## select、poll、epoll之间的区别总结

select，poll，epoll都是IO多路复用的机制。I/O多路复用就通过一种机制，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。**但select，poll，epoll本质上都是同步I/O，因为他们都需要在读写事件就绪后自己负责进行读写，也就是说这个读写过程是阻塞的**，而异步I/O则无需自己负责进行读写，异步I/O的实现会负责把数据从内核拷贝到用户空间。关于这三种IO多路复用的用法，前面三篇总结写的很清楚，并用服务器回射echo程序进行了测试。

### [IO多路复用之select总结](http://www.cnblogs.com/Anker/p/3258674.html)

**1、基本概念**

　　IO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。IO多路复用适用如下场合：

（1）当客户处理多个描述字时（一般是交互式输入和网络套接口），必须使用I/O复用。

（2）当一个客户同时处理多个套接口时，而这种情况是可能的，但很少出现。

（3）如果一个TCP服务器既要处理监听套接口，又要处理已连接套接口，一般也要用到I/O复用。

（4）如果一个服务器即要处理TCP，又要处理UDP，一般要使用I/O复用。

（5）如果一个服务器要处理多个服务或多个协议，一般要使用I/O复用。

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。

**2、select函数**

　　该函数准许进程指示内核等待多个事件中的任何一个发送，并只在有一个或多个事件发生或经历一段指定的时间后才唤醒。函数原型如下：

#include <sys/select.h>

#include <sys/time.h>

int select(int maxfdp1,fd\_set \*readset,fd\_set \*writeset,fd\_set \*exceptset,const struct timeval \*timeout)

返回值：就绪描述符的数目，超时返回0，出错返回-1

函数参数介绍如下：

（1）第一个参数maxfdp1指定待测试的描述字个数，它的值是待测试的最大描述字加1（因此把该参数命名为maxfdp1），描述字0、1、2...maxfdp1-1均将被测试。

因为文件描述符是从0开始的。

（2）中间的三个参数readset、writeset和exceptset指定我们要让内核测试读、写和异常条件的描述字。如果对某一个的条件不感兴趣，就可以把它设为空指针。struct fd\_set可以理解为一个集合，这个集合中存放的是文件描述符，可通过以下四个宏进行设置：

          void FD\_ZERO(fd\_set \*fdset);           //清空集合

          void FD\_SET(int fd, fd\_set \*fdset);   //将一个给定的文件描述符加入集合之中

          void FD\_CLR(int fd, fd\_set \*fdset);   //将一个给定的文件描述符从集合中删除

          int FD\_ISSET(int fd, fd\_set \*fdset);   // 检查集合中指定的文件描述符是否可以读写

（3）timeout告知内核等待所指定描述字中的任何一个就绪可花多少时间。其timeval结构用于指定这段时间的秒数和微秒数。

         struct timeval{

                   long tv\_sec;   //seconds

                   long tv\_usec;  //microseconds

       };

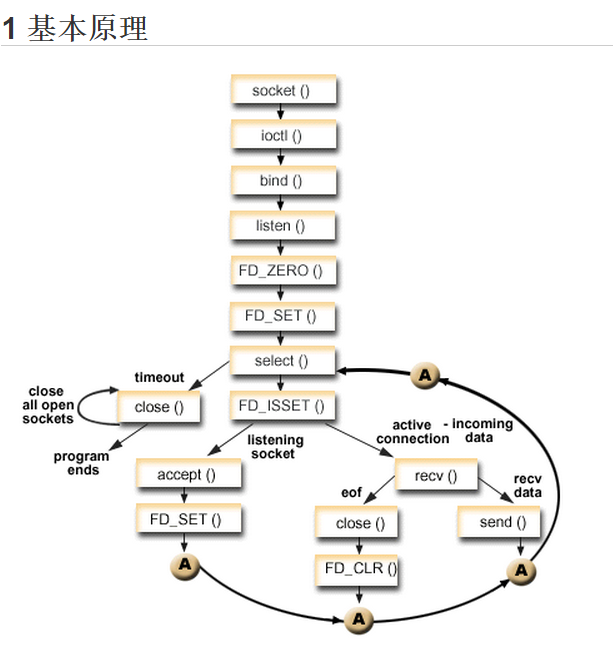
这个参数有三种可能：

（1）永远等待下去：仅在有一个描述字准备好I/O时才返回。为此，把该参数设置为空指针NULL。

（2）等待一段固定时间：在有一个描述字准备好I/O时返回，但是不超过由该参数所指向的timeval结构中指定的秒数和微秒数。

（3）根本不等待：检查描述字后立即返回，这称为轮询。为此，该参数必须指向一个timeval结构，而且其中的定时器值必须为0。

 原理图：



### [IO多路复用之poll总结](http://www.cnblogs.com/Anker/p/3261006.html)

**1、基本知识**

　　poll的机制与select类似，与select在本质上没有多大差别，管理多个描述符也是进行轮询，根据描述符的状态进行处理，但是poll没有最大文件描述符数量的限制。poll和select同样存在一个缺点就是，包含大量文件描述符的数组被整体复制于用户态和内核的地址空间之间，而不论这些文件描述符是否就绪，它的开销随着文件描述符数量的增加而线性增大。

**2、poll函数**

　　函数格式如下所示：

# include <poll.h>

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

pollfd结构体定义如下：

**struct pollfd {**

**int fd;         /\* 文件描述符 \*/**  
**short events;         /\* 等待的事件 \*/**  
**short revents;       /\* 实际发生了的事件 \*/**  
**} ;**

　　每一个pollfd结构体指定了一个被监视的文件描述符，可以传递多个结构体，指示poll()监视多个文件描述符。每个结构体的events域是监视该文件描述符的事件掩码，由用户来设置这个域。revents域是文件描述符的操作结果事件掩码，内核在调用返回时设置这个域。events域中请求的任何事件都可能在revents域中返回。合法的事件如下：

　　POLLIN 　　　　　　　　有数据可读。

　　POLLRDNORM 　　　　  有普通数据可读。

　　POLLRDBAND　　　　　 有优先数据可读。

　　POLLPRI　　　　　　　　 有紧迫数据可读。

　　POLLOUT　　　　　　      写数据不会导致阻塞。

　　POLLWRNORM　　　　　  写普通数据不会导致阻塞。

　　POLLWRBAND　　　　　   写优先数据不会导致阻塞。

　　POLLMSGSIGPOLL 　　　　消息可用。

　　此外，revents 域中还可能返回下列事件：  
　　POLLER　　     指定的文件描述符发生错误。

　　POLLHUP　　  指定的文件描述符挂起事件。

　　POLLNVAL　　 指定的文件描述符非法。

这些事件在events域中无意义，因为它们在合适的时候总是会从revents中返回。

　　使用poll()和select()不一样，你不需要显式地请求异常情况报告。  
　　POLLIN | POLLPRI等价于select()的读事件，POLLOUT |POLLWRBAND等价于select()的写事件。POLLIN等价于POLLRDNORM |POLLRDBAND，而POLLOUT则等价于POLLWRNORM。例如，要同时监视一个文件描述符是否可读和可写，我们可以设置 events为POLLIN |POLLOUT。在poll返回时，我们可以检查revents中的标志，对应于文件描述符请求的events结构体。如果POLLIN事件被设置，则文件描述符可以被读取而不阻塞。如果POLLOUT被设置，则文件描述符可以写入而不导致阻塞。这些标志并不是互斥的：它们可能被同时设置，表示这个文件描述符的读取和写入操作都会正常返回而不阻塞。

　　timeout参数指定等待的毫秒数，无论I/O是否准备好，poll都会返回。timeout指定为负数值表示无限超时，使poll()一直挂起直到一个指定事件发生；timeout为0指示poll调用立即返回并列出准备好I/O的文件描述符，但并不等待其它的事件。这种情况下，poll()就像它的名字那样，一旦选举出来，立即返回。

　　返回值和错误代码  
　　成功时，poll()返回结构体中revents域不为0的文件描述符个数；如果在超时前没有任何事件发生，poll()返回0；失败时，poll()返回-1，并设置errno为下列值之一：  
　　EBADF　　       一个或多个结构体中指定的文件描述符无效。

　　EFAULTfds　　 指针指向的地址超出进程的地址空间。

　　EINTR　　　　  请求的事件之前产生一个信号，调用可以重新发起。

　　EINVALnfds　　参数超出PLIMIT\_NOFILE值。

　　ENOMEM　　     可用内存不足，无法完成请求。

### [IO多路复用之epoll总结](http://www.cnblogs.com/Anker/p/3263780.html)

**1、基本知识**

　　epoll是在2.6内核中提出的，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。

**2、epoll接口**

　　epoll操作过程需要三个接口，分别如下：

#include <sys/epoll.h>

int epoll\_create(int size);

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

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

1. **int epoll\_create(int size);**

创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大。这个参数不同于select()中的第一个参数，给出最大监听的fd+1的值。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。

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

epoll的事件注册函数，它不同与select()是在监听事件时告诉内核要监听什么类型的事件epoll的事件注册函数，它不同与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。第一个参数是epoll\_create()的返回值，第二个参数表示动作，用三个宏来表示：

EPOLL\_CTL\_ADD：注册新的fd到epfd中；  
EPOLL\_CTL\_MOD：修改已经注册的fd的监听事件；  
EPOLL\_CTL\_DEL：从epfd中删除一个fd；

第三个参数是需要监听的fd，第四个参数是告诉内核需要监听什么事，struct epoll\_event结构如下：

struct epoll\_event {

\_\_uint32\_t events; /\* Epoll events \*/

epoll\_data\_t data; /\* User data variable \*/

};

events可以是以下几个宏的集合：  
EPOLLIN ：表示对应的文件描述符可以读（包括对端SOCKET正常关闭）；  
EPOLLOUT：表示对应的文件描述符可以写；  
EPOLLPRI：表示对应的文件描述符有紧急的数据可读（这里应该表示有带外数据到来）；  
EPOLLERR：表示对应的文件描述符发生错误；  
EPOLLHUP：表示对应的文件描述符被挂断；  
EPOLLET： 将EPOLL设为边缘触发(Edge Triggered)模式，这是相对于水平触发(Level Triggered)来说的。  
EPOLLONESHOT：只监听一次事件，当监听完这次事件之后，如果还需要继续监听这个socket的话，需要再次把这个socket加入到EPOLL队列里

**（3） int epoll\_wait(int epfd, struct epoll\_event \* events, int maxevents, int timeout);**  
　　等待事件的产生，类似于select()调用。参数events用来从内核得到事件的集合，maxevents告之内核这个events有多大，这个maxevents的值不能大于创建epoll\_create()时的size，参数timeout是超时时间（毫秒，0会立即返回，-1将不确定，也有说法说是永久阻塞）。该函数返回需要处理的事件数目，如返回0表示已超时。

**3、工作模式**

　　epoll对文件描述符的操作有两种模式：LT（level trigger）和ET（edge trigger）。LT模式是默认模式，LT模式与ET模式的区别如下：

　　LT模式：**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。**

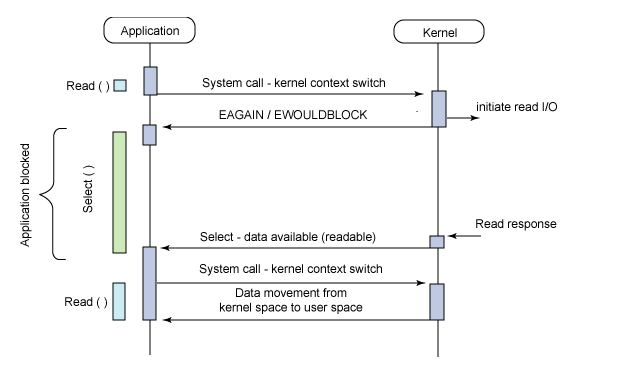
　　ET模式：**当epoll\_wait检测到描述符事件发生并将此事件通知应用程序，应用程序必须立即处理该事件。如果不处理，下次调用epoll\_wait时，不会再次响应应用程序并通知此事件。**

**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。**

### [select、poll、epoll之间的区别总结](http://www.cnblogs.com/Anker/p/3265058.html)

**1、select实现**

**select的调用过程如下所示：**

****

（1）使用copy\_from\_user从用户空间拷贝fd\_set到内核空间

（2）注册回调函数\_\_pollwait

（3）遍历所有fd，调用其对应的poll方法（对于socket，这个poll方法是sock\_poll，sock\_poll根据情况会调用到tcp\_poll,udp\_poll或者datagram\_poll）

（4）以tcp\_poll为例，其核心实现就是\_\_pollwait，也就是上面注册的回调函数。

（5）\_\_pollwait的主要工作就是把current（当前进程）挂到设备的等待队列中，不同的设备有不同的等待队列，对于tcp\_poll来说，其等待队列是sk->sk\_sleep（注意把进程挂到等待队列中并不代表进程已经睡眠了）。在设备收到一条消息（网络设备）或填写完文件数据（磁盘设备）后，会唤醒设备等待队列上睡眠的进程，这时current便被唤醒了。

（6）poll方法返回时会返回一个描述读写操作是否就绪的mask掩码，根据这个mask掩码给fd\_set赋值。

（7）如果遍历完所有的fd，还没有返回一个可读写的mask掩码，则会调用schedule\_timeout是调用select的进程（也就是current）进入睡眠。当设备驱动发生自身资源可读写后，会唤醒其等待队列上睡眠的进程。如果超过一定的超时时间（schedule\_timeout指定），还是没人唤醒，则调用select的进程会重新被唤醒获得CPU，进而重新遍历fd，判断有没有就绪的fd。

（8）把fd\_set从内核空间拷贝到用户空间。

**总结：**

**select的几大缺点：**

**（1）每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大**

**（2）同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大**

**（3）select支持的文件描述符数量太小了，默认是1024**

**2 poll实现**

　　poll的实现和select非常相似，只是描述fd集合的方式不同，poll使用pollfd结构而不是select的fd\_set结构，其他的都差不多。

关于select和poll的实现分析，可以参考下面几篇博文：

<http://blog.csdn.net/lizhiguo0532/article/details/6568964#comments>

<http://blog.csdn.net/lizhiguo0532/article/details/6568968>

<http://blog.csdn.net/lizhiguo0532/article/details/6568969>

<http://www.ibm.com/developerworks/cn/linux/l-cn-edntwk/index.html?ca=drs->

<http://linux.chinaunix.net/techdoc/net/2009/05/03/1109887.shtml>

**3、epoll**

　　epoll既然是对select和poll的改进，就应该能避免上述的三个缺点。那epoll都是怎么解决的呢？在此之前，我们先看一下epoll和select和poll的调用接口上的不同，select和poll都只提供了一个函数——select或者poll函数。而epoll提供了三个函数，epoll\_create,epoll\_ctl和epoll\_wait，epoll\_create是创建一个epoll句柄；epoll\_ctl是注册要监听的事件类型；epoll\_wait则是等待事件的产生。

　　对于第一个缺点，epoll的解决方案在epoll\_ctl函数中。每次注册新的事件到epoll句柄中时（在epoll\_ctl中指定EPOLL\_CTL\_ADD），会把所有的fd拷贝进内核，而不是在epoll\_wait的时候重复拷贝。epoll保证了每个fd在整个过程中只会拷贝一次。

　　对于第二个缺点，epoll的解决方案不像select或poll一样每次都把current轮流加入fd对应的设备等待队列中，而只在epoll\_ctl时把current挂一遍（这一遍必不可少）并为每个fd指定一个回调函数，当设备就绪，唤醒等待队列上的等待者时，就会调用这个回调函数，而这个回调函数会把就绪的fd加入一个就绪链表）。epoll\_wait的工作实际上就是在这个就绪链表中查看有没有就绪的fd（利用schedule\_timeout()实现睡一会，判断一会的效果，和select实现中的第7步是类似的）。

　　对于第三个缺点，epoll没有这个限制，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。

**总结：**

（1）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。

（2）select，poll每次调用都要把fd集合从用户态往内核态拷贝一次，并且要把current往设备等待队列中挂一次，而epoll只要一次拷贝，而且把current往等待队列上挂也只挂一次（在epoll\_wait的开始，注意这里的等待队列并不是设备等待队列，只是一个epoll内部定义的等待队列）。这也能节省不少的开销。

简短的总结一下：

基本上select有3个缺点:

1. 连接数受限
2. 查找配对速度慢
3. 数据由内核拷贝到用户态

poll改善了第一个缺点

epoll改了三个缺点.

### 关于epoll

现如今，网络通讯中用[epoll](http://www.cppfans.org/tag/epoll)(linux)和IOCP(windows)几乎是大家津津乐道的东西，不为别的，就因为高效，所以大家喜欢用。IOCP的基础东西已经讲过了，可翻阅[《IOCP浅析》](http://www.cppfans.org/1054.html) [《IOCP浅析[二]——IOCP出现的意义和函数接口》](http://www.cppfans.org/1089.html).

**什么是**[**epoll**](http://www.cppfans.org/tag/epoll)**？**

epoll是[Linux](http://www.cppfans.org/tag/linux)下多路复用IO接口select/poll的增强版本，它能显著提高程序在大量并发连接中只有少量活跃的情况下的系统CPU利用率，因为它会复用文件描述符集 合来传递结果而不用迫使开发者每次等待事件之前都必须重新准备要被侦听的文件描述符集合，另一点原因就是获取事件的时候，它无须遍历整个被侦听的描述符 集，只要遍历那些被内核IO事件异步唤醒而加入Ready队列的描述符集合就行了。epoll除了提供select/poll那种IO事件的电平触发 （Level Triggered）外，还提供了边沿触发（Edge Triggered），这就使得用户空间程序有可能缓存IO状态，减少epoll\_wait/epoll\_pwait的调用，提高应用程序效率。[Linux](http://www.cppfans.org/tag/linux)2.6内核中对/dev/epoll设备的访问的封装（system epoll）。

这个使我们开发网络应用程序更加简单，并且更加高效。

**为什么要使用epoll？**

同样，我们在linux系统下，影响效率的依然是I/O操作，linux提供给我们select/poll/epoll等多路复用I/O方式*(kqueue暂时没研究过)*，为什么我们对epoll情有独钟呢？原因如下：

1.文件描述符数量的对比。

epoll并没有fd(文件描述符)的上限，它只跟系统内存有关，我的2G的ubuntu下查看是20480个，轻松支持20W个fd。可使用如下命令查看：

cat /proc/sys/fs/file-max

再来看select/poll，有一个限定的fd的数量，linux/posix\_types.h头文件中

#define \_\_FD\_SETSIZE    1024

2.效率对比。

当然了，你可以修改上述值，然后重新编译内核，然后再次写代码，这也是没问题的，不过我先说说select/poll的机制，估计你马上会作废上面修改枚举值的想法。

select/poll会因为监听fd的数量而导致效率低下，因为它是轮询所有fd，有数据就处理，没数据就跳过，所以fd的数量会降低效率；而epoll只处理就绪的fd，它有一个就绪设备的队列，每次只轮询该队列的数据，然后进行处理。*(先简单讲一下，第二篇还会详细讲解)*

3.内存处理方式对比。

不管是哪种I/O机制，都无法避免fd在操作过程中拷贝的问题，而epoll使用了mmap(是指文件/对象的内存映射，被映射到多个内存页上)，所以同一块内存就可以避免这个问题。

btw:TCP/IP协议栈使用内存池管理sk\_buff结构，你还可以通过修改内存池pool的大小，毕竟linux支持各种微调内核。

**epoll的工作方式**

epoll分为两种工作方式LT和ET。

LT(level triggered) 是默认/缺省的工作方式，同时支持 block和no\_block socket。这种工作方式下，内核会通知你一个fd是否就绪，然后才可以对这个就绪的fd进行I/O操作。就算你没有任何操作，系统还是会继续提示fd已经就绪，不过这种工作方式出错会比较小，传统的select/poll就是这种工作方式的代表。

ET(edge-triggered) 是高速工作方式，仅支持no\_block socket，这种工作方式下，当fd从未就绪变为就绪时，内核会通知fd已经就绪，并且内核认为你知道该fd已经就绪，不会再次通知了，除非因为某些操作导致fd就绪状态发生变化。如果一直不对这个fd进行I/O操作，导致fd变为未就绪时，内核同样不会发送更多的通知，因为only once。所以这种方式下，出错率比较高，需要增加一些检测程序。

**LT可以理解为水平触发，只要有数据可以读，不管怎样都会通知。而ET为边缘触发，只有状态发生变化时才会通知，可以理解为电平变化**。

**如何使用epoll？**

使用epoll很简单，只需要

#include <sys/epoll.h>

有三个关键函数：

int epoll\_create(int size);

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

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

当然了，不要忘记关闭函数.

============分割线==============

这篇就讲到这里了，下面两篇主要是函数介绍，效率分析，例子。

转载请注明：[C++爱好者博客](http://www.cppfans.org/) » [浅析epoll-为何多路复用I/O要使用epoll](http://www.cppfans.org/1417.html)

前一篇大致讲了一下[epoll](http://www.cppfans.org/tag/epoll)是个什么东西，优点等内容，这篇延续上一篇的内容，主要是分析[epoll](http://www.cppfans.org/tag/epoll)的函数，epoll[高性能](http://www.cppfans.org/tag/%e9%ab%98%e6%80%a7%e8%83%bd)的深入分析。

**epoll的三大函数**

1.创建epoll fd函数

int epoll\_create(int size);

epoll\_create()创建一个epoll的事例，通知内核需要监听size个fd。size指的并不是最大的后备存储设备，而是衡量内核内部结构大小的一个提示。当创建成功后，会占用一个fd，所以记得在使用完之后调用close()，否则fd可能会被耗尽。

Note:自从[Linux](http://www.cppfans.org/tag/linux)2.6.8版本以后，size值其实是没什么用的，不过要大于0，因为内核可以动态的分配大小，所以不需要size这个提示了。

创建还有另外一个函数

int epoll\_create1(int flag);

这个函数是在linux 2.6.27中加入的，当你在看陈硕的muduo时可以看到这个函数，其实它和epoll\_create差不多，不同的是epoll\_create1函数的参数是flag，当flag是0时，表示和epoll\_create函数完全一样，不需要size的提示了。

当flag = EPOLL\_CLOEXEC，创建的epfd会设置FD\_CLOEXEC

当flag = EPOLL\_NONBLOCK，创建的epfd会设置为非阻塞

一般用法都是使用EPOLL\_CLOEXEC.

Note:关于FD\_CLOEXEC，现在网上好多都说的有点问题，我翻阅了一些资料，请教了一些人，大约明白它的意思了。

它是fd的一个标识说明，用来设置文件close-on-exec状态的。当close-on-exec状态为0时，调用exec时，fd**不会**被关闭；状态非零时则**会**被关闭，这样做可以防止fd泄露给执行exec后的进程。关于exec的用法，大家可以去自己查阅下，或者直接man exec。

2.epoll事件的注册函数

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

select是在监听时告诉内核要监听的事件，而epoll\_ctl是先注册需要监听的事件。

第一个参数epfd，为epoll\_create返回的的epoll fd。

第二个参数op表示操作值。有三个操作类型，

EPOLL\_CTL\_ADD  // 注册目标fd到epfd中，同时关联内部event到fd上

EPOLL\_CTL\_MOD // 修改已经注册到fd的监听事件

EPOLL\_CTL\_DEL // 从epfd中删除/移除已注册的fd，event可以被忽略，也可以为NULL

第三个参数fd表示需要监听的fd。

第四个参数event表示需要监听的事件。

typedef union epoll\_data {

void        \*ptr;

int          fd;

uint32\_t     u32;

uint64\_t     u64;

} epoll\_data\_t;

struct epoll\_event {

uint32\_t     events;      /\* Epoll events \*/

epoll\_data\_t data;        /\* User data variable \*/

};

event参数是一个枚举的集合，可以用” | “来增加事件类型，枚举如下：

EPOLLIN:表示关联的fd可以进行读操作了。  
EPOLLOUT:表示关联的fd可以进行写操作了。  
EPOLLRDHUP(since [Linux](http://www.cppfans.org/tag/linux) 2.6.17):表示套接字关闭了连接，或者关闭了正写一半的连接。  
EPOLLPRI:表示关联的fd有紧急优先事件可以进行读操作了。  
EPOLLERR:表示关联的fd发生了错误，epoll\_wait会一直等待这个事件，所以一般没必要设置这个属性。  
EPOLLHUP:表示关联的fd挂起了，epoll\_wait会一直等待这个事件，所以一般没必要设置这个属性。  
EPOLLET:设置关联的fd为ET的工作方式，epoll的默认工作方式是LT。  
EPOLLONESHOT (since Linux 2.6.2):设置关联的fd为one-shot的工作方式。表示只监听一次事件，如果要再次监听，需要把socket放入到epoll队列中。

3.epoll等待事件函数

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);

上面两个函数的参数含义：

第一个参数:表示epoll\_wait等待epfd上的事件

第二个参数:events指针携带有epoll\_data\_t数据

第三个参数:maxevents告诉内核events有多大，该值必须大于0

第四个参数:timeout表示超时时间(单位：毫秒)

epoll\_pwait(since linux 2.6.19)允许一个应用程序安全的等待，直到fd设备准备就绪，或者捕获到一个信号量。其中sigmask表示要捕获的信号量。

函数如果等待成功，则返回fd的数字；0表示等待fd超时，其他错误号请查看errno

函数到这里就讲完了，下一篇会写一个例子给大家看下这些函数是如何使用的。

============

epoll支持水平触发和边缘触发，理论上来说边缘触发性能更高，但是使用更加复杂，因为任何意外的丢失事件都会造成请求处理错误。Nginx就使用了epoll的边缘触发模型。

这里提一下水平触发和边缘触发就绪通知的区别，这两个词来源于计算机硬件设计。它们的区别是只要句柄满足某种状态，水平触发就会发出通知；而只有当句柄状态改变时，边缘触发才会发出通知。例如一个socket经过长时间等待后接收到一段100k的数据，两种触发方式都会向程序发出就绪通知。假设程序从这个socket中读取了50k数据，并再次调用监听函数，水平触发依然会发出就绪通知，而边缘触发会因为socket“有数据可读”这个状态没有发生变化而不发出通知且陷入长时间的等待。

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netstat 的recv-q会持续增加

===============

通常来说，et方式是比较危险的方式，如果要使用et方式，那么，应用程序应该 1、将socket设置为non-blocking方式 2、epoll\_wait收到event后，read或write需要读到没有数据为止，write需要写到没有数据为止（对于non-blocking socket来说，EAGAIN通常是无数据可读，无数据可写的返回状态）；

我们最近遇到一个问题，就是由于在使用epoll的过程中，缓冲区的数据没有读完，造成后续的通信失败。

表现现象就是，使用netstat -an观察时，这个socket的recv-q值不为0.

## 调度算法

1. **先来先服务**(FCFS, First Come First Serve)

先来先服务(FCFS)调度算法是一种最简单的调度算法，该算法既可用于作业调度，也可用于进程调度。当在作业调度中采用该算法时，每次调度都是从后备作业队列中选择一个或多个最先进入该队列的作业，将它们调入内存，为它们分配资源、创建进程，然后放入就绪队列。在进程调度中采用FCFS算法时，则每次调度是从就绪队列中选择一个最先进入该队列的进程，为之分配处理机，使之投入运行。该进程一直运行到完成或发生某事件而阻塞后才放弃处理机。

缺点：比较有利于长作业，而不利于短作业。 有利于CPU繁忙的作业，而不利于I/O繁忙的作业。

1. **短作业优先**(SJF, Shortest Job First)

最短优先调度算法是指对短作业或短进程优先调度的算法。它们可以分别用于作业调度和进程调度。短作业优先(SJF)的调度算法是从后备队列中选择一个或若干个估计运行时间最短的作业，将它们调入内存运行。而短进程优先(SPF)调度算法则是从就绪队列中选出一个估计运行时间最短的进程，将处理机分配给它，使它立即执行并一直执行到完成，或发生某事件而被阻塞放弃处理机时再重新调度。

缺点：长作业的运行得不到保证。

**3.1高优先权优先调度算法-（优先权调度算法的类型）**

为了照顾紧迫型作业，使之在进入系统后便获得优先处理，引入了最高优先权优先(FPF)调度算法。此算法常被用于批处理系统中，作为作业调度算法，也作为多种操作系统中的进程调度算法，还可用于实时系统中。当把该算法用于作业调度时，系统将从后备队列中选择若干个优先权最高的作业装入内存。当用于进程调度时，该算法是把处理机分配给就绪队列中优先权最高的进程，这时，又可进一步把该算法分成如下两种。

1) 非抢占式优先权算法

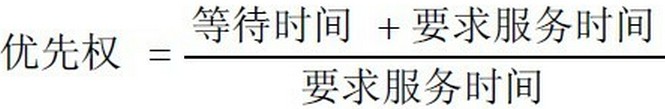
在这种方式下，系统一旦把处理机分配给就绪队列中优先权最高的进程后，该进程便一直执行下去，直至完成；或因发生某事件使该进程放弃处理机时，系统方可再将处理机重新分配给另一优先权最高的进程。这种调度算法主要用于批处理系统中；也可用于某些对实时性要求不严的实时系统中。

2) 抢占式优先权调度算法

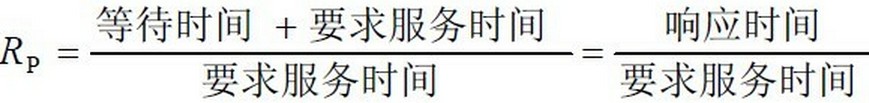
在这种方式下，系统同样是把处理机分配给优先权最高的进程，使之执行。但在其执行期间，只要又出现了另一个其优先权更高的进程，进程调度程序就立即停止当前进程(原优先权最高的进程)的执行，重新将处理机分配给新到的优先权最高的进程。因此，在采用这种调度算法时，是每当系统中出现一个新的就绪进程i 时，就将其优先权Pi与正在执行的进程j 的优先权Pj进行比较。如果Pi≤Pj，原进程Pj便继续执行；但如果是Pi>Pj，则立即停止Pj的执行，做进程切换，使i 进程投入执行。显然，这种抢占式的优先权调度算法能更好地满足紧迫作业的要求，故而常用于要求比较严格的实时系统中，以及对性能要求较高的批处理和分时系统中。

**3.2高优先权优先调度算法-（高响应比优先调度算法）**

在批处理系统中，短作业优先算法是一种比较好的算法，其主要的不足之处是长作业的运行得不到保证。如果我们能为每个作业引入前面所述的动态优先权，并使作业的优先级随着等待时间的增加而以速率a 提高，则长作业在等待一定的时间后，必然有机会分配到处理机。该优先权的变化规律可描述为：



由于等待时间与服务时间之和就是系统对该作业的响应时间，故该优先权又相当于响应比RP。据此，又可表示为：



由上式可以看出：

(1) 如果作业的等待时间相同，则要求服务的时间愈短，其优先权愈高，因而该算法有利于短作业。

(2) 当要求服务的时间相同时，作业的优先权决定于其等待时间，等待时间愈长，其优先权愈高，因而它实现的是先来先服务。

(3) 对于长作业，作业的优先级可以随等待时间的增加而提高，当其等待时间足够长时，其优先级便可升到很高，从而也可获得处理机。

简言之，该算法既照顾了短作业，又考虑了作业到达的先后次序，不会使长作业长期得不到服务。因此，该算法实现了一种较好的折衷。当然，在利用该算法时，每要进行调度之前，都须先做响应比的计算，这会增加系统开销。

**4.1基于时间片的轮转调度算法-（时间片轮转法）**

在早期的时间片轮转法中，系统将所有的就绪进程按先来先服务的原则排成一个队列，每次调度时，把CPU 分配给队首进程，并令其执行一个时间片。时间片的大小从几ms 到几百ms。当执行的时间片用完时，由一个计时器发出时钟中断请求，调度程序便据此信号来停止该进程的执行，并将它送往就绪队列的末尾；然后，再把处理机分配给就绪队列中新的队首进程，同时也让它执行一个时间片。这样就可以保证就绪队列中的所有进程在一给定的时间内均能获得一时间片的处理机执行时间。换言之，系统能在给定的时间内响应所有用户的请求。

**4.2基于时间片的轮转调度算法-（多级反馈队列调度算法）**

前面介绍的各种用作进程调度的算法都有一定的局限性。如短进程优先的调度算法，仅照顾了短进程而忽略了长进程，而且如果并未指明进程的长度，则短进程优先和基于进程长度的抢占式调度算法都将无法使用。而多级反馈队列调度算法则不必事先知道各种进程所需的执行时间，而且还可以满足各种类型进程的需要，因而它是目前被公认的一种较好的进程调度算法。在采用多级反馈队列调度算法的系统中，调度算法的实施过程如下所述。

(1) 应设置多个就绪队列，并为各个队列赋予不同的优先级。第一个队列的优先级最高，第二个队列次之，其余各队列的优先权逐个降低。该算法赋予各个队列中进程执行时间片的大小也各不相同，在优先权愈高的队列中，为每个进程所规定的执行时间片就愈小。例如，第二个队列的时间片要比第一个队列的时间片长一倍，……，第i+1个队列的时间片要比第i个队列的时间片长一倍。

(2) 当一个新进程进入内存后，首先将它放入第一队列的末尾，按FCFS原则排队等待调度。当轮到该进程执行时，如它能在该时间片内完成，便可准备撤离系统；如果它在一个时间片结束时尚未完成，调度程序便将该进程转入第二队列的末尾，再同样地按FCFS原则等待调度执行；如果它在第二队列中运行一个时间片后仍未完成，再依次将它放入第三队列，……，如此下去，当一个长作业(进程)从第一队列依次降到第n队列后，在第n 队列便采取按时间片轮转的方式运行。

(3) 仅当第一队列空闲时，调度程序才调度第二队列中的进程运行；仅当第1～(i-1)队列均空时，才会调度第i队列中的进程运行。如果处理机正在第i队列中为某进程服务时，又有新进程进入优先权较高的队列(第1～(i-1)中的任何一个队列)，则此时新进程将抢占正在运行进程的处理机，即由调度程序把正在运行的进程放回到第i队列的末尾，把处理机分配给新到的高优先权进程。

**5. 电梯调度算法**

高层建筑中电梯请求不断地到来，控制电梯的计算机能够很容易地跟踪顾客按下请求的顺序。如果使用先来先服务算法调度，同时如果电梯负载很重，那么大部分时间电梯将停留在电梯的中部区域，而电梯两端区域的请求将不得不等待，直到负载中的统计波动使得中部区域没有请求位置，这样导致远离中部区域的请求得到的服务很差。因此获得最小响应时间的目标和公平性之间存在着冲突。

大多数电梯使用电梯算法来协调效率和公平性这两个相互冲突的目标。电梯算法电梯保持按一个方向移动，直到在那个方向上没有请求位置，然后改变方向。

电梯算法(elevation algorithm)需要软件维护一个二进制位，即当前方向位：向上(up)或向下(down)。当一个请求处理完成之后，电梯的驱动程序检查该位，如果是up，电梯移至下一个更高的未完成的请求。如果更高的位置没有未完成的请求，则方向位取反。当方向位设置为down时，同时存在一个低位置的请求，则移向该位置。

现在我们明白了，电梯的上下箭头按钮是为了告诉电梯你想向上还是向下去），而不是让电梯向上还是向下。

举例：电梯在上行，5楼有上召和下召。电梯会停5楼，但它是为上召服务的，所以下召灯还会保持点亮。然后启动向上，直到服务完上行的所有请求。转下行，到五楼时还是会停。这时是服务5楼下召的。

电梯处理请求规则：

电梯有移动方向，各楼层的请求有请求方向，这里维护一个请求表（记录请求ID，请求方向，该请求的停靠楼层）。因为电梯会按照移动方向移动，直到该方向没有请求（请求包括请求ID和停靠楼层的请求），所以不会根据请求方向突然改变电梯的移动方向。因此，电梯在移动过程中只处理与“电梯移动方向”相同的“请求方向”的请求。如电梯向下移动，只处理向下的请求，且该请求的方向也向下（停靠楼层请求无方向）。

实时调度算法:

## 死锁

原因:

1. 竞争资源
2. 程序推进顺序不当

必要条件:

1. 互斥条件
2. 请求和保持条件
3. 不剥夺条件
4. 环路等待条件

处理死锁基本方法:

1. 预防死锁(摒弃除1以外的条件)
2. 避免死锁(银行家算法)
3. 检测死锁(资源分配图)
4. 解除死锁
   1. 剥夺资源
   2. 撤销进程

**所谓死锁**：是指两个或两个以上的进程在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。由于资源占用是互斥的，当某个进程提出申请资源后，使得有关进程在无外力协助下，永远分配不到必需的资源而无法继续运行，这就产生了一种特殊现象死锁。

虽然进程在运行过程中，可能发生死锁，但死锁的发生也必须具备一定的条件，死锁的发生必须具备以下四个必要条件。

1）**互斥条件**：指进程对所分配到的资源进行排它性使用，即在一段时间内某资源只由一个进程占用。如果此时还有其它进程请求资源，则请求者只能等待，直至占有资源的进程用毕释放。

2）**请求和保持条件**：指进程已经保持至少一个资源，但又提出了新的资源请求，而该资源已被其它进程占有，此时请求进程阻塞，但又对自己已获得的其它资源保持不放。

3）**不剥夺条件**：指进程已获得的资源，在未使用完之前，不能被剥夺，只能在使用完时由自己释放。

4）**环路等待条件**：指在发生死锁时，必然存在一个进程——资源的环形链，即进程集合{P0，P1，P2，···，Pn}中的P0正在等待一个P1占用的资源；P1正在等待P2占用的资源，……，Pn正在等待已被P0占用的资源。

在系统中已经出现死锁后，应该及时检测到死锁的发生，并采取适当的措施来解除死锁。目前处理死锁的方法可归结为以下四种：

1)**预防死锁。**

　　这是一种较简单和直观的事先预防的方法。方法是通过设置某些限制条件，去破坏产生死锁的四个必要条件中的一个或者几个，来预防发生死锁。预防死锁是一种较易实现的方法，已被广泛使用。但是由于所施加的限制条件往往太严格，可能会导致系统资源利用率和系统吞吐量降低。

2)**避免死锁**。

　　该方法同样是属于事先预防的策略，但它并不须事先采取各种限制措施去破坏产生死锁的的四个必要条件，而是在资源的动态分配过程中，用某种方法去防止系统进入不安全状态，从而避免发生死锁。

3)**检测死锁。**

　　这种方法并不须事先采取任何限制性措施，也不必检查系统是否已经进入不安全区，此方法允许系统在运行过程中发生死锁。但可通过系统所设置的检测机构，及时地检测出死锁的发生，并精确地确定与死锁有关的进程和资源，然后采取适当措施，从系统中将已发生的死锁清除掉。

4)**解除死锁**。

　　这是与检测死锁相配套的一种措施。当检测到系统中已发生死锁时，须将进程从死锁状态中解脱出来。常用的实施方法是撤销或挂起一些进程，以便回收一些资源，再将这些资源分配给已处于阻塞状态的进程，使之转为就绪状态，以继续运行。死锁的检测和解除措施，有可能使系统获得较好的资源利用率和吞吐量，但在实现上难度也最大。

列举说明linux系统的各类异步机制

**死锁**：是指两个或两个以上的进程（或线程）在执行过程中，因争夺资源而造成的一种互相等待的现象，若无外力作用，它们都将无法推进下去。此时称系统处于死锁状态或系统产生了死锁，这些永远在互相等待的进程称为死锁进程。

死锁发生的四个条件

1、互斥条件：线程对资源的访问是排他性的，如果一个线程对占用了某资源，那么其他线程必须处于等待状态，直到资源被释放。

2、请求和保持条件：线程T1至少已经保持了一个资源R1占用,但又提出对另一个资源R2请求，而此时，资源R2被其他线程T2占用，于是该线程T1也必须等待，但又对自己保持的资源R1不释放。

3、不剥夺条件：线程已获得的资源，在未使用完之前，不能被其他线程剥夺，只能在使用完以后由自己释放。

4、环路等待条件：在死锁发生时，必然存在一个“进程-资源环形链”，即：{p0,p1,p2,...pn},进程p0（或线程）等待p1占用的资源，p1等待p2占用的资源，pn等待p0占用的资源。（最直观的理解是，p0等待p1占用的资源，而p1而在等待p0占用的资源，于是两个进程就相互等待）

**活锁**：是指线程1可以使用资源，但它很礼貌，让其他线程先使用资源，线程2也可以使用资源，但它很绅士，也让其他线程先使用资源。这样你让我，我让你，最后两个线程都无法使用资源。

关于“死锁与活锁”的比喻：

死锁：迎面开来的汽车A和汽车B过马路，汽车A得到了半条路的资源（满足死锁发生条件1：资源访问是排他性的，我占了路你就不能上来，除非你爬我头上去），汽车B占了汽车A的另外半条路的资源，A想过去必须请求另一半被B占用的道路（死锁发生条件2：必须整条车身的空间才能开过去，我已经占了一半，尼玛另一半的路被B占用了），B若想过去也必须等待A让路，A是辆兰博基尼，B是开奇瑞QQ的屌丝，A素质比较低开窗对B狂骂：快给老子让开，B很生气，你妈逼的，老子就不让（死锁发生条件3：在未使用完资源前，不能被其他线程剥夺），于是两者相互僵持一个都走不了（死锁发生条件4：环路等待条件），而且导致整条道上的后续车辆也走不了。

例如：马路中间有条小桥，只能容纳一辆车经过，桥两头开来两辆车A和B，A比较礼貌，示意B先过，B也比较礼貌，示意A先过，结果两人一直谦让谁也过不去。

饥饿：是指如果线程T1占用了资源R，线程T2又请求封锁R，于是T2等待。T3也请求资源R，当T1释放了R上的封锁后，系统首先批准了T3的请求，T2仍然等待。然后T4又请求封锁R，当T3释放了R上的封锁之后，系统又批准了T4的请求......，T2可能永远等待。

关于”饥饿“的比喻：

在“首堵”北京的某一天，天气阴沉，空气中充斥着雾霾和地沟油的味道，某个苦逼的临时工交警正在处理塞车，有两条道A和B上都堵满了车辆，其中A道堵的时间最长，B相对相对堵的时间较短，这时，前面道路已疏通，交警按照最佳分配原则，示意B道上车辆先过，B道路上过了一辆又一辆，A道上排队时间最长的确没法通过，只能等B道上没有车辆通过的时候再等交警发指令让A道依次通过，这也就是ReentrantLock显示锁里提供的不公平锁机制（当然了，ReentrantLock也提供了公平锁的机制，由用户根据具体的使用场景而决定到底使用哪种锁策略），不公平锁能够提高吞吐量但不可避免的会造成某些线程的饥饿。

## 4. 虚拟内存技术

虚拟存储器是指具有请求调入功能和置换功能,能从逻辑上对内存容量加以扩充的一种存储系统.

## 5. 分页和分段

分页: 用户程序的地址空间被划分成若干固定大小的区域，称为“页”，相应地，内存空间分成若干个物理块，页和块的大小相等。可将用户程序的任一页放在内存的任一块中，实现了离散分配。

分段: 将用户程序地址空间分成若干个大小不等的段，每段可以定义一组相对完整的逻辑信息。存储分配时，以段为单位，段与段在内存中可以不相邻接，也实现了离散分配。

**分页与分段的主要区别**

1. 页是信息的物理单位,分页是为了实现非连续分配,以便解决内存碎片问题,或者说分页是由于系统管理的需要.段是信息的逻辑单位,它含有一组意义相对完整的信息,分段的目的是为了更好地实现共享,满足用户的需要.
2. 页的大小固定,由系统确定,将逻辑地址划分为页号和页内地址是由机器硬件实现的.而段的长度却不固定,决定于用户所编写的程序,通常由编译程序在对源程序进行编译时根据信息的性质来划分.
3. 分页的作业地址空间是一维的.分段的地址空间是二维的.

# 数据库 MySQL

## 1. 事务

**数据库事务**(Database Transaction) ，是指作为单个逻辑工作单元执行的一系列操作，要么完全地执行，要么完全地不执行。

事务（*transaction*）是数据库管理系统的执行单位，可以是一个数据库操作（如Select操作）或者是一组操作序列。事务ACID属性，即原子性（*Atomicity*）、一致性(*Consistency*)、隔离性（*Isolation*）、持久性（*Durability*）。

**原子性**：保证事务中的所有操作全部执行或全部不执行。例如执行转账事务，要么转账成功，要么失败。成功，则金额从转出帐户转入到目的帐户，并且两个帐户金额将发生相应的变化；失败，则两个账户的金额都不变。不会出现转出帐户扣了钱，而目的帐户没有收到钱的情况。

**一致性**：保证数据库始终保持数据的一致性——事务操作之前是一致的，事务操作之后也是一致的，不管事务成功与否。如上面的例子，转账之前和之后数据库都保持数据上的一致性。

**隔离性**：多个事务并发执行的话，结果应该与多个事务串行执行效果是一样的。显然最简单的隔离就是将所有事务都串行执行：先来先执行，一个事务执行完了才允许执行下一个。但这样数据库的效率低下，如：两个不同的事务只是读取同一批数据，这样完全可以并发进行。为了控制并发执行的效果就有了不同的隔离级别。下面将详细介绍。

**持久性**：持久性表示事务操作完成之后，对数据库的影响是持久的，即使数据库因故障而受到破坏，数据库也应该能够恢复。通常的实现方式是采用日志。

## 2. 事务隔离级别

**事务隔离级别**（*transaction isolation levels*）：隔离级别就是对对事务并发控制的等级。[ANSI](http://en.wikipedia.org/wiki/American_National_Standards_Institute)/ [ISO](http://en.wikipedia.org/wiki/International_Organization_for_Standardization) [SQL](http://en.wikipedia.org/wiki/SQL)将其分为串行化（SERIALIZABLE）、可重复读（REPEATABLE READ）、读已提交（READ COMMITED）、读未提交（READ UNCOMMITED）四个等级。为了实现隔离级别通常数据库采用锁（Lock）。一般在编程的时候只需要设置隔离等级，至于具体采用什么锁则由数据库来设置。首先介绍四种等级，然后举例解释后面三个等级（可重复读、读已提交、读未提交）中会出现的并发问题。

**可串行化**（SERIALIZABLE）：所有事务都一个接一个地串行执行，这样可以避免幻读（phantom reads）。对于基于锁来实现并发控制的数据库来说，串行化要求在执行范围查询（如选取年龄在10到30之间的用户）的时候，需要获取范围锁（range lock）。如果不是基于锁实现并发控制的数据库，则检查到有违反串行操作的事务时，需要滚回该事务。

**可重复读**（REPEATABLE READ）：所有被Select获取的数据都不能被修改，这样就可以避免一个事务前后读取数据不一致的情况。但是却没有办法控制幻读，因为这个时候其他事务不能更改所选的数据，但是可以增加数据，因为前一个事务没有范围锁。Mysql默认的隔离等级

**读已提交**（READ COMMITED）：被读取的数据可以被其他事务修改。这样就可能导致不可重复读。也就是说，事务的读取数据的时候获取读锁，但是读完之后立即释放（不需要等到事务结束），而写锁则是事务提交之后才释放。释放读锁之后，就可能被其他事物修改数据。该等级也是SQL Server默认的隔离等级。

**读未提交**（READ UNCOMMITED）：这是最低的隔离等级，允许其他事务看到没有提交的数据。这种等级会导致脏读（Dirty Read）。

**1.** **脏读** ：脏读就是指当一个事务正在访问数据，并且对数据进行了修改，而这种修改还没有提交到数据库中，这时，另外一个事务也访问这个数据，然后使用了这个数据。

**2.** **不可重复读** ：是指在一个事务内，多次读同一数据。在这个事务还没有结束时，另外一个事务也访问该同一数据。那么，在第一个事务中的两 次读数据之间，由于第二个事务的修改，那么第一个事务两次读到的的数据可能是不一样的。这样就发生了在一个事务内两次读到的数据是不一样的，因此称为是不 可重复读。例如，一个编辑人员两次读取同一文档，但在两次读取之间，作者重写了该文档。当编辑人员第二次读取文档时，文档已更改。原始读取不可重复。如果 只有在作者全部完成编写后编辑人员才可以读取文档，则可以避免该问题。

**3.** **幻读** : 是指当事务不是独立执行时发生的一种现象，例如第一个事务对一个表中的数据进行了修改，这种修改涉及到表中的全部数据行。 同时，第二个事务也修改这个表中的数据，这种修改是向表中插入一行新数据。那么，以后就会发生操作第一个事务的用户发现表中还有没有修改的数据行，就好象 发生了幻觉一样。例如，一个编辑人员更改作者提交的文档，但当生产部门将其更改内容合并到该文档的主复本时，发现作者已将未编辑的新材料添加到该文档中。 如果在编辑人员和生产部门完成对原始文档的处理之前，任何人都不能将新材料添加到文档中，则可以避免该问题。



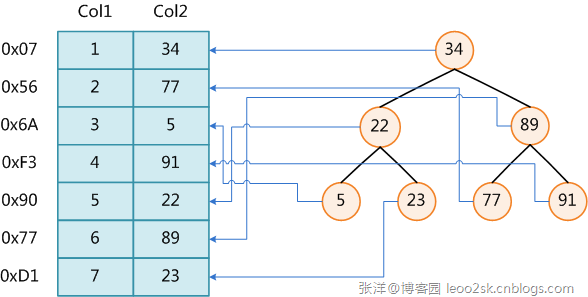
## 3. 数据库索引

**说白了，索引问题就是一个查找问题。。。**

**数据库索引**，是数据库管理系统中一个排序的数据结构，以协助快速查询、更新数据库表中数据。**索引的实现通常使用B树及其变种B+树**。

在数据之外，数据库系统还维护着满足特定查找算法的数据结构，这些数据结构以某种方式引用（指向）数据，这样就可以在这些数据结构上实现高级查找算法。这种数据结构，就是索引。

为表设置索引要付出代价的：一是增加了数据库的存储空间，二是在插入和修改数据时要花费较多的时间(因为索引也要随之变动)。



上图展示了一种可能的索引方式。左边是数据表，一共有两列七条记录，最左边的是数据记录的物理地址（注意逻辑上相邻的记录在磁盘上也并不是一定物理相邻的）。为了加快Col2的查找，可以维护一个右边所示的二叉查找树，每个节点分别包含索引键值和一个指向对应数据记录物理地址的指针，这样就可以运用二叉查找在O(log2n)的复杂度内获取到相应数据。

创建索引可以大大提高系统的性能。

第一，通过创建唯一性索引，可以保证数据库表中每一行数据的唯一性。

第二，可以大大加快数据的检索速度，这也是创建索引的最主要的原因。

第三，可以加速表和表之间的连接，特别是在实现数据的参考完整性方面特别有意义。

第四，在使用分组和排序子句进行数据检索时，同样可以显著减少查询中分组和排序的时间。

第五，通过使用索引，可以在查询的过程中，使用优化隐藏器，提高系统的性能。

也许会有人要问：增加索引有如此多的优点，为什么不对表中的每一个列创建一个索引呢？因为，增加索引也有许多不利的方面。

第一，创建索引和维护索引要耗费时间，这种时间随着数据量的增加而增加。

第二，索引需要占物理空间，除了数据表占数据空间之外，每一个索引还要占一定的物理空间，如果要建立聚簇索引，那么需要的空间就会更大。

第三，当对表中的数据进行增加、删除和修改的时候，索引也要动态的维护，这样就降低了数据的维护速度。

索引是建立在数据库表中的某些列的上面。在创建索引的时候，应该考虑在哪些列上可以创建索引，在哪些列上不能创建索引。**一般来说，应该在这些列上创建索引：**在经常需要搜索的列上，可以加快搜索的速度；在作为主键的列上，强制该列的唯一性和组织表中数据的排列结构；在经常用在连接的列上，这些列主要是一些外键，可以加快连接的速度；在经常需要根据范围进行搜索的列上创建索引，因为索引已经排序，其指定的范围是连续的；在经常需要排序的列上创建索引，因为索引已经排序，这样查询可以利用索引的排序，加快排序查询时间；在经常使用在WHERE子句中的列上面创建索引，加快条件的判断速度。

同样，对于有些列不应该创建索引。**一般来说，不应该创建索引的的这些列具有下列特点：**

第一，对于那些在查询中很少使用或者参考的列不应该创建索引。这是因为，既然这些列很少使用到，因此有索引或者无索引，并不能提高查询速度。相反，由于增加了索引，反而降低了系统的维护速度和增大了空间需求。

第二，对于那些只有很少数据值的列也不应该增加索引。这是因为，由于这些列的取值很少，例如人事表的性别列，在查询的结果中，结果集的数据行占了表中数据行的很大比例，即需要在表中搜索的数据行的比例很大。增加索引，并不能明显加快检索速度。

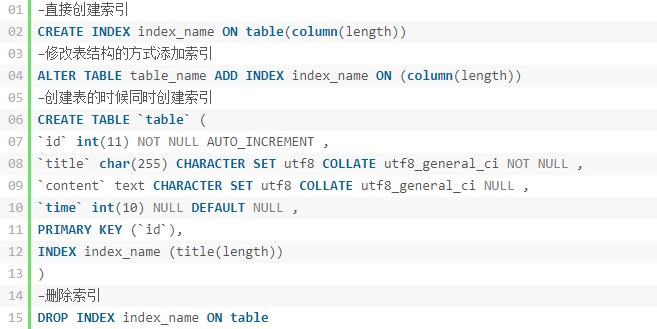
第三，对于那些定义为text, image和bit数据类型的列不应该增加索引。这是因为，这些列的数据量要么相当大，要么取值很少。

第四，当修改性能远远大于检索性能时，不应该创建索引。这是因为，**修改性能和检索性能是互相矛盾的**。当增加索引时，会提高检索性能，但是会降低修改性能。当减少索引时，会提高修改性能，降低检索性能。因此，当修改性能远远大于检索性能时，不应该创建索引。

根据数据库的功能，可以在数据库设计器中创建三种索引：**唯一索引、主键索引和聚集索引**。

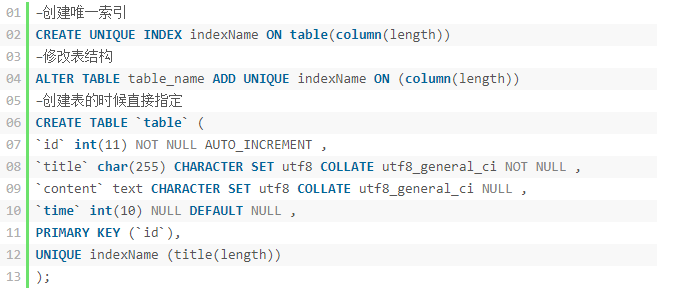
**普通索引：**

1. 普通索引：最基本的MySQL数据库索引，它没有任何限制



**唯一索引**

唯一索引是不允许其中任何两行具有相同索引值的索引。当现有数据中存在重复的键值时，大多数数据库不允许将新创建的唯一索引与表一起保存。数据库还可能防止添加将在表中创建重复键值的新数据。与普通索引类似，不同的就是：索引列的值必须唯一，但允许有空值（注意和主键不同）。如果是组合索引，则列值的组合必须唯一，创建方法和普通索引类似。

**主键索引**

数据库表经常有一列或列组合，其值唯一标识表中的每一行。该列称为表的主键。

在数据库关系图中为表定义主键将自动创建主键索引，主键索引是唯一索引的特定类型。该索引要求主键中的每个值都唯一。当在查询中使用主键索引时，它还允许对数据的快速访问。

CREATE TABLE mytable( ID INT NOT NULL, username VARCHAR(16) NOT NULL,PRIMARY KEY(ID) );

也可以用 ALTER 命令

**全文索引（FULLTEXT）**



**聚集索引**

在聚集索引中，表中行的物理顺序与键值的逻辑（索引）顺序相同。一个表只能包含一个聚集索引。

如果某索引不是聚集索引，则表中行的物理顺序与键值的逻辑顺序不匹配。与非聚集索引相比，聚集索引通常提供更快的数据访问速度。

**局部性原理与磁盘预读**

由于存储介质的特性，磁盘本身存取就比主存慢很多，再加上机械运动耗费，磁盘的存取速度往往是主存的几百分分之一，因此为了提高效率，要尽量减少磁盘I/O。为了达到这个目的，磁盘往往不是严格按需读取，而是每次都会预读，即使只需要一个字节，磁盘也会从这个位置开始，顺序向后读取一定长度的数据放入内存。这样做的理论依据是计算机科学中著名的**局部性原理**：**当一个数据被用到时，其附近的数据也通常会马上被使用。程序运行期间所需要的数据通常比较集中。**

由于磁盘顺序读取的效率很高（不需要寻道时间，只需很少的旋转时间），因此对于具有局部性的程序来说，预读可以提高I/O效率。

预读的长度一般为页（page）的整倍数。页是计算机管理存储器的逻辑块，硬件及操作系统往往将主存和磁盘存储区分割为连续的大小相等的块，每个存储块称为一页（在许多操作系统中，页得大小通常为4k），主存和磁盘以页为单位交换数据。当程序要读取的数据不在主存中时，会触发一个缺页异常，此时系统会向磁盘发出读盘信号，磁盘会找到数据的起始位置并向后连续读取一页或几页载入内存中，然后异常返回，程序继续运行。

**B-/+Tree索引的性能分析**

到这里终于可以分析B-/+Tree索引的性能了。

上文说过一般使用磁盘I/O次数评价索引结构的优劣。先从B-Tree分析，根据B-Tree的定义，可知检索一次最多需要访问h个节点。数据库系统的设计者巧妙利用了磁盘预读原理，将一个节点的大小设为等于一个页，这样每个节点只需要一次I/O就可以完全载入。为了达到这个目的，在实际实现B-Tree还需要使用如下技巧：

每次新建节点时，直接申请一个页的空间，这样就保证一个节点物理上也存储在一个页里，加之计算机存储分配都是按页对齐的，就实现了一个node只需一次I/O。

**B-Tree中一次检索最多需要h-1次I/O（根节点常驻内存），渐进复杂度为O(h)=O(logdN)。**一般实际应用中，出度d是非常大的数字，通常超过100，因此h非常小（通常不超过3）。

而红黑树这种结构，h明显要深的多。由于逻辑上很近的节点（父子）物理上可能很远，无法利用局部性，所以红黑树的I/O渐进复杂度也为O(h)，效率明显比B-Tree差很多。

**综上所述，用B-Tree作为索引结构效率是非常高的。**

**应该花时间学习B-树和B+树数据结构**

=============================================================================================================

1）B树

B树中每个节点包含了键值和键值对于的数据对象存放地址指针，所以成功搜索一个对象可以不用到达树的叶节点。

成功搜索包括节点内搜索和沿某一路径的搜索，成功搜索时间取决于关键码所在的层次以及节点内关键码的数量。

在B树中查找给定关键字的方法是：首先把根结点取来，在根结点所包含的关键字K1,…,kj查找给定的关键字（可用顺序查找或二分查找法），若找到等于给定值的关键字，则查找成功；否则，一定可以确定要查的关键字在某个Ki或Ki+1之间，于是取Pi所指的下一层索引节点块继续查找，直到找到，或指针Pi为空时查找失败。

2）B+树

B+树非叶节点中存放的关键码并不指示数据对象的地址指针，非也节点只是索引部分。所有的叶节点在同一层上，包含了全部关键码和相应数据对象的存放地址指针，且叶节点按关键码从小到大顺序链接。如果实际数据对象按加入的顺序存储而不是按关键码次数存储的话，叶节点的索引必须是稠密索引，若实际数据存储按关键码次序存放的话，叶节点索引时稀疏索引。B+树有2个头指针，一个是树的根节点，一个是最小关键码的叶节点。

所以 B+树有两种搜索方法：

一种是按叶节点自己拉起的链表顺序搜索。

一种是从根节点开始搜索，和B树类似，不过如果非叶节点的关键码等于给定值，搜索并不停止，而是继续沿右指针，一直查到叶节点上的关键码。所以无论搜索是否成功，都将走完树的所有层。

B+ 树中，数据对象的插入和删除仅在叶节点上进行。

这两种处理索引的数据结构的不同之处：

a，B树中同一键值不会出现多次，并且它有可能出现在叶结点，也有可能出现在非叶结点中。而B+树的键一定会出现在叶结点中，并且有可能在非叶结点中也有可能重复出现，以维持B+树的平衡。

b，因为B树键位置不定，且在整个树结构中只出现一次，虽然可以节省存储空间，但使得在插入、删除操作复杂度明显增加。B+树相比来说是一种较好的折中。

c，B树的查询效率与键在树中的位置有关，最大时间复杂度与B+树相同(在叶结点的时候)，最小时间复杂度为1(在根结点的时候)。而B+树的时候复杂度对某建成的树是固定的。

## 4. Mysql

查询语句书写顺序：select – from- where- group by- having- order by-limit

查询语句执行顺序：from - where -group by - having - select - order by-limit

**4.**

[DDL](https://www.baidu.com/s?wd=DDL&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvmvnsnW03nHTLPyuBuWD0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtYPHbdrH6LPj6)(data definition language)：数据库模式定义语言，关键字：create  
[DML](https://www.baidu.com/s?wd=DML&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvmvnsnW03nHTLPyuBuWD0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtYPHbdrH6LPj6)(data manipulation language)：数据操纵语言，关键字：Insert、delete、update  
[DCL](https://www.baidu.com/s?wd=DCL&tn=44039180_cpr&fenlei=mv6quAkxTZn0IZRqIHckPjm4nH00T1YvmvnsnW03nHTLPyuBuWD0IAYqnWm3PW64rj0d0AP8IA3qPjfsn1bkrjKxmLKz0ZNzUjdCIZwsrBtEXh9GuA7EQhF9pywdQhPEUiqkIyN1IA-EUBtYPHbdrH6LPj6)(Data Control Language)：数据库控制语言 ，关键字：grant、remove  
DQL：数据库查询语言，关键字：select

## 5. [union和union all的区别](http://www.cnblogs.com/xiangshu/articles/2054447.html)

**Union因为要进行重复值扫描，所以效率低。如果合并没有刻意要删除重复行，那么就使用Union All**

相同个数的列，并且每个列的类型是一样的。但列名则不一定需要相同

union 和 union all都可以将多个结果集合并  
Union：对两个结果集进行并集操作，不包括重复行，同时进行默认规则的排序；   
Union All：对两个结果集进行并集操作，包括重复行，不进行排序；   
Intersect：对两个结果集进行交集操作，不包括重复行，同时进行默认规则的排序；   
Minus：对两个结果集进行差操作，不包括重复行，同时进行默认规则的排序。   
可以在最后一个结果集中指定Order by子句改变排序方式。

## 6. where与having

where针对表中的列发挥作用，查询数据

having是对查询结果中得到的列发挥作用，筛选数据(也就是说从查询出的结果集再次进行筛选

## 7. 内连接,外连接

**内连接**

连接的数据表相对应的匹配字段完全相等的连接。连接关键字是 inner join

**外连接**

分为左外连接与右外连接、全连接。

左连接的结果集包括指定的左表全部数据与匹配的右表数据,右表中没匹配的全为空值.关键字 left join

右连接的结果集包含指定的右表全部数据与匹配的左边数据,左边中没匹配的全为空值.关键字 right join

**全连接**返回左右数据表的所有行.关键字 full join

自然连接，不需要人为指定连接字段，自然会自动找同名字段进行连接，会删除连接后的重复列。 关键字 natural

生成笛卡尔积－它不使用任何匹配或者选取条件，而是直接将一个数据源中的每个行与另一个数据源的每个

效率问题:

1.inner join比left join快 注:inner join  内连接等价于下面的sql: SELECT A.name, B.address FROM A, B WHERE A.id = B.A\_id 所以一般要用一般的连接就可以了.

2.连接字段建索引

# Redis原理

**Redis是什么？**

1. 是一个完全开源免费的key-value内存数据库
2. 通常被认为是一个数据结构服务器，主要是因为其有着丰富的数据结构 strings、map、 list、sets、 sorted sets

**Redis数据库**

​ 通常局限点来说，Redis也以消息队列的形式存在，作为内嵌的List存在，满足实时的高并发需求。在使用缓存的时候，redis比memcached具有更多的优势，并且支持更多的数据类型，把redis当作一个中间存储系统，用来处理高并发的数据库操作

* **速度快**：使用标准C写，所有数据都在内存中完成，读写速度分别达到10万/20万
* **持久化**：对数据的更新采用Copy-on-write技术，可以异步地保存到磁盘上，主要有两种策略，一是根据时间，更新次数的快照（save 300 10 ）二是基于语句追加方式(Append-only file，aof)
* **自动操作**：对不同数据类型的操作都是自动的，很安全
* **快速的主--从复制**，官方提供了一个数据，Slave在21秒即完成了对Amazon网站10G key set的复制。
* **Sharding技术**： 很容易将数据分布到多个Redis实例中，数据库的扩展是个永恒的话题，在关系型数据库中，主要是以添加硬件、以分区为主要技术形式的纵向扩展解决了很多的应用场景，但随着web2.0、移动互联网、云计算等应用的兴起，这种扩展模式已经不太适合了，所以近年来，像采用主从配置、数据库复制形式的，Sharding这种技术把负载分布到多个特理节点上去的横向扩展方式用处越来越多。

**Redis缺点**

* 是数据库容量受到物理内存的限制,不能用作海量数据的高性能读写,因此Redis适合的场景主要局限在较小数据量的高性能操作和运算上。
* Redis较难支持在线扩容，在集群容量达到上限时在线扩容会变得很复杂。为避免这一问题，运维人员在系统上线时必须确保有足够的空间，这对资源造成了很大的浪费。

# 乐观锁与悲观锁

Web应用往往面临多用户环境，这种情况下的并发写入控制， 几乎成为每个开发人员都必须掌握的一项技能。

在并发环境下，有可能会出现脏读（Dirty Read）、不可重复读（Unrepeatable Read）、 幻读（Phantom Read）、更新丢失（Lost update）等情况。具体的表现可以自行搜索。

为了应对这些问题，主流数据库都提供了锁机制，并引入了事务隔离级别的概念。 这里我们都不作解释了，拿这些关键词一搜，网上大把大把的。

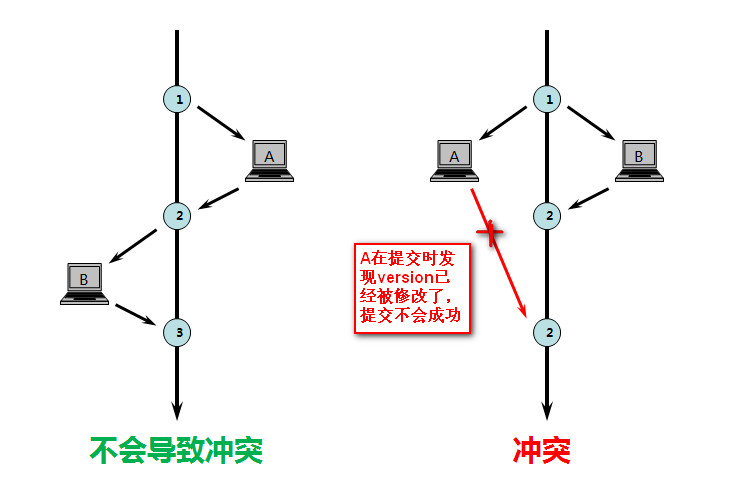
但是，就于具体开发过程而言，一般分为悲观锁和乐观锁两种方式来解决并发冲突问题。

## 乐观锁

**乐观锁介绍：**

乐观锁（ Optimistic Locking ） 相对悲观锁而言，乐观锁假设认为数据一般情况下不会造成冲突，所以在数据进行提交更新的时候，才会正式对数据的冲突与否进行检测，如果发现冲突了，则让返回用户错误的信息，让用户决定如何去做。那么我们如何实现乐观锁呢，一般来说有以下2种方式：

1.使用数据版本（Version）记录机制实现，这是乐观锁最常用的一种实现方式。何谓数据版本？即为数据增加一个版本标识，一般是通过为数据库表增加一个数字类型的 “version” 字段来实现。当读取数据时，将version字段的值一同读出，数据每更新一次，对此version值加一。当我们提交更新的时候，判断数据库表对应记录的当前版本信息与第一次取出来的version值进行比对，如果数据库表当前版本号与第一次取出来的version值相等，则予以更新，否则认为是过期数据。用下面的一张图来说明

：

如上图所示，如果更新操作顺序执行，则数据的版本（version）依次递增，不会产生冲突。但是如果发生有不同的业务操作对同一版本的数据进行修改，那么，先提交的操作（图中B）会把数据version更新为2，当A在B之后提交更新时发现数据的version已经被修改了，那么A的更新操作会失败。

2.乐观锁定的第二种实现方式和第一种差不多，同样是在需要乐观锁控制的table中增加一个字段，名称无所谓，字段类型使用时间戳（timestamp）, 和上面的version类似，也是在更新提交的时候检查当前数据库中数据的时间戳和自己更新前取到的时间戳进行对比，如果一致则OK，否则就是版本冲突。

## 悲观锁介绍（百科）：

悲观锁，正如其名，它指的是对数据被外界（包括本系统当前的其他事务，以及来自外部系统的事务处理）修改持保守态度，因此，在整个数据处理过程中，将数据处于锁定状态。悲观锁的实现，往往依靠数据库提供的锁机制（也只有数据库层提供的锁机制才能真正保证数据访问的排他性，否则，即使在本系统中实现了加锁机制，也无法保证外部系统不会修改数据）。

**使用场景举例**：以MySQL InnoDB为例

商品goods表中有一个字段status，status为1代表商品未被下单，status为2代表商品已经被下单，那么我们对某个商品下单时必须确保该商品status为1。假设商品的id为1。

## InnoDB和MyISAM

MyISAM 适合于一些需要大量查询的应用，但其对于有大量写操作并不是很好。甚至你只是需要update一个字段，整个表都会被锁起来，而别的进程，就算是读进程都无法操作直到读操作完成。另外，MyISAM 对于 SELECT COUNT(\*) 这类的计算是超快无比的。

InnoDB 的趋势会是一个非常复杂的存储引擎，对于一些小的应用，它会比 MyISAM 还慢。他是它支持“行锁” ，于是在写操作比较多的时候，会更优秀。并且，他还支持更多的高级应用，比如：事务。

InnoDB和MyISAM是许多人在使用MySQL时最常用的两个表类型，这两个表类型各有优劣，视具体应用而定。基本的差别为：MyISAM类型不支持事务处理等高级处理，而InnoDB类型支持。MyISAM类型的表强调的是性能，其执行数度比InnoDB类型更快，但是不提供事务支持，而InnoDB提供事务支持以及外部键等高级数据库功能。

# 网络相关

## 三次握手

        所谓三次握手（Three-Way Handshake）即建立TCP连接，就是指建立一个TCP连接时，需要客户端和服务端总共发送3个包以确认连接的建立。在socket编程中，这一过程由客户端执行connect来触发，整个流程如下图所示：



图2 TCP三次握手

        （1）第一次握手：Client将标志位SYN置为1，随机产生一个值seq=J，并将该数据包发送给Server，Client进入SYN\_SENT状态，等待Server确认。  
        （2）第二次握手：Server收到数据包后由标志位SYN=1知道Client请求建立连接，Server将标志位SYN和ACK都置为1，ack=J+1，随机产生一个值seq=K，并将该数据包发送给Client以确认连接请求，Server进入SYN\_RCVD状态。  
        （3）第三次握手：Client收到确认后，检查ack是否为J+1，ACK是否为1，如果正确则将标志位ACK置为1，ack=K+1，并将该数据包发送给Server，Server检查ack是否为K+1，ACK是否为1，如果正确则连接建立成功，Client和Server进入ESTABLISHED状态，完成三次握手，随后Client与Server之间可以开始传输数据了。

**SYN攻击：**

在三次握手过程中，Server发送SYN-ACK之后，收到Client的ACK之前的TCP连接称为半连接（half-open connect），此时Server处于SYN\_RCVD状态，当收到ACK后，Server转入ESTABLISHED状态。SYN攻击就是Client在短时间内伪造大量不存在的IP地址，并向Server不断地发送SYN包，Server回复确认包，并等待Client的确认，由于源地址是不存在的，因此，Server需要不断重发直至超时，这些伪造的SYN包将产时间占用未连接队列，导致正常的SYN请求因为队列满而被丢弃，从而引起网络堵塞甚至系统瘫痪。SYN攻击时一种典型的DDOS攻击，检测SYN攻击的方式非常简单，即当Server上有大量半连接状态且源IP地址是随机的，则可以断定遭到SYN攻击了，

## 四次挥手

四次挥手

 三次握手耳熟能详，四次挥手估计就http://blog.chinaunix.net/kindeditor/plugins/emoticons/images/20.gif，所谓四次挥手（Four-Way Wavehand）即终止TCP连接，就是指断开一个TCP连接时，需要客户端和服务端总共发送4个包以确认连接的断开。在socket编程中，这一过程由客户端或服务端任一方执行close来触发，整个流程如下图所示：

  
图3 TCP四次挥手

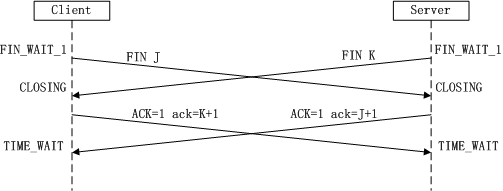
由于TCP连接时全双工的，因此，每个方向都必须要单独进行关闭，这一原则是当一方完成数据发送任务后，发送一个FIN来终止这一方向的连接，收到一个FIN只是意味着这一方向上没有数据流动了，即不会再收到数据了，但是在这个TCP连接上仍然能够发送数据，直到这一方向也发送了FIN。首先进行关闭的一方将执行主动关闭，而另一方则执行被动关闭，上图描述的即是如此。

（1）第一次挥手：Client发送一个FIN，用来关闭Client到Server的数据传送，Client进入FIN\_WAIT\_1状态。

（2）第二次挥手：Server收到FIN后，发送一个ACK给Client，确认序号为收到序号+1（与SYN相同，一个FIN占用一个序号），Server进入CLOSE\_WAIT状态。

（3）第三次挥手：Server发送一个FIN，用来关闭Server到Client的数据传送，Server进入LAST\_ACK状态。

（4）第四次挥手：Client收到FIN后，Client进入TIME\_WAIT状态，接着发送一个ACK给Server，确认序号为收到序号+1，Server进入CLOSED状态，完成四次挥手。  
        上面是一方主动关闭，另一方被动关闭的情况，实际中还会出现同时发起主动关闭的情况，具体流程如下图：

  
图4 同时挥手

        流程和状态在上图中已经很明了了，在此不再赘述，可以参考前面的四次挥手解析步骤。

四、附注  
        关于三次握手与四次挥手通常都会有典型的面试题，在此提出供有需求的XDJM们参考：  
        （1）三次握手是什么或者流程？四次握手呢？答案前面分析就是。  
        （2）为什么建立连接是三次握手，而关闭连接却是四次挥手呢？  
        这是因为服务端在LISTEN状态下，收到建立连接请求的SYN报文后，把ACK和SYN放在一个报文里发送给客户端。而关闭连接时，当收到对方的FIN报文时，仅仅表示对方不再发送数据了但是还能接收数据，己方也未必全部数据都发送给对方了，所以己方可以立即close，也可以发送一些数据给对方后，再发送FIN报文给对方来表示同意现在关闭连接，因此，己方ACK和FIN一般都会分开发送。

## ARP协议

地址解析协议(Address Resolution Protocol)，其基本功能为透过目标设备的IP地址，查询目标的MAC地址，以保证通信的顺利进行。它是IPv4网络层必不可少的协议，不过在IPv6中已不再适用，并被邻居发现协议（NDP）所替代。

**urllib和urllib2的区别**

这个面试官确实问过,当时答的urllib2可以Post而urllib不可以.

1. urllib提供urlencode方法用来GET查询字符串的产生，而urllib2没有。这是为何urllib常和urllib2一起使用的原因。
2. urllib2可以接受一个Request类的实例来设置URL请求的headers，urllib仅可以接受URL。这意味着，你不可以伪装你的User Agent字符串等。

## GET和POST与数据如何传递没有关系

     GET和POST是由[HTTP协议定义](http://www.w3.org/Protocols/rfc2616/rfc2616.html)的。在HTTP协议中，Method和Data（URL， Body， Header）是正交的两个概念，也就是说，使用哪个Method与应用层的数据如何传输是没有相互关系的。

     HTTP没有要求，如果Method是POST数据就要放在BODY中。也没有要求，如果Method是GET，数据（参数）就一定要放在URL中而不能放在BODY中。

     那么，网上流传甚广的这个说法是从何而来的呢？我[在HTML标准中，找到了相似的描述](http://www.w3.org/TR/REC-html40/interact/forms.html#h-17.13.1)。这和网上流传的说法一致。但是这只是HTML标准对HTTP协议的用法的约定。怎么能当成GET和POST的区别呢？

    而且，现代的Web Server都是支持GET中包含BODY这样的请求。虽然这种请求不可能从浏览器发出，但是现在的Web Server又不是只给浏览器用，已经完全地超出了HTML服务器的范畴了。

     知道这个有什么用？我不想解释了，有时候就得自己痛一次才记得住。

## HTTP协议对GET和POST都没有对长度的限制

     HTTP协议明确地指出了，HTTP头和Body都没有长度的要求。而对于URL长度上的限制，有两方面的原因造成：

     1. 浏览器。据说早期的浏览器会对URL长度做限制。据说IE对URL长度会限制在2048个字符内（流传很广，而且无数同事都表示认同）。但我自己试了一下，我构造了90K的URL通过IE9访问live.com，是正常的。网上的东西，哪怕是Wikipedia上的，也不能信。

     2. 服务器。URL长了，对服务器处理也是一种负担。原本一个会话就没有多少数据，现在如果有人恶意地构造几个几M大小的URL，并不停地访问你的服务器。服务器的最大并发数显然会下降。另一种攻击方式是，把告诉服务器Content-Length是一个很大的数，然后只给服务器发一点儿数据，嘿嘿，服务器你就傻等着去吧。哪怕你有超时设置，这种故意的次次访问超时也能让服务器吃不了兜着走。有鉴于此，多数服务器出于安全啦、稳定啦方面的考虑，会给URL长度加限制。但是这个限制是针对所有HTTP请求的，与GET、POST没有关系

## 安全不安全和GET、POST没有关系

     我觉得这真是中国特色。我讲个小段子，大家应该可以体会出这个说法多么的可笑。

      觉得POST数据比GET数据安全的人会说

*“防君子不防小人；中国小白多，能防小白用户就行了。”*

*“哼，”*我不以为然，*“那你怎么不说，URL参数都Encode过了，或是Base64一下，小白也看不懂啊。”*

     那人反驳道，*“Encode太简单了，聪明点儿的小白很容易就可以Decode并修改掉。”*

     我笑道，*“五十步笑百步耳，再聪明点儿的小白还会截包并重发呢，Opera就有这功能。”*

     那人阴险地祭出**神器——最终解释权**，说，***“这个不算小白。”***

     我日啊。

**最后一点儿感想**

     我之前一直做Windows桌面应用，对Web开发无甚了解，直到一年多前转做服务器端开发，才开始接触到HTTP。（注意，我说的是HTTP，不是HTML。服务器开放接口是基于REST理念设计的，使用的协议是HTTP，但是传输的内容不是HTML。这不是Web Server，而是一个Web Service）

     所以我对于GET和POST的理解，是纯粹地来源于HTTP协议。他们只有**一点**根本区别，简单点儿说，一个用于获取数据，一个用于修改数据。具体的请参考RFC文档。

     如果一个人一开始就做Web开发，很可能把HTML对HTTP协议的使用方式，当成HTTP协议的唯一的合理使用方式。从而犯了以偏概全的错误。

     可能有人会觉得我钻牛角尖。我只是不喜欢模棱两可，不喜欢边界不清、概念不明，不喜欢“拿来主义”，也不喜欢被其它喜欢钻牛角尖的人奚落得无地自容。

     “知之为知之，不知为不知，是知也。”

## 从浏览器地址栏输入url到显示页面的步骤

1.在浏览器地址栏输入URL

2.浏览器查看缓存，如果请求资源在缓存中并且新鲜，跳转到转码步骤

如果资源未缓存，发起新请求

如果已缓存，检验是否足够新鲜，足够新鲜直接提供给客户端，否则与服务器进行验证。

检验新鲜通常有两个HTTP头进行控制Expires和Cache-Control：

HTTP1.0提供Expires，值为一个绝对时间表示缓存新鲜日期

HTTP1.1增加了Cache-Control: max-age=6000,值为以秒为单位的最大新鲜时间

3.浏览器解析URL获取协议，主机，端口，path

4.浏览器组装一个HTTP（GET）请求报文

5.浏览器获取主机ip地址(DNS查询)，过程如下：

浏览器缓存

本机缓存

hosts文件

路由器缓存

ISP DNS缓存

DNS递归查询（可能存在负载均衡导致每次IP不一样）

6.打开一个socket与目标IP地址，端口建立TCP链接，三次握手如下：

客户端发送一个TCP的SYN = 1，Seq = X的包到服务器端口

服务器发回SYN = 1， ACK = X + 1， Seq = Y的响应包

客户端发送ACK = Y + 1， Seq = Z

7.TCP链接建立后发送HTTP请求

8.服务器接受请求并解析，将请求转发到服务程序，如虚拟主机使用HTTP Host头部判断请求的服务程序

9.服务器检查HTTP请求头是否包含缓存验证信息如果验证缓存新鲜，返回304等对应状态码

10.处理程序读取完整请求并准备HTTP响应，可能需要查询数据库等操作

11.服务器将响应报文通过TCP连接发送回浏览器

12.浏览器接收HTTP响应，然后根据情况选择关闭TCP连接或者保留重用，关闭TCP连接的四次挥手如下：

主动方发送FIN = 1， Ack = Z， Seq = X报文

被动方发送ACK = X + 1， Seq = Z报文

被动方发送FIN = 1， ACK = X， Seq = Y报文

主动方发送ACK = Y， Seq = X报文

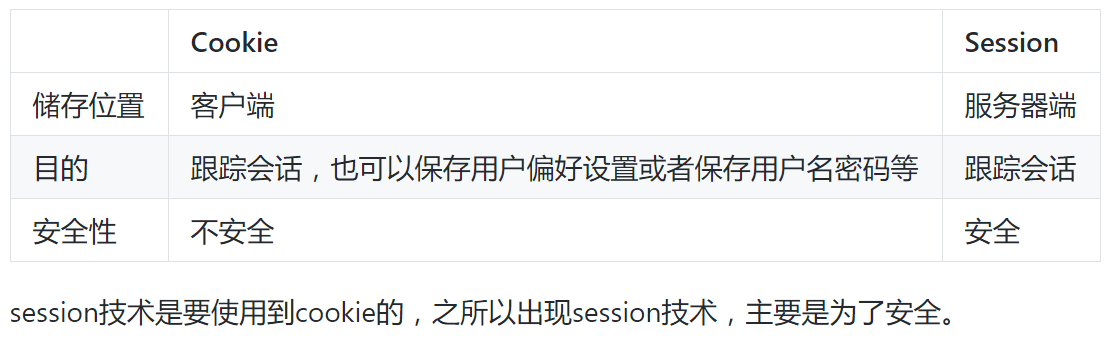
13.浏览器检查响应状态码：是否为1XX，3XX， 4XX， 5XX，这些情况处理与2XX不同

14.如果资源可缓存，进行缓存

15.对响应进行解码（例如gzip压缩）

16.根据资源类型决定如何处理（假设资源为HTML文档）

## Cookie和Session



## 网站用户密码保存

1. 明文保存
2. 明文hash后保存,如md5
3. MD5+Salt方式,这个salt可以随机
4. 知乎使用了Bcrypy(好像)加密

# HTTP和HTTPS



# 幂等 Idempotence

HTTP方法的幂等性是指一次和多次请求某一个资源应该具有同样的**副作用**。(注意是副作用)

GET http://www.bank.com/account/123456，不会改变资源的状态，不论调用一次还是N次都没有副作用。请注意，这里强调的是一次和N次具有相同的副作用，而不是每次GET的结果相同。GET http://www.news.com/latest-news这个HTTP请求可能会每次得到不同的结果，但它本身并没有产生任何副作用，因而是满足幂等性的。

DELETE方法用于删除资源，有副作用，但它应该满足幂等性。比如：DELETE http://www.forum.com/article/4231，调用一次和N次对系统产生的副作用是相同的，即删掉id为4231的帖子；因此，调用者可以多次调用或刷新页面而不必担心引起错误。

POST所对应的URI并非创建的资源本身，而是资源的接收者。比如：POST http://www.forum.com/articles的语义是在http://www.forum.com/articles下创建一篇帖子，HTTP响应中应包含帖子的创建状态以及帖子的URI。两次相同的POST请求会在服务器端创建两份资源，它们具有不同的URI；所以，POST方法不具备幂等性。

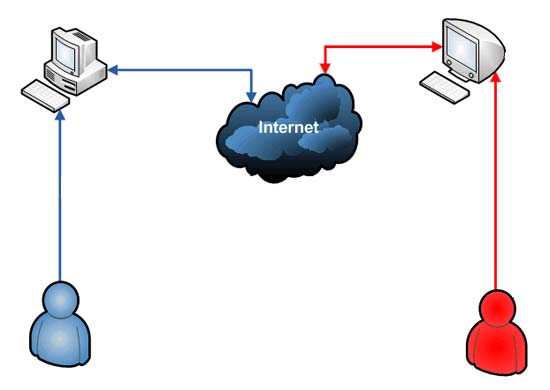
PUT所对应的URI是要创建或更新的资源本身。比如：PUT http://www.forum/articles/4231的语义是创建或更新ID为4231的帖子。对同一URI进行多次PUT的副作用和一次PUT是相同的；因此，PUT方法具有幂等性。

# RESTFul

越来越多的人开始意识到，**网站即软件**，而且是一种新型的软件。

这种"互联网软件"采用客户端/服务器模式，建立在分布式体系上，通过互联网通信，具有高延时（high latency）、高并发等特点。

网站开发，完全可以采用软件开发的模式。但是传统上，软件和网络是两个不同的领域，很少有交集；软件开发主要针对单机环境，网络则主要研究系统之间的通信。互联网的兴起，使得这两个领域开始融合，**现在我们必须考虑，如何开发在互联网环境中使用的软件。**



RESTful架构，就是目前最流行的一种互联网软件架构。它结构清晰、符合标准、易于理解、扩展方便，所以正得到越来越多网站的采用。

但是，到底什么是RESTful架构，并不是一个容易说清楚的问题。下面，我就谈谈我理解的RESTful架构。

**一、起源**

REST这个词，是[Roy Thomas Fielding](http://en.wikipedia.org/wiki/Roy_Fielding)在他2000年的[博士论文](http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm)中提出的。

**二、名称**

Fielding将他对互联网软件的架构原则，定名为REST，即Representational State Transfer的缩写。我对这个词组的翻译是"表现层状态转化"。

如果一个架构符合REST原则，就称它为RESTful架构。

**要理解RESTful架构，最好的方法就是去理解Representational State Transfer这个词组到底是什么意思，它的每一个词代表了什么涵义。**如果你把这个名称搞懂了，也就不难体会REST是一种什么样的设计。

**三、资源（Resources）**

REST的名称"表现层状态转化"中，省略了主语。"表现层"其实指的是"资源"（Resources）的"表现层"。

**所谓"资源"，就是网络上的一个实体，或者说是网络上的一个具体信息。**它可以是一段文本、一张图片、一首歌曲、一种服务，总之就是一个具体的实在。你可以用一个URI（统一资源定位符）指向它，每种资源对应一个特定的URI。要获取这个资源，访问它的URI就可以，因此URI就成了每一个资源的地址或独一无二的识别符。

所谓"上网"，就是与互联网上一系列的"资源"互动，调用它的URI。

**四、表现层（Representation）**

"资源"是一种信息实体，它可以有多种外在表现形式。**我们把"资源"具体呈现出来的形式，叫做它的"表现层"（Representation）。**

比如，文本可以用txt格式表现，也可以用HTML格式、XML格式、JSON格式表现，甚至可以采用二进制格式；图片可以用JPG格式表现，也可以用PNG格式表现。

URI只代表资源的实体，不代表它的形式。严格地说，有些网址最后的".html"后缀名是不必要的，因为这个后缀名表示格式，属于"表现层"范畴，而URI应该只代表"资源"的位置。它的具体表现形式，应该在HTTP请求的头信息中用Accept和Content-Type字段指定，这两个字段才是对"表现层"的描述。

**五、状态转化（State Transfer）**

访问一个网站，就代表了客户端和服务器的一个互动过程。在这个过程中，势必涉及到数据和状态的变化。

互联网通信协议HTTP协议，是一个无状态协议。这意味着，所有的状态都保存在服务器端。因此，**如果客户端想要操作服务器，必须通过某种手段，让服务器端发生"状态转化"（State Transfer）。而这种转化是建立在表现层之上的，所以就是"表现层状态转化"。**

客户端用到的手段，只能是HTTP协议。具体来说，就是HTTP协议里面，四个表示操作方式的动词：GET、POST、PUT、DELETE。它们分别对应四种基本操作：**GET用来获取资源，POST用来新建资源（也可以用于更新资源），PUT用来更新资源，DELETE用来删除资源。**

**六、综述**

综合上面的解释，我们总结一下什么是RESTful架构：

　　（1）每一个URI代表一种资源；

　　（2）客户端和服务器之间，传递这种资源的某种表现层；

　　（3）客户端通过四个HTTP动词，对服务器端资源进行操作，实现"表现层状态转化"。

**七、误区**

RESTful架构有一些典型的设计误区。

**最常见的一种设计错误，就是URI包含动词。**因为"资源"表示一种实体，所以应该是名词，URI不应该有动词，动词应该放在HTTP协议中。

举例来说，某个URI是/posts/show/1，其中show是动词，这个URI就设计错了，正确的写法应该是/posts/1，然后用GET方法表示show。

如果某些动作是HTTP动词表示不了的，你就应该把动作做成一种资源。比如网上汇款，从账户1向账户2汇款500元，错误的URI是：

　　POST /accounts/1/transfer/500/to/2

正确的写法是把动词transfer改成名词transaction，资源不能是动词，但是可以是一种服务：

　　POST /transaction HTTP/1.1  
　　Host: 127.0.0.1  
　　from=1&to=2&amount=500.00

**另一个设计误区，就是在URI中加入版本号**：

　　http://www.example.com/app/1.0/foo

　　http://www.example.com/app/1.1/foo

　　http://www.example.com/app/2.0/foo

因为不同的版本，可以理解成同一种资源的不同表现形式，所以应该采用同一个URI。版本号可以在HTTP请求头信息的Accept字段中进行区分（参见[Versioning REST Services](http://www.informit.com/articles/article.aspx?p=1566460)）：

　　Accept: vnd.example-com.foo+json; version=1.0

　　Accept: vnd.example-com.foo+json; version=1.1

　　Accept: vnd.example-com.foo+json; version=2.0

# RPC

RPC（Remote Procedure Call Protocol）——远程过程调用协议，它是一种通过网络从远程计算机程序上请求服务，而不需要了解底层网络技术的协议。RPC协议假定某些传输协议的存在，如TCP或UDP，为通信程序之间携带信息数据。在OSI网络通信模型中，RPC跨越了传输层和应用层。RPC使得开发包括网络分布式多程序在内的应用程序更加容易。

总结:服务提供的两大流派.传统意义以方法调用为导向通称RPC。为了企业SOA,若干厂商联合推出webservice,制定了wsdl接口定义,传输soap.当互联网时代,臃肿SOA被简化为http+xml/json.但是简化出现各种混乱。以资源为导向,任何操作无非是对资源的增删改查，于是统一的REST出现了.

进化的顺序: RPC -> SOAP -> RESTful

# CGI和WSGI

CGI是通用网关接口，是连接web服务器和应用程序的接口，用户通过CGI来获取动态数据或文件等。 CGI程序是一个独立的程序，它可以用几乎所有语言来写，包括perl，c，lua，python等等。

WSGI, Web Server Gateway Interface，是Python应用程序或框架和Web服务器之间的一种接口，WSGI的其中一个目的就是让用户可以用统一的语言(Python)编写前后端。

# 中间人攻击

在GFW里屡见不鲜的,呵呵.

中间人攻击（Man-in-the-middle attack，通常缩写为MITM）是指攻击者与通讯的两端分别创建独立的联系，并交换其所收到的数据，使通讯的两端认为他们正在通过一个私密的连接与对方直接对话，但事实上整个会话都被攻击者完全控制。