我对网络IO的理解，你理解了吗？

作者：爱比特编程 2019-12-26 09:15:44

存储 # 存储软件

Unix/Linux系统下IO主要分为磁盘IO，网络IO，我今天主要说一下对网络IO的理解，网络IO主要是socket套接字的读(read)、写(write)，socket在Linux系统被抽象为流(stream)。

Unix/Linux系统下IO主要分为磁盘IO，网络IO，我今天主要说一下对网络IO的理解，网络IO主要是socket套接字的读(read)、写(write)，socket在Linux系统被抽象为流(stream)。

网络IO模型

在Unix/Linux系统下，IO分为两个不同阶段：

- 等待数据准备好
- 从内核向进程复制数据

阻塞式I/O

阻塞式I/O(blocking I/O)是最简单的一种，默认情况下，socket 套接字的系统调用都是阻塞的，我以recv/recvfrom 理解一下网络IO的模型。当应用层的系统调用recv/recvfrom时，开启Linux的系统调用，开始准备数据，然后将数据从内核态复制到用户态，然后通知应用程序获取数据，整个过程都是阻塞的。两个阶段都会被阻塞。

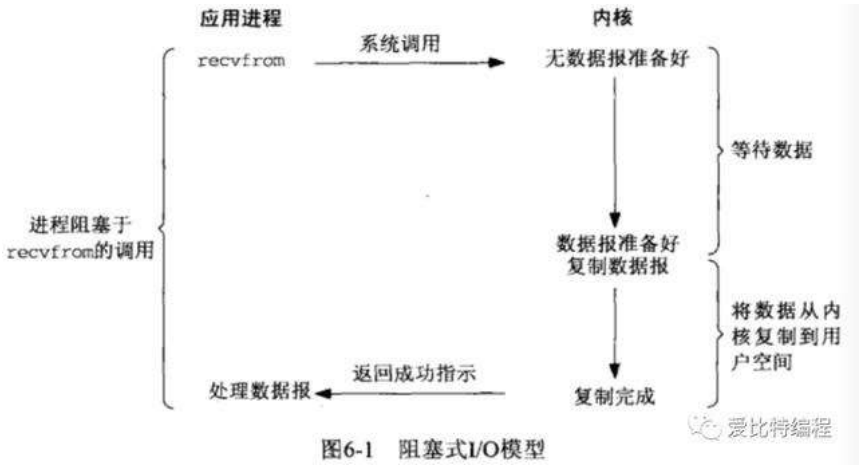


图6-1 阻塞式I/O模型

相似话题

容灾备份
77内容

存储架构
36内容

存储设备
153内容

超
205内

查看全

51CTO技术栈公众号

编辑推荐

再谈Raid 5和Raid 6的写性能

ECC内存和普通内存有什么区别 内存吗

云存储技术的原理是什么？百析

分布式存储与传统SAN、NAS与劣势？

Redis内存淘汰策略，看这一

相关专题

戴尔超融合研讨会

终
最
202

202
及
202

我收藏的内

图片来源于《Unix网络编程卷1》

阻塞I/O下开发的后台服务，一般都是通过多进程或者线程取出来请求，但是开辟进程或者线程是非常消耗系统资源的，当大量请求时，因为需要开辟更多的进程或者线程有可能将系统资源耗尽，因此这种模式不适合高并发的系统。

非阻塞式I/O

非阻塞IO(non-blocking I/O)在调用后，内核马上返回给进程，如果数据没有准备好，就返回一个error，进程可以先去干其他事情，一会再次调用，直到数据准备好为止，循环往返的系统调用的过程称为轮询(pool)，然后在从内核态将数据拷贝到用户态，但是这个拷贝的过程还是阻塞的。

我还是以recv/recvfrom为例说一下，首选需要将socket套接字设置成为非阻塞，进程开始调用recv/recvfrom，如果内核没有准备好数据时，立即返回给进程一个error码(在Linux下是EAGINE的错误码)，进程接到error返回后，先去干其他的事情，进入了轮询，只等到数据准备好，然后将数据拷贝到用户态。

需要通过ioctl 函数将socket套接字设置成为非阻塞

```
1.    ioctl(fd, FIONBIO, &nb);
```

复制

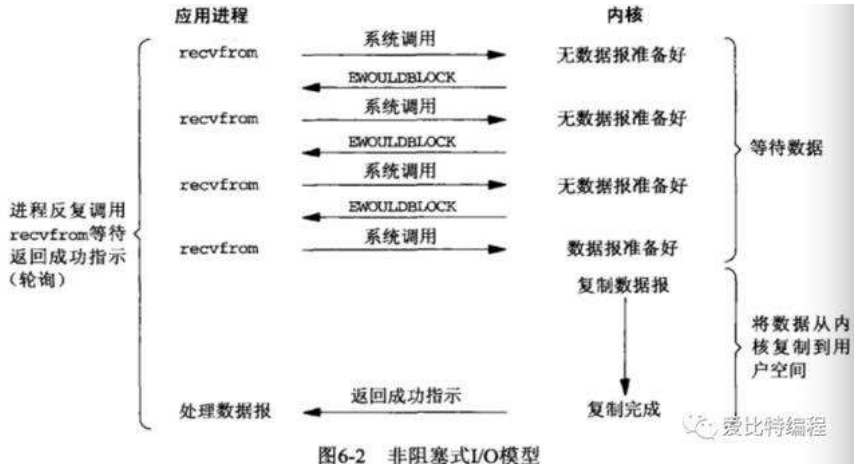


图6-2 非阻塞式I/O模型

非阻塞I/O模型

图片来源于《Unix网络编程卷1》

非阻塞I/O的第一阶段不会阻塞，但是第二个阶段将数据从内核态拷贝到用户态时会有阻塞。在开发后台服务，由于非阻塞I/O需要通过轮询的方式去知道是否数据准备好，轮询的方式特别耗CPU的资源。

I/O多路复用

在Linux下提供一种I/O多路复用(I/O multiplexing)的机制，这个机制允许同时监听多个socket套接字描述符fd，一旦某个fd就绪(就绪一般是有数据可读或者可写)时，能够通知进程进行相应的读写操作。

在Linux下有三个I/O多路复用的函数Select、Poll、Epoll，但是它们都是同步IO，因为它们都需要在数据准备好后，读写数据是阻塞的。

51CTO技术栈公众号

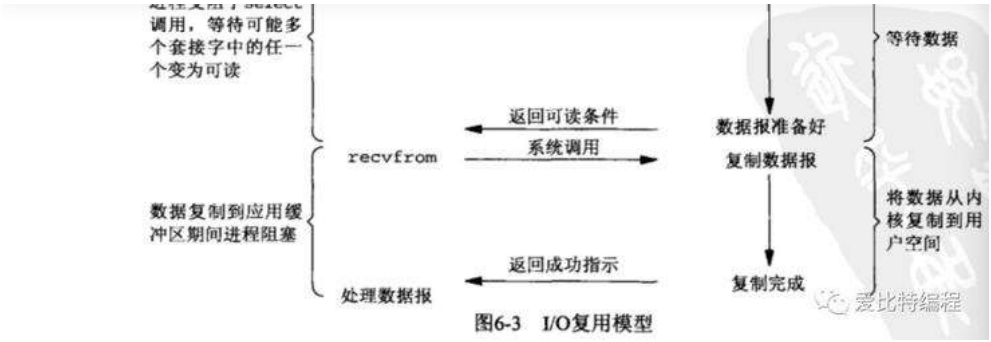


图6-3 I/O复用模型

I/O多路复用模型

图片来源于《Unix网络编程卷1》

I/O多路复用是Linux处理高并发的技术，Epoll比Select、Poll性能更优越，后面会讲到它们的区别。优秀的高并发服务例如Nginx、Redis都是采用Epoll+Non-Blocking I/O的模式。

信号驱动式I/O

信号驱动式I/O是通过信号的方式通知数据准备好，然后再讲数据拷贝到应用层，拷贝阶段也是阻塞的。

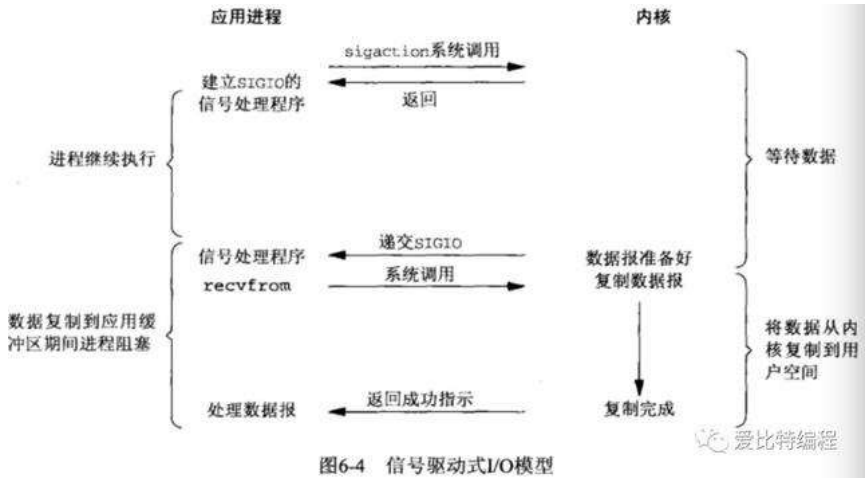


图6-4 信号驱动式I/O模型

信号驱动式I/O

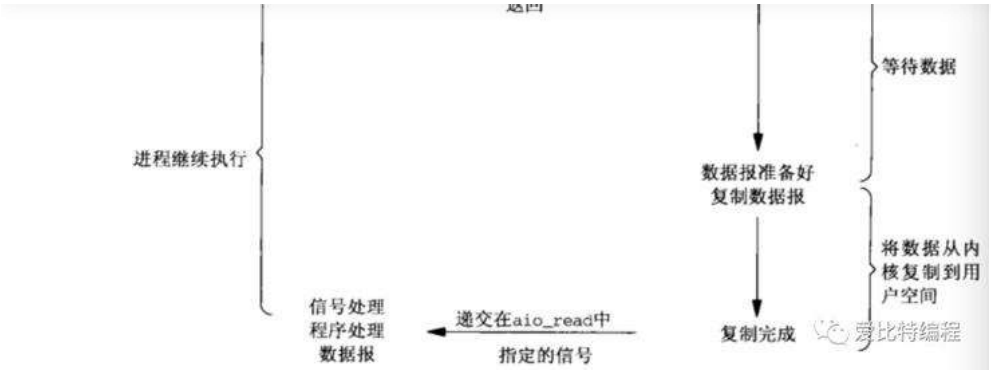
图片来源于《Unix网络编程卷1》

异步I/O

异步I/O(asynchronous I/O或者AIO)，数据准备通知和数据拷贝两个阶段都在内核态完成，两个阶段都不会阻塞，真正的异步I/O。

进程调用read/readfrom时，内核立刻返回，进程不会阻塞，进程可以去干其他的事情，当内核通知进程数据已经完成后，进程直接可以处理数据，不需要再拷贝数据，因为内核已经将数据从内核态拷贝到用户态，进程可以直接处理数据。

51CTO技术栈公众号



异步I/O模型

图片来源于《Unix网络编程卷1》

Linux对AIO支持不好，因此使用的不是太广泛。

同步和异步区别、阻塞和非阻塞的区别

同步和异步区别

对于这两个东西，POSIX其实是有官方定义的。A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;An asynchronous I/O operation does not cause the requesting process to be blocked;

一个同步I/O操作会引起请求进程阻塞，只到这个I/O请求结束。

一个异步I/O操作不会引起请求进程阻塞。

从这个官方定义中，不管是Blocking I/O还是Non-Blocking I/O，其实都是synchronous I/O。因为它们一定都会阻塞在第二阶段拷贝数据那里。只有异步IO才是异步的。

同步	IO multiplexing (select/poll/epoll)		
	阻塞	非阻塞	
异步	Linux	Windows	.NET
	AIO	IOCP	BeginInvoke/EndInvoke

爱比特编程

同步异步对比

图片来源于知乎

阻塞和非阻塞的区别

阻塞和非阻塞主要区别其实是在第一阶段等待数据的时候。但是在第二阶段，阻塞和非阻塞其实是没有区别的。程序必须等待内核把收到的数据复制到进程缓冲区来。换句话说，非阻塞也不是真的一点都不"阻塞"，只是在不能立刻得到结果的时候不会傻乎乎地等在那里而已。

IO多路复用

Select、Poll、Epoll的区别

Select、poll、epoll都是I/O多路复用的机制，I/O多路复用就是通过一种机制，一个进程可以监视多个文件描述符fd，一旦某个描述符就绪(一般是读就绪或者写就绪)，能够通知程序进行相应的读写操作。但select，poll，epoll本质上都是同

51CTO技术栈公众号

select

```
int select (int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct ti…  *
```

复制

select 函数监视的文件描述符分3类，分别是writefds、readfds、和exceptfds。调用后select函数会阻塞，直到有描述符就绪(有数据 可读、可写、或者有except)，或者超时(timeout指定等待时间，如果立即返回设为null即可)，函数返回。当select函数返回后，可以 通过遍历fdset，来找到就绪的描述符。

select支持几乎所有的平台，跨平台是它的优点。

select缺点是：1)单个进程支持监控的文件描述符数量有限，Linux下一般是1024，可以修改提升限制，但是会造成效率低下。2)select通过轮询方式通知消息，效率比较低。

51CTO技术栈公众号

poll

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

复制

不同于select使用三个位图来表示三个fdset的方式，poll使用一个pollfd的指针实现。

```
struct pollfd {
 int fd; /* file descriptor */
 short events; /* requested events to watch */
 short revents; /* returned events witnessed */
};
```

复制

pollfd结构包含了要监视的event和发生的event，不再使用select“参数-值”传递的方式。同时，pollfd并没有最大数量限制(但是数量过大后性能也是会下降)。和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符。

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。

epoll

epoll是在2.6内核中提出的，是之前的select和poll的增强版本，是Linux特有的。相对于select和poll来说，epoll更加灵活，没有描述符限制。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。

```
int epoll_create(int size)；//创建一个epoll的句柄，size用来告诉内核这个监听的数目一共有多大
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)；
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

复制

执行epoll_create时，创建了红黑树和就绪list链表;执行epoll_ctl时，如果增加fd，则检查在红黑树中是否存在，存在则立即返回，不存在则添加到红黑树中，然后向内核注册回调函数，用于当中断事件到来时向准备就绪的list链表中插入数据。执行epoll_wait时立即返回准备就绪链表里的数据即可。

工作模式

1. LT模式

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。如果你不做任何操作，内核还是会继续通知你的。

2. ET模式

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

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。

详细对比

系统调用	select	poll	epoll
事件集合	内核会修改用户注册的文件描述符集来反馈其中的就绪事件，这使得用户每次调用select都要重置这三个文件描述符集	使用pollfd.events传入事件，使用pollfd.revents反馈就绪事件	使用内核事件表来管理用户事件，epoll_wait的events仅用来保存就绪事件
应用程序索引就绪文件描述符的时间复杂度	O(n)	O(n)	O(1)
最大支持文件描述符个数	有限制，一般是1024	65535	65535
工作模式	LT	LT	支持ET高效模式
内核实现	采用轮询方式	采用轮询方式	采用回调方式

三种I/O多路复用对比

Nginx中Epoll+非阻塞IO

Nginx高并发主要是通过Epoll模式+非阻塞I/O

Nginx对I/O多路复用进行封装，封装在结构体struct ngx_event_s，同时将事件封装在ngx_event_actions_t结构中。

复制

```
typedef struct {
    ngx_int_t (*add)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    ngx_int_t (*del)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    ngx_int_t (*enable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);
    ngx_int_t (*disable)(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags);

    ngx_int_t (*add_conn)(ngx_connection_t *c);
    ngx_int_t (*del_conn)(ngx_connection_t *c, ngx_uint_t flags);

    ngx_int_t (*notify)(ngx_event_handler_pt handler);

    ngx_int_t (*process_events)(ngx_cycle_t *cycle, ngx_msec_t timer,
    ngx_uint_t flags);

    ngx_int_t (*init)(ngx_cycle_t *cycle, ngx_msec_t timer);
    void (*done)(ngx_cycle_t *cycle);
} ngx_event_actions_t;
```

初始化epoll句柄

复制

```
static ngx_int_t
ngx_epoll_init(ngx_cycle_t *cycle, ngx_msec_t timer)
{
    ngx_epoll_conf_t *epcf;

```

51CTO技术栈公众号

```
    ep = epoll_create(cycle->connection_n / 2);

    if (ep == -1) {
    ngx_log_error(NGX_LOG_EMERG, cycle->log, ngx_errno,
    "epoll_create() failed");
    return NGX_ERROR;
    }
    ...
    }
}
```

将fd设置为非阻塞

复制

```
(ngx_nonblocking(s) == -1) #nginx将fd设置非阻塞
```

设置事件触发

复制

```
static ngx_int_t
ngx_epoll_add_event(ngx_event_t *ev, ngx_int_t event, ngx_uint_t flags)
{
    int op;
    uint32_t events, prev;
    ngx_event_t *e;
    ngx_connection_t *c;
    struct epoll_event ee;

    c = ev->data;

    events = (uint32_t) event;

    if (event == NGX_READ_EVENT) {
    e = c->write;
    prev = EPOLLOUT;
#if (NGX_READ_EVENT != EPOLLIN|EPOLLRDHUP)
    events = EPOLLIN|EPOLLRDHUP;
#endif

    } else {
    e = c->read;
    prev = EPOLLIN|EPOLLRDHUP;
#if (NGX_WRITE_EVENT != EPOLLOUT)
    events = EPOLLOUT;
#endif
    }

    if (e->active) {
    op = EPOLL_CTL_MOD;
    events |= prev;

    } else {
    op = EPOLL_CTL_ADD;
    }
```

51CTO技术栈公众号

```
    events &= ~EPOLLRDHUP;
    }
#endif

    ee.events = events | (uint32_t) flags;
    ee.data.ptr = (void *) ((uintptr_t) c | ev->instance);

    ngx_log_debug3(NGX_LOG_DEBUG_EVENT, ev->log, 0,
    "epoll add event: fd:%d op:%d ev:%08XD",
    c->fd, op, ee.events);

    if (epoll_ctl(ep, op, c->fd, &ee) == -1) {
    ngx_log_error(NGX_LOG_ALERT, ev->log, ngx_errno,
    "epoll_ctl(%d, %d) failed", op, c->fd);
    return NGX_ERROR;
    }

    ev->active = 1;
#if 0
    ev->oneshot = (flags & NGX_ONESHOT_EVENT) ? 1 : 0;
#endif

    return NGX_OK;
}
```

处理就绪的事件

复制

```
static ngx_int_t
ngx_epoll_process_events(ngx_cycle_t *cycle, ngx_msec_t timer, ngx_uint_t flags)
{
    int events;
    uint32_t revents;
    ngx_int_t instance, i;
    ngx_uint_t level;
    ngx_err_t err;
    ngx_event_t *rev, *wev;
    ngx_queue_t *queue;
    ngx_connection_t *c;

    /* NGX_TIMER_INFINITE == INFTIM */

    ngx_log_debug1(NGX_LOG_DEBUG_EVENT, cycle->log, 0,
    "epoll timer: %M", timer);

    events = epoll_wait(ep, event_list, (int) nevents, timer);
    ...
}
```

51CTO技术栈公众号

责任编辑：武晓燕 来源： 今日头条