本文档配合主要对如下demo进行配合说明: [借助redis已有的网络相关.c和.h文件，半小时快速实现一个epoll异步网络框架，程序demo](https://github.com/y123456yz/middleware_development_learning/tree/master/%E7%AC%AC%E4%B8%80%E9%98%B6%E6%AE%B5-%E6%89%8B%E6%8A%8A%E6%89%8B%E6%95%99%E4%BD%A0%E5%81%9A%E5%88%86%E5%B8%83%E5%BC%8F%E7%BC%93%E5%AD%98%E4%BA%8C%E6%AC%A1%E5%BC%80%E5%8F%91%E3%80%81%E6%80%A7%E8%83%BD%E4%BC%98%E5%8C%96/%E5%BC%82%E6%AD%A5%E7%BD%91%E7%BB%9C%E6%A1%86%E6%9E%B6%E9%9B%B6%E5%9F%BA%E7%A1%80%E5%AD%A6%E4%B9%A0/asyn_network)

# 1.epoll出现背景

epoll 是 linux 内核为处理大批量文件描述符(网络文件描述符主要是socket返回的套接字fd和accept处理的新连接fd)而作了改进的 poll，是 linux 下多路复用 io接口 select/poll 的增强版本。在 linux 的网络编程中，很长时间都在使用 select 来做事件触发。在 2.6 内核中，有一种替换它的机制，就是 epoll。epoll替换select和poll的主要原因如下:

1. select最多处理1024(内核代码fd\_setsize宏定义)个连接。
2. select和poll采用轮训方式检测内核网络事件，算法事件复杂度为o（n），n为连接数，效率低下。

epoll克服了select和poll的缺点，采用回调方式来检测就绪事件，算法时间复杂度o(1)，相比于select和poll，效率得到了很大的提升。

借助epoll的事件回调通知机制，工作线程可以在没有网络事件通知的时候做其他工作，这样可以最大限度的利用系统cpu资源，服务端不用再阻塞等待客户端网络事件，而是依赖epoll事件通知机制来避免同步等待。

# 2. epoll系统调用接口

## 2.1 epoll\_create函数

函数声明：int epoll\_create(int size)

该函数生成一个epoll专用的文件描述符，其中的参数是指定生成描述符的最大范围。在linux-2.4.32内核中根据size大小初始化哈希表的大小，在linux2.6.10内核中该参数无用，使用红黑树管理所有的文件描述符，而不是hash。

重点:该函数返回的fd将作为其他epoll系统接口的参数。

## epoll\_ctl函数

函数声明：int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event event)

该函数用于控制某个文件描述符上的事件，可以注册事件，修改事件，删除事件。epoll\_wait个参数说明如下：

epfd: epoll事件集文件描述符，也就是epoll\_create返回值

op: 对fd描述符进行的操作类型，可以是添加注册事件、修改事件、删除事件，分别对应宏定义: **epoll\_ctl\_add、epoll\_ctl\_mod、epoll\_ctl\_del。**

fd: 操作的文件描述符。

event: 需要操作的fd对应的epoll\_event事件对象，对象数据来源为fd和op。

## 2.3 epoll\_wait函数

函数声明:int epoll\_wait(int epfd, struct epoll\_event events, int maxevents, int timeout)，该函数用于轮询i/o事件的发生，改函数参数说明如下:

epfd: epoll事件集文件描述符，也就是epoll\_create返回值。

Events：该epoll时间集上的所有epoll\_event信息，每个fd对应的epoll\_event都存入到该数组中，数组每个成员对应一个fd描述符。

Maxevents: 也就是events数组长度。

Timeout: 超时时间，如果在这个超时时间内内核没有I/O网络事件通知，则会超时返回，如果在超时时间内有时间通知，则立马返回

# 3. epoll I/O对路复用主要代码实现流程

代码实现主要由以下几个阶段组成：

1. 创建套接字获取sd，bind然后listen该套接字sd。
2. 把步骤1中的sd添加到epoll事件集中，epoll只关注sd套接字上的新连接请求，新连接对应的事件为读事件AE\_READABLE (EPOLLIN)。并设置新连接事件到来时对应的回调函数为MainAcceptTcpHandler。
3. 在新连接回调函数MainAcceptTcpHandler中，获取新连接，并返回该新连接对应的文件描述符fd，同时把新连接的fd添加到epoll事件集中，该fd开始关注epoll读事件，如果检测到该fd对应的读事件(客户端发送的数据服务端收到后，内核会触发该fd对应的epoll读事件)，则触发读数据回调函数MainReadFromClient。多个连接，每个连接有各自的文件描述符事件结构(该结构记录了各自的私有数据、读写回调函数等)，并且每个连接fd有各自的已就绪事件结构。不同连接有不同的结构信息，最终借助epoll实现I/O多路复用。
4. 进入aeMain事件循环函数中，循环检测步骤2中的新连接事件和步骤3中的数据读事件。如果有对应的epoll事件，则触发epoll\_wait返回，并执行对应事件的回调。

# 4. epoll I/O多路复用主要数据结构及函数说明

## 4.1 主要数据结构

struct aeEventLoop结构用于记录整个epoll事件的各种信息，主要成员如下:

typedef struct aeEventLoop {

// 目前已注册的最大描述符

int maxfd; /\* highest file descriptor currently registered \*/

// 目前已追踪的最大描述符

int setsize; /\* max number of file descriptors tracked \*/

// 用于生成时间事件 id

long long timeEventNextId;

// 最后一次执行时间事件的时间

time\_t lastTime; /\* Used to detect system clock skew \*/

// 已注册的文件事件，每个fd对应一个该结构，events实际上是一个数组

aeFileEvent \*events; /\* Registered events \*/

// 已就绪的文件事件，参考aeApiPoll，数组结构

aeFiredEvent \*fired; /\* Fired events \*/

// 时间事件，所有的定时器时间都添加到该链表中

aeTimeEvent \*timeEventHead;

}

该结构主要由文件描述符事件(即网络I/O事件，包括socket/bind/listen对应的sd文件描述符和accept获取到的新连接文件描述符)和定时器事件组成，其中文件事件主要由events、fired、maxfd、setsize，其中events和fired为数组类型，数组大小为setsize。

events数组成员类型为aeFileEvent，每个成员代表一个注册的文件事件，文件描述符与数组游标对应，例如如果fd=10，则该fd对应的文件事件为event数组的第十个成员Events[10]。

fired数组成员类型为aeFiredEvent，每个成员代表一个就绪的文件事件，文件描述符和数组游标对应，例如如果fd=10，则该fd对应的已就绪的文件事件为fired数组的第十个成员fired [10]。

Setsize为events文件事件数组和fired就绪事件数组的长度，初始值为REDIS\_MAX\_CLIENTS + REDIS\_EVENTLOOP\_FDSET\_INCR。aeCreateEventLoop中提前分配好events和fired数组空间。

maxfd为所有文件描述符中最大的文件描述符，该描述符的作用是调整setsize大小来扩大events和fired数组长度，从而保证存储所有的事件，不会出现数组越界。

**何时扩大events和fireds数组长度？**

例如redis最开始设置的默认最大连接数为REDIS\_MAX\_CLIENTS，如果程序运行一段时间后，我们想调大最大连接数，这时候就需要调整数组长度。

**为什么events和fireds数组长度需要加REDIS\_EVENTLOOP\_FDSET\_INCR？**

因为redis程序中除了网络相关accept新连接的描述符外，程序中也会有普通文件描述符，例如套接字socket描述符、日志文件、rdb文件、aof文件、syslog等文件描述符，确保events和fireds数组长度大于配置的最大连接数，从而避免数组越界。

## 4.2 主要函数实现

主要函数功能请参考以下几个函数:

aeCreateFileEvent

aeDeleteFileEvent

aeProcessEvents

aeApiAddEvent

aeApiDelEvent

aeApiPoll

# 5. 定时器实现原理

## 5.1 定时器主要代码流程

Redis的定时器实际上是借助epoll\_wait实现的，epoll\_wait的超时时间参数timeout是定时器链表中距离当前时间最少的时间差，例如现在是8点1分，我们有一个定时器需要8点1分5秒执行，那么这里epoll\_wait的timeout参数就会设置为5s。

epoll\_wait函数默认等待网络I/O事件，如果8点1分到8点1分5秒这段时间内没有网络I/O事件到来，那么到了8点1分5秒的时候，epoll\_wait就会超时返回。Epoll\_wait返回后就会在aeMain循环体中遍历定时器链表，获取到定时器到达时间比当前时间少的定时器，运行该定时器的对应回调函数。

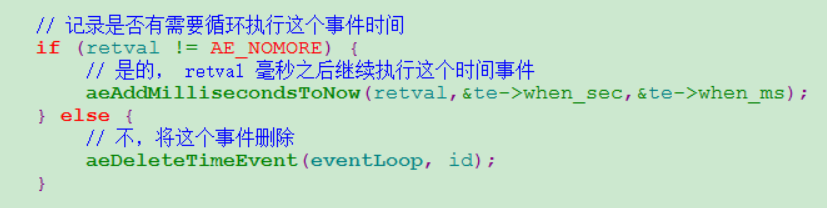
如果在8点1分3秒过程中有网络事件到达，epoll\_wait会在3秒钟返回，返回后处理完对应的网络事件回调函数，然后继续aeMain循环体中遍历定时器链表，获取离当前时间最近的定时器时间为5-3=2秒，也就是还有2秒该定时器才会到期，于是在下一个epoll\_wait中，设置timeout超时时间为2秒，以此循环。

## 5.2 两种不同的定时器(周期性定时器、一次性定时器)

周期性定时器: 指的是定时器到期对应的回调函数执行后，需要重新设置该定时器的超时时间，以备下一个周期继续执行。

一次性定时器: 本次定时时间到执行完对应的回调函数后，把该定时器从定时器链表删除。

两种定时器代码主要代码流程区别如下:



## 5.3 主要数据结构及函数实现

主要数据结构如下:

typedef struct aeEventLoop {

// 时间事件，所有的定时器时间都添加到该链表中

aeTimeEvent \*timeEventHead;

}



主要函数实现参考:

aeCreateTimeEvent

aeDeleteTimeEvent

aeSearchNearestTimer

processTimeEvents

# 6. 常用套接字选项设置

套接字选项可以通过setsockopt函数进行设置，函数声明如下:

 int setsockopt( int socket, int level, int option\_name, const void \*option\_value, size\_t option\_len);

setsockopt参数说明如下:

socket: 可以是bind/listen对应的sd，也可以是accept获取到的新连接fd。

level: 参数level是被设置的选项的级别，套接字级别对应 SOL\_SOCKET，tcp网络设置级别对应SOL\_SOCKET.

option\_name: 选项类型。

optval：[指针](https://baike.baidu.com/item/%E6%8C%87%E9%92%88" \t "_blank)，指向存放选项待设置的新值的[缓冲区](https://baike.baidu.com/item/%E7%BC%93%E5%86%B2%E5%8C%BA)。

optlen：optval缓冲区长度。

## 6.1 SOL\_SOCKET级别套接字选项

Level级别为SOL\_SOCKET的option\_name常用类型如下（说明:网络I/O的文件描述符句柄有两类，一类是针对socket()/bind/listen对应的sd，一类是新连接到来后accept返回的新的连接句柄fd）:

SO\_REUSEADDR: 复用地址，针对socket()/bind/listen对应的sd，避免服务端进程退出再重启后出现error:98，Address already in use。

SO\_RECVBUF: 设置连接fd对应的内核网络协议栈接收缓冲区buf大小，每个连接都会有一个recv buf来接收客户端发送的数据。实际应用中，使用默认值就可以，但如果连接过多，负载过大，内存可能吃不消，这时候可以调小该值。

SO\_SNDBUF：设置连接fd对应的内核网络协议栈发送缓冲区buf大小，每个连接都会有一个send buf来缓存需要发送的连接数据。实际应用中，使用默认值就可以，但如果连接过多，负载过大，内存可能吃不消，这时候可以调小该值。

SO\_KEEPALIVE：针对socket()/bind/listen对应的sd，设置TCP的keepalive机制，由内核网络协议栈实现连接保活。通过该设置可以判断对端异常断电、网络不通的连接问题(如网线松动)。因为客户端异常断电或者网线松动，服务端是不会有epoll异常事件通知的。如果没有设计应用层保活超时报文，则可以依赖协议栈keepalive来检测连接是否异常。

SO\_LINGER：决定关闭连接fd的方式，因为关闭连接的时候，该fd对应的内核协议栈buf可能数据还没有发送出去，如果强制立即关闭可能会出现丢数据的情况。可以根据传入optval参数决定立即关闭连接(可能丢数据)，还是等待数据发送完毕后关闭释放连接或者超时关闭连接。

SO\_RCVTIMEO：针对sd和新连接fd，接收数据超时时间，这个针对阻塞读方式。如果read超过这么多时间还没有获取到内核协议栈数据，则超时返回。

SO\_SNDTIMEO：针对sd和新连接fd，发送数据超时时间，这个针对阻塞写方式。如果write超过这么多时间还没有把数据成功写入到内核协议栈，则超时返回。

## 6.2 IPPROTO\_TCP级别套接字选项

Level级别为IPPROTO\_TCP的option\_name常用类型如下：

TCP\_NODELAY：针对连接fd，是否启用naggle算法，一般禁用，这样可以保证服务端快速回包，降低时延。

# 阻塞、非阻塞

服务端和网络I/O相关相关的几个系统函数主要是:accept()接收客户端连接、read()从内核网络协议栈读取客户端发送来的数据、write()写数据到内核协议栈buf，然后由内核调度发送出去。请参考阻塞demo和非阻塞demo。

## 7.1阻塞及其demo程序验证说明

网上相关说明很多，但是都比较抽象。这里以服务端调用accept为例说明：accept()函数会进行系统调用，从应用层走到内核空间，最终调用内核函数SYSCALL\_DEFINE3()，如果accept对应的sd(socket/bind/listen对应的文件描述符)是阻塞调用(如果不进行设置，默认就是阻塞调用)，SYSCALL\_DEFINE3()对应的函数会判断内核是否收到客户端新连接，如果没有则一直等待，直到有新连接到来或者超时才会返回。

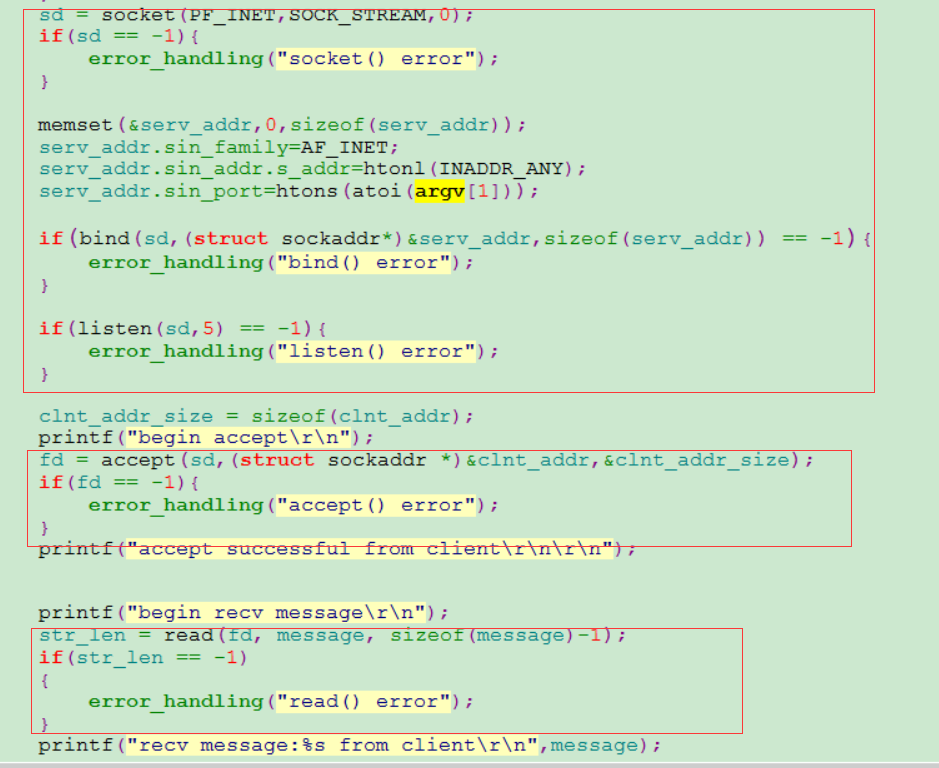
从上面的描述可以看出，如果是阻塞方式，accept()所在的现场会一直等待，整个线程不能做其他事情，这就是阻塞。Accept()阻塞超时时间可以通过上面的SO\_RCVTIMEO设置。

Read()和write()阻塞操作过程和acept()类似，只有在接收到数据和写数据到协议栈成功才会返回，或者超时返回，超时时间分别可以通过SO\_RCVTIMEO和SO\_SNDTIMEO设置。

下面以如下demo为例，来体验阻塞和非阻塞，以下是阻塞操作例子，分别对应服务端和客户端代码:

[服务端阻塞操作程序demo github地址](https://github.com/y123456yz/middleware_development_learning)

[对应客户端程序demo github地址](https://github.com/y123456yz/middleware_development_learning)





从上面的程序，服务端创建套接字后，绑定地址后开始监听，然后阻塞accept()等待客户端连接。如果客户端有连接，则开始阻塞等待read()读客户端发送来的数据，读到数据后打印返回，程序执行结束。

客户端程序创建好套接字，设置好需要连接的服务器Ip和端口，先延时10秒钟才开始连接服务器，连接成功后再次延时10秒，然后发送”block test message”字符串给服务端。

通过CRT开两个窗口，同时启动服务端和客户端程序，服务端打印信息如下:

[root@localhost block\_noblock\_demo]# gcc block\_server.c -o block\_server

[root@localhost block\_noblock\_demo]#

[root@localhost block\_noblock\_demo]# ./block\_server 1234

begin accept //在这里阻塞等待客户端连接

accept successful from client

begin recv message //这里阻塞等待客户端发送数据过来

recv message:block test message from client

[root@localhost block\_noblock\_demo]#

客户端打印信息如下:

[root@localhost block\_noblock\_demo]# gcc block\_client.c -o block\_client

[root@localhost block\_noblock\_demo]# ./block\_client 127.0.0.1 1234

begin connect //begin和end见有10s延时

end connect

begin send message //begin和end间有10s延时

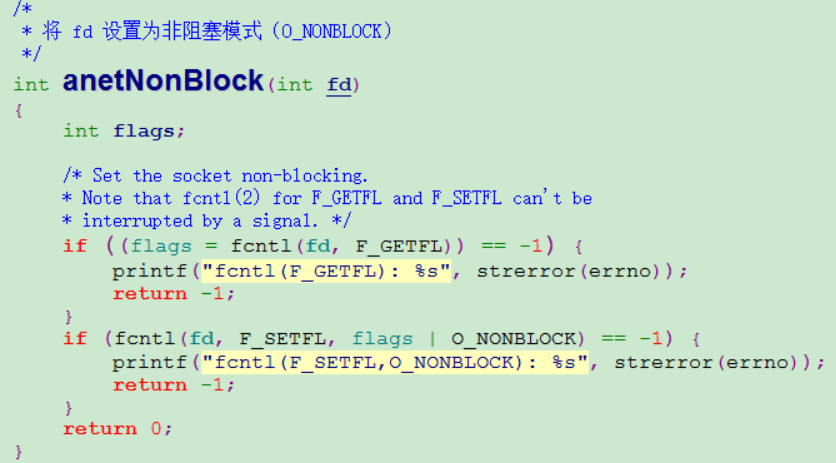
end send message:block test message to server

从运行服务端程序和客户端程序的打印可以看出，如果客户端不发起连接，服务端accept()函数会阻塞等待，知道有新连接到来才会返回。同时启用服务端和客户端程序，服务端accept()函数10s才会返回，因为客户端我故意做了10s延时。Read()阻塞读函数过程和accept()类似。

Write()阻塞验证过程，服务端设置好该链接对应的内核网络协议栈发送缓存区大小，然后传递很大的一个数据给write函数，期望把这个大数据通过write函数写入到内核协议栈发送缓存区。如果内核协议栈缓存区可用buf空间比需要write的数据大，则数据通过write函数拷贝到内核发送缓存区后会立马返回。为了验证write的阻塞过程，我这里故意让客户端不去读数据，这样服务端write的数据就会缓冲到协议栈发送缓冲区，如果缓冲区空间没那么大。Write就会一直等待内核调度把发送缓冲区数据通过网卡发送出去，这样就会腾出空间，继续拷贝用户态write需要写的数据。由于这里我故意让客户端不读数据，该链接对应的发送缓冲区很快就会写满，由于我想要写的数据比这个buf缓冲区大很多，那么write函数就需要阻塞等待，直到把期望发送的数据全部写入到该发送缓存区才会返回，或者超过系统默认的write超时时间才会返回。

## 7.2非阻塞及其demo程序验证说明

非阻塞通过系统调用fcntl设置，函数代码如下:

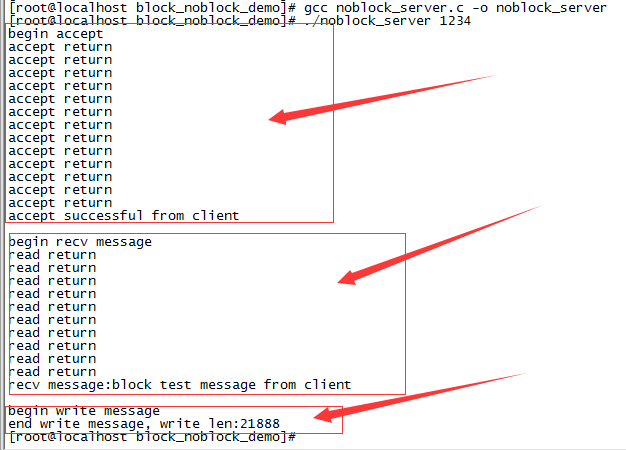


函数中的fd文件描述符可以是socket/bind/listen对应的sd，也可以是accept获取到的新连接fd。非阻塞程序demo github地址如下:

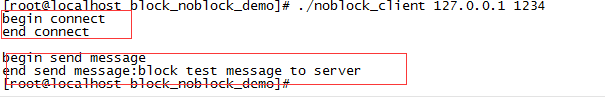
[服务端非阻塞程序demo github地址](https://github.com/y123456yz/middleware_development_learning)

[对应客户端非阻塞程序demo github地址](https://github.com/y123456yz/middleware_development_learning)

编译程序，先启动服务端程序，然后启动客户端程序，验证方法如下：



对应客户端

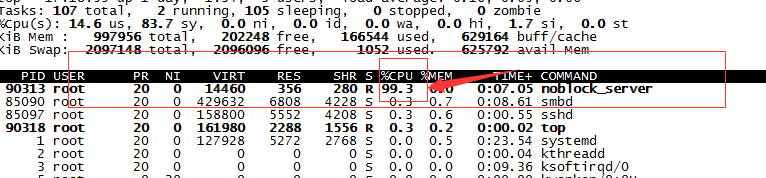


客户端启动后，延迟10秒向服务端发起连接，连接建立成功后，延迟10秒向服务端发送数据。服务端启动后，设置sd为非阻塞，开始accept等待接收客户端连接，如果accept系统调用没有获取到客户端连接，则延时1秒钟，然后继续accept。由于客户端启动后要延迟10s钟才发起连接，因此accept会有十次accept return打印。read过程和accept类似，可以查看demo代码。

我们知道write操作是把用户态向要发送的数据拷贝到内核态连接对应的send buf缓冲区，如果是阻塞方式，如果内核缓存区空间不够，则write会阻塞等待。但是如果我们把新连接的fd设置为非阻塞，及时内核发送缓冲区空间不够，write也会立马返回，并返回本次写入到内核空间的数据量，不会阻塞等待。可以通过运行demo自己来体验这个过程。

大家应该注意到，则非阻塞操作服务端demo中，accept(),read()如果没有返回我们想要的连接或者数据，demo中做了sleep延时，为什么这里要做点延时呢？

原因是如果不做延时，这里会不停的进行accept read系统调用，系统调用过程是个非常消耗性能的过程，会造成CPU的大量浪费。假设我们不加sleep延时，通过top可以查看到如下现象:



# 同步、异步

同步和异步是比较抽象的概念，还是用程序demo来说明。

## 8.1 同步

上面的[服务端阻塞操作程序demo github地址](https://github.com/y123456yz/middleware_development_learning)和[对应客户端程序demo github地址](https://github.com/y123456yz/middleware_development_learning)实际上都是同步调用的过程。这两个demo都是单线程的，以accept()调用为例，不管是阻塞操作还是非阻塞操作，由于服务端不知道客户端合适发起连接，因此只能阻塞等待，或者非阻塞轮训查询。不管是阻塞等待还是轮训查询，效率都非常低下，整个线程不能做其他工作，CPU完全利用不起来。

## 8.2 异步

[借助redis已有的网络相关.c和.h文件，半小时快速实现一个epoll异步网络框架，程序demo](https://github.com/y123456yz/middleware_development_learning/tree/master/%E7%AC%AC%E4%B8%80%E9%98%B6%E6%AE%B5-%E6%89%8B%E6%8A%8A%E6%89%8B%E6%95%99%E4%BD%A0%E5%81%9A%E5%88%86%E5%B8%83%E5%BC%8F%E7%BC%93%E5%AD%98%E4%BA%8C%E6%AC%A1%E5%BC%80%E5%8F%91%E3%80%81%E6%80%A7%E8%83%BD%E4%BC%98%E5%8C%96/%E5%BC%82%E6%AD%A5%E7%BD%91%E7%BB%9C%E6%A1%86%E6%9E%B6%E9%9B%B6%E5%9F%BA%E7%A1%80%E5%AD%A6%E4%B9%A0/asyn_network)，这个demo是异步操作。还是以该demo的accept为例说明，从这个demo可以看出，sd设置为非阻塞，借助epoll机制，当有新的连接事件到来后，触发epoll\_wait返回，并返回所有的文件描述符对应的读写事件，这样就触发执行对应的新连接回调函数MainAcceptTcpHandler。借助epoll事件通知机制，就避免了前面两个demo的阻塞等待过程和轮训查询过程，整个accept()操作由事件触发，不必轮训等待。本异步网络框架demo也是单线程，就不存在前面两个demo只能做accept这一件事，如果没有accept事件到来，本异步网络框架demo线程还可以处理其他已有连接的读写事件，这样线程CPU资源也就充分利用起来了。

打个形象的比喻，假设我们每年单位都有福利体检，体检后一到两周出体检报告，想要获取体检报告有两种方式。第一种方式: 体检结束一周后，你就坐在医院一直等，直到体检报告出来，整个过程你是无法正常去单位上班的(这就相当于前面的服务端阻塞demo方式)。第二种方式:你每天都跑去体检医院询问，我的体检报告出了吗，如果没有，第二天有去体检医院，以此重复，直到有一天你去体检医院拿到体检报告。在你每天去医院询问是否已经出体检报告的过程中，你是不能正常上班的(这种方式类似于前面的服务端非阻塞demo方式)。第三种方式：你每天正常上班，等体检医院打电话通知你拿体检报告，你再去拿，电话通知你拿体检报告的过程就相当于异步事件通知，这样你就可以正常上班了。第一、二种方式就是同步操作，第三种方式就是异步操作。

**总结: 同步和异步的区别就是异步操作借助epoll的事件通知机制，从而可以充分利用CPU资源。**