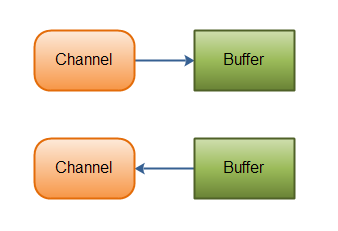
# NIO概述

## 组成部分

1. Java NIO 的核心部分组成Channels、Buffers、Selectors。其它组件，如 Pipe 和 FileLock，只不过是与三个核心组件共同使用的工具类。

## Channel 和 Buffer

1. 基本上，所有的 IO 在 NIO 中都从一个 Channel 开始。Channel 有点像流。 数据可以从 Channel 读到 Buffer 中，也可以从 Buffer 写到 Channel 中。Channel 和 Buffer 都有好几种类型



1. Channel的不同类型（java.nio.channels）

FileChannel：用于读取，写入，映射和操作文件的通道。

DatagramChannel：（UDP连接）面向数据报的套接字的可选通道。

SocketChannel：（TCP连接）用于面向流的可选通道

ServerSocketChannel：（服务器套接字）用于面向流的侦听套接字的可选通道。

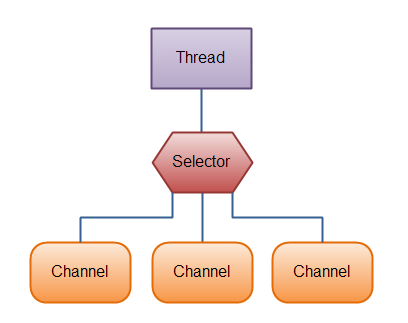
1. Buffer的不同类型（java.nio）

ByteBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer

ShortBuffer这些 Buffer 缓冲区覆盖了你能通过 IO 发送的基本数据类型：byte, short, int, long, float, double 和 char。Java NIO 还有个 MappedByteBuffer，用于表示内存映射文件。

## Selector（java.nio.channels）

1. Selector 允许单线程处理多个 Channel。如果你的应用打开了多个连接（通道），但每个连接的流量都很低，使用 Selector 就会很方便。



1. 作用：

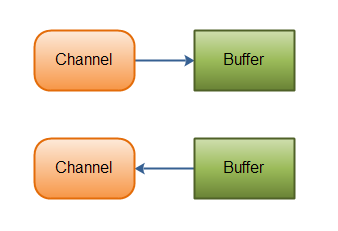
要使用 Selector，得向 Selector 注册 Channel，然后调用它的 select() 方法。这个方法会一直阻塞到某个注册的通道有事件就绪。一旦这个方法返回，线程就可以处理这些事件，事件的例子有如新连接进来，数据接收等。

# Channel

## 通道（Channel）的特点

Java NIO 的通道类似流，但又有些不同：

1. **既可以从通道中读取数据，又可以写数据到通道**。但流的读写通常是单向的。
2. 通道可以**异步**地读写。
3. 通道中的数据总是要先读到一个 Buffer，或者总是要从一个 Buffer 中写入。从通道读取数据到缓冲区，从缓冲区写入数据到通道

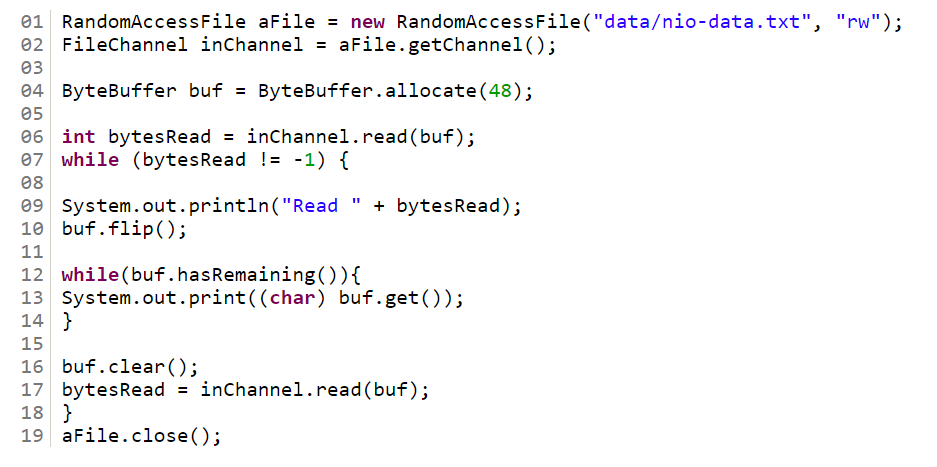


## Channel 的实现

1. FileChannel 从文件中读写数据。
2. DatagramChannel 能通过 UDP 读写网络中的数据。
3. SocketChannel 能通过 TCP 读写网络中的数据。
4. ServerSocketChannel 可以监听新进来的 TCP 连接，像 Web 服务器那样。对每一个新进来的连接都会创建一个 SocketChannel。

## 基本的 Channel 示例

具体见java文件 channel实例



# Buffer

## 本质

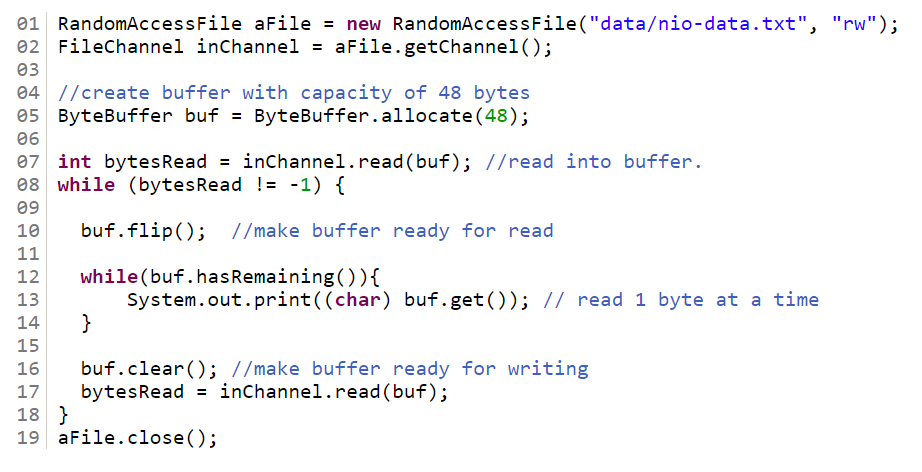
Java NIO 中的 Buffer 用于和 NIO 通道进行交互。如你所知，数据是从通道读入缓冲区，从缓冲区写入到通道中的。

**缓冲区本质上是一块可以写入数据，然后可以从中读取数据的内存。这块内存被包装成 NIO Buffer 对象，并提供了一组方法，用来方便的访问该块内存。**

## Buffer 的基本用法

* 1. 执行步骤

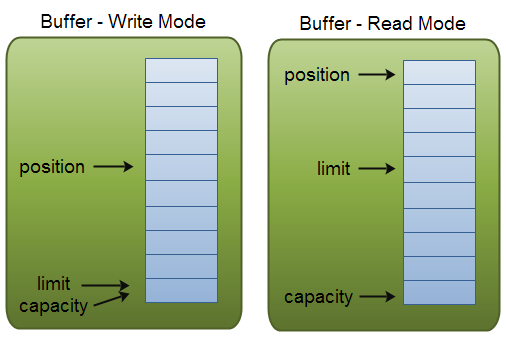
1. 写入数据到 Buffer
2. 调用flip()方法 //重置指针位置
3. 从 Buffer 中读取数据
4. 调用clear()方法或者compact()方法 //清除缓存区内容
   1. 使用注意
5. 当向 buffer 写入数据时，buffer 会记录下写了多少数据。一旦要读取数据，需要通过 flip() 方法将 Buffer 从写模式切换到读模式。在读模式下，可以读取之前写入到 buffer 的所有数据。
6. 一旦读完了所有的数据，就需要清空缓冲区，让它可以再次被写入。有两种方式能清空缓冲区：调用 clear() 或 compact() 方法。**clear() 方法会清空整个缓冲区。compact() 方法只会清除已经读过的数据。**任何未读的数据都被移到缓冲区的起始处，新写入的数据将放到缓冲区未读数据的后面。
   1. 实例代码



## Buffer 的属性

* 1. 简介

1. Buffer有三个属性：capacity、position、limit。
2. position 和 limit 的含义取决于 Buffer 处在读模式还是写模式。不管 Buffer 处在什么模式，capacity 的含义总是一样的。



* 1. 具体介绍

1. Capacity

作为一个内存块，Buffer 有一个固定的大小值，也叫 “capacity”. 你只能往里写 capacity 个 byte、long，char 等类型。一旦 Buffer 满了，需要将其清空（通过读数据或者清除数据）才能继续写数据往里写数据。

1. Position

**写数据到Buffer模式下，position 表示当前的位置。**初始的 position 值为 0. 当一个 byte、long 等数据写到 Buffer 后， position 会向前移动到下一个可插入数据的 Buffer 单元。position 最大可为 capacity – 1.

**从Buffer读数据模式下，也是从某个特定位置读。**当将 Buffer 从写模式切换到读模式，position 会被重置为 0. 当从 Buffer 的 position 处读取数据时，position 向前移动到下一个可读的位置。

1. Limit

在写模式下，Buffer 的 limit 表示你最多能往 Buffer 里写多少数据。 写模式下，limit 等于 Buffer 的 capacity。

**在读模式下， limit 表示你最多能读到多少数据。**因此，当切换 Buffer 到读模式时，limit 会被设置成写模式下的 position 值。换句话说，你能读到之前写入的所有数据（limit 被设置成已写数据的数量，这个值在写模式下就是 position）

## Buffer 的类型

1. ByteBuffer、MappedByteBuffer、CharBuffer、DoubleBuffer、FloatBuffer、IntBuffer、LongBuffer、ShortBuffer

这些 Buffer 类型代表了不同的数据类型。换句话说，就是可以通过 char，short，int，long，float 或 double 类型来操作缓冲区中的字节。

## Buffer 的分配

1. 要想获得一个 Buffer 对象首先要进行分配。 每一个 Buffer 类都有一个 allocate 方法。

具体实例：

ByteBuffer buf = ByteBuffer.allocate(48);

CharBuffer buf = CharBuffer.allocate(1024);

## 向 Buffer 中写数据

* 1. 写数据的两种方式

1. 从 Channel 写到 Buffer，相当于输入到程序中。

int bytesRead = inChannel.read(buf);

1. 通过 Buffer 的 put() 方法写到 Buffer 里。

buf.put(127);

* 1. flip()方法

1. flip 方法将 Buffer 从写模式切换到读模式。调用 flip() 方法会将 position 设回 0，并将 limit 设置成之前 position 的值。

## 从 Buffer 中读取数据

* 1. 读取数据的两种方式

1. 从 Buffer 读取数据到 Channel。

int bytesWritten = inChannel.write(buf);

1. 使用 get() 方法从 Buffer 中读取数据

byte aByte = buf.get();

* 1. Rewind()方法

Buffer.rewind() 将 position 设回 0，**可以重读 Buffer 中的所有数据**。limit 保持不变，仍然表示能从 Buffer 中读取多少个元素（byte、char 等）。

## 常用方法

* 1. clear() 与 compact() 方法

1. 一旦读完 Buffer 中的数据，需要让 Buffer 准备好再次被写入。可以通过 clear() 或 compact() 方法来完成。
2. **如果调用的是 clear() 方法，position 将被设回 0，limit 被设置成 capacity 的值。**换句话说，Buffer 被清空了。**Buffer 中的数据并未清除**，只是这些标记告诉我们可以从哪里开始往 Buffer 里写数据。

如果 Buffer 中有一些未读的数据，调用 clear() 方法，数据将 “被遗忘”，意味着不再有任何标记会告诉你哪些数据被读过，哪些还没有。

1. 如果 Buffer 中仍有未读的数据，且后续还需要这些数据，但是此时想要先先写些数据，那么使用 compact() 方法。

**compact() 方法将所有未读的数据拷贝到 Buffer 起始处。然后将 position 设到最后一个未读元素正后面。limit 属性依然像 clear() 方法一样，设置成 capacity。现在 Buffer 准备好写数据了，但是不会覆盖未读的数据。**

* 1. mark() 与 reset() 方法

通过调用 Buffer.mark() 方法，可以标记 Buffer 中的一个特定 position。之后可以通过调用 Buffer.reset() 方法恢复到这个 position。

buffer.mark();

buffer.reset();

* 1. equals()

当同时满足下列条件时，表示两个 Buffer 相等本质上比较的是buffer中的剩余元素。

1. 有相同的类型（byte、char、int 等）。
2. Buffer 中剩余的 byte、char 等的个数相等。
3. Buffer 中所有剩余的 byte、char 等都相同。
   1. compareTo() 方法

compareTo() 方法比较两个 Buffer 的剩余元素 (byte、char 等)， 如果满足下列条件，则认为一个 Buffer“小于” 另一个 Buffer：

1. 第一个不相等的元素小于另一个 Buffer 中对应的元素 。
2. 所有元素都相等，但第一个 Buffer 比另一个先耗尽 (第一个 Buffer 的元素个数比另一个少)。

# 四．Scatter和Gather

## 1. 简介

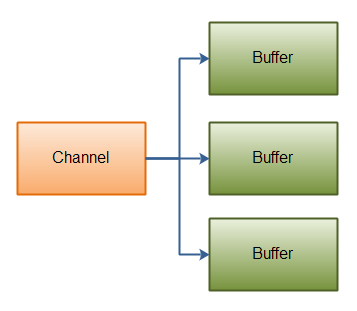
(1) 分散（scatter）从 Channel 中读取是指**在读操作时将读取的数据写入多个 buffer 中**。因此，Channel 将从 Channel 中读取的数据 “分散（scatter）” 到多个 Buffer 中。

(2) 聚集（gather）写入 Channel 是指**在写操作时将多个 buffer 的数据写入同一个 Channel**，因此，Channel 将多个 Buffer 中的数据 “聚集（gather）” 后发送到 Channel。

1. scatter / gather 经常用于需要将传输的数据分开处理的场合，例如传输一个由消息头和消息体组成的消息，将消息体和消息头分散到不同的 buffer 中，这样可以方便的处理消息头和消息体。

## 2. Scattering Reads

(1) Scattering Reads 是指数据从一个 channel 读取到多个 buffer 中。



(2) 实例代码

ByteBuffer header = ByteBuffer.allocate(128);

ByteBuffer body = ByteBuffer.allocate(1024);

ByteBuffer[] bufferArray = { header, body };

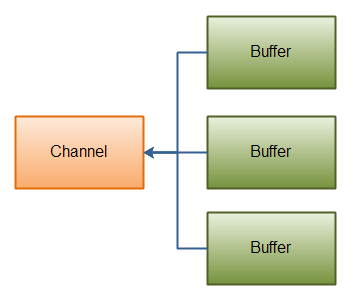
channel.read(bufferArray);

(3) 注意 buffer 首先被插入到数组，然后再将数组作为 channel.read() 的输入参数。read() 方法按照 buffer 在数组中的顺序将从 channel 中读取的数据写入到 buffer，**当一个 buffer 被写满后，channel 紧接着向另一个 buffer 中写**。

1. **Scattering Reads 在移动下一个 buffer 前，必须填满当前的 buffer，这也意味着它不适用于动态消息** (译者注：消息大小不固定)。换句话说，如果存在消息头和消息体，消息头必须完成填充（例如 128byte），Scattering Reads 才能正常工作。

## 3. Gathering Writes

(1) Gathering Writes 是指数据从多个 buffer 写入到同一个 channel。



(2) 代码实例

ByteBuffer header = ByteBuffer.allocate(128);

ByteBuffer body = ByteBuffer.allocate(1024);

//write data into buffers

ByteBuffer[] bufferArray = { header, body };

channel.write(bufferArray);

1. buffers 数组是 write() 方法的入参，**write() 方法会按照 buffer 在数组中的顺序，将数据写入到 channel，注意只有 position 和 limit 之间的数据才会被写入。**因此，如果一个 buffer 的容量为 128byte，但是仅仅包含 58byte 的数据，那么这 58byte 的数据将被写入到 channel 中。因此与 Scattering Reads 相反，Gathering Writes 能较好的处理动态消息。

# 五、通道之间的数据传输

## 简介

在 Java NIO 中，**如果两个通道中有一个是 FileChannel**，那你可以直接将数据从一个 channel传输到另外一个 channel。

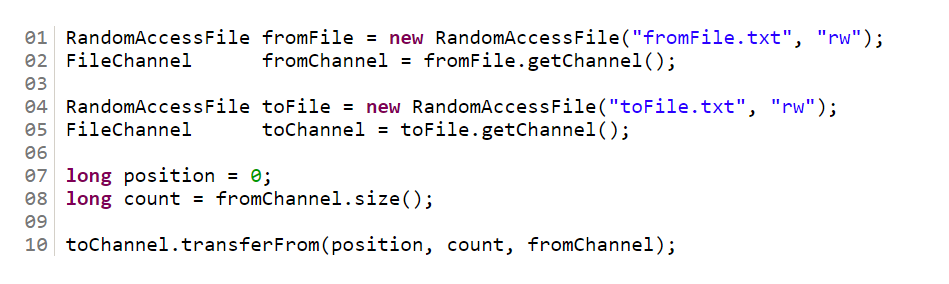
## transferFrom()

FileChannel 的 transferFrom() 方法可以将数据从源通道传输到 FileChannel 中（译者注：这个方法在 JDK 文档中的解释为将字节从给定的可读取字节通道传输到此通道的文件中）。

1. 实例代码(从FileChannel 到 FileChannel)

代码纠正：10行应为

toChannel.transferFrom(fromChannel,position,count)



1. 方法的输入参数

position 表示从 position 处开始向目标文件写入数据，

count 表示最多传输的字节数

如果源通道的所有字节小于 count 个字节，则所传输的字节数要小于请求的字节数。

1. 此外要注意，在 SoketChannel 的实现中，SocketChannel 只会传输此刻准备好的数据（可能不足 count 字节）。因此，SocketChannel 可能不会将请求的所有数据 (count 个字节) 全部传输到 FileChannel 中。

## transferTo()

1. transferTo() 方法将数据从 FileChannel 传输到其他的 channel 中。

和上一个方法一样，例子暂时省略。

# 六、选择器Selector

## 1. 简介

(1) Selector（选择器）是 Java NIO 中能够**检测一到多个 NIO 通道**，并**能够知晓通道是否为诸如读写事件做好准备的组件**。这样，一个单独的线程可以管理多个 channel，从而管理多个网络连接。

1. 仅用单个线程来处理多个 Channels 的好处是，只需要更少的线程来处理通道。事实上，可以只用一个线程处理所有的通道。对于操作系统来说，线程之间上下文切换的开销很大，而且每个线程都要占用系统的一些资源（如内存）。因此，使用的线程越少越好。

## 2. 通道Channel在Selector注册

2.1 Selector 的创建

通过调用 Selector.open() 方法创建一个 Selector

Selector selector = Selector.open();

2.2 向 Selector 注册通道

(1) 为了将 Channel 和 Selector 配合使用，必须将 channel 注册到 selector 上。

**//与 Selector 一起使用时，Channel 必须处于非阻塞模式下。这意味着不能将 FileChannel 与 Selector 一起使用，因为 FileChannel 不能切换到非阻塞模式。而套接字通道都可以。**

channel.configureBlocking(false);

**//register() 方法的第二个参数。这是一个 “interest 集合”，意思是在通过 Selector 监听 Channel 时对什么事件感兴趣。**

SelectionKey key = channel.register(selector,Selectionkey.OP\_READ);

(2) 通道触发了一个事件意思是该事件已经就绪。所以，某个 channel 成功连接到另一个服务器称为 “连接就绪”。一个 server socket channel 准备好接收新进入的连接称为 “接收就绪”。一个有数据可读的通道可以说是 “读就绪”。等待写数据的通道可以说是 “写就绪”。总共有四种事件：Connect、Accept、Read、Write

1. 这四种事件用 SelectionKey 的四个常量来表示：

SelectionKey.OP\_CONNECT

SelectionKey.OP\_ACCEPT

SelectionKey.OP\_READ

SelectionKey.OP\_WRITE

1. 如果你对不止一种事件感兴趣，那么可以用 “位或” 操作符将常量连接起来

int interestSet = SelectionKey.OP\_READ | SelectionKey.OP\_WRITE;

## 3. 选择键SelectionKey

3.1 代表channel注册到selector上的选择键

(1) 当向 Selector 注册 Channel 时，register() 方法会返回一个 SelectionKey 对象。

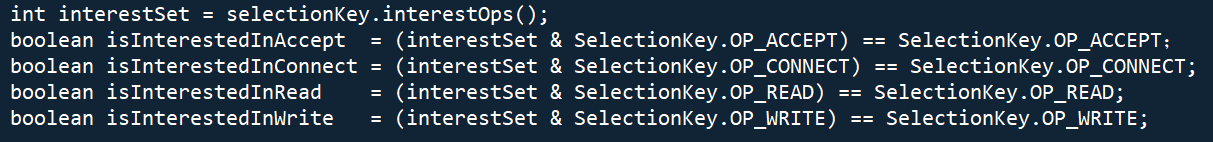
SelectionKey key = channel.register(selector,Selectionkey.OP\_READ);

(2) SelectionKey的属性

interest 集合、ready 集合、Channel、Selector、附加的对象（可选）

* 1. interest 集合

1. interest 集合是你所选择的感兴趣的事件集合，可以通过 SelectionKey 读写 interest 集合。

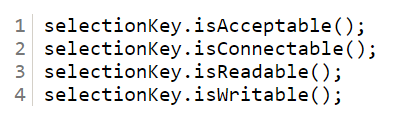


* 1. ready 集合

1. ready 集合是通道已经准备就绪的操作的集合。在一次选择 (Selection) 之后，你会首先访问这个 ready set。

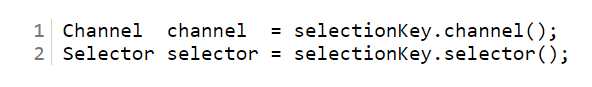
int readySet = selectionKey.readyOps();

1. 可以用像检测 interest 集合那样的方法，来检测 channel 中什么事件或操作已经就绪。但是，也可以使用以下四个方法，它们都会返回一个布尔类型：



* 1. Channel + Selector

1. 从 SelectionKey 访问 Channel 和 Selector 很简单。



* 1. 附加的对象

1. 可以将一个对象或者更多信息附着到 SelectionKey 上，这样就能方便的识别某个给定的通道。
2. 可以附加 与通道一起使用的 Buffer，或是包含聚集数据的某个对象。

//attach将对象附加到selectionKey上

selectionKey.attach(theObject);

//利用attachment取出该对象

Object attachedObj = selectionKey.attachment();

1. 可以在用 register() 方法向 Selector 注册 Channel 的时候附加对象

SelectionKey key = channel.register(selector, SelectionKey.OP\_READ, theObject);

## 通过 Selector 选择通道

4.1 简介

(1) 一旦向 Selector 注册了一或多个通道，就可以调用几个重载的 select()方法。这些方法返回你所感兴趣的事件（如连接、接受、读或写）已经准备就绪的那些通道。换句话说，如果想要读事件就绪的通道，select()方法会返回读事件已经就绪的那些通道。

* 1. select()方法

1. int select()

select()阻塞到至少有一个通道在你注册的事件上就绪了。

select() 方法返回的 int 值表示有多少通道已经就绪。本质是，距离上次调用select()方法后有几个通道准备就绪，比如上次查询出一个，然后有另一个通道准备就绪，此次再查也只有后就绪的这一个。

1. int select(long timeout)

select(long timeout)和 select() 一样，除了最长会阻塞 timeout 毫秒 (参数)。

1. int selectNow()

selectNow()不会阻塞，不管什么通道就绪都立刻返回

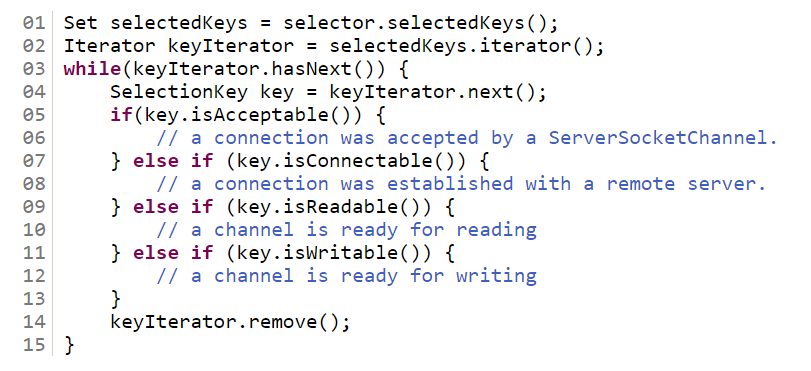
4.2 selectedKeys()方法

(1) 一旦调用了 select()方法，并且返回值表明有一个或更多个通道就绪了，然后可以通过调用 selector 的 selectedKeys()方法，访问 “已选择键集（selected key set）” 中的就绪通道。

Set selectedKeys = selector.selectedKeys();

Iterator<SelectionKey> keys = selector.selectedKeys().iterator();

1. 可以遍历这个已选择的键集合来访问就绪的通道。



1. 注意每次迭代末尾的 keyIterator.remove() 调用。Selector 不会自己从已选择键集中移除 SelectionKey 实例。必须在处理完通道时自己移除。下次该通道变成就绪时，Selector 会再次将其放入已选择键集中。
2. SelectionKey.channel() 方法返回的通道需要转型成你要处理的类型，如 ServerSocketChannel 或 SocketChannel 等。

ServerSocketChannel server = (ServerSocketChannel) key.channel();

* 1. wakeUp()

1. 某个线程调用 select() 方法后阻塞了，即使没有通道已经就绪，也有办法让其从 select() 方法返回。只要让其它线程在第一个线程调用 select() 方法的那个对象上调用 Selector.wakeup() 方法即可。阻塞在 select() 方法上的线程会立马返回。
2. 如果有其它线程调用了 wakeup() 方法，但当前没有线程阻塞在 select() 方法上，下个调用 select() 方法的线程会立即 “醒来（wake up）”。
   1. close()

用完 Selector 后调用其 close() 方法会关闭该 Selector，且使注册到该 Selector 上的所有 SelectionKey 实例无效。通道本身并不会关闭。

## 实例代码

