# Netty 学习笔记

https://netty.io/

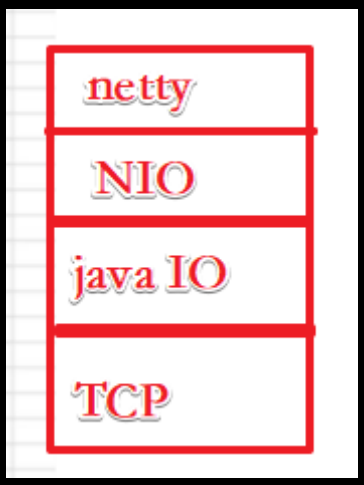
# 尚硅谷视频教程

https://www.bilibili.com/video/BV1DJ411m7NR

## Netty 介绍

### Netty 是什么

1. Netty 是 JBOSS 提供的 java 开源框架
2. Netty 是一个异步的、基于事件驱动的网络应用框架
3. Netty 主要针对在 TCP 协议下，面向 Clients 端的高并发应用，或者 P2P 场景下大量数据持续传输的应用
4. Netty 本质是一个 **NIO 框架**

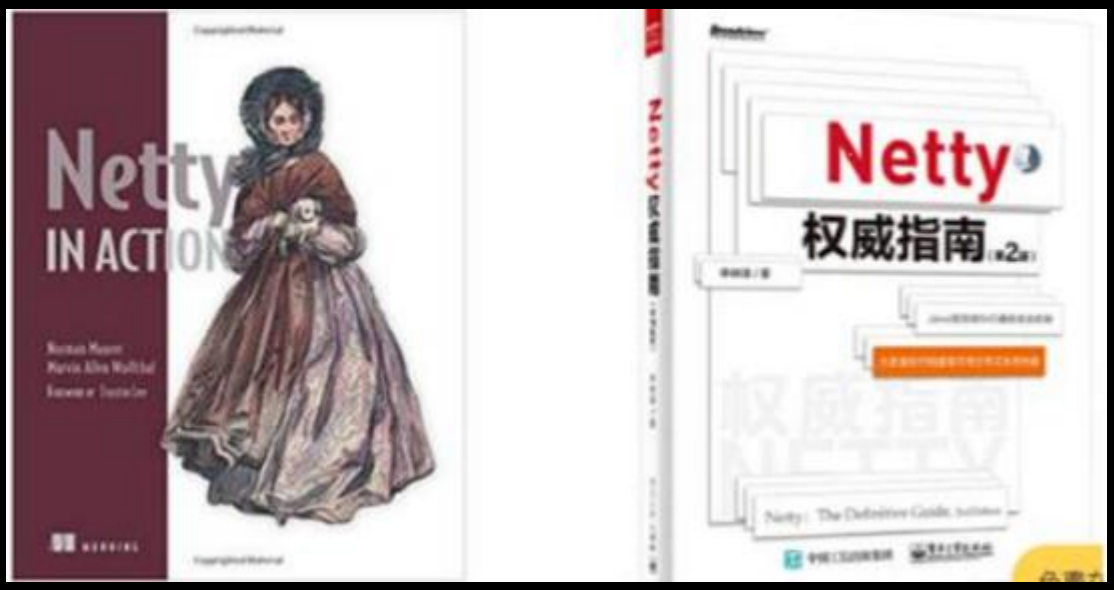



### 应用场景

1. 互联网行业：节点之间，RPC，如 dubbo
2. 游戏行业
3. 大数据领域：序列化组件 Avro…

Akka
Flink
Spark 等

学习资料：

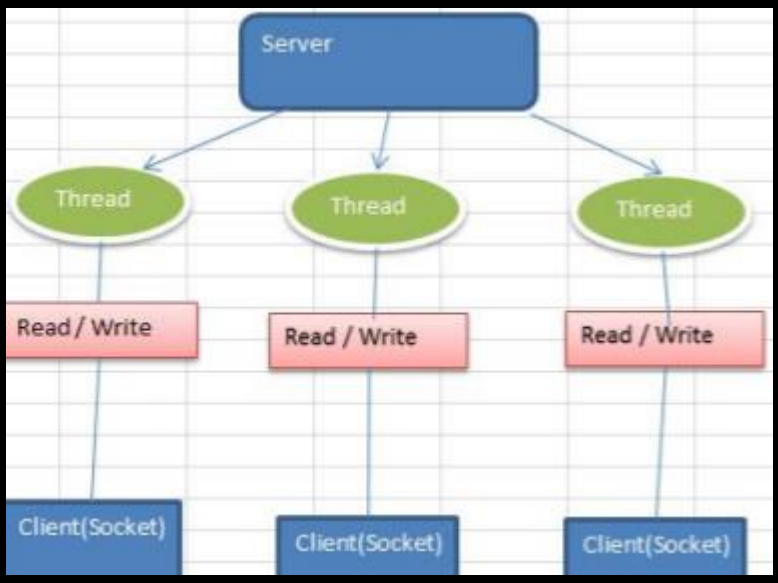



# IO 模型

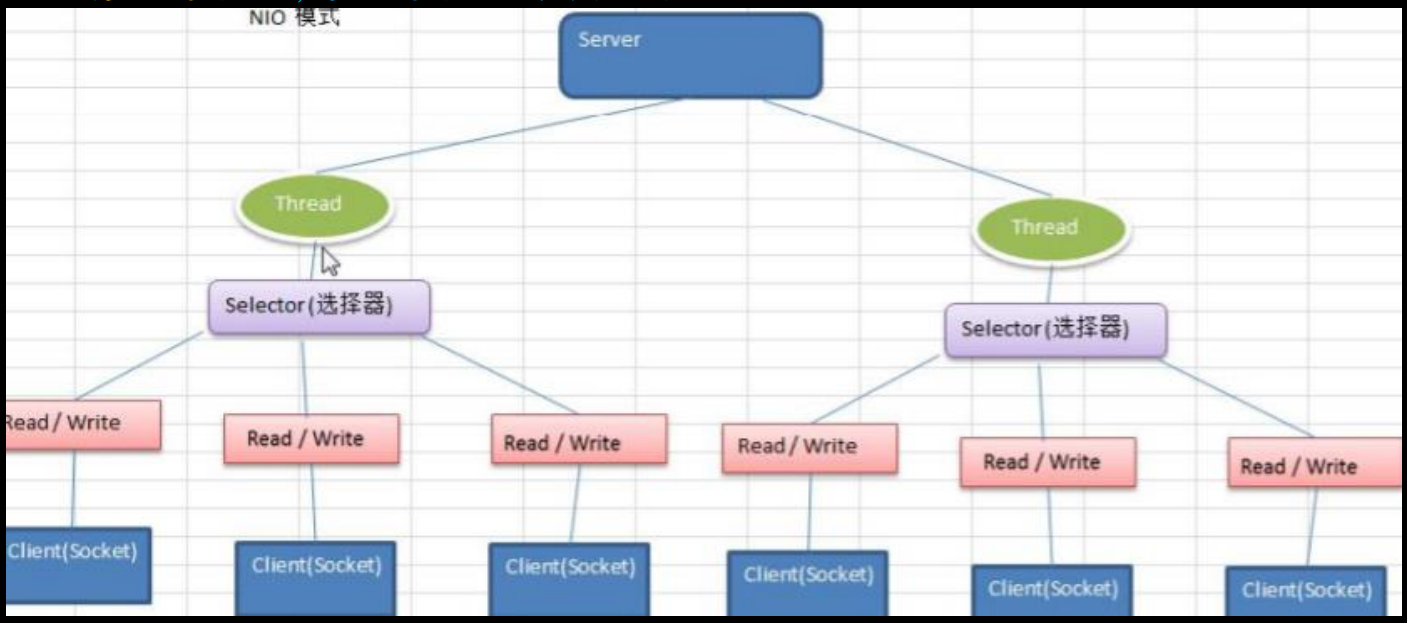
## Java 三种 IO 模型

IO 模型：用什么通道进行数据的发送和接收
Java 支持三种 IO 模型：
1. BIO：同步并阻塞
2. NIO：异步非阻塞
3. AIO：异步非阻塞

BIO：
同步并阻塞，传统阻塞型，服务器实现模式为：一个连接一个线程 ，客户端有连接请求时就服务器就启动一个线程进行处理，如果这个连接不做任何事情就会造成必须要的资源开销。




NIO：同步非阻塞，服务器实现模式为 一个线程处理多个请求/连接 ，即客户端发送的连接请求都会注册到多路复用器上，多路复用器轮询请求进行处理




Java AIO：NIO2， 异步非阻塞，引入异步通道的概念，采用 Proactor 模式，简化程序编写，有效的请求才会启动线程。使用较少。

适用场景：

1. BIO：连接少，固定的架构，jdk1.4 以前唯一的选择
2. NIO：连接多、短连接，如聊天服务器、弹幕系统，编程复杂，jdk1.4 开始
3. AIO：连接数目多、连接长，充分调用 OS 参与并发操作，jdk1.7 开始

## BIO

传统的 java io 编程，相关类接口在 java.io 中

尽管 BIO 对一个请求使用一个线程，但是仍然可以通过 线程池 提升性能。

BIO 编程：

1. 服务器启动 ServerSocket
2. 客户端启动 Socket 对服务器进行通信，默认情况下服务器需要对每个客户建立一个线程与之通信
3. 客户端发送请求后，先咨询服务器是否有线程响应，如果没有则会等待，或者被拒绝
4. 如果有相应，客户端线程会等待请求结束或再继续执行

启动服务端后可以使用 **telnet** 连接：

**telnet ip port**

如 telnet 127.0.0.1 6000 (注意没使用:)

使用 send 命令可以发送消息

```
public class BIOServer {
    /**
     * 这些等待的地方就是:  阻塞
     * */
    public static void main(String[] args) throws Exception {
        ExecutorService threadPool = Executors.newCachedThreadPool();
        final ServerSocket serverSocket = new ServerSocket(6666);
        System.out.println("服务器启动了...");
        while (true) {
            System.out.println("服务器等待连接...");
            final Socket accept = serverSocket.accept();
            threadPool.execute(new Runnable() {
                public void run() {
                    try {
                        System.out.println("服务器连接到了一个客户端，处理的线程:
"+Thread.currentThread().getName());
                        handleSocket(accept);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            });
        }

    }

    public static void handleSocket(Socket socket) throws IOException {
        InputStream inputStream = socket.getInputStream();
        byte[] bytes = new byte[1024];
        while (true) {
            System.out.println( Thread.currentThread().getName()+" waiting to read...");
            int read = inputStream.read(bytes);
            if (read==-1){
                break;
            }
            System.out.println(Thread.currentThread().getName()+" read: "+ new
String(bytes, 0 , read));
        }
    }
}
```

# NIO

Java nio: java non-blocking IO，是指 JDK 提供的新 API，从 JDK1.4 开始，java 提供了一系列改进输入/输出的新特性，被统称为 NIO(new IO)，同步非阻塞的
NIO 相关的类都是在 java.nio 包下
NIO 有三大核心：

1. **Channel：通道**
2. **Buffer：缓冲区**
3. **Selector：选择器**

NIO 是面向缓冲区的/面向块的，数据读取到一个它稍后处理的缓冲区，需要时可以在缓冲区中前后移动，增加处理灵活性，非阻塞式高伸缩性

NIO 中，一个线程从某个通道发送或者读取数据，它仅能得到目前可用的数据，如果没有数据可用，就不会做什么，所有通道都空闲时可以做自己的事情。

通俗理解：NIO 是可以做到用一个线程来处理多个操作的。假设有 10000 个请求过来,根据实际情况，可以分配 50 或者 100 个线程来处理。不像之前的阻塞 IO 那样，非得分配 10000 个。

**HTTP2.0 使用了多路复用的技术**，做到同一个连接并发处理多个请求，而且并发请求的数量比 HTTP1.1 大了好几个数量级

buffer 案例

```
public class IntBufferDemo {
    public static void main(String[] args) {
        IntBuffer intBuffer = IntBuffer.allocate(5);
        for (int i = 0; i <intBuffer.capacity() ; i++) {
            intBuffer.put(i);
        }

        // 读取之前，把 buffer 的当前指针指向 0
        // 用于读与写的切换
        intBuffer.flip();

        while (intBuffer.hasRemaining()){
            System.out.println(intBuffer.get());
        }
    }
}
```

# BIO vs. NIO

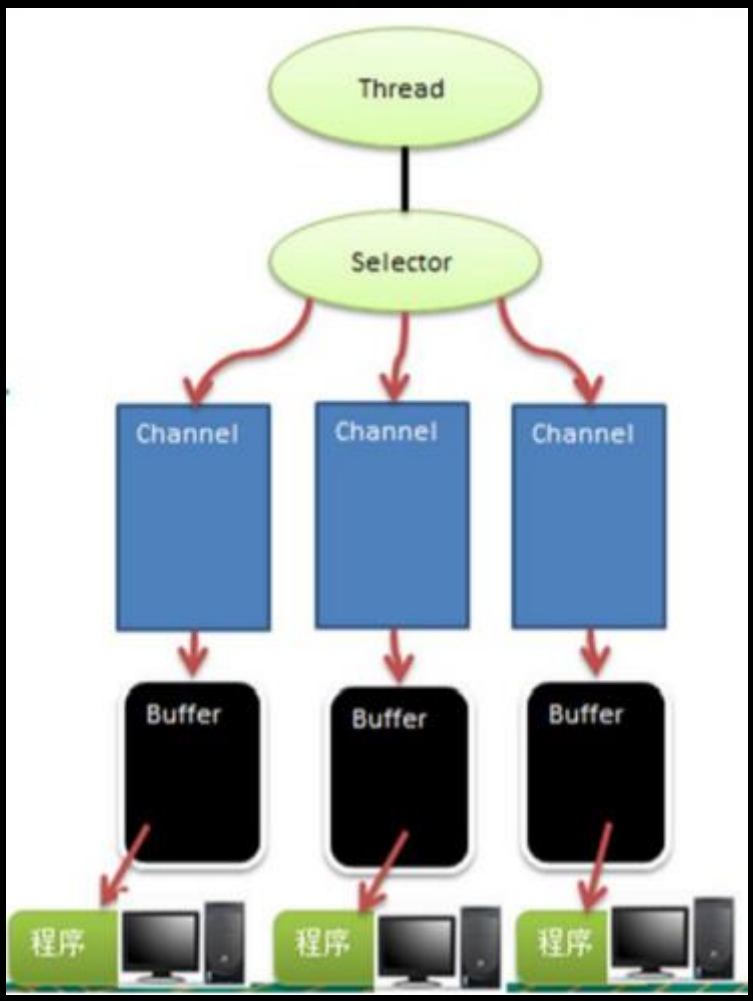
1. BIO 以流的方式处理，NIO 以块的方式处理，块 io 效率高很多
2. BIO 阻塞 NIO 非阻塞
3. BIO 基于字节流和字符流进行操作，而 NIO 基于 channel 和 buffer 进行操作，数据在二者之间流动。Selector 用于监听多个通道的事件，使用单个线程就可以监听多个客户端。

| | BIO | NIO | AIO |
| --- | --- | --- | --- |
| IO 模型 | 同步阻塞 | 同步非阻塞（多路复用） | 异步非阻塞 |
| 编程难度 | 简单 | 复杂 | 复杂 |
| 可靠性 | 差 | 好 | 好 |
| 吞吐量 | 低 | 高 | 高 |

**举例说明**

1) **同步阻塞**：到理发店理发，就一直等理发师，直到轮到自己理发。
2) **同步非阻塞**：到理发店理发，发现前面有其它人理发，给理发师说下，先干其他事情，一会过来看是否轮到自己.
3) **异步非阻塞**：给理发师打电话，让理发师上门服务，自己干其它事情，理发师自己来家给你理发

## NIO 三大组件

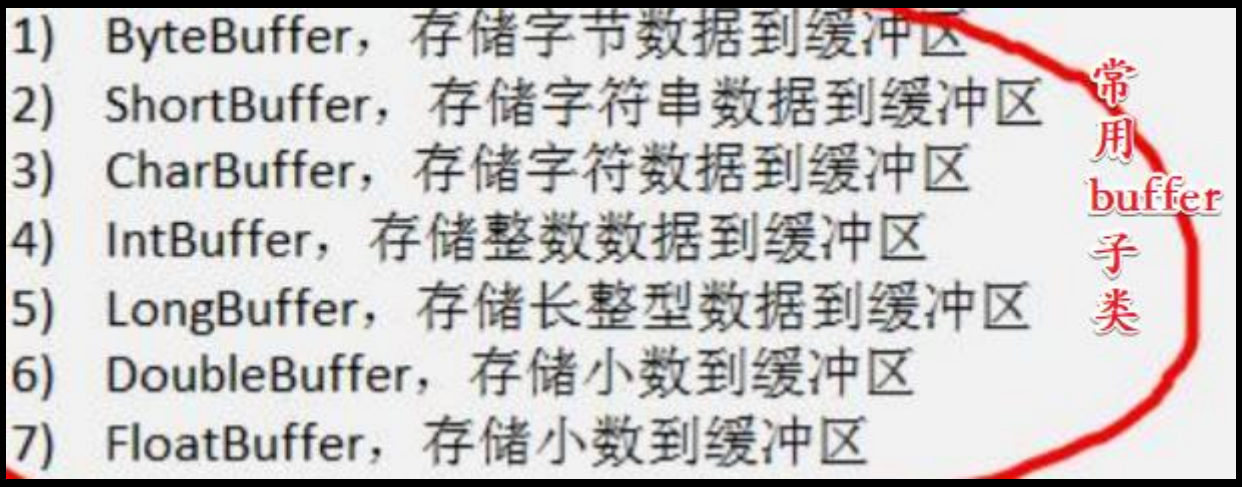



1. 一个 channel 对应一个 buffer

2. Selector 对应一个线程，一个线程对应多个 channel
3. 多个 channel 注册到 selector
4. 程序切换到那个 channel，是由事件 Event 决定的，select 会根据不同的事件，在各个通道上切换
5. Buffer 本质是一个内存块，底层是一个数组
6. 数据读写通过 buffer，BIO 中分为读写流，不是双向的。NIO 的 buffer 可读可写，需要使用 flip 方法切换，channel 是双向的，返回底层操作系统的执行情况，底层就是双向的。

## Buffer

缓冲区 Buffer：本质是一个可读可写的内存块，数组容器对象。
Channel 提供从文件、网络读取数据的渠道，但是读取或者写入的数据都必须经由 buffer：


Buffer 类定义了所有的缓冲区都具有的属性：
1. Capacity：容量
2. Limit：缓冲区的当前终点，不能对超过 limit 的位置进行读写，limit 可以变更
3. Position：下一个要被操作的位置
4. Mark：标记

```
public abstract class Buffer {
  //JDK1.4时，引入的api
  public final int capacity( )//返回此缓冲区的容量
  public final int position( )//返回此缓冲区的位置
  public final Buffer position (int newPositio)//设置此缓冲区的位置
  public final int limit( )//返回此缓冲区的限制
  public final Buffer limit (int newLimit)//设置此缓冲区的限制
  public final Buffer mark( )//在此缓冲区的位置设置标记
  public final Buffer reset( )//将此缓冲区的位置重置为以前标记的位置
  public final Buffer clear( )//清除此缓冲区,即将各个标记恢复到初始状态，但是数据并没有真正擦除
  public final Buffer flip( )//反转此缓冲区
  public final Buffer rewind( )//重绕此缓冲区
  public final int remaining( )//返回当前位置与限制之间的元素数
  public final boolean hasRemaining( )//告知在当前位置和限制之间是否有元素
  public abstract boolean isReadOnly( );//告知此缓冲区是否为只读缓冲区

  //JDK1.6时引入的api
  public abstract boolean hasArray();//告知此缓冲区是否具有可访问的底层实现数组
  public abstract Object array();//返回此缓冲区的底层实现数组
  public abstract int arrayOffset();//返回此缓冲区的底层实现数组中第一个缓冲区元素的偏移量
  public abstract boolean isDirect();//告知此缓冲区是否为直接缓冲区
```

Buffer 类相关方法一览

Flip: 翻转

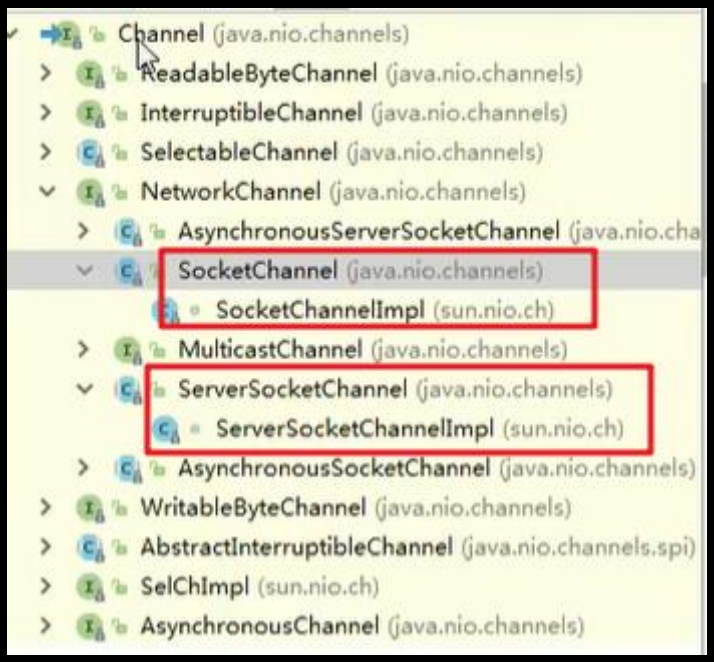
```
    public final Buffer flip() {
        limit = position;
        position = 0;
        mark = -1;
        return this;
    }
```

除了 bool 之外，都有一个 buffer 和基本类型对应
ByteBuffer 最常用

## Channel

NIO 的通道类似与流，但是有些区别：
1. 通道可以同时进行读写，流只能读或者写
2. 通道可以实现异步读写
3. 通道可以从缓冲读数据，也可以写到数据到缓冲

Channel 是 nio 中的一个接口，在 java.nio.channels 包下




常用的有

1. **FileChannel: 文件读写**
2. DatagramChannel: UDP 数据读写
3. ServerSocketChannel: TCP 数据读写
4. Socket: TCP 数据读写

FileChannel 主要用来对本地文件进行 IO 操作，常见的方法有

```
public int read(ByteBuffer dst) ，从通道读取数据并放到缓冲区中
public int write(ByteBuffer src) ，把缓冲区的数据写到通道中
public long transferFrom(ReadableByteChannel src, long position, long count)，从目标通道中复制数据到当前通道
public long transferTo(long position, long count, WritableByteChannel target)，把数据从当前通道复制给目标通道
```

案例：


```
    /**
     * 通过 FileChannel 将字符串写入文件
     * */
    public static void main(String[] args) throws IOException {
        String str = "Hi, 许涛!";

        // 打开一个文件输出流(需要从内存输出到文件，故是输出流)
        FileOutputStream fileOutputStream = new FileOutputStream("fileChannel.txt");


        // 获取 FileChannel，实现为 FileChannelImpl
```

```
        FileChannel channel = fileOutputStream.getChannel();

        // 声明 ByteBuffer，并将字符串加入其中
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        buffer.put(str.getBytes());

        // 读写反转
        buffer.flip();

        // 将 buffer 中的数据写入 channel
        channel.write(buffer);

        // 关闭输出流
        fileOutputStream.close();
    }
```

```
    /**
     * 从文件写入 buffer
     * */
    public static void main(String[] args) throws IOException {
        // 输入流
        FileInputStream inputStream = new FileInputStream("fileChannel.txt");

        // 获取 channel
        FileChannel channel = inputStream.getChannel();

        // 开一个 buffer
        ByteBuffer buffer = ByteBuffer.allocate(1024);

        // 读取到 buffer
        channel.read(buffer);

        // 反转
        buffer.flip();

        // 创建一个字节数组用于存储，注意大小是 buffer 的 limit 或者在 get 的时候需要指定大小为 limit
        // 或者不使用 bytes，直接使用 buffer.array();
        byte[] bytes = new byte[1024];
        buffer.get(bytes, 0, buffer.limit());

        System.out.println(new String(bytes));
    }
```

```
    /**
     * 使用 FileChannel 拷贝文件
     * */
    public static void main(String[] args) throws IOException {
        // 创建输入流
        File file = new File("fileChannel.txt");
        FileInputStream fileInputStream = new FileInputStream(file);
        // 获取 channel
        FileChannel inputStreamChannel = fileInputStream.getChannel();

        // 创建输出流
        FileOutputStream fileOutputStream = new FileOutputStream("fileChannelCopy.txt");
        // 获取 channel
        FileChannel outputStreamChannel = fileOutputStream.getChannel();


        // 开启一个 buffer
        ByteBuffer byteBuffer = ByteBuffer.allocate((int) file.length());

        // 循环写入 buffer
        while (true){
            // !重要：  读过一次之后，必须要将 position、limit 等置为 0
            byteBuffer.clear();
```

```
            int read = inputStreamChannel.read(byteBuffer);
            if (read == -1){
                break;
            }

            // 反转
            byteBuffer.flip();
            // 将 buffer 中内容写入 channel(当全部写入 Channel 后[或者缓冲区慢？]才会真正写入文件)
            outputStreamChannel.write(byteBuffer);
        }
        fileInputStream.close();
        fileOutputStream.close();
    }
```

拷贝还可以使用 channel 的 **transferFrom/transferTo** ,更加直接

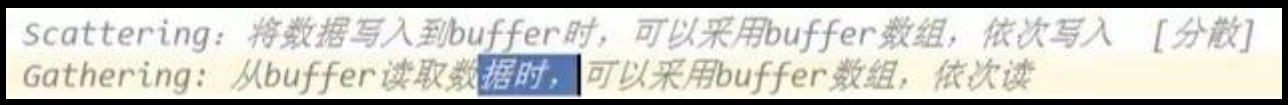
ByteBuffer 支持类型化的 put 和 get, put 放入的是什么数据类型, get 就应该使用相应的数据类型来取出, 否则可能有 BufferUnderflowException 异常

可以将一个普通 Buffer 转成只读 Buffer

NIO 还提供了 MappedByteBuffer, 可以让**文件直接在内存（堆外的内存）中进行修改**, 而如何同步到文件由 NIO 来完成

```
    public static void main(String[] args) throws Exception {
        RandomAccessFile randomAccessFile = new RandomAccessFile("1.txt", "rw");
        //获取对应的通道
        FileChannel channel = randomAccessFile.getChannel();
        /**
         * 参数 1: FileChannel.MapMode.READ_WRITE 使用的读写模式
         * 参数 2:  0 :  可以直接修改的起始位置
         * 参数 3: 5: 是映射到内存的大小(不是索引位置) ,即将 1.txt 的多少个字节映射到内存
         * 可以直接修改的范围就是 0-5
         * 实际类型 DirectByteBuffer
         */
        MappedByteBuffer mappedByteBuffer = channel.map(FileChannel.MapMode.READ_WRITE, 0,
5);
        mappedByteBuffer.put(0, (byte) 'H');
        mappedByteBuffer.put(3, (byte) '9');
        mappedByteBuffer.put(5, (byte) 'Y');//IndexOutOfBoundsException
        randomAccessFile.close();
        System.out.println("修改成功~~");
    }
```

前面我们讲的读写操作,都是通过一个 Buffer 完成的,NIO 还支持 通过多个 Buffer (即 **Buffer 数组**) 完成读写操作, 即 **Scattering 和 Gathering**

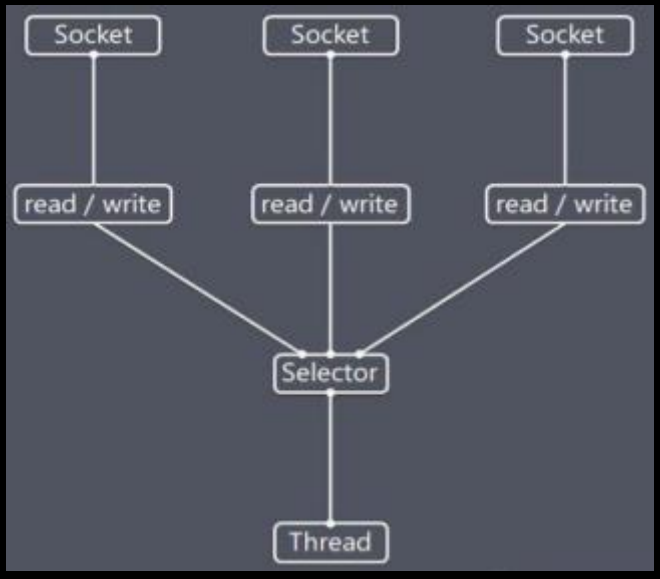



## Selector

Selector 能够检测多个注册的通道上是否有事件发生, 如果有, 则获取事件然后针对每个事件进行

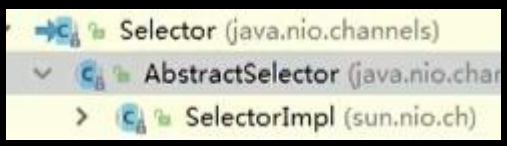
相应的处理，于是可以一个线程管理多个通道，即管理多个连接和请求。

只有在 连接/通道 真正有读写事件发生时，才会进行读写，大大减少系统开销
避免了多线程之间的线程切换开销




1. Netty 的 IO 线程 NIOEventLoop 聚合了 selector(多路复用器)，可以同时处理成百上千个客户端连接。
2. 没有需要读写的 channel 时，线程可以做其他事
3. 读写都是非阻塞的，效率高
4. 一个 IO 线程可以并发处理 N 个客户端连接，根本上解决 BIO 一个线程一个连接的问题。

Selector 类与方法:




Selector 是一个抽象类

```
public abstract class Selector implements Closeable {
public static Selector open();//得到一个选择器对象
public int select(long timeout);//监控所有注册的通道，当其中有 IO 操作可以进行时，将
对应的 SelectionKey 加入到内部集合中并返回，参数用来设置超时时间
public Set<SelectionKey> selectedKeys();//从内部集合中得到所有的 SelectionKey
```

NIO 中的 ServerSocketChannel 功能类似 ServerSocket，SocketChannel 功能类似 Socket

1. selector.select()//阻塞

2. selector.select(1000);//阻塞 1000 毫秒，在 1000 毫秒后返回
3. selector.wakeup();//唤醒 selector
4. selector.selectNow();//不阻塞，立马返还

## SelectionKey

Selector 和网络通道的注册关系

1. int OP_ACCEPT：有新的网络连接可以 accept，值为 16
2. int OP_CONNECT：代表连接已经建立，值为 8
3. int OP_READ：代表读操作，值为 1
4. int OP_WRITE：代表写操作，值为 4

```
public abstract class SelectionKey {
    public abstract Selector selector();//得到与之关联的
Selector 对象
    public abstract SelectableChannel channel();//得到与之关
联的通道
    public final Object attachment();//得到与之关联的共享数
据
    public abstract SelectionKey interestOps(int ops);//设置或
改变监听事件
    public final boolean isAcceptable();//是否可以 accept
    public final boolean isReadable();//是否可以读
    public final boolean isWritable();//是否可以写
}
```

## ServerSocketChannel SocketChannel

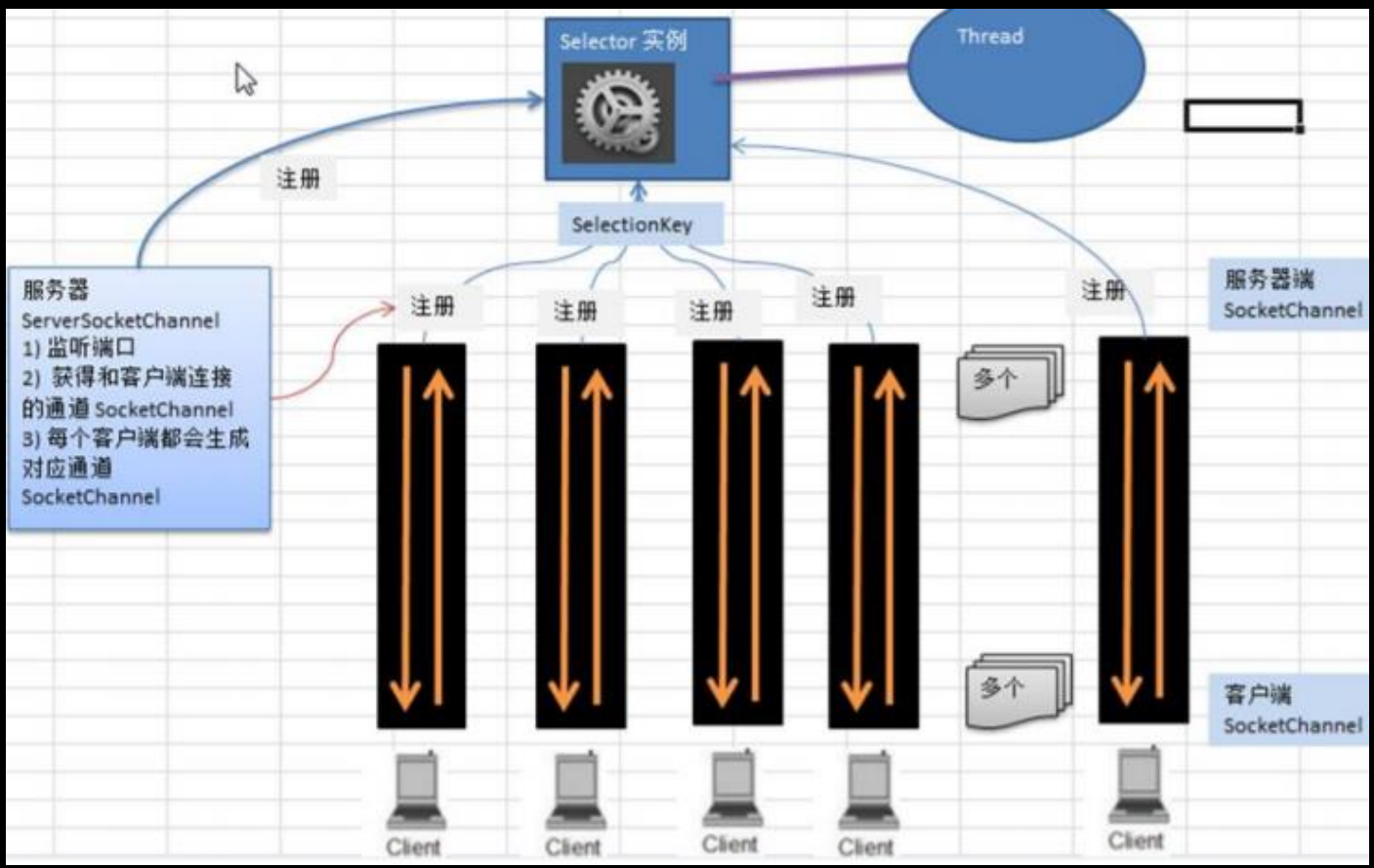
ServerSocketChannel 在服务器端监听新的客户端 Socket 连接

```
public abstract class ServerSocketChannel
    extends AbstractSelectableChannel
    implements NetworkChannel{
public static ServerSocketChannel open()，得到一个 ServerSocketChannel 通道
public final ServerSocketChannel bind(SocketAddress local)，设置服务器端端口
号
public final SelectableChannel configureBlocking(boolean block)，设置阻塞或非
阻塞模式，取值 false 表示采用非阻塞模式
public SocketChannel accept()，接受一个连接，返回代表这个连接的通道对
象
public final SelectionKey register(Selector sel, int ops)，注册一个选择器并设置
监听事件
}
```

SocketChannel，网络 IO 通道，**具体负责进行读写操作**。NIO 把缓冲区的数据写入通道，或者把通道里的数据读到缓冲区

```
public abstract class SocketChannel
    extends AbstractSelectableChannel
    implements ByteChannel, ScatteringByteChannel, GatheringByteChannel,
NetworkChannel{
public static SocketChannel open();//得到一个 SocketChannel 通道
public final SelectableChannel configureBlocking(boolean block);//设置阻塞或非阻塞
模式，取值 false 表示采用非阻塞模式
public boolean connect(SocketAddress remote);//连接服务器
public boolean finishConnect();//如果上面的方法连接失败，接下来就要通过该方法
完成连接操作
public int write(ByteBuffer src);//往通道里写数据
public int read(ByteBuffer dst);//从通道里读数据
public final SelectionKey register(Selector sel, int ops, Object att);//注册一个选择器并
设置监听事件，最后一个参数可以设置共享数据
public final void close();//关闭通道
```

## NIO 非阻塞原理




1. 当客户端连接时，通过 ServerSocketChannel 得到 SocketChannel
2. Selector 进行监听，select 方法返回有事件发生的通道个数

3. 将 socketChannel 注册到 selector 上，register(Selector sel, int ops)
4. 注册后返回一个 SelectionKey，和该 selector 关联(加入 selector 的集合)
5. 进一步得到有事件发生的各个 SelectionKey
6. 通过 selectionKey 反向获取 SocketChannel
7. 通过 socketchannel 完成业务

## NIO 案例

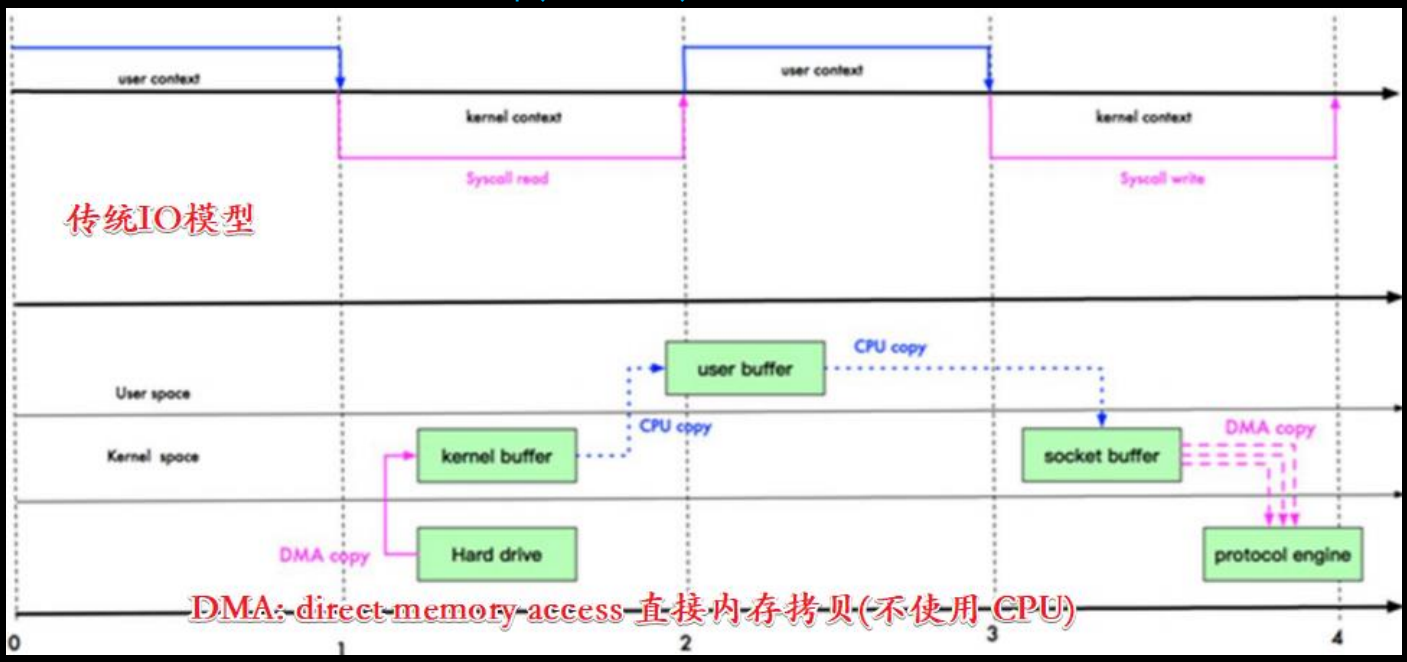
```
public class NIOServer {
    public static void main(String[] args) throws IOException {
        // 创建一个 Selector
        Selector selector = Selector.open();
        //创建 ServerSocketChannel
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        // 绑定端口，监 b 听
        serverSocketChannel.socket().bind(new InetSocketAddress(6666));

        // 设置非阻塞
        serverSocketChannel.configureBlocking(false);

        // 把 serverSocketChannel 注册到 selector， 操作类型为 OP_ACCEPT
        serverSocketChannel.register(selector, SelectionKey.OP_ACCEPT);

        while (true) {
            if (selector.select(3000) == 0) {
                System.out.println("等待了 3 秒，没有事件发生!");
                continue;
            }
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                // 是通道的连接事件
                if (key.isAcceptable()) {
                    // key 是 OP_ACCEPT 类型，表示有一个客户端在请求连接，可以获取连接
                    // accept 本身是阻塞的，当这个地方并不会阻塞，因为已经知道了有客户端连接，不会
等待
                    // 我有问题：如果获取了一个 key，但是当 accept 的时候如果客户端取消请求，不就是
获取到 null 了吗？
                    SocketChannel socketChannel = serverSocketChannel.accept();
                    // 必须要设置为非阻塞
                    socketChannel.configureBlocking(false);
                    // 也需要注册到 selector，关注事件为 OP_READ，并且关联一个 buffer
                    socketChannel.register(selector, SelectionKey.OP_READ,
ByteBuffer.allocate(1024));
                }
                // 是读事件
                if (key.isReadable()) {
                    // 根据 key 获取 channel
                    SocketChannel channel = (SocketChannel) key.channel();
                    // 获取关联的 buffer
                    ByteBuffer buffer = (ByteBuffer) key.attachment();
                    // 从 channel 中获取数据，存储到 buffer
                    channel.read(buffer);
                    System.out.println("read: "+ new String(buffer.array()));
                }
                // 需要手动把该 key 删除，多线程下防止产生重复？？？
                iterator.remove();
            }
        }
    }
}
```

```
public class NIOClient {
    public static void main(String[] args) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.configureBlocking(false);
        SocketAddress socketAddress = new InetSocketAddress("127.0.0.1", 6666);
        if (!socketChannel.connect(socketAddress)){
            while (!socketChannel.finishConnect()){
                System.out.println("连接没有完成, 做其他事情中~");
            }
        }
        // 连接完成
        String msg = "hello, Edwin Xu!";
        ByteBuffer buffer = ByteBuffer.wrap(msg.getBytes());
        socketChannel.write(buffer);
        System.in.read();
    }
}
```

## NIO 与零拷贝

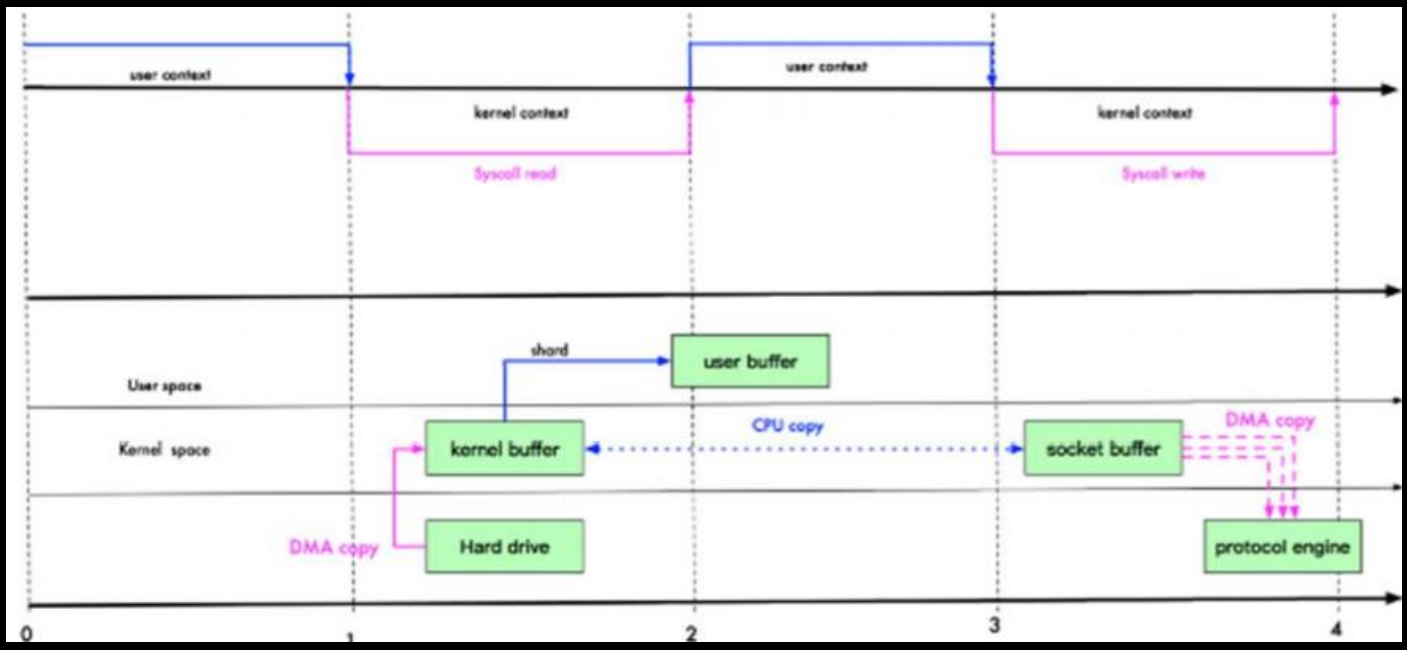
零拷贝用于 IO 性能优化

Java 程序中, 常用的零拷贝有 mmap(内存映射) 和 sendFile。

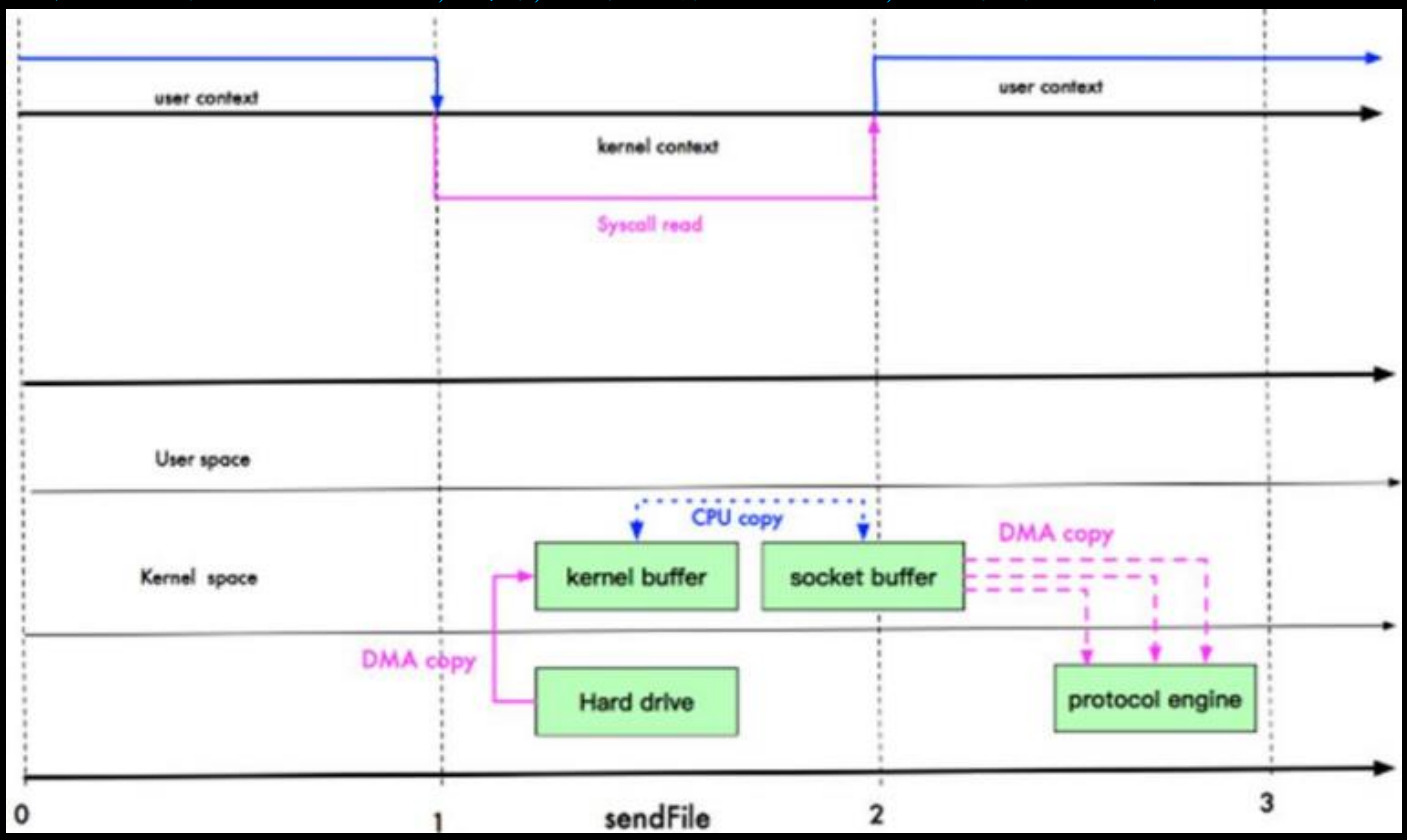



mmap 优化:

mmap 通过内存映射, 将**文件映射到内核缓冲区**, 同时, 用户空间可以共享内核空间的数据。这样, 在进行网络传输时, 就可以减少内核空间到用户空间的拷贝次数




sendFile 优化:
Linux 2.1 版本 提供了 sendFile 函数，其基本原理如下：数据根本不经过用户态，直接从内核缓冲区进入到 Socket Buffer，同时，由于和用户态完全无关，就减少了一次上下文切换




Linux 在 2.4 版本中，做了一些修改，避免了从内核缓冲区拷贝到 Socket buffer 的操作，直接拷贝到协议栈，从而再一次减少了数据拷贝

零拷贝的再次理解
零拷贝，是从操作系统的角度来说的。因为内核缓冲区之间，没有数据是重复的
零拷贝不仅仅带来更少的数据复制，还能带来其他的性能优势

mmap 和 sendFile 的区别:
1. mmap 适合小数据量读写，sendFile 适合大文件传输
2. mmap 需要 4 次上下文切换，3 次数据拷贝；sendFile 需要 3 次上下文切换，最少 2 次数据拷贝。
3. sendFile 可以利用 DMA 方式，减少 CPU 拷贝，mmap 则不能（必须从内核拷贝到 Socket 缓冲区）

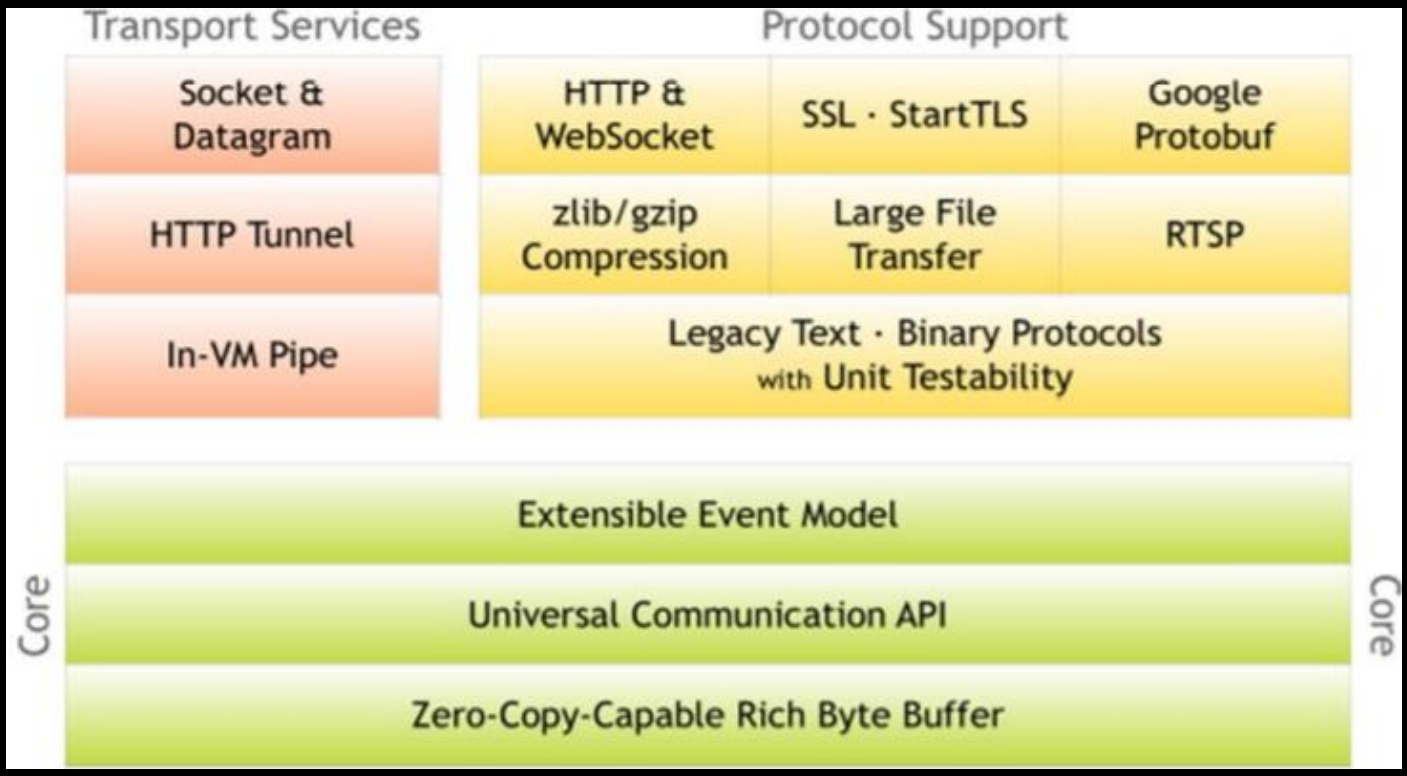
NIO 零拷贝案例:
1) 使用传统的 IO 方法传递一个大文件?
2) 使用 NIO 零拷贝方式传递(transferTo)一个大文件?
3) 看看两种传递方式耗时时间分别是多少?

**零拷贝： fileChannel.transferTo();**

## Netty 概述

原生 NIO 的问题:
1. 类库和 api 复杂，使用麻烦
2. 需要扎实的编程技能：java 多线程、reactor 模式、网络编程
3. 开发工作量、难度大：断连重连、网络闪断、半包读写、失败缓存、网络拥塞等
4. JDK NIO 的 bug：如 epool bug，导致 selector 空轮询，导致 CPU100%

官网介绍:
Netty is an asynchronous event-driven network application framework
for rapid development of maintainable high performance protocol servers & clients




Netty 的优点:
对 JDK 自带的 NIO 进行了封装。
1. 设计优雅:
   a) 支持阻塞和非阻塞
   b) 灵活可扩展的事件模型，分离关注点
   c) 可定制化的线程模型: 单线程、多线程、线程池
2. 使用方便
3. 高性能、高吞吐量；低延迟，低资源消耗；最小化不必要的内存复制
4. 安全: SSL/TLS 、startTLS
5. 社区活跃、不断更新

版本说明:
netty 版本分为 netty3.x 和 netty4.x、netty5.x
因为 Netty5 出现重大 bug，已经被官网废弃了，目前推荐使用的是 Netty4.x 的稳定版本

https://bintray.com/netty/downloads/netty/

# Netty 高性能架构设计

## 线程模型

不能的线程模型对程序的性能影响很大。

目前存在的线程模型：

1. 传统阻塞 IO 服务模型
2. Reactor 模式

   根据 Reactor 的数量和处理资源线程的数量不同，有三种典型的实现：

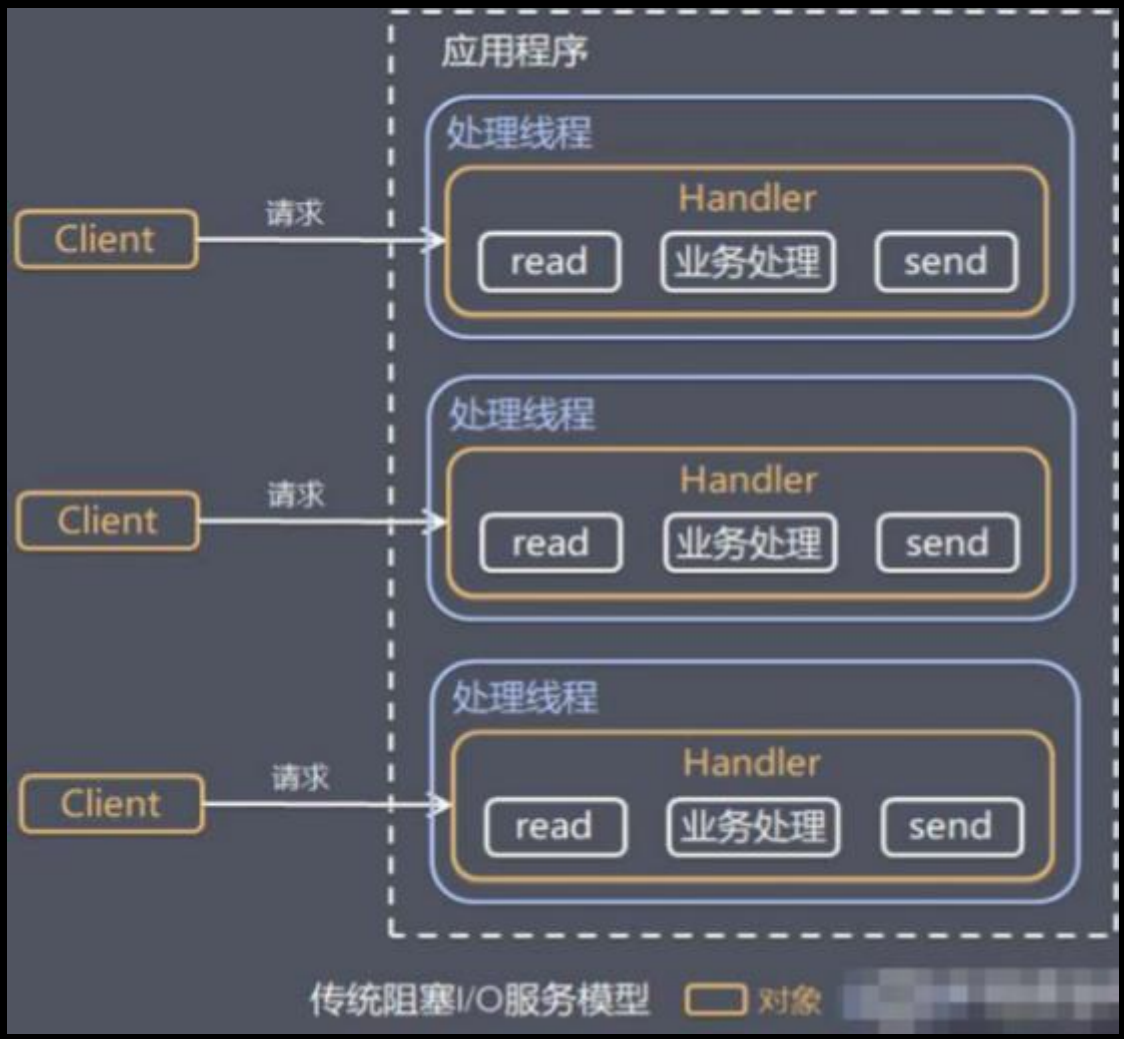
   a) 单 Reactor 单线程

   b) 单 Reactor 多线程

   c) 主从 Reactor 多线程

Netty 线程模式(**Netty 主要基于主从 Reactor 多线程模型**做了一定的改进，其中主从 Reactor 多线程模型有多个 Reactor)

## 传统阻塞 I/O 服务模型



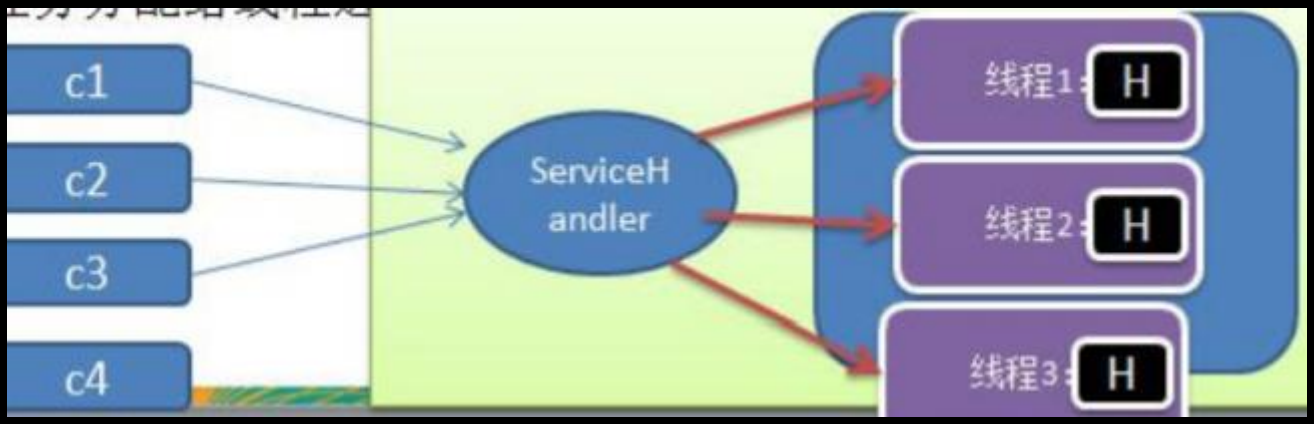

一个请求一个线程，

## Reactor 模型

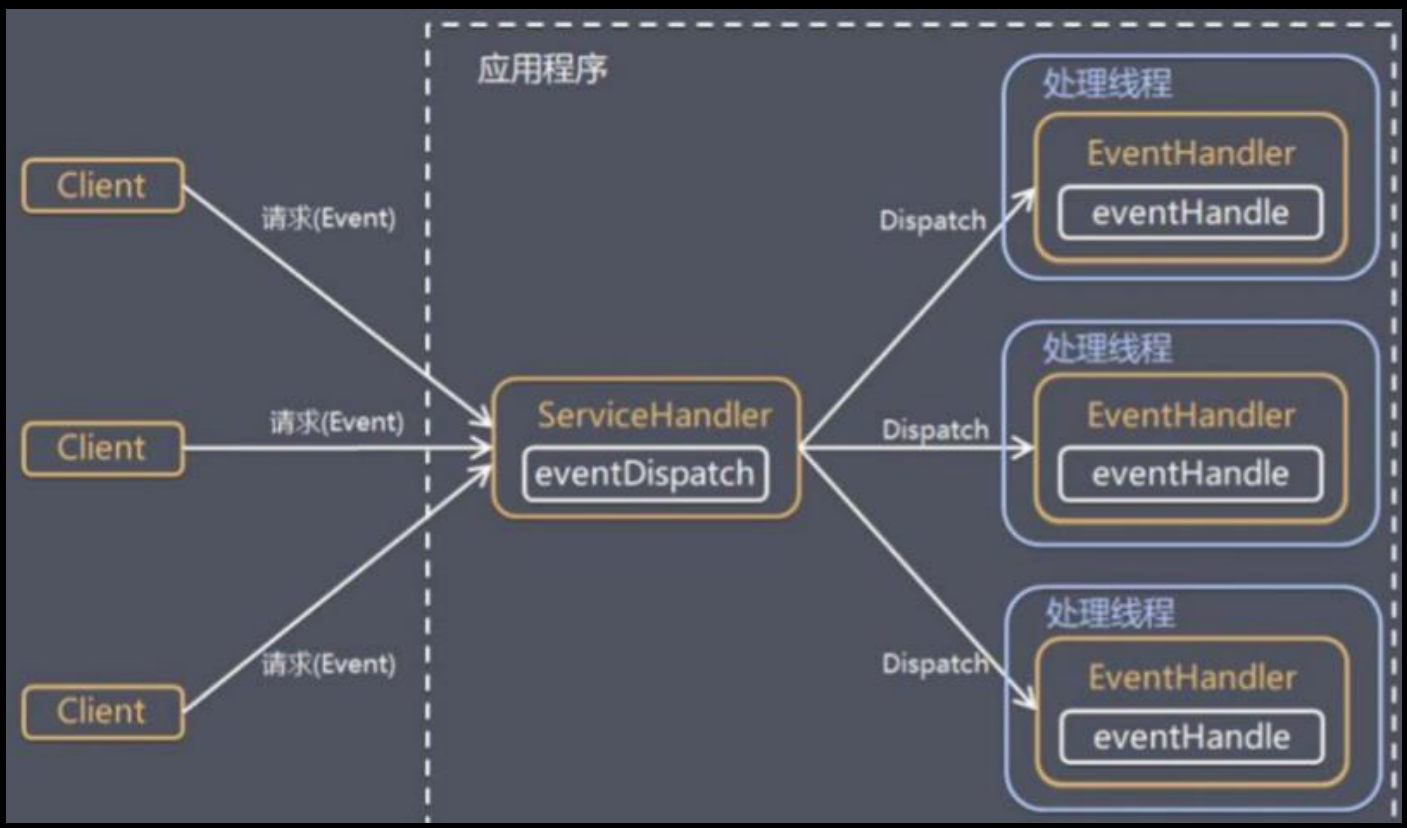
针对传统阻塞 I/O 服务模型的 2 个缺点，解决方案：

1. **基于 I/O 复用模型**：多个连接共用一个阻塞对象，应用程序只需要在一个阻塞对象等待，无需阻塞等待所有连接。当某个连接有新的数据可以处理时，操作系统通知应用程序，线程从阻塞状态返回，开始进行业务处理

2. 基于线程池复用线程资源：不必再为每个连接创建线程，将连接完成后的业务处理任务分配给线程进行处理，一个线程可以处理多个连接的业务




复用结合线程池，就是 Reactor 模式基本设计思想




1. Reactor 模式，通过一个或多个输入同时传递给服务处理器的模式(基于**事件驱动**)
2. 服务器端程序处理传入的多个请求,并将它们同步分派到相应的处理线程， 因此 Reactor 模式也叫 Dispatcher 模式
3. Reactor 模式使用 IO 复用监听事件，收到事件后，分发给某个线程(进程)，这点就是网络服务器高并发处理关键

Reactor 模式中核心组成：

1. Reactor：Reactor 在一个单独的线程中运行，负责**监听和分发事件**，分发给适当的处理程序来对 IO 事件做出反应。 它就像公司的电话接线员，它接听来自客户的电话并将线路转移到适当的联系人；

2. Handlers：处理程序执行 I/O 事件要完成的实际事件，类似于客户想要与之交谈的公司中的实际官员。Reactor 通过调度适当的处理程序来响应 I/O 事件，处理程序执行非阻塞操作

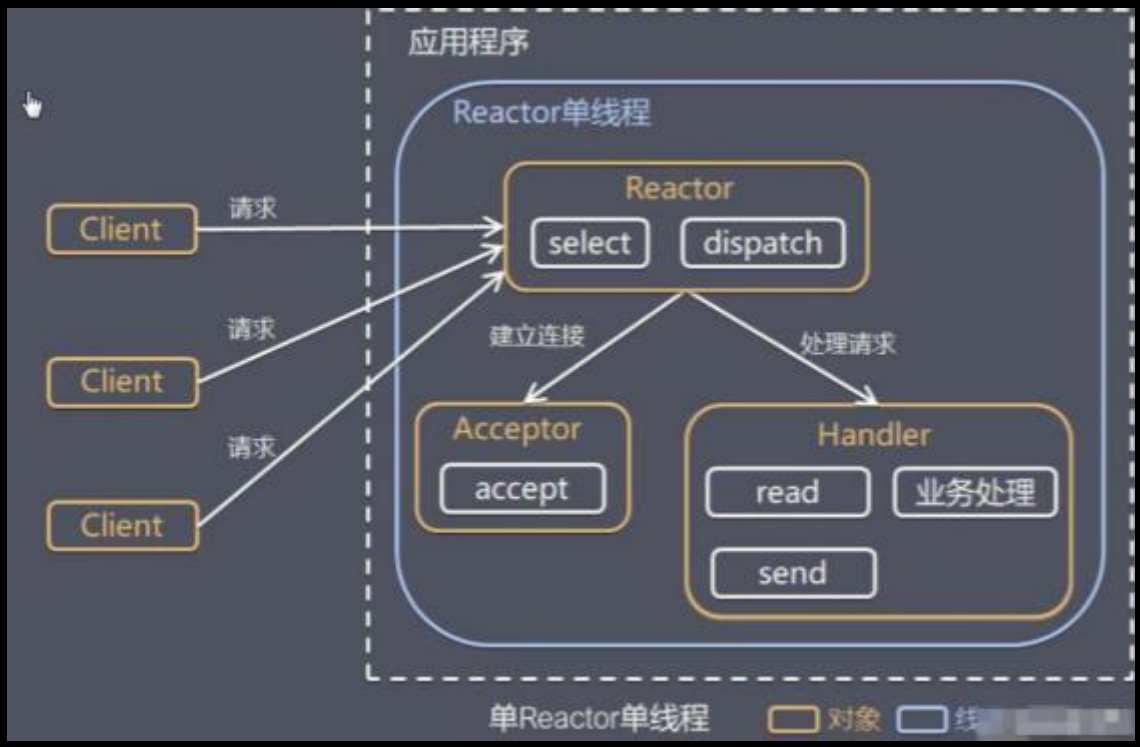
Reactor 对应的叫法：

- **反应器模式**
- **分发者模式(Dispatcher)**
- **通知者模式(notifier)**

三种 reactor：

1）单 Reactor 单线程
2）单 Reactor 多线程
3）主从 Reactor 多线程

## 单 Reactor 单线程

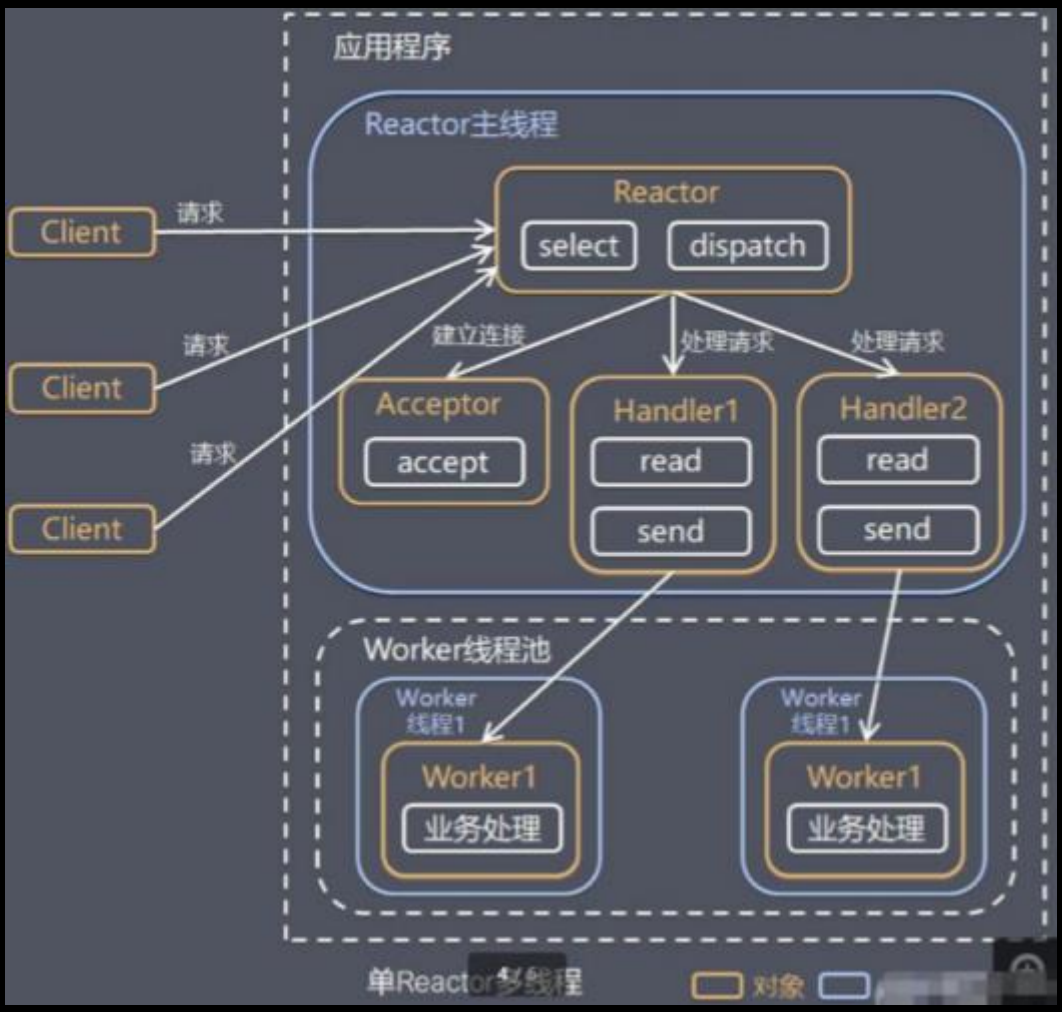



1）Select 是前面 I/O 复用模型介绍的标准网络编程 API，可以实现应用程序通过一个阻塞对象监听多路连接请求
2）Reactor 对象通过 Select 监控客户端请求事件，收到事件后通过 Dispatch 进行分发
3）如果是建立连接请求事件，则由 Acceptor 通过 Accept 处理连接请求，然后创建一个 Handler 对象处理连接完成后的后续业务处理
4）如果不是建立连接事件，则 Reactor 会分发调用连接对应的 Handler 来响应
5）Handler 会完成 Read→业务处理→Send 的完整业务流程

1）优点：模型简单，没有多线程、进程通信、竞争的问题，全部都在一个线程中完成
2）缺点：性能问题，只有一个线程，无法完全发挥多核 CPU 的性能。Handler 在处理某个连接上的业务时，整个进程无法处理其他连接事件，很容易导致性能瓶颈
3）缺点：可靠性问题，线程意外终止，或者进入死循环，会导致整个系统通信模块不可用，不能接

收和处理外部消息，造成节点故障
4）使用场景：客户端的数量有限，业务处理非常快速，比如 Redis 在业务处理的时间复杂度 O(1) 的情况

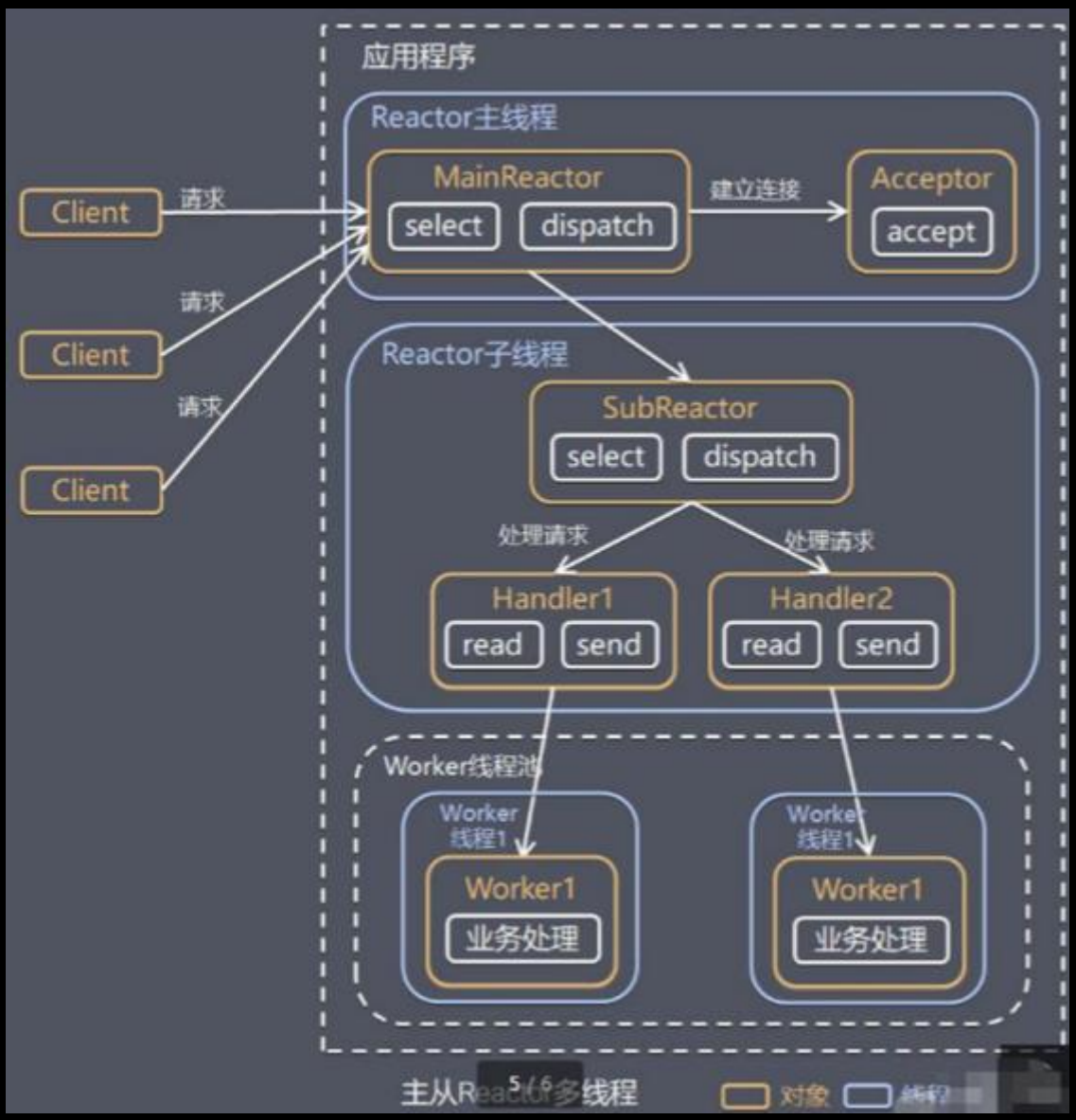
## 单 Reactor 多线程

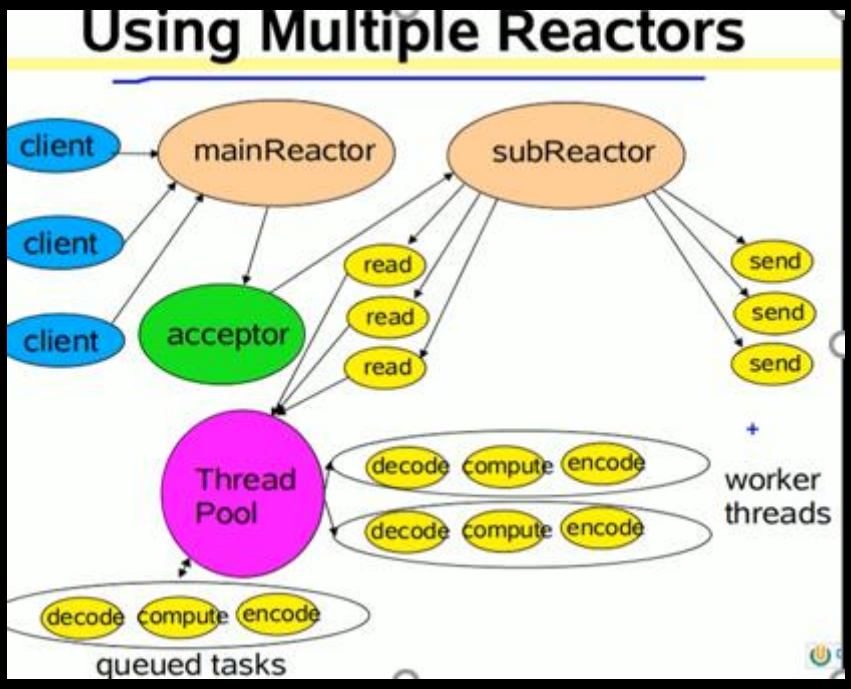



1. Reactor 对象通过 select 监控客户端请求事件，收到事件后，通过 dispatch 进行分发
2. 如果建立连接请求，则右 Acceptor 通过 accept 处理连接请求，然后创建一个 Handler 对象处理完成连接后的各种事件
3. 如果不是连接请求，则由 reactor 分发调用连接对应的 handler 来处理
4. handler 只负责响应事件，不做具体的业务处理，通过 read 读取数据后，会分发给后面的 worker 线程池的某个线程处理业务
5. worker 线程池会分配独立线程完成真正的业务，并将结果返回给 handler
6. handler 收到响应后，通过 send 将结果返回给 client

优点：可以充分的利用多核 cpu 的处理能力
缺点：多线程数据共享和访问比较复杂， reactor 处理所有的事件的监听和响应，在单线程运行，在高并发场景容易出现性能瓶颈.

## 主从 Reactor 多线程




《Scalable IO in Java》

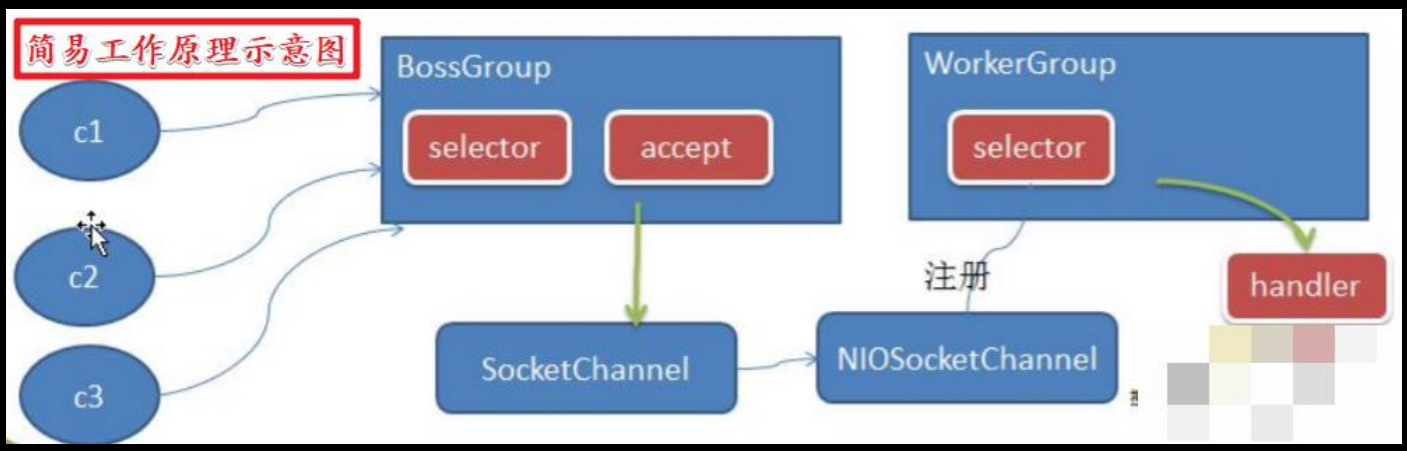

1) Reactor 主线程 MainReactor 对象通过 select 监听**连接事件**, 收到事件后，通过 Acceptor 处理连接事件
2) 当 Acceptor 处理连接事件后，MainReactor 将连接分配给 SubReactor
3) subreactor 将连接加入到连接队列进行监听,并创建 handler 进行各种事件处理
4) 当有新事件发生时， subreactor 就会调用对应的 handler 处理
5) handler 通过 read 读取数据，分发给后面的 worker 线程处理
6) worker 线程池分配独立的
7) handler 收到响应的结果后，再通过 send 将结果返回给 client
8) Reactor 主线程可以对应多个 Reactor 子线程，即 MainRecator 可以关联多个 SubReactor

1) 优点：父线程与子线程的数据交互简单职责明确，父线程只需要接收新连接，子线程完成后续的业务处理。
2) 优点：父线程与子线程的数据交互简单，Reactor 主线程只需要把新连接传给子线程，子线程无需返回数据。
3) 缺点：编程复杂度较高

举例说明：
1) 单 Reactor 单线程，前台接待员和服务员是同一个人，全程为顾客服
2) 单 Reactor 多线程，1 个前台接待员，多个服务员，接待员只负责接待
3) 主从 Reactor 多线程，多个前台接待员，多个服务生

## Netty 模型

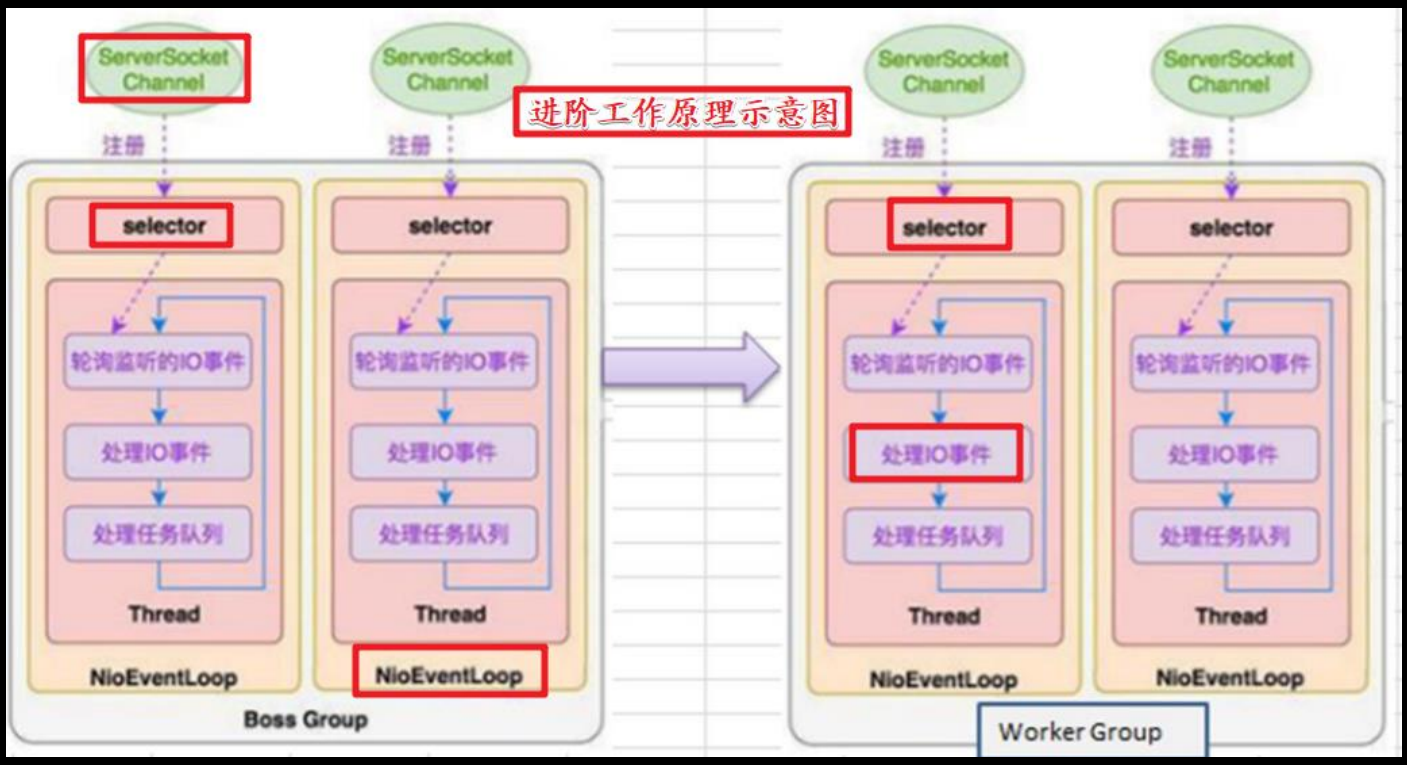
Netty 主要基于 **主从 Reactor 多线程魔性** 改进

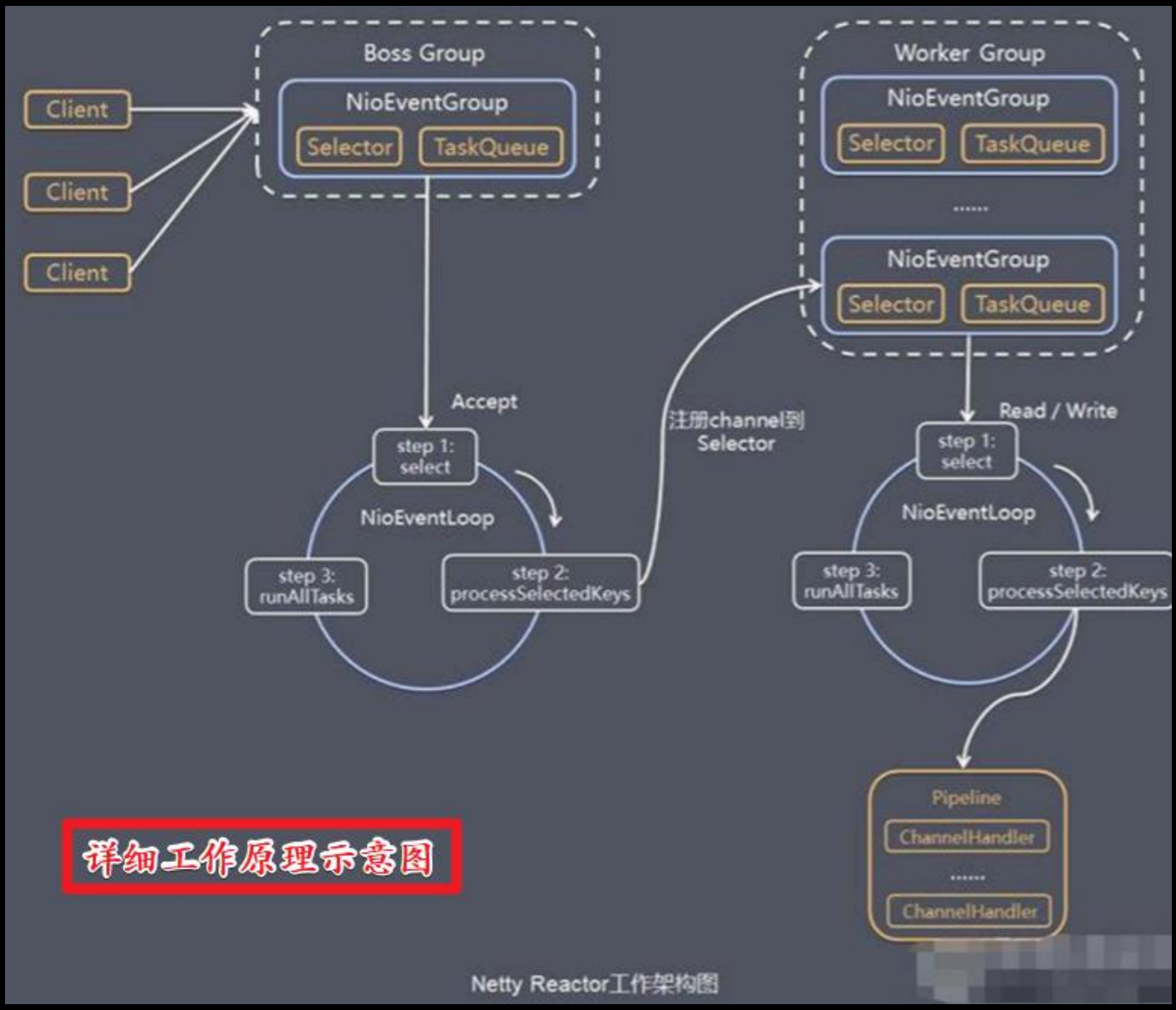



1) **BossGroup** 线程维护 Selector ，只关注 Accecpt

2) 当接收到 Accept 事件，获取到对应的 SocketChannel，封装成 NIOScoketChannel 并注册到 Worker 线程(事件循环)，并进行维护

3) 当 Worker 线程监听到 selector 中通道发生自己感兴趣的事件后，就进行处理(就由 handler)， 注意 handler 已经加入到通道






1) Netty 抽象出两组线程池
   1. BossGroup 专门负责接收客户端的连接
   2. WorkerGroup 专门负责网络的读写
2) BossGroup 和 WorkerGroup 类型都是 **NioEventLoopGroup**
3) NioEventLoopGroup 相当于一个事件循环组，这个组中含有多个事件循环，每一个事件循环是 **NioEventLoop**
4) NioEventLoop 表示一个不断循环的执行处理任务的线程，每个 NioEventLoop 都有一个 selector，用于监听绑定在其上的 socket 的网络通讯
5) NioEventLoopGroup 可以有多个线程，即可以含有多个 NioEventLoop
6) 每个 Boss NioEventLoop 循环执行的步骤有 3 步
   1. 轮询 accept 事件
   2. 处理 accept 事件，与 client 建立连接，生成 NioScocketChannel，并将其注册到某个 worker NIOEventLoop 上的 selector
   3. 处理任务队列的任务，即 runAllTasks
7) 每个 Worker NIOEventLoop 循环执行的步骤
   1. 轮询 read, write 事件
   2. 处理 i/o 事件，即 read，write 事件，在对应 NioScocketChannel 处理
   3. 处理任务队列的任务，即 runAllTasks
8) 每个 Worker NIOEventLoop 处理业务时，会使用 pipeline(管道)，pipeline 中包含了

channel ，即通过 pipeline 可以获取到对应通道，管道中维护了很多的处理器。

案例：

1) Netty 服务器在 6668 端口监听，客户端能发送消息给服务器 "hello，服务器~"

2) 服务器可以回复消息给客户端 "hello，客户端~"

3) 目的：对 Netty 线程模型 有一个初步认识，便于理解 Netty 模型理论

```
NettyServer:
public class NettyServer {
    public static void main(String[] args)  {
        // 创建两个线程组 bossGroup workerGroup。 都是‘无限循环’
        // bossGroup 只会处理连接请求
        // 参数 nThreads：含有的子线程 NioEventLoop 的个数
        NioEventLoopGroup bossGroup = new NioEventLoopGroup(1);
        // workerGroup 处理读写业务
        // nThreads 默认为 实际 cpu 核数 * 2
        NioEventLoopGroup workerGroup = new NioEventLoopGroup();

        // 创建服务器端的启动对象
        ServerBootstrap bootstrap = new ServerBootstrap();
        try {
            // 链式编程 配置
            // 设置两个线程组
            bootstrap.group(bossGroup, workerGroup)
                    // 服务器通道采用 NioServerSocketChannel
                    .channel(NioServerSocketChannel.class)
                    .option(ChannelOption.SO_BACKLOG, 128)
                    // 保持 alive
                    .childOption(ChannelOption.SO_KEEPALIVE, true)
                    // 创建一个通道初始化对象
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        // 给 pipeline 设置处理器
                        @Override
                        protected void initChannel(SocketChannel ch) throws Exception {
                            ch.pipeline().addLast(new NettyServerHandler());
                        }
                    });
            System.out.println("the netty server is ready!");

            // 绑定端口、启动服务器，返回 Future 对象
            ChannelFuture cf = bootstrap.bind(6666).sync();

            // 对关闭通道进行监听
            cf.channel().closeFuture().sync();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }finally {
            bossGroup.shutdownGracefully();
            workerGroup.shutdownGracefully();
        }
    }
}


NettyServerHandler:
public class NettyServerHandler extends ChannelInboundHandlerAdapter {
    //读取数据实际(这里我们可以读取客户端发送的消息)
    /**
    1. ChannelHandlerContext ctx:上下文对象，含有 管道 pipeline ，通道 channel，地址
    2. Object msg: 就是客户端发送的数据 默认 Object
    */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
```

```
        System.out.println("服务器读取线程 " + Thread.currentThread().getName());
        System.out.println("server ctx =" + ctx);
        System.out.println("看看 channel 和 pipeline 的关系");
        Channel channel = ctx.channel();
        //本质是一个双向链接, 出站入站
        ChannelPipeline pipeline = ctx.pipeline();
        //将 msg 转成一个 ByteBuf
        ByteBuf buf = (ByteBuf) msg;
        //ByteBuf 是 Netty 提供的, 不是 NIO 的 ByteBuffer. ByteBuf buf = (ByteBuf) msg;
        System.out.println("客户端发送消息是:" + buf.toString(CharsetUtil.UTF_8));
        System.out.println("客户端地址:" + channel.remoteAddress());
    }

    /**
     * 数据读取完毕
     * */
    @Override
    public void channelReadComplete(ChannelHandlerContext ctx) throws Exception {
        //writeAndFlush 是 write + flush
        //将数据写入到缓存, 并刷新
        //一般讲, 我们对这个发送的数据进行编码
        ctx.writeAndFlush(Unpooled.copiedBuffer("hello, 客户端~(>^ω^<)喵",
CharsetUtil.UTF_8));
    }

    /**
     * 处理异常, 一般是需要关闭通道
     * */
    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
        ctx.close();
    }
}



NettyClient:
public class NettyClient {
    public static void main(String[] args)  {
        // 客户端需要一个时间循环组
        NioEventLoopGroup group = new NioEventLoopGroup();

        // 创建一个启动对象: client 使用的是 Bootstrap 而不是 ServerBootstrap
        Bootstrap bootstrap = new Bootstrap();

        // 链式编程配置
        // 设置线程组
        bootstrap.group(group)
                // 设置客户端通道实现
                .channel(NioSocketChannel.class)
                .handler(new ChannelInitializer<SocketChannel>() {
                    @Override
                    protected void initChannel(SocketChannel ch) throws Exception {
                        // 使用自己的 handler
                        ch.pipeline().addLast(new NettyClientHandler());
                    }
                });
        System.out.println("client is ready...");

        try {
            // 启动客户端, 连接服务器
            ChannelFuture future = bootstrap.connect(new InetSocketAddress("127.0.0.1",
6666))
                    .sync();
            // 关闭通道的监听
            future.channel().closeFuture().sync();
        }catch (Exception e) {
            e.printStackTrace();
```

```
        }finally {
            // 优雅第关闭
            group.shutdownGracefully();
        }
    }
}


public class NettyClientHandler extends ChannelInboundHandlerAdapter {
    /**
     * 当通道就绪时触发
     * */
    @Override
    public void channelActive(ChannelHandlerContext ctx) throws Exception {
        System.out.println("client : " +ctx);
        ctx.writeAndFlush(Unpooled.copiedBuffer("edwinxu", CharsetUtil.UTF_8));
    }

    /**
     * 当通道有读取事件时触发
     * */
    @Override
    public void channelRead(ChannelHandlerContext ctx, Object msg) throws Exception {
        ByteBuf buf = (ByteBuf) msg;
        System.out.println("服务器回复: "+buf.toString(CharsetUtil.UTF_8));
        System.out.println("服务器地址: "+ctx.channel().remoteAddress());
    }

    @Override
    public void exceptionCaught(ChannelHandlerContext ctx, Throwable cause) throws
Exception {
        ctx.close();
    }
}
```

任务队列中的 Task 有 3 种典型使用场景

1. 用户程序自定义的普通任务

```
Handler 里面:

ctx.channel().eventLoop().execute(new Runnable() {
@Override
public void run() {
try {
  Thread.sleep(5 * 1000);
  ctx.writeAndFlush(Unpooled.copiedBuffer("hello, 客户端~(>^ω^<)喵 2",
CharsetUtil.UTF_8));
  System.out.println("channel code=" + ctx.channel().hashCode());
} catch (Exception ex) {
  System.out.println("发生异常" + ex.getMessage());
}
}
});
```

2. 用户自定义定时任务
3. 非当前 Reactor 线程调用 Channel 的各种方法

1) Netty 抽象出两组线程池，BossGroup 专门负责接收客户端连接，WorkerGroup 专门负责网络读写操作。

2) **NioEventLoop 表示一个不断循环执行处理任务的线程**，每个 NioEventLoop 都有一个 selector，用于监听绑定在其上的 socket 网络通道。
3) NioEventLoop 内部采用串行化设计，从消息的读取->解码->处理->编码->发送，始终由 IO 线程 NioEventLoop 负责
  NioEventLoopGroup 下包含多个 NioEventLoop
  每个 NioEventLoop 中包含有一个 Selector，一个 taskQueue
  每个 NioEventLoop 的 Selector 上可以注册监听多个 NioChannel
  每个 NioChannel 只会绑定在唯一的 NioEventLoop 上
  每个 NioChannel 都绑定有一个自己的 ChannelPipeline

## 异步模型

tty 中的 I/O 操作是异步的，包括 Bind、Write、Connect 等操作会简单的返回一个 **ChannelFuture**。
调用者并不能立刻获得结果，而是通过 **Future-Listener** 机制，用户可以方便的主动获取或者通过通知机制获得 IO 操作结果
Netty 的异步模型是建立在 future 和 callback 的之上的。callback 就是回调。重点说 Future，它的核心思想是：假设一个方法 fun，计算过程可能非常耗时，等待 fun 返回显然不合适。那么可以在调用 fun 的时候，立马返回一个 Future，后续可以通过 Future 去监控方法 fun 的处理过程(即 ： Future-Listener 机制)

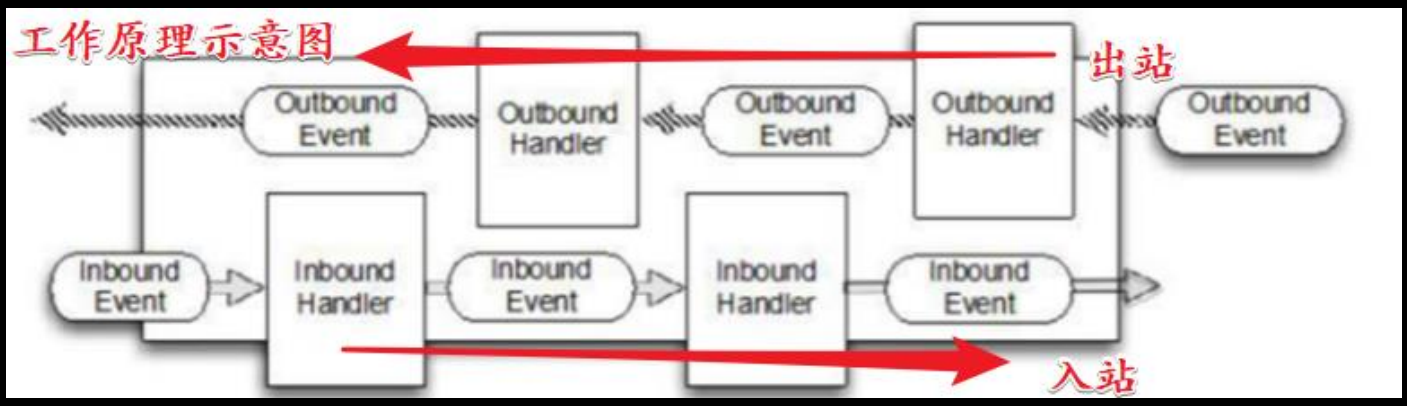
Future 说明:
表示异步的执行结果，可以通过它提供的方法来检测执行是否完成

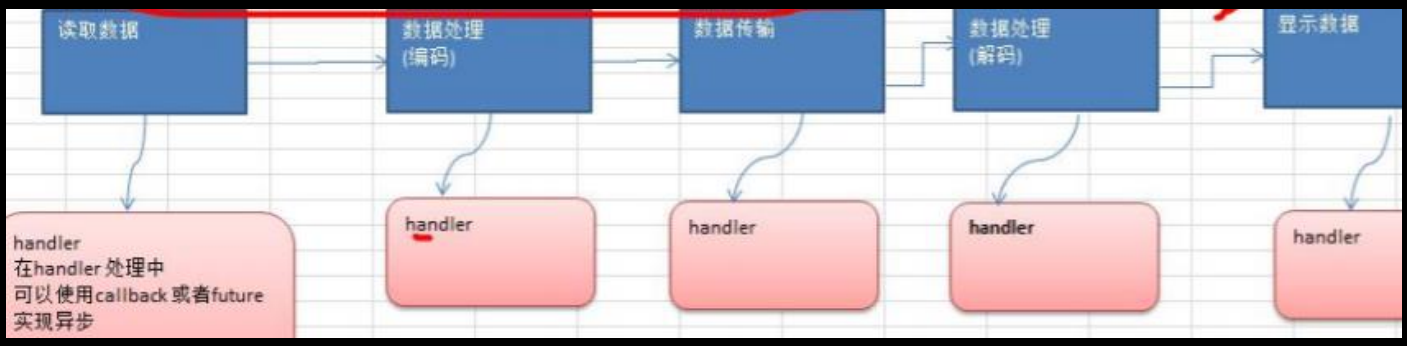
```
public class Task implements Callable<Integer> {
    public Integer call() throws Exception {
        Thread.sleep(10 * 1000);
        return 1;
    }
}

    /**
     * 一般不会这样用
     * */
    public static void main(String[] args) throws Exception {
        Task task = new Task();
        // 这里会阻塞，直到 Task 执行完
        Integer future = task.call();
        System.out.println(future);
    }


    //Future 一般结合 ExecutorService 使用
    public static void main(String[] args) throws ExecutionException, InterruptedException
{
        ExecutorService executor = Executors.newCachedThreadPool();
        Future<Integer> future = executor.submit(new Task());
        while (!future.isDone()){
            System.out.println("Task 还在执行中！ ");
        }
        System.out.println("Task 执行结果: "+future.get());
    }
```

ChannelFuture 是一个接口 ：

```
public interface ChannelFuture extends Future<Void>
```

我们可以添加监听器，当监听的事件发生时，就会通知到监听器




当 Future 对象刚刚创建时，处于非完成状态，调用者可以通过返回的 ChannelFuture 来获取操作执行的状态，注册监听函数来执行完成后的操作。

1. 通过 isDone 方法来判断当前操作是否完成；
2. 通过 isSuccess 方法来判断已完成的当前操作是否成功；
3. 通过 getCause 方法来获取已完成的当前操作失败的原因；
4. 通过 isCancelled 方法来判断已完成的当前操作是否被取消；
5. 通过 addListener 方法来注册监听器，当操作已完成(isDone 方法返回完成)，将会通知指定的监听器；如果 Future 对象已完成，则通知指定的监听器

```
        //启动服务器(并绑定端口)
        ChannelFuture cf = bootstrap.bind(6668).sync();
        //给 cf 注册监听器，监控我们关心的事件
        cf.addListener(new ChannelFutureListener() {
            @Override
            public void operationComplete(ChannelFuture future) throws Exception {
                if (cf.isSuccess()) {
                    System.out.println("监听端口 6668 成功");
                } else {
                    System.out.println("监听端口 6668 失败");
                }
            }
        });
```

## Netty 核心模块组件

### Bootstrap、ServerBootstrap

一个 Netty 应用通常由一个 Bootstrap 开始，主要作用是配置整个 Netty 程序，串联各个组件，

Netty 中 Bootstrap 类是客户端程序的启动引导类，ServerBootstrap 是服务端启动引导类

```
public ServerBootstrap group(EventLoopGroup parentGroup, EventLoopGroup childGroup)，该方法用于服务器端，用来设置两个 EventLoop
public B group(EventLoopGroup group) ，该方法用于客户端，用来设置一个 EventLoop
public B channel(Class<? extends C> channelClass)，该方法用来设置一个服务器端的通道实现
public <T> B option(ChannelOption<T> option, T value)，用来给 ServerChannel 添加配置
public <T> ServerBootstrap childOption(ChannelOption<T> childOption, T value)，用来给接收到的通道添加配置
public ServerBootstrap childHandler(ChannelHandler childHandler)，该方法用来设置业务处理类（自定义的 handler）
public ChannelFuture bind(int inetPort) ，该方法用于服务器端，用来设置占用的端口号
public ChannelFuture connect(String inetHost, int inetPort) ，该方法用于客户端，用来连接服务器端
```

## Future、ChannelFuture

Netty 中所有的 IO 操作都是异步的，不能立刻得知消息是否被正确处理。但是可以过一会等它执行完成或者直接注册一个监听，具体的实现就是通过 Future 和 ChannelFutures，他们可以注册一个监听，当操作执行成功或失败时监听会自动触发注册的监听事件

常见的方法有
Channel channel()，返回当前正在进行 IO 操作的通道
ChannelFuture sync()，等待异步操作执行完毕

Sync():同步化，阻塞在这里，直到状态改变

## Channel

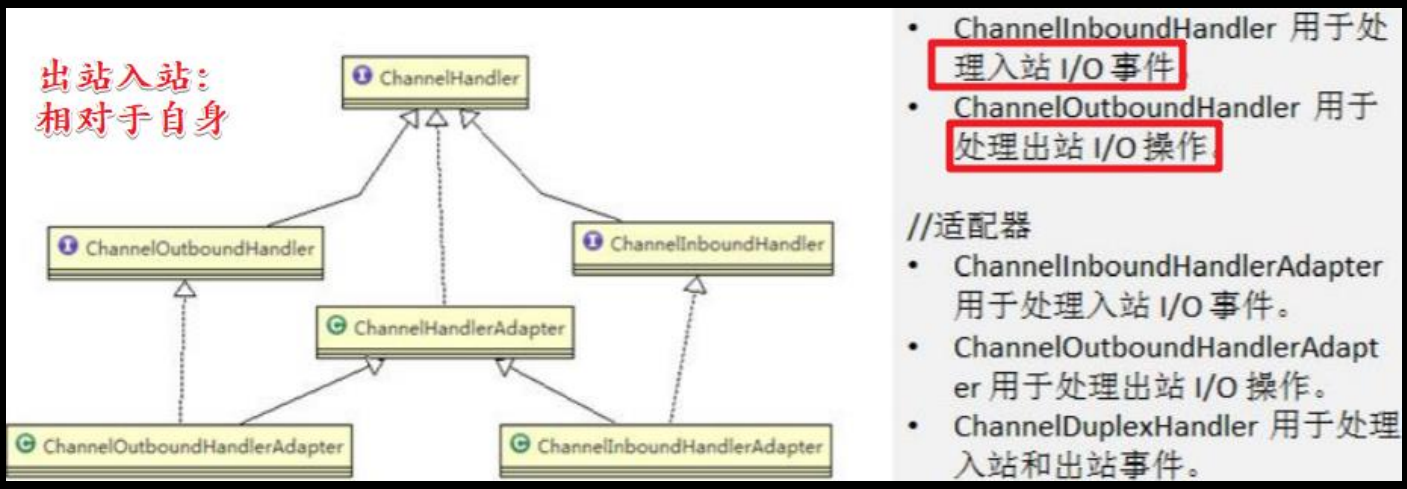
- Netty 网络通信的组件，能够用于执行网络 I/O 操作。
- 通过 Channel 可获得当前网络连接的通道的状态
- 通过 Channel 可获得 网络连接的配置参数 （例如接收缓冲区大小）
- Channel 提供异步的网络 I/O 操作(如建立连接，读写，绑定端口)，异步调用意味着任何 I/O 调用都将立即返回，并且不保证在调用结束时所请求的 I/O 操作已完成
- 调用立即返回一个 ChannelFuture 实例，通过**注册监听器到 ChannelFuture** 上，可以 I/O 操作成功、失败或取消时回调通知调用方
- 支持关联 I/O 操作与对应的处理程序
- 不同协议、不同的阻塞类型的连接都有不同的 Channel 类型与之对应，常用的 Channel 类型：
  1. NioSocketChannel，异步的客户端 TCP Socket 连接。
  2. NioServerSocketChannel，异步的服务器端 TCP Socket 连接。
  3. NioDatagramChannel，异步的 UDP 连接。
  4. NioSctpChannel，异步的客户端 Sctp 连接。
  5. NioSctpServerChannel，异步的 Sctp 服务器端连接，这些通道涵盖了 UDP 和 TCP 网络 IO 以及文件 IO

## Selector

Netty 基于 Selector 对象实现 I/O 多路复用，通过 Selector 一个线程可以监听多个连接的 Channel 事件
当向一个 Selector 中注册 Channel 后，Selector 内部的机制就可以自动不断地查询(Select)这些注册的 Channel 是否有已就绪的 I/O 事件（例如可读，可写，网络连接完成等），这样程序就可以很简单地使用一个线程高效地管理多个 Channel

## ChannelHandler 及其实现类

1) ChannelHandler 是一个接口，**处理 I/O 事件或拦截 I/O 操作，并将其转发到其 ChannelPipeline(业务处理链)中的下一个处理程序。**
2) ChannelHandler 本身并没有提供很多方法，因为这个接口有许多的方法需要实现，方便使用期间，可以继承它的子类

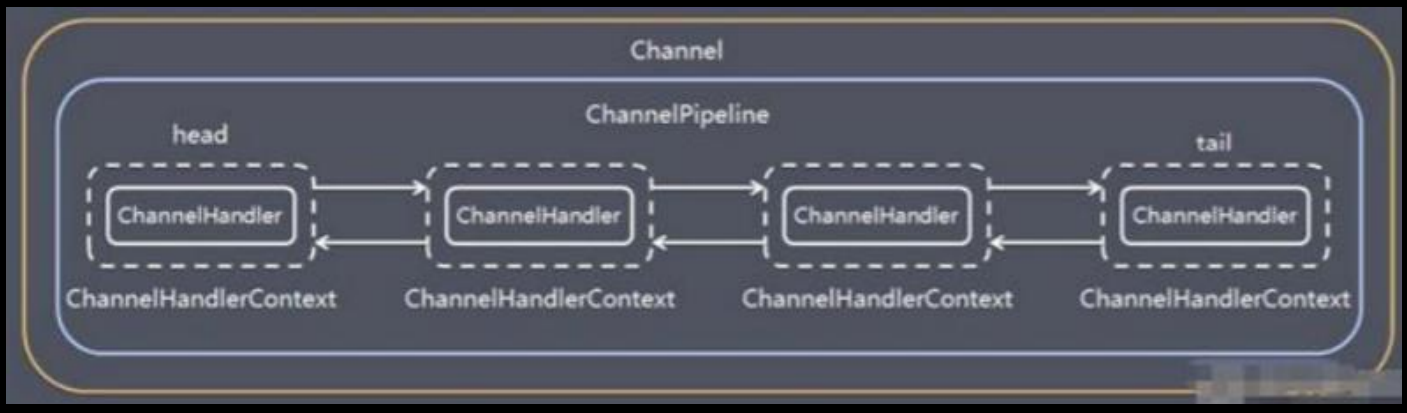



我们经常需要自定义一个 Handler 类去继承 ChannelInboundHandlerAdapter，然后通过重写相应方法实现业务逻辑

## Pipeline 和 ChannelPipeline

ChannelPipeline 是重点

1. **ChannelPipeline 是一个 Handler 的集合，它负责处理和拦截 inbound 或者 outbound 的事件和操作，相当于一个贯穿 Netty 的链。(也可以这样理解：ChannelPipeline 是 保存 ChannelHandler 的 List，用于处理或拦截 Channel 的入站事件和出站操作)**
2. ChannelPipeline 实现了一种高级形式的拦截过滤器模式，使用户可以完全控制事件的处理方式，以及 Channel 中各个的 ChannelHandler 如何相互交互
3. 在 Netty 中每个 Channel 都有且仅有一个 ChannelPipeline 与之对应，它们的组成关系如下




**一个 Channel 包含一个 ChannelPipeline，而 ChannelPipeline 中维护了一个由 ChannelHandlerContext 组成的双向链表，并且每一个 ChannelHandlerContext 中关联着一个 ChannelHandler。**

**入站和出站事件在一个双向链表中，入站事件会从链表 head 往后传递到最后一个 handler，出站事件从链表 tail 往前传递到第一个 handler。两种类型的 handler 互不干扰**

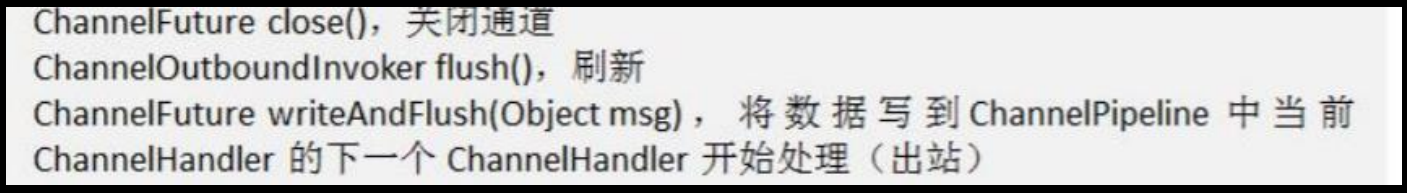
```
ChannelPipeline addFirst(ChannelHandler... handlers)，把一个业务处理类（handler）添加到链中的第一个位置
ChannelPipeline addLast(ChannelHandler... handlers)，把一个业务处理类（handler）添加到链中的最后一个位置
```

(责任链模式)

## ChannelHandlerContext

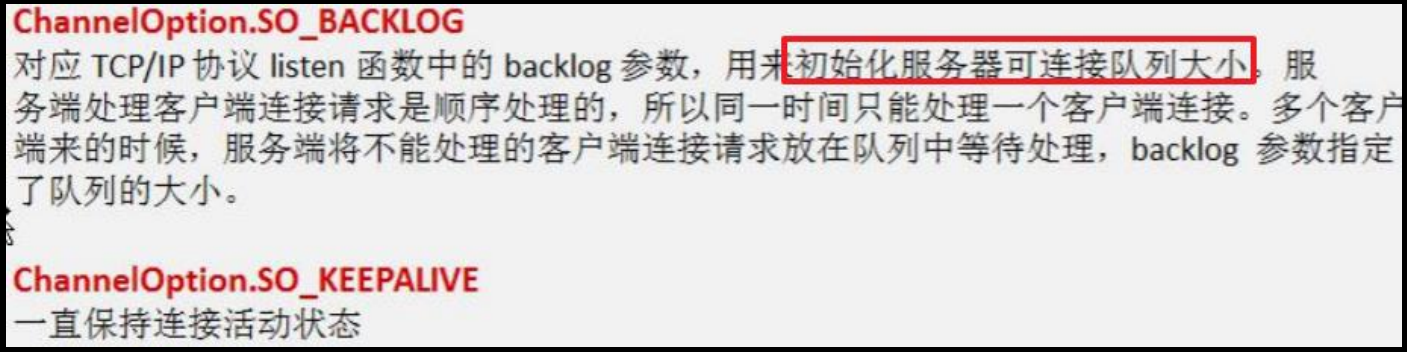
保存 Channel 相关的所有上下文信息，同时**关联一个 ChannelHandler 对象**

即 ChannelHandlerContext 中包含一个**具体的事件处理器** ChannelHandler，同时 ChannelHandlerContext 中也绑定了对应的 pipeline 和 Channel 的信息，方便对 ChannelHandler 进行调用.




## ChannelOption

Netty 在创建 Channel 实例后,一般都需要设置 ChannelOption 参数

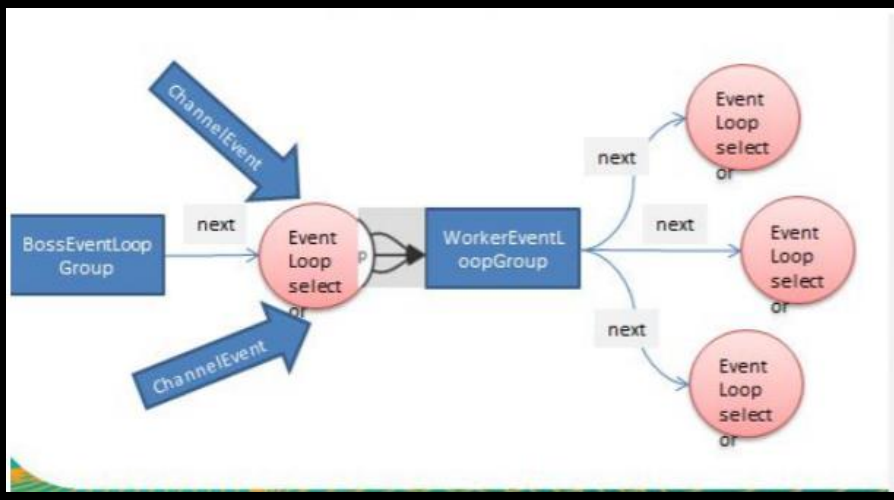


## EventLoopGroup 和其实现类 NioEventLoopGroup

EventLoopGroup 是一组 EventLoop 的抽象，Netty 为了更好的利用多核 CPU 资源，一般会有多个 EventLoop 同时工作，**每个 EventLoop 维护着一个 Selector** 实例。

EventLoopGroup 提供 next 接口，可以从组里面按照一定规则获取其中一个 EventLoop 来处理任务。在 Netty 服务器端编程中，我们一般都需要提供两个 EventLoopGroup，例如：**BossEventLoopGroup 和 WorkerEventLoopGroup。**

通常一个服务端口即一个 ServerSocketChannel 对应一个 Selector 和一个 EventLoop 线程。BossEventLoop 负责接收客户端的连接并将 SocketChannel 交给 WorkerEventLoopGroup 来进行 IO 处理



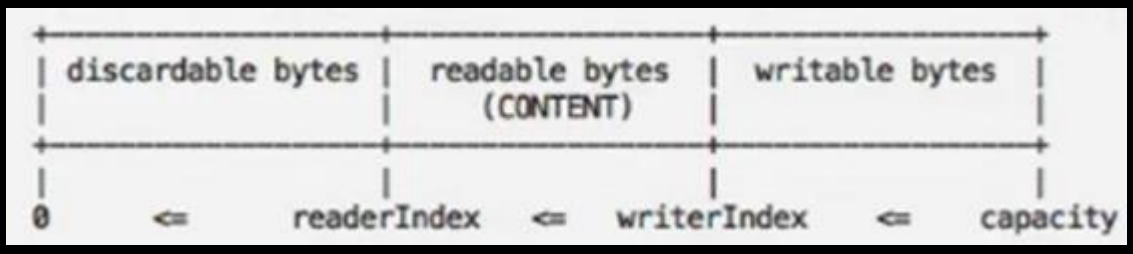

- BossEventLoopGroup 通常是一个单线程的 EventLoop，EventLoop 维护着一个注册了 ServerSocketChannel 的 Selector 实例BossEventLoop 不断轮询 Selector 将连接事件分离出来
- 通常是 OP_ACCEPT 事件，然后将接收到的 SocketChannel 交给 WorkerEventLoopGroup
- WorkerEventLoopGroup 会由 next 选择其中一个 EventLoop来将这个 SocketChannel 注册到其维护的 Selector 并对其后续的 IO 事件进行处理

常用方法

```
public NioEventLoopGroup(), 构造方法
public Future<?> shutdownGracefully(), 断开连接，关闭线程
```

## Unpooled 类

ByteBuf: netty 提供的 buffer
不需要反转 flip
分别维护读写的索引：readerIndex、writerIndex



Unpooled 类:
Netty 提供一个专门用来操作缓冲区(即 Netty 的数据容器)的工具类

```
//通过给定的数据和字符编码返回一个 ByteBuf 对象（类似于 NIO 中的 ByteBuffer 但有区别）
public static ByteBuf copiedBuffer(CharSequence string, Charset charset)
```

## ChannelGroup

管理了一群 Channel

## WebSocket 长连接开发

聊天室？？

Websocket 协议
协议升级

长连接性能优良

看到了 73 集，看不下去了

# 其他知识

UTF-8 中，一个汉字三个字节
回车换行是 2 字节
输入流和输出流，把内存(程序)作为参考点，向内存中写，就是输入，从内存中读取，就是输出。

GB vs GiB
1GB（Gigabyte）= 1000 MB
1GiB （Gibibyte）= 1024 MiB
硬盘厂商用的都是 Gigabyte

被 final 修饰的变量，有三种赋值方式。
1. 在定义时直接赋值。
2. 声明时不赋值，在 constructor 中赋值（最常用的方式）
3. 声明时不赋值，在构造代码块中赋值

被 final static 修饰的变量，有两种赋值方式
1. 在定义时直接赋值.
2. 在静态代码块里赋值

# 这十个软件,让你的电脑舒适度提升 1400%

https://www.bilibili.com/video/BV137411K7V1/?spm_id_from=autoNext

7-Zip 解压缩软件
Geek Uninstaller 卸载软件
PotPlayer 播放器
IDM 下载器
Clover 使用资源管理器的插件
QuickLook Windows 快速预览
Notepads Windows 代码软件
Wox 一个小插件
Everything 快速搜索本地磁盘文件
链接: https://pan.baidu.com/s/1RRQAktCo1tNnuTmicK4uLQ
提取码: guhj

附: 官网:
7-Zip https://sparanoid.com/lab/7z/
Geek Uninstaller https://geekuninstaller.com/
Microsoft Edge https://www.microsoft.com/en-us/edge
PotPlayer https://potplayer.daum.net/?lang=zh_CN
IDM http://www.internetdownloadmanager.com/
Clover http://cn.ejie.me/
QuickLook Windows 商店搜索即可
Wox https://github.com/Wox-launcher/Wox/releases
Everything https://www.voidtools.com/zh-cn/
我自己整理的官方安装包:
链接: https://pan.baidu.com/s/1RRQAktCo1tNnuTmicK4uLQ
提取码: guhj