NIO网络编程

JAVA NIO

1. 始于Java 1.4，提供了新的JAVA IO操作非阻塞API。来替代JAVA IO和JAVA Networking相关API。
2. NIO中有三个核心组件
    - Buffer缓冲区
    - Channel通道
    - Selector选择器

Buffer缓冲区

缓冲区 本质上是一个可以写入数据的内存块(类似数组)，然后可以再次读取。此内存块包含在NIO Buffer对象中，该对象提供了一组方法，可以更轻松地使用内存块。相比较直接对数组的操作，Buffer API更加容易操作和管理。

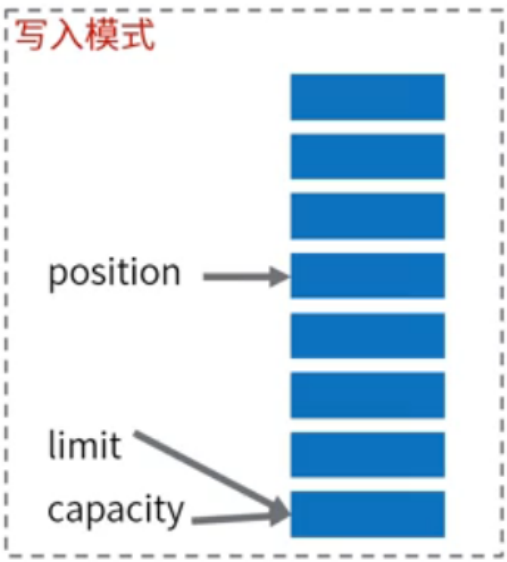
使用Buffer进行数据写入与读取，需要进行如下四个步骤：

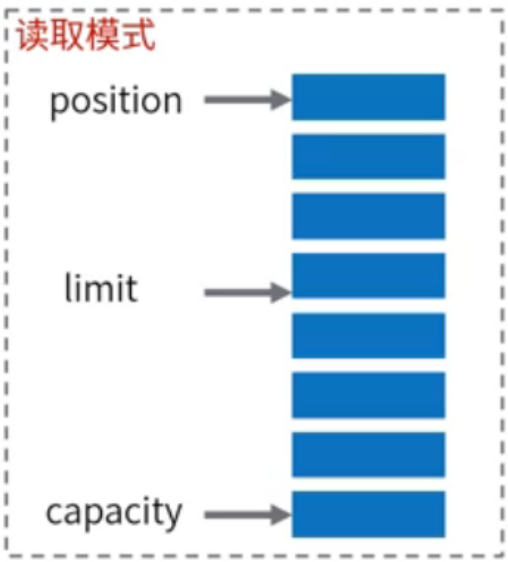
1. 将数据写入缓冲区。
2. 调用buffer.flip()，转换为读取模式。
3. 缓冲区读取数据。
4. 调用buffer.clear()或buffer.compact()清除缓冲区。

Buffer工作原理

Buffer三个重要属性：

1. capacity容量：作为一个内存块，Buffer具有一定的固定大小，也称为容量。
2. position位置：写入模式时代表写数据的位置。读取模式时代表读取数据的位置。
3. limit限制：写入模式，限制等于buffer的容量。读取模式下，limit等于写入的数据量。






```
public class BufferDemo {
    public static void main(String[] args) {
```

```
        ByteBuffer byteBuffer = ByteBuffer.allocate(4);
        System.out.println(String.format("初始化: capacity容量: %s, position位置: %s, limit限制: %s", byteBuffer.capacity(),
                byteBuffer.position(), byteBuffer.limit()));

        byteBuffer.put((byte) 1);
        byteBuffer.put((byte) 2);
        byteBuffer.put((byte) 5);
        System.out.println(String.format("初始化: capacity容量: %s, position位置: %s, limit限制: %s", byteBuffer.capacity(),
                byteBuffer.position(), byteBuffer.limit()));

        System.out.println("开始读取");
        // 转换为读取模式
        byteBuffer.flip();
        byte a = byteBuffer.get();
        System.out.println(a);
        byte b = byteBuffer.get();
        System.out.println(b);
        System.out.println(String.format("读取两个字节后: capacity容量: %s, position位置: %s, limit限制: %s", byteBuffer.capacity(),
                byteBuffer.position(), byteBuffer.limit()));

        // 继续写入3个字节, 此时读取模式下, limit=3, position=2继续写入只能覆盖写入一条数据
        // clear()方法清除整个缓冲区, compact()清除已经读取的数据, 转为写入模式
        byteBuffer.compact();
        byteBuffer.put((byte) 3);
        byteBuffer.put((byte) 4);
        byteBuffer.put((byte) 5);
        System.out.println(String.format("最终情况: capacity容量: %s, position位置: %s, limit限制: %s", byteBuffer.capacity(),
                byteBuffer.position(), byteBuffer.limit()));

        // rewind 重置position为0
        // mark() 标记position的位置
        // reset() 重置position为上次mark()标记的位置
    }
}


初始化: capacity容量: 4, position位置: 0, limit限制: 4
初始化: capacity容量: 4, position位置: 3, limit限制: 4
开始读取
1
2
读取两个字节后: capacity容量: 4, position位置: 2, limit限制: 3
最终情况: capacity容量: 4, position位置: 4, limit限制: 4
```

ByteBuffer内存类型

ByteBuffer为性能关键代码提供了**直接内存(direct堆外)**和**非直接内存(heap堆)**两种实现。堆外内存获取的方式：

ByteBuffer directBuffer = ByteBuffer.allocateDirect(noBytes);

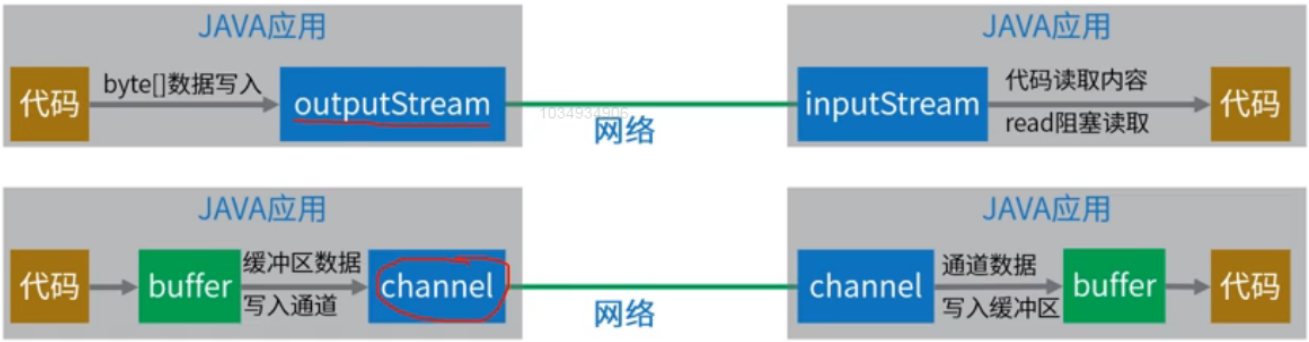
好处：

1. 进行网络IO或者文件IO时比heapBuffer少拷贝一次。(file/socket->OS memory->jvm heap)GC会移动对象内存，在写file或者socket的过程中，JVM的实现中，会先把数据复制到堆外，再进行写入。
2. GC范围之外，降低GC压力，但实现了自动管理。DirectByteBuffer中有一个Cleaner对象(PhantomReference)，Cleaner被GC前会执行clean方法，触发DirectByteBuffer中定义的Deallocator。

建议：

1. 性能确实可观的时候才去使用；分配给大型，长寿命；(网络传输，文件读写场景)。
2. 通过虚拟机参数MaxDirectMemorySize限制大小，防止耗尽整个机器内存。

Channel通道



1. Channel的API涵盖了UDP/TCP网络和文件IO。FileChannel，DatagramChannel，SocketChannel，ServerSocketChannel。
2. 和标准IO的区别：
    - 在一个通道内进行读取和写入，stream通常是单向的(input和output)。
    - 可以非阻塞读取和写入通道。
    - 通道始终读取或写入缓冲区。

SocketChannel

SocketChannel用于建立TCP网络连接，类似java.net.socket。有两种创建socketChannel形式：

1. 客户端主动发起和服务器的连接。
2. 服务端获取的新连接。

```
// 客户端主动发起连接的方式
SocketChannel socketChannel = SocketChannel.open () ;
socketChannel .configureBlocking( false ); // 设置为非阻塞模式
socketChannel.connect (new InetSocketAddress ( “http://163.com” , 80) ) ;

channel.write(byteBuffer); // 发送请求数据 – 向通道写入数据

int bytesRead = socketChannel.read ( byteBuffer ) ; // 读取服务端返回 – 读取缓冲区的数据

socketChannel.close (); // 关闭连接
```

wirte写：write()在尚未写入任何内容时就可能返回了。需要在循环中调用write()。

read读：read()方法可能直接返回而根本不读取任何数据，根据返回的int值判断读取了多少字节。

ServerSocketChannel

ServerSocketChannel可以监听新建的TCP连接通道，类似ServerSocket。

```
// 创建网络服务端
ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
serverSocketChannel.configureBlocking(false); // 设置为非阻塞模式
serverSocketChannel.socket().bind(new InetSocketAddress(8080)); // 绑定端口
while(true){
  SocketChannel socketChannel =  serverSocketChannel.accept(); // 获取新tcp连接通道
  if(socketChannel != null){
   // tcp请求 读取/响应
  }
}
```

serverSocketChannel.accept()：如果该通道处于非阻塞模式，那么如果没有挂起的连接，该方法将立即返回null。必须检查返回的SocketChannel是否为null。但是这种低效的循环检查，不是NIO服务端的正确使用方法。

```
public class NIOServer {
    public static void main(String[] args) throws IOException {
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        // 设置为非阻塞
        serverSocketChannel.configureBlocking(false);
        serverSocketChannel.bind(new InetSocketAddress(8080));
        System.out.println("启动成功");
        while (true) {
            SocketChannel socketChannel = serverSocketChannel.accept();
            if (null != socketChannel) {
                System.out.println("收到连接: " + socketChannel.getRemoteAddress());
                socketChannel.configureBlocking(false);
                try {
                    ByteBuffer requestBuffer = ByteBuffer.allocate(1024);
                    while (socketChannel.isOpen() && socketChannel.read(requestBuffer) !=
-1) {
                        // 长连接情况下，需要手动判断数据有没有读取结束
                        if (requestBuffer.position() > 0) {
                            break;
                        }
                    }

                    // 如果没数据了，则不继续后面的处理
                    if (requestBuffer.position() == 0) {
                        continue;
                    }
                    requestBuffer.flip();

                    byte[] content = new byte[requestBuffer.limit()];
                    requestBuffer.get(content);
                    System.out.println(new String(content));
                    System.out.println("收到数据，来自: " +
socketChannel.getRemoteAddress());

                    String response = "HTTP/1.1 200OK\r\n" +
```

```
                            "Content-Length: 11\r\n\r\n" +
                            "Hello World";
                    ByteBuffer byteBuffer = ByteBuffer.wrap(response.getBytes());
                    while (byteBuffer.hasRemaining()) {
                        // 非阻塞
                        socketChannel.write(byteBuffer);
                    }
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}

public class NIOClient {
    public static void main(String[] args) throws IOException {
        SocketChannel socketChannel = SocketChannel.open();
        socketChannel.configureBlocking(false);
        socketChannel.connect(new InetSocketAddress("127.0.0.1", 8080));
        while (!socketChannel.finishConnect()) {
            // 没连接上一直等待
            Thread.yield();
        }
        Scanner scanner = new Scanner(System.in);
        System.out.println("请输入: ");
        String message = scanner.nextLine();
        ByteBuffer byteBuffer = ByteBuffer.wrap(message.getBytes());
        while (byteBuffer.hasRemaining()) {
            socketChannel.write(byteBuffer);
        }

        System.out.println("收到服务端响应");
        ByteBuffer responseBuffer = ByteBuffer.allocate(1024);

        while (socketChannel.isOpen() && socketChannel.read(responseBuffer) != -1) {
            if (responseBuffer.position() > 0) {
                break;
            }
        }
        responseBuffer.flip();
        byte[] content = new byte[responseBuffer.limit()];
        responseBuffer.get(content);
        System.out.println(new String(content));
        scanner.close();
        socketChannel.close();
    }
}
```

改进后的服务端写法

```
public class NIOServer1 {
```

```
    /**
     * 已经建立连接的集合
     */
    private static ArrayList<SocketChannel> channels = new ArrayList<>();

    public static void main(String[] args) throws Exception {
        // 创建网络服务端
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.configureBlocking(false); // 设置为非阻塞模式
        serverSocketChannel.socket().bind(new InetSocketAddress(8080)); // 绑定端口
        System.out.println("启动成功");
        while (true) {
            SocketChannel socketChannel = serverSocketChannel.accept(); // 获取新tcp连接通道
            // tcp请求 读取/响应
            if (socketChannel != null) {
                System.out.println("收到新连接 : " + socketChannel.getRemoteAddress());
                socketChannel.configureBlocking(false); // 默认是阻塞的,一定要设置为非阻塞
                channels.add(socketChannel);
            } else {
                // 没有新连接的情况下,就去处理现有连接的数据,处理完的就删除掉
                Iterator<SocketChannel> iterator = channels.iterator();
                while (iterator.hasNext()) {
                    SocketChannel ch = iterator.next();
                    try {
                        ByteBuffer requestBuffer = ByteBuffer.allocate(1024);

                        if (ch.read(requestBuffer) == 0) {
                            // 等于0,代表这个通道没有数据需要处理,那就待会再处理
                            continue;
                        }
                        while (ch.isOpen() && ch.read(requestBuffer) != -1) {
                            // 长连接情况下,需要手动判断数据有没有读取结束 (此处做一个简单的判断: 超过
0字节就认为请求结束了)
                            if (requestBuffer.position() > 0) break;
                        }
                        if(requestBuffer.position() == 0) continue; // 如果没数据了, 则不继续
后面的处理
                        requestBuffer.flip();
                        byte[] content = new byte[requestBuffer.limit()];
                        requestBuffer.get(content);
                        System.out.println(new String(content));
                        System.out.println("收到数据,来自：" + ch.getRemoteAddress());

                        // 响应结果 200
                        String response = "HTTP/1.1 200 OK\r\n" +
                                "Content-Length: 11\r\n\r\n" +
                                "Hello World";
                        ByteBuffer buffer = ByteBuffer.wrap(response.getBytes());
                        while (buffer.hasRemaining()) {
                            ch.write(buffer);
                        }
                        iterator.remove();
                    } catch (IOException e) {
```

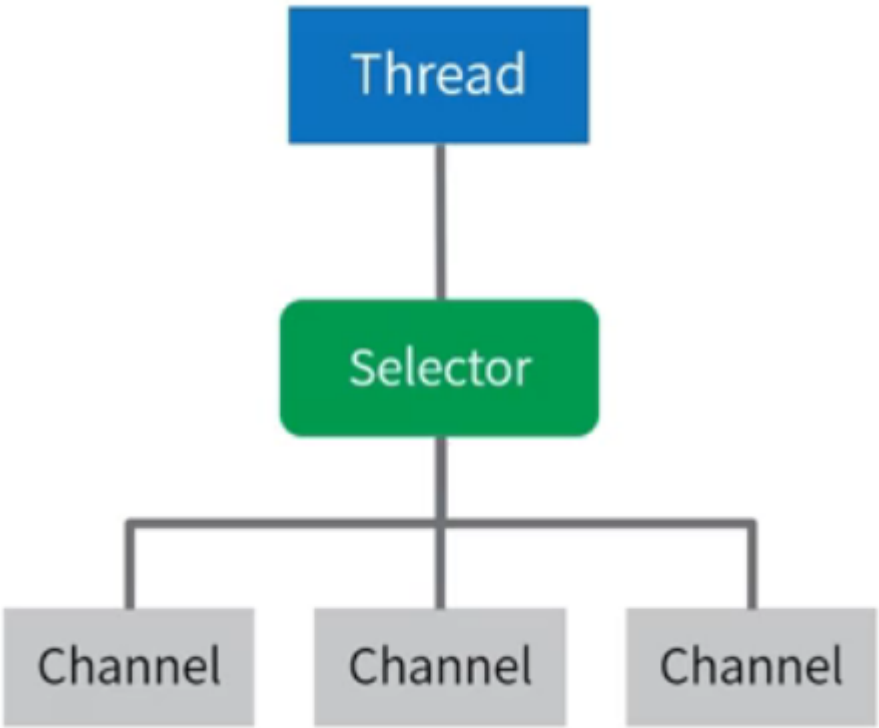
```
                        e.printStackTrace();
                        iterator.remove();
                    }
                }
            }
        }
        // 用到了非阻塞的API，再设计上,和BIO可以有很大的不同
        // 问题：轮询通道的方式,低效,浪费CPU
    }
}
```

Select选择器

Seclector是一个Java NIO组件，可以检查一个或多个NIO通道，并确定哪些通道已经准备好进行读取或写入。实现单个线程可以管理多个通道，从而管理多个网络连接。

一个线程使用Selector监听过个channel的不同事件：

1. Connect连接(SelectionKey.OP_CONNECT)。
2. Accept准备就绪(OP_ACCEPT)。
3. Read读取(OP_READ)。
4. Write写入(OP_WRITE)。



实现一个线程处理多个通道的核心概念：**事件驱动机制**。

非阻塞的网络通道下，开发者通过Selector注册对通道感兴趣的事件类型，线程通过监听事件来触发相应的代码执行。(更底层时操作系统的多路复用机制)。

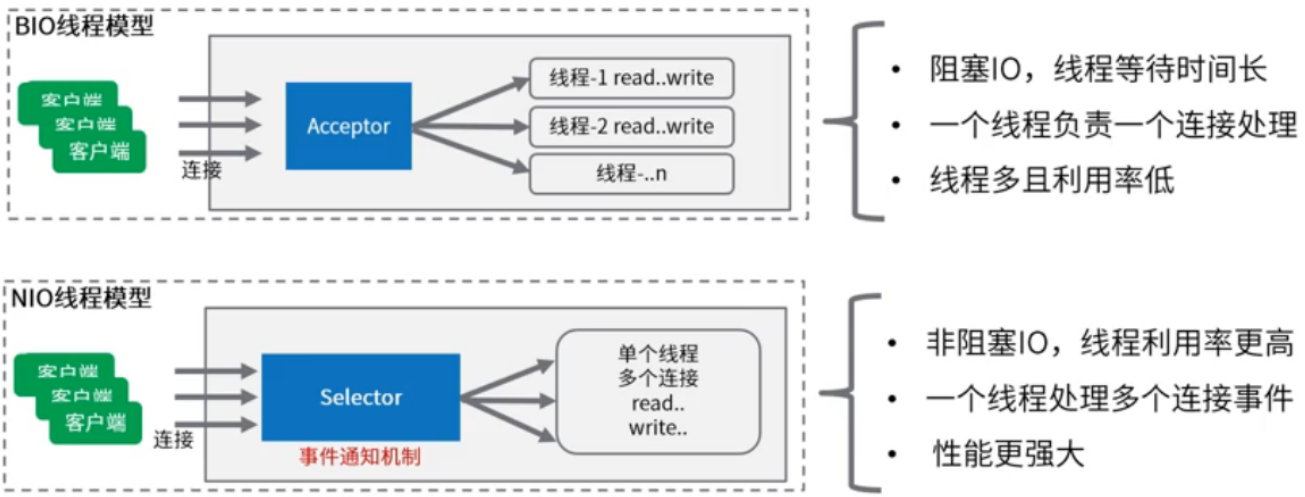
```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ); //注册感兴趣的事件
while(true) { // 由accept轮询，变成了事件通知的方式。
 int readyChannels = selector.select(); // select 收到新的事件，方法才会返回
 if(readyChannels == 0) continue;
 Set<SelectionKey> selectedKeys = selector.selectedKeys();
 Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
 while(keyIterator.hasNext()) {
  SelectionKey key = keyIterator.next();
  // 判断不同的事件类型，执行对应的逻辑处理
  // key.isAcceptable()) / key.isConnectable() / key.isReadable() /  key.isWritable()}
  keyIterator.remove();
 }
}
```

再次改写

```
public class NIOServer2 {
    public static void main(String[] args) throws IOException {
        ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.configureBlocking(false);

        Selector selector = Selector.open();
        // 将serverSocketChannel注册到selector上
        SelectionKey selectionKey = serverSocketChannel.register(selector, 0,
serverSocketChannel);
        // 对serverSocketChannel上的accept事件感兴趣
        selectionKey.interestOps(SelectionKey.OP_ACCEPT);

        serverSocketChannel.socket().bind(new InetSocketAddress(8080));

        System.out.println("启动成功");

        while (true) {
            // 不在轮询通道,改用下面轮询事件的方式,select方法有阻塞效果,直到有事件通知才会有返回
            selector.select();
            // 获取事件
            Set<SelectionKey> selectionKeys = selector.selectedKeys();
            Iterator<SelectionKey> iterator = selectionKeys.iterator();
            while (iterator.hasNext()) {
                SelectionKey key = iterator.next();
                iterator.remove();
                if (key.isAcceptable()) {
                    SocketChannel socketChannel = serverSocketChannel.accept();
                    socketChannel.configureBlocking(false);
                    socketChannel.register(selector, SelectionKey.OP_READ, socketChannel);
                    System.out.println("收到新连接 : " + socketChannel.getRemoteAddress());
                }

                if (key.isReadable()) {
                    SocketChannel ch = (SocketChannel) key.attachment();
                    try {
```

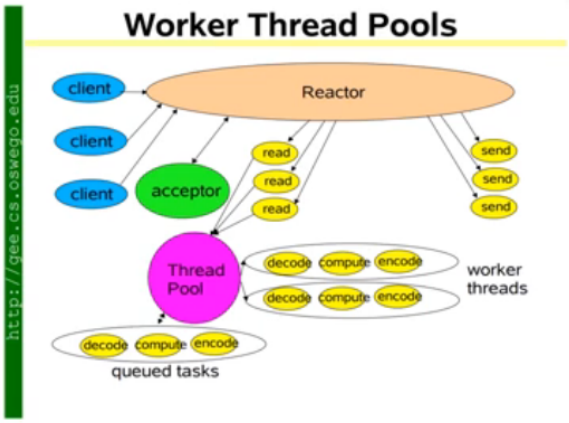
```
                        ByteBuffer requestBuffer = ByteBuffer.allocate(1024);

                        if (ch.read(requestBuffer) == 0) {
                            // 等于0,代表这个通道没有数据需要处理,那就待会再处理
                            continue;
                        }
                        while (ch.isOpen() && ch.read(requestBuffer) != -1) {
                            // 长连接情况下,需要手动判断数据有没有读取结束 (此处做一个简单的判断: 超过
0字节就认为请求结束了)

                            if (requestBuffer.position() > 0) break;
                        }
                        if (requestBuffer.position() == 0) continue; // 如果没数据了, 则不继续
后面的处理

                        requestBuffer.flip();
                        byte[] content = new byte[requestBuffer.limit()];
                        requestBuffer.get(content);
                        System.out.println(new String(content));
                        System.out.println("收到数据,来自：" + ch.getRemoteAddress());

                        // 响应结果 200
                        String response = "HTTP/1.1 200 OK\r\n" +
                                "Content-Length: 11\r\n\r\n" +
                                "Hello World";
                        ByteBuffer buffer = ByteBuffer.wrap(response.getBytes());
                        while (buffer.hasRemaining()) {
                            ch.write(buffer);
                        }
                    } catch (IOException e) {
                        key.cancel();
                    }
                }
            }
        }
    }
}
```

NIO对比BIO

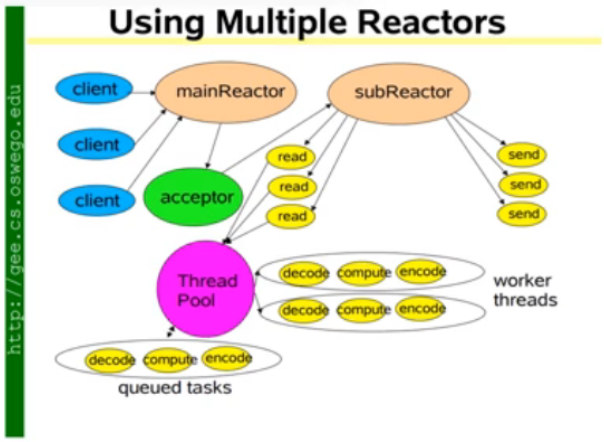


如果你的程序需要支撑大量的连接，使用NIO是最好的方式。Tomcat8中，已经完全去除BIO相关的网络处理代码，默认采用NIO进行网络处理。

NIO与多线程结合的改进方案



Reactor线程接收请求 -> 分发给线程池处理请求



mainReactor接收 -> 分发给subReactor读写 -> 具体业务逻辑分发给单独的线程池处理

最终服务端如下

```
public class NIOServerV3 {

    /** 处理业务操作的线程 */
    private static ExecutorService workPool = Executors.newCachedThreadPool();

    /**
     * 封装了selector.select()等事件轮询的代码
     */
    abstract class ReactorThread extends Thread {

        Selector selector;
        LinkedBlockingQueue<Runnable> taskQueue = new LinkedBlockingQueue<>();

        /**
         * Selector监听到有事件后,调用这个方法
         */
        public abstract void handler(SelectableChannel channel) throws Exception;

        private ReactorThread() throws IOException {
            selector = Selector.open();
        }

        volatile boolean running = false;

        @Override
        public void run() {
            // 轮询Selector事件
            while (running) {
                try {
                    // 执行队列中的任务
```

```
                    Runnable task;
                    while ((task = taskQueue.poll()) != null) {
                        task.run();
                    }
                    selector.select(1000);

                    // 获取查询结果
                    Set<SelectionKey> selected = selector.selectedKeys();
                    // 遍历查询结果
                    Iterator<SelectionKey> iter = selected.iterator();
                    while (iter.hasNext()) {
                        // 被封装的查询结果
                        SelectionKey key = iter.next();
                        iter.remove();
                        int readyOps = key.readyOps();
                        // 关注 Read 和 Accept两个事件
                        if ((readyOps & (SelectionKey.OP_READ | SelectionKey.OP_ACCEPT)) !=
0 || readyOps == 0) {
                            try {
                                SelectableChannel channel = (SelectableChannel)
key.attachment();
                                channel.configureBlocking(false);
                                handler(channel);
                                if (!channel.isOpen()) {
                                    key.cancel(); // 如果关闭了,就取消这个KEY的订阅
                                }
                            } catch (Exception ex) {
                                key.cancel(); // 如果有异常,就取消这个KEY的订阅
                            }
                        }
                    }
                    selector.selectNow();
                } catch (IOException e) {
                    e.printStackTrace();
                }
            }
        }

        private SelectionKey register(SelectableChannel channel) throws Exception {
            // 为什么register要以任务提交的形式，让reactor线程去处理？
            // 因为线程在执行channel注册到selector的过程中，会和调用selector.select()方法的线程争用
同一把锁
            // 而select()方法是在eventLoop中通过while循环调用的，争抢的可能性很高，为了让register能
更快的执行，就放到同一个线程来处理
            FutureTask<SelectionKey> futureTask = new FutureTask<>(() ->
channel.register(selector, 0, channel));
            taskQueue.add(futureTask);
            return futureTask.get();
        }

        private void doStart() {
            if (!running) {
                running = true;
```

```
                start();
            }
        }
    }

    private ServerSocketChannel serverSocketChannel;
    // 1、创建多个线程 - accept处理reactor线程 (accept线程)
    private ReactorThread[] mainReactorThreads = new ReactorThread[1];
    // 2、创建多个线程 - io处理reactor线程  (I/O线程)
    private ReactorThread[] subReactorThreads = new ReactorThread[8];

    /**
     * 初始化线程组
     */
    private void newGroup() throws IOException {
        // 创建IO线程,负责处理客户端连接以后socketChannel的IO读写
        for (int i = 0; i < subReactorThreads.length; i++) {
            subReactorThreads[i] = new ReactorThread() {
                @Override
                public void handler(SelectableChannel channel) throws IOException {
                    // work线程只负责处理IO处理，不处理accept事件
                    SocketChannel ch = (SocketChannel) channel;
                    ByteBuffer requestBuffer = ByteBuffer.allocate(1024);
                    while (ch.isOpen() && ch.read(requestBuffer) != -1) {
                        // 长连接情况下,需要手动判断数据有没有读取结束 (此处做一个简单的判断: 超过0字节
就认为请求结束了)
                        if (requestBuffer.position() > 0) break;
                    }
                    if (requestBuffer.position() == 0) return; // 如果没数据了, 则不继续后面的
处理
                    requestBuffer.flip();
                    byte[] content = new byte[requestBuffer.limit()];
                    requestBuffer.get(content);
                    System.out.println(new String(content));
                    System.out.println(Thread.currentThread().getName() + "收到数据,来自：" +
ch.getRemoteAddress());

                    // TODO 业务操作 数据库、接口...
                    workPool.submit(() -> {
                    });

                    // 响应结果 200
                    String response = "HTTP/1.1 200 OK\r\n" +
                            "Content-Length: 11\r\n\r\n" +
                            "Hello World";
                    ByteBuffer buffer = ByteBuffer.wrap(response.getBytes());
                    while (buffer.hasRemaining()) {
                        ch.write(buffer);
                    }
                }
            };
        }
```

```
        // 创建mainReactor线程，只负责处理serverSocketChannel
        for (int i = 0; i < mainReactorThreads.length; i++) {
            mainReactorThreads[i] = new ReactorThread() {
                AtomicInteger incr = new AtomicInteger(0);

                @Override
                public void handler(SelectableChannel channel) throws Exception {
                    // 只做请求分发，不做具体的数据读取
                    ServerSocketChannel ch = (ServerSocketChannel) channel;
                    SocketChannel socketChannel = ch.accept();
                    socketChannel.configureBlocking(false);
                    // 收到连接建立的通知之后，分发给I/O线程继续去读取数据
                    int index = incr.getAndIncrement() % subReactorThreads.length;
                    ReactorThread workEventLoop = subReactorThreads[index];
                    workEventLoop.doStart();
                    SelectionKey selectionKey = workEventLoop.register(socketChannel);
                    selectionKey.interestOps(SelectionKey.OP_READ);
                    System.out.println(Thread.currentThread().getName() + "收到新连接 : " +
socketChannel.getRemoteAddress());
                }
            };
        }


    }

    /**
     * 初始化channel,并且绑定一个eventLoop线程
     *
     * @throws IOException IO异常
     */
    private void initAndRegister() throws Exception {
        // 1、 创建ServerSocketChannel
        serverSocketChannel = ServerSocketChannel.open();
        serverSocketChannel.configureBlocking(false);
        // 2、 将serverSocketChannel注册到selector
        int index = new Random().nextInt(mainReactorThreads.length);
        mainReactorThreads[index].doStart();
        SelectionKey selectionKey =
mainReactorThreads[index].register(serverSocketChannel);
        selectionKey.interestOps(SelectionKey.OP_ACCEPT);
    }

    /**
     * 绑定端口
     *
     * @throws IOException IO异常
     */
    private void bind() throws IOException {
        //  1、 正式绑定端口，对外服务
        serverSocketChannel.bind(new InetSocketAddress(8080));
        System.out.println("启动完成，端口8080");
    }
```

```
    public static void main(String[] args) throws Exception {
        NIOServerV3 nioServerV3 = new NIOServerV3();
        nioServerV3.newGroup(); // 1、 创建main和sub两组线程
        nioServerV3.initAndRegister(); // 2、 创建serverSocketChannel，注册到mainReactor线程上
的selector上
        nioServerV3.bind(); // 3、 为serverSocketChannel绑定端口
    }
}
```