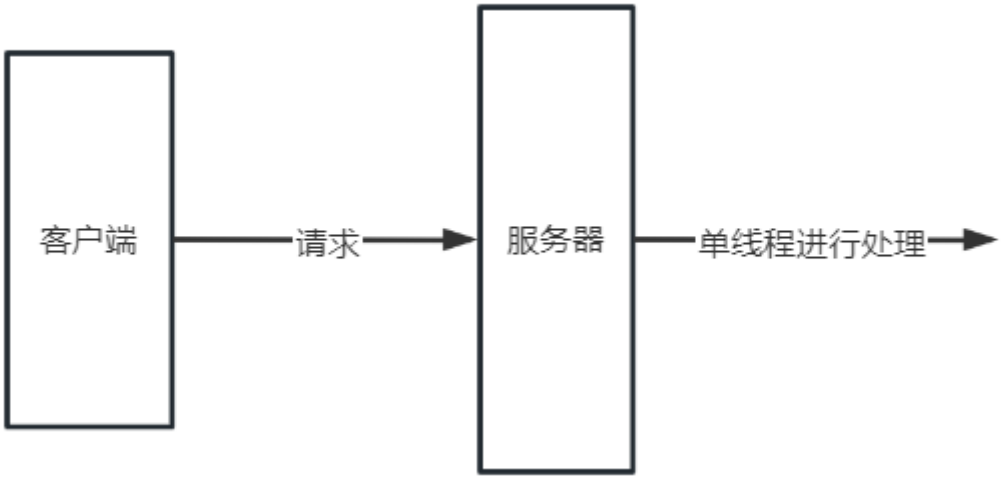
IO演进

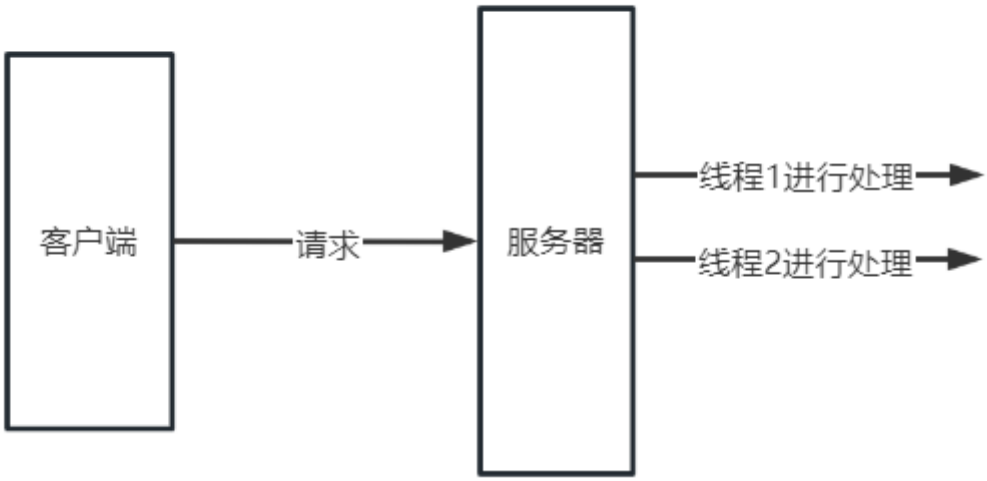
最原始方式，服务器只有一个主线程接受请求并进行处理

```
int sock_client = accept(serverfd, (struct sockaddr *)(&client_addr), (unsigned
int *)(&size));
//处理请求
```

一次只能服务于一个客户端



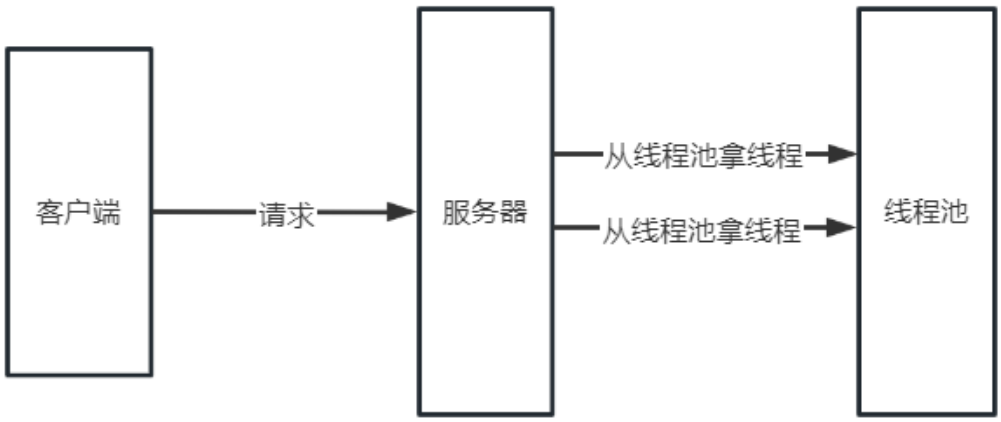
如果要让服务器服务多个客户端，那么最直接的方式就是一接收后就为每一条连接创建线程去进行处理。



```
int sock_client = accept(serverfd, (struct sockaddr *)(&client_addr), (unsigned
int *)(&size));
new thread(....)。//交给线程后立即监听下一个请求
```

- 处理完业务逻辑后，随着连接关闭后线程也同样要销毁了，但是这样不停地创建和销毁线程，不仅会带来性能开销，也会造成浪费资源，
- 而且如果要连接几万条连接，创建几万个线程去应对也是不现实的。

使用线程池，也就是不用再为每个连接创建线程，将连接分配给线程，然后一个线程可以处理多个连接的业务。




```
int sock_client = accept(serverfd, (struct sockaddr *)(&client_addr), (unsigned
int *)(&size));
threadpool.execute(....)。//交给线程后立即监听下一个请求
```

这种方式会只要有连接，我就从线程池拿一个线程来处理。但很多时候线程一般采用「read -> 业务处理 -> send」的处理流程，如果当前连接没有数据可读，那么线程会阻塞在 `read` 操作上（socket 默认情况是阻塞 I/O），那么线程就没办法继续处理其他连接的业务。这就造成了线程的浪费。（比如我们登陆QQ了，但是很多时候其实是挂在那里，但挂在那里也浪费了一个线程，这样累计浪费的资源就很多）。

总结现在的问题就是以上所有方式都是主动创建线程去读，读不了就阻塞（改为非阻塞的话，轮巡非常消耗CPU，另外很多时候也必须拿到了数据才能进行业务操作，反而效果还不如阻塞）。

只有当连接上真正有数据交互的时候，线程才去发起读请求呢?

实现这一技术的就是 I/O 多路复用。I/O 多路复用技术会用一个系统调用函数来监听我们所有关心的连接，也就说可以在一个监控线程里面监控很多的连接。

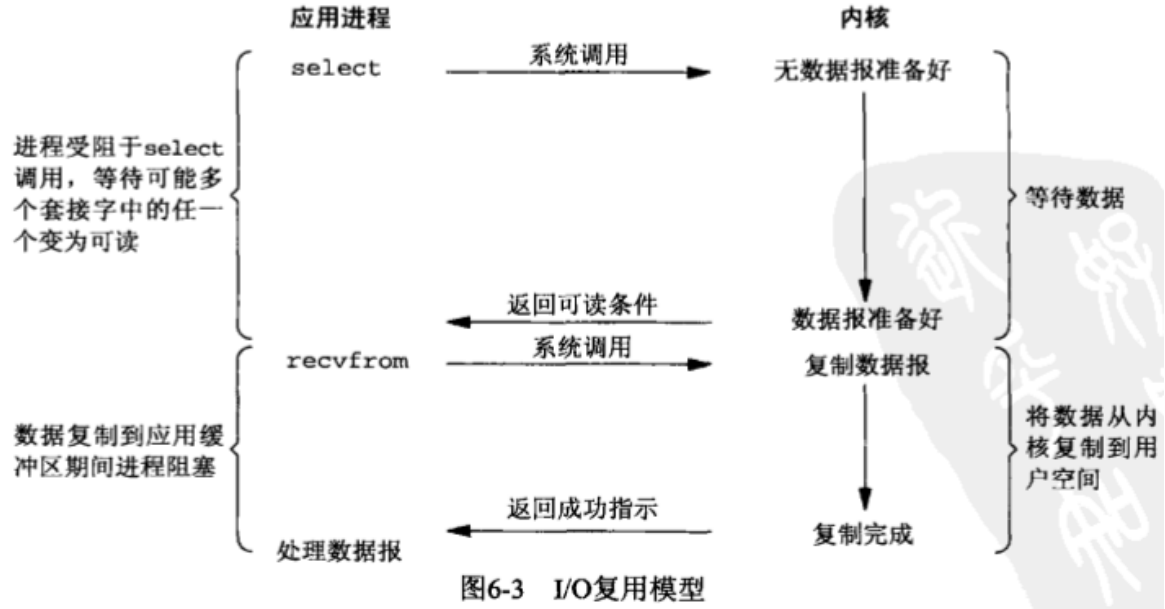


图6-3　I/O复用模型

刚连接上时没有线程创建，直接调用select监听。没有数据就阻塞，有数据就唤醒进行处理（可以单线程，也可以多线程）。

如果每次都用最基础的多路复用代码来写程序，效率就很低。所以基于面向对象的思想，对 I/O 多路复用作了一层封装同时做一些扩展。就形成了Reactor模式。

Reactor模式

Reactor 模式主要由 **Reactor** 和**处理资源池**这两个核心部分组成，它俩负责的事情如下：

- Reactor 负责**监听和分发事件**，**事件类型**包含**连接事件、读写事件**；
- 处理资源池负责处理事件，如 read -> 业务逻辑 -> send；

Reactor 模式具体来说还分几种，可以应对不同的业务场景，原因在于

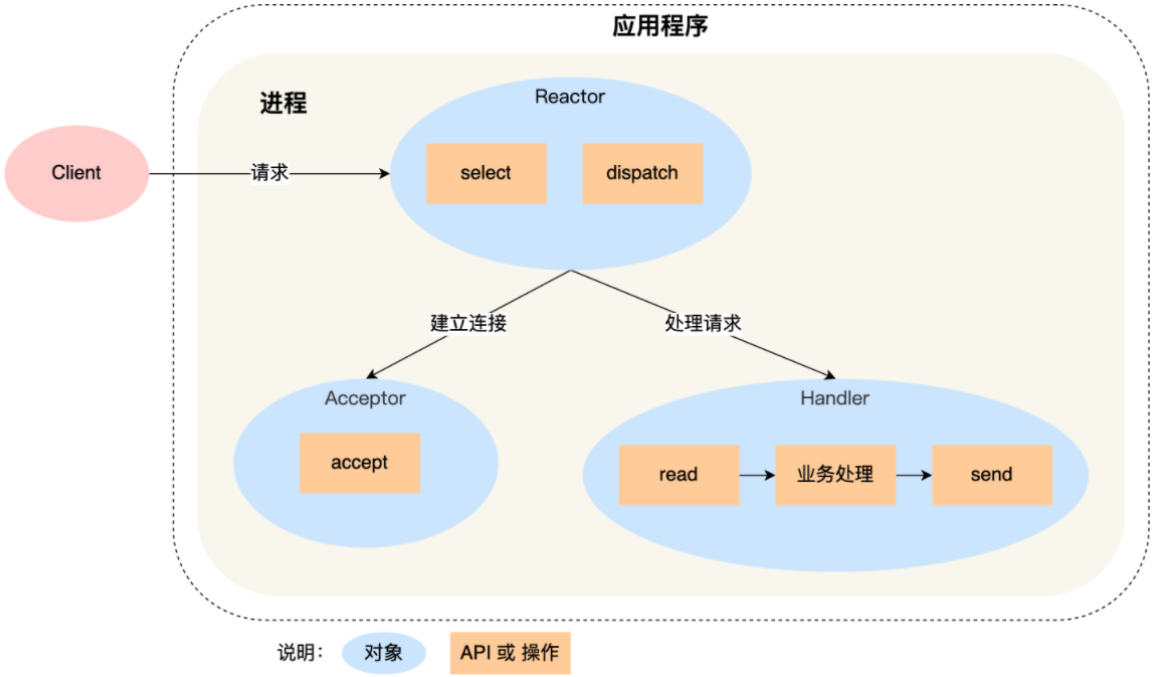
- Reactor 的数量可以只有一个，也可以有多个；
- 处理资源池可以是单个进程 / 线程，也可以是多个进程 /线程

一般来说排列组合后就是

- 单Reactor单进程下的单线程
- 单Reactor单进程下的多线程
- 多Reactor单进程下的多线程

其它的要么不常见，要么就是很效果很糟糕直接不使用。

单 Reactor 单线程（单进程）



可以看到进程里有 **Reactor、Acceptor、Handler** 这三个对象：

- Reactor 对象的作用是监听和分发事件；
- Acceptor 对象的作用是建立连接；
- Handler 对象的作用是处理业务；

过程

- Reactor 对象通过 select （IO 多路复用接口） 监听事件，收到事件后通过 dispatch 进行分发，具体分发给 Acceptor 对象还是 Handler 对象，还要看收到的事件类型；
- 如果是连接建立的事件，则交由 Acceptor 对象进行处理，**Acceptor 对象会通过 accept 方法 获取连接，并创建一个 Handler 对象来处理后续的响应事件**；
- 如果不是连接建立事件， 则交由当前连接对应的 Handler 对象来进行响应；
- Handler 对象通过 read -> 业务处理 -> send 的流程来完成完整的业务流程。

单 Reactor 单进程的方案因为全部工作都在同一个进程内完成，所以实现起来比较简单，不需要考虑进程间通信，也不用担心多进程竞争。

但是，这种方案存在 2 个缺点：

- 第一个缺点，因为只有一个线程，**无法充分利用多核 CPU 的性能**；
- 第二个缺点，Handler 对象在业务处理时，整个进程是无法处理其他连接的事件的，**如果业务处理耗时比较长，那么就造成响应的延迟**；

所以，单 Reactor 单进程的方案**不适用计算机密集型的场景，只适用于业务处理非常快速的场景。**

Redis 是由 C 语言实现的，它采用的正是「单 Reactor 单进程」的方案，因为 Redis 业务处理主要是在内存中完成，操作的速度是很快的，性能瓶颈不在 CPU 上，所以 Redis 对于命令的处理是单线程的方案。（存取命令get,set操作非常快）

伪代码

最原始的IO复用代码

```
int main(int argc, char **argv)
{
    while (1)
    {
```

```
        /*初始化文件描述符号到集合*/
        FD_ZERO(&client_fdset);
        /*加入服务器描述符*/
        //初步添加进去，我要监听的
        FD_SET(serverfd, &client_fdset);
        /*设置超时时间*/
        tv.tv_sec = 30; /*30秒*/
        tv.tv_usec = 0;
        /*把活动的句柄加入到文件描述符中*/
        for (int i = 0; i < 5; ++i)
        {
            /*程序中Listen中参数设为5,故i必须小于5*/
            if (client_sockfd[i] != 0)
            {
                // 不等于0才放进去进行监听。
                //初步添加进去，我要监听的
                FD_SET(client_sockfd[i], &client_fdset);
            }
        }
        /*printf("put sockfd in fdset!\n");*/
        /*select函数*/
        // 阻塞监听clientz_fdset这个文件句柄集合
        ret = select(maxsock + 1, &client_fdset, NULL, NULL, &tv);
        //返回后，client只存储了真正响应了的文件句柄
        if (ret < 0)
        {
            perror("select error!\n");
            break;
        }
        else if (ret == 0)
        {
            printf("timeout!\n");
            continue;
        }
        /*轮询各个文件描述符*/

        // 必须循环完，因为ret只知道是几个，但是不知道具体是哪几个有响应。
        for (int i = 0; i < conn_amount; ++i)
        {
            /*FD_ISSET检查client_sockfd是否可读写，>0可读写*/
            // 如果存在里面，那么就说明有响应码？确实是
            if (FD_ISSET(client_sockfd[i], &client_fdset))
            {
                printf("start recv from client[%d]:\n", i);
                ret = recv(client_sockfd[i], buffer, 1024, 0);
                // 没收到，或者读出来为0
                if (ret <= 0)
                {
                    printf("client[%d] close\n", i);
                    close(client_sockfd[i]);
                    FD_CLR(client_sockfd[i], &client_fdset);
                    client_sockfd[i] = 0;
                }
                else
                {
                    printf("recv from client[%d] :%s\n", i, buffer);
                }
            }
```

```
        } // 这里是for结束了

        /*检查是否有新的连接，如果收，接收连接，加入到client_sockfd中*/
        //如果输入dd后，它会立刻返回，相当于就是serverfd没有响应就不会存在于client_fdset里面了，这里判断就不会成功，就不会进去了
        if (FD_ISSET(serverfd, &client_fdset))
        {
            /*接受连接*/
            struct sockaddr_in client_addr;
            size_t size = sizeof(struct sockaddr_in);
            int sock_client = accept(serverfd, (struct sockaddr *)(&client_addr), (unsigned int *)(&size));
            if (sock_client < 0)
            {
                perror("accept error!\n");
                continue;
            }
            /*把连接加入到文件描述符集合中*/
            if (conn_amount < 5)
            {
                client_sockfd[conn_amount++] = sock_client;
                bzero(buffer, 1024);
                strcpy(buffer, "this is server! welcome!\n");
                send(sock_client, buffer, 1024, 0);
                printf("new connection client[%d] %s:%d\n", conn_amount, inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));
                bzero(buffer, sizeof(buffer));

                ret = recv(sock_client, buffer, 1024, 0);
                if (ret < 0)
                {
                    perror("recv error!\n");
                    close(serverfd);
                    return -1;
                }
                printf("recv : %s\n", buffer);
                if (sock_client > maxsock)
                {
                    maxsock = sock_client;
                }
                else
                {
                    printf("max connections!!!quit!!\n");
                    break;
                }
            } // 这里对应conn_amount
        }     // 这里对应新连接
    }         // 这里是while结束了


}
```

现在的代码

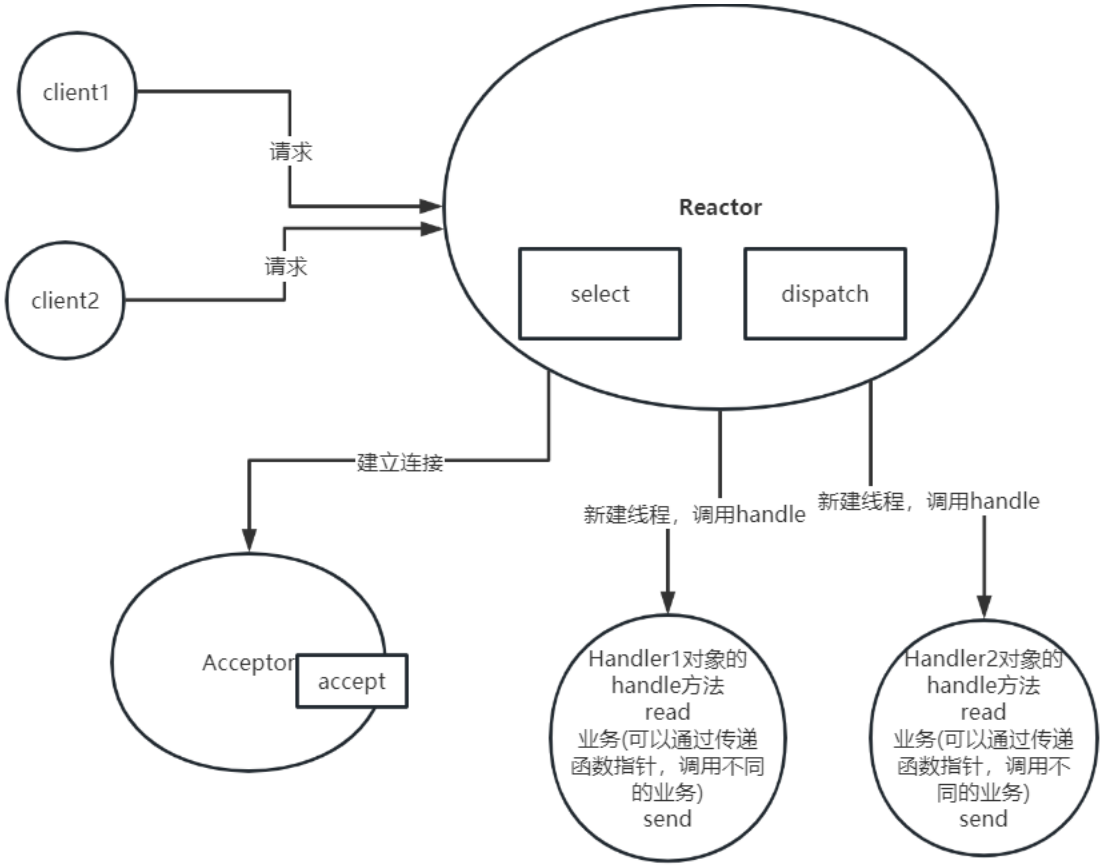
reactor.cpp

```
int main(int argc, char **argv)
{
    while (1)
    {
        /*初始化文件描述符号到集合*/
        FD_ZERO(&client_fdset);
        /*加入服务器描述符*/
        //初步添加进去，我要监听的
        FD_SET(serverfd, &client_fdset);
        /*设置超时时间*/
        tv.tv_sec = 30; /*30秒*/
        tv.tv_usec = 0;
        /*把活动的句柄加入到文件描述符中*/
        for (int i = 0; i < 5; ++i)
        {
            /*程序中Listen中参数设为5,故i必须小于5*/
            if (client_sockfd[i] != 0)
            {
                // 不等于0才放进去进行监听。
                //初步添加进去，我要监听的
                FD_SET(client_sockfd[i], &client_fdset);
            }
        }
        /*printf("put sockfd in fdset!\n");*/
        /*select函数*/
        // 阻塞监听clientz_fdset这个文件句柄集合
        ret = select(maxsock + 1, &client_fdset, NULL, NULL, &tv);
        //返回后，client只存储了真正响应了的文件句柄
        if (ret < 0)
        {
            perror("select error!\n");
            break;
        }
        else if (ret == 0)
        {
            printf("timeout!\n");
            continue;
        }
        /*轮询各个文件描述符*/
        // 必须循环完，因为ret只知道是几个，但是不知道具体是哪几个有响应。
        for (int i = 0; i < conn_amount; ++i)
        {
            /*FD_ISSET检查client_sockfd是否可读写，>0可读写*/
            // 如果存在里面，那么就说明有响应码？确实是
            if (FD_ISSET(client_sockfd[i], &client_fdset))
            {
                handler[i].handle();
            }
        } // 这里是for结束了

        /*检查是否有新的连接，如果收，接收连接，加入到client_sockfd中*/
        //如果输入dd后，它会立刻返回，相当于就是serverfd没有响应就不会存在于client_fdset里面了，这里判断就不会成功，就不会进去了
        if (FD_ISSET(serverfd, &client_fdset))
        {
            acceptor.handle();
```

```
        }       // 这里对应新连接
    }           // 这里是while结束了


}
```

其实就是最基础的select,poll,epoll，只不过使用的面向对象，把代码解耦合了下，原先是全部冗余在一起。

单 Reactor 单进程下的多线程



- Reactor 对象通过 select （IO 多路复用接口） 监听事件，收到事件后通过 dispatch 进行分发，具体分发给 Acceptor 对象还是 Handler 对象，还要看收到的事件类型；
- 如果是连接建立的事件，则交由 Acceptor 对象进行处理，Acceptor 对象会通过 accept 方法 获取连接，并创建一个 Handler 对象来处理后续的响应事件；
- 如果不是连接建立事件， 则创建新的线程并调用Handle对象的handle方法

单 Reator 多线程的方案优势在于**能够充分利用多核 CPU 的性能**

伪代码

```
int main(int argc, char **argv)
{
    while (1)
    {
        /*初始化文件描述符号到集合*/
        FD_ZERO(&client_fdset);
        /*加入服务器描述符*/
        //初步添加进去，我要监听的
```

```
        FD_SET(serverfd, &client_fdset);
        /*设置超时时间*/
        tv.tv_sec = 30; /*30秒*/
        tv.tv_usec = 0;
        /*把活动的句柄加入到文件描述符中*/
        for (int i = 0; i < 5; ++i)
        {
            /*程序中Listen中参数设为5,故i必须小于5*/
            if (client_sockfd[i] != 0)
            {
                // 不等于0才放进去进行监听。
                //初步添加进去，我要监听的
                FD_SET(client_sockfd[i], &client_fdset);
            }
        }
        /*printf("put sockfd in fdset!\n");*/
        /*select函数*/
        // 阻塞监听clientz_fdset这个文件句柄集合
        ret = select(maxsock + 1, &client_fdset, NULL, NULL, &tv);
        //返回后，client只存储了真正响应了的文件句柄
        if (ret < 0)
        {
            perror("select error!\n");
            break;
        }
        else if (ret == 0)
        {
            printf("timeout!\n");
            continue;
        }
        /*轮询各个文件描述符*/
        // 必须循环完，因为ret只知道是几个，但是不知道具体是哪几个有响应。
        for (int i = 0; i < conn_amount; ++i)
        {
            /*FD_ISSET检查client_sockfd是否可读写，>0可读写*/
            // 如果存在里面，那么就说明有响应码？确实是
            if (FD_ISSET(client_sockfd[i], &client_fdset))
            {
                new thread(handler[i].handle(function)...);
                //更好的方式
                threadPool.execute(handler[i].handle(function)...);
            }
        } // 这里是for结束了

        /*检查是否有新的连接，如果收，接收连接，加入到client_sockfd中*/
        //如果输入dd后，它会立刻返回，相当于就是serverfd没有响应就不会存在于client_fdset里面了，这里判断就不会成功，就不会进去了
        if (FD_ISSET(serverfd, &client_fdset))
        {
            //也可以不创建线程
            acceptor.accept();
            //创建线程
            new thread(acceptor.handle());
            //更好的方式
            threadPool.execute(acceptor.handle);
        }     // 这里对应新连接
    }         // 这里是while结束了
```

```
}
```

这里创建线程完全不同于前面的创建线程。

这里是必须做事的时候了（描述符已经响应了），所以创建线程并不浪费。（利用线程池）

而前面是完全可能只是挂起在浪费线程。

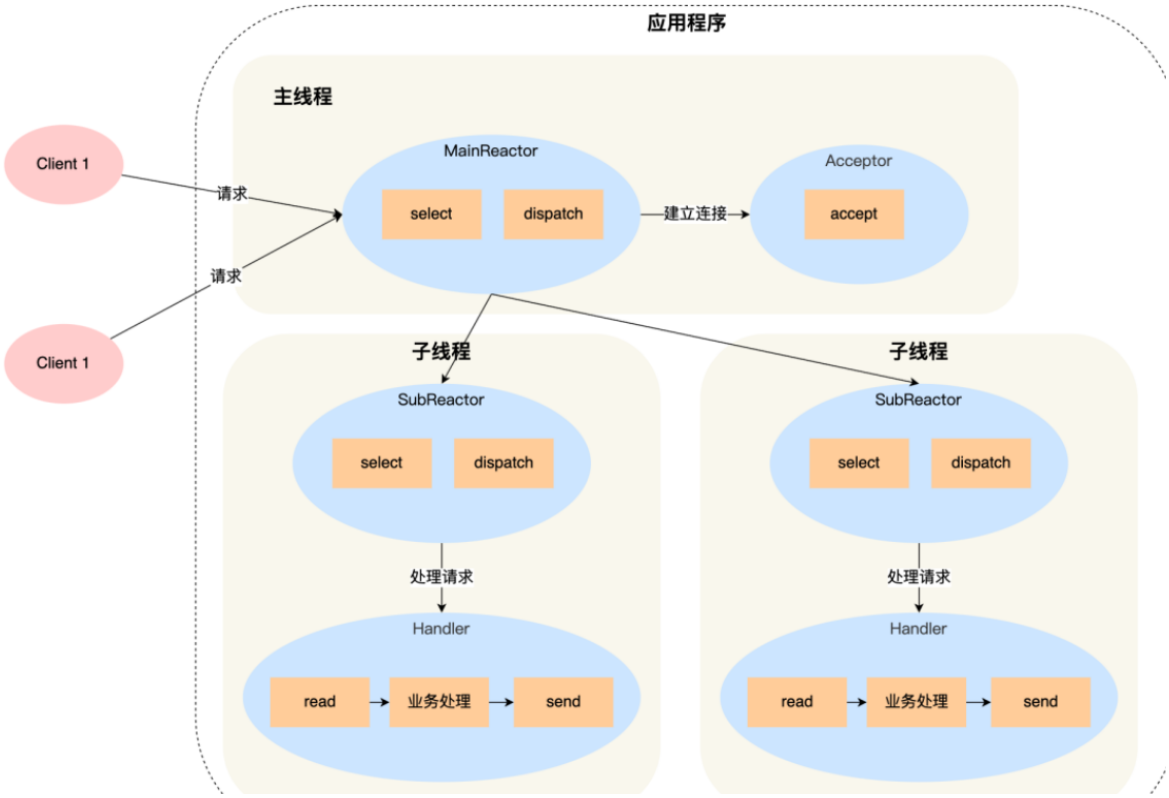
所以这个模型可以理解为基础的IO多路复用+OOP封装+线程池

多 Reactor 单进程下的多线程

「单 Reactor」的模式还有个问题，**因为一个 Reactor 对象承担所有事件（新建连接事件和业务处理事件）的监听和响应，而且只在主线程中运行，在面对瞬间高并发的场景时，容易成为性能的瓶颈的地方。**

```
int main(int argc, char **argv)
{
    while (1)
    {
        /*初始化文件描述符号到集合*/
        FD_ZERO(&client_fdset);
        /*加入服务器描述符*/
        //初步添加进去，我要监听的
        FD_SET(serverfd, &client_fdset);
        /*设置超时时间*/
        tv.tv_sec = 30; /*30秒*/
        tv.tv_usec = 0;
        /*把活动的句柄加入到文件描述符中*/
        for (int i = 0; i < 5; ++i)
        {
            /*程序中Listen中参数设为5,故i必须小于5*/
            if (client_sockfd[i] != 0)
            {
                // 不等于0才放进去进行监听。
                //初步添加进去，我要监听的
                FD_SET(client_sockfd[i], &client_fdset);
            }
        }
        /*printf("put sockfd in fdset!\n");*/
        /*select函数*/
        // 阻塞监听clientz_fdset这个文件句柄集合
        ret = select(maxsock + 1, &client_fdset, NULL, NULL, &tv);
        //返回后，client只存储了真正响应了的文件句柄
        if (ret < 0)
        {
            perror("select error!\n");
            break;
        }
        else if (ret == 0)
        {
            printf("timeout!\n");
            continue;
        }
```

```
        /*轮询各个文件描述符*/
        // 必须循环完，因为ret只知道是几个，但是不知道具体是哪几个有响应。
        for (int i = 0; i < conn_amount; ++i)
        {
            /*FD_ISSET检查client_sockfd是否可读写，>0可读写*/
            // 如果存在里面，那么就说明有响应码？确实是
            if (FD_ISSET(client_sockfd[i], &client_fdset))
            {
                new thread(handler[i].handle(function)...);
                //更好的方式
                threadPool.execute(handler[i].handle(function)...);
            }
        } // 这里是for结束了

        /*检查是否有新的连接，如果收，接收连接，加入到client_sockfd中*/
        //如果输入dd后，它会立刻返回，相当于就是serverfd没有响应就不会存在于client_fdset里面了，这里判断就不会成功，就不会进去了
        if (FD_ISSET(serverfd, &client_fdset))
        {
            //也可以不创建线程
            acceptor.accept();
            //创建线程
            new thread(acceptor.handle());
            //更好的方式
            threadPool.execute(acceptor.handle);
        }     // 这里对应新连接
    }         // 这里是while结束了


}
```



要解决「单 Reactor」的问题，就是将「单 Reactor」实现成「多 Reactor」，这样就产生了第 **多 Reactor 多进程 / 线程**的方案。

方案详细说明如下：

- 主线程中的 MainReactor 对象通过 select 监控**连接建立事件**，收到事件后通过 Acceptor 对象中的 accept 获取连接，将新的连接（**新的sockfd，自己不再监听了**）分配**给某个子线程**；
- 子线程中的 SubReactor 对象将 MainReactor 对象分配的连接加入 select 继续进行监听，并创建一个 Handler 用于处理连接的响应事件。
- 如果有新的事件发生时，SubReactor 对象会调用当前连接对应的 Handler 对象来进行响应。
- Handler 对象通过 read -> 业务处理 -> send 的流程来完成完整的业务流程。

多 Reactor 多线程的方案虽然看起来复杂的，但是实际实现时比单 Reactor 多线程的方案要简单的多，原因如下：

- 主线程和子线程分工明确，主线程只负责接收新连接，子线程负责完成后续的业务处理。
- 主线程和子线程的交互很简单，主线程只需要把新连接传给子线程，子线程无须返回数据，直接就可以在子线程将处理结果发送给客户端。

这张图里面还可以再Handler这里继续进化，即每个描述符响应后都使用线程池里面的线程进行处理。

Proactor

Proactor 是异步网络模型。

阻塞 I/O 好比，你去饭堂吃饭，但是饭堂的菜还没做好，然后你就一直在那里等啊等，等了好长一段时间终于等到饭堂阿姨把菜做出来（数据准备的过程），但是你还得继续等阿姨把菜（内核空间）打到你的饭盒里（用户空间），经历完这两个过程，你才可以离开。

非阻塞 I/O 好比，你去了饭堂，问阿姨菜做好了没有，阿姨告诉你没，你就离开了，过几十分钟，你又来饭堂问阿姨，阿姨说做好了，于是你叫阿姨帮你把菜打到你的饭盒里，第二个过程你是得一直在那里等待的。

异步 I/O 好比，你直接告诉饭堂阿姨**将菜做好并把菜打到饭盒里**后给你打电话，说完你就走了干你的事情去了，整个过程你都不需要任何等待。

异步 I/O 是「内核数据准备好」和「数据从内核态拷贝到用户态」这**两个过程都不用等待。**

现在我们再来理解 Reactor 和 Proactor 的区别，就比较清晰了。

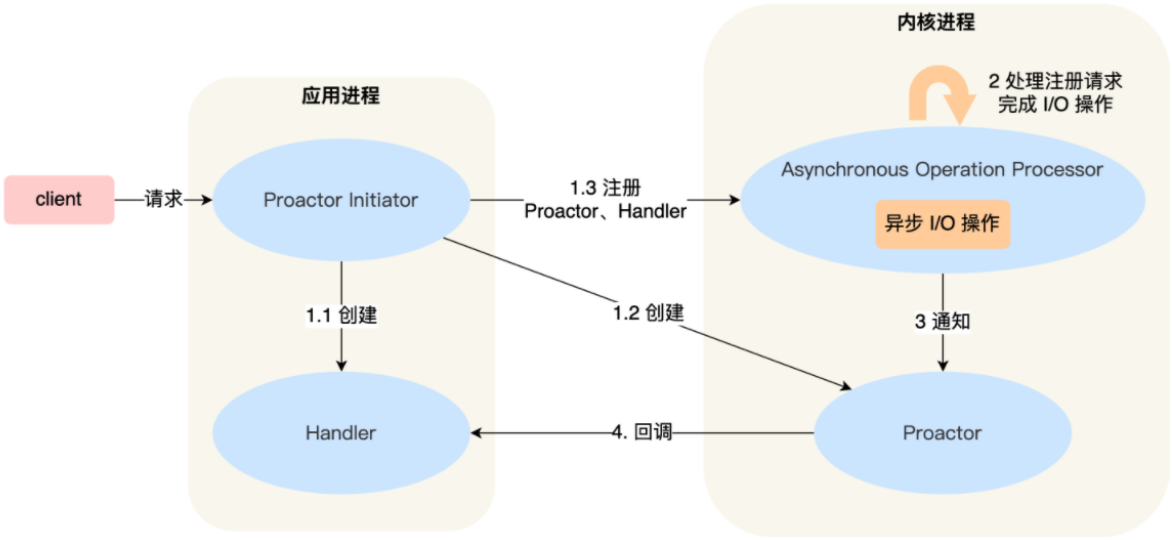
- **Reactor 是非阻塞同步网络模式，感知的是就绪可读写事件（菜做好没）。**在每次感知到有事件发生（比如可读就绪事件）后，就需要应用进程主动调用 read 方法来完成数据的读取，也就是要应用进程主动将 socket 接收缓存中的数据读到应用进程内存中，这个过程是同步的，读取完数据后应用进程才能处理数据。
- **Proactor 是异步网络模式， 感知的是已完成的读写事件（菜放进餐盒里面没）。**在发起异步读写请求时，**需要传入数据缓冲区的地址（用来存放结果数据）等信息**，这样系统内核才可以自动帮我们把数据的读写工作完成，这里的读写工作全程由操作系统来做，并不需要像 Reactor 那样还需要应用进程主动发起 read/write 来读写数据，操作系统完成读写工作后，就会通知应用进程直接处理数据。

因此，**Reactor 可以理解为「来了事件操作系统通知应用进程，让应用进程来处理」**，而 **Proactor 可以理解为「来了事件操作系统来处理，处理完再通知应用进程」。**这里的「事件」就是有新连接、有数据可读、有数据可写的这些 I/O 事件这里的「处理」包含从驱动读取到内核以及从内核读取到用户空间。

相同点：无论是 Reactor，还是 Proactor，都是一种基于「事件分发」的网络编程模式，

区别在于 **Reactor 模式是基于「待完成」的 I/O 事件，而 Proactor 模式则是基于「已完成」的 I/O 事件。**

接下来，一起看看 Proactor 模式的示意图：



介绍一下 Proactor 模式的工作流程：

- Proactor Initiator（Proactor发起者） 负责创建 Proactor（等待通知机制） 和 Handler 对象（数据彻底就绪后的操作），并将 Proactor 和 Handler 都通过 Asynchronous Operation Processor（异步操作处理机） 注册到内核（即告知内核最终要把哪些数据读出到哪里）；
- Asynchronous Operation Processor（异步操作处理机） 负责处理注册请求，并处理 I/O 操作；
- Asynchronous Operation Processor 彻底完成 I/O 操作后通知 Proactor；
- Proactor 根据不同的事件类型回调不同的 Handler 进行业务处理；
- Handler 完成业务处理；