该系列博文会告诉你如何从计算机网络的基础知识入手，一步步地学习Java网络基础，从socket到nio、bio、aio和netty等网络编程知识，并且进行实战，网络编程是每一个Java后端工程师必须要学习和理解的知识点，进一步来说，你还需要掌握Linux中的网络编程原理，包括IO模型、网络编程框架netty的进阶原理，才能更完整地了解整个Java网络编程的知识体系，形成自己的知识框架。

>

- 文章一：JAVA 中原生的 socket 通信机制

当前环境

1. jdk == 1.8

代码地址

git 地址：https://github.com/jasonGeng88/java-network-programming

知识点

- nio 下 I/O 阻塞与非阻塞实现
- SocketChannel 介绍
- I/O 多路复用的原理
- 事件选择器与 SocketChannel 的关系
- 事件监听类型
- 字节缓冲 ByteBuffer 数据结构

场景

接着上一篇中的站点访问问题，如果我们需要并发访问10个不同的网站，我们该如何处理?

在上一篇中，我们使用了 `java.net.socket` 类来实现了这样的需求，以一线程处理一连接的方式，并配以线程池的控制，貌似得到了当前的最优解。可是这里也存在一个问题，连接处理是同步的，也就是并发数量增大后，大量请求会在队列中等待，或直接异常抛出。

为解决这问题，我们发现元凶处在"一线程一请求"上，如果一个线程能同时处理多个请求，那么在高并发下性能上会大大改善。这里就借住 JAVA 中的 nio 技术来实现这一模型。 BIO存在的问题

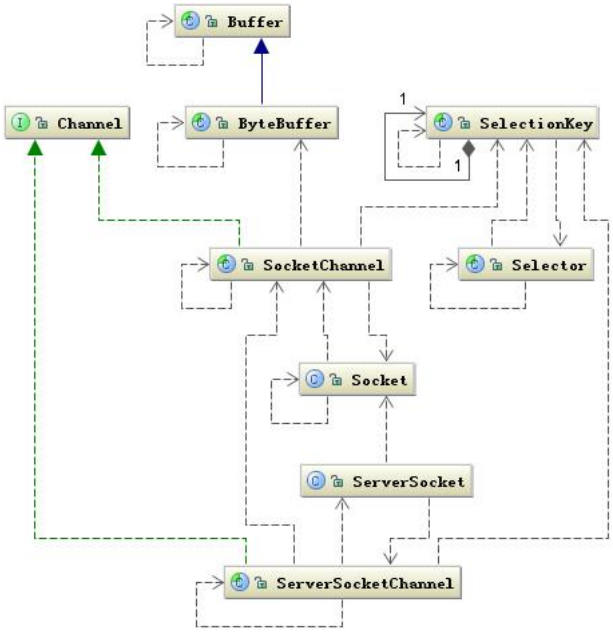
nio 的阻塞实现

关于什么是 nio，从字面上理解为 New IO，就是为了弥补原本 I/O 上的不足，而在 JDK 1.4 中引入的一种新的 I/O 实现方式。简单理解，就是它提供了 I/O 的阻塞与非阻塞的两种实现方式（*当然，默认实现方式是阻塞的。*）。

下面，我们先来看下 nio 以阻塞方式是如何处理的。

建立连接

有了上一篇 socket 的经验，我们的第一步一定也是建立 socket 连接。只不过，这里不是采用 `new socket()` 的方式，而是引入了一个新的概念 `SocketChannel` 。它可以看作是 socket 的一个完善类，除了提供 Socket 的相关功能外，还提供了许多其他特性，如后面要讲到的向选择器注册的功能。



类图如下：

建立连接代码实现：

```
// 初始化 socket，建立 socket 与 channel 的绑定关系
SocketChannel socketChannel = SocketChannel.open();
// 初始化远程连接地址
SocketAddress remote = new InetSocketAddress(this.host, port);
// I/O 处理设置阻塞，这也是默认的方式，可不设置
socketChannel.configureBlocking(true);
// 建立连接
socketChannel.connect(remote);
```

获取 socket 连接

因为是同样是 I/O 阻塞的实现，所以后面的关于 socket 输入输出流的处理，和上一篇的基本相同。唯一差别是，这里需要通过 channel 来获取 socket 连接。

- 获取 socket 连接

```
Socket socket = socketChannel.socket();
```

- 处理输入输出流

```
PrintWriter pw = getWriter(socketChannel.socket());
BufferedReader br = getReader(socketChannel.socket());
```

完整示例

```
package com.jason.network.mode.nio;

import com.jason.network.constant.HttpConstant;
import com.jason.network.util.HttpUtil;

import java.io.*;
import java.net.InetSocketAddress;
import java.net.Socket;
import java.net.SocketAddress;
import java.nio.channels.SocketChannel;

public class NioBlockingHttpClient {

    private SocketChannel socketChannel;
    private String host;

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

        for (String host: HttpConstant.HOSTS) {

            NioBlockingHttpClient client = new NioBlockingHttpClient(host, HttpConstant.PORT);
            client.request();

        }


    }

    public NioBlockingHttpClient(String host, int port) throws IOException {
        this.host = host;
        socketChannel = SocketChannel.open();
        socketChannel.socket().setSoTimeout(5000);
        SocketAddress remote = new InetSocketAddress(this.host, port);
        this.socketChannel.connect(remote);
    }

    public void request() throws IOException {
        PrintWriter pw = getWriter(socketChannel.socket());
        BufferedReader br = getReader(socketChannel.socket());

        pw.write(HttpUtil.compositeRequest(host));
        pw.flush();
        String msg;
        while ((msg = br.readLine()) != null){
            System.out.println(msg);
        }
    }

    private PrintWriter getWriter(Socket socket) throws IOException {
        OutputStream out = socket.getOutputStream();
        return new PrintWriter(out);
    }

    private BufferedReader getReader(Socket socket) throws IOException {
        InputStream in = socket.getInputStream();
        return new BufferedReader(new InputStreamReader(in));
    }
}
```

nio 的非阻塞实现

原理分析

nio 的阻塞实现，基本与使用原生的 socket 类似，没有什么特别大的差别。

下面我们来看看它真正强大的地方。到目前为止，我们将的都是阻塞 I/O。何为阻塞 I/O，看下图：

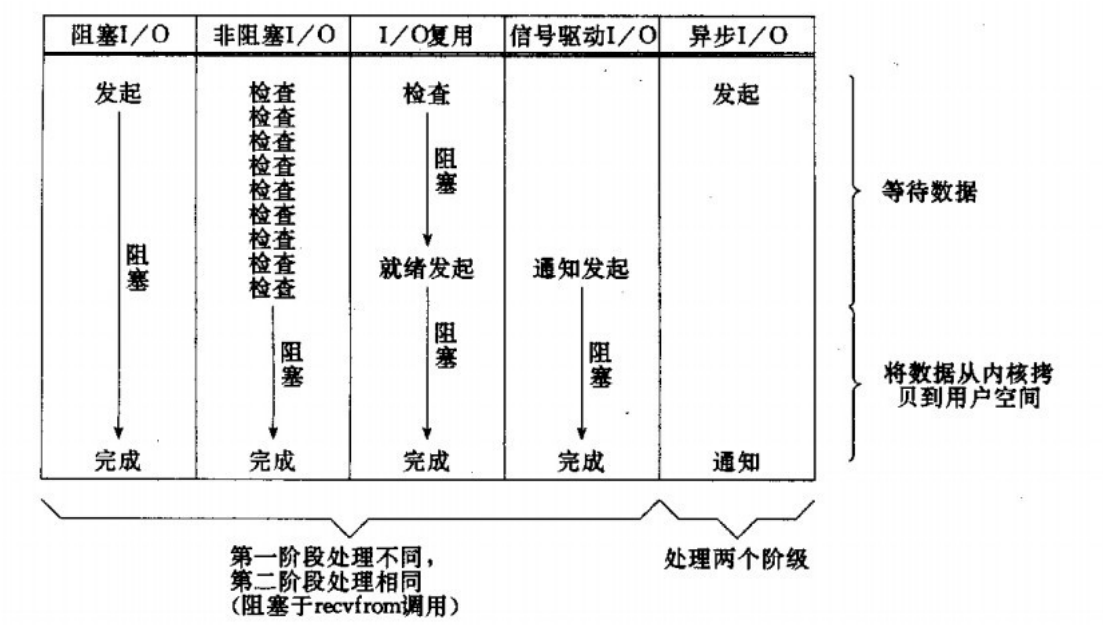


图 6.6　五个 I/O 模型的比较

我们主要观察图中的前三种 I/O 模型，关于异步 I/O，一般需要依靠操作系统的支持，这里不讨论。

从图中可以发现，阻塞过程主要发生在两个阶段上：

- 第一阶段：等待数据就绪；
- 第二阶段：将已就绪的数据从内核缓冲区拷贝到用户空间；

这里产生了一个从内核到用户空间的拷贝，主要是为了系统的性能优化考虑。假设，从网卡读到的数据直接返回给用户空间，那势必会造成频繁的系统中断，因为从网卡读到的数据不一定是完整的，可能断断续续的过来。通过内核缓冲区作为缓冲，等待缓冲区有足够的数据，或者读取完结后，进行一次的系统中断，将数据返回给用户，这样就能避免频繁的中断产生。

了解了 I/O 阻塞的两个阶段，下面我们进入正题。看看一个线程是如何实现同时处理多个 I/O 调用的。从上图中的非阻塞 I/O 可以看出，仅仅只有第二阶段需要阻塞，第一阶段的数据等待过程，我们是不需要关心的。不过该模型是频繁地去检查是否就绪，造成了 CPU 无效的处理，反而效果不好。如果有一种类似的好莱坞原则— “不要给我们打电话，我们会打给你” 。这样一个线程可以同时发起多个 I/O 调用，并且不需要同步等待数据就绪。在数据就绪完成的时候，会以事件的机制，来通知我们。这样不就实现了单线程同时处理多个 IO 调用的问题了吗？即所说的“I/O 多路复用模型”。

废话讲了一大堆，下面就来实际操刀一下。

创建选择器

由上面分析可以，我们得有一个选择器，它能监听所有的 I/O 操作，并且以事件的方式通知我们哪些 I/O 已经就绪了。

代码如下：

```
import java.nio.channels.Selector;

...

private static Selector selector;
static {
    try {
        selector = Selector.open();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

创建非阻塞 I/O

下面，我们来创建一个非阻塞的 `SocketChannel` ，代码与阻塞实现类型，唯一不同是 `socketChannel.configureBlocking(false)` 。

注意：只有在 `socketChannel.configureBlocking(false)` 之后的代码，才是非阻塞的，如果 `socketChannel.connect()` 在设置非阻塞模式之前，那么连接操作依旧是阻塞调用的。

```
SocketChannel socketChannel = SocketChannel.open();
SocketAddress remote = new InetSocketAddress(host, port);
// 设置非阻塞模式
socketChannel.configureBlocking(false);
socketChannel.connect(remote);
```

建立选择器与 socket 的关联

选择器与 socket 都创建好了，下一步就是将两者进行关联，好让选择器和监听到 Socket 的变化。这里采用了以 `SocketChannel` 主动注册到选择器的方式进行关联绑定，这也就解释了，为什么不直接 `new Socket()`，而是以 `SocketChannel` 的方式来创建 socket。

代码如下：

```
socketChannel.register(selector,
                       SelectionKey.OP_CONNECT
                       | SelectionKey.OP_READ
                       | SelectionKey.OP_WRITE);
```

上面代码，我们将 socketChannel 注册到了选择器中，并且对它的连接、可读、可写事件进行了监听。

具体的事件监听类型如下：

操作类型	值	描述	所属对象
OP_READ	1 << 0	读操作	SocketChannel
OP_WRITE	1 << 2	写操作	SocketChannel
OP_CONNECT	1 << 3	连接socket操作	SocketChannel
OP_ACCEPT	1 << 4	接受socket操作	ServerSocketChannel

选择器监听 socket 变化

现在，选择器已经与我们关心的 socket 进行了关联。下面就是感知事件的变化，然后调用相应的处理机制。

这里与 Linux 下的 selector 有点不同，nio 下的 selecotr 不会去遍历所有关联的 socket。我们在注册时设置了我们关心的事件类型，每次从选择器中获取的，只会是那些符合事件类型，并且完成就绪操作的 socket，减少了大量无效的遍历操作。

```
public void select() throws IOException {
        // 获取就绪的 socket 个数
    while (selector.select() > 0){

        // 获取符合的 socket 在选择器中对应的事件句柄 key
        Set keys = selector.selectedKeys();

                // 遍历所有的key
        Iterator it = keys.iterator();
        while (it.hasNext()){

                        // 获取对应的 key，并从已选择的集合中移除
            SelectionKey key = (SelectionKey)it.next();
            it.remove();

            if (key.isConnectable()){
                // 进行连接操作
                connect(key);
            }
            else if (key.isWritable()){
                // 进行写操作
                write(key);
            }
            else if (key.isReadable()){
                // 进行读操作
                receive(key);
            }
        }
    }
}
```

注意：这里的 `selector.select()` 是同步阻塞的，等待有事件发生后，才会被唤醒。这就防止了CPU 空转的产生。当然，我们也可以给它设置超时时间， `selector.select(long timeout)` 来结束阻塞过程。

处理连接就绪事件

下面，我们分别来看下，一个 socket 是如何来处理连接、写入数据和读取数据的（*这些操作都是阻塞的过程，只是我们将等待就绪的过程变成了非阻塞的了*）。

处理连接代码：

```
// SelectionKey 代表 SocketChannel 在选择器中注册的事件句柄
private void connect(SelectionKey key) throws IOException {
        // 获取事件句柄对应的 SocketChannel
    SocketChannel channel = (SocketChannel) key.channel();

   // 真正的完成 socket 连接
    channel.finishConnect();

   // 打印连接信息
    InetSocketAddress remote = (InetSocketAddress) channel.socket().getRemoteSocketAddress();
    String host = remote.getHostName();
    int port = remote.getPort();
    System.out.println(String.format("访问地址: %s:%s 连接成功!", host, port));
}
```

处理写入就绪事件

```
// 字符集处理类
private Charset charset = Charset.forName("utf8");

private void write(SelectionKey key) throws IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    InetSocketAddress remote = (InetSocketAddress) channel.socket().getRemoteSocketAddress();
    String host = remote.getHostName();

        // 获取 HTTP 请求，同上一篇
    String request = HttpUtil.compositeRequest(host);

        // 向 SocketChannel 写入事件
    channel.write(charset.encode(request));

    // 修改 SocketChannel 所关心的事件
    key.interestOps(SelectionKey.OP_READ);
}
```

这里有两个地方需要注意：

- 第一个是使用 `channel.write(charset.encode(request));` 进行数据写入。有人会说，为什么不能像上面同步阻塞那样，通过 `PrintWriter` 包装类进行操作。因为 `PrintWriter` 的 `write()` 方法是阻塞的，也就是说要等数据真正从 socket 发送出去后才返回。

这与我们这里所讲的阻塞是不一致的，这里的操作虽然也是阻塞的，但它发生的过程是在数据从用户空间到内核缓冲区拷贝过程。至于系统将缓冲区的数据通过 socket 发送出去，这不在阻塞范围内。也解释了为什么要用 `Charset` 对写入内容进行编码了，因为缓冲区接收的格式是 `ByteBuffer`。

- 第二，选择器用来监听事件变化的两个参数是 `interestOps` 与 `readyOps`。
    - interestOps：表示 `SocketChannel` 所关心的事件类型，也就是告诉选择器，当有这几种事件发生时，才来通知我。这里通过 `key.interestOps(SelectionKey.OP_READ);` 告诉选择器，之后我只关心"读就绪"事件，其他的不用通知我了。
    - readyOps：表示 `SocketChannel` 当前就绪的事件类型。以 `key.isReadable()` 为例，判断依据就是： `return (readyOps() & OP_READ) != 0;`

处理读取就绪事件

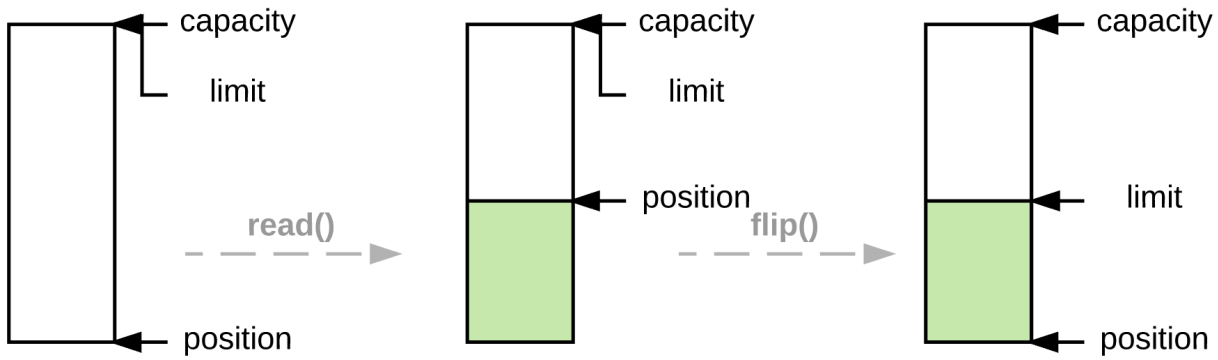
```
private void receive(SelectionKey key) throws IOException {
    SocketChannel channel = (SocketChannel) key.channel();
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    channel.read(buffer);
    buffer.flip();
    String receiveData = charset.decode(buffer).toString();

        // 当再没有数据可读时，取消在选择器中的关联，并关闭 socket 连接
    if ("".equals(receiveData)) {
        key.cancel();
        channel.close();
        return;
    }
```

```
        System.out.println(receiveData);
    }
```

这里的处理基本与写入一致，唯一要注意的是，这里我们需要自行处理去缓冲区读取数据的操作。首先会分配一个固定大小的缓冲区，然后从内核缓冲区中，拷贝数据至我们刚分配固定缓冲区上。这里存在两种情况：

- 我们分配的缓冲区过大，那多余的部分以0补充（*初始化时，其实会自动补0*）。
- 我们分配的缓冲去过小，因为选择器会不停的遍历。只要 `SocketChannel` 处理读就绪状态，那下一次会继续读取。当然，分配过小，会增加遍历次数。

最后，将一下 `ByteBuffer` 的结构，它主要有 position, limit,capacity 以及 mark 属性。以 `buffer.flip();` 为例，讲下各属性的作用（*mark 主要是用来标记之前 position 的位置，是在当前 postion 无法满足的情况下使用的，这里不作讨论*）。



从图中看出，

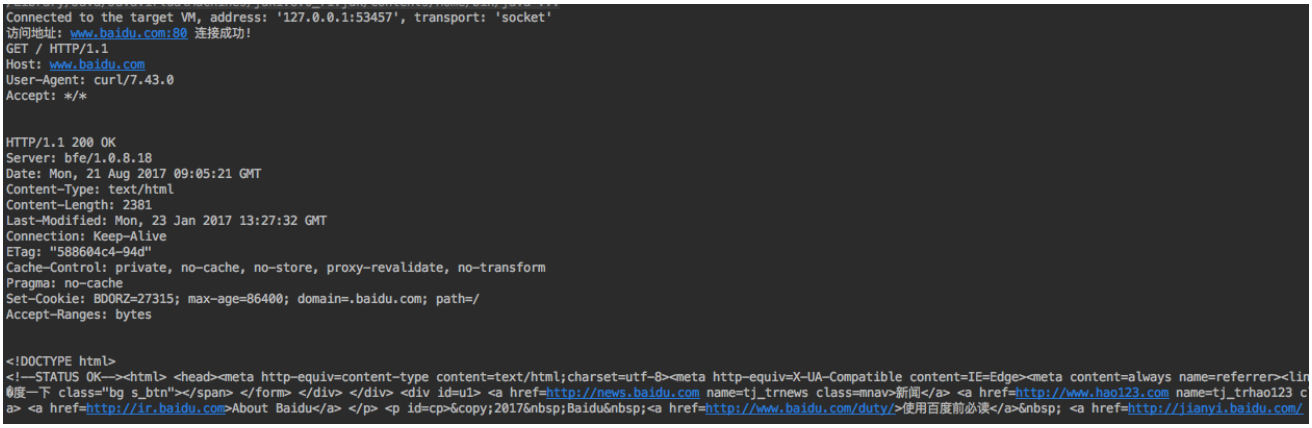
- 容量（capacity）：表示缓冲区可以保存的数据容量；
- 极限（limit）：表示缓冲区的当前终点，即写入、读取都不可超过该重点；
- 位置（position）：表示缓冲区下一个读写单元的位置；

完整代码

```
package com.jason.network.mode.nio;

import com.jason.network.constant.HttpConstant;
import com.jason.network.util.HttpUtil;

import java.io.IOException;
import java.net.InetSocketAddress;
import java.net.SocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.SelectionKey;
import java.nio.channels.Selector;
import java.nio.channels.SocketChannel;
import java.nio.charset.Charset;
import java.util.Iterator;
import java.util.Set;

public class NioNonBlockingHttpClient {

    private static Selector selector;
    private Charset charset = Charset.forName("utf8");

    static {
        try {
            selector = Selector.open();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

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

        NioNonBlockingHttpClient client = new NioNonBlockingHttpClient();

        for (String host: HttpConstant.HOSTS) {

            client.request(host, HttpConstant.PORT);
```

```
        }

        client.select(); 选择器监听所有的socketchannel感兴趣的事件

    }

    public void request(String host, int port) throws IOException { 客户端所有的socket请求（socketchannel），绑定到同一个选择器上进行监听
        SocketChannel socketChannel = SocketChannel.open();
       socketChannel.socket().setSoTimeout(5000);
        SocketAddress remote = new InetSocketAddress(host, port);
       socketChannel.configureBlocking(false);
        socketChannel.connect(remote); 非阻塞模式下，这里不阻塞，接着执行下面代码
        socketChannel.register(selector,
                        SelectionKey.OP_CONNECT 当前socketchannle感兴趣的事件注册到客户端选择器上selector
                        | SelectionKey.OP_READ
                        | SelectionKey.OP_WRITE);
    }

    public void select() throws IOException {
              while (selector.select(500) > 0){选择器selector.select()方法阻塞，等待注册的socketchannel感兴趣事件发生
            Set keys = selector.selectedKeys();

            Iterator it = keys.iterator();

            while (it.hasNext()){

                SelectionKey key = (SelectionKey)it.next();
                it.remove();

                if (key.isConnectable()){
                    connect(key);
                }
                else if (key.isWritable()){
                    write(key);
                }
                else if (key.isReadable()){
                    receive(key);
                }
            }
        }
    }

    private void connect(SelectionKey key) throws IOException {
        SocketChannel channel = (SocketChannel) key.channel(); 想进行连接操作的socketchannel准备就绪
        channel.finishConnect();
        InetSocketAddress remote = (InetSocketAddress) channel.socket().getRemoteSocketAddress();
        String host = remote.getHostName();
        int port = remote.getPort();
        System.out.println(String.format("访问地址：%s:%s 连接成功!", host, port));
    }

    private void write(SelectionKey key) throws IOException {  想进行写操作的socketchannel准备就绪
        SocketChannel channel = (SocketChannel) key.channel();
        InetSocketAddress remote = (InetSocketAddress) channel.socket().getRemoteSocketAddress();
        String host = remote.getHostName();

        String request = HttpUtil.compositeRequest(host);
        System.out.println(request);

        channel.write(charset.encode(request));
        key.interestOps(SelectionKey.OP_READ);
    }


    private void receive(SelectionKey key) throws IOException { 想进行读操作的socketchannel准备就绪
        SocketChannel channel = (SocketChannel) key.channel();
        ByteBuffer buffer = ByteBuffer.allocate(1024);
        channel.read(buffer);
        buffer.flip();
        String receiveData = charset.decode(buffer).toString();

        if ("".equals(receiveData)) {
            key.cancel();
            channel.close();
            return;
        }

        System.out.println(receiveData);
    }
}
```

示例效果



总结

本文从 nio 的阻塞方式讲起，介绍了阻塞 I/O 与非阻塞 I/O 的区别，以及在 nio 下是如何一步步构建一个 IO 多路复用的模型的客户端。文中需要理解的内容比较多，如果有理解错误的地方，欢迎指正~

后续

- Netty 下的异步请求实现