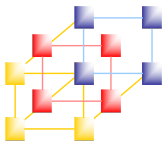
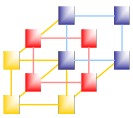
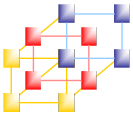


Unit 8
Non-blocking Socket



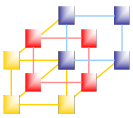
大綱

- Java NIO
- Non-blocking Socket



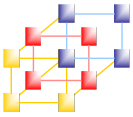
Java NIO

- Java.nio package 提供一個 high-speed, block-oriented 的 I/O 函式庫 (包括 non-blocking I/O)
- 傳統的 Java 程式設計，I/O 使用 streaming 的方式來處理
  - All I/O is viewed as the movement of single bytes, one at a time, through an object called a Stream.
- Java NIO 的 I/O 使用區塊 (block) 的方式來處理
  - NIO moves the most time-consuming I/O activities (namely, filling and draining buffers) back into the operating system, thus allowing for a great increase in speed.



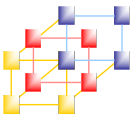
Channel and Buffer

- Channel 與 Buffer 為 NIO 的主要物件
  - Channel 就如同原 I/O package 中的 streams，所有要傳送的資料都需利用 Channel object 來傳送
  - Buffer 為一個容器 (Container Object)，所有要送到 Channel 中的資料都需先放在 Buffer



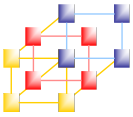
Channel

- SocketChannel
  - 適用於 TCP 協定的資料讀取與傳送
  - 運用於 Client 端
- SocketServerChannel
  - 用於 TCP 協定的伺服器，傾聽本機的埠號，等待外來的連線
  - 運用於 Server 端
- DatagramChannel
  - 用於UDP協定的封包傳送



Buffer 的型別

- ByteBuffer – 位元組緩衝區
- CharBuffer – 字元緩衝區
- ShortBuffer – 短整數short緩衝區
- IntBuffer – 整數int緩衝區
- LongBuffer – 長整數long緩衝區
- FloatBuffer – 浮點數float緩衝區
- DoubleBuffer – 雙精確度浮點數double緩衝區



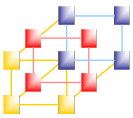
Buffer 的分類

- 直接緩衝區
  - 可跳過JVM的處理，直接對應到作業系統的記憶體來存取緩衝區資料
  - 適合較大檔案

```
ByteBuffer buf = ByteBuffer.allocateDirect(20);
```

- 間接緩衝區
  - 由JVM負責所有的容量分配與存取動作
  - 使用上會耗費JVM本身所分配的記憶體
  - 實務設計常採用間接緩衝區來處理

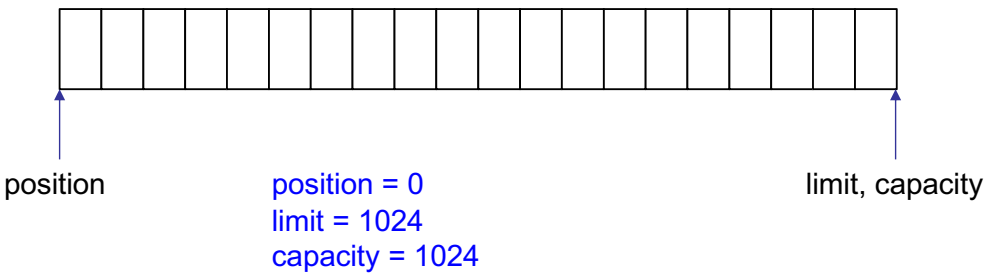
```
ByteBuffer buf = ByteBuffer.allocate(20);
```

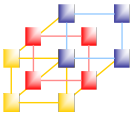



ByteBuffer 指標 (1/8)

```
ByteBuffer buffer = ByteBuffer.allocate(1024);
```

- capacity – 是這個緩衝區的總量，這個數字在後續的各項操作都不會改變
- position – 是指資料可以從這裡開始"處理"
- limit – 是指資料可以從 position "處理"到 limit 所指的位置



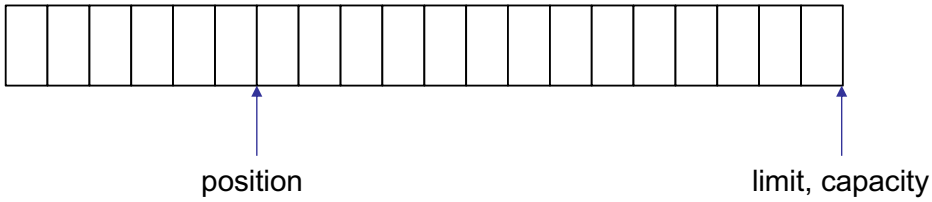


ByteBuffer 指標 (2/8)

- 讀取一段資料到緩衝區

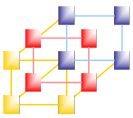
```
socketChannel.read(buffer); //從 socket 寫入 buffer
```

  - 假設寫入的資料有 124 個 bytes，指標的變化



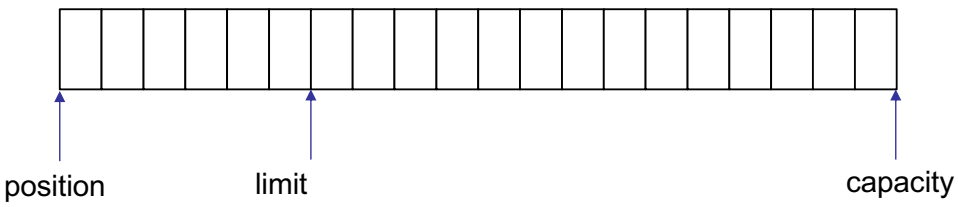
position = 124
limit = 1024
capacity = 1024

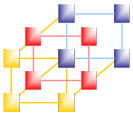
  - position 指向下一個可將資料寫入緩衝區的位置，limit 仍是指可被寫入資料的限制(不可超過這個位置)



ByteBuffer 指標 (3/8)

- 資料處理前先呼叫 flip()
  - limit 會被設定為原本 position 的值，即 124
  - position 被設定為0 (之後的程式可透過 get() 來讀取 position 到 limit 這段區間的資料
    - position 的角色在 flip 前後產生了變化，在呼叫 flip() 之前是指向可被寫入的位置，呼叫 flip() 後變成了正要被讀取的位置

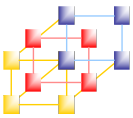




ByteBuffer 指標 (4/8)

```
byte b = buffer.get();
```

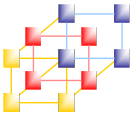
- 每呼叫一次 get()，即會傳回 position 所指位置的資料，並將 position 加1
- position 最多可以加到 limit，如果 position 已經等於 limit，再呼叫 get()，會拋出 BufferUnderflowException 例外
- method - get(i)，可以指定要讀取特定位置的資料，這個 method 呼叫後除了傳回值外，不會移動 position



ByteBuffer 指標 (5/8)

- ByteBuffer wrap(byte[] array)
  - 將 byte[] 陣列中的資料，放入 ByteBuffer 裡
  - 以下的程式片段，會產生一個長為 100 bytes 的 ByteBuffer，並放入 b 的資料，各指標變化如下: position = 0、limit = 100、capacity = 100

```
byte[] b = new byte[100];
// put something to b
ByteBuffer buffer = ByteBuffer.wrap(b);
```

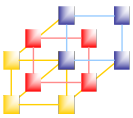


ByteBuffer 指標 (6/8)

- 從 ByteBuffer 讀出、寫入 socket

```
socketChannel.write(buffer);
if (buffer.hasRemaining()) {
     buffer.compact();
} else {
     buffer.clear();
}
```

  - 假設第一行 write 只寫了 90 個bytes，剩下 10 bytes 在 Buffer，程式就可能要再 write 一次
  - 判斷有沒有剩下的方法是用 hasRemaining()
    - 這一個 method 是判斷 position 和 limit 間是否還有資料
    - 假設，只從 buffer 讀出 90 bytes，指標移動如下:
      position = 90、limit = 100、capacity = 100

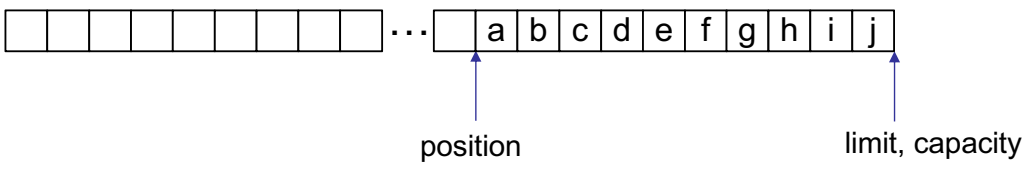


ByteBuffer 指標 (7/8)

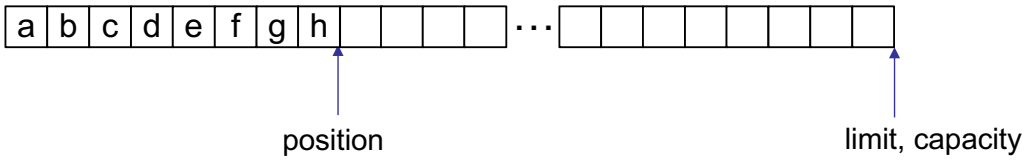
- 如果 Buffer 還有 10 bytes 未處理，假設我們希望再放入一些資料到 ByteBuffer 裡

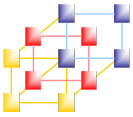
```
buffer.compact();
```

  - 執行 compact() 後未寫出的 10 bytes 資料會被移到最前面 index 0 ~ 9 的位置，指標變化如下:
    position = 10、limit = 100、capacity = 100




```
buffer.compact();
```





ByteBuffer 指標 (8/8)

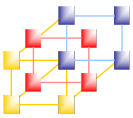
- ByteBuffer put(byte b)
  - 一次只放一個 byte，也可以放一個 byte array

```
byte b = new byte(0x55);
buffer.put(b);
```

  - 放入一個 byte，position 會加 1，其餘不變

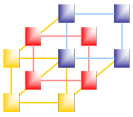
```
byte[] c = new byte[] { 'A', 'B', 'C', 'D', 'E',
'F', 'G' };
buffer.put(c);
```

  - 放入了 7 個 bytes， position 會加 7



Non-Blocking Socket 簡介

- 在 UNIX 系統下，使用者建立一個 BSD socket 之後，該 socket 的預設狀態是「阻擾」（Blocking）模式
- 使用者可以利用 ioctl() 函式來變更為「非阻擾」（Non-Blocking）模式

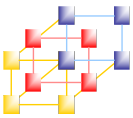


Blocking 模式

- 使用者在呼叫了某一個 blocking 函式之後，程式必須等待該函式呼叫完成（或失敗），並且回返（return）之後才能再繼續執行下一個指令
- Example

```
n = recv( socket_num, data, 50 );
memcpy( buf, data, n );
```

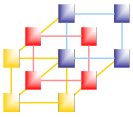
  - 在 Blocking 模式下，如果對方沒有傳送資料過來的話，那麼這個程式將會一直被阻攔在 recv()，而不會執行到 memcpy()
  - 等對方資料過來後，才會自 recv() 呼叫回返，然後執行 memcpy() 函式



Non-Blocking 模式 (1/2)

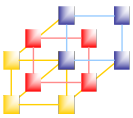
- 使用者呼叫了一個 non-blocking 函式之後，不管要求的條件或狀況是否成立或完成，都會馬上自該函式呼叫回返，接著執行程式的下一個指令
- Example

```
n = recv( socket_num, data, 50 );
if ( n > 0 )     /* 收到資料 */
   memcpy( buf, data, n );
else     /*沒有收到資料 */
   ...
```

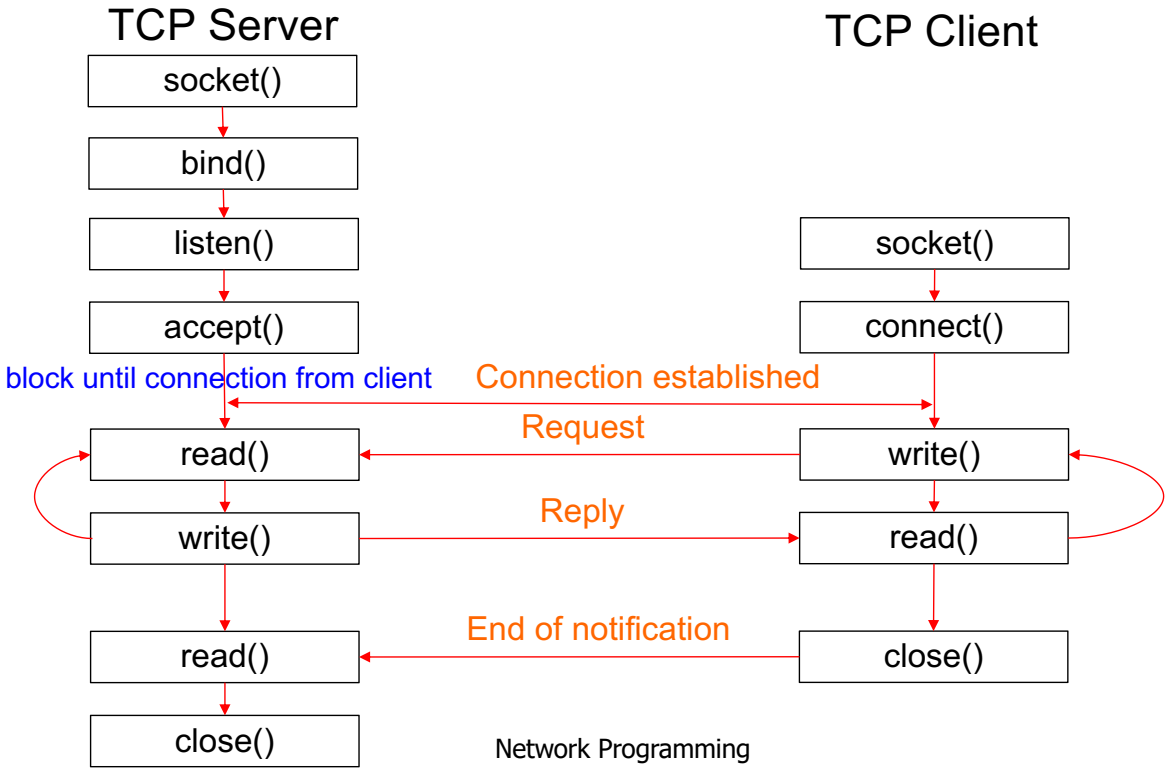


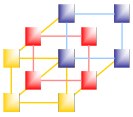
Non-Blocking 模式 (2/2)

- 在呼叫 recv() 函式時，不管對方是否已經送資料來了沒有，程式都不會被阻攔在 recv() 上
  - 如果此時的確有資料了，那麼就會被放在 data 這個暫存區（buffer）
  - 如果沒有資料，那麼就會回返錯誤的訊息
- 由於 non-blocking 模式在呼叫 recv() 時，並不一定有資料，且不知道什麼時候對方會送資料來，所以在使用 non-Blocking 模式時，程式必須常常再次呼叫 recv()，以檢查資料是否來了 (polling)



TCP 網路應用程式模型





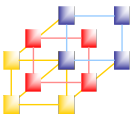
Non-blocking Server (1/2)

- 建立一個 ServerSocketChannel 並將此 channel 設成 non-blocking 模式

```
ServerSocketChannel ssc = ServerSocketChannel.open();
ssc.configureBlocking(false);
```

- 將所建立的 scoket 綁定於 port

```
ServerSocket serverSocket = ssc.socket();
serverSocket.bind(new InetSocketAddress(port));
```

Non-blocking Server (2/2)

- 接受連線

```
SocketChannel sc = ssc.accept();
```

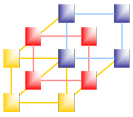
  - 這一個動作將以 non-blocking 方式進行
    - 如果有進來的連線要求，ssc.accept() 將回傳 SocketChannel 的值
    - 如果沒有連線的要求，ssc.accept() 將回傳 null

- 接收訊息

```
ByteBuffer b = ByteBuffer.allocate(100);
int    len = sc.read(b); // read message from sc
```

  - sc.read(b) 如要以 non-blocking 方式進行，在 ssc.accept() 建立連線後，須執行

```
sc.configureBlocking(false);
```

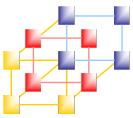


Non-blocking Client

- 建立一個 SocketChannel 並連到 Server

```
SocketChannel sc = SocketChannel.open();
sc.configureBlocking(false);
sc.connect(new InetSocketAddress(args[0], port));
while( !sc.finishConnect() );
```

  - 如果 SocketChannel 還沒連線成功，sc.finishConnect() 將回傳 false



Selector (1/2)

- Selector 提供一個機制來註冊我們有興趣的 I/O events，並提供方法讓我們了解那些已註冊的 events 已發生
- 建立一個 selector

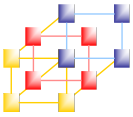
```
Selector selector = Selector.open();
```

- 註冊有興趣的 I/O events

```
SelectionKey key = ssc.register(selector, EVENT-TYPE);
```

- 等待 event 發生(blocking mode)

```
int num = selector.select();
```



Selector (2/2)

- Opens a selector

```
public static Selector open() throws IOException
```

- Selects a set of keys whose corresponding channels are ready for I/O operations

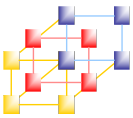
```
public abstract int select(long timeout) throws
IOException
public abstract int select() throws IOException
```

- Returns this selector's selected-key set

```
public abstract Set<SelectionKey> selectedKeys()
```

- Closes this selector

```
public abstract void close() throws IOException
```

SelectionKey

- 用來記錄一個 selector 已註冊的資料
  - OP_ACCEPT – for socket-accept operations
  - OP_CONNECT – for socket-connect operations
  - OP_READ – for read operations
  - OP_WRITE – for write operations
- Retrieves this key's ready-operation set

```
public abstract int readyOps()
public final boolean isReadable()
public final boolean isWritable()
public final boolean isConnectable()
public final boolean isAcceptable()
```