

# 1. I/O多路复用（I/O多路转接）

I/O 多路复用使得程序能同时监听多个文件描述符，能够提高程序的性能，Linux 下实现 I/O 多路复用的系统调用主要有 select、poll 和 epoll。

## 2. select

主旨思想：

1. 首先要构造一个关于文件描述符的列表，将要监听的文件描述符添加到该列表中。
2. 调用一个系统函数，监听该列表中的文件描述符，直到这些描述符中的一个或者多个进行I/O操作时，该函数才返回。
  - a.这个函数是阻塞
  - b.函数对文件描述符的检测的操作是由内核完成的
3. 在返回时，它会告诉进程有多少（哪些）描述符要进行I/O操作。

```
// sizeof(fd_set) = 128      1024
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/select.h>
int select(int nfd, fd_set *readfds, fd_set *writefds,
           fd_set *exceptfds, struct timeval *timeout);
```

- 参数：

- nfd : 委托内核检测的最大文件描述符的值 + 1
- readfds : 要检测的文件描述符的读的集合，委托内核检测哪些文件描述符的读的属性
  - 一般检测读操作
  - 对应的是对方发送过来的数据，因为读是被动的接收数据，检测的就是读缓冲区
- writefds : 要检测的文件描述符的写的集合，委托内核检测哪些文件描述符的写的属性
  - 委托内核检测写缓冲区是不是还可以写数据（不满的就可以写）
- exceptfds : 检测发生异常的文件描述符的集合
- timeout : 设置的超时时间

```
struct timeval {
    long    tv_sec;           /* seconds */
    long    tv_usec;         /* microseconds */
};
```

  - NULL : 永久阻塞，直到检测到了文件描述符有变化
  - tv\_sec = 0 tv\_usec = 0, 不阻塞
  - tv\_sec > 0 tv\_usec > 0, 阻塞对应的时间

- 返回值 :

- -1 : 失败
- >0(n) : 检测的集合中有n个文件描述符发生了变化

```
// 将参数文件描述符fd对应的标志位设置为0
void FD_CLR(int fd, fd_set *set);
// 判断fd对应的标志位是0还是1, 返回值 : fd对应的标志位的值, 0, 返回0, 1, 返回1
int FD_ISSET(int fd, fd_set *set);
// 将参数文件描述符fd 对应的标志位, 设置为1
void FD_SET(int fd, fd_set *set);
```

```
// fd_set一共有1024 bit，全部初始化为0
void FD_ZERO(fd_set *set);
```

## 3. poll

```
#include <poll.h>
struct pollfd {
    int    fd;           /* 委托内核检测的文件描述符 */
    short  events;        /* 委托内核检测文件描述符的什么事件 */
    short  revents;       /* 文件描述符实际发生的事件 */
};

struct pollfd myfd;
myfd.fd = 5;
myfd.events = POLLIN | POLLOUT;

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

- 参数:
  - **fds** : 是一个struct pollfd 结构体数组，这是一个需要检测的文件描述符的集合
  - **nfds** : 这个是第一个参数数组中最后一个有效元素的下标 + 1
  - **timeout** : 阻塞时长
    - 0 : 不阻塞
    - 1 : 阻塞，当检测到需要检测的文件描述符有变化，解除阻塞
    - >0 : 阻塞的时长
- 返回值:
  - 1 : 失败
  - >0 (n) : 成功,n表示检测到集合中有n个文件描述符发生变化

事件	常值	作为events的值	作为revents的值	说明
读事件	POLLIN	✓	✓	普通或优先带数据可读
	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
写事件	POLLOUT	✓	✓	普通或优先带数据可写
	POLLWRNORM	✓	✓	普通数据可写
	POLLWRBAND	✓	✓	优先级带数据可写
错误事件	POLLERR		✓	发生错误
	POLLHUP		✓	发生挂起
	POLLNVAL		✓	描述不是打开的文件

## 4. epoll

```
#include <sys/epoll.h>
```

// 创建一个新的epoll实例。在内核中创建了一个数据，这个数据中有两个比较重要的数据，一个是需要检测的文件描述符的信息（红黑树），还有一个是就绪列表，存放检测到数据发送改变的文件描述符信息（双向链表）。

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

- 参数:
  - size** : 目前没有意义了。随便写一个数，必须大于0
- 返回值:
  - 1 : 失败
  - > 0 : 文件描述符，操作epoll实例的

```

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 */
};

```

常见的Epoll检测事件：

- EPOLLIN
- EPOLLOUT
- EPOLLERR

// 对epoll实例进行管理：添加文件描述符信息，删除信息，修改信息

```

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

```

- 参数：
  - epfd : epoll实例对应的文件描述符
  - op : 要进行什么操作
    - EPOLL\_CTL\_ADD: 添加
    - EPOLL\_CTL\_MOD: 修改
    - EPOLL\_CTL\_DEL: 删除
  - fd : 要检测的文件描述符
  - event : 检测文件描述符什么事情

// 检测函数

```

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

```

- 参数：
  - epfd : epoll实例对应的文件描述符
  - events : 传出参数，保存了发送了变化的文件描述符的信息
  - maxevents : 第二个参数结构体数组的大小
  - timeout : 阻塞时间
    - 0 : 不阻塞
    - -1 : 阻塞，直到检测到fd数据发生变化，解除阻塞
    - > 0 : 阻塞的时长（毫秒）
- 返回值：
  - 成功，返回发送变化的文件描述符的个数 > 0
  - 失败 -1

Epoll 的工作模式：

- LT 模式（水平触发）

假设委托内核检测读事件 -> 检测fd的读缓冲区

读缓冲区有数据 -> epoll检测到了会给用户通知

- 用户不读数据，数据一直在缓冲区，epoll 会一直通知
- 用户只读了一部分数据，epoll会通知
- 缓冲区的数据读完了，不通知

LT (level - triggered) 是缺省的工作方式，并且同时支持 block 和 no-block socket。在这种做法中，内核告诉你一个文件描述符是否就绪了，然后你可以对这个就绪的 fd 进行 IO 操作。如果你不作任何操作，内核还是会继续通知你的。

- ET 模式 (边沿触发)

假设委托内核检测读事件 -> 检测fd的读缓冲区

读缓冲区有数据 -> epoll检测到了会给用户通知

- a.用户不读数据，数据一致在缓冲区中，epoll下次检测的时候就不通知了
- b.用户只读了一部分数据，epoll不通知
- c.缓冲区的数据读完了，不通知

ET (edge - triggered) 是高速工作方式，只支持 no-block socket。在这种模式下，当描述符从未就绪变为就绪时，内核通过epoll告诉你。然后它会假设你知道文件描述符已经就绪，并且不会再为那个文件描述符发送更多的就绪通知，直到你做了某些操作导致那个文件描述符不再为就绪状态了。但是请注意，如果一直不对这个 fd 作 IO 操作（从而导致它再次变成未就绪），内核不会发送更多的通知（only once）。

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。

```
struct epoll_event {
    uint32_t     events;      /* Epoll events */
    epoll_data_t data;       /* User data variable */
};
```

常见的Epoll检测事件：

- EPOLLIN
- EPOLLOUT
- EPOLLERR
- EPOLLET