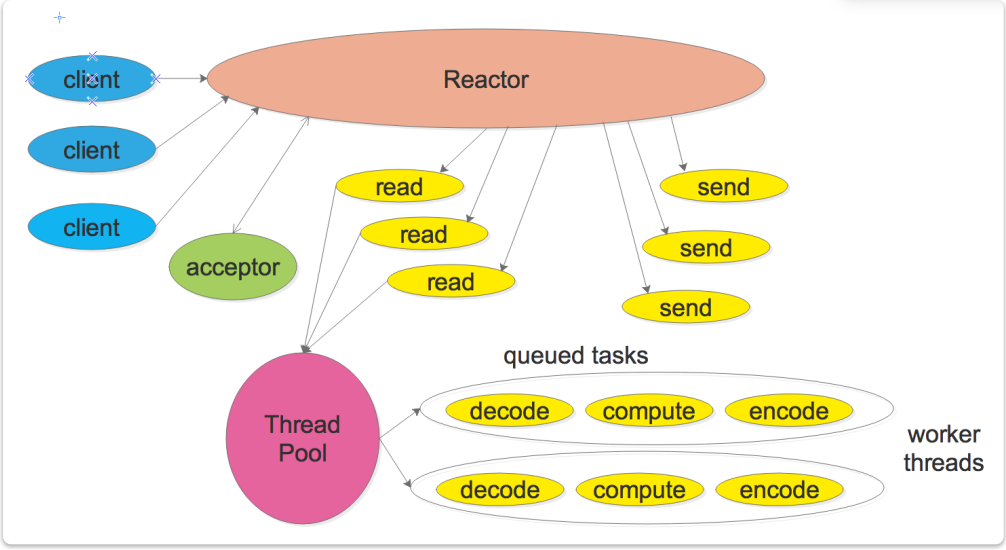
方案	model	UNP 对应	阻塞/非阻塞	多进程	多线程	IO 复用	长连接	并发性	多核	开销	多连接互通	顺序性	线程数确定	特点
0	accept+read/write	0	阻塞	no	no	no	no	无	no	低	no	yes	yes	一次服务一个客户
1	accept+fork	1	阻塞	yes	no	no	yes	低	yes	高	no	yes	no	process-per-connection
2	accept+thread	6	阻塞	no	yes	no	yes	中	yes	中	yes	yes	no	thread-per-connection
3	prefork	2/3/4/5	阻塞	yes	no	no	yes	低	yes	高	no	yes	no	见 UNP
4	pre threaded	7/8	阻塞	no	yes	no	yes	中	yes	中	yes	yes	no	见 UNP
5	poll (reactor)	sec6.8	非阻塞	no	no	yes	yes	高	no	低	yes	yes	yes	单线程 reactor
6	reactor+thread-per-task	无	非阻塞	no	yes	yes	yes	中	yes	中	yes	no	no	thread-per-request
7	reactor+worker thread	无	非阻塞	no	yes	yes	yes	中	yes	中	yes	yes	no	worker-thread-per-connection
8	reactor+thread poll	无	非阻塞	no	yes	yes	yes	高	yes	低	yes	no	yes	主线程 IO，工作线程计算
9	multiple reactors	无	非阻塞	no	yes	yes	yes	高	yes	低	yes	yes	yes	one loop per thread

方案5



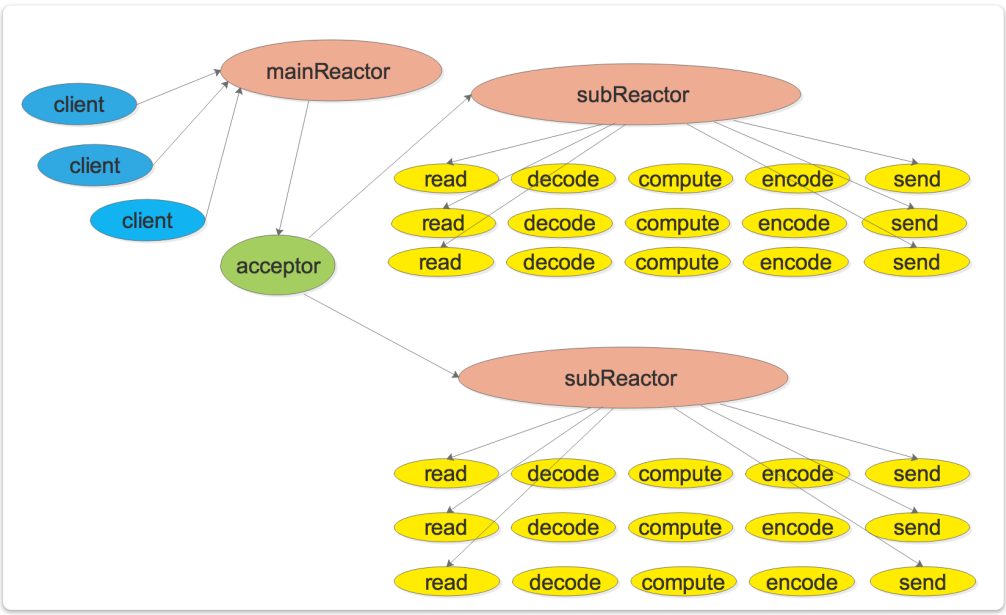
典型 reactor 模式，多核服务器中只能使用一个核

方案8



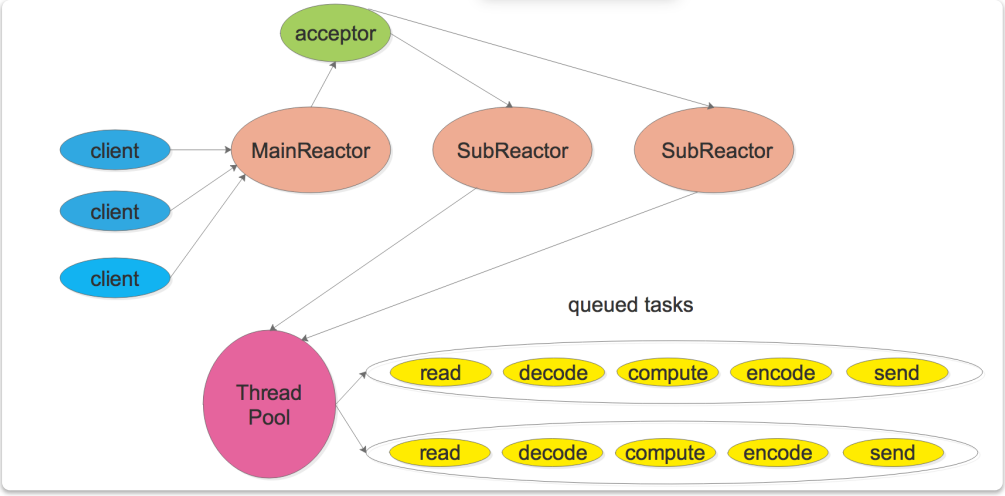
全部的 IO 工作都在一个 reactor 线程完成，而计算任务交给 thread pool。适用于计算量大、IO处理简单的场景。消息返回顺序是乱序的，需要客户端自己维护。

方案9



multiple reactors，Netty 内置的多线程方案，一个 main reactor 拖着多个子 reactor。
一个 main reactor 负责 accept 连接，然后把连接挂在某个 sub reactor 中，连接在子 reactor 中完成IO读写。
相较于方案8，少了两次 thread 切换。

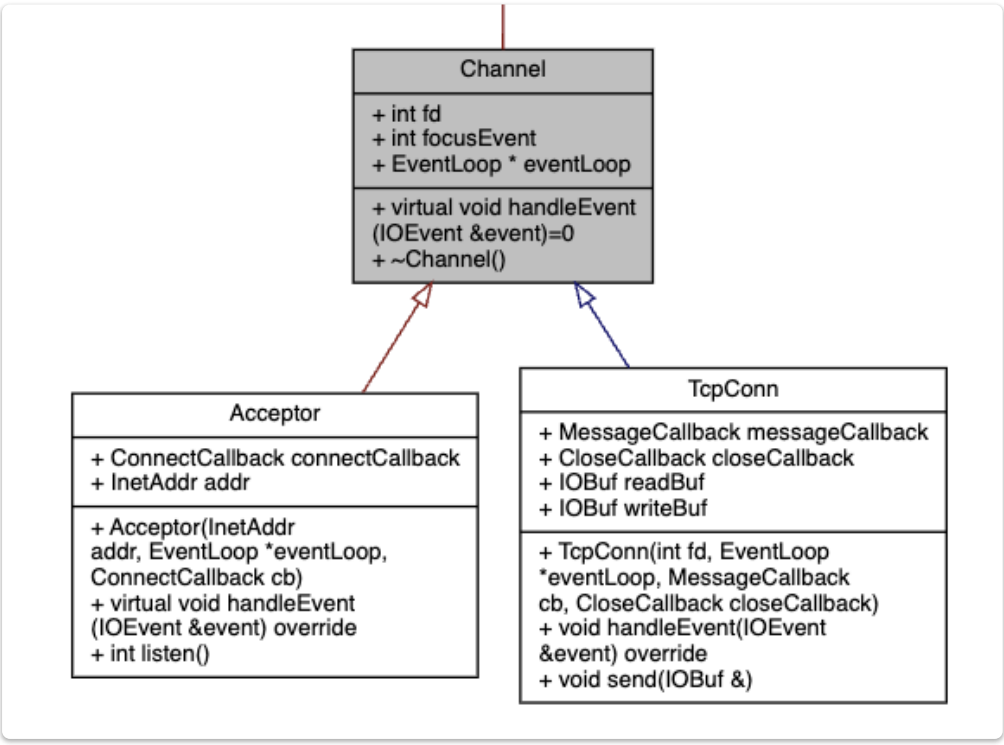
方案10

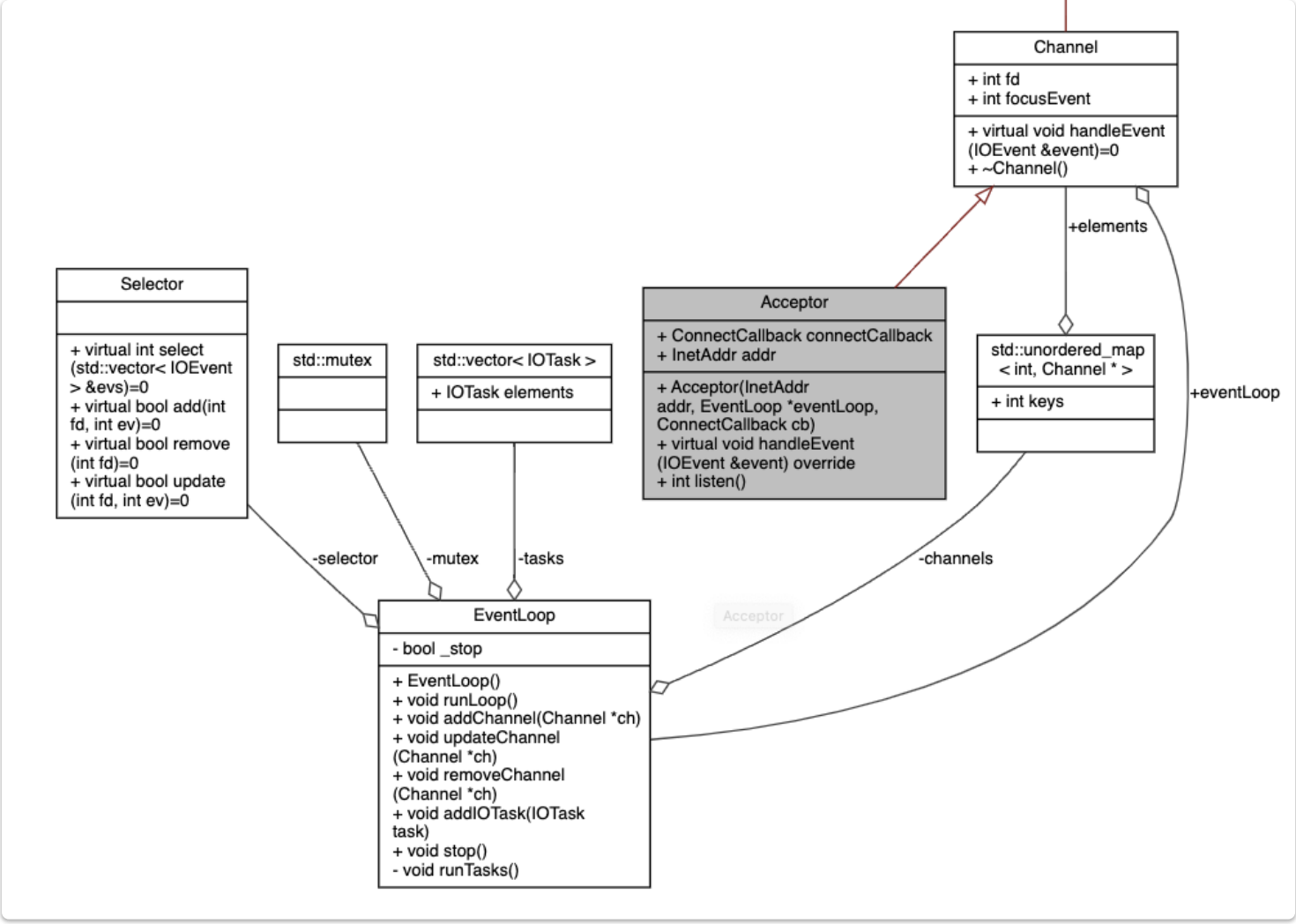


方案8+方案9混合态。

我们来实现一个 reactor 服务器

https://git.in.zhihu.com/fengyuwei/protobuf-rpc





基于 reactor 模式实现了一个单线程 TcpServer 服务器，对外提供 messageCallback 定义了tcp消息的回调函数。

添加 protobuf 功能，实现一个 rpc server

```
// client
int main() {
    MyChannel channel;
    channel.init("127.0.0.1", 8081);

    echo::EchoRequest request;
    echo::EchoResponse response;
    request.set_msg("hello, myrpc.");

    echo::EchoService_Stub stub(&channel);
    MyController cntl;
    stub.Echo(&cntl, &request, &response, NULL);
    std::cout << "resp:" << response.msg() << std::endl;

    return 0;
}


// server
class MyEchoService : public echo::EchoService {
public:
  virtual void Echo(::google::protobuf::RpcController* /* controller */,
```

```
                        const ::echo::EchoRequest* request,
                        ::echo::EchoResponse* response,
                        ::google::protobuf::Closure* done) {
        std::cout << request->msg() << std::endl;
        response->set_msg(
                std::string("I have received '") + request->msg() + std::string("'"));
        done->Run();
    }
};//MyEchoService

int main() {
    InetAddr addr4;
    addr4.sin_family = AF_INET;
    addr4.sin_port = htons(8081);
    addr4.sin_addr.s_addr = INADDR_ANY;

    TcpServer* tcpServer = new TcpServer(addr4);
    RpcServer* rpcServer = new RpcServer(tcpServer);
    MyEchoService echo_service;
    rpcServer->addService(&echo_service);
    rpcServer->start();

    return 0;
}
```

1. 为什么 server 端只需要实现`MyEchoService::Echo`函数，client端只需要调用`EchoService_Stub::Echo`就能发送和接收对应格式的数据？中间的调用流程是怎么样子的？
2. 如果 server 端接收多种 pb 数据（例如还有一个 method `rpc Post(DeepLinkReq) returns (DeepLinkResp);`），那么怎么区分接收到的是哪个格式？
3. 区分之后，又如何构造出对应的对象来？例如`MyEchoService::Echo`参数里的`EchoRequest EchoResponse`，因为 rpc 框架并不清楚这些具体类和函数的存在，框架并不清楚具体类的名字，也不清楚 method 名字，却要能够构造对象并调用这个函数？

后记

- reactor 之后，现在处理 IO 并发还有什么模式？
- 实现 multi reactor 模式服务器