实验4 Epoll与Reactor模式

互联网络程序设计实验

实验目标

- 封装Epoll
- 按照Reactor模式构建io引擎

基础概念

- 同步和异步
  - 同步：两个同步任务相互依赖，并且一个任务必须以依赖于另一任务的某种方式执行。比如：在A->B事件模型中，你需要先完成A才能执行B。换句话说，同步调用中被调用者未处理完请求之前，调用不返回，调用者会一直等待结果的返回。
  - 异步：两个异步的任务完全独立的，一方的执行不需要等待另一方的执行。换句话说，异步调用中一调用就返回结果不需要等待结果返回，当结果返回的时候通过回调函数或者其它方式拿着结果再做相关事情。

基础概念

- 阻塞和非阻塞
  - 阻塞： 阻塞就是发起一个请求，调用者一直等待请求结果返回，也就是当前线程会被挂起，无法从事其它任务，只有当条件就绪才能继续。
  - 非阻塞： 非阻塞就是发起一个请求，调用者不用一直等着结果返回，可以先去干其它事情。

IO

- 常见的IO类型有AIO, BIO, NIO, IO复用
  - BIO：同步并阻塞，即客户端有连接请求时服务器端就需要启动一个线程进行处理。如果这个连接不做任何事情会造成不必要的线程开销。
  - NIO：同步非阻塞，服务器实现模式为一个请求一个线程，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询到连接有I/O请求时才启动一个线程进行处理。

IO

- AIO：异步非阻塞，服务器实现模式为一个有效请求一个线程，客户端的I/O请求都是由OS先完成了再通知服务器应用去启动线程进行处理。
- IO复用：O多路复用是指内核一旦发现进程指定的一个或者多个IO条件准备读取，它就通知该进程。与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。

实验原理

- Epoll是linux上提供的一种IO复用机制，其使用一种机制，等待大量的读写事件，和以往的机制完全不同，传统的IO方式是每个IO请求都是必须阻塞，等待其完成为止，或者非阻塞，IO不能立刻成功就马上返回。
- Epoll使用了完全不同的方式，其让需要等待IO的事件向其注册，并在没有其它操作时阻塞，直到注册的至少一个事件可用时为止或者时间到为止。
- 通过这样的方式编程，可以使一个线程同时启用大量的IO，同时不用带来轮询的开销。
- 这种方式替代了传统的多进程或者多线程的编程模型，减少了进程或线程切换的开销，能够达到很好的性能要求。但随之而来的是代码量的增加。

实验要求： EPOLL封装

- Epoll是一个文件
- Epoll的主要操作
  - 创建
  - 销毁
  - 注册事件
  - 注销事件
  - 修改事件
  - 等待事件发生
- 事件如何描述?

EPOLL机制的系统调用

1、创建 epoll 文件描述符：

```
#include <sys/epoll.h>
int epoll_create(int size);
```

2、注册，更改或者删除 epoll 中的文件描述符及等待的事件：

```
#include <sys/epoll.h>
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
```

EPOLL机制的系统调用

3、等待注册的事件的到来：

```
#include <sys/epoll.h>
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
int epoll_pwait(int epfd, struct epoll_event *events, int maxevents, int timeout, const sigset_t *sigmask);
```

4、关闭 epoll 文件描述符：

```
#include <unistd.h>
int close(int fd);
```

EPOLL实现DEMO

```
#include <iostream>
#include <sys/socket.h>
#include <sys/epoll.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

using namespace std;

#define MAXLINE 5
#define OPEN_MAX 100
#define LISTENQ 20
#define SERV_PORT 5000
#define INFTIM 1000
```

EPOLL实现DEMO

```
int main(int argc, char* argv[])
{
    int listen_fd, connfd_fd, socket_fd, epfd, nfds;
    ssize_t n;
    char line[MAXLINE];
    socklen_t clilen;

    //声明epoll_event结构体的变量,ev用于注册事件,数组用于回传要处理的事件
    struct epoll_event ev,events[20];
    //生成用于处理accept的epoll专用的文件描述符
    epfd=epoll_create(5);
    struct sockaddr_in clientaddr;
    struct sockaddr_in serveraddr;
    listen_fd = socket(AF_INET, SOCK_STREAM, 0);
    //设置与要处理的事件相关的文件描述符
    ev.data.fd = listen_fd;
    //设置要处理的事件类型
    ev.events=EPOLLIN|EPOLLET;
    //注册epoll事件
    epoll_ctl(epfd,EPOLL_CTL_ADD,listen_fd,&ev);
```

EPOLL实现DEMO

```
 42      memset(&serveraddr, 0, sizeof(serveraddr));
 43      serveraddr.sin_family = AF_INET;
 44      serveraddr.sin_addr.s_addr = htonl(INADDR_ANY);
 45      serveraddr.sin_port = htons(SERV_PORT);
 46
  47              if   (bind(listen_fd,   (struct   sockaddr*)&serveraddr,
sizeof(serveraddr)) == -1)
 48      {
  49                          printf("bind   socket   error:   %s(errno:
%d)\n",strerror(errno),errno);
 50          exit(0);
 51      }
 52
 53      if (listen(listen_fd, LISTENQ) == -1)
 54      {
 55          exit(0);
 56      }
 57
```

EPOLL实现DEMO

```
58      for ( ; ; )
59      {
60          //等待epoll事件的发生
61          nfds = epoll_wait(epfd,events,20,500);
62          //处理所发生的所有事件
63          for (int i = 0; i < nfds; ++i)
64          {
65              if (events[i].data.fd == listen_fd)//如果新监测到一个SOCKET用
户连接到了绑定的SOCKET端口，建立新的连接。
66
67              {
68                   connfd_fd = accept(listen_fd,(sockaddr *)&clientaddr,
&clilen);
69                  if (connfd_fd < 0){
70                      perror("connfd_fd < 0");
71                      exit(1);
72                  }
73                  char *str = inet_ntoa(clientaddr.sin_addr);
74                  cout << "accapt a connection from " << str << endl;
75                  //设置用于读操作的文件描述符
76                  ev.data.fd = connfd_fd;
77                  //设置用于注测的读操作事件
78                  ev.events = EPOLLIN|EPOLLET;
79                  //注册ev
80                  epoll_ctl(epfd,EPOLL_CTL_ADD,connfd_fd,&ev);
81              }
```

EPOLL实现DEMO

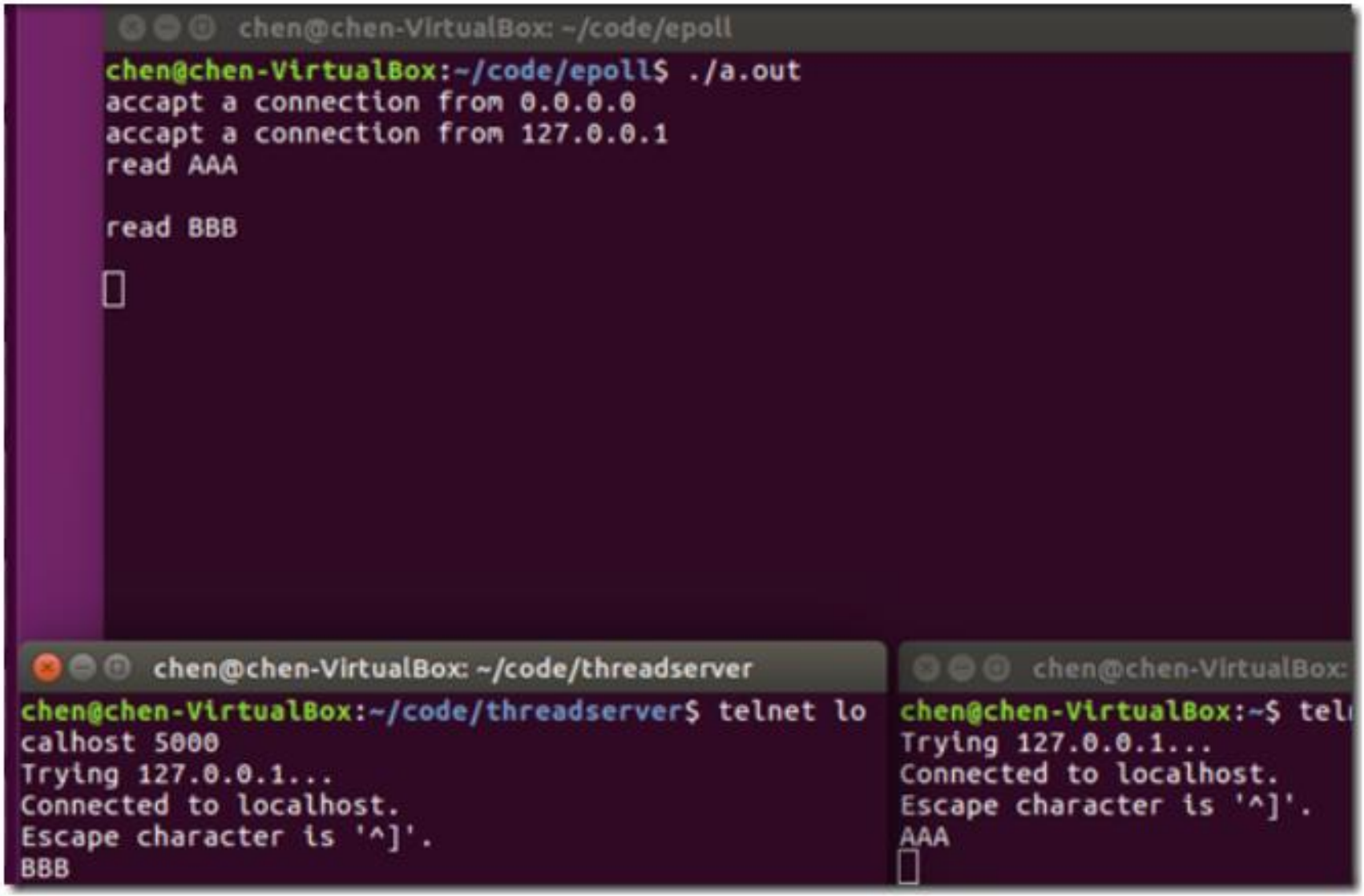
```
 82              else if (events[i].events&EPOLLIN)//如果是已经连接的用户，并且
收到数据，那么进行读入。
 83              {
 84                  memset(&line,'\0', sizeof(line));
 85                  if ( (socket_fd = events[i].data.fd) < 0)
 86                      continue;
 87                  if ( (n = read(socket_fd, line, MAXLINE)) < 0) {
 88                      if (errno == ECONNRESET) {
 89                          close(socket_fd);
 90                          events[i].data.fd = -1;
 91                      } else
 92                          std::cout<<"readline error"<<std::endl;
 93                  } else if (n == 0) {
 94                      close(socket_fd);
 95                      events[i].data.fd = -1;
 96                  }
 97                  cout << line << endl;
 98                  //设置用于写操作的文件描述符
 99                  ev.data.fd = socket_fd;
100                  //设置用于注测的写操作事件
101                  ev.events = EPOLLOUT|EPOLLET;
102                  //修改socket_fd上要处理的事件为EPOLLOUT
103                  //epoll_ctl(epfd,EPOLL_CTL_MOD,socket_fd,&ev);
104              }
```

EPOLL实现DEMO

```
105                 else if (events[i].events&EPOLLOUT) // 如果有数据发送
106                 {
107                     socket_fd = events[i].data.fd;
108                     write(socket_fd, line, n);
109                     //设置用于读操作的文件描述符
110                     ev.data.fd = socket_fd;
111                     //设置用于注测的读操作事件
112                     ev.events = EPOLLIN|EPOLLET;
113                     //修改socket_fd上要处理的事件为EPOLIN
114                     epoll_ctl(epfd,EPOLL_CTL_MOD,socket_fd,&ev);
115                 }
116             }
117         }
118         return 0;
119 }
```

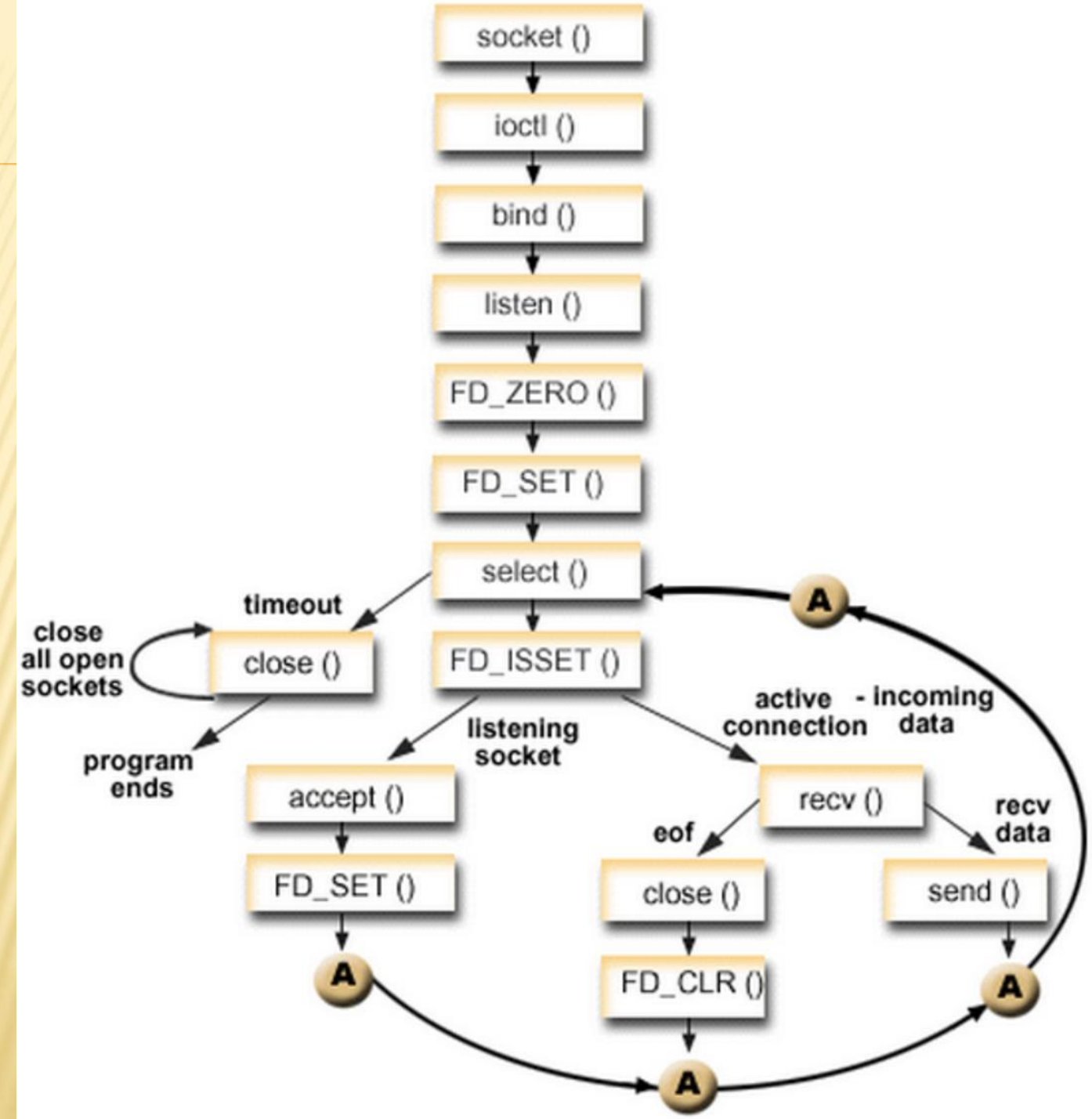
EPOLL实现DEMO

执行效果如下：



LINUX中其它IO复用

- select
  - select　　函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述符就绪（有数据可读、可写、或者有except），或者超时（timeout指定等待时间，如果立即返回设为null即可），函数返回。当select函数返回后，可以通过遍历fdset，来找到就绪的描述符。


socket ()
ioctl ()
bind ()
listen ()
FD_ZERO ()
FD_SET ()
select ()
timeout
close all open sockets
close ()
program ends
FD_ISSET ()
listening socket
accept ()
FD_SET ()
A
active connection
- incoming data
recv ()
eof
recv data
close ()
FD_CLR ()
send ()
A
A
A

LINUX中其它IO复用

- poll
  - poll本质上和select没有区别，它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。这个过程经历了多次无谓的遍历。

REACTOR实验原理

- Reactor是一种事件驱动机制。和普通函数调用的不同之处在于：应用程序不是主动调用某个API完成处理，而是恰恰相反，Reactor逆置了事件处理流程，应用程序需要提供相应的接口并注册到Reactor上，如果相应的事件发生，Reactor将主动调用应用程序注册的接口，这些接口又称为“回调函数”。

REACTOR实验原理

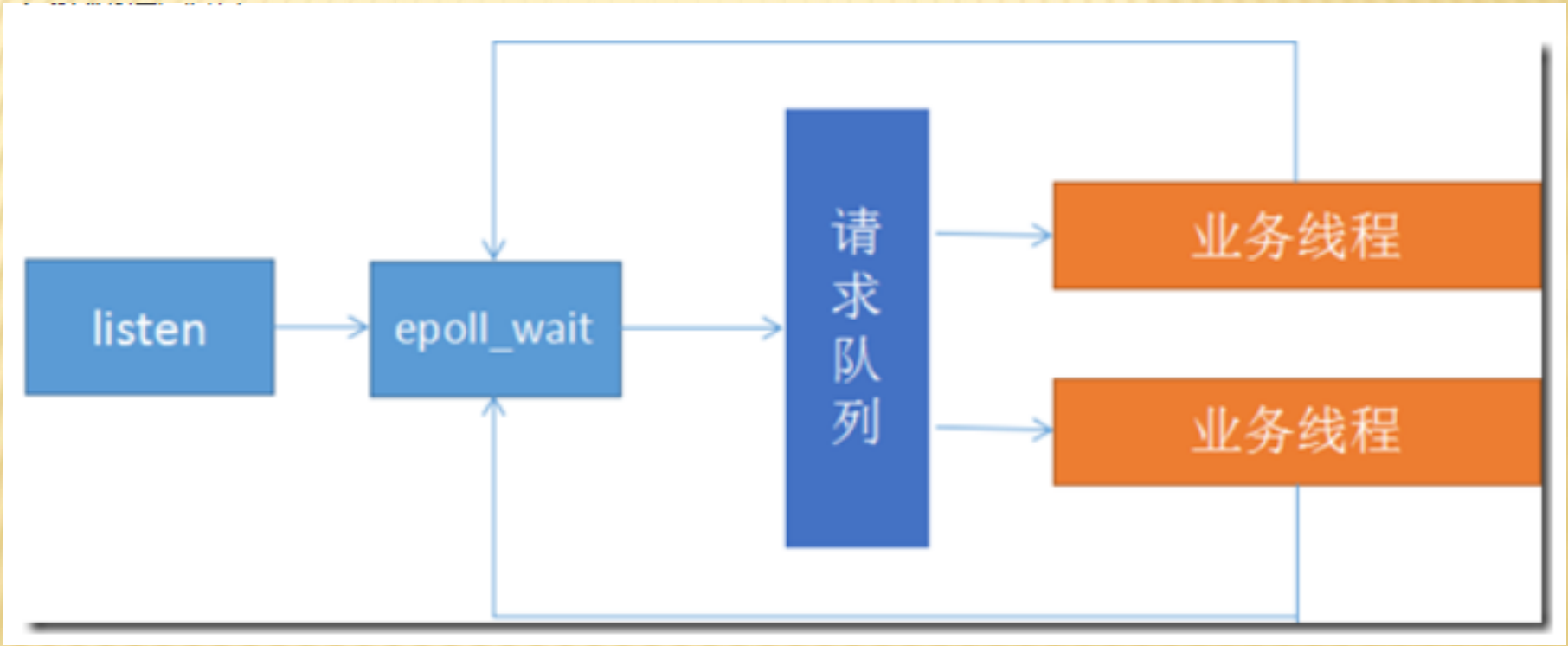
- Reactor模式是编写高性能网络服务器的必备技术之一，它具有如下优点：
- 1）响应快，不必为单个同步时间所阻塞，虽然Reactor本身依然是同步的；
- 2）编程相对简单，可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销；
- 3）可扩展性，可以方便地通过增加Reactor实例个数来充分利用CPU资源；
- 4）可复用性，reactor框架本身与具体事件处理逻辑无关，具有很高的复用性。

REACTOR模式

- Reactor模式实现非常简单，使用同步IO模型，即业务线程处理数据需要主动等待或询问，主要特点是利用epoll监听listen描述符是否有响应，及时将客户连接信息放于一个队列，epoll和队列都是在主进程/线程中，由子进程/线程来接管各个描述符，对描述符进行下一步操作，包括connect和数据读写。主程读写就绪事件。

REACTOR模式

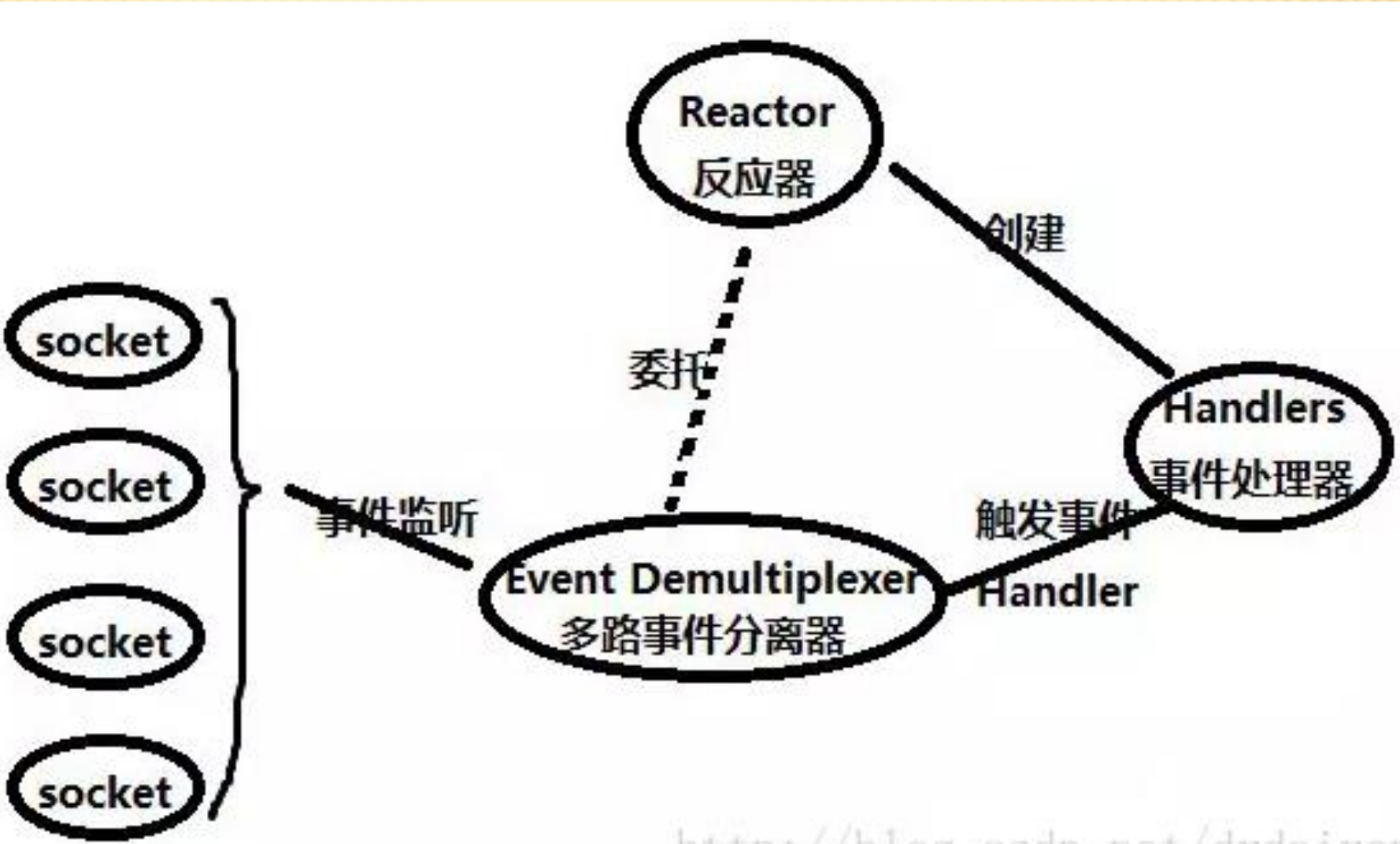
- 大致流程图如下：



REACTOR模式

- 参考文献
- 主要对象
  - IDispach接口
  - Demultiplexer对象
  - Handle
  - Handler

REACTOR模式


Reactor
反应器
创建
委托
Handlers
事件处理器
socket
socket
socket
socket
事件监听
Event Demultiplexer
多路事件分离器
触发事件
Handler
http://blog.csdn.net/drdairen

REACTOR模式

- 上图就是基于socket的reactor模式，socket作为文件fd，被多路事件分离器监听，这个多路事件分离器一般就是指Select/poll/epoll这种IO多路复用的机制，select/poll/epoll将事件分离出来后，委托给反应器去寻找创建对应的事件处理器，当创建完成后，以后有socket事件进来，直接由select分离出来，交由对应的Handler去处理。

REACTOR模式

- 标准的Reactor模式图一共有5个角色：
- 1、描述符（handle）：
- 由操作系统提供，用于识别每一个事件，如Socket描述符、文件描述符fd等。在Linux中，它用一个整数来表示。 事件可以来自外部，如来自客户端的连接请求、数据等。事件也可以来自内部，如定时器事件。
- 2、同步事件分离器（demultiplexer）：
- 是一个函数，用来等待一个或多个事件的发生。调用者会被阻塞，直到分离器分离的描述符集上有事件发生。 Linux的select/poll/epoll函数是一个经常被使用的分离器。
- 3、事件处理器接口（event handler）：
- 是由一个或多个模板函数组成的接口。这些模板函数描述了和应用程序相关的对某个事件的操作。

REACTOR模式

- 4、具体的事件处理器：
- 是事件处理器接口的实现。它实现了应用程序提供的某个服务。也就是libev中的回调函数callback函数。每个具体的事件处理器总和一个描述符相关。它使用描述符来识别事件、识别应用程序提供的服务。
- 5、Reactor 管理器（reactor）：
- 定义了一些接口，用于应用程序控制事件调度，以及应用程序注册、删除事件处理器和相关的描述符。它是事件处理器的调度核心。（如libev中的loop）Reactor管理器使用同步事件分离器来等待事件的发生。一旦事件发生，Reactor管理器先是分离每个事件，然后调度事件处理器，最后调用相关的模板函数来处理这个事件。

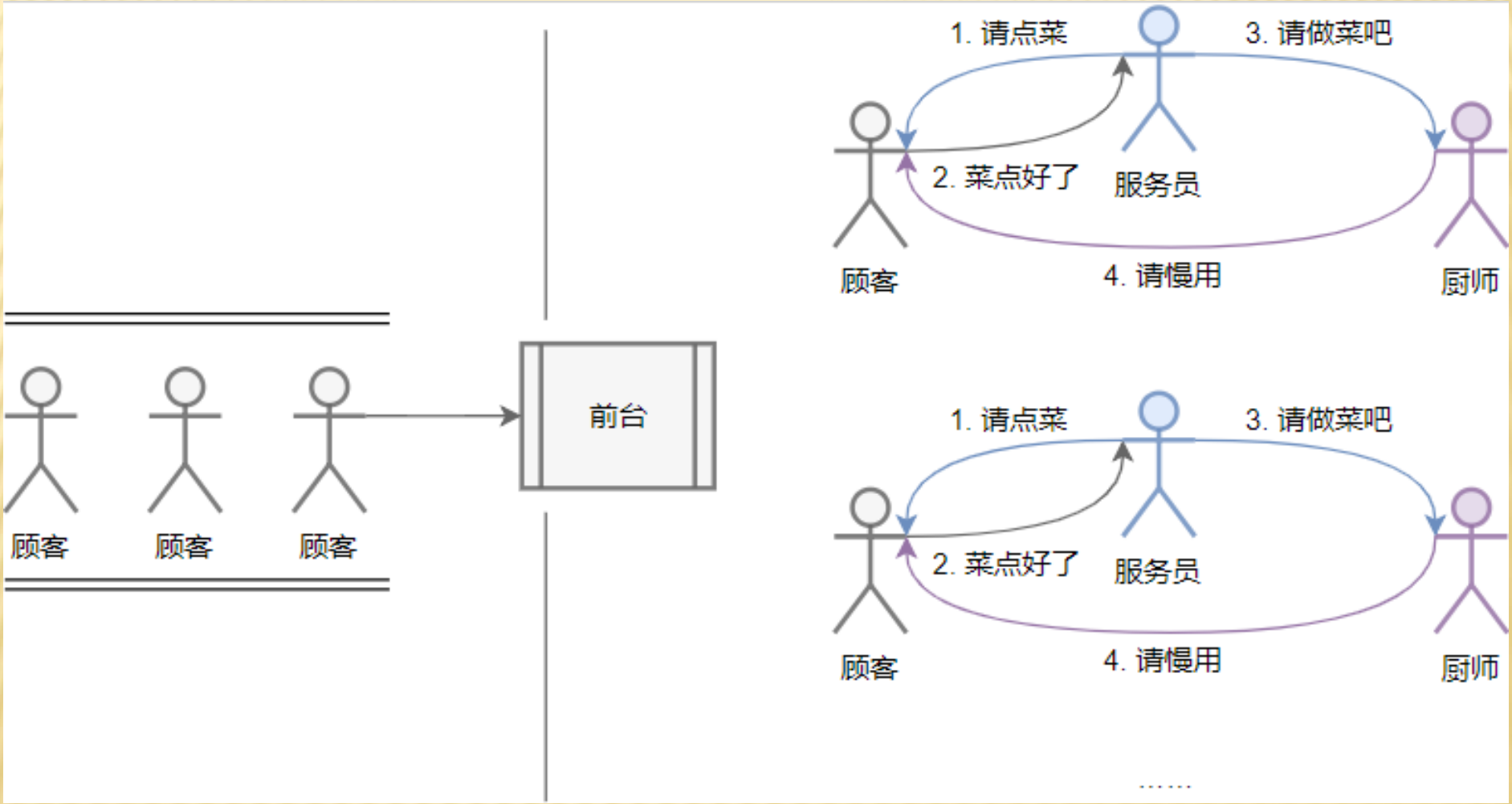
REACTOR模式

- I/O多路复用又被称为“事件驱动”，就是通过一种机制，一个进程可以监视多个描述符，一旦某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作，技术上是通过调用操作系统的select、pselect、poll、epoll来实现。
- 与多进程和多线程技术相比，I/O多路复用技术的最大优势是系统开销小，系统不必创建进程/线程，也不必维护这些进程/线程，从而大大减小了系统的开销。
- Reactor是一种应用在服务器端的开发模式，目的是提高服务端程序的并发能力，其实就是实现了I/O多路复用这种I/O模型。

REACTOR模式

- 传统的多线程方式
- 每个顾客都有自己的服务团队（线程），在人少的情况下是可以良好的运作的，流程如下：
- 1. 服务员给出菜单，并等待点菜
- 2. 顾客查看菜单，并点菜
- 3. 服务员把菜单交给厨师，厨师照着做菜
- 4. 厨师做好菜后端到餐桌上

REACTOR模式


顾客
顾客
顾客
前台
1. 请点菜
3. 请做菜吧
2. 菜点好了
服务员
顾客
4. 请慢用
厨师
1. 请点菜
3. 请做菜吧
2. 菜点好了
服务员
顾客
4. 请慢用
厨师
.....

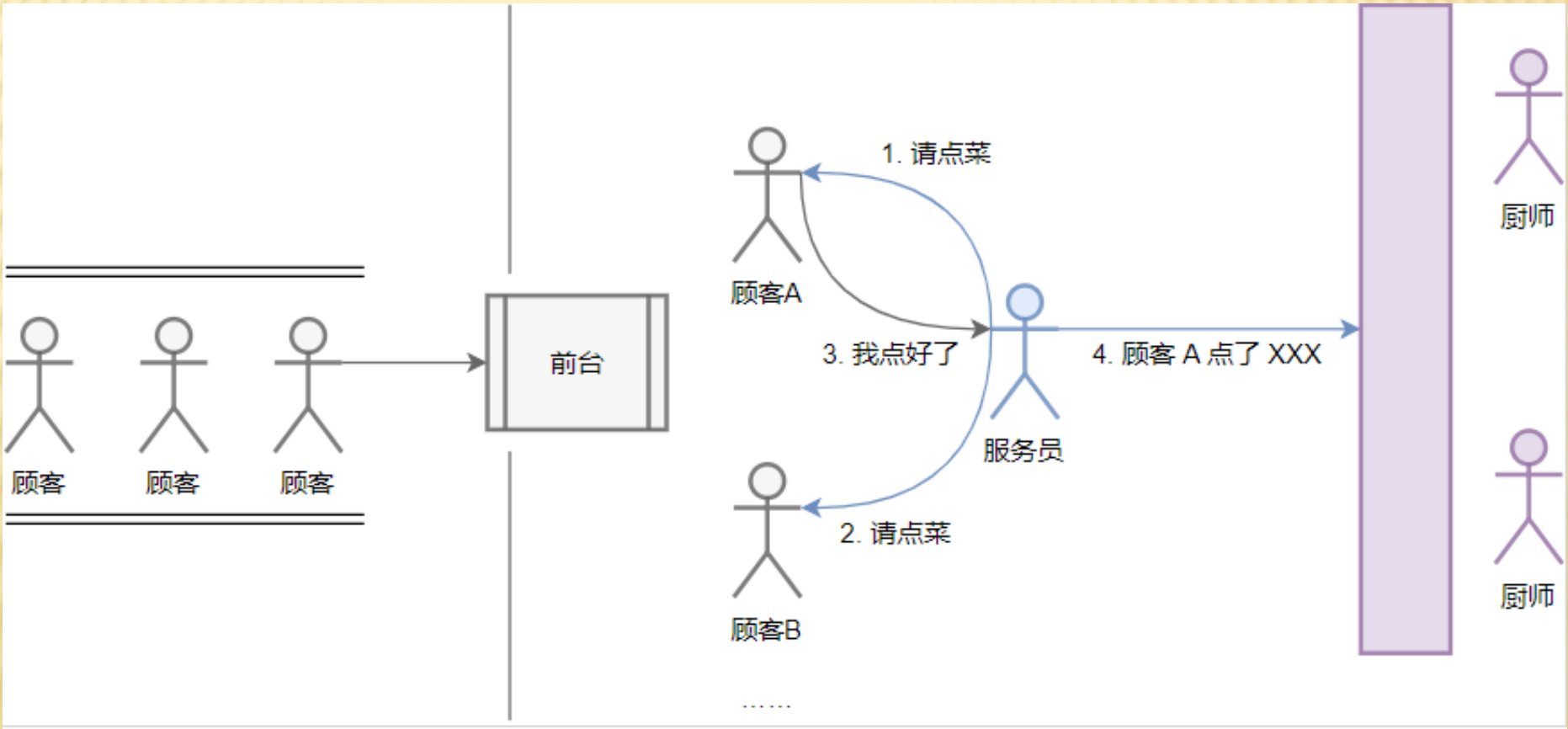
REACTOR模式

- 现在餐厅的口碑好，顾客人数不断增加，这时服务员就有点处理不过来了。这时老板发现，每个服务员在服务完客人后，都要去休息一下，因此老板就说，“你们都别休息了，在旁边待命”。这样可能 10 个服务员也来得及服务 20 个顾客了。这也是“线程池”的方式， 通过重用线程来减少线程的创建和销毁时间，从而提高性能。

REACTOR模式

- Reactor方式
- 但是客人又进一步增加了，仅仅靠剥削服务员的休息时间也没有办法服务这么多客人。老板仔细观察，发现其实服务员并不是一直在干活的，大部分时间他们只是站在餐桌旁边等客人点菜。
- 于是老板就对服务员说，客人点菜的时候你们就别傻站着了，先去服务其他客人，有客人点好的时候喊你们再过去。对应于下图：

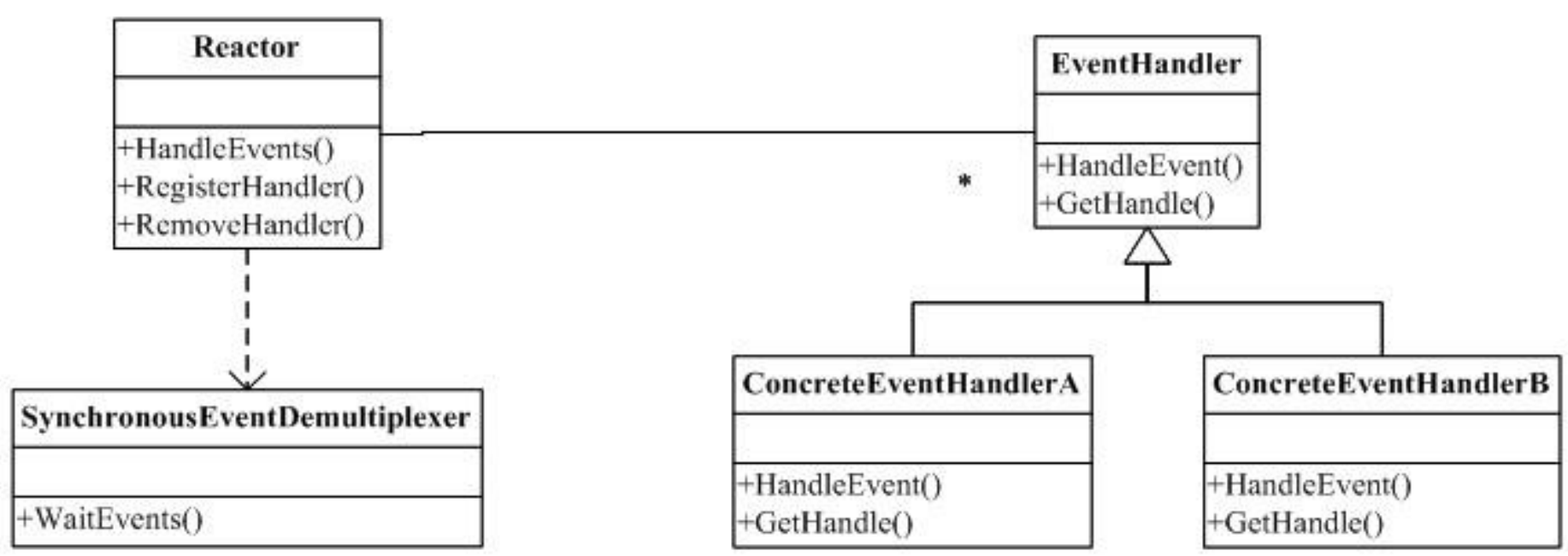
REACTOR模式



REACTOR模式

- 最后，老板发现根本不需要那么多的服务员，于是裁了一波员，最终甚至可以只有一个服务员。
- 这就是 Reactor 模式的核心思想：减少等待。当遇到需要等待 IO 时，先释放资源，而在 IO 完成时， 再通过事件驱动 (event driven) 的方式，继续接下来的处理。从整体上减少了资源的消耗。

REACTOR模式


Reactor
+HandleEvents()
+RegisterHandler()
+RemoveHandler()
EventHandler
+HandleEvent()
+GetHandle()
*
SynchronousEventDemultiplexer
+WaitEvents()
ConcreteEventHandlerA
+HandleEvent()
+GetHandle()
ConcreteEventHandlerB
+HandleEvent()
+GetHandle()

REACTOR模式

- 在上图中，可以看到主要有以下四种角色：
  1. Reactor:
  Reactor是Reactor模式中最为关键的角色，它是该模式最终向用户提供接口的类。用户可以向Reactor中注册EventHandler(3)，然后Reactor在“反应(react)”的时候，发现用户注册的fd上有事件发生，就会回调用户的事件处理函数。下面是一个简单的设计：

REACTOR模式

```
1  class Reactor
2  {
3  public:
4
5      /// 构造函数
6      Reactor();
7
8      /// 析构函数
9      ~Reactor();
10
11     /// 向reactor中注册关注事件evt的handler(可重入)
12     /// @param  handler 要注册的事件处理器
13     /// @param  evt     要关注的事件
14     /// @retval 0       注册成功
15     /// @retval -1      注册出错
16     int RegisterHandler(EventHandler * handler, event_t evt);
17
18     /// 从reactor中移除handler
19     /// @param  handler 要移除的事件处理器
20     /// @retval 0       移除成功
21     /// @retval -1      移除出错
22     int RemoveHandler(EventHandler * handler);
23
24     /// 处理事件,回调注册的handler中相应的事件处理函数
25     /// @param  timeout 超时时间(毫秒)
26     void HandleEvents(int timeout = 0);
27
28 private:
29
30     ReactorImplementation * m_reactor_impl; ///< reactor的实现类
31 };
```

REACTOR模式

- 2. SynchrousEventDemultiplexer: SynchrousEventDemultiplexer也是Reactor中一个比较重要的角色，它是Reactor用来检测用户注册的fd上发生的事件的利器，通过Reactor得知了哪些fd上发什么了什么样的事件，然后以此为依据，来多路分发事件，回调用户的事件处理函数。下面是一个简单的设计:

REACTOR模式

```
1  class EventDemultiplexer
2  {
3  public:
4
5      /// 获取有事件发生的所有句柄以及所发生的事件
6      /// @param  events  获取的事件
7      /// @param  timeout 超时时间
8      /// @retval 0       没有发生事件的句柄(超时)
9      /// @retval 大于0   发生事件的句柄个数
10     /// @retval 小于0   发生错误
11     virtual int WaitEvents(std::map<handle_t , event_t> * events,
12                            int timeout = 0) = 0;
13
14     /// 设置句柄handle关注evt事件
15     /// @retval 0     设置成功
16     /// @retval 小于0 设置出错
17     virtual int RequestEvent(handle_t handle, event_t evt) = 0;
18
19     /// 撤销句柄handle对事件evt的关注
20     /// @retval 0     撤销成功
21     /// @retval 小于0 撤销出错
22     virtual int UnrequestEvent(handle_t handle, event_t evt) = 0;
23 };
24 </handle_t>
```

REACTOR模式

- 3. EventHandler:
  EventHander是用户和Reactor打交道的工具，用户通过向Reactor注册自己的EventHandler，可以告知Reactor在特定事件发生的时候该帮我做些什么。下面是一个简单的设计：

REACTOR模式

```
1  class EventHandler
2  {
3  public:
4
5      /// 获取该handler所对应的句柄
6      virtual handle_t GetHandle() = 0;
7
8      /// 处理读事件的回调函数
9      virtual void HandleRead() {}
10
11     /// 处理写事件的回调函数
12     virtual void HandleWrite() {}
13
14     /// 处理出错事件的回调函数
15     virtual void HandleError() {}
16
17 protected:
18
19     /// 构造函数,只能子类调
20     EventHandler() {}
21
22     /// 析构函数,只能子类调
23     virtual ~EventHandler() {}
24 };
```

REACTOR模式

- 4. ConcreteEventHandler: ConcreteEventHandler是EventHandler的子类，EventHandler是Reactor所用来规定接口的基类，用户自己的事件处理器都必须从EventHandler继承。

The End