网络 IO 管理教程

网络 IO，会涉及到两个系统对象，一个是用户空间调用 IO 的进程或者线程，另一个是内核空间的内核系统，比如发生 IO 操作 **read** 时，它会经历两个阶段：

1. 等待数据准备就绪
2. 将数据从内核拷贝到进程或者线程中。

因为在以上两个阶段上各有不同的情况，所以出现了多种网络 IO 模型

服务器模型 Reactor 与 Proactor

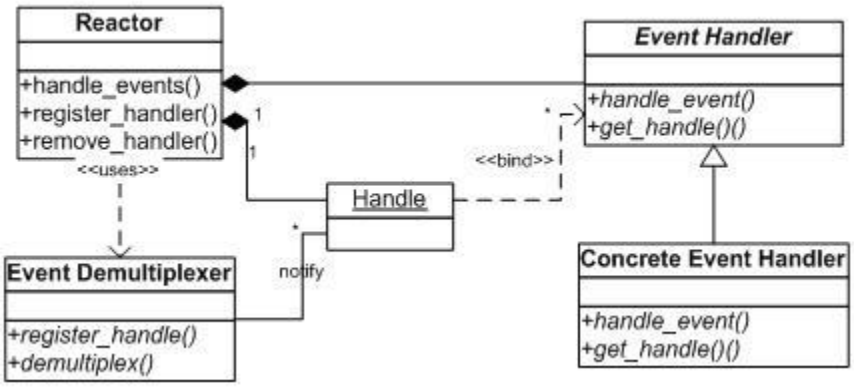
对高并发编程，网络连接上的消息处理，可以分为两个阶段：等待消息准备好、消息处理。当使用默认的阻塞套接字时（例如上面提到的 1 个线程捆绑处理 1 个连接），往往是把这两个阶段合而为一，这样操作套接字的代码所在的线程就得睡眠来等待消息准备好，这导致了高并发下线程会频繁的睡眠、唤醒，从而影响了 CPU 的使用效率。

高并发编程方法当然就是把两个阶段分开处理。即，等待消息准备好的代码段，与处理消息的代码段是分离的。当然，这也要求套接字必须是非阻塞的，否则，处理消息的代码段很容易导致条件不满足时，所在线程又进入了睡眠等待阶段。那么问题来了，等待消息准备好这个阶段怎么实现？它毕竟还是等待，这意味着线程还是要睡眠的！解决办法就是，线程主动查询，或者让 1 个线程为所有连接而等待！这就是 IO 多路复用了。多路复用就是处理等待消息准备好这件事的，但它可以同时处理多个连接！它也可能“等待”，所以它也会导致线程睡眠，然而这不要紧，因为它一对多、它可以监控所有连接。这样，当我们的线程被唤醒执行时，就一定是有一些连接准备好被我们的代码执行了。

作为一个高性能服务器程序通常需要考虑处理三类事件： I/O 事件，定时事件及信号。两种高效的事件处理模型：Reactor 和 Proactor。

Reactor 模型

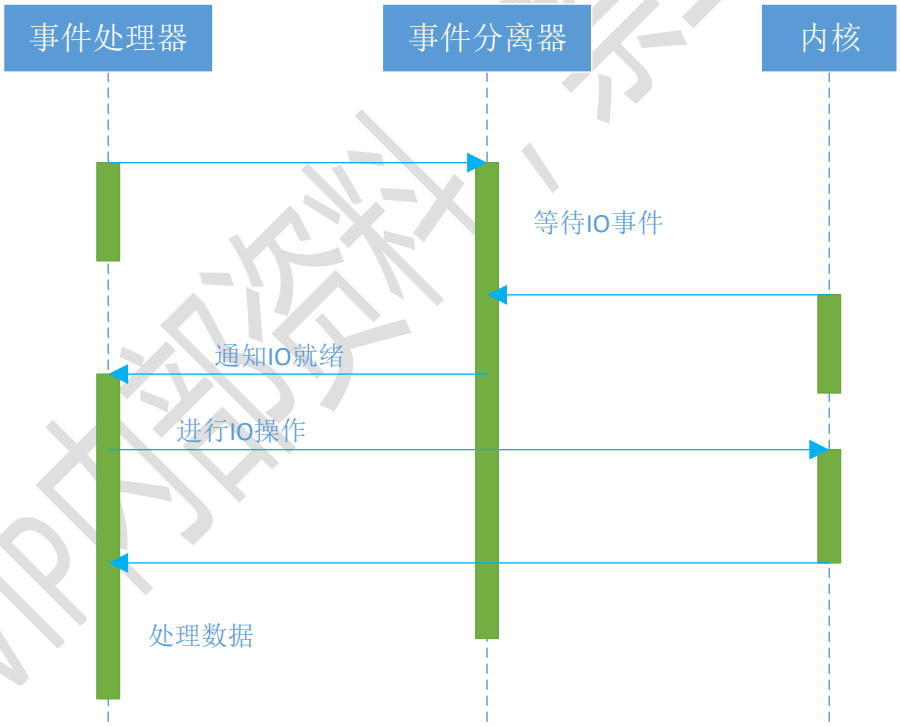
首先来回想一下普通函数调用的机制：程序调用某函数，函数执行，程序等待，函数将结果和控制权返回给程序，程序继续处理。Reactor 释义“反应堆”，是一种事件驱动机制。和普通函数调用的不同之处在于：应用程序不是主动的调用某个 API 完成处理，而是恰恰相反，Reactor 逆置了事件处理流程，应用程序需要提供相应的接口并注册到 Reactor 上，如果相应的时间发生，Reactor 将主动调用应用程序注册的接口，这些接口又称为“回调函数”。



Reactor 模式是处理并发 I/O 比较常见的一种模式，用于同步 I/O，中心思想是将所有要处理的 I/O 事件注册到一个中心 I/O 多路复用器上，同时主线程/进程阻塞在多路复用器上；一旦有 I/O 事件到来或是准备就绪(文件描述符或 socket 可读、写)，多路复用器返回并将事先注册的相应 I/O 事件分发到对应的处理器中。

Reactor 模型有三个重要的组件：

- 多路复用器：由操作系统提供，在 linux 上一般是 select, poll, epoll 等系统调用。
- 事件分发器：将多路复用器中返回的就绪事件分到对应的处理函数中。
- 事件处理器：负责处理特定事件的处理函数。



具体流程如下：

1. 注册读就绪事件和相应的事件处理器；
2. 事件分离器等待事件；
3. 事件到来，激活分离器，分离器调用事件对应的处理器；
4. 事件处理器完成实际的读操作，处理读到的数据，注册新的事件，然后返还控制权。

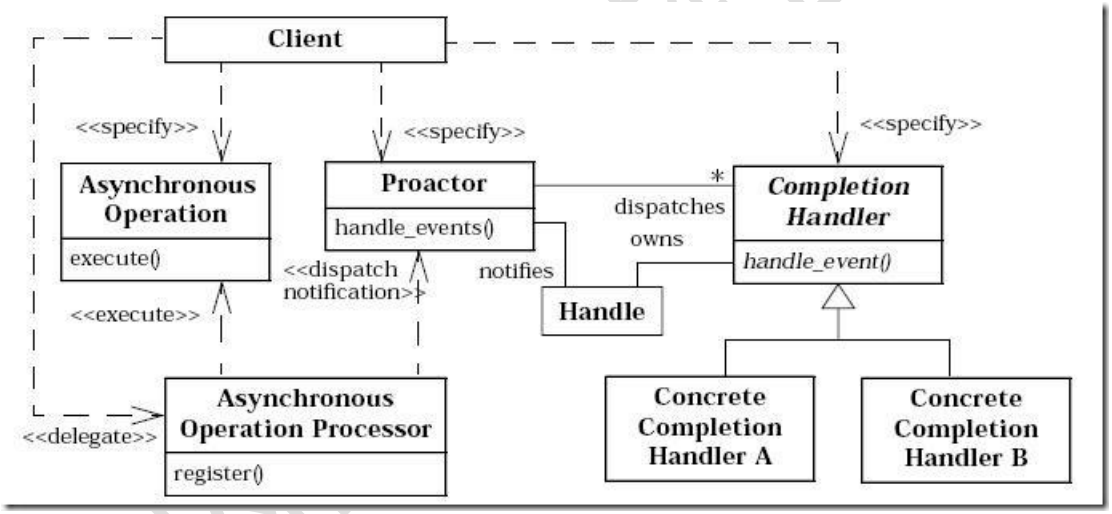
Reactor 模式是编写高性能网络服务器的必备技术之一，它具有如下的优点：

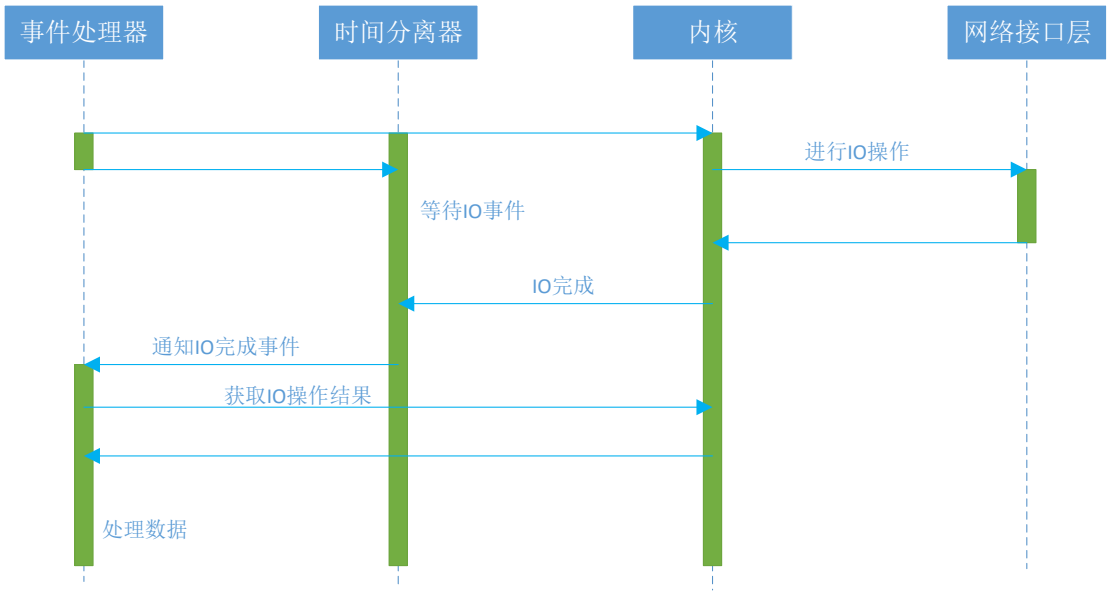
- 响应快，不必为单个同步时间所阻塞，虽然 Reactor 本身依然是同步的；
- 编程相对简单，可以最大程度的避免复杂的多线程及同步问题，并且避免了多线程/进程的切换开销；
- 可扩展性，可以方便的通过增加 Reactor 实例个数来充分利用 CPU 资源；
- 可复用性，reactor 框架本身与具体事件处理逻辑无关，具有很高的复用性；

Reactor 模型开发效率上比起直接使用 IO 复用要高，它通常是单线程的，设计目标是希望单线程使用一颗 CPU 的全部资源，但也有附带优点，即每个事件处理中很多时候可以不考虑共享资源的互斥访问。可是缺点也是明显的，现在的硬件发展，已经不再遵循摩尔定律，CPU 的频率受制于材料的限制不再有大的提升，而改为是从核数的增加上提升能力，当程序需要使用多核资源时，Reactor 模型就会悲剧，为什么呢？

如果程序业务很简单，例如只是简单的访问一些提供了并发访问的服务，就可以直接开启多个反应堆，每个反应堆对应一颗 CPU 核心，这些反应堆上跑的请求互不相关，这是完全可以利用多核的。例如 Nginx 这样的 http 静态服务器。

Proactor 模型





具体流程如下：

1. 处理器发起异步操作，并关注 I/O 完成事件
2. 事件分离器等待操作完成事件
3. 分离器等待过程中，内核并行执行实际的 I/O 操作，并将结果数据存入用户自定义缓冲区，最后通知事件分离器读操作完成
4. I/O 完成后，通过事件分离器呼唤处理器
5. 事件处理器处理用户自定义缓冲区中的数据

从上面的处理流程，我们可以发现 proactor 模型最大的特点就是 Proactor 最大的特点是使用异步 I/O。所有的 I/O 操作都交由系统提供的异步 I/O 接口去执行。工作线程仅仅负责业务逻辑。在 Proactor 中，用户函数启动一个异步的文件操作。同时将这个操作注册到多路复用器上。多路复用器并不关心文件是否可读或可写而是关心这个异步读操作是否完成。异步操作是操作系统完成，用户程序不需要关心。多路复用器等待直到有完成通知到来。当操作系统完成了读文件操作——将读到的数据复制到了用户先前提供的缓冲区之后，通知多路复用器相关操作已完成。多路复用器再调用相应的处理程序，处理数据。

Proactor 增加了编程的复杂度，但给工作线程带来了更高的效率。Proactor 可以在系统态将读写优化，利用 I/O 并行能力，提供一个高性能单线程模型。在 windows 上，由于没有 epoll 这样的机制，因此提供了 IOCP 来支持高并发， 由于操作系统做了较好的优化，windows 较常采用 Proactor 的模型利用完成端口来实现服务器。在 linux 上，在 2.6 内核出现了 aio 接口，但 aio 实际效果并不理想，它的出现，主要是解决 poll 性能不佳的问题，但实际上经过测试，epoll 的性能高于 poll+aio，并且 aio 不能处理 accept，因此 linux 主要还是以 Reactor 模型为主。

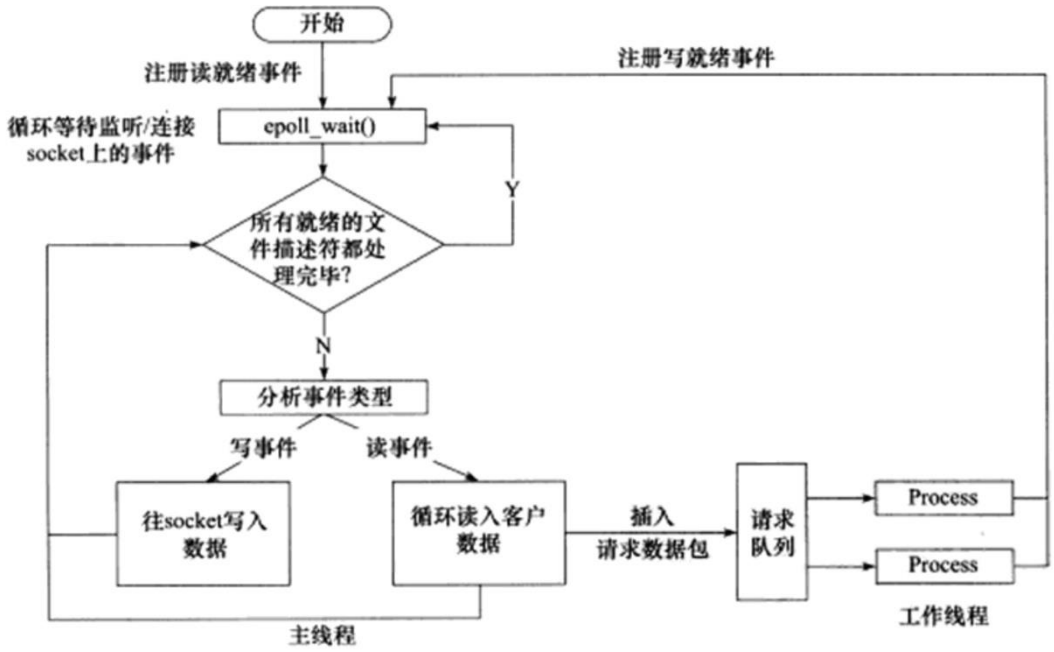
在不使用操作系统提供的异步 I/O 接口的情况下，还可以使用 Reactor 来模拟 Proactor，差别是：使用异步接口可以利用系统提供的读写并行能力，而在模拟的情况下，这需要在用户态实现。具体的做法只需要这样：

1. 注册读事件（同时再提供一段缓冲区）
2. 事件分离器等待可读事件
3. 事件到来，激活分离器，分离器（立即读数据，写缓冲区）调用事件处理器

4. 事件处理器处理数据，删除事件(需要再用异步接口注册)

我们知道，Boost.asio 库采用的即为 Proactor 模型。不过 Boost.asio 库在 Linux 平台采用 epoll 实现的 Reactor 来模拟 Proactor，并且另外开了一个线程来完成读写调度。

同步 I/O 模拟 Proactor 模型



1. 主线程往 epoll 内核事件表中注册 socket 上的读就绪事件。
2. 主线程调用 epoll_wait 等待 socket 上有数据可读。
3. 当 socket 上有数据可读时，epoll_wait 通知主线程。主线程从 socket 循环读取数据，直到没有更多数据可读，然后将读取到的数据封装成一个请求对象并插入请求队列。
4. 睡眠在请求队列上的某个工作线程被唤醒，它获得请求对象并处理客户请求，然后往 epoll 内核事件表中注册 socket 上的写就绪事件。
5. 主线程调用 epoll_wait 等待 socket 可写。
6. 当 socket 可写时，epoll_wait 通知主线程。主线程往 socket 上写入服务器处理客户请求的结果。

两个模式的相同点，都是对某个 IO 事件的事件通知(即告诉某个模块，这个 IO 操作可以进行或已经完成)。在结构上两者也有相同点：demultiplexor 负责提交 IO 操作(异步)、查询设备是否可操作(同步)，然后当条件满足时，就回调注册处理函数。

不同点在于，异步情况下(Proactor)，当回调注册的处理函数时，表示 IO 操作已经完成；同步情况下(Reactor)，回调注册的处理函数时，表示 IO 设备可以进行某个操作(can read or can write)，注册的处理函数这个时候开始提交操作。

Libevent，libev，libuv

libevent :名气最大，应用最广泛，历史悠久的跨平台事件库；

libev :较 libevent 而言，设计更简练，性能更好，但对 Windows 支持不够好；
libuv :开发 node 的过程中需要一个跨平台的事件库，他们首选了 libev，但又要支持 Windows，故重新封装了一套，linux 下用 libev 实现，Windows 下用 IOCP 实现；

优先级

libevent: 激活的事件组织在优先级队列中，各类事件默认的优先级是相同的，可以通过设置事件的优先级使其优先被处理
libev: 也是通过优先级队列来管理激活的时间，也可以设置事件优先级
libuv: 没有优先级概念，按照固定的顺序访问各类事件

事件循环

libevent: event_base 用于管理事件
libev: 激活的事件组织在优先级队列中，各类事件默认的优先级是相同的，
libuv: 可以通 过设置事件的优先级 使其优先被处理

线程安全

event_base 和 loop 都不是线程安全的，一个 event_base 或 loop 实例只能在用户的一个线程内访问（一般是主线程），注册到 event_base 或者 loop 的 event 都是串行访问的，即每个执行过程中，会按照优先级顺序访问已经激活的事件，执行其回调函数。所以在仅使用一个 event_base 或 loop 的情况下，回调函数的执行不存在并行关系