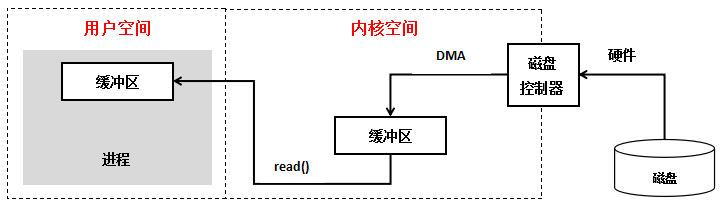
# IO模型



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

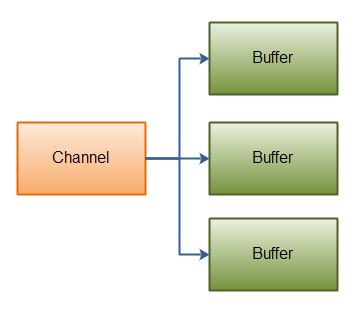
3、Gathering write操作：从多个buffer向一个channel中写入数据

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

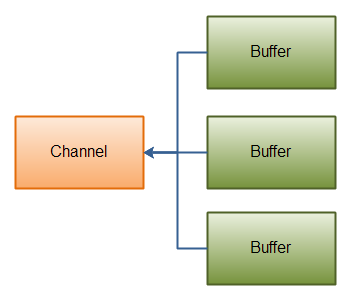


2、用法

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

1、Java NIO FileChannel是一种用于连接文件的channel，通过FileChannel，我们可以向文件写入数据，或者从文件读取数据。作为标准Java IO API的替代方式

2、FileChannel不能设置为non-blocking模式，它只能工作于blocking模式

## Open a FileChannel

1、我们可以通过InputStream、OutputStream或者RandomAccessFile来获取一个FileChannel

1. RandomAccessFile aFile     = **new** RandomAccessFile("data/nio-data.txt", "rw");
2. FileChannel      inChannel = aFile.getChannel();

## Reading Data from a FileChannel

1、调用FileChannel#read()方法可以从FlieChannel中读取数据并存入Buffer中

1. ByteBuffer buf = ByteBuffer.allocate(48);
3. **int** bytesRead = inChannel.read(buf);
4. read方法的返回值代表读入Buffer的数据数量，如果返回-1则代表到达文件尾

## Writing Data to a FileChannel

1、调用FileChannel#write()方法，可以从Buffer中读取数据并写入到FileChannel中

1. String newData = "New String to write to file..." + System.currentTimeMillis();
3. ByteBuffer buf = ByteBuffer.allocate(48);
4. buf.clear();
5. buf.put(newData.getBytes());
7. buf.flip();
9. **while**(buf.hasRemaining()) {
10. channel.write(buf);
11. }
12. 注意到，FileChannel#write()方法置于一个while循环中，因为我们没法保证有write会写入多少数据，因此我们循环调用write方法，直至Buffer没有数据可读
13. 问题：FileChannel不是阻塞的吗???为什么调用一次write不会把所有的Buffer全部写入

## Closing a FileChannel

1、当我们使用FileChannel后，我们必须显式关闭它

1. channel.close();

## FileChannel Position

1、当我们通过FileChannel写入或读取数据时，在任意时刻，一定位于一个确定的位置，我们可以通过FileChannel#position()方法获取当前位置(position)

2、我们可以通过FileChannel#position(long pos)显式设定文件位置

1. **long** pos channel.position();
3. channel.position(pos +123);

3、如果我们将position定位到文件尾以及以后

1. 调用FileChannel#read()将返回-1，因为到达了文件尾
2. 调用FileChannel#write()将产生"file hole"问题，因为position标记的位置可能位于硬盘中的一个gap

## FileChannel Size

1、FileChannel#size()方法将会返回该channel连接文件的大小

1. **long** fileSize = channel.size();

## FileChannel Truncate

1. /\*\*
2. \* Truncates this channel's file to the given size.
3. \*
4. \* <p> If the given size is less than the file's current size then the file
5. \* is truncated, discarding any bytes beyond the new end of the file.  If
6. \* the given size is greater than or equal to the file's current size then
7. \* the file is not modified.  In either case, if this channel's file
8. \* position is greater than the given size then it is set to that size.
9. \* </p>
10. \*
11. \* @param  size
12. \*         The new size, a non-negative byte count
13. \*
14. \* @return  This file channel
15. \*
16. \* @throws  NonWritableChannelException
17. \*          If this channel was not opened for writing
18. \*
19. \* @throws  ClosedChannelException
20. \*          If this channel is closed
21. \*
22. \* @throws  IllegalArgumentException
23. \*          If the new size is negative
24. \*
25. \* @throws  IOException
26. \*          If some other I/O error occurs
27. \*/
28. **public** **abstract** FileChannel truncate(**long** size) **throws** IOException;

## FileChannel Force

1. /\*\*
2. \* Forces any updates to this channel's file to be written to the storage
3. \* device that contains it.
4. \*
5. \* <p> If this channel's file resides on a local storage device then when
6. \* this method returns it is guaranteed that all changes made to the file
7. \* since this channel was created, or since this method was last invoked,
8. \* will have been written to that device.  This is useful for ensuring that
9. \* critical information is not lost in the event of a system crash.
10. \*
11. \* <p> If the file does not reside on a local device then no such guarantee
12. \* is made.
13. \*
14. \* <p> The <tt>metaData</tt> parameter can be used to limit the number of
15. \* I/O operations that this method is required to perform.  Passing
16. \* <tt>false</tt> for this parameter indicates that only updates to the
17. \* file's content need be written to storage; passing <tt>true</tt>
18. \* indicates that updates to both the file's content and metadata must be
19. \* written, which generally requires at least one more I/O operation.
20. \* Whether this parameter actually has any effect is dependent upon the
21. \* underlying operating system and is therefore unspecified.
22. \*
23. \* <p> Invoking this method may cause an I/O operation to occur even if the
24. \* channel was only opened for reading.  Some operating systems, for
25. \* example, maintain a last-access time as part of a file's metadata, and
26. \* this time is updated whenever the file is read.  Whether or not this is
27. \* actually done is system-dependent and is therefore unspecified.
28. \*
29. \* <p> This method is only guaranteed to force changes that were made to
30. \* this channel's file via the methods defined in this class.  It may or
31. \* may not force changes that were made by modifying the content of a
32. \* {@link MappedByteBuffer <i>mapped byte buffer</i>} obtained by
33. \* invoking the {@link #map map} method.  Invoking the {@link
34. \* MappedByteBuffer#force force} method of the mapped byte buffer will
35. \* force changes made to the buffer's content to be written.  </p>
36. \*
37. \* @param   metaData
38. \*          If <tt>true</tt> then this method is required to force changes
39. \*          to both the file's content and metadata to be written to
40. \*          storage; otherwise, it need only force content changes to be
41. \*          written
42. \*
43. \* @throws  ClosedChannelException
44. \*          If this channel is closed
45. \*
46. \* @throws  IOException
47. \*          If some other I/O error occurs
48. \*/
49. **public** **abstract** **void** force(**boolean** metaData) **throws** IOException;

1、FileChannel#force()方法会将数据从channel写入到磁盘dist中。因为操作系统可能会将数据缓存在内存中，因此我们无法确定数据是否已经写入磁盘中了，调用该方法可以显式写入磁盘

1. channel.force(**true**);
2. 其中参数代表是否将文件元数据一并写入磁盘，元数据指的是访问权限，文件类型等信息

# Java NIO SocketChannel

1、Java NIO SocketChannel是用于连接TCP socket的channel，Java NIO相当于Java Networking的套接字

2、创建SocketChannel有两种方式

1. 打开一个SocketChannel，然后连接到Internet中的某服务中
2. 当ServerSocketChannel监听到有新连接时，会创建SocketChannel

## Opening a SocketChannel

1、用法如下

1. SocketChannel socketChannel = SocketChannel.open();
2. socketChannel.connect(**new** InetSocketAddress("http://jenkov.com", 80));

## Closing a SocketChannel

1、当我们使用完SocketChannel后，必须调用SocketChannel#close()方法显示关闭

1. socketChannel.close();

## Reading from a SocketChannel

1、我们可以从SocketChannel中读取数据并写入到Buffer中

1. ByteBuffer buf = ByteBuffer.allocate(48);
3. **int** bytesRead = socketChannel.read(buf);
4. 返回值表示读取的数据数量，当返回-1时表示连接已经关闭

## Writing to a SocketChannel

1、我们可以从Buffer中读取数据并写入到SocketChannel中

1. String newData = "New String to write to file..." + System.currentTimeMillis();
3. ByteBuffer buf = ByteBuffer.allocate(48);
4. buf.clear();
5. buf.put(newData.getBytes());
7. buf.flip();
9. **while**(buf.hasRemaining()) {
10. channel.write(buf);
11. }
12. 注意到，SocketChannel#write()方法置于一个while循环中，因为我们没法保证有write会写入多少数据，因此我们循环调用write方法，直至Buffer没有数据可读

## Non-blocking Mode

1、我们将SocketChannel设置为non-blocking模式

2、在non-blocking模式下，SocketChannel#connect()、read()、write()将会处于非阻塞模式

### connect()

1、如果SocketChannel处于非阻塞模式，当我们调用SocketChannel#connect()方法时，无论连接是否建立，方法会直接返回，我们可以通过调用SocketChannel#finishConnect()方法来判断连接是否已经建立

1. socketChannel.configureBlocking(**false**);
2. socketChannel.connect(**new** InetSocketAddress("http://jenkov.com", 80));
4. **while**(! socketChannel.finishConnect() ){
5. //非阻塞模式的意义就在这里，可以在等待的同时做点别的事情，但是与异步还是有区别
6. //wait, or do something else...
7. }

### write()

1、如果SocketChannel处于非阻塞模式，SocketChannel#write()方法将会在调用后直接返回，此时可能任何数据都没有写入到SocketChannel中，因此我们必须在循环中调用write()方法

2、求一个例子

1. byteBuffer.flip();
2. **while** (socketChannel.write(byteBuffer) > 0) {
3. //other process
4. }

### read()

1、如果SocketChannel处于非阻塞模式，SocketChannel#read()方法将会在调用后直接返回，此时可能尚未从SocketChannel中读取任何数据

2、求一个例子

1. byteBuffer.clear();
2. **while** (socketChannel.read(byteBuffer) > 0) {
3. byteBuffer.flip();
4. //process byteBuffer
5. byteBuffer.clear();
6. }

### Non-blocking Mode with Selectors

1、直接在非阻塞模式下调用SocketChannel#write、read方法并不好用，但是配合Selector，这种模式能够很好地工作

# Java NIO ServerSocketChannel

1、Java NIO ServerSocketChannel是用于监听TCP连接的channel，就像标准Java Networking的ServerSocket一样

1. ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
3. serverSocketChannel.socket().bind(**new** InetSocketAddress(9999));
5. **while**(**true**){
6. SocketChannel socketChannel =
7. serverSocketChannel.accept();
9. //do something with socketChannel...
10. }

## Opening a ServerSocketChannel

1. ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();

## Closing a ServerSocketChannel

1. serverSocketChannel.close();

## Listening for Incoming Connections

1、调用ServerSocketChannel#accept()方可以开启监听连接功能，当accept方法返回时，它将返回一个SocketChannel代表这个新连接。因此accpet方法会阻塞直至新连接到达

2、我们可以在循环中调用accept()方法，这样可以使得ServerSocketChannel一直处于工作状态，不只监听一次连接

1. **while**(**true**){
2. SocketChannel socketChannel =
3. serverSocketChannel.accept();
5. //do something with socketChannel...
6. }

## Non-blocking Mode

1、ServerSocketChannel可以被设置为non-blocking模式

2、在non-blocking模式下，accept()方法在调用后将直接返回，此时可能返回null(代表没有新连接)，因此我们需要判断返回的SocketChannel是否为null来判断是否有新连接

1. ServerSocketChannel serverSocketChannel = ServerSocketChannel.open();
3. serverSocketChannel.socket().bind(**new** InetSocketAddress(9999));
4. serverSocketChannel.configureBlocking(**false**);
6. **while**(**true**){
7. SocketChannel socketChannel =
8. serverSocketChannel.accept();
10. **if**(socketChannel != **null**){
11. //do something with socketChannel...
12. }
13. }

# Java NIO Non-blocking Server

1、即便我们理解了Java NIO的non-blocking特性(Selector，Channel，Buffer等)，设计一个非阻塞服务器仍然是一项艰巨的任务

2、相比于阻塞IO，non-blocking IO面临着几个重要的挑战

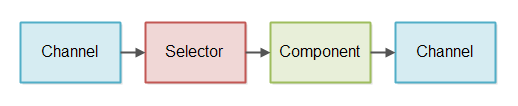
3、同时找到设计non-blocking的相关信息也是非常难的，下面将要介绍的non-blocking server将基于Jakob Jenkov个人的理解

## Non-blocking Server-GitHub Repository

https://github.com/jjenkov/java-nio-server

## Non-blocking IO Pipelines

1、non-blocking IO pipeline如下



1. 上述操作包括在non-blocking模式下的读写操作
2. 一个component利用一个Selector来检查Source Channel是否可读
3. 当可读时，component从Source Channel读取数据，然后根据输入产生一些输出
4. 输出将会写入到Destination Channel

2、一个non-blocking IO pipeline并不需要同时可读可写，有些pipelines仅需要读数据，而另一些pipeline仅需要写入数据

3、上面的示例仅仅含有一个component，一些non-blocking IO pipeline可能含有多个component来处理输入数据

4、一些non-blocking IO pipeline可能同时会从多个Channel中读取数据，例如从多个SocketChannel中读取数据

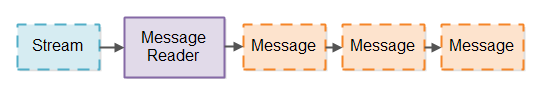
5、上面的实例将整个流程简化了，在图中，这个component是一个通过Selector从channel中读取数据的组件。事实上，并不是channel将数据通过Selector传入component

The flow of control in the above diagram is also simplified. It is the component that initiates the reading of data from the Channel via the Selector. It is not the Channel that pushes the data into the Selector and from there into the component, even if that is what the above diagram suggests.

## Non-blocking vs. Blocking IO Pipelines

1、non-blocking和blocking IO的最大区别是：数据如何从底层Channel中读取

2、IO pipelines通常从stream中读取数据(例如从socket或file)然后将数据分成若干连续的message。现在将这个用于分割数据的组件称为Message Reader，下面是示意图



3、blocking IO pipeline可以使用InputStream接口(每次从底层Channel读取一个字节)，并且InputStream可以阻塞直至数据准备好。这种特性将决定了blocking Message Reader的实现方式

4、使用blocking IO接口可以大大简化Message Reader的实现，blocking Message Reader不需要考虑以下情况

1. 当数据没有到达stream
2. 只有部分message到达流，因此数据解析需要延迟

5、类似的，使用blocking IO接口可以简化Message Write(将多个message组合成一个stream的组件)的实现，blocking Message Write不需要考虑以下情况

1. 当只有部分message到达时，数据合并操作将需要延迟

### Blocking IO Pipeline Drawbacks

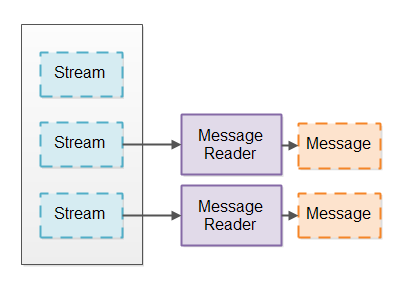
1、尽管blocking Message Reader易于实现，但是其缺点也是显而易见的，对于每个stream，必须开启一个thread来进行split或merge操作。其原因是因为blocking IO会阻塞当前线程直至数据可读，这就意味着一个单线程在阻塞时不能从其他stream中读取数据

2、如果这种blocking IO pipeline作为服务器的一个组件来处理大量的并发连接，那么对于每个连接，服务器必须启动一个新的线程来进行IO操作

1. 对于连接数量只有几百的服务器而言，这种方式不失为一种合理选择
2. 但是对于拥有百万量级并发连接的服务器而言，这种方式就显得很不合理

* 在Java虚拟机中，每个线程会消耗320K(32bit)和1024K(64bit)的资源(程序计数器、Java虚拟机栈、本地方法栈，暂不考虑运行时Java堆的消耗)。因此那么多的线程会消耗大量的内存资源，显然是不合理的

3、为了降低线程数量，大多数服务器维护一个线程池，线程池从任务队列中提取任务进行IO操作，如下图所示



3、然而，采用线程池虽然可以降低线程数量，但是仍然会出现严重问题，如果线程池中的所有线程所处理的IO任务被阻塞，那么其他IO任务将不会得到处理，直至线程池有一个空闲线程

4、可以通过设定弹性的线程数量来解决这个问题(ThreadPoolExecutor)，例如线程池中的所有核心线程都忙碌，那么就开启新的线程来接受新的任务，但是这样做仍然会造成上述提到的两个问题，并不能从本质上解决

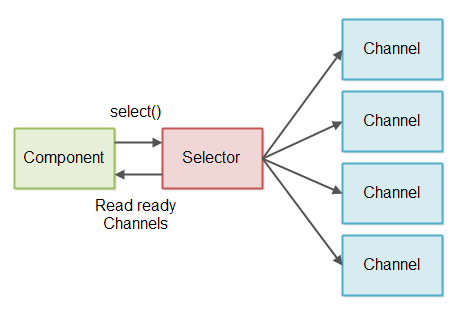
Some server designs try to mitigate this problem by having some elasticity in the number of threads in the thread pool. For instance, if the thread pool runs out of threads, the thread pool might start more threads to handle the load. This solution means that it takes a higher number of slow connections to make the server unresponsive. But remember, there is still an upper limit to how many threads you can have running. So, this would not scale well with 1.000.000 slow connections.

## Basic Non-blocking IO Pipeline Design

1、一个non-blocking IO pipeline可以利用一个单线程来从多个stream中读取数据。这要求stream必须可以切换到non-blocking模式。处于non-blocking模式中，stream可能返回0或更多bytes

1. 返回0表示无任何数据可读
2. 返回1+表示有这么多数据可读

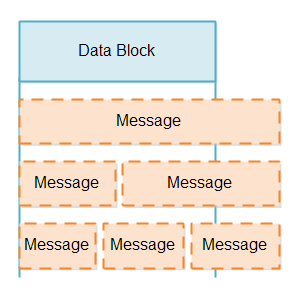
2、为了避免循环检查返回值，我们采用Java NIO Selector。一个或多个SelectableChannel实例可以被注册到Selector中。当我们调用Selector的select()或者selectNow()时，将返回可读的SelectableChannel，设计示意图如下



## Reading Partial Messages

1、当我们从SelectableChannel中读取数据到数据块(data block)中时，我们并不知道数据块含有的数据量多余或是少于一个message，如下图所示

1. 一个数据块可能只能包含了一个message的部分，因为数据块大小小于message的长度
2. 一个数据块可能包含了一个完整的message，因为数据块大小不小于message的长度



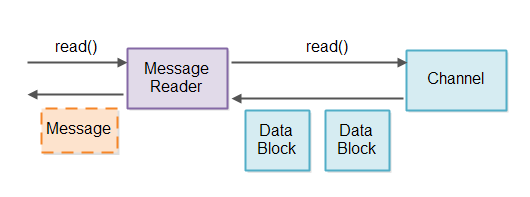
2、在处理partial message时，将面临以下挑战

1. 检测当前data block中是否包含了整个message(detecting full message)
2. 在message的剩余部分到达之前，我们持有的partial message能做什么(storing partial message)

3、我们可以通过Message Reader来检测数据块中的数据是否包含至少一个完整的message，如果数据块包含了一个或多个完整message，那么这些message就可以被送往pipeline下一个步骤进行处理。这些检查操作将会循环执行，且必须效率很高

4、当一个数据块中仅包含了一个partial message时，这个partial message需要被存储下来，以便该message的剩余部分到达时可以进行合并操作

5、detecting full message和storing message将由Message Reader来完成，为了避免混淆不同Channel的data，我们将为每个Channel生成一个Message Reader实例，如下图所示



1. 当从Selector获取了一个可读数据的Channel实例后，MessageReader负责从Channel中读取数据并且将其分割成数个message。如果有任何full message读取完毕，那么这些full message将会被送到pipeline的下一个处理环节进行处理
2. Message Reader必须是协议相关的，因为Message Reader读取数据需要知道数据格式，这样才能检测message是否完整。另外，如果协议相关的配置信息能够作为Message Reader构造的参数，那么就可以实现配置对象的复用，Message Reader通过该传入的配置对象来获取数据相关格式信息

## Storing Partial Messages

1、我们必须将接受到的partial message存储起来，直至整个full message接受完毕，我们必须考虑partial message存储的实现方式

2、以下是两个设计因素

1. 我们想要尽可能少地拷贝数据，因为拷贝的越多，性能越差
2. 我们想让full message存储在连续的字节序列中，这样可以让消息的解析(parsing message)更加容易

### A Buffer Per Message Reader

1、显而易见，partial message需要被存储在buffer(缓存)中。一个简单的思路就是为每个MessageReader创建一个buffer，但是问题是这个buffer应该有多大？它至少应该比最大的Message要更大，如果存在数量非常大的连接，那么MessageReader占用的内存资源将会非常多

### Resizable Bufferss

1、另一种方案就是实现一种resizable buffer，即每个Message Reader含有一个大小可变的buffer。大小可变的buffer有多种可实现的方式，每一种各有优劣，下面将详细讨论

### Resize by Copy

1、在这种实现方式下，一个resizable buffer初始化大小较小，例如4KB，当message的大小大于buffer时，buffer进行自动扩容，扩容时会拷贝已有数据

2、resize-by-copy的优势：一个message的所有数据都被连续存储在字节数组中，这使得数据解析变得容易

3、resize-by-copy的劣势：会导致大量的数据拷贝

4、为了减少数据拷贝，我们必须精心设计buffer的大小，以减小扩容时拷贝的次数，例如大多数request/response小于4KB，那么第一级数量就设为4KB，另一部分Message大于4KB但是小于128KB，一般是文件，第二级大小就为128KB。另外还有一部分Message大于128KB，那么第三级大小就调整为Message的大小即可

5、一旦一个message接收完毕，那么buffer将会调整到最初的大小，便于下一个message使用

### Resize by Append

1、另一种实现resizable-buffer的方式是让一个buffer包含多个数组，当需要进行resize操作时，就分配一个新的数组，并将数据写入这个新的buffer中

2、有两种方式来实现resize by append

1. 一种方法是分配单独的字节数组并保留这些字节数组的列表
2. 另一种方法是分配较大的共享字节数组的片段，然后保留分配给缓冲区的片段的列表

3、resize by append的优势：没有数据会被拷贝

4、resize by append的劣势：Message数据的存储并不连续，会导致数据解析变得困难

### TLV Encoded Messages

1、一些协议消息格式使用TLV格式(类型，长度，值)进行编码。这意味着，当message到达时，message的长度等信息将会被记录在message的头部，我们立即能得到message的大小，因此可以分配与message大小相同的buffer

2、TLV编码使得内存分配变得非常简单，我们能够立即得知message的大小，并分配相等大小的buffer，不会造成buffer的浪费

3、TLV Encoded Messages的劣势：由于我们在一开始就分配了与message大小一致的buffer，一些slow connection会发送一些非常大的消息，这会使得服务器的内存在一开始就被分配完，但是仅仅只有少部分是真正被利用的

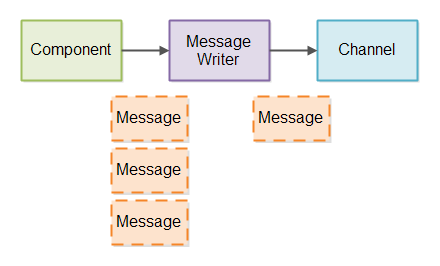
4、一种解决方案是设定超时时间，例如10-15秒。这使得服务器能够从一些极端情况中恢复，但是仍然会有一段时间无法响应。另外服务器也容易受到DoS(Denial of Service)攻击

## Writing Partial Messages

1、如何在non-blocking IO pipeline中写入数据同样是一个挑战，当我们在non-blocking模式下调用Channel#write()时，无法保证有多少个字节写入到了ByteBuffer中。write()方法返回写入字节的数量，因此保存写入字节数量是非常有必要的(用于确定数据写入完毕???)

2、为了实现partial messages的写入，我们同样为每个Channel创建一个Message Writer，用于记录当前message共有多少byte已经写入到channel中

3、如果同时有多个message到达Message Writer，那么这些message必须在Message Writer中进行排队，并依次写入Channel，如下图所示



4、For the Message Writer to be able to send messages that were only partially sent earlier, the Message Writer needs to be called from time to time, so it can send more data.

5、考虑下面一种情况：我们有非常多的连接，因此会有非常多的Message Writer实例，检查一百万个Message Writer实例看是否能够写入数据是非常慢的。首先，大多数Message Writer没有数据需要传输。我们没有必要检查所有Message Writer。其次，不是所有的Channel都准备好接收数据(可以向Channel写入数据)，我们不想浪费时间尝试将数据写入无法接受任何数据的Channel

6、检查一个Channel是否可写，可以借助于Selector，我们只要向Selector注册Channel监听感兴趣的事件即可。但是，我们不必注册所有的Channel。假设，我们有一百万个连接，其中大多数都处于空闲状态，但是这一百万个Channel全部注册到Selector中。当我们调用select()方法，那么大多数Channel都会处于write-ready状态，因此我们将调用Selector#selectedKeys并通过SelectionKey获取关联的Channel，然后检查Channel关联的Message Writer是否有数据需要写入

6、**为了避免检查所有的Message Writer实例，我们采用以下two-step策略**

1. 当消息写入到Message Writer中，Message Writer将其关联的Channel注册到Selector中(如果尚未注册的话)
2. 服务器将会通过Selector获取所有处于write-ready状态的Channel，遍历这些Channel，并取出其关联的Message Writer，通过Message Writer将数据写入到Channel中去，然后将Channel从Selector中注销

## Putting it All Together

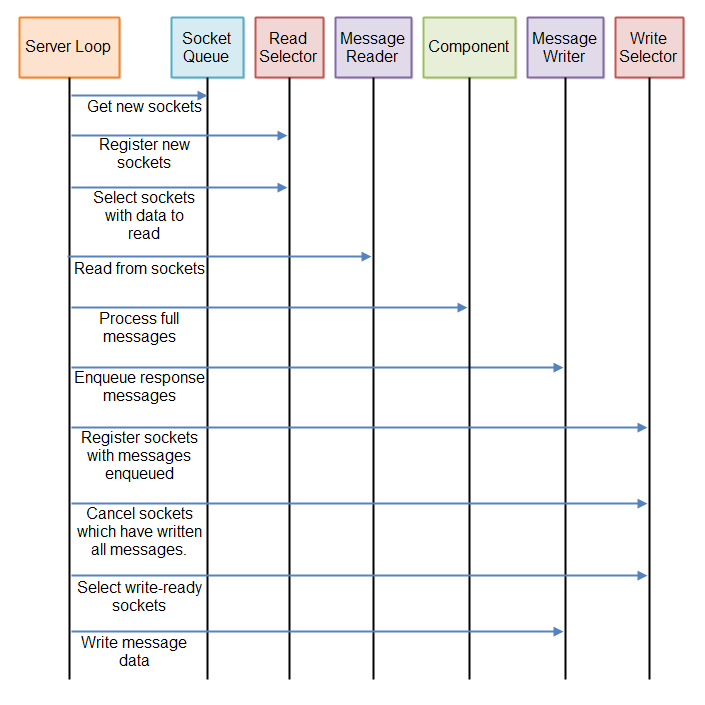
1、正如我们所见，一个non-blocking服务器需要时不时检查输入数据，判断是否有一个完整的message接受完毕。服务器可能需要检查多次直至一个或多个full message接受

2、同样地，一个non-blocking服务器需要时不时检查，判断是否有数据需要写入Channel，如果是，那么服务器还需要检查是否有相关的连接的Channel已经处于write-ready状态。仅仅在一个message第一次入队时检查一次是不够的，因为这个message可能会被分段写入

3、总之，一个non-blocking服务器需要定期执行三个pipelines

1. The read pipeline which checks for new incoming data from the open connections.
2. The process pipeline which processes any full messages received.
3. The write pipeline which checks if it can write any outgoing messages to any of the open connections.
4. 一个read pipeline，用于从打开的连接(Channel)检查新的输入数据
5. 一个process pipeline，用于处理(业务逻辑相关)接收到的数据
6. 一个write pipeline，用于从打开的连接(Channel)写入数据

4、这三个pipeline在一个循环中重复执行。如果没有处于正在排队的message，我们可以省略Write pipeline；如果么有新的full message到达，我们可以省略process pipeline。以下是流程图



# Java NIO DatagramChannel

1、Java NIO DatagramChannel用于发送UDP数据包，因为UDP不是面向连接的网络协议，我们不能像其他channel那样定义read和write方法

## Opening a DatagramChannel

1. DatagramChannel channel = DatagramChannel.open();
2. channel.socket().bind(**new** InetSocketAddress(9999));

1、这里与SocketChannel不同，SocketChannel通过connect方法开启连接，由于UDP不是面向连接的，因此发送数据前可以不建立连接

## Receiving Data

1、通过DatagramChannel#receive()方法读取channel中的数据

1. ByteBuffer buf = ByteBuffer.allocate(48);
2. buf.clear();
4. channel.receive(buf);

2、receive方法会读取channel中的数据并写入到Buffer中，如果接受数据包包含了多余Buffer容量的数据，那么多余的数据将被直接丢弃

## Sending Data

1、通过DatagramChannel#send()方法向channel中写入数据

1. String newData = "New String to write to file..."
2. + System.currentTimeMillis();
4. ByteBuffer buf = ByteBuffer.allocate(48);
5. buf.clear();
6. buf.put(newData.getBytes());
7. buf.flip();
9. **int** bytesSent = channel.send(buf, **new** InetSocketAddress("jenkov.com", 80));

* 在上面这个例子中，向jenkov.com服务器的80端口发送一段数据，因为服务器没有在监听，因此什么都不会发生
* 我们不会察觉到数据包是否被接受，因为UDP不保证数据的传输

## Connecting to a Specific Address

1、DatagramChannel可以连接到网络中一个确定的地址，由于UDP并不是面向连接的协议，因此这种方式的连接并不创建一个真正的连接(这一点与TCP的SocketChannel不同)。DatagramChannel的这种连接可以锁定DatagramChannel，使得我们只能从这个确定地址接受或者发送数据

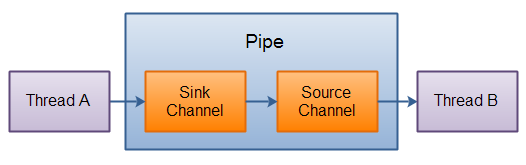
1. channel.connect(**new** InetSocketAddress("jenkov.com", 80));

2、"连接后"，我们可以调用DatagramChannel#read()或write()方法，就像之前那些Channel一样，但是仍然不能保证数据传输的可靠性

1. **int** bytesRead = channel.read(buf);
2. **int** bytesWritten = channel.write(buf);

# Java NIO Pipe

1、Java NIO Pipe是一个建立在线程之间的单向数据连接，一个Pipe拥有source channel和sink channel，可以向sink Channel写入数据，可以从source channel写入数据，模型如下



## Creating a Pipe

1、通过Pipe.open()打开一个Pipe

1. Pipe pipe = Pipe.open();

## Writing to a Pipe

1、为了向Pipe写入数据，我们需要通过pipe对象获取一个sink channel

1. Pipe.SinkChannel sinkChannel = pipe.sink();

2、然后就与正常Channel一样，我们可以通过write()方法向sink channel写入数据

1. String newData = "New String to write to file..." + System.currentTimeMillis();
3. ByteBuffer buf = ByteBuffer.allocate(48);
4. buf.clear();
5. buf.put(newData.getBytes());
7. buf.flip();
9. **while**(buf.hasRemaining()) {
10. sinkChannel.write(buf);
11. }

## Reading from a Pipe

1、为了从Pipe读取数据，我们需要通过pipe对象获取一个source channel

1. Pipe.SourceChannel sourceChannel = pipe.source();

2、然后就与正常Channel一样，我们可以通过read()方法从source channel读取数据

1. ByteBuffer buf = ByteBuffer.allocate(48);
3. **int** bytesRead = inChannel.read(buf);

# Java NIO vs IO

1、当我们学习完Java NIO和Java IO API后，有一个问题：何时我们使用IO，何时使用NIO

## Main Differences Betwen Java NIO and IO

1、IO和NIO的主要区别如下

1. IO面向Stream，NIO面向Buffer
2. IO是阻塞的，NIO**可以是**非阻塞的
3. NIO有Selectors

## Stream Oriented vs. Buffer Oriented

1、Java IO和Java NIO第一个区别就是IO是面向Stream的，NIO是面向Buffer的，但这是什么意思呢？

2、Java IO面向Stream是指：在同一时刻我们可以从一个stream读取一个或多个byte，读取多少byte取决于我们，这些读取的字节不在任何地方进行缓存，我们无法在一个stream进行前向或者后向移动，如果我们需要执行移动的操作，则我们需要自己进行缓存

2、Java NIO面向Buffer是指：读取的数据首先会被存入一个buffer便于后续处理，我们可以在这个缓存中自由地前向或后向移动，这一点为后续数据处理提供了非常大的灵活性。但是我们必须注意到，我们需要检查buffer中是否包含了一个完整的数据段(有意义的数据总是分段的)，并且我们不能覆盖那些我们尚未处理的数据

## Blocking vs. Non-blocking IO

1、Java IO是阻塞的，这意味着，当一个线程调用read()或write()方法时，线程会被阻塞直至数据读写完毕，在此期间，此线程不能做任何事

2、Java NIO是可以是非阻塞的，当处于非阻塞模式下时，一个线程调用read()方法从channel中读取数据，该方法会立即返回，并得知当前可读的数据数量。如果没有数据可读，此线程还能做其他事情，而IO则会阻塞直至可读

3、对于非阻塞模式下的write()方法也是如此，一个线程调用write()方法向channel中写入数据，如果返回的结果表明没有数据可写，则该线程可以做其他事情，而IO则会阻塞直至可写

4、What threads spend their idle time on when not blocked in IO calls, is usually performing IO on other channels in the meantime. That is, a single thread can now manage multiple channels of input and output.

## Selectors

1、Java NIO Selector允许一个单线程来监视多个channel的输入，我们可以将channel注册到selector中，然后调用Selector#select()方法来获取read-ready、write-ready等的channel

## How NIO and IO Influences Application Design

1、关于Java NIO和Java IO的选择可能会影响应用设计的以下几个方面

1. API可能不同
2. 处理数据的流程不同
3. 处理数据的线程数量不同

### The API Calls

1、显然Java IO和Java NIO的API是不同的，对于Java IO，我们从Inputstream一个个地读取字节；对于Java NIO，我们从Channel中读取的数据首先要存入Buffer中

### The Processing of Data

1、Java IO和Java NIO处理数据的流程是不同的

2、给定数据如下

Name: Anna

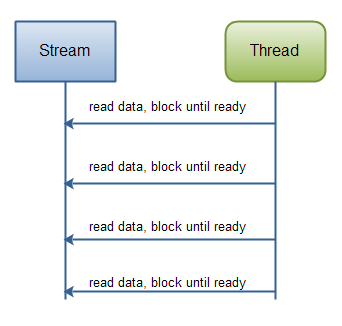
Age: 25

Email: anna@mailserver.com

Phone: 1234567890

3、Java IO的操作过程

1. InputStream input = ... ; // get the InputStream from the client socket
3. BufferedReader reader = **new** BufferedReader(**new** InputStreamReader(input));
5. String nameLine   = reader.readLine();
6. String ageLine    = reader.readLine();
7. String emailLine  = reader.readLine();
8. String phoneLine  = reader.readLine();
9. 注意到，处理状态取决于代码执行的地方。例如，当我们第一次调用read.readLine()方法返回时，我们可以确定第一行已经被读取了，readLine()会阻塞直至第一行完全被读取，所以我们知道这一行就代表name。类似地，第二次调用readLine()，我们知道这一行包含的就是age
10. 正如我们所见，程序尽在数据被读取时才会进行处理，因此在每一步，我们确切地知道读取的数据是什么
11. 模型如下图所示



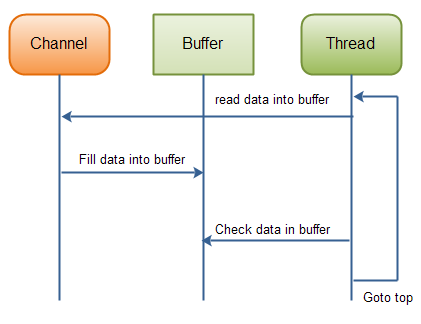
4、Java NIO操作过程如下

1. ByteBuffer buffer = ByteBuffer.allocate(48);
3. **int** bytesRead = inChannel.read(buffer);

* 第二行从channel中读取数据并存入ByteBuffer中，当方法返回时，我们不能确定channel中的所有数据是否已经读取完毕。我们仅仅知道ByteBuffer中包含了一些字节，这使得数据处理变得有点困难
* 考虑以下情况，当第一次调用read(buffer)方法时，只有半行数据读入了ByteBuffer中，例如"Name:An"。此时我们无法处理数据，我们需要等到一行数据全部被读入ByteBuffer中才行，在此之前，我们无法处理数据
* 那么我们如何知道buffer中包含的数据是否包含了一个full message(进行数据处理的最小单元)，遗憾的是，并不知道。我们只能通过查看处于buffer中的数据来判断，前提是我们必须事先知道数据格式
* 如果在数据可以处理前，我们不得不对buffer中的数据进行多次检查，那么这将变得非常低效，例如

1. ByteBuffer buffer = ByteBuffer.allocate(48);
3. **int** bytesRead = inChannel.read(buffer);
5. **while**(! bufferFull(bytesRead) ) {
6. bytesRead = inChannel.read(buffer);  //会直接返回，因为是非阻塞模式，可能信道中的数据未全部读入buffer中
7. }

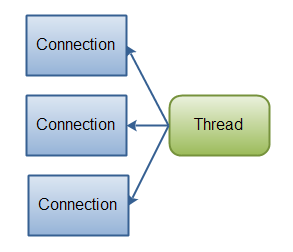
* bufferFull()方法必须记录读入到buffer中的字节数量，当读满时返回true，否则返回false
* 需要注意的是，bufferFull会扫描ByteBuffer，扫描的过程中会改变position的值，因此当扫描完成后，必须还原position的位置。否则会出现未定义的错误
* 当buffer包含了一个full message时，这个message才能被处理，否则只能等待message完全到达



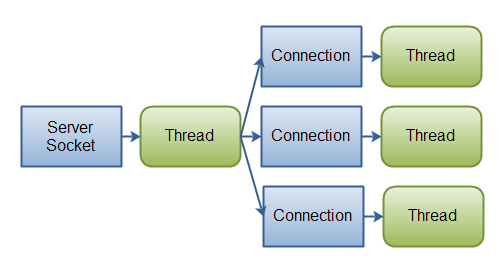
## Summary

1、NIO允许我们仅用单个或者少量线程管理多个channel(网络连接或文件(FileChannel不支持非阻塞的，那这里的文件指什么???))，但这种方式会使得数据的处理变得更加复杂，尤其是读取数据的时候(如何读取有意义的Message，不同的场景有不同的做法，即依赖具体的文本协议)

2、如果需要管理数量非常大的连接，而且这些连接在同一时刻仅有少量处于活跃状态，那么使用NIO会带来非常大的增益，模型如下图所示



3、如果需要管理少量连接，但是这些连接具有非常大的数据量，那么传统的IO就是一种非常合适的方案，模型如下图所示



# Java NIO Path

1、Java Path在Java 7中被添加到了Java NIO库中。Java Path接口现在位于java.nio.file中

2、一个Java Path代表了文件系统中的一个路径，可以代表一个文件或者一个文件夹，可以是绝对或者相对路径。绝对路径是指从文件系统根路径开始的full path，而相对路径是指从某个路径开始的relative path。相对路径更加复杂

3、Java Path接口与操作系统的path环境变量无任何关系，不要搞混

4、在某种程度上，java.nio.file.Path类似于java.io.File，它们之间存在着细微的差别，在很多情况下，我们可以用Path代替File来用

## Creating a Path Instance

1、我们通过一个静态方法Paths.get()来获取java.nio.file.Path的实例

1. **import** java.nio.file.Path;
2. **import** java.nio.file.Paths;
4. **public** **class** PathExample {
6. **public** **static** **void** main(String[] args) {
8. Path path = Paths.get("c:\\data\\myfile.txt");
10. }
11. }

* 其中Paths.get()是一个工厂方法

### Creating an Absolute Path

1、用Paths.get()方法来获取Path的实例

1. Path path = Paths.get("c:\\data\\myfile.txt");

* 这里绝对路径是"c:\data\myfile.txt"，双斜线"\\"在Java String中必须的，因为"\"在Java中是转义字符，表示其后面的字符作为一般字符来看待(固定搭配除外)，因此"\\"代表将一个"\"写入字符串
* 上面的写法是针对于Windows操作系统的，对于Unix操作系统，用以下路径表示方式

1. Path path = Paths.get("/home/jakobjenkov/myfile.txt");

* 这里绝对路径是"/home/jakobjenkov/myfile.txt"

### Creating a Relative Path

1、相对路径是指基于某个base path的路径，因此绝对路径为base path以及relative path的组合

2、Java NIO支持相对路径，我们可以通过以下方式来创建一个相对路径，同样调用Path.get()方法，不过是另一个重载版本，接受两个参数，一个base path，一个relative path

1. Path projects = Paths.get("d:\\data", "projects");
3. Path file     = Paths.get("d:\\data", "projects\\a-project\\myfile.txt");

3、支持"."与".."

## Path.normalize()

1、Path#normalize()方法可以除去路径中的所有"."和".."，下面是例子

1. String originalPath =
2. "d:\\data\\projects\\a-project\\..\\another-project";
4. Path path1 = Paths.get(originalPath);
5. System.out.println("path1 = " + path1);
7. Path path2 = path1.normalize();
8. System.out.println("path2 = " + path2);

# Java NIO Files

1、Java NIO Files(java.nio.file.Files)提供了多个方法用于操作多个文件

2、java.nio.file.Files配合java.nio.file.Path进行工作

## Files.exists()

1、Files#exists()方法用于判断一个Path是否在文件系统中存在

2、这意味着，允许生成一个Path，其代表的路径在文件系统中并不存在

1. Path path = Paths.get("data/logging.properties");
3. **boolean** pathExists =
4. Files.exists(path,
5. **new** LinkOption[]{ LinkOption.NOFOLLOW\_LINKS});

* 第二个参数告诉此方法用何种方式判断Path是否存在，上述例子中，只有path所指向的路径代表一个符号链接文件时，才返回true

## Files.createDirectory()

1、Files.createDirectory()静态方法通过一个Path实例来产生一个代表文件夹的Path的实例

1. Path path = Paths.get("data/subdir");
3. **try** {
4. Path newDir = Files.createDirectory(path);
5. } **catch**(FileAlreadyExistsException e){
6. // the directory already exists.
7. } **catch** (IOException e) {
8. //something else went wrong
9. e.printStackTrace();
10. }

* 如果要文件夹已经存在，则抛出FileAlreadyExistsException异常
* 若有其他问题则抛出IOException异常，例如要创建的文件夹其父文件夹也不存在

## Files.copy()

1、Files.copy()静态方法，将文件从一个path拷贝到另一个path中

1. Path sourcePath      = Paths.get("data/logging.properties");
2. Path destinationPath = Paths.get("data/logging-copy.properties");
4. **try** {
5. Files.copy(sourcePath, destinationPath);
6. } **catch**(FileAlreadyExistsException e) {
7. //destination file already exists
8. } **catch** (IOException e) {
9. //something else went wrong
10. e.printStackTrace();
11. }

* 如果目标文件已存在，则抛出java.nio.file.FileAlreadyExistsException
* 其他异常情况则抛出IOException，例如源文件不存在等

### Overwriting Existing Files

1、我们可以通过Files.copy()静态方法重写一个已存在的文件

1. Path sourcePath      = Paths.get("data/logging.properties");
2. Path destinationPath = Paths.get("data/logging-copy.properties");
4. **try** {
5. Files.copy(sourcePath, destinationPath,
6. StandardCopyOption.REPLACE\_EXISTING);
7. } **catch**(FileAlreadyExistsException e) {
8. //destination file already exists
9. } **catch** (IOException e) {
10. //something else went wrong
11. e.printStackTrace();
12. }

## Files.move()

1、Java NIO Files允许将文件移动到另一个路径中，移动一个文件就等于给文件改名，因此该方法可以移动文件或者在同一路径下给文件改名。java.io.File也有改名操作，即renameTo()方法

1. Path sourcePath      = Paths.get("data/logging-copy.properties");
2. Path destinationPath = Paths.get("data/subdir/logging-moved.properties");
4. **try** {
5. Files.move(sourcePath, destinationPath,
6. StandardCopyOption.REPLACE\_EXISTING);
7. } **catch** (IOException e) {
8. //moving file failed.
9. e.printStackTrace();
10. }

* 第三个参数代表允许覆盖，可选

## Files.delete()

1、Files.delete()静态方法可以删除一个文件或文件夹

1. Path path = Paths.get("data/subdir/logging-moved.properties");
3. **try** {
4. Files.delete(path);
5. } **catch** (IOException e) {
6. //deleting file failed
7. e.printStackTrace();
8. }

* 如果被删除的文件不存在，则抛出IOException异常

## Files.walkFileTree()

1、Files.walkFileTree()方法可以遍历一个目录，该方法需要两个参数，第一个参数是Path的实例，用于指定文件夹路径；第二个参数是FileVisitor的实例，用于指定匹配策略

1. **public** **interface** FileVisitor {
3. **public** FileVisitResult preVisitDirectory(
4. Path dir, BasicFileAttributes attrs) **throws** IOException;
6. **public** FileVisitResult visitFile(
7. Path file, BasicFileAttributes attrs) **throws** IOException;
9. **public** FileVisitResult visitFileFailed(
10. Path file, IOException exc) **throws** IOException;
12. **public** FileVisitResult postVisitDirectory(
13. Path dir, IOException exc) **throws** IOException {
15. }

* 以上方法在遍历的不同阶段会被调用
* preVisitDirectory()方法在访问任何一个目录前调用，postVisitDirectory()方法在访问目录后调用
* visitFile()方法在在访问任何一个文件时被调用，visitFileFailed()在访问文件失败时被调用，例如，对于某些文件，我们可能并没有权限
* 所有的方法均返回一个FileVisitResult枚举实例，该实例包含四种可能值

1. CONTINUE：file walk正常继续执行
2. TERMINATE：终止
3. SKIP\_SIBLINGS：继续进行，但是不访问当前文件或文件夹的siblings???
4. SKIP\_SUBTREE：继续进行，但是不访问子目录结构，该值仅仅可能由preVisitDirectory()方法返回

### Searching For Files

1. Path rootPath = Paths.get("data");
2. String fileToFind = File.separator + "README.txt";
4. **try** {
5. Files.walkFileTree(rootPath, **new** SimpleFileVisitor<Path>() {
7. @Override
8. **public** FileVisitResult visitFile(Path file, BasicFileAttributes attrs) **throws** IOException {
9. String fileString = file.toAbsolutePath().toString();
10. //System.out.println("pathString = " + fileString);
12. **if**(fileString.endsWith(fileToFind)){
13. System.out.println("file found at path: " + file.toAbsolutePath());
14. **return** FileVisitResult.TERMINATE;
15. }
16. **return** FileVisitResult.CONTINUE;
17. }
18. });
19. } **catch**(IOException e){
20. e.printStackTrace();
21. }

### Deleting Directories Recursively

1、Files.walkFileTree()可以用于删除一个不为空的目录，Files.delete()仅仅能删除一个空目录

1. Path rootPath = Paths.get("data/to-delete");
3. **try** {
4. Files.walkFileTree(rootPath, **new** SimpleFileVisitor<Path>() {
5. @Override
6. **public** FileVisitResult visitFile(Path file, BasicFileAttributes attrs) **throws** IOException {
7. System.out.println("delete file: " + file.toString());
8. Files.delete(file);
9. **return** FileVisitResult.CONTINUE;
10. }
12. @Override
13. **public** FileVisitResult postVisitDirectory(Path dir, IOException exc) **throws** IOException {
14. Files.delete(dir);
15. System.out.println("delete dir: " + dir.toString());
16. **return** FileVisitResult.CONTINUE;
17. }
18. });
19. } **catch**(IOException e){
20. e.printStackTrace();
21. }

## Additional Methods in the Files Class

1、java.nio.file.Files还有其他强大的功能，例如创建连接文件，判断文件大小，设定文件权限等等，具体参考JavaDoc

# Java NIO AsynchronousFileChannel

1、AsynchronousFileChannel在Java 7被添加到了Java NIO，允许进行异步的读写操作

## Creating an AsynchronousFileChannel

1、通过静态方法打开AsynchronousFileChannel

1. Path path = Paths.get("data/test.xml");
3. AsynchronousFileChannel fileChannel =
4. AsynchronousFileChannel.open(path, StandardOpenOption.READ);

* 第一参数是Path的实例，代表文件路径
* 第二个参数是指定AsynchronousFileChannel允许进行的操作

## Reading Data

1、AsynchronousFileChannel读取数据有两种方式

## Reading Data Via a Future

1、读数据的第一种方式是调用AsynchronousFileChannel#read()方法，该方法返回一个Future

1. Future<Integer> operation = fileChannel.read(buffer, 0);

* 第一参数是ByteBuffer的实例
* 第二个参数是文件中指定的位置
* 该方法会立即返回，无论读操作是否完成，我们可以通过Future#isDone来查看是否完成读操作

2、详细例子

1. AsynchronousFileChannel fileChannel =
2. AsynchronousFileChannel.open(path, StandardOpenOption.READ);
4. ByteBuffer buffer = ByteBuffer.allocate(1024);
5. **long** position = 0;
7. Future<Integer> operation = fileChannel.read(buffer, position);
9. **while**(!operation.isDone());  //等待读完毕，会浪费CPU
11. buffer.flip();
12. **byte**[] data = **new** **byte**[buffer.limit()];
13. buffer.get(data);
14. System.out.println(**new** String(data));
15. buffer.clear();

* 这种方式效率较为低下，其原因是我们不得不等待读取操作完成

## Reading Data Via a CompletionHandler

1、读取数据的第二种方式是调用AsynchronousFileChannel#read()的另一个重载版本，需要传入一个CompletionHandler作为参数(回调)

1. fileChannel.read(buffer, position, buffer, **new** CompletionHandler<Integer, ByteBuffer>() {
2. @Override
3. **public** **void** completed(Integer result, ByteBuffer attachment) {
4. System.out.println("result = " + result);
6. attachment.flip();
7. **byte**[] data = **new** **byte**[attachment.limit()];
8. attachment.get(data);
9. System.out.println(**new** String(data));
10. attachment.clear();
11. }
13. @Override
14. **public** **void** failed(Throwable exc, ByteBuffer attachment) {
16. }
17. });

* 一旦读取操作完成，CompletionHandler#compelted()方法会被调用，其中Integer代表读取的byte数量，attachment是read方法传入的第三个参数，是泛型参数
* 如果读取操作失败，CompletionHandler#failed()方法会被调用

2、方法源码

1. /\*\*
2. \* Reads a sequence of bytes from this channel into the given buffer,
3. \* starting at the given file position.
4. \*
5. \* <p> This method initiates the reading of a sequence of bytes from this
6. \* channel into the given buffer, starting at the given file position. The
7. \* result of the read is the number of bytes read or {@code -1} if the given
8. \* position is greater than or equal to the file's size at the time that the
9. \* read is attempted.
10. \*
11. \* <p> This method works in the same manner as the {@link
12. \* AsynchronousByteChannel#read(ByteBuffer,Object,CompletionHandler)}
13. \* method, except that bytes are read starting at the given file position.
14. \* If the given file position is greater than the file's size at the time
15. \* that the read is attempted then no bytes are read.
16. \*
17. \* @param   <A>
18. \*          The type of the attachment
19. \* @param   dst
20. \*          The buffer into which bytes are to be transferred
21. \* @param   position
22. \*          The file position at which the transfer is to begin;
23. \*          must be non-negative
24. \* @param   attachment
25. \*          The object to attach to the I/O operation; can be {@code null}
26. \* @param   handler
27. \*          The handler for consuming the result
28. \*
29. \* @throws  IllegalArgumentException
30. \*          If the position is negative or the buffer is read-only
31. \* @throws  NonReadableChannelException
32. \*          If this channel was not opened for reading
33. \*/
34. **public** **abstract** <A> **void** read(ByteBuffer dst,
35. **long** position,
36. A attachment,
37. CompletionHandler<Integer,? **super** A> handler);

## Writing Data

1、与AsynchronousFileChannel#read()相同，AsynchronousFileChannel#write()也有两种方式

## Writing Data Via a Future

1、写入数据的第一种方式是调用AsynchronousFileChannel#write()方法，如下

1. Path path = Paths.get("data/test-write.txt");
2. AsynchronousFileChannel fileChannel =
3. AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);
5. ByteBuffer buffer = ByteBuffer.allocate(1024);
6. **long** position = 0;
8. buffer.put("test data".getBytes());
9. buffer.flip();
11. Future<Integer> operation = fileChannel.write(buffer, position);
12. buffer.clear();
14. **while**(!operation.isDone());  //等待写完毕，会浪费CPU
16. System.out.println("Write done");

* 显然这种方式效率低下，
* 如果文件不存在，则会抛出java.nio.file.NoSuchFileException异常，我们可以用如下方式来进行确认

1. **if**(!Files.exists(path)){
2. Files.createFile(path);
3. }

## Writing Data Via a CompletionHandler

1、写入数据的第二种方式是调用AsynchronousFileChannel#write()方法的另一个重载版本，接受四个参数

1. Path path = Paths.get("data/test-write.txt");
2. **if**(!Files.exists(path)){
3. Files.createFile(path);
4. }
5. AsynchronousFileChannel fileChannel =
6. AsynchronousFileChannel.open(path, StandardOpenOption.WRITE);
8. ByteBuffer buffer = ByteBuffer.allocate(1024);
9. **long** position = 0;
11. buffer.put("test data".getBytes());
12. buffer.flip();
14. fileChannel.write(buffer, position, buffer, **new** CompletionHandler<Integer, ByteBuffer>() {
16. @Override
17. **public** **void** completed(Integer result, ByteBuffer attachment) {
18. System.out.println("bytes written: " + result);
19. }
21. @Override
22. **public** **void** failed(Throwable exc, ByteBuffer attachment) {
23. System.out.println("Write failed");
24. exc.printStackTrace();
25. }
26. });

* 写入操作完成时CompletionHandler#completed()方法被调用
* 写入操作失败时CompletionHandler#failed()方法被调用

2、方法源码

1. /\*\*
2. \* Writes a sequence of bytes to this channel from the given buffer, starting
3. \* at the given file position.
4. \*
5. \* <p> This method works in the same manner as the {@link
6. \* AsynchronousByteChannel#write(ByteBuffer,Object,CompletionHandler)}
7. \* method, except that bytes are written starting at the given file position.
8. \* If the given position is greater than the file's size, at the time that
9. \* the write is attempted, then the file will be grown to accommodate the new
10. \* bytes; the values of any bytes between the previous end-of-file and the
11. \* newly-written bytes are unspecified.
12. \*
13. \* @param   <A>
14. \*          The type of the attachment
15. \* @param   src
16. \*          The buffer from which bytes are to be transferred
17. \* @param   position
18. \*          The file position at which the transfer is to begin;
19. \*          must be non-negative
20. \* @param   attachment
21. \*          The object to attach to the I/O operation; can be {@code null}
22. \* @param   handler
23. \*          The handler for consuming the result
24. \*
25. \* @throws  IllegalArgumentException
26. \*          If the position is negative
27. \* @throws  NonWritableChannelException
28. \*          If this channel was not opened for writing
29. \*/
30. **public** **abstract** <A> **void** write(ByteBuffer src,
31. **long** position,
32. A attachment,
33. CompletionHandler<Integer,? **super** A> handler);

# Interrupt block NIO

1、阻塞式的SocketChannel#read()/write()和ServerSocketChannel#accept()会被Thread#interrupt()方法中断，原理是其内部会检测中断标志位，若被中断则抛出ClosedByInterruptException异常(IOException的子类)，这一点与wait、sleep、join方法响应中断时抛出InterruptedException异常不同