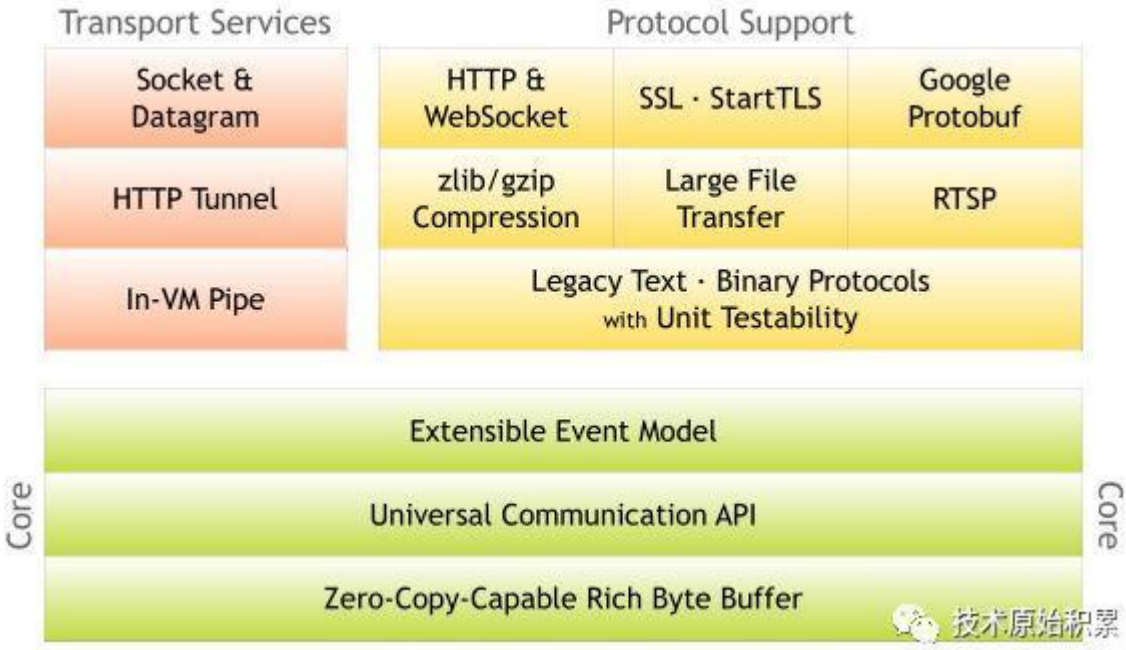
高性能网络通信框架 Netty-基础概念篇

一、前言

Netty 是一种可以轻松快速的开发协议服务器和客户端网络应用程序的 NIO 框架，它大大简化了 TCP 或者 UDP 服务器的网络编程,但是你仍然可以访问和使用底层的 API,Netty 只是对其进行了高层的抽象。Netty 的简易和快速开发并不意味着由它开发的程序将失去可维护性或者存在性能问题。Netty 是被精心设计的，它的设计参考了许多协议的实现，比如 FTP，SMTP，HTTP 和各种二进制和基于文本的传统协议，因此 Netty 成功的实现了兼顾快速开发，性能，稳定性，灵活性为一体，不需要为了考虑一方面原因而妥协其他方面。



二、基础概念

Channel 也就是通道，这个概念是在 JDK NIO 类库里面提供的一个概念，JDK 中其实现类有客户端套接字通道 java.nio.channels.SocketChannel 和服务端监听套接字通道 java.nio.channels.ServerSocketChannel，Channel 的出现是为了支持异步 IO 操作，JDK 里面的通道是 java.nio.channels.Channel。io.netty.channel.Channel 是 Netty 框架自己定义的一个通道接口，Netty 实现的客户端 NIO 套接字通道是 NioSocketChannel，提供的服务器端 NIO 套接字通道是 NioServerSocketChannel。

- NioSocketChannel

客户端套接字通道，内部管理了一个 Java NIO 中的

java.nio.channels.SocketChannel 实例，用来创建 SocketChannel 实例和设置该实例的属性，并调用 Connect 方法向服务端发起 TCP 链接等。

- NioServerSocketChannel

  服务器端监听套接字通道，内部管理了一个 Java NIO 中的 java.nio.channels.ServerSocketChannel 实例，用来创建 ServerSocketChannel 实例和设置该实例属性，并调用该实例的 bind 方法在指定端口监听客户端的链接。

- Channel 与 socket 的关系

  在 Netty 中 Channel 有两种，对应客户端套接字通道 NioSocketChannel，内部管理 java.nio.channels.SocketChannel 套接字，对应服务器端监听套接字通道 NioServerSocketChannel，其内部管理自己的 java.nio.channels.ServerSocketChannel 套接字。也就是 Channel 是对 socket 的装饰或者门面，其封装了对 socket 的原子操作。

- EventLoopGroup

  Netty 之所以能提供高性能网络通讯，其中一个原因是因为它使用 Reactor 线程模型。在 netty 中每个 EventLoopGroup 本身是一个线程池，其中包含了自定义个数的 NioEventLoop,每个 NioEventLoop 是一个线程，并且每个 NioEventLoop 里面持有自己的 selector 选择器在

Netty 中客户端持有一个 EventLoopGroup 用来处理网络 IO 操作，在服务器端持有两个 EventLoopGroup，其中 boss 组是专门用来接收客户端发来的 TCP 链接请求的，worker 组是专门用来具体处理完成三次握手的链接套接字的网络 IO 请求的。

- Channel 与 EventLoop 的关系

Netty 中 NioEventLoop 是 EventLoop 的一个实现，每个 NioEventLoop 中会管理自己的一个 selector 选择器和监控选择器就绪事件的线程；每个 Channel 只会关联一个 NioEventLoop ；

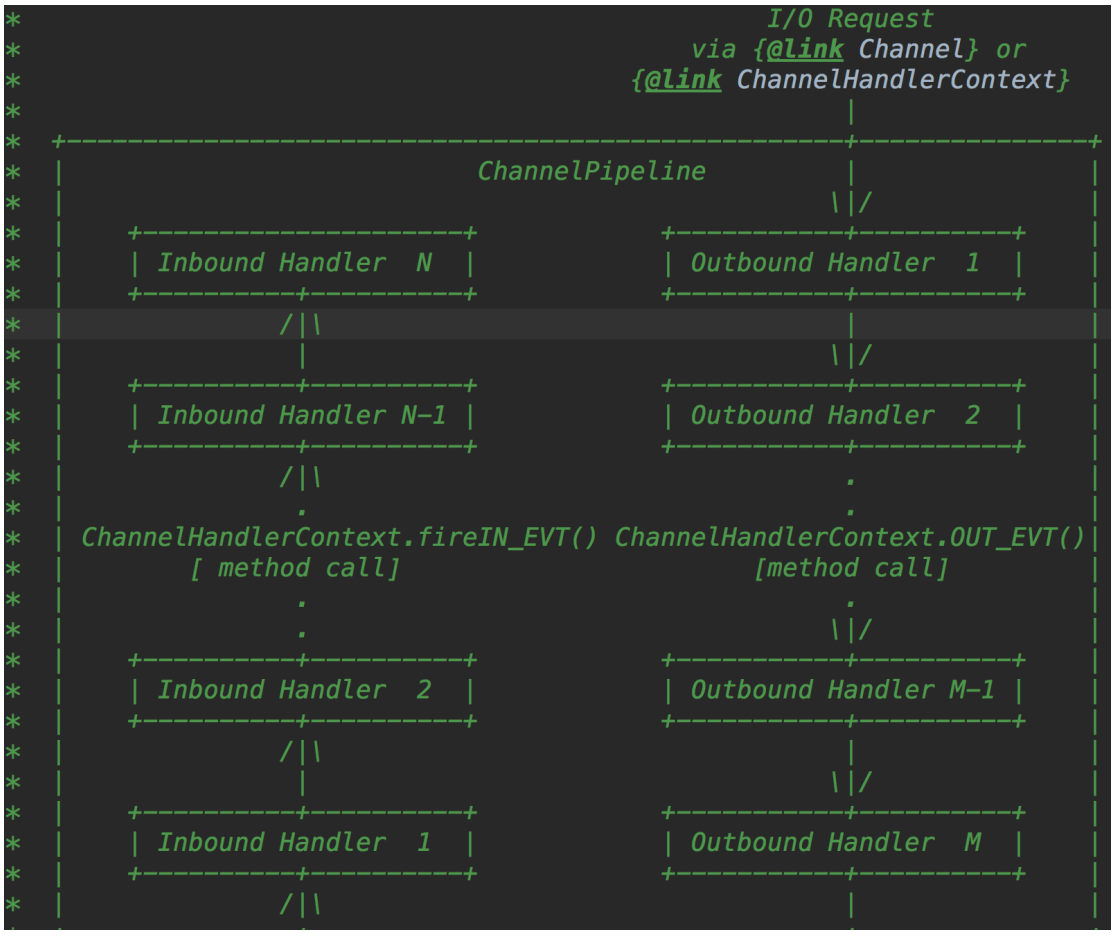
当 Channel 是客户端通道 NioSocketChannel 时候，会注册 NioSocketChannel 管理的 SocketChannel 实例到自己关联的 NioEventLoop 的 selector 选择器上，然后 NioEventLoop 对应的线程会通过 select 命令监控感兴趣的网络读写事件。当 Channel 是服务端通道 NioServerSocketChannel 时候，NioServerSocketChannel 本身会被注册到 boss EventLoopGroup 里面的某一个 NioEventLoop 管理的 selector 选择器上，而完成三次握手的链接套接字是被注册到了 worker EventLoopGroup 里面的某一个 NioEventLoop 管理的 selector 选择器上；

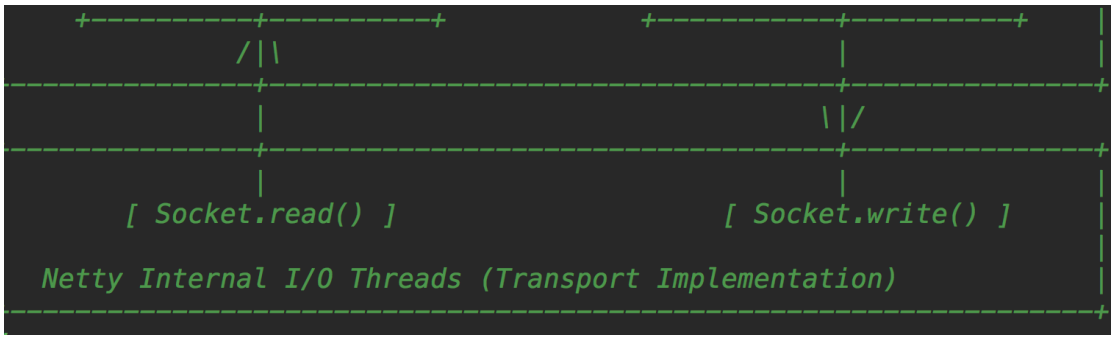
需要注意是多个 Channel 可以注册到同一个 NioEventLoop 管理的 selector 选择器上，这时候 NioEventLoop 对应的单

个线程就可以处理多个 Channel 的就绪事件；但是每个 Channel 只能注册到一个固定的 NioEventLoop 管理的 selector 选择器上。

- ChannelPipeline

Netty 中的 ChannelPipeline 类似于 Tomcat 容器中的 Filter 链，属于设计模式中的责任链模式，其中链上的每个节点就是一个 ChannelHandler。在 netty 中每个 Channel 有属于自己的 ChannelPipeline，对从 Channel 中读取或者要写入 Channel 中的数据进行依次处理,如下图是 netty 源码里面的一个图：





需要注意一点是虽然每个 Channel（更底层说是每个 socket）有自己的 ChannelPipeline，但是每个 ChannelPipeline 里面可以复用一个 ChannelHandler。

三 、Netty 客户端底层与 Java NIO 对应关系

在讲解 Netty 客户端程序时候我们提到指定 NioSocketChannel 用于创建客户端 NIO 套接字通道的实例，下面我们来看 NioSocketChannel 是如何创建一个 Java NIO 里面的 SocketChannel 的。

首先我们来看 NioSocketChannel 的构造函数：

```
public NioSocketChannel() {
    this(DEFAULT_SELECTOR_PROVIDER);
}
```

其中 DEFAULT_SELECTOR_PROVIDER 定义如下：

```
    private static final SelectorProvider
DEFAULT_SELECTOR_PROVIDER =
SelectorProvider.provider();
```

然后继续看

```
//这里的 provider 为 DEFAULT_SELECTOR_PROVIDER
    public
NioSocketChannel(SelectorProvider provider)
{
        this(newSocket(provider));
    }
```

其中 newSocket 代码如下：

```
    private static SocketChannel
newSocket(SelectorProvider provider) {
        try {
            return
provider.openSocketChannel();
        } catch (IOException e) {
```

```
            throw new
ChannelException("Failed to open a
socket.", e);
        }
    }
```

所以 NioSocketChannel 内部是管理一个客户端的 SocketChannel 的，这个 SocketChannel 就是讲 Java NIO 时候的 SocketChannel,也就是创建 NioSocketChannel 实例对象时候相当于执行了 Java NIO 中：

```
SocketChannel socketChannel =
SocketChannel.open();
```

另外在 NioSocketChannel 的父类 AbstractNioChannel 的构造函数里面默认会记录队 op_read 事件感兴趣，这个后面当链接完成后会使用到：

```
    protected
AbstractNioByteChannel(Channel parent,
SelectableChannel ch) {
```

```
        super(parent, ch,
SelectionKey.OP_READ);

    }
```

另外在 NioSocketChannel 的父类 AbstractNioChannel 的构造函数里面设置了该套接字为非阻塞的

```
protected AbstractNioChannel(Channel
parent, SelectableChannel ch, int
readInterestOp) {

        super(parent);

        this.ch = ch;

        this.readInterestOp =
readInterestOp;

        try {

            ch.configureBlocking(false);

        } catch (IOException e) {

         ...

        }
```

```
    }
```

下面我们看 Netty 里面是哪里创建的 NioSocketChannel 实例，哪里注册到选择器的。

下面我们看下 Bootstrap 的 connect 操作代码：

```
    public ChannelFuture
connect(InetAddress inetHost, int inetPort)
{

        return connect(new
InetSocketAddress(inetHost, inetPort));

    }
```

类似 Java NIO 传递了一个 InetSocketAddress 对象用来记录服务端 ip 和端口：

```
    public ChannelFuture
connect(SocketAddress remoteAddress) {

       ...

        return
doResolveAndConnect(remoteAddress,
config.localAddress());
```

```
    }
```

下面我们看下 doResolveAndConnect 的代码：

```
private ChannelFuture
doResolveAndConnect(final SocketAddress
remoteAddress, final SocketAddress
localAddress) {

       //(1)

       final ChannelFuture regFuture =
initAndRegister();

       final Channel channel =
regFuture.channel();



       if (regFuture.isDone()) {

           if (!regFuture.isSuccess()) {

               return regFuture;

           }

            //(2)
```

```
            return
doResolveAndConnect0(channel,
remoteAddress, localAddress,
channel.newPromise());
        }
            ...
       }
}
```

首先我们来看代码（1）initAndRegister：

```
    final ChannelFuture initAndRegister() {
        Channel channel = null;
        try {
            //(1.1)
            channel =
channelFactory.newChannel();
            //(1.2)
            init(channel);
```

```
        } catch (Throwable t) {
            ...
        }
        //(1.3)
        ChannelFuture regFuture =
config().group().register(channel);
        if (regFuture.cause() != null) {
            if (channel.isRegistered()) {
                channel.close();
            } else {

channel.unsafe().closeForcibly();
            }
        }
}
```

其中（1.1）作用就是创建一个 NioSocketChannel 的实例，代码（1.2）是具体设置内部套接字的选项的。

代码（1.3)则是具体注册客户端套接字到选择器的，其首先会调用 NioEventLoop 的 register 方法，最后调用 NioSocketChannelUnsafe 的 register 方法：

```
public final void register(EventLoop
eventLoop, final ChannelPromise promise) {
          ...
          AbstractChannel.this.eventLoop =
eventLoop;

          if (eventLoop.inEventLoop()) {
             register0(promise);
          } else {
             try {
                eventLoop.execute(new
Runnable() {
                   @Override
```

```
                    public void run() {
                        register0(promise);
                    }
                });
            } catch (Throwable t) {
                ...
            }
        }
    }
```

其中 register0 内部调用 doRegister，其代码如下：

```
protected void doRegister() throws
Exception {
        boolean selected = false;
        for (;;) {
            try {
```

```
                //注册客户端 socket 到当前
eventloop 的 selector 上
                selectionKey =
javaChannel().register(eventLoop().unwrappe
dSelector(), 0, this);
                return;
            } catch (CancelledKeyException e)
{
                ...
            }
        }
 }
```

到这里代码（1）initAndRegister 的流程讲解完毕了，下面我们来看代码（2）的

```
  public final void connect(
```

```
            final SocketAddress
remoteAddress, final SocketAddress
localAddress, final ChannelPromise promise)
{

        ...

          try {

              ...


              boolean wasActive =
isActive();

              if (doConnect(remoteAddress,
localAddress)) {


fulfillConnectPromise(promise, wasActive);

              } else {

                ◦ ◦ ◦

              }

          } catch (Throwable t) {
```

```
            ...
        }
    }
```

其中 doConnect 代码如下：

```
    protected boolean
doConnect(SocketAddress remoteAddress,
SocketAddress localAddress) throws
Exception {
        ...
        boolean success = false;
        try {
            //2.1
            boolean connected =
SocketUtils.connect(javaChannel(),
remoteAddress);
            //2.2
            if (!connected) {
```

```
selectionKey().interestOps(SelectionKey.OP_
CONNECT);

            }

            success = true;

            return connected;

        } finally {

            if (!success) {

                doClose();

            }

        }

    }
```

其中 2.1 具体调用客户端套接字的 connect 方法，等价于 Java NIO 里面的。

代码 2.2 由于 connect 方法是异步的，所以类似 JavaNIO 调用 connect 方法进行判断，如果当前没有完成链接则设置对 op_connect 感兴趣。

最后一个点就是何处进行的从选择器获取就绪的事件的,具体是在该客户端套接关联的 NioEventLoop 里面的做的，每个 NioEventLoop 里面有一个线程用来循环从选择器里面获取就绪的事件，然后进行处理：

```
 protected void run() {

        for (;;) {

            try {

                ...


select(wakenUp.getAndSet(false));

                ...

                processSelectedKeys();

                ...

            } catch (Throwable t) {

                handleLoopException(t);

            }

            ...
```

```
        }

    }
```

其中 select 代码如下：

```
 private void select(boolean oldWakenUp)
throws IOException {

       Selector selector = this.selector;

       try {

           ...

           for (;;) {

               ...

               int selectedKeys =
selector.select(timeoutMillis);

               selectCnt ++;



             ...

       } catch (CancelledKeyException e) {
```

```
            ...
        }
 }
```

可知会从选择器选取就绪的事件，其中 processSelectedKeys 代码如下：

```
 private void processSelectedKeys() {
        ...

processSelectedKeysPlain(selector.selectedK
eys());
        ...
    }
```

可知会获取已经就绪的事件集合，然后交给 processSelectedKeysPlain 处理，后者循环调用 processSelectedKey 具体处理每个事件，代码如下：

```
 private void
processSelectedKey(SelectionKey k,
AbstractNioChannel ch) {

      ...

       try {

           //(3)如果是 op_connect 事件

           int readyOps = k.readyOps();

           if ((readyOps &
SelectionKey.OP_CONNECT) != 0) {

               int ops = k.interestOps();

               ops &=
~SelectionKey.OP_CONNECT;

               k.interestOps(ops);

               //3.1

               unsafe.finishConnect();

           }

           //4
```

```
            if ((readyOps &
SelectionKey.OP_WRITE) != 0) {
                ch.unsafe().forceFlush();
            }
            //5
            if ((readyOps &
(SelectionKey.OP_READ |
SelectionKey.OP_ACCEPT)) != 0 || readyOps
== 0) {
                unsafe.read();
            }
        } catch (CancelledKeyException
ignored) {

unsafe.close(unsafe.voidPromise());
        }
    }
```

代码（3）如果当前事件 key 为 op_connect 则去掉 op_connect，然后调用 NioSocketChannel 的 doFinishConnect：

```
    protected void doFinishConnect() throws
Exception {
        if (!javaChannel().finishConnect())
{
            throw new Error();
        }
    }
```

可知是调用了客户端套接字的 finishConnect 方法，最后会调用 NioSocketChannel 的 doBeginRead 方法设置对 op_read 事件感兴趣：

```
    protected void doBeginRead() throws
Exception {
       ...
        final int interestOps =
selectionKey.interestOps();
```

```
        if ((interestOps & readInterestOp)
== 0) {


selectionKey.interestOps(interestOps |
readInterestOp);

        }

    }
```

这里 interestOps 为 op_read,上面在讲解 NioSocketChannel 的构造函数时候提到过。

代码（5）如果当前是 op_accept 事件说明是服务器监听套接字获取到了一个链接套接字，如果是 op_read,则说明可以读取客户端发来的数据了，如果是后者则会激活管线里面的所有 handler 的 channelRead 方法，这里会激活我们自定义的 NettyClientHandler 的 channelRead 读取客户端发来的数据，然后在向客户端写入数据。

四 总结

本节讲解了 Netty 客户端底层如何使用 Java NIO 进行实现的，可见与我们前面讲解的 Java NIO 设计的客户端代码步骤是一致的，只是 netty 对其进行了封装，方便了我

们使用，了解了这些对深入研究 netty 源码提供了一个骨架指导。

五、Netty(二) 从线程模型的角度看 Netty 为什么是高性能

传统 IO

在 Netty 以及 NIO 出现之前，我们写 IO 应用其实用的都是用 java.io.* 下所提供的包。

比如下面的伪代码：

```
ServeSocket serverSocket = new ServeSocket(
8080);

Socket socket = serverSocket.accept() ;

BufferReader in = .... ;



String request ;



while((request = in.readLine()) != null){

    new Thread(new Task()).start()
```

```
}
```

大概是这样，其实主要想表达的是：**这样一个线程只能处理一个连接**。

如果是 100 个客户端连接那就得开 100 个线程，1000 那就得 1000 个线程。

要知道线程资源非常宝贵，每次的创建都会带来消耗，而且每个线程还得为它分配对应的栈内存。

即便是我们给 JVM 足够的内存，大量线程所带来的上下文切换也是受不了的。

并且传统 IO 是阻塞模式，每一次的响应必须的是发起 IO 请求，处理请求完成再同时返回，直接的结果就是性能差，吞吐量低。

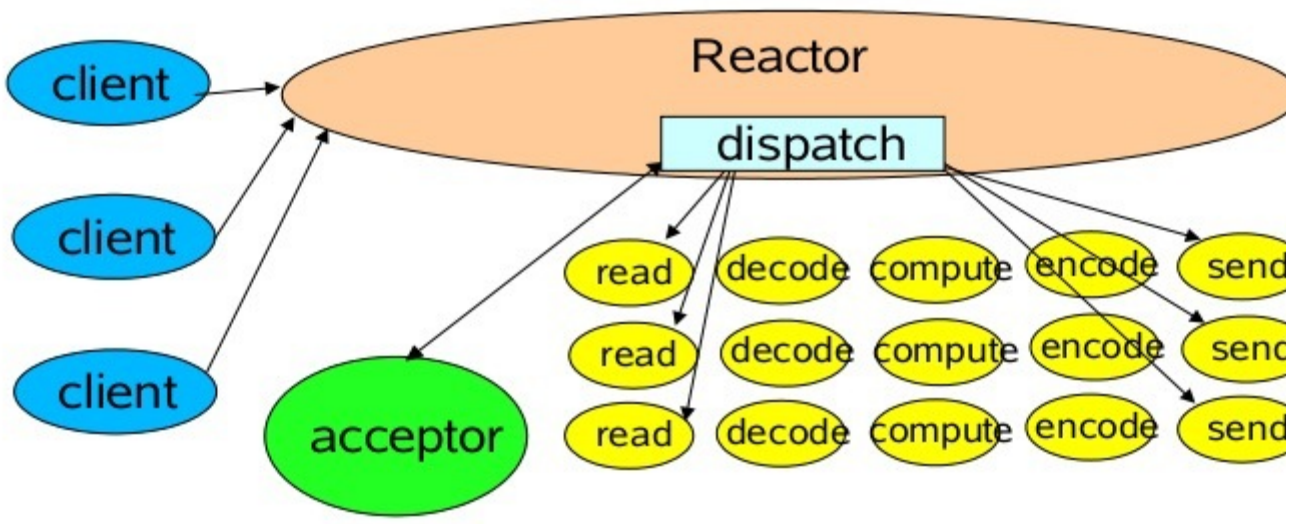
Reactor 模型

因此业界常用的高性能 IO 模型是 Reactor。

它是一种异步、非阻塞的事件驱动模型。

通常也表现为以下三种方式：

单线程



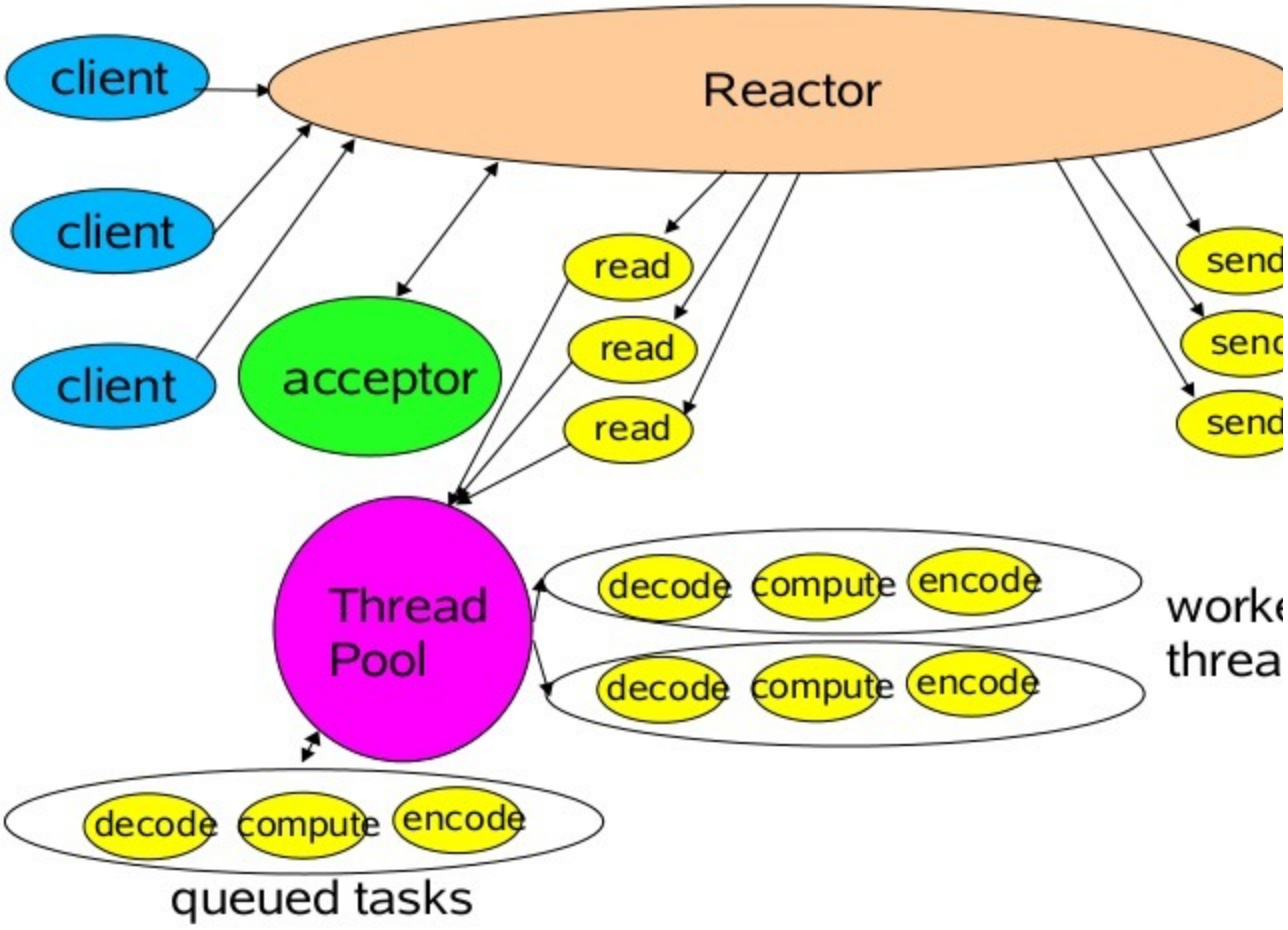
从图中可以看出：

它是由一个线程来接收客户端的连接，并将该请求分发到对应的事件处理 handler 中，整个过程完全是异步非阻塞的；并且完全不存在共享资源的问题。所以理论上来说吞吐量也还不错。

但由于是一个线程，对多核 CPU 利用率不高，一旦有大量的客户端连接上来性能必然下降，甚至会有大量请求无法响应。

最坏的情况是一旦这个线程哪里没有处理好进入了死循环那整个服务都将不可用！

多线程



因此产生了多线程模型。

其实最大的改进就是将原有的事件处理改为了多线程。

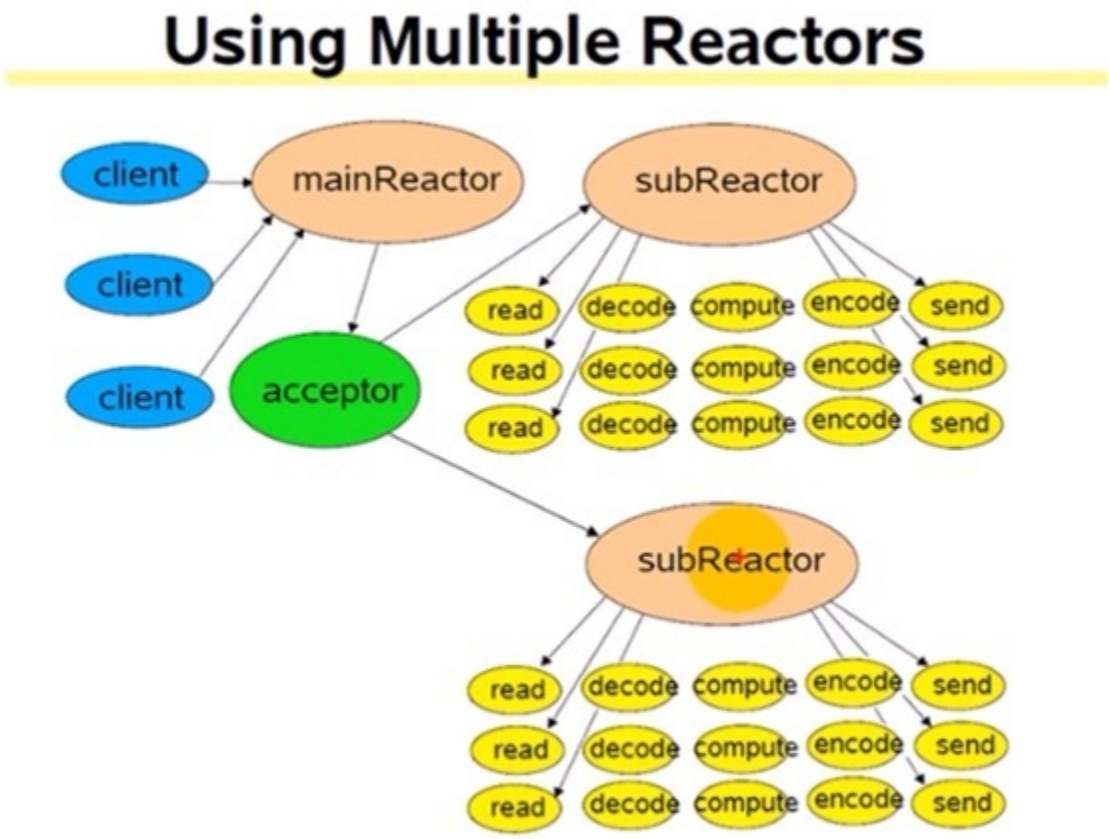
可以基于 Java 自身的线程池实现，这样在大量请求的处理上性能提示是巨大的。

虽然如此，但理论上来说依然有一个地方是单点的；那就是处理客户端连接的线程。

因为大多数服务端应用或多或少在连接时都会处理一些业务，如鉴权之类的，当连接的客户端越来越多时这一个线程依然会存在性能问题。

于是又有了下面的线程模型。

主从多线程



该模型将客户端连接那一块的线程也改为多线程，称为主线程。

同时也是多个子线程来处理事件响应，这样无论是连接还是事件都是高性能的。

Netty 实现

以上谈了这么多其实 Netty 的线程模型与之的类似。

我们回到之前 SpringBoot 整合长连接心跳机制 中的服务端代码：

```
    private EventLoopGroup boss = new NioEv
entLoopGroup();

    private EventLoopGroup work = new NioEv
entLoopGroup();



    /**
     * 启动 Netty
     *
     * @return
     * @throws InterruptedException
     */
    @PostConstruct
    public void start() throws InterruptedE
xception {
```

```
        ServerBootstrap bootstrap = new Ser
verBootstrap()
                .group(boss, work)
                .channel(NioServerSocketCha
nnel.class)
                .localAddress(new InetSocke
tAddress(nettyPort))
                //保持长连接
                .childOption(ChannelOption.
SO_KEEPALIVE, true)
                .childHandler(new Heartbeat
Initializer());


        ChannelFuture future = bootstrap.bi
nd().sync();
        if (future.isSuccess()) {
            LOGGER.info("启动 Netty 成功");
        }
```

```
    }
```

其实这里的 boss 就相当于 Reactor 模型中处理客户端连接的线程池。

work 自然就是处理事件的线程池了。

那么如何来实现上文的三种模式呢？其实也很简单：

单线程模型：

```
private EventLoopGroup group = new NioEvent
LoopGroup();

ServerBootstrap bootstrap = new ServerBoots
trap()

                .group(group)

                .childHandler(new Heartbeat
Initializer());
```

多线程模型：

```
private EventLoopGroup boss = new NioEventL
oopGroup(1);
```

```
private EventLoopGroup work = new NioEventL
oopGroup();

ServerBootstrap bootstrap = new ServerBoots
trap()

                .group(boss,work)

                .childHandler(new Heartbeat
Initializer());
```

主从多线程:

```
private EventLoopGroup boss = new NioEventL
oopGroup();

private EventLoopGroup work = new NioEventL
oopGroup();

ServerBootstrap bootstrap = new ServerBoots
trap()

                .group(boss,work)

                .childHandler(new Heartbeat
Initializer());
```

相信大家一看也明白。

总结

其实看过了 Netty 的线程模型之后能否对我们平时做高性能应用带来点启发呢？

我认为是可以的：

- 接口同步转异步处理。
- 回调通知结果。
- 多线程提高并发效率。

无非也就是这些，只是做了这些之后就会带来其他问题：

- 异步之后事务如何保证？
- 回调失败的情况？
- 多线程所带来的上下文切换、共享资源的问题。

这就是一个博弈的过程，想要做到一个尽量高效的应用是需要不断磨合试错的。