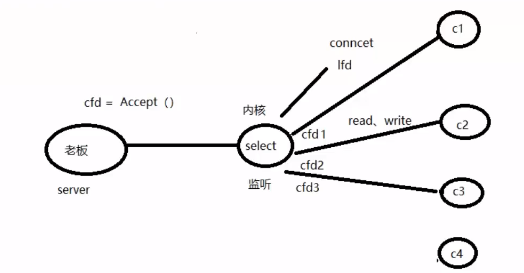
# I/O复用

## 背景

假设有这样的应用场景，服务器一边不停从连接上读数据，一边在侦听连接上的数据。这就是调用一个accept和一个recv，但是在调用accept接受信息的时候会阻塞，怎么能同时recv数据呢？用非阻塞的套接字，这样会耗尽所有的CPU。



注：采用上述方案，只要server调用accept就会接收到数据，因为内核的select已经给判断好存在连接和读写操作了。

可以采用的方案：

1. 多进程
2. 多线程
3. 非阻塞忙轮询：非阻塞IO
4. 响应式：多路IO

### 非阻塞IO

如果在打开文件时设定了O\_NOBLOCK标志，会以阻塞方式打开文件。如果I/O系统调用不能立即完成，则会返回错误而不是阻塞进程（这是与阻塞I/O的本质区别！）。

非阻塞式I/O可以让我们周期性检查（“轮询”）某个文件描述符上是否可执行I/O操作（**非阻塞忙轮询**）。但是，这种轮询通常是我们不希望看到的，如果轮询的频率不高，那么应用程序响应I/O事件的延时可能会达到无法接受的程序。换句话说，**在一个紧凑的循环中做轮询就是在浪费CPU**。

### 多进程

如果不希望进程在对文件描述符执行I/O操作时被阻塞，我们可以创建一个新的进程来执行I/O。此时，父进程可以去处理其他任务，子进程将阻塞直到完成I/O操作。如果需要处理多个文件描述符上的I/O，此时可以为每个文件描述符创建一个子进程。这种方法的问题在于开销昂贵且复杂。创建及维护进程对于系统来说都是有比较大的开销的，而且一般来说子进程需要某种IPC机制来通知父进程有关I/O操作的状态，这无形增加了编程的难度。

### 多线程

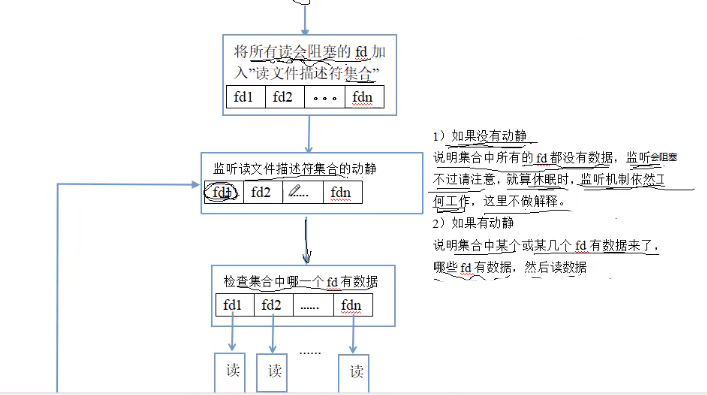
使用多线程而不是多进程，这将占用较少的系统资源。但是线程之间仍然需要通信，以告知其他线程有关I/O操作的状态，这将使得编程变得复杂。尤其是如果我们使用线程池技术来最小化需要处理大量并发客户的线程数量时。（多线程特别有用的一个地方是如果应用程序需要调用一个会执行阻塞式I/O操作的第三方库，那么可以通过在分离的线程中调用这个库从而避免应用阻塞。）而且，多线程涉及到**上下文切换**，代价比较高（**IO多路复用是单线程！**）。

### IO多路复用

select可以让你同时监听多个套接字，使用的基本步骤：

1. 创建套接字，并设置结构体sockaddr\_in的成员变量作为参数；
2. 建立连接；
3. 发送消息并等待消息；
4. 调用多路同步IO select
5. 发送和接收消息
6. 关闭套接字

## 原理



注：在我们自行编写的多线程编程中，是采用while循环判断fd是否被占用，这样用户态判断非常低效，IO多路复用都是将这个fd的判断放在内核态。

多路IO复用的原理：

1. 首先，多路IO会将所有读阻塞的文件描述符fd加入集合中；
2. 多路IO机制会监听读文件描述符集合的动静，查看哪些描述符上面有数据；
3. 检查哪些文件描述符有数据，进行处理。

**注：**如果是阻塞写的话，需要将文件描述符加入写集合，对于99%的情况来说，写操作不会阻塞，所以**一般情况下对于写来说，使用IO多路没有意义**。

对于多路IO来说，**只有操作阻塞的fd才有意义**，如果文件描述符不是阻塞的，使用多路IO没有意义（比如fd1为非阻塞，即不论是否有无数据，监听机制发现该文件描述符fd1非阻塞状态，就会认为它是有动静的，然后去读该fd1上的数据，但是又读不到数据，接着又去监听fd1，继续死循环）。

**IO多路复用的优势：**

多进程实现：**开销太大**，不建议使用。

非阻塞方式：**CPU空转，耗费CPU资源**，不建议使用。

多线程：**线程上下文切换**。

多路IO：使用多路IO时，多路IO机制由于在监听时如果没有动静的话，监听会休眠，因此开销页非常低。相比多进程和非阻塞来说，多路IO是个很好的方式（**IO多路复用是单线程**）。

注：**多路IO不需要多进程、多线程以“多线任务”的方式实现，也不需要用到非阻塞**。

## 应用

下述情况需要使用I/O复用技术：

1. 客户端需要同时处理多个socket
2. 客户端需要同时处理用户输入和网络连接
3. 服务器需要同时处理监听socket和连接socket
4. 服务器需要同时处理TCP请求和UDP请求
5. 服务器需要同时监听多个端口

注：这里的I/O指的是网络套接字socket的I/O。I/O复用虽然能够监听多个端口，但是其本身是阻塞的，如果需要提高效率需要采用多进程多线程技术。

**总结：**



参考资料

[https://devarea.com/linux-io-multiplexing-select-vs-poll-vs-epoll/#.XhrUFvkza1u](https://devarea.com/linux-io-multiplexing-select-vs-poll-vs-epoll/" \l ".XhrUFvkza1u)

<https://notes.shichao.io/unp/ch6/>

<https://www.ulduzsoft.com/2014/01/select-poll-epoll-practical-difference-for-system-architects/>

# select()

## 原型

#include <sys/select.h>

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

**功能：监听**集合中的描述符有没有动静，如果没有动静就阻塞；如果有动静就成功返回，返回值为集合中有动静的fd的数量。

**参数：**

nfds—被监听文件描述符的总数，**通常设置为select监听所有文件描述符最大值+1**（一般从3开始，0/1/2系统输入/输出/错误）。nfds用于说明需要关心描述符的范围，这个范围必须覆盖所有集合中的文件描述符。

注：比如“读集合”中包含0、3、6这3个文件描述符，写集合和异常集合为NULL（没有），此时nfds=6+1（此时读文件描述符最大值为6）。表示监听范围包含7个描述符，描述符从0开始计算，所以监听范围包含的文件描述符为：0、1、2、3、4、5、6。这里集合中只包含了0、3、6这3个fd，但是实际却需要监听7个fd（这是由select实现机制决定的）。

readfds, writefds, exceptfds—可读/可写/异常文件描述符集合，在调用select的时候，通过这3个参数传入自己感兴趣的文件描述符

注：这3个参数叫做传入传出参数，传入的集合是需要监听的文件描述符集合，传出的是实际有事件发生的，比如readfds=3、5、6，实际传出可能只有3、5，select返回值是实际事件总个数。

timeout—设置select函数的超时时间（select调用失败时返回timeout不确定）。

select函数监听集合时，如果没有任何动静就阻塞（休眠）。

1、**如果timeout设置为NULL，select会永远阻塞监听下去，直到被信号中断或者集合中的某些文件描述符有动静了**。

2、如果不想休眠太久的话，就可以设置超时时间，如果时间到了但是集合中的fd没有任何动静，select就返回，然后不再阻塞，超时时的返回值为0。

3、设置timeval里时间均为0，检查描述字后立即返回，轮询。

注：可以通过设置超时时间和readfds, writefds, exceptfds为空，实现精准的定时操作。

可以访问fd\_set结构体（**bitmap类型**）的函数：

FD\_ZERO(fd\_set\* fdset); // 清除fdset的所有位

FD\_SET(int fd,fd\_set\* fdset); // 设置fdset的位fd

FD\_CLR(int fd,fd\_set\* fdset); // 清除fdset的位fd（用的不多）

int FD\_ISSET(int fd,fd\_set\* fdset); // 测试fdset的位fd是否被设置

注：**fd\_set其实就是一个bitmap的结构，大小为1024**。

struct timeval结构体：

struct timeval{

long tv\_sec;//秒数

long tv\_usec;//微秒数

};

如果将两个参数都设置为0，则select立即返回；如果设置为NULL，则select会阻塞。

**返回值：**

1. -1：说明函数调用失败，errno被设置

Select调用出错的情况有很多，比如select在阻塞时被信号唤醒从而导致出错返回，errno被设置为EINTR错误号，这个错误号表示函数是被信号中断而出错返回的。

如果不想被信号中断：可以忽略、屏蔽这些信号；手动重启select的调用。

1. 0：说明超时时间到并且集合中没有一个文件描述符有响应
2. >0：**集合中fd有动静时，函数返回有动静的文件描述符的数量**。

**socket可读的条件：**

1. socket接收缓冲区中字节大于等于低水位标记
2. socket对方关闭连接（应该返回0）
3. 监听socket上有新的连接
4. socket上有待处理的错误

**socket可写的条件：**

1. socket中发送缓冲区字节大于低水位
2. socket对方写操作关闭
3. socket上使用非阻塞connect连接成功或失败之后
4. socket上有未处理的错误

## 原理

借助内核，select来监听，客户端连接、数据通信事件。



执行select的过程：

1. select函数将**bitmap**从**用户态拷贝到内核态**，交给内核去判断每个fd是否有数据（内核处理比用户态快，用户态的判断其实也是用户判断后交给内核再次判断，这样会有用户态和内核态的上下文切换）；
2. 如果监听fd没有数据，则select处于阻塞状态；
3. 如果监听fd有数据：

**3.1 内核**会将**有数据的fd置位**（标识为有数据到来）；

3.2 select会返回（不再阻塞）；

3.3 读取fd上的数据，进行相应的处理。

**传入的是需要监听的fd，传出的是真正有数据的fd（传出的已经被修改了）。**

注：上述将fd的bitmap结构置位，导致bitmap直接被破坏，不能继续重用，导致后面查找具体哪一个bitmap中被置位时需要再次遍历bitmap结构。

## 特点

Select将轮询工作交给内核处理，省去内核态和用户态的切换，提高效率。

### 优点

Select()调用的优点：

1. 跨平台
2. select对超时值提供了很好的精度，精确到微秒，而poll是毫秒。

### 缺点

select()调用的缺点：

1. **单个进程可监视的fd数量被限制**（fd\_set仅包含一个整型数组，这个整型数组的每一位标记一个文件描述符fd\_set能容纳的文件描述符数量是由一个宏决定的）；

注：bitmap中有1024位，监听的设置为1，不监听的设置为0，不管监听1个还是1024个fd都需要使用1024那么大，有时候有点浪费。

1. 对socket采用**轮询**的方法；
2. 需要维护一个用来存放大量fd的数据结构（**fd不可重用**，每次都需要创建一个bitmap，都需要调用FD\_ZERO、FD\_SET初始化）；
3. FD\_SET拷贝到内核态，**用户态和内核态切换仍有较大的开销**；
4. **select返回时知道FD\_SET有一个bitmap置位，但是不知道哪一/几位置位，需要再次遍历bitmap**，时间复杂度为O(n)。

## 实例

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <assert.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <stdlib.h>

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port num\r\n");

exit(1);

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

int ret = 0;

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int listenfd = socket(AF\_INET,SOCK\_STREAM,0);

assert(listenfd >= 0);

ret = bind(listenfd,(struct sockaddr\*)&address,sizeof(address));

assert(ret != -1);

ret = listen(listenfd,5);

assert(ret != -1);

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

if(connfd < 0)

{

printf("accept error\r\n");

exit(1);

}

char buf[1024];

fd\_set read\_fds;

fd\_set exceptions\_fds;

FD\_ZERO(&read\_fds);

FD\_ZERO(&exceptions\_fds);

while(1)

{

memset(buf,0x00,sizeof(buf));

FD\_SET(connfd,&read\_fds);

FD\_SET(connfd,&exceptions\_fds);

ret = select(connfd+1,&read\_fds,NULL,&exceptions\_fds,NULL);

if(ret < 0)

{

printf("select error\r\n");

exit(1);

}

if(FD\_ISSET(connfd,&read\_fds))//判断是否有事件

{

ret = recv(connfd,buf,sizeof(buf)-1,0);

if(ret <= 0)

{

break;

}

printf("recv data form %d and buf is %s\r\n",connfd,buf);

}

else if(FD\_ISSET(connfd,&exceptions\_fds))

{

ret = recv(connfd,buf,sizeof(buf)-1,MSG\_OOB);

if(ret <=0 )

{

break;

}

printf("recv oob data fomr %d and buf is %s\r\n",connfd,buf);

}

}

close(connfd);

close(listenfd);

return 0;

}

# poll()

## 原型

#include <poll.h>

int poll(struct pollfd\* fds,nfds\_t nfds,int timeout);

**参数说明：**

nfds：监听数组的实际有效监听个数。

timeout：超时时长（>0：超时时长；-1：阻塞等待；0：不阻塞）。

**返回值：**返回满足对应监听事件的文件描述符总个数。

struct pollfd结构体：

struct pollfd{

int fd; //待监听的文件描述符

**short events; //注册的事件（关心的事件），待监听的文件描述符对应的监听事件**

**short revents;//实际发生的事件/返回的事件，由内核填充**

};

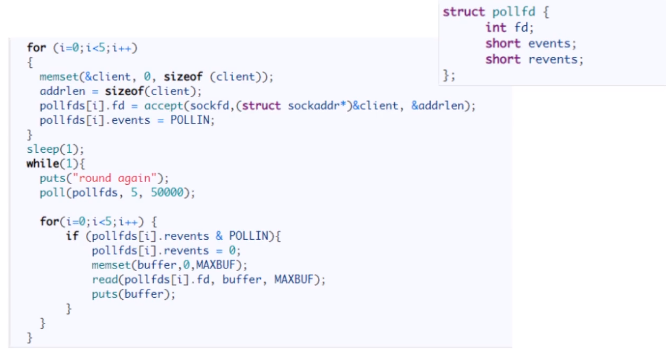
注：这里与select的区别在于不是bitmap维护的fd。

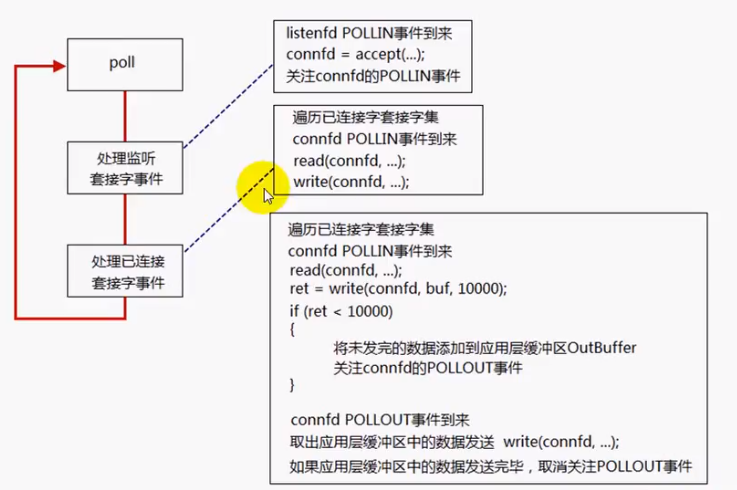
**poll事件类型：**





## 原理





poll的基本原理跟select一致：

1. 用户态维护一个fd（不是bitmap维护），拷贝到内核态，交给内核监听fd上的数据；
2. 如果监听fd没有数据，则**poll处于阻塞状态**；
3. 如果监听fd有数据：

3.1 内核会将有数据的pollfd中的**revents置位**；

3.2 poll返回；

3.3 判断revents是否被置位，进行数据读写；

**3.4将revent恢复成原来的状态，可以继续重用**。

## 特点

### 优点

poll解决了select的诸多缺点：

1. 不要求计算最大文件描述符+1的大小。
2. 应付大数量的文件描述符时比select要快。
3. **revents使用自带数组**（将监听事件集合和返回事件集合分离），解决了select监听事件1024的大小限制（**没有最大连接数的限制是基于链表存储的**）。
4. **fd可以重用（内核态判断后置位revent）。**

### 缺点

1、大量的fd数组被整体复制于内核态和用户态之间，即**需要用户态和内核态的切换，而不管这样的复制是不是有意义**。

2、同select相同的是调用结束后需要**轮询**来获取就绪描述符，需要O(n)时间复杂度。

## 实例

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <assert.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <stdlib.h>

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port num\r\n");

exit(1);

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

int ret = 0;

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int listenfd = socket(AF\_INET,SOCK\_STREAM,0);

assert(listenfd >= 0);

ret = bind(listenfd,(struct sockaddr\*)&address,sizeof(address));

assert(ret != -1);

ret = listen(listenfd,5);

assert(ret != -1);

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

if(connfd < 0)

{

printf("accept error\r\n");

exit(1);

}

char buf[1024];

fd\_set read\_fds;

fd\_set exceptions\_fds;

FD\_ZERO(&read\_fds);

FD\_ZERO(&exceptions\_fds);

while(1)

{

memset(buf,0x00,sizeof(buf));

FD\_SET(connfd,&read\_fds);

FD\_SET(connfd,&exceptions\_fds);

ret = select(connfd+1,&read\_fds,NULL,&exceptions\_fds,NULL);

if(ret < 0)

{

printf("select error\r\n");

exit(1);

}

if(FD\_ISSET(connfd,&read\_fds))

{

ret = recv(connfd,buf,sizeof(buf)-1,0);

if(ret <= 0)

{

break;

}

printf("recv data form %d and buf is %s\r\n",connfd,buf);

}

else if(FD\_ISSET(connfd,&exceptions\_fds))

{

ret = recv(connfd,buf,sizeof(buf)-1,MSG\_OOB);

if(ret <=0 )

{

break;

}

printf("recv oob data fomr %d and buf is %s\r\n",connfd,buf);

}

}

close(connfd);

close(listenfd);

return 0;

}

# epoll()

epoll是Linux内核中的一种可扩展IO事件处理机制，最早在Linux 2.5.44内核中引入，可被用于代替POSIX select和poll 系统调用，并且在具有大量应用程序请求时能够获得较好的性能（此时被监视的文件描述符数目非常大，与旧的select和poll系统调用完成操作所需O(n)不同，epoll能在O(1)时间内完成操作，所以性能相当高），epoll与FreeBSD的kqueue类似，都向用户空间提供了自己的文件描述符来进行操作。

epoll是之前的select和poll的增强版本。相对于select和poll来说，epoll更加灵活，**没有描述符限制**。epoll使用一个文件描述符管理多个描述符，将用户关系的文件描述符的事件存放到内核的一个事件表中，这样**在用户空间和内核空间的copy只需一次**。

## 原型

### epoll\_create

#include<sys/epoll.h

int epoll\_create(int size);

**功能：**该函数生成一个epoll专用的文件描述符（创建一个epoll的句柄）。

**参数：**

size：用来告诉内核这个监听的数目一共有多大，参数size并不是限制了epoll所能监听的描述符最大个数，只是对内核初始分配内部数据结构的一个建议。

自从 linux 2.6.8 之后，size参数是被忽略的，也就是说可以填只有大于0的任意值。需要注意的是，当创建好epoll句柄后，它就是会占用一个fd值，在linux下如果查看/proc/进程id/fd/，是能够看到这个fd的，所以在使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

**返回值：**

成功：epoll 专用的文件描述符；

失败：-1。

### epoll\_ctl

#include<sys/epoll.h

int epoll\_ctl(int epfd,int op,int fd,struct epoll\_event\* event);

**功能：**epoll的事件注册函数，它不同于select()是在监听事件时告诉内核要监听什么类型的事件，而是在这里先注册要监听的事件类型。

**参数：**

epfd：epoll专用的文件描述符，epoll\_create()的返回值。

op参数的类型（对该监听红黑树结构的操作）：



fd：需要监听的文件描述符；

event：告诉内核要监听什么事件；

注：epoll是个事件表，这是放置在内核中的，我们只需要调用系统函数往这个事件表上注册、修改、删除关注的套接字描述符即可，不像select和poll系统调用需要往集合上注册，而且注册到集合上的文件描述符需要在内核态和用户态来回切换。

struct epoll\_event结构体：

struct epoll\_event

{

\_\_uint32\_t events; //epoll事件表（事件类型）

epoll\_data\_t data; //用户数据

};

**返回值：**成功：0，失败：-1。

### epoll\_wait

**原型：**

#include<sys/epoll.h

int epoll\_wait(int epfd,struct epoll\_event\* events,int maxevents,int

timeout);

**作用：**在一段超时时间内，等待一组文件描述符上的事件。收集在epoll监控的事件中已经发送的事件，类似于select调用。

**epoll\_wait监测到事件是对数据进行重排序（红黑树），最后只需要遍历前几个数据即可，而不是select和poll中的数组置位操作！**

**参数：**

epfd：epoll专用的文件描述符，epoll\_create()的返回值；

events：分配好的epoll\_event结构体数组，**epoll将会把发生的事件赋值到events数组中**（events不可以是空指针，内核只负责把数据复制到这个events数组中，不会去帮助我们在用户态中分配内存）；

maxevents：告知内核这个events有多大；

timeout：超时时间，单位为毫秒，为-1时，函数为阻塞。

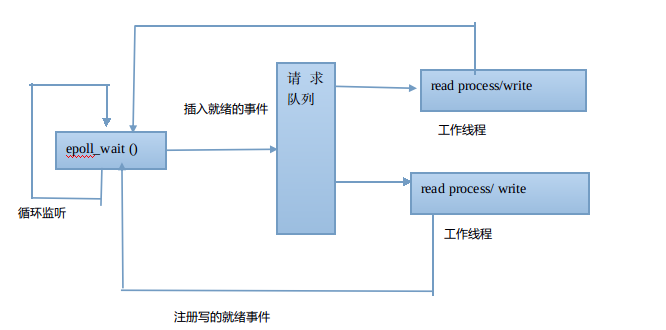
**返回值：**成功返回事件数，失败-1

**说明：**epoll如果监测到事件，就把该事件从内核的事件表中复制到第二个参数event指向的数组中（这里只拷贝就绪事件，而select/poll拷贝的是所有的事件）。

## 原理

### Reactor模型

IO模型实现reactor 模式的工作流程：



1、主线程向epoll内核事件表内注册socket上的可读就绪事件；

2、主线程调用epoll\_wait()等待socket上有数据可读；

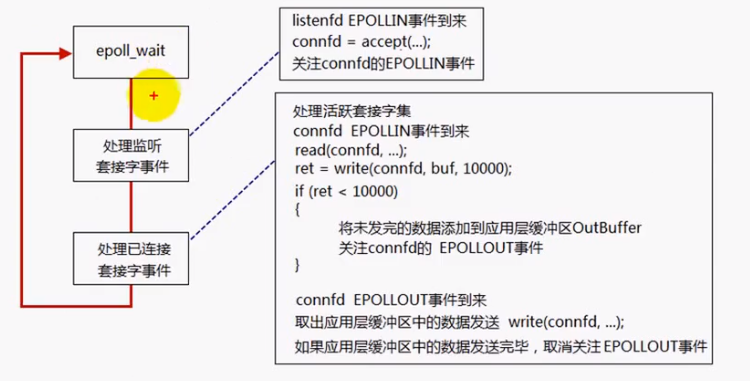
3、当socket上有数据可读，epoll\_wait 通知主线程。主线程从socket可读事件放入请求队列。

4、睡眠在请求队列上的某个可读工作线程被唤醒，从socket上读取数据，处理客户的请求。

然后向 epoll内核事件表里注册写的就绪事件

1. 主线程调用epoll\_wait()等待数据可写。

**epoll反应堆模型**，是libevent核心思想的实现。



从事服务端开发，少不了要接触网络编程。epoll 作为 Linux 下高性能网络服务器的必备技术至关重要，nginx、Redis、Skynet 和大部分游戏服务器都使用到这一多路复用技术。

epoll 很重要，但是 epoll 与 select 的区别是什么呢？epoll 高效的原因是什么？

**epoll原理与select/poll基本类似，但是没有置位的操作：**

1. 监听epfd（这个**epfd是用户态和内核态共享的**，不需要切换）；
2. 如果epfd没有数据，则epoll处于阻塞状态；
3. 如果epfd监听到数据：

3.1 对监听的fd进行**重排**（红黑树的旋转，而不是像select那样置位）；

3.2 epoll\_wait返回，返回值n就是触发事件的fd个数（最后遍历返回值n个事件即可，遍历复杂度O(1)）。

注：**这样就解决了select和poll中未能解决的问题：用户态和内核态拷贝，遍历复杂度O(n)**。

这里监听到对应的事件，即epfd有数据的时候，就会对存储epfd的结构，即红黑树进行旋转，将最新监听的epfd置于红黑树顶端，这样读取的时候只需要读取红黑树顶部节点，而不需要采用select那样效率低下的置位操作。如果最后有3个epfd有数据，最后遍历3个即可，不需要像select那样遍历bitmap的所有元素，**遍历的时间复杂度O(1)**。

<https://www.cnblogs.com/Joy-Hu/p/10762239.html>

### 网卡接收数据

下边是一个典型的计算机结构图，计算机由 CPU、存储器（内存）与网络接口等部件组成，了解 epoll 本质的第一步，要从硬件的角度看计算机怎样接收网络数据。



计算机结构图（图片来源：Linux内核完全注释之微型计算机组成结构）

下图展示了网卡接收数据的过程。

在 ① 阶段，网卡收到网线传来的数据；

经过 ② 阶段的硬件电路的传输；

最终 ③ 阶段将数据写入到内存中的某个地址上。

这个过程涉及到 DMA 传输、IO 通路选择等硬件有关的知识，但我们只需知道：网卡会把接收到的数据写入内存。



网卡接收数据的过程

通过硬件传输，网卡接收的数据存放到内存中，操作系统就可以去读取它们。

### 如何知道接收了数据？

了解 epoll 本质的第二步，要从 CPU 的角度来看数据接收。理解这个问题，要先了解一个概念——中断。

计算机执行程序时，会有优先级的需求。比如，当计算机收到断电信号时，它应立即去保存数据，保存数据的程序具有较高的优先级（电容可以保存少许电量，供 CPU 运行很短的一小段时间）。

一般而言，由硬件产生的信号需要 CPU 立马做出回应，不然数据可能就丢失了，所以它的优先级很高。CPU 理应中断掉正在执行的程序，去做出响应；当 CPU 完成对硬件的响应后，再重新执行用户程序。中断的过程如下图，它和函数调用差不多，只不过函数调用是事先定好位置，而中断的位置由“信号”决定。



中断程序调用

以键盘为例，当用户按下键盘某个按键时，键盘会给 CPU 的中断引脚发出一个高电平，CPU 能够捕获这个信号，然后执行键盘中断程序。下图展示了各种硬件通过中断与 CPU 交互的过程。



CPU 中断（图片来源：net.pku.edu.cn）

现在可以回答“如何知道接收了数据？”这个问题了：当网卡把数据写入到内存后，网卡向 CPU 发出一个中断信号，操作系统便能得知有新数据到来，再通过网卡中断程序去处理数据。

### 进程阻塞为什么不占用 CPU 资源？

了解epoll本质的第三步，要从操作系统进程调度的角度来看数据接收。阻塞是进程调度的关键一环，指的是进程在等待某事件（如接收到网络数据）发生之前的等待状态，recv、select和epoll都是阻塞方法。下边分析一下进程阻塞为什么不占用 CPU 资源？

为简单起见，我们从普通的 recv 接收开始分析，先看看下面代码：

//创建socket

int s = socket(AF\_INET, SOCK\_STREAM, 0);

//绑定

bind(s, ...)

//监听

listen(s, ...)

//接受客户端连接

int c = accept(s, ...)

//接收客户端数据

recv(c, ...);

//将数据打印出来

printf(...)

这是一段最基础的网络编程代码，先新建 socket 对象，依次调用 bind、listen 与 accept，最后调用 recv 接收数据。recv 是个阻塞方法，当程序运行到 recv 时，它会一直等待，直到接收到数据才往下执行。

那么阻塞的原理是什么？

#### 工作队列

操作系统为了支持多任务，实现了进程调度的功能，会把进程分为“运行”和“等待”等几种状态。运行状态是进程获得CPU使用权，正在执行代码的状态；等待状态是阻塞状态，比如上述程序运行到recv时，程序会从运行状态变为等待状态，接收到数据后又变回运行状态。操作系统会分时执行各个运行状态的进程，由于速度很快，看上去就像是同时执行多个任务。

下图的计算机中运行着 A、B与C三个进程，其中进程A执行着上述基础网络程序，一开始，这3个进程都被操作系统的工作队列所引用，处于运行状态，会分时执行。



工作队列中有 A、B 和 C 三个进程

#### 等待队列

当进程A执行到创建socket的语句时，操作系统会创建一个由文件系统管理的socket对象（如下图）。这个socket对象包含了发送缓冲区、接收缓冲区与等待队列等成员。等待队列是个非常重要的结构，它指向所有需要等待该socket事件的进程。



创建 socket

当程序执行到recv时，操作系统会将进程 A从工作队列移动到该socket的等待队列中（如下图）。由于工作队列只剩下了进程B和C，依据进程调度，CPU会轮流执行这两个进程的程序，不会执行进程A的程序。所以进程A被阻塞，不会往下执行代码，也不会占用CPU资源。



**socket的等待队列**

注：操作系统添加等待队列只是添加了对这个“等待中”进程的引用，以便在接收到数据时获取进程对象、将其唤醒，而非直接将进程管理纳入自己之下。上图为了方便说明，直接将进程挂到等待队列之下。

#### 唤醒进程

当socket接收到数据后，操作系统将该socket等待队列上的进程重新放回到工作队列，该进程变成运行状态，继续执行代码。同时由于socket的接收缓冲区已经有了数据，recv可以返回接收到的数据。

### 内核接收网络数据全过程

这一步，贯穿网卡、中断与进程调度的知识，叙述阻塞 recv下，内核接收数据的全过程。

如下图所示，进程在recv阻塞期间，计算机收到了对端传送的数据（步骤①），数据经由网卡传送到内存（步骤②），然后网卡通过中断信号通知CPU有数据到达，CPU执行中断程序（步骤③）。

此处的中断程序主要有两项功能，先将网络数据写入到对应 socket 的接收缓冲区里面（步骤④），再唤醒进程 A（步骤⑤），重新将进程 A 放入工作队列中。



内核接收数据全过程

唤醒进程的过程如下图所示：



**唤醒进程**

以上是内核接收数据全过程，这里我们可能会思考两个问题：

其一，操作系统如何知道网络数据对应于哪个 socket？

其二，如何同时监视多个 socket 的数据？

第一个问题：因为一个 socket 对应着一个端口号，而网络数据包中包含了 ip 和端口的信息，内核可以通过端口号找到对应的 socket。当然，为了提高处理速度，操作系统会维护端口号到 socket 的索引结构，以快速读取。

第二个问题是多路复用的重中之重，也正是本文后半部分的重点。

### 同时监视多个socket的简单方法

服务端需要管理多个客户端连接，而recv只能监视单个socket，这种矛盾下，人们开始寻找监视多个socket 的方法。epoll的要义就是高效地监视多个socket。

从历史发展角度看，必然先出现一种不太高效的方法，人们再加以改进，正如 select 之于epoll。

先理解不太高效的select，才能够更好地理解epoll的本质。

假如能够预先传入一个socket列表，如果列表中的socket都没有数据，挂起进程，直到有一个 socket 收到数据，唤醒进程。这种方法很直接，也是 select 的设计思想。

为方便理解，我们先复习select的用法。在下边的代码中，先准备一个数组 fds，让 fds 存放着所有需要监视的socket。然后调用 select，如果 fds 中的所有 socket 都没有数据，select 会阻塞，直到有一个 socket 接收到数据，select 返回，唤醒进程。用户可以遍历 fds，通过 FD\_ISSET 判断具体哪个 socket 收到数据，然后做出处理。

int s = socket(AF\_INET, SOCK\_STREAM, 0);

bind(s, ...);

listen(s, ...);

int fds[] = 存放需要监听的socket;

while(1){

int n = select(..., fds, ...)

for(int i=0; i < fds.count; i++){

if(FD\_ISSET(fds[i], ...)){

//fds[i]的数据处理

}

}}

**select的流程**

select 的实现思路很直接，假如程序同时监视如下图的 sock1、sock2 和 sock3 三个 socket，那么在调用 select 之后，操作系统把进程A分别加入这三个socket的等待队列中。



操作系统把进程 A 分别加入这三个 socket 的等待队列中

当任何一个 socket 收到数据后，中断程序将唤起进程。下图展示了 sock2 接收到了数据的处理流程：

注：recv 和 select 的中断回调可以设置成不同的内容。



sock2 接收到了数据，中断程序唤起进程 A

所谓唤起进程，就是将进程从所有的等待队列中移除，加入到工作队列里面，如下图所示：



将进程 A 从所有等待队列中移除，再加入到工作队列里面

经由这些步骤，当进程 A 被唤醒后，它知道至少有一个 socket 接收了数据。程序只需遍历一遍 socket 列表，就可以得到就绪的 socket。

这种简单方式行之有效，在几乎所有操作系统都有对应的实现。

但是简单的方法往往有缺点，主要是：

其一，每次调用select都需要将进程加入到所有监视 socket 的等待队列，每次唤醒都需要从每个队列中移除。这里涉及了两次遍历，而且每次都要将整个fds列表传递给内核，有一定的开销。正是因为遍历操作开销大，出于效率的考量，才会规定 select 的最大监视数量，默认只能监视 1024 个 socket。

其二，进程被唤醒后，程序并不知道哪些 socket 收到数据，还需要遍历一次。

那么，有没有减少遍历的方法？有没有保存就绪 socket 的方法？这两个问题便是 epoll 技术要解决的。

补充说明： 本节只解释了 select 的一种情形。当程序调用 select 时，内核会先遍历一遍 socket，如果有一个以上的 socket 接收缓冲区有数据，那么 select 直接返回，不会阻塞。这也是为什么 select 的返回值有可能大于 1 的原因之一。如果没有 socket 有数据，进程才会阻塞。

### epoll的设计思路

epoll 是在 select 出现 N 多年后才被发明的，是 select 和 poll（poll 和 select 基本一样，有少量改进）的增强版本。epoll 通过以下一些措施来改进效率：

**措施一：功能分离**

select 低效的原因之一是将“维护等待队列”和“阻塞进程”两个步骤合二为一。如下图所示，每次调用 select 都需要这两步操作，然而大多数应用场景中，需要监视的 socket 相对固定，并不需要每次都修改。epoll 将这两个操作分开，先用 epoll\_ctl 维护等待队列，再调用 epoll\_wait 阻塞进程。显而易见地，效率就能得到提升。



相比 select，epoll 拆分了功能

为方便理解后续的内容，我们先了解一下 epoll 的用法。如下的代码中，先用epoll\_create创建一个epoll对象epfd（红黑树），再通过epoll\_ctl将需要监视的socket添加到epfd中，最后调用epoll\_wait等待数据：

int s = socket(AF\_INET, SOCK\_STREAM, 0);

bind(s, ...)

listen(s, ...)

int epfd = epoll\_create(...);//构造红黑树结构

epoll\_ctl(epfd, ...); //将所有需要监听的socket添加到epfd中，设置红黑树结构

while(1){

int n = epoll\_wait(...)

for(接收到数据的socket){

//处理

}

}

功能分离，使得 epoll 有了优化的可能。

**措施二：就绪列表**

select 低效的另一个原因在于程序不知道哪些 socket 收到数据，只能一个个遍历。如果内核维护一个“就绪列表”，引用收到数据的 socket，就能避免遍历。如下图所示，计算机共有三个 socket，收到数据的 sock2 和 sock3 被就绪列表 rdlist 所引用。当进程被唤醒后，只要获取 rdlist 的内容，就能够知道哪些 socket 收到数据。



就绪列表示意图

### epoll的原理与工作流程



epoll 的原理和工作流程：

#### 创建epoll对象

如下图所示，当某个进程调用 epoll\_create 方法时，内核会创建一个 eventpoll 对象（也就是程序中 epfd 所代表的对象）。eventpoll 对象也是文件系统中的一员，和 socket 一样，它也会有等待队列。

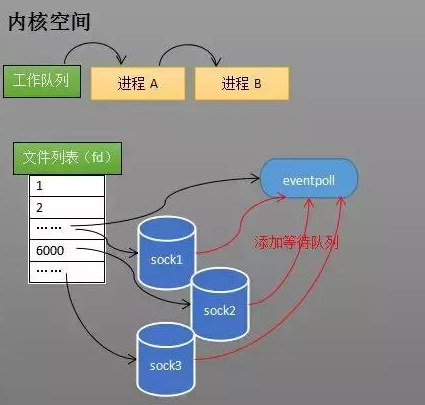


内核创建 eventpoll 对象

创建一个代表该 epoll 的 eventpoll 对象是必须的，因为内核要维护“就绪列表”等数据，“就绪列表”可以作为 eventpoll 的成员。

#### 维护监视列表

创建 epoll 对象后，可以用 **epoll\_ctl 添加或删除所要监听的 socket**。以添加 socket 为例，如下图，如果通过 epoll\_ctl 添加 sock1、sock2 和 sock3 的监视，内核会将 eventpoll 添加到这三个 socket 的等待队列中。

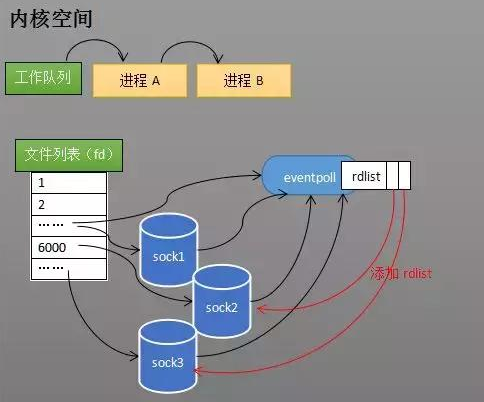


#### 添加所要监听的 socket

当 socket 收到数据后，中断程序会操作 eventpoll 对象，而不是直接操作进程。

#### 接收数据

当 socket 收到数据后，中断程序会给 eventpoll 的“就绪列表”添加 socket 引用。如下图展示的是 sock2 和 sock3 收到数据后，中断程序让 rdlist 引用这两个 socket。



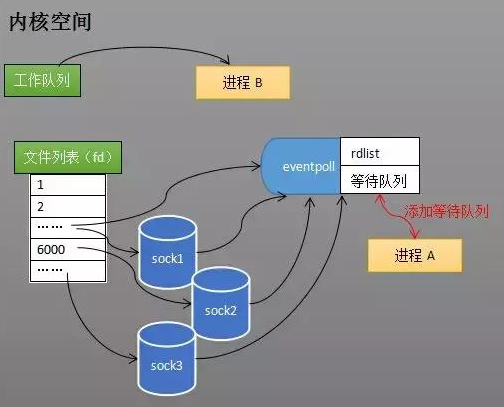
#### 给就绪列表添加引用

eventpoll对象相当于socket和进程之间的中介，socket的数据接收并不直接影响进程，而是通过改变eventpoll的就绪列表来改变进程状态。

当程序执行到epoll\_wait时，如果rdlist已经引用了socket，那么epoll\_wait直接返回，如果rdlist为空，阻塞进程。

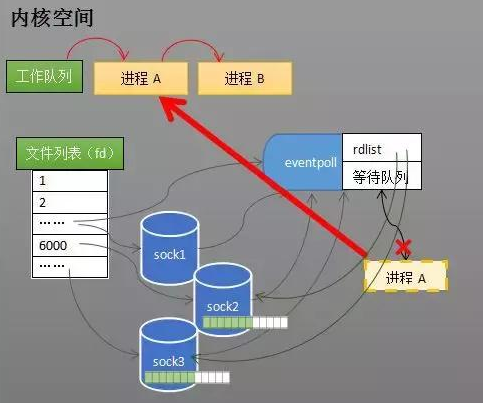
#### 阻塞和唤醒进程

假设计算机中正在运行进程 A 和进程B，在某时刻进程 A 运行到了 epoll\_wait 语句。如下图所示，内核会将进程A放入 eventpoll 的等待队列中，阻塞进程。



epoll\_wait 阻塞进程

当 socket 接收到数据，中断程序一方面修改 rdlist，另一方面唤醒 eventpoll 等待队列中的进程，进程 A 再次进入运行状态（如下图）。也因为 rdlist 的存在，进程 A 可以知道哪些 socket 发生了变化。



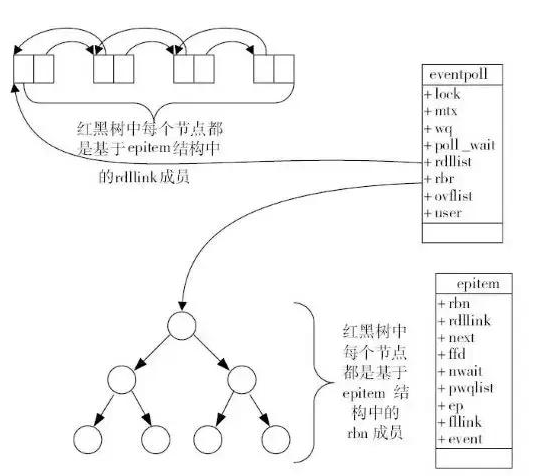
epoll 唤醒进程

### epoll的实现细节

至此，相信读者对epoll的本质已经有一定的了解。但我们还需要知道 eventpoll 的数据结构是什么样子？

此外，就绪队列应该应使用什么数据结构？eventpoll应使用什么数据结构来管理通过epoll\_ctl添加或删除的 socket？

如下图所示，eventpoll包含了lock、mtx、wq（等待队列）与rdlist等成员，其中rdlist和rbr是我们所关心的。



epoll原理示意图，图片来源：《深入理解Nginx：模块开发与架构解析(第二版)》，陶辉

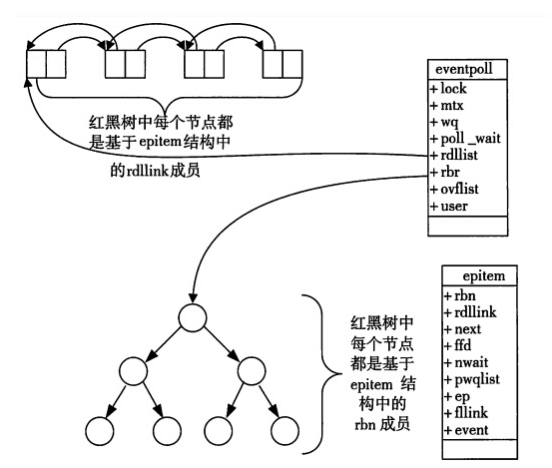
#### 就绪列表的数据结构/双向链表

就绪列表引用着就绪的socket，所以它应能够快速的插入数据。

程序可能随时调用epoll\_ctl 添加监视socket，也可能随时删除。当删除时，若该socket已经存放在就绪列表中，它也应该被移除。所以就绪列表应是一种能够快速插入和删除的数据结构。

双向链表就是这样一种数据结构，epoll使用双向链表来实现**就绪队列**（对应上图的rdllist）。

**通过红黑树和双链表数据结构，并结合回调机制，造就了epoll的高效。**



#### 索引结构/红黑树

既然epoll 将“维护监视队列”和“进程阻塞”分离，也意味着需要有个数据结构来保存监视的socket，至少要方便地添加和移除，还要便于搜索，以避免重复添加。红黑树是一种自平衡二叉查找树，搜索、插入和删除时间复杂度都是O(log(N))，效率较好，**epoll 使用了红黑树作为索引结构**（对应上图的rbr）。

注：因为操作系统要兼顾多种功能，以及由更多需要保存的数据，rdlist并非直接引用socket，而是通过epitem间接引用，红黑树的节点也是epitem对象。同样，文件系统也并非直接引用着socket。为方便理解，本文中省略了一些间接结构。



注：epoll之所以那么高效，因为在内核中epoll是通过链表和红黑树来实现的，红黑树的作用就是事件表，链表就是返回的就绪事件的文件描述符。

## 特点

### 优点

1. 用户态和内核态**共享epfd**，不存在用户态和内核态的拷贝。

select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，而epoll只要一次拷贝，这也能节省不少的开销。

2、epoll在select和poll的基础上引入了eventpoll作为中间层，使用了先进的数据结构，是一种高效的多路复用技术。

3、突破1024描述符的限制。监视的描述符数量不受限制，它所支持的fd上限是最大可以打开文件的数目，这个数字一般远大于1024，举个例子，在1GB内存的机器上大约是10万左右，具体数目可以cat /proc/sys/fs/file-max察看,一般来说这个数目和系统内存关系很大。select的最大缺点就是进程打开的fd是有数量限制的。这对于连接数量比较大的服务器来说根本不能满足。虽然也可以选择多进程的解决方案(Apache就是这样实现的)，不过虽然Linux上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。

4、因为epoll内核中实现时根据每个fd上的callback函数来实现的，只有活跃的socket才会主动调用callback，所以在活跃socket较少的情况下，使用epoll没有前面两者的线性下降的性能问题，但是所有socket都很活跃的情况下，可能会有性能问题。

I/O的效率不会随着监视fd的数量的增长而下降。select，poll实现需要自己不断轮询所有fd集合，直到设备就绪，期间可能要睡眠和唤醒多次交替。而epoll其实也需要调用epoll\_wait()不断轮询就绪链表，期间也可能多次睡眠和唤醒交替，但是它是设备就绪时，调用回调函数，把就绪fd放入就绪链表中，并唤醒在epoll\_wait()中进入睡眠的进程。虽然都要睡眠和交替，但是select和poll在“醒着”的时候要遍历整个fd集合，而epoll在“醒着”的时候只要判断一下就绪链表是否为空就行了，这节省了大量的CPU时间。这就是回调机制带来的性能提升。

epoll基于事件驱动，epoll\_ctl注册事件并注册callback回调函数，epoll\_wait只返回发生的事件避免了像select和poll对事件的整个轮询操作。

### 缺点

不能跨平台。

## 工作模式

从man手册中，得到ET和LT的具体描述如下

EPOLL事件有两种模型：

Edge Triggered(ET)

//高速工作方式，错误率比较大，只支持no\_block socket (非阻塞socket)

LevelTriggered(LT)

//缺省工作方式，支持blocksocket和no\_blocksocket，错误率比较小。

假如有这样一个例子：(LT方式，即默认方式下，内核会继续通知，可以读数据，ET方式，内核不会再通知，可以读数据)

1、我们已经把一个用来从管道中读取数据的文件句柄(RFD)添加到epoll描述符

2、这个时候从管道的另一端被写入了2KB的数据

3、调用epoll\_wait(2)，并且它会返回RFD，说明它已经准备好读取操作

4、然后我们读取了1KB的数据

5、调用epoll\_wait(2)......

### 边沿触发

**Edge Triggered工作模式：**

如果我们在第1步将RFD添加到epoll描述符的时候使用了EPOLLET标志，那么在第5步调用epoll\_wait(2)之后将有可能会挂起，因为剩余的数据还存在于文件的输入缓冲区内，而且数据发出端还在等待一个针对已经发出数据的反馈信息。只有在监视的文件句柄上发生了某个事件的时候ET工作模式才会汇报事件。因此在第5步的时候，调用者可能会放弃等待仍在存在于文件输入缓冲区内的剩余数据。在上面的例子中，会有一个事件产生在RFD句柄上，因为在第2步执行了一个写操作，然后，事件将会在第3步被销毁。因为第4步的读取操作没有读空文件输入缓冲区内的数据，因此我们在第5步调用epoll\_wait(2)完成后，是否挂起是不确定的。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。最好以下面的方式调用ET模式的epoll接口，在后面会介绍避免可能的缺陷。(LT方式可以解决这种缺陷)

i 基于非阻塞文件句柄

ii 只有当read(2)或者write(2)返回EAGAIN时(认为读完)才需要挂起，等待。但这并不是说每次read()时都需要循环读，直到读到产生一个EAGAIN才认为此次事件处理完成，当read()返回的读到的数据长度小于请求的数据长度时(即小于sizeof(buf))，就可以确定此时缓冲中已没有数据了，也就可以认为此事读事件已处理完成。

### 边沿非阻塞触发

### 水平触发

**Level Triggered工作模式(默认的工作方式)**

相反的，以LT方式调用epoll接口的时候，它就相当于一个速度比较快的poll(2)，并且无论后面的数据是否被使用，因此他们具有同样的职能。因为即使使用ET模式的epoll，在收到多个chunk的数据的时候仍然会产生多个事件。调用者可以设定EPOLLONESHOT标志，在epoll\_wait(2)收到事件后epoll会与事件关联的文件句柄从epoll描述符中禁止掉。因此当EPOLLONESHOT设定后，使用带有EPOLL\_CTL\_MOD标志的epoll\_ctl(2)处理文件句柄就成为调用者必须作的事情。

//没有对就绪的fd进行IO操作，内核会不断的通知。

LT(leveltriggered)是缺省的工作方式，并且同时支持block和no-blocksocket。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，所以，这种模式编程出错误可能性要小一点。传统的select/poll都是这种模型的代表。

//没有对就绪的fd进行IO操作，内核不会再进行通知。

ET(edge-triggered)是高速工作方式，只支持no-blocksocket。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)，不过在TCP协议中，ET模式的加速效用仍需要更多的benchmark确认。

另外，当使用epoll的ET模型(epoll的非默认工作方式)来工作时，当产生了一个EPOLLIN事件后，读数据的时候需要考虑的是当recv()返回的大小如果等于要求的大小，即sizeof(buf)，那么很有可能是缓冲区还有数据未读完，也意味着该次事件还没有处理完，所以还需要再次读取：

while(rs) //ET模型

{

buflen = recv(activeevents[i].data.fd, buf, sizeof(buf), 0);

if(buflen < 0)

{

/\*由于是非阻塞的模式，所以当errno为EAGAIN时，表示当前缓冲区已无数 据可读，在这里就当作是该次事件已处理处\*/

if(errno== EAGAIN || errno == EINT)

//即当buflen<0且errno=EAGAIN时，表示没有数据了。(读/写都是这样)

break;

else

### 总结



水平触发模式（以读为例）：只要fd对应的缓冲区有数据；epoll\_wait返回；返回的次数与发送次数无关。

边沿触发模式（以读为例）：客户端发一次数据，server的epoll\_\_wait返回一次；不在乎数据是否读完。

注：LT模式编程简单，效率低，ET模式工作效率高。

## 应用

### Redis

### Nginx

### JavaNIO

## 实例

#include <sys/types.h>

#include <sys/socket.h>

#include <netinet/in.h>

#include <arpa/inet.h>

#include <assert.h>

#include <stdio.h>

#include <unistd.h>

#include <errno.h>

#include <string.h>

#include <fcntl.h>

#include <stdlib.h>

#include <sys/epoll.h>

#include <pthread.h>

#define MAX\_SOCKET\_NUMBERS 1024

#define BUF\_SIZE 1024

int setnonblocking(int fd);

void addfd(int epollfd,int fd, int flag);

void lt(struct epoll\_event\* events,int number,int epollfd,int listenfd);

void et(struct epoll\_event\* events,int number,int epollfd,int listenfd);

int main(int argc,char\* argv[])

{

if(argc <= 2)

{

printf("please input ip address and port num\r\n");

exit(1);

}

const char\* ip = argv[1];

int port = atoi(argv[2]);

int ret = 0;

struct sockaddr\_in address;

bzero(&address,sizeof(address));

address.sin\_family = AF\_INET;

inet\_pton(AF\_INET,ip,&address.sin\_addr);

address.sin\_port = htons(port);

int listenfd = socket(AF\_INET,SOCK\_STREAM,0);

assert(listenfd >= 0);

ret = bind(listenfd,(struct sockaddr\*)&address,sizeof(address));

assert(ret != -1);

ret = listen(listenfd,5);

assert(ret != -1);

struct epoll\_event events[MAX\_SOCKET\_NUMBERS];

int epollfd = epoll\_create(5);

assert( epollfd != -1);

addfd(epollfd,listenfd,1);

while(1)

{

int ret = epoll\_wait(epollfd,events,MAX\_SOCKET\_NUMBERS,-1);

if(ret < 0)

{

printf("epoll\_wait error\r\n");

exit(1);

}

lt(events,ret,epollfd,listenfd);

// et(events,ret,epollfd,listenfd);

}

close(listenfd);

return 0;

}

int setnonblocking(int fd)

{

int old\_option = fcntl(fd,F\_GETFL);

int new\_option = old\_option | O\_NONBLOCK;

fcntl(fd,F\_SETFL,new\_option);

return new\_option;

}

void addfd(int epollfd,int fd,int flag)

{

struct epoll\_event event;

memset(&event,0x00,sizeof(event));

event.data.fd = fd;

event.events = EPOLLIN;

if(flag )

{

event.events |= EPOLLET;

}

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

setnonblocking(fd);

}

void lt(struct epoll\_event\* events,int number ,int epollfd,int listenfd)

{

char buf[BUF\_SIZE];

int i;

for(i=0;i<number;i++)

{

int sockfd = events[i].data.fd;

if(sockfd == listenfd)

{

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

if(connfd < 0)

{

printf("accept error\r\n");

exit(1);

}

addfd(epollfd,connfd,0);

}

else if(events[i].events & EPOLLIN)

{

printf("LT once\r\n");

memset(buf,0x00,sizeof(buf));

int ret = recv(sockfd,buf,sizeof(buf)-1,0);

if(ret <=0)

{

printf("recv 0\r\n");

close(sockfd);

continue;

}

printf("recv data form %d buf is %s\r\n",sockfd,buf);

}

else

{

printf("somthing else happen\r\n");

}

}

}

void et(struct epoll\_event\* events,int number,int epollfd,int listenfd)

{

char buf[BUF\_SIZE];

int i;

for(i=0;i<number;i++)

{

int sockfd = events[i].data.fd;

if(sockfd == listenfd )

{

struct sockaddr\_in client\_address;

socklen\_t client\_addresslen = sizeof(client\_address);

int connfd = accept(listenfd,(struct sockaddr\*)&client\_address,

&client\_addresslen);

addfd(epollfd,connfd,0);

}

else if(events[i].events & EPOLLIN)

{

printf("ET once\r\n");

while(1)

{

memset(buf,0x00,sizeof(buf));

int ret = recv(sockfd,buf,sizeof(buf)-1,0);

if(ret < 0)

{

if((errno == EAGAIN ) || (errno == EWOULDBLOCK))

{

printf("read later\r\n");

break;

}

close(sockfd);

break;

}

else if(ret == 0)

{

close(sockfd);

}

else

{

printf("recv data from %d buf is %s\r\n",sockfd,buf);

}

}

}

else

{

printf("somthin else happen\r\n");

}

}

}

# 对比

