高级IO

本节重点

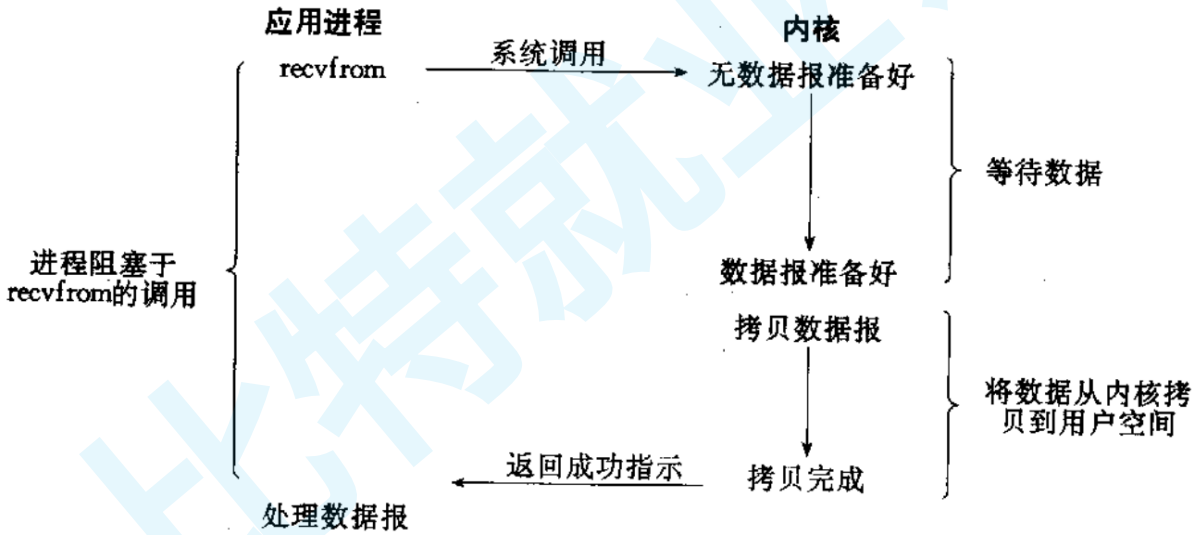
> - 理解五种IO模型的基本概念, 重点是IO多路转接.
> - 掌握select编程模型, 能够实现select版本的TCP服务器.
> - 掌握poll编程模型, 能够实现poll版本的TCP服务器.
> - 掌握epoll编程模型, 能够实现epoll版本的TCP服务器.
> - 理解epoll的LT模式和ET模式.
> - 理解select和epoll的优缺点对比.

五种IO模型

[钓鱼例子]

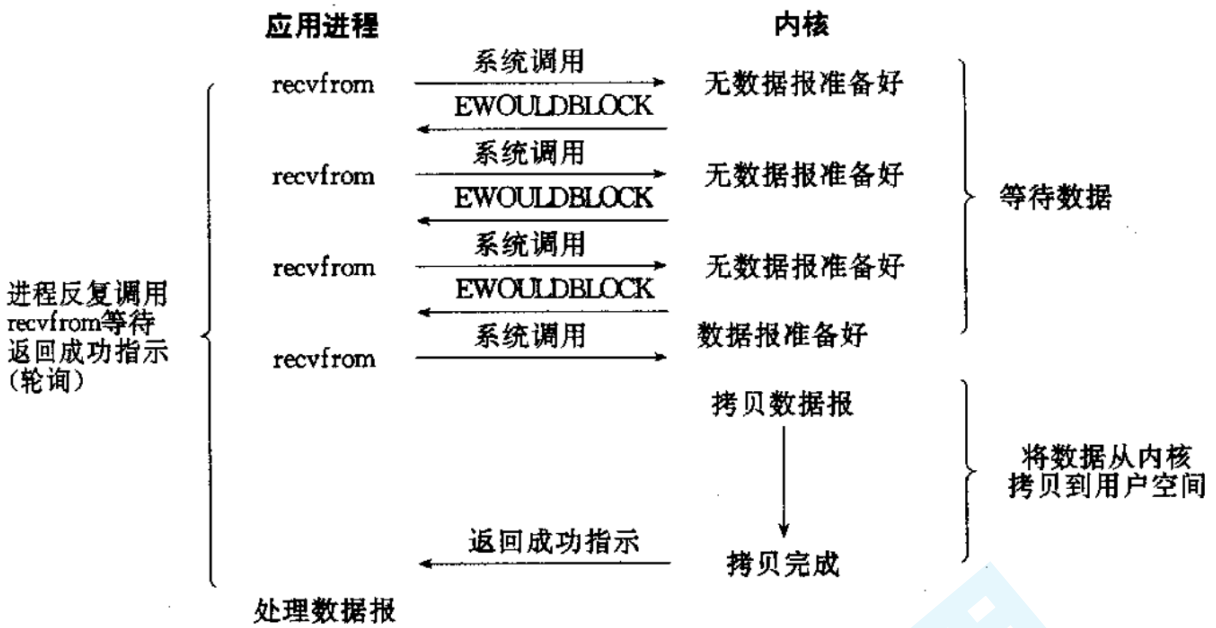
> - 阻塞IO: 在内核将数据准备好之前, 系统调用会一直等待. 所有的套接字, 默认都是阻塞方式.

阻塞IO是最常见的IO模型.



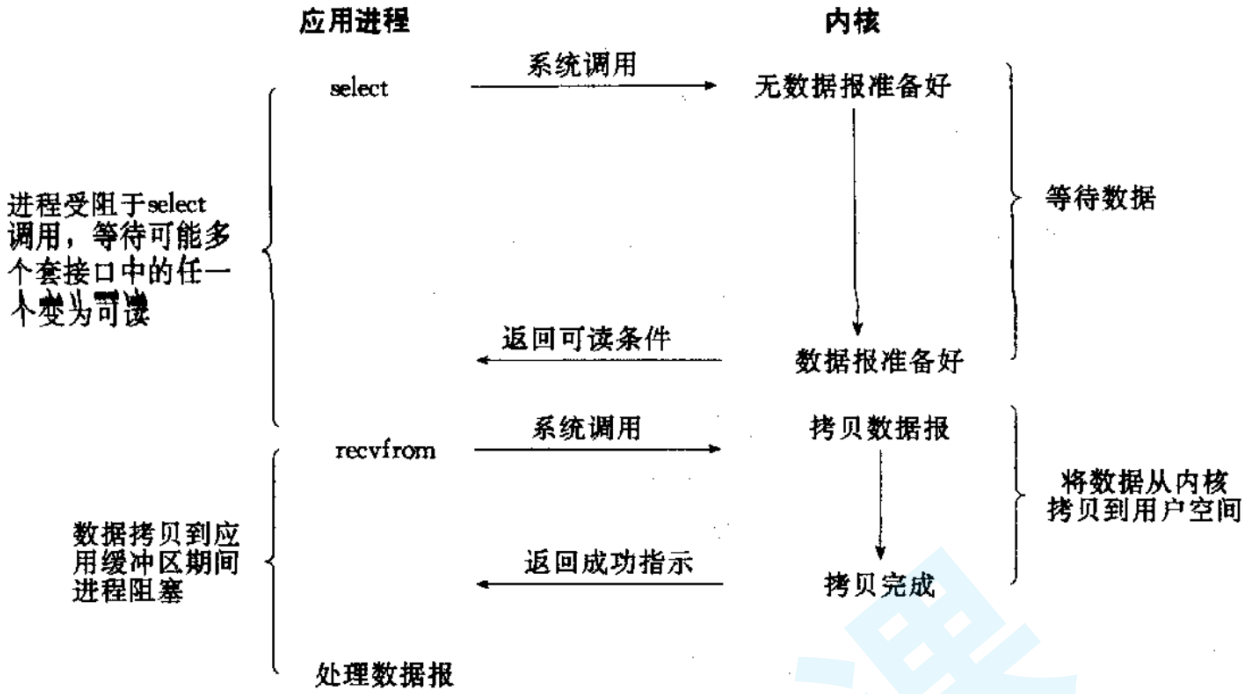
> - 非阻塞IO: 如果内核还未将数据准备好, 系统调用仍然会直接返回, 并且返回EWOULDBLOCK错误码.

非阻塞IO往往需要程序员循环的方式反复尝试读写文件描述符, 这个过程称为**轮询**. 这对CPU来说是较大的浪费, 一般只有特定场景下才使用.

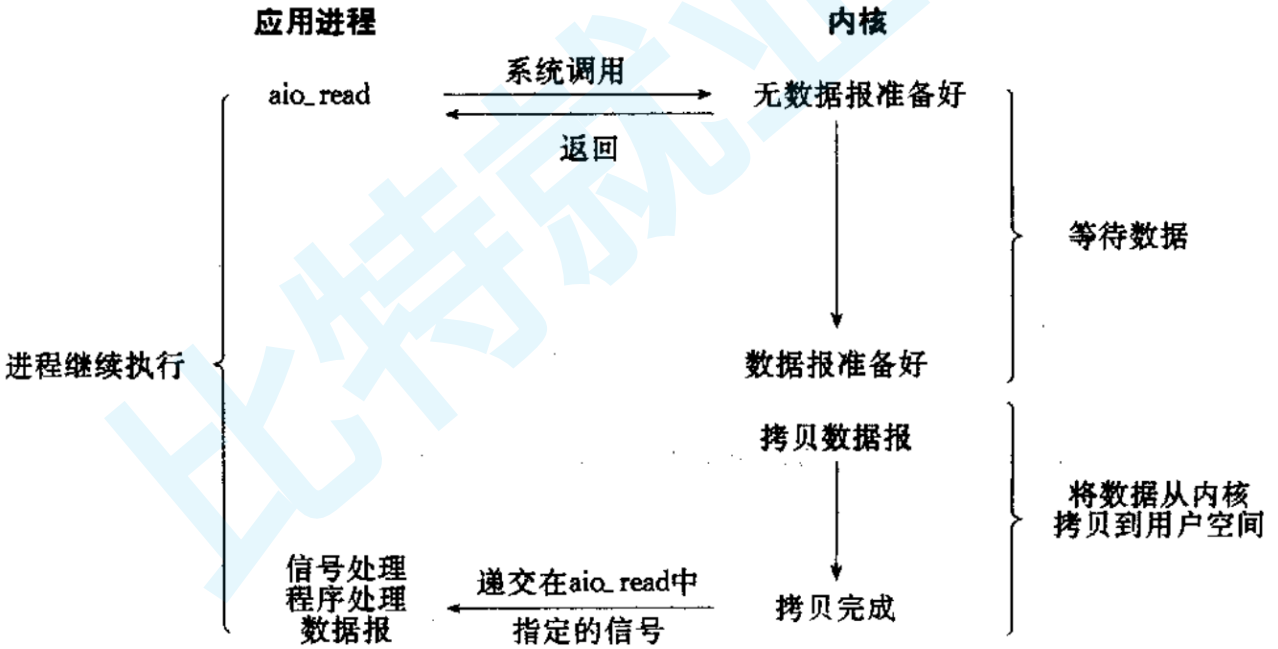


- 信号驱动IO: 内核将数据准备好的时候, 使用SIGIO信号通知应用程序进行IO操作.

- IO多路转接: 虽然从流程图上看起来和阻塞IO类似. 实际上最核心在于IO多路转接能够同时等待多个文件描述符的就绪状态.



- 异步IO: 由内核在数据拷贝完成时, 通知应用程序(而信号驱动是告诉应用程序何时可以开始拷贝数据).



小结

- 任何IO过程中, 都包含两个步骤. 第一是**等待**, 第二是**拷贝**. 而且在实际的应用场景中, 等待消耗的时间往往都远远高于拷贝的时间. 让IO更高效, 最核心的办法就是让等待的时间尽量少.

高级IO重要概念

在这里, 我们要强调几个概念

同步通信 vs 异步通信(synchronous communication/ asynchronous communication)

同步和异步关注的是消息通信机制.

> - 所谓同步，就是在发出一个**调用**时，在没有得到结果之前，该**调用**就不返回. 但是一旦调用返回，就得到返回值了; 换句话说，就是由**调用者**主动等待这个**调用**的结果;
> - 异步则是相反，**调用**在发出之后，这个调用就直接返回了，所以没有返回结果; 换句话说，当一个异步过程调用发出后，调用者不会立刻得到结果; 而是在**调用**发出后，**被调用者**通过状态、通知来通知调用者，或通过回调函数处理这个调用.

另外, 我们回忆在讲多进程多线程的时候, 也提到同步和互斥. 这里的同步通信和进程之间的同步是完全不想干的概念.

> - 进程/线程同步也是进程/线程之间直接的制约关系
> - 是为完成某种任务而建立的两个或多个线程，这个线程需要在某些位置上协调他们的工作次序而等待、传递信息所产生的制约关系. 尤其是在访问临界资源的时候.

同学们以后在看到 "同步" 这个词, 一定要先搞清楚大背景是什么. 这个同步, 是同步通信异步通信的同步, 还是同步与互斥的同步.

阻塞 vs 非阻塞

阻塞和非阻塞关注的是程序在等待调用结果（消息，返回值）时的状态.

> - 阻塞调用是指调用结果返回之前，当前线程会被挂起. 调用线程只有在得到结果之后才会返回.
> - 非阻塞调用指在不能立刻得到结果之前，该调用不会阻塞当前线程.

理解这四者的关系

[**妖怪蒸唐僧的例子**]

其他高级IO

非阻塞IO，纪录锁，系统V流机制，I/O多路转接（也叫I/O多路复用）,readv和writev函数以及存储映射IO（mmap），这些统称为高级IO.

我们此处重点讨论的是I/O多路转接

非阻塞IO

fcntl

一个文件描述符, 默认都是阻塞IO.

函数原型如下.

```
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd, ... /* arg */ );
```

传入的cmd的值不同, 后面追加的参数也不相同.

fcntl函数有5种功能:

> - 复制一个现有的描述符（cmd=F_DUPFD）.
> - 获得/设置文件描述符标记(cmd=F_GETFD或F_SETFD).
> - 获得/设置文件状态标记(cmd=F_GETFL或F_SETFL).
> - 获得/设置异步I/O所有权(cmd=F_GETOWN或F_SETOWN).
> - 获得/设置记录锁(cmd=F_GETLK,F_SETLK或F_SETLKW).

我们此处只是用第三种功能, 获取/设置文件状态标记, 就可以将一个文件描述符设置为非阻塞.

实现函数SetNoBlock

基于fcntl, 我们实现一个SetNoBlock函数, 将文件描述符设置为非阻塞.

```
void SetNoBlock(int fd) {
  int fl = fcntl(fd, F_GETFL);
  if (fl < 0) {
    perror("fcntl");
    return;
  }
  fcntl(fd, F_SETFL, fl | O_NONBLOCK);
}
```

> - 使用F_GETFL将当前的文件描述符的属性取出来(这是一个位图).
> - 然后再使用F_SETFL将文件描述符设置回去. 设置回去的同时, 加上一个O_NONBLOCK参数.

轮询方式读取标准输入

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

void SetNoBlock(int fd) {
  int fl = fcntl(fd, F_GETFL);
  if (fl < 0) {
    perror("fcntl");
    return;
  }
  fcntl(fd, F_SETFL, fl | O_NONBLOCK);
}

int main() {
  SetNoBlock(0);
  while (1) {
    char buf[1024] = {0};
    ssize_t read_size = read(0, buf, sizeof(buf) - 1);
    if (read_size < 0) {
      perror("read");
      sleep(1);
      continue;
```

```
    }
    printf("input:%s\n", buf);
  }
  return 0;
}
```

I/O多路转接之select

初识select

系统提供select函数来实现多路复用输入/输出模型.

> - select系统调用是用来让我们的程序监视多个文件描述符的状态变化的;
> - 程序会停在select这里等待，直到被监视的文件描述符有一个或多个发生了状态改变;

select函数原型

select的函数原型如下: #include <sys/select.h>

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

参数解释:

> - 参数nfds是需要监视的最大的文件描述符值+1;
> - rdset,wrset,exset分别对应于需要检测的可读文件描述符的集合，可写文件描述符的集 合及异常文件描述符的集合;
> - 参数timeout为结构timeval，用来设置select()的等待时间

参数timeout取值:

> - NULL：则表示select（）没有timeout，select将一直被阻塞，直到某个文件描述符上发生了事件;
> - 0：仅检测描述符集合的状态，然后立即返回，并不等待外部事件的发生。
> - 特定的时间值：如果在指定的时间段里没有事件发生，select将超时返回。

关于fd_set结构

```
62
63 /* fd_set for select and pselect.  */
64 typedef struct
65   {
66     /* XPG4.2 requires this member name.  Otherwise avoid the name
67        from the global namespace.  */
68 #ifdef __USE_XOPEN
69     __fd_mask fds_bits[__FD_SETSIZE / __NFDBITS];
70 # define __FDS_BITS(set) ((set)->fds_bits)
71 #else
72     __fd_mask __fds_bits[__FD_SETSIZE / __NFDBITS];
73 # define __FDS_BITS(set) ((set)->__fds_bits)
74 #endif
75   } fd_set;
```

```
53 /* The fd_set member is required to be an array of longs.  */
54 typedef long int __fd_mask;
55
```

其实这个结构就是一个整数数组, 更严格的说, 是一个 "位图". 使用位图中对应的位来表示要监视的文件描述符.

提供了一组操作fd_set的接口, 来比较方便的操作位图.

```
    void FD_CLR(int fd, fd_set *set);      // 用来清除描述词组set中相关fd 的位
    int  FD_ISSET(int fd, fd_set *set);    // 用来测试描述词组set中相关fd 的位是否为真
    void FD_SET(int fd, fd_set *set);      // 用来设置描述词组set中相关fd的位
    void FD_ZERO(fd_set *set);             // 用来清除描述词组set的全部位
```

关于timeval结构
timeval结构用于描述一段时间长度，如果在这个时间内，需要监视的描述符没有事件发生则函数返回，返回值为0。

```
28 /* A time value that is accurate to the nearest
29    microsecond but also has a range of years.  */
30 struct timeval
31   {
32     __time_t tv_sec;    /* Seconds.  */
33     __suseconds_t tv_usec;  /* Microseconds.  */
34   };
```

函数返回值：

> - 执行成功则返回文件描述词状态已改变的个数
> - 如果返回0代表在描述词状态改变前已超过timeout时间，没有返回
> - 当有错误发生时则返回-1，错误原因存于errno，此时参数readfds，writefds, exceptfds和timeout的值变成不可预测。

> **错误值可能为：**
>
> - EBADF 文件描述词为无效的或该文件已关闭
> - EINTR 此调用被信号所中断
> - EINVAL 参数n 为负值。
> - ENOMEM 核心内存不足

常见的程序片段如下:

```
fs_set readset;
FD_SET(fd,&readset);
select(fd+1,&readset,NULL,NULL,NULL);
if(FD_ISSET(fd,readset)){……}
```

理解select执行过程

理解select模型的关键在于理解fd_set,为说明方便，取fd_set长度为1字节，fd_set中的每一bit可以对应一个文件描述符fd。则1字节长的fd_set最大可以对应8个fd.

> *（1）执行fd_set set; FD_ZERO(&set);则set用位表示是0000,0000。 *（2）若fd = 5,执行FD_SET(fd,&set);后set变为0001,0000(第5位置为1) *（3）若再加入fd = 2，fd=1,则set变为0001,0011 *（4）执行select(6,&set,0,0,0)阻塞等待 *（5）若fd=1,fd=2上都发生可读事件，则select返回，此时set变为0000,0011。注意：没有事件发生的fd=5被清空。

socket就绪条件

读就绪

- socket内核中, 接收缓冲区中的字节数, 大于等于低水位标记SO_RCVLOWAT. 此时可以无阻塞的读该文件描述符, 并且返回值大于0;
- socket TCP通信中, 对端关闭连接, 此时对该socket读, 则返回0;
- 监听的socket上有新的连接请求;
- socket上有未处理的错误;

写就绪

- socket内核中, 发送缓冲区中的可用字节数(发送缓冲区的空闲位置大小), 大于等于低水位标记SO_SNDLOWAT, 此时可以无阻塞的写, 并且返回值大于0;
- socket的写操作被关闭(close或者shutdown). 对一个写操作被关闭的socket进行写操作, 会触发SIGPIPE信号;
- socket使用非阻塞connect连接成功或失败之后;
- socket上有未读取的错误;

异常就绪(选学)

- socket上收到带外数据. 关于带外数据, 和TCP紧急模式相关(回忆TCP协议头中, 有一个紧急指针的字段), 同学们课后自己收集相关资料.

select的特点

- 可监控的文件描述符个数取决与sizeof(fd_set)的值. 我这边服务器上sizeof(fd_set) = 512，每bit表示一个文件描述符，则我服务器上支持的最大文件描述符是512*8=4096.
- 将fd加入select监控集的同时，还要再使用一个数据结构array保存放到select监控集中的fd，
  - 一是用于再select 返回后，array作为源数据和fd_set进行FD_ISSET判断。
  - 二是select返回后会把以前加入的但并无事件发生的fd清空，则每次开始select前都要重新从array取得fd逐一加入(FD_ZERO最先)，扫描array的同时取得fd最大值maxfd，用于select的第一个参数。

备注: fd_set的大小可以调整，可能涉及到重新编译内核. 感兴趣的同学可以自己去收集相关资料.

select缺点

- 每次调用select, 都需要手动设置fd集合, 从接口使用角度来说也非常不便.
- 每次调用select，都需要把fd集合从用户态拷贝到内核态，这个开销在fd很多时会很大
- 同时每次调用select都需要在内核遍历传递进来的所有fd，这个开销在fd很多时也很大
- select支持的文件描述符数量太小.

select使用示例: 检测标准输入输出

只检测标准输入:

```
#include <stdio.h>
#include <unistd.h>
#include <sys/select.h>

int main() {
  fd_set read_fds;
  FD_ZERO(&read_fds);
  FD_SET(0, &read_fds);
```

```
  for (;;) {
    printf("> ");
    fflush(stdout);
    int ret = select(1, &read_fds, NULL, NULL, NULL);
    if (ret < 0) {
      perror("select");
      continue;
    }
    if (FD_ISSET(0, &read_fds)) {
      char buf[1024] = {0};
      read(0, buf, sizeof(buf) - 1);
      printf("input: %s", buf);
    } else {
      printf("error! invaild fd\n");
      continue;
    }
    FD_ZERO(&read_fds);
    FD_SET(0, &read_fds);
  }
  return 0;
}
```

说明:

> - 当只检测文件描述符0（标准输入）时，因为输入条件只有在你有输入信息的时候，才成立，所以如果一直不输入，就会产生超时信息。

select使用示例

使用 select 实现字典服务器

tcp_select_server.hpp

```
#pragma once
#include <vector>
#include <unordered_map>
#include <functional>
#include <sys/select.h>
#include "tcp_socket.hpp"

// 必要的调试函数
inline void PrintFdSet(fd_set* fds, int max_fd) {
  printf("select fds: ");
  for (int i = 0; i < max_fd + 1; ++i) {
    if (!FD_ISSET(i, fds)) {
      continue;
    }
    printf("%d ", i);
  }
  printf("\n");
}

typedef std::function<void (const std::string& req, std::string* resp)> Handler;
```

```cpp
// 把 Select 封装成一个类. 这个类虽然保存很多 TcpSocket 对象指针, 但是不管理内存
class Selector {
public:
  Selector() {
    // [注意!] 初始化千万别忘了!!
    max_fd_ = 0;
    FD_ZERO(&read_fds_);
  }

  bool Add(const TcpSocket& sock) {
    int fd = sock.GetFd();
    printf("[Selector::Add] %d\n", fd);
    if (fd_map_.find(fd) != fd_map_.end()) {
      printf("Add failed! fd has in Selector!\n");
      return false;
    }
    fd_map_[fd] = sock;
    FD_SET(fd, &read_fds_);
    if (fd > max_fd_) {
      max_fd_ = fd;
    }
    return true;
  }

  bool Del(const TcpSocket& sock) {
    int fd = sock.GetFd();
    printf("[Selector::Del] %d\n", fd);
    if (fd_map_.find(fd) == fd_map_.end()) {
      printf("Del failed! fd has not in Selector!\n");
      return false;
    }
    fd_map_.erase(fd);
    FD_CLR(fd, &read_fds_);

    // 重新找到最大的文件描述符, 从右往左找比较快
    for (int i = max_fd_; i >= 0; --i) {
      if (!FD_ISSET(i, &read_fds_)) {
        continue;
      }
      max_fd_ = i;
      break;
    }
    return true;
  }

  // 返回读就绪的文件描述符集
  bool Wait(std::vector<TcpSocket>* output) {
    output->clear();
    // [注意] 此处必须要创建一个临时变量, 否则原来的结果会被覆盖掉
    fd_set tmp = read_fds_;


    // DEBUG
```

```
    PrintFdSet(&tmp, max_fd_);

    int nfds = select(max_fd_ + 1, &tmp, NULL, NULL, NULL);
    if (nfds < 0) {
      perror("select");
      return false;
    }
    // [注意!] 此处的循环条件必须是 i < max_fd_ + 1
    for (int i = 0; i < max_fd_ + 1; ++i) {
      if (!FD_ISSET(i, &tmp)) {
        continue;
      }
      output->push_back(fd_map_[i]);
    }
    return true;
  }

private:
  fd_set read_fds_;
  int max_fd_;
  // 文件描述符和 socket 对象的映射关系
  std::unordered_map<int, TcpSocket> fd_map_;
};

class TcpSelectServer {
public:
  TcpSelectServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {

  }

  bool Start(Handler handler) const {
    // 1. 创建 socket
    TcpSocket listen_sock;
    bool ret = listen_sock.Socket();
    if (!ret) {
      return false;
    }
    // 2. 绑定端口号
    ret = listen_sock.Bind(ip_, port_);
    if (!ret) {
      return false;
    }
    // 3. 进行监听
    ret = listen_sock.Listen(5);
    if (!ret) {
      return false;
    }
    // 4. 创建 Selector 对象
    Selector selector;
    selector.Add(listen_sock);
    // 5. 进入事件循环
    for (;;) {

      std::vector<TcpSocket> output;
```

```
      bool ret = selector.Wait(&output);
      if (!ret) {
        continue;
      }
      // 6. 根据就绪的文件描述符的差别，决定后续的处理逻辑
      for (size_t i = 0; i < output.size(); ++i) {
        if (output[i].GetFd() == listen_sock.GetFd()) {
          // 如果就绪的文件描述符是 listen_sock，就执行 accept，并加入到 select 中
          TcpSocket new_sock;
          listen_sock.Accept(&new_sock, NULL, NULL);
          selector.Add(new_sock);
        } else {
          // 如果就绪的文件描述符是 new_sock，就进行一次请求的处理
          std::string req, resp;
          bool ret = output[i].Recv(&req);
          if (!ret) {
            selector.Del(output[i]);
            // [注意!] 需要关闭 socket
            output[i].Close();
            continue;
          }
          // 调用业务函数计算响应
          handler(req, &resp);
          // 将结果写回到客户端
          output[i].Send(resp);
        }
      }  // end for
    }  // end for (;;)
    return true;
  }

private:
  std::string ip_;
  uint16_t port_;
};
```

dict_server.cc

这个代码和之前相同, 只是把里面的 server 对象改成 TcpSelectServer 类即可.

客户端和之前的客户端完全相同, 无需单独开发.

I/O多路转接之poll [选学]

poll函数接口

比特就业课

```
#include <poll.h>

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

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

参数说明

- fds是一个poll函数监听的结构列表. 每一个元素中, 包含了三部分内容: 文件描述符, 监听的事件集合, 返回的事件集合.
- nfds表示fds数组的长度.
- timeout表示poll函数的超时时间, 单位是毫秒(ms).

events和revents的取值:

事　　件	描　　述	是否可作为输入	是否可作为输出
POLLIN	数据（包括普通数据和优先数据）可读	是	是
POLLRDNORM	普通数据可读	是	是
POLLRDBAND	优先级带数据可读（Linux 不支持）	是	是
POLLPRI	高优先级数据可读，比如 TCP 带外数据	是	是
POLLOUT	数据（包括普通数据和优先数据）可写	是	是
POLLWRNORM	普通数据可写	是	是
POLLWRBAND	优先级带数据可写	是	是
POLLRDHUP	TCP 连接被对方关闭，或者对方关闭了写操作，它由 GNU 引入	是	是
POLLERR	错误	否	是
POLLHUP	挂起。比如管道的写端被关闭后，读端描述符上将收到 POLLHUP 事件	否	是
POLLNVAL	文件描述符没有打开	否	是

返回结果

- 返回值小于0, 表示出错;
- 返回值等于0, 表示poll函数等待超时;
- 返回值大于0, 表示poll由于监听的文件描述符就绪而返回.

socket就绪条件

同select

poll的优点

使用一个pollfd的指针实现使用三个位图来表示三个

- 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. 接口使用比select更方便.

> - poll并没有最大数量限制 (但是数量过大后性能也是会下降).

poll的缺点

poll中监听的文件描述符数目增多时

> - 和select函数一样，poll返回后，需要轮询pollfd来获取就绪的描述符.
> - 每次调用poll都需要把大量的pollfd结构从用户态拷贝到内核中.
> - 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.

poll示例: 使用poll监控标准输入

```
#include <poll.h>
#include <unistd.h>
#include <stdio.h>

int main() {
  struct pollfd poll_fd;
  poll_fd.fd = 0;
  poll_fd.events = POLLIN;

  for (;;) {
    int ret = poll(&poll_fd, 1, 1000);
    if (ret < 0) {
      perror("poll");
      continue;
    }
    if (ret == 0) {
      printf("poll timeout\n");
      continue;
    }
    if (poll_fd.revents == POLLIN) {
      char buf[1024] = {0};
      read(0, buf, sizeof(buf) - 1);
      printf("stdin:%s", buf);
    }
  }
}
```

I/O多路转接之epoll

epoll初识

按照man手册的说法: **是为处理大批量句柄而作了改进的poll**.
它是在2.5.44内核中被引进的(epoll(4) is a new API introduced in Linux kernel 2.5.44)
它几乎具备了之前所说的一切优点，被公认为性能最好的多路I/O就绪通知方法

epoll的相关系统调用

epoll 有3个相关的系统调用.

epoll_create

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

创建一个epoll的句柄.

> - 自从linux2.6.8之后，size参数是被忽略的.
> - 用完之后, 必须调用close()关闭.

epoll_ctl

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

epoll的事件注册函数.

> - 它不同于select()是在监听事件时告诉内核要监听什么类型的事件, 而是在这里先注册要监听的事件类型.
> - 第一个参数是epoll_create()的返回值(epoll的句柄).
> - 第二个参数表示动作，用三个宏来表示.
> - 第三个参数是需要监听的fd.
> - 第四个参数是告诉内核需要监听什么事.

第二个参数的取值:

> - `EPOLL_CTL_ADD`：注册新的fd到epfd中；
> - `EPOLL_CTL_MOD`：修改已经注册的fd的监听事件；
> - `EPOLL_CTL_DEL`：从epfd中删除一个fd；

struct epoll_event结构如下:

比特就业课

```
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 */
} __EPOLL_PACKED;
```

events可以是以下几个宏的集合：

- EPOLLIN ：表示对应的文件描述符可以读 (包括对端SOCKET正常关闭);
- EPOLLOUT：表示对应的文件描述符可以写;
- EPOLLPRI：表示对应的文件描述符有紧急的数据可读 (这里应该表示有带外数据到来);
- EPOLLERR：表示对应的文件描述符发生错误;
- EPOLLHUP：表示对应的文件描述符被挂断;
- EPOLLET ：将EPOLL设为边缘触发(Edge Triggered)模式, 这是相对于水平触发(Level Triggered)来说的.
- EPOLLONESHOT：只监听一次事件, 当监听完这次事件之后, 如果还需要继续监听这个socket的话, 需要再次把这个socket加入到EPOLL队列里.

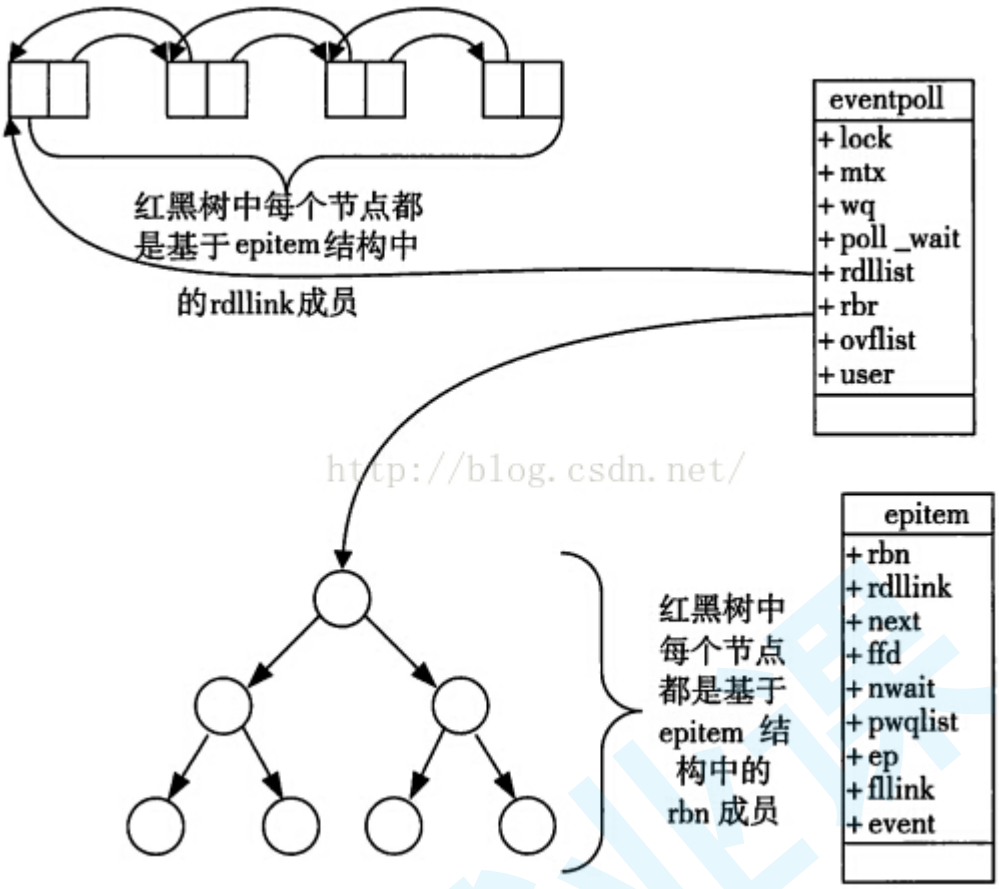
epoll_wait

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

收集在epoll监控的事件中已经发送的事件.

- 参数events是分配好的epoll_event结构体数组.
- epoll将会把发生的事件赋值到events数组中 (events不可以是空指针，内核只负责把数据复制到这个events数组中，不会去帮助我们在用户态中分配内存).
- maxevents告之内核这个events有多大，这个 maxevents的值不能大于创建epoll_create()时的size.
- 参数timeout是超时时间 (毫秒，0会立即返回，-1是永久阻塞).
- 如果函数调用成功，返回对应I/O上已准备好的文件描述符数目，如返回0表示已超时, 返回小于0表示函数失败.

epoll工作原理



- 当某一进程调用epoll_create方法时，Linux内核会创建一个eventpoll结构体，这个结构体中有两个成员与epoll的使用方式密切相关.

```
struct eventpoll{
    ....
    /*红黑树的根节点，这颗树中存储着所有添加到epoll中的需要监控的事件*/
    struct rb_root  rbr;
    /*双链表中则存放着将要通过epoll_wait返回给用户的满足条件的事件*/
    struct list_head rdlist;
    ....
};
```

- 每一个epoll对象都有一个独立的eventpoll结构体，用于存放通过epoll_ctl方法向epoll对象中添加进来的事件.
- 这些事件都会挂载在红黑树中，如此，重复添加的事件就可以通过红黑树而高效的识别出来(红黑树的插入时间效率是lgn，其中n为树的高度).
- 而所有添加到epoll中的事件都会与设备(网卡)驱动程序建立回调关系，也就是说，当响应的事件发生时会调用这个回调方法.
- 这个回调方法在内核中叫ep_poll_callback,它会将发生的事件添加到rdlist双链表中.
- 在epoll中，对于每一个事件，都会建立一个

m结构体.

```
struct epitem{
    struct rb_node  rbn;//红黑树节点
    struct list_head    rdllink;//双向链表节点
    struct epoll_filefd  ffd;  //事件句柄信息
    struct eventpoll *ep;    //指向其所属的eventpoll对象
    struct epoll_event event; //期待发生的事件类型
}
```

> - 当调用epoll_wait检查是否有事件发生时，只需要检查eventpoll对象中的rdlist双链表中是否有epitem元素即可.
> - 如果rdlist不为空，则把发生的事件复制到用户态，同时将事件数量返回给用户. 这个操作的时间复杂度是O(1).

总结一下, epoll的使用过程就是三部曲:

> - 调用epoll_create创建一个epoll句柄;
> - 调用epoll_ctl, 将要监控的文件描述符进行注册;
> - 调用epoll_wait, 等待文件描述符就绪;

epoll的优点(和 select 的缺点对应)

> - 接口使用方便: 虽然拆分成了三个函数, 但是反而使用起来更方便高效. 不需要每次循环都设置关注的文件描述符, 也做到了输入输出参数分离开
> - 数据拷贝轻量: 只在合适的时候调用 `EPOLL_CTL_ADD` 将文件描述符结构拷贝到内核中, 这个操作并不频繁(而select/poll都是每次循环都要进行拷贝)
> - 事件回调机制: 避免使用遍历, 而是使用回调函数的方式, 将就绪的文件描述符结构加入到就绪队列中, epoll_wait 返回直接访问就绪队列就知道哪些文件描述符就绪. 这个操作时间复杂度O(1). 即使文件描述符数目很多, 效率也不会受到影响.
> - 没有数量限制: 文件描述符数目无上限.

注意!!

网上有些博客说, epoll中使用了内存映射机制

> - 内存映射机制: 内核直接将就绪队列通过mmap的方式映射到用户态. 避免了拷贝内存这样的额外性能开销.

这种说法是不准确的. 我们定义的struct epoll_event是我们在用户空间中分配好的内存. 势必还是需要将内核的数据拷贝到这个用户空间的内存中的.

请同学们对比总结select, poll, epoll之间的优点和缺点(重要, 面试中常见).

epoll工作方式

你妈喊你吃饭的例子

```
你正在吃鸡，眼看进入了决赛圈，你妈饭做好了，喊你吃饭的时候有两种方式：
1. 如果你妈喊你一次，你没动，那么你妈会继续喊你第二次，第三次...(亲妈，水平触发)
2. 如果你妈喊你一次，你没动，你妈就不管你了(后妈，边缘触发)
```

epoll有2种工作方式-水平触发(LT)和边缘触发(ET)

假如有这样一个例子:

> - 我们已经把一个tcp socket添加到epoll描述符
> - 这个时候socket的另一端被写入了2KB的数据
> - 调用epoll_wait，并且它会返回. 说明它已经准备好读取操作
> - 然后调用read, 只读取了1KB的数据
> - 继续调用epoll_wait.....

水平触发Level Triggered 工作模式

epoll默认状态下就是LT工作模式.

> - 当epoll检测到socket上事件就绪的时候, 可以不立刻进行处理. 或者只处理一部分.
> - 如上面的例子, 由于只读了1K数据, 缓冲区中还剩1K数据, 在第二次调用 `epoll_wait` 时, `epoll_wait` 仍然会立刻返回并通知socket读事件就绪.
> - 直到缓冲区上所有的数据都被处理完, `epoll_wait` 才不会立刻返回.
> - 支持阻塞读写和非阻塞读写

边缘触发Edge Triggered工作模式

如果我们在第1步将socket添加到epoll描述符的时候使用了EPOLLET标志, epoll进入ET工作模式.

> - 当epoll检测到socket上事件就绪时, 必须立刻处理.
> - 如上面的例子, 虽然只读了1K的数据, 缓冲区还剩1K的数据, 在第二次调用 `epoll_wait` 的时候, `epoll_wait` 不会再返回了.
> - 也就是说, ET模式下, 文件描述符上的事件就绪后, 只有一次处理机会.
> - ET的性能比LT性能更高(`epoll_wait` 返回的次数少了很多). Nginx默认采用ET模式使用epoll.
> - 只支持非阻塞的读写

select和poll其实也是工作在LT模式下. epoll既可以支持LT, 也可以支持ET.

对比LT和ET

LT是 epoll 的默认行为. 使用 ET 能够减少 epoll 触发的次数. 但是代价就是强逼着程序猿一次响应就绪过程中就把所有的数据都处理完.

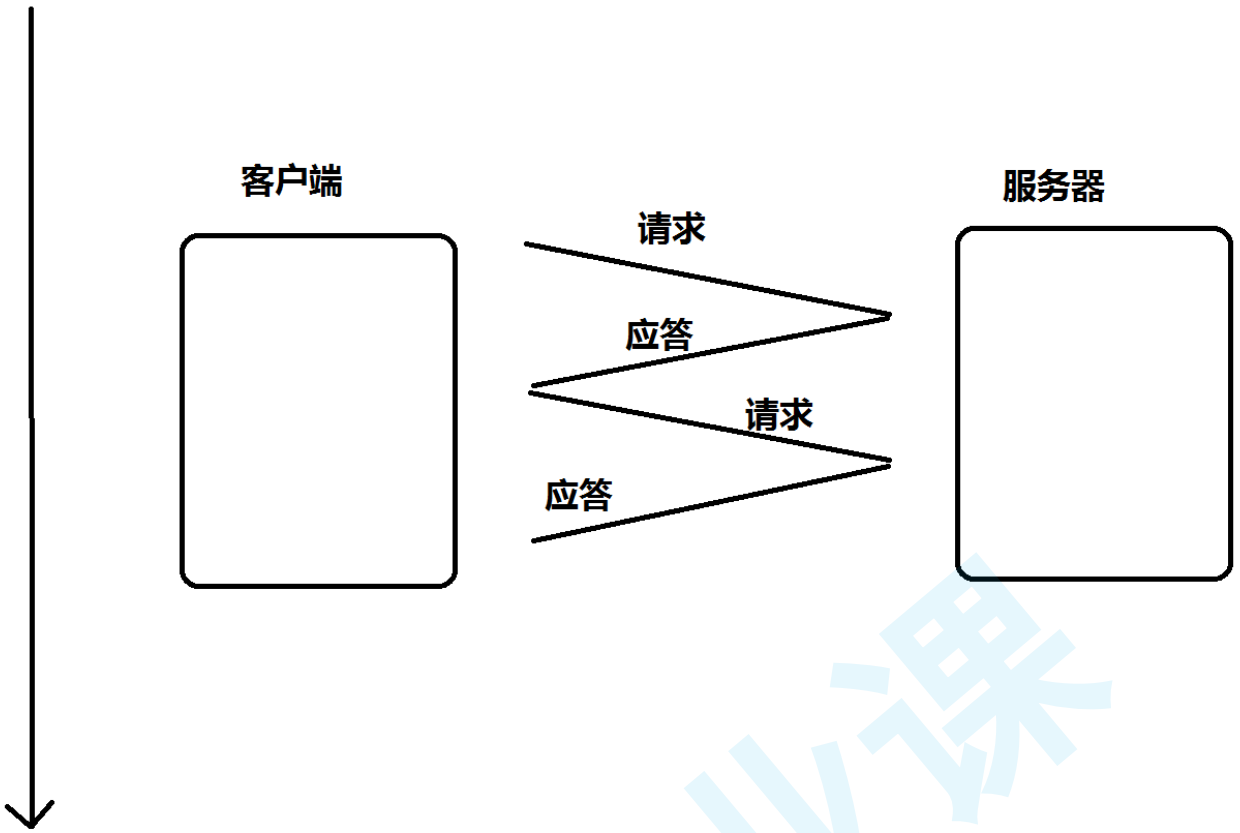
相当于一个文件描述符就绪之后, 不会反复被提示就绪, 看起来就比 LT 更高效一些. 但是在 LT 情况下如果也能做到每次就绪的文件描述符都立刻处理, 不让这个就绪被重复提示的话, 其实性能也是一样的.

另一方面, ET 的代码复杂程度更高了.

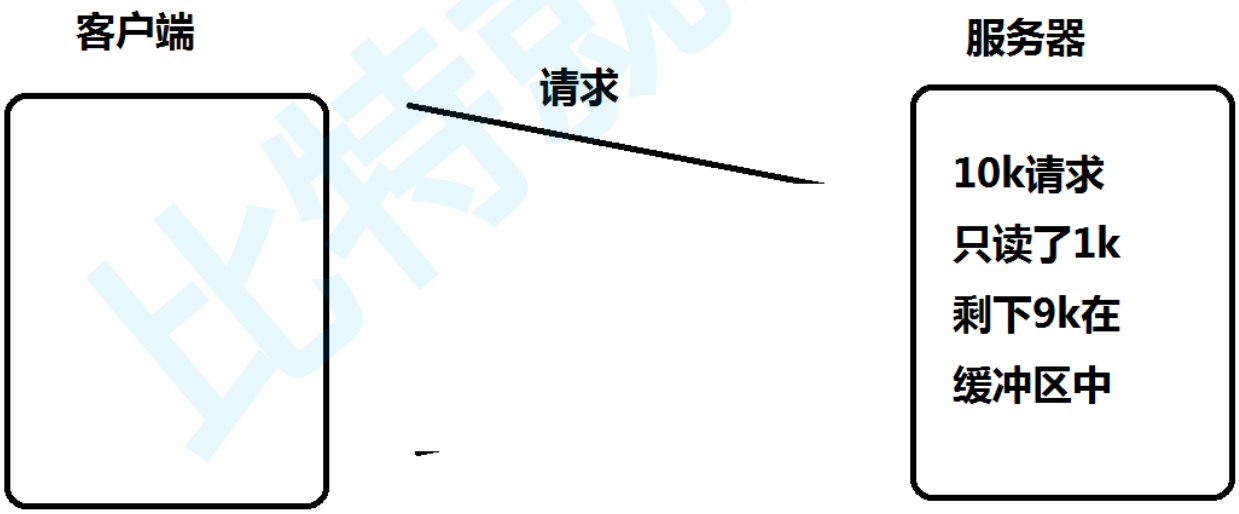
理解ET模式和非阻塞文件描述符

使用 ET 模式的 epoll, 需要将文件描述设置为非阻塞. 这个不是接口上的要求, 而是 "工程实践" 上的要求.

假设这样的场景: 服务器接受到一个10k的请求, 会向客户端返回一个应答数据. 如果客户端收不到应答, 不会发送第二个10k请求.



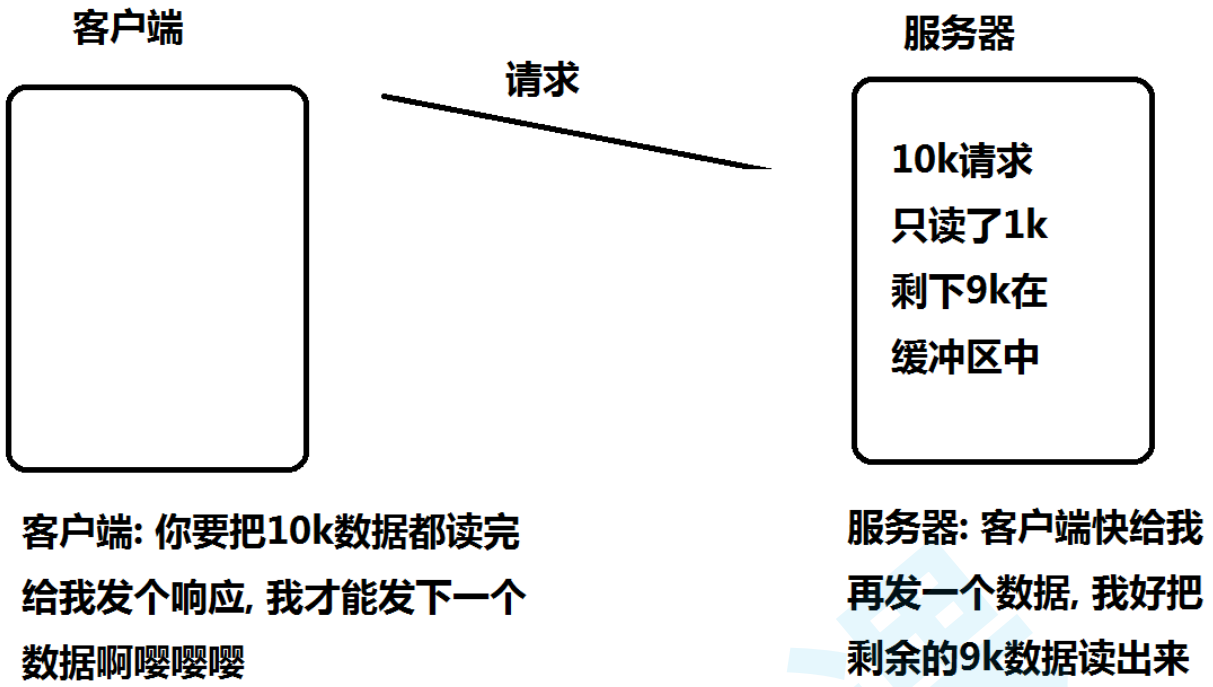
如果服务端写的代码是阻塞式的read, 并且一次只 read 1k 数据的话(read不能保证一次就把所有的数据都读出来, 参考 man 手册的说明, 可能被信号打断), 剩下的9k数据就会待在缓冲区中.



此时由于 epoll 是ET模式, 并不会认为文件描述符读就绪. `epoll_wait` 就不会再次返回. 剩下的 9k 数据会一直在缓冲区中. 直到下一次客户端再给服务器写数据. `epoll_wait` 才能返回

但是问题来了.

> - 服务器只读到1k个数据, 要10k读完才会给客户端返回响应数据.
> - 客户端要读到服务器的响应
> - 客户端发送了下一个请求, `epoll_wait` 才会返回, 才能去读缓冲区中剩余的数据.



所以, 为了解决上述问题(阻塞read不一定能一下把完整的请求读完), 于是就可以使用非阻塞轮训的方式来读缓冲区, 保证一定能把完整的请求都读出来.

而如果是LT没这个问题. 只要缓冲区中的数据没读完, 就能够让 `epoll_wait` 返回文件描述符读就绪.

epoll的使用场景

epoll的高性能, 是有一定的特定场景的. 如果场景选择的不适宜, epoll的性能可能适得其反.

> - 对于多连接, 且多连接中只有一部分连接比较活跃时, 比较适合使用epoll.

例如, 典型的一个需要处理上万个客户端的服务器, 例如各种互联网APP的入口服务器, 这样的服务器就很适合epoll.

如果只是系统内部, 服务器和服务器之间进行通信, 只有少数的几个连接, 这种情况下用epoll就并不合适. 具体要根据需求和场景特点来决定使用哪种IO模型.

epoll中的惊群问题(选学)

惊群问题有些面试官可能会问到. 建议同学们课后自己查阅资料了解一下问题的解决方案.

参考 http://blog.csdn.net/fsmiy/article/details/36873357

epoll示例: epoll服务器(LT模式)

tcp_epoll_server.hpp

```
//////////////////////////////////////////////////////
// 封装一个 Epoll 服务器, 只考虑读就绪的情况
//////////////////////////////////////////////////////

#pragma once
#include <vector>

#include <functional>
```

```cpp
#include <sys/epoll.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string&, std::string* resp)> Handler;

class Epoll {
public:
  Epoll() {
    epoll_fd_ = epoll_create(10);
  }

  ~Epoll() {
    close(epoll_fd_);
  }

  bool Add(const TcpSocket& sock) const {
    int fd = sock.GetFd();
    printf("[Epoll Add] fd = %d\n", fd);
    epoll_event ev;
    ev.data.fd = fd;
    ev.events = EPOLLIN;
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, fd, &ev);
    if (ret < 0) {
      perror("epoll_ctl ADD");
      return false;
    }
    return true;
  }

  bool Del(const TcpSocket& sock) const {
    int fd = sock.GetFd();
    printf("[Epoll Del] fd = %d\n", fd);
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_DEL, fd, NULL);
    if (ret < 0) {
      perror("epoll_ctl DEL");
      return false;
    }
    return true;
  }

  bool Wait(std::vector<TcpSocket>* output) const {
    output->clear();
    epoll_event events[1000];
    int nfds = epoll_wait(epoll_fd_, events, sizeof(events) / sizeof(events[0]), -1);
    if (nfds < 0) {
      perror("epoll_wait");
      return false;
    }
    // [注意!] 此处必须是循环到 nfds, 不能多循环
    for (int i = 0; i < nfds; ++i) {
      TcpSocket sock(events[i].data.fd);
      output->push_back(sock

    }
```

北京就业课

```
    return true;
  }

private:
  int epoll_fd_;
};

class TcpEpollServer {
public:
  TcpEpollServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {

  }

  bool Start(Handler handler) {
    // 1. 创建 socket
    TcpSocket listen_sock;
    CHECK_RET(listen_sock.Socket());
    // 2. 绑定
    CHECK_RET(listen_sock.Bind(ip_, port_));
    // 3. 监听
    CHECK_RET(listen_sock.Listen(5));
    // 4. 创建 Epoll 对象, 并将 listen_sock 加入进去
    Epoll epoll;
    epoll.Add(listen_sock);
    // 5. 进入事件循环
    for (;;) {
      // 6. 进行 epoll_wait
      std::vector<TcpSocket> output;
      if (!epoll.Wait(&output)) {
        continue;
      }
      // 7. 根据就绪的文件描述符的种类决定如何处理
      for (size_t i = 0; i < output.size(); ++i) {
        if (output[i].GetFd() == listen_sock.GetFd()) {
          // 如果是 listen_sock, 就调用 accept
          TcpSocket new_sock;
          listen_sock.Accept(&new_sock);
          epoll.Add(new_sock);
        } else {
          // 如果是 new_sock, 就进行一次读写
          std::string req, resp;
          bool ret = output[i].Recv(&req);
          if (!ret) {
            // [注意!!] 需要把不用的 socket 关闭
            // 先后顺序别搞反. 不过在 epoll 删除的时候其实就已经关闭 socket 了
            epoll.Del(output[i]);
            output[i].Close();
            continue;
          }
          handler(req, &resp);
          output[i].Send(resp);
        }  // end for

      }  // end for (;;)
```

```
    }
    return true;
  }

private:
  std::string ip_;
  uint16_t port_;
};
```

dict_server.cc 只需要将 server 对象的类型改成 TcpEpollServer 即可.

epoll示例: epoll服务器(ET模式)

基于 LT 版本稍加修改即可

1. 修改 tcp_socket.hpp, 新增非阻塞读和非阻塞写接口
2. 对于 accept 返回的 new_sock 加上 EPOLLET 这样的选项

注意: 此代码暂时未考虑 listen_sock ET 的情况. 如果将 listen_sock 设为 ET, 则需要非阻塞轮询的方式 accept. 否则会导致同一时刻大量的客户端同时连接的时候, 只能 accept 一次的问题.

tcp_socket.hpp

```
// 以下代码添加在 TcpSocket 类中
// 非阻塞 IO 接口
bool SetNoBlock() {
  int fl = fcntl(fd_, F_GETFL);
  if (fl < 0) {
    perror("fcntl F_GETFL");
    return false;
  }
  int ret = fcntl(fd_, F_SETFL, fl | O_NONBLOCK);
  if (ret < 0) {
    perror("fcntl F_SETFL");
    return false;
  }
  return true;
}

bool RecvNoBlock(std::string* buf) const {
  // 对于非阻塞 IO 读数据，如果 TCP 接受缓冲区为空，就会返回错误
  // 错误码为 EAGAIN 或者 EWOULDBLOCK，这种情况也是意料之中，需要重试
  // 如果当前读到的数据长度小于尝试读的缓冲区的长度，就退出循环
  // 这种写法其实不算特别严谨(没有考虑粘包问题)
  buf->clear();
  char tmp[1024 * 10] = {0};
  for (;;) {
    ssize_t read_size = recv(fd_, tmp, sizeof(tmp) - 1, 0);
    if (read_size < 0) {
      if (errno == EWOULDBLOCK || errno == EAGAIN) {
        continue;
      }

      perror("recv");
```

```
        return false;
      }
      if (read_size == 0) {
        // 对端关闭, 返回 false
        return false;
      }
      tmp[read_size] = '\0';
      *buf += tmp;
      if (read_size < (ssize_t)sizeof(tmp) - 1) {
        break;
      }
    }
    return true;
  }

  bool SendNoBlock(const std::string& buf) const {
    // 对于非阻塞 IO 的写入, 如果 TCP 的发送缓冲区已经满了, 就会出现出错的情况
    // 此时的错误号是 EAGAIN 或者 EWOULDBLOCK. 这种情况下不应放弃治疗
    // 而要进行重试
    ssize_t cur_pos = 0;  // 记录当前写到的位置
    ssize_t left_size = buf.size();
    for (;;) {
      ssize_t write_size = send(fd_, buf.data() + cur_pos, left_size, 0);
      if (write_size < 0) {
        if (errno == EAGAIN || errno == EWOULDBLOCK) {
          // 重试写入
          continue;
        }
        return false;
      }
      cur_pos += write_size;
      left_size -= write_size;
      // 这个条件说明写完需要的数据了
      if (left_size <= 0) {
        break;
      }
    }
    return true;
  }
```

tcp_epoll_server.hpp

```
//////////////////////////////////////////////////////
// 封装一个 Epoll ET 服务器
// 修改点:
// 1. 对于 new sock, 加上 EPOLLET 标记
// 2. 修改 TcpSocket 支持非阻塞读写
// [注意!] listen_sock 如果设置成 ET, 就需要非阻塞调用 accept 了
// 稍微麻烦一点, 此处暂时不实现
//////////////////////////////////////////////////////


#pragma once
```

```cpp
#include <vector>
#include <functional>
#include <sys/epoll.h>
#include "tcp_socket.hpp"

typedef std::function<void (const std::string&, std::string* resp)> Handler;

class Epoll {
public:
  Epoll() {
    epoll_fd_ = epoll_create(10);
  }

  ~Epoll() {
    close(epoll_fd_);
  }

  bool Add(const TcpSocket& sock, bool epoll_et = false) const {
    int fd = sock.GetFd();
    printf("[Epoll Add] fd = %d\n", fd);
    epoll_event ev;
    ev.data.fd = fd;
    if (epoll_et) {
      ev.events = EPOLLIN | EPOLLET;
    } else {
      ev.events = EPOLLIN;
    }
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_ADD, fd, &ev);
    if (ret < 0) {
      perror("epoll_ctl ADD");
      return false;
    }
    return true;
  }

  bool Del(const TcpSocket& sock) const {
    int fd = sock.GetFd();
    printf("[Epoll Del] fd = %d\n", fd);
    int ret = epoll_ctl(epoll_fd_, EPOLL_CTL_DEL, fd, NULL);
    if (ret < 0) {
      perror("epoll_ctl DEL");
      return false;
    }
    return true;
  }

  bool Wait(std::vector<TcpSocket>* output) const {
    output->clear();
    epoll_event events[1000];
    int nfds = epoll_wait(epoll_fd_, events, sizeof(events) / sizeof(events[0]), -1);
    if (nfds < 0) {
      perror("epoll_wait");

      return false;
```

北特就业课

```
    }
    // [注意!] 此处必须是循环到 nfds, 不能多循环
    for (int i = 0; i < nfds; ++i) {
      TcpSocket sock(events[i].data.fd);
      output->push_back(sock);
    }
    return true;
  }

private:
  int epoll_fd_;
};

class TcpEpollServer {
public:
  TcpEpollServer(const std::string& ip, uint16_t port) : ip_(ip), port_(port) {

  }

  bool Start(Handler handler) {
    // 1. 创建 socket
    TcpSocket listen_sock;
    CHECK_RET(listen_sock.Socket());
    // 2. 绑定
    CHECK_RET(listen_sock.Bind(ip_, port_));
    // 3. 监听
    CHECK_RET(listen_sock.Listen(5));
    // 4. 创建 Epoll 对象, 并将 listen_sock 加入进去
    Epoll epoll;
    epoll.Add(listen_sock);
    // 5. 进入事件循环
    for (;;) {
      // 6. 进行 epoll_wait
      std::vector<TcpSocket> output;
      if (!epoll.Wait(&output)) {
        continue;
      }
      // 7. 根据就绪的文件描述符的种类决定如何处理
      for (size_t i = 0; i < output.size(); ++i) {
        if (output[i].GetFd() == listen_sock.GetFd()) {
          // 如果是 listen_sock, 就调用 accept
          TcpSocket new_sock;
          listen_sock.Accept(&new_sock);
          epoll.Add(new_sock, true);
        } else {
          // 如果是 new_sock, 就进行一次读写
          std::string req, resp;
          bool ret = output[i].RecvNoBlock(&req);
          if (!ret) {
            // [注意!!] 需要把不用的 socket 关闭
            // 先后顺序别搞反. 不过在 epoll 删除的时候其实就已经关闭 socket 了
            epoll.Del(output

            output[i].Close();
```

```
            continue;
          }
          handler(req, &resp);
          output[i].SendNoBlock(resp);
          printf("[client %d] req: %s, resp: %s\n", output[i].GetFd(),
              req.c_str(), resp.c_str());
        }  // end for
      }  // end for (;;)
    }
    return true;
  }

private:
  std::string ip_;
  uint16_t port_;
};
```

参考资料

epoll详解

apache/nginx网络模型

比特就业课