

# Scalable IO in Java

Doug Lea

State University of New York at Oswego

dl@cs.oswego.edu

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

译者: [顿悟源码](#)

代码实现: <https://github.com/tonwu/Reactor>

# 大纲

---

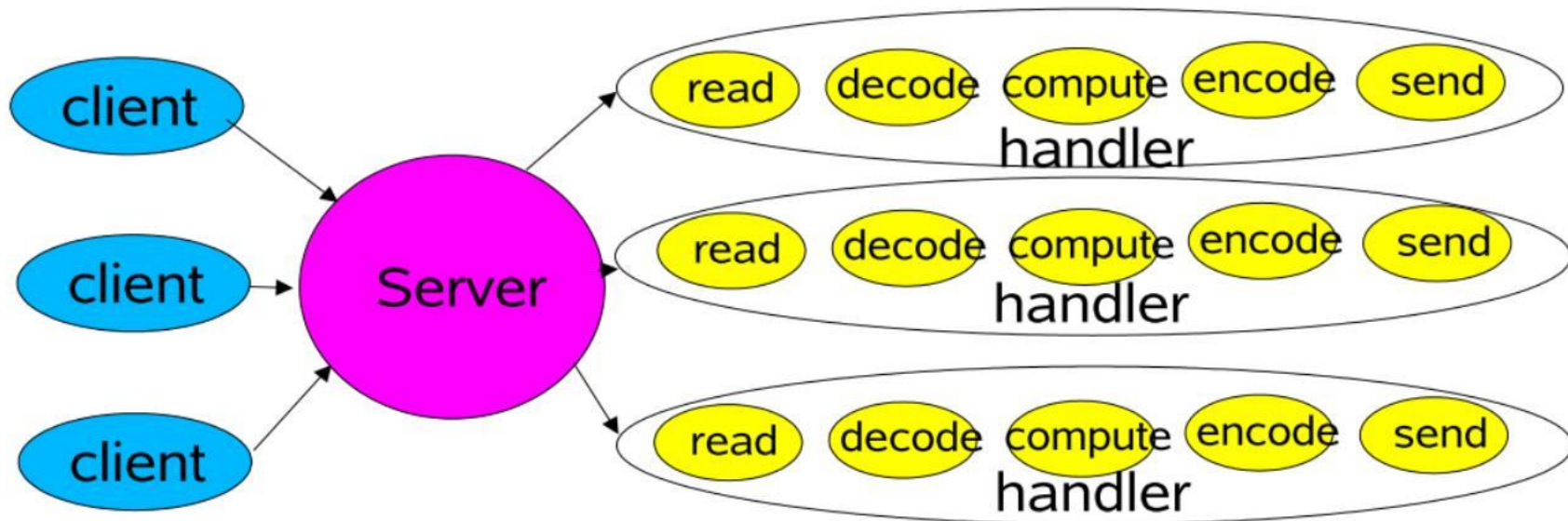
- 可扩展的网络服务
- 事件驱动
- **Reactor** 模式
  - 基本版本
  - 多线程版本
  - 其他变种
- **java.nio** 非阻塞 **IO APIs** 的介绍

# 网络服务

---

- Web 服务，分布式对象等
- 大部分都有相同的基本结构：
  - 读取请求，**Read request**
  - 解析请求，**Decode request**
  - 业务处理，**Process service**
  - 编码响应，**Encode reply**
  - 发送响应，**Send reply**
- 但每一步的本质和开销不同
  - 解析 XML，传输文件，生成 Web 页面，计算服务，...

# 典型的服务端设计



每个 handler 可能在各自的线程中

# 经典 ServerSocket 循环

```
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) { /* ... */ }
    }
}
```

*Note: most exception handling elided from code examples*

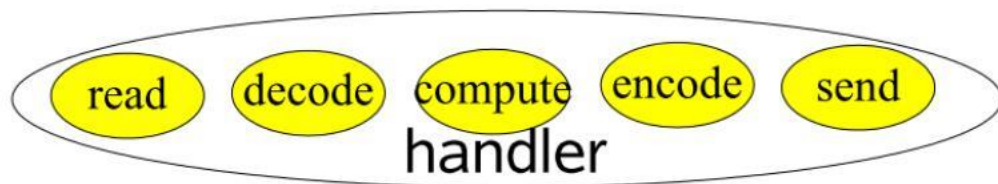
# 可扩展点

---

- 在负载不断增加的情况下，如何优雅的缓解压力（更多的客户端）
- 随着系统资源的增加，如何持续改进（CPU，内存，硬盘，带宽）
- 并且满足可用性和性能目标
  - 低延迟
  - 满足高峰需求
  - 可调节的服务质量
- 分而治之通常是实现任何可伸缩性的目标最佳方法

# 分而治之

- 将处理分为多个小任务
  - 每个任务执行一个非阻塞操作
- 当任务被激活时就执行它
  - 这里，通常由一个 **IO** 事件触发



- **java.nio** 支持这种机制
  - 非阻塞的读和写
  - 分配任务，为感兴趣的 **IO** 事件
- 无限可能的变化
  - 一系列事件驱动的设计

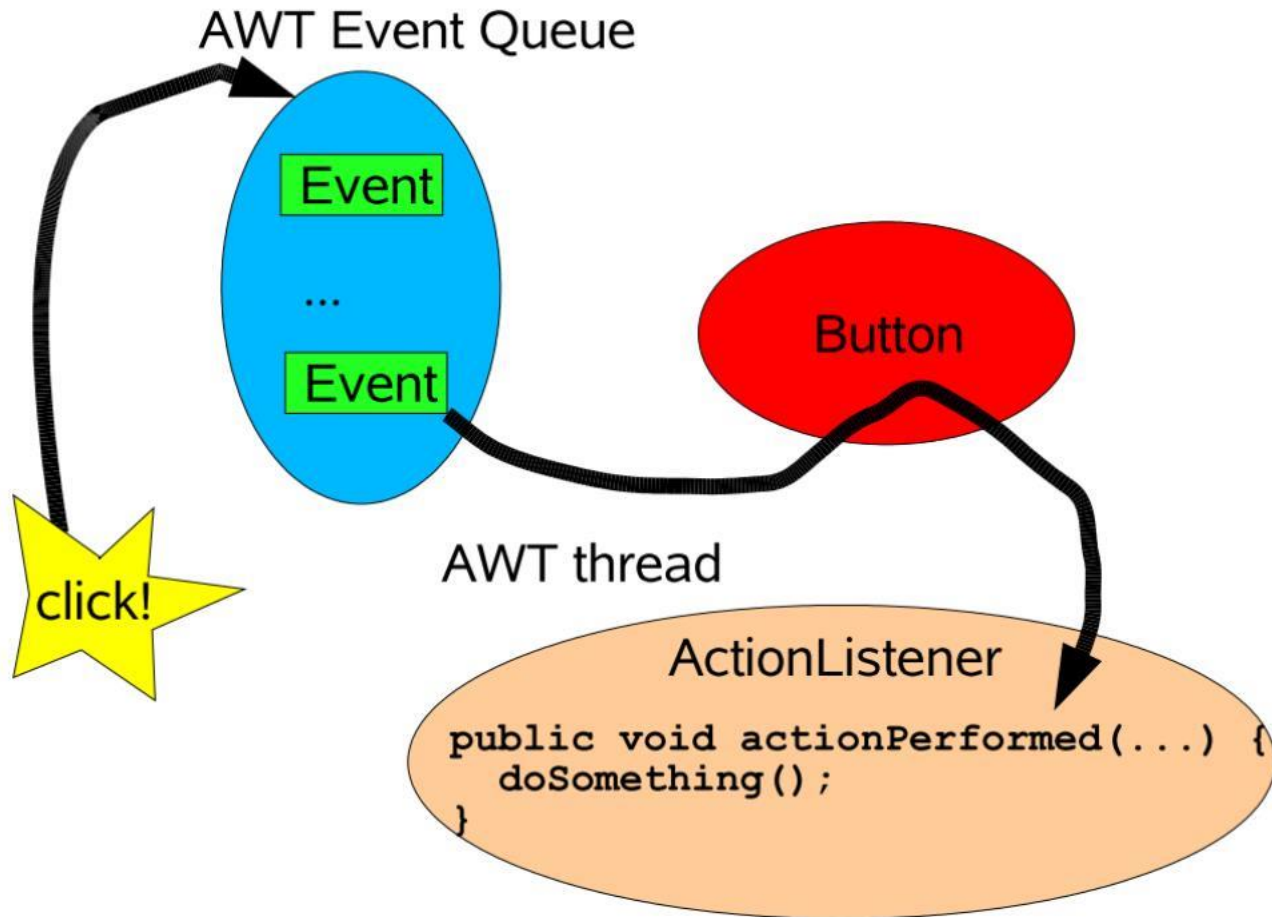
# 事件驱动设计

---

- 通常比其他模型更高效
  - 更少的资源：不需要一客户端一线程
  - 较少的开销：更少的上下文切换，更少的锁定
  - 但分派可能较慢：必须手动将操作绑定到事件
- 通常编程比较困难
  - 必须分解成简单的非阻塞操作
    - 类似于 GUI 事件驱动的操作
    - 不能消除所有阻塞：GC, page faults 等
  - 必须跟踪服务的逻辑状态



# 背景：AWT 中的事件

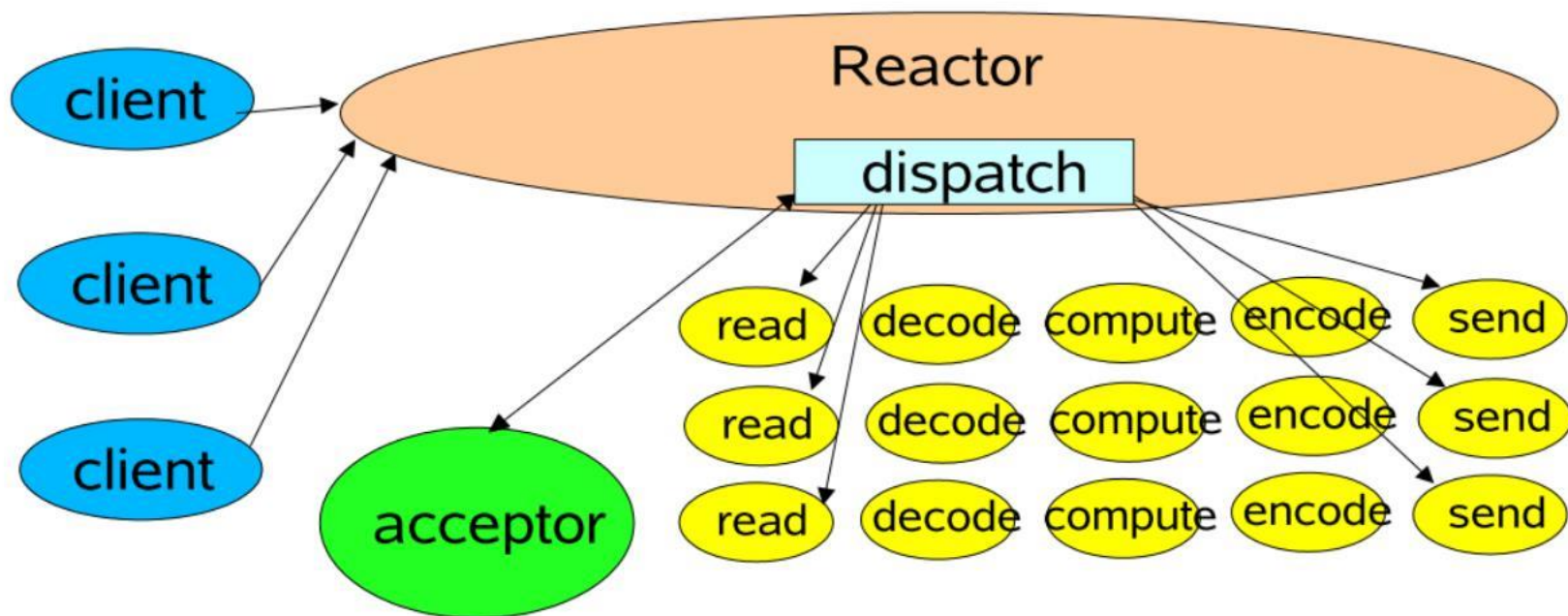


事件驱动 IO 使用类似的想法，但在设计上有所不同

# Reactor 模式

- **Reactor** 通过分派合适的Handler来响应IO事件
  - 类似于 AWT 线程
- **Handlers** 执行非阻塞操作
  - 类似于 AWT 的 ActionListeners
- 通过绑定 Handlers 来管理事件
  - 类似于 AWT 的 addActionListener
- See Schmidt et al, *Pattern-Oriented Software Architecture, Volume 2* (POSA2)  
Also Richard Stevens's networking books,  
Matt Welsh's SEDA framework, etc

# Reactor 基本设计



单线程版本

# java.nio 提供的支持

---

- **Channels**
  - 连接到文件, Socket 等, 支持非阻塞读取
- **Buffers**
  - 类似数组的对象, 可由通道直接读取或写入
- **Selectors**
  - 通知哪组通道有 IO 事件
- **SelectionKeys**
  - 维护 IO 事件的状态和绑定信息

# Reactor 1: Setup

---

```
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

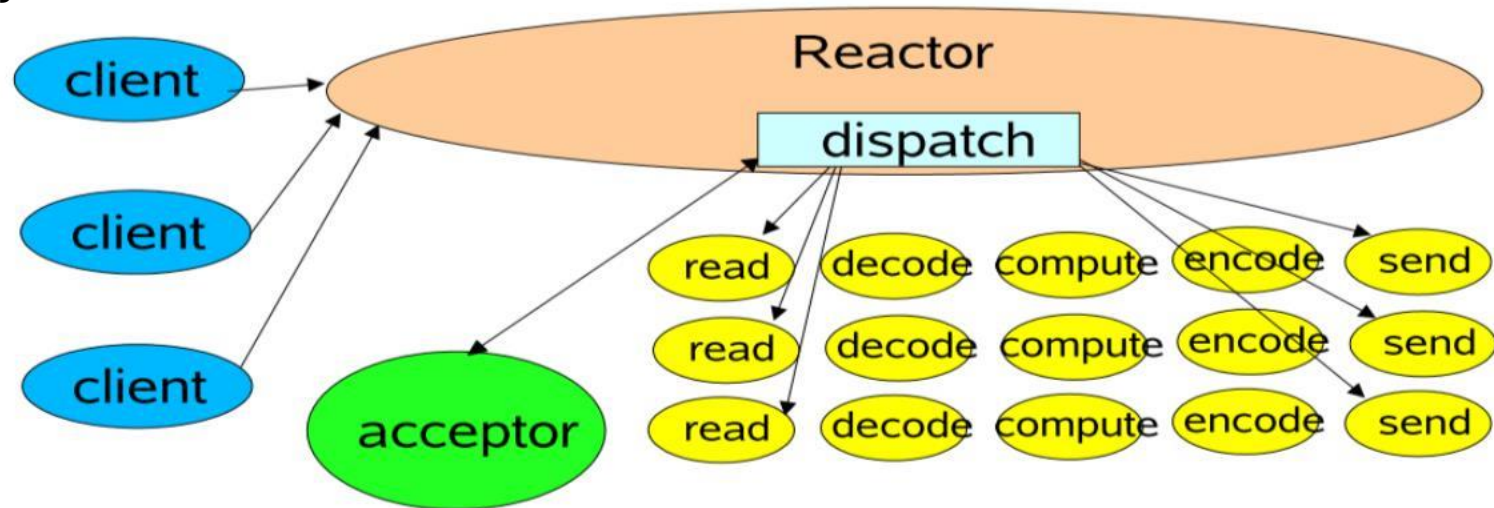
---

```
// class Reactor continued
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

```
// 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

---

```
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

---

```
// class Handler continued
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();
}
}
```

# 一个状态一个 Handler

GoF 状态对象模式的简单使用

重新绑定适当的 Handler 作为附件，附加对象

```
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();
    }
}
}
```

# 多线程设计

---

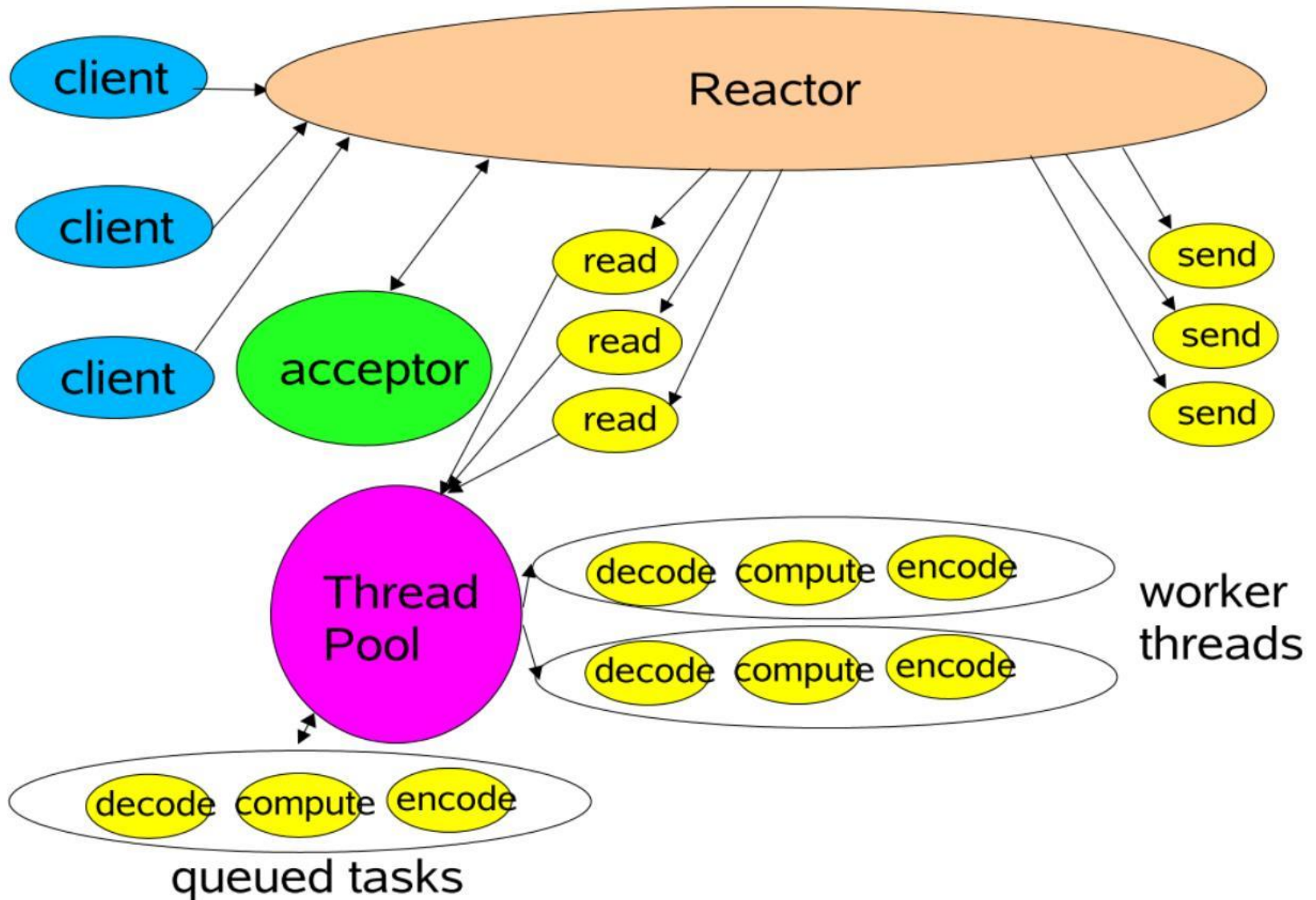
- 为了可伸缩性适当的增加线程
  - 主要适用于多处理器
- **Worker 线程**
  - **Reactor** 应迅速触发 **Handler**
    - **Handler** 的处理过程会导致 **Reactor** 变慢
  - 将非 **IO** 操作的处理分离到其他线程
- **多个 Reactor 线程**
  - **Reactor** 线程可以饱和 **IO**
  - 将负载分配到其他 **Reactor**
    - 负载平衡以匹配 **CPU** 和 **IO** 速率

# Worker 线程

- 将非 IO 操作从 Reactor 线程中分离，提高其速度
  - 类似于 POA2 Proactor 设计
- 比将计算限制处理改造成事件驱动简单
  - 应该还是纯粹的无阻塞计算
    - 充足的计算处理远比那点开销重要
- 但是处理与 IO 重叠的地方比较困难
  - 最好的方式是第一次读的时候读取全部的输入
- 使用线程池，线程数可调节控制
  - 通常需要的线程数比客户端数少的多

译者注：compute-bound 也叫做 CPU-bound，简单来说如果 CPU 快程序就快，就说明程序是受 CPU 限制的，此外还有 I/O-bound，memory-bound。

# Worker 线程池



# Handler 使用线程池

```
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;
            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(); }
    }
}
```

# 协调任务

---

- 切换 (Handoffs)
  - 每个任务的启用，触发或调用下一个
  - 通常很快，但是很脆弱
- 回调每个 handler 的分派者
  - 设置状态，附加的对象等
  - GoF 中介模式的一种变种
- 队列 (Queues)
  - 比如，跨阶段传递缓冲区
- Futures
  - 当每个任务产生结果时
  - 在顶层使用 `join` 或 `wait/notify` 进行协调

# 使用线程池

---

- 可调节的工作线程池
- 主要方法 `execute(Runnable r)`
- 线程池的控制：
  - 任务队列的种类（任何通道）
  - 最大和最小线程数
  - 缓和在线线程的需求
  - 保持存活间隔，直到空闲线程死亡
    - 如有必要，稍后将被新的替换
  - 饱和策略
    - 阻塞，删除，生产者运行，等

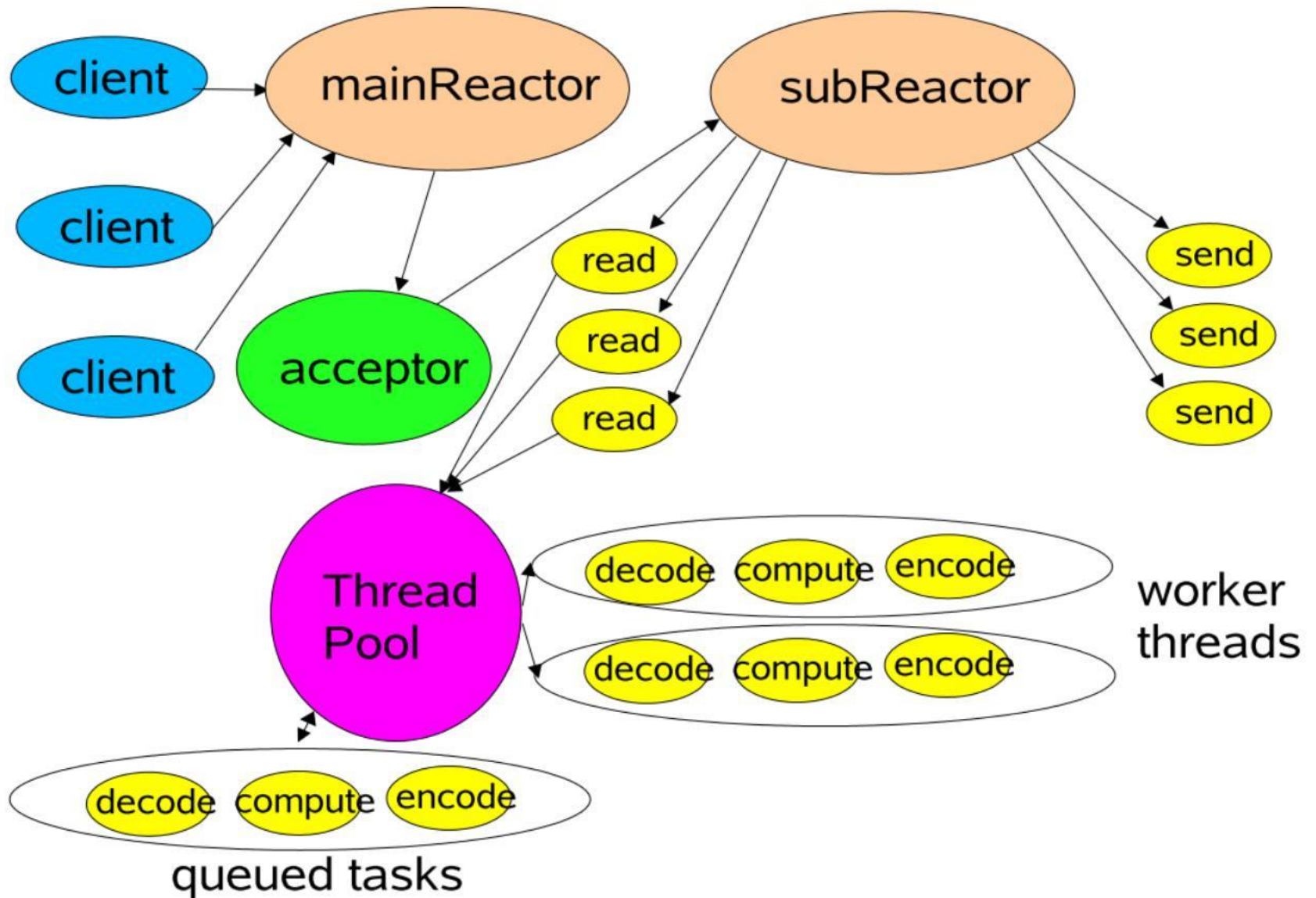


# 多个 Reactor 线程

- 使用 Reactor 池
  - 匹配 CPU 和 IO 的速率
  - 静态或动态结构
    - 有自己的 Selector、Thread、dispatch loop
  - 主 acceptor 向其他 reactor 分配

```
Selector[] selectors; // also create threads
int next = 0;
class Acceptor { // ...
    public synchronized void run() { ...
        Socket connection = serverSocket.accept();
        if (connection != null)
            new Handler(selectors[next], connection);
        if (++next == selectors.length) next = 0;
    }
}
```

# 使用多个 Reactor



# java.nio其他特性

---

- 一个 Reactor 多个 Selector
  - 将不同的 Handler 绑定不同的 IO 事件
  - 可能需要仔细同步才能协调
- 文件传输
  - file-to-net 或 net-to-file的自动拷贝
- 内存映射文件
  - 通过缓冲区访问文件
- 直接缓冲区
  - 有时可以实现零拷贝传输
  - 但是有初始化和回收开销
  - 适合长连接的应用

# 基于连接的扩展

---

- 替换单个请求服务，而是：
  - 客户端连接
  - 客户端发送一系列消息/请求
  - 客户端断开连接
- 示例
  - 数据库和事物监听器
  - 多人游戏、聊天等
- 可扩展基本服务模型
  - 处理多个长连接客户端
  - 跟踪客户端和会话状态（包括删除）
  - 跨主机的分布式服务

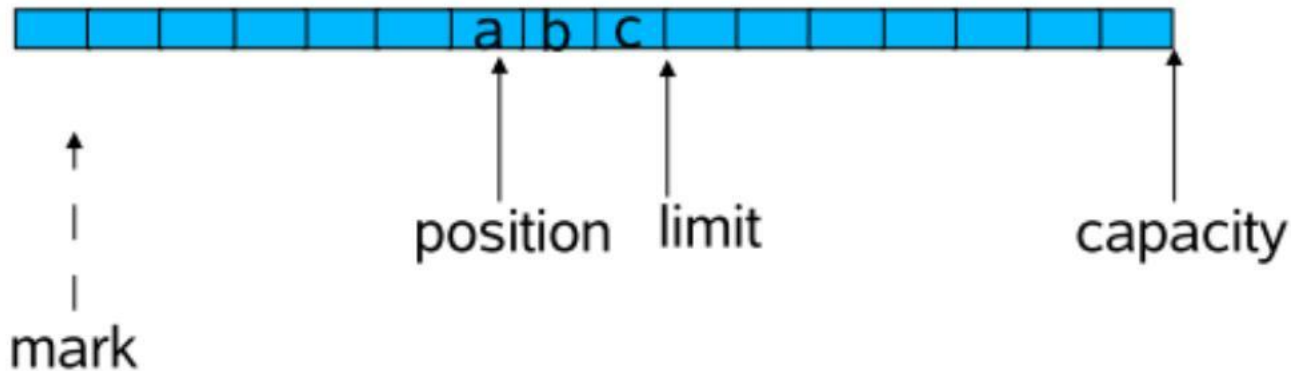
# API 介绍

---

- Buffer
- ByteBuffer  
(CharBuffer, LongBuffer, etc not shown.)
- Channel
- SelectableChannel
- SocketChannel
- ServerSocketChannel
- FileChannel
- Selector
- SelectionKey

# Buffer

```
abstract class Buffer {  
    int    capacity(); // 总容量  
    int    position(); // 当前可读或可写的位置  
    Buffer position(int newPosition);  
    int    limit();    // 可读或可写的位置上限  
    Buffer limit(int newLimit);  
    Buffer mark();      // 设置标志位, mark = position  
    Buffer reset();     // 重置当前位置为标志位, position = mark  
    Buffer clear();     // position=0;mark=-1;limit=capacity;  
    Buffer flip();      // limit=position;position=0;mark=-1;  
    Buffer rewind();    // 重新读取, position=0;mark=-1;  
    int    remaining(); // 返回limit和position之间的元素个数  
    boolean hasRemaining(); // 判断limit和position之间是否有元素  
    boolean isReadOnly(); // 只读  
}
```



# ByteBuffer

```
abstract class ByteBuffer extends Buffer {
    static ByteBuffer allocateDirect(int capacity);
    static ByteBuffer allocate(int capacity); // 创建对象
    static ByteBuffer wrap(byte[] src, int offset, int len);
    static ByteBuffer wrap(byte[] src); // 以上创建字节缓冲区对象
    boolean isDirect(); // 是否使用直接内存
    ByteOrder order(); // 字节序
    ByteOrder order(ByteOrder bo);
    ByteBuffer slice(); // 剩余元素的视图，类似复制
    ByteBuffer duplicate(); // 创建一个新的引用，不复制数据
    ByteBuffer compact(); // 释放部分已读数据
    ByteBuffer asReadOnlyBuffer();
    byte get(); // 读取字节
    byte get(int index);
    ByteBuffer get(byte[] dst, int offset, int length);
    ByteBuffer get(byte[] dst);
    ByteBuffer put(byte b); // 写入字节
    ByteBuffer put(int index, byte b);
    ByteBuffer put(byte[] src, int offset, int length);
    ByteBuffer put(ByteBuffer src);
    ByteBuffer put(byte[] src);
    . . .
}
```

# Channel

---

```
interface Channel {  
    boolean isOpen();  
    void close() throws IOException;  
}
```

```
interface ReadableByteChannel extends Channel {  
    int read(ByteBuffer dst) throws IOException;  
}
```

```
interface WritableByteChannel extends Channel {  
    int write(ByteBuffer src) throws IOException;  
}
```

```
interface ScatteringByteChannel extends ReadableByteChannel {  
    int read(ByteBuffer[] dsts, int offset, int length)  
        throws IOException;  
    int read(ByteBuffer[] dsts) throws IOException;  
}
```

```
interface GatheringByteChannel extends WritableByteChannel {  
    int write(ByteBuffer[] srcs, int offset, int length)  
        throws IOException;  
    int write(ByteBuffer[] srcs) throws IOException;  
}
```



# SelectableChannel

---

```
abstract class SelectableChannel implements Channel {
    int        validOps(); // 返回通道支持的操作
    boolean    isRegistered();
    SelectionKey keyFor(Selector sel);
    SelectionKey register(Selector sel, int ops)
                    throws ClosedChannelException; // 注册
    void        configureBlocking(boolean block)
                    throws IOException; // 设置阻塞或非阻塞
    boolean    isBlocking();
    Object     blockingLock(); // 锁对象, 安全修改阻塞状态
}
```

# SocketChannel

```
abstract class SocketChannel implements ByteChannel ... {
    static SocketChannel open() throws IOException;
    Socket    socket(); // 通道关联的 Socket 对象
    int       validOps();
    boolean   isConnected();
    boolean   isConnectionPending();
    boolean   isInputOpen();
    boolean   isOutputOpen();
    boolean   connect(SocketAddress remote) throws IOException;
    boolean   finishConnect() throws IOException;
    void      shutdownInput() throws IOException;
    void      shutdownOutput() throws IOException;
    int       read(ByteBuffer dst) throws IOException;
    int       read(ByteBuffer[] dsts, int offset, int length)
                throws IOException;
    int       read(ByteBuffer[] dsts) throws IOException;
    int       write(ByteBuffer src) throws IOException;
    int       write(ByteBuffer[] srcs, int offset, int length)
                throws IOException;
    int       write(ByteBuffer[] srcs) throws IOException;
}
```

# ServerSocketChannel

---

```
abstract class ServerSocketChannel extends ... {  
    static ServerSocketChannel open() throws IOException;  
    int          validOps();  
    ServerSocket socket();  
    SocketChannel accept() throws IOException;  
}
```

# FileChannel

```
abstract class FileChannel implements ... {
    int  read(ByteBuffer dst);
    int  read(ByteBuffer[] dsts);
    int  write(ByteBuffer src);
    int  write(ByteBuffer[] srcs);
    long position();
    void position(long newPosition);
    long size();
    void truncate(long size);
    void force(boolean flushMetadataToo);
    int  transferTo(long position, int count,
                   WritableByteChannel dst); // 零拷贝
    int  transferFrom(ReadableByteChannel src,
                     long position, int count);
    FileLock lock(long position, long size, boolean shared);
    FileLock lock();
    FileLock tryLock(long pos, long size, boolean shared);
    FileLock tryLock();
    static final int MAP_RO, MAP_RW, MAP_COW;
    MappedByteBuffer map(int mode, long position, int size);
}
NOTE: ALL methods throw IOException
```

# Selector

---

```
abstract class Selector {  
    static Selector open() throws IOException;  
    Set  keys(); // 返回已注册的 key 集合  
    Set  selectedKeys();  
    int  selectNow() throws IOException;  
    int  select(long timeout) throws IOException;  
    int  select() throws IOException;  
    void wakeup();  
    void close() throws IOException;  
}
```

# SelectionKey

```
abstract class SelectionKey {
    static final int    OP_READ,      OP_WRITE,
                      OP_CONNECT, OP_ACCEPT;
    SelectableChannel channel(); // 注册的通道
    Selector          selector(); // 关联的 Selector
    boolean           isValid();   // key 是否有效
    void              cancel();   // key立即失效, 下次select()清理
    int               interestOps(); // 感兴趣的事件
    void              interestOps(int ops);
    int               readyOps();
    boolean           isReadable();
    boolean           isWritable();
    boolean           isConnectable();
    boolean           isAcceptable();
    Object            attach(Object ob); // 附加一个对象
    Object            attachment(); // 获取附加的对象
}
```