Netty5.0 架构剖析和源码解读

作者：李林锋 © 版权所有　　email: neu_lilinfeng@sina.com

1. 概述

1.1. JAVA 的 IO 演进

1.1.1. 传统 BIO 通信的弊端

在 JDK 1.4 推出 JAVA NIO1.0 之前，基于 JAVA 的所有 Socket 通信都采用了同步阻塞模式（BIO），这种一请求一应答的通信模型简化了上层的应用开发，但是在可靠性和性能方面存在巨大的弊端。所以，在很长一段时间，大型的应用服务器都采用 C 或者 C++开发。当并发访问量增大、响应时间延迟变大后，采用 JAVA BIO 作为服务端的软件只有通过硬件不断的扩容来满足访问量的激增，它大大增加了企业的成本，随着集群的膨胀，系统的可维护性也面临巨大的挑战，解决这个问题已经刻不容缓。

首先，我们通过下面这幅图来看下采用 BIO 的服务端通信模型：采用 BIO 通信模型的

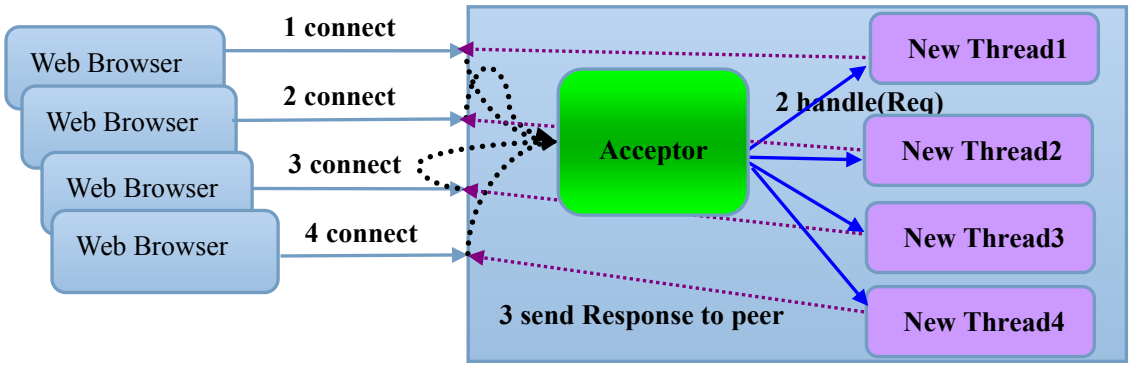


图 1.1.1-1　BIO 通信模型图

服务端，通常由一个独立的 Acceptor 线程负责监听客户端的连接，接收到客户端连接之后为客户端连接创建一个新的线程处理请求消息，处理完成之后，返回应答消息给客户端，线程销毁，这就是典型的一请求一应答模型。该架构最大的问题就是不具备弹性伸缩能力，当并发访问量增加后，服务端的线程个数和并发访问数成线性正比，由于线程是 JAVA 虚拟机非常宝贵的系统资源，当线程数膨胀之后，系统的性能急剧下降，随着并发量的继续增加，可能会发生句柄溢出、线程堆栈溢出等问题，并导致服务器最终宕机。

有读者可能有疑问：为什么不让一个线程处理多个 Socket 连接，这样不就能打破一连接一线程模型吗？由于 java.net.Socket 通过 java.io.InputStream 和 java.io.OutputStream 来进行网络读写操作，InputStream 和 OutputStream 的读写操作都是阻塞模式，所以，当某个 Socket 链路的读写操作没有完成时，排在后面的 Socket 连接是无法得到处理的，长时间的等待可能会导致超时，因此，在同步阻塞模型下，一个线程处理多个客户端连接没有意义，反而会导致后面排队的 Socket 连接处理不及时引起客户端超时，所以通常会采用每个 Socket 链路独占一个线程的模型。

```
/**
 * Reads the next byte of data from the input stream. The value byte is
 * returned as an <code>int</code> in the range <code>0</code> to
 * <code>255</code>. If no byte is available because the end of the stream
 * has been reached, the value <code>-1</code> is returned. This method
 * blocks until input data is available, the end of the stream is detected,
 or an exception is thrown.
 *
 * <p> A subclass must provide an implementation of this method.
 *
 * @return     the next byte of data, or <code>-1</code> if the end of the
 *             stream is reached.
 * @exception  IOException  if an I/O error occurs.
 */
public abstract int read() throws IOException;
```

图 1.1.1-2　InputStream 阻塞读

后来针对传统的一连接一线程的模型进行了优化，采用线程池和任务队列实现一种叫做伪异步的 IO 通信框架，它的架构图如下：

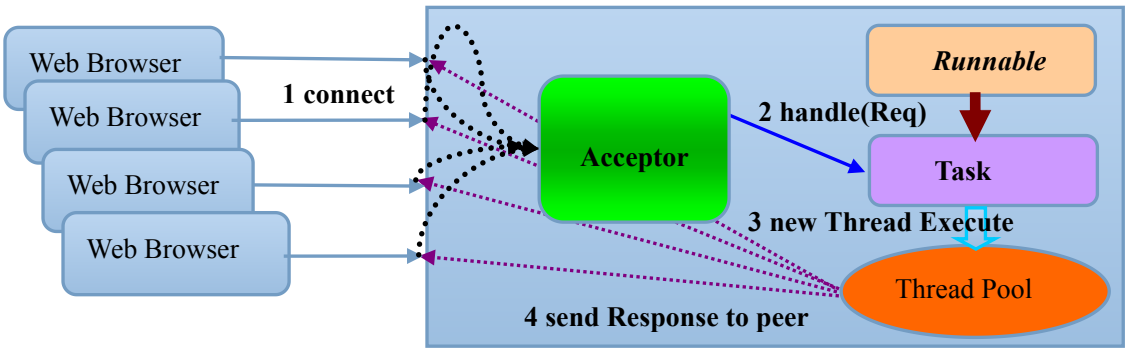


图 1.1.1-3 伪异步服务端框架

服务端线程接收到客户端连接之后，不创建独立的线程，而是将 Socket 连接封装成 Task，将 Task 放入线程池的任务队列中执行，这样就可以有效控制线程的规模，防止线程膨胀导致的系统崩溃，利用线程池，可以重用线程，性能相比于传统的一连接一线程有很大提升。

伪异步通信框架能够缓解 BIO 面临的问题，但是无法从根本上解决问题，由于 IO 的读写操作会被阻塞，当并发量增加或者网络 IO 时延增大之后，线程的执行时间会被拉长，它导致缓存在任务队列中的任务不断堆积，最终导致内存溢出或者拒绝新任务的执行。

由于网络的时延、客户端的执行速度和服务器的处理能力不同，导致网络 IO 的执行时间不可控，如果 IO 读写被阻塞，阻塞时间往往也是不可控的（或者超时），它会导致 IO 线程的不可预期性阻塞，降低系统的处理能力和网络吞吐量。在大规模高并发、高性能的服务器端，使用 JAVA 的同步 IO 来构建服务端是无法满足性能、可扩展性和可靠性要求的。

1.1.2. Linux 的网络 IO 模型简介

Linux 的内核将所有外部设备都可以看做一个文件来操作，那么我们对与外

部设备的操作都可以看做对文件进行操作。我们对一个文件的读写，都通过调用内核提供的系统调用；内核给我们返回一个 file descriptor（fd,文件描述符）。而对一个 socket 的读写也会有相应的描述符，称为 socketfd(socket 描述符)，描述符就是一个数字，指向内核中一个结构体（文件路径，数据区等一些属性）。

根据 Unix 网络编程对 IO 模型的分类，Unix 提供了五种 IO 模型，分别如下：

1. 阻塞 IO 模型：最常用的 I/O 模型就是阻塞 I/O 模型，缺省情形下，所有文件操作都是阻塞的。我们以套接口为例来讲解此模型。在进程空间中调用 recvfrom，其系统调用直到数据报到达且被拷贝到应用进程的缓冲区中或者发生错误才返回，期间一直在等待。我们就说进程在从调用 recvfrom 开始到它返回的整段时间内是被阻塞的。

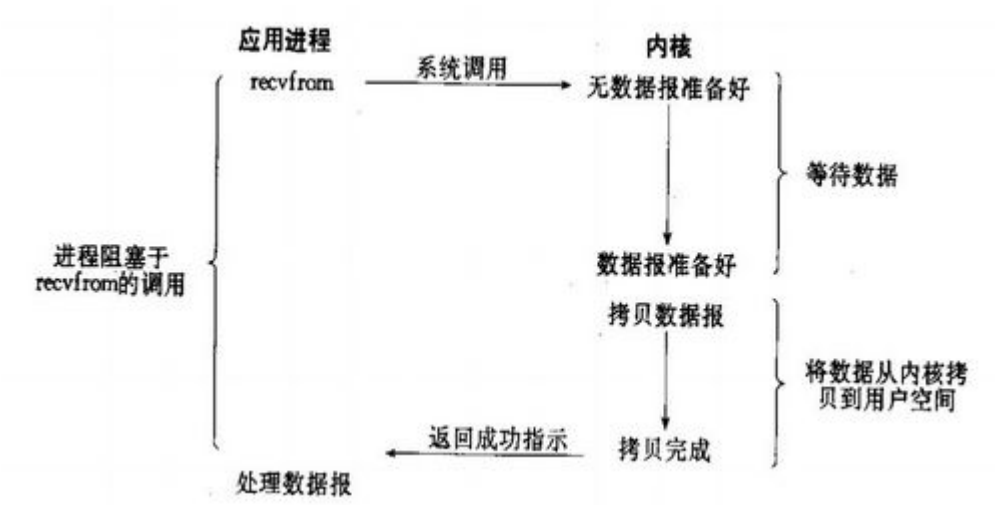


图 1.1.2-1 阻塞 IO 模型

2. 非阻塞 IO 模型：recvfrom 从应用层到内核的时候，如果该缓冲区没有数据的话，就直接返回一个 EWOULDBLOCK 错误，一般都对非阻塞 I/O 模型进行轮询检查这个状态，看内核是不是有数据到来。

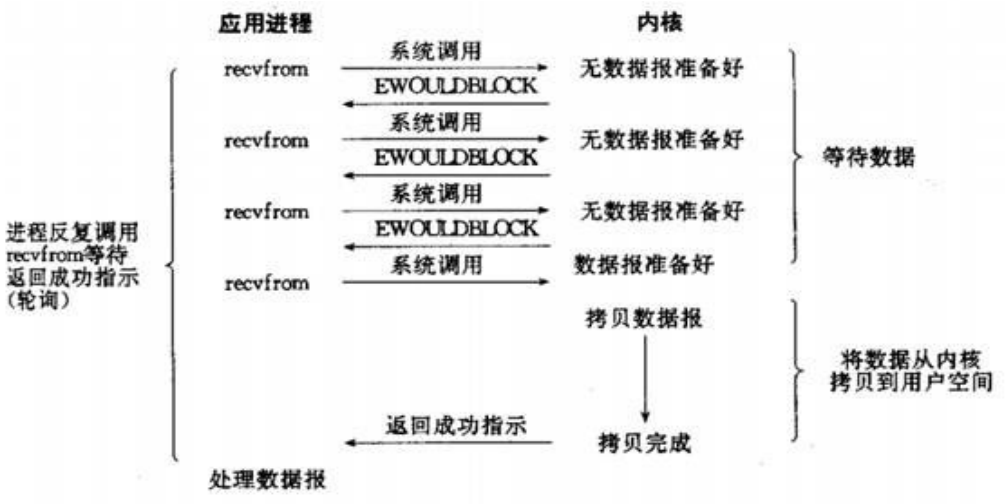


图 1.1.2-2 非阻塞 IO

3. IO 复用模型：Linux 提供 select/poll，进程通过将一个或多个 fd 传递给 select 或 poll 系统调用，阻塞在 select;这样 select/poll 可以帮我们侦测许多 fd 是否就绪。但是 select/poll 是顺序扫描 fd 是否就绪，而且支持的 fd 数量有限。linux 还提供了一个 epoll 系统调用，epoll 是基于事件驱动方式，而不是顺序扫描,当有 fd 就绪时，立即回调函数 rollback。

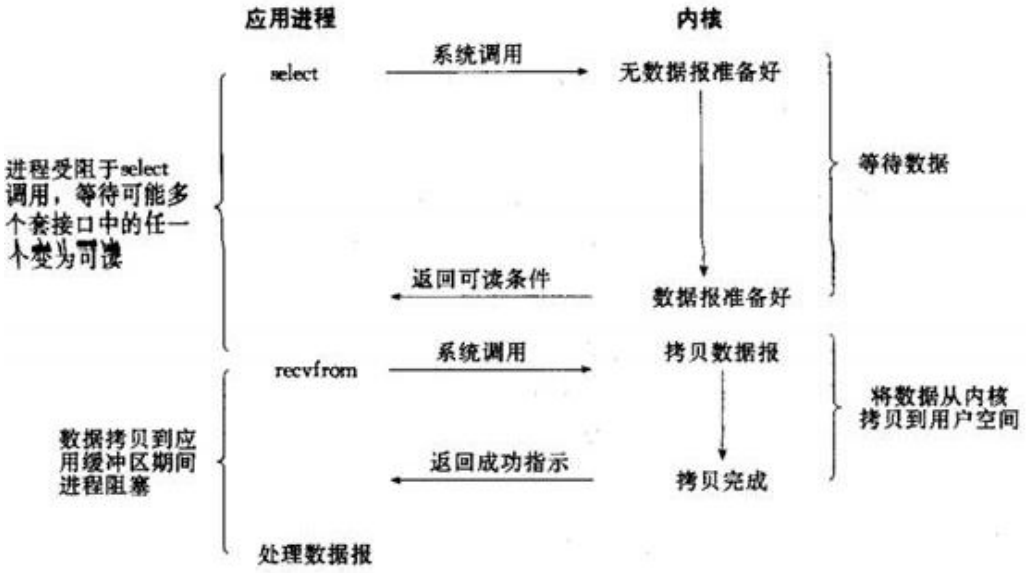


图 1.1.2-3 IO 复用模型

4. 信号驱动 IO 模型:首先开启套接口信号驱动 I/O 功能，并通过系统调用 sigaction 执行一个信号处理函数（此系统调用立即返回，进程继续工作，它是非阻塞的）。当数据准备就绪时，就为该进程生成一个 SIGIO 信号。随即可以在信号处理程序中调用 recvfrom 来读数据，并通知主循环函数处理数据。

图 1.1.2-4 信号驱动 IO

5. 异步 IO：告知内核启动某个操作，并让内核在整个操作完成后(包括将数据从内核拷贝到用户自己的缓冲区)通知我们。这种模型与信号驱动模型的主要区别是：信号驱动 I/O：由内核通知我们何时可以启动一个 I/O 操作；异步 I/O 模型：由内核通知我们 I/O 操作何时完成。

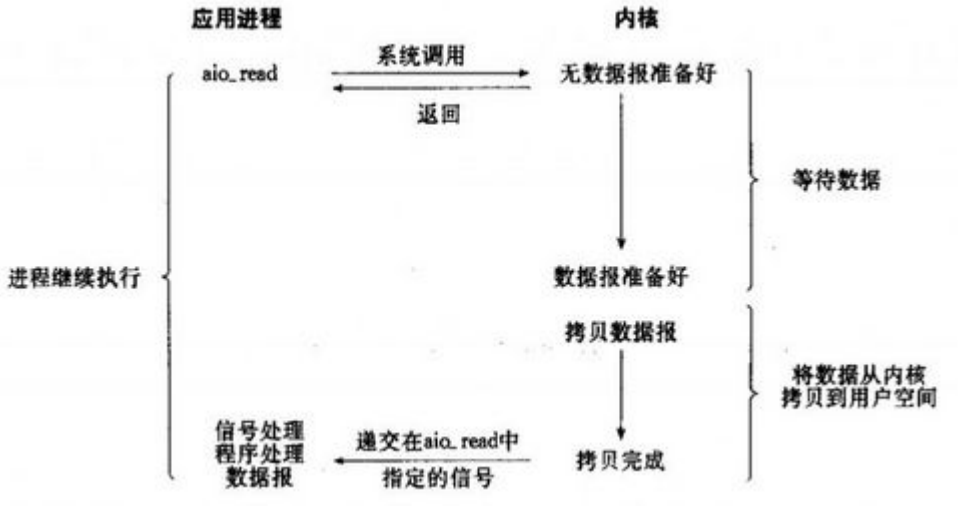


图 1.1.2-5 非阻塞 IO

1.1.3. IO 复用技术介绍

在 IO 编程过程中，当需要处理多个请求时，可以使用多线程和 IO 复用的方式进行处理。上图介绍了整个 IO 复用的过程，它通过把多个 IO 的阻塞复用到一个 select 之类的阻塞上，从而使得系统在单线程的情况下同时支持处理多个请求。和多线程/进程比较，I/O 多路复用的最大优势是系统开销小，系统不需要建立新的进程或者线程，也不必维护这些线程和进程。IO 复用常见的应用场景：

1. 服务器需要同时处理多个处于监听状态和多个连接状态的套接字；
2. 服务器需要处理多种网络协议的套接字。

目前支持 I/O 复用的系统调用有 select、pselect、poll、epoll，在 Linux 网络编程过程中，很长一段时间都使用 select 做事件触发，然而 select 逐渐暴露出了一些缺陷，使得 linux 不得不在新的内核中寻找出替代方案，那就是 epoll。其实，epoll 与 select 原理类似，只不过，epoll 作出了一些重大改进，具体如下：

1. 支持一个进程打开的 socket 描述符(FD)不受限制（仅受限于操作系统的最大文件句柄数）：select 有个比较大的缺陷就是一个进程所打开的 FD 是有一定限制的，由 FD_SETSIZE 设置，默认值是 2048。对于那些需要支持的上万连接数目的大型服务器来说显然太少了。这时候你可以选择修改这个宏然后重新编译内核，不过资料也同时指出这样会带来网络效率的下降，二是可以选择多进程的解决方案(传统的 Apache 方案)，不过虽然 linux 上面创建进程的代价比较小，但仍旧是不可忽视的，加上进程间数据同步远比不上线程间同步的高效，所以也不是一种完美的方案。不过 epoll 则没有这个限制，它所支持的 FD 上限是最大可以打开文件的数目，这个数字一般远大于 2048,

举个例子,在 1GB 内存的机器上大约是 10 万左右，具体数目可以 cat /proc/sys/fs/file-max 察看,一般来说这个数目和系统内存关系很大；

2. IO 效率可能随 FD 数目增加而线性下降：传统的 select/poll 另一个致命弱点就是当你拥有一个很大的 socket 集合，由于网络延时，任一时间只有部分的 socket 是"活跃"的，但是 select/poll 每次调用都会线性扫描全部的集合，导致效率呈现线性下降。但是 epoll 不存在这个问题，它只会对"活跃"的 socket 进行操作---这是因为在内核实现中 epoll 是根据每个 fd 上面的 callback 函数实现的。那么，只有"活跃"的 socket 才会主动的去调用 callback 函数，其他 idle 状态 socket 则不会，在这点上，epoll 实现了一个"伪"AIO，因为这时候推动力在 os 内核。在一些 benchmark 中，如果所有的 socket 基本上都是活跃的---比如一个高速 LAN 环境，epoll 并不比 select/poll 效率高太多，相反，如果过多使用 epoll_ctl,效率相比还有稍微的下降。但是一旦使用 idle connections 模拟 WAN 环境,epoll 的效率就远在 select/poll 之上了；

3. 使用 mmap 加速内核与用户空间的消息传递：无论是 select,poll 还是 epoll 都需要内核把 FD 消息通知给用户空间，如何避免不必要的内存拷贝就很重要，在这点上，epoll 是通过内核于用户空间 mmap 同一块内存实现的；

4. epoll 的 API 更加简单:包括创建一个 epoll 描述符，添加监听事件，阻塞等待所监听的事件发生,关闭 epoll 描述符。

值得说明的是，用来克服 select/poll 缺点的方法不只有 epoll。epoll 只是一种 Linux 的实现方案，在 freeBSD 下有 kqueue，而 dev/poll 是最古老的 Solaris 的方案，使用难度依次递增。kqueue 是 freebsd 的宠儿，kqueue 实际上是一个功能相当丰富的 kernel 事件队列，它不仅仅是 select/poll 的升级，而且可以处理 signal、目录结构变化、进程等多种事件。Kqueue 是边缘触发的。/dev/poll 是 Solaris 的产物，是这一系列高性能 API 中最早出现的。Kernel 提供一个特殊的设备文件/dev/poll 。应用程序打开这个文件得到操纵 fd_set 的句柄，通过写入 pollfd 来修改它，一个特殊 ioctl 调用用来替换 select，不过由于出现的年代比较早，所以/dev/poll 的接口实现比较原始。

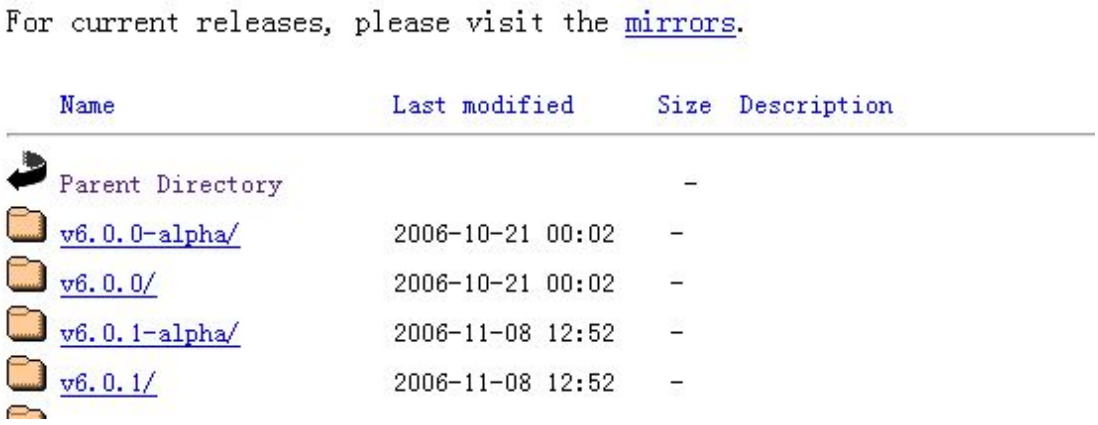
1.1.4. JAVA 的异步 IO

从 JDK1.4 开始，JDK 提供了一套专门的类库支持非阻塞 I/O，可以在 java.nio 包及其子包中找到相关的类和接口。由于这套 API 是新提供的 I/O API，因此，也叫 New I/O，这就是 JAVA NIO 的由来。非阻塞 IO API 由三个主要部分组成：缓冲区（Buffers）、通道（Channels）和 Selector 组成。

NIO 是基于事件驱动思想来实现的，它采用 Reactor 模式实现，主要用来解决 BIO（同步阻塞 IO）模型中一个服务端无法同时并发处理大量客户端连接的问题。NIO 基于 Selector 进行轮询，当 socket 有数据可读、可写、连接完成、新的 TCP 请求接入事件时，操作系统内核会触发 Selector 返回准备就绪的 SelectionKey 的集合，通过 SelectableChannel 进行读写操作。由于 JDK 的 Selector 底层基于 epoll 实现，因此不受 2048 连接数的限制，理论上可以同时处理操作系统最大文件句柄个数的连接。SelectableChannel 的读写操作都是异

步非阻塞的，当由于数据没有就绪导致读半包时，立即返回，不会同步阻塞等待数据就绪，当 TCP 缓冲区数据就绪之后，会触发 Selector 的读事件，驱动下一次读操作。因此，一个 Reactor 线程就可以同时处理 N 个客户端的连接，这就解决了之前 BIO 的一连接一线程的弊端，使 JAVA 服务端的并发读写能力得到极大的提升。

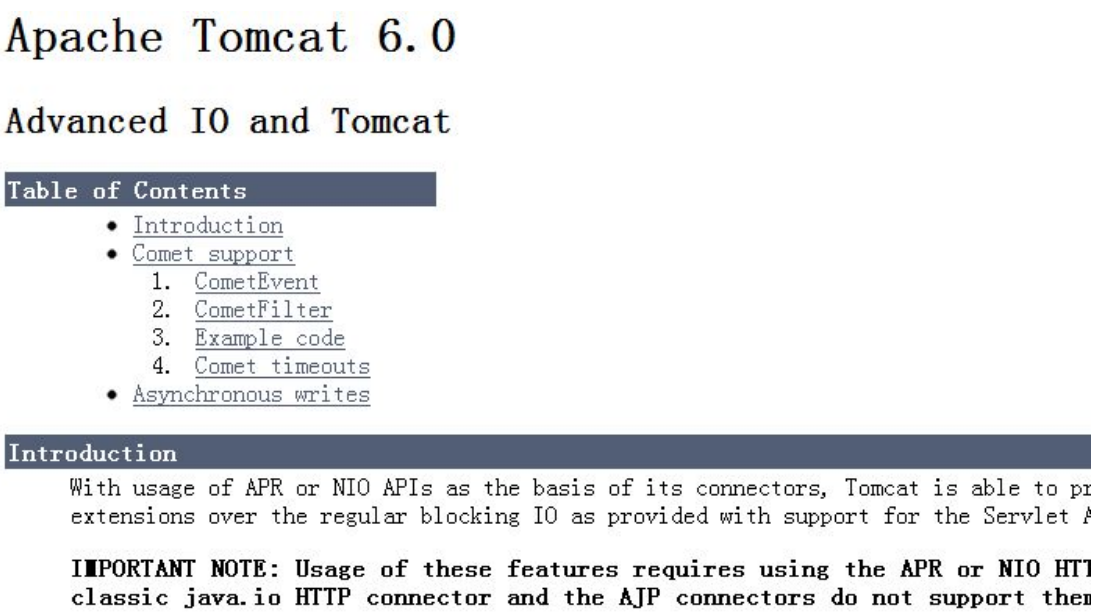
在 2004 年 JDK 提供 NIO 包以后的很长一段时间，NIO 并没有在业界得到大规模普及和应用，2006 年 10 月，Tomcat V6.0.0 版本正式发布，它提供了基于 NIO 的异步 HTTP Connector 能力，版本信息如下：


For current releases, please visit the mirrors.

Name	Last modified	Size	Description
Parent Directory		-	
v6.0.0-alpha/	2006-10-21 00:02	-	
v6.0.0/	2006-10-21 00:02	-	
v6.0.1-alpha/	2006-11-08 12:52	-	
v6.0.1/	2006-11-08 12:52	-	

图 1.1.4-1　Tomcat V6.0.0 归档日期

Tomcat 在 V6.0 版本开始对 NIO 提供支持，但是默认没有打开，需要通过修改 server.xml 配置来打开 NIO 功能，由于 Tomcat6.0 使用的是 Servlet2.5，因此并没有统一的规范来支持 NIO，Tomcat 的实现与 Jetty 的实现就不同。在 Tomcat6.0 中使用 NIO，除了修改配置以外，还需要额外修改代码，实现一个称为 event() 的方法来处理不同的事件，比较繁琐。因此，大多数使用者并没有使用 Tomcat 的 NIO 功能。


Apache Tomcat 6.0

Advanced IO and Tomcat

Table of Contents

- Introduction
- Comet support
  1. CometEvent
  2. CometFilter
  3. Example code
  4. Comet timeouts
- Asynchronous writes

Introduction

With usage of APR or NIO APIs as the basis of its connectors, Tomcat is able to pr
extensions over the regular blocking IO as provided with support for the Servlet A

IMPORTANT NOTE: Usage of these features requires using the APR or NIO HT
classic java.io HTTP connector and the AJP connectors do not support them

图 1.1.4-2 Tomcat 对 NIO 的支持

在 JAVA 企业领域，由于基于 NIO 的编程难度非常大，需要注意的细节和对编程技巧的要求都很高，传统的 JAVA 程序员很难驾驭，由于大多数的 JAVA 程序员对 Unix 的网络编程并不熟悉，这就意味着仅仅依靠 JDK 的 JAVA DOC 和示例很难编写出高质量、可以商用的 NIO 程序来。所以，在 NIO 推出 10 年后，很多公司仍然采用传统的 BIO 进行 JAVA 通信开发，JAVA NIO 的商业化普及方兴未艾。

1.1.5. 业界主流的 NIO 框架介绍

随着移动互联网的发展和大数据时代的到来，大规模分布式服务框架、分布式流计算框架已经成为架构主流，分布式服务节点之间的通信形式往往是内部长连接，例如阿里巴巴的 dubbo 协议、FaceBook 的 Thrift 协议，为了提升节点间的通信吞吐量、提升通信性能，目前主流的内部通信框架均使用 NIO 框架，对于大公司、技术积累比较深的团队可能会使用自研的 NIO 框架来满足个性化或者行业特殊的需求，但是大多数架构师会选择业界主流的 NIO 框架进行异步通信开发。

目前，业界主流的 NIO 框架主要有两款：Mina 和 Netty，两者都使用 Apache LICENSE-2.0 进行开源。不同之处是 Mina 是 Apache 基金会的官方 NIO 框架，Netty 之前是 Jboss 的 NIO 框架，后来脱离 Jboss 独立申请了 netty.io 域名，与 Jboss 脱离关系，并对版本进行了重构，导致 API 无法向上兼容。

Mina 和 Netty 还有一段历史渊源，Mina 最初版本的架构师是 Trustin Lee，后来，由于种种原因，Trustin Lee 离开了 Mina 社区加入到了 Netty 团队，重新开发了 Netty。很多读者会发现 Netty 中透着 Mina 的影子，两个框架的架构理念也有很多相似之处，甚至一些代码都非常相似，原因就在这里。

目前，Mina 和 Netty 的应用已经非常广泛，很多开源框架都使用两者做底层的 NIO 框架，例如 Hadoop 的通信组件 Avro 使用 Netty 做底层的通信框架，Openfire 则使用 Mina 做底层通信框架，两者的应用可以说是不分伯仲。

由于业界很少有针对 Netty 的架构和源码进行系统性分析的文章和书籍，很多架构师、设计师和 NIO 爱好者又想深入了解 Netty，本文作者从 2009 年开始 JAVA NIO 领域的设计和开发工作，并将其应用在核心的电信级商用软件中，经受了全球数百个商用局点和电信级海量数据的冲击和考验。从 2011 年开始研究并使用 Mina 和 Netty，对两者的架构和源码实现都非常熟悉。为了满足更多读者深入了解 Netty 的愿望，本文从架构和关键源码分析两部分入手，引领读者尽快入门，以期达到事半功倍的学习效果。

2. NIO 入门

2.1. NIO 服务端

首先，我们通过一个时序图来看下如何创建一个 NIO 服务端并启动监听，接收多个客户端的连接，进行消息的异步读写。

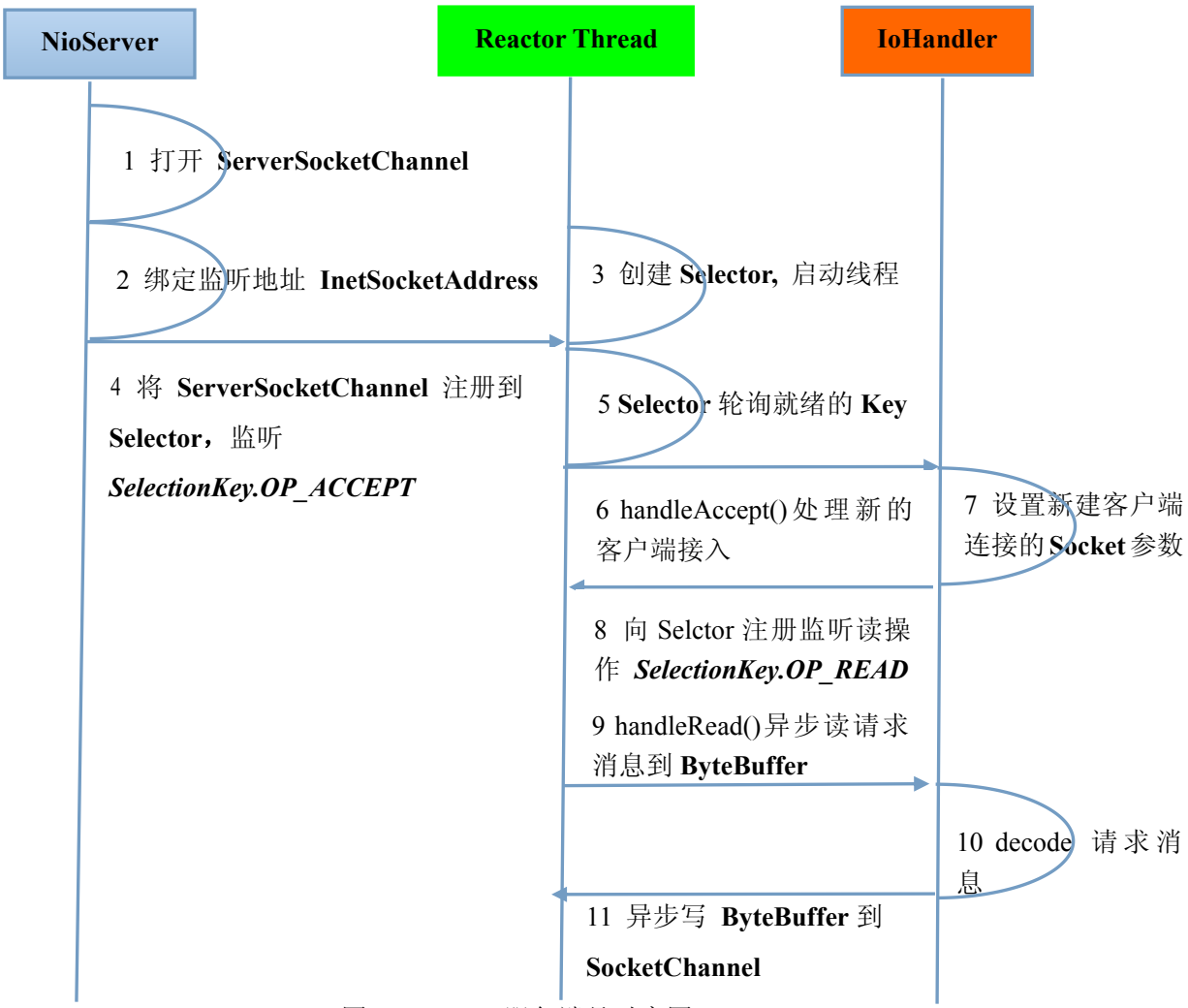


图 2.1-1 NIO 服务端是时序图

作为入门教程，上图仅仅描述了创建 NIO 服务端的最基础步骤，像半包读写、编解码、IO 线程上下文切换、异常处理、线程池等均没有描述，后续在分析 Netty 源码的时候，会做相关的介绍。

下面我们就通过代码来描述 NIO 服务端的创建过程：

步骤一：打开 ServerSocketChannel，用于监听客户端的连接，它是所有客户端连接的父管道，代码示例如下：

```
ServerSocketChannel acceptorSvr = ServerSocketChannel.open();
```

步骤二：绑定监听端口，设置连接为非阻塞模式，示例代码如下：

```
acceptorSvr.socket().bind(new InetSocketAddress(InetAddress.getByName("IP"), port));
acceptorSvr.configureBlocking(false);
```

步骤三：创建 Reactor 线程，创建多路复用器并启动线程，代码如下：

```
Selector selector = Selector.open();
New Thread(new ReactorTask()).start();
```

步骤四：将 ServerSocketChannel 注册到 Reactor 线程的多路复用器 Selector 上，监听 ACCEPT 事件，代码如下：

```
SelectionKey key = acceptorSvr.register( selector, SelectionKey.OP_ACCEPT, ioHandler);
```

步骤五：多路复用器在线程 run 方法的无限循环体内轮询准备就绪的 Key，代码如下：

```
int num = selector.select();
Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();
while (it.hasNext()) {
        SelectionKey key = (SelectionKey)it.next();
        // ... deal with I/O event ...
}
```

步骤六：多路复用器监听到有新的客户端接入，处理新的接入请求，完成 TCP 三次握手，建立物理链路，代码示例如下：

```
SocketChannel channel = svrChannel.accept();
```

步骤七：设置客户端链路的 TCP 参数，示例代码如下：

```
channel.configureBlocking(false);
channel.socket().setReuseAddress(true);
```

步骤八：将新接入的客户端连接注册到 Reactor 线程的多路复用器上，监听读操作，用来读取客户端发送的网络消息，代码如下：

```
SelectionKey key = socketChannel.register( selector, SelectionKey.OP_READ, ioHandler);
```

步骤九：异步读取客户端请求消息到缓冲区，示例代码如下：

```
int    readNumber =    channel.read(receivedBuffer);
```

步骤十：对 ByteBuffer 进行编解码，如果有半包消息指针 Reset，继续读取后续的报文，将解码成功的消息封装成 Task，投递到业务线程池中，进行业务逻辑编排，示例代码如下：

```
Object message = null;
while(buffer.hasRemain())
{
        byteBuffer.mark();
        Object message = decode(byteBuffer);
        if (message == null)
        {
            byteBuffer.reset();
            break;
        }
        messageList.add(message );
}
if (!byteBuffer.hasRemain())
    byteBuffer.clear();
else
    byteBuffer.compact();
if (messageList != null & !messageList.isEmpty())
{
    for(Object messageE : messageList)
        handlerTask(messageE);
}
```

步骤十一：将 POJO 对象 encode 成 ByteBuffer，调用 SocketChannel 的异步 write 接口，将消息异步发送给客户端，示例代码如下：

```
socketChannel.write(buffer);
```

注意：如果发送区 TCP 缓冲区满，会导致写半包，此时，需要注册监听写操作位，循环写，直到整包消息写入 TCP 缓冲区，此处不赘述，后续 Netty 源码分析章节会详细分析 Netty 的处理方式。

2.2. NIO 客户端

类似 NIO 服务端，我们首先通过时序图来看下 NIO 客户端的创建、与服务端连接的建立以及消息的读写。

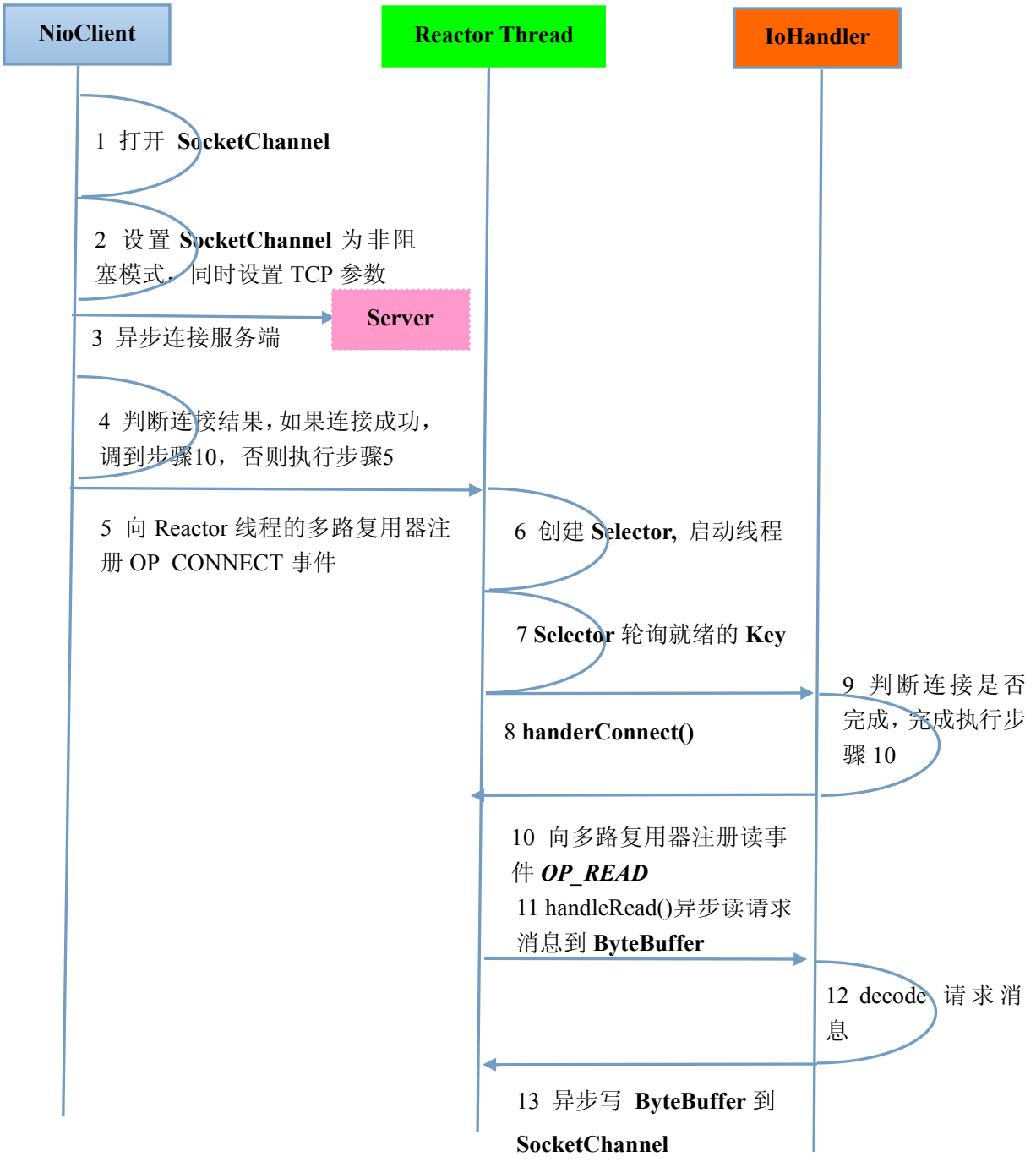


图 2.2-1 客户端创建时序图

步骤一：打开 SocketChannel，绑定客户端端口（可选，默认系统会随机分配一个可用的端口），示例代码如下：

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

步骤二：设置 SocketChannel 为非阻塞模式，同时设置客户端连接的 TCP 参数，示例代码如下：

```
clientChannel.configureBlocking(false);
socket.setReuseAddress(true);
socket.setReceiveBufferSize(BUFFER_SIZE);
socket.setSendBufferSize(BUFFER_SIZE);
```

步骤三：异步连接服务端，示例代码如下：

```
boolean connected = clientChannel.connect(new InetSocketAddress(“ip”,port));
```

步骤四：判断是否连接成功，如果连接成功，则直接注册读事件到多路复用器中，如果当前没有连接成功（异步连接，返回 false，说明客户端已经发送 sync 包，服务端没有返回 ack 包，物理链路还没有建立），示例代码如下：

```
if (connected)
{
      clientChannel.register( selector, SelectionKey.OP_READ, ioHandler);
}
else
{
      clientChannel.register( selector, SelectionKey.OP_CONNECT, ioHandler);
}
```

步骤五：向 Reactor 线程的多路复用器注册 OP_CONNECT 事件，监听服务端的 TCP ACK 应答，示例代码如下：

```
clientChannel.register( selector, SelectionKey.OP_CONNECT, ioHandler);
```

步骤六：创建 Reactor 线程，创建多路复用器并启动线程，代码如下：

```
Selector selector = Selector.open();
New Thread(new ReactorTask()).start();
```

步骤七：多路复用器在线程 run 方法的无限循环体内轮询准备就绪的 Key，代码如下：

```
int num = selector.select();
Set selectedKeys = selector.selectedKeys();
Iterator it = selectedKeys.iterator();
while (it.hasNext()) {
        SelectionKey key = (SelectionKey)it.next();
        // ... deal with I/O event ...
}
```

步骤八：接收 connect 事件进行处理，示例代码如下：

```
if (key.isConnectable())
   //handlerConnect();
```

步骤九：判断连接结果，如果连接成功，注册读事件到多路复用器，示例代码如下：

```
if (channel.finishConnect())
   registerRead();
```

步骤十：注册读事件到多路复用器：

```
clientChannel.register( selector, SelectionKey.OP_READ, ioHandler);
```

步骤十一：异步读客户端请求消息到缓冲区，示例代码如下：

```
int    readNumber =    channel.read(receivedBuffer);
```

步骤十二：对 ByteBuffer 进行编解码，如果有半包消息指针 Reset，继续读取后续的报文，将解码成功的消息封装成 Task，投递到业务线程池中，进行业务逻辑编排，示例代码如下：

```
Object message = null;
while(buffer.hasRemain())
{
        byteBuffer.mark();
        Object message = decode(byteBuffer);
        if (message == null)
        {
            byteBuffer.reset();
            break;
        }
        messageList.add(message );
}
if (!byteBuffer.hasRemain())
    byteBuffer.clear();
else
    byteBuffer.compact();
if (messageList != null & !messageList.isEmpty())
{
    for(Object messageE : messageList)
        handlerTask(messageE);
}
```

步骤十三：将 POJO 对象 encode 成 ByteBuffer，调用 SocketChannel 的异步 write 接口，将消息异步发送给客户端，示例代码如下：

```
socketChannel.write(buffer);
```

注意：客户端同样存在读写半包的问题，实际上，由于客户端和服务端对于读写的处理是一致的，因此，往往会抽象成统一的父类进行处理，无论是 Netty 还是 Mina，都采用了抽象复用的架构设计来降低代码的冗余。

3. Netty 源码分析

3.1. 服务端创建

3.1.1. 服务端启动辅助类 ServerBootstrap

当我们直接使用 JDK NIO 的类库开发基于 NIO 的异步服务端时，需要使用到多路复用器 Selector、ServerSocketChannel、SocketChannel、ByteBuffer、SelectionKey 等等，相比于传统的 BIO 开发，NIO 的开发要复杂很多，开发出稳定、高性能的异步通信框架，一直是个难题。

Netty 为了向使用者屏蔽 NIO 通信的底层细节，在和业务交互的边界做了封装，目的就是为了降低业务开发工作量，降低开发难度。ServerBootstrap 是 Socket 服务端的启动辅助类，业务只需要和这个类打交道就能开发出 NIO 的服务端。

首先通过构造函数创建 ServerBootstrap，我们会惊讶的发现 ServerBootstrap 只有一个无参的构造函数，作为启动辅助类这让人不可思议，因为它需要与多个其它组件或者类交互。ServerBootstrap 构造函数没有参数的根本原因是因为它的参数太多了，而且未来也可能会发生变化，为了解决这个问题，就需要引入 Builder 模式。《Effective Java》第二版第 2 条建议 遇到多个构造器参数时要考虑用构建器，关于多个参数构造函数的缺点和使用构建器的优点大家可以查阅《Effective Java》,在此不再详述。

随后，通常会创建两个 EventLoopGroup，代码如下图所示，至于为什么要初始化两个看上去一样的 NioEventLoopGroup，后续介绍 NioEventLoopGroup 的时候会做详细说明。

```
EventLoopGroup bossGroup = new NioEventLoopGroup();
EventLoopGroup workerGroup = new NioEventLoopGroup();
```

通过 ServerBootstrap 的构建方法 group 将两个 EventLoopGroup 传入，代码如下：

```
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup) {
    super.group(parentGroup);
    if (childGroup == null) {
        throw new NullPointerException("childGroup");
    }
    if (this.childGroup != null) {
        throw new IllegalStateException("childGroup set already");
    }
    this.childGroup = childGroup;
    return this;
}
```

，其中父 NioEventLoopGroup 被传入了父类构造函数中，代码如下：

Bootstrap.java | EchoServer.java | AbstractBootstrap.java

```
     */
    @SuppressWarnings("unchecked")
    public B group(EventLoopGroup group) {
        if (group == null) {
            throw new NullPointerException("group");
        }
        if (this.group != null) {
            throw new IllegalStateException("group set already");
        }
        this.group = group;
        return (B) this;
    }
```

父类 AbstractBootstrap 持有了一个 EventLoopGroup 类型的成员变量 NioEventLoopGroup。

线程组和线程类型设置完成后，需要设置服务端 Channel，Netty 通过 Channel 工厂类来创建不同类型的 Channel，对于服务端，需要创建 NioServerSocketChannel,所以，通过指定 Channel 类型的方式创建 Channel 工厂：

```
public ServerBootstrap channel(Class<? extends ServerChannel> channelClass) {
    if (channelClass == null) {
        throw new NullPointerException("channelClass");
    }
    return channelFactory(new ServerBootstrapChannelFactory<ServerChannel>(channelClass));
}
```

ServerBootstrapChannelFactory 是 ServerBootstrap 的内部静态类，职责是根据 Channel 的类型通过反射创建 Channel 的实例，服务端需要创建的是 NioServerSocketChannel 实例，代码如下：

```
@Override
public T newChannel(EventLoop eventLoop, EventLoopGroup childGroup) {
    try {
        Constructor<? extends T> constructor = clazz.getConstructor(EventLoop.class, EventLoopGroup.class);
        return constructor.newInstance(eventLoop, childGroup);
    } catch (Throwable t) {
        throw new ChannelException("Unable to create Channel from class " + clazz, t);
    }
}
```

指定 NioServerSocketChannel 后，需要设置 TCP 的一些参数，作为服务端，主要是要设置 TCP 的 backlog 参数，底层 C 的对应接口定义如下：

int listen(int fd, int backlog);

backlog 指定了内核为此套接口排队的最大连接个数，对于给定的监听套接口，内核要维护两个队列，未链接队列和已连接队列，根据 TCP 三路握手过程中三个分节来分隔这两个队列。服务器处于 listen 状态时收到客户端 syn 分节(connect)时在未完成队列中创建一个新的条目，然后用三路握手的第二个分节即服务器的 syn 响应及对客户端 syn 的 ack,此条目在第三个分节到达前(客户端对服务器 syn 的 ack)一直保留在未完成连接队列中，如果三路握手完成，该条目将从未完成连接队列搬到已完成连接队列尾部。当进程调用 accept 时，从已完成队列中的头部取出一个条目给进程，当已完成队列为空时进程将睡眠，直到有条目在已完成连

接队列中才唤醒。backlog 被规定为两个队列总和的最大值，大多数实现默认值为 5，但在高并发 web 服务器中此值显然不够，lighttpd 中此值达到 128*8 。需要设置此值更大一些的原因是未完成连接队列的长度可能因为客户端 SYN 的到达及等待三路握手第三个分节的到达延时而增大。Netty 默认的 backlog 为 100，当然，用户可以修改默认值，后续会做详细介绍。

TCP 参数设置完成后，我们可以为启动辅助类和其父类分别指定 Handler，对于 Hanlder 后续会有专门章节详述，此处不展开讲。两类 Handler 的用途不同，子类中的 Hanlder 是 NioServerSocketChannel 对应的 ChannelPipeline 的 Handler，父类中的 Hanlder 是客户端新接入的连接 SocketChannel 对应的 ChannelPipeline 的 Handler。两者的区别我们通过下图来展示：

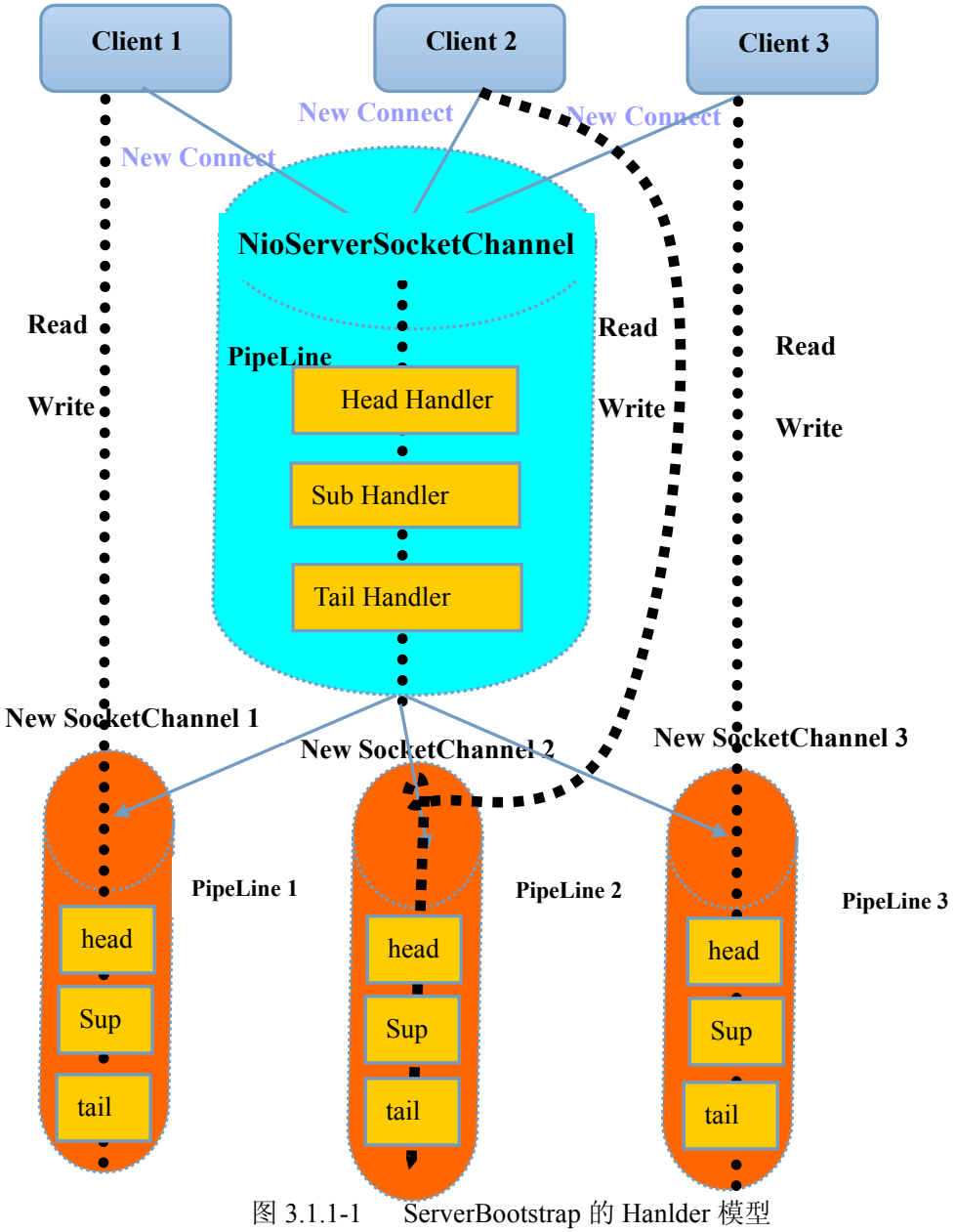


图 3.1.1-1　ServerBootstrap 的 Hanlder 模型

本质区别就是：ServerBootstrap 中的 Handler 是 NioServerSocketChannel 使用的，所有连接该监听端口的客户端都会执行它，父类 AbstractBootstrap 中的 Handler 是个工厂类，它为每个新接入的客户端都创建一个新的 Handler，代码如下：

```
.handler(new LoggingHandler(LogLevel.INFO))
.childHandler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(
                //new LoggingHandler(LogLevel.INFO),
                new EchoServerHandler());
    }
});
```

服务端启动的最后一步，就是绑定本地端口，启动服务，下面我们来分析下这部分代码：

```
private ChannelFuture doBind(final SocketAddress localAddress) {
    final ChannelFuture regFuture = initAndRegister();  NO. 1
    final Channel channel = regFuture.channel();
    if (regFuture.cause() != null) {
        return regFuture;
    }

    final ChannelPromise promise;
    if (regFuture.isDone()) {   NO. 2
        promise = channel.newPromise();
        doBind0(regFuture, channel, localAddress, promise);
    } else {
        // Registration future is almost always fulfilled already, but just in case it's not.
        promise = new DefaultChannelPromise(channel, GlobalEventExecutor.INSTANCE);
        regFuture.addListener(new ChannelFutureListener() {   NO. 3
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                doBind0(regFuture, channel, localAddress, promise);
            }
        });
    }

    return promise;
}
```

我们先看下 NO.1， 首先创建 Channel，createChannel 由子类 ServerBootstrap

```
final ChannelFuture initAndRegister() {
    Channel channel;
    try {
        channel = createChannel();
    } catch (Throwable t) {
        return VoidChannel.INSTANCE.newFailedFuture(t);
    }
```

实现，创建新的 NioServerSocketChannel，它有两个参数，参数 1 是从父类的 NIO 线程池中顺序获取一个 NioEventLoop，它就是服务端用于监听和接收客户端连接的 Reactor 线程。第二个参数就是所谓的 workerGroup 线程池，它就是处理 IO 读写的 Reactor 线程组，后续会详细介绍它的实现和工作原理。

```
@Override
Channel createChannel() {
    EventLoop eventLoop = group().next();
    return channelFactory().newChannel(eventLoop, childGroup);
}
```

NioServerSocketChannel 创建成功后对它进行初始化，初始化工作主要有三点：

1. 设置 Socket 参数和 NioServerSocketChannel 的附加属性，代码如下：

```
synchronized (options) {
    channel.config().setOptions(options);
}

final Map<AttributeKey<?>, Object> attrs = attrs();
synchronized (attrs) {
    for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
        @SuppressWarnings("unchecked")
        AttributeKey<Object> key = (AttributeKey<Object>) e.getKey();
        channel.attr(key).set(e.getValue());
    }
}
```

2. 将 AbstractBootstrap 的 Handler 添加到 NioServerSocketChannel 的 PipeLine 中，代码如下：

```
ChannelPipeline p = channel.pipeline();
if (handler() != null) {
    p.addLast(handler());
}
```

3. 将用于服务端注册的 Handler --ServerBootstrapAcceptor 添加到 PipeLine 中，代码如下：

```
p.addLast(new ChannelInitializer<Channel>() {
    @Override
    public void initChannel(Channel ch) throws Exception {
        ch.pipeline().addLast(new ServerBootstrapAcceptor(currentChildHandler, currentChildOptions,
                currentChildAttrs));
    }
});
```

好，到此处，启动服务端监听的相关资源已经初始化完毕，就剩下最后一步注册 NioServerSocketChannel 到 Reactor 线程的多路复用器上，然后轮询连接事件。在分析这个过程之前，我们先通过下图看看目前 NioServerSocketChannel 的 PipeLine 的组成。

图 3.1.1-2 NioServerSocketChannel 的 PipeLine

3.1.2. NioServerSocketChannel 的注册

当 NioServerSocketChannel 初始化完成之后，需要将它注册到 Reactor 线程的多路复用器上监听新客户端的接入，代码如下：

```
@Override
public final void register(final ChannelPromise promise) {
    if (eventLoop.inEventLoop()) {分支一
        register0(promise);
    } else {
        try {  分支二
            eventLoop.execute(new Runnable() {
                @Override
                public void run() {
                    register0(promise);
                }
            });
        } catch (Throwable t) {
            logger.warn(
                    "Force-closing a channel whose registration task was not accepted by an event loop: {}",
                    AbstractChannel.this, t);
            closeForcibly();
            closeFuture.setClosed();
            promise.setFailure(t);
        }
```

首先判断是否是 NioEventLoop 发起的操作，如果是，则不会存在并发操作，直接执行 Channel 注册；如果由其它线程发起，则封装成一个 Task 放入消息队列中异步执行。此处，由于是由 ServerBootstrap 所在线程执行的注册操作，所以执行分支二，代码如下：

```
private void register0(ChannelPromise promise) {
    try {
        // check if the channel is still open as it could be closed in the mean time when the register
        // call was outside of the eventLoop
        if (!ensureOpen(promise)) {
            return;
        }
        doRegister();
        registered = true;
        promise.setSuccess();
        pipeline.fireChannelRegistered();
        if (isActive()) {
            pipeline.fireChannelActive();
        }
    } catch (Throwable t) {
        // Close the channel directly to avoid FD leak.
        closeForcibly();
        closeFuture.setClosed();
        if (!promise.tryFailure(t)) {
            logger.warn(
                    "Tried to fail the registration promise, but it is complete already. " +
                            "Swallowing the cause of the registration failure:", t);
        }
```

将 NioServerSocketChannel 注册到 NioEventLoop 的 selector 上，代码如下：

```
protected void doRegister() throws Exception {
    boolean selected = false;
    for (;;) {
        try {
            selectionKey = javaChannel().register(eventLoop().selector, 0, this);
            return;
        } catch (CancelledKeyException e) {
```

大家可能会很诧异，应该注册 OP_ACCEPT（16）到多路复用器上，怎么注册个 0 呢？0 表示只注册，不监听任何网络操作。这样做的原因如下：

1. 注册方法是多态的，它既可以被 NioServerSocketChannel 用来监听客户端的

连接接入，也可以用来注册 SocketChannel，用来监听网络读或者写操作；

2. 通过 SelectionKey 的 interestOps(int ops)方法可以方便的修改监听对象，所以，此处注册需要获取 SelectionKey 并给 AbstractNioChannel 的成员变量 selectionKey 赋值。

注册成功之后，触发 ChannelRegistered 事件，方法如下：

```
promise.setSuccess();
pipeline.fireChannelRegistered();
```

Netty 的 HeadHandler 不需要处理 ChannelRegistered 事件，所以，直接直接调用下一个 Handler,代码如下：

```
@Skip
@Override
public void channelRegistered(ChannelHandlerContext ctx) throws Exception {
    ctx.fireChannelRegistered();
}
```

当 ChannelRegistered 事件传递到 TailHandler 后结束，TailHandler 也不关心 ChannelRegistered 事件，因此是空实现，代码如下：

```
static final class TailHandler extends ChannelHandlerAdapter {

    @Override
    public void channelRegistered(ChannelHandlerContext ctx) throws Exception { }
```

ChannelRegistered 事件传递完成后，判断 ServerSocketChannel 监听是否成功，如果成功，需要出发 NioServerSocketChannel 的 ChannelActive 事件，代码如下：

```
if (isActive()) {
    pipeline.fireChannelActive();
}
```

isActive()也是个多态方法，如果是服务端，判断监听是否启动，如果是客户端，判断 TCP 连接是否完成。ChannelActive 事件在 PipeLine 中传递，完成之后根据配置决定是否自动触发 Channel 的读操作，代码如下：

```
@Override
public ChannelPipeline fireChannelActive() {
    head.fireChannelActive();

    if (channel.config().isAutoRead()) {
        channel.read();
    }

    return this;
}
```

AbstractChannel 的读操作出发 PipeLine 的读操作，最终调用到 HeadHandler 的读操作，代码如下：

```
@Override
public void read(ChannelHandlerContext ctx) {
    unsafe.beginRead();
}
```

继续看 AbstractUnsafe 的 beginRead 方法，代码如下：

```
@Override
public void beginRead() {
    if (!isActive()) {
        return;
    }

    try {
        doBeginRead();
    } catch (final Exception e) {
        invokeLater(new Runnable() {
            @Override
            public void run() {
                pipeline.fireExceptionCaught(e);
            }
        });
        close(voidPromise());
    }
}
```

由于不同类型的 Channel 对读操作的准备工作不同，因此，beginRead 也是个多态方法，对于 NIO 通信，无论是客户端还是服务端，都是要修改网络监听操作位为自身感兴趣的，对于 NioServerSocketChannel 感兴趣的操作是 OP_ACCEPT（16），于是重新修改注册的操作位为 OP_ACCEPT，代码如下：

```
@Override
protected void doBeginRead() throws Exception {
    if (inputShutdown) {
        return;
    }

    final SelectionKey selectionKey = this.selectionKey;
    if (!selectionKey.isValid()) {
        return;
    }

    final int interestOps = selectionKey.interestOps();
    if ((interestOps & readInterestOp) == 0) {
        selectionKey.interestOps(interestOps | readInterestOp);
    }
```

在某些场景下，当前监听的操作类型和 Chanel 关心的网络事件是一致的，不需要重复注册，所以增加了&操作的判断，只有两者不一致，才需要重新注册操作位。

JDK SelectionKey 有四种操作类型，分别为：

1. OP_READ = 1 << 0；
2. OP_WRITE = 1 << 2；
3. OP_CONNECT = 1 << 3；
4. OP_ACCEPT = 1 << 4。

由于只有四种网络操作类型，所以用 4 bit 就可以表示所有的网络操作位，由于

JAVA 语言没有 bit 类型，所以使用了整形来表示，每个操作位代表一种网络操作类型，分别为：0001、0010、0100、1000, 这样做的好处是可以非常方便的通过位操作来进行网络操作位的状态判断和状态修改，提升操作性能。

好，接着刚才的继续讲，由于创建 NioServerSocketChannel 将 readInterestOp 设置成了 OP_ACCEPT，所以，我们重新设置关注客户端的网络连接操作，代码如下：

```
/**
 * Create a new instance
 */
public NioServerSocketChannel(EventLoop eventLoop, EventLoopGroup childGroup) {
    super(null, eventLoop, childGroup, newSocket(), SelectionKey.OP_ACCEPT);
    config = new DefaultServerSocketChannelConfig(this, javaChannel().socket());
}
```

到此，服务端监听启动部分源码已经分析完成，接下来，让我们继续分析一个新的客户端是如何接入的。

3.1.3. 新的客户端接入

负责处理网络读写、连接和客户端请求接入的 Reactor 线程就是 NioEventLoop，下面我们分析下 NioEventLoop 是如何处理新的客户端连接接入的。当多路复用器检测到新的准备就绪的 Channel 时，默认执行

processSelectedKeysOptimized 方法，代码如下：

```
if (selectedKeys != null) {
    processSelectedKeysOptimized(selectedKeys.flip());
} else {
    processSelectedKeysPlain(selector.selectedKeys());
}
```

由于 Channel 的 Attachment 是 NioServerSocketChannel,所以执行

processSelectedKey 方法，根据就绪的操作位，执行不同的操作，此处，由于

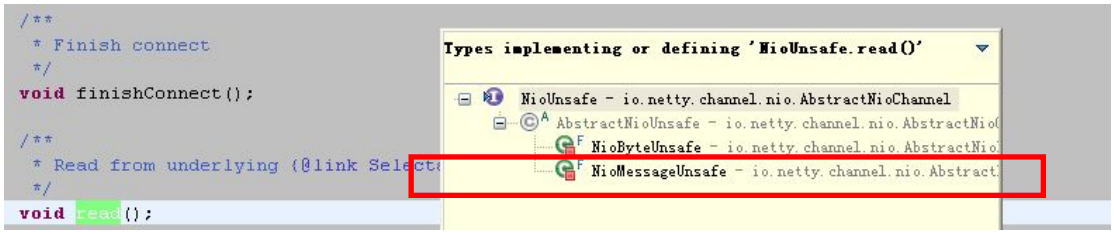
```
final Object a = k.attachment();

if (a instanceof AbstractNioChannel) {
    processSelectedKey(k, (AbstractNioChannel) a);
} else {
    @SuppressWarnings("unchecked")
    NioTask<SelectableChannel> task = (NioTask<SelectableChannel>) a;
    processSelectedKey(k, task);
}
```

监听的是连接操作，所以执行 unsafe.read()方法，由于不同的 Channel 执行不同

```
try {
    int readyOps = k.readyOps();
    // Also check for readOps of 0 to workaround possible JDK bug which may otherwise lead
    // to a spin loop
    if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) != 0 || readyOps == 0) {
        unsafe.read();
        if (!ch.isOpen()) {
            // Connection already closed - no need to handle write.
            return;
        }
    }
```

的操作，所以 NioUnsafe 被设计成接口，由不同的 Channel 内部的 NioUnsafe 实现类负责具体实现，我们发现 read()方法的实现有两个，分别是 NioByteUnsafe

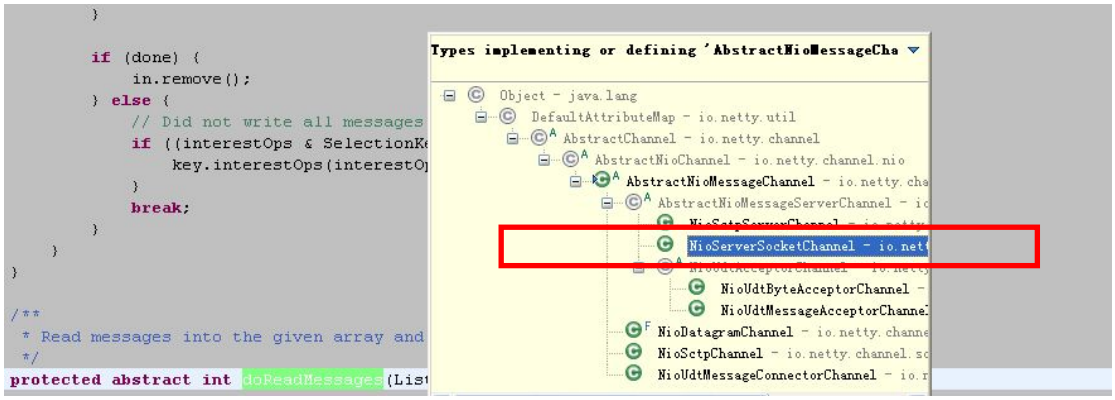
和 NioMessageUnsafe,后面有专门章节介绍 Netty 的 Unsafe 接口，此处不展开介绍，对于 NioServerSocketChannel，它使用的是 NioMessageUnsafe，下面我们继续分析



它的 read()方法，代码如下：

```
try {
    for (;;) {
        int localRead = doReadMessages(readBuf);
        if (localRead == 0) {
            break;
        }
```

展开 doReadMessages，继续分析，对于 NioServerSocketChannel 就是接收新的客



户端连接并设置 TCP 参数，代码如下：首先通过

ServerSocketChannel.accept()创建 SocketChannel，通过装饰模式将 SocketChannel

```
@Override
protected int doReadMessages(List<Object> buf) throws Exception {
    SocketChannel ch = javaChannel().accept();

    try {
        if (ch != null) {
            buf.add(new NioSocketChannel(this, childEventLoopGroup().next(), ch));
            return 1;
        }
    } catch (Throwable t) {
        logger.warn("Failed to create a new channel from an accepted socket.", t);

        try {
            ch.close();
        } catch (Throwable t2) {
            logger.warn("Failed to close a socket.", t2);
        }
    }

    return 0;
```

包装成 Netty 的 NioSocketChannel，然后加入到消息列表 buf 中返回并退出循环。

```
int size = readBuf.size();
for (int i = 0; i < size; i ++) {
    pipeline.fireChannelRead(readBuf.get(i));
}
```

接收到新的客户端连接后，触发 PipeLine 的 ChannelRead 方法，代码如上所示。

首先执行 headChannelHandlerContext 的 fireChannelRead 方法，事件在 PipeLine 中传递，执行 ServerBootstrapAcceptor 的 channelRead 方法，代码如下：

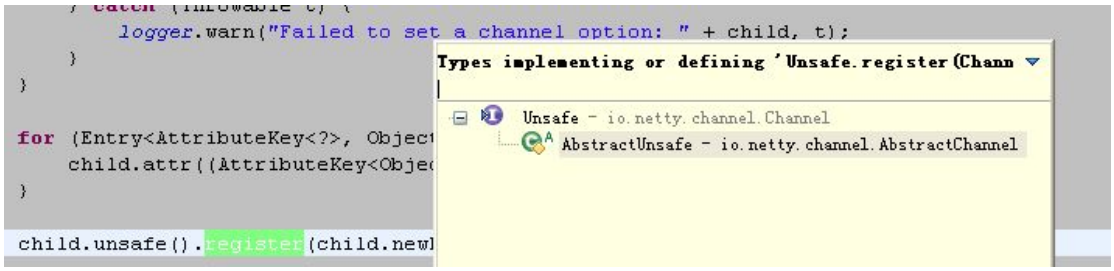
```
@Override
@SuppressWarnings("unchecked")
public void channelRead(ChannelHandlerContext ctx, Object msg) {
    Channel child = (Channel) msg;

    child.pipeline().addLast(childHandler);

    for (Entry<ChannelOption<?>, Object> e: childOptions) {
        try {
            if (!child.config().setOption((ChannelOption<Object>) e.getKey(), e.getValue())) {
                logger.warn("Unknown channel option: " + e);
            }
        } catch (Throwable t) {
            logger.warn("Failed to set a channel option: " + child, t);
        }
    }

    for (Entry<AttributeKey<?>, Object> e: childAttrs) {
        child.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
    }

    child.unsafe().register(child.newPromise());
}
```

第一步：将启动时传入的 childHandler 加入到客户端 SocketChannel 的 PipeLine 中

第二步：设置客户端 SocketChannel 的 TCP 参数

第三步：注册 SocketChannel 到多路复用器

channelRead 主要执行如上图所示的三个方法，下面我们展开看下 NioSocketChannel 的 register 方法，代码如下所示：



我们发现 NioSocketChannel 的注册方法与 ServerSocketChannel 的一致，也是将 Channel 注册到 Reactor 线程的多路复用器上，由于注册的操作位是 0，所以，此时 NioSocketChannel 还不能读取客户端发送的消息，那什么时候修改监听操作位为 OP_READ 呢，别着急，继续看代码。

执行完注册操作之后，紧接着出发 ChannelReadComplete 事件，我们继续分析 ChannelReadComplete 在 PipeLine 中的处理流程：Netty 的 Header 和 Tail 本身

```
@Override
public ChannelPipeline fireChannelReadComplete() {
    head.fireChannelReadComplete();
    if (channel.config().isAutoRead()) {
        read();
    }
    return this;
}
```

不关注 ChannelReadComplete 事件就直接透传，执行完 ChannelReadComplete 后，最终执行 PipeLine 的 read（）方法，最终执行 HeadHandler 的 read（）方法，代码如下：

```
@Override
public void read(ChannelHandlerContext ctx) {
    unsafe.beginRead();
}
```

后面的代码已经在之前的 3.1.2 章节介绍过，用来修改网络操作位为读操作，创建 NioSocketChannel 的时候已经将 AbstractNioChannel 的 readInterestOp 设置为

```
protected AbstractNioByteChannel(Channel parent, EventLoop eventLoop, SelectableChannel ch) {
    super(parent, eventLoop, ch, SelectionKey.OP_READ);
}
```

OP_READ，这样，执行 selectionKey.interestOps(interestOps | readInterestOp)操作时就会把操作位设置为 OP_READ。

这样，新接入的客户端连接就正式建立，并被挂载到 Reactor 的 Selector 上用来监听读操作。

3.2. 客户端创建

3.2.1. 客户端连接辅助类 Bootstrap

如果直接使用 JAVA 的类库创建基于 NIO 的客户端连接比较繁琐，需要对 NIO 有深刻的理解和实践基础才能写出高质量的客户端程序。如同服务端，Netty 也试图屏蔽客户端连接创建的细节，通过辅助类 Bootstrap 来屏蔽细节降低开发难度。

首先，创建 Bootstrap 的实例，类似 ServerBootstrap,客户端也使用 Builder 模式来构造。对于客户端，由于它不需要监听和处理来自客户端的连接，所以，只需要一个 Reactor 线程组即可，代码如下：

```
// Configure the client.
EventLoopGroup group = new NioEventLoopGroup();
try {
    Bootstrap b = new Bootstrap();
    b.group(group)
```

完成连接辅助类和 Reactor 线程组的初始化操作后，继续设置发起连接的 Channel 为 NioSocketChannel，代码如下：如同服务端启动辅助类，客户端辅助类采用

```
b.group(group)
 .channel(NioSocketChannel.class)
```

工厂模式创建 NioSocketChannel，BootstrapChannelFactory 是 Bootstrap 的内部静态工厂类，用于根据 Channel 的类型和构造函数反射创建新的 NioSocketChannel，代码如下所示：

```
if (channelClass == null) {
    throw new NullPointerException("channelClass");
}
return channelFactory(new BootstrapChannelFactory<Channel>(channelClass));
```

```
private static final class BootstrapChannelFactory<T extends Channel> implements ChannelFactory<T> {

    private final Class<? extends T> clazz;

    BootstrapChannelFactory(Class<? extends T> clazz) {
        this.clazz = clazz;
    }

    @Override
    public T newChannel(EventLoop eventLoop) {
        try {
            Constructor<? extends T> constructor = clazz.getConstructor(EventLoop.class);
            return constructor.newInstance(eventLoop);
        } catch (Throwable t) {
            throw new ChannelException("Unable to create Channel from class " + clazz, t);
        }
    }
```

Channel 工厂初始化完成后，设置 TCP 参数，然后设置 Handler，由于此时 NioSocketChannel 还没有真正创建，所以，PipeLine 也没有创建，Netty 预置一个负责创建业务 Handler 的初始化 Hanlder 工厂到启动辅助类中，当 initChannel 方法被执行时再创建业务 Handler，代码如下：

```
.handler(new ChannelInitializer<SocketChannel>() {
    @Override
    public void initChannel(SocketChannel ch) throws Exception {
        ch.pipeline().addLast(
                //new LoggingHandler(LogLevel.INFO),
                new EchoClientHandler(firstMessageSize));
    }
});
```

一切准备就绪后，发起连接操作，代码如下：

```
private ChannelFuture doConnect(final SocketAddress remoteAddress, final SocketAddress localAddress) {
    final ChannelFuture regFuture = initAndRegister(); 第一步：初始化 SocketChannel 并注册
    final Channel channel = regFuture.channel();
    if (regFuture.cause() != null) {
        return regFuture;
    }

    final ChannelPromise promise = channel.newPromise();
    if (regFuture.isDone()) {  第二步：判断注册操作是否完成，完成直接发起连接，
        doConnect0(regFuture, channel, remoteAddress, localAddress, promise);
    } else {          第三步：如果注册操作没有完成，设置监听器，完成后被回调
        regFuture.addListener(new ChannelFutureListener() {
            @Override 执行连接操作
            public void operationComplete(ChannelFuture future) throws Exception {
                doConnect0(regFuture, channel, remoteAddress, localAddress, promise);
            }
        });
```

第一步，初始化 NioSocketChannel，设置 TCP 参数，注册 SocketChannel 到 Reactor 线程的多路复用器中，代码如下：

```
@Override
Channel createChannel() {
    EventLoop eventLoop = group().next();
    return channelFactory().newChannel(eventLoop);
}
```

从 Reactor 线程池中顺序获取 NioEventLoop 线程

创建 NioSocketChannel

初始化 NioSocketChannel，将预置的 Hanlder 加入到 NioSocketChannel 的 PipeLine 中，设置客户端连接的 TCP 参数，代码如下：

```
ChannelPipeline p = channel.pipeline();
p.addLast(handler());

final Map<ChannelOption<?>, Object> options = options();
synchronized (options) {
    for (Entry<ChannelOption<?>, Object> e: options.entrySet()) {
        try {
            if (!channel.config().setOption((ChannelOption<Object>) e.getKey(), e.getValue())) {
                logger.warn("Unknown channel option: " + e);
            }
        } catch (Throwable t) {
            logger.warn("Failed to set a channel option: " + channel, t);
        }
    }
}

final Map<AttributeKey<?>, Object> attrs = attrs();
synchronized (attrs) {
    for (Entry<AttributeKey<?>, Object> e: attrs.entrySet()) {
        channel.attr((AttributeKey<Object>) e.getKey()).set(e.getValue());
    }
}
```

发起注册操作，注册操作在 3.1 章节创建服务端的时候已经详细讲解过，这里不再重复讲解。

第二步：判断 NioSocketChannel 是否注册成功，由于是异步注册，通常返回是 False，执行第三步操作，当 NioSocketChannel 注册成功后，发起异步连接操作：

```
channel.eventLoop().execute(new Runnable() {
    @Override
    public void run() {
        if (regFuture.isSuccess()) {
            if (localAddress == null) {
                channel.connect(remoteAddress, promise);
            } else {
                channel.connect(remoteAddress, localAddress, promise);
            }
            promise.addListener(ChannelFutureListener.CLOSE_ON_FAILURE);
        } else {
            promise.setFailure(regFuture.cause());
        }
    }
});
```

根据客户端是否指定本地绑定地址执行不同的分支，下面具体分析 AbstractChannel 发起的连接操作，代码如下：

```
@Override
public ChannelFuture connect(SocketAddress remoteAddress) {
    return pipeline.connect(remoteAddress);
}
```

首先调用 NioSocketChannel 的 PipeLine，执行连接操作：最终会调用到 HeadHandler 的 connect 方法，代码如下：

```
@Override
public void connect(
        ChannelHandlerContext ctx,
        SocketAddress remoteAddress, SocketAddress localAddress,
        ChannelPromise promise) throws Exception {
    unsafe.connect(remoteAddress, localAddress, promise);
}
```

展开 AbstractNioUnsafe 的 connect 进行分析，代码如下：

```
boolean wasActive = isActive();
if (doConnect(remoteAddress, localAddress)) {
```

首先获取当前的连接状态进行缓存，然后发起连接操作，代码如下：

```
@Override
protected boolean doConnect(SocketAddress remoteAddress, SocketAddress localAddress) throws Exception {
    if (localAddress != null) {
        javaChannel().socket().bind(localAddress);
    }

    boolean success = false;
    try {
        boolean connected = javaChannel().connect(remoteAddress);
        if (!connected) {
            selectionKey().interestOps(SelectionKey.OP_CONNECT);
        }
        success = true;
        return connected;
    } finally {
        if (!success) {
            doClose();
        }
    }
}
```

如果指定了本地绑定端口，执行绑定操作

发起异步 TCP 连接，可能连接成功，也可能暂时没有连接成功

如果没有立即连接成功，则监听连接操作

大家需要注意的是，SocketChannel 执行 connect()操作后有三种结果：

1. 连接成功，返回 True；
2. 暂时没有连接上，服务端没有返回 ACK 应答，连接结果不确定，返回 False；
3. 连接失败，直接抛出 IO 异常。

如果是第二种结果，需要将 NioSocketChannel 中的 selectionKey 设置为 OP_CONNECT，监听连接结果。

异步连接返回之后，需要判断连接结果，如果连接成功，则触发 ChannelActive 事件，代码如下：

```
private void fulfillConnectPromise(ChannelPromise promise, boolean wasActive) {
    // trySuccess() will return false if a user cancelled the connection attempt.
    boolean promiseSet = promise.trySuccess();

    // Regardless if the connection attempt was cancelled, channelActive() event should be triggered,
    // because what happened is what happened.
    if (!wasActive && isActive()) {
        pipeline().fireChannelActive();
    }
```

ChannelActive 事件处理在 3.1 章节已经详细说明，最终会将 NioSocketChannel 中的 selectionKey 设置为 SelectionKey.OP_READ，用于监听网络读操作。

如果没有立即连接上服务端，则执行如下分支，

```
int connectTimeoutMillis = config().getConnectTimeoutMillis();
if (connectTimeoutMillis > 0) {
    connectTimeoutFuture = eventLoop().schedule(new Runnable() {
        @Override
        public void run() {
            ChannelPromise connectPromise = AbstractNioChannel.this.connectPromise;
            ConnectTimeoutException cause =
                    new ConnectTimeoutException("connection timed out: " + remoteAddress);
            if (connectPromise != null && connectPromise.tryFailure(cause)) {
                close(voidPromise());
            }
        }
    }, connectTimeoutMillis, TimeUnit.MILLISECONDS);
}

promise.addListener(new ChannelFutureListener() {
    @Override
    public void operationComplete(ChannelFuture future) throws Exception {
        if (future.isCancelled()) {
            if (connectTimeoutFuture != null) {
                connectTimeoutFuture.cancel(false);
            }
            connectPromise = null;
            close(voidPromise());
```

上面的操作有两个目的：

1. 根据连接超时事件设置定时任务，超时时间到之后触发校验，如果发现连接并没有完成，则关闭连接句柄，释放资源，设置异常堆栈并发起去注册；
2. 设置连接结果监听器，如果接收到连接完成通知则判断连接是否被取消，如果被取消则关闭连接句柄，释放资源，发起取消注册操作。

3.2.2. 服务端返回 ACK 应答，客户端连接成功

让我们重新回到 Reactor 线程 NioEventLoop 中，看看如何处理客户端连接，当服务端返回 ACK 应答后，触发 Selector 轮询出就绪的 SocketChannel，代码如下：

```
if ((readyOps & SelectionKey.OP_CONNECT) != 0) {
    // remove OP_CONNECT as otherwise Selector.select(..) will always return without blocking
    // See https://github.com/netty/netty/issues/924
    int ops = k.interestOps();
    ops &= ~SelectionKey.OP_CONNECT;
    k.interestOps(ops);

    unsafe.finishConnect();
}
```

首先将 OP_CONNECT 从 selector 上摘除掉，然后调用 AbstractNioChannel 的 finishConnect 方法，判断异步连接的结果，代码如下：

```
@Override
public void finishConnect() {
    // Note this method is invoked by the event loop only if the connection attempt was
    // neither cancelled nor timed out.

    assert eventLoop().inEventLoop();
    assert connectPromise != null;

    try {
        boolean wasActive = isActive();
        doFinishConnect();
```

首先缓存连接状态，当前返回 False，然后执行 doFinishConnect 方法判断连接结果，代码如下：

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

通过 SocketChannel 的 finishConnect 方法判断连接结果，执行该方法返回三种结果：

1. 连接成功返回 True；
2. 连接失败返回 False；
3. 发生链路被关闭、链路中断等异常，连接失败。

只要连接失败，就抛出 Error()，由调用方执行句柄关闭等资源释放操作，如果返回成功，则执行 fulfillConnectPromise 方法，该方法之前已经介绍过，它负责将 SocketChannel 修改为读操作，用来监听网络的读事件，代码如下：

```
private void fulfillConnectPromise(ChannelPromise promise, boolean wasActive) {
    // trySuccess() will return false if a user cancelled the connection attempt.
    boolean promiseSet = promise.trySuccess();

    // Regardless if the connection attempt was cancelled, channelActive() event should be triggered,
    // because what happened is what happened.
    if (!wasActive && isActive()) {
        pipeline().fireChannelActive();
    }
```

如果连接超时时仍然没有接收到服务端的 ACK 应答消息，则由定时任务关闭客户端连接，将 SocketChannel 从 Reactor 线程的多路复用器上摘除，释放资源。

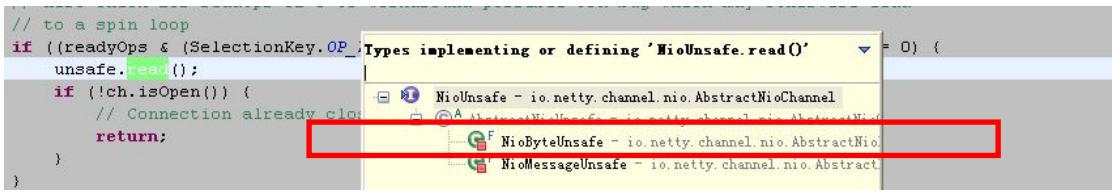
至此，客户端创建连接源码分析完毕。下面章节将介绍消息的读写机制。

3.3. 读操作

3.3.1. 异步读取消息

NioEvnetLoop 作为 Reactor 线程，负责轮询多路复用器，获取就绪的通道执行网络的连接、客户端请求接入、读和写。

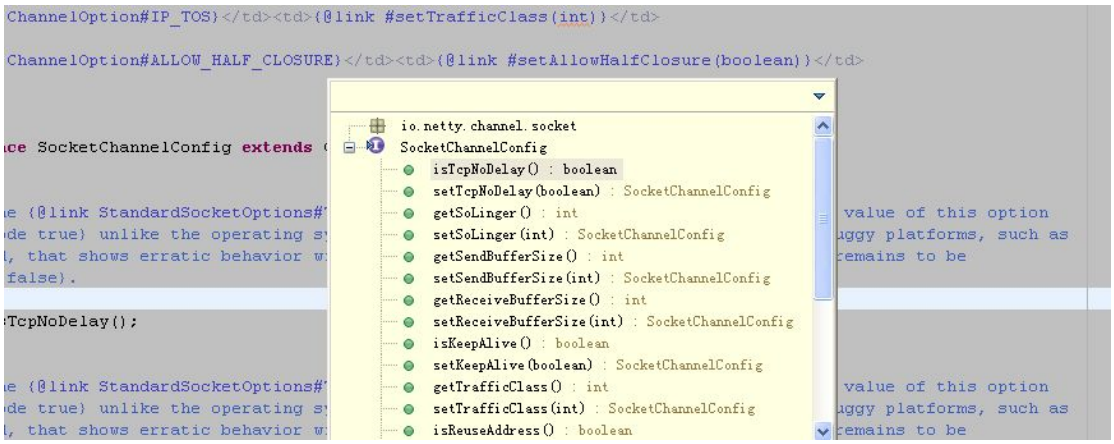
当多路复用器检测到读操作后，执行如下方法：不同的 Channel 对应不同的



NioUnsafe, 此处对应的是 NioByteUnsafe, 下面我们进入它的父类 AbstractNioByteChannel 类进行详细分析：

```
final ChannelConfig config = config();
final ChannelPipeline pipeline = pipeline();
final ByteBufAllocator allocator = config.getAllocator();
final int maxMessagesPerRead = config.getMaxMessagesPerRead();
RecvByteBufAllocator.Handle allocHandle = this.allocHandle;
if (allocHandle == null) {
    this.allocHandle = allocHandle = config.getRecvByteBufAllocator().newHandle();
}
if (!config.isAutoRead()) {
    removeReadOp();
}
```

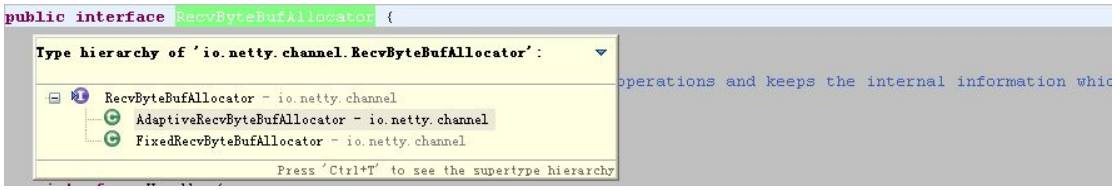
首先，获取 NioSocketChannel 的 SocketChannelConfig，它主要用于设置客户端连接的 TCP 参数，接口如下：



我们重点看红框中标出的代码：如果首次调用，从 SocketChannelConfig 的 RecvByteBufAllocator 中创建 Handle。下面我们对 RecvByteBufAllocator 做下简单的代码分析：RecvByteBufAllocator 默认有两种实现，分别是：AdaptiveRecvByteBufAllocator 和 FixedRecvByteBufAllocator。由于

FixedRecvByteBufAllocator 的实现比较简单，我们重点分析

AdaptiveRecvByteBufAllocator 的实现。



顾名思义，AdaptiveRecvByteBufAllocator 指的是缓冲区大小可以动态调整的

ByteBuf 分配器。下面看下它的成员变量：

```
static final int DEFAULT_MINIMUM = 64;
static final int DEFAULT_INITIAL = 1024;
static final int DEFAULT_MAXIMUM = 65536;

private static final int INDEX_INCREMENT = 4;
private static final int INDEX_DECREMENT = 1;
```

它分别定义了三个系统默认值：最小缓冲区长度 64 字节、初始容量 1024 字节、

最大容量 65536 字节。

还定义了两个动态调整容量时的步进参数：扩张的步进索引为 4、收缩的步进索引为 1。

最后，定义了长度的向量表 SIZE_TABLE 并初始化它，它的初始值如下：

```
0-->16 1-->32 2-->48 3-->64 4-->80 5-->96 6-->112 7-->128 8-->144
9-->160
10-->176 11-->192 12-->208 13-->224 14-->240 15-->256 16-->272
17-->288 18-->304
19-->320 20-->336 21-->352 22-->368 23-->384 24-->400 25-->416
26-->432 27-->448
28-->464 29-->480 30-->496 31-->512 32-->1024 33-->2048 34-->4096
35-->8192 36-->16384
37-->32768 38-->65536 39-->131072 40-->262144 41-->524288
42-->1048576 43-->2097152 44-->4194304 45-->8388608

46-->16777216 47-->33554432 48-->67108864 49-->134217728
50-->268435456 51-->536870912 52-->1073741824
```

向量数组的每个值都对应一个 Buffer 容量，当容量小于 512 的时候，由于缓冲区已经比较小，需要降低步进值，容量每次下调的幅度要小些；当大于 512 时，说明需要解码的消息码流比较大，这时采用调大步进幅度的方式减少动态扩张的频率，所以它采用 512 的倍数进行扩张。

接下来我们重点分析下 AdaptiveRecvByteBufAllocator 的方法：

方法一：getSizeTableIndex(final int size)，代码如下：

```
private static int getSizeTableIndex(final int size) {
    for (int low = 0, high = SIZE_TABLE.length - 1;;) {
        if (high < low) {
            return low;
        }
        if (high == low) {
            return high;
        }

        int mid = low + high >>> 1;
        int a = SIZE_TABLE[mid];
        int b = SIZE_TABLE[mid + 1];
        if (size > b) {
            low = mid + 1;
        } else if (size < a) {
            high = mid - 1;
        } else if (size == a) {
            return mid;
        } else {
            return mid + 1;
        }
    }
```

根据容量 Size 查找容量向量表对应的索引：这是个典型的二分查找法，由于它的算法非常经典，也比较简单，此处不再赘述。

下面我们分析下它的内部静态类 HandleImpl，首先，还是看下它的成员变量：

```
private final int minIndex;
private final int maxIndex;
private int index;
private int nextReceiveBufferSize;
private boolean decreaseNow;
```

它有五个成员变量，分别是：对应向量表的最小索引、最大索引、当前索引、下一次预分配的 Buffer 大小和是否立即执行容量收缩操作。

我们重点分析它的 record(int actualReadBytes)方法：当 NioSocketChannel 执行完读操作后，会计算获得本次轮询读取的总字节数，它就是参数 actualReadBytes，执行 record 方法，根据实际读取的字节数对 ByteBuf 进行动态伸缩和扩张，代码如下：

```
@Override
public void record(int actualReadBytes) {
    if (actualReadBytes <= SIZE_TABLE[Math.max(0, index
            - INDEX_DECREMENT - 1)]) {
        if (decreaseNow) {
            index = Math.max(index - INDEX_DECREMENT, minIndex);
            nextReceiveBufferSize = SIZE_TABLE[index];
            decreaseNow = false;
        } else {
            decreaseNow = true;
        }
    } else if (actualReadBytes >= nextReceiveBufferSize) {
        index = Math.min(index + INDEX_INCREMENT, maxIndex);
        nextReceiveBufferSize = SIZE_TABLE[index];
        decreaseNow = false;
    }
}
```

首先，对当前索引做步进缩减，然后获取收缩后索引对应的容量，与实际读取的字节数进行比对，如果发现小于收缩后的容量，则重新对当前索引进行赋值，取收缩后的索引和最小索引中的较大者作为最新的索引，然后，为下一次缓冲区容量分配赋值--新的索引对用容量向量表中的容量。相反，如果当前实际读取的字节数大于之前预分配的初始容量，则说明实际分配的容量不足，需要动态扩张。重新计算索引，选取当前索引+扩张步进 和 最大索引中的较小作为当前索引值，然后对下次缓冲区的容量值进行重新分配，完成缓冲区容量的动态扩张。

通过上述分析我们得知，AdaptiveRecvByteBufAllocator 就是根据本次读取的实际字节数对下次接收缓冲区的容量进行动态分配。

使用动态缓冲区分配器的优点如下：

1. Netty 作为一个通用的 NIO 框架，并不对客户的应用场景进行假设，你可能使用它做流媒体传输，也可能用它做聊天工具，不同的应用场景，传输的码流大小千差万别；无论初始化分配的是 32K 还是 1M，都会随着应用场景的变化而变得不适应，因此，Netty 根据上次实际读取的码流大小对下次的接收 Buffer 缓冲区进行预测和调整，能够最大限度的满足不同行业的应用场景；

2. 性能更高，容量过大会导致内存占用开销增加,后续的 Buffer 处理性能会下降；容量过小时需要频繁的内存扩张来接收大的请求消息，同样会导致性能下降；

3. 更节约内存：设想，假如通常情况下请求消息平均值为 1M 左右，接收缓冲区大小为 1.2M；突然某个客户发送了一个 10M 的流媒体附件，接收缓冲区扩张为 10M 以接纳该附件，如果缓冲区不能收缩，每次缓冲区创建都会分配 10M 的内存，但是后续所有的消息都是 1M 左右，这样会导致内存的浪费，如果并发客户端过多、Reactor 线程个数过多，可能会发生内存溢出，最终宕机。

好，看完了 AdaptiveRecvByteBufAllocator，我们继续分析读操作：

```
try {
    int byteBufCapacity = allocHandle.guess();
    int totalReadAmount = 0;
    do {
        byteBuf = allocator.ioBuffer(byteBufCapacity);
```

首先通过接收缓冲区分配器的 Handler 计算获得下次预分配的缓冲区容量 byteBufCapacity，紧接着根据缓冲区容量进行缓冲区分配，Netty 的缓冲区种类很多，此处重点介绍的是消息的读取，因此对缓冲区不展开说明。

接收缓冲区 ByteBuf 分配完成后，进行消息的异步读取，代码如下：

```
int localReadAmount = doReadBytes(byteBuf);
```

它是个抽象方法，具体实现在 NioSocketChannel 中，代码如下：

```
@Override
protected int doReadBytes(ByteBuf byteBuf) throws Exception {
    return byteBuf.writeBytes(javaChannel(), byteBuf.writableBytes());
}
```

其中 javaChannel()返回的是 SocketChannel,代码如下：

```
@Override
protected SocketChannel javaChannel() {
    return (SocketChannel) super.javaChannel();
}
```

其中 byteBuf.writableBytes()返回本次可读的最大长度，我们继续展开看最终是如何从 Channel 中读取码流的，代码如下：

```
@Override
public int writeBytes(ScatteringByteChannel in, int length) throws IOException {
    ensureWritable(length);
    int writtenBytes = setBytes(writerIndex, in, length);
    if (writtenBytes > 0) {
        writerIndex += writtenBytes;
    }
    return writtenBytes;
}
```

对 setBytes 方法展开代码如下：

```
@Override
public int setBytes(int index, ScatteringByteChannel in, int length) throws IOException {
    ensureAccessible();
    try {
        return in.read((ByteBuffer) internalNioBuffer().clear().position(index).limit(index + length));
    } catch (ClosedChannelException e) {
        return -1;
    }
}
```

由于 SocketChannel 的 read 方法参数是 JAVA NIO 的 ByteBuffer,所以，需要先将 Netty 的 ByteBuf 转换成 JDK 的 ByteBuffer，随后调用 ByteBuffer 的 clear 方法对指针进行重置用于新消息的读取，随后将 position 指针指到初始读的 index，读取的上限设置为 index + 读取的长度。最后调用 read 方法将 SocketChannel 中就绪的码流读取到 ByteBuffer 中，完成消息的读取，返回读取的字节数。

完成消息的异步读取后，需要对本次读取的字节数进行判断，有三种可能：

1. 返回 0，表示没有就绪的消息可读；
2. 返回值大于 0，读到了消息；
3. 返回值-1，表示发生了 IO 异常，读取失败。

下面我们继续看 Netty 的处理逻辑，首先对读取的字节数进行判断，如果等于或

```
    if (localReadAmount <= 0) {
        // not was read release the buffer
        byteBuf.release();
        close = localReadAmount < 0;
        break;
    }
```

者小于 0，表示没有就绪的消息可读或者发生了 IO 异常，此时需要释放接收缓冲区，如果读取的字节数小于 0，则需要将 close 状态位置位，用于关闭连接，释放句柄资源。置位完成之后，退出循环。

完成一次异步读之后，就会触发一次 ChannelRead 事件，这里特别需要提醒大家的是，完成一次读操作，并不意味着读到了一条完整的消息，因为 TCP 底层存在组包和粘包，所以，一次读操作可能包含多条消息，也可能是一条不完整的消息，所以，不要把它跟读取的消息个数等同起来。我曾经发现有同事在没有做任何半包处理的情况下，以 ChannelRead 的触发次数做计数器来进行性能分析和统计，是完全错误的。当然，如果你使用了针对半包的 Decode 类或者自己做了特殊封装，对 ChannelRead 事件进行拦截，屏蔽 Netty 的默认机制，也能够实现一次 ChannelRead 对应一条完整消息的效果，此处也不再展开说明了，当你掌握了 Netty 的编解码技巧之后，自然就知道如何实现这种效果了。

触发和完成 ChannelRead 事件调用之后，将接收缓冲区释放，代码如下：

```
    byteBuf = null;
```

好，我们继续分析，因为一次读操作未必能够完成 TCP 缓冲区的全部读取工作，所以，读操作在循环体中进行，每次读取操作完成之后，会对读取的字节数进行累加，代码如下：

```
if (totalReadAmount >= Integer.MAX_VALUE - localReadAmount) {
    // Avoid overflow.
    totalReadAmount = Integer.MAX_VALUE;
    break;
}

totalReadAmount += localReadAmount;
```

在累加之前，需要对长度上限做保护，如果累计读取的字节数已经发生溢出，则将读取到的字节数设置为整形的最大值，然后退出循环，原因是本次循环已经读取过多的字节，需要退出。否则会影响后面排队的 Task 任务和写操作的执行。如果没有溢出，则执行累加操作。

```
    if (localReadAmount < writable) {
        // Read less than what the buffer can hold,
        // which might mean we drained the recv buffer completely.
        break;
    }
```

最后，对本次读取的字节数进行判断，如果小于缓冲区可写的容量，说明 TCP 缓冲区已经没有就绪的字节可读，读取操作已经完成，需要退出循环。如果仍然有未读的消息，则继续执行读操作。连续的读操作会阻塞排在后面的任务队列中待执行的 Task，以及写操作，所以，对连续读操作做了上限控制，默认值为 16 次，无论 TCP 缓冲区有多少码流需要读取，只要连续 16 次没有读完，都需要强制退出，等待下次 selector 轮询周期再执行。

```
    }
} while (++ messages < maxMessagesPerRead);
```

完成多路复用器本轮读操作之后，触发 ChannelReadComplete 事件。随后调用接收缓冲区容量分配器的 Hanlder 的记录方法，将本次读取的总字节数传入到 record()方法中进行缓冲区的动态分配，为下一次读取选取更加合适的缓冲区容量，代码如下：

```
allocHandle.record(totalReadAmount);
```

上面我们提到，如果读到的返回值为-1，表明发生了 IO 异常，需要关闭连接，释放资源，代码如下：

```
if (close) {
    closeOnRead(pipeline);
    close = false;
}
```

至此，请求消息的异步读取源码分析我们已经完成。下面，我们继续分析写操作是如何执行的。

3.4. 写操作

3.4.1. 异步消息发送

Netty 的写操作和将消息真正刷新到 SocketChannel 中是分开的，因此我们分

成两个小结来介绍，首先介绍消息的写操作。

下面我们从 ChannelHandlerContext 开始分析，首先调用它的 write 方法，异步发送消息，代码如下：

```
@Override
public ChannelFuture write(Object msg, ChannelPromise promise) {
    DefaultChannelHandlerContext next = findContextOutbound(MASK_WRITE);
    next.invoker.invokeWrite(next, msg, promise);
    return promise;
}
```

类似 Mina 的 FilterChain，它实际上是个职责链，消息在职责链中传递，最终它会调用 HeadHandler 的 write 方法，代码如下：

```
@Override
public void write(ChannelHandlerContext ctx, Object msg, ChannelPromise promise) throws Exception {
    unsafe.write(msg, promise);
}
```

它由子类 AbstractUnsafe 实现，代码如下：

```
@Override
public void write(Object msg, ChannelPromise promise) {
    if (!isActive()) {
        // Mark the write request as failure if the channel is inactive.
        if (isOpen()) {
            promise.tryFailure(NOT_YET_CONNECTED_EXCEPTION);
        } else {
            promise.tryFailure(CLOSED_CHANNEL_EXCEPTION);
        }
        // release message now to prevent resource-leak
        ReferenceCountUtil.release(msg);
    } else {
        outboundBuffer.addMessage(msg, promise);
    }
}
```

首先对链路的状态进行判断，如果已经断开连接，则需要设置回调结果异常信息，同时，释放需要发送的消息。注意：此处的消息通常是经过编码后的 ByteBuf，因此，需要释放。

如果链路正常，则将需要发送的 ByteBuf 加入到 outboundBuffer 中，下面，

我们重点分析 ChannelOutboundBuffer 的 addMessage 方法。代码如下：

```
void addMessage(Object msg, ChannelPromise promise) {
    int size = channel.estimatorHandle().size(msg);
    if (size < 0) {
        size = 0;
    }

    Entry e = buffer[tail++];
    e.msg = msg;
    e.pendingSize = size;
    e.promise = promise;
    e.total = total(msg);

    tail &= buffer.length - 1;

    if (tail == flushed) {
        addCapacity();
    }

    // increment pending bytes after adding message to the unflushed arrays.
    // See https://github.com/netty/netty/issues/1619
    incrementPendingOutboundBytes(size);
}
```

首先，我们获取 ByteBuf 的可读字节数，实际上也就是需要发送的字节数。

然后，从环形 Entry 数组中获取可用的 Entry，将指针+1，接着进行一系列的赋值操作，例如将 Entry 的 Message 设置为需要发送的 ByteBuf 等。设置完成后需要进行一次判断，如果当前指针已经达到唤醒数组的尾部，即：tail = buffer.length; 此时需要重新将指针调整为起始位置 0。由于环形数组的初始容量为 32，后面容量的扩张是 32 的 N 倍，所以通过&操作就能将指针重新指到起始位置，实现环形队列，代码如下：

```
tail &= buffer.length - 1;
```

指针重绕后，需要对尾部指针 tail 和需要刷新的位置 flushed 进行判断，如果两者相等，说明指针重绕后已经到达需要刷新的位置，再继续使用就会覆盖尚未发送的消息，因此，需要对环形队列进行动态扩容，动态扩展的代码如下：

```
private void addCapacity() {
    int p = flushed;
    int n = buffer.length;
    int r = n - p; // number of elements to the right of p
    int s = size();

    int newCapacity = n << 1;
    if (newCapacity < 0) {
        throw new IllegalStateException();
    }

    Entry[] e = new Entry[newCapacity];
    System.arraycopy(buffer, p, e, 0, r);
    System.arraycopy(buffer, 0, e, r, p);
    for (int i = n; i < e.length; i++) {
        e[i] = new Entry();
    }

    buffer = e;
    flushed = 0;
    unflushed = s;
    tail = n;
}
```

第一步：将尚未刷新的消息拷贝到新的数组中

第二步：将已经释放的 Entry 拷贝填充到数组组中

第三步：填充新扩容的数组

首先，保存需要刷新的位置索引，计算还有多少个消息没有被刷新，然后执行扩容操作，将环形数组的 Size 扩展为原来的 2 倍。扩容以后，需要对新的环形数组进行填充，填充分为三步：

1. 将尚未刷新的消息拷贝到数组的首部；
2. 原来数组中已经刷新并释放的 Entry 可以重用，所以，将其拷贝到尚未刷新消息的后面；
3. 最后扩容的数组全部重新初始化。

对扩容后的数组初始化后，需要对指针进行重新置位，具体如下：

1. 由于尚未刷新的消息在数组首部，所以 flushed 为 0；
2. 由于未刷新的消息从 0 开始，所以 unflushed = unflushed - flushed & buffer.length - 1；
3. 下次新的消息写入需要放入扩容后的数组中，所以 tail = buffer.length。（这个设置值得推敲）。

将需要发送的消息写入环形发送数组之后，计算当前需要发送消息的总字节数是否达到一次发送的高水位线，如果达到，触发 hannelWritabilityChanged 事件，代码如下：

```
void incrementPendingOutboundBytes(int size) {
    // Cache the channel and check for null to make sure we not produce a NPE in case of the Channel gets
    // recycled while process this method.
    Channel channel = this.channel;
    if (size == 0 || channel == null) {
        return;
    }

    long oldValue = totalPendingSize;
    long newWriteBufferSize = oldValue + size;
    while (!TOTAL_PENDING_SIZE_UPDATER.compareAndSet(this, oldValue, newWriteBufferSize)) {
        oldValue = totalPendingSize;
        newWriteBufferSize = oldValue + size;
    }

    int highWaterMark = channel.config().getWriteBufferHighWaterMark();

    if (newWriteBufferSize > highWaterMark) {
        if (WRITABLE_UPDATER.compareAndSet(this, 1, 0)) {
            channel.pipeline().fireChannelWritabilityChanged();
        }
    }
```

这段代码理解起来非常简单，不再展开说明。只对红框中标出的部分做解释：

它仿照了 JDK1.5 以后新增的原子类的自旋操作解决多线程并发操作问题，循环判断，如果需要更新的变量值没有发生变化并且更新成功退出，否则取其它线程更新后的新值重新计算并重新赋值，这个就是自旋，通过它可以解决多线程并发修改一个变量的无锁化问题。

至此，我们完成了消息异步发送的代码分析，接下来，我们继续分析消息的刷新操作，flush 负责将发送环形数组中缓存的消息写入到 SocketChannel 中发送给对方。

3.4.2. Flush 操作

Flush 操作负责将 ByteBuffer 消息写入到 SocketChannel 中发送给对方，下面我们首先从发起 Flush 操作的类入口，进行详细分析。

DefaultChannelHandlerContext 的 flush 方法，代码如下：

```
@Override
public ChannelHandlerContext flush() {
    DefaultChannelHandlerContext next = findContextOutbound(MASK_FLUSH);
    next.invoker.invokeFlush(next);
    return this;
}
```

最终会调用的 HeadHandler 的 flush 操作，代码如下：

```
@Override
public void flush(ChannelHandlerContext ctx) throws Exception {
    unsafe.flush();
}
```

我们重点分析 AbstractUnsafe 的 flush 操作，代码如下：

```
public void flush() {
    ChannelOutboundBuffer outboundBuffer = this.outboundBuffer;
    if (outboundBuffer == null) {
        return;
    }

    outboundBuffer.addFlush();
    flush0();
}
```

首先将发送环形数组的 unflushed 指针修改为 tail，标识本次要发送的消息范围。然后调用 flush0 进行发送，由于 flush0 代码非常简单，我们重点分析 doWrite 方法，代码如下：

```
@Override
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    for (;;) {
        // Do non-gathering write for a single buffer case.
        final int msgCount = in.size();
        if (msgCount <= 1) {
            super.doWrite(in);
            return;
        }
```

首先计算需要发送的消息个数（unflushed - flush）,如果只有 1 个消息需要发送，则调用父类的写操作，我们分析 AbstractNioByteChannel 的 doWrite()方法，代码如下：

```
@Override
protected void doWrite(ChannelOutboundBuffer in) throws Exception {
    int writeSpinCount = -1;

    for (;;) {
        Object msg = in.current(true);
        if (msg == null) {
            // Wrote all messages.
            clearOpWrite();
            break;
        }
```

因为只有一条消息需要发送，所以直接从 ChannelOutboundBuffer 中获取当前需要发送的消息，代码如下：

```
public Object current(boolean preferDirect) {
    if (isEmpty()) {
        return null;
    } else {
        // TODO: Think of a smart way to handle ByteBufHolder messages
        Object msg = buffer[flushed].msg;
        if (threadLocalDirectBufferSize <= 0 || !preferDirect) {
            return msg;
        }
        if (msg instanceof ByteBuf) {
            ByteBuf buf = (ByteBuf) msg;
            if (buf.isDirect()) {
                return buf;
            } else {
                int readableBytes = buf.readableBytes();
                if (readableBytes == 0) {
                    return buf;
                }

                // Non-direct buffers are copied into JDK's own internal direct buffer on every I/O.
                // We can do a better job by using our pooled allocator. If the current allocator does not
                // pool a direct buffer, we use a ThreadLocal based pool.
                ByteBufAllocator alloc = channel.alloc();
                ByteBuf directBuf;
                if (alloc.isDirectBufferPooled()) {
                    directBuf = alloc.directBuffer(readableBytes);
                } else {
                    directBuf = ThreadLocalPooledByteBuf.newInstance();
                }
```

首先，获取需要发送的消息，如果消息为 ByteBuf 且它分配的是 JDK 的非堆内存，则直接返回。

对返回的消息进行判断，如果为空，说明该消息已经发送完成并被回收，然后执行清空 OP_WRITE 操作位的 clearOpWrite 方法，代码如下：

```
protected final void clearOpWrite() {
    final SelectionKey key = selectionKey();
    final int interestOps = key.interestOps();
    if ((interestOps & SelectionKey.OP_WRITE) != 0) {
        key.interestOps(interestOps & ~SelectionKey.OP_WRITE);
    }
}
```

继续向下分析，如果需要发送的 ByteBuf 已经没有可写的字节，说明已经发送完

```
ByteBuf buf = (ByteBuf) msg;
int readableBytes = buf.readableBytes();
if (readableBytes == 0) {
    in.remove();
    continue;
}
```

成，将该消息从环形队列中删除，然后继续循环。下面我们分析下 ChannelOutboundBuffer 的 remove 方法:

```
public boolean remove() {
    if (isEmpty()) {
        return false;
    }

    Entry e = buffer[flushed];
    Object msg = e.msg;
    if (msg == null) {
        return false;
    }

    ChannelPromise promise = e.promise;
    int size = e.pendingSize;

    e.clear();

    flushed = flushed + 1 & buffer.length - 1;

    safeRelease(msg);

    promise.trySuccess();
    decrementPendingOutboundBytes(size);

    return true;
}
```

首先判断环形队列中是否还有需要发送的消息，如果没有，则直接返回。如果非空，则首先获取 Entry，然后对其进行资源释放，同时把需要发送的索引 flushed 进行更新。所有操作执行完之后，调用 decrementPendingOutboundBytes 减去已经发送的字节数，该方法跟 incrementPendingOutboundBytes 类似，会进行发送低水位的判断和事件通知，此处不再赘述。

我们接着继续对消息的发送进行分析，代码如下：首先将半包标致设置为 false,

```
    boolean setOpWrite = false;
    boolean done = false;
    long flushedAmount = 0;
    if (writeSpinCount == -1) {
        writeSpinCount = config().getWriteSpinCount();
    }
    for (int i = writeSpinCount - 1; i >= 0; i --) {
        int localFlushedAmount = doWriteBytes(buf);
        if (localFlushedAmount == 0) {
            setOpWrite = true;
            break;
        }

        flushedAmount += localFlushedAmount;
        if (!buf.isReadable()) {
            done = true;
            break;
        }
    }
```

第一步：对写入的字节个数进行判读，如果为 0 说明 TCP 的发送缓冲区已满，需要退出并监听写操作

从 DefaultSocketChannelConfig 中获取循环发送的次数，进行循环发送，对发送方法 doWriteBytes 展开分析，如下：

```
@Override
protected int doWriteBytes(ByteBuf buf) throws Exception {
    final int expectedWrittenBytes = buf.readableBytes();
    final int writtenBytes = buf.readBytes(javaChannel(), expectedWrittenBytes);
    return writtenBytes;
}
```

对于红框中的代码说明如下：ByteBuf 的 readBytes()方法的功能是将当前 ByteBuf 中的可写字节数组写入到指定的 Channel 中。方法的第一个参数是 Channel，此处就是 SocketChannel，第二个参数是写入的字节数组长度，它等于 ByteBuf 的可读字节数，返回值是写入的字节个数。由于我们将 SocketChannel 设置为异步非阻塞模式，所以写操作不会阻塞。

从写操作中返回，需要对写入的字节数进行判断，如果为 0，说明 TCP 发送缓冲

区已满，不能继续再向里面写入消息，因此，将写半包标致设置为 true，然后退出循环，执行后续排队的其它任何或者读操作，等待下一次 Selector 的轮询继续触发写操作。

对写入的字节数进行累加，判断当前的 ByteBuf 中是否还有没有发送的字节，如果没有可发送的字节，则将 done 设置为 true，退出循环。

从循环发送状态退出后，首先根据实际发送的字节数更新发送进度，实际就是发送的字节数和需要发送的字节数的一个比值。执行完成进度更新后，判断本轮循环是否将需要发送的消息中所有需要发送的字节全部发送完成，如果发送完成，

```
in.progress(flushedAmount);

if (done) {
    in.remove();
} else {
    incompleteWrite(setOpWrite);
    break;
}
```

则将该消息从循环队列中删除；否则，将设置多路复用器的 OP_WRITE 操作位，用于通知 Reactor 线程还有没有发送完成的消息，需要继续发送，直到全部发送完成。

好，到此我们分析完了单条消息的发送，现在我们重新将注意力转回到 NioSocketChannel，看看多条消息的发送过程，代码如下：

```
// Ensure the pending writes are made of ByteBufs only.
ByteBuffer[] nioBuffers = in.nioBuffers();
if (nioBuffers == null) {
    super.doWrite(in);
    return;
}
```

从 ChannelOutboundBuffer 获取需要发送的 ByteBuffer 列表，由于 Netty 使用的是 ByteBuf，因此，需要做下内部类型转换，代码如下：

```
public ByteBuffer[] nioBuffers() {
    long nioBufferSize = 0;
    int nioBufferCount = 0;
    final int mask = buffer.length - 1;
    final ByteBufAllocator alloc = channel.alloc();
    ByteBuffer[] nioBuffers = this.nioBuffers;
    Object m;
    int i = flushed;
    while (i != unflushed && (m = buffer[i].msg) != null) {
        if (!(m instanceof ByteBuf)) {
            this.nioBufferCount = 0;
            this.nioBufferSize = 0;
            return null;
        }
```

声明各种局部变量并赋值，从 flushed 开始循环获取需要发送的 ByteBuf。首先对要发送的 Message 进行判断，如果不是 Netty 的 ByteBuf，则返回空。

```
    Entry entry = buffer[i];
    ByteBuf buf = (ByteBuf) m;
    final int readerIndex = buf.readerIndex();
    final int readableBytes = buf.writerIndex() - readerIndex;

    if (readableBytes > 0) {
        nioBufferSize += readableBytes;
        int count = entry.count;
        if (count == -1) {
            entry.count = count = buf.nioBufferCount();
        }
```

获取可写的字节个数，如果大于 0，对需要发送的缓冲区字节总数进行累加。然后从当前 Entry 中获取 ByteBuf 包含的最大 ByteBuffer 个数。

对包含的 ByteBuffer 个数进行累加，如果超过 ChannelOutboundBuffer 预先分配

```
int neededSpace = nioBufferCount + count;
if (neededSpace > nioBuffers.length) {
    this.nioBuffers = nioBuffers = expandNioBufferArray(nioBuffers, neededSpace, nioBufferCount);
}
```

的数组上限，则进行数组扩张。扩张的代码如下：

```
private static ByteBuffer[] expandNioBufferArray(ByteBuffer[] array, int neededSpace, int size) {
    int newCapacity = array.length;
    do {
        // double capacity until it is big enough
        // See https://github.com/netty/netty/issues/1890
        newCapacity <<= 1; 翻倍扩张，避免频繁的数组拷贝

        if (newCapacity < 0) {
            throw new IllegalStateException();
        }

    } while (neededSpace > newCapacity);

    ByteBuffer[] newArray = new ByteBuffer[newCapacity];
    System.arraycopy(array, 0, newArray, 0, size);

    return newArray;
}
```

由于频繁的数组扩张会导致频繁的数组拷贝，影响性能，所以，Netty 采用了翻倍扩张的方式，新的数组创建之后，将老的数据内容拷贝到新创建的数组中返回。

ByteBuffer 创建完成之后，需要将要刷新的 ByteBuf 转换成 ByteBuffer 并存到发送数据中。由于 ByteBuf 的实现不同，所以，它们内部包含的 ByteBuffer 个数是不同的，例如 UnpooledHeapByteBuf，它基于 JVM 堆内存的字节数组实现，它

```
if (buf.isDirect() || threadLocalDirectBufferSize <= 0) {
    if (count == 1) {
        ByteBuffer nioBuf = entry.buf;
        if (nioBuf == null) {
            // cache ByteBuffer as it may need to create a new ByteBuffer instacne if its a
            // derived buffer
            entry.buf = nioBuf = buf.internalNioBuffer(readerIndex, readableBytes);
        }
        nioBuffers[nioBufferCount ++] = nioBuf;
```

只包含 1 个 ByteBuffer。对 Entry 中缓存的 ByteBuffer 进行判断，如果为空，则调用 ByteBuf 的 internalNioBuffer 方法，将当前的 ByteBuf 转换为 JDK 的 ByteBuffer，我们以 UnpooledHeapByteBuf 为例看下 internalNioBuffer 的实现：

```
@Override
public ByteBuffer internalNioBuffer(int index, int length) {
    return (ByteBuffer) internalNioBuffer().clear().position(index).limit(index + length);
}
```

获取 ByteBuffer 实例，然后调用它的 clear()方法对它的指针进行初始化，随后将 Position 指针设置为 index, limit 指针设置为 index + length。这些初始化操作完成之后 ByteBuffer 就可以被正确的读写。

下面我们看另一个分支，如果 ByteBuf 包含 NIO ByteBuffer 数组，那就获取 Entry

```
} else {
    ByteBuffer[] nioBufs = entry.buffers;
    if (nioBufs == null) {
        // cached ByteBuffers as they may be expensive to create in terms of Object allocation
        entry.buffers = nioBufs = buf.nioBuffers();
    }
    nioBufferCount = fillBufferArray(nioBufs, nioBuffers, nioBufferCount);
}
```

缓存的 ByteBuffer 数组，如果为空，则从当前需要刷新的 ByteBuf 中获取它的 ByteBuffer 数组。完成赋值操作后，调用 fillBufferArray 进行赋值。

对循环变量 i 赋值，完成本轮循环，代码如下：

```
        }
    }
    i = i + 1 & mask;
}
```

当 i = unflushed 时，说明需要刷新的消息全部赋值完成，循环执行结束。

对 ByteBuffer 数组进行判断，看是否还有单个需要发送的消息，如果没有则直接

```
if (nioBuffers == null) {
    super.doWrite(in);
    return;
}
```

返回，有则发送。

在批量发送缓冲区的消息之前，先对一系列的局部变量进行赋值，首先，获取需要发送的 ByteBuffer 数组个数 nioBufferCnt，然后，从 ChannelOutboundBuffer 中获取需要发送的总字节数，从 NioSocketChannel 中获取 NIO 的 SocketChannel, 是否发送完成标识设置为 false，是否有写半包标致设置为 false。

```
int nioBufferCnt = in.nioBufferCount();
long expectedWrittenBytes = in.nioBufferSize();

final SocketChannel ch = javaChannel();
long writtenBytes = 0;
boolean done = false;
boolean setOpWrite = false;
```

继续分析循环发送的代码，代码如下：

```
for (int i = config().getWriteSpinCount() - 1; i >= 0; i --) {
    final long localWrittenBytes = ch.write(nioBuffers, 0, nioBufferCnt);
```

就像循环读一样，我们需要对一次 Selector 轮询的写操作次数进行上限控制，因为如果 TCP 的发送缓冲区满，TCP 处于 KEEP-ALIVE 状态，消息是发送不出去的，如果不对上限进行控制，就会常时间的处于发送状态，Reactor 线程无法及时读取其它消息和执行排队的 Task。所以，我们必须对循环次数上限做控制。

调用 NIO SocketChannel 的 write 方法，它有三个参数：第一个是需要发送的 ByteBuffer 数组，第二个是数组的偏移量，第三个参数是发送的 ByteBuffer 个数。返回值是写入 SocketChannel 的字节个数。

下面对写入的字节进行判断，如果为 0，说明 TCP 发送缓冲区已满，再写很有可能还是写不进去，因此从循环中跳出，同时将写半包标识设置为 True,用于向多

```
if (localWrittenBytes == 0) {
    setOpWrite = true;
    break;
}
```

路复用器注册写操作位，告诉多路复用器有没发完的半包消息，你要继续轮询出就绪的 SocketChannel 继续发送。

```
expectedWrittenBytes -= localWrittenBytes;
writtenBytes += localWrittenBytes;
if (expectedWrittenBytes == 0) {
    done = true;
    break;
}
```

发送操作完成后进行两个计算：需要发送的字节数要减去已经发送的字节数；发送的字节总数+已经发送的字节数。更新完这两个变量后，判断缓冲区中所有的消息是否已经发送完成，如果是，则把发送完成标识设置为 True 同时退出循环。如果没有发送完成，则继续循环。

从循环发送中退出之后，首先对发送完成标识 done 进行判断，如果发送完成，则循环释放已经发送的消息，代码如红框中标识所示：

环形数组的发送缓冲区释放完成后，取消半包标识，告诉多路复用器消息已经全部发送完成。

```
if (done) {
    // Release all buffers
    for (int i = msgCount; i > 0; i --) {
        in.remove();
    }

    // Finish the write loop if no new messages were flushed by in.remove().
    if (in.isEmpty()) {
        clearOpWrite();
        break;
    }
```

当缓冲区中的消息没有发送完成，甚至某个 ByteBuffer 只发送了一半，出现了半包发送，该怎么办？下面我们继续看看 Netty 是如何处理的。

```
for (int i = msgCount; i > 0; i --) {
    final ByteBuf buf = (ByteBuf) in.current();
    final int readerIndex = buf.readerIndex();
    final int readableBytes = buf.writerIndex() - readerIndex;

    if (readableBytes < writtenBytes) {
        in.progress(readableBytes);
        in.remove();
        writtenBytes -= readableBytes;
    } else if (readableBytes > writtenBytes) {
        buf.readerIndex(readerIndex + (int) writtenBytes);
        in.progress(writtenBytes);
        break;
    } else { // readableBytes == writtenBytes
        in.progress(readableBytes);
        in.remove();
        break;
    }
}
```

首先，我们循环遍历发送缓冲区，对消息的发送结果进行分析，下面具体展开进行说明：

1. 从 ChannelOutboundBuffer 弹出第一条发送的 ByteBuf; 然后获取该 ByteBuf 的可读索引和可读字节数；
2. 对可读字节数和发送的总字节数进行判断，如果发送的字节数大于可读的字节说，说明它已经被完全发送出去，更新 ChannelOutboundBuffer 的发送进度信息，将已经发送的 ByteBuf 删除，释放相关资源，最后，发送的字节数要减去第一条发送的字节数，就是后面消息发送的总字节数；然后继续循环判断第二条消息、第三条消息.....
3. 如果可读的消息大于已经发送的总消息数，说明这条消息没有被完全发送成功，也就是出现了所谓的“写半包”，此时，需要更新可读的索引为当前索引 + 已经发送的总字节数，然后更新 ChannelOutboundBuffer 的进度信息，退出循环；
4. 如果可读字节数等于已经发送的字节数总和，则说明最后一次发送的消息是个全包消息，更新发送进度信息，将最后一条完全发送的消息从缓冲区中删除，最后退出循环。

最后，因为缓冲区中待刷新的消息没有全部发送完成，所以需要更新 SocketChannel 的注册监听位，将其修改为 OP_WRITE, 在下一次轮询中继续发送没有发送出去的消息。

至此，消息的异步发送和刷新全部分析完成。

4. Netty 架构

4.1. 逻辑架构

Netty 采用了比较典型的三层网络架构进行设计和开发，逻辑架构图如下所示：

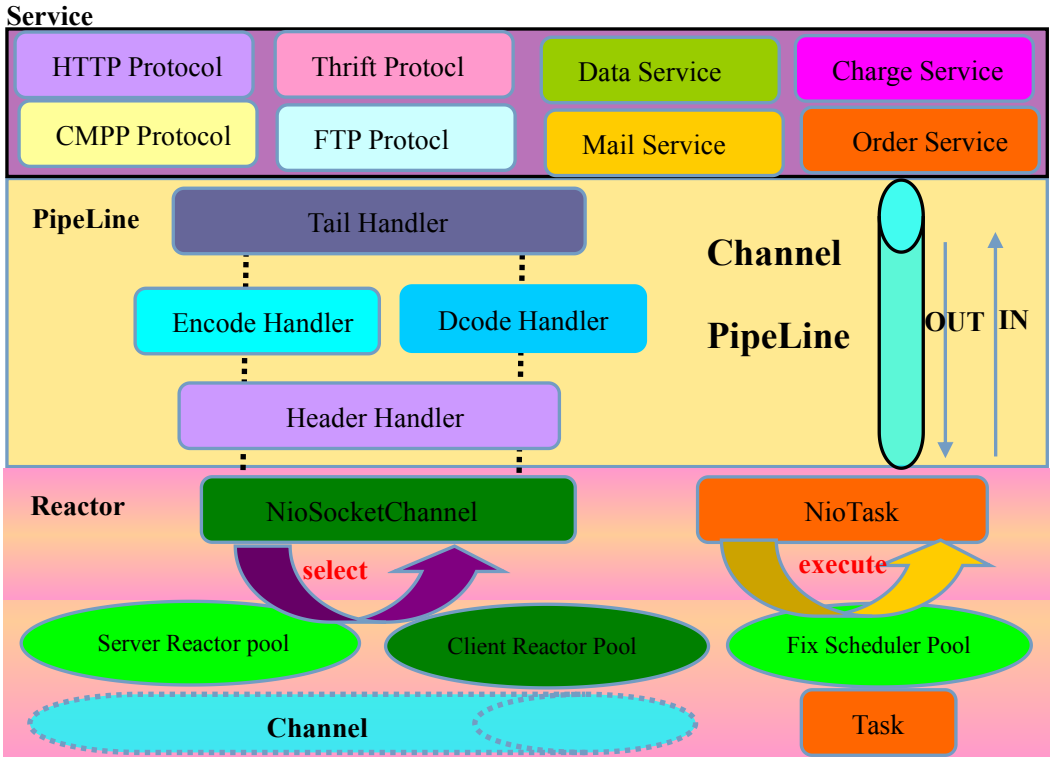


图 4.1-1 Netty 逻辑架构图

第一层：Reactor 通信调度层，它由一系列辅助类完成，包括 Reactor 线程 NioEventLoop 以及其父类、NioSocketChannel/NioServerSocketChannel 以及其父类、ByteBuffer 以及由其衍生出来的各种 Buffer、Unsafe 以及其衍生出的各种内部类等。该层的主要职责就是监听网络的读写和连接操作，负责将网络层的数据读取到内存缓冲区中，然后触发各种网络事件，例如连接创建、连接激活、读事件、写事件等等，将这些事件触发到 PipeLine 中，由 PipeLine 充当的职责链来进行后续的处理；

第二层：职责链 PipeLine，它负责事件在职责链中的有序传播，同时负责动态的编排职责链，职责链可以选择监听和处理自己关心的事件，它可以拦截处理和向后/向前传播事件，不同的应用的 Handler 节点的功能也不同，通常情况下，往往会开发编解码 Hanlder 用于消息的编解码，它可以将外部的协议消息转换成内部的 POJO 对象，这样上层业务侧只需要关心处理业务逻辑即可，不需要感知底层的协议差异和线程模型差异，实现了架构层面的分层隔离；

第三层：业务逻辑编排层，业务逻辑编排层通常有两类：一类是纯粹的业务逻辑编排，还有一类是其它的应用层协议插件，用于特性协议相关的会话和链路管理，例如 CMPP 协议，用于管理和中国移动短信的对接。

架构的不同层次，需要关心和处理的对象都不同，通常情况下，对于业务开发，只需要关心第二和第三层即可，由于应用层协议栈往往是开发一次，到处运行。这样，实际上对于业务开发和使用者来说，只需要关心第三层的业务逻辑开发即可。各种应用协议以插件的形式提供，只有协议开发人员关注和管理，其他业务开发人员不需要关心。这种分层的架构设计理念实现了 NIO 框架各层之间的解耦，非常方便上层业务协议栈的开发和业务的定制。

正是由于 Netty 的分层架构设计非常合理，基于 Netty 的各种应用服务器和协议栈开发才能够如雨后春笋般的得到快速发展。

5. 附录

5.1. 作者简介

李林锋：华为技术有限公司平台中间件架构与设计部 架构师设计师，电信行业工作 6 年，NIO 领域 4 年工作经验，曾参与多个重要的电信软件平台设计和骨架代码开发工作。2010 年，参与设计和开发的某平台产品获得软件公司总裁技术创新奖二等奖。

联系方式：新浪微博：李林锋 hw 邮箱：neu_lilinfeng@sina.com

个人主页：http://blog.sina.com.cn/u/1725503810

5.2. 使用声明