**一、 什么是IO？**

我们都知道unix世界里，一切皆文件，而文件是什么呢？文件就是一串二进制流，不管socket,还是FIFO、管道、终端，对我们来说，一切都是文件，一切都是流。在信息 交换的过程中，**我们都是对这些流进行数据的收发操作，简称为I/O操作(input and output)**，往**流中读出数据，**[**系统**](https://www.2cto.com/os/)**调用read，写入数据，系统调用write。**

**二、 文件描述符**

那么计算机中这么多流，我们是如何知道要操作哪个流呢？

这时候我们需要有一个能够对文件进行定位的标识符，那么**文件描述符**就应运而生。文件描述符是内核为了高效管理已经被打开的文件所创建的索引，他是一个从0开始的整数，程序所有执行的I/O操作都是通过文件描述符进行的。其中，**在程序刚刚启动时，0,1,2三个文件描述符已经被占用了**，0代表标准输入设备stdin（比如键盘），1代表标准输出设备stdout（显示器），2代表标准错误stderr.POSIX标准要求每次打开文件时（含socket）必须使用当前进程中最小可用的文件描述符号码,因此再打开一个文件，它的文件描述符会是3。  
既然文件描述符是文件的索引，那么它有没有**最大限制**呢？对的，文件描述符是系统的一个重要的资源，实际中最大打开的文件数是系统内存的10%，这个是**系统级限制**，有系统就会有用户，**用户级限制是单个进程最大打开的文件数，一般是1024**，可以使用ulimit -n命令查看。

**系统是如何通过文件描述符定位到文件的呢？**  
系统为每一个进程维护了一个**文件描述符表**，这是一个进程级的文件描述符表，它通过文件描述符所对应的文件指针指向系统级的**打开文件描述符表**中的一个打开文件句柄，句柄中存储了打开文件相应的全部信息，包括文件偏移量、状态标示、访问模式文件类型以及文件属性等等，其中有一个inode指针，它指向了**i-node表** 中该文件的表项。

**三、 阻塞与非阻塞**

为了让大家能顺利掌握IO复用，请允许我再唠叨一下阻塞与非阻塞。  
什么是程序的阻塞呢？想象这种情形，比如你等快递，但快递一直没来，你会怎么做？有两种方式：  
**快递没来，我可以先去睡觉，然后快递来了给我打电话叫我去取就行了。(**非阻塞忙轮询**)  
快递没来，我就不停的给快递打电话说：擦，怎么还没来，给老子快点，直到快递来。(**阻塞**)**  
很显然，你无法忍受第二种方式，不仅耽搁自己的时间，也会让快递很想打你。  
而在计算机世界，这两种情形就对应阻塞和非阻塞忙轮询。  
非阻塞忙轮询：数据没来，进程就不停的去检测数据，直到数据来。  
阻塞：数据没来，啥都不做，直到数据来了，才进行下一步的处理。  
 先说说阻塞，为了了解阻塞是如何进行的，我们来讨论缓冲区，以及内核缓冲区，最终把I/O事件解释清楚。

缓冲区的引入是为了减少频繁I/O操作而引起频繁的系统调用（你知道它很慢的），当你操作一个流时，更多的是以缓冲区为单位进行操作，这是相对于用户空间而言。对于内核来说，也需要缓冲区。

因为一个线程只能处理一个套接字的I/O事件，如果想同时处理多个，**可以利用非阻塞忙轮询的方式,**伪代码如下：

[?](https://www.2cto.com/kf/201609/545578.html)

|  |  |
| --- | --- |
| 1  2 | while true { for i in stream[]  { if i has data read until unavailable } } |

我们只要把**所有流从头到尾查询一遍**，就可以处理多个流了，但这样做很不好，因为如果所有的流都没有I/O事件，白白浪费CPU时间片。正如有一位科学家所说，**计算机所有的问题都可以增加一个中间层来解决**，同样，为了避免这里cpu的空转，我们**不让这个线程亲自去检查流中是否有事件**，而是**引进了一个代理(一开始是select,后来是poll)，**这个代理很牛，它**可以同时观察许多流的I/O事件，如果没有事件，代理就阻塞，线程就不会挨个挨个去轮询了**，伪代码如下：

[?](https://www.2cto.com/kf/201609/545578.html)

|  |  |
| --- | --- |
| 1  2 | while true { select(streams[]) //这一步死在这里，知道有一个流有I/O事件时，才往下执行   for i in streams[] { if i has data read until unavailable } } |

**四、 IO多路复用**

**I/O多路复用 (单个线程，通过记录跟踪每个I/O流(sock)的状态，来同时管理多个I/O流 。)**

**重要的事情再说一遍： I/O multiplexing 这里面的 multiplexing 指的其实是在单个线程通过记录跟踪每一个Sock(I/O流)的状态(对应空管塔里面的Fight progress strip槽)来同时管理多个I/O流**.

发明它的原因，是尽量多的提高服务器的吞吐能力。

是不是听起来好拗口，看个图就懂了



在同一个线程里面， **通过拨开关的方式，来同时传输多个I/O流**， (学过EE的人现在可以站出来义正严辞说这个叫“时分复用”了）。

什么，你还没有搞懂“一个请求到来了，**nginx使用epoll接收请求的过程是怎样的**”，

多看看这个图就了解了。提醒下，**ngnix会有很多链接进来， epoll会把他们都监视起来，然后像拨开关一样，谁有数据就拨向谁，然后调用相应的代码处理。**

**select, poll, epoll 都是I/O多路复用的具体的实现，之所以有这三个鬼存在，其实是他们出现是有先后顺序的。**

I/O多路复用这个概念被提出来以后， select是第一个实现 (1983 左右在BSD里面实现的)。

select 被实现以后，很快就暴露出了很多问题。

* **select 会修改传入的参数数组**，这个对于一个需要调用很多次的函数，是非常不友好的。
* **select 如果任何一个sock(I/O stream)出现了数据，select 会返回所有的sock，但是并不会告诉你是那个sock上有数据，于是你只能自己一个一个的找，**10几个sock可能还好，要是几万的sock每次都找一遍，这个无谓的开销就颇有海天盛筵的豪气了。
* **select 只能监视1024个链接，** 这个跟草榴没啥关系哦，linux 定义在头文件中的，参见*FD\_SETSIZE。*
* select 不是线程安全的，如果你把一个sock加入到select, 然后突然另外一个线程发现，尼玛，这个sock不用，要收回。对不起，这个select 不支持的，如果你丧心病狂的竟然关掉这个sock, select的标准行为是。。呃。。不可预测的， 这个可是写在文档中的哦.
* 于是14年以后(1997年）一帮人又实现了poll, poll 修复了select的很多问题，比如
* **poll 去掉了1024个链接的限制，**于是要多少链接呢， 主人你开心就好。
* poll 从设计上来说，**不再修改传入数组，**不过这个要看你的平台了，所以行走江湖，还是小心为妙。

**其实拖14年那么久也不是效率问题， 而是那个时代的硬件实在太弱，一台服务器处理1千多个链接简直就是神一样的存在了，select很长段时间已经满足需求。**

但是poll仍然不是线程安全的， 这就意味着，不管服务器有多强悍，你也只能在一个线程里面处理一组I/O流。你当然可以那多进程来配合了，不过然后你就有了多进程的各种问题。

于是5年以后, 在2002, 大神 Davide Libenzi 实现了epoll.

epoll 可以说是I/O 多路复用最新的一个实现，epoll 修复了poll 和select绝大部分问题, 比如：

* epoll 现在是线程安全的。
* epoll 现在**不仅告诉你sock组里面数据，**还会告诉你**具体哪个sock有数据，**你不用自己去找了。

**select**

函数介绍：  
　　该函数准许**应用程序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：

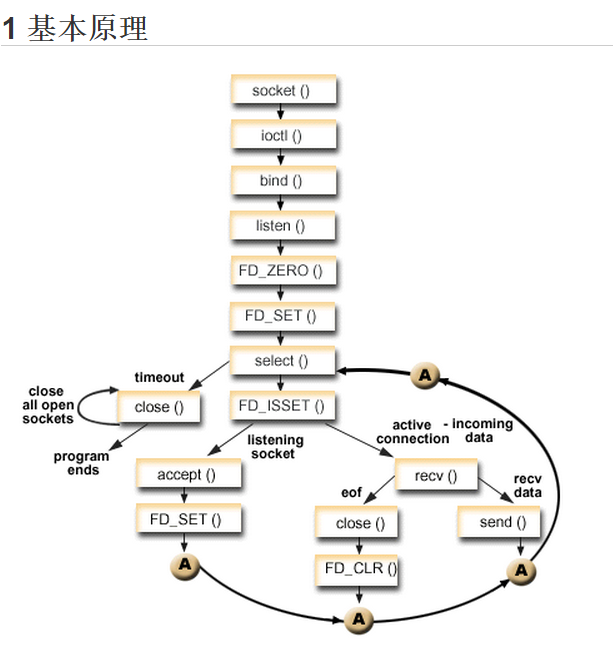
|  |  |
| --- | --- |
| 1  2  3  4 | include <sys select.h="">  include <sys time.h="">    int select(int maxfdp1,fd\_set \***readset**,fd\_set \***writeset**,fd\_set \***exceptset**,const struct timeval \*timeout)</sys></sys> |

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（可以看为是socket的事件）**，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。

|  |  |
| --- | --- |
| 1  2  3  4 | 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：



函数说明：  
I/O多路复用这个概念被提出来以后， select是第一个实现 (1983 左右在BSD里面实现的)。

**select 被实现以后，很快就暴露出了很多问题。**  
1) 对socket进行扫描时是线性扫描，即采用轮询的方法，效率较低。  
当套接字比较多的时候，每次select()都要通过遍历**FD\_SETSIZE个Socket来完成调度**，**不管哪个Socket是活跃的，都遍历一遍**。这会浪费很多CPU时间。  
2) select **如果任何一个sock(I/O stream)出现了数据，select 仅仅会返回，但是并不会告诉你是那个sock上有数据，于是你只能遍历所有的socket自己一个一个的找**  
3) select **只能监视1024个链接**， linux 定义在头文件中的，**参见FD\_SETSIZE。**  
4) **需要维护一个用来存放大量fd的数据结构，这样会使得用户空间和内核空间在传递该结构时复制开销大。**  
5) select 不是线程安全的，如果你把一个sock加入到select, 然后突然另外一个线程发现，这个sock不用，要收回。对不起，这个select 不支持的，如果你丧心病狂的竟然关掉这个sock, select的标准行为是。。呃。。不可预测的。

1）**这3个bitmap有大小限制（FD\_SETSIZE，常为1024）；**

**2）由于这3个集合在返回时会被内核修改，因此我们每次调用时都需要重新设置；**

**3）我们在调用完成后需要扫描这3个集合（即扫描所有的socket）才能知道哪些fd的读/写事件发生了，一般情况下全量集合比较大而实际发生读/写事件的fd比较少，效率比较低下；**

**4）（将用户传入的数组拷贝到内核空间）内核在每次调用时都需要扫描这3个fd集合(套接字集合)，然后查看哪些fd的事件实际发生，并将它传到设备等待队列。在读/写比较稀疏的情况下同样存在效率问题**

**poll**

于是14年以后(1997年）一帮人又实现了poll, poll 修复了select的很多问题，比如  
**1) poll 去掉了1024个链接的限制，于是要多少链接呢， 主人你开心就好。  
2) poll 从设计上来说，不再修改传入所有的数组（所有的socket），需要传递的是一个pollfd结构的数组，不过这个要看你的平台了。**

但是poll仍然不是线程安全的， 这就意味着，不管服务器有多强悍，你也只能在一个线程里面处理一组I/O流。

你当然可以那多进程来配合了，不过然后你就有了多进程的各种问题。  
poll本质上和select没有区别，**它将用户传入的数组拷贝到内核空间，然后查询每个fd对应的设备状态，如果设备就绪则在设备等待队列中加入一项并继续遍历，如果遍历完所有fd后没有发现就绪设备，则挂起当前进程，直到设备就绪或者主动超时，被唤醒后它又要再次遍历fd。**这个过程经历了多次无谓的遍历。  
 它没有最大连接数的限制，原因是它是**基于链表来存储的**，但是同样有一个缺点：  
**1) 大量的fd的数组被整体复制于用户态和内核地址空间之间，而不管这样的复制是不是有意义。  
2) poll还有一个特点是“水平触发”，如果报告了fd后，没有被处理，那么下次poll时会再次报告该fd。**

select() 和 poll() 系统调用的本质一样，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（但是数量过大后性能也是会下降）。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。

**epoll**

于是5年以后, 在2002, 大神 Davide Libenzi 实现了epoll.  
epoll 可以说是I/O 多路复用最新的一个实现，epoll 修复了poll 和select绝大部分问题, 比如： **1. 没有最大并发连接的限制，能打开的FD的上限远大于1024（1G的内存上能监听约10万个端口）。**2. 效率提升，不是轮询的方式，不会随着FD数目的增加效率下降。

**只有活跃可用的FD**才会调用callback函数；即Epoll最大的优点就在于它**只管你“活跃”的连接**，而跟连接总数无关，因此在实际的网络环境中，Epoll的效率就会远远高于select和poll。  
3. 内存拷贝，利用mmap()文件映射内存加速与内核空间的消息传递；**即epoll使用mmap减少复制开销**。（共享内存的方式实现）

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。  
原理：epoll支持水平触发和边缘触发，最大的特点在于边缘触发，它只告诉进程哪些fd刚刚变为就绪态，并且只会通知一次。还有一个特点是，**epoll使用“事件”的就绪通知方式，通过epoll\_ctl注册fd，一旦某个文件描述符就绪，内核就会采用类似callback的回调机制来激活该fd，epoll\_wait便可以收到通知。（此处去掉了遍历文件描述符，而是通过监听回调的的机制。这正是epoll的魅力所在。）**

**三种方式总结：**

从select那里仅仅知道了，**有I/O事件发生了，却并不知道是哪那几个流（可能有一个，多个，甚至全部），我们只能无差 别轮询所有流，找出能读出数据，或者写入数据的流，对他们进行操作。所以select具有O(n)的无差别轮询复杂度，同时处理的流越多，无差别轮询时间就越长。**  
epoll可以理解为event poll，不同于忙轮询和无差别轮询，**epoll会把哪个流发生了怎样的I/O事件通知我们。**所以我们说epoll实际上**是事件驱动（每个事件关联上fd）的**，此时我们对这些流的操作都是有意义的。（复杂度降低到了O(1)）伪代码如下：

|  |  |
| --- | --- |
| 1 | while true { active\_stream[] = epoll\_wait(epollfd) for i in active\_stream[] { read or write till } } |

可以看到，select和epoll最大的区别就是：select只是告诉你一定数目的流有事件了，至于哪个流有事件，还得你一个一个地去轮询，而 epoll会把发生的事件告诉你，通过发生的事件，就自然而然定位到哪个流了。不能不说epoll跟select相比，是质的飞跃，这也是一种牺牲空间，换取时间的思想，毕竟现在硬件越来越便宜了。  
 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。