Java 网络编程

大部分程序用来从数据源读取数据，经过处理后将数据流输出到目的地，如下图所示：



数据输入和输出包括：

* Files，文件
* Pipes，管道
* Network Connections，网络连接
* In-Memory Buffers，内存缓存，例如Array
* System.in,System.out及System.error，Java标准输入、输出和错误输出

# Java IO

Java IO是用来读写数据(输入和输出)的API，流是核心的概念，其是一个连续的数据流，其提供了各种流类和接口用以获取不同种类的数据，并通过标准的方法输入和输出数据，在Java IO中流分为两种形式：

* 字节流(8 bit)，以字节为单位进行读写
* 字符流(16 bit)，以字符为单位进行读写

对于这两种流，其输入输出接口是不同的，如下表所示：

|  |  |  |
| --- | --- | --- |
| 抽象基类 | 字节流 | 字符流 |
| 输入流 | InputStream | Reader |
| 输出流 | OutputStream | Writer |

## 1.1 流使用示例

下面以文件流的使用为例，详述这两种流的使用方法

1. **文件字符流**

*FileReader fr = new FileReader("e:\\fileop.txt");*

*char[] buf = new char[1024];*

*int len = 0;*

*while ( (len = fr.read(buf)) != -1) {*

*System.out.println(new String(buf,0,len));}*

*fr.close();*

*FileWriter fw = new FileWriter("e:\\fileop.txt");*

*fw.write("FW Write MSG");*

*fw.close();*

文件流读数据的使用步骤如下：

* 创建流对象，将已存在的文件加载进流

*FileReader fr = new FileReader("e:\\fileop.txt");*

* 创建临时存放数据的数组

*char[] ch = new char[1024]*

* 调用流对象的读取方法将流中的数据读入到数组中

*ch = fr.read(buf)*

其文件写入过程类似。

1. **文件字节流**

为了提高数据的读写速度，Java API提供带缓冲功能的流类，使用这些流类中，会创建内部缓冲区数组，根据数据操作单位可以将缓冲流分为：

* BufferedInputStream和BufferedOutputStream，以字节流为单位
* BufferedReader和BufferedWriter，以字符流为单位

缓冲流要套接在对应的节点流之上，对读写的数据提供缓冲功能，提高了读写的效率，对于输出的缓冲流写出的数据先在内存中缓冲，使用flush将会使内存中的数据写出。

*File f = new File("e:\\fileop.txt");*

*InputStream in = new FileInputStream(f);*

*byte b[] = new byte[(int)f.length()];*

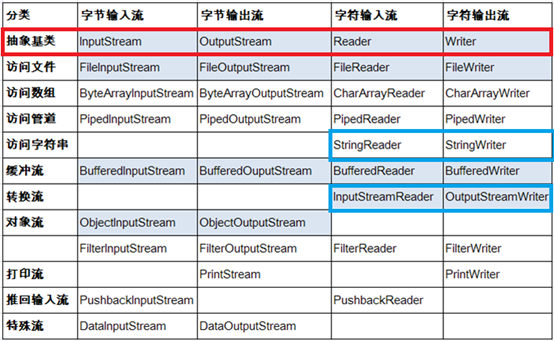
*in.read(b);*

*in.close();*

*System.out.println(new String(b));*

## **1.2 Java IO类**

下表是通过输入、输出及基于字节或者字符，以及其他比如缓冲、解析之类的特点用途来划分的Java IO类的表格：



1. **InputStream&Reader接口**

InputStream和Reader是所有输入流的基类，类图如下所示：



对于普通文件流FileInputStream，其从文件中获得字节流，其核心成员变量为：

*private final FileDescriptor fd; //文件描述符，打开的连接*

*private final String path; //文件路径*

*private FileChannel channel = null;*

BufferedInputStream缓冲输入流，其本质上通过内部缓冲区数组实现的，其缓冲区为:

*protected volatile byte buf[];*

读入输入流数据时，将数据分配填入到缓冲区中，如下图所示：

*private byte[] getBufIfOpen() throws IOException {*

*byte[] buffer = buf;*

*if (buffer == null)*

*throw new IOException("Stream closed");*

*return buffer;*

*}*

1. OutputStream&Writer

其是所有输出类的基类，如下所示：



详细使用不再介绍

http://ifeve.com/java-io/

# **Java网络编程**

网络连接是Java IO主要操作的资源，在Linux中将所有外部设备都看做一个文件来操作，对一个文件的读写操作会调用内存提供的系统命令，返回文件描述符(fd, file descriptor)，对于一个Socket(网络连接)的读写也会有相应的描述符，称为socketfd(socket描述符)，其指向内核中的结构体，在SocketImpl中，成员变量如下：

*public abstract class SocketImpl implements SocketOptions {*

*Socket socket = null;*

*ServerSocket serverSocket = null;*

*protected FileDescriptor fd; //代表该socket的文件描述符*

*protected InetAddress address;*

*protected int port;*

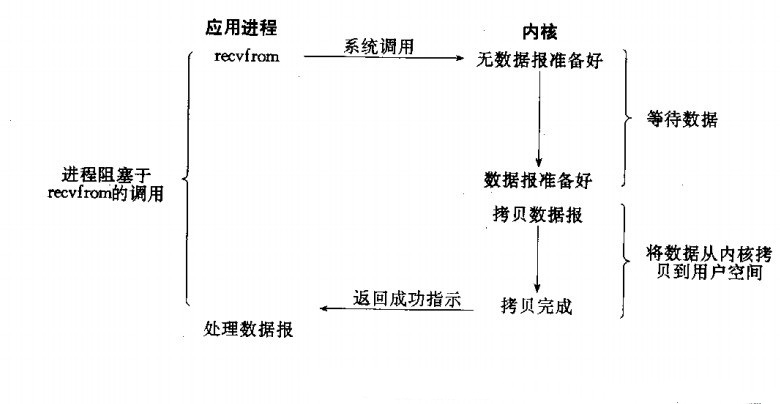
*}*

在java.net包中提供两种常见的网络协议支持：

* TCP，传输控制协议，其保障了两个应用程序之间的可靠通信，用于互联网协议
* UDP，用户数据报协议，无连接协议，提供应用程序之间要发送数据的数据包

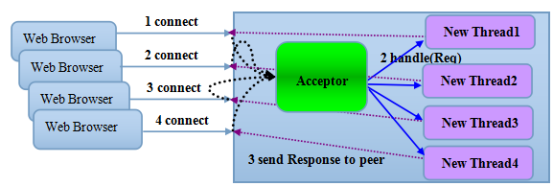
## **2.1 Socket编程(BIO)**

Linux网络编程中最常用的是阻塞IO模型，缺省情形下所有文件操作都是阻塞的，其执行流程如下图所示：



Socket(套接字)使用TCP提供两台计算机之间的通信机制，客户端程序创建一个套接字，并尝试连接服务器的套接字。当连接建立时，服务器会创建一个Socket对象，客户端和服务器通过Socket对象的写入和读取来进行通信。

在JDK 1.4推出NIO之前，基于Java的所有Socket通信都是采用同步阻塞模式(BIO)，同步阻塞I/O模型如下图：



采用BIO通信模型的服务端，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创建新的线程处理请求信息，处理完成之后返回应答消息给客户端，最后将线程销毁。这是典型的一对一的应答模型。

Java Socket分为客户端Socket和服务端ServerSocket，下面是使用示例：

1. **服务端ServerSocket使用**
   * 绑定特定端口创建ServerSocket对象
   * 使用ServerSocket的accept方法监听这个端口的请求连接，accept会一直阻塞到某个请求连接与客户端建立连接，此时accept将返回客户端与服务端的连接的Socket对象
   * 通过Socket对象的getInputStream与getOutputStream方法获得与客户端通信的输入流和输出流，进行通信交互
   * 完成交互后关闭连接

代码示例如下：

*public class BIOEchoServer {*

*private static final ExecutorService executor =*

*Executors.newCachedThreadPool();*

*public static void main(String[] args) throws IOException {*

*int port = 8082;*

*ServerSocket serverSocket = null;*

*try {*

*serverSocket = new ServerSocket(port);*

*Socket socket = null;*

*while(true) {*

*socket = serverSocket.accept();*

*executor.submit(new BIOEchoServerHandler(socket)); //处理客户端请求*

*}*

*} finally {*

*if(serverSocket != null) {*

*serverSocket.close();*

*}*

*}*

*}*

*}*

其中BIOEchoServerHandler的处理逻辑核心代码如下：

*BufferedReader reader = null;*

*BufferedWriter writer = null;*

*try {*

*reader = new BufferedReader(new InputStreamReader(this.socket.getInputStream()));*

*writer = new BufferedWriter(new OutputStreamWriter(this.socket.getOutputStream()));*

*while (true) {*

*String line = reader.readLine();*

*if(line == null) break;*

*writer.write(line + "\n");*

*writer.flush();*

*}*

服务器应用程序通过使用ServerSocket类获取端口，并侦听客户端请求

1. **客户端Socket的使用**
   * 创建Socket对象，使用创建Socket连接远程主机
   * 建立连接后，从Socket得到输入流与输出流，Socket是全双工通道，可以使用两个流与服务器之间相互发送数据

客户端代码示例如下：

*public class BIOEchoClient {*

*public static void main(String[] args) throws IOException {*

*int port = 8082;*

*String serverIP = "127.0.0.1";*

*Socket socket = null;*

*BufferedReader reader = null;*

*BufferedWriter writer = null;*

*try {*

*socket = new Socket(serverIP,port);*

*reader = new BufferedReader(new InputStreamReader(socket.getInputStream()));*

*writer = new BufferedWriter(new OutputStreamWriter(socket.getOutputStream()));*

*writer.write("Hello, Block IO.\n"); //输出及读取*

*writer.flush();*

*String echo = reader.readLine();*

*System.out.println("Echo:" + echo);*

*}....*

*}*

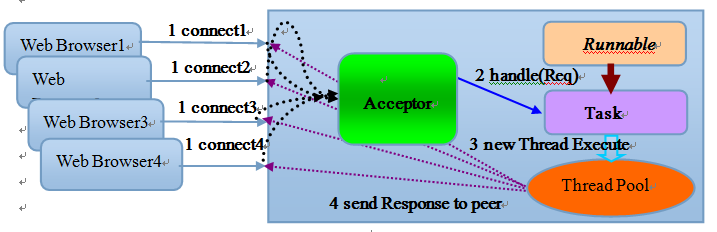
*}*

java.net.Socket类代表客户端和服务器都用来互相沟通的套接字，客户端获取Socket对象并实例化，服务器获取一个Socket对象则通过accept方法返回值。

1. 伪异步通信模型

Java BIO模型最大的问题是缺乏弹性伸缩能力，当客户端并发访问量增加后，服务端的线程个数和客户端并发访问数呈现1:1的正比关系，当线程数膨胀之后，系统的性能会急剧下降，当并发访问量继续增大，最终会导致进程宕机或者僵死，不能对外提供服务。

为了解决这个问题，一种常用做法是对线程模型进行优化，通过线程池来处理多个客户端的请求接入，形成客户端个数M与最大线程池数N的比例关系，其中M可以远大于N，通过对线程池进行线程池资源管理，防止由于海量并发导致的线程池耗尽，模型图如下所示：



即示例中

*while(true) {*

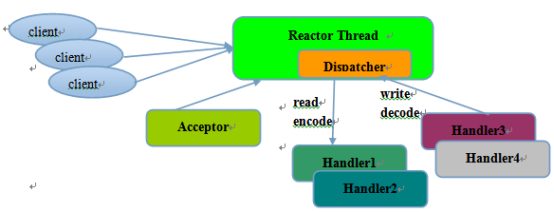
*socket = serverSocket.accept();*

*executor.submit(new BIOEchoServerHandler(socket)); //处理客户端请求*

*}*

## **2.2 NIO编程，Unblocking IO**

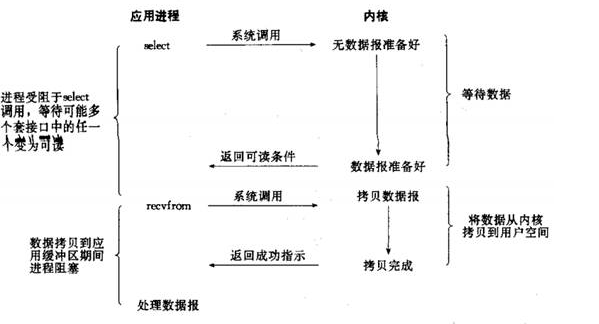
JDK 1.4之后引入NIO，弥补同步阻塞IO的不足，它在标准Java代码中提供高速、面向块的IO，其核心思路是处理多个客户端请求使用IO多路复用技术，其通信模型如下图原理如下：



NIO弥补了同步阻塞的不足，它在标准Java代码中提供了高速、面向块的I/O，通过定义包含数据的类，以及通过以块的形式处理这些数据。

### 2.2.1 NIO核心概念

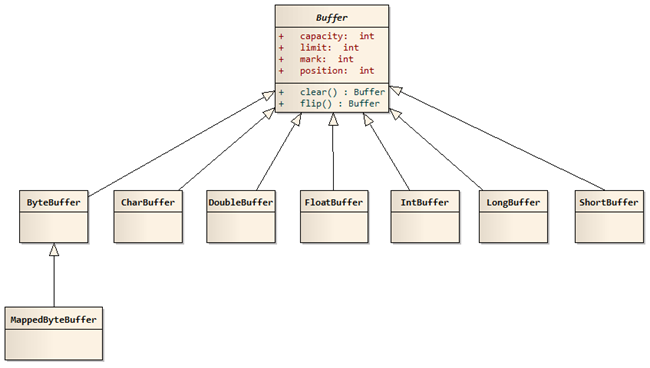
在NIO库中，所有数据都是用缓冲区(Buffer)中处理的，Buffer是一个对象，其包含一些要写入或者要读出的数据。在读取数据时，其直接读到缓冲区中，在写入数据时，写入缓冲区中，任何时候访问NIO中的数据，都是通过缓冲区进行操作，其I/O复用模型如下图所示：



1. **缓冲区Buffer**

缓冲区实质上是一个数组，通常它是字节数组(ByteBuffer)，也可以使用其他种类的数组。但是一个缓冲区不仅仅是数组，提提供了对数据结构化访问以及维护读写位置等信息。

最常用的缓冲区是ByteBuffer，其提供一组功能用户操作byte数组，除了ByteBuffer还有其他一些缓冲区，Java基本类型都对应一种缓冲区，具体如下：



每个Buffer类都是Buffer接口的子实例，Buffer类都有完全一样的操作，只是其所处理的数据类型不一样，ByteBuffer除了一般缓冲区的操作之外还提供一些特有操作，方便网络读写。



目前NIO库中支持两种ByteBuffer，HeapByteBuffer和DirectByteBuffer，这两种不同的地方在于分配的buffer是否在Java Heap上，DirectByteBuffer在性能上比HeapByteBuffer高几倍，HeapByteBuffer的方法如下：

*ByteBuffer.allocate(int capacity);参数大小为字节的数量*

DirectByteBuffer的分配方法如下：

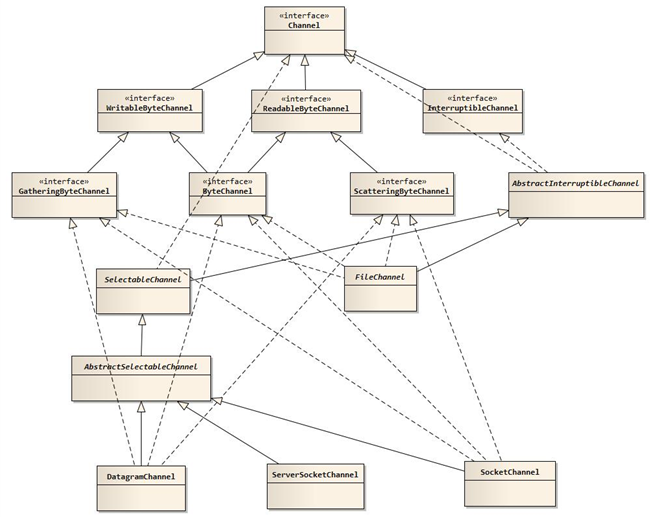
*ByteBuffer.allocateDirect(int capacity);*

在DirectByteBuffer中内存是通过unsafe.allocateMemory()来实现的，这个unsafe默认情况下java代码是没有能力可以调用到的，不过你可以通过反射的手段得到实例进而做操作，当然需要保证的是程序的稳定性，既然叫unsafe的，就是告诉你这不是安全的，其实并不是不安全，而是交给程序员来操作，它可能会因为程序员的能力而导致不安全，而并非它本身不安全。

1. **Channel**

通道是一个对象，网络数据都是通过Channel进行读取和写入，通道与流不同点在于通道是双向（全双工）的，流(InputStream或者OutputStream)只是在一个方向上移动，通道可以同时用于读写。Channel中的数据读写都是通过Buffer对象来处理，在执行时都要经过缓冲区。

在NIO中，提供多种通道对象，而所有的通道对象都实现了Channel接口，其类图如下所示：



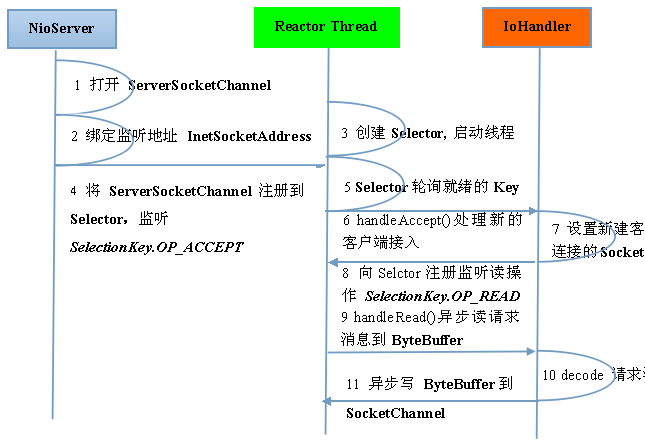
Channel分为两个大类：分别用于网络读写的SelectableChannel和用于文件操作的FileChannel。使用较多的ServerSocketChanel和SocketChannel都是SelectableChannel的子类。

1. **多路复用器Selector**

Java NIO编程的基础是Selector，其不断轮询注册在其上的Channel，如果某个Channel上面有新的TCP连接接入、读和写事件，这个Channel就处于就绪状态，Selector轮询出来后通过SelectionKey可以获取就绪的集合，进行后续的I/O操作。

多路复用器Selector可以同时轮询多个Channel，由于JDK使用了epoll代替传统的select实现，其并没有最大连接句柄1024/2048的限制，这也就意味着只需要一个线程负责Selector的轮询，就可以接入成千上万的客户端。

NIO服务端通信序列图如下所示：



### **2.2.2 NIO示例**

1) 定义NIO Server

*public class NIOServer {*

*public static void main(String[] args) throws IOException {*

*ServerSocketChannel ssChannel = ServerSocketChannel.open();*

*ssChannel.bind(new InetSocketAddress(9898));*

*ssChannel.configureBlocking(false);*

*Selector selector = Selector.open();*

*System.out.println("Selector Open Success");*

*ssChannel.register(selector, SelectionKey.OP\_ACCEPT);*

*while (selector.select() > 0 ) {*

*Iterator<SelectionKey> it = selector.selectedKeys().iterator();*

*while(it.hasNext()) {*

*SelectionKey key = it.next();*

*if(key.isAcceptable()) {*

*SocketChannel socketChannel = ssChannel.accept();*

*socketChannel.configureBlocking(false);*

*socketChannel.register(selector, SelectionKey.OP\_READ);*

*} else if (key.isReadable()) {*

*SocketChannel channel = (SocketChannel) key.channel();*

*readMsg(channel);*

*}*

*}*

*it.remove();}}*

*private static void readMsg(SocketChannel channel) throws IOException {*

*ByteBuffer buf = ByteBuffer.allocate(1024);*

*int len = 0;*

*System.out.println("Read Msg from Channel!");*

*while((len = channel.read(buf)) > 0) {*

*buf.flip();*

*byte[] bytes = new byte[1024];*

*buf.get(bytes,0,len);*

*System.out.println(new String(bytes,0,len));*

*}}}*

2)定义NIOClient

*public class NIOClient {*

*public static void main(String[] args) throws IOException{*

*SocketChannel sChannel = SocketChannel.open();*

*sChannel.connect(new InetSocketAddress("127.0.0.1",9898));*

*ByteBuffer buf = ByteBuffer.allocate(1024);*

*sChannel.configureBlocking(false);*

*Scanner scanner = new Scanner(System.in);*

*while(scanner.hasNext()) {*

*String msg = scanner.nextLine();*

*buf.put((new Date() + ": " + msg).getBytes());*

*buf.flip();*

*sChannel.write(buf);*

*buf.clear();*

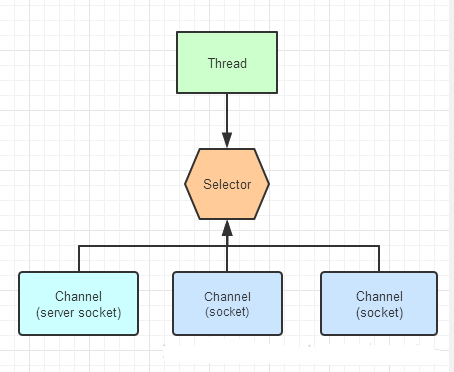
*}*

*}}*

### **2.2.3 Selector源码分析**

Java NIO是面向通道和缓冲区，数据总是从通道中读到buffer缓冲区内或者从buffer写入到通道中。在BIO模型下需要为每个Channel分配线程或者通过线程池处理请求，浪费资源，因此在Java NIO中使用Selector来检查Nio Channel的状态处于什么状态，依次管理多个网络连接，使用Selector管理多个Channel的好处是可以有更多的线程来处理channel。

使用Selector管理多个Channel的结构图如下所示：



下面是相关概念：

* Selector

管理被注册的通道集合信息和它们的就绪状态。通道是和选择器一起被注册的，并且使用选择器来更新通道的就绪状态。然后选择将被激发的线程挂起，直到有就绪的通道。

* SelectableChannel

可选择通道，这个抽象类提供了实现通道的可选择性所需要的公共方法。它是所有支持就绪检查的通道类的父类。所有Socket通道都是可选择的，包括从Pipe对象中获得的通道。SelectableChannel可以被注册到Selector对象上，同时可以指定对那个选择器而言。一个通道可以被注册到多个选择器上，但是对每个选择器而言只能被注册一次。

* SelectionKey

选择键封装了特定的通道与特定选择器的注册关系。选择键对象被SelectableChannel

register返回并提供一种表示这种注册关系的标记。选择键包含两个比特集(以整数的形式进行编码)，指示了该注册关系所关系的通道操作，以及通道已经准备好的操作。

Selector的使用流程如下：

1. **创建Selector**

Selector对象是通过调用静态工厂方法open来示例化的，如下：

*Selector selector = Selector.open();*

类方法open实际上通过默认的SelectorProvider对象获取一个新的实例，如下：

*public static Selector open() throws IOException {*

*return SelectorProvider.provider().openSelector();*

*}*

1. **将Channel注册到Selector**

要实现Selector管理Channel，需要将Channel注册到相应的Selector上，如下：

*ssChannel.configureBlocking(false);*

*Selector selector = Selector.open();*

*ssChannel.register(selector, SelectionKey.OP\_ACCEPT)*

通过调用通道的register方法将它注册到选择器上，与Selector一起使用时，Channel必须处于非阻塞模型下

1. **为SelectionKey绑定附加对象**

可以将一个或者多个附加对象绑定到SelectionKey上来识别给定的通道，有两种方法：

* 在注册的时候直接绑定

*if(key.isAcceptable()) {*

*SocketChannel socketChannel = ssChannel.accept();*

*socketChannel.configureBlocking(false);*

*socketChannel.register(selector, SelectionKey.OP\_READ); //将socket注册到Selector*

*}*

* 在绑定完成后附加

*selectionKey.attach(theObject);//绑定*

绑定之后，可以通过对应的SelectionKey取出该对象

1. **通过Selector选择通道**

选择器维护注册过的通道集合，这种注册关系封装在SelectionKey当中，在Selector中维护了三种类型的SelectionKey集合

*private Set<SelectionKey> publicKeys; //已注册key*

*Private Set<SelectionKey> cancelledKeys //已取消key集合*

*private Set<SelectionKey> publicSelectedKeys; //已选择 Key集合*

# **Netty NIO**

直接使用JDK NIO类库进行开发比较繁琐，需要熟练掌握Selector、ServerSocketChannel、SocketChannel、ByteBuffer等。由于NIO编程涉及到Reactor模型，必须对网络编程非常熟悉，才能编写出高质量的NIO程序。同时客户端面临断连重连、网络闪断、半包读写、失败缓存等。

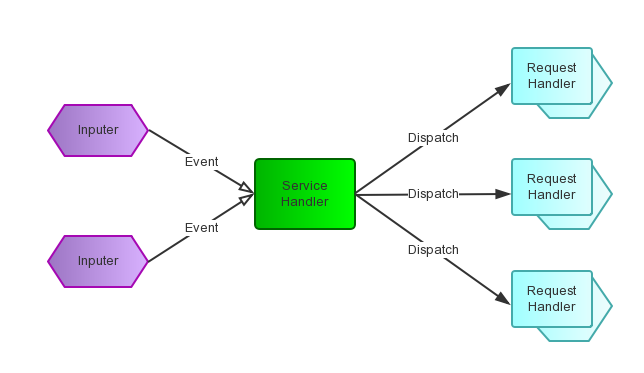
Netty是目前最流行的NIO框架之一，其在健壮性、功能、性能、可定制性和可扩展性已经在很多项目中验证，目前很多RPC框架使用Netty来构建高性能的异步通信功能。

## **3.1 Reactor模式**

Reacotr设计模式用于处理多客户端并发输入的事件处理模式，其使用多路复用器将请求分发到相关的请求处理handler上。核心概念如下：

1. 事件驱动
2. 一个或者多个并发输入源
3. Service Handler，同步将输入请求（event）多路复用的分发给Request Handler
4. Request Handler

如下图所示:



在结构上类似生产者消费者模式，即有一个或者多个生产者将事件放入一个Queue中，消费者主动从Queue中Poll事件来处理。在Reactor模式则没有Queue来做缓冲，每当Event输入到Service Handler之后，Service Handler主动的根据不同的Event类型将其分发给对应的Request Handler来处理

Java NIO可以很好的与Reactor模式结合，NIO的Reactor的核心是Selector，其核心和Reactor循环（Eveng Loop），在示例中代码如下：

*while (selector.select() > 0 ) {*

*Iterator<SelectionKey> it = selector.selectedKeys().iterator();*

*while(it.hasNext()) {*

*SelectionKey key = it.next();*

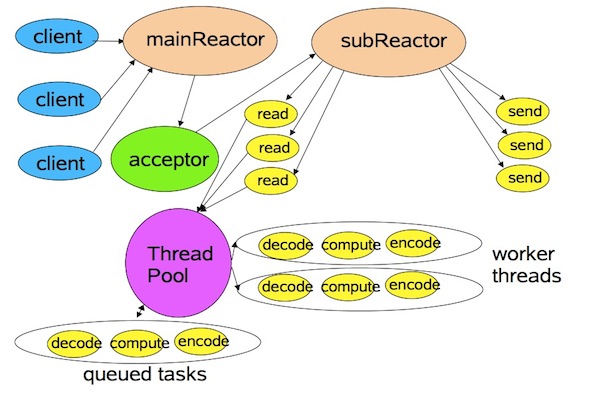
*dispatch(key)*

*}*

*}*

*}*

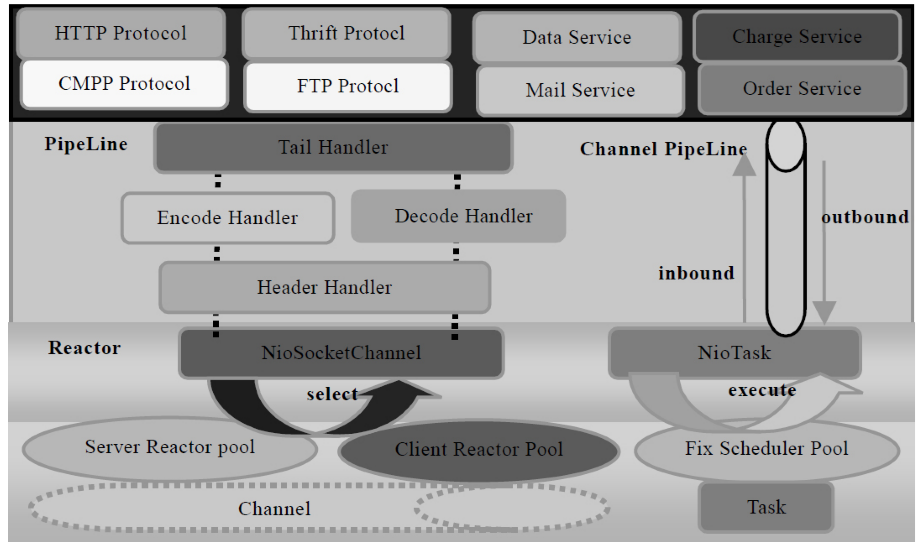
Netty中的Reactor模型，如下图所示：



在这个模式下，Main Reactor只有一个负责响应Client的连接请求，并建立连接，在Netty中使用NIO Selector。SubReactor可以有一个或者多个，每个SubReactor在独立的线程中执行，并且唯一一个独立的Selector。这样做的好处是，Sub Reactor会执行一些比较耗时的IO操作，例如消息的读写，使用多个线程执行有利于发挥CPU的运算能力，减少IO等待时间。

## **3.2 Netty逻辑架构**

逻辑架构图如下所示：



1. Reactor通信调度层，主要职责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。将这些事件触发到PipeLine中，由PipeLine管理的职责链后续进行处理。其由一系列辅助类完成，包括Reactor线程NioEventLoop及其父类, NioSocketChannel

/NioServerSocketChannel及其父类、ByteBuffer以及其衍生出来的各种Buffer、Unsafe以及其衍生出来的各种内部类。

1. PipeLine，PipeLine是职责链 ChannelPipeLine，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，也可以拦截处理和向后/向前传播时间。不同应用的Handler节点的功能也不同，通常条件下往往会开发编解码Handler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，这样上层业务则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，实现层面的分层隔离
2. 业务逻辑编排层，一类是纯粹的业务逻辑编排，还有一类是其他的应用层协议插件，用户特定协议相关的会话和链路管理。
3. 分层设计，架构的不同层面，需要关系和处理的对象不同，通常情况下，对于业务开发者，只需要关心职责链的拦截和业务Handler的编排，因为应用层协议栈往往是开发一次，到处运行，实际上对于业务开发者来说，只需要关心服务层的业务逻辑开发即可。各种应用协议以插件形式提供，只有协议人员需要关协议插件，对于其他业务开发人员来说，只需要关心业务逻辑定制即可。这种分层的架构设计理念实现了NIO架构设计理念实现NIO框架各层之间的解耦，便于上层业务协议栈的开发和业务逻辑的定制。

## **3.3 Netty NIO使用示例**

1) 在pom中添加依赖

*<dependency>*

*<groupId>io.netty</groupId>*

*<artifactId>netty-all</artifactId>*

*<version>4.1.17.Final</version>*

*</dependency>*

1. 定义Handler，该协议定义接收到请求后将信息输出

*public class DiscardHandler extends ChannelInboundHandlerAdapter {*

*@Override*

*public void channelRead(ChannelHandlerContext ctx, Object msg) {*

*ByteBuf in = (ByteBuf) msg;*

*try {*

*while (in.isReadable()) {*

*System.out.println((char) in.readByte());*

*System.out.flush();*

*}*

*} finally {*

*ReferenceCountUtil.release(msg);*

*}*

*}*

*@Override*

*public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause)*

*throws Exception {*

*cause.printStackTrace();*

*ctx.close();*

*}*

*}*

Handler继承ChannelInboundHandlerAdapter类，从Channel中读取信息并输出。

1. 定义NettyServer

*public class DiscardServer {*

*private int port;*

*public DiscardServer(int port) {*

*this.port = port;*

*}*

*public void run() throws Exception {*

*EventLoopGroup bossGroup = new NioEventLoopGroup(1);*

*EventLoopGroup workerGroup = new NioEventLoopGroup();*

*try {*

*ServerBootstrap b = new ServerBootstrap();*

*b.group(bossGroup, workerGroup)*

*.channel(NioServerSocketChannel.class)*

*.childHandler(new ChannelInitializer<SocketChannel>() {*

*public void initChannel(SocketChannel ch) throws Exception {*

*ch.pipeline().addLast(new DiscardHandler());*

*}*

*})*

*.option(ChannelOption.SO\_BACKLOG,128)*

*.childOption(ChannelOption.SO\_KEEPALIVE, true);*

*ChannelFuture f = b.bind(port).sync();*

*f.channel().closeFuture().sync();*

*} finally {*

*bossGroup.shutdownGracefully();*

*workerGroup.shutdownGracefully();*

*}*

*}*

*public static void main(String[] args) {*

*int port = 8088;*

*try {*

*new DiscardServer(port).run();*

*} catch (Exception ex) {*

*ex.printStackTrace();*

*}*

*}*

*}*

http://wiki.jikexueyuan.com/project/java-nio-zh/java-nio-selector.html

http://ifeve.com/java-io-network/

http://tutorials.jenkov.com/java-io/networking.html

http://www.runoob.com/java/java-networking.html