概念理解

同步异步 阻塞非阻塞

阻塞/非阻塞是看用户线程在系统调用下的处理方式，比如读取IO数据时，要发起系统调用切换到内核态，如果此时用户线程挂起，就是阻塞IO，如果立即返回，那就是非阻塞IO。 主要是看这种情况下用户线程的处理方式。

同步/异步是看数据准备的结果，同步的话，如果返回，那必定是准备好了数据，如果是异步的话，返回不一定数据准备好，要等待一个事件回调来处理。

BIO同步阻塞　NIO同步非阻塞　AIO异步非阻塞　Netty异步非阻塞

Unix系统下的5种IO模型：（主要理解EPOLL模型，其他模型知道有就行）

阻塞IO：

定义：进程在进行IO操作时会挂起，会一直阻塞到内核缓冲区数据准备好并复制到用户缓冲区之后。
例子：老王去钓鱼，把带有鱼饵的钓竿放进水里后就做在河边一直盯着，啥也不干，等到鱼上钩才把鱼钓上来放进桶里。期间什么也干不了。

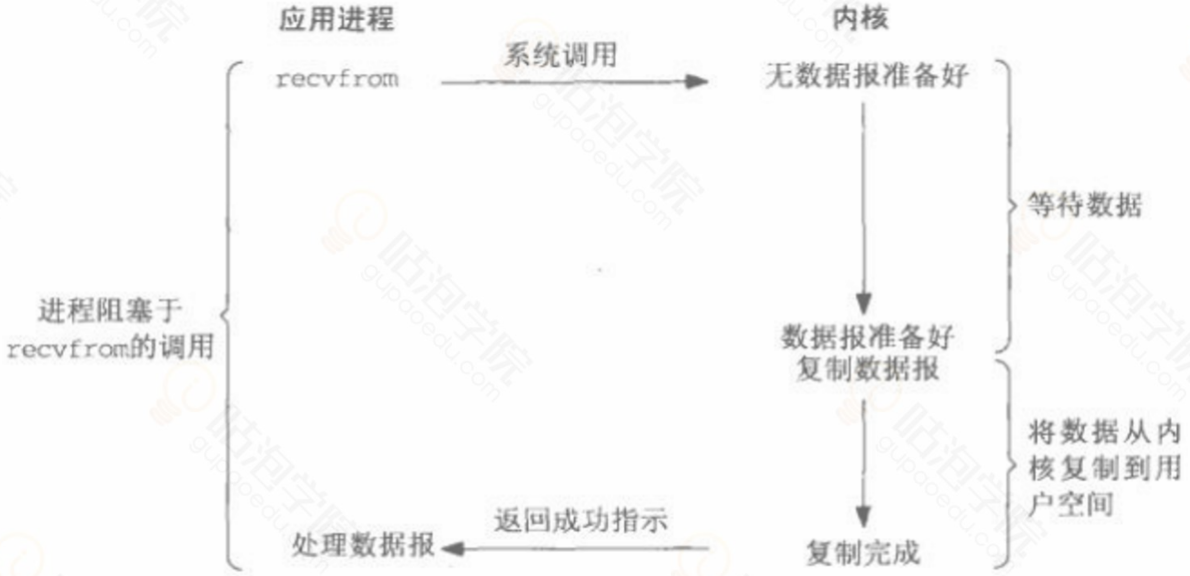


图6-1　阻塞式I/O模型

流程解释：

1. 用户进程需要进行IO操作时，会进行一次系统调用，进入到内核态，此时用户进程被挂起。处于阻塞状态。此时进程不会再占用cpu资源。
2. 内核进行数据的准备，把需要的数据填充到内核缓冲区。
3. 内核缓冲区数据填充完毕，把数据从内核缓冲区复制到用户缓冲区。
4. 数据复制完毕，返回，从内核态从新切换到用户态，进程进入就绪状态等待cpu执行。

总结：

- 用户进程从进行系统调用进入内核态后，会一直挂起，阻塞到数据从内核缓冲区复制到用户缓冲区完毕。期间不会消耗cpu资源。
- 适用并发量小的网络应用开发。

非阻塞IO：

定义：用户进程在进行IO操作后，不会被挂起，还是会继续执行逻辑。
例子：老王去钓鱼，把带有鱼饵的钓竿放进水里后，期间可以玩手机，看微信，只是要不断地把眼睛看向合理，看鱼有没有上钩，没有上钩就继续玩手机，有就收杆，把鱼放进桶里。

图6-2　非阻塞式I/O模型

流程说明：

1. 用户进程需要进行IO操作时，会进行一次系统调用，进入到内核态，如果数据没有准备好，立即返回EWOULDBLOCK。此时不会造成进程阻塞。进程还可以继续处理其他事情。
2. 进程会轮询查看内核数据是否准备好，如果没有准备好，就继续立即返回EWOULDBLOCK，不阻塞进程。
3. 轮询到数据准备好了后，进行数据复制，从内核缓冲区复制到用户缓冲区。在此期间进程会挂起，处于阻塞状态，知道数据复制完成。
4. 数据复制完毕后返回，进程转为就绪态，等待cpu调度。

总结：

- 非阻塞IO因为要用轮询代替了阻塞，使得进程在内核数据准备期间不会阻塞，可以执行其他事情，但是因为要轮询，所以会对CPU资源造成较大的消耗。
- 在进行内核态往用户态数据复制过程中，进程还是会处于阻塞状态的。
- 虽然非阻塞IO可以使得进程在IO内核数据准备期间不阻塞，可以执行其他事情，但是，由于轮询需要消耗较大的cpu资源，所以会使得服务端处理和响应请求会有较大的延时。
- 适用并发量较小、且不需要及时响应的网络应用开发。

IO多路复用：

IO多路复用有基于select/poll的对路复用，也有基于epoll的多路复用。

基于select/poll的多路复用：

多个网络IO连接可以注册到一个复路器select上，然后由一个进程或者线程调用该复路器，调用该复路器会使得进程或者线程挂起，处于阻塞状态，内核会轮询监视该复路器上的每一个连接，一旦有一个连接的数据准备好了，该select会返回，然后进程或者线程退出阻塞状态，然后该进程或者线程会进行系统调用，把内核缓冲区的数据复制到用户缓冲区。

例子：老王去河边钓鱼，不过他有点贪心，一次性使用多个鱼竿钓鱼（假设10个），然后把十个鱼竿放进合理，然后就眼睛不断从左往右循环看每一根鱼竿是否有鱼上钩，其中一根鱼竿上钩了，就把鱼钓起放进桶里。

流程说明：

1. 连接到服务进程上的多个套接字连接会注册到复路器select上。
2. 进程调用select系统调用。进入内核态，应用进程挂起，内核会对select上的连接进行监视轮询，监视连接的数据是否准备好。此间进程会处于阻塞状态。
3. 一旦select复用器上的一个或者多个连接数据准备好了，select会返回，然后进程会取消阻塞状态并再发起系统调用recvfrom把内核缓冲区的数据复制到用户缓冲区，此时进程又会被挂起，此时的系统调用recvfrom时内核缓冲区数据必定是准备好的。
4. 数据复制完成返回。

与阻塞IO的区别：

- 基于select/poll的IO复用与阻塞IO类似，只不过阻塞IO是一直阻塞在recvfrom系统调用等待数据准备好并复制到缓冲区，而IO复用会先阻塞在select或者poll系统调用，监视某一个连接数据准备好的再返回，然后调用recvfrom系统调用进行数据复制，只是此时的recvfrom不再需要等待数据准备，可以直接复制。时间较短。主要阻塞在select/poll系统调用上。
- 基于select/poll的IO复用实际上比阻塞IO多了一个步骤，多进行了一次系统调用，只不过IO复用的优势不是在于处于一个连接更快，而是可以处理更多的连接。
- 在处理连接数不高的情况下，基于select/poll的IO多路复用的服务器不一定会比多线程+阻塞IO的服务器性能高，可能延时会更大。

总结：

- 基于select或者poll模型的IO多路复用基本是一样的只是基于select的模型默认能同时接收的连接数是1024个，因为一个进程默认最多打开1024个fd文件描述符。而基于poll模型的IO多路复用就没有限制，因为是基于链表来存储的。
- 基于select/poll的IO多路复用也不能设置太大的连接数，因为监视复用器的连接数据是否准备好是采用循环进行无差别的监视，时间复杂度为O(n)，如果连接数太大的话，循环监视一次的时间会相对长，反而会降低监视的效率。

基于epoll的IO多路复用：

回顾基于select/poll的IO多路复用缺点：

- select的模型默认能同时接收的连接数是1024个，因为一个进程默认最多打开1024个fd文件描述符。而基于poll模型的IO多路复用就没有限制。
- 对复路器上的连接的监视轮询是线性时间复杂度O(n),也就是说随着连接数的增加，对复路器的监视效率会降低。

基于epoll的IO多路复用的改进：

- 一个进程打开的fd连接文件描述符没有限制。会限制于内存大小，1GB内存大概可以打开10w个。
- 利用每个文件描述符fd上的callback函数来实现异步回调，省略了对复路器上的连接监视轮询的开销。时间复杂度O(1)，就不会随着连接数的增多而降低。

例子：老王去河边钓鱼，不过他有点贪心，一次性使用多个鱼竿钓鱼（假设10个）（不过老王这从使用的是升级版的鱼竿，每根鱼竿上绑着一个小铃铛，当有鱼上钩时铃铛会响），然后把十个鱼竿放进河里，因为使用了升级版鱼竿，使用老王不用从左到右盯着鱼竿，但是也什么也干不了，只能那发呆。等到某一根鱼竿有鱼上钩，铃铛就会响，然后老王就把那根鱼竿收杆，把鱼放进桶里。

流程：

1. 连接到服务进程上的多个套接字连接会注册到复路器上。
2. 进程调用epoll系统调用。进入内核态，应用进程挂起。
3. 一旦复用器上的某个连接数据准备好了，就会通过该连接套接字描述符fd上的回调函数通知应用进程并，然后进程会取消阻塞状态并再发起系统调用recvfrom把内核缓冲区的数据复制到用户缓冲区，此时进程又会被挂起，此时的系统调用recvfrom时内核缓冲区数据必定是准备好的。
4. 数据复制完成返回。

基于select/poll的IO复用与基于epoll的IO复用对比：

- 基于select的IO复用会有最大连接数限制，基于poll的IO复用没有限制，但是这两个都会随着连接数的增加而性能线性降低。而基于epoll的IO复用采用的是异步回调方式通知用户进程，不会随着连接数增加而性能线性降低。
- 在select poll epoll都会阻塞进程。之后的recvfrom系统调用也会阻塞进程，只是调用recvfrom代表着数据已经准备好了，可以直接复制，所以IO复用的阻塞主要集中在select poll epoll上。

信号驱动：

老王去河边钓鱼，用的升级版鱼竿，放下鱼竿后他可以干其他事，玩游戏，刷微博等，等听到铃铛响了之后，就把鱼放到桶里。

图 6.4 信号驱动 I/O 模型

流程说明：

1. 在Socket连接上安装一个信号处理函数，然后进程调用sigaction系统调用，但是立即返回，进程不用阻塞。继续执行。

2. 当某个Socket的数据准备好了之后，进程会收到一个SIGIO信号，可以在信号处理函数中调用recvfrom进行数据的复制。复制过程中进程阻塞。
3. 复制完成返回。

异步IO：

异步IO是最快的。

例子：老王去河边钓鱼，用的终极版鱼竿（自动放进桶里并播放响铃），放下鱼竿后他可以干其他事，玩游戏，刷微博等，鱼上钩了以后，终极版鱼竿会自动收杆并把鱼放进桶里后，响铃会响，通知老王把鱼煮了。

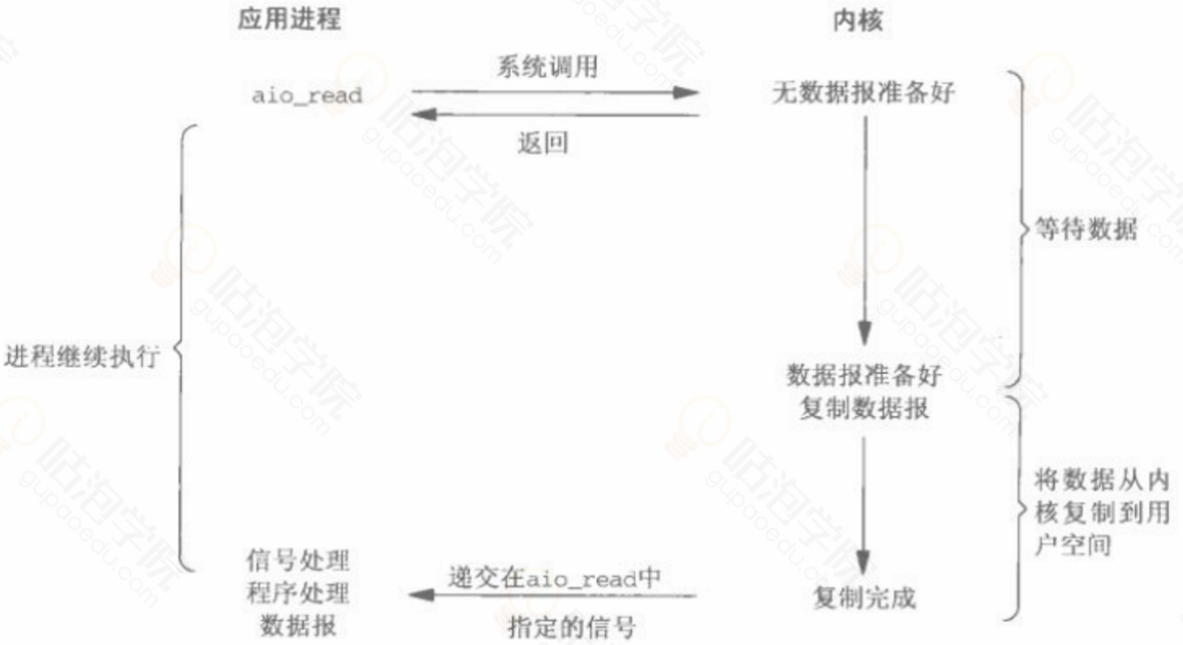


图6-5　异步I/O模型

流程说明：

1. 1. 应用进程接收到IO连接，发起一次aio_read系统调用，然后立即返回，进程不阻塞，可以继续执行。
   2. 等待数据准备好。
   3. 数据准备好了以后，内核直接把数据从内核缓冲区复制到用户缓冲区，而无需用户进程发起系统调用再进行复制。
   4. 数据复制完以后返回指定信号给用户进程，此时数据已经在用户的缓冲区了，所以用户进程可以直接在用户缓冲区拿数据处理。

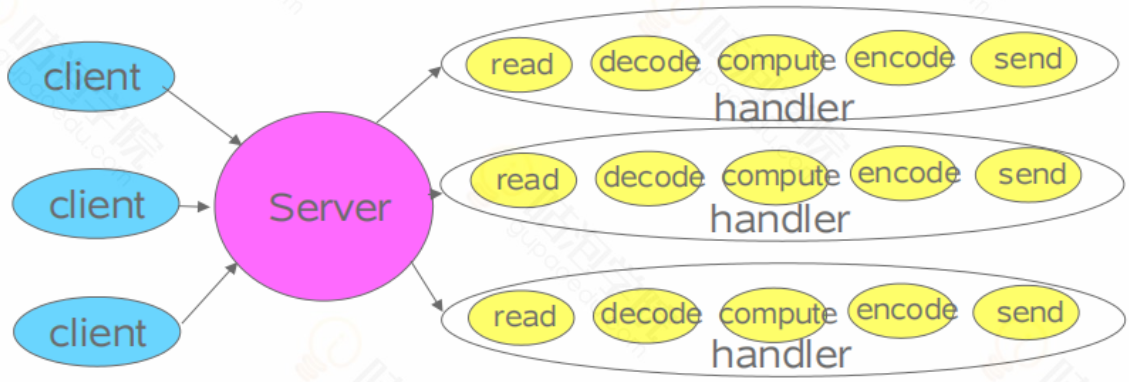
Reactor模式（重点）

《**Scalable IO in Java**》http://gee.cs.oswego.edu/dl/cpjslides/nio.pdf

网络io流程：

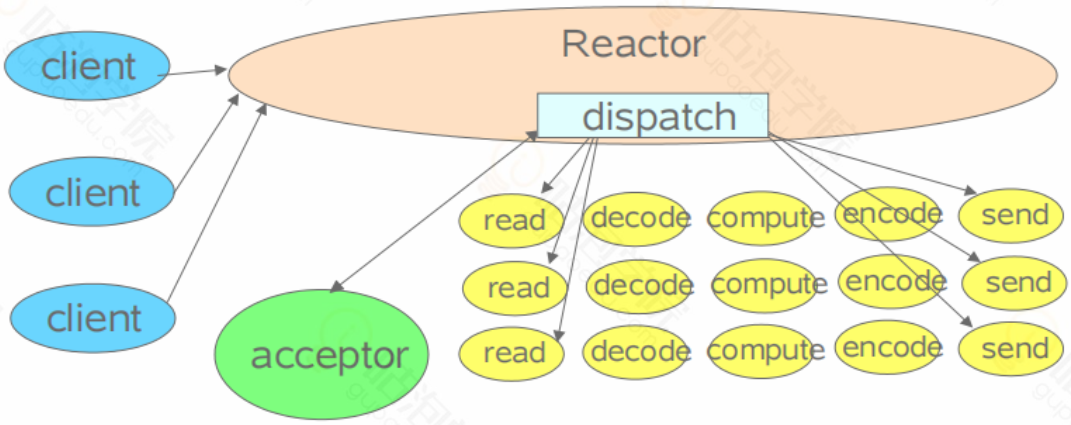
- 读取请求数据
- 对请求数据进行解码
- 对数据进行业务处理
- 对回复数据进行编码
- 发送回复

传统的-阻塞式




```
public class BioSocketServer {
    public static void main(String[] args) throws IOException {
        //建立socket连接，并监听端口
        ServerSocket serverSocket = new ServerSocket(8888);
        while (true) {
            //这里阻塞等待连接
            Socket client = serverSocket.accept();
            //当有连接建立时线程处理 只是演示 真实环境所有线程都应该通过线程池创建
            new Thread(new Runnable() {
                @Override
                public void run() {
                    try {
                        //处理器
                        handler(client);
                    } catch (IOException e) {
                        e.printStackTrace();
                    }
                }
            }).start();
        }
    }

    /**
     * 处理器
     */
    private static void handler(Socket client) throws IOException {
        byte[] bytes = new byte[1024];
        // 第二个阻塞，获取客户端发来的数据，假如客户端没有发数据，则会一直阻塞
        int read = clientSocket.getInputStream().read(bytes);
        if (read != -1) {
            System.out.println( new String(bytes, 0, read));
        }
    }
}
```

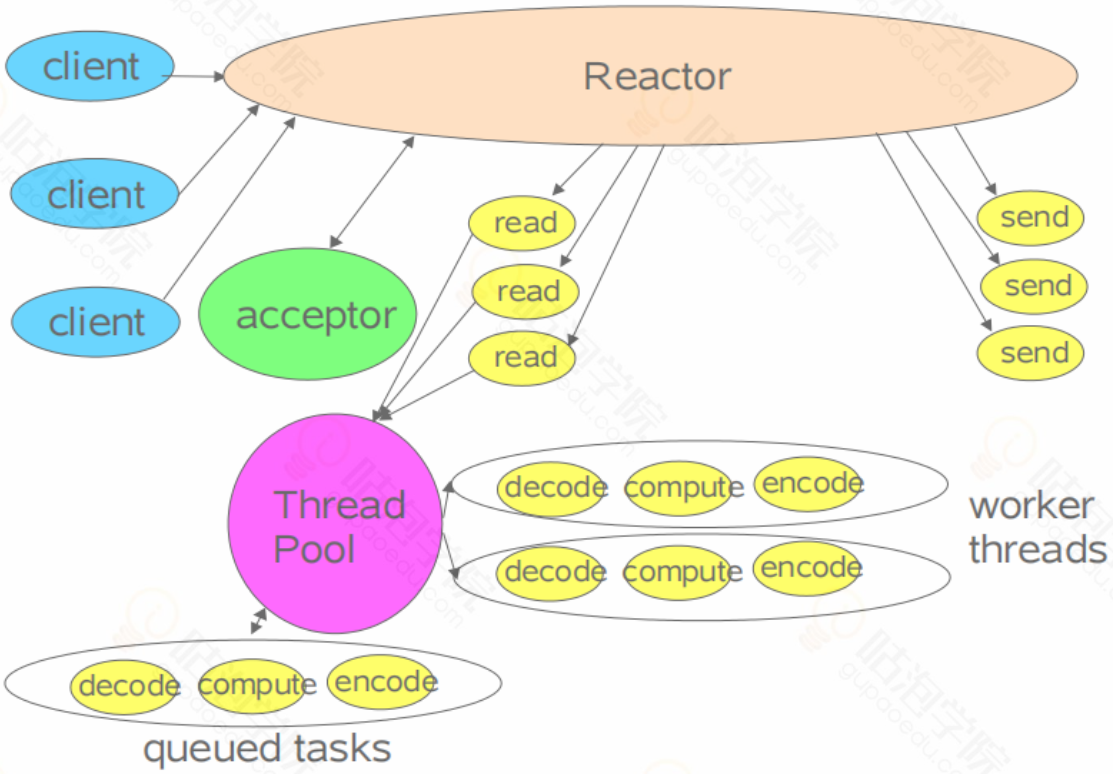
基础的reactor模型-单reactor单线程处理




```
public class NioSelectorSignThreadServer {
    public static void main(String[] args) throws IOException {
        // 创建Nio连接通道
        ServerSocketChannel serverSocket = ServerSocketChannel.open();
        //绑定监听端口
        serverSocket.socket().bind(new InetSocketAddress(8888));
        // 设置ServerSocketChannel为非阻塞
        serverSocket.configureBlocking(false);
        // 打开Selector处理Channel，即创建epoll 这里底层调用了系统的epoll_create方法，创建了epoll对象
        Selector selector = Selector.open();
        // 把ServerSocketChannel注册到selector上，且绑定OP_ACCEPT事件  这里底层调用了epoll_ctl将事件加入到epoll中
        serverSocket.register(selector, SelectionKey.OP_ACCEPT);//标注1
        System.out.println("nio服务启动成功");
        while (true) {
            // 这里是阻塞的 nio并非是异步非阻塞 而是同步阻塞
            // 阻塞等待需要处理的事件发生  这里调用了系统的epoll_wait方法,轮训就绪队列rdlist
            // 返回的select是此次获取的已就绪事件个数
            int select = selector.select();
            if(select <= 0){
                continue;
            }
            // 获取selector中注册的全部事件的就绪事件 SelectionKey 实例
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            // 遍历SelectionKey对事件进行处理
            while (iterator.hasNext()) {
                SelectionKey selectionKey = iterator.next();
                // OP_ACCEPT事件(相对于server端，accept事件就是客户端注册事件，在客户端就是OP_CONNECT事件)
                // 开始进行连接处理，和读事件注册
                if (selectionKey.isAcceptable()) {
                    // 注意这里的channel类型，因为OP_ACCEPT事件是ServerSocketChannel注册的(标注1)，所以OP_ACCEPT事件返回的对象也是ServerSocketChannel
                    ServerSocketChannel server = (ServerSocketChannel) selectionKey.channel();
                    SocketChannel socketChannel = server.accept();
```

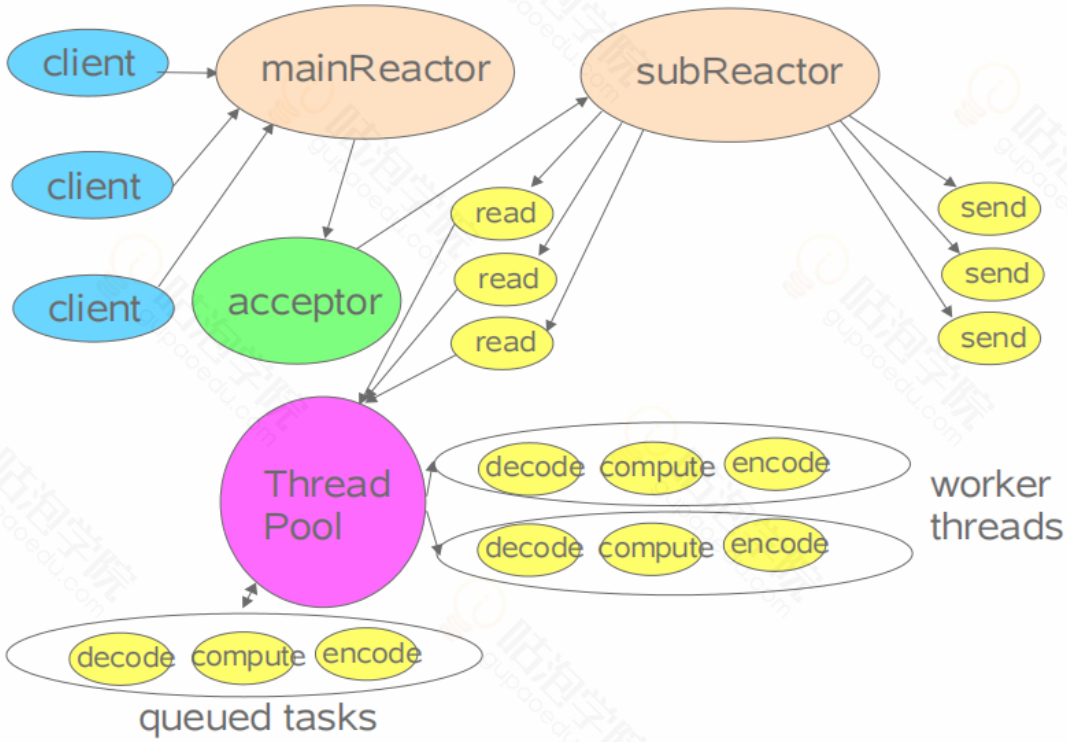
```
                    // 这里要注意将客户端连接注册成了异步，select模式属于异步，必须要注册成
异步才支持select模式
                    socketChannel.configureBlocking(false);
                    // 注册需要的事件 读(OP_READ)事件
                    socketChannel.register(selector,
SelectionKey.OP_READ|SelectionKey.OP_WRITE);
                    System.out.println("客户端连接成功");

                    // 读事件，处理客户端传过来的数据
                } else if (selectionKey.isReadable()) {
                    SocketChannel socketChannel = (SocketChannel)
selectionKey.channel();
                    ByteBuffer byteBuffer = ByteBuffer.allocateDirect(1024);
                    int len = socketChannel.read(byteBuffer);
                    // 如果有数据，把数据打印出来
                    if (len > 0) {
                        System.out.println("接收到消息：" + new
String(byteBuffer.array()));
                        selectionKey.interestOps(SelectionKey.OP_READ);
                        // 如果客户端断开连接，关闭Socket
                    } else if (len == -1) {
                        System.out.println("客户端断开连接");
                        socketChannel.close();
                    }
                    //写事件 水平触发 需要移除该事件 需要时再注册
                } else if(selectionKey.isWritable()){
                    SocketChannel socketChannel = (SocketChannel)
selectionKey.channel();
                    System.out.println("write事件");
                    // NIO事件触发是水平触发
                    selectionKey.interestOps(SelectionKey.OP_READ);
                }
                //将已处理的事件移除，防止二次处理
                iterator.remove();
            }
        }
    }
}
```

多线程处理-单reactor-多线程处理



*将上面的处理select事件部分改为多线程即可

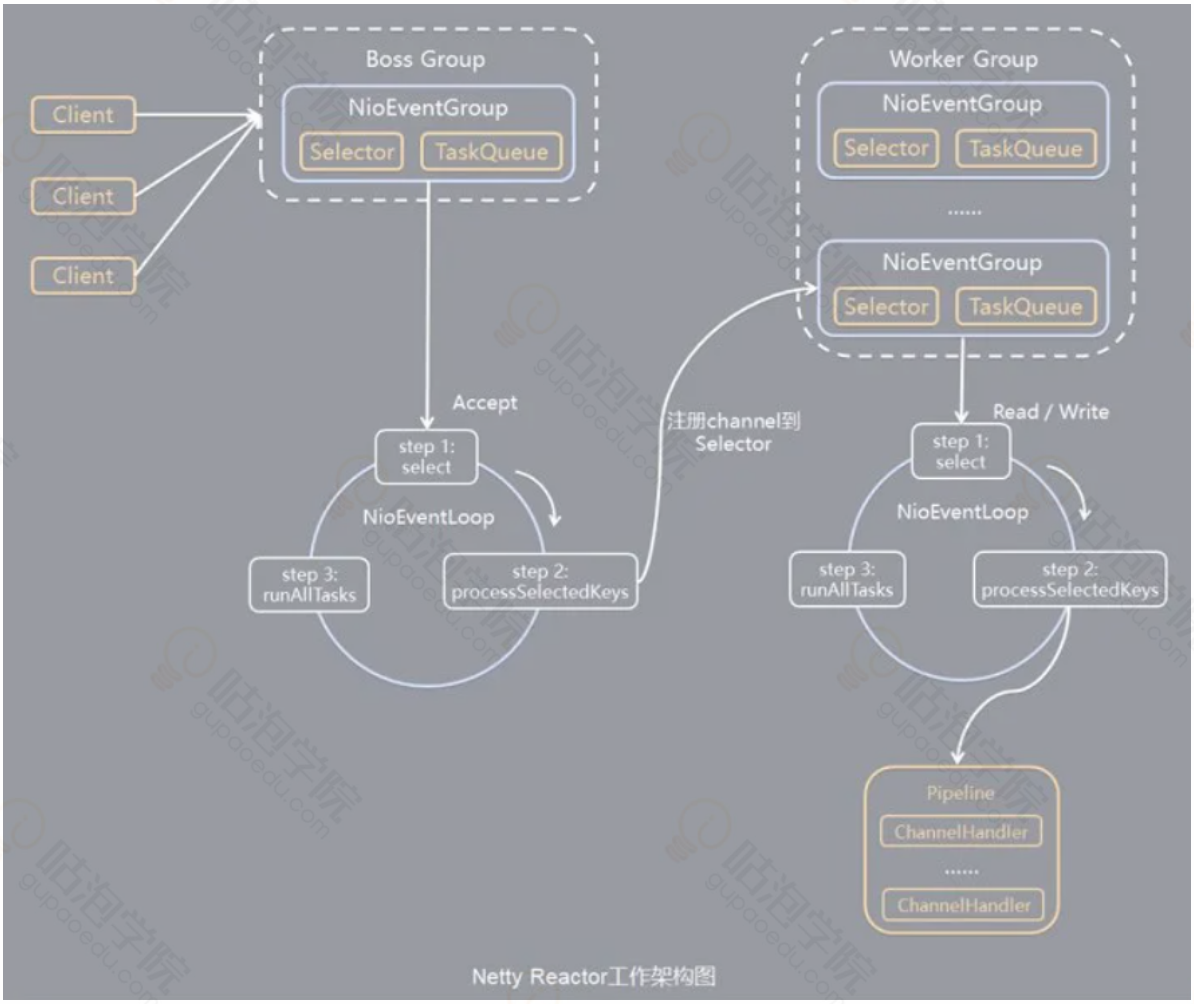
多个反应器的多线程模式-多reactor多线程处理



*比如netty的parentGroup、childGroup 属于主从reactor模式

```
public class NettyServer {
    public static void main(String[] args) throws InterruptedException {
```

```
        NioEventLoopGroup parentGroup = new NioEventLoopGroup();
        NioEventLoopGroup childGroup = new NioEventLoopGroup();

        try{
            ServerBootstrap bootstrap = new ServerBootstrap();
            bootstrap.group(parentGroup,childGroup)
                    .channel(NioServerSocketChannel.class)
                    .childHandler(new ChannelInitializer<SocketChannel>() {
                        @Override
                        protected void initChannel(SocketChannel ch) throws
Exception {
                            ChannelPipeline pipeline = ch.pipeline();
//                            pipeline.addLast(new FixedLengthFrameDecoder(6));
                            pipeline.addLast(new
DelimiterBasedFrameDecoder(10240, Unpooled.copiedBuffer("####".getBytes())));

                            pipeline.addLast(new
StringDecoder(CharsetUtil.UTF_8));
                            pipeline.addLast(new ServerHandler3());

//                            pipeline.addLast(new ChatSocketSer)
                        }
                    });
            ChannelFuture future = bootstrap.bind(8888).sync();
            System.out.println("启动成功");
            future.channel().closeFuture().sync();
        }finally {
            parentGroup.shutdownGracefully();
            childGroup.shutdownGracefully();
        }
    }

}
```



epoll模型 select/poll、epoll

Selector实际上就是封装的底层epoll实现

```
private native int epollCreate();
private native void epollCtl(int epfd, int opcode, int fd, int events);
private native int epollWait(long pollAddress, int numfds, long timeout,int
epfd) throws IOException;
```

	select	poll	epoll(jdk 1.5及以上)
操作方式	遍历	遍历	回调
底层实现	数组	链表	哈希表
IO效率	每次调用都进行线性遍历，时间复杂度为O(n)	每次调用都进行线性遍历，时间复杂度为O(n)	事件通知方式，每当有IO事件就绪，系统注册的回调函数就会被调用，时间复杂度O(1)
最大连接	有上限1024	无上限	无上限

reactor是多路复用其中一种模式，epoll是reactor的一种实现

Buffer

重要的属性

- **容量(Capacity)**
- **上界(Limit)**
- **位置(Position)**
- **标记(Mark)**

重要的方法

- **flip:**

```
limit = position;
position = 0;
mark = -1;
```

- **mark:**
-
```
mark = position;
```

- **reset:**
-
```
position = mark;
```

- **clear:**
-
```
position = 0;
limit = capacity;
mark = -1;
```

零拷贝

linux支持的方式(关注下面两种方式):

- mmap内存映射：用户空间和内核空间的地址映射
- sendfile：

java实现

MappedByteBuffer

FileChannel.transferTo

DirectByteBuffer