Reactor 模式

讲到高性能 IO 绕不开 Reactor 模式，它是大多数 IO 相关组件如 Netty、Redis 在使用的 IO 模式，为什么需要这种模式，它是如何设计来解决高性能并发的呢？

背景：

While 循环

最最原始的网络编程思路就是服务器用一个 while 循环，不断监听端口是否有新的套接字连接，如果有，那么就调用一个处理函数处理，类似：

```
while(true){
socket = accept();
handle(socket)
}
```

这种方法的最大问题是无法并发，效率太低，如果当前的请求没有处理完，那么后面的请求只能被阻塞，服务器的吞吐量太低。

多线程

之后，想到了使用多线程，也就是很经典的 connection per thread，每一个连接用一个线程处理，类似：

```
while(true){
socket = accept();
new thread(socket);
}
```

tomcat 服务器的早期版本确实是这样实现的。多线程的方式确实一定程度上极大地提高了服务器的吞吐量，因为之前的请求在 read 阻塞以后，不会影响到后续的请求，因为他们在不同的线程中。这也是为什么通常会讲“一个线程只能对应一个 socket”的原因。最开始对这句话很不理解，线程中创建多个 socket 不行吗？语法上确实可以，但是实际上没有用，每一个 socket 都是阻塞的，所以在一个线程里只能处理一个 socket，就算 accept 了多个也没用，前一个 socket 被阻塞了，后面的是无法被执行到的。

缺点在于资源要求太高，系统中创建线程是需要比较高的系统资源的，如果连接数太高，系统无法承受，而且，线程的反复创建-销毁也需要代价。

线程池

线程池本身可以缓解线程创建-销毁的代价，这样优化确实会好很多，不过还是存在一些问题的，就是线程的粒度太大。每一个线程把一次交互的事情全部做了，包括读取和返回，甚至连接，表面上似乎连接不在线程里，但是如果线程不够，有了新的连接，也无法得到处理，所以，目前的方案线程里可以看成要做三件事，连接，读取和写入。

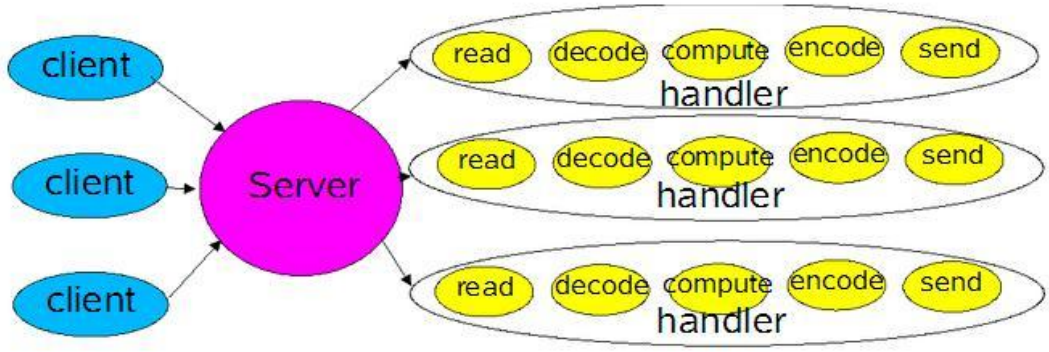
线程同步的粒度太大了，限制了吞吐量。应该把一次连接的操作分为更细的粒度或者过程，这些更细的粒度是更小的线程。整个线程池的数目会翻倍，但是线程更简单，任务更加单一。这其实就是 Reactor 出现的原因，在 Reactor 中，这些被拆分的小线程或者子过程对应的是 handler，每一种 handler 会出处理一种 event。这里会有一个全局的管理者 selector，我们需要把 channel 注册感兴趣的事件，那么这个 selector 就会不断在 channel 上检测是否有该类型的事件发生，如果没有，那么主线程就会被阻塞，否则就会调用相应的事件处理函数即 handler 来处理。典型的事件有连接，读取和写入，当然我们就需要为这些事件分别提供处理器，每一个处理器可以采用线程的方式实现。一个连接来了，显示被读取线程或者 handler 处理了，然后再执行写入，那么之前的读取就可以被后面的请求复用，吞吐量就提高了。

Reactor 模式演化过程

Classic service design

几乎所有的网络连接都会经过读请求内容——》解码——》计算处理——》编码回复——》回复的过程，

对于传统的服务设计，每一个到来的请求，系统都会分配一个线程去处理，这样看似合乎情理，但是，当系统请求量瞬间暴增时，会直接把系统拖垮。因为在高并发情况下，系统创建的线程数量是有限的。传统系统设计如下图所示：



传统的服务代码实现如下所示：

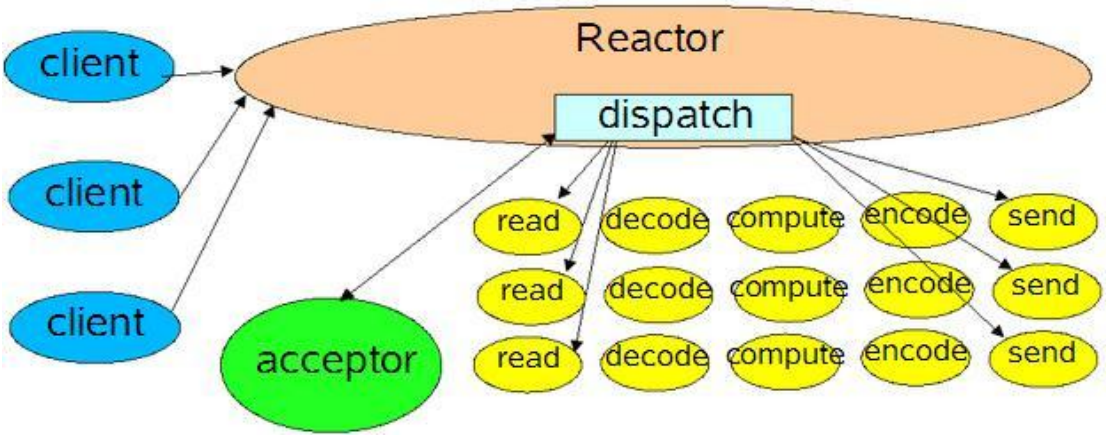
```
class Server implements Runnable {
        public void run() {
            try {
                //创建服务端连接
                ServerSocket ss = new ServerSocket(PORT);
                //不停创建线程处理新的请求
                while (!Thread.interrupted())
                    new Thread(new Handler(ss.accept())).start();
                // or, single-threaded, or a thread pool
            } catch (IOException ex) {
                /* ... */ }
        }
        //处理请求的 handler
        static class Handler implements Runnable {
            final Socket socket;
            Handler(Socket s) {
                socket = s;
            }
            public void run() {
                try {
                    byte[] input = new byte[MAX_INPUT];
                    socket.getInputStream().read(input);
                    byte[] output = process(input);
                    socket.getOutputStream().write(output);
                } catch (IOException ex) {
                    /* ... */ }
```

```
            }
            private byte[] process(byte[] cmd) {
                /* ... */ }
        }
    }
```

显然，传统的一对一的线程处理无法满足新需求的变化。对此，考虑到了线程池的使用，这样就使得线程可以被复用，大大降低创建线程和销毁线程的时间。然而，线程池并不能很好满足高并发线程的需求，当海量请求到来时，线程池中的工作线程达到饱和状态，这时可能就导致请求被抛弃，无法完成客户端的请求。对此，考虑到将一次完整的请求切分成几个小的任务，每一个小任务都是非阻塞的；对于读写操作，使用 NIO 对其进行读写；不同的任务将被分配到与想关联的处理器上进行处理，每个处理器都是通过异步回调机制实现。这样就大大提供系统吞吐量，减少响应时间。这就是下面将要介绍的 Reactor 模式。

这种模型由于 IO 在阻塞时会一直等待，因此在用户负载增加时，性能下降的非常快。
server 导致阻塞的原因：
1、serversocket 的 accept 方法，阻塞等待 client 连接，直到 client 连接成功。
2、线程从 socket inputstream 读入数据，会进入阻塞状态，直到全部数据读完。
3、线程向 socket outputstream 写入数据，会阻塞直到全部数据写完。

改进：采用基于事件驱动的设计，当有事件触发时，才会调用处理器进行数据处理。
单线程版的 Reactor 模式如下图所示。对于客户端的所以请求，都又一个专门的线程去进行处理，这个线程无线循环去监听是否又客户的请求到来，一旦收到客户端的请求，就将其分发给响应的处理器进行处理。



Reactor：负责响应 IO 事件，当检测到一个新的事件，将其发送给相应的 Handler 去处理。
Handler：负责处理非阻塞的行为，标识系统管理的资源；同时将 handler 与事件绑定。
Reactor 为单个线程，需要处理 accept 连接，同时发送请求到处理器中。
由于只有单个线程，所以处理器中的业务需要能够快速处理完。

Reactor 1: Setup

在 Reactor 模式中，我们需要进行一些基本设置，首先需要创建一个 Selector 和一个 ServerSocketChannel ，将监听的端口绑定到 Channel 中，还需要设置 Channel 为非阻塞，并

在 Selector 上注册自己感兴趣的时事件，可以是连接事件，也可以是读写事件。代码如下所示：

```
//定义 reactor，其中包括 Selector 和 ServerSocketChannel
//将 ServerSocketChannel 和事件类型绑定到 Seletor 上，设置　serverSocket 为非阻塞
class Reactor implements Runnable {
    final Selector selector;
    final ServerSocketChannel serverSocket;

    Reactor(int port) throws IOException {
        selector = Selector.open();
        serverSocket = ServerSocketChannel.open();
        serverSocket.socket().bind(new InetSocketAddress(port));
        serverSocket.configureBlocking(false);
        SelectionKey sk = serverSocket.register(selector, SelectionKey.OP_ACCEPT);
        sk.attach(new Acceptor());
    }
    /*
     * Alternatively, use explicit SPI provider: SelectorProvider p =
     * SelectorProvider.provider(); selector = p.openSelector();
     * serverSocket = p.openServerSocketChannel();
     */
}
```

Reactor 2: Dispatch Loop

下面这段代码可以看做是 boss 线程，它负责接收请求并安排给对应的 handle 处理。可以看出，只要当前线程不中断就会一直监听，其中 selector.select()是阻塞的，一旦又请求到来时，就会从 selector 中获取到对应的 SelectionKey ，然后将其下发给后续处理程序(工作线程)进行处理。

```
// class Reactor continued
//无限循环等待网络请求的到来
//其中 selector.select();会阻塞直到有绑定到 selector 的请求类型对应的请求到来，一旦收到事件，处理分发到对应的 handler，并将这个事件移除
public void run() { // normally in a new Thread
    try {
        while (!Thread.interrupted()) {
            selector.select();
            Set selected = selector.selectedKeys();
            Iterator it = selected.iterator();
            while (it.hasNext())
                dispatch((SelectionKey)(it.next());
                selected.clear();
            }
```

```
        } catch (IOException ex) {
            /* ... */ }
        }

    }

    void dispatch(SelectionKey k) {
        Runnable r = (Runnable) (k.attachment());
        if (r != null)
            r.run();
    }
```

Reactor 3: Acceptor

Acceptor 也是一个线程，在其 run 方法中，通过判断 serverSocket.accept()方法来获取 SocketChannel，只要 SocketChannel 不为空，则创建一个 handler 进行相应处理。

```
    // class Reactor continued
    class Acceptor implements Runnable { // inner
        public void run() {
            try {
                SocketChannel c = serverSocket.accept();
                if (c != null)
                    new Handler(selector, c);
            } catch (IOException ex) {
                /* ... */ }
        }
    }
```

Reactor 4: Handler setup

从下方代码可看出，一个 handler 就是一个线程，其中的 SocketChannel 被设置成非阻塞。默认在 Selector 上注册了读事件并绑定到 SocketChannel 上。

```
    final class Handler implements Runnable {
        final SocketChannel socket;
        final SelectionKey sk;
        ByteBuffer input = ByteBuffer.allocate(MAXIN);
        ByteBuffer output = ByteBuffer.allocate(MAXOUT);
        static final int READING = 0, SENDING = 1;
        int state = READING;

        Handler(Selector sel, SocketChannel c) throws IOException {
            socket = c;
            c.configureBlocking(false);
            // Optionally try first read now
```

```
            sk = socket.register(sel, 0);
            sk.attach(this);
            sk.interestOps(SelectionKey.OP_READ);
            sel.wakeup();
        }

        boolean inputIsComplete() {
            /* ... */ }

        boolean outputIsComplete() {
            /* ... */ }

        void process() {
            /* ... */ }
    }
```

Reactor 5: Request handling

针对不同的请求事件进行处理，代码实现如下所示：

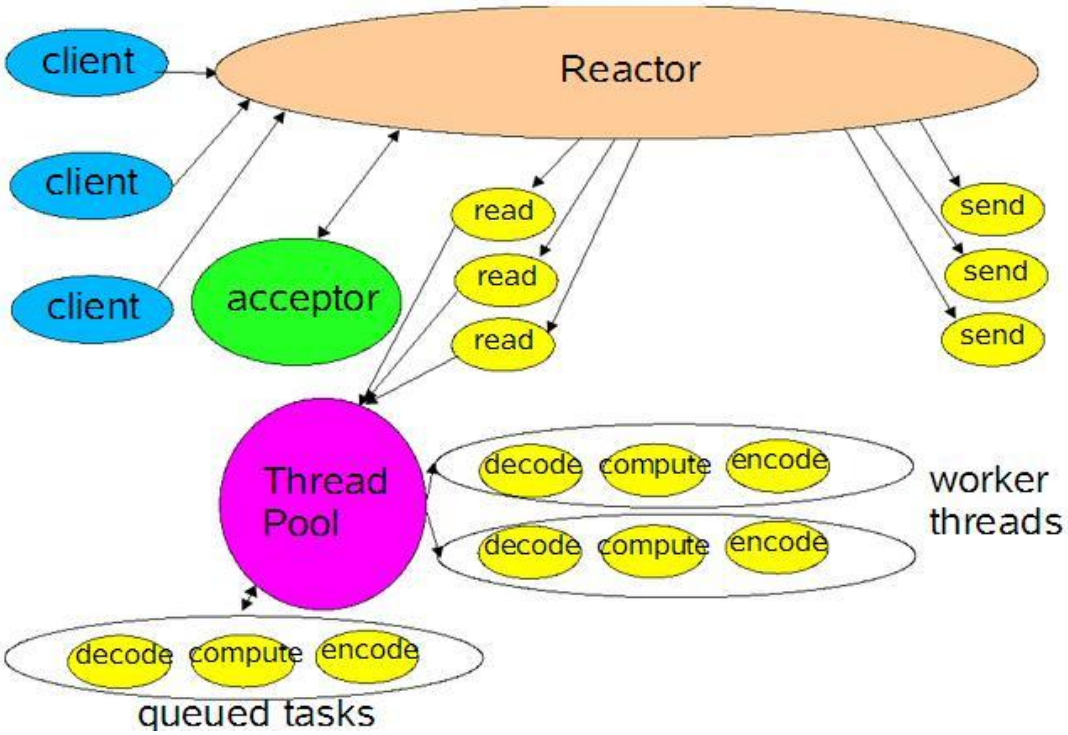
```
    // class Handler continued
    //具体的请求处理，可能是读事件、写事件等
    public void run() {
        try {
            if (state == READING)
                read();
            else if (state == SENDING)
                send();
        } catch (IOException ex) {
            /* ... */ }
    }

    void read() throws IOException {
        socket.read(input);
        if (inputIsComplete()) {
            process();
            state = SENDING;
            // Normally also do first write now
            sk.interestOps(SelectionKey.OP_WRITE);
        }
    }

    void send() throws IOException {
        socket.write(output);
        if (outputIsComplete())
```

```
            sk.cancel();
        }
```

改进：使用多线程处理业务逻辑。



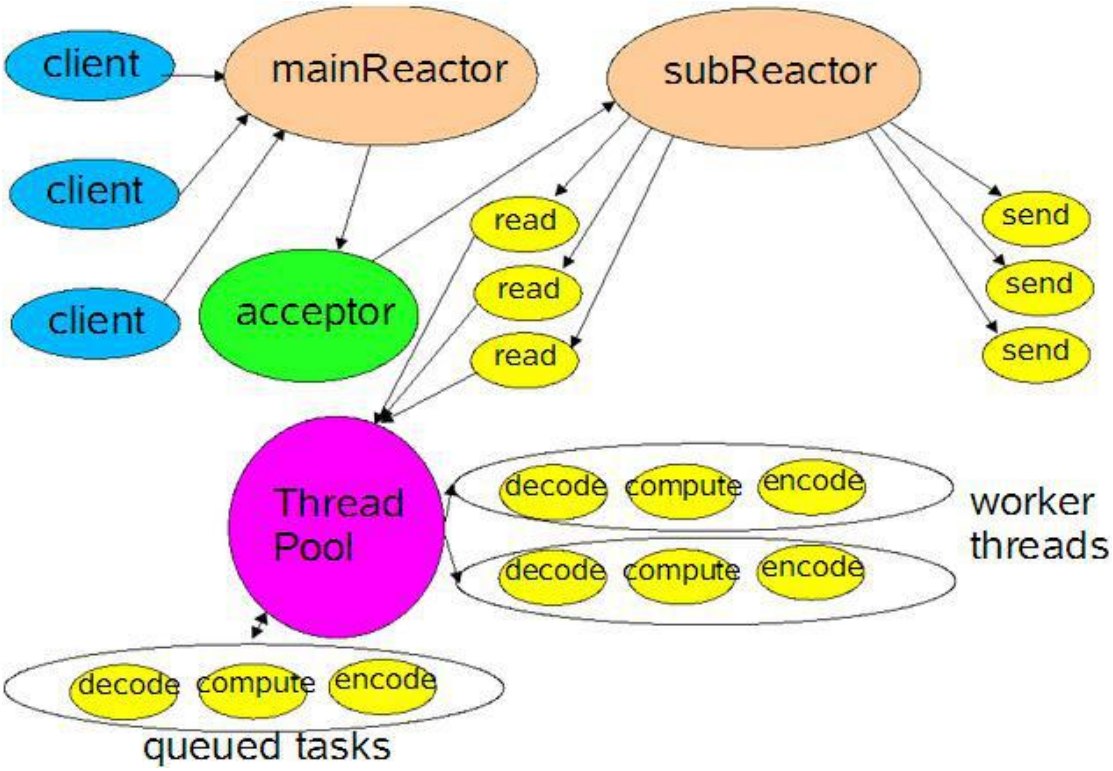
在 handler 中使用线程池来处理任务。代码实现如下所示：

```
    //这里将具体的业务处理线程设置线程池，提供线程复用
    class Handler implements Runnable {
        // uses util.concurrent thread pool
        static PooledExecutor pool = new PooledExecutor(...);
        static final int PROCESSING = 3;
        // ...
        synchronized void read() { // ...
            socket.read(input);
            if (inputIsComplete()) {
                state = PROCESSING;
                //从线程池中指派线程去完成工作
                pool.execute(new Processer());
            }
        }

        synchronized void processAndHandOff() {
            process();
            state = SENDING; // or rebind attachment
            sk.interest(SelectionKey.OP_WRITE);
        }
```

```
        class Processer implements Runnable {
            public void run() {
                processAndHandOff();
            }
        }
    }
```

将处理器的执行放入线程池，多线程进行业务处理。但 Reactor 仍为单个线程。
继续改进：对于多个 CPU 的机器，为充分利用系统资源，将 Reactor 拆分为两部分。
Using Multiple Reactors

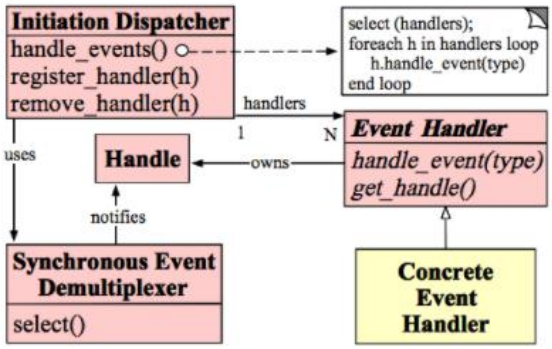


mainReactor 负责监听连接，accept 连接给 subReactor 处理，为什么要单独分一个 Reactor 来处理监听呢？因为像 TCP 这样需要经过 3 次握手才能建立连接，这个建立连接的过程也是要耗时间和资源的，单独分一个 Reactor 来处理，可以提高性能。

概念
reactor 设计模式，是一种基于事件驱动的设计模式。Reactor 框架是 ACE 各个框架中最基础的一个框架，其他框架都或多或少地用到了 Reactor 框架。
在事件驱动的应用中，将一个或多个客户的服务请求分离（demultiplex）和调度（dispatch）给应用程序。在事件驱动的应用中，同步地、有序地处理同时接收的多个服务请求。
reactor 模式与外观模式有点像。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 。当一个主体发生改变时，所有依属体都得到通知。
优点
1）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；

2）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；
3）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；
4）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；
缺点
1）相比传统的简单模型，Reactor 增加了一定的复杂性，因而有一定的门槛，并且不易于调试。
2）Reactor 模式需要底层的 Synchronous Event Demultiplexer 支持，比如 Java 中的 Selector 支持，操作系统的 select 系统调用支持，如果要自己实现 Synchronous Event Demultiplexer 可能不会有那么高效。
3） Reactor 模式在 IO 读写数据时还是在同一个线程中实现的，即使使用多个 Reactor 机制的情况下，那些共享一个 Reactor 的 Channel 如果出现一个长时间的数据读写，会影响这个 Reactor 中其他 Channel 的相应时间，比如在大文件传输时，IO 操作就会影响其他 Client 的相应时间，因而对这种操作，使用传统的 Thread-Per-Connection 或许是一个更好的选择，或则此时使用 Proactor 模式。

架构模式
架构图



Handles ： 表示操作系统管理的资源，我们可以理解为 fd。
Synchronous Event Demultiplexer ： 同步事件分离器，阻塞等待 Handles 中的事件发生。

Initiation Dispatcher ： 初始分派器，作用为添加 Event handler（事件处理器）、删除 Event handler 以及分派事件给 Event handler。也就是说，Synchronous Event Demultiplexer 负责等待新事件发生，事件发生时通知 Initiation Dispatcher，然后 Initiation Dispatcher 调用 event handler 处理事件。
Event Handler ： 事件处理器的接口
Concrete Event Handler ： 事件处理器的实际实现，而且绑定了一个 Handle。因为在实际情况中，我们往往不止一种事件处理器，因此这里将事件处理器接口和实现分开，与 C++、Java 这些高级语言中的多态类似

模块交互
1）我们注册 Concrete Event Handler 到 Initiation Dispatcher 中。
2）Initiation Dispatcher 调用每个 Event Handler 的 get_handle 接口获取其绑定的 Handle。
3）Initiation Dispatcher 调用 handle_events 开始事件处理循环。在这里，Initiation Dispatcher

会将步骤 2 获取的所有 Handle 都收集起来，使用 Synchronous Event Demultiplexer 来等待这些 Handle 的事件发生。

4）当某个（或某几个）Handle 的事件发生时，Synchronous Event Demultiplexer 通知 Initiation Dispatcher。

5）Initiation Dispatcher 根据发生事件的 Handle 找出所对应的 Handler。

6）Initiation Dispatcher 调用 Handler 的 handle_event 方法处理事件