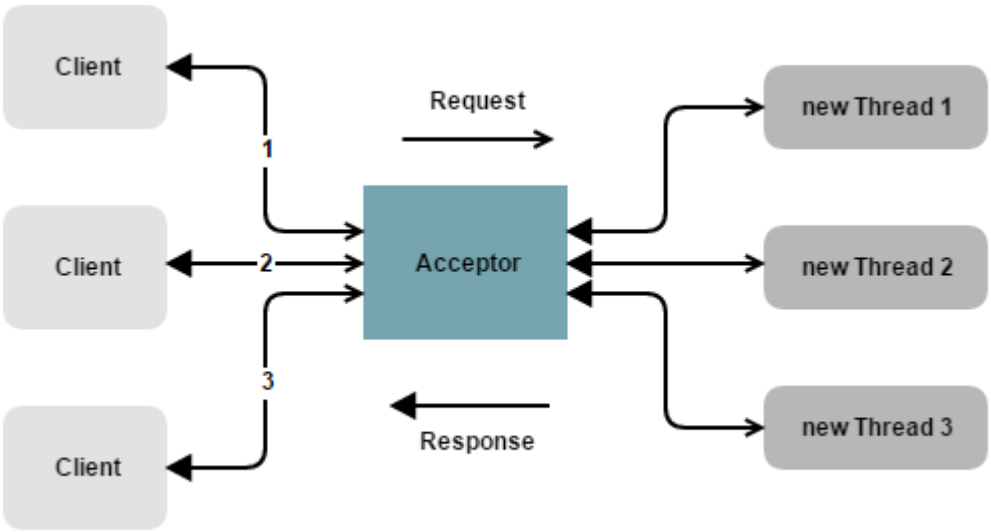
从Blocking I/O 到Netty

一. Java Blocking I/O



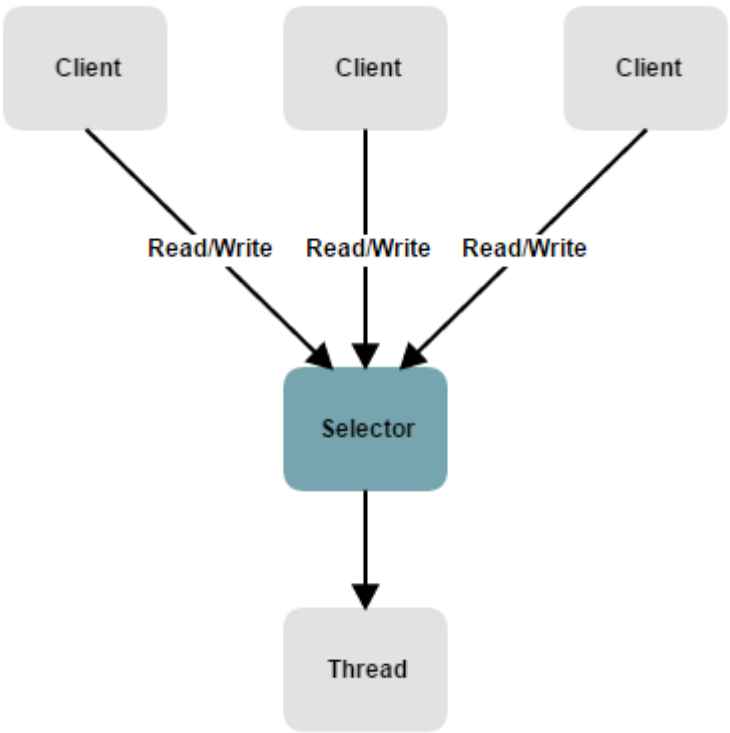

```
{
    ExecutorService executor = Excutors.newFixedThreadPollExecutor(100);//线程池

    ServerSocket serverSocket = new ServerSocket();
    serverSocket.bind(8088);
    while(!Thread.currentThread.isInturrupted()){//主线程死循环等待新连接到来
        Socket socket = serverSocket.accept();//blocking
        executor.submit(new ConnectIOnHandler(socket));//为新的连接创建新的线程
    }

    class ConnectIOnHandler extends Thread{
        private Socket socket;
        public ConnectIOnHandler(Socket socket){
            this.socket = socket;
        }
        public void run(){
            while(!Thread.currentThread.isInturrupted()&&!socket.isClosed()){死循环处理读写事件
                String someThing = socket.read()....//读取数据(blocking)
                if(someThing!=null){
                    ......//处理数据
                        socket.write()....//写数据
```

```
                }
                                                                    }
        }
    }
```

不足:

1、线程的创建和销毁成本很高
2、线程的切换成本是很高
3、线程数量过多,使系统负载压力过大。
4、没有充分利用多核CPU

二. Java NO Blocking I/O or New I/O



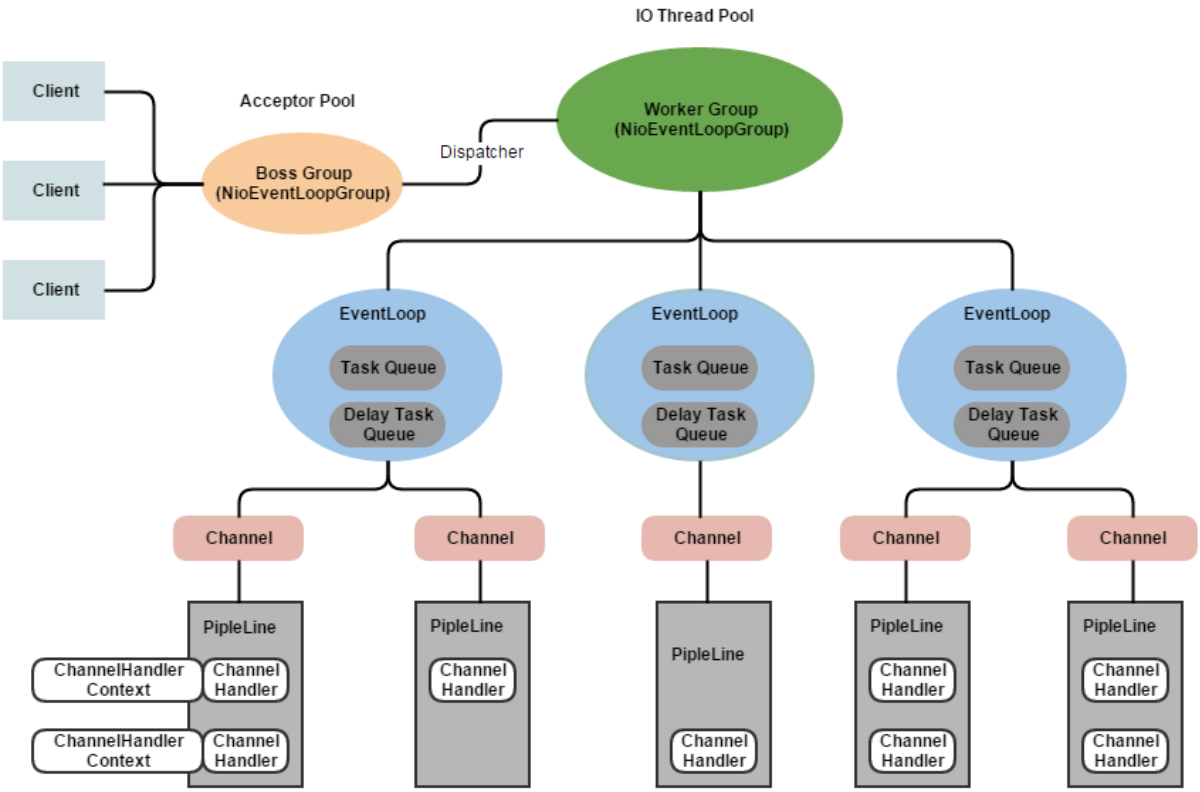

```
while (true) {
    //无事件到底阻塞
    selector.select();
    Iterator<SelectionKey> keys = this.selector.selectedKeys().iterator();
    while (keys.hasNext()) {
        SelectionKey key = keys.next();
        keys.remove();
        handler(key);
    }
}
/**
```

```
     * 处理不同事件的请求
     * @param key
     */
    private void handler(SelectionKey key) throws IOException {
        if (key.isAcceptable()) {
            handleAccept(key);
        } else if (key.isReadable()) {
            handleRead(key);
        }
    }
    //处理连接请求
    private void handleAccept(SelectionKey key){
        ...
    }
    //处理读操作
    private void handleRead(SelectionKey key){
        ...
    }
```

`NIO`和 `BIO`的对比

- IO 基于流(Stream oriented), 而 NIO 基于 Buffer (Buffer oriented)
- IO 操作是阻塞的, 而 NIO 操作是非阻塞的
- IO 没有 selector 概念, 而 NIO 有 selector 概念.

三. Java Netty



Netty工作原理

3.1 Netty基于服务端例子

```
private void startServer() throws InterruptedException {
    //创建boss接收进来的连接
    EventLoopGroup boss = new NioEventLoopGroup();
    //创建worker处理已经接收的连接
    EventLoopGroup worker = new NioEventLoopGroup();
    try {
        //创建nio辅助启动类
        ServerBootstrap bootstrap = new ServerBootstrap();
        bootstrap.group(boss, worker).channel(NioServerSocketChannel.class)
            .childHandler(new ChannelInitializer<SocketChannel>() {
                @Override
                protected void initChannel(SocketChannel channel) throws Exception {
                    channel.pipeline().addLast(new EchoServerHandler());
                }
            });
        //绑定端口准备接收进来的连接
        ChannelFuture future = bootstrap.bind(port).sync();
        //等待服务器socket关闭
        future.channel().closeFuture().sync();
    } finally {
        boss.shutdownGracefully();
        worker.shutdownGracefully();

    }
```

```
}
```

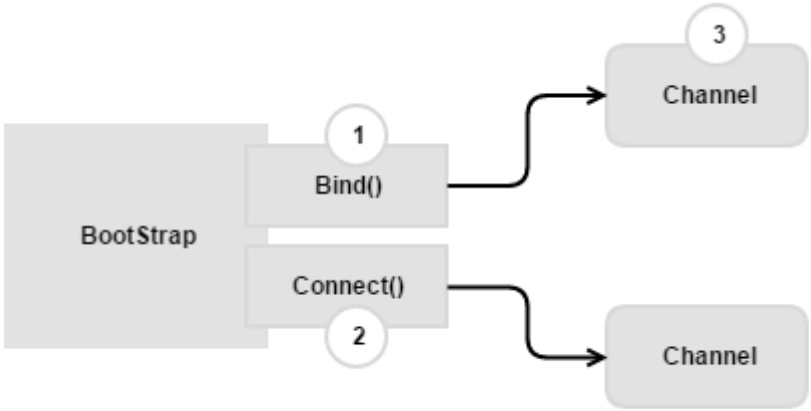
3.2 Netty核心组件

3.2.1 `Bootstrap 和 ServerBootstrap`

Bootstrapping 有两种类型，一种是用于客户端的Bootstrap，一种是用于服务端的`ServerBootstrap`

分类	Bootstrap	ServerBootstrap
网络功能	连接到远程主机和端口	绑定本地端口
EventLoopGroup 数量	1	2

- **Bootstrap如何引导客户端:**

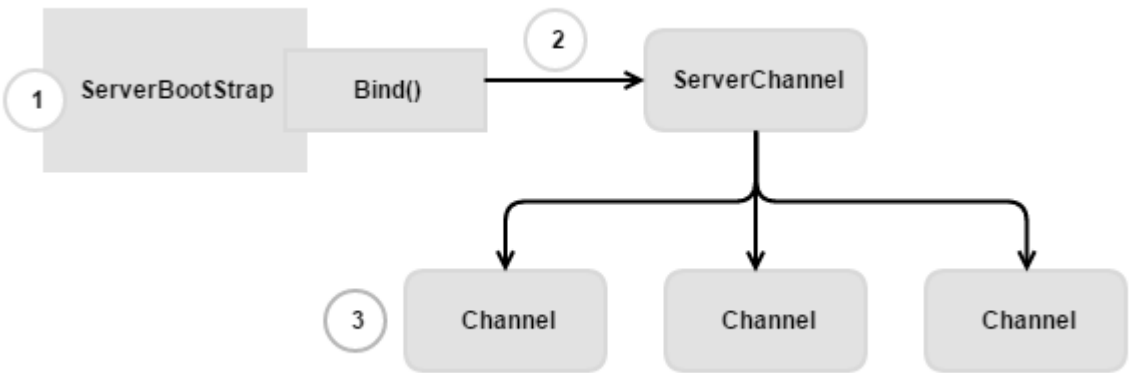


1.当 bind() 调用时，Bootstrap 将创建一个新的管道, 当 connect() 调用在 Channel 来建立连接

2.Bootstrap 将创建一个新的管道, 当 connect() 调用时

3.新的 Channel

- **ServerBootstrap如何引导服务端:**



1.当调用 bind() 后 ServerBootstrap 将创建一个新的管道，这个管道将会在绑定成功后接收子管道

2.接收新连接给每个子管道

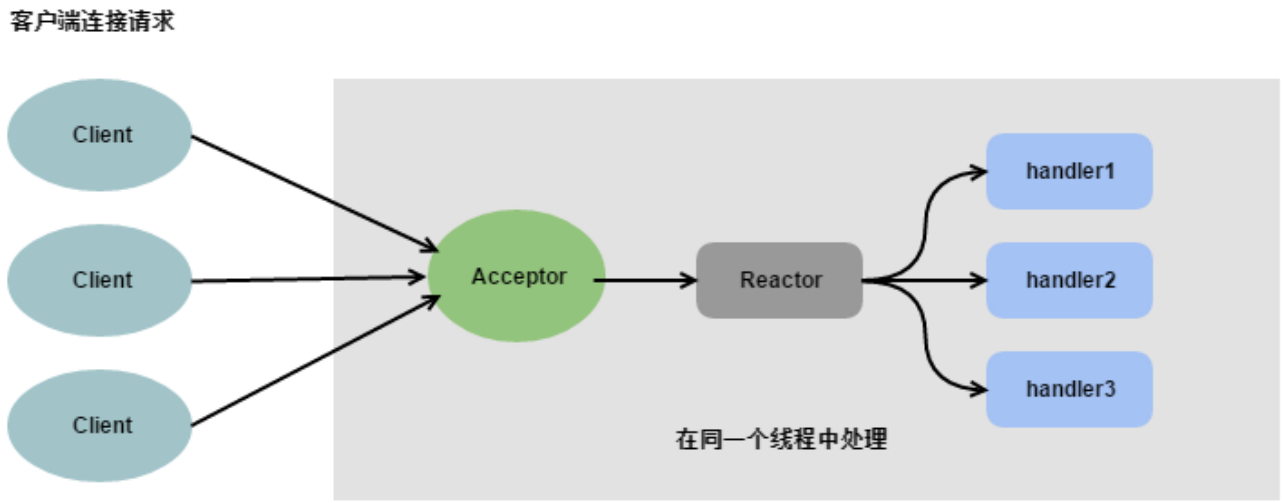
3.接收连接的 Channel

3.2.2 `EventLoopGroup`

Netty 中`EventLoopGroup`是 Reactor 模型的一个实现

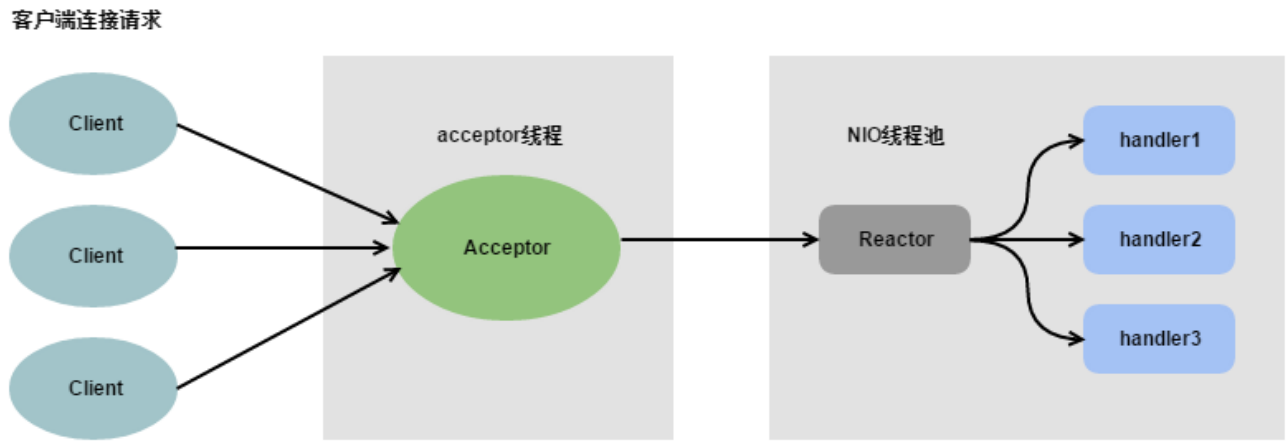
> 什么是Reactor呢？可以这样理解，Reactor就是一个执行while (true) { selector.select(); ...}循环的线程，会源源不断的产生新的事件，称作反应堆很贴切。 事件又分为连接事件、IO读和IO写事件，一般把连接事件单独放一线程里处理，即主Reactor（MainReactor），IO读和IO写事件放到另外的一组线程里处理，即从Reactor（SubReactor），从Reactor线程数量一般为2*(CPUs - 1)。 所以在运行时，MainReactor只处理Accept事件，连接到来，马上按照策略转发给从Reactor之一，只处理连接，故开销非常小；每个SubReactor管理多个连接，负责这些连接的读和写，属于IO密集型线程，读到完整的消息就丢给业务线程池处理业务，处理完比后，响应消息一般放到队列里，SubReactor会去处理队列，然后将消息写回。

Reactor单线程模型：



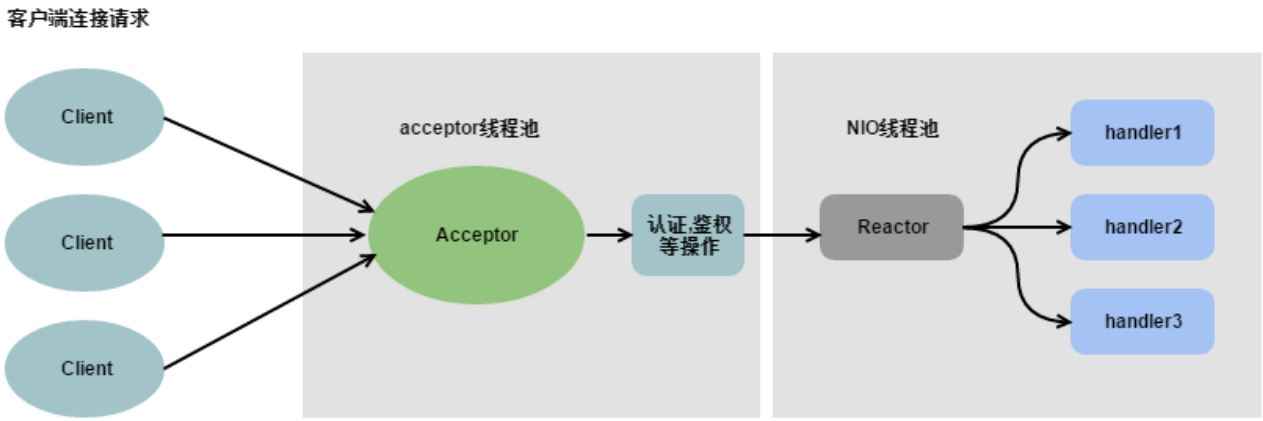
> 所谓单线程, 即 acceptor 处理和 handler 处理都在一个线程中处理. 这个模型的坏处显而易见: 当其中某个 handler 阻塞时, 会导致其他所有的 client 的 handler 都得不到执行, 并且更严重的是, handler 的阻塞也会导致整个服务不能接收新的 client 请求(因为 acceptor 也被阻塞了). 因为有这么多的缺陷, 因此单线程Reactor模型用的比较少.

Reactor多线程模型:



- 有专门一个线程, 即 Acceptor 线程用于监听客户端的TCP连接请求.
- 客户端连接的 IO 操作都是由一个特定的 NIO 线程池负责. 每个客户端连接都与一个特定的 NIO 线程绑定, 因此在这个客户端连接中的所有 IO 操作都是在同一个线程中完成的.
- 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。

Reactor主从多线程模型



- 从主线程池中随机选择一个Reactor线程作为Acceptor线程，用于绑定监听端口，接收客户端连接；
- Acceptor线程接收客户端连接请求之后创建新的`SocketChannel`，将其注册到主线程池的其它Reactor线程上，由其负责接入认证、IP黑白名单过滤、握手等操作；
- 步骤2完成之后，业务层的链路正式建立，将`SocketChannel`从主线程池的Reactor线程的多路复用器上摘除，重新注册到NIO线程池的线程上，用于处理I/O的读写操作。

注意:

> 服务器端的`ServerSocketChannel`只绑定到了`bossGroup`中的一个线程, 因此在调用`Java NIO`的Selector.select 处理客户端的连接请求时, 实际上是在一个线程中的, 所以对只有一个服务的应用来说, `bossGroup`设置多个线程是没有什么作用的, 反而还会造成资源浪费.

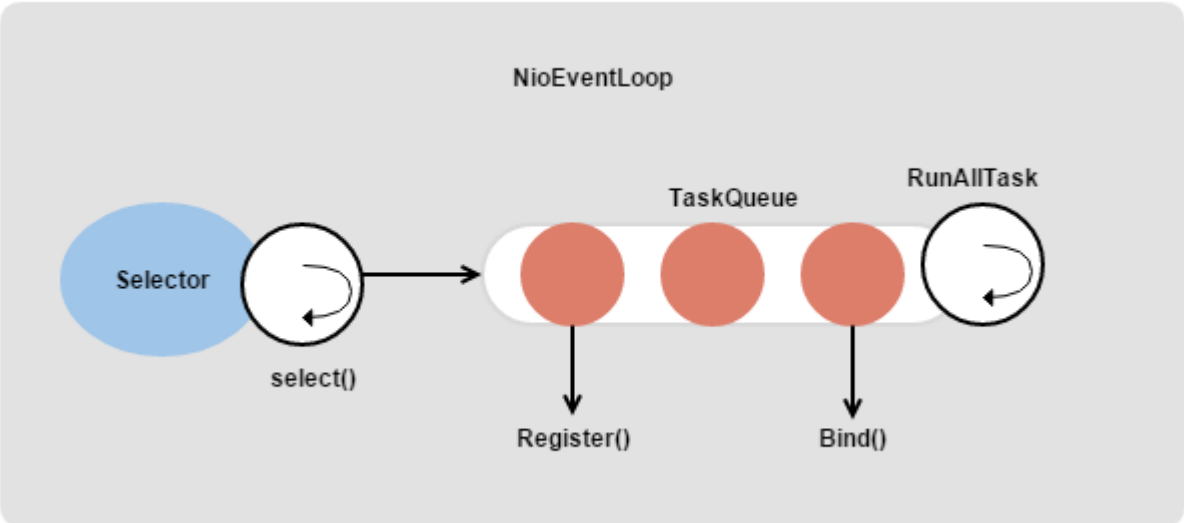
`NioEventLoopGroup`与 Reactor 线程模型的对应

```
//单线程模型
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup)
 .channel(NioServerSocketChannel.class)
 ...
```

```
//多线程模型
EventLoopGroup bossGroup = new NioEventLoopGroup(1);
EventLoopGroup workerGroup = new NioEventLoopGroup();
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup, workerGroup)
 .channel(NioServerSocketChannel.class)
 ...
```

```
//主从多线程模型
EventLoopGroup bossGroup = new NioEventLoopGroup(4);
EventLoopGroup workerGroup = new NioEventLoopGroup();
ServerBootstrap b = new ServerBootstrap();
b.group(bossGroup, workerGroup)
 .channel(NioServerSocketChannel.class)
 ...
```

3.2.3 `EventLoop`



`NioEventLoop`主要干两件事:

1. IO 事件的处理
    1. 作为服务端Acceptor线程，负责处理客户端的请求接入；
    2. 作为客户端Connecor线程，负责注册监听连接操作位，用于判断异步连接结果；

3. 作为IO线程，监听网络读操作位，负责从SocketChannel中读取报文；
4. 作为IO线程，负责向SocketChannel写入报文发送给对方，如果发生写半包，会自动注册监听写事件，用于后续继续发送半包数据，直到数据全部发送完成；

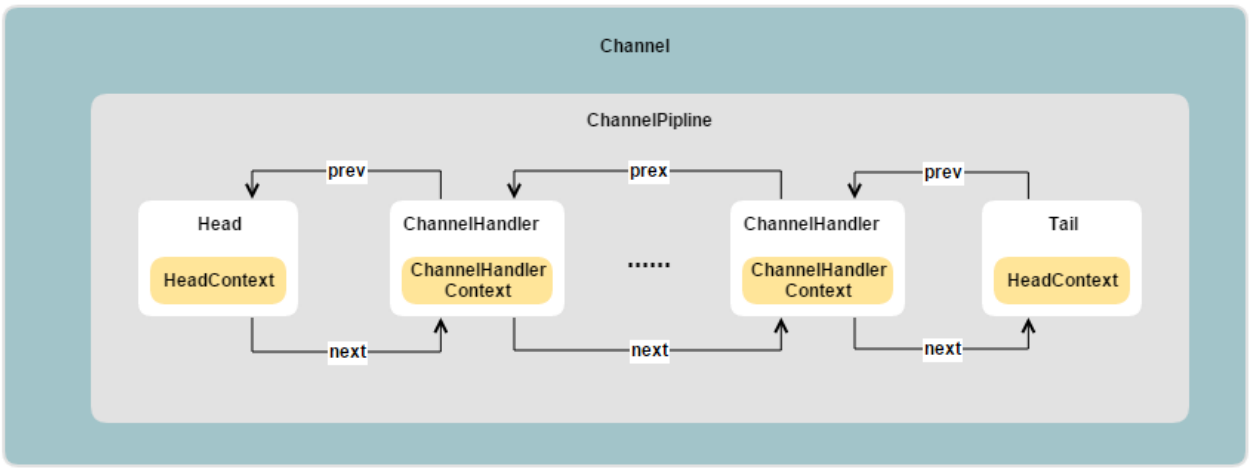
2. 非IO任务

1.作为定时任务线程，可以执行定时任务，例如链路空闲检测和发送心跳消息等；

2.作为线程执行器可以执行普通的任务线程（Runnable）。

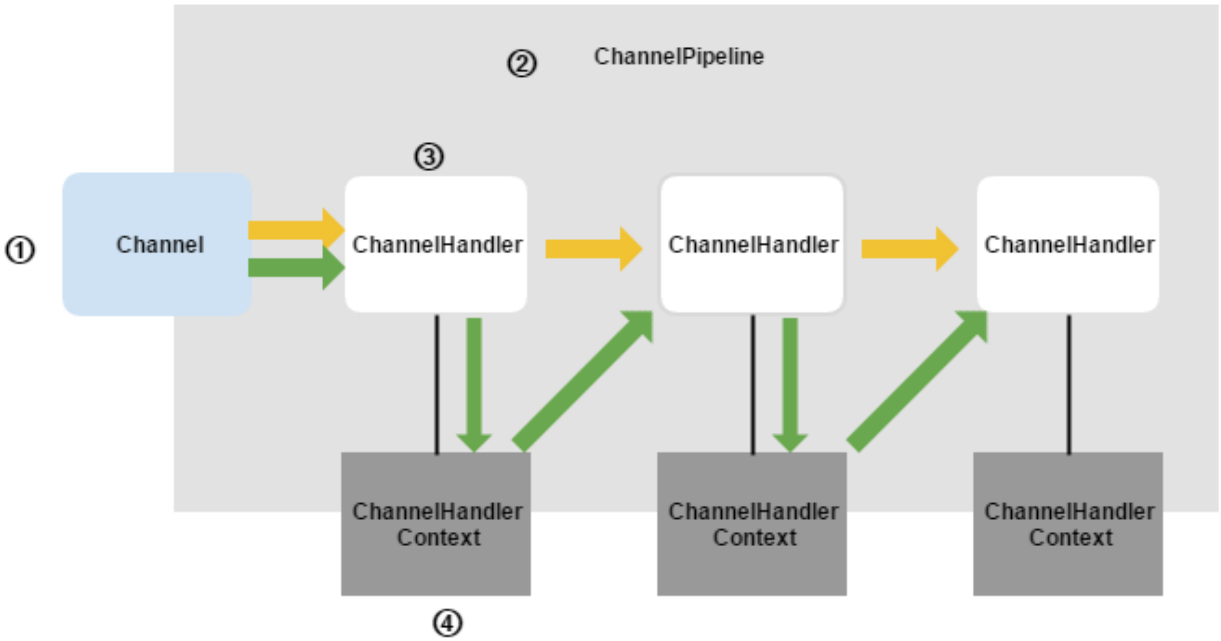
```
//为了保证定时任务的执行不会因为过度挤占IO事件的处理，Netty提供了IO执行比例供用户设置，用户可以设置分
//配给IO的执行比例，防止因为海量定时任务的执行导致IO处理超时或者积压。默认是1:1
final int ioRatio = this.ioRatio;//默认为50
if (ioRatio == 100) {
    try {
        processSelectedKeys();
    } finally {
        // Ensure we always run tasks.
        runAllTasks();
    }
} else {
    final long ioStartTime = System.nanoTime();
    try {
        processSelectedKeys();
    } finally {
        // Ensure we always run tasks.
        final long ioTime = System.nanoTime() - ioStartTime;
        runAllTasks(ioTime * (100 - ioRatio) / ioRatio);
    }
}
```

3.2.4 Channel，`ChannelHandler`，`ChannelPipeline`，`ChannelHandlerContext`



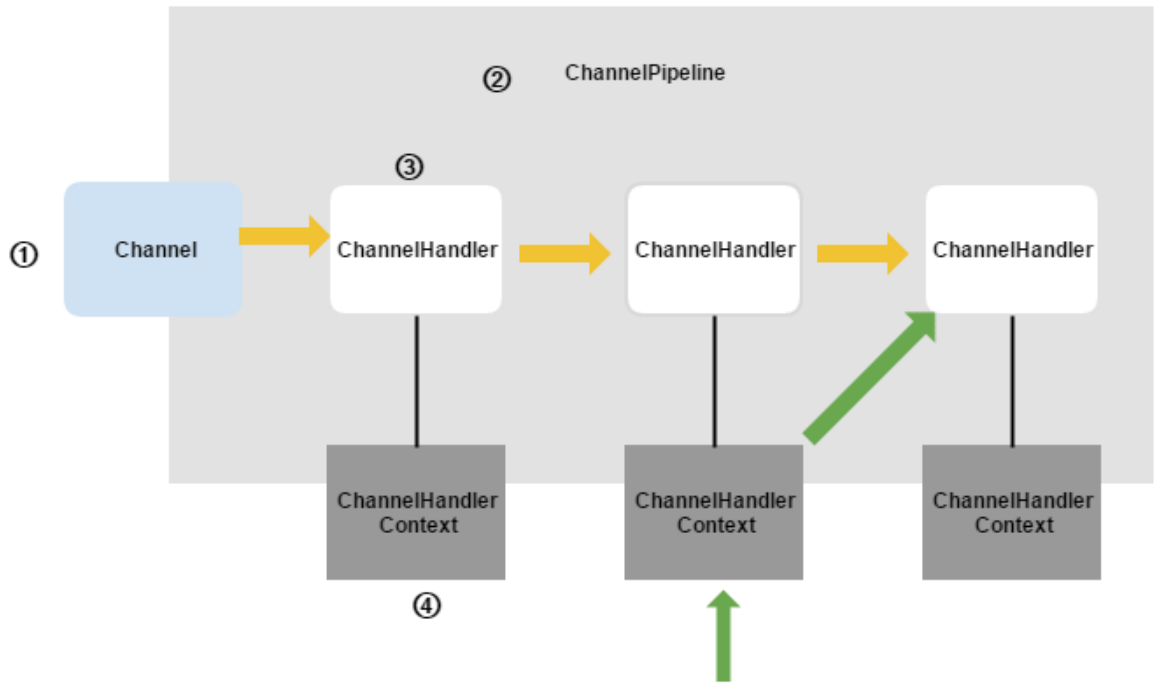
- 事件传播1：从`Channel`或者`ChannelPipeline`进行事件传播会把事件在整个管道中传播如下

```
hannelHandlerContext ctx = context;
ChannelPipeline pipeline = ctx.pipeline(); //1
pipeline.write(Unpooled.copiedBuffer("Netty in Action", CharsetUtil.UTF_8));
```

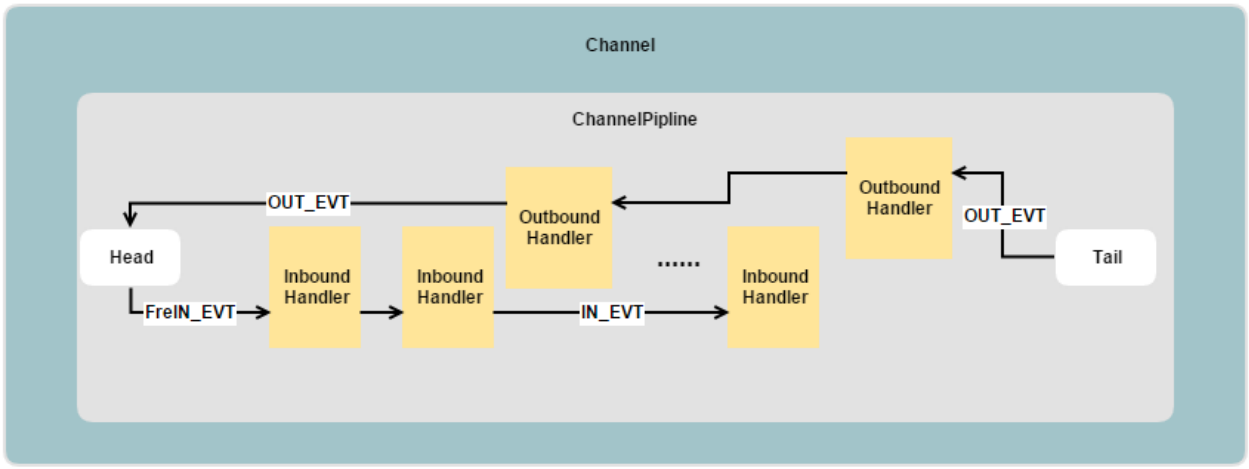


- 事件传播2：从特定的 `ChannelHandler` 进行事件传播如下：

```
ChannelHandlerContext ctx = context;
ctx.write(Unpooled.copiedBuffer("Netty in Action",CharsetUtil.UTF_8));
```



- 更为详细的事件传播：`ChannelInboundHandler`和`ChannelOutboundHandler`入站和出站处理

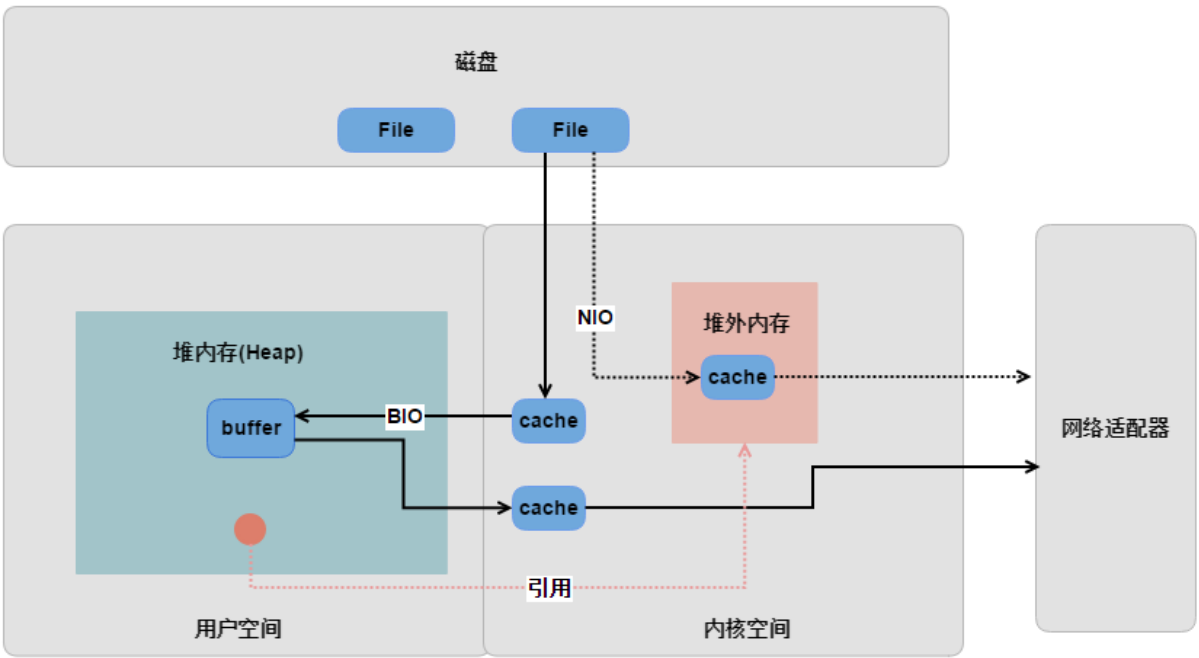


四.高性能Netty

4.1.非阻塞事件驱动框架

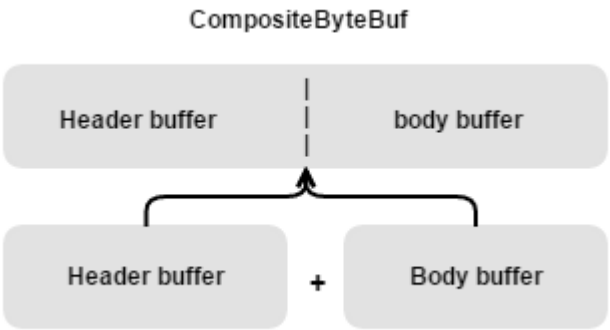
> Netty的IO线程`NioEventLoop`由于聚合了多路复用器Selector，可以同时并发处理成百上千个客户端Channel，由于读写操作都是非阻塞的，这就可以充分提升IO线程的运行效率，避免由于频繁IO阻塞导致的线程挂起。

4.2 零拷贝



> Netty的接收和发送`ByteBuffer`采用DIRECT BUFFERS，直接使用堆外直接内存进行Socket读写。

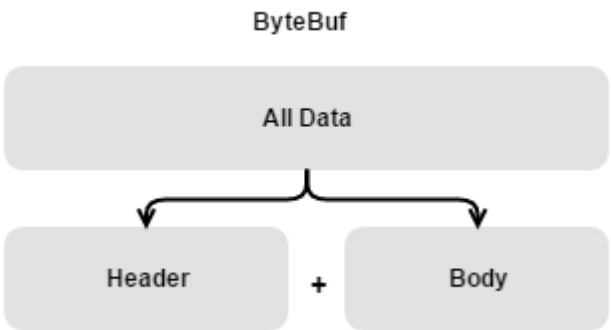
4.2.2 聚合零拷贝




```
ByteBuf header = ...
ByteBuf body = ...

CompositeByteBuf compositeByteBuf = Unpooled.compositeBuffer();
compositeByteBuf.addComponents(true, header, body);
```

虽然看起来`CompositeByteBuf`是由两个`ByteBuf`组合而成的，不过在`CompositeByteBuf`内部，这两个`ByteBuf`都是单独存在的，`CompositeByteBuf`只是逻辑上是一个整体。如下：

4.2.3 通过`CompositeByteBuf`实现零拷贝（分解）




```
ByteBuf byteBuf = ...
ByteBuf header = byteBuf.slice(0, 5);
ByteBuf body = byteBuf.slice(5, 10);
```

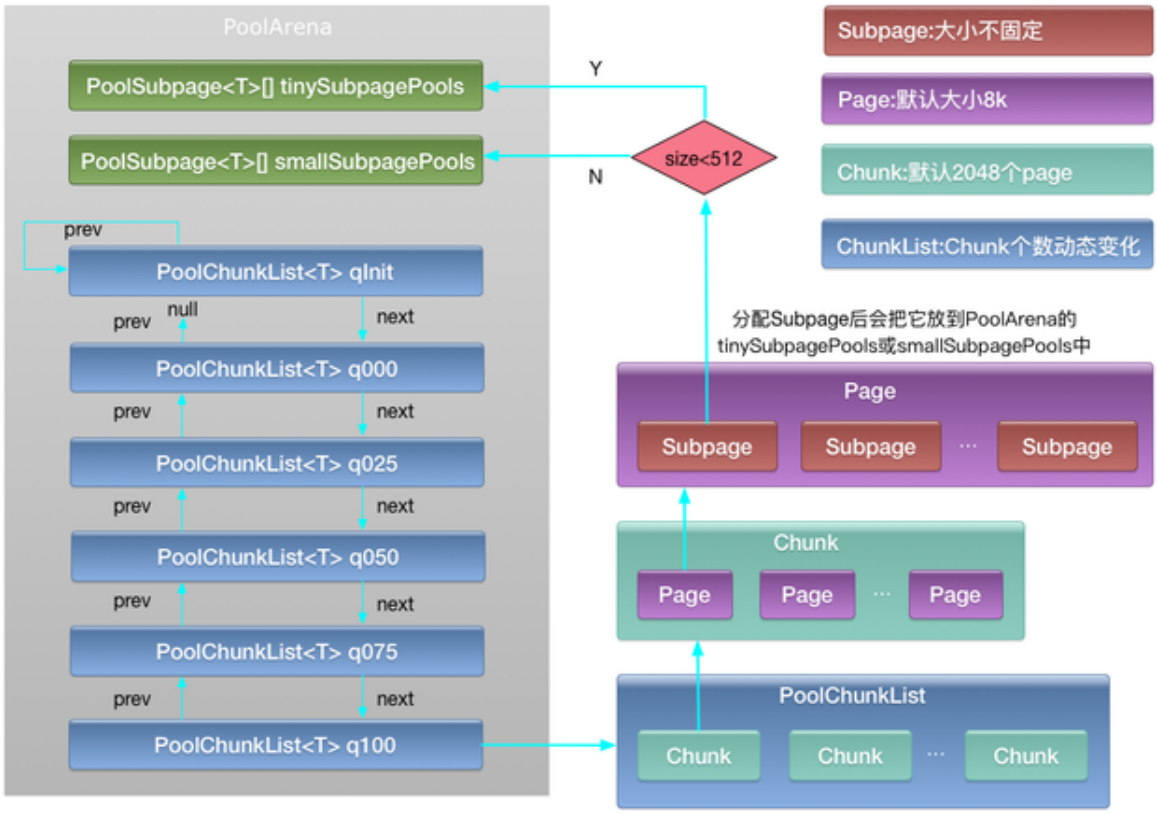
4.2.4 通过 wrap 操作实现零拷贝(包装)

```
byte[] bytes = ...
ByteBuf byteBuf = Unpooled.buffer();
byteBuf.writeBytes(bytes);

//Netty
byte[] bytes = ...
ByteBuf byteBuf = Unpooled.wrappedBuffer(bytes);
```

4.2.5 通过 `FileRegion` 文件通道传输

```
//传统io
byte[] temp = new byte[1024];
FileInputStream in = new FileInputStream(srcFile);
FileOutputStream out = new FileOutputStream(destFile);
int length;
while ((length = in.read(temp)) != -1) {
    out.write(temp, 0, length);
}

//netty
RandomAccessFile srcFile = new RandomAccessFile(srcFileName, "r");
FileChannel srcFileChannel = srcFile.getChannel();

RandomAccessFile destFile = new RandomAccessFile(destFileName, "rw");
FileChannel destFileChannel = destFile.getChannel();

long position = 0;
long count = srcFileChannel.size();
srcFileChannel.transferTo(position, count, destFileChannel);
```

4.3 内存池的使用

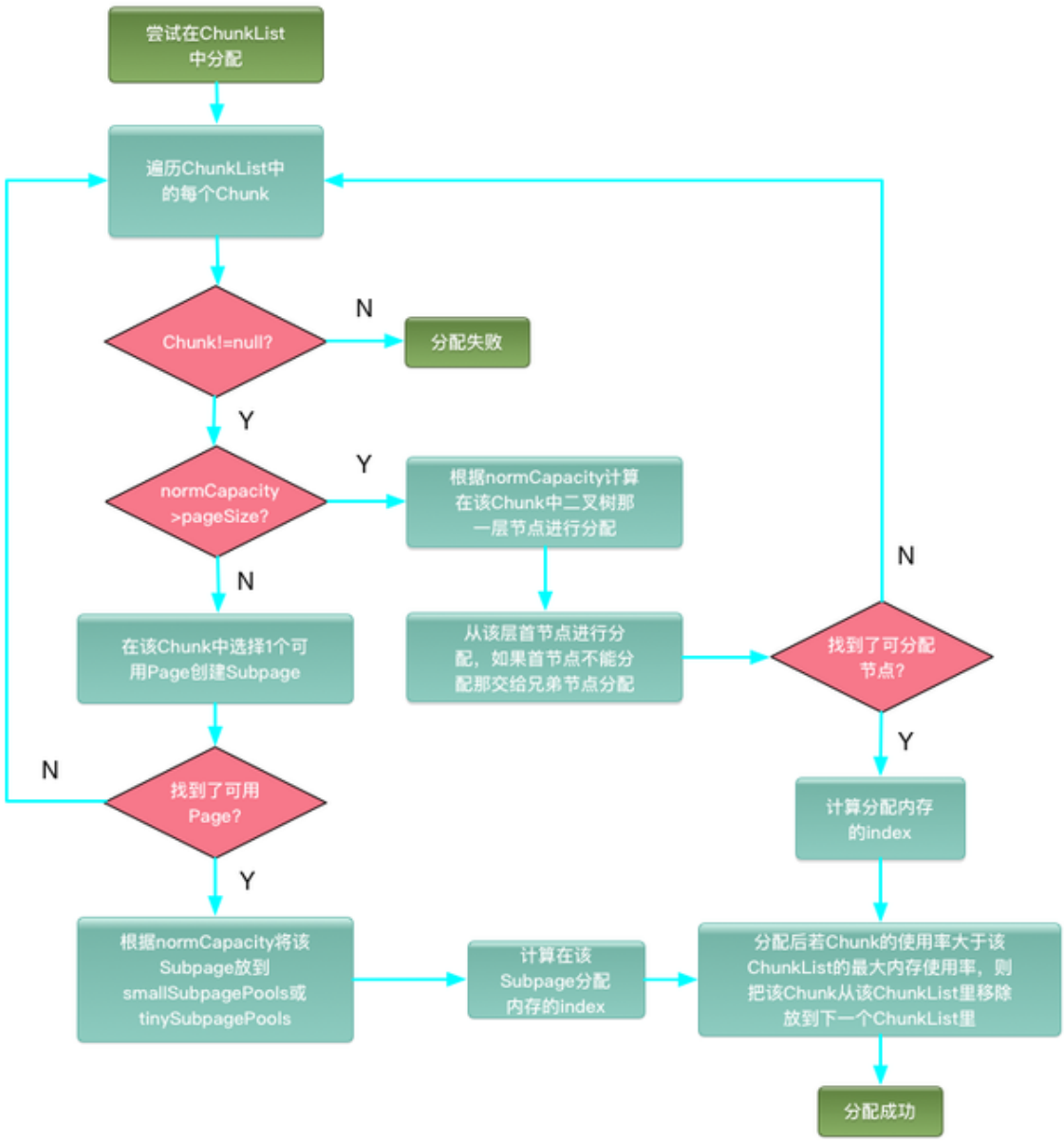



```
PoolChunkList<T> qInit：存储内存利用率0-25%的chunk
PoolChunkList<T> q000：存储内存利用率1-50%的chunk
PoolChunkList<T> q025：存储内存利用率25-75%的chunk
PoolChunkList<T> q050：存储内存利用率50-100%的chunk
PoolChunkList<T> q075：存储内存利用率75-100%的chunk
PoolChunkList<T> q100：存储内存利用率100%的chunk
```

`PoolArena`中申请内存:

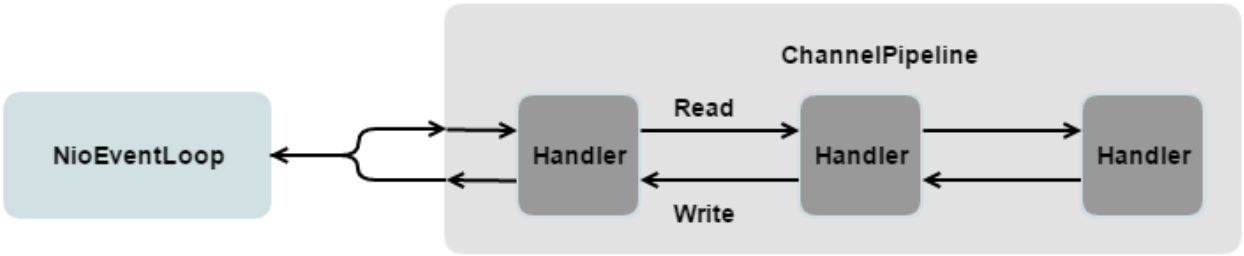
- 对于小于pageSize大小的内存，会在`tinySubpagePools`或s`mallSubpagePools`中分配，`tinySubpagePools`用于分配小于512字节的内存，`smallSubpagePools`用于分配大于512小于pageSize的内存。
- 对于大于pageSize小于chunkSize大小的内存，会在PoolChunkList的Chunk中分配。
- 对于大于`chunkSize`大小的内存，直接创建非池化Chunk来分配内存，并且该Chunk不会放在内存池中重用。

`q050、q025、q000、qInit、q075`这些PoolChunkList里申请内存:



4.4.无锁化的串行设计理念

> 为了尽可能提升性能，Netty采用了串行无锁化设计，在IO线程内部进行串行操作，避免多线程竞争导致的性能下降。表面上看，串行化设计似乎CPU利用率不高，并发程度不够。但是，通过调整NIO线程池的线程参数，可以同时启动多个串行化的线程并行运行，这种局部无锁化的串行线程设计相比一个队列-多个工作线程模型性能更优



Netty的NioEventLoop读取到消息之后，直接调用ChannelPipeline的fireChannelRead(Object msg)，只要用户不主动切换线程，一直会由NioEventLoop调用到用户的Handler，期间不进行线程切换，这种串行化处理方式避免了多线程操作导致的锁的竞争，从性能角度看是最优的。

4.5 高性能的序列化框架

> Netty默认提供了对Google Protobuf的支持，通过扩展Netty的编解码接口，用户可以实现其它的高性能序列化框架