两种高效的服务器设计模型：Reactor和Proactor模型

I/O模型

在文章《unix网络编程》（12）五种I/O模型中提到了五种I/O模型，其中前四种：阻塞模型、非阻塞模型、信号驱动模型、I/O复用模型都是**同步模型**；还有一种是**异步模型**。

Reactor模型

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，如果相应的事件发生，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”。用“好莱坞原则”来形容Reactor再合适不过了：不要打电话给我们，我们会打电话通知你。

Reactor模式与Observer模式在某些方面极为相似：当一个主体发生改变时，所有依属体都得到通知。不过，观察者模式与单个事件源关联，而反应器模式则与多个事件源关联 。

在Reactor模式中，有5个关键的参与者：

- **描述符**（handle）：由操作系统提供的资源，用于识别每一个事件，如Socket描述符、文件描述符、信号的值等。在Linux中，它用一个整数来表示。事件可以来自外部，如来自客户端的连接请求、数据等。事件也可以来自内部，如信号、定时器事件。
- **同步事件多路分离器**（event demultiplexer）：事件的到来是随机的、异步的，无法预知程序何时收到一个客户连接请求或收到一个信号。所以程序要循环等待并处理事件，这就是事件循环。在事件循环中，等待事件一般使用I/O复用技术实现。在linux系统上一般是select、poll、epol_waitl等系统调用，用来等待一个或多个事件的发生。I/O框架库一般将各种I/O复用系统调用封装成统一的接口，称为事件多路分离器。调用者会被阻塞，直到分离器分离的描述符集上有事件发生。
- **事件处理器**（event handler）：I/O框架库提供的事件处理器通常是由一个或多个模板函数组成的接口。这些模板函数描述了和应用程序相关的对某个事件的操作，用户需要继承它来实现自己的事件处理器，即具体事件处理器。因此，事件处理器中的回调函数一般声明为虚函数，以支持用户拓展。
- **具体的事件处理器**（concrete event handler）：是事件处理器接口的实现。它实现了应用程序提供的某个服务。每个具体的事件处理器总和一个描述符相关。它使用描述

符来识别事件、识别应用程序提供的服务。

- **Reactor 管理器**（reactor）：定义了一些接口，用于应用程序控制事件调度，以及应用程序注册、删除事件处理器和相关的描述符。它是事件处理器的调度核心。Reactor管理器使用同步事件分离器来等待事件的发生。一旦事件发生，Reactor管理器先是分离每个事件，然后调度事件处理器，最后调用相关的模 板函数来处理这个事件。

可以看出，**是Reactor管理器**并不是应用程序**负责等待事件、分离事件和调度事件**。Reactor并没有被具体的事件处理器调度，而是**管理器调度具体的事件处理器，由事件处理器对发生的事件作出处理**，这就是Hollywood原则。应用程序要做的仅仅是实现一个具体的事件处理器，然后把它注册到Reactor管理器中。接下来的工作由管理器来完成：如果有相应的事件发生，Reactor会主动调用具体的事件处理器，由事件处理器对发生的事件作出处理。

应用场景

场景：　　长途客车在路途上，有人上车有人下车，但是乘客总是希望能够在客车上得到休息。
传统做法：　每隔一段时间（或每一个站），司机或售票员对每一个乘客询问是否下车。
Reactor做法：汽车是乘客访问的主体（Reactor），乘客上车后，到售票员（acceptor）处登记，之后乘客便可以休息睡觉去了，当到达乘客所要到达的目的地时（指定的事件发生，乘客到了下车地点），售票员将其唤醒即可。

为什么使用Reactor

网络编程为什么要用反应堆？有了I/O复用，有了epoll已经可以使服务器并发几十万连接的同时，维持高TPS了，难道这还不够吗？

答案是，技术层面足够了，但在软件工程层面却是不够的。

程序使用IO复用的难点在哪里呢？

1个请求可能由多次IO处理完成，但相比传统的单线程完整处理请求生命期的方法，IO复用在人的大脑思维中并不自然，因为，程序员编程中，处理请求A的时候，假定A请求必须经过多个IO操作A1-An（两次IO间可能间隔很长时间），每经过一次IO操作，再调用IO复用时，IO复用的调用返回里，非常可能不再有A，而是返回了请求B。即请求A会经常被请求B打断，处理请求B时，又被C打断。这种思维下，编程容易出错。

形象例子：

本部分和下部分内容来自：高性能网络编程6--reactor反应堆与定时器管理

传统编程方法就好像是到了银行营业厅里，每个窗口前排了长队，业务员们在窗口后一个个的解决客户们的请求。一个业务员可以尽情思考着客户A依次提出的问题，例如：
“我要买2万XX理财产品。“

“看清楚了，5万起售。”

“等等，查下我活期余额。”

“余额5万。”

“那就买 5万吧。”

业务员开始录入信息。

”对了，XX理财产品年利率8%？”

“是预期8%，最低无利息保本。“

”早不说，拜拜，我去买余额宝。“

业务员无表情的删着已经录入的信息进行事务回滚。

”下一个！“

用了**IO复用**则是大师业务员开始挑战极限，在超大营业厅里给客户们人手一个牌子，黑压压的客户们都在大厅中，有问题时举牌申请提问，大师目光敏锐点名指定某人提问，该客户迅速得到大师的答复后，要经过一段时间思考，查查自己的银袋子，咨询下LD，才能再次进行下一个提问，直到得到完整的满意答复退出大厅。例如：大师刚指导A填写转帐单的某一项，B又来申请兑换泰铢，给了B兑换单后，C又来办理定转活，然后D与F在争抢有限的圆珠笔时出现了不和谐现象，被大师叫停业务，暂时等待。

这就是**基于事件驱动的IO复用编程**比起传统1线程1请求的方式来，有难度的设计点了，客户们都是上帝，既不能出错，还不能厚此薄彼。
当没有反应堆时，我们可能的设计方法是这样的：大师把每个客户的提问都记录下来，当客户A提问时，首先查阅A之前问过什么做过什么，这叫联系上下文，然后再根据上下文和当前提问查阅有关的银行规章制度，有针对性的回答A，并把回答也记录下来。当圆满回答了A的所有问题后，删除A的所有记录。

在程序中：

某一瞬间，服务器共有10万个并发连接，此时，一次IO复用接口的调用返回了100个活跃的连接等待处理。先根据这100个连接找出其对应的对象，这并不难，epoll的返回连接数据结构里就有这样的指针可以用。接着，循环的处理每一个连接，找出这个对象此刻的上下文状态，再使用read、write这样的网络IO获取此次的操作内容，结合上下文状态查询此时应当选择哪个业务方法处理，调用相应方法完成操作后，若请求结束，则删除对象及其上下文。

这样，我们就**陷入了面向过程编程方法之中**了，在面向应用、快速响应为王的移动互联网时代，这样做早晚得把自己玩死。**我们的主程序需要关注各种不同类型的请求，在不同状态下，对于不同的请求命令选择不同的业务处理方法**。这会导致随着请求类型的增加，请求

状态的增加，请求命令的增加，主程序复杂度快速膨胀，导致维护越来越困难，苦逼的程序员再也不敢轻易接新需求、重构。

反应堆是解决上述软件工程问题的一种途径，它也许并不优雅，开发效率上也不是最高的，但**其执行效率与面向过程的使用IO复用却几乎是等价的**，所以，无论是nginx、memcached、redis等等这些高性能组件的代名词，都义无反顾的一头扎进了反应堆的怀抱中。

反应堆模式可以在软件工程层面，**将事件驱动框架分离出具体业务，将不同类型请求之间用OO的思想分离**。通常，**反应堆不仅使用IO复用处理网络事件驱动，还会实现定时器来处理时间事件的驱动（请求的超时处理或者定时任务的处理）**，就像下面的示意图：

这幅图有5点意思：

（1）处理应用时基于OO思想，**不同的类型的请求处理间是分离的**。例如，A类型请求是用户注册请求，B类型请求是查询用户头像，那么当我们把用户头像新增多种分辨率图片时，更改B类型请求的代码处理逻辑时，完全不涉及A类型请求代码的修改。

（2）**应用处理请求的逻辑，与事件分发框架完全分离**。什么意思呢？即写应用处理时，不用去管何时调用IO复用，不用去管什么调用epoll_wait，去处理它返回的多个socket连接。应用代码中，只关心如何读取、发送socket上的数据，如何处理业务逻辑。事件分发框架有一个抽象的事件接口，所有的应用必须实现抽象的事件接口，通过这种抽象才把应用与框架进行分离。

（3）反应堆上提供注册、移除事件方法，供应用代码使用，而**分发事件方法，通常是循环的调用而已**，是否提供给应用代码调用，还是由框架简单粗暴的直接循环使用，这是框架的自由。

（4）IO多路复用也是一个抽象，它可以是具体的select，也可以是epoll，它们只必须提供采集到某一瞬间所有待监控连接中活跃的连接。

（5）定时器也是由反应堆对象使用，它必须至少提供4个方法，包括添加、删除定时器事件，这该由应用代码调用。最近超时时间是需要的，这会被反应堆对象使用，用于确认select或者epoll_wait执行时的阻塞超时时间，防止IO的等待影响了定时事件的处理。遍历也是由反应堆框架使用，用于处理定时事件。

Reactor的几种模式

参考资料：**Scalable IO in Java**

在web服务中，很多都涉及基本的操作：read request、decode request、process service、encod reply、send reply等。

1 单线程模式

这是最简单的单Reactor单线程模型。Reactor线程是个多面手，负责多路分离套接字，Accept新连接，并分派请求到处理器链中。该模型适用于处理器链中业务处理组件能快速完成的场景。不过这种单线程模型不能充分利用多核资源，所以实际使用的不多。

---

2 多线程模式（单Reactor）

该模型在事件处理器（Handler）链部分采用了多线程（线程池），也是后端程序常用的模型。

---

3 多线程模式（多个Reactor）

比起第二种模型，它是将Reactor分成两部分，mainReactor负责监听并accept新连接，然后将建立的socket通过多路复用器（Acceptor）分派给subReactor。subReactor负责多路分离已连接的socket，读写网络数据；业务处理功能，其交给worker线程池完成。通常，subReactor个数上可与CPU个数等同。

Proacotr模型

Proactor是和异步I/O相关的。

在Reactor模式中，事件分离者等待某个事件或者可应用或个操作的状态发生（比如文件描述符可读写，或者是socket可读写），事件分离器就把这个事件传给事先注册的处理器（事件处理函数或者回调函数），由后者来做实际的读写操作。

在Proactor模式中，事件处理者(或者代由事件分离者发起)直接发起一个异步读写操作(相当于请求)，而实际的工作是由操作系统来完成的。发起时，需要提供的参数包括用于存放读到数据的缓存区，读的数据大小，或者用于存放外发数据的缓存区，以及这个请求完后的回调函数等信息。事件分离者得知了这个请求，它默默等待这个请求的完成，然后转发完成事件给相应的事件处理者或者回调。

可以看出两者的区别：Reactor是在事件发生时就通知事先注册的事件（读写由处理函数完成）；Proactor是在事件发生时进行异步I/O（读写由OS完成），待IO完成事件分离器才调度处理器来处理。

举个例子，将有助于理解Reactor与Proactor二者的差异，以读操作为例（类操作类似）。
在Reactor（同步）中实现读：
- 注册读就绪事件和相应的事件处理器
- 事件分离器等待事件
- 事件到来，激活分离器，分离器调用事件对应的处理器。
- 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。
Proactor（异步）中的读：
- 处理器发起异步读操作（注意：操作系统必须支持异步IO）。在这种情况下，处理器无视IO就绪事件，**它关注的是完成事件**。
- 事件分离器等待操作完成事件
- 在分离器等待过程中，操作系统利用并行的内核线程执行实际的读操作，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。
- 事件分离器呼唤处理器。
- 事件处理器处理用户自定义缓冲区中的数据，然后启动一个新的异步操作，并将控制权返回事件分离器。

现行做法

开源C++框架：ACE

开源C++开发框架 ACE 提供了大量平台独立的底层并发支持类(线程、互斥量等)。 同时在更高一层它也提供了独立的几组C++类，用于实现Reactor及Proactor模式。 尽管它们都是平台独立的单元，但他们都提供了不同的接口。ACE Proactor在MS-Windows上无论是性能还在健壮性都更胜一筹，这主要是由于Windows提供了一系列高效的底层异步API。(这段可能过时了点吧) 不幸的是，**并不是所有操作系统都为底层异步提供健壮的支持**。举例来说， 许多Unix系统就有麻烦。因此， **ACE Reactor可能是Unix系统上更合适的解决方案**。正因为系统底层的支持力度不一，为了在各系统上有更好的性能，开发者不得不维护独立的好几份代码: 为Windows准备的ACE Proactor以及为Unix系列提供的ACE Reactor。真正的异步模式需要操作系统级别的支持。由于事件处理者及操作系统交互的差异，为Reactor和Proactor设计一种通用统一的外部接口是非常困难的。这也是设计通行开发框架的难点所在。

ACE是一个大型的中间件产品，代码20万行左右，过于宏大，一堆的设计模式，架构了一层又一层，使用的时候，要根据情况，看从那一层来进行使用。支持跨平台。
设计模式 ：ACE主要应用了Reactor,Proactor等；
层次架构 ：ACE底层是C风格的OS适配层，上一层基于C++的wrap类，再上一层是一些框架 (Accpetor,Connector,Reactor,Proactor等)，最上一层是框架上服务；
可移植性 ：ACE支持多种平台，可移植性不存在问题，据说socket编程在linux下有不少bugs；
事件分派处理 ：ACE主要是注册handler类，当事件分派时，调用其handler的虚挂勾函数。实现 ACE_Handler/ACE_Svc_Handler/ACE_Event_handler等类的虚函数；
涉及范围 ：ACE包含了日志，IPC,线程池，共享内存，配置服务，递归锁，定时器等；
线程调度 ：ACE的Reactor是单线程调度，Proactor支持多线程调度；

发布方式 ：ACE是开源免费的，不依赖于第三方库，一般应用使用它时，以动态链接的方式发布动态库；开发难度 ：基于ACE开发应用，对程序员要求比较高，要用好它，必须非常了解其框架。在其框架下开发，往往new出一个对象，不知在什么地方释放好。

C网络库：libevent

libevent是一个C语言写的网络库，官方主要支持的是类linux操作系统，最新的版本添加了对windows的IOCP的支持。在跨平台方面主要通过select模型来进行支持。
设计模式 ：**libevent为Reactor模式**；
层次架构：livevent在不同的操作系统下，做了多路复用模型的抽象，可以选择使用不同的模型，通过事件函数提供服务；
可移植性 ：libevent主要支持linux平台，freebsd平台，其他平台下通过select模型进行支持，效率不是太高；
事件分派处理 ：libevent**基于注册的事件回调函数来实现事件分发**；
涉及范围 ：libevent只提供了简单的网络API的封装，线程池，内存池，递归锁等均需要自己实现；
线程调度 ：libevent的线程调度需要自己来注册不同的事件句柄；
发布方式 ：libevent为开源免费的，一般编译为静态库进行使用；
开发难度 ：基于libevent开发应用，相对容易，具体可以参考memcached这个开源的应用，里面使用了 libevent这个库。

改进方案：模拟异步

在改进方案中，我们**将Reactor原来位于事件处理器内的read/write操作移至分离器**(不妨将这个思路称为“模拟异步”)，以此寻求将Reactor多路同步IO转化为模拟异步IO。

以读操作为例子，**改进过程如下**：
- 注册读就绪事件及其处理器，并为分离器提供数据缓冲区地址，需要读取数据量等信息。
- 分离器等待事件（如在select()上等待）
- 事件到来，激活分离器。分离器执行一个非阻塞读操作（它有完成这个操作所需的全部信息），最后调用对应处理器。
- 事件处理器处理用户自定义缓冲区的数据，注册新的事件（当然同样要给出数据缓冲区地址，需要读取的数据量等信息），最后将控制权返还分离器。

如我们所见，通过对多路IO模式功能结构的改造，可将Reactor转化为Proactor模式。改造前后，模型实际完成的工作量没有增加，只不过参与者间对工作职责稍加调换。没有工作量的改变，自然不会造成性能的削弱。对如下各步骤的比较，可以证明工作量的恒定：
标准/典型的Reactor：
- 步骤1：等待事件到来（Reactor负责）
- 步骤2：将读就绪事件分发给用户定义的处理器（Reactor负责）
- 步骤3：读数据（用户处理器负责）
- 步骤4：处理数据（用户处理器负责）

改进实现的模拟Proactor：
- 步骤1：等待事件到来（Proactor负责）
- 步骤2：得到读就绪事件，执行读数据（现在由Proactor负责）
- 步骤3：将读完成事件分发给用户处理器（Proactor负责）
- 步骤4：处理数据（用户处理器负责）

对于不提供异步IO API的操作系统来说，这种办法可以隐藏socket API的交互细节，从而对外暴露一个完整的异步接口。借此，我们就可以进一步构建完全可移植的，平台无关的，有通用对外接口的解决方案。上述方案已经由Terabit P/L公司（http://www.terabit.com.au/）实现为TProactor。它有两个版本：C++和JAVA的。C++版本采用ACE跨平台底层类开发，为所有平台提供了通用统一的主动式异步接口。

Boost.Asio类库

Boost.Asio类库，其就是**以Proactor这种设计模式来实现**，参见：Proactor（The Boost.Asio library is based on the Proactor pattern. This design note outlines the advantages and disadvantages of this approach.），其设计文档链接：http://asio.sourceforge.net/boost_asio_0_3_7/libs/asio/doc/design/index.html

参考资料

1、Reactor构架模式及框架概述

2、高性能网络编程6--reactor反应堆与定时器管理

3、**Scalable IO in Java**

4、Reactor - An Object Behavioral Pattern for Demultiplexing and Dispatching Handles for Synchronous Events

5、两种高性能I/O设计模式(Reactor/Proactor)的比较

6、Reactor模式及在DSS中的体现

7、高性能I/O设计模式Reactor和Proactor

8、Comparing Two High-Performance I/O Design Patterns