12 多路转接 select

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:

```
C
#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.

```
C
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：

```
C
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于 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:

```c
#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

```
C
#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;

// 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.