

**三種IO簡述與推出的時間軸**

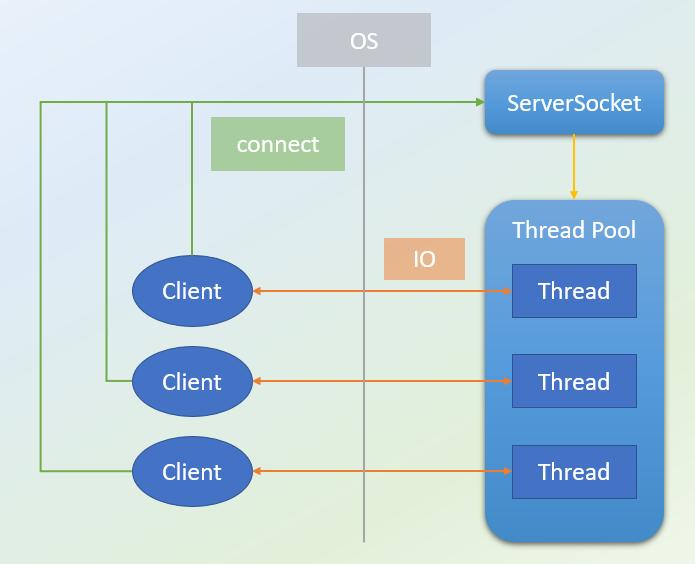
**名詞釋疑**

|  |  |
| --- | --- |
| Synchronous (同步)  Asynchronous (非同步) | 在這裡意義為在執行IO動作時, 是由jvm處理或是交由OS處理來區分. 若為jvm處理則為Synchronous, 否則為Asynchronous. |
| Block(阻塞)  Non-Block(非阻塞) | 依據"進行IO操作時是否需要等待"作為區分. 若操作IO時當下的Thread需要等待則為Block, 否則為Non-Block. |



初探BIO

在jdk 1.4之前java原生所提供的IO操作, 也是三種IO模型裡最簡單、最直覺的, 流程大致上是:

1. 等待連線
2. 取得連線, 分派該連線給其他Thread處理
3. 回頭繼續等待連線

如上圖所示, 可以很清楚看到每一個Client連線將會由一個Thread負責處理, 雖然採用了Thread Pool的機制將減少在建立Thread時花費的硬體資源, 但在面對高併發請求時就很容易榨乾硬體資源, 因為就要有相對應的Thread數量來處理.

不過其實這種client-thread一對一的關係並不是阻礙高併發請求的原因, 真正的原因在於進行IO操作時是Block(阻塞)的, 一旦進行了IO的操作, 當下的Thread會被Block(等待網路層回應), 而在等待期間它啥事都不能做, 就像你的某項工作因為需要別人的回應才能決定下一動, 而這等待期間你卻無法去做其他工作一樣的意思. 這種狀況就造成了效率不佳, 因為Thread的存在需要消耗硬體資源, 但卻常常在偷懶!

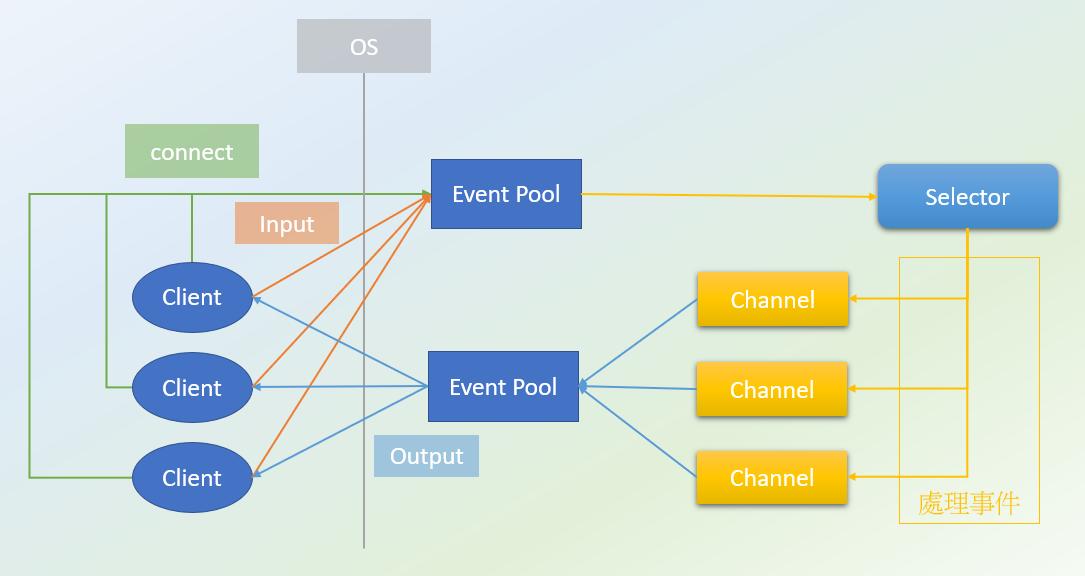
初探NIO

從jdk 1.4開始提供, 雖然官方稱之為New IO, 但這名字並沒有體現這個IO模型的特點, 因此大多數的人仍用Non-Block來理解NIO. 然而NIO主要就是為了解決BIO進行IO操作時Blocak所造成效率不佳的問題。

NIO是基於Reactor(事件驅動), 也就是有IO事件發生時再來處理就好. 因此導入了一個Selector概念, 將連線的發生、IO資料輸入/輸出都視為"事件(Event)", 在此只要操作Selector取出事件作相對應的事情即可. 這概念可以想像成底層將這些"事件(Event)"都塞到了一個queue裡, 然後操作這個Selector不斷的從這個queue取出這些event進行處理即可. 但因為所有的事情都視為event, 所以所有client的連線事件都會混在一起被select出來處理, 因此需要比較複雜的程式判斷才不至於混淆。

操作流程大致上是:

1. 操作selector取出event
2. (1)若是connect事件需要完成TCP三方交握, 完成後會取得一個與這個Client專屬的Channel, 再將此Channel註冊回Selector, Selector將會監聽此Channel Input事件的發生。  
   (2)若是Input事件, 則可以透過ByteBuffer將Input的內容取出, 若當下ByteBuffer不足以取出所有Input內容, 那麼下次select一樣有會此次Channel的Input事件被取出, 直到程式將Input全部取完。  
   (3)上述的操作皆可以使用與select同個Thread做事, 或交由Thread Pool去處理. 若使用同個來Thread處理Input事件, 那概念就會像是Node.js單線程操作一樣, 遇到IO時一樣交由底層所控制的Thread去執行, 當下的Thread可以回來繼續處理業務邏輯.
3. 回頭繼續取出事件



NIO在IO操作時引入了兩個新個概念, "Channel"與"ByteBuffer".

* "Channel": 對應BIO的InputStream, OutputStream. BIO的I/O Stream是單向操作, 而NIO的Channel是雙向操作, 讀/寫均透過此物件.
* "ByteBuffer": NIO在進行IO操作時均要使用ByteBuffer給予Channel. 概念就是要output的內容時會先塞到ByteBuffer, 再丟給Channel去output(底層將調用別的Thread進行output動作); 而Input事件發生時, 底層會先將內容暫存, 我們程式一樣透過ByteBuffer取用. 這是我們程式與底層IO間的重要橋樑, 讓我們可以以非同步狀態操作IO, 進而讓Thread不再因為等待IO而偷懶.

初探AIO

從jdk 1.7開始提供, 又稱為NIO 2.0. 而從網路上其他文章的介紹看來, 主要與NIO最大的差別是IO是否由jvm處理. NIO在處理IO的時候, 仍然是由jvm處理, 也就是NIO底層仍然會調用Thread進行處理; 而AIO在處理IO時是委託OS進行處理, 不占用jvm資源. 而IO委託OS進行處理, 就意味著OS需要支援IO非同步操作, 但現在兩大OS: Linux有[epoll](https://zh.wikipedia.org/wiki/Epoll)機制, Windows有[IOCP](https://zh.wikipedia.org/wiki/IOCP)機制, 所以基本上不用擔心囉.

另外我個人研究完NIO與AIO的程式操作後, 認為另外一個最大的差別就是改進了NIO設計不良的操作...(有些操作真的滿莫名其妙的), 不過jdk 1.4有那時候的時空背景, 用現在的思維去看難免不合時宜, 總之從我們使用IO模型的角度來看(不管模型內部), 在NIO裡接受到事件時交由Thread Pool去做事情, 概念上就如同AIO了.

AIO的操作流程完全基於callback function(也是listener的概念), 在操作接受連線、讀取input等動作時, 是準備一個callback給channel, 在底層準備好這些事情時將會呼叫這些callback相對應的method, 就不需要像NIO一樣需要占用一個Thread不斷select事件來處理. 而在操作output動作時, 則是跟NIO一樣準備好ByteBuffer丟給Channel進行output, 並同時給定一個callback, 底層將在完成output事情後呼叫對應的method.

*由於AIO的模型與NIO模型一樣, 只有模型實作與程式操作的不同, 因此就不在提供AIO模型圖了。*

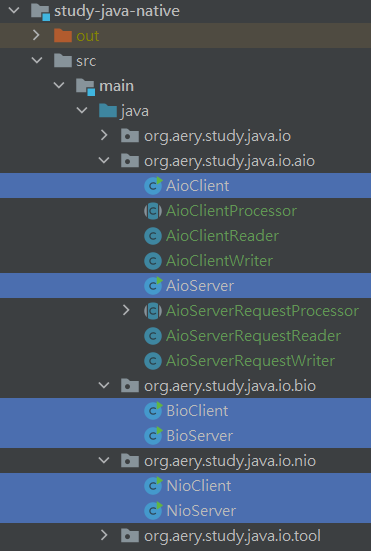
實戰

Code在study-java-native的package:org.aery.study.java.io裡, 請自行服用.

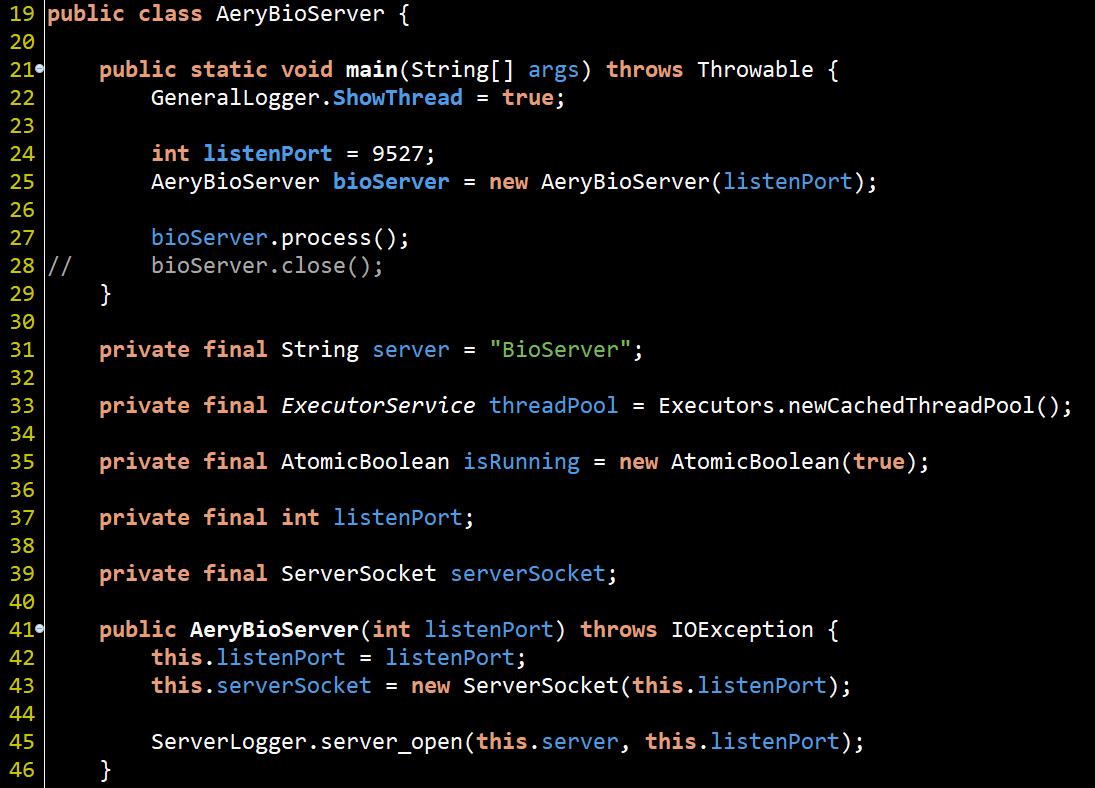
三種IO模型都各會有一個Servert程式一個Client物件來展示如何使用. BIO因為操作簡單, 所以設計server會將收到的訊息由console印出, client也是從console輸入訊息來展示; NIO、AIO較為複雜, 除了server一樣由consle印出訊息外, client則是透過swing開啟兩個視窗, 一個是印出server回應的訊息, 另一個是輸入訊息的視窗, 以此展示. 另外為了方便辨識server上每個client連線, 在server與client建立好連線後, 將會有一個對應的RID(Random ID)來識別.

下面各IO實戰解說只針對server的程式, client的程式碼基本上也是大同小異. 而client在輸入方面基本上是給server啥, server就回應啥, 但有兩個關鍵字有特殊操作:

* "exit": client會主動中斷連線, 結束程式.
* "close": client送出此字串給server之後, server會主動關閉連線.



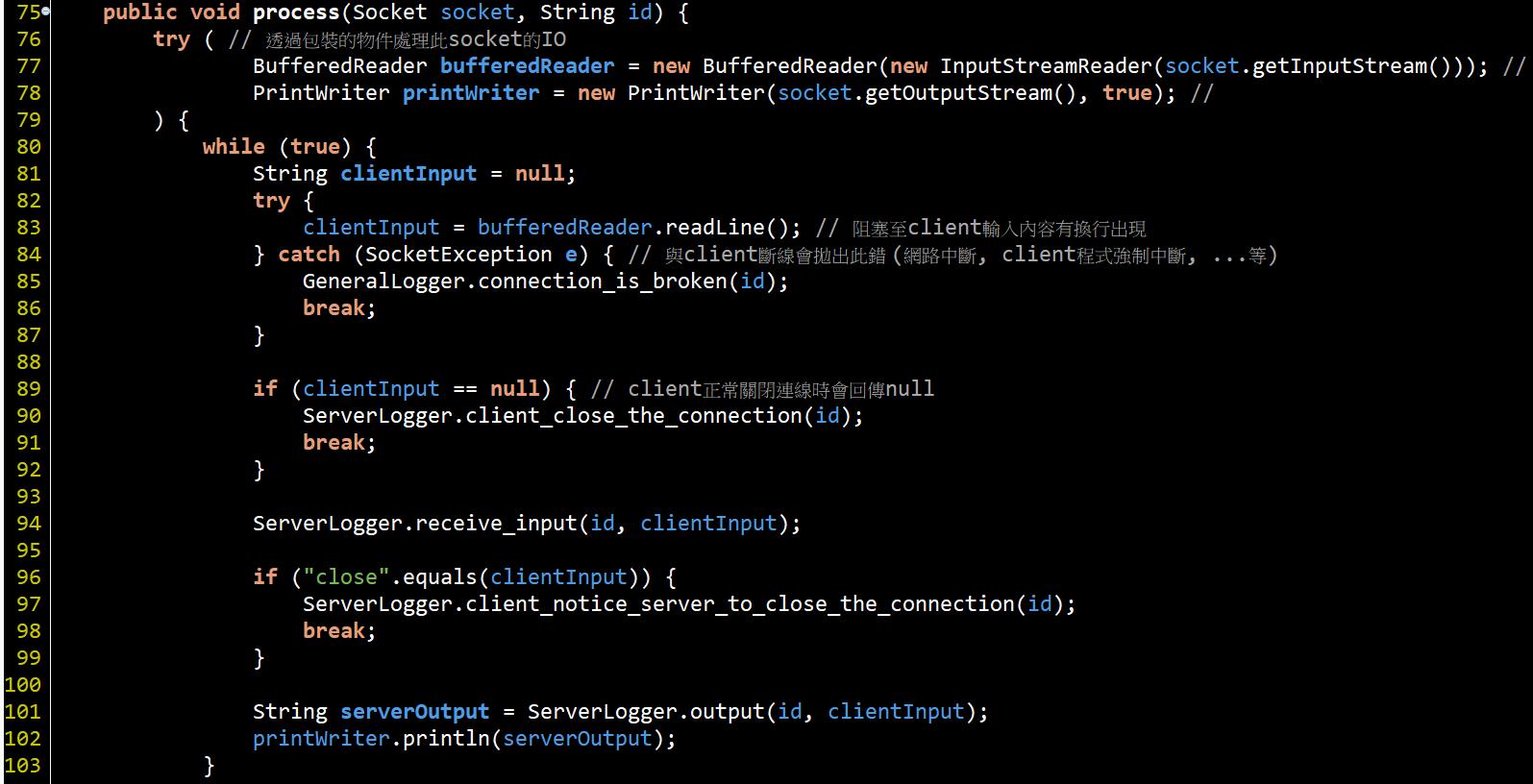
實戰BIO



非常簡單, 在#43行將會開始監聽port接受連線進來, 接著執行process()開始處理連線.



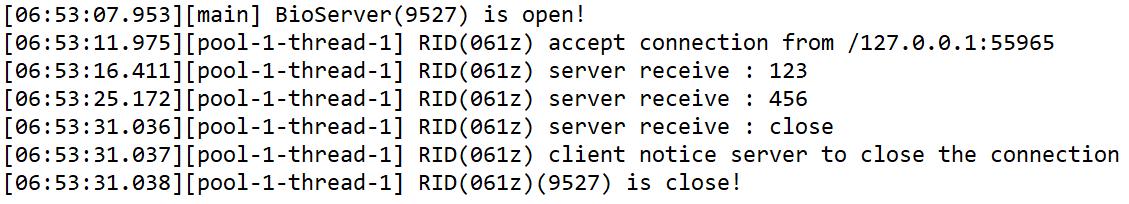
主Thread會在#51行阻塞(等待)有連線進來, 一但有連線進來才會被喚醒取得一個對應client的Socket, 然後就可以從Socket上取得一些client連線資訊以及最重要的InputStream、OutputStream物件, 取得輸入與輸出訊息。



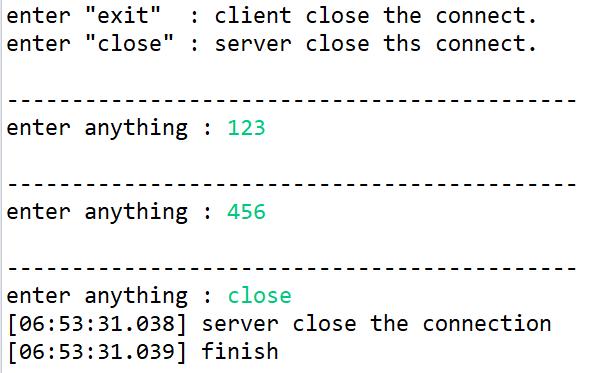
#76為使用1.7提供的auto close方法, 在try()內放入有實作Closeable的物件, 在最後離開try-catch時就會自動呼叫close().

#77、#78是為了IO操作方便進行I/O Stream包裹. 因此使用此server時, client的輸入要以換"行為(\n)"界, 這段程式才會從#83行離開往下執行, 然後#102將訊息output給client.

**執行畫面**



(server console)



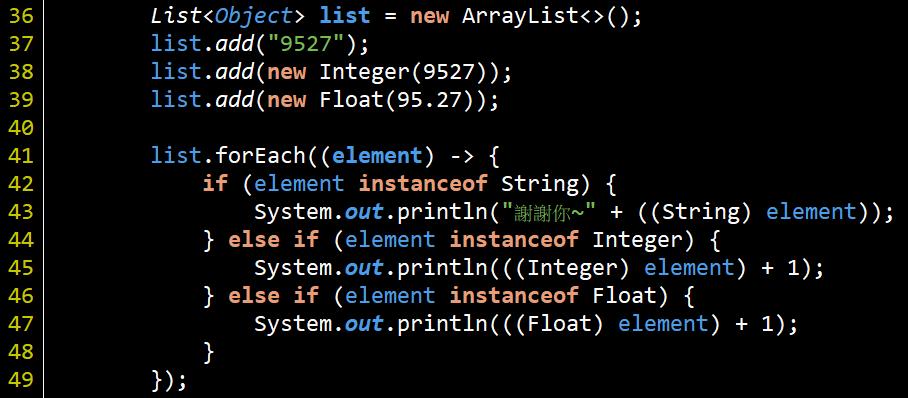
(client console)

BIO就是這樣, 是不是很簡單勒?

實戰NIO



#46打開的Selector就是"多路複用器"(網路上大家都這麼稱), 一開始看到"多路複用器"詞會想說到底是啥鬼東西, 因為這個用詞太難理解了, 但其實概念很簡單, 有點像是下面這段code:



這個list裡塞入了異質的物件, 在取用時就要先判斷型別才能正確操作. 而Selector就像這個list一樣, 會註冊不同的Channel, 在select出來之後就要先判斷註冊的Channel是啥再進行操作. 對應程式的操作就是 #51 serverChannel.register(this.selector, SelectionKey.OP\_ACCEPT); 這段, 第二個參數就是ops, 用來表達這個Channel會發生什麼事件, 在select出來之後就要根據此設定判斷做相對應的處理.



#59或#60就是在select事件, 若註冊在selector裡的Channel有事件發生(連線、I/O等)就會獲得一個Set<SelectionKey>物件. SelectionKey其實就是對應每一個你註冊至這個selector裡的Channel, 此SelectionKey就會包含該Channel與當初register時使用的ops, 用以判斷當初註冊進來的Channel型別以供處理.

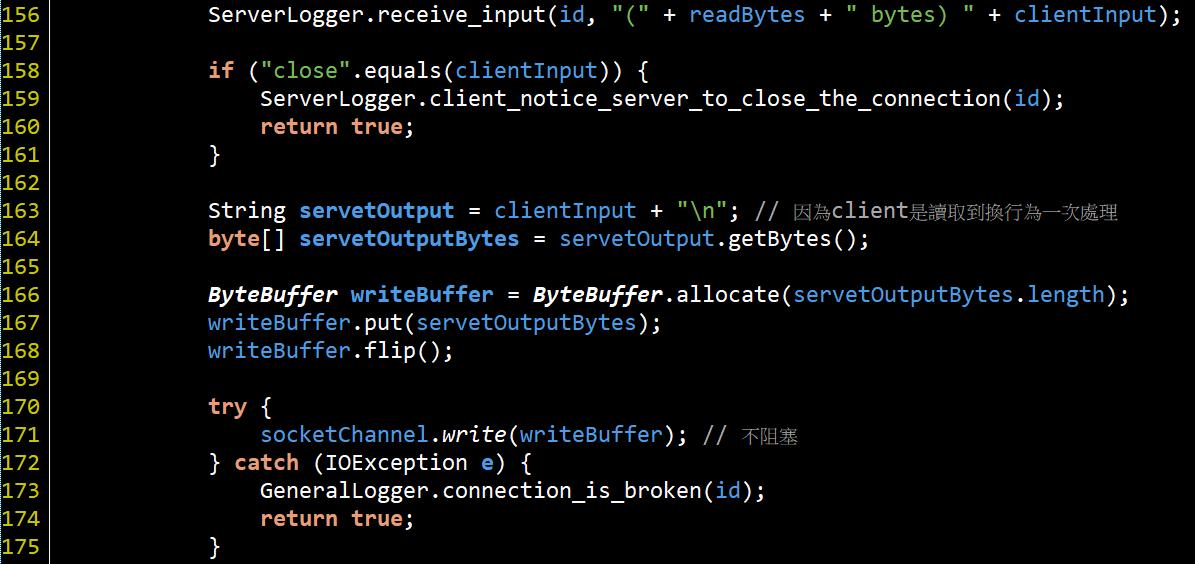


#91~#100就是最一開始server開啟監聽port的Channel所發生的事件, 因為當初在註冊的時候只使用 SelectionKey.OP\_ACCEPT , 因此這裡使用isAcceptable()來判斷. 接著#93會操作OS完成TCP三方交握並獲得與該client專屬的Channel, 接著將這個Channel註冊回selector, 讓selector可以監控這個client channel事件的發生.

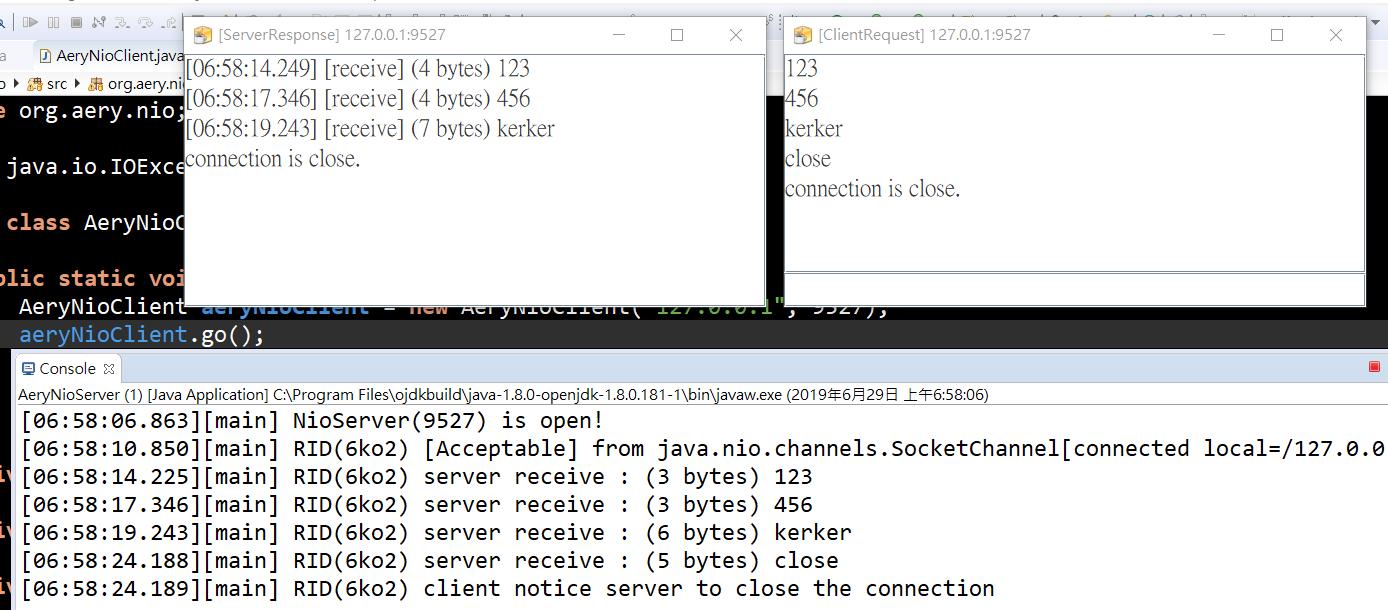
#102~#111就是處理剛剛註冊進來的client channel, 因為當初註冊的時候是使用, 所以這裡才用isReadable()來判斷, 所以這個事件被觸發基本就是只有Input進來, 要來處理輸入的東西了.



#131就是準備ByteBuffer在#135取得底層暫存的Input內容, 然後在#149~#154將該input內容轉為字串後在#156印出, 接著再將這個輸入如下圖程式碼回應給client

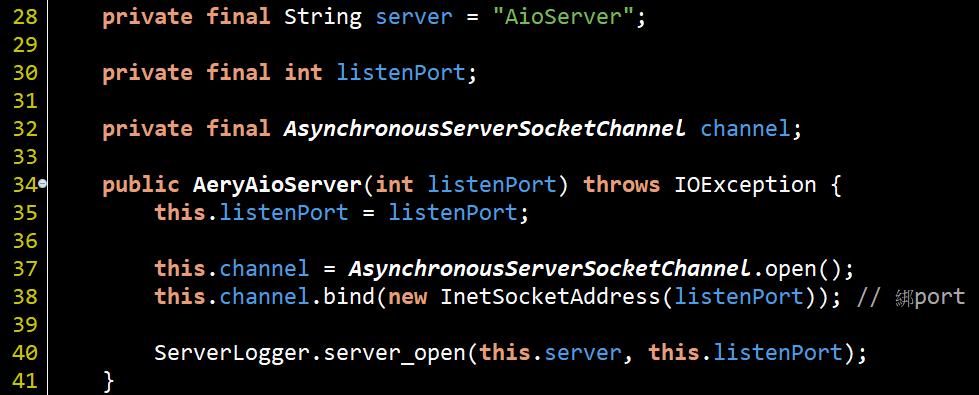


**執行畫面**



NIO操作大致如上說明, 但實際應用上會比這還要複雜, 因為有可能發生用來讀取input的ByteBuffer不夠大沒辦法一次讀完input內容, 所以要暫存到下一次讀取再串起處理,等之類的問題. 不過通常越高效的功能就伴隨著越複雜的程式操作, 但這設計真的很不好操作(再次強調)!

實戰AIO



這裡建立監聽port的方式是不是有種反璞歸真的fu哩, 但前文有提過AIO完全基於callback, 所以重點在下圖程式碼

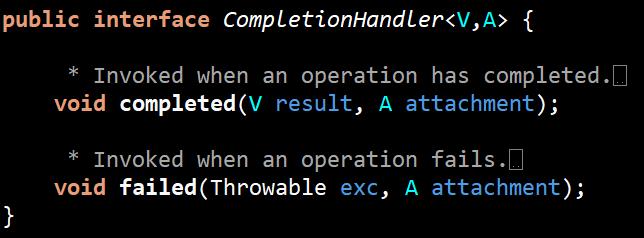


#46~#52很重要, 因為AIO不再像BIO、NIO那樣需要控制一條Thread來等待連線進來, 因此若這邊不讓main Thread等待, 就會因為沒有其他非Daemon-Thread存在, 程式就會直接結束, 所以這邊讓main Thread進行等待, 直到被操作關閉再喚醒結束.

而#62這行就是先前不斷提到的callbac操作, 這邊在跟Channel講說"兄弟, 我要監聽這個port的連線, 有連線發生通知第二個參數物件餒", 大致上就是這樣. 因此我們可以先來看看channel.appept()要接受什麼樣的參數:



原來第二參數必須為CompletionHandler的interface實作, 那麼再看看這個interface



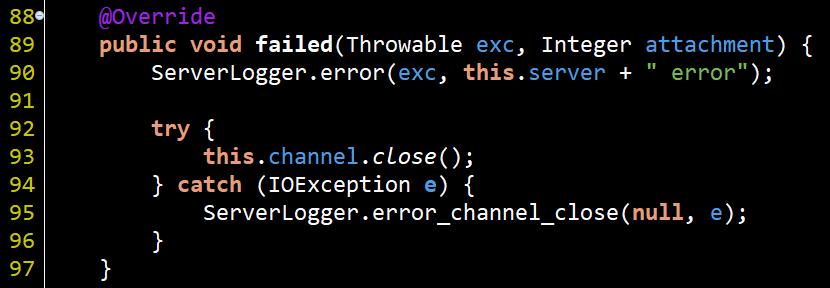
從CompletionHandler的method看來可以很快理解, 傳入accept的第二個參數, 在相對應的任務完成後會呼叫completed(), 通知說任務完成了, 你可以做你該做的事情了; 而當發生錯誤時相對的就會呼叫failed()通知你爆炸了!

快速了解channel.appept()意義後, 我們回頭來看#62行, 所以其實AeryAioServer的class長這樣



相對應的實作





completed實作裡會獲得一個AsynchronousSocketChannel, 其實就是對應這個連線client的專屬Channel, 而AeryAioServer在這邊將請求的處理交給了AioServerRequestReader去處理, 然後在#74通知開始處理請求.

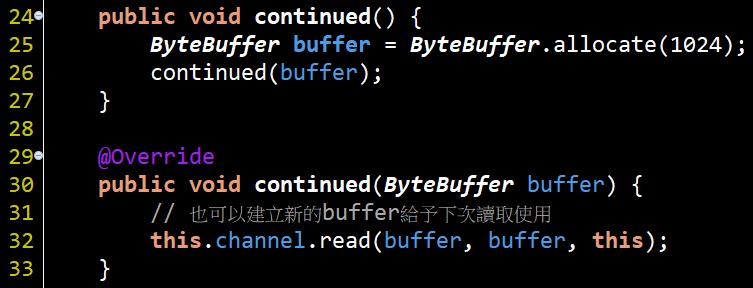
#75是重點, 若沒有再操作channel繼續接受連線事件, 然後...就沒有然後了..., 因為AIO全部基於callback, 所以沒有給他下一個事件的callback, 就會沒有然後了! 這也像是BIO、NIO裡處理請求的loop只做一次的意思, 只是AIO的這個loop是method層級(類似迭代概念).

接著來看看AioServerRequestReader在做啥吧

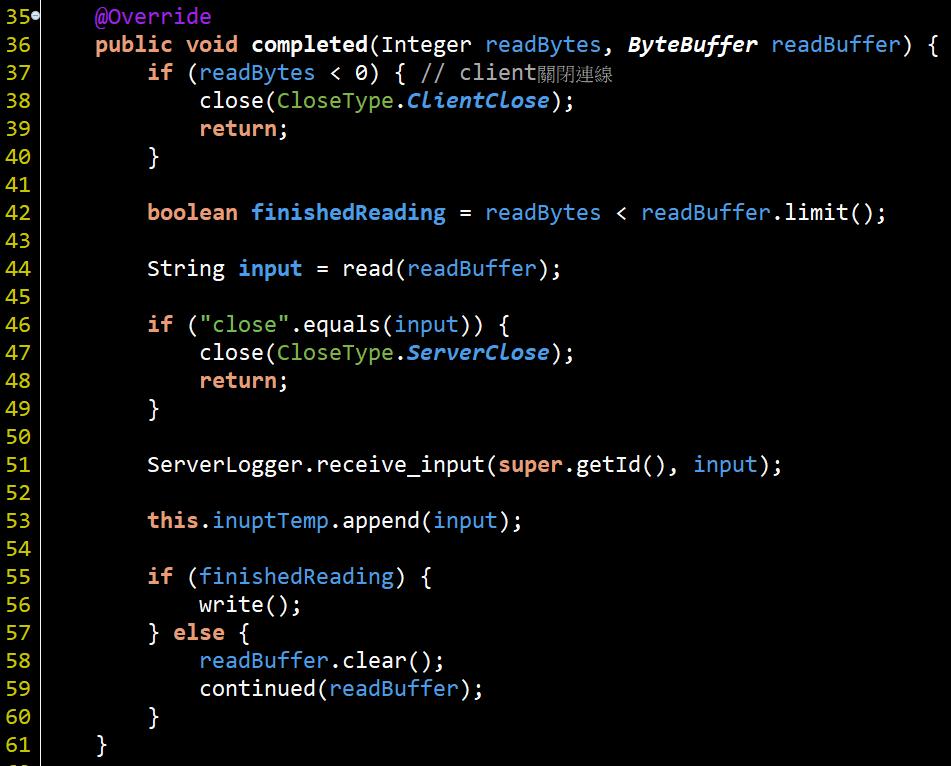




從繼承關係看出來AioServerRequestReader屬於CompletionHandler實作, 因為在channel的read或write操作時的callback也是這個interface的時做, 因此AioServerRequestReader.continued()裡長這樣



#30可以看到在跟channel說"兄弟, ByteBuffer我幫你準備好了, 就這麼大, 有讀到東西記得來叫我做事欸", 所以AeryAioServer裡的#74就是在做這件事. 於是下一個流程就是AioServerRequestReader的completed()被呼叫

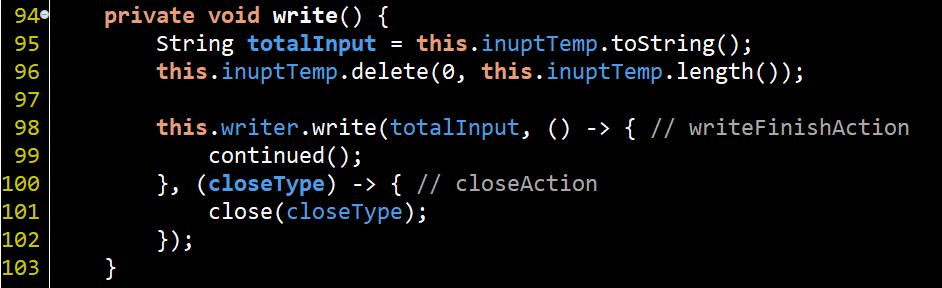


在這裡實作會看到拿到兩個參數, 第一個Integer表示此次讀了多少個byte, 因此可以用來與ByteBuffer的大小比對是否有塞滿, 就意味著會不會需要下一次的讀取.

因此在#55~#60就是判斷該次讀取事件完成後, 是不是要繼續讀取下一次input或準備要回應client(#56的write).

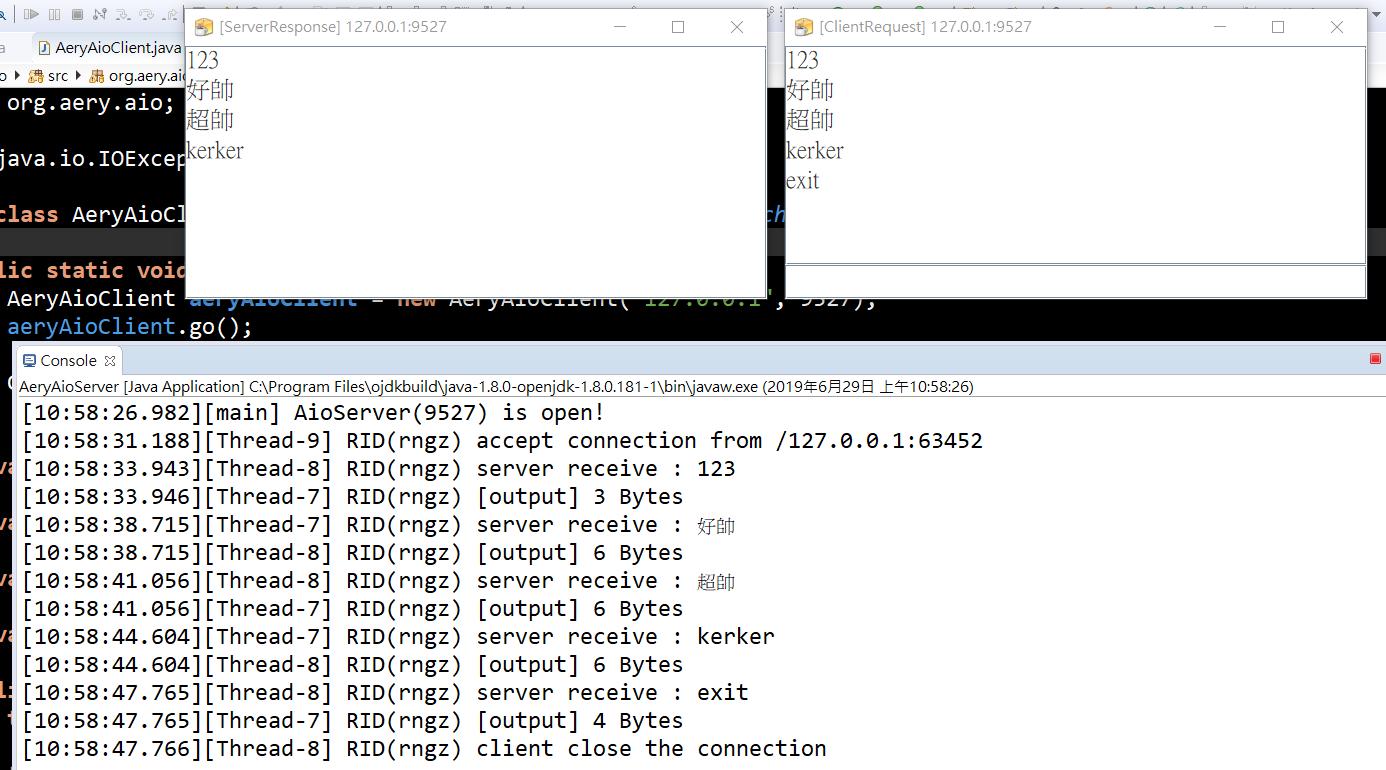
#44的read同NIO操作, 不多做解釋

再來就是#56的write()操作



#98的write是AioServerRequestWriter物件, 主要負責對channel輸出(回應client), 重點在於#99跟#101這兩個callback function, 當輸出完成後, 會呼叫writeFinishAction這裡(#99), 繼續讓AioServerRequestReader可以準備讀取這個channel下一次的input內容, 否則又要像只執行一次的迴圈一樣. 若中途client主動關閉連線或斷線的話, 將會呼叫closeAction(#101)來關閉這個channel.

**執行畫面**



可以看到server端處理的thread不一樣了, 原因是由別人完成事件之後呼叫callback執行.

總結

其實BIO透過一點程式技巧仍然可以達到NIO, AIO的效果, 但需要大量加工, 且AIO有透過OS層面操作, 在效能上仍有差異, 但此文主要介紹與實作三種IO模型遇到的問題與要解決的問題, 因此這裡並不討論OS層面實作細節.

在三種IO模型裡, 很有趣的是包含了幾種設計思維, 可以看到程式一路走來思維的演進. 除了了解越多底層概念, 越能在問題發生時有更多面向的追查, 也可以吸收前人們的經驗、學習大神們解決問題的思路, 這也是為什麼我喜歡研究底層或其他open source的source code的原因啦.