《Scalable IO in Java》 是java.util.concurrent包的作者，大师Doug Lea关于分析与构建可伸缩的高性能IO服务的一篇经典文章，在文章中Doug Lea通过各个角度，循序渐进的梳理了服务开发中的相关问题，以及在解决问题的过程中服务模型的演变与进化，文章中基于Reactor反应器模式的几种服务模型架构，也被Netty、Mina等大多数高性能IO服务框架所采用，因此阅读这篇文章有助于你更深入了解Netty、Mina等服务框架的编程思想与设计模式。

原文连接：<http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf>

## 一、网络服务

在一般的网络或分布式服务等应用程序中，大都具备一些相同的处理流程，例如：

①　读取请求数据；

②　对请求数据进行解码；

③　对数据进行处理；

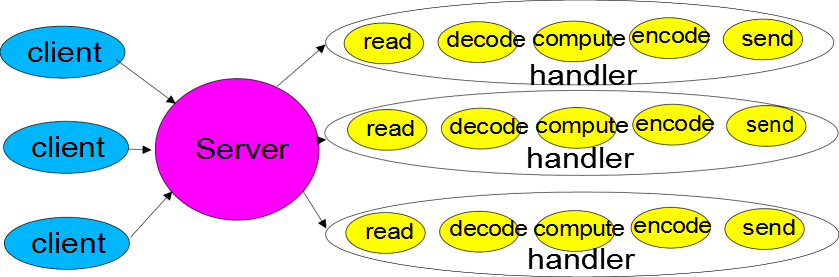
④　对回复数据进行编码；

⑤　发送回复；

当然在实际应用中每一步的运行效率都是不同的，例如其中可能涉及到xml解析、文件传输、web页面的加载、计算服务等不同功能。

### 1、传统的服务设计模式

在一般的网络服务当中都会为每一个连接的处理开启一个新的线程，我们可以看下大致的示意图：



每一个连接的处理都会对应分配一个新的线程，下面我们看一段经典的Server端Socket服务代码：

[IMG_257](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

class Server implements Runnable {

public void run() {

try {

ServerSocket ss = new ServerSocket(PORT);

while (!Thread.interrupted())

new Thread(new Handler(ss.accept())).start();

// or, single-threaded, or a thread pool

} catch (IOException ex) {

/\* ... \*/ }

}

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) {

/\* ... \*/ }

}

}

[IMG_258](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

### 2、构建高性能可伸缩的IO服务

在构建高性能可伸缩IO服务的过程中，我们希望达到以下的目标：

①　能够在海量负载连接情况下优雅降级；

②　能够随着硬件资源的增加，性能持续改进；

③　具备低延迟、高吞吐量、可调节的服务质量等特点；

而分发处理就是实现上述目标的一个最佳方式。

### 3、分发模式

分发模式具有以下几个机制：

①　将一个完整处理过程分解为一个个细小的任务；

②　每个任务执行相关的动作且不产生阻塞；

③　在任务执行状态被触发时才会去执行，例如只在有数据时才会触发读操作；

在一般的服务开发当中，IO事件通常被当做任务执行状态的触发器使用，在hander处理过程中主要针对的也就是IO事件；



java.nio包就很好的实现了上述的机制：

①　非阻塞的读和写

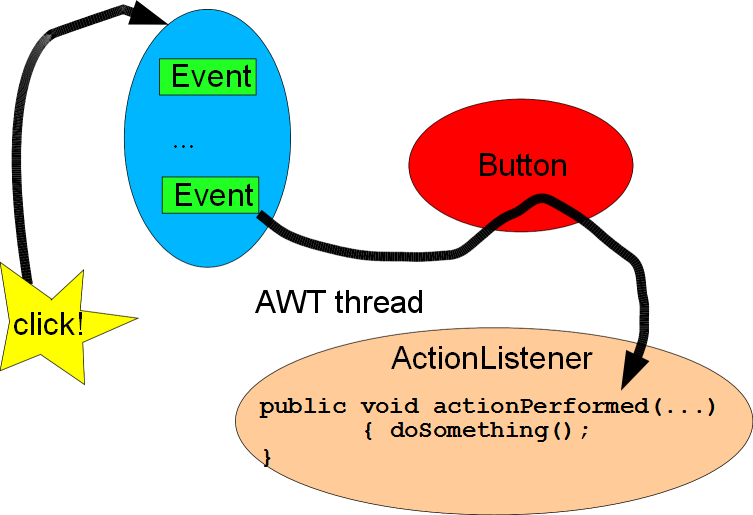
②　通过感知IO事件分发任务的执行

所以结合一系列基于事件驱动模式的设计，给高性能IO服务的架构与设计带来丰富的可扩展性；

## 二、基于事件驱动模式的设计

基于事件驱动的架构设计通常比其他架构模型更加有效，因为可以节省一定的性能资源，事件驱动模式下通常不需要为每一个客户端建立一个线程，这意味这更少的线程开销，更少的上下文切换和更少的锁互斥，但任务的调度可能会慢一些，而且通常实现的复杂度也会增加，相关功能必须分解成简单的非阻塞操作，类似与GUI的事件驱动机制，当然也不可能把所有阻塞都消除掉，特别是GC， page faults(内存缺页中断)等。由于是基于事件驱动的，所以需要跟踪服务的相关状态（因为你需要知道什么时候事件会发生）;

下图是AWT中事件驱动设计的一个简单示意图，可以看到，在不同的架构设计中的基于事件驱动的IO操作使用的基本思路是一致的；



## ****三、Reactor模式****

Reactor也可以称作反应器模式，它有以下几个特点：

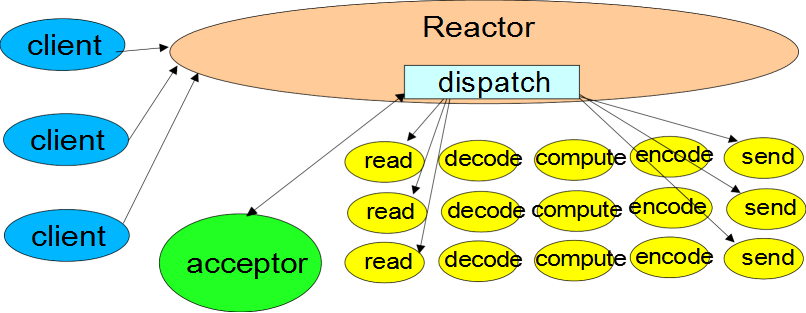
①　Reactor模式中会通过分配适当的handler(处理程序)来响应IO事件，类似与AWT 事件处理线程；

②　每个handler执行非阻塞的操作，类似于AWT ActionListeners 事件监听

③　通过将handler绑定到事件进行管理，类似与AWT addActionListener 添加事件监听；

### 1、单线程模式

下图展示的就是单线程下基本的Reactor设计模式



首先我们明确下java.nio中相关的几个概念：

### ****Channels****

支持非阻塞读写的socket连接；

### ****Buffers****

用于被Channels读写的字节数组对象

### ****Selectors****

用于判断channle发生IO事件的选择器

### ****SelectionKeys****

负责IO事件的状态与绑定

Ok，接下来我们一步步看下基于Reactor模式的服务端设计代码示例：

第一步  Rector线程的初始化

[IMG_262](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

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); //注册accept事件

sk.attach(new Acceptor()); //调用Acceptor()为回调方法 }

public void run() {

try {

while (!Thread.interrupted()) {//循环 selector.select();

Set selected = selector.selectedKeys();

Iterator it = selected.iterator();

while (it.hasNext())

dispatch((SelectionKey)(it.next()); //dispatch分发事件 selected.clear();

}

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

}

void dispatch(SelectionKey k) {

Runnable r = (Runnable)(k.attachment()); //调用SelectionKey绑定的调用对象

if (r != null)

r.run();

}

// Acceptor 连接处理类

class Acceptor implements Runnable { // inner

public void run() {

try {

SocketChannel c = serverSocket.accept();

if (c != null)

new Handler(selector, c);

}

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

}

}

}

[IMG_263](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

第二步 Handler处理类的初始化

[IMG_264](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

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); //将Handler绑定到SelectionKey上 sk.interestOps(SelectionKey.OP\_READ);

sel.wakeup();

}

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

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

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

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();

}

}

[IMG_265](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

下面是基于GoF状态对象模式对Handler类的一个优化实现，不需要再进行状态的判断。

[IMG_266](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

class Handler { // ...

public void run() { // initial state is reader socket.read(input);

if (inputIsComplete()) {

process();

sk.attach(new Sender()); sk.interest(SelectionKey.OP\_WRITE);

sk.selector().wakeup();

}

}

class Sender implements Runnable {

public void run(){ // ... socket.write(output);

if (outputIsComplete()) sk.cancel();

}

}

}

[IMG_267](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

### 2、多线程设计模式

在多处理器场景下，为实现服务的高性能我们可以有目的的采用多线程模式：

  1、增加Worker线程，专门用于处理非IO操作，因为通过上面的程序我们可以看到，反应器线程需要迅速触发处理流程，而如果处理过程也就是process()方法产生阻塞会拖慢反应器线程的性能，所以我们需要把一些非IO操作交给Woker线程来做；

  2、拆分并增加反应器Reactor线程，一方面在压力较大时可以饱和处理IO操作，提高处理能力；另一方面维持多个Reactor线程也可以做负载均衡使用；线程的数量可以根据程序本身是CPU密集型还是IO密集型操作来进行合理的分配；

#### 2.1 多线程模式

Reactor多线程设计模式具备以下几个特点：

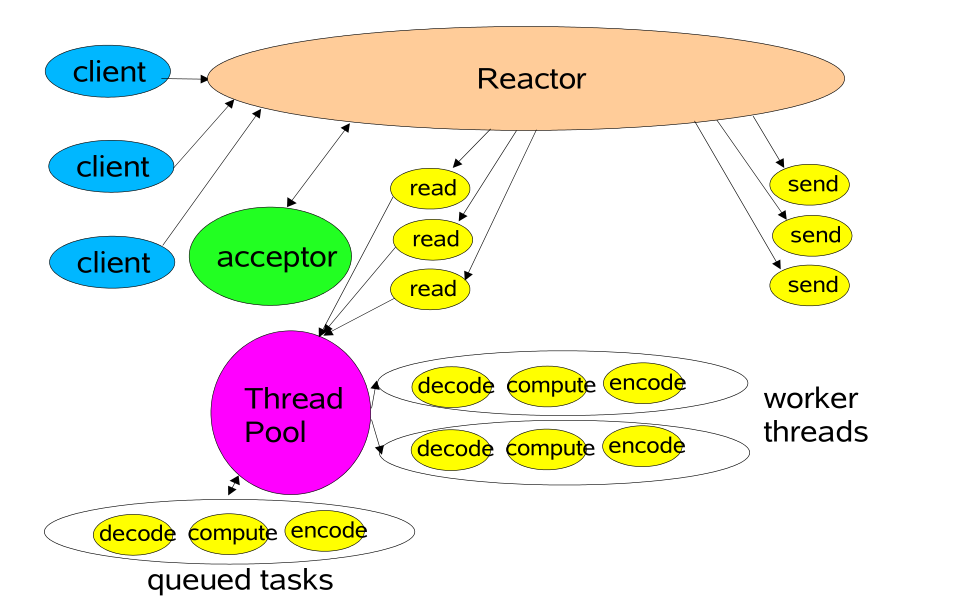
①　通过卸载非IO操作来提升Reactor 线程的处理性能，这类似与POSA2 中Proactor的设计；

②　比将非IO操作重新设计为事件驱动的方式更简单；

③　但是很难与IO重叠处理，最好能在第一时间将所有输入读入缓冲区；（这里我理解的是最好一次性读取缓冲区数据，方便异步非IO操作处理数据）

④　可以通过线程池的方式对线程进行调优与控制，一般情况下需要的线程数量比客户端数量少很多；

下面是Reactor多线程设计模式的一个示意图与示例代码（我们可以看到在这种模式中在Reactor线程的基础上把非IO操作放在了Worker线程中执行）：



[IMG_269](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

class Handler implements Runnable {

// uses util.concurrent thread pool

static PooledExecutor pool = new PooledExecutor(...);//声明线程池

static final int PROCESSING = 3;

// ...

synchronized void read() { // ... socket.read(input);

if (inputIsComplete()) {

state = PROCESSING;

pool.execute(new Processer());//处理程序放在线程池中执行 }

}

synchronized void processAndHandOff() {

process();

state = SENDING; // or rebind attachment sk.interest(SelectionKey.OP\_WRITE);

}

class Processer implements Runnable {

public void run() {

processAndHandOff();

}

}

}

[IMG_270](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

当你把非IO操作放到线程池中运行时，你需要注意以下几点问题：

①　任务之间的协调与控制，每个任务的启动、执行、传递的速度是很快的，不容易协调与控制；

②　每个hander中dispatch的回调与状态控制；

③　不同线程之间缓冲区的线程安全问题；

④　需要任务返回结果时，任务线程等待和唤醒状态间的切换；

为解决上述问题可以使用PooledExecutor线程池框架，这是一个可控的任务线程池，主函数采用execute(Runnable r)，它具备以下功能，可以很好的对池中的线程与任务进行控制与管理：

①　可设置线程池中最大与最小线程数；

②　按需要判断线程的活动状态，及时处理空闲线程；

③　当执行任务数量超过线程池中线程数量时，有一系列的阻塞、限流的策略；

#### ****2.2 基于多个反应器的多线程模式****

这是对上面模式的进一步完善，使用反应器线程池，一方面根据实际情况用于匹配调节CPU处理与IO读写的效率，提高系统资源的利用率，另一方面在静态或动态构造中每个反应器线程都包含对应的Selector,Thread,dispatchloop,下面是一个简单的代码示例与示意图（Netty就是基于这个模式设计的，一个处理Accpet连接的mainReactor线程，多个处理IO事件的subReactor线程）：

[IMG_271](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)

Selector[] selectors; // Selector集合，每一个Selector 对应一个subReactor线程

//mainReactor线程

class Acceptor { // ...

public synchronized void run() {

//...

Socket connection = serverSocket.accept();

if (connection != null)

new Handler(selectors[next], connection);

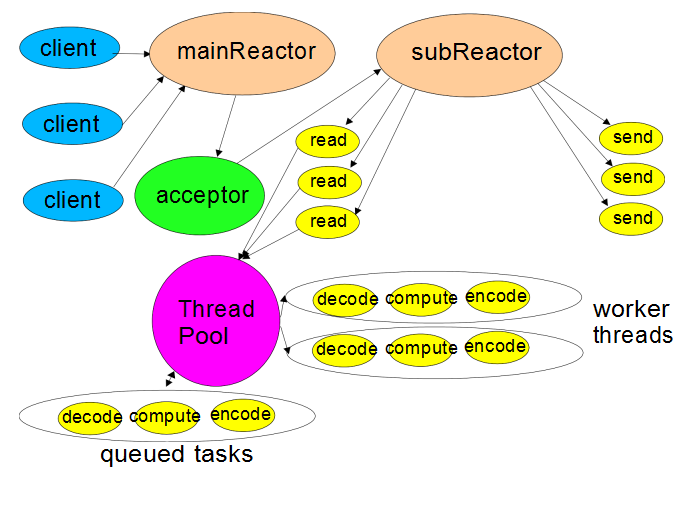
if (++next == selectors.length)

next = 0;

}

}

[IMG_272](https://www.cnblogs.com/dafanjoy/p/javascript:void(0);)



在服务的设计当中，我们还需要注意与java.nio包特性的结合：

一是注意线程安全，每个selectors 对应一个Reactor 线程，并将不同的处理程序绑定到不同的IO事件，在这里特别需要注意线程之间的同步；

二是java nio中文件传输的方式：

①　Memory-mapped files 内存映射文件的方式，通过缓存区访问文件；

②　Direct buffers直接缓冲区的方式，在合适的情况下可以使用零拷贝传输，但同时这会带来初始化与内存释放的问题（需要池化与主动释放）;