

- 3.2.1 handleKey
- 3.3.2 handleConnectableKey
- 3.3.3 handleReadableKey
- 3.3.4 handleWritableKey
- 3.3 send
- 3.4 main
- 666. 彩蛋

```
32:         Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
33:         while (iterator.hasNext()) {
34:             SelectionKey key = iterator.next();
35:             iterator.remove(); // 移除下面要处理的 SelectionKey
36:             if (!key.isValid()) { // 忽略无效的 SelectionKey
37:                 continue;
38:             }
39:
40:             handleKey(key);
41:         }
42:     }
43: }
44:
45: private void handleKey(SelectionKey key) throws IOException {
46:     // 接受连接就绪
47:     if (key.isAcceptable()) {
48:         handleAcceptableKey(key);
49:     }
50:     // 读就绪
51:     if (key.isReadable()) {
52:         handleReadableKey(key);
53:     }
54:     // 写就绪
55:     if (key.isWritable()) {
56:         handleWritableKey(key);
57:     }
58: }
59:
60: private void handleAcceptableKey(SelectionKey key) throws IOException {
61:     // 接受 Client Socket Channel
62:     SocketChannel clientSocketChannel = ((ServerSocketChannel) key.channel()).accept();
63:     // 配置为非阻塞
64:     clientSocketChannel.configureBlocking(false);
65:     // log
```

or.select(30 * 1000L);

译 Channel 数量: " + selectNums);

的 SelectionKey 集合

```
    " Channel");
    el 到 Selector
    r(selector, SelectionKey.OP_READ, new ArrayList<String>());

    SelectionKey key) throws IOException {

    hannel = (SocketChannel) key.channel();

    ecUtil.read(clientSocketChannel);

    F Channel");
```

文章目录

- 1. 概述
- 2. 服务端
 - 2.1 构造方法
 - 2.2 handleKeys
 - 2.2.1 handleKey
 - 2.2.2 handleAcceptableKey
 - 2.2.3 handleReadableKey
 - 2.2.4 handleWritableKey
 - 2.3 main
- 3. 客户端
 - 3.1 构造方法
 - 3.2 handleKeys

[3.2.1 handleKey](#)
[3.3.2 handleConnectableKey](#)
[3.3.3 handleReadableKey](#)
[3.3.4 handleWritableKey](#)
[3.3 send](#)
[3.4 main](#)
[666. 彩蛋](#)

```

        ister(selector, 0);

        0) {
            til.newString(readBuffer);
            数据: " + content);
    
```

```

86:
87:         // 添加到响应队列
88:         List<String> responseQueue = (ArrayList<String>) key.attachment();
89:         responseQueue.add("响应: " + content);
90:         // 注册 Client Socket Channel 到 Selector
91:         clientSocketChannel.register(selector, SelectionKey.OP_WRITE, key.attachment());
92:     }
93: }
94:
95: @SuppressWarnings("Duplicates")
96: private void handleWritableKey(SelectionKey key) throws ClosedChannelException {
97:     // Client Socket Channel
98:     SocketChannel clientSocketChannel = (SocketChannel) key.channel();
99:
100:    // 遍历响应队列
101:    List<String> responseQueue = (ArrayList<String>) key.attachment();
102:    for (String content : responseQueue) {
103:        // 打印数据
104:        System.out.println("写入数据: " + content);
105:        // 返回
106:        CodecUtil.write(clientSocketChannel, content);
107:    }
108:    responseQueue.clear();
109:
110:    // 注册 Client Socket Channel 到 Selector
111:    clientSocketChannel.register(selector, SelectionKey.OP_READ, responseQueue);
112: }
113:
114: public static void main(String[] args) throws IOException {
115:     NioServer server = new NioServer();
116: }
117:
118: }
    
```

文章目录

1. 概述
2. 服务端
 - 2.1 构造方法
 - 2.2 handleKeys
 - 2.2.1 handleKey
 - 2.2.2 handleAcceptableKey
 - 2.2.3 handleReadableKey
 - 2.2.4 handleWritableKey
 - 2.3 main
3. 客户端
 - 3.1 构造方法
 - 3.2 handleKeys

- 3.2.1 [handleKey](#)
- 3.3.2 [handleConnectableKey](#)
- 3.3.3 [handleReadableKey](#)
- 3.3.4 [handleWritableKey](#)
- 3.3 [send](#)
- 3.4 [main](#)
- 666. [彩蛋](#)

- `serverSocketChannel`，在【第 7 至 12 行】的代码进行初始化，重点是此
- `selector` 属性，选择器，在【第 14 至 18 行】的代码进行初始化，重点是此处将 `serverSocketChannel` 到 `selector` 中，并对 `SelectionKey.OP_ACCEPT` 事件感兴趣。这样子，在客户端连接服务端时，我们就可以处理该 IO 事件。
- 第 19 行：调用 `#handleKeys()` 方法，基于 `Selector` 处理 IO 事件。

2.2 `handleKeys`

对应【第 22 至 43 行】的代码。

- 第 23 行：死循环。本文的示例，不考虑服务端关闭的逻辑。
- 第 24 至 29 行：调用 `Selector#select(long timeout)` 方法，每 30 秒阻塞等待有就绪的 IO 事件。此处的 30 秒为笔者随意写的，实际也可以改成其他超时时间，或者 `Selector#select()` 方法。当不存在就绪的 IO 事件，直接 `continue`，继续下一次阻塞等待。
- 第 32 行：调用 `Selector#selectedKeys()` 方法，获得有就绪的 IO 事件(也可以称为“选择的”) `Channel` 对应的 `SelectionKey` 集合。
 - 第 33 行至 35 行：遍历 `iterator`，进行逐个 `SelectionKey` 处理。重点注意下，处理完需要进行移除，具体原因，在《[精尽 Netty 源码分析 —— NIO 基础 \(四\) 之 Selector](#)》[10. 简单 `Selector` 示例] 有详细解析。
 - 第 36 至 38 行：在遍历的过程中，可能该 `SelectionKey` 已经失效，直接 `continue`，不进行处理。
 - 第 40 行：调用 `#handleKey()` 方法，逐个 `SelectionKey` 处理。

2.2.1 `handleKey`

对应【第 45 至 58 行】的代码。

- 通过调用 `SelectionKey` 的 `#isAcceptable()`、`#isReadable()`、`#isWritable()` 方法，**分别**判断 `Channel` 是接受连接就绪，还是读就绪，或是写就绪，并调用相应的 `#handleXXXX(SelectionKey key)` 方法，处理对应的 IO 事件。

文章目录

- 1. [概述](#)
- 2. [服务端](#)
 - 2.1 [构造方法](#)
 - 2.2 [handleKeys](#)
 - 2.2.1 [handleKey](#)
 - 2.2.2 [handleAcceptableKey](#)
 - 2.2.3 [handleReadableKey](#)
 - 2.2.4 [handleWritableKey](#)
 - 2.3 [main](#)
- 3. [客户端](#)
 - 3.1 [构造方法](#)
 - 3.2 [handleKeys](#)

多个事件感兴趣，所以此处的代码都是 `if` 判断，而不是 `if else` 并未编写同时对于一个 `Channel` 的多个事件感兴趣，后续我们会在 `Netty`，所以此处不需要做相应的判断和处理。

。

`opt()` 方法，获得连接的客户端的 `SocketChannel`。

3.2.1 handleKey
 3.3.2 handleConnectableKey
 3.3.3 handleReadableKey
 3.3.4 handleWritableKey
 3.3 send
 3.4 main
 666. 彩蛋

阻塞，否则无法使用 Selector。

使用 Logger 而不要使用 System.out 进行输出。

Selector 中，并对 SelectionKey.OP_READ 事件感兴趣。这样子，在
 处理该 IO 事件。

件感兴趣呢？因为这个时候，服务端一般不会主动向客户端发送消息，
 事件感兴趣。

selector selector, int ops, Object attachment) 方法的第 3

1 参数，我们注册了 SelectionKey 的 attachment 属性为 new ArrayList<String>()，这又是为什么呢？

结合下面的 #handleReadableKey(Selection key) 方法，我们一起解析。

2.2.3 handleReadableKey

对应【第 71 至 93 行】的代码。

- 第 73 行：调用 SelectionKey#channel() 方法，获得该 SelectionKey 对应的 SocketChannel，即客户端的 SocketChannel。
- 第 75 行：调用 CodecUtil#read(SocketChannel channel) 方法，读取数据。具体代码如下：

```
// CodecUtil.java

public static ByteBuffer read(SocketChannel channel) {
    // 注意，不考虑拆包的处理
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    try {
        int count = channel.read(buffer);
        if (count == -1) {
            return null;
        }
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    return buffer;
}
```

- 考虑到示例的简单性，数据的读取，就不考虑拆包的处理。不理解的朋友，可以自己 Google 下。
- 调用 SocketChannel#read(ByteBuffer) 方法，读取 Channel 的缓冲区的数据到 ByteBuffer 中。若返回的结果(count)为 -1，意味着客户端连接已经断开，我们直接返回 null。为什么是返回 null 呢？下面继续

四公晓

文章目录

1. 概述
 2. 服务端
 2.1 构造方法
 2.2 handleKeys
 2.2.1 handleKey
 2.2.2 handleAcceptableKey
 2.2.3 handleReadableKey
 2.2.4 handleWritableKey
 2.3 main
 3. 客户端
 3.1 构造方法
 3.2 handleKeys

1 时，意味着客户端的连接已经断开，因此取消注册 selector 对该
 通过调用注册方法，并且第 2 个参数 ops 为 0，可以达到取消注册的
 注释，测试下效果就很容易明白了。

ng(ByteBuffer) 方法，格式化为字符串，并进行打印。代码如下：

```
ByteBuffer buffer) {
    remaining();
```

3.2.1 handleKey
3.3.2 handleConnectableKey
3.3.3 handleReadableKey
3.3.4 handleWritableKey
3.3 send
3.4 main
666. 彩蛋

```
    , buffer.position(), bytes, 0, buffer.remaining());

    "TF-8");

    eption e) {
        e);
```

- 注意，需要调用 `ByteBuffer#flip()` 方法，将 `ByteBuffer` 从**写**模式切换到**读**模式。
- 第 86 行：一般在此处，我们可以进行一些业务逻辑的处理，并返回处理的相应结果。例如，我们熟悉的 `Request / Response` 的处理。当然，考虑到性能，我们甚至可以将逻辑的处理，丢到逻辑线程池。
- 🤡 如果不理解，木有关系，在《[精尽 Dubbo 源码分析 —— NIO 服务器（二）之 Transport 层](#)》[8. [Dispatcher](#)] 中，有详细解析。
- 😊 考虑到示例的简洁性，所以在【第 88 至 89 行】的代码中，我们直接返回（"响应：" + 请求内容）给客户端。
- 第 88 行：通过调用 `SelectionKey#attachment()` 方法，获得我们**附加**在 `SelectionKey` 的响应队列(`responseQueue`)。可能有胖友会问啦，为什么不调用 `SocketChannel#write(ByteBuf)` 方法，直接写数据给客户端呢？虽然大多数情况下，`SocketChannel` 都是**可写**的，但是如果写入比较频繁，超过 `SocketChannel` 的缓存区大小，就会导致数据“**丢失**”，并未写给客户端。
- 所以，此处笔者在示例中，处理的方式为添加响应数据到 `responseQueue` 中，并在【第 91 行】的代码中，注册客户端的 `SocketChannel` 到 `selector` 中，并对 `SelectionKey.OP_WRITE` 事件感兴趣。这样子，在 `SocketChannel` **写就绪**时，在 `#handleWritableKey(SelectionKey key)` 方法中，统一处理写数据给客户端。
- 当然，还是因为是示例，所以这样的实现方式不是最优。在 `Netty` 中，具体的实现方式是，先尝试调用 `SocketChannel#write(ByteBuf)` 方法，写数据给客户端。若写入失败(方法返回结果为 0)时，再进行类似笔者的上述实现方式。牛逼！`Netty`！
- 如果不太理解分享的原因，可以再阅读如下两篇文章：
 - 《[深夜对话：NIO 中 SelectionKey.OP_WRITE 你了解多少](#)》
 - 《[Java.nio 中 socketChannel.write\(\) 返回 0 的简易解决方案](#)》
- 第 91 行：有一点需要注意，`Channel#register(Selector selector, int ops, Object attachment)` 方法的第 3 个参数，需要继续传入响应队列(`responseQueue`)，因为每次注册生成**新的** `SelectionKey`。若不传入，下面的 `#handleWritableKey(SelectionKey key)` 方法，会获得不到响应队列(`responseQueue`)。

2.2.4 handleWritableKey

对应【第 96 至 112 行】的代码。

第 96 行：调用 `SelectionKey#attachment()` 方法，获得该 `SelectionKey` 对应的 `SocketChannel`，即客户端的

文章目录

1. 概述
2. 服务端
 - 2.1 构造方法
 - 2.2 handleKeys
 - 2.2.1 handleKey
 - 2.2.2 handleAcceptableKey
 - 2.2.3 handleReadableKey
 - 2.2.4 handleWritableKey
 - 2.3 main
3. 客户端
 - 3.1 构造方法
 - 3.2 handleKeys

`ment()` 方法，获得我们**附加**在 `SelectionKey` 的响应队列(

`Channel, content)` 方法，写入响应数据给客户端。代码如下：

```
nel channel, String content) {
    allocate(1024);
```

3.2.1 handleKey

3.3.2 handleConnectableKey

3.3.3 handleReadableKey


3.3.4 handleWritableKey

3.3 send

3.4 main

666. 彩蛋

```
try {
    // 注意，不考虑写入超过 Channel 缓存区上限。
    channel.write(buffer);
} catch (IOException e) {
    throw new RuntimeException(e);
}
```

- 代码比较简单，**还是要注意**，需要调用 `ByteBuffer#flip()` 方法，将 `ByteBuffer` 从**写**模式切换到**读**模式。
- 第 111 行：**注意**，再结束写入后，需要**重新**注册客户端的 `SocketChannel` 到 `selector` 中，并对 `SelectionKey.OP_READ` 事件感兴趣。为什么呢？其实还是我们在上文中提到的，大多数情况下，`SocketChannel` **都是写就绪的**，如果不取消掉注册掉对 `SelectionKey.OP_READ` 事件感兴趣，就会导致反复触发无用的写事件处理。
 感兴趣的胖友，可以将这行代码进行注释，测试下效果就很容易明白了。

2.3 main

对应【第 114 至 116 行】

- 比较简单，就是创建一个 `NioServer` 对象。

撸到此处，我们可以直接通过 `telnet 127.0.0.1 8080` 的方式，连接服务端，进行读写数据的测试。

3. 客户端

客户端的实现代码，绝大多数和服务端相同，所以我们分析的相对会简略一些。不然，自己都嫌弃自己太啰嗦了。

```
1: public class NioClient {
2:
```

文章目录

1. 概述

2. 服务端

2.1 构造方法

2.2 handleKeys

2.2.1 handleKey

2.2.2 handleAcceptableKey

2.2.3 handleReadableKey

2.2.4 handleWritableKey

2.3 main

3. 客户端

3.1 构造方法

3.2 handleKeys

```
ketChannel;

onseQueue = new ArrayList<String>();

d = new CountdownLatch(1);

eption, InterruptedException {
el
tChannel.open();

reBlocking(false);
```

[3.2.1 handleKey](#)
[3.3.2 handleConnectableKey](#)
[3.3.3 handleReadableKey](#)
[3.3.4 handleWritableKey](#)
[3.3 send](#)
[3.4 main](#)
[666. 彩蛋](#)

el 到 Selector
 r(selector, SelectionKey.OP_CONNECT);
 (new InetSocketAddress(8080));

```

23:         public void run() {
24:             try {
25:                 handleKeys();
26:             } catch (IOException e) {
27:                 e.printStackTrace();
28:             }
29:         }
30:     }).start();
31:
32:     if (connected.getCount() != 0) {
33:         connected.await();
34:     }
35:     System.out.println("Client 启动完成");
36: }
37:
38: @SuppressWarnings("Duplications")
39: private void handleKeys() throws IOException {
40:     while (true) {
41:         // 通过 Selector 选择 Channel
42:         int selectNums = selector.select(30 * 1000L);
43:         if (selectNums == 0) {
44:             continue;
45:         }
46:
47:         // 遍历可选择的 Channel 的 SelectionKey 集合
48:         Iterator<SelectionKey> iterator = selector.selectedKeys().iterator();
49:         while (iterator.hasNext()) {
50:             SelectionKey key = iterator.next();
51:             iterator.remove(); // 移除下面要处理的 SelectionKey
52:             if (!key.isValid()) { // 忽略无效的 SelectionKey
53:                 continue;
54:             }
55:
56:             handleKey(key);

```

文章目录

[1. 概述](#)
[2. 服务端](#)
 [2.1 构造方法](#)
 [2.2 handleKeys](#)
 [2.2.1 handleKey](#)
 [2.2.2 handleAcceptableKey](#)
 [2.2.3 handleReadableKey](#)
 [2.2.4 handleWritableKey](#)
 [2.3 main](#)
[3. 客户端](#)
 [3.1 构造方法](#)
 [3.2 handleKeys](#)

eKey(SelectionKey key) throws IOException {

y);

- 3.2.1 handleKey
- 3.3.2 handleConnectableKey
- 3.3.3 handleReadableKey
- 3.3.4 handleWritableKey
- 3.3 send
- 3.4 main
- 666. 彩蛋

```
key(SelectionKey key) throws IOException {  
77: // 完成连接  
78: if (!clientSocketChannel.isConnectionPending()) {  
79:     return;  
80: }  
81: clientSocketChannel.finishConnect();  
82: // log  
83: System.out.println("接受新的 Channel");  
84: // 注册 Client Socket Channel 到 Selector  
85: clientSocketChannel.register(selector, SelectionKey.OP_READ, responseQueue);  
86: // 标记为已连接  
87: connected.countDown();  
88: }  
89:  
90: @SuppressWarnings("Duplications")  
91: private void handleReadableKey(SelectionKey key) throws ClosedChannelException {  
92:     // Client Socket Channel  
93:     SocketChannel clientSocketChannel = (SocketChannel) key.channel();  
94:     // 读取数据  
95:     ByteBuffer readBuffer = CodecUtil.read(clientSocketChannel);  
96:     // 打印数据  
97:     if (readBuffer.position() > 0) { // 写入模式下,  
98:         String content = CodecUtil.newString(readBuffer);  
99:         System.out.println("读取数据: " + content);  
100:    }  
101: }  
102:  
103: @SuppressWarnings("Duplications")  
104: private void handleWritableKey(SelectionKey key) throws ClosedChannelException {  
105:     // Client Socket Channel  
106:     SocketChannel clientSocketChannel = (SocketChannel) key.channel();  
107:  
108:     // 遍历响应队列  
109:     List<String> responseQueue = (ArrayList<String>) key.attachment();  
110:     for (String content : responseQueue) {
```

文章目录

- 1. 概述
- 2. 服务端
 - 2.1 构造方法
 - 2.2 handleKeys
 - 2.2.1 handleKey
 - 2.2.2 handleAcceptableKey
 - 2.2.3 handleReadableKey
 - 2.2.4 handleWritableKey
 - 2.3 main
- 3. 客户端
 - 3.1 构造方法
 - 3.2 handleKeys

```
    System.out.println("数据: " + content);  
  
    clientSocketChannel.write(content);  
  
    clientSocketChannel.close();  
  
    clientSocketChannel.register(selector, SelectionKey.OP_READ, responseQueue);  
  
    return true;  
  
    String content) throws ClosedChannelException {
```

- 3.2.1 handleKey
- 3.3.2 handleConnectableKey
- 3.3.3 handleReadableKey
- 3.3.4 handleWritableKey
- 3.3 send
- 3.4 main
- 666. 彩蛋

```
131:
132:     public static void main(String[] args) throws IOException, InterruptedException {
133:         NioClient client = new NioClient();
134:         for (int i = 0; i < 30; i++) {
135:             client.send("nihao: " + i);
136:             Thread.sleep(1000L);
137:         }
138:     }
139:
140: }
```

整块代码我们可以分成 3 部分：

- 构造方法：初始化 NIO 客户端。
- #handleKeys() 方法：基于 Selector 处理 IO 操作。
- #main(String[] args) 方法：创建 NIO 客户端，并向服务器发送请求数据。

下面，我们逐小节来分享。

3.1 构造方法

对应【第 3 至 36 行】的代码。

- clientSocketChannel 属性，客户端的 SocketChannel，在【第 9 至 13 行】和【第 19 行】的代码进行初始化，重点是此处连接了指定服务端。
- selector 属性，选择器，在【第 14 至 17 行】的代码进行初始化，重点是此处将 clientSocketChannel 到 selector 中，并对 SelectionKey.OP_CONNECT 事件感兴趣。这样子，在客户端连接服务端成功时，我们就可以处理该 IO 事件。
- responseQueue 属性，直接声明为 NioClient 的成员变量，是为了方便 #send(String content) 方法的实现。
- 第 21 至 30 行：调用 #handleKeys() 方法，基于 Selector 处理 IO 事件。比较特殊的是，我们是启动了一个线程进

文章目录

- 1. 概述
- 2. 服务端
 - 2.1 构造方法
 - 2.2 handleKeys
 - 2.2.1 handleKey
 - 2.2.2 handleAcceptableKey
 - 2.2.3 handleReadableKey
 - 2.2.4 handleWritableKey
 - 2.3 main
- 3. 客户端
 - 3.1 构造方法
 - 3.2 handleKeys

们需要调用发送请求数据的方法，不能直接在主线程，轮询处理 IO 事件。严格来说，也是应该这样处理。阻塞等待客户端成功连接上服务端。具体的 handleConnectableKey(SelectionKey key) 方法中调用。当然，除还可以通过如下方式：

3.2.1 handleKey
 3.3.2 handleConnectableKey
 3.3.3 handleReadableKey
 3.3.4 handleWritableKey
 3.3 send
 3.4 main
 666. 彩蛋

对应【第 61 至 74 行】的代码。

大体逻辑和 NioServer 中的该方法一模一样，差别将对 `SelectionKey.OP_WRITE` 事件的处理改成对 `SelectionKey.OP_CONNECT` 事件的处理。

3.3.2 handleConnectableKey

对应【第 76 至 88 行】的代码。

- 第 77 至 81 行：判断客户端的 `SocketChannel` 上是否**正在进行连接**的操作，若是，则完成连接。
- 第 83 行：打印日志。
- 第 85 行：注册客户端的 `SocketChannel` 到 `selector` 中，并对 `SelectionKey.OP_READ` 事件感兴趣。这样子，在客户端接收到服务端的消息(数据)时，我们就可以处理该 IO 事件。
- 第 87 行：调用 `CountDownLatch#countDown()` 方法，结束 `NioClient` 构造方法中的【第 32 至 34 行】的阻塞等待连接完成。

3.3.3 handleReadableKey

对应【第 91 至 101 行】的代码。

大体逻辑和 NioServer 中的该方法一模一样，**去掉响应请求的相关逻辑**。😈 如果不去掉，就是客户端和服务端互发消息的“死循环”了。

文章目录

1. 概述
2. 服务端
 - 2.1 构造方法
 - 2.2 handleKeys
 - 2.2.1 handleKey
 - 2.2.2 handleAcceptableKey
 - 2.2.3 handleReadableKey
 - 2.2.4 handleWritableKey
 - 2.3 main
3. 客户端
 - 3.1 构造方法
 - 3.2 handleKeys

码。

[3.2.1 handleKey](#)
[3.3.2 handleConnectableKey](#)
[3.3.3 handleReadableKey](#)
[3.3.4 handleWritableKey](#)
[3.3 send](#)
[3.4 main](#)
[666. 彩蛋](#)

码。

- 第 124 行：添加到响应队列(`responseQueue`)中。
- 第 126 行：打印日志。
- 第 128 行：注册客户端的 `SocketChannel` 到 `selector` 中，并对 `SelectionKey.OP_WRITE` 事件感兴趣。具体的原因，和 `NioServer` 的 `#handleReadableKey(SelectionKey key)` 方法的【第 88 行】一样。
- 第 129 行：调用 `Selector#wakeup()` 方法，唤醒 `#handleKeys()` 方法中，`Selector#select(long timeout)` 方法的阻塞等待。
 - 因为，在 `Selector#select(long timeout)` 方法的实现中，是以调用**当时**，对 `SocketChannel` 的感兴趣的事件。
 - 所以，在【第 128 行】的代码中，即使修改了对 `SocketChannel` 的感兴趣的事件，也不会结束 `Selector#select(long timeout)` 方法的阻塞等待。因此，需要进行唤醒操作。
 - 🐼 感兴趣的胖友，可以将这行代码进行注释，测试下效果就很容易明白了。

3.4 main

对应【第 132 至 137 行】的代码。

- 第 133 行：创建一个 `NioClient` 对象。
- 第 134 至 137 行：每秒发送一次请求。考虑到代码没有处理拆包的逻辑，所以增加了间隔 1 秒的 `sleep`。

666. 彩蛋

呼呼，凌晨 1 点。困累，写的有点着急了。简单 Review 了一遍，如果有不正确的，烦请斧正！谢谢！

推荐阅读文章如下：

- 《【NIO系列】—— Reactor 模式》
- 《lanux/java-demo/nio/example》

文章目录

1. 概述
2. 服务端
 - 2.1 构造方法
 - 2.2 handleKeys
 - 2.2.1 handleKey
 - 2.2.2 handleAcceptableKey
 - 2.2.3 handleReadableKey
 - 2.2.4 handleWritableKey
 - 2.3 main
3. 客户端
 - 3.1 构造方法
 - 3.2 handleKeys