# 标题1

## 标题

## 标题

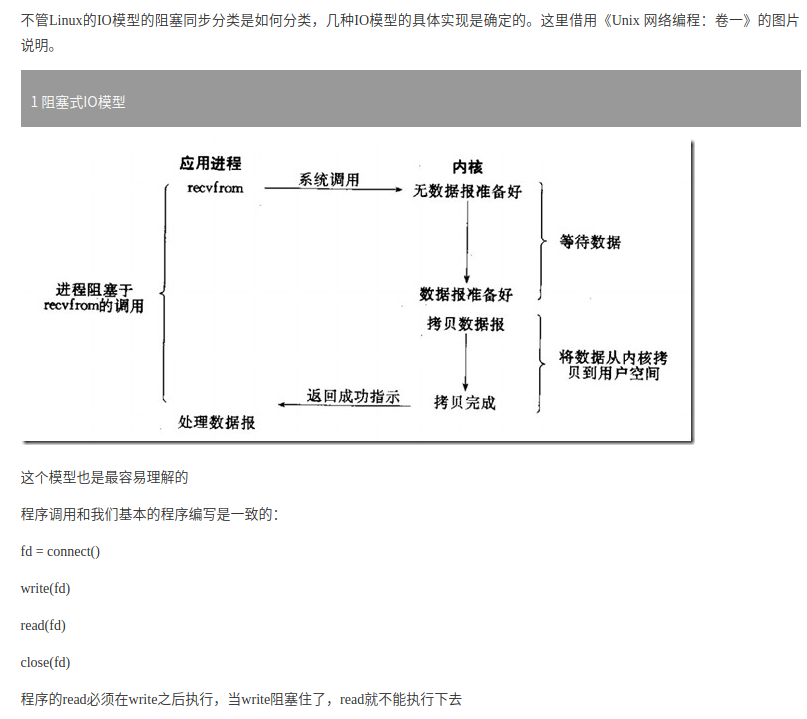
# 网络I/O模型

## 参考资料

<https://blog.csdn.net/lltaoyy/article/details/54861749>

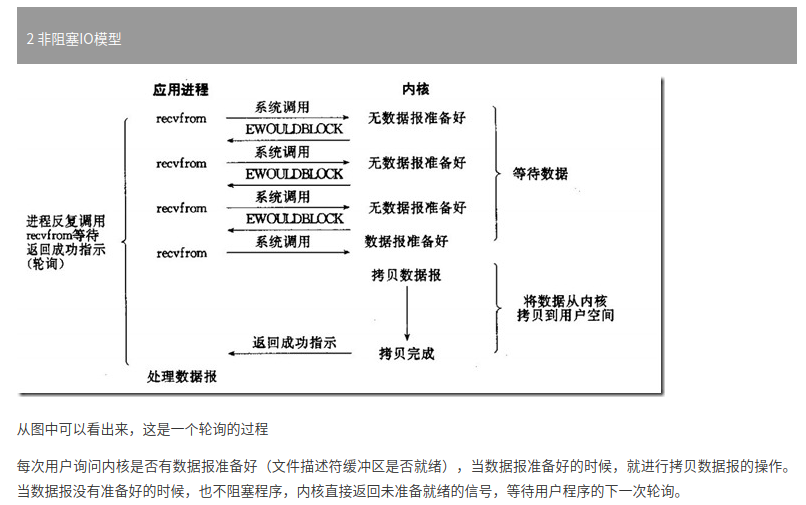
## 阻塞（blocking I/O）

### 定义



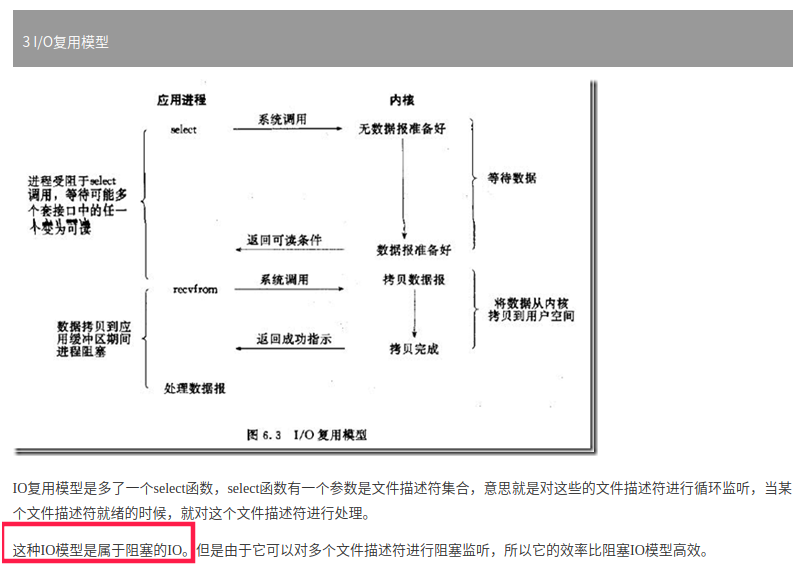
## 非阻塞（non-blocking I/O）

### 定义



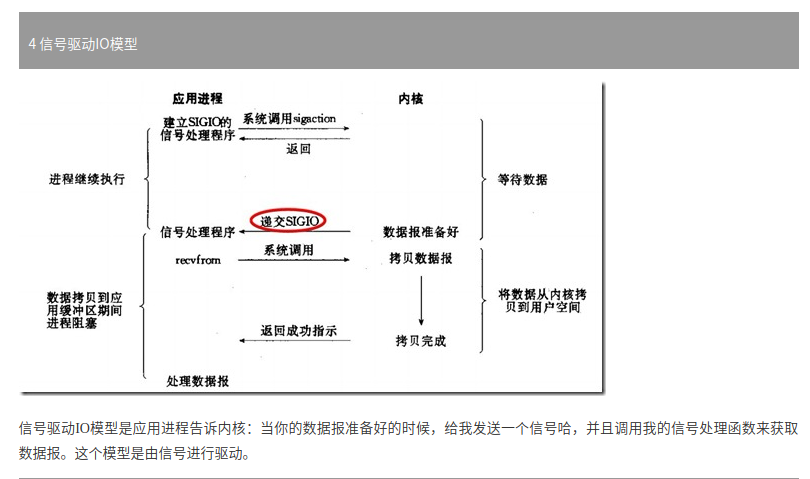
## I/O多路复用（I/O multiplexing）

### 定义



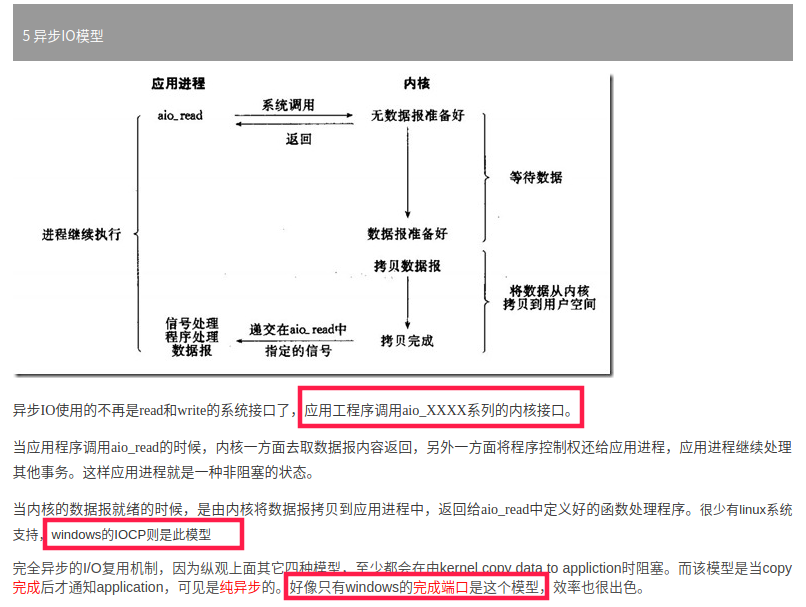
## 信号驱动I/O（signal driven I/O）

### 定义



## 异步I/O（asynchronous I/O）

### 定义



## 重要函数分类

### Select

select对应第3种模型。

### IOCP

IOCP对应第5种模型。

### epoll

那么epoll与kqueue呢？其实也于select属于同一种模型，只是更高级一些，可以看作有了第4种模型的某些特性，如callback机制。

那么，为什么epoll,kqueue比select高级？

答案是，他们无轮询。因为他们用callback取代了。想想看，当套接字比较多的时候，每次select()都要通过遍历FD\_SETSIZE个Socket来完成调度,不管哪个Socket是活跃的,都遍历一遍。这会浪费很多CPU时间。如果能给套接字注册某个回调函数，当他们活跃时，自动完成相关操作，那就避免了轮询，这正是epoll与kqueue做的。

windows or \*nix （IOCP or kqueue/epoll）？

诚然，Windows的IOCP非常出色，目前很少有支持asynchronous I/O的系统，但是由于其系统本身的局限性，大型服务器还是在UNIX下。而且正如上面所述，kqueue/epoll 与 IOCP相比，就是多了一层从内核copy数据到应用层的阻塞，从而不能算作asynchronous I/O类。但是，这层小小的阻塞无足轻重，kqueue与epoll已经做得很优秀了。

# 网络设计模式

## Reactor

### 实例

#### Libevent

#### Epoll

#### Kqueue

#### ACE Reactor

## Proactor

### 实例

#### IOCP

#### ACE Proactor

## Select模型

### 实例

#### Java NIO

# 网络线程模型

## 单线程

### Reactor单线程

#### 只有一个线程

## 多线程

### Reactor多线程

#### 一个Reactor

#### 一个专门线程accept

#### 一组线程处理网络读写

## 主从多线程

### 主从Reactor多线程

#### 多个Reactor

## Master/Worker多线程

### 一个I/O线程

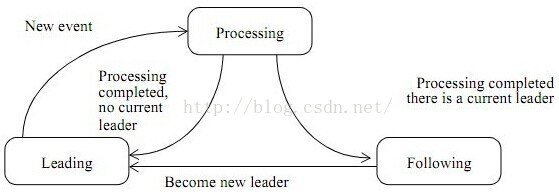
IO线程收到消息后，需要动态分配内存，将请求放在该内存中，然后放入一个队列。

### 多个Worker线程

多个Worker对这个队列进行互斥的操作。

## Leader/Follower多线程

### 状态图



上图就是L/F多线程模型的状态变迁图，共6个关键点：

（1）线程有3种状态：领导leading，处理processing，追随following；

（2）假设共N个线程，其中只有1个leading线程（等待任务），x个processing线程（处理），余下有N-1-x个following线程（空闲）；

（3）有一把锁，谁抢到就是leading；

（4）事件/任务来到时，leading线程会对其进行处理，从而转化为processing状态，处理完成之后，又转变为following；

（5）丢失leading后，following会尝试抢锁，抢到则变为leading，否则保持following；

（6）following不干事，就是抢锁，力图成为leading；

### Leader

永远最多只有一个leader。

所有follower都在等待成为 leader。

线程池启动时会自动产生一个Leader负责等待网络IO事件。

当有一个事件产生时，Leader线程首先通知一个Follower线程将 其提拔为新的Leader，然后自己就去干活了，去处理这个网络事件，处理完毕后加入Follower线程等待队列，等待下次成为Leader。

### Follower

### Processor

### 接口

#### 等待成为Leader（waitToBeLeader）

#### 提升新Leader（promoteNewLeader）

## 半同步/半异步（Half-Sync/Half-Async)

### 同步线程

有一个线程池不断的从队列里获取任务，同步处理，处理完之后再把响应数据由IO线程返回给用户。

### 异步线程

启动一个IO线程使用epoll处理网络数据。当收到一个完整的请求包，把请求放到任务队列。

### 高效半同步/半异步

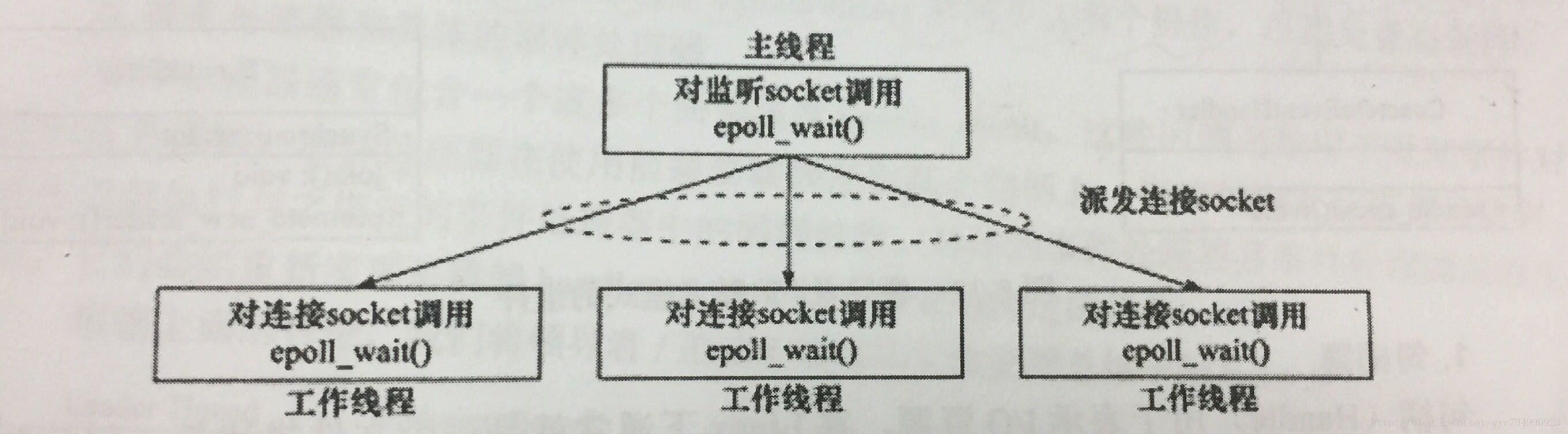
##### 每个工作线程一个I/O Loop

【1】主线程只管理监听socket、连接，得到新的连接socket由工作线程来管理。当有新的连接到来时，主线程就接受并将新返回的连接socket派发给某个工作线程，此后，该socket上任何I/O操作都由被选中的工作线程来处理，知道客户端关闭连接。

【2】主线程向工作线程派发socket的方式，是往它和工作线程之间的管道里写数据。工作线程检测到管道上有数据可读时，就分析是否是一个新的客户端连接请求到来，如果是，就把该socket上的读写事件注册到自己的epoll内核事件表中。

【3】此模式每个线程都维持自己的事件循环，各自监听不同的事件。

【4】



## 半同步/半反应堆（Half Sync/Half Reactor）

# 服务器编程

## 创建socket

### 函数接口

int socket(int domain, int type, int protocol);

### 参数详解

#### Domain

AF\_INET IPv4 Internet protocols ip(7)

AF\_INET6 IPv6 Internet protocols ipv6(7)

#### Type

SOCK\_STREAM：TCP

SOCK\_DGRAM：UDP

#### Protocol

一般传0，表示使用默认的，一般前面2个参数情况下一般只有一个协议。

## 绑定本地端口

### 函数接口

int bind(int sockfd, const struct sockaddr \*addr, socklen\_t addrlen);

### 参数详解

#### Sockfd

通过socket函数创建的文件描述符。

#### Addr

一般通过struct sockaddr\_in来赋值，然后强转(const sockaddr \*)&addr\_in。

#### Addrlen

sizeof(addr\_in)。

## 监听

### 参考资料

<https://blog.csdn.net/shisiye15/article/details/7818637>

<http://blog.chinaunix.net/uid-24782829-id-3456109.html>

### 函数接口

int listen(int sockfd, int backlog);

### 参数详解

#### Sockfd

通过socket函数创建的文件描述符。

#### Backlog

已完成队列的最大长度，可设置512\1024。

## 循环接受客户端连接

### 函数接口

int accept(int socket, struct sockaddr \*restrict address, socklen\_t \*restrict address\_len);

### 参数详解

#### Socket

Listen中使用的socket。

#### Address

sockaddr\_in addr\_client;

int clientlen = sizeof(addr\_client);

int client\_sock;

if ((client\_sock = accept(listener, (struct sockaddr \*) &addr\_client, &clientlen)) < 0) {

printf("Failed to accept client connection \n");

}

fprintf(stdout, "Client connected: %s \n", inet\_ntoa(addr\_client.sin\_addr));

#### Address\_len

## 接收数据

### 函数接口

ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

### 参数详解

## 发送数据

### 函数接口

ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

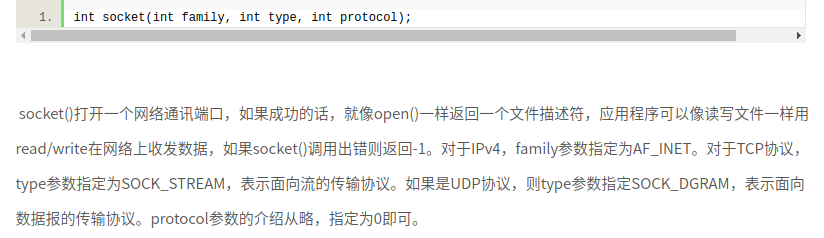
### 参数详解

# 客户端编程

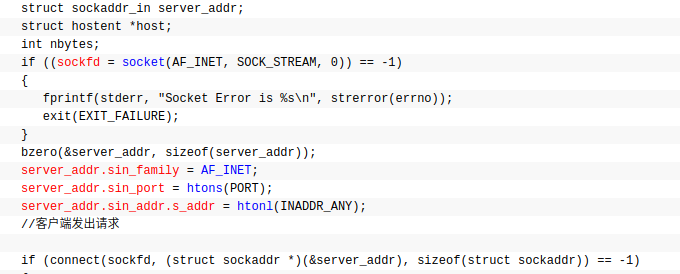
## 参考资料

<https://blog.csdn.net/u011068702/article/details/56678997>

## 创建socket



## 连接服务器



# 重要API

## Linux

### 参考资料

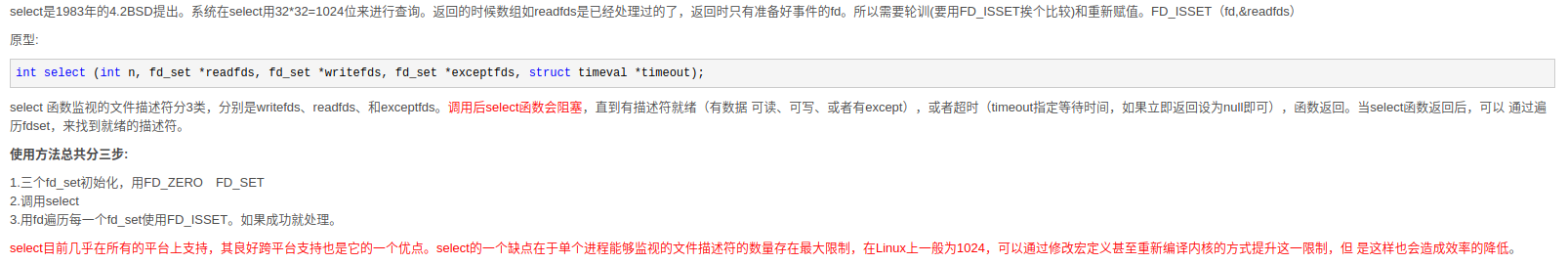
<http://www.cnblogs.com/duanxz/p/5155926.html>

### Select

#### 函数接口

int select (int n, fd\_set \*readfds, fd\_set \*writefds, fd\_set \*exceptfds, struct timeval \*timeout);

#### 详细解释



### Epoll

#### 函数接口



#### 详细解释

#### LT（Level Trigger）

当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序可以不立即处理该事件。下次调用epoll\_wait时，会再次响应应用程序并通知此事件。

LT(level triggered)是缺省的工作方式，并且同时支持block和no-block socket.在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的fd进行IO操作。如果你不作任何操作，内核还是会继续通知你的。

#### ET（Edge Trigger）

##### 详细解释

当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll\_wait时，不会再次响应应用程序并通知此事件。

ET(edge-triggered)是高速工作方式，只支持no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了(比如，你在发送，接收或者接收请求，或者发送接收的数据少于一定量时导致了一个EWOULDBLOCK 错误）。但是请注意，如果一直不对这个fd作IO操作(从而导致它再次变成未就绪)，内核不会发送更多的通知(only once)

ET模式在很大程度上减少了epoll事件被重复触发的次数，因此效率要比LT模式高。epoll工作在ET模式的时候，必须使用非阻塞套接口，以避免由于一个文件句柄的阻塞读/阻塞写操作把处理多个文件描述符的任务饿死。

##### Recv注意



##### 样例代码

#define IPADDRESS "127.0.0.1"

#define PORT 8787

#define MAXSIZE 1024

#define LISTENQ 5

#define FDSIZE 1000

#define EPOLLEVENTS 100

listenfd = socket\_bind(IPADDRESS,PORT);

struct epoll\_event events[EPOLLEVENTS];

//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)

{

int i;

int fd;

//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);

}

}

//添加事件

static void add\_event(int epollfd,int fd,int state){

struct epoll\_event ev;

ev.events = state;

ev.data.fd = fd;

epoll\_ctl(epollfd,EPOLL\_CTL\_ADD,fd,&ev);

}

//处理接收到的连接

static void handle\_accpet(int epollfd,int listenfd){

int clifd;

struct sockaddr\_in cliaddr;

socklen\_t cliaddrlen;

clifd = accept(listenfd,(struct sockaddr\*)&cliaddr,&cliaddrlen);

if (clifd == -1)

perror("accpet error:");

else {

printf("accept a new client: %s:%d\n",inet\_ntoa(cliaddr.sin\_addr),cliaddr.sin\_port); //添加一个客户描述符和事件

add\_event(epollfd,clifd,EPOLLIN);

}

}

//读处理

static void do\_read(int epollfd,int fd,char \*buf){

int nread;

nread = read(fd,buf,MAXSIZE);

if (nread == -1) {

perror("read error:");

close(fd); //记住close fd

delete\_event(epollfd,fd,EPOLLIN); //删除监听

}

else if (nread == 0) {

fprintf(stderr,"client close.\n");

close(fd); //记住close fd

delete\_event(epollfd,fd,EPOLLIN); //删除监听

}

else {

printf("read message is : %s",buf);

//修改描述符对应的事件，由读改为写

modify\_event(epollfd,fd,EPOLLOUT);

}

}

//写处理

static void do\_write(int epollfd,int fd,char \*buf) {

int nwrite;

nwrite = write(fd,buf,strlen(buf));

if (nwrite == -1){

perror("write error:");

close(fd); //记住close fd

delete\_event(epollfd,fd,EPOLLOUT); //删除监听

}else{

modify\_event(epollfd,fd,EPOLLIN);

}

memset(buf,0,MAXSIZE);

}

//删除事件

static void delete\_event(int epollfd,int fd,int state) {

struct epoll\_event ev;

ev.events = state;

ev.data.fd = fd;

epoll\_ctl(epollfd,EPOLL\_CTL\_DEL,fd,&ev);

}

//修改事件

static void modify\_event(int epollfd,int fd,int state){

struct epoll\_event ev;

ev.events = state;

ev.data.fd = fd;

epoll\_ctl(epollfd,EPOLL\_CTL\_MOD,fd,&ev);

}

## Windows

### select

### IOCP

#### 参考资料

<https://www.cnblogs.com/talenth/p/7068392.html>

<https://msdn.microsoft.com/zh-cn/library/windows/desktop/ms737524(v=vs.85).aspx>

#### 函数接口

typedef struct \_OVERLAPPED {

ULONG\_PTR Internal;

ULONG\_PTR InternalHigh;

union {

struct {

DWORD Offset;

DWORD OffsetHigh;

};

PVOID Pointer;

};

HANDLE hEvent;

} OVERLAPPED, \*LPOVERLAPPED;

HANDLE WINAPI CreateIoCompletionPort(

\_In\_ HANDLE FileHandle,

\_In\_opt\_ HANDLE ExistingCompletionPort,

\_In\_ ULONG\_PTR CompletionKey,

\_In\_ DWORD NumberOfConcurrentThreads

);

BOOL WINAPI GetQueuedCompletionStatus(

\_In\_ HANDLE CompletionPort,

\_Out\_ LPDWORD lpNumberOfBytes,

\_Out\_ PULONG\_PTR lpCompletionKey,

\_Out\_ LPOVERLAPPED \*lpOverlapped,

\_In\_ DWORD dwMilliseconds

);

#### 创建完成端口

HANDLE m\_hIOCompletionPort = CreateIoCompletionPort(INVALID\_HANDLE\_VALUE, NULL, 0, 0 );

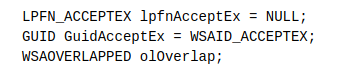
NumberOfConcurrentThreads为0，表示跟CPU个数相同的线程。

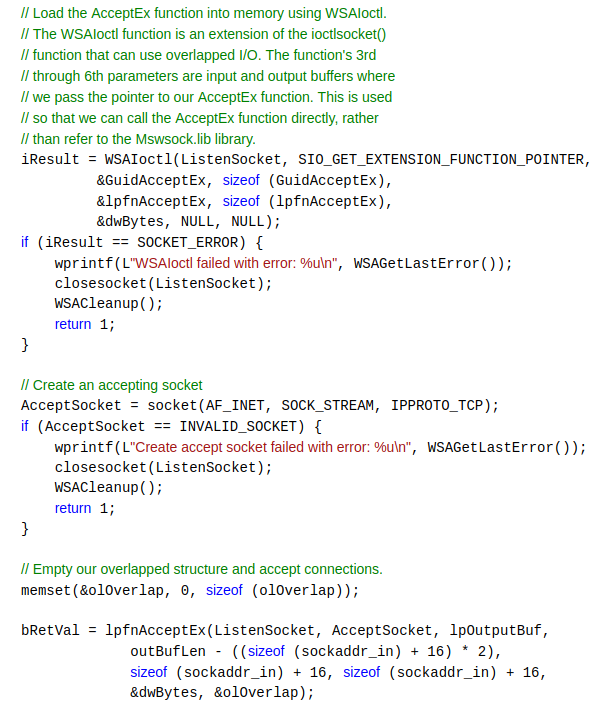
#### 创建socket，并绑定到完成端口上



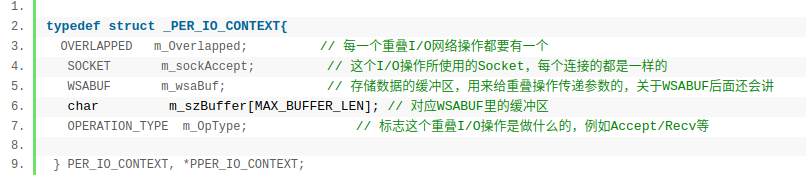
CreateIoCompletionPort((HANDLE) ListenSocket, hCompPort, (u\_long) 0, 0);

#### 监听Socket上投递AcceptEx请求

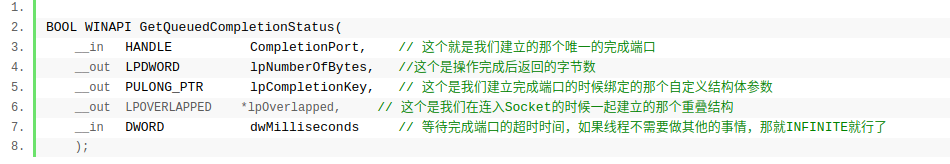


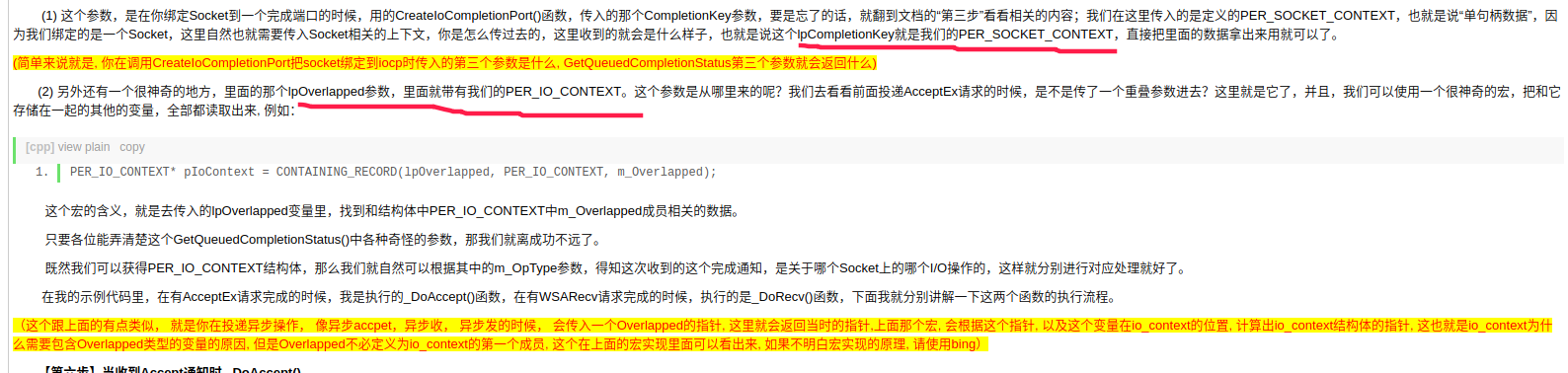


#### Worker线程

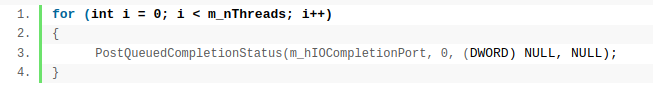


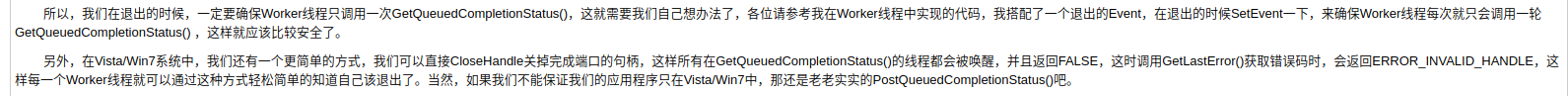




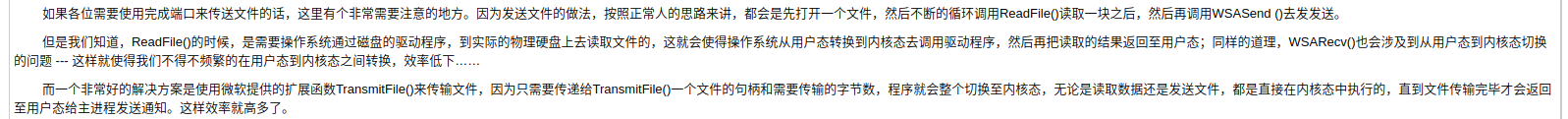


#### 关闭完成端口





#### 传输文件



## Mac

### kqueue