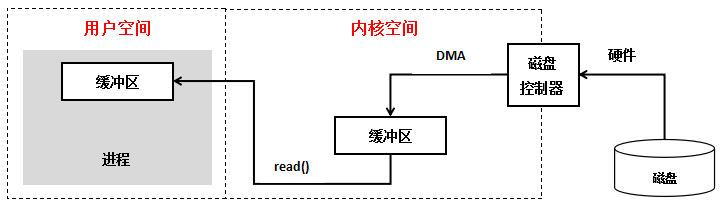
# IO模型



1、linux系统IO分为内核准备数据和将数据从内核拷贝到用户空间两个阶段

## 基本概念

### 用户空间内核空间

1、现在操作系统都是采用虚拟存储器，那么对32位操作系统而言，它的寻址空间(虚拟储存空间)为4G(2的32次方)。操作系统的核心是内核，独立于普通的应用程序，可以访问受保护的内存空间，也有访问底层硬件设备的所有权限。为了保证用户进程不能直接操作内核，保证内核的安全，操作系统将虚拟空间划分为两个部分，一个部分为内核空间，一部分为用户空间。

2、如何分配这两个空间的大小也是有讲究的，如windows 32位操作系统，默认的用户空间：内核空间的比例是1:1；而在32位Linux系统中的默认比例是3:1(3G用户空间，1G内核空间)

3、内核空间和用户空间是现代操作系统的两种工作模式，内核模块运行在内核空间，而用户态应用程序运行在用户空间。它们代表不同的级别，而对系统资源具有不同的访问权限。内核模块运行在最高级别(内核态)，这个级下所有的操作都受系统信任，而应用程序运行在较低级别(用户态)。在这个级别，处理器控制着对硬件的直接访问以及对内存的非授权访问。内核态和用户态有自己的内存映射，即自己的地址空间

4、处理器总处于以下状态中的一种

1. 内核态，运行于进程上下文，内核代表进程运行于内核空间
2. 内核态，运行于中断上下文，内核代表硬件运行于内核空间
3. 用户态，运行于用户空间

5、用户空间的应用程序，通过系统调用，进入内核空间。由内核代表该进程运行于内核空间，这就涉及到上下文的切换，用户空间和内核空间具有不同的地址映射，通用或专用的寄存器组，而用户空间的进程要传递很多变量、参数给内核，内核也要保存用户进程的一些寄存器、变量等，以便系统调用结束后回到用户空间继续执行

6、所谓的"进程上下文"，就是一个进程在执行的时候，CPU的所有寄存器中的值、进程的状态以及堆栈上的内容，当内核需要切换到另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的进程上下文，以便再次执行该进程时，能够恢复切换时的状态，继续执行

### 进程切换

1、为了控制进程的执行，内核必须要有能力挂起正在CPU上运行的进程，并恢复以前挂起的某个进程的执行。这种行为成为进程的切换。任何进程都是在操作系统内核的支持下运行的，是与内核紧密相关的。

2、进程切换的过程，会经过下面这些变化

1. 保存处理机上下文，包括程序计数器和其他寄存器
2. 更新PCB(process control block)信息
3. 将进程的PCB移入相应的队列，如就绪、在某事件阻塞等队列
4. 选择另外一个进程执行，并更新PCB
5. 更新内存管理的数据结构
6. 恢复处理机上下文

3、PCB通常包含如以下的信息：

1. 进程标识符(唯一)
2. 进程当前状态，通常同一状态的进程会被放到同一个队列
3. 进程的程序和数据地址
4. 进程资源清单。列出所拥有的除CPU以外的资源记录
5. 进程优先级。反应进程的紧迫程度
6. CPU现场保护区。记录中断时的CPU状态
7. 进程队列的PCB的链接字
8. 进程相关的其他信息。记账用的，如占用CPU多长时间等

* **另一种版本**

1. 进程标识符(内部，外部)
2. 处理机的信息(通用寄存器，指令计数器，PSW，用户的栈指针)
3. 进程调度信息(进程状态，进程的优先级，进程调度所需的其它信息，事件)
4. 进程控制信息(程序的数据的地址，资源清单，进程同步和通信机制，链接指针)

### 同步、异步、阻塞、非阻塞

1、在讨论这个问题的时候，是需要有具体的上下文的(context)，不同的上下文下，其含义可能不太一致

2、本小节的讨论所基于的上下文：Linux环境下的network IO

3、首先看Stevens给出的定义(POSIX的定义)

A synchronous I/O operation causes the requesting process to be blocked until that I/O operation completes;

An asynchronous I/O operation does not cause the requesting process to be blocked;

* 两者的区别就在于synchronous IO做"IO operation"的时候会将process阻塞。按照这个定义，之前所述的blocking IO，non-blocking IO，IO multiplexing都属于synchronous IO
* 定义中所指的"IO operation"是指真实的IO操作(数据报从内核拷贝到用户空间的过程)

4、**同步与异步(用户是否等待操作完成)**：描述的是用户线程与内核的交互方式，同步指用户线程发起IO请求后需要等待或者轮询内核IO操作完成后才能继续执行；而异步是指用户线程发起IO请求后仍然继续执行，当内核IO操作完成后会通知用户线程，或者调用用户线程注册的回调函数

5、**阻塞与非阻塞(内核在操作完成前是否返回)**：描述是用户线程调用内核IO操作的方式，阻塞是指IO操作需要彻底完成后才返回到用户空间；而非阻塞是指IO操作被调用后立即返回给用户一个状态值，无需等到IO操作彻底完成

## 阻塞IO



1、在这个模型中，应用程序为了执行这个read操作，会调用相应的一个system call，将系统控制权交给内核，然后就进行等待(这个等待的过程就是被阻塞了)，内核开始执行这个system call，执行完毕后会向应用程序返回响应，应用程序得到响应后，就不再阻塞，并进行后面的工作。

2、优点：

1. 能够及时返回数据，无延迟。

3、缺点：

1. 对用户来说处于等待就要付出性能代价。

## 非阻塞IO



1、当用户进程发出read操作时，调用相应的system call，这个system call会立即从内核中返回。但是在返回的这个时间点，内核中的数据可能还没有准备好，也就是说内核只是很快就返回了system call，只有这样才不会阻塞用户进程，对于应用程序，虽然这个IO操作很快就返回了，但是它并不知道这个IO操作是否真的成功了，为了知道IO操作是否成功，应用程序需要主动的循环去问内核。

2、优点：

1. 能够在等待的时间里去做其他的事情。

3、缺点：

1. 任务完成的响应延迟增大了，因为每过一段时间去轮询一次read操作，而任务可能在两次轮询之间的任意时间完成，这对导致整体数据吞吐量的降低。

## 多路复用IO



1、I/O multiplexing 这里面的 multiplexing 指的其实是在**单个线程(内核级线程)**通过记录跟踪每一个Sock(I/O流)的状态来同时管理多个I/O流

2、如果IO多路复用配合Reactor设计模式，可以从select调用的阻塞中解放出来，一旦有sock准备好，来主动通知，这样用户在等待数据准备好之前，可以做自己的事情

3、select

1. select 会修改传入的参数数组，这个对于一个需要调用很多次的函数，是非常不友好的
2. select 如果任何一个sock(I/O stream)出现了数据，select 仅仅会返回，但是并不会告诉你是那个sock上有数据，于是你只能自己一个一个的找，10几个sock可能还好，要是几万的sock每次都找一遍，这个无谓的开销就颇有海天盛筵的豪气了
3. select 只能监视1024个链接，linux将其定义在头文件中，参见FD\_SETSIZE
4. select 不是线程安全的，如果你把一个sock加入到select，然后突然另外一个线程发现，尼玛，这个sock不用，要收回。对不起，这个select 不支持的，如果你丧心病狂的竟然关掉这个sock，select的标准行为是不可预测的， 这个可是写在文档中的哦。

4、poll

1. 去掉了1024个链接的限制
2. poll 从设计上来说，不再修改传入数组
3. 但是poll仍然不是线程安全的， 这就意味着，不管服务器有多强悍，你也只能在一个线程里面处理一组I/O流

5、epoll

1. epoll 现在是线程安全的
2. epoll 现在不仅告诉你sock组里面数据，还会告诉你具体哪个sock有数据，你不用自己去找了

### 剖析

1、场景：假设现在一台服务器监听了n个端口，这n个端口各有一个连接，n个连接都有在进行IO操作

1. 这n个连接必然是在n个线程中的，暂不去考虑是通过线程池来建立新连接还是每个新连接会创建一个新的线程

2、若采用的是同步阻塞的方式



1. 为什么会这么想？由于用户线程被阻塞了，那么谁来执行数据等待以及数据拷贝的操作？那自然是内核线程???

3、若采用的是IO多路复用的方式



1. 每个连接在创建时，都会向一个Selector进行注册，注册其监听的socket，然后调用select方法

## 信号驱动IO



## 异步IO



# Java NIO Tutorial

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

1、Java NIO(New IO)是java 1.4之后出现的新的IO API，意在代替标准Java IO以及Java Networking API

2、Java NIO提供了不同于标准IO的工作方式

## Java NIO:Channels and Buffers

1、在标准Java IO API中，提供的是字节流或者字符流

2、在NIO中，将提供channels以及buffer。数据总是从channel读到buffer，或者从buffer写入channel

## Java NIO:Non-blocking IO

1、Java NIO提供非阻塞的IO方式2

2、例如，一个线程请求从channel读数据到buffer中，在数据从channel读入到buffer的过程中，这个线程可以做其他的事情。一旦数据读入到buffer，线程可以继续处理剩余事情。向channel写入数据也是如此

## Java NIO:Selectors

1、Java NIO保留了一个概念：selectors。一个selector是一个对象，它可以监视多个channel中的事件(例如开启连接，数据到达等等)。因此，只需一个线程即可监视所有的channel

# Java NIO Overriew

1、Java NIO的三个核心组件

1. Channels
2. Buffers
3. Selectors

2、Java NIO类库含有非常多的类以及组件，远超上面提到的三个，但是Channel，Buffer和Selector是构成Java NIO API的核心

## Channels and Buffers

1、通常，在NIO中所有的IO操作起始于Channel。一个Channel就像是一个流，数据可以从Channel读入到Buffer中，当然也可以从Buffer写入到Channel中，如下图所示



2、Channel的实现类有以下几种

1. FileChannel
2. DatagramChannel
3. SocketChannel
4. ServerSocketChannel

* 可以看到，这些channel涵盖了文件IO以及UDP和TCP的网络IO

3、Buffer的实现类有以下几种

1. ByteBuffer
2. CharBuffer
3. DoubleBuffer
4. FloatBuffer
5. IntBuffer
6. LongBuffer
7. ShortBuffer

* 可以看到，这些Buffer涵盖了所有基本数据类型
* 此外还有MappedBytepBuffer，用于内存映射文件，将在后面进行讲解

## Selectors

1、一个Selector允许一个线程来管理多个Channel。如果你的应用有非常多的连接，但是每个连接仅有非常低的流量时，这种方式将非常有用



2、在使用Selector时，我们需要将Channel注册到Selector中，然后调用select()方法，这个方法会阻塞当前线程，直到注册的信道有事件触发。当select()方法返回时，线程可以处理这些事件，事件包括连接、数据到达等

# Java NIO Channel

1、Java NIO Channels与Stream非常相似，但存在以下几点不同

1. 我们可以从Channel读数据也可以从Channel写数据，而一般的字节流或者字符流都是单向的
2. Channels的读写可以是异步的
3. Channels只能从Buffer读入数据(对应的是向Channel写数据)，或者向Buffer写入数据(对应的是从Channel读数据)

2、我们可以从channel中读取数据写入到buffer，或者从buffer取出数据并写入到channel中，如下图所示



## Channel Implementations

1、以下是Channel的实现类

1. FileChannel：从文件读数据/向文件写数据
2. DatagramChannel：通过UDP协议，写入或者读取数据
3. SocketChannel：通过TCP协议，写入或读取数据
4. ServerSoccketChannel：允许我们监听TCP连接，就像一个web server，每一个信的连接都会创建一个SocketChannel

## Basic Channel Example

1. RandomAccessFile aFile = **new** RandomAccessFile("data/nio-data.txt", "rw");
2. FileChannel inChannel = aFile.getChannel();
4. ByteBuffer buf = ByteBuffer.allocate(48);
6. **int** bytesRead = inChannel.read(buf);
7. **while** (bytesRead != -1) {
9. System.out.println("Read " + bytesRead);
10. buf.flip();
12. **while**(buf.hasRemaining()){
13. System.out.print((**char**) buf.get());
14. }
16. buf.clear();
17. bytesRead = inChannel.read(buf);
18. }
19. aFile.close();

# Java NIO Buffer

## Basic Buffer Usage

1、用Buffer读写数据的步骤

1. 将数据写入Buffer
2. 调用buffer.flip()
3. 从Buffer读数据
4. 调用buffer.clear()或者buffer.compact()

2、当你将数据写入Buffer时，buffer必须记录写入的数据量；当你需要从buffer中读取数据时，需要调用flip()方法将buffer的模式从writing模式转为reading模式；在reading模式中，buffer允许你读取上一次写入的所有数据

3、当你读完所有数据后，需要调用clear()方法或者compact()方法将buffer转为writing模式以便于buffer可以再一次被写入数据

* clear()方法清空整个buffer
* compact()方法仅仅清除你已经读取的数据，其余未被读取的数据将会被移动到buffer的开始，数据将会被写入到未读取数据之后

4、简单例子

1. RandomAccessFile aFile = **new** RandomAccessFile("data/nio-data.txt", "rw");
2. FileChannel inChannel = aFile.getChannel();
4. //create buffer with capacity of 48 bytes
5. ByteBuffer buf = ByteBuffer.allocate(48);
7. **int** bytesRead = inChannel.read(buf); //read into buffer.
8. **while** (bytesRead != -1) {
10. buf.flip();  //make buffer ready for read
12. **while**(buf.hasRemaining()){
13. System.out.print((**char**) buf.get()); // read 1 byte at a time
14. }
16. buf.clear(); //make buffer ready for writing
17. bytesRead = inChannel.read(buf);
18. }
19. aFile.close();

## Buffer Cpacity，Position and Limit

1、一个buffer本质上就是一块内存可读可写的内存区域。这块区域被封装成NIO Buffer对象，这个对象提供了一系列方法简化我们对内存块读写的操作

2、一个Buffer有3个重要属性

1. capacity
2. position
3. limit

* posiiton和limit的意义取决于buffer所处的模式(read模式或write模式)
* capacity在不同模式下的意义相同
* 如下图所示



### Capacity

1、对于一个内存块而言，一个buffer一旦初始化完毕，其大小就不再改变，称为capacity，最多只能读取或者写入capacity个数据

### Position

1、当我们向Buffer写入数据时，对于某一时刻，我们总是处于某一个position，在这之前都是已写入的数据，之后都是未写入的区域，每当写入一个数据后，position就自增1

1. position初始化为0，position最大为capacity-1

2、当我们从Buffer写入数据时，对于某一时刻，我们总是处于某一个position，在之前都是已经读取的数据，之后到limit为止都是未读取的数据，每当我们读取一个数据后，position就自增1

3、当我们调用flip方法将buffer从writing模式转为reading模式后，position就被重置为0

### Limit

1、在writing模式下，limit意味着我们能写入buffer的最大数据量，limit与capacity相等

2、当我们调用flip()方法将Buffer从writing模式转为reading模式时，limit表示我们能从buffer读取的最大数据量，因此flip()方法会将limit设置为与position相同的值，这是显然的，写了多少数据你就最多只能读取多少数据

## Buffer Types

1、Buffer类型有如下几种

1. ByteBuffer
2. MappedByteBuffer
3. CharBuffer
4. DoubleBuffer
5. FloatBuffer
6. IntBuffer
7. LongBuffer
8. ShortBuffer

* 以上这些都是抽象类

2、另外还有MappedByteBuffer，这个有点特殊，后面将会介绍

## Allocating a Buffer

1、Buffer对象只能通过每种类型的静态方法allocate来创造

1. ByteBuffer buf = ByteBuffer.allocate(48);
2. CharBuffer buf = CharBuffer.allocate(1024);

## Writing Data to a Buffer

1、我们将数据写入Buffer有两种方式

1. 从Channel将数据写入Buffer中，对应Channel#read()方法
2. 我们将数据直接写入Buffer中，对应Buffer#put()方法，其中put方法会有多种变体
3. **int** bytesRead = inChannel.read(buf); //read into buffer.
4. buf.put(127);

2、只有position到limit之间的空间才能写入数据，因此在写入前需要调用clear或者compact来调整position与limit标志位

## flip()

1. /\*\*
2. \* Flips this buffer.  The limit is set to the current position and then
3. \* the position is set to zero.  If the mark is defined then it is
4. \* discarded.
5. \*
6. \* <p> After a sequence of channel-read or <i>put</i> operations, invoke
7. \* this method to prepare for a sequence of channel-write or relative
8. \* <i>get</i> operations.  For example:
9. \*
10. \* <blockquote><pre>
11. \* buf.put(magic);    // Prepend header
12. \* in.read(buf);      // Read data into rest of buffer
13. \* buf.flip();        // Flip buffer
14. \* out.write(buf);    // Write header + data to channel</pre></blockquote>
15. \*
16. \* <p> This method is often used in conjunction with the {@link
17. \* java.nio.ByteBuffer#compact compact} method when transferring data from
18. \* one place to another.  </p>
19. \*
20. \* @return  This buffer
21. \*/
22. **public** **final** Buffer flip() {
23. limit = position;
24. position = 0;
25. mark = -1;
26. **return** **this**;
27. }

1、通过源码可以看出，并没有存在模式这种字段，一个buffer在任意时刻都是可读可写的

2、flip()方法是的buffer从writing模式转为reading模式，让buffer做好让人读取数据的准备

## Read Data from a Buffer

1、从Buffer读取数据有两种方式

1. 从Buffer中读取数据并写入channel，对应Channel#write()方法
2. 从Buffer中读取数据并为我们所用，对应Buffer#get()方法
3. //read from buffer into channel.
4. **int** bytesWritten = inChannel.write(buf);
5. **byte** aByte = buf.get();

2、只有position

## rewind()

1. /\*\*
2. \* Rewinds this buffer.  The position is set to zero and the mark is
3. \* discarded.
4. \*
5. \* <p> Invoke this method before a sequence of channel-write or <i>get</i>
6. \* operations, assuming that the limit has already been set
7. \* appropriately.  For example:
8. \*
9. \* <blockquote><pre>
10. \* out.write(buf);    // Write remaining data
11. \* buf.rewind();      // Rewind buffer
12. \* buf.get(array);    // Copy data into array</pre></blockquote>
13. \*
14. \* @return  This buffer
15. \*/
16. **public** **final** Buffer rewind() {
17. position = 0;
18. mark = -1;
19. **return** **this**;
20. }

1、这个方法可以让我们重复读取一个Buffer中的内容

## clear() and compact()

1、当我们读取完Buffer的数据后，我们需要将buffer调整到writing模式，可以通过调用clear()或者compact()方法来置为标志位

1. /\*\*
2. \* Clears this buffer.  The position is set to zero, the limit is set to
3. \* the capacity, and the mark is discarded.
4. \*
5. \* <p> Invoke this method before using a sequence of channel-read or
6. \* <i>put</i> operations to fill this buffer.  For example:
7. \*
8. \* <blockquote><pre>
9. \* buf.clear();     // Prepare buffer for reading
10. \* in.read(buf);    // Read data</pre></blockquote>
11. \*
12. \* <p> This method does not actually erase the data in the buffer, but it
13. \* is named as if it did because it will most often be used in situations
14. \* in which that might as well be the case. </p>
15. \*
16. \* @return  This buffer
17. \*/
18. **public** **final** Buffer clear() {
19. position = 0;
20. limit = capacity;
21. mark = -1;
22. **return** **this**;
23. }

2、其中Buffer并没有compact方法，而是存在于Buffer的子类中，例如ByteBuffer，下面是HeapByteBuffer#compact

1. **public** ByteBuffer compact() {
3. System.arraycopy(hb, ix(position()), hb, ix(0), remaining());
4. position(remaining());
5. limit(capacity());
6. discardMark();  //将mark标记为-1
7. **return** **this**;
8. }

3、clear()以及compact()方法将buffer从reading模式转为writing模式，让buffer做好让人写入的准备

## mark() and reset()

1. /\*\*
2. \* Sets this buffer's mark at its position.
3. \*
4. \* @return  This buffer
5. \*/
6. **public** **final** Buffer mark() {
7. mark = position;
8. **return** **this**;
9. }
10. /\*\*
11. \* Resets this buffer's position to the previously-marked position.
12. \*
13. \* <p> Invoking this method neither changes nor discards the mark's
14. \* value. </p>
15. \*
16. \* @return  This buffer
17. \*
18. \* @throws  InvalidMarkException
19. \*          If the mark has not been set
20. \*/
21. **public** **final** Buffer reset() {
22. **int** m = mark;
23. **if** (m < 0)
24. **throw** **new** InvalidMarkException();
25. position = m;
26. **return** **this**;
27. }

1、用法

1. buffer.mark();
3. //call buffer.get() a couple of times, e.g. during parsing.
5. buffer.reset();  //set position back to mark.

## equals() and compareTo()

### equals()

1、equals判断

1. Buffer类型是否相同，ByteBuffer、CharBuffer等
2. 是否含有相同的剩余数据(remaining bytes、char etc)
3. 所有剩余数据都相同

2、Buffer并没有重写Object的equals方法，在其子类，例如ByteBuffer中，才重写了equals方法，ByteBuffer#equals()方法如下

1. /\*\*
2. \* Tells whether or not this buffer is equal to another object.
3. \*
4. \* <p> Two byte buffers are equal if, and only if,
5. \*
6. \* <ol>
7. \*
8. \*   <li><p> They have the same element type,  </p></li>
9. \*
10. \*   <li><p> They have the same number of remaining elements, and
11. \*   </p></li>
12. \*
13. \*   <li><p> The two sequences of remaining elements, considered
14. \*   independently of their starting positions, are pointwise equal.
15. \*   </p></li>
16. \*
17. \* </ol>
18. \*
19. \* <p> A byte buffer is not equal to any other type of object.  </p>
20. \*
21. \* @param  ob  The object to which this buffer is to be compared
22. \*
23. \* @return  <tt>true</tt> if, and only if, this buffer is equal to the
24. \*           given object
25. \*/
26. **public** **boolean** equals(Object ob) {
27. **if** (**this** == ob)
28. **return** **true**;
29. **if** (!(ob **instanceof** ByteBuffer))
30. **return** **false**;
31. ByteBuffer that = (ByteBuffer)ob;
32. **if** (**this**.remaining() != that.remaining())
33. **return** **false**;
34. **int** p = **this**.position();
35. **for** (**int** i = **this**.limit() - 1, j = that.limit() - 1; i >= p; i--, j--)
36. **if** (!equals(**this**.get(i), that.get(j)))
37. **return** **false**;
38. **return** **true**;
39. }
40. /\*\*
41. \* Returns the number of elements between the current position and the
42. \* limit.
43. \*
44. \* @return  The number of elements remaining in this buffer
45. \*/
46. **public** **final** **int** remaining() {
47. **return** limit - position;
48. }

### compareTo()

1、可以用字符串compare的逻辑来理解，先比较对应的字符是否相等，若全部相等，则看哪个长哪个短

1. /\*\*
2. \* Compares this buffer to another.
3. \*
4. \* <p> Two byte buffers are compared by comparing their sequences of
5. \* remaining elements lexicographically, without regard to the starting
6. \* position of each sequence within its corresponding buffer.
7. \* Pairs of {@code byte} elements are compared as if by invoking
8. \* {@link Byte#compare(byte,byte)}.
10. \*
11. \* <p> A byte buffer is not comparable to any other type of object.
12. \*
13. \* @return  A negative integer, zero, or a positive integer as this buffer
14. \*          is less than, equal to, or greater than the given buffer
15. \*/
16. **public** **int** compareTo(ByteBuffer that) {
17. **int** n = **this**.position() + Math.min(**this**.remaining(), that.remaining());
18. **for** (**int** i = **this**.position(), j = that.position(); i < n; i++, j++) {
19. **int** cmp = compare(**this**.get(i), that.get(j));
20. **if** (cmp != 0)
21. **return** cmp;
22. }
23. **return** **this**.remaining() - that.remaining();
24. }

# Java NIO Scatter/Gather

1、Java NIO有内建的scatter/gather的支持。

2、Scattering read操作：从一个channel中读取数据到多个buffer中

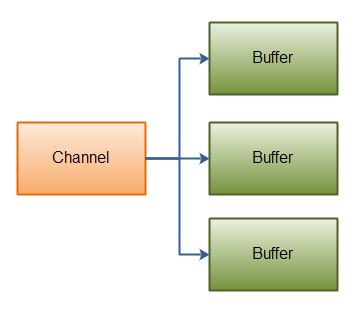
3、Gathering write操作：从多个buffer向一个channel中写入数据

4、Scatter/gather适用于如下场景

1. 我们将数据的处理分成多个部分，例如一个Request，分为Head和Body两部分

## Scattering Reads

1、Scattering read从一个channel中读取数据到多个buffer中

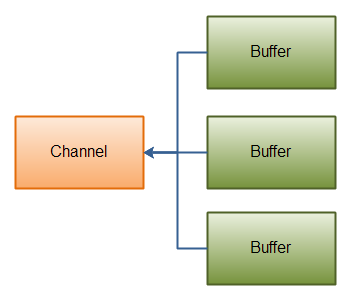


2、用法

1. ByteBuffer header = ByteBuffer.allocate(128);
2. ByteBuffer body   = ByteBuffer.allocate(1024);
4. ByteBuffer[] bufferArray = { header, body };
6. channel.read(bufferArray);
7. channel按照数组中buffer出现的顺序进行写入，当一个buffer写满后，才会写入下一个buffer中
8. 因此如果要将这种性质应用于上述提到的Head以及Body的分离，那么必须保证Head的大小是正确的，因为分块的操作仅仅是依据前一个buffer是否写满

## Gathering Writes

1、Gathering wirate将数据从多个buffer写入到一个channel中



2、用法

1. ByteBuffer header = ByteBuffer.allocate(128);
2. ByteBuffer body   = ByteBuffer.allocate(1024);
4. //write data into buffers
6. ByteBuffer[] bufferArray = { header, body };
8. channel.write(bufferArray);
9. 只有处于position与limit之间的数据才会被写入到channel中
10. gathering write可以很好地处理大小动态变化的信息片段，因为只有有效部分才会被写入channel
11. 这与scatter不太一样，因为从channel中读取数据写入到buffer中时，无法进行信息片段的有效切割，只能依据buffer的大小来进行分隔

# Java NIO Channel to Channel Transfers

1、我们可以直接从一个channel传递数据到另一个channel中，对于FileChannel，提供了transferTo方法以及transferFrom方法

## transferFrom()

1、FileChannel#transferFrom()方法可以将数据从一个source channel传输到当前channel

2、用法

1. RandomAccessFile fromFile = **new** RandomAccessFile("fromFile.txt", "rw");
2. FileChannel      fromChannel = fromFile.getChannel();
4. RandomAccessFile toFile = **new** RandomAccessFile("toFile.txt", "rw");
5. FileChannel      toChannel = toFile.getChannel();
7. **long** position = 0;
8. **long** count    = fromChannel.size();
10. toChannel.transferFrom(fromChannel, position, count);
11. position指的是：destination channel的起始位置，也就是方法调用对象代表的channel
12. count：指的是：最大传输数据(这里是byte)数量，如果fromChannel中的数据数量少于count，那么实际传输的数量以fromChannel总数量为准

3、另外，有些SocketChannel的实现中，可能仅仅传递SocketChannel现有(调用发生时刻)的数据，即便稍后SocketChannel会有更多的数据准备就绪

## transferTo()

1、FileChannel#transferTo()方法将数据传递到destination channel中

2、用法

1. RandomAccessFile fromFile = **new** RandomAccessFile("fromFile.txt", "rw");
2. FileChannel      fromChannel = fromFile.getChannel();
4. RandomAccessFile toFile = **new** RandomAccessFile("toFile.txt", "rw");
5. FileChannel      toChannel = toFile.getChannel();
7. **long** position = 0;
8. **long** count    = fromChannel.size();
10. fromChannel.transferTo(position, count, toChannel);
11. position指的是：source channel的起始位置，也就是方法调用对象代表的channel
12. count指的是：传输的最大数据数量，source channel总数据量小于该值，那么实际传输的数据量就source channel总数据量

# Java NIO Selector

1、Selector是一个Java NIO组件，它可以检查1个或者多个NIO channel，可以确定哪个channel可以读或者写。通过这种方式，一个单线程就可以管理多个channel，也就是可以管理多个网络连接

## Why Use a Selector?

1、使用一个线程去处理多个channel的好处就是我们可以大大缩减处理channel所需要的线程数量。事实上，我们可以用一个线程处理所有channel。因为线程间的切换代价非常高昂，同时每一个线程消耗着操作系统一部分的资源。因此在保证效率的前提下，线程数量越少越好



## Creating a Selector

1、通过Selector.open()方法可以获取一个Selector实例

1. Selector selector = Selector.open();

## Registering Channels with the Selector

1、为了让Selector管理Channel，我们必须将Channel注册到Selector当中，我们必须调用SelectableChannel抽象类的register()方法进行注册

1. channel.configureBlocking(**false**);
3. SelectionKey key = channel.register(selector, SelectionKey.OP\_READ);

* Channel必须使用non-blocking模式，这意味着，我们不能用Selector管理那些不支持non-blocking模式的Channel，例如FileChannel
* SocketChannel支持non-blocking模式
* register的第二个参数是指定感兴趣事件的类型，我们指定以下几种类型

1. Connect：一个channel连接另一个server成功，即"connect ready"
2. Accept：一个server socket channel接受新连接，即"accept ready"
3. Read：一个channel有准备好数据待读取，即"ready ready"
4. Write：一个channel准备好写入数据，即"write ready"

2、上述提到的4种类型分别对应于SelectionKey的四个静态常量

1. SelectionKey.OP\_READ = 1 << 0
2. SelectionKey.OP\_WRITE = 1 << 2
3. SelectionKey.OP\_CONNECT = 1 << 3
4. SelectionKey.OP\_ACCEPT = 1 << 4

* 可以看出将四中类型置于不同的位上

## SelectionKey's

1、一个SelectionKey包含了以下几个重要属性

1. The interest set
2. The ready set
3. The Channel
4. The Selector
5. An attached object(optional)

### Interest Set

1、Interest Set用于描述注册channel时所传入的感兴趣事件的类型，可以通过位运算来判断是否是某种感兴趣的事件

1. **int** interestSet = selectionKey.interestOps();
3. **boolean** isInterestedInAccept  = interestSet & SelectionKey.OP\_ACCEPT;
4. **boolean** isInterestedInConnect = interestSet & SelectionKey.OP\_CONNECT;
5. **boolean** isInterestedInRead    = interestSet & SelectionKey.OP\_READ;
6. **boolean** isInterestedInWrite   = interestSet & SelectionKey.OP\_WRITE;

### Ready Set

1、Ready Set将channle准备就绪的事件置于bit位中，我们可以通过与Interest Set相似的方式来判断channel对于哪类事件准备就绪

2、我们也可以调用方法来判断

1. selectionKey.isAcceptable();
2. selectionKey.isConnectable();
3. selectionKey.isReadable();
4. selectionKey.isWritable();

### Channel+Selector

1、我们可以从SelectionKey对象中获取Channel对象以及Selector对象

1. Channel  channel  = selectionKey.channel();
3. Selector selector = selectionKey.selector();

### Attaching Objects

1、我们可以向SelectionKey添加一个对象，这便于我们识别Channel或者传输更多有效信息，用法如下

1. selectionKey.attach(theObject);
3. Object attachedObj = selectionKey.attachment();

2、我们也可以在注册Channel的同时添加这个额外的对象

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

## Selecting Channels via a Selector

1、当我们向Selector注册过Channel后，我们便可以调用几种Selector#select形式中的一种。这些方法会返回那些对感兴趣事件准备就绪的channel数量，可能是一个也可能是多个

2、select方法的形式

1. int Selector#select()：会阻塞直至有一个channel在感兴趣事件上准备就绪
2. int Selector#select(long timeout)：在设定事件内阻塞直至超时或至少有一个channel在感兴趣事件上准备就绪
3. int Selector#selectNow()：不阻塞，立即返回

* 返回值为int，代表有多少个channel在感兴趣的事件上准备就绪
* That is, how many channels that became ready since last time you called select(). If you call select() and it returns 1 because one channel has become ready, and you call select() one more time, and one more channel has become ready, it will return 1 again. If you have done nothing with the first channel that was ready, you now have 2 ready channels, but only one channel had become ready between each select() call

### selectKeys

1、当我们调用select方法并返回后，返回值指示我们有一个或多个channel准备就绪，我们可以通过selected key set来访问准备就绪的channel，调用方法Selector#selectedKeys

1. Set<SelectionKey> selectedKeys = selector.selectedKeys();

2、当我们向Selector注册channel时，会调用SelectableChannel.register()方法，该方法返回一个SelectionKey对象，这个对象包含了Channel、感兴趣事件类型、Selector等信息。我们也可以通过Selector.selectedKeySet()方法来获取这些SelectionKey

1. Set<SelectionKey> selectedKeys = selector.selectedKeys();
3. Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
5. **while**(keyIterator.hasNext()) {
7. SelectionKey key = keyIterator.next();
9. **if**(key.isAcceptable()) {
10. // a connection was accepted by a ServerSocketChannel.
12. } **else** **if** (key.isConnectable()) {
13. // a connection was established with a remote server.
15. } **else** **if** (key.isReadable()) {
16. // a channel is ready for reading
18. } **else** **if** (key.isWritable()) {
19. // a channel is ready for writing
20. }
22. keyIterator.remove();
23. }

* 注意到keyIterator.remove()调用，Selector不会自动将SelectionKey从selected key集合中删除，我们必须在处理完channel相关过程后，手动删除。当下一次该信道channel再次准备就绪时，Selector会再将这个SelectionKey添加到selected key集合中

3、通过SelectionKey.channel()方法返回的Channel必须转型到恰当的类型，例如ServerSocketChannel以及SocketChannel

## wakeUp

1、调用Selector#select()方法被阻塞的线程可以在没有channel准备继续的情况下返回：通过另一个线程调用Selector.wakeup()方法可以唤醒第一个阻塞在select()方法上的线程

2、值得注意的是wakeup()类似于LockSupport.unpark()，可以获取一次许可

## Full Selector Example

1. Selector selector = Selector.open();
3. channel.configureBlocking(**false**);
5. SelectionKey key = channel.register(selector, SelectionKey.OP\_READ);

8. **while**(**true**) {
10. **int** readyChannels = selector.select();
12. **if**(readyChannels == 0) **continue**;

15. Set<SelectionKey> selectedKeys = selector.selectedKeys();
17. Iterator<SelectionKey> keyIterator = selectedKeys.iterator();
19. **while**(keyIterator.hasNext()) {
21. SelectionKey key = keyIterator.next();
23. **if**(key.isAcceptable()) {
24. // a connection was accepted by a ServerSocketChannel.
26. } **else** **if** (key.isConnectable()) {
27. // a connection was established with a remote server.
29. } **else** **if** (key.isReadable()) {
30. // a channel is ready for reading
32. } **else** **if** (key.isWritable()) {
33. // a channel is ready for writing
34. }
36. keyIterator.remove();
37. }
38. }

# Java NIO FileChannel

# Java NIO SocketChannel

# Java NIO ServerSocketChannel

# Java NIO Non-blocking Server

# Java NIO DatagramChannel

# Java NIO Pipe

# Java NIO vs IO

# Java NIO Path

# Java NIO Files

# Java NIO AsynchronousFileChannel