Reactor模型

无论是C++还是Java编写的网络框架，大多数都是基于Reactor模式进行设计和开发，Reactor模式基于事件驱动，特别适合处理海量的I/O事件。

#### 单线程模型

Reactor单线程模型，指的是所有的IO操作都在同一个NIO线程上面完成，NIO线程的职责如下：

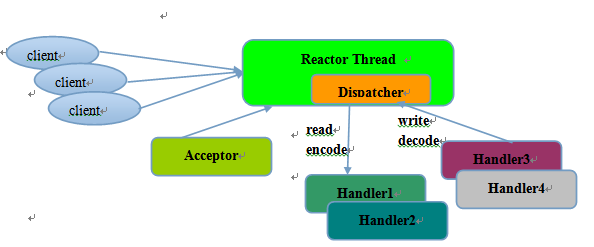
1）作为NIO服务端，接收客户端的TCP连接；

2）作为NIO客户端，向服务端发起TCP连接；

3）读取通信对端的请求或者应答消息；

4）向通信对端发送消息请求或者应答消息。

Reactor单线程模型示意图如下所示：



对于一些小容量应用场景，可以使用单线程模型。但是对于高负载、大并发的应用场景却不合适，主要原因如下：

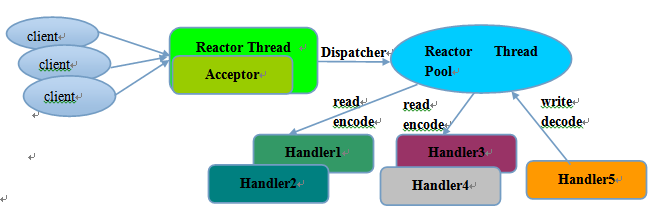
1）一个NIO线程同时处理成百上千的链路，性能上无法支撑，即便NIO线程的CPU负荷达到100%，也无法满足海量消息的编码、解码、读取和发送；

2）当NIO线程负载过重之后，处理速度将变慢，这会导致大量客户端连接超时，超时之后往往会进行重发，这更加重了NIO线程的负载，最终会导致大量消息积压和处理超时，成为系统的性能瓶颈；

3）可靠性问题：一旦NIO线程意外跑飞，或者进入死循环，会导致整个系统通信模块不可用，不能接收和处理外部消息，造成节点故障。

#### 多线程模型

Rector多线程模型与单线程模型最大的区别就是有一组NIO线程处理IO操作，它的原理图如下：



Reactor多线程模型的特点：

1）有专门一个NIO线程-Acceptor线程用于监听服务端，接收客户端的TCP连接请求；

2）网络IO操作-读、写等由一个NIO线程池负责，线程池可以采用标准的JDK线程池实现，它包含一个任务队列和N个可用的线程，由这些NIO线程负责消息的读取、解码、编码和发送；

3）1个NIO线程可以同时处理N条链路，但是1个链路只对应1个NIO线程，防止发生并发操作问题。

在绝大多数场景下，Reactor多线程模型都可以满足性能需求；但是，在极个别特殊场景中，一个NIO线程负责监听和处理所有的客户端连接可能会存在性能问题。例如并发百万客户端连接，或者服务端需要对客户端握手进行安全认证，但是认证本身非常损耗性能。在这类场景下，单独一个Acceptor线程可能会存在性能不足问题，为了解决性能问题，产生了第三种Reactor线程模型-主从Reactor多线程模型。