第一章　BIO

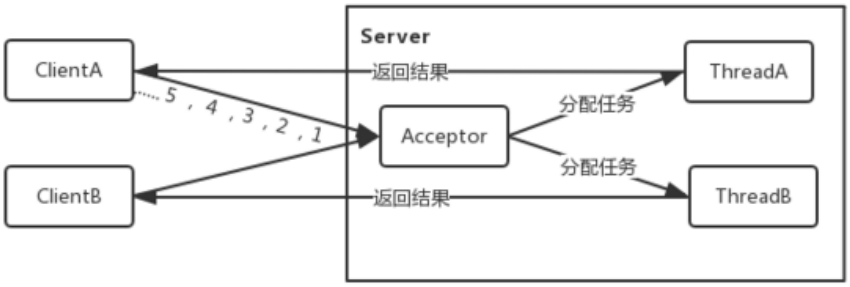
1. BIO 概述

BIO 全称 Block-IO 是一种**阻塞同步**的通信模式。何为阻塞同步？以数据读取过程为例，**阻塞**指的是等待数据准备的过程，而**同步**指的是数据从内核拷贝至进程的阶段。BIO 的特点为**模式简单，使用方便，但并发处理能力低，通信耗时，依赖网速**。

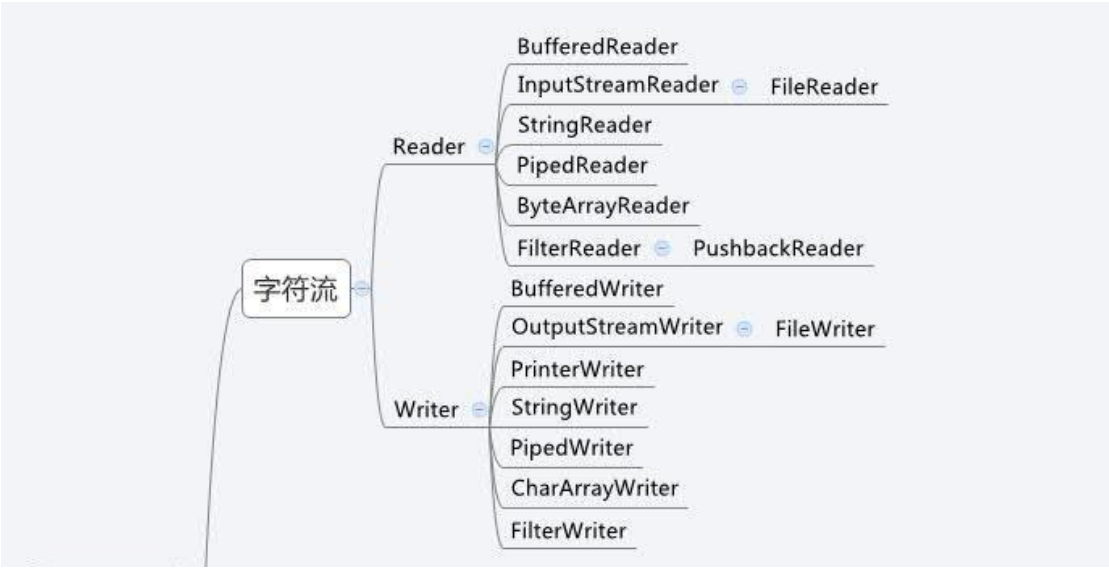
2. BIO 的设计原理

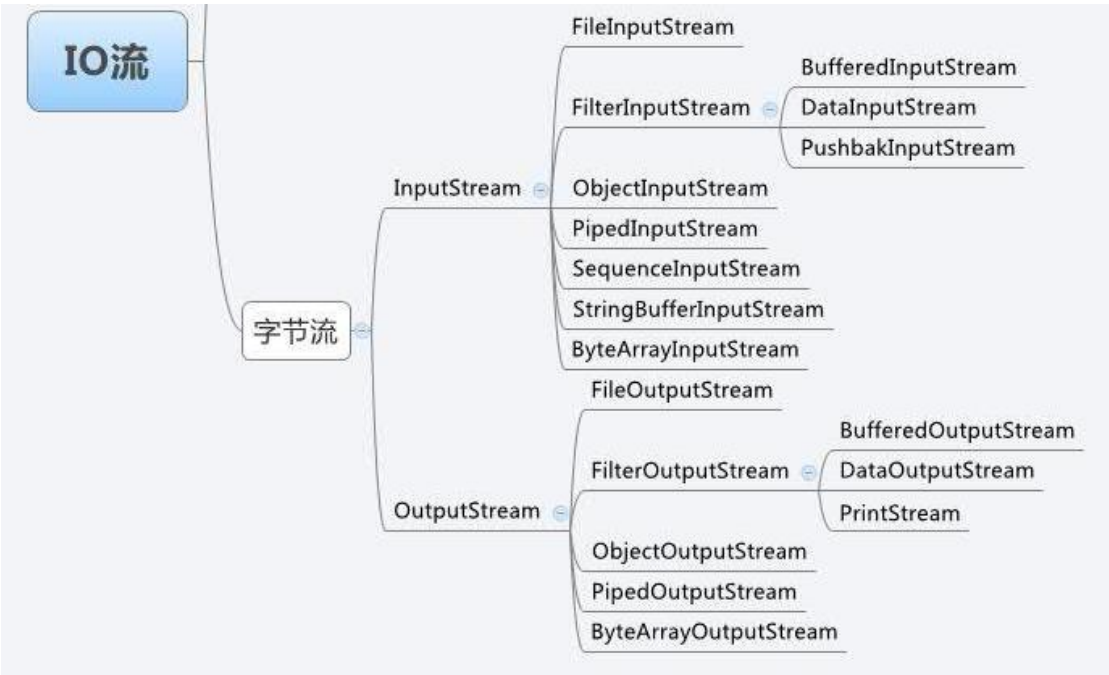
服务器通过一个 Acceptor 线程负责监听客户端请求和为每个客户端创建一个新的线程进行链路处理。典型的一请求一应答模式。若客户端数量增多，频繁地创建和销毁线程会给服务器打开很大的压力。后改良为用线程池的方式代替新增线程，被称为**伪异步 IO**。

服务器提供 IP 地址和监听的端口，客户端通过 TCP 的三次握手与服务器连接，连接成功后，双方才能通过套接字(Socket)通信。



3. 常见的 IO 流





4. 代码解析

服务端：第一步，设置端口监听；第二步，等待监听结果（阻塞）；第三步，创建并处理线程任务。

```
server = new ServerSocket(PORT); // ServerSocket 启动监听端口
/*--------------传统的新增线程处理----------------*/
while (true) {
    socket = server.accept();
    new Thread(new ITDragonBIOServerHandler(socket)).start();
}
```

客户端：第一步，向服务器发起连接；第二步，向服务器发送请求并接收来自服务器的处理结果。

```
socket = new Socket(IP_ADDRESS, PORT); // Socket 发起连接操作。
```

改良后的服务端：第一步，设置端口监听；第二步，创建线程池；第三步，等待监听结果（阻塞）；第四步，创建线程任务并加入线程到线程池中；第五步，处理线程池中的线程任务。

```
server = new ServerSocket(PORT); // ServerSocket 启动监听端口
```

```
/*--------------通过线程池处理缓解高并发给程序带来的压力（伪异步IO编程）----------------*/
executor = new ThreadPoolExecutor( corePoolSize: 10,  maximumPoolSize: 100,  keepAliveTime: 1000,
        TimeUnit.SECONDS, new ArrayBlockingQueue<Runnable>( capacity: 50));
while (true) {
    socket = server.accept();  // 服务器监听：阻塞，等待Client请求
    ITDragonBIOServerHandler serverHandler = new ITDragonBIOServerHandler(socket);
    executor.execute(serverHandler);
}
```

5. BIO【JDK 1.0】小结

BIO 模型中通过 Socket 和 ServerSocket 完成套接字通道的实现。阻塞，同步，建立连接耗时。

实现 BIO 程序开发：同步阻塞 IO 操作，*每一个线程都只会管理一个客户端的链接*，这种操作的本质是存在有程序阻塞的问题。此问题可以通过引入伪异步 IO 的方式进行一定程度的改善。

程序问题：性能不高、多线程的利用率不高、如果大规模的用户访问，有可能会造成服务器端资源耗尽。

第二章　NIO

1. NIO 概述

NIO 全称 New IO，也叫 Non-Block IO 是一种**非阻塞同步**的通信模式。NIO 是对 BIO 的补充和完善，目的在于提升 IO 的处理效率。

2. NIO 的设计原理

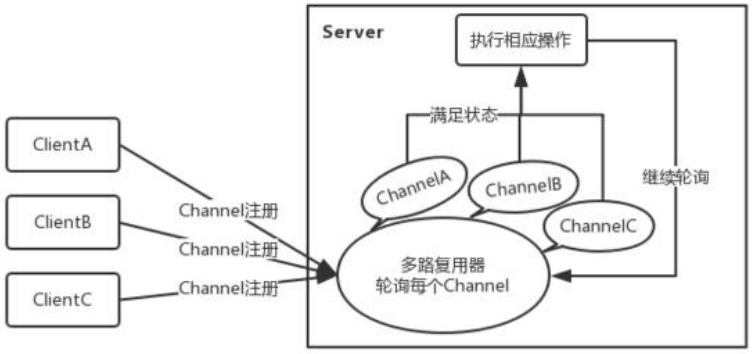
客户端和服务器之间通过 **Channel** 通信。NIO 可以在 Channel 进行读写操作。这些 Channel 都会被注册在 **Selector 多路复用器**上。Selector 通过一个线程不停的**轮询**这些 Channel。找出已经准备就绪的 Channel 执行 IO 操作。NIO 通过一个线程轮询，实现千万个客户端的请求，这就是非阻塞 NIO 的特点。

1）**缓冲区 Buffer**：它是 NIO 与 BIO 的一个重要区别。BIO 是将数据直接写入或读取到 Stream 对象中。而 NIO 的数据操作都是在缓冲区中进行的。缓冲区实际上是一个数组。Buffer 最常见的类型是 ByteBuffer，另外还有 CharBuffer，ShortBuffer，IntBuffer，LongBuffer，FloatBuffer，DoubleBuffer。

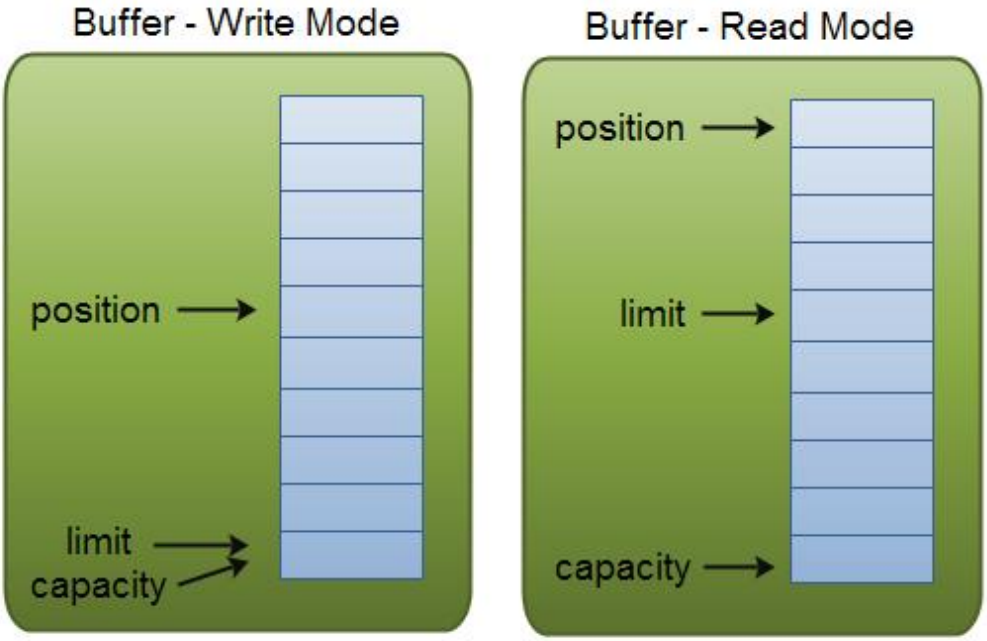
2）**通道 Channel**：和流不同，通道是**双向**的。NIO 可以通过 Channel 进行数据的读或写或同时读写操作。通道分为两大类：一类是**网络读写**（SelectableChannel），一类是用于**文件操作**（FileChannel），我们使用的 SocketChannel 和 ServerSocketChannel 都是 SelectableChannel 的子类。

3）**多路复用器 Selector**：NIO 编程的基础。多路复用器提供选择已经就绪的任务的能力。就是 Selector 会不断地轮询注册在其上的通道（Channel），如果某个通道处于就绪状态，会被 Selector 轮询出来，然后通过 SelectionKey 可以取得就绪的 Channel 集合，从而进行后续的 IO 操作。*服务器端只要提供一个线程负*

责 Selector 的轮询，就可以接入成千上万个客户端，这就是 JDK NIO 库的巨大进步。



3. Buffer 缓冲区的读写模式



Capacity 指开辟的缓冲区大小。写模式下，Position 指缓冲区中当前状态下数据写入到了缓冲区的什么位置，而 Limit 和 Capacity 相等。读模式下，Limit 指缓冲区中当前状态下数据可以被读取到什么位置，而 Position 等于缓冲区的起始位置。图中表示的含义为当 Buffer 由写模式切换到读模式时 Position 和 Limit 的变化。

4. 代码解析

1）把通道注册到多路复用器上。

```
// 1.开启多路复用器
selector = Selector.open();
// 2.打开服务器通道(网络读写通道)
ServerSocketChannel channel = ServerSocketChannel.open();
// 3.设置服务器通道为非阻塞模式，true为阻塞，false为非阻塞
channel.configureBlocking(false);
// 4.绑定端口
channel.socket().bind(new InetSocketAddress(PORT));
// 5.把通道注册到多路复用器上，并监听阻塞事件
/**
 * SelectionKey.OP_READ   : 表示关注读数据就绪事件
 * SelectionKey.OP_WRITE  : 表示关注写数据就绪事件
 * SelectionKey.OP_CONNECT: 表示关注socket channel的连接完成事件
 * SelectionKey.OP_ACCEPT : 表示关注server-socket channel的accept事件
 */
channel.register(selector, SelectionKey.OP_ACCEPT);
```

2）对注册在多路复用器上的通道进行轮询

```
Selector selector = Selector.open();
channel.configureBlocking(false);
SelectionKey key = channel.register(selector, SelectionKey.OP_READ);
while(true) {
  int readyChannels = selector.select();
  if(readyChannels == 0) continue;
  Set selectedKeys = selector.selectedKeys();
  Iterator keyIterator = selectedKeys.iterator();
  while(keyIterator.hasNext()) {
    SelectionKey key = keyIterator.next();
    if(key.isAcceptable()) {
        // a connection was accepted by a ServerSocketChannel.
    } else if (key.isConnectable()) {
        // a connection was established with a remote server.
    } else if (key.isReadable()) {
        // a channel is ready for reading
    } else if (key.isWritable()) {
        // a channel is ready for writing
    }
    keyIterator.remove();
  }
}
```

3）Buffer 缓冲区的读操作

```
// 1.清空缓冲区数据
readBuffer.clear();
// 2.获取在多路复用器上注册的通道
SocketChannel socketChannel = (SocketChannel) selectionKey.channel();
// 3.读取数据，返回
int count = socketChannel.read(readBuffer);
// 4.返回内容为-1 表示没有数据
if (-1 == count) {
    selectionKey.channel().close();
    selectionKey.cancel();
    return ;
}
// 5.有数据则在读取数据前进行复位操作
readBuffer.flip();
// 6.根据缓冲区大小创建一个相应大小的bytes数组，用来获取值
byte[] bytes = new byte[readBuffer.remaining()];
// 7.接收缓冲区数据
readBuffer.get(bytes);
// 8.打印获取到的数据
System.out.println("NIO Server : " + new String(bytes)); // 不能用bytes.toString()
```

4）Buffer 缓冲区的写操作

```
// 4.打开通道
socketChannel = SocketChannel.open();
// 5.连接服务器
socketChannel.connect(inetSocketAddress);
while (true) {
    // 6.定义一个字节数组，然后使用系统录入功能：
    byte[] bytes = new byte[BUFFER_SIZE];
    // 7.键盘输入数据
    System.in.read(bytes);
    // 8.把数据放到缓冲区中
    byteBuffer.put(bytes);
    // 9.对缓冲区进行复位
    byteBuffer.flip();
    // 10.写出数据
    socketChannel.write(byteBuffer);
    // 11.清空缓冲区数据
    byteBuffer.clear();
}
```

5. NIO【JDK 1.4】小结

NIO 模型中通过 SocketChannel 和 ServerSocketChannel 完成套接字通道的实现。非阻塞/阻塞，同步，避免 TCP 建立连接使用三次握手带来的开销。

NIO 模型减少了数组操作，利用了缓存数据方便的保存和清空操作进行 IO 的处理。

Reactor 模型提倡的是：公共注册，统一操作。

第三章　AIO

1. AIO 概述

AIO 也叫 NIO2.0 ，是一种**非阻塞异步**的通信模式。在 NIO 的基础上引入了新的**异步通道**的概念，并提供了异步文件通道和异步套接字通道的实现。

2. AIO 的设计原理

AIO 的核心思想是：**去主函数等待时间**。异步 channel API 提供了两种方式监控/控制异步操作(connect, accept, read，write 等)。

第一种方式是返回 **java.util.concurrent.Future 对象**，检查 Future 的状态可以得到操作是完成还是失败，还是进行中，future.get 阻塞当前进程。

第二种方式为操作提供一个**回调**参数 **java.nio.channels.CompletionHandler**，这个回调类包含 completed、failed 两个方法。

3. 代码解析

1）通过 Future 读取数据

```
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.READ);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

Future<Integer> operation = fileChannel.read(buffer, position);

while(!operation.isDone());

buffer.flip();
byte[] data = new byte[buffer.limit()];
buffer.get(data);
System.out.println(new String(data));
buffer.clear();
```

这个Demo创建了一个AsynchronousFileChannel，然后创建一个ByteBuffer，它被传递给 read()方法作为参数，以及一个 0 的位置。在调用 read()之后，这个示例循环，直到返回的 isDone()方法返回 true（**此处的循环可以使用 futrue.get()方法将线程阻塞**）。当然，这不是非常有效地使用 CPU，但是您需要等到读取操作完成之后才会执行。读取操作完成后，数据读取到 ByteBuffer 中，然后进入一个字符串并打印到 System.out 中。

2）通过 CompletionHandler 读取数据

```
fileChannel.read(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {
    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("result = " + result);

        attachment.flip();
        byte[] data = new byte[attachment.limit()];
        attachment.get(data);
        System.out.println(new String(data));
        attachment.clear();
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {

    }
});
```

一旦读取操作完成，将调用 CompletionHandler 的 completed()方法。对于completed()方法的参数传递一个整数，它告诉我们读取了多少字节，以及传递给read()方法的“附件”。“附件”是 read()方法的第三个参数。在本例中，它是

ByteBuffer，数据也被读取。您可以自由选择要附加的对象。如果读取操作失败，则将调用 CompletionHandler 的 failed()方法。

3）通过 Future 写数据

```
Path path = Paths.get("data/test-write.txt");
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("test data".getBytes());
buffer.flip();

Future<Integer> operation = fileChannel.write(buffer, position);
buffer.clear();

while(!operation.isDone());

System.out.println("Write done");
```

首先，AsynchronousFileChannel 以写模式打开。然后创建一个 ByteBuffer，并将一些数据写入其中。然后，ByteBuffer 中的数据被写入到文件中。最后，示例检查返回的 Future，以查看写操作完成时的情况。

4）通过 CompletionHandler 写数据

```
Path path = Paths.get("data/test-write.txt");
if(!Files.exists(path)){
    Files.createFile(path);
}
AsynchronousFileChannel fileChannel =
    AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);

ByteBuffer buffer = ByteBuffer.allocate(1024);
long position = 0;

buffer.put("test data".getBytes());
buffer.flip();

fileChannel.write(buffer, position, buffer, new CompletionHandler<Integer, ByteBuffer>() {

    @Override
    public void completed(Integer result, ByteBuffer attachment) {
        System.out.println("bytes written: " + result);
    }

    @Override
    public void failed(Throwable exc, ByteBuffer attachment) {
        System.out.println("Write failed");
        exc.printStackTrace();
    }
});
```

当写操作完成时，将会调用 CompletionHandler 的 completed()方法。如果由于某种原因而写失败，则会调用 failed()方法。

4. AIO【JDK 1.7】小结

AIO 模型中通过 AsynchronousSocketChannel 和 AsynchronousServerSocket-Channel 完成套接字通道的实现。非阻塞，异步。

第四章　总结

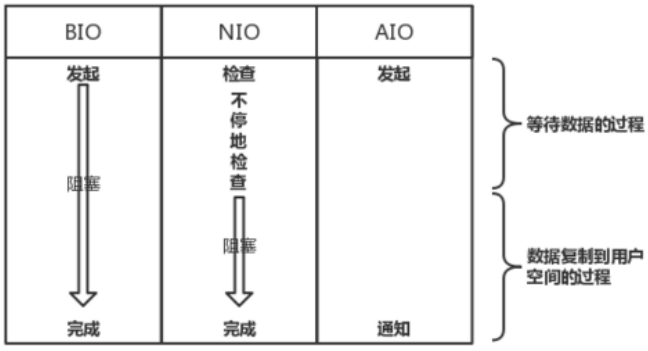
属性\模型	阻塞 BIO	非阻塞 NIO	异步 AIO
blocking	阻塞并同步	非阻塞但同步	非阻塞并异步
线程数（server:client）	1:1	1: N	0: N
复杂度	简单	较复杂	复杂
吞吐量	低	高	高

BIO（同步阻塞 IO）：在进行处理的时候是通过一个线程进行操作，并且 IO 实现通讯的时候采用的是阻塞模式；你现在通过水壶烧水，在 BIO 的世界里面，烧水这一过程你需要从头一直监视到结尾。

NIO（同步非阻塞 IO）：不断的进行烧水状态的判断，同时你可以做其他的事情。

AIO（异步非阻塞 IO）：烧水的过程你不用关注，如果水一旦烧好了，就会给你一个反馈。

最后，我们用一张图总结 BIO、NIO、AIO 的工作模式，更多解释可参考 https://www.jianshu.com/p/8ad464ed516e（强烈推荐）



博主 GitHub 源码

###

参考资料

Java aio 编程 https://colobu.com/2014/11/13/java-aio-introduction/

Netty 序章之 BIO NIO AIO 演变

https://segmentfault.com/a/1190000012976683#articleHeader5

浅谈“阻塞同步” https://www.jianshu.com/p/8ad464ed516e

Java NIO AsynchronousFileChannel https://www.jianshu.com/p/b38f8c596193

Java NIO 系列教程推荐

http://tutorials.jenkov.com/java-nio/asynchronousfilechannel.html

https://www.jianshu.com/p/465ecd909f8c（译）