Linux网络编程演进过程

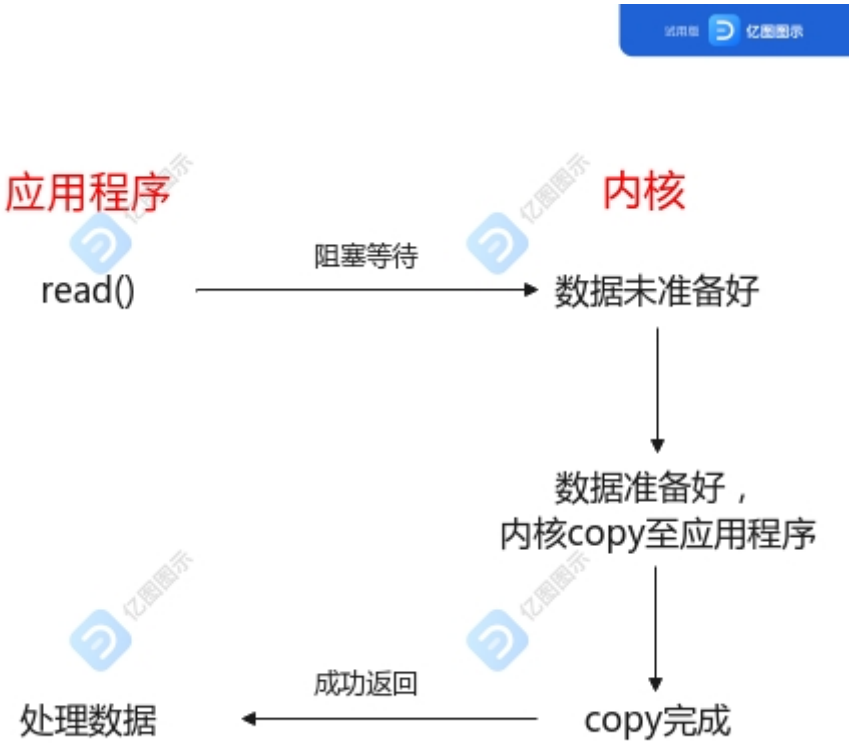
一、IO模型：阻塞、非阻塞、同步、异步

以从内核读取数据为例（向内核写入数据是逆过程）：

应用进程调用系统函数，如read()函数，从内核中读取缓冲区数据时，内核首先会**检查数据是否准备好**，当数据准备好后，**从内核拷贝数据到应用进程**，拷贝完成后，向应用进程返回成功。

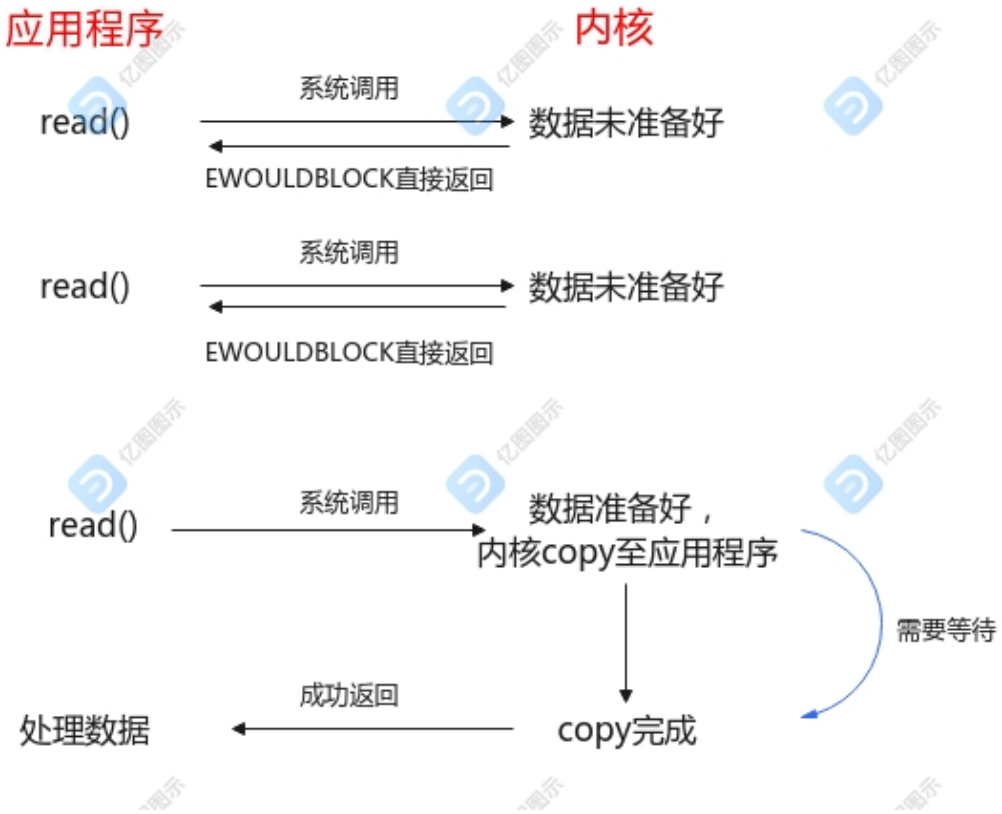
- 阻塞IO：

  当应用程序调用read()函数时，需要阻塞等待至内核将数据准备好，并且等待内核将数据copy至应用程序才最终返回。



- 非阻塞IO：

  当应用程序调用read()函数时，不需要阻塞等待至内核将数据准备好，若内核此时没有将数据准备好可直接返回，但是若数据准备好，copy的这个过程，是需要等待内核将数据copy至应用程序才最终返回。

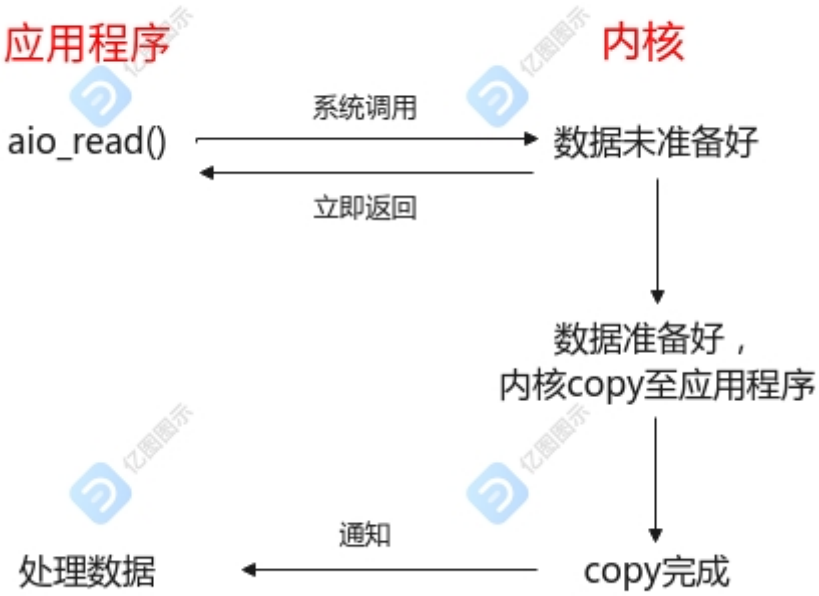


- 同步IO：

  上面提到的阻塞和非阻塞IO，在内核将准备好的数据copy至应用程序的过程中都需要等待，这是一个同步过程。我们指的同步是内核态的数据copy到用户程序缓冲区的过程，因此上面所说的两种都是同步IO。

- 异步IO：

  异步的IO进行系统调用后（如aio_read函数）可以立即返回，整个copy过程无需用户等待，内核会自动完成，完成copy动作后通知应用程序。



二、演进过程

整体演进过程：TCP/IP基本socket----->IO多路复用----->Reactor模型----->Proactor模型

- TCP/IP基本socket：

  服务端的基本TCP/IP socket处理网络数据的过程就是创建socket，bind，listen，然后accept产生一个连接socket。调用recv系统调用接受数据，处理数据后，向客户端返回相应结果。

```
#include<strings.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<signal.h>
#include<unistd.h>
#include<stdlib.h>
#include<assert.h>
#include<stdio.h>
#include<string.h>
#include<errno.h>

int main(int argc, char* argv[])
{
    if(argc<=2)
    {
        printf("Usage:%s ip_address port_number\n",basename(argv[0]));
        return 1;
    }

    const char *ip = argv[1];
    int port = atoi(argv[2]);
    int sock = socket(PF_INET,SOCK_STREAM,0);
    assert(sock>0);

    struct sockaddr_in address; //ipv4
    bzero(&address, sizeof(address));
    address.sin_family = AF_INET;
    inet_pton(AF_INET,ip,&address.sin_addr);
    address.sin_port = htons(port);

    int ret = bind(sock,(struct sockaddr*)&address,sizeof(address));
    assert(ret!=-1);
    ret = listen(sock, 5);
    assert(ret!=-1);

    struct sockaddr_in client; //ipv4
    socklen_t client_addrlenth = sizeof(client);
    int connfd = accept(sock, (struct sockaddr*)&client, &client_addrlenth);
    if(connfd<0)
    {
        printf("errno is:%d\n",errno);
    }
    else
    {
        char remote[INET_ADDRSTRLEN];
```

```
        printf("connected with ip: %s and port:
%d\n",inet_ntop(AF_INET,&client.sin_addr,remote,INET_ADDRSTRLEN),ntohs(client.sin_port));
        close(connfd);
    }
    close(sock);
    return 0;
}
```

服务器程序一般不能设计成一次只服务一个客户端，那要同时服务多个连接怎么办呢？对每个客户连接新创建一个进程/线程？但是系统能创建的进程/线程数是有限的，我们从单进程单线程的角度看linux网络编程的演进过程。

这里需要看一下阻塞IO对服务程序产生的影响。对于listenfd（监听socket），如果是阻塞IO，没有连接请求时，则会阻塞在accept()；对于connfd（已连接socket），如果是阻塞IO，没有可读数据时，则会阻塞到recv();

```
#include <sys/socket.h>
#include <sys/types.h>
#include <arpa/inet.h>
#include <assert.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <stdio.h>
#include <errno.h>

int setnoblocking(int fd)
{
    int old_option = fcntl(fd, F_GETFL);
    int new_option = old_option|O_NONBLOCK;
    fcntl(fd, F_SETFL,new_option);
    return old_option;
}

void recv_data(int connfd)
{
    char buf[1024];
    memset(buf, 0, 1024);
    int ret = recv(connfd,buf,1024-1,0);//若connfd是阻塞IO，则会被阻塞在这里
    if(ret<=0)
    {
        if(EAGAIN == errno || errno==EWOULDBLOCK) //若connfd是非阻塞IO，则errno会返回非阻塞
的再次尝试
            printf("not blocking, next time read\n");
        else
        {
            close(connfd);
            printf("client socket has been closed\n");
        }
    }
    else
        printf("recv data: %s\n",buf);
}
```

```
int main(int argc,char* argv[])
{
    if(argc<=2)
    {
        printf("Usage:%s ip_address port_number\n",basename(argv[0]));
        return 1;
    }
    u_int8_t ret;
    const char* ip = argv[1];
    uint16_t port = atoi(argv[2]);

    struct sockaddr_in address;
    bzero(&address, sizeof(address));
    address.sin_family = AF_INET;
    inet_pton(PF_INET, ip, &address.sin_addr);
    address.sin_port = htons(port);

    int listenfd = socket(PF_INET,SOCK_STREAM,0);
    ret = bind(listenfd,(struct sockaddr*)&address,sizeof(address));
    assert(ret!=1);
    ret = listen(listenfd, 5);
    setnoblocking(listenfd);  //设置listenfd非阻塞IO
    assert(ret!=1);

    while(1) //想要接受新的连接，就要不停循环检测，但是这样浪费CPU资源
    {
        struct sockaddr_in client;
        socklen_t client_len = sizeof(client);
        int connfd = accept(listenfd, (struct sockaddr*)&client, &client_len);//若listenfd是
阻塞IO，则会被阻塞在这里
        if(connfd>0)
        {
            printf("accepted\n");
            // setnoblocking(connfd);  //设置connfd非阻塞IO
            recv_data(connfd);
            continue;
        }
        else if(EAGAIN == errno || errno==EWOULDBLOCK) //若listenfd是非阻塞IO，则errno会返回
非阻塞的再次尝试
        {
            printf("nothing accepted,not blocking, now could do something other\n");
            continue;
        }
        else
            printf("something wrong\n");
    }
    close(listenfd);
    return 0;
}
```

那么这里面就有问题了，如果想要接受新的客户端连接，就要不停地去循环，调用accept，如果listenfd是阻塞IO，那么就可能会被阻塞在accept函数，程序被阻塞住不能处理其他事情；而非阻塞的IO可以直接返回，但若要及时处理客户端的链接，仍然要不停的循环检测，**这种用户态的不停循环会占用很高的CPU导致性能浪费**。

读取数据也是同样的问题，对于一个connfd，如果是阻塞的IO，那么可能会阻塞到recv()函数，程序被阻塞住不能处理其他事情；而非阻塞的IO可以直接返回，但是要及时处理可能到达的数据，或者同时有很多connfd需要检测，仍然需要不停的循环，这种用户态的不停循环会导致CPU性能浪费。

那么让内核帮助我们去检测有没有事件发生是一种比较合适的方式，让内核有新的事件发生才通知我们用户应用程序。内核态的检测不会像用户态的while循环那样极大的消耗CPU资源。而让内核帮助我们检测有无新事件发生就产生了IO多路复用技术，使用IO多路复用技术+非阻塞IO可以很好的解决上面提到的缺点。（***可以使用top命令，对比上面代码和使用IO多路复用时，对于非阻塞IO在等待用户连接时的CPU占用情况***）

- Linux的IO多路复用：select、poll、epoll

  对于select和poll，它们需要将socketfd都放到一个文件描述符集合中，然后调用select或poll函数，交给（**copy**给）内核来检查文件描述符集合中是否有事件发生。内核的检查方式很简单——就是**遍历**。内核发现由新事件发生，就会把发生事件的socketfd标记相应事件，然后将整个文件描述符集合**copy**返回给用户态程序。用户态程序拿到后，也不知道是整个文件描述符集合中具体哪个发生了事件，仍然再**遍历**一遍，找出发生事件的socketfd，然后开始处理这些fd对应的事件。

  综上，一次检测+处理事件，需要copy两遍文件描述符集合和两次遍历。

```
//select()的调用形式为：
#include <sys/select.h>
#include <sys/time.h>
int select(int maxfd, fd_set *readfds, fd_set *writefds, fe_set *exceptfds, const struct
timeval *timeout);
```

```
//poll()的调用形式为：
#include <poll.h>
int poll(struct pollfd* fds, nfds_t nfds, int timeout)
```

以上两种IO多路复用在大并发时不如epoll，epoll可以精准返回，到底是哪些socketfd有事件发生，并且也不需要将文件描述符集合全量copy给内核。我们这里以epoll为例实现并发访问处理。

```
//epoll系列：
#include <sys/epoll.h>
int epoll_create(int size);
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout);
```

使用epoll的例子：

```
#include "sys/epoll.h"
#include "sys/socket.h"
#include "sys/types.h"
#include <arpa/inet.h>
#include <assert.h>
#include <fcntl.h>
#include <netinet/in.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include "stdio.h"
#include "errno.h"
```

```c
int setnoblocking(int fd)
{
    int old_option = fcntl(fd, F_GETFL);
    int new_option = old_option|O_NONBLOCK;
    fcntl(fd, F_SETFL,new_option);
    return old_option;
}

void add_fd(int epfd,int fd)
{
    struct epoll_event ev;
    ev.events = EPOLLIN;
    ev.data.fd = fd;
    epoll_ctl(epfd, EPOLL_CTL_ADD, fd, &ev);
    setnoblocking(fd);
}

void recv_data(int connfd)
{
    char buf[1024];
    memset(buf, 0, 1024);
    int ret = recv(connfd,buf,1024-1,0);
    if(ret<=0)
    {
        if(EAGAIN == errno || errno==EWOULDBLOCK)
            printf("not blocking, next time read\n");
        else
        {
            close(connfd);
            printf("client socket has been close\n");
        }
    }
    else
        printf("recv data: %s\n",buf);
}

int main(int argc,char* argv[])
{
    if(argc<=2)
    {
        printf("Usage:%s ip_address port_number\n",basename(argv[0]));
        return 1;
    }
    u_int8_t ret;
    const char* ip = argv[1];
    uint16_t port = atoi(argv[2]);

    struct sockaddr_in address;
    bzero(&address, sizeof(address));
    address.sin_family = AF_INET;
    inet_pton(PF_INET, ip, &address.sin_addr);
    address.sin_port = htons(port);

    int listenfd = socket(PF_INET,SOCK_STREAM,0);
    ret = bind(listenfd,(struct sockaddr*)&address,sizeof(address));
    assert(ret!=1);
    ret = listen(listenfd, 5);
```

```
    assert(ret!=1);

    int epfd = epoll_create(1024);
    add_fd(epfd,listenfd);

    while(1)
    {
        struct epoll_event ev[1024];
        int n = epoll_wait(epfd, ev, 1024, -1);
        for(int i=0;i<n;i++)
        {
            if(ev[i].data.fd == listenfd)
            {
                struct sockaddr_in client;
                socklen_t client_len = sizeof(client);
                int connfd = accept(listenfd, (struct sockaddr*)&client, &client_len);
                add_fd(epfd,connfd);
            }
            else if(ev[i].events & EPOLLIN)
            {
                recv_data(ev[i].data.fd);
            }
            else
            {
                printf("something wrong\n");
            }
        }
    }

    close(listenfd);
    return 0;

}
```

epoll支持边缘触发（ET）和水平触发（LT）两种模式，还有EPOLLONESHOT事件来保证一次只有一个线程在处理当前fd，本篇重点关注linux网络编程的演进过程，这些暂不讨论。

- Reactor模型：

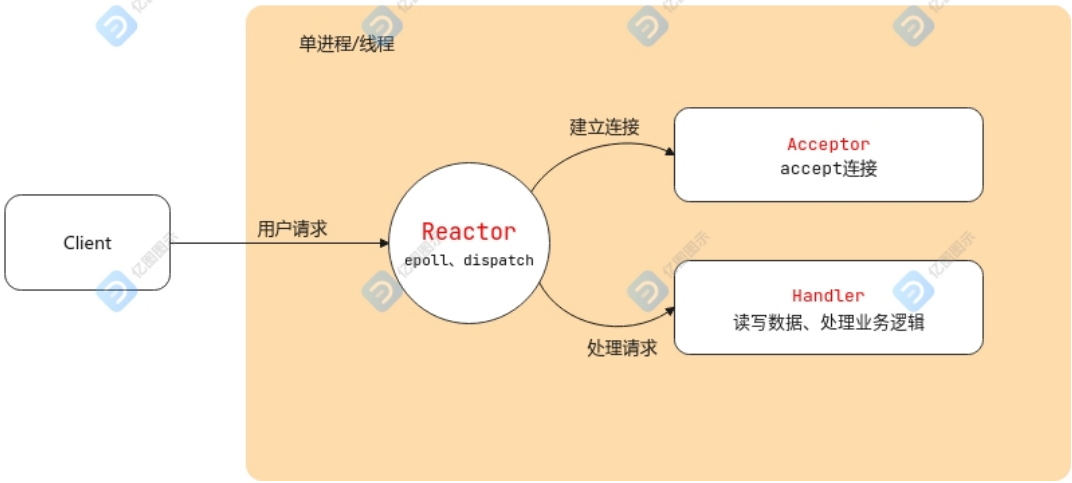
  其实所谓Reactor是对IO多路复用技术做了一层封装，相当于将epoll等IO多路复用技术的编程方式**由面向过程转为面向对象**。这样我们可以不关注底层实现，而主要关注业务层面。Reactor是对事件的反应堆，当事件到来时，Dispatch（分发）给某个进程/线程去处理。

  因此Reactor模型有两个关键部分：Reactor和Handler

  - Reactor：负责监听事件和Dispatch事件，包括连接事件、读写事件等。因此通常要提供注册事件、取消注册、分发任务等方法。
  - Handler：处理被分发的事件，如读取数据，业务处理，写入数据等。

  Reactor模型非常灵活，可以自由设计Reactor的数量和与进程/线程的组合方式，如：

  - 单Reactor+单进程/线程



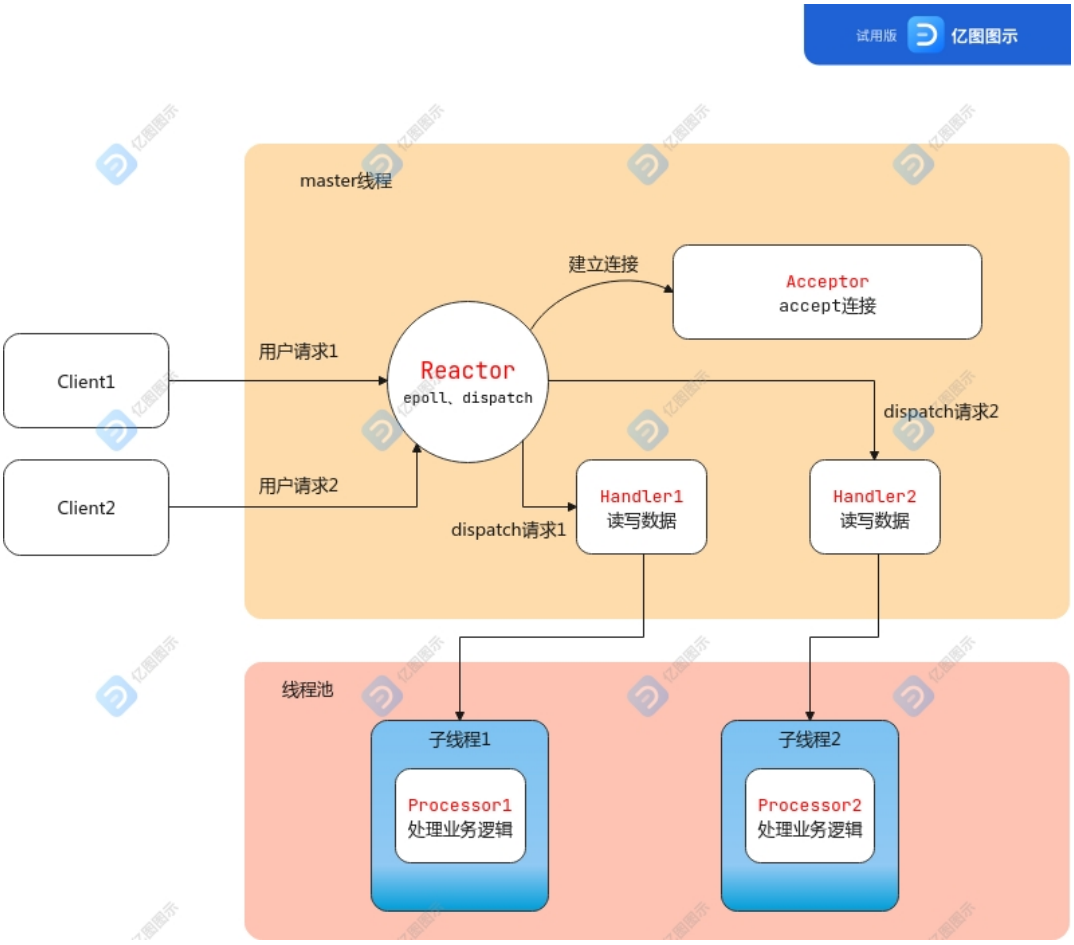
Reactor通过epoll_wait监听事件，收到事件后通过dispatch方法进行分发（需要区别是建立连接的事件还是处理其他读写事件）；

对于建立连接，则dispatch给Acceptor，Acceptor针对此连接创建新的Handler对象，将connfd（已连接socketfd）加入Reactor继续监听；

对于其他如读写事件，则dispatch给相应的Handler对象进行读写以及业务逻辑的处理；

由于单Reactor+单进程/线程**无法利用CPU多核性能**，并且性能与Handler的处理能力十分相关，由于是单线程，**如果Handler业务处理的慢了，那么会影响整个系统的性能**。所以后面出现了单Reactor+多进程/线程的模型。

- 单Reactor+多进程/线程



Reactor通过epoll_wait监听事件，收到事件后通过dispatch方法进行分发（需要区别是建立连接的事件还是处理其他读写事件）；

对于建立连接，则dispatch给Acceptor，Acceptor针对此连接创建新的Handler对象，将connfd（已连接socketfd）加入Reactor继续监听；

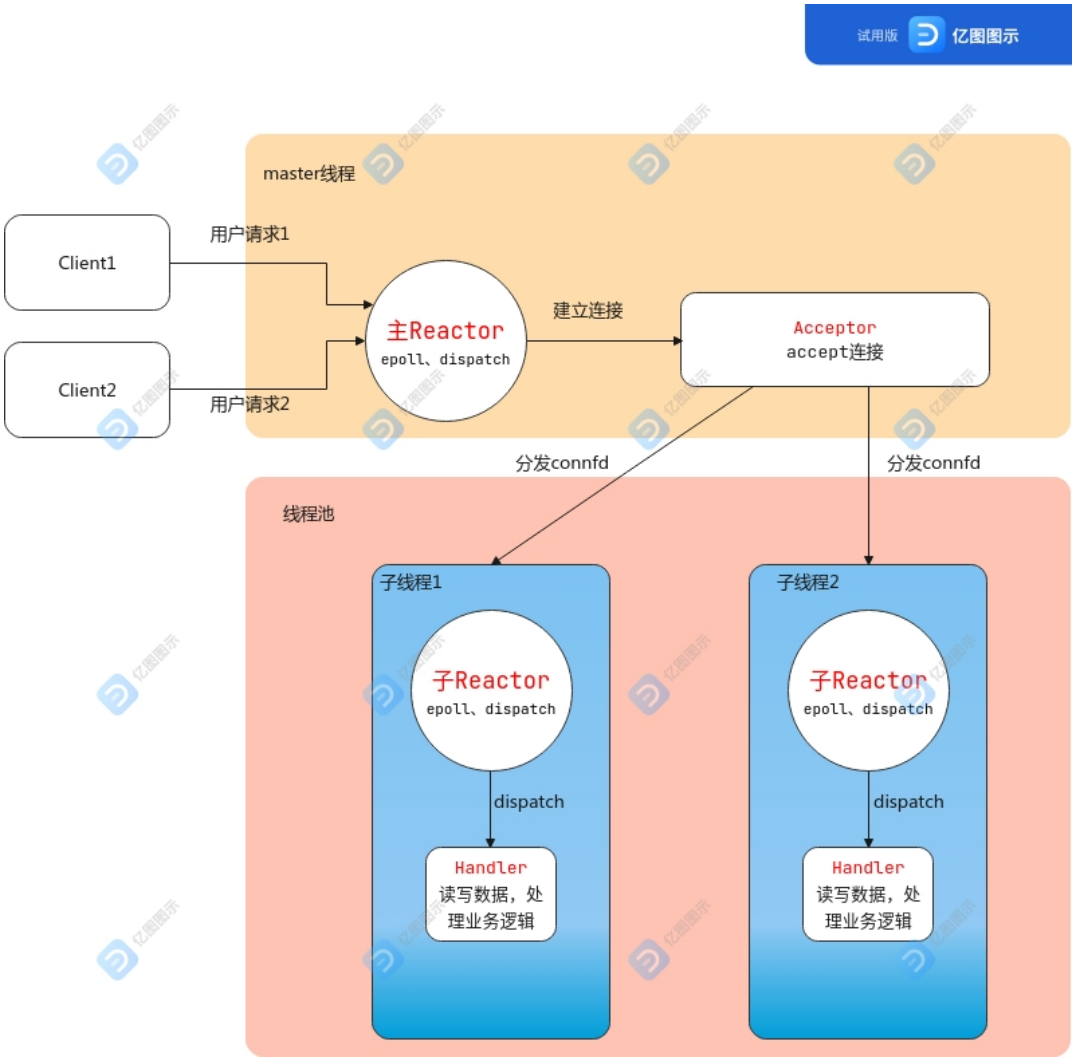
对于其他如读写事件，则dispatch给相应的Handler对象进行读取，而Handler对象读取完数据后，将数据发送给子线程对应的Processor进行业务处理，Processor处理之后在将处理后的数据返回给对应的Handler，Handler在响应回客户端；

这种"单Reactor+多进程/线程"相比于"单Reactor+单进程/线程"，不仅**可以利用CPU多核，而且主线程中的Handler不再负责业务处理，如果业务处理较慢也不会影响主线程的其他任务**。但是，由于使用了多线程，那么就很可能涉及到**共享资源的竞争**，如只有主线程一个Reactor，对共享资源的操作要注意线程安全。

单Reactor的另一个问题是：**由于只有一个Reactor在主线程运行，所有的监听和响应都由它自己负责，当有很大的并发请求时，可能成为性能瓶颈**。

所以后面出现了多Reactor的方式。

- 多Reactor+多进程/线程



主Reactor通过epoll_wait监听事件，注意这里只负责监听连接事件，收到连接事件后给Acceptor；

Acceptor建立连接，针对此连接，将connfd（已连接socketfd）分发给不同子线程；

子线程拿到connfd，加入到自己线程的子Reactor继续监听后续事件，并且创建一个Handler对象用于处理后续事件；

Handler负责读写数据和处理业务逻辑；

> 这样的模型线程间逻辑清晰，主线程只负责接受新建连接，子线程负责业务处理，由于子线程有自己的Reactor，业务处理后的结果可以直接在本线程返回给客户端。

- Proactor模型：

上面的Reactor模型都是基于同步IO模型进行阐述的，而Proactor模型需要用到异步IO。

Reactor的同步IO，内核通知应用程序事件发生，感知的是**就绪的可读写事件（接到通知后，自己现去内核读取）；**

Proactor的异步IO，感知的是**已完成的读写事件（接到通知后，数据已在用户空间，可直接使用）**。

不过当前Linux的异步IO还不算完善，没有真正的实现。

参考：《Linux高性能服务器编程》、小林Coding微信公众号