发出Web请求到服务器都经历了什么（七）网络IO模型优化socket阻塞调用

上文socket与TCP连接的关系讲socket时，从调用listen()后，我们就可以通过netstat查看对应端口号是否被监听。进入监听状态后，调用accept()从内核获取客户端连接，关于accept还有很多细节可以展开。

当accept接收到多个客户端请求时，服务端如何处理这些请求，以及如何进行数据传输呢？我们需要优化我们的网络IO模型，支持更多的客户端

我们能accept多少客户端请求

* 从TCP连接考虑

我们理论上能支持连接多少客户端呢，TCP由4元组确认：本机IP端口、目标IP端口。因为服务端本机IP端口固定因此，TCP最大连接数（2^48）=客户端IP数（2^32）* 客户端端口数（2^16）。

* 从FD文件描述符FD考虑

socket实际是一个文件，有相应的文件描述符FD，Linux下单个进程的文件描述符一般为1024。为什么是1024?

```
// include/uapi/linux/posix_types.h
#define __FD_SETSIZE    1024

typedef struct {
    unsigned long fds_bits[__FD_SETSIZE / (8 * sizeof(long))];
} __kernel_fd_set;
```

* 从硬件资源角度考虑

C10K问题即并发1万请求，如果服务器的内存只有2GB，网卡是千兆，能支持并发1万请求吗？如果每个请求处理占用不到 200KB 的内存和 100Kbit 的网络带宽就可以满足并发 1 万个请求。但要考虑服务器网络I/O模型能不能支持。

网络I/O模型
=======

什么是IO，IO其实就是读写操作，对网卡对硬盘对文件都有IO。IO对于应用程序来说，其实就是通过向内核发起系统调用，完成对IO的间接访问。应用程序发起IO分为两个阶段：

1. IO调用

应用程序向内核发起系统调用。

2. IO执行

2.1 数据准备阶段：内核等待IO设备准备数据

2.2 数据拷贝阶段：数据从内核缓冲区拷贝到用户缓冲区

阻塞式I/O模式

当阻塞式I/O模式下，一个已连接套接字进行读写操作时会阻塞当前线程/进程，直到操作完成或超时，才会去处理其他客户端的请求。

```
```

```
//创建网络通信套接字
listenfd = socket();
//绑定
bind(listenfd);
//监听
listen(listenfd);
while(1){
    //阻塞 等待建立连接
    connfd = accept(listen_fd,addr,addr_len);
    //读数据 阻塞
    read(connfd，buf);
    //处理
    handler(buf);
    //关闭
    close(connfd);
}
```

可以将服务端理解为上面这个流程，某个accept和read都会造成阻塞，某个socket阻塞会影响到其他socket处理。

* accept阻塞
* read阻塞

根据上文提到的IO模型步骤2，read分为等待读就绪和读数据。

+ 数据准备阶段：等待读就绪

当等待数据到达网卡+网卡的数据拷贝到内核缓冲区之后，会将文件描述符connfd从读未就绪改为读已就绪。
+ 数据拷贝阶段：读数据

内核缓冲区拷贝到用户缓冲区，read可以返回到达的字节数。

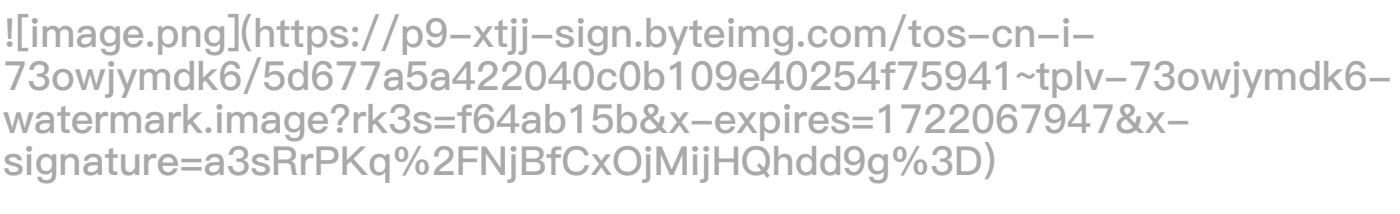
这时候可以考虑使用多线程/多进程，来处理阻塞部分，每来一个客户端都开辟新的线程。但是客户端较多时，会造成资源浪费形成新的瓶颈。

* 多进程阻塞式IO

父进程只关心监听socket，子进程只关心已连接socket。


* 多线程阻塞式IO

同进程里的线程可以共享进程的部分资源，比如文件描述符列表、进程空间、代码、全局数据、堆、共享库等，只需要切换线程的私有数据、寄存器等不共享的数据，开销比多进程小。



```
listenfd = socket();   // 打开一个网络通信套接字
bind(listenfd);        // 绑定
listen(listenfd);      // 监听
while(1) {
  connfd = accept(listenfd);  // 阻塞 等待建立连接
  newThreadDeal(connfd)       // 当有新连接建立时创建一个新线程处理连接
}

newThreadDeal(connfd){
  int n = read(connfd, buf);  // 阻塞 读数据
  doSomeThing(buf);  // 处理数据
  close(connfd);     // 关闭连接
}
```

非阻塞式I/O模式

非阻塞式I/O模式即同步非阻塞式I/O模式，对accept来说不一定要接收到连接也会往下执行，但是这里收到的fd可能是个非法值如负数。会造成CPU空转。

```
```

```
setNonblocking(fd);
accept_fd = accept(listen_fd,addr,addr_len);//负数代表还没有连接
if(accept_fd > 0){
  fd_list.add(fd);
}
```

对读操作read来说,当未就绪读操作时可以返回-1，当不为-1，已就绪时再进行读取。因此在数据准备阶段不阻塞，read会不断调用直到有网卡中数据拷贝到内核缓冲区了，但是读数据阶段（内核缓冲区拷贝到用户缓冲区）仍然是阻塞的。

```
arr = new Arr[];
listenfd = socket();   // 打开一个网络通信套接字
bind(listenfd);        // 绑定
listen(listenfd);      // 监听
while(1) {
  connfd = accept(listenfd);  // 阻塞 等待建立连接
  arr.add(connfd);
}

// 异步线程检测 连接是否可读
new Tread(){
  for(connfd : arr){
    // 非阻塞 read 最重要的是提供了我们在一个线程内管理多个文件描述符的能力
    int n = read(connfd, buf);  // 检测 connfd 是否可读
    if(n != -1){
      newThreadDeal(buf);   // 创建新线程处理
      close(connfd);        // 关闭连接
      arr.remove(connfd);   // 移除已处理的连接
    }
  }
}

newTheadDeal(buf){
  doSomeThing(buf);  // 处理数据
}
```

IO多路复用

服务器不会阻塞在某个连接之上，同时监听多个已连接套接字，根据事件类型进行处理，提高并发性能和吞吐量。其实一个时刻还是只能处理一个请求，但是处理每个请求事件，耗时1ms以内，很像1个CPU并发多个进程，所以也叫时分多路复用。

我们想要把非阻塞IO中不断循环read，让read去判断可读FD的操作交给系统去做，这样可以避免频繁地进行系统调用。不管是select、poll还是epoll，都是通过一个系统调用从内核中获取多个事件。获取事件时，先把所有FD传给内核，再由内核返回产生了事件的连接，在用户态处理这些连接请求。

select

* select用户态FD放到集合中

将已连接的Socket放到一个FD文件描述符集合中。

```
arr = new Arr[];
listenfd = socket();   // 打开一个网络通信套接字
bind(listenfd);        // 绑定
listen(listenfd);      // 监听
while(1) {
connfd = accept(listenfd);  // 阻塞 等待建立连接
arr.add(connfd);
}
```

* 将用户态中FD拷贝到内核中

调用select将用户态中FD拷贝到内核中，让内核检查是否有网络事件。
* 内核检查是否有网络事件

内核检查是否有网络事件的方式就是遍历FD集合，检查到有事件产生，将Socket标记为可读可写（0->1），接着再把整个FD集合拷贝回用户态。
* 用户态遍历找到可读可写Socket

这时只返回可读可写FD的数量，用户不知道具体是哪个FD可读可写，用户态再遍历可读或可写的Socket。

```
// 异步线程检测 通过 select 判断是否有连接可读
new Tread(){
  while(select(arr) > 0){
    for(connfd : arr){
      if(connfd can read){
        // 如果套接字可读 创建新线程处理
        newTheadDeal(connfd);
        arr.remove(connfd);   // 移除已处理的连接
      }
    }
  }
}

newTheadDeal(connfd){
    int n = read(connfd, buf);  // 阻塞读取数据
    doSomeThing(buf);  // 处理数据
    close(connfd);          // 关闭连接
}
```

如果内核没有检查到就绪FD。select并不是继续遍历，直到FD就绪终止遍历。而是遍历完一次内核空间中FD如果没有就绪的，那么就将当前用户的进程阻塞起来，当客户端发送消息，会通过网络传输到达服务器网卡，网卡会通过DMA将这个数据包写入指定内存。处理完成后将通过中断信号告诉CPU有新的数据包到达。CPU收到中断信号后会进行响应中断，调用中断处理程序进行处理。

根据这个数据包的IP和端口号找到对应socket，将数据包保存到这个socket的一个接收队列。检查这个socket对应的等待队列中是否有进程正在阻塞等待，如果有的话就会唤醒该进程。唤醒后会再检查一次内核中的fd集合，检查到如果有fd就绪就会打上标记，停止阻塞，返回给用户空间。

fd_set入参时表示监听了哪些FD，回参时表示哪些FD就绪了。

```
int select(int nfds,
        fd_set *readfds,
        fd_set *writefds,
```

```
        fd_set *exceptfds,
        struct timeval *timeout);

```

* 2次遍历
* 2次拷贝
* 使用固定长度最大限制FD_SETSIZE`1024`的BitsMap，只能监听0-1023FD

poll

针对select主要做了以下2点优化：

* 使用动态数组，链表形式。优化seelct1024的限制。
* 分开监听和就绪的事件。为了避免fd_set复用导致每次调用完都要重置，分开进行存储。

```
//要监听的文件描述符集合
struct pollfd{
    int fd; //监听的文件描述符
    short events; //监听的事件
    short revents; //就绪的事件
}
// pollfd 要监听的文件描述符集合
// nfds 文件描述符数量
// timeout 本次调用超时时间
// return >0 已就绪的文件描述符数 <0 出错 =0 超时
int poll(struct pollfd *fds,
        unsigned int nfds,
        int timeout);

```

epoll

epoll通过epoll_create创建了一个epoll对象，再通过epoll_ctl将要监听的socket添加到该对象中，最后调用epoll_wait()等待数据。支持连接的上限为系统定义的进程打开的最大文件描述符个数。

* 红黑树结构

通过epoll_ctl()将需要监听的socket加入内核中的红黑树中。使用红黑树来跟踪进程所有待检测的FD，不像select、poll传入整个fd，因此减少内核和用户空间以及数据拷贝。
* 事件驱动机制/事件通知模式

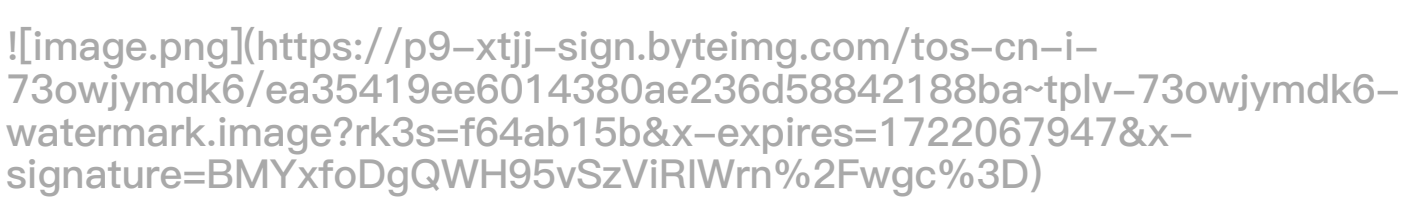
内核里维护了一个链表来记录就绪事件，当FD有数据可读时，调用epoll_wait()可以得到通知，只返回有事件发生的文件描述符个数。但事件通知模式有2种：

+ LT水平触发 LevelTriggered【默认】

当被监控的Socket上有可读事件发生时，不断通知。只要满足事件的条件，比如内核中有数据需要读，就一直不断地把这个事件传递给用户。

select、poll只有水平触发，epoll默认使用水平触发。
+ ET边缘触发 EdgeTriggered

当被监控的Socket描述符上有可读事件发生，只通知一次。只有第一次满足条件的时候才触发，之后就不会再传递同样的事件了。




```
listenfd = socket();   // 打开一个网络通信套接字
bind(listenfd);        // 绑定
listen(listenfd);      // 监听
int epfd = epoll_create(...); // 创建 epoll 对象
while(1) {
  connfd = accept(listenfd);  // 阻塞 等待建立连接
  epoll_ctl(connfd, ...);  // 将新连接加入到 epoll 对象
}

// 异步线程检测 通过 epoll_wait 阻塞获取可读的套接字
new Tread(){
```

```
  while(arr = epoll_wait()){
    for(connfd : arr){
        // 仅返回可读套接字
        newTheadDeal(connfd);
    }
  }
}

newTheadDeal(connfd){
    int n = read(connfd, buf);  // 阻塞读取数据
    doSomeThing(buf);  // 处理数据
    close(connfd);         // 关闭连接
}

```

多路复用过程其实就是高性能IO设计模式中Reactor模式：

1. 对客户端注册感兴趣事件
2. 专门有一个线程去轮询客户端是否有事件发生
3. 有事件发生进行处理再去轮询

综上，IO复用的效率高是因为减少了系统调用，并且通过一个线程管理多个文件描述符。

事件驱动 IO

发起读请求后，等待读就绪事件通知再进行数据读取。

异步 IO

检测到有事件发生，发起异步操作交由内核线程处理。内核线程完成IO操作后，发送一个通知告知操作完成，即异步IO就是高性能IO设计模式中的Proactor模式。发起读请求后，等待操作系统读取完成后通知，完全将功能交给操作系统实现。

也就是第二阶段操作内核拷贝到用户内核的步骤也非阻塞了。只需要留下一个回调函数（如Future#get），异步线程读完之后就知道怎么做了。

参考：

1. [xiaolincoding.com/os/8_networ…](http://cxyroad.com/”https://xiaolincoding.com/os/8_network_system/selete_poll_epoll.html#%E6%9C%80%E5%9F%BA%E6%9C%AC%E7%9A%84-socket-%E6%A8%A1%E5%9E%8B”)
2. [puppylpg.github.io/posts/2022/…](http://cxyroad.com/”https://puppylpg.github.io/posts/2022/02/24/io-nio-aio/”)
原文链接: https://juejin.cn/post/7393293636685971492