**目录**

[1 简介和概述 2](#_Toc484531956)

[基本概念 2](#_Toc484531957)

[2 缓冲区 3](#_Toc484531958)

[2.1 缓冲区基础 3](#_Toc484531959)

[2.2 创建缓冲区 4](#_Toc484531960)

[2.3 复制缓冲区 5](#_Toc484531961)

[2.4 缓冲区案例 5](#_Toc484531962)

[3 通道 6](#_Toc484531963)

[3.1 通道基础 6](#_Toc484531964)

[3.2 文件通道 7](#_Toc484531965)

[3.3 Socket通道 8](#_Toc484531966)

[3.4 通道案例 10](#_Toc484531967)

[3.4.1 Socket通道案例 10](#_Toc484531968)

[3.4.2 文件通道案例 14](#_Toc484531969)

[4 选择器 16](#_Toc484531970)

[4.1 选择器基础 16](#_Toc484531971)

[4.2 建立选择器 17](#_Toc484531972)

[4.3 使用选择器 17](#_Toc484531973)

[4.4 选择器案例 19](#_Toc484531974)

# 1 简介和概述

## 基本概念

Java平台提供了一整套I/O，其抽象程度各有不同。然而，离冰冷的现实越远，要想搞清楚来龙去脉就越难——不管使用哪一种抽象，情况都是如此。JDK1.4的NIO软件包引入了一套新的抽象用于I/O处理。与以往不同的是，新的抽象把重点放在了如何缩短抽象与现实之间的距离上面。NIO抽象与现实中存在的实体有着非常真实直接的交互关系。要想最大限度地满足Java应用程序的密集I/O需求，理解这些新的抽象，以及与其他发生交互作用的I/O服务（其重要性不亚于抽象），正是关键所在。

本章假定您熟知基本的I/O概念，NIO类模拟I/O函数，因此，必须掌握操作系统层面的处理细节，才能理解新的I/O模型。

NIO是java New IO的简称，在JDK1.4里面提供的心api。Sun官方标榜的特性如下：

1、为所有的原始类型提供（Buffer）缓存支持。

2、字符集编码解码解决方案。

3、Channel：一个新的原始I/O抽象。

4、支持锁和内存映射文件的文件访问接口。

5、提供多路（non-bloking）非阻塞式的高伸缩性网络I/O。

**集成的I/O**

在JDK1.4中原来的I/O包和NIO已经很好地集成了。java.io.\*已经以NIO为基础重新实现了，所以现在它可以利用NIO的一些特性。

例如，java.io.\*包中的一些类包含以块的形式读写数据的方法，这使得即使在更面向流的系统中，处理速度也会更快。

也可以用NIO库实现标准I/O功能。例如，可以容易地使用块I/O一次一个字节地移动数据。但是正如您会看到的，NIO还提供了原I/O包中所没有的许多好处。

NIO创建目的是让Java程序员可以实现高速I/O而不需编写自定义的本机代码。NIO将最耗时的I/O操作（即填充和提取缓冲区）转移回操作系统，因而可以极大地提高速度。

流与块的比较

原来的I/O库（在java.io.\*中）与NIO最重要的区别是数据打包和传输方式。正如前面提到的，原来的I/O以流的方式处理数据，而NIO以块的方式处理数据。

面向流的I/O系统一次一个字节的处理数据。一个输入流产生一个字节的数据，一个输出流消费一个字节的数据。为流式数据创建过滤器非常容易。链接几个过滤器，以便每个过滤器只负责单个复杂处理机制的一部分，这样也是相对简单的。不利的一面是，面向流的I/O通常相当慢。

一个面向块的I/O系统以块的形式处理数据。每一个操作都在一步中产生或者消费一个数据块。按块处理数据比按（流式的）字节处理数据要快得多。但是面向I/O缺少一些面向流的I/O所具有的优雅性和简单性。

# 2 缓冲区

## 2.1 缓冲区基础

概念上，**缓冲区**是包在一个对象内的基本数据元素**数组**。Buffer类相比一个简单数组的优点是：它将关于数据的数据内容和信息包含在一个单一的对象中。Buffer类以及它专有的子类定义了一个用于处理数据缓冲区的API。

所有的缓冲区都具有四个属性来提供关于其所包含的数据元素的信息。它们是：

1.容量（Capacity）2.上界（Limit）3.位置（Position）4.标记（Mark）

**容量（Capacity）**

缓冲区能够容纳的数据元素的最大数量。这一容量在缓冲区创建时被设定，并且永远不能被改变。

**上界（Limit）**

缓冲区的第一个不能被读或写的元素。或者说，缓冲区中现存元素的计数。

**位置（Position）**

下一个要被读或写的元素的索引。位置会自动由相应的get()和put()函数更新。

**标记（Mark）**

一个备忘位置。调用mark()来设定 mark = position。调用reset()设定position = mark。标记在设定前是未定义的（undefined）。

缓冲区的分类：

1.字节缓冲区ByteBuffer

2.字符缓冲区CharBuffer

3.双精浮点型（double）缓冲区DoubleBuffer

4.单精浮点型（float）缓冲区FloatBuffer

5.整型（int）缓冲区IntBuffer

6.长整型（long）缓冲区LongBuffer

7.短整型（short）缓冲区ShortBuffer

注：没有布尔类型的缓冲区

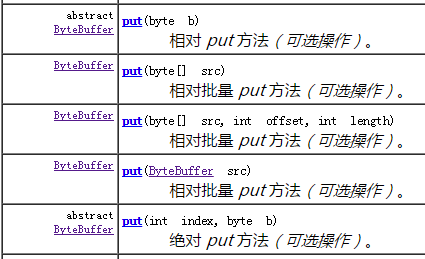
上述的各类型的缓冲区 都提供了读和写的方法 get和put方法，也提供了一些批量的put和get方法。而且缓冲区可以通过allocation创建，此方法通过wrapping将一个现有（数据类型）数组包装到缓冲区中来为缓冲区内容分配空间，或者通过创建现有字节缓冲区的视图来创建。

缓冲区的操作：

1.存：通过相关的Buffer类和put方法进行操作。

2.取：通过相关的Buffer类的get方法进行操作。

还有其他一些get/put方法的重载方式，参见API文档



## 2.2 创建缓冲区

Java.nio软件包中有七种主要的缓冲区类，每一种都具有一种Java语言中的非布尔类型的原始类型数据。这些类没有一种能够直接实例化。他们都是抽象类，但是都包含静态工厂方法用来创建相应类的新实例。

虽然有七种类型的缓冲区类，但是他们的创建方式基本都是类似的。

新的缓冲区是由分配或包装操作创建的。**分配操作**创建一个缓冲区对象并分配一个私有的空间来储存容量大小的数据元素。**包装操作**创建一个缓冲区对象但是不分配任何空间来储存数据元素。它使用您所提供的数组作为存储空间来储存缓冲区中的数据元素。

要分配一个容量为100个char变量的CharBuffer:

CharBuffer charBuffer = CharBuffer.allocate(100);

如果您想提供您自己的数组用作缓冲区的备份存储器，请调用wrap()函数：

char[] myArray = new char[100];

CharBuffer charbuffer = CharBuffer.wrap(myArray);

通过allocate()或者wrap()函数创建的缓冲区通常都是间接的。

间接的缓冲区使用备份数组，像我们之前讨论的，您可以通过上面流出的API函数获得对这些数组的存取权。Boolean型函数hasArray()告诉您这个缓冲区是否有一个可存取的备份数组。如果这个函数的返回true，array()函数会返回这个缓冲区对象所使用的数组存储空间的引用。

## 2.3 复制缓冲区

如我们刚刚所讨论的那样，可以创建描述从外部存储到数组中的数据元素的缓冲区对象。但是缓冲区不限于管理数组中的外部数据。它们也能管理其他缓冲区中的外部数据。当一个管理其他缓冲器所包含的数据元素的缓冲器被创建时，这个缓冲器被称为视图缓冲器。大多数的视图缓冲器都是ByteBuffer的视图。在继续前往字节缓冲器的细节之前，我们先将注意力放在所有缓冲器类型的共同视图上。

视图存储器总是通过调用已存在的存储器实例中的函数来创建。使用已存在的存储器实例中的工厂方法意味着视图对象为原始存储器的内部实现细节私有。数据元素可以直接存取，无论它们是存储在数组中还是以一些其他的方式，而不需经过原始缓冲区对象的get()/put() API。如果原始缓冲区是直接缓冲区，该缓冲区的视图会具有同样的效率优势。映像缓冲区也是如此。

Duplicate()函数创建了一个与原始缓冲区相似的新缓冲区。两个缓冲区共享数据元素，拥有同样的容量，但每个缓冲区拥有各自的位置，上界和标记属性。对一个缓冲区内的数据元素所做的改变会反应在另外一个缓冲区上。这一副本缓冲区具有与原始缓冲区同样的数据视图。如果原始的缓冲区为只读，或者直接缓冲区，新的缓冲区将继续这些属性。

注意：复制一个缓冲区会创建一个新的Buffer对象，但并不复制数据。原始缓冲区和副本都会操作同样的数据元素。

## 2.4 缓冲区案例

Eclipse 中创建项目JavaNIO

创建包buffer

创建类BufferTest

代码如下：

|  |
| --- |
| **package** buffer;  **import** java.nio.IntBuffer;  **public** **class** BufferTest {  **public** **static** **void** main(String[] args) {  // 创建指定长度的缓冲区  IntBuffer buff = IntBuffer.*allocate*(10);  **int**[] array = **new** **int**[] { 3, 5, 1 };  // 使用数组来创建一个缓冲区视图[3, 5, 1]  buff = IntBuffer.*wrap*(array);  // 利用数组的某一个区间来创建视图  // buff = IntBuffer.wrap(array, 0, 2);  // 对缓冲区某个位置上面进行元素修改  buff.put(0, 7);  // 遍历缓冲区中的数据  System.***out***.println("缓冲区数据如下：");  **for** (**int** i = 0; i < buff.limit(); i++) {  // 调用这个get方法会把pos自动递增  System.***out***.print(buff.get() + "\t");  }    System.***out***.println("\n原始数据如下：");  **for** (**int** a : array) {  System.***out***.print(a + "\t");  }    System.***out***.println("\n调用flip方法之前，Buffer类信息：" + buff);  buff.flip();// 对缓冲区进行反转，（limit = pos; pos = 0）  System.***out***.println("\n调用flip方法之后，Buffer类信息：" + buff);  buff.clear(); // 也是把pos清零      IntBuffer buff2 = buff.duplicate(); // duplicate函数是缓冲区  System.***out***.println(buff2); // 复制的缓冲区buff2与原始缓冲区buff具有同样的属性。    //// 0 <= mark <= position <= limit <= capacity    }  } |

在JDK1.8运行结果如下：

|  |
| --- |
| 缓冲区数据如下：  7 5 1  原始数据如下：  7 5 1  调用flip方法之前，Buffer类信息：java.nio.HeapIntBuffer[pos=3 lim=3 cap=3]  调用flip方法之后，Buffer类信息：java.nio.HeapIntBuffer[pos=0 lim=3 cap=3]  java.nio.HeapIntBuffer[pos=0 lim=3 cap=3] |

# 3 通道

## 3.1 通道基础

通道（Channel）是java.nio的第二个主要创新。它们既不是一个扩展也不是一项增强，而是全新、极好的Java I/O示例，提供与I/O服务的直接连接。Channel用于在字节缓冲区和位于通道另一侧的实体（通常是一个文件或套接字）之间有效地传输数据。

通道可以形象地比喻为银行出纳窗口使用的启动导管。您的薪水支票就是您要传送的信息，载体（Carrier）就好比一个缓冲区。您先填充缓冲区（将您的支票放到载体上），接着将缓冲“写”到通道中（将载体丢进导管中），然后信息负载就被传送到通道另一侧的I/O服务（银行出纳员）。

通道是一种途径，借助该途径，可以用最小的开销来访问操作系统本身的I/O服务。缓冲区则是通道内部用来发送和接收数据的端点。

与缓冲区不同，通道API主要由接口指定。不同的操作系统上通道实现（ChannelImplementation）会有根本性的差异，所以通道API仅仅描述了可以做什么。因此很自然地，通道实现经常使用操作系统的本地代码。通道接口允许您以一种受控且可移植的方式来访问底层的I/O服务。

您可以从顶层的Channel接口看到，对所有通道来说只有两种共同的操作：检查一个通道是否打开（isOpen()）和关闭一个打开的通道（close()）。所有有趣的东西都是那些Channel接口以及它的子接口的类。

通道是访问I/O服务的导管。I/O可以分为广义的两大类别：File I/O和Stream I/O。那么相应地有两种类型的通道也就不足为怪了，它们是文件（file）通道和套接字（socket）通道。你会发现有一个FileChannel类和三个socket通道类：SocketChannel、ServerSocketChannel和DatagramChannel。

通道可以以多种方式创建。Socket通道有可以直接创建新socket通道的工厂方法。但是一个FileChannel对象却只能通过在一个打开的RandomAccessFile、FileInputStream或FileOutputStream对象上调用getChannel()方法来获取。您不能直接创建一个FileChannel对象。File和socket通道会在后面的章节中予以详细讨论。

**使用通道**

通道可以是单向（unidirectional）或者双向（bidirectional）。一个channel类可能实现定义read()方法的ReadableByteChannel接口，而另一个channel类也许实现WritableByteChannel接口以提供write()方法。实现这两种接口其中之一的类都是单向的，只能在一个方向上传输数据。如果一个类**同时实现这两个接口**，那么它是双向的，可以双向传输数据。

通道会连接一个特定I/O服务且通道实例（channel instance）的性能受它所连接的I/O服务的特征限制，记住这很重要。一个连接到只读文件Channel实例不能进行写操作，即使该实例所属的类可能有write()方法。基于此，程序员需要知道通道是如何打开的，避免试图尝试一个底层I/O服务不允许的操作。

通道可以以阻塞（blocking）或非阻塞（nonblocking）模式运行。非阻塞模式的通道永远不会让调用的线程休眠。请求的操作要么立即完成，要么返回一个结果表明未进行任何操作。只有面向流的（stream-oriented）的通道，如sockets和pipes才能使用非阻塞模式。

**关闭通道**

通过调用通道的close方法进行关闭，但是可能会导致关闭底层I/O服务时发生阻塞（非阻塞模式和阻塞模式都一样）

通过isOpen方法来测试通道的开放状态，如果返回true，那么说明通道可以使用。反之，说明通道可以使用。反之，说明通道已经关闭，不能使用。

## 3.2 文件通道

FileChannel类

文件通道总是非阻塞式的，因此不能被置于阻塞模式。现代操作系统都有复杂的缓存的**预取机制**，使得本地磁盘I/O操作延迟很少。网络文件系统一般而言延迟会多些，不过却也因该优化而受益。面向流的I/O的非阻塞范例对于面向文件的操作无多大意义，这是由文件I/O本质上的不同性质造成的。对于文件I/O，最强大之处在于异步I/O（asynchronous I/O），它允许一个进程可以从操作系统请求一个或多个I/O操作而不必等待这些操作的完成。发起请求的进程之后会收到它请求的I/O操作已完成的通知。异步I/O是一种高级性能，当前的很多操作系统都不具备。以后的NIO增强也会把异步I/O纳入考虑范围。

FileChannel对象不能直接创建。一个FileChannel实例只能通过在一个打开的file对象（RandomAccessFile， FileInputStream或FileOutputStream）上调用getChannel()方法获取。调用getChannel()方法会返回一个连接到相同文件的FileChannel对象且该FileChannel对象具有与file对象相同的范文权限，然后您就可以使用该通道对象来利用强大的FileChannel API了：

FileChannel对象是**线程安全（thread-safe）**的。多个进程可以在同一个实例上并发调用方法而不会引起任何问题，不过并非所有的操作都是多线程的（multithreaded）。影响通道位置或者影响文件大小的操作都是单线程的（single-threaded）。如果有一个线程已经在执行会影响通道位置或文件大小的操作，那么其他尝试进行此类操作之一的线程必须等待。并发行为也会受到底层的操作系统或文件系统影响。

同大多数I/O相关的类一样，FileChannel是一个反映Java虚拟机外部一个具体对象的抽象。FileChannel类保证同一个Java虚拟机的所有实例看到的某个文件的视图均是一致的，但是Java虚拟机却不能对超出它控制范围的因素提供担保。通过一个FileChannel实例看到的某个文件的视图同通过一个外部的非Java进程看到的该文件的视图可能一致，也可能不一致。多个进程发起的并发文件访问的语义高度取决于底层的操作系统和（或）文件系统。一般而言，由运行在不同Java虚拟机上的FileChannel对象发起的对某个文件的并发访问和由非Java进程发起的对该文件的并发访问是一致的。

**访问文件**

在通道这块我们可以使用FileChannel的read和write方法进行文件的访问，以及配合position()进行文件的操作。

FIleChannel位置（position）是从底层的文件描述符获得的，该position同时被作为通道引用获取来源的文件对象共享。这也就意味着一个对象对该position的更新可以被另一个对象看到。

position能够决定文件中哪一处的数据接下来将被读或写。类似于缓冲区的get()和put()方法，当字节被read()或write()方法传输时，文件position会自动更新。如果position值达到了文件大小的值（文件大小的值可以通过size()方法返回），read()方法会返回一个文件尾的条件值（-1）。可是不同于缓冲区的是，如果实现write()方法时position前进到超过文件大小的值，该文件会扩展，以容纳新写入的字节。

## 3.3 Socket通道

新的socket通道类可以非阻塞模式并且是可选择的。这两个性能可以激活大程序（如网络服务区和中间件组件）巨大的可伸缩性和灵活性。本节中我们会看到，再也没有为两个socket连接使用一个线程的必要了，也避免了管理大量线程所需的上下文交换总开销。借助新的NIO类，一个或几个线程就可以管理成百上千的活动。socket连接了并且只有很少甚至可能没有性能损失。

全部socket通道类（DatagramChannel、SocketChannel和ServerSocketChannel）在被实例化时都会创建一个对等socket对象。这些事我们所熟悉的来自java.net的类（Socket、ServerSocket和DatagramSocket），它们已经被更新以识别通道。对等socket可以通过调用socket()方法从一个通道上获取。此外，这三个java.net类现在都有getChannel()方法。

虽然每个socket通道（在java.nio.channels包中）都有一个关联的java.net socket对象，却并非所有的socket都有一个关联的通道。如果您用传统方式（直接实例化）创建了一个Socket对象，它就不会有关联的SocketChannel并且它的getChannel()方法将总是返回null。

**非阻塞模式**

Socket通道可以在非阻塞模式下运行。这个说法虽然简单却有着深远的含义。传统Java Socket的阻塞性质曾经是Java程序可伸缩性的重要制约之一。非阻塞I/O是许多复杂的、高性能的程序构建的基础。

设置或重新设置一个通道的阻塞模式是很简单的，只要调用configureBlocking()方法即可，传递参数值为true则设为阻塞模式，参数值为false值设为非阻塞模式。真的，就这么简答！您可以通过调用isBlocking()方法来判断某个socket通道当前处于哪种模式。

服务器端的使用经常会考虑到非阻塞socket通道，因为他们使同时管理很多socket通道变得更容易。但是在客户端使用一个或几个非阻塞式的socket通道也是有益处的，例如，借助非阻塞socket通道，GUI程序可以专注于用户请求并且同时维护与一个或多个服务器的会话。在很多程序上，非阻塞模式都是有用的。

**ServerSocketChannel**

它是一个基于通道的socket监听器。它同我们所熟悉的java.net.ServerSocket执行相同的基本任务，不过它增加了通道语义，因此能够在非阻塞模式下运行。用静态的open()工厂方法创建一个新的ServerSocketChannel对象，将会返回同一个未绑定的java.net.ServerSocket关联的通道。该对等ServerSocket可以通过在返回的ServerSocketChannel上调用socket()方法来获取。作为ServerSocketChannel的对等体被创建的SerSocket对象依赖通道实现。这些socket关联的SocketImpl能识别通道。

通道不能被封装在随意的socket对象外面。由于ServerSocketChannel没有bind()方法，因此有必要取出对等的socket并使用它来绑定到一个端口以开始监听连接。我们也是使用对等ServerSocket的API来根据需要设置其他的socket选项。

和java.net.ServerSocket一样，ServerSocketChannel也有accept()方法。一旦您创建了一个ServerSocketChannel并用对等socket绑定了它，然后您就可以在其中一个上调用accept()。如果您选择在ServerSocket上调用accept()方法，那么它会同任何其他的ServerSocket表现一样的行为：

总是阻塞并返回一个java.net.Socket对象。如果您选择在ServerSocketChannel上调用accept()方法则会返回SocketChannel类型对象，返回的对象能够在非阻塞模式下运行。假设系统已经有一个安全管理器（security manager），两种形式的方法调用都执行相同的安全检查。

如果以非阻塞模式被调用，当没有传入连接在等待时，ServerSocketChannel.accept()会立即返回null。正式这种检查连接而不阻塞的能力实现了可伸缩性并降低了复杂性。可选择性也因此得到了实现。我们可以使用一个选择器实例来注册一个ServerSocketChannel对象以实现新连接达到时自动通知的功能。

**SocketChannel**

Socket和SocketChannel类封装点对点、有序的网络连接，类似于我们所熟知并喜爱的TCP/IP网络连接。SocketChannel扮演客户端发起同一个监听服务器的连接。直到连接成功，它才能收到数据并且只会连接到的地址接收。

每个SocketChannel对象创建时都是同一个对等的java.net.Socket对象串联的。静态的open()方法可以创建一个新的SocketChannel对象，而在新创建的SocketChannel上调用socket()方法能返回它对等的Socket对象；在该Socket上调用getChannel()方法则能返回最初的那个SocketChannel。

新创建的SocketChannel虽已打开却是未连接的。在一个未连接的SocketChannel对象上尝试一个I/O操作会导致NotYetConnectedException异常。我们可以通过在通道上直接调用connect()方法或在通道关联的Socket对象上调用connect()来讲socket通道连接。一旦一个socket通道被连接，它将保持连接状态知道被关闭。您可以通过调用布尔类型的isConnected()方法来测试某个SocketChannel当前是否已经连接。

如果您选择使用传统方式进行连接——通过在对等Socket对象上调用connect()方法，那么传统的连接语义将适用于此。线程在连接建立好或超时过期之前都将保持阻塞。如果您选择通过在通道上直接调用connect()方法来建立连接并且通道处于阻塞模式（默认模式），那么连接过程实际上是一样的。

在SocketChannel上并没有一种connect()方法可以让您指定超时（timeout）值，当connect()方法在非阻塞模式下被调用时SocketChannel提供并发连接；它发起队请求地址的连接并且立即返回值。如果返回值是true，说明连接立即建立了（这可能是本地还回连接）；如果连接不能立即建立，connect()方法会返回false且并发地继续连接建立过程。

面向流的socket建立连接状态需要一定的时间，因为两个待连接系统之间必须进行包对话以建立维护流socket所需的状态信息。跨越开放互联网连接到远程系统会特别耗时。假如某个SocketChannel上当前正由一个并发连接，iSConnectPending()方法就会返回true值。

调用finishConnect()方法来完成连接过程，该方法任何时候都可以安全地进行调用。假如在一个非阻塞模式的SocketChannel对象上调用finishConnect()方法，将可能出现下列情形之一：

1、connect()方法尚未被调用。那么将产生NoConnectionPendingException异常。

2、连接建立过程正在进行，尚未完成。那么什么都不会发生，finishConnect()方法会立即返回false值。

3、在非阻塞模式下调用connect()方法之后，SocketChannel又被切换回了阻塞模式。那么如果有必要的话，调用线程阻塞直到连接建立完finishConnect()方法接着就会返回true值。

4、在初次调用connect()或最后一次调用finishConnect()之后，连接建立过程已经完成。那么SocketChannel对象的内部状态将被更新到已连接状态，finishConnect()方法会返回true值，然后SocketChannel对象就可以被用来传输数据了。

5、连接已经建立。那么什么都不会发生，finishConnect()方法会返回true值。

## 3.4 通道案例

### 3.4.1 Socket通道案例

打开Eclipse，在之前创建的Java项目JavaNIO中创建包channel，创建类NioChannelServer

代码如下：

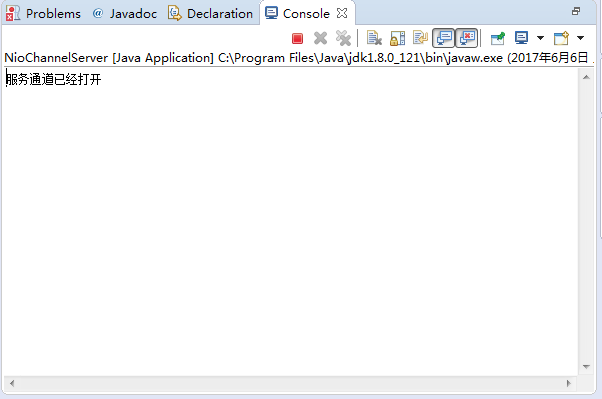
|  |
| --- |
| **package** channel;  **import** java.io.IOException;  **import** java.net.InetSocketAddress;  **import** java.nio.ByteBuffer;  **import** java.nio.IntBuffer;  **import** java.nio.channels.ServerSocketChannel;  **import** java.nio.channels.SocketChannel;  **public** **class** NioChannelServer {  **private** ByteBuffer buff = ByteBuffer.*allocate*(1024);  // 创建一个int缓冲区的视图 此缓冲内容的更改在新缓冲区中是可见的。反之亦然。  **private** IntBuffer intBuff = buff.asIntBuffer();  **private** SocketChannel clientChannel = **null**;  **private** ServerSocketChannel serverChannel = **null**;  /\*\*  \* 打开服务端的通道  \* **@throws** IOException  \*/  **public** **void** openChannel() **throws** IOException {  // 建立一个新的连接的通道  serverChannel = ServerSocketChannel.*open*();  // 为新的通道设置访问的端口  serverChannel.socket().bind(**new** InetSocketAddress(8888));  System.***out***.println("服务通道已经打开");  }  /\*\*  \* 等待客户端的请求连接  \* **@throws** IOException  \*/  **public** **void** waitReqConn() **throws** IOException {  **while** (**true**) {  clientChannel = serverChannel.accept();  **if** (**null** != clientChannel) {  System.***out***.println("新的连接加入！");  }  processReq();// 处理请求  clientChannel.close();  }  }  /\*\*  \* 处理请求过来的数据  \* **@throws** IOException  \*/  **public** **void** processReq() **throws** IOException {  System.***out***.println("开始读取和处理客户端数据...");  buff.clear();// 把当前位置设置为0， 上限值修改为容量的值。  clientChannel.read(buff);  **int** result = intBuff.get(0) + intBuff.get(1);  buff.flip();  buff.clear();  // 修改视图，原来的缓冲区也会变化  intBuff.put(0, result);  clientChannel.write(buff);  System.***out***.println("读取和处理客户端数据完成。");  }    **public** **void** start() {  **try** {  // 打开服务通道  openChannel();  // 监听客户端请求  waitReqConn();  clientChannel.close();  System.***out***.println("服务端处理完毕。");  } **catch** (IOException e) {  e.printStackTrace();  }  }  **public** **static** **void** main(String[] args) {  **new** NioChannelServer().start();  }  } |

在创建NioChannelClient类，代码如下：

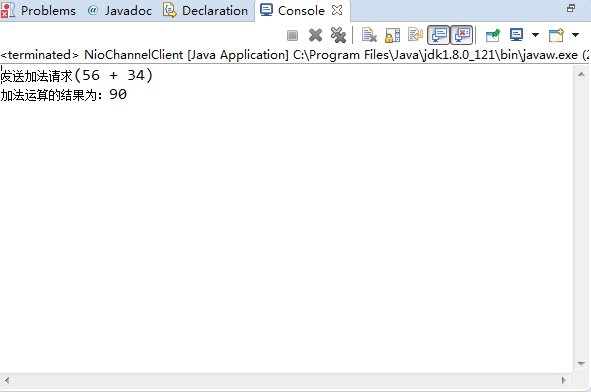
|  |
| --- |
| **package** channel;  **import** java.io.IOException;  **import** java.net.InetSocketAddress;  **import** java.nio.ByteBuffer;  **import** java.nio.IntBuffer;  **import** java.nio.channels.SocketChannel;  **public** **class** NioChannelClient {  **private** SocketChannel channel = **null**;  **private** ByteBuffer buff = ByteBuffer.*allocate*(8);  **private** IntBuffer intBuff = buff.asIntBuffer();  /\*\*  \* 与服务器指定的地址和端口建立连接通道  \*  \* **@return**  \* **@throws** IOException  \*/  **public** SocketChannel connect() **throws** IOException {  **return** SocketChannel.*open*(**new** InetSocketAddress("127.0.0.1", 8888));  }  /\*\*  \* 发送加法请求到服务器  \* **@param** a  \* **@param** b  \* **@throws** IOException  \*/  **public** **void** sendRequest(**int** a, **int** b) **throws** IOException {  buff.clear();  intBuff.put(0, a);  intBuff.put(1, b);  channel.write(buff);  System.***out***.println("发送加法请求(" + a + " + " + b + ")");  }  /\*\*  \* 接收服务器运算结果  \* **@return**  \* **@throws** IOException  \*/  **public** **int** receiveResult() **throws** IOException {  buff.clear();  channel.read(buff);  **return** intBuff.get(0);  }  /\*\*  \* 获得加法运算的结果  \* **@param** a  \* **@param** b  \* **@return**  \*/  **public** **int** getSum(**int** a, **int** b) {  **int** result = 0;  **try** {  channel = connect();  sendRequest(a, b);  result = receiveResult();  } **catch** (IOException e) {  e.printStackTrace();  }  **return** result;  }  **public** **static** **void** main(String[] args) {  **int** result = **new** NioChannelClient().getSum(56, 34);  System.***out***.println("加法运算的结果为：" + result);  }  } |

运行结果

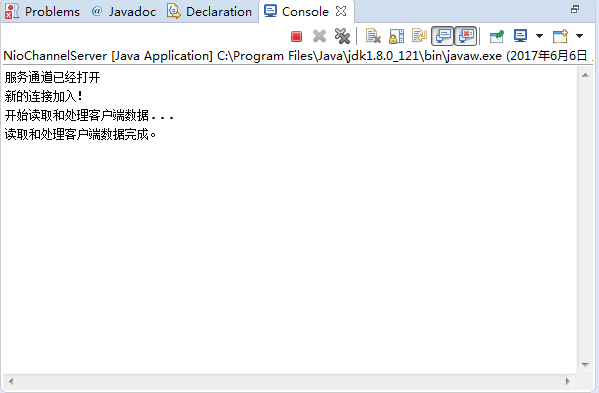
服务端运行结果：



此处等待客户端的运行，此时运行客户端程序，结果如下：



此时服务端控制台已经输出：



### 3.4.2 文件通道案例

打开Eclipse，在JavaNIO项目的根目录下创建一个文本文件a.txt,内容如下：

|  |
| --- |
| 今天是个不好的天气——下雨。  这是一个服务器上面的demo资源。 |

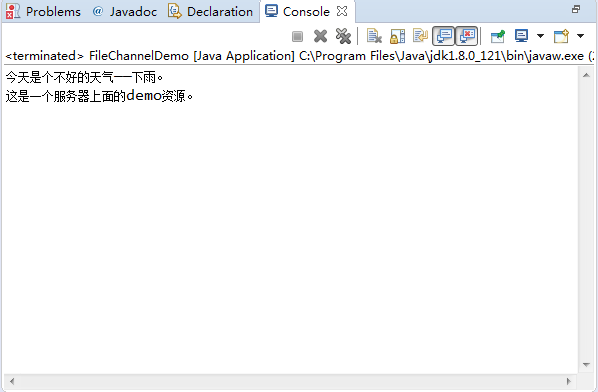
在channel包下创建类FileChannelDemo

代码如下：

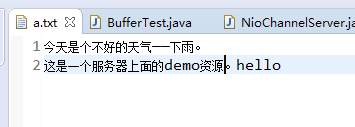
|  |
| --- |
| **package** channel;  **import** java.io.FileInputStream;  **import** java.io.FileNotFoundException;  **import** java.io.FileOutputStream;  **import** java.io.IOException;  **import** java.nio.ByteBuffer;  **import** java.nio.channels.FileChannel;  **public** **class** FileChannelDemo {  **public** **static** **void** main(String[] args) {  *fileChannelDemo*();  }  /\*\*  \* 文件通道  \*/  **public** **static** **void** fileChannelDemo() {  **try** {  // 定义缓冲区对象  ByteBuffer buff = ByteBuffer.*allocate*(1024);  // 通过文件输入流获得文件通道对象(读取操作)  FileChannel inFc = **new** FileInputStream("a.txt").getChannel();    // 追加写入文件 这里的true代表追加  FileChannel outFc = **new** FileOutputStream("a.txt", **true**).getChannel();    // 读取数据  buff.clear();  **int** len = inFc.read(buff);  System.***out***.println(**new** String(buff.array(), 0, len));    // 写数据  ByteBuffer buf2 = ByteBuffer.*wrap*("hello".getBytes());  outFc.write(buf2);    // 关闭通道  outFc.close();  inFc.close();  } **catch** (FileNotFoundException e) {  e.printStackTrace();  } **catch** (IOException e) {  e.printStackTrace();  }  }  } |

运行结果：

控制台打印如下内容



文本文件的内容如下



说明追加文件内容成功

# 4 选择器

## 4.1 选择器基础

通道处于就绪状态后，就可以在缓冲区之间传送数据。可以采用非阻塞模式来检查通道是否就绪，但非阻塞模式还会做别的任务，当有多个通道同时存在时，很难将检查通道是否就绪与其他任务玻璃开来，或者说是这样做很复杂，即使完成了这样的功能，但每检查一次通道的就绪状态，就至少有一次系统调用，代价十分昂贵。当您轮询每个通道的就绪状态时，刚被检查的一个处于未就绪状态的通道，突然处于就绪状态，在下一次轮询之前是不会被察觉的。操作系统拥有这种检查就绪状态并通知就绪的能力，因此要充分利用操作系统提供的服务。在JAVA中，Selector类提供了这种抽象，拥有询问通道是否已经准备好执行每个I/O操作的能力，所以可以利用选择器来很好地解决以上问题。

**流程结构图：**



**选择器（Selector）**

选择器类管理着一个被注册的通道集合的信息和他们的就绪状态。通道是和选择器一起被注册的，并且使用选择器来更新通道的 就绪状态。

可选择通道（SelectableChannel）

这个抽象类提供了通道的可选择所需要的公共方法，FileChannel对象不是可选择的，因为他们没有继承SelectableChannel。所有socket通道都是可选择的。包括从管道（Pipe）对象中获得的通道。SelectableChannel可以被注册到Selector对象上，一个通道可以被注册到多个选择器上，但对每个选择器而言只能被注册一次。

选择键（SelectionKey）

选择键封装了通道与选择器的注册关系。选择键对象被SelectableChannel.register返回并提供一个表示这种注册关系的标记。通道在被注册到一个选择器上之前，必须先设置为非阻塞模式（通过调用configureBloking(false)）。

调用可选择通道的register()方法会将它注册到一个选择器上。如果视图注册一个处于阻塞状态的通道，register()将抛出未检查的IllegalBlockingModeException异常。此外，通道一旦被注册，就不能回到阻塞状态。试图这么做的话，将在调用configureBlocking()方法时将抛出IllegalBlockingModeException异常。并且，试图注册一个已经关闭的SelectableChannel实例的话，也将抛出ClosedChannelException异常。

键的interest（感兴趣的操作）集合和ready（已经准备好的操作）集合是和特定的通道相关的。每个通道的实现，将定义它自己的选择键类。在register()方法中可以构造它并将它传递给所提供的选择器对象。

如果我们要使用非阻塞I/O编写服务器处理程序，大致的步骤如下：

1、向Selector对象注册感兴趣的事件。

2、从Selector中获取感兴趣的事件。

3、根据不同的事件进行相应的处理。

## 4.2 建立选择器

通过以下代码我们可以建立监控三个Socket通道的选择器：

Selector selector = Selector.open();

channel1.register(selector,SelectionKey.OP\_READ);

channel2.register(selector,SelectionKey.OP\_WRITE);

channel3.register(selector,SelectionKey.OP\_READ | SelectionKey.OP\_WRITE);

// Wait up to 10 seconds for a channel to become ready

readyCount = selector.select(10000);

select方法时阻塞方法，直到过了十秒或者至少有一个通道的I/O操作准备好。

## 4.3 使用选择器

如下给出了一些选择器方法：

|  |  |
| --- | --- |
| abstract void close() | 关闭此选择器 |
| abstract Boolean isOpen() | 判断选择器是否已打开 |
| static Selector open() | 打开一个选择器 |
| abstract SelectorProvider provider() | 返回创建此通道的提供者 |
| abstract int select() | 返回一组键的个数，其相应的通道已为I/O操作准备就绪 |
| abstract int select(long timeout) | 同上，指定了阻塞时间 |
| abstract int selectNow() | select()方法的非阻塞形式。不等于select(0)（无限期阻塞） |
| abstract Set keys() | 返回此选择器的键集 |
| abstract Set selectedKeys() | 返回此选择器上相应的通道I/O操作准备就绪的选择键集 |
| abstract Selector wakeup() | 使尚未返回的第一个选择操作立即返回 |

关于选择键的方法：

|  |  |
| --- | --- |
| Object attach(Object ob) | 将给定的对象附加到此键 |
| Object attachment() | 获取当前的附加对象 |
| abstract void cancel() | 请求取消此键的通道到其选择器的注册 |
| abstract Boolean isValid() | 告知此键是否有效 |
| abstract SelectableChannel channel() | 返回与此键相关的通道 |
| abstract Selector selector() | 返回此选择器创建的键 |
| abstract int interestOps() | 获取此键的interest集合，是通道被注册时传进来的值 |
| abstract SelectionKey interestOps(int ops) | 将此键的interest集合设置为给定值 |
| abstract int readyOps() | 获取此键的ready操作集合，为interest的子集 |
| boolean isAcceptable() | 测试此键的通道是否已经准备好接受新的套接字连接 |
| boolean isConnectable() | 测试此键的通道是否已经完成其套接字连接操作 |
| boolean isReadable() | 测试此键的通道是否已经准备好进行读取 |
| boolean isWritable() | 测试此键的通道是否已经准备好进行写入 |

在SelectionKey中，用静态常量定义了四种I/O操作：OP\_READ 1、 OP\_WRITE 4、OP\_CONNECT 8、OP\_ACCEPT 16，这4个值任何2、3、4个相加结果都不相同，因此可以用validOps()方法返回值确定SelectableChannel支持的操作。

当通道关闭时，所有相关的键会自动取消。当选择器关闭时，所有被注册到该选择的通道都将被注销，并且相关的键将立即被无效化（取消）。一旦键被无效化，调用它的与选择器相关的方法就将抛出CancelledKeyException。

前面已经列出了关于选择方法列表，下面介绍基本用法。

通过keys()方法已注册键的集合，集合可能是空的。这个已注册的键的集合不是可以直接修改的，试图这么做的话将java.lang.UnsupportedOperationException。

已选择的键的集合(Selected key set)是已注册的键的集合的子集，这个集合的每个成员都是相关的通道被选择判断为已经准备好的。这个集合通过selectedKeys()方法返回（可能是空的）。

已取消的键的集合（Cancelled key set）是已注册的键的集合的子集。这个集合包含了cancel()方法被调用过的键（这个键已经被无效化），但它们还没有被注销，这个集合是选择器对象的私有成员，因而无法直接访问。

以下三种方式可以唤醒select()方法中睡眠的现成：

a、调用Selector对象的wakeup()方法将使得选择上的第一个还没有返回的选择操作立即返回。如果当前没有在进行中的选择，那么下一次对select()方法的一种形式的调用将立即返回。后续的选择操作将正常进行。在选择操作之间多次调用wakeup()方法与调用它一次没有什么不同。

b、如果选择器的close()方法被调用，那么任何一个在选择操作中阻塞的线程都将被唤醒，就像wakeup()方法被调用了一样。但是与选择器相关的通道将被注销，键将被取消。

c、如果睡眠中的线程interupt()方法被调用，它的返回状态被设置。如果被唤醒的线程之后将试图在通道上执行I/O操作，通道将立即关闭，然后线程将捕捉到一个异常。Selector对象将被捕捉InterruptedException异常并调用wakeup()方法。

## 4.4 选择器案例

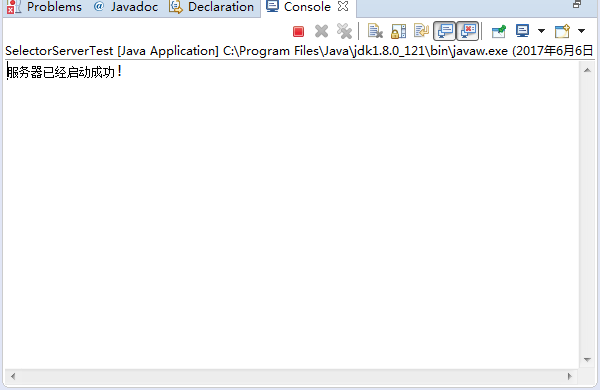
打开Eclipse，在JavaNIO项目创建selector包，并创建类SelectorServerTest，代码如下：

|  |
| --- |
| **package** selector;  **import** java.io.IOException;  **import** java.net.InetSocketAddress;  **import** java.nio.ByteBuffer;  **import** java.nio.channels.SelectionKey;  **import** java.nio.channels.Selector;  **import** java.nio.channels.ServerSocketChannel;  **import** java.nio.channels.SocketChannel;  **import** java.util.Iterator;  **public** **class** SelectorServerTest {  **private** Selector selector;  **private** ServerSocketChannel serverChannel = **null**;  **private** **int** keys = 0;  /\*\*  \* 初始化服务端的连接通道和管理器，已经注册事件  \* **@throws** IOException  \*/  **public** **void** initServer() **throws** IOException {  **this**.selector = Selector.*open*();// SelectorProvider.provider().open();  serverChannel = ServerSocketChannel.*open*();  serverChannel.socket().bind(**new** InetSocketAddress("127.0.0.1", 8888));  serverChannel.configureBlocking(**false**);  // 把serverChannel这个通道注册到通道管理器对象acceptSelector中去，当有客户端连接时触发  SelectionKey key = serverChannel.register(**this**.selector, SelectionKey.***OP\_ACCEPT***);  }  /\*\*  \* 对客户端的请求(通道上面感兴趣的事件)进行监听  \* **@throws** IOException  \*/  **public** **void** listen() **throws** IOException {  System.***out***.println("服务器已经启动成功!");  **while** (**true**) {  // 让通道管理器至少选择一个通道  keys = **this**.selector.select();  Iterator it = **this**.selector.selectedKeys().iterator();  **if** (keys > 0) {  // 进行轮询  **while** (it.hasNext()) {  SelectionKey key = (SelectionKey) it.next();  it.remove();  // 客户端请求连接事件  **if** (key.isAcceptable()) {  serverChannel = (ServerSocketChannel) key.channel();  // 获得和客户端连接的通道  SocketChannel channel = serverChannel.accept();  channel.configureBlocking(**false**);// 设置为非阻塞模式    // 给客户端发送消息  channel.write(ByteBuffer.*wrap*(**new** String("hello client.").getBytes()));  // 还需要读取客户端过来的数据，所以注册一个去读取数据的事件  channel.register(**this**.selector, SelectionKey.***OP\_READ***);  }  **else** **if** (key.isReadable()) {  read(key);  }  }  }  **else** {  System.***out***.println("Select finished without any keys.");  }  }  }  /\*\*  \* 根据SelectionKey对象来读取客户端发送的到通道里面的数据  \* **@param** key  \* **@throws** IOException  \*/  **public** **void** read(SelectionKey key) **throws** IOException {  SocketChannel channel = (SocketChannel) key.channel();  // 缓冲区  ByteBuffer buff = ByteBuffer.*allocate*(1024);  **int** len = channel.read(buff);  String msg = "服务器收到的消息为：" + **new** String(buff.array(), 0, len);  System.***out***.println(msg);  }  /\*\*  \* 启动服务  \*/  **public** **void** start() {  **try** {  SelectorServerTest ns = **new** SelectorServerTest();  ns.initServer();  ns.listen();  } **catch** (IOException e) {  e.printStackTrace();  }  }  **public** **static** **void** main(String[] args) {  **new** SelectorServerTest().start();  }  } |

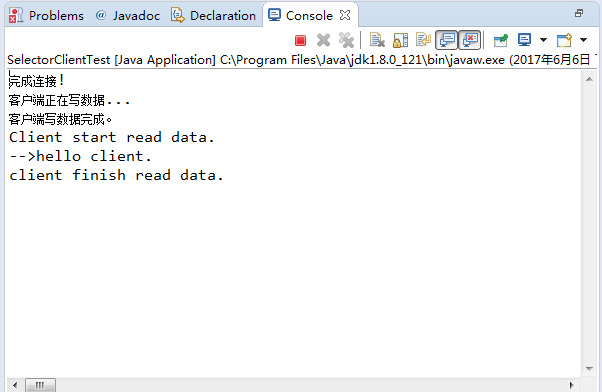
创建类SelectorClientTest，代码如下：

|  |
| --- |
| **package** selector;  **import** java.io.IOException;  **import** java.net.InetSocketAddress;  **import** java.nio.ByteBuffer;  **import** java.nio.channels.SelectionKey;  **import** java.nio.channels.Selector;  **import** java.nio.channels.SocketChannel;  **import** java.util.Iterator;  /\*\*  \* NIO选择器的实例——客户端  \* **@author** lenovo  \*  \*/  **public** **class** SelectorClientTest {  **private** Selector selector;  **private** ByteBuffer outBuff = ByteBuffer.*allocate*(1024);  **private** ByteBuffer inBuff = ByteBuffer.*allocate*(1024);  **private** **int** keys = 0;  **private** SocketChannel channel = **null**;  **public** **void** initClient() **throws** IOException {  // 获得一个socket通道，并没有进行连接  channel = SocketChannel.*open*();  // 获得一个通道管理器  selector = Selector.*open*();  // 设置为非阻塞  channel.configureBlocking(**false**);  // 连接服务器  channel.connect(**new** InetSocketAddress("127.0.0.1", 8888));  // 注册客户端连接服务器的事件  channel.register(**this**.selector, SelectionKey.***OP\_CONNECT***);  }  /\*\*  \* 监听在通道上面进行注册的事件  \* **@throws** IOException  \*/  **public** **void** listen() **throws** IOException {  // 设置轮询  **while** (**true**) {  keys = **this**.selector.select();  **if** (keys > 0) {  // 获得通道管理器事件注册的集合  Iterator<SelectionKey> it = **this**.selector.selectedKeys().iterator();  **while** (it.hasNext()) {  SelectionKey key = it.next();  // 测试此通道是否完成套接字的连接  **if** (key.isConnectable()) {  // 获得与服务器相连的通道  SocketChannel channel = (SocketChannel) key.channel();  // 如果正在连接 就连接完成  **if** (channel.isConnectionPending()) {  channel.finishConnect();  System.***out***.println("完成连接!");  }  channel.register(**this**.selector, SelectionKey.***OP\_WRITE***);  }  // 在通道上面进行写操作  **else** **if** (key.isWritable()) {  SocketChannel channel = (SocketChannel) key.channel();  outBuff.clear();  // outBuff.flip();  System.***out***.println("客户端正在写数据...");  channel.write(outBuff.*wrap*("我是clientA".getBytes()));  channel.register(**this**.selector, SelectionKey.***OP\_READ***);  System.***out***.println("客户端写数据完成。");  }  // 在通道上面进行读取  **else** **if** (key.isReadable()) {  SocketChannel channel = (SocketChannel) key.channel();  inBuff.clear();  System.***out***.println("Client start read data.");  channel.read(inBuff);  System.***out***.println("-->" + **new** String(inBuff.array()));  System.***out***.println("client finish read data.");  }  }  } **else** {  // System.out.println("没有找到感兴趣的事件。");  }  }  }  /\*\*  \* 启动  \*/  **public** **void** start() {  **try** {  initClient();  listen();  } **catch** (IOException e) {  e.printStackTrace();  }  }    **public** **static** **void** main(String[] args) {  **new** SelectorClientTest().start();  }  } |

运行服务器端程序



运行客户端程序：



此时服务端控制台打印：

