一. 简析 pselect() 与 select()的异同

1.函数原型

```
int select(int nfds, fd_set *readfds, fd_set *writefds,fd_set *exceptfds, struct timeval *timeout);

int pselect(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, const struct timespec *timeout,const sigset_t *sigmask);
```

2. 二者在参数上的几点区别：

1. pselect的超时等待时间精度更高

pselect()函数的第五个参数为const struct timespec 类型的指针，该类型定义如下：

```
struct timespec {
    long tv_sec;    //超时的秒数
    long tv_nsec;   //超时的纳秒数
}
```

而select()函数的对应参数类型为struct timeval 类型的指针，该类型定义如下：

```
struct timeval {
    long tv_sec; //超时的秒数
    long tv_usec; //超时的微秒数
}
```

可以看出，pselect的超时时间结构是一个“**纳秒**” 级别的结构，**比select的精度更高（微秒）**。不过在Linux平台下，内核调度的精度为10毫秒级，因此即使设置了更精确的时间数，使用起来也达不到设置的精度。

2. pselect传入的超时等待时间结构体不会被修改

pselect()函数传入的超时等待时间结构指针为**const**类型，因此**该参数在pselect()中是不能够被修改的**，这与select()函数不同。

在select()函数中，每次传入的timeval结构体需要初始化（重新赋值），因为该函数在执行的过程中会修改结构体里时间的值。

而pselect()函数则不需要每次初始化超时等待时间。

3. pselect()新增了第六个参数 const sigset_t *sigmask 用来屏蔽信号。

该参数指向一个信号集合，该集合中保存的信号就是pselect在执行的过程中将会屏蔽掉的信号。

当select() 正在阻塞等待的时候，它能够被信号中断返回。针对这样的情况，可以使用pselect()函数，屏蔽信号。

二. poll()函数简介

1. 函数原型

```
int poll(struct pollfd *fds, nfds_t nfds, int timeout);
```

2.参数

1. struct pollfd *fd;

   该参数指向的结构体定义如下：

   ```
   struct pollfd {
       int   fd;         /* file descriptor */
       short events;     /* requested events */
       short revents;    /* returned events */
   };
   ```

   可以看到结构体中共有三个成员：

   1. fd用来保存文件描述符，若fd为负，那么对应的events就被忽略，而revents被返回为0。这种方法可以用来忽略某个文件描述符，但是要注意不能忽略“0”。
   2. events是输入参数
   3. revents是输出参数。

   与events 和 revents相关的宏：

   POLLIN： 可读

   POLLPRI： 紧急可读

   POLLOUT： 可写

   POLLRDHUP： 对方的套接字关闭连接或者连接中途停止写入。

   POLLERR： 异常

   POLLHUP：挂起，在从套接字或管道读取时，对方关闭通道。或EOF。

常量	说明
POLLIN	普通或优先级带数据可读
POLLRDNORM	普通数据可读
POLLRDBAND	优先级带数据可读
POLLPRI	高优先级数据可读
POLLOUT	普通数据可写
POLLWRNORM	普通数据可写
POLLWRBAND	优先级带数据可写
POLLERR	发生错误
POLLHUP	发生挂起
POLLNVAL	描述字不是一个打开的文件

2. nfds_t nfds

   nfds 表示第一个参数指向的结构体数组的元素个数

3. timeout

   超时等待的 毫秒数。

3. 返回值

1. 成功，返回revents为非零值的结构体数（大于0）。
2. 超时，返回0.
3. 失败，返回-1，并设置errno。

4.注意

1. poll没有最大连接数的限制

2. “水平触发” 报告了fd后若没有被处理，那么下此次poll还会再次报告该fd。

三. 简析 poll() 与 select()的异同

同：

1. 与select()一样，poll()返回后也需要通过“轮询”pollfd的方式来取得“就绪”的文件描述符

异：

1. poll有“水平触发”的特点，报告了fd之后若没被处理，下次还会再次报告该fd。
2. poll不需要开发者计算最大文件描述符+1的大小
3. 在文件描述符数量很大的情况下，poll的速度更快。
4. poll没有最大连接数量的限制，因为它是用链表来储存的。

四. epoll简介（epoll_create() & epoll_ctl() & epoll_wait()）

1. 函数原型

```
#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fd（使用完毕需要close）

参数size告诉内核这个监听的数目一共有多大。

2. **`int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)`**

   epoll的事件注册函数，用来注册要监听的事件类型。

   参数：

   1. int epfd : epoll_create 返回的文件描述符
   2. int op : 传入要进行的动作，用宏定义表示：

      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；
   3. int fd : 需要监听的fd
   4. struct epoll_event *event : 告诉内核要监听什么事。

      struct epoll_event 结构体定义如下：

      ```
      struct epoll_event {
          __uint32_t events;  /* Epoll events */
          epoll_data_t data;  /* User data variable */
      };
      typedef union epoll_data {
          void        *ptr;
      ```

```
    int          fd;
    uint32_t     u32;
    uint64_t     u64;
} epoll_data_t;
```

使用方法（封装函数）：

```
void add_event(int epollfd, int fd, int state) {
    struct epoll_event ev;
    ev.events = state; // 传入宏定义
    ev.data.fd = fd;
    epoll_ctl(epollfd, EPOLL_CTL_ADD, &ev);
}
```

删改对应函数自行类推。

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);`**

   等待事件的发生，返回需要处理的事件数目，返回0表示超时。

   参数:

   1. int epfd
   2. struct epoll_event *events： 用来从内核得到事件的集合
   3. int maxevents : 告诉内核这个events有多大
   4. int timeout： 超时时间（毫秒）

五. 简析 epoll 与 select()的异同

1. epoll不需要通过轮询的方式来找到“就绪”的文件描述符。只要到制定的结构体数组中去取即可，epoll_wait的返回值代表“就绪”的fd数量。
2. epoll没有对fd数量的限制，上限是最大可以打开文件的数目（1g内存 ≈ 几打开十万个文件）。
3. epoll效率不会随fd数目增长而线性下降，epoll只对活跃的socket进行操作，因为只有活跃的socket才会主动地调用callback函数。（因此，若活跃的fd占所有的fd比重非常大时，epoll的效率可能并不高。但在正常网络环境其效率则远胜于select 和 poll）
4. epoll避免了“不必要的内存拷贝”，epoll内部是通过“内核与用户空间mmap同一块内存”实现的。

六. 摘抄：

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。

	Select	Poll	Epoll
支持最大连接数	1024（x86） or 2048（x64）	无上限	无上限
IO效率	每次调用进行线性遍历，时间复杂度为O（N）	每次调用进行线性遍历，时间复杂度为O（N）	使用“事件”通知方式，每当fd就绪，系统注册的回调函数就会被调用，将就绪fd放到rdllist里面，这样epoll_wait返回的时候我们就拿到了就绪的fd。时间发复杂度O（1）
fd拷贝	每次select都拷贝	每次poll都拷贝	调用epoll_ctl时拷贝进内核并由内核保存，之后每次epoll_wait不拷贝

总结：

综上，在选择select，poll，epoll时要根据具体的使用场合以及这三种方式的自身特点。**

1、表面上看epoll的性能最好，但是在连接数少并且连接都十分活跃的情况下，select和poll的性能可能比epoll好，毕竟epoll的通知机制需要很多函数回调。

2、select低效是因为每次它都需要轮询。但低效也是相对的，视情况而定，也可通过良好的设计改善