# select/poll/epoll对比分析

Dec 6, 2015



select/poll/epoll都是IO多路复用机制，可以同时监控多个描述符，当某个描述符就绪(读或写就绪)，则立刻通知相应程序进行读或写操作。本质上select/poll/epoll都是同步I/O，即读写是阻塞的。

# 一、select

原型：

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

从select函数监控3类文件描述符：writefds、readfds、exceptfds。调用select函数后会阻塞，直到描述符准备就绪（有数据可读、可写、或者出现异常）或者超时，函数便返回。当select函数返回后，可以通过遍历描述符集合，找到就绪的描述符。

**select缺点**

- 单进程能够监控的文件描述符的数量存在最大限制，在Linux上一般为1024，可以通过修改宏定义增大上限，但同样存在效率低的弱势;
- IO效随着监视的描述符数量的增长，其效率也会线性下降;

# 二、poll

原型：

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

其中pollfd表示监视的描述符集合，如下

```
struct pollfd {
    int fd; //文件描述符
    short events; //监视的请求事件
    short revents; //已发生的事件
};
```

pollfd结构包含了要监视的event和发生的event，并且pollfd并没有最大数量限制（但数量过大同样会导致性下降）。 和select函数一样，当poll函数返回后，可以通过遍历描述符集合，找到就绪的描述符。

从上面看，select和poll都需要在返回后，通过遍历文件描述符来获取已经就绪的socket。事实上，同时连接的大量客户端在一时刻可能只有很少的处于就绪状态，因此随着监视的描述符数量的增长，其效率也会线性下降。

# 三、epoll

epoll是在2.6内核中提出的，是select和poll的增强版。相对于select和poll来说，epoll更加灵活，没有描述符数量限制。epoll使用一个文件描述符管理多个描述符，将用户空间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。epoll机制是Linux最高效的I/O复用机制，在一处等待多个文件句柄的I/O事件。

select/poll都只有一个方法，而epoll的操作过程有3个方法，分别是`epoll_create()`，`epoll_ctl()`，`epoll_wait()`。

## 3.1 epoll_create()

```
int epoll_create(int size);
```

用于创建一个epoll的句柄，size是指监听的描述符个数， 现在内核支持动态扩展，该值的意义仅仅是初次分配的fd个数，后面空间不够时会动态扩容。 当创建完epoll句柄后，占用一个fd值.

```
ls /proc/<pid>/fd/  //可通过终端执行，看到该fd
```

使用完epoll后，必须调用close()关闭，否则可能导致fd被耗尽。

## 3.2 epoll_ctl()

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

用于对需要监听的文件描述符(fd)执行op操作，比如将fd加入到epoll句柄。

- epfd：是epoll_create()的返回值；
- op：表示op操作，用三个宏来表示，分别代表添加、删除和修改对fd的监听

事件；
    - EPOLL_CTL_ADD(添加)
    - EPOLL_CTL_DEL(删除)
    - EPOLL_CTL_MOD（修改）
- fd：需要监听的文件描述符；
- epoll_event：需要监听的事件，struct epoll_event结构如下：

```
struct epoll_event {
  __uint32_t events;  /* Epoll事件 */
  epoll_data_t data;  /*用户可用数据*/
};
```

events可取值：(表示对应的文件描述符的操作)

- EPOLLIN ：可读（包括对端SOCKET正常关闭）；
- EPOLLOUT：可写；
- EPOLLERR：错误；
- EPOLLHUP：中断；
- EPOLLPRI：高优先级的可读（这里应该表示有带外数据到来）；
- EPOLLET： 将EPOLL设为边缘触发模式，这是相对于水平触发来说的。
- EPOLLONESHOT：只监听一次事件，当监听完这次事件之后就不再监听该事件

## 3.3 epoll_wait()

```
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

- epfd：等待epfd上的io事件，最多返回maxevents个事件；
- events：用来从内核得到事件的集合；
- maxevents：events数量，该maxevents值不能大于创建epoll_create()时的size；
- timeout：超时时间（毫秒，0会立即返回）。

该函数返回需要处理的事件数目，如返回0表示已超时。

# 四、对比

在 select/poll中，进程只有在调用一定的方法后，内核才对所有监视的文件描述符进行扫描，而epoll事先通过epoll_ctl()来注册一个文件描述符，一旦基于某个文件描述符就绪时，内核会采用类似callback的回调机制，迅速激活这个文件描述符，当进程调用epoll_wait() 时便得到通知。(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。)

**epoll优势**

1. 监视的描述符数量不受限制，所支持的FD上限是最大可以打开文件的数目，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，以3G的手机来说这个值为20-30万。
2. IO效率不会随着监视fd的数量增长而下降。epoll不同于select和poll轮询的方式，而是通过每个fd定义的回调函数来实现的，只有就绪的fd才会执行回调函数。

如果没有大量的idle-connection或者dead-connection，epoll的效率并不会比select/poll高很多，但是当遇到大量的idle-connection，就会发现epoll的效率大大高于select/poll。

---

另外，想更详细地了解select/poll/epoll机制，可查看http://www.cnblogs.com/Anker/p/3265058.html (http://www.cnblogs.com/Anker/p/3265058.html)

---




gityuan@gmail.com (mailto:gityuan@gmail.com) · Github

(https://github.com/yuanhuihui) · 天道酬勤 · © 2015 Yuanhh · Jekyll

(https://github.com/jekyll/jekyll) theme by HyG (https://github.com/Gaohaoyang)