核心概念：epoll instance

内核数据结构是两个list：

1. interest list(epoll set):进程已经注册的一个感兴趣的文件描述符的集合
2. ready list：已经准备好IO的文件描述符的集合，是interest list的子集

三个重要的函数：

1. epoll\_create:创建一个新的epoll instance，并且返回指向那个instance的文件描述符
2. epoll\_ctl:通过这个函数注册描述符进interest list

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

对epfd所指的那个eoll instance的interest list进行增加、修改、删除事件

操作op的目标文件描述符是fd

成功，返回0

1. epoll\_wait:等待IO事件，如果没有可用事件，就阻塞

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

timeout指定epoll\_wait阻塞的最大时间

调用被阻塞，除非以下发生：1、文件描述符传递一个事件；2、调用被信号终止；3、超时

成功，返回处于就绪状态的文件描述符的数量

水平触发(EPOLLLT)和 边缘触发(EPOLLET)：

Level\_triggered(水平触发)：当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据一次性全部读写完(如读写缓冲区太小)，那么下次调用 epoll\_wait()时，它还会通知你在上没读写完的文件描述符上继续读写，当然如果你一直不去读写，它会一直通知你！！！如果系统中有大量你不需要读写的就绪文件描述符，而它们每次都会返回，这样会大大降低处理程序检索自己关心的就绪文件描述符的效率！！！

 select(),poll()模型都是水平触发模式

Edge\_triggered(边缘触发)：当被监控的文件描述符上有可读写事件发生时，epoll\_wait()会通知处理程序去读写。如果这次没有把数据全部读写完(如读写缓冲区太小)，那么下次调用epoll\_wait()时，它不会通知你，也就是它只会通知你一次，直到该文件描述符上出现第二次可读写事件才会通知你！！！这种模式比水平触发效率高，系统不会充斥大量你不关心的就绪文件描述符！！！

Epoll的思想是首先获取一个epoll instance，然后在上面注册文件描述符。而select和poll都是在内核空间构建一个大的文件描述符的数组。

因为有事件发生的文件描述符的数量/被控制的总的文件描述符的数量的比率非常小，可以认为时间复杂度是常数。

The reason of using a file descriptor as the reference to an epoll instance is that this makes the epoll instance also pollable

Epoll\_create函数的源代码：

数据结构：struct eventpoll

为struct eventpoll 数据结构分配空间并初始化:

error = ep\_alloc(&ep);

从进程中得到一个未使用过的文件描述符：

fd = get\_unused\_fd\_flags(O\_RDWR | (flags & O\_CLOEXEC));

获得一个匿名的inode，不太懂？？

file = anon\_inode\_getfile("[eventpoll]", &eventpoll\_fops, ep,

O\_RDWR | (flags & O\_CLOEXEC));

把文件描述符和匿名inode绑定起来，然后把fd返回给调用进程。

fd\_install(fd, file);

return fd;

epoll instance如何记住文件描述符？？

使用红黑树记录一个epoll instance所管理的所有文件描述符。其中根节点用ep\_alloc()函数初始化。

Struct epitem 树中的每个节点会指向这样一个数据结构，应该是红黑树中的节点项

当使用epoll\_ctl对epoll instance增加一个文件描述符时，首先会调用ep\_find()尝试放置一个struct eqitem

红黑树中的key：

struct epoll\_filefd {

struct file \*file; // pointer to the target file struct corresponding to the fd

int fd; // target file descriptor number

} \_\_packed;

排序二叉树中的比较函数，先比较文件地址，再比较文件描述符

/\* Compare RB tree keys \*/

static inline int ep\_cmp\_ffd(struct epoll\_filefd \*p1,

struct epoll\_filefd \*p2)

{

return (p1->file > p2->file ? +1:

(p1->file < p2->file ? -1 : p1->fd - p2->fd));

}

Epoll\_ctl函数源代码：

首先调用ep\_find()查找相关节点，没有找到时，调用ep\_insert()添加一个文件描述符到红黑树中，

ep\_insert()函数源代码：

首先检查被现在的用户所观察的用户数量是否超过max\_user\_wathches

user\_watches = atomic\_long\_read(&ep->user->epoll\_watches);

if (unlikely(user\_watches >= max\_user\_watches))

return -ENOSPC;

然后从kernel slab allocator中分配内存：

if (!(epi = kmem\_cache\_alloc(epi\_cache, GFP\_KERNEL)))

return -ENOMEM;

然后进行一系列的初始化：

/\* Item initialization follow here ... \*/

INIT\_LIST\_HEAD(&epi->rdllink);

INIT\_LIST\_HEAD(&epi->fllink);

INIT\_LIST\_HEAD(&epi->pwqlist);

epi->ep = ep;

ep\_set\_ffd(&epi->ffd, tfile, fd);

epi->event = \*event;

epi->nwait = 0;

epi->next = EP\_UNACTIVE\_PTR;

然后会尝试注册回调函数到文件描述符中。

一些结构体：

其中poll\_queue\_proc是一种回调函数：

typedef void (\*poll\_queue\_proc)(struct file \*, wait\_queue\_head\_t \*, struct poll\_table\_struct \*);

typedef struct poll\_table\_struct {

poll\_queue\_proc \_qproc;

unsigned long \_key;

} poll\_table;

\_key其实是一个事件类型的掩码，也就是可以收到什么类型的掩码？

把两种结构体结合在一起，表示两者的联系

struct ep\_pqueue {

poll\_table pt;

struct epitem \*epi;

};

初始化struct ep\_pqueue，将回调函数指向ep\_ptable\_queue\_proc

/\* Initialize the poll table using the queue callback \*/

epq.epi = epi;

init\_poll\_funcptr(&epq.pt, ep\_ptable\_queue\_proc);

然后调用ep\_item\_poll(epi, &epq.pt);

调用相应文件的poll实现

ep\_ptable\_queue\_proc函数的实现

static void ep\_ptable\_queue\_proc(struct file \*file, wait\_queue\_head\_t \*whead,

poll\_table \*pt)

{

struct epitem \*epi = ep\_item\_from\_epqueue(pt);

struct eppoll\_entry \*pwq;

if (epi->nwait >= 0 && (pwq = kmem\_cache\_alloc(pwq\_cache, GFP\_KERNEL))) {

init\_waitqueue\_func\_entry(&pwq->wait, ep\_poll\_callback);

pwq->whead = whead;

pwq->base = epi;

add\_wait\_queue(whead, &pwq->wait);

list\_add\_tail(&pwq->llink, &epi->pwqlist);

epi->nwait++;

} else {

/\* We have to signal that an error occurred \*/

epi->nwait = -1;

}

}

It is vital that epoll keeps track the head of the wait queue for the monitored file. Otherwise epoll would not be able to unregister from the wait queue later

pwq->wait might be the most important thing in the whole epoll implementation because it will be used for:

1. monitoring events happening on that particular monitored file
2. wake up other processes as necessary

然后把pwq加到eppoll\_entry链表的尾部，struct epitem内有个成员变量就是eppoll\_entry的链表 不懂？？？

Linux如何通知epoll instance事件发生？

//SIG\_IGN,系统函数，忽略信号的处理程序,客户端发送RST包后，服务器还调用write会触发

//忽略pipe信号的原因：

//TCP的11种状态

//如果服务端主动关闭套接字close 编程中的close与四次挥手不太一样，当某一端调用close，也就意味这一端既不接收数据，也不发送数据

//而客户端调用了一次write，服务器会接收一个RST segment（TCP传输层）

//如果客户端再次调用了write， 这个时候服务端就会产生SIGPIPE信号，而此信号默认的处理方式就是关闭进程

//关闭进程不符合高并发服务器的特点

//TIME\_WAIT状态 对高并发服务器的影响

//应避免服务器端出现此状态

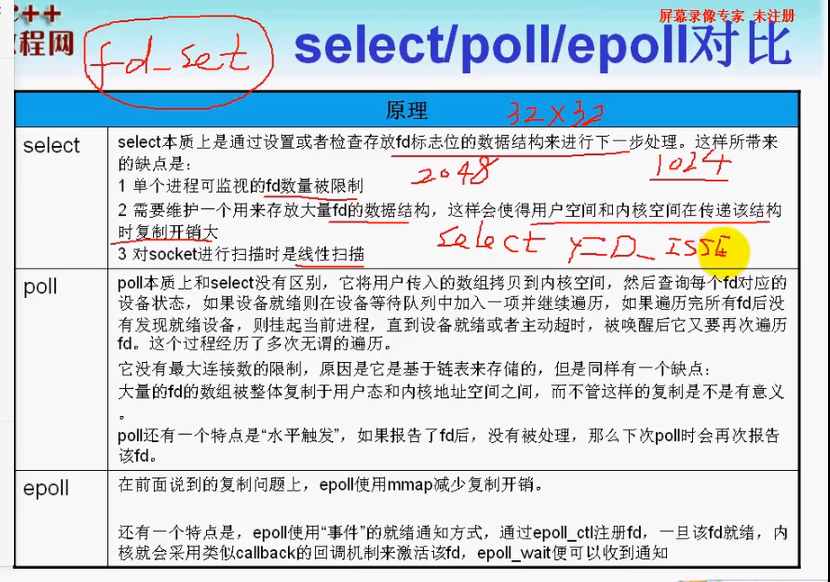
//如果服务器端主动断开连接(先于client调用close，)服务器端会进入TIME\_WAIT状态，会浪费一些资源

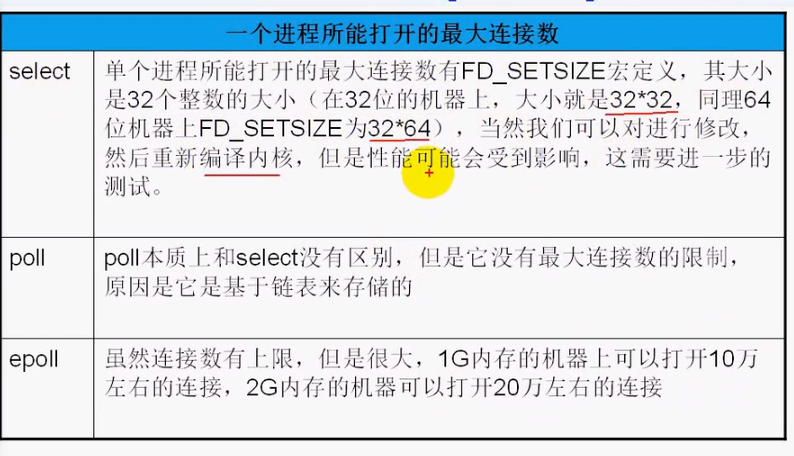
//所以在协议设计上，应该让客户端主动断开连接，

//如果客户端不活跃了，一些客户端不断开连接，这样会占用服务器端的连接资源

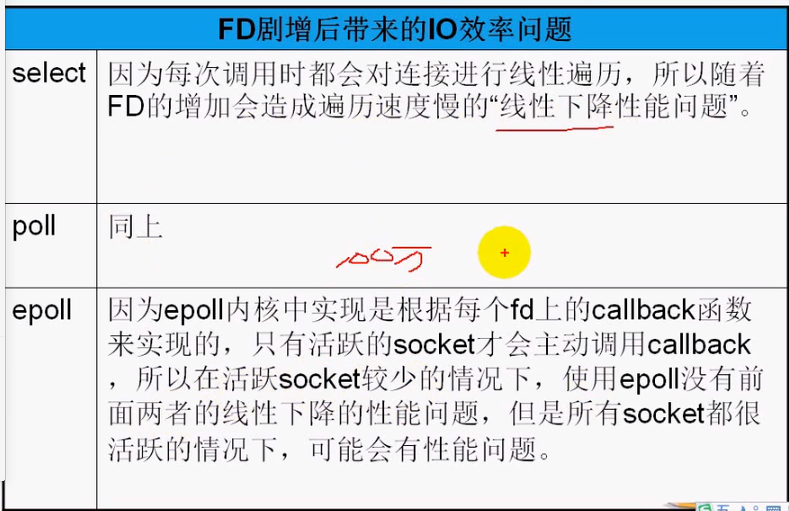
//服务端也要有个机制踢掉不活跃的连接 close

Epoll\_wait的block和non-block版本，根据timeout是否为0判断





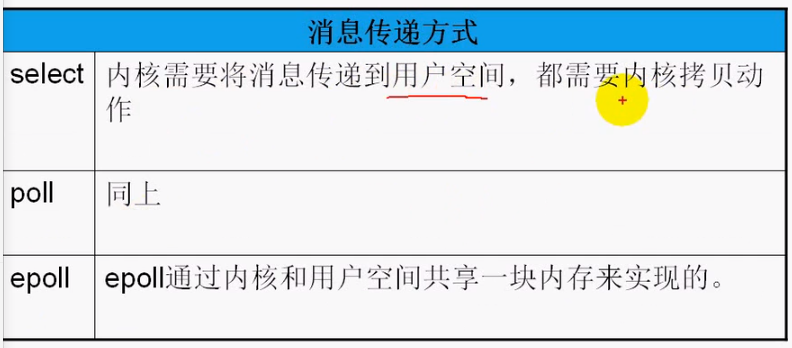
Epoll连接数没有上限



如果已连接套接字不太大，并且这些套接字非常活跃

Poll select 一次性遍历返回活跃的文件描述符

Epoll 内部实现更复杂 更复杂的代码逻辑。优势在于处理大量的连接



Bilibli视频 从零开始开发服务器框架

##开发环境

Cmake

##项目路径

bin ---二进制

build ---中间文件路径

cmake ---cmake函数文件夹

CmakeList.txt ---cmake的定义文件

lib ---库的输出路径

Makefile

sylar –源代码路径

tests ---测试路径

##日志系统

1. Log4j 所用到的类
   * 1. Logger(定义日志类别) 有级别 有name /Appender列表，输出到目的地的集合
     2. Formatter（日志格式）
     3. Appender（日志输出地方）

日志输出地方有很多，多态 虚函数

高并发服务器模型：Linux环境

1. 多进程并发服务器 客户端数量低时可以考虑 进程多时会出现排队现象
2. 多线程。。
3. 多路IO服用 :select/poll/epoll

时间片 :分时复用技术

点到点：PC IP地址

端到端: PC唯一进程 端口号

Socket返回一个文件描述符

用epoll实现高并发

Epoll实现原理：

<https://idndx.com/2014/09/01/the-implementation-of-epoll-1/>

epoll首先需要注册文件描述符，epoll只会关注注册的文件描述符，而不是所有的

服务器开发中常用的库：

Libevent libev nginx 木铎

Github上的那个项目

关于Reactor模式: 堵塞、非堵塞的区别是在于第一阶段，即数据准备阶段。无论是堵塞还是非堵塞，都是用应用主动找内核要数据，而read数据的过程是‘堵塞’的，直到数据读取完。

同步、异步的区别在于第二阶段，若由请求者主动的去获取数据，则为同步操作，需要说明的是：read/write操作也是‘堵塞’的，直到数据读取完。  
若数据的read都由kernel内核完成了(在内核read数据的过程中，应用进程依旧可以执行其他的任务)，这就是异步操作。

下面所说的读应该都是第二步的过程。

换句话说，BIO里用户最关心“我要读”，NIO里用户最关心"我可以读了"，在AIO模型里用户更需要关注的是“读完了”。  
NIO一个重要的特点是：socket主要的读、写、注册和接收函数，在等待就绪阶段都是非阻塞的，真正的I/O操作是同步阻塞的（消耗CPU但性能非常高）。  
NIO是一种同步非阻塞的I/O模型，也是I/O多路复用的基础。

一般情况下，I/O 复用机制需要事件分发器。 事件分发器的作用，将那些读写事件源分发给各读写事件的处理者。  
涉及到事件分发器的两种模式称为：Reactor和Proactor。 Reactor模式是基于同步I/O的，而Proactor模式是和异步I/O相关的

reactor只是一种设计模式，是一种基于事件响应的模式

Reactor模式的核心是解决多请求问题，如果有特别多的请求同时发生，不会因为线程池被短时间占满而拒绝服务。我们一般实现多请求的模块，会采用线程池的实现方案，这种方案对于并发量不是特别大的场景是足够用的

但是线程池方案的最大缺点就是，如果瞬间有大并发则会一下子耗满线程，整个服务陷入阻塞中，后续请求将无法接入。基于Reactor模式实现的方案，会有一个Dispatcher先接收event，然后快速分发给相应的耗时eventHandler处理器去处理，这样就不会阻塞请求的接收。

Reactor模式的缺点也很明显，模型复杂，因为涉及到内部回调，多线程处理，不容易调试；需要操作系统底层支持，这就导致不同操作系统可能会产生不一样的结果。所以总的来说如果并发要求不是那么高，使用传统的阻塞线程池模型足够了，而且调试、查问题都会简单很多；如果我们的使用场景是会产生瞬时大并发，可以使用Reactor模式来实现，目前大部分的NIO框架或者容器都是实现了Reactor模式

[边缘触发(Edge Trigger)和水平触发也叫条件触发(Level Trigger)](http://blog.csdn.net/josunna/article/details/6269235)：

**水平触发(level-triggered，也被称为条件触发)LT:** 只要满足条件，就触发一个事件(只要有数据没有被获取，内核就不断通知你)  
**边缘触发(edge-triggered)ET:** 每当状态变化时，触发一个事件epoll有EPOLLLT和EPOLLET两种触发模式，LT是默认的模式，ET是“高速”模式。

所以在ET模式下，read一个fd的时候一定要把它的buffer读光，也就是说一直读到read的返回值小于请求值，或者 遇到EAGAIN错误。

随着连接数增 加，select和poll的性能是严重非线性下降。

举个读socket的例子，假定经过长时间的沉默后，现在来了100个字节，这时无论边缘触发和条件触发都会产生一个read ready notification通知应用程序可读。应用程序读了50个字节，然后重新调用api等待io事件。这时条件触发的api会因为还有50个字节可读从 而立即返回用户一个read ready notification。而边缘触发的api会因为可读这个状态没有发生变化而陷入长期等待。 因此在使用边缘触发的api时，要注意每次都要读到socket返回EWOULDBLOCK为止，否则这个socket就算废了。而使用条件触发的api 时，如果应用程序不需要写就不要关注socket可写的事件，否则就会无限次的立即返回一个write ready notification。大家常用的select就是属于条件触发这一类，长期关注socket写事件会出现CPU 100%的毛病。

Epoll的优点：

1. **支持一个进程打开大数目的socket描述符(FD)** ：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2048,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。
2. **IO效率不随FD数目增加而线性下降** ：select/poll每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是epoll不存在这个问题，它只会对"活跃"的socket进行 操作---这是因为在内核实现中epoll是根据每个fd上面的callback函数实现的。
3. **使用mmap加速内核与用户空间的消息传递。** ：论是select,poll还是epoll都需要内核把FD消息通知给用户空间，如何避免不必要的内存拷贝就 很重要，在这点上，epoll是通过内核于用户空间mmap同一块内存实现的。

协程比线程更轻量级，是线程的线程

！！！！！

枚举enum 是全局的

枚举类 enum class 两者的区别

解析 http报文，使用 ragle

彻底禁止对象的拷贝 继承Noncopyable类

POLLOUT事件触发条件：当内核缓冲区不满时（可以容纳数据）。但是不能一开始就关注POLLOUT事件，因为开始时缓冲区为空，会一直触发POLLOUT事件，导致busy\_loop忙等待。

实现epoll监听只要3步骤（只需要3个API函数）

epoll （红黑树） 内部用到的两个数据结构：红黑树；链表

解决select poll的问题：

（1）每次都要传送数组和集合到内核，然后内核在返回出来

（2）每次处理事件都需要遍历处理事件

1、告诉内核一次监听那些文件描述符

内核回创建一个链表，将lfd，cfd存在链表上，在将链表的内容拷贝给用户

1. 返回发生事件的文件描述符

Epoll 的本质！！

<https://blog.csdn.net/songchuwang1868/article/details/89877739>

网络数据通过网线传到网卡：

**网卡会把接收到的数据写入内存。**

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。

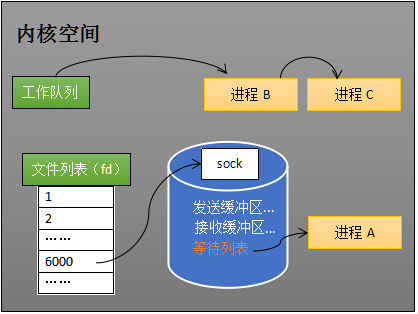
**进程阻塞为什么不占用cpu资源？**

**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、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**。



Windows下的多路复用：

IOCP：以后补上

# [Reactor与Proactor区别](https://www.cnblogs.com/wanpengcoder/p/5340115.html)

<https://www.cnblogs.com/wanpengcoder/p/5340115.html>

<https://blog.csdn.net/caiwenfeng_for_23/article/details/8458299>

应该就是同步IO和异步IO的区别，然后Reactor用到了多路转接

如网络编程中accept之后等待数据到达，并且读取数据为例：

Reactor: 基于同步IO

1. 线程等待读取socket数据，将socketfd添加到事件分派器中，如添加到epoll；

2. 事件分派器阻塞等待socketfd可读事件发生；

3. 若数据到达，socketfd变成可读状态，事件分派器通知线程(或者回调)处理；

4. 线程阻塞完成从socket读数据；

--我要读，告诉你，你发现可以读的时候通知我，然后我去读；

Proactor: 基于异步IO

1. 线程等待读取socket数据，将存储数据的缓冲区和和读事件请求交给事件分派器；

2. 事件分派器等待socket数据到达；

3. 若数据到达，事件分派器不通知线程读取，而是直接完成数据的读取；

4. 通知线程读取数据完成，并已经存入提供的缓冲区中；

--我要读，告诉你，你帮我读好，再通知我；

同步和异步是针对**应用程序和内核**的交互而言的；

       同步指的是用户进程触发IO操作并等待或者轮询的去查看IO操作是否就绪，

       异步是指用户进程触发IO操作以后便开始做自己的事情，而当IO操作已经完成的时候会得到IO完成的通知。

<https://daimojingdeyu.iteye.com/blog/828696>

<https://xmuzyq.iteye.com/blog/783218>

reactor模式在实际生活中的类比:

在单核的机上，多线程并不能提高系统的性能，除非在有一些阻塞的情况发生。否则线程切换的开销会使处理的速度变慢。就像你一个人做两件事情，1、削一个苹果。2、切一个西瓜。那你可以一件一件的做，我想你也会一件一件的做。如果这个时候你使用多线程，一会儿削苹果，一会切西瓜，可以相像究竟是哪个速度快。这也就是说为什么在单核机上多线程来处理可能会更慢。

但当有阻碍操作发生时，多线程的优势才会显示出来，现在你有另外两件事情去做，1、削一个苹果。2、烧一壶开水。我想没有人会去做完一件再做另一件，你肯定会一边烧水，一边就把苹果削了。

Reactor和Proactor模式的主要区别就是真正的读取和写入操作是有谁来完成的，Reactor中需要应用程序自己读取或者写入数据，而Proactor模式中，应用程序不需要进行实际的读写过程，它只需要从缓存区读取或者写入即可，操作系统会读取缓存区或者写入缓存区到真正的IO设备.