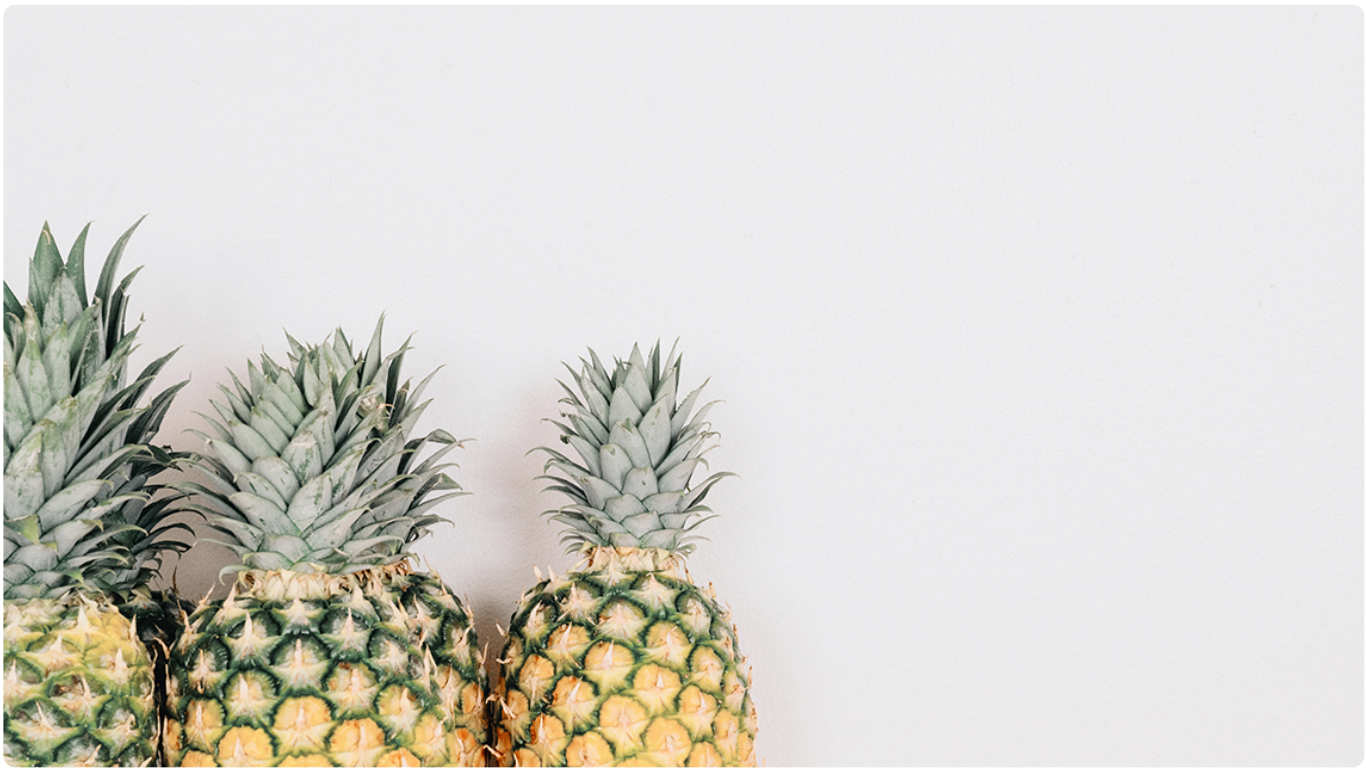
23 | Linux利器：epoll的前世今生

2019-09-30 盛延敏

网络编程实战 进入课程 >



讲述：冯永吉
时长 11:52 大小 10.88M

你好，我是盛延敏，这里是网络编程实战第 23 讲，欢迎回来。

性能篇的前三讲，非阻塞 I/O 加上 I/O 多路复用，已经渐渐帮助我们在高性能网络编程这个领域搭建了初步的基石。但是，离最终的目标还差那么一点，如果说 I/O 多路复用帮我们打开了高性能网络编程的窗口，那么今天的主题——epoll，将为我们增添足够的动力。

我在文稿中放置了一张图，这张图来自 The Linux Programming Interface(No Starch Press)。这张图直观地为我们展示了 select、poll、epoll 几种不同的 I/O 复用技术在面对不同文件描述符大小时的表现差异。

Number of File Descriptors	poll() CPU time	select() CPU time	epoll() CPU time
10	0.61	0.73	0.41
100	2.9	3	0.42
1000	35	35	0.53
10000	990	930	0.66

从图中可以明显地看到，epoll 的性能是最好的，即使在多达 10000 个文件描述的情况下，其性能的下降和有 10 个文件描述符的情况相比，差别也不是很大。而随着文件描述符的增大，常规的 select 和 poll 方法性能逐渐变得很差。

那么，epoll 究竟使用了什么样的“魔法”，取得了如此令人惊讶的效果呢？接下来，我们就来一起分析一下。

epoll 的用法

在分析对比 epoll、poll 和 select 几种技术之前，我们先看一下怎么使用 epoll 来完成一个服务器程序，具体的原理我将在 29 讲中进行讲解。

epoll 可以说是和 poll 非常相似的一种 I/O 多路复用技术，有些朋友将 epoll 归为异步 I/O，我觉得这是不正确的。本质上 epoll 还是一种 I/O 多路复用技术， epoll 通过监控注册的多个描述字，来进行 I/O 事件的分发处理。不同于 poll 的是，epoll 不仅提供了默认的 level-triggered（条件触发）机制，还提供了性能更为强劲的 edge-triggered（边缘触发）机制。至于这两种机制的区别，我会在后面详细展开。

使用 epoll 进行网络程序的编写，需要三个步骤，分别是 epoll_create，epoll_ctl 和 epoll_wait。接下来我对这几个 API 详细展开讲一下。

epoll_create

复制代码

```
int epoll_create(int size);
int epoll_create1(int flags);
        返回值：若成功返回一个大于 0 的值，表示 epoll 实例；若返回 -1 表示出错
```

epoll_create() 方法创建了一个 epoll 实例，从 Linux 2.6.8 开始，参数 size 被自动忽略，但是该值仍需要一个大于 0 的整数。这个 epoll 实例被用来调用 epoll_ctl 和 epoll_wait，

如果这个 epoll 实例不再需要，比如服务器正常关机，需要调用 close() 方法释放 epoll 实例，这样系统内核可以回收 epoll 实例所分配使用的内核资源。

关于这个参数 size，在一开始的 epoll_create 实现中，是用来告知内核期望监控的文件描述字大小，然后内核使用这部分的信息来初始化内核数据结构，在新的实现中，这个参数不再被需要，因为内核可以动态分配需要的内核数据结构。我们只需要注意，每次将 size 设置成一个大于 0 的整数就可以了。

epoll_create1() 的用法和 epoll_create() 基本一致，如果 epoll_create1() 的输入 size 大小为 0，则和 epoll_create() 一样，内核自动忽略。可以增加如 EPOLL_CLOEXEC 的额外选项，如果你有兴趣的话，可以研究一下这个选项有什么意义。

epoll_ctl

复制代码

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
        返回值：若成功返回 0；若返回 -1 表示出错
```

在创建完 epoll 实例之后，可以通过调用 epoll_ctl 往这个 epoll 实例增加或删除监控的事件。函数 epll_ctl 有 4 个入口参数。

第一个参数 epfd 是刚刚调用 epoll_create 创建的 epoll 实例描述字，可以简单理解成是 epoll 句柄。

第二个参数表示增加还是删除一个监控事件，它有三个选项可供选择：

EPOLL_CTL_ADD： 向 epoll 实例注册文件描述符对应的事件；

EPOLL_CTL_DEL：向 epoll 实例删除文件描述符对应的事件；

EPOLL_CTL_MOD： 修改文件描述符对应的事件。

第三个参数是注册的事件的文件描述符，比如一个监听套接字。

第四个参数表示的是注册的事件类型，并且可以在这个结构体里设置用户需要的数据，其中最为常见的是使用联合结构里的 fd 字段，表示事件所对应的文件描述符。

复制代码

```
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 */
 };
```

我们在前面介绍 poll 的时候已经接触过基于 mask 的事件类型了，这里 epoll 仍旧使用了同样的机制，我们重点看一下这几种事件类型：

EPOLLIN：表示对应的文件描述字可以读；

EPOLLOUT：表示对应的文件描述字可以写；

EPOLLRDHUP：表示套接字的一端已经关闭，或者半关闭；

EPOLLHUP：表示对应的文件描述字被挂起；

EPOLLET：设置为 edge-triggered，默认为 level-triggered。

epoll_wait

复制代码

```
int epoll_wait(int epfd, struct epoll_event *events, int maxevents, int timeout);
  返回值：成功返回的是一个大于 0 的数，表示事件的个数；返回 0 表示的是超时时间到；若出错返回 -1
```

epoll_wait() 函数类似之前的 poll 和 select 函数，调用者进程被挂起，在等待内核 I/O 事件的分发。

这个函数的第一个参数是 epoll 实例描述字，也就是 epoll 句柄。

第二个参数返回给用户空间需要处理的 I/O 事件，这是一个数组，数组的大小由 epoll_wait 的返回值决定，这个数组的每个元素都是一个需要待处理的 I/O 事件，其中

events 表示具体的事件类型，事件类型取值和 epoll_ctl 可设置的值一样，这个 epoll_event 结构体里的 data 值就是在 epoll_ctl 那里设置的 data，也就是用户空间和内核空间调用时需要的数据。

第三个参数是一个大于 0 的整数，表示 epoll_wait 可以返回的最大事件值。

第四个参数是 epoll_wait 阻塞调用的超时值，如果这个值设置为 -1，表示不超时；如果设置为 0 则立即返回，即使没有任何 I/O 事件发生。

epoll 例子

代码解析

下面我们把原先基于 poll 的服务器端程序改造成基于 epoll 的：

复制代码

```
#include "lib/common.h"

#define MAXEVENTS 128

char rot13_char(char c) {
    if ((c >= 'a' && c <= 'm') || (c >= 'A' && c <= 'M'))
        return c + 13;
    else if ((c >= 'n' && c <= 'z') || (c >= 'N' && c <= 'Z'))
        return c - 13;
    else
        return c;
}

int main(int argc, char **argv) {
    int listen_fd, socket_fd;
    int n, i;
    int efd;
    struct epoll_event event;
    struct epoll_event *events;

    listen_fd = tcp_nonblocking_server_listen(SERV_PORT);

    efd = epoll_create1(0);
    if (efd == -1) {
        error(1, errno, "epoll create failed");
    }

    event.data.fd = listen_fd;
    event.events = EPOLLIN | EPOLLET;
```

```
    if (epoll_ctl(efd, EPOLL_CTL_ADD, listen_fd, &event) == -1) {
        error(1, errno, "epoll_ctl add listen fd failed");
    }

    /* Buffer where events are returned */
    events = calloc(MAXEVENTS, sizeof(event));

    while (1) {
        n = epoll_wait(efd, events, MAXEVENTS, -1);
        printf("epoll_wait wakeup\n");
        for (i = 0; i < n; i++) {
            if ((events[i].events & EPOLLERR) ||
                (events[i].events & EPOLLHUP) ||
                (!(events[i].events & EPOLLIN))) {
                fprintf(stderr, "epoll error\n");
                close(events[i].data.fd);
                continue;
            } else if (listen_fd == events[i].data.fd) {
                struct sockaddr_storage ss;
                socklen_t slen = sizeof(ss);
                int fd = accept(listen_fd, (struct sockaddr *) &ss, &slen);
                if (fd < 0) {
                    error(1, errno, "accept failed");
                } else {
                    make_nonblocking(fd);
                    event.data.fd = fd;
                    event.events = EPOLLIN | EPOLLET; //edge-triggered
                    if (epoll_ctl(efd, EPOLL_CTL_ADD, fd, &event) == -1) {
                        error(1, errno, "epoll_ctl add connection fd failed");
                    }
                }
                continue;
            } else {
                socket_fd = events[i].data.fd;
                printf("get event on socket fd == %d \n", socket_fd);
                while (1) {
                    char buf[512];
                    if ((n = read(socket_fd, buf, sizeof(buf))) < 0) {
                        if (errno != EAGAIN) {
                            error(1, errno, "read error");
                            close(socket_fd);
                        }
                        break;
                    } else if (n == 0) {
                        close(socket_fd);
                        break;
                    } else {
                        for (i = 0; i < n; ++i) {
                            buf[i] = rot13_char(buf[i]);
                        }
                        if (write(socket_fd, buf, n) < 0) {
                            error(1, errno, "write error");
```

```
82                          }
83                      }
84                  }
85              }
86          }
87      }
88
89      free(events);
90      close(listen_fd);
91  }
```

程序的第 23 行调用 epoll_create0 创建了一个 epoll 实例。

28-32 行，调用 epoll_ctl 将监听套接字对应的 I/O 事件进行了注册，这样在有新的连接建立之后，就可以感知到。注意这里使用的是 edge-triggered（边缘触发）。

35 行为返回的 event 数组分配了内存。

主循环调用 epoll_wait 函数分发 I/O 事件，当 epoll_wait 成功返回时，通过遍历返回的 event 数组，就直接可以知道发生的 I/O 事件。

第 41-46 行判断了各种错误情况。

第 47-61 行是监听套接字上有事件发生的情况下，调用 accept 获取已建立连接，并将该连接设置为非阻塞，再调用 epoll_ctl 把已连接套接字对应的可读事件注册到 epoll 实例中。这里我们使用了 event_data 里面的 fd 字段，将连接套接字存储其中。

第 63-84 行，处理了已连接套接字上的可读事件，读取字节流，编码后再回应给客户端。

实验

启动该服务器：

复制代码

```
1 $./epoll01
2 epoll_wait wakeup
3 epoll_wait wakeup
4 epoll_wait wakeup
5 get event on socket fd == 6
```

```
epoll_wait wakeup
get event on socket fd == 5
epoll_wait wakeup
get event on socket fd == 5
epoll_wait wakeup
get event on socket fd == 6
epoll_wait wakeup
get event on socket fd == 6
epoll_wait wakeup
get event on socket fd == 6
epoll_wait wakeup
get event on socket fd == 5
```

再启动几个 telnet 客户端，可以看到有连接建立情况下，epoll_wait 迅速从挂起状态结束；并且套接字上有数据可读时，epoll_wait 也迅速结束挂起状态，这时候通过 read 可以读取套接字接收缓冲区上的数据。

复制代码

```
$telnet 127.0.0.1 43211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
fasfsafas
snfsfnsnf
^]
telnet> quit
Connection closed.
```

edge-triggered VS level-triggered

对于 edge-triggered 和 level-triggered， 官方的说法是一个是边缘触发，一个是条件触发。也有文章从电子脉冲角度来解读的，总体上，给初学者的带来的感受是理解上有困难。

我在文稿里面给了两个程序，我们用这个程序来说明一下这两者之间的不同。

在这两个程序里，即使已连接套接字上有数据可读，我们也不调用 read 函数去读，只是简单地打印出一句话。

第一个程序我们设置为 edge-triggered，即边缘触发。开启这个服务器程序，用 telnet 连接上，输入一些字符，我们看到，服务器端只从 epoll_wait 中苏醒过一次，就是第一次有数据可读的时候。

复制代码

```
$./epoll02
epoll_wait wakeup
epoll_wait wakeup
get event on socket fd == 5
```

复制代码

```
$telnet 127.0.0.1 43211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
asfafas
```

第二个程序我们设置为 level-triggered，即条件触发。然后按照同样的步骤来一次，观察服务器端，这一次我们可以看到，服务器端不断地从 epoll_wait 中苏醒，告诉我们有数据需要读取。

复制代码

```
$./epoll03
epoll_wait wakeup
epoll_wait wakeup
get event on socket fd == 5
epoll_wait wakeup
get event on socket fd == 5
epoll_wait wakeup
get event on socket fd == 5
epoll_wait wakeup
get event on socket fd == 5
...
```

这就是两者的区别，条件触发的意思是只要满足事件的条件，比如有数据需要读，就一直不断地把这个事件传递给用户；而边缘触发的意思是只有第一次满足条件的时候才触发，之后

就不会再传递同样的事件了。

一般我们认为，边缘触发的效率比条件触发的效率要高，这一点也是 epoll 的杀手锏之一。

epoll 的历史

早在 Linux 实现 epoll 之前，Windows 系统就已经在 1994 年引入了 IOCP，这是一个异步 I/O 模型，用来支持高并发的网络 I/O，而著名的 FreeBSD 在 2000 年引入了 Kqueue——一个 I/O 事件分发框架。

Linux 在 2002 年引入了 epoll，不过相关工作的讨论和设计早在 2000 年就开始了。如果你感兴趣的话，可以http://lkml.iu.edu/hypermail/linux/kernel/0010.3/0003.html"> 点击这里看一下里面的讨论。

为什么 Linux 不把 FreeBSD 的 kqueue 直接移植过来，而是另辟蹊径创立了 epoll 呢?

让我们先看下 kqueue 的用法，kqueue 也需要先创建一个名叫 kqueue 的对象，然后通过这个对象，调用 kevent 函数增加感兴趣的事件，同时，也是通过这个 kevent 函数来等待事件的发生。

复制代码

```
int kqueue(void);
int kevent(int kq, const struct kevent *changelist, int nchanges,
           struct kevent *eventlist, int nevents,
           const struct timespec *timeout);
void EV_SET(struct kevent *kev, uintptr_t ident, short filter,
            u_short flags, u_int fflags, intptr_t data, void *udata);

struct kevent {
  uintptr_t  ident;      /* identifier (e.g., file descriptor) */
  short      filter;    /* filter type (e.g., EVFILT_READ) */
  u_short    flags;      /* action flags (e.g., EV_ADD) */
  u_int      fflags;    /* filter-specific flags */
  intptr_t   data;      /* filter-specific data */
  void       *udata;     /* opaque user data */
};
```

Linus 在他最初的设想里，提到了这么一句话，也就是说他觉得类似 select 或 poll 的数组方式是可以的，而队列方式则是不可取的。

So sticky arrays of events are good, while queues are bad. Let’ s take that as one of the fundamentals.

在最初的设计里，Linus 等于把 keque 里面的 kevent 函数拆分了两个部分，一部分负责事件绑定，通过 bind_event 函数来实现；另一部分负责事件等待，通过 get_events 来实现。

复制代码

```
struct event {
     unsigned long id; /* file descriptor ID the event is on */
     unsigned long event; /* bitmask of active events */
};

int bind_event(int fd, struct event *event);
int get_events(struct event * event_array, int maxnr, struct timeval *tmout);
```

和最终的 epoll 实现相比，前者类似 epoll_ctl，后者类似 epoll_wait，不过原始的设计里没有考虑到创建 epoll 句柄，在最终的实现里增加了 epoll_create，支持了 epoll 句柄的创建。

2002 年，epoll 最终在 Linux 2.5.44 中首次出现，在 2.6 中趋于稳定，为 Linux 的高性能网络 I/O 画上了一段句号。

总结

Linux 中 epoll 的出现，为高性能网络编程补齐了最后一块拼图。epoll 通过改进的接口设计，避免了用户态 - 内核态频繁的数据拷贝，大大提高了系统性能。在使用 epoll 的时候，我们一定要理解条件触发和边缘触发两种模式。条件触发的意思是只要满足事件的条件，比如有数据需要读，就一直不断地把这个事件传递给用户；而边缘触发的意思是只有第一次满足条件的时候才触发，之后就不会再传递同样的事件了。

思考题

理解完了 epoll，和往常一样，我给你布置两道思考题：

第一道，你不妨试着修改一下第 20 讲中 select 的例子，即在已连接套接字上有数据可读，也不调用 read 函数去读，看一看你的结果，你认为 select 是边缘触发的，还是条件触发的?

第二道，同样的修改一下第 21 讲 poll 的例子，看看你的结果，你认为 poll 是边缘触发的，还是条件触发的?

你可以在 GitHub 上上传你的代码，并写出你的疑惑，我会和你一起交流，也欢迎把这篇文章分享给你的朋友或者同事，一起交流一下。

上一篇　22 | 非阻塞I/O：提升性能的加速器

精选留言 (5)

写留言

空想家
2019-09-30

LT + non-blocking 和 ET + non-blocking 有什么区别吗？性能谁更好一点?

epoll 的惊群问题会讲吗?

1

Hale
2019-09-30

epoll这两种模式有使用的不同场景吗？既然边缘触发优于条件触发，那什么场景下会使用条件触发?

Hale
2019-09-30

poll是条件触发，只要条件满足，每次都会触发

展开

刘丹
2019-09-30

提个小建议，能否把代码解说（例如：第 41-46 行判断了各种错误情况）作为注释放在代码里?

Steiner
2019-09-29

epoll_wait的意思是把发生的事件和fd装载到events这个数组里吗