0328_epoll

1. 快速的理解部分概念，快速的看一下epoll相关的接口
2. 讲解epoll的工作原理
3. epoll服务器 -- 封装
4. 工作模式
5. 如何基于epoll设计一个完整的服务器 --- Reactor模式

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

       int epoll_create(int size);
       int epoll_create1(int flags);
```

这个size目前是废弃的
随便写
返回的是一个文件描述符
我们叫做epoll模式

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

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

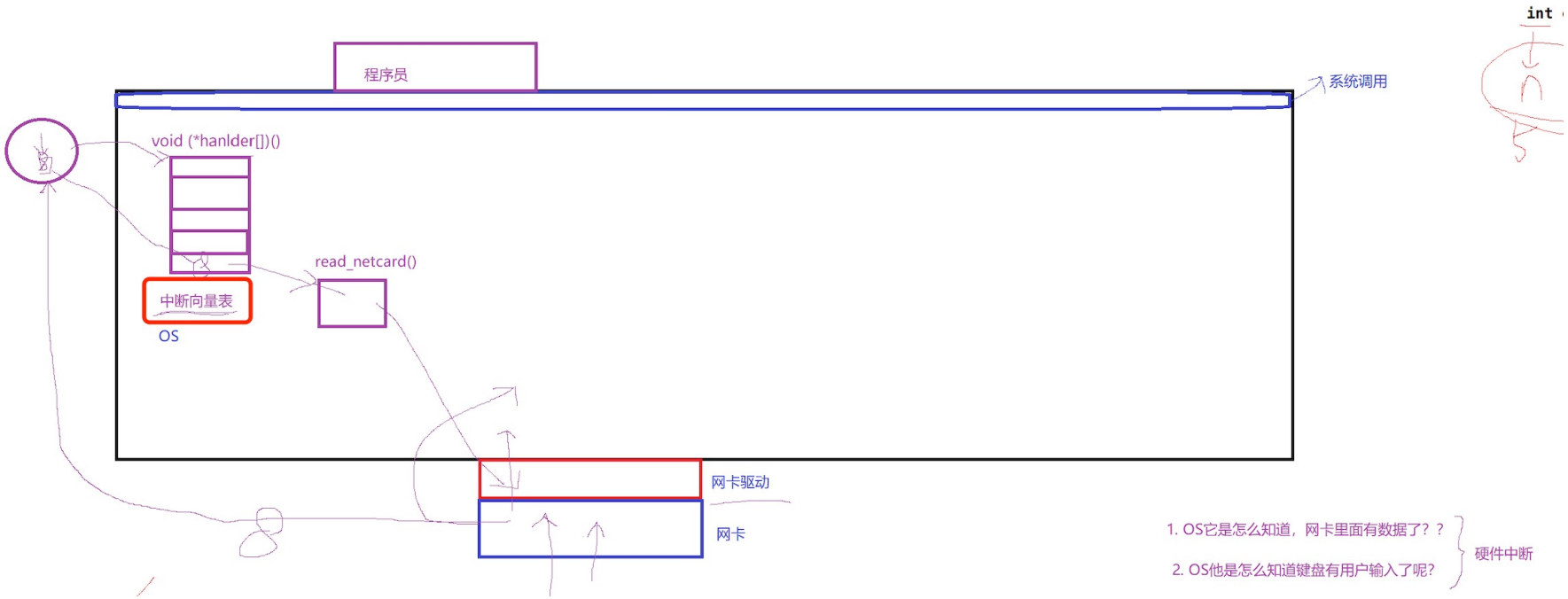
```
SYNOPSIS
       #include <sys/epoll.h>

       int epoll_wait(int epfd, struct epoll_event *events,
                      int maxevents, int timeout);
       int epoll_pwait(int epfd, struct epoll_event *events,
                      int maxevents, int timeout,
                      const sigset_t *sigmask);
```

1. 无论是select还是poll，都需要自己维护一个数组，来进行保存fd与特定事件的
2. select or poll 都要遍历
3. 两个的工作模式，都是通过这些系统调用，告诉内核，你要帮我关心，哪些fd上的哪些event

开始讲EPOLL的原理

先提一些问题：

1. OS他是怎么知道，网卡里面有数据了？ ⇨ 中断的方式
2. OS他是怎么知道键盘有用户输入了？



首先，创建一个epoll模式之后
在操作系统里面先维护一棵红黑树！！
这个红黑树，在内核里面，用户不关心。

红黑树节点里面放什么，最核心的字段有两个：
1. int fd;
2. Short event;

然后，创建epoll之后，OS还会帮我们维护一个就绪队列！！
他的每一个节点，核心的字段：
int fd;
short revents;

一开始这个队列是空的，如果红黑树上有就绪的节点，OS就会帮我们构建一个节点，放到这个就绪队列里面去！

所以这个队列解决的是，内核到用户的问题

此时要注意，这个红黑树，是完成了我们之前说的用户到内核的这部分内容
就是用户告诉内核，你需要帮我关心哪些fd，我就帮你用红黑树存好
这个红黑树，其实就相当于select，poll的数组
只不过是，这个红黑树我们不用自己维护

创建epoll之后，第三件事是：

回向底层注册一个回调方法

```
void callback()
{
    //1. 根据红黑树上节点要关心的时间，结合已经发生的时间
    //2. 自动根据fd和已经发生的时间，构建就绪节点
    //3. 自动将构建好的节点，插入到就绪队列中
}
```

以上的一整套东西，叫做epoll模型！

此时三个接口就能理解了！

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

       int epoll_create(int size);
       int epoll_create1(int flags);
```

做三件事：

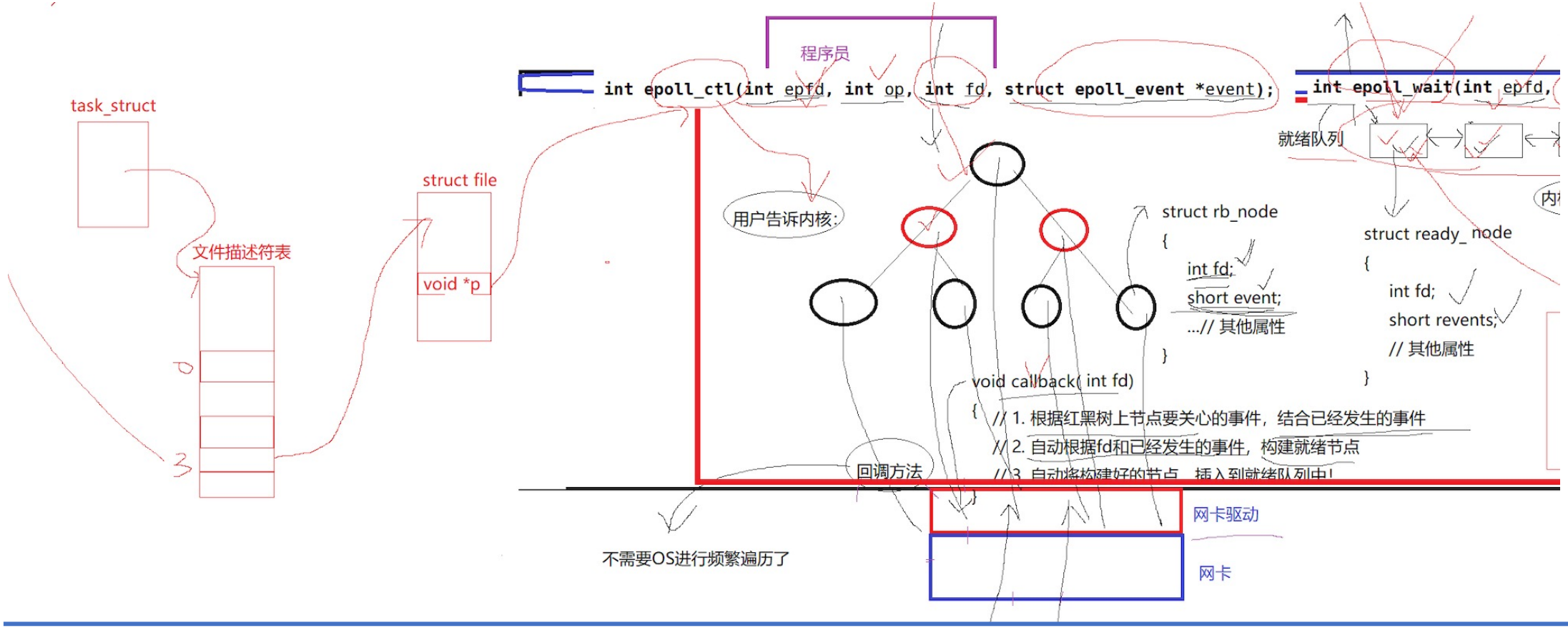
1. 创建红黑树
2. 创建队列
3. 创建回调

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

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

修改底层的红黑树

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

       int epoll_wait(int epfd, struct epoll_event *events,
                      int maxevents, int timeout);
       int epoll_pwait(int epfd, struct epoll_event *events,
                      int maxevents, int timeout,
                      const sigset_t *sigmask);
```

捞取就绪的事件



一些细节：

1. 红黑树是要key的 -> 文件描述符
2. 用户只需要设置关系，获取结果即可，不用关心fd与event的管理
3. epoll为什么高效呢？？
4. 底层只要有fd就绪了，OS就会自己给我构建节点，连接到就绪队列中去，上层只需要不断的从就绪队列中将数据拿走
5. 如果底层没有就绪事件呢？我们的上层应该怎么办？

生产者消费者模型，线程安全？

底层已经保证了这个动作是线程安全的

写代码！

```
blic:
  EpollServer(const int& port = __default_port) : __port(port)
  {
      //1. 创建listensock
      __listen_sock = Sock::Socket();
      Sock::Bind(__listen_sock, __port);
      Sock::Listen(__listen_sock);
      //2. 创建epoll模型 -> 这个也可以封装一下
      __epfd = Epoll::CreateEpoll();
      logMessage(DEBUG, "init sucess, listensock: %d, epfd: %d", __listen_sock, __epfd);
      //3. 将listensock先添加到epoll中，让epoll帮我们管理起来！ -- 封装一下
      Epoll::CtlEpoll(__epfd, EPOLL_CTL_ADD, __listen_sock, EPOLLIN);
  }
  ~EpollServer() {}
  void start()
```

```
 7  class Epoll
 8  {
 9  public:
10      static int CreateEpoll()
11      {
12          int epfd = epoll_create(__gsize);
13          if(epfd > 0) return epfd;
14          exit(5); // epoll模型创建失败，直接终止
15      }
16  private:
17      static const int __gsize = 256;
18  public:
19      static bool CtlEpoll(int epfd, int oper, int sock, uint32_t events)
20      {
21          struct epoll_event ev;
22          ev.events = events;
23          ev.data.fd = sock;
24          int n = epoll_ctl(epfd, oper, sock, &ev);
25          return n == 0;
26      }
27  };
```

```
    }
private:
    static const int __gsize = 256;
public:
    static bool CtlEpoll(int epfd, int oper, int sock, uint32_t events)
    {
        struct epoll_event ev;
        ev.events = events;
        ev.data.fd = sock;
        int n = epoll_ctl(epfd, oper, sock, &ev);
        return n == 0;
    }
    static int WaitEpoll(int epfd, struct epoll_event revs[], int num, int timeout)
    {
        return epoll_wait(epfd, revs, num, timeout);
    }
};
```

细节一：
如果就绪队列里面的节点很多！revs[]装不下怎么办？
不影响，可以等下一次再拿

细节二：
关于epollwait的返回值的问题：
有几个fd上的事件就绪，就返回几，epoll返回的时候，会将所有就绪的event按顺序放入revs数组中！**有返回值个！**