



# ICS-24

# Concurrent Programming

By 胡仕豪

# 并发编程的困难

## 1、数据竞争 (Data Race)

多个线程同时访问共享数据，且至少有一个线程对共享数据进行了写操作，从而导致程序出现未定义的行为。

## 2、死锁(Dead lock)

两个或多个线程互相等待对方释放资源，从而导致所有相关线程都无法继续执行的情况。

## 3、活锁(Live lock)

两个或多个线程互相等待对方释放资源，其中某个线程不断运行尝试解决问题但总是失败。

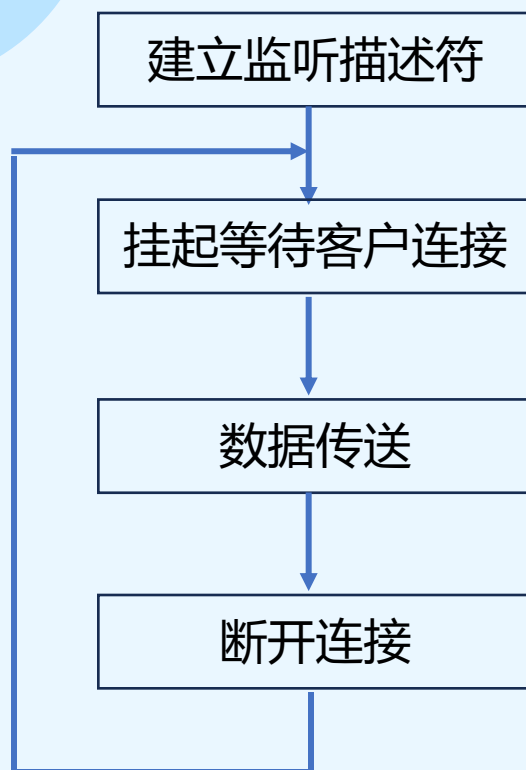
## 4、饿死(Starvation)

进程因为某种原因长时间无法获得所需资源，从而无法执行。

# 并发编程的技术-构建并发服务器

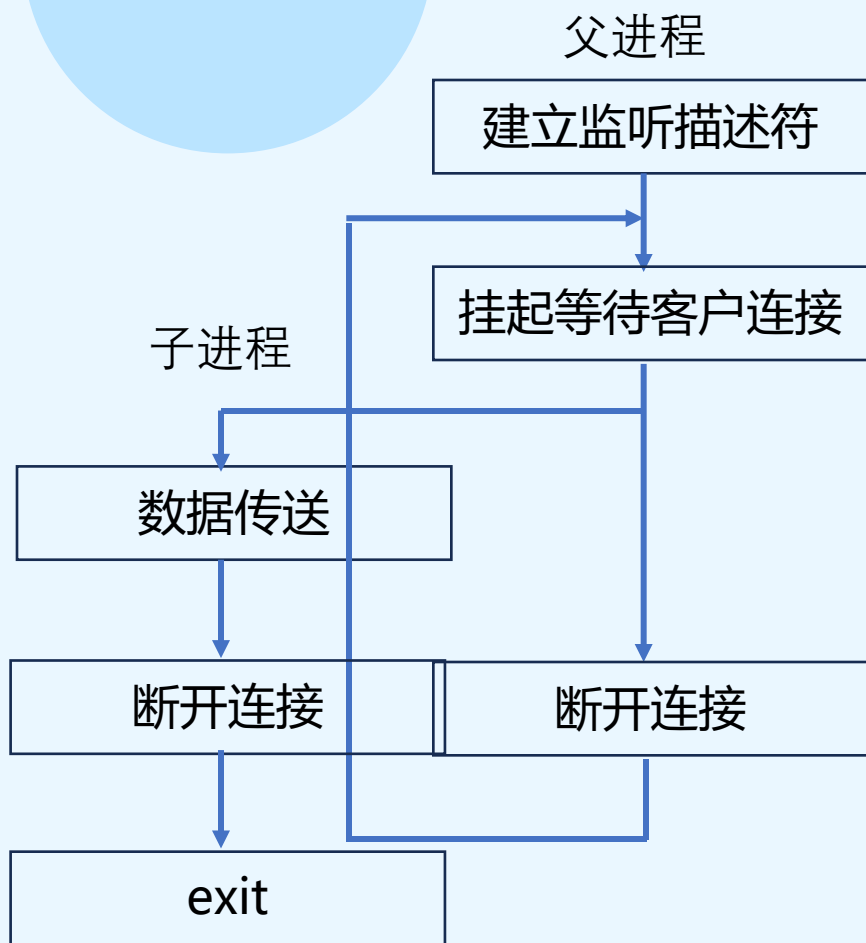
- 1、基于进程(Process-based)
- 2、基于线程(Thread-based)
- 3、基于I/O多路复用(Event-based)

# 迭代服务器



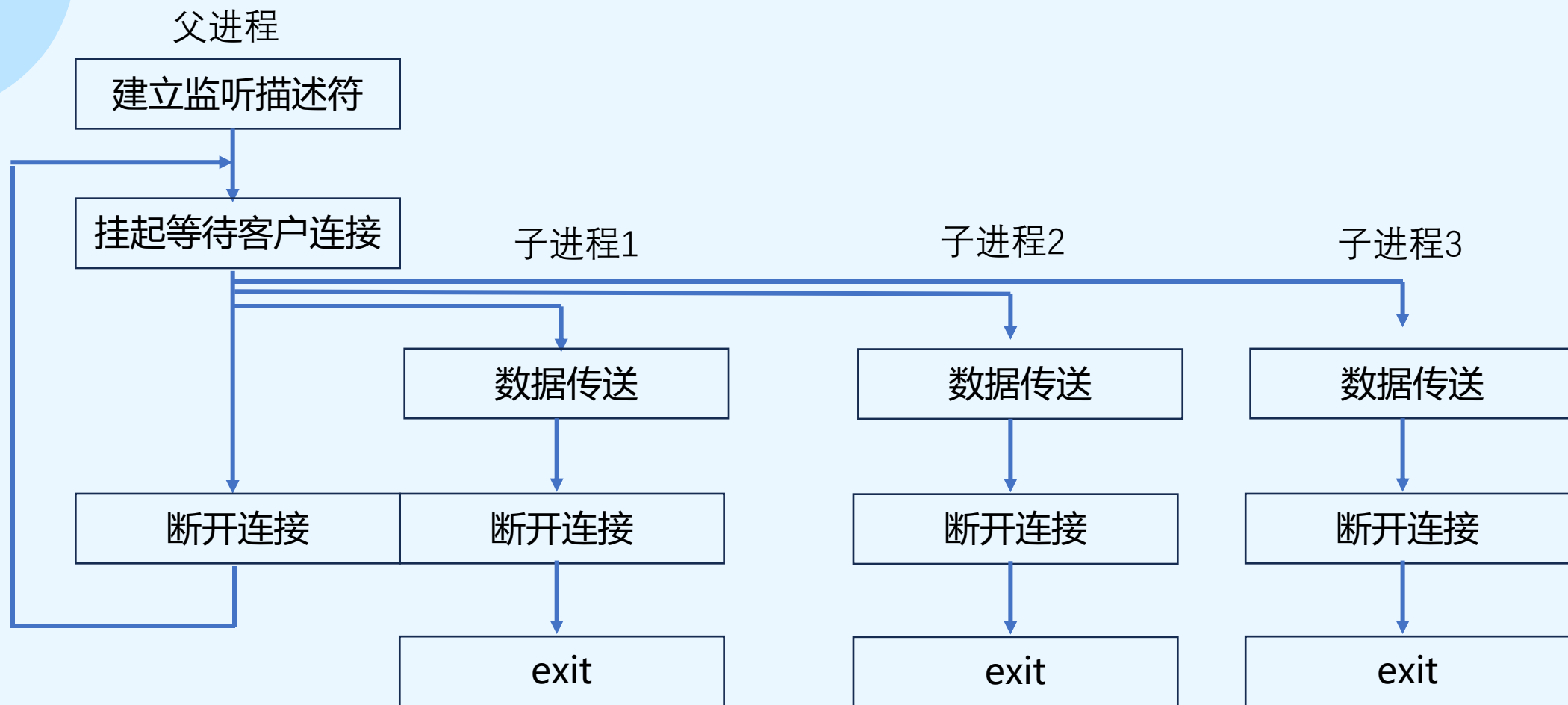
```
listenfd = Open_listenfd(argv[1]);  
while(1) {  
    clientlen = sizeof(struct sockaddr_storage);  
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);  
  
    echo(connfd);  
  
    Close(connfd);  
}
```

# 基于进程的并发服务器

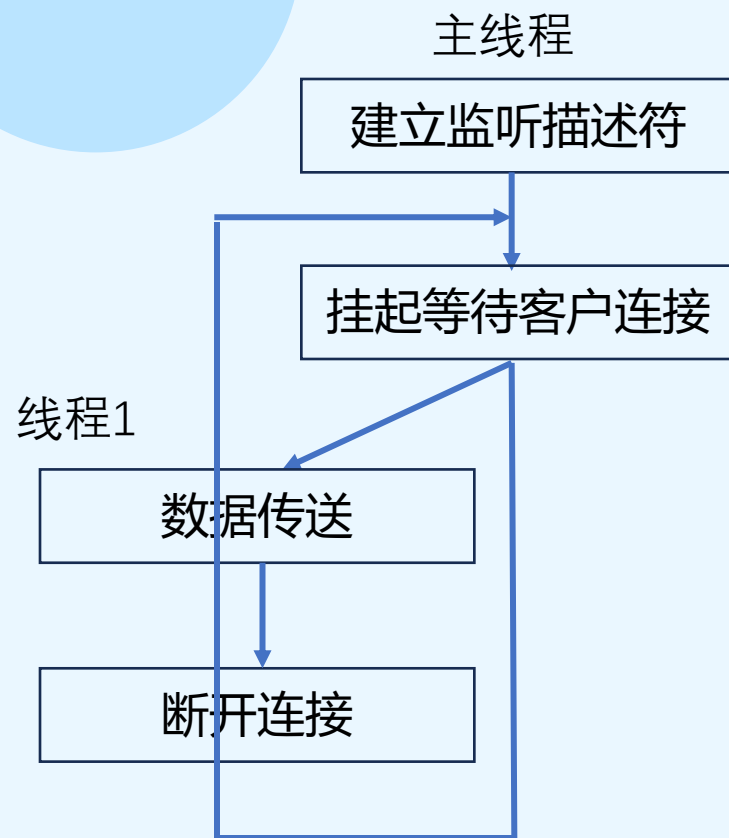


```
listenfd = Open_listenfd(argv[1]);
while(1) {
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    if (Fork() == 0){
        echo(connfd);
        Close(connfd);
        exit(0);
    }
    Close(connfd);
}
exit(0);
```

# 基于进程的并发服务器



# 基于线程的并发服务器



```
listenfd = Open_listenfd(argv[1]);

while(1) {
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

创建一个线程，让它去做事thread

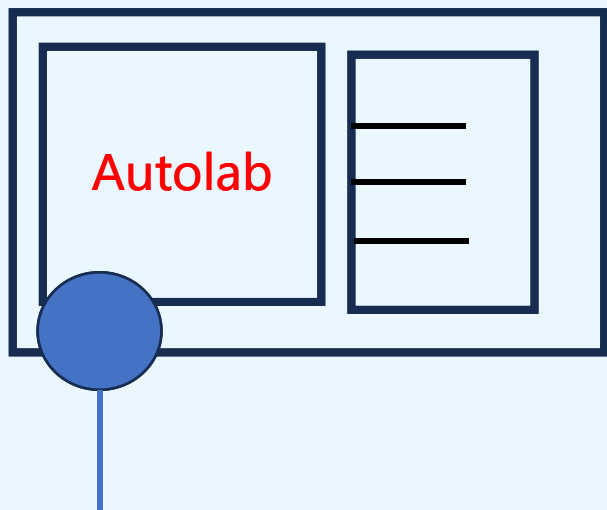
thread

```
echo(connfd);

Close(connfd);
```

回收?

# 进程

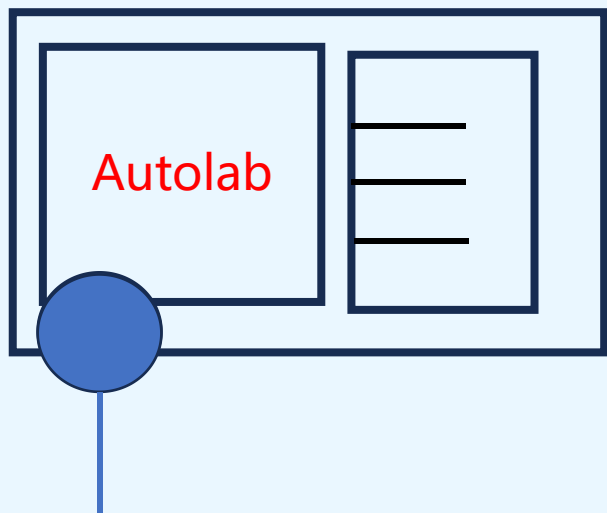


03

06



# 线程



03

06

# 进程与线程

1、每个进程包含了独立的**进程上下文**和**内存空间**

**进程上下文**包含了程序员可见的状态和内核上下文

程序员可见的状态包括寄存器、条件码cc、栈指针、pc

内核上下文包括虚拟内存结构、描述符表、堆指针

**内存空间**包含了栈、代码、数据、堆等等

2、每个线程包含了独立的线程上下文：程序员可见的状态和栈，它们具有独立的逻辑控制流。

3、每个线程具有自己的tid，我们无法区分线程，它们具有相同的权限，它们是并发的。

# 线程相关函数

## Posix Threads (Pthreads) Interface

- ***Pthreads***: Standard interface for ~60 functions that manipulate threads from C programs
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining your thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
    - `exit()` [terminates all threads]
    - `return` [terminates current thread]
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
    - `pthread_mutex_[un]lock`

# 线程相关函数

```
int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg)
```

tid: 对等线程的tid将存在这里

attr: 对等线程的属性存在这里

f: 对等线程将负责这部分工作（例程）

arg: 对等线程将带着这里的一个参数开始

成功返回0，失败返回非零

```
int pthread_join(pthread_t tid, void **thread_return)
```

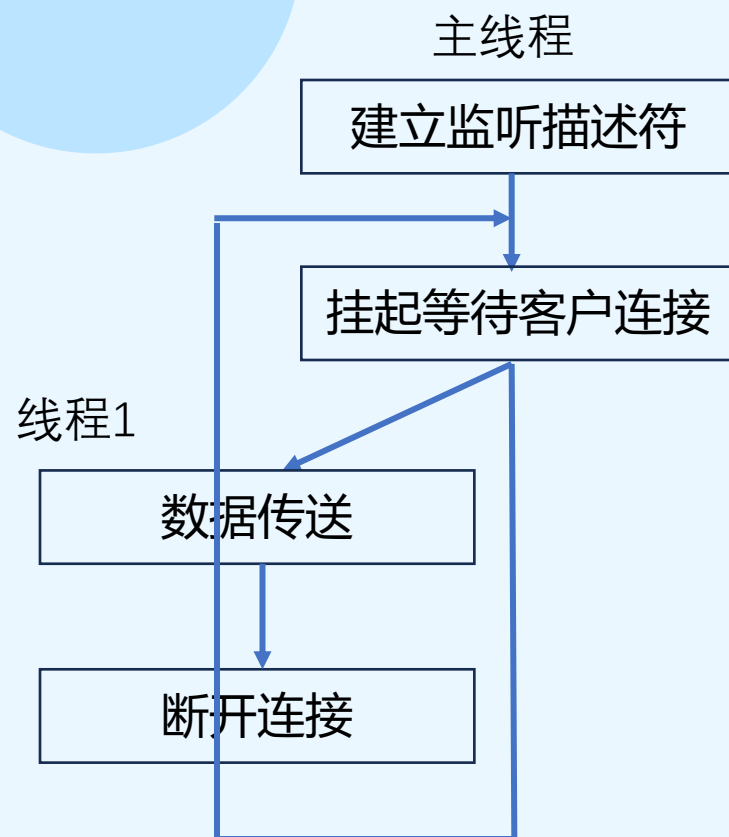
tid: 要回收的对等线程的tid

thread\_return: 对等线程的例程

成功返回0，失败返回非零

```
int pthread_detach(pthread_t tid)
```

# 基于线程的并发服务器



```
listenfd = Open_listenfd(argv[1]);
while(1) {
    clientlen = sizeof(struct sockaddr_storage);
    connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

    创建一个线程，让它去做事thread
    pthread_create(&tid, NULL, thread, *connfd)

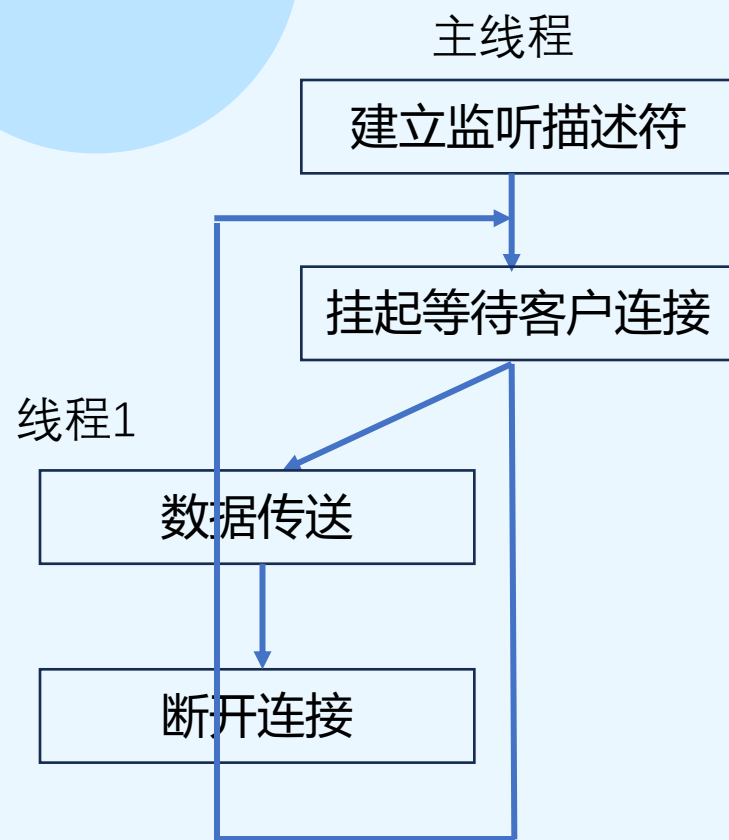
}
```

thread

```
int connfd = *connfd;
pthread_detach(pthread_self());

echo(connfd);
Close(connfd);
return NULL;
```

# 基于线程的并发服务器

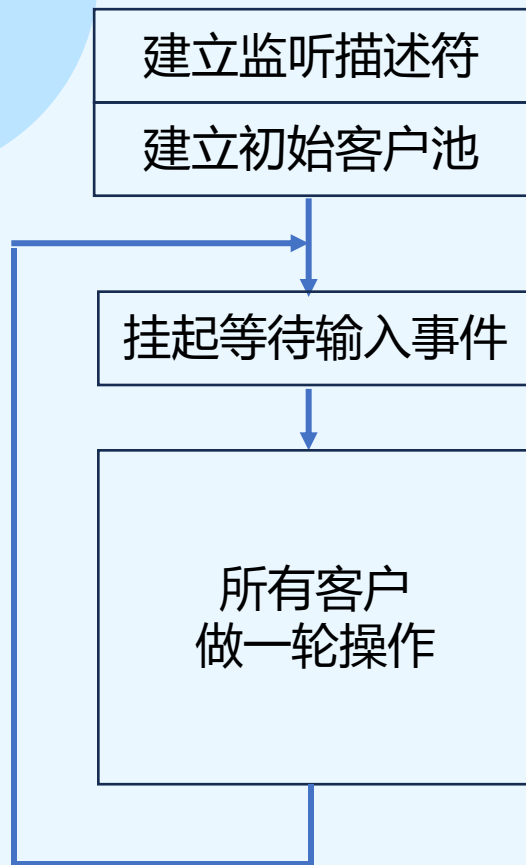


```
listenfd = Open_listenfd(argv[1]);
while(1) {
    clientlen = sizeof(struct sockaddr_storage);
    connfdp = Malloc(sizeof(int));
    *connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
    pthread_create(&tid, NULL, thread, connfdp);
}
```

```
void *thread(void *vargp){
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```

# 基于I/O多路复用的并发事件驱动服务器

自循环状态机



```
listenfd = Open_listenfd(argv[1]);
```

```
while(1) {
```

等待某些描述符准备好可读

如果监听描述符可读，增加一个客户到客户池

扫描客户池，看看哪些客户描述符可读，并完成数据传送或断开连接。

```
}
```

# 挂起等待输入事件select

等待某些描述符准备好可读

```
int select(int n, fd_set *fdset, NULL, NULL, NULL)
```

fd\_set类似于**未被阻塞的信号集合**，是一个大小为n的位向量，表示描述符为k的文件是否在集合中。

可以用一系列宏归零位向量、清空状态、设置状态、检测状态。

当集合中存在文件描述符可读时，select返回可读描述符个数，否则一直挂起。注意返回时select会把fdset改为可读描述符集合。



# 基于I/O多路复用的并发事件驱动服务器

建立监听描述符

建立初始客户池

挂起等待输入事件

所有客户  
做一轮操作

自循环状态机

```
listenfd = Open_listenfd(argv[1]);
```

```
init_pool(listenfd, &pool);
```

```
while(1) {
```

```
    pool.ready_set = pool.read_set;
```

```
    pool.nready = Select(pool.maxfd+1, &pool.ready_set, NULL, NULL, NULL);
```

```
    if (FD_ISSET(listenfd, &pool.ready_set)) {
```

```
        clientlen = sizeof(struct sockaddr_storage);
```

```
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
```

```
        add_client(connfd, &pool);
```

```
    }
```

```
    check_client(&pool);
```

```
}
```

# 客户池及其相关函数

```
typedef struct {  
    int maxfd;  
    fd_set read_set;  
    fd_set ready_set;  
    int nready;  
    int maxi;  
    int clientfd[FD_SETSIZE];  
    rio_t clientrio[FD_SETSIZE];  
} pool;
```

```
init_pool(listenfd, &pool);
```


获得一个只包含活跃的监听描述符的客户池

```
add_client(connfd, &pool);
```

找到一个客户空闲位，把客户放进去，将它与对应的读缓冲区联系起来，更新其它参数，并客户设置为活跃的。

```
check_client(&pool);
```

检索所有准备好的客户，要么读到文本写给客户，要么读到EOF关闭连接并更新其它参数。



思考题：在基于I/O多路复用的并发事件驱动服务器中中某个时刻，临时变量 `nready == 4`，那么从此刻起到下一次自循环状态机到达“挂起等待输入事件”这一状态的过程中，在 `check_client` 函数中可能会调用几次 `Rio_readlineb` 函数？



**Thanks**