引言

为什么要学I/O多路复用呢?

之前我们学习多进程、多线程就是为了解决高并发问题。

但是，多进程和多线程都有其缺点！

多进程

1. 进程数量的限制
2. 进程的创建、销毁及切换代价较高
3. 受限于CPU核心数
   - 如果是单核，同一时刻只可能由一个进程在跑
4. 进程间内存隔离
5. 进程间通信代价较高

多线程

1. 受限于CPU核心数！

   这样就会影响响应能力！

之前我们写的项目的致命缺陷

1. 阻塞！--导致响应能力受限(*例如我们写的chatroom的accept和recv*)
   - 但是如果改成非阻塞，那么就必须得循环地去检测是否有数据来！就得花更多的时间、更多的cpu次数！
2. 所以就没有办法感知I/O！

所以，我们引入了I/O多路复用(多路转接)， 它的特征

1. 可以感知I/O(是否可读、可写、异常)，即I/O是否ready

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，从而大大减小了系统的开销。

关于I/O多路复用的三个函数

三个函数之间的总结对比

即三个函数的简单对比及各优缺点

函数名	函数原型	功能描述	RETRUN VALUE
select	int select(int nfds, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);	monitor multiple fds, waiting until one of more fds become "ready" for I/O operation	-1 error; 0 timeout ; number of ready success;
poll	int poll(struct pollfd *fds, nfds_t nfds, int timeout);	performs a similar task to select	基本同上
epoll(API)		performs a similar task to poll	
	int epoll_create(int size);		-1 error; fd success(返回一个引用新epoll实例的fd)
	int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)	performs control operations on the epoll instance	-1 error; 0 success
	int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout)	waits for events on the epoll instance	基本同poll

相同点

> 1. 都是用来感知I/O操作的，功能基本相同

不同点

> *这三个函数是不断优化的*！
>
> 1. *从传参角度*

- select要传入三个`fd_set`的指针，这样即使你成功返回，你还得循环三个数组去找哪些是ready的
- poll只需要传入一个结构体指针，这样如果成功返回，你就只需要循环那一个数组就行
- epoll_create只传入一个size(实际上被忽略了)， 就可返回一个引用新epoll实例的fd，之后想要关注某个fd只需调用epoll_ctl就可

2. ***select的几大缺点(poll也基本差不多)***

- 每次调用，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
- 同时每次调用时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
- select支持的fd数量太小了，默认是1024

3. ***epoll对select和poll的改进***

- 首先select和poll都只提供了一个函数，而epoll则是提供了3个函数
  - `epoll_create`是创建一个引用新epoll实例的fd
  - `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`的解决方案是在`epoll_ctl`函数中。每次注册新的事件到epfd中时，会把所有的fd拷贝进内核，而不是在`epoll_wait`的时候重复拷贝
- 对于第二个缺点，epoll的解决方案不想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
- 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的文件的数目，这个数字一般远大于2048，比如，在1GB内存的机器上大约是10W左右，具体数目可以`cat /proc/sys/fs/file-max`查看，一般来说这个数目和系统内存关系很大。

参考资料：www.cnblogs.com/Anker/p/3265058.html