目录

[一、实现并发的方式 1](#_Toc200967701)

[1.多进程 实现并发服务 1](#_Toc200967702)

[2.多线程 实现并发服务 1](#_Toc200967703)

[线程池 1](#_Toc200967704)

[3.select()多路I/O复用 实现并发服务(性能差，推荐下面epoll) 3](#_Toc200967705)

[4.epoll\_ctl()多路I/O复用 实现并发服务 3](#_Toc200967706)

[int epoll\_create(int size); 3](#_Toc200967707)

[int epoll\_create1(int flags); 3](#_Toc200967708)

[int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event); 3](#_Toc200967709)

[epoll\_event 结构体 4](#_Toc200967710)

[int epoll\_wait(int epfd, struct epoll\_event \*events,int maxevents, int timeout); 5](#_Toc200967711)

[5.epoll反应堆模型 5](#_Toc200967712)

# 一、实现并发的方式

## 1.多进程 实现并发服务

## 2.多线程 实现并发服务

### 线程池

线程池基于线程(thread)、条件变量(condition)、互斥量(mutex)

线程池没有系统函数可以调用，需要自己写函数

作用：在大量请求到来之前先创建一些线程，这样可以在请求来到的时候立即进行处理，省去了创建线程的开销。

开始时创建一些线程，此时还没有客户端向服务器发送请求，这些线程没有任务可做，于是一直等待任务(等待条件变量haveTask)。一段时间后有客户端向服务器发送请求，此时产生任务(条件变量满足haveTask)，线程开始处理任务。当线程处理完任务后，回到线程池继续等待任务。

当线程快要不够用时，需要进行扩容。相反，当空闲线程过多时要考虑减容。

扩容减容的事交给单独的一个线程做，称之为管理员线程。

任务结构体 和 线程池结构体 如下：

typedef struct Task

{

void (\*function)(void\* arg);

void\* arg;

}Task;

typedef struct ThreadPool

{

int minThreadNum; // 最小线程数量

int maxThreadNum; // 最大线程数量

int busyThreadNum; // 忙线程数

int totalThreadNum; // 线程总数(不包括管理员线程)

int exitThreadNum; // 要销毁的线程个数

int shutdown; // 是否要销毁线程池, 销毁为1, 不销毁为0

pthread\_mutex\_t poolMutex; // 针对整个的线程池加锁

pthread\_mutex\_t busyMutex; // 针对busyThreadNum变量加锁

pthread\_cond\_t haveSlot; // 任务队列是否有空位

pthread\_cond\_t haveTask; // 是否有任务可做

Task\* taskArray; //任务数组

int taskArrayCapacity; //任务数组总容量

int taskArrayTaskNum; // 当前任务个数

int taskArrayFront; // 数组头下标 取任务

int taskArrayRear; // 数组尾下标 放任务

pthread\_t managerThreadId; // 管理员线程ID

pthread\_t \*threadIds; // 工作线程ID数组

} ThreadPool;

## 3.select()多路I/O复用 实现并发服务(性能差，推荐下面epoll)

## 4.epoll\_ctl()多路I/O复用 实现并发服务

### int epoll\_create(int size);

参数：

size: 指示这个新的 epoll 实例预期要监视的文件描述符数量。在较新的内核版本中(特别是从Linux 2.6.8开始)，这个参数实际上被忽略了，但是仍然需要传入一个大于0的值。

返回值：

成功时，返回一个表示新创建的 epoll 实例的文件描述符。失败时，返回 -1 并设置 errno 来指示错误类型。

注意事项:

由于 epoll\_create 的 size 参数在现代内核中已被忽略，推荐使用 epoll\_create1() 函数来代替 epoll\_create()。epoll\_create1() 提供了更多的标志选项，并且不需要指定一个无用的 size 参数。

### int epoll\_create1(int flags);

参数：

flags: 可以传递 0 或者特定的标志位，如 EPOLL\_CLOEXEC，该标志会在执行 exec 家族函数时自动关闭文件描述符。

### int epoll\_ctl(int epfd, int op, int fd, struct epoll\_event \*event);

参数：

epfd: epoll 实例文件描述符

op: 操作类型

EPOLL\_CTL\_ADD: 添加一个新的文件描述符，并注册要监听的事件

EPOLL\_CTL\_MOD: 修改某文件描述符要监听的事件

EPOLL\_CTL\_DEL: 移除一个文件描述符，并不再监听该文件描述符的事件

fd: 需要操作的文件描述符

event: 指向 epoll\_event 结构体的指针。对于 EPOLL\_CTL\_DEL 操作，这个参数可以为 NULL

返回值：

成功时返回 0；失败时返回 -1 并设置 errno 来指示错误类型。

注意事项：

event.data.fd 用于存储事件对应的文件描述符，这样可以在epoll\_wait()返回的events中判断是否是该文件描述符

epoll\_ctl 中的 fd 参数告诉内核操作哪个文件描述符

### epoll\_event 结构体

struct epoll\_event

{

uint32\_t events; // Epoll events (bit mask)

epoll\_data\_t data; // User data variable

};

events: 位掩码，用于表示需要监听的事件集合。常见的事件包括：

EPOLLIN: 文件描述符已准备好进行读操作。

EPOLLOUT: 文件描述符已准备好进行写操作。

EPOLLRDHUP: 对端关闭连接(TCP连接)。

EPOLLPRI: 带有紧急数据的文件描述符。

EPOLLERR: 文件描述符发生错误，总是会被监视，不需要作为事件标志加入。

EPOLLHUP: 文件描述符被挂起，总是会被监视，不需要作为事件标志加入。

判断事件用 位与 操作： if( events[i].events & EPOLLIN )

data: 一个联合体(结构定义在下面)，允许用户将任意数据与文件描述符关联起来。

常用 data.fd，它直接保存了文件描述符本身，方便回调处理。

union epoll\_data

{

void \*ptr;

int fd;

uint32\_t u32;

uint64\_t u64;

}

typedef union epoll\_data\_t epoll\_data

### int epoll\_wait(int epfd, struct epoll\_event \*events,int maxevents, int timeout);

参数：

epfd: epoll 实例的文件描述符

events: 指向 epoll\_event 数组的指针，用于储存返回就绪事件的信息。需要在调用前分配好内存。

maxevents: 指定 events 数组可以容纳的最大事件数。必须大于0。

timeout: 超时时间(以毫秒为单位)。如果设置为 -1，则 epoll\_wait 将无限期阻塞直到至少有一个事件发生；如果设置为 0，则立即返回不管有没有事件发生；如果是正数，则表示最大等待的时间长度。

返回值：

成功时，返回就绪事件的数量。如果超时时间内没有任何事件发生，则返回 0。

失败时，返回 -1 并设置 errno 来指示错误类型。

返回的 events[i] 中，events[i].events 用于确定事件类型，

events[i].data 是之前在 epoll\_ctl() 中通过 event.data 自己定义的数据

epoll\_wait() 有两种触发方式：水平触发(Level Triggered, LT) 和 边缘触发(Edge Triggered, ET)

水平触发: 当文件描述符处于可读(或可写)状态时触发，若一直可读就一直触发

边缘触发: 当文件描述符变为可读(或可写)状态时触发，若一直可读，则只在变为可读的一刻触发

epoll\_wait() 默认采用水平触发

## 5.epoll反应堆模型

epoll反应堆模型基于epoll的非阻塞边缘触发模式

设想有一客户端与服务端通信，客户端向服务端发送请求，服务端处理请求后准备返回数据。但此时客户端因某种原因无法接收数据，这种情况下服务端会一直阻塞下去或直接返回错误。等到客户端可以接收数据时，服务端已经阻塞很长时间或者因之前报错而没有返回数据。前者会浪费服务端资源，后者导致客户端不能正常获取数据，总之都是潜在的问题。

解决方法是将读操作和写操作分离，单独监听可读和可写操作。

没有使用反应堆模型的流程：

epoll\_wait()等待cfd可读 ⇒ cfd可读状态触发事件 ⇒ 读数据 ⇒ 处理数据 ⇒ 发送数据 ⇒ epoll\_wait()等待cfd可读 ⇒ …

中间不能做其他操作

使用反应堆模型的流程：

epoll\_wait()等待cfd可读 ⇒ cfd可读状态触发事件 ⇒

读数据 ⇒ 处理数据 ⇒ 移除监听cfd可读 ⇒ 添加监听cfd可写 ⇒

(期间可以做其他操作)

epoll\_wait()等待cfd可写 ⇒ cfd可写状态触发事件 ⇒

发送数据 ⇒ 移除监听cfd可写 ⇒ 添加监听cfd可读 ⇒

epoll\_wait()等待cfd可读 ⇒ …