我們在寫程式時, 有不少時間都是在看別人的 Code.
例如看 Team 的 Code, 看小組整合的 Code, 若一開始沒規劃怎麼看,
就會"嚕看嚕苦 (台語)"

不管是參考也好, 從 Open Source 抓下來研究也好, 為了瞭解箇中含意, 在有限的時間下, 不免會對龐大的 Source Code 解讀感到壓力.
網路上有一篇關於分析看 Code 的方法, 做為程式設計師的您, 不妨參考看看,
換個角度來分析. 也能更有效率的解讀你想要的程式碼片段.

六個章節：

(1) 讀懂程式碼, 使心法皆為我所用.
(2) 摸清架構, 便可輕鬆掌握全貌.
(3) 優質工具在手, 讀懂程式非難事.
(4) 望文生義, 進而推敲組件的作用.
(5) 找到程式入口, 再由上而下抽絲剝繭.
(6) 閱讀的樂趣, 透過程式碼認識作者.

程式碼是別人寫的，只有原作者才真的了解程式碼的用途及涵義。許多程式人心裡都有一種不自覺的恐懼感，深怕被迫去碰觸其他人所寫的程式碼。但是，與其抗拒接收別人的程式碼，不如徹底了解相關的語言和慣例，當成是培養自我實力的基石。

閱讀他人的程式碼(1)─讀懂程式碼，使心法皆為我所用

程式碼是別人寫的，只有原作者才真的了解程式碼的用途及涵義。許多程式人心裡都有一種不自覺的恐懼感，深怕被迫去碰觸其他人所寫的程式碼。但是，與其抗拒接收別人的程式碼，不如徹底了解相關的語言和慣例，當成是培養自我實力的基石。

對大多數的程式人來說，撰寫程式碼或許是令人開心的一件事情，但我相信，有更多人視閱讀他人所寫成的程式碼為畏途。許多人寧可自己重新寫過一遍程式碼，也不願意接收別人的程式碼，進而修正錯誤、維護它們、甚至加強功能。

這其中的關鍵究竟在何處呢？若是一語道破，其實也很簡單，程式碼是別人寫的，只有原作者才真的了解程式碼的用途及涵義。許多程式人心裡都有一種不自覺的恐懼感，深怕被迫去碰觸其他人所寫的程式碼。這是來自於人類內心深處對於陌生事物的原始恐懼。

讀懂別人寫的程式碼，讓你收穫滿滿

不過，基於許多現實的原因，程式人時常受迫要去接收別人的程式碼。例如，同事離職了，必須接手他遺留下來的工作；也有可能你是剛進部門的菜鳥，而同事經驗值夠了、升級了，風水輪流轉，一代菜鳥換菜鳥。甚至，你的公司所承接的專案，必須接手或是整合客戶前一個廠商所遺留下來的系統，你們手上只有那套系統的原始碼（運氣好時，還有數量不等的文件）。

諸如此類的故事，其實時常在程式人身邊或身上持續上演著。許多程式人都將接手他人的程式碼，當做一件悲慘的事情。每個人都不想接手別人所撰寫的程式碼，因為不想花時間去探索，寧可將生產力花在產生新的程式碼，而不是耗費在了解這些程式碼上。

很遺憾的是，上述的情況對程式人來說很難避免。我們總是必須碰觸到其他人所寫成的程式碼，甚至必須了解它、加以修改。對於這項需求，在現今開放原始碼的風氣如此盛行的今日，正如之前的「程式設計2.0」文中所提到的，你可以透過開放原始碼學習到新的技術、學習到高手的架構設計，大幅提高學習的效率及效果。你甚至可以直接自開放原始碼專案中抽取、提煉出自己所需的程式碼，站在巨人的肩膀上，直接由彼端獲得所需的生產

力。從這個觀點來看，讀懂別人所寫的程式碼，就不再只是從負面觀點的「被迫接收」，而是極具正面價值的「汲取養份」。

先了解系統架構與行為模式，再細讀

倘若撰寫程式碼是程式人的重要技藝之一，那麼讀懂別人的程式碼、接著加以修改，也勢必是另一個重要的技藝。

如果你不能熟悉這項工作，不僅在遭逢你所不願面對的局面時，無法解決眼前接手他人程式碼的難題，更重要的是，當你看著眼前現成的程式碼，卻不知如何從中擷取自己所需，導致最後只能入寶山空手回，望之興嘆。

接觸他人的程式碼，大致上可以分為三種程度：一、了解，二、修改、擴充，三、抽取、提煉。

了解別人的程式碼是最基礎的工作，倘若不能了解自己要處理的程式碼，就甭論修改或擴充，更不可能去蕪存菁，從中萃取出自己所需，回收再利用別人所撰寫的程式碼。

雖說是「閱讀」，但程式碼並不像文章或小說一樣，透過這種做法，便能夠獲得一定程度的了解。閱讀文章或小說時，幾乎都是循序地閱讀，你只消翻開第一頁，一行行閱讀下去即可。但是，有許多程式人在試著閱讀其他人的程式碼時，卻往往有不知如何讀起的困難。

或許找到系統的第一頁（也就是程式碼執行的啟始點）並不難，但是複雜度高的系統，有時十分龐大，有時千頭萬緒。

從程式碼的啟始點開始讀起，一來要循序讀完所有的程式碼曠日費時，二來透過這種方式來了解系統，很難在腦中構建出系統的面貌，進而了解到系統真正的行為。所以，閱讀程式碼的重點，不在於讀完每一行程式碼，而是在於有效率地透過探索及閱讀，從而了解系統的架構及行為模式。以便在你需要了解任何片段的細節實作時，能夠很快在腦上對映到具體的程式碼位置，直到那一刻，才是細讀的時機。

熟悉溝通語言與慣例用語

不論如何，有些基本的準備，是閱讀他人程式碼時必須要有的。

首先，你最好得了解程式碼寫成的程式語言。想要讀懂法文寫成的小說，總不能連法文都不懂吧。有些情況則很特殊。我們雖然不懂該程式碼撰寫所用的語言，但是因為現代語言的高階化，而且流行的程式語言多半都是血統相近，所以即使不那麼熟悉，有時也可勉力為之。

除了認識所用語言之外，再來就是要先確認程式碼所用的命名慣例（naming convention）。了解命名慣例很重要，不同的程式人或開發團隊，差異可能很大。

這命名慣例涵蓋的範圍通常包括了變數的名稱、函式的名稱、類別（如果是物件導向的話）的名稱、原始碼檔案、甚至是專案建構目錄的名稱。倘若使用了像設計模式之類的方法，這些名稱更有一些具體的表述方式。

命名慣例有點像是程式人在程式語言之上，另行建構的一組溝通行話。程式人會透過共通約束、遵守的命名慣例，來表達一些較高階的概念。例如，有名的匈牙利式命名法，便將

變數名稱以屬性、型別、說明合併在一起描述。對程式人來說，這種方式能夠提供更豐富的資訊，以了解該變數的作用及性質。

對程式碼閱讀來說，熟悉這個做法之所以重要，是因為當你了解整個系統所採用的慣例時，你便能試著以他們所共同操用的語彙來進行理解。倘若，不能了解其所用的慣例，那麼這些額外提供的資訊，就無法為你所用。像以設計模式寫成的程式碼，同樣處處充滿著模式的名稱，諸如：Factory、Facade、Proxy等等。以這些名稱指涉的類別，也直接透過名稱，表達了它們自身的作用。對於懂得這命名慣例的讀者來說，不需要深入探索，也能很快捕捉到這些類別的意義。

當你拿到一套必須閱讀的程式碼時，最好先取得命名慣例的說明文件。然而，並不是每套程式碼都附有此類的說明文件。另一個方式，就是自己到程式碼中，大略瀏覽一遍，有經驗的程式人可以輕易發掘出該系統所用的命名慣例。

常見的命名方式不脫那幾類，這時候經驗就很重要，倘若你知道的慣例越多，就越能輕易識別他人所用的慣例。如果運氣很糟，程式碼所用的慣例是前所未見的，那麼你也得花點時間歸納，憑自己的力量找出這程式碼命名上的規則。

掌握程式碼撰寫者的心態與習慣

大多數的程式碼，基本上都依循一致的命名慣例。不過運氣更差的時候，一套系統中可能會充斥著多套命名慣例。這有可能是因為開發團隊由多組人馬所構成，每組人馬都有不同的文化，而在專案開發管理又沒有管控得宜所造成。最糟的情況，程式碼完全沒有明顯的慣例可言，這時候閱讀的難度就更高了。

想要閱讀程式碼，得先試著體會程式碼作者的「心」。想要這麼做，就得多了解對方所使用的語言，以及慣常運用的語彙。在下一回中，我們將繼續探討閱讀程式碼的相關議題。

閱讀他人的程式碼(2) 摸清架構，便可輕鬆掌握全貌

在本文中，我們的重點放在：要了解一個系統，最好是採取由上至下的方式。先試著捕捉系統架構性的觀念，不要過早鑽進細節，因為那通常對於你了解全貌，沒有多大的幫助。閱讀程式碼不需要從第一行讀起，我們的目的並不是在於讀遍每一段程式碼。

基於許多原因，程式人需要閱讀其他人所寫成的程式碼。而對程式設計2.0時代的程式人來說，最正面的價值在於，能讀懂別人程式的人，才有能力從中萃取自己所需的程式，藉以提高生產力。

閱讀程式碼的目的，在於了解全貌而非細節

想要讀懂別人程式碼的根本基礎，便是了解對方所用的程式語言及命名慣例。有了這個基礎之後，才算是具備了基本的閱讀能力。正如我之前提到的──想要讀懂法文寫成的小說，總不能連法文都不懂吧。閱讀程式碼和閱讀文學作品，都需要了解撰寫所用的語言及作者習用的語彙。

但我們在閱讀文學作品通常是採循序的方式，也就是從第一頁開始，一行一行地讀下去，依循作者為你鋪陳的步調，逐漸進到他為你準備好的世界裡。

閱讀程式碼卻大大不同。我們很少從第一行開始讀起，因為除非它是很簡單的單執行緒程式，否則很少這麼做。因為要是這麼做，就很難了解整個系統的全貌。

是的，我們這邊提到了一個重點，閱讀程式碼的目的在於了解系統的全貌，而不是在於只是為了地毯式的讀遍每一段程式碼。

就拿物件導向程式語言所寫成的系統來說，整個系統被拆解、分析成為一個個獨立的類別。閱讀個別類別的程式碼，或許可以明白每項類別物件個別的行為。但對於各類別物件之間如何交互影響、如何協同工作，又很容易陷入盲人摸象的困境。這是因為各類別的程式碼，只描述個別物件的行為，而片段的閱讀就只能造就片面的認識。

由上而下釐清架構後，便可輕易理解組成關係

如果你想要跳脫困境，不想浪費大量時間閱讀程式碼，卻始終只能捕捉到對系統片段認識，就必須轉換到另一種觀點來看待系統。從個別的類別行為著手，是由下至上（Bottom-Up）的方法；在閱讀程式碼時，卻應該先採由上至下（Top-Down）的方式。對程式碼的閱讀來說，由上至下意謂著，你得先了解整個系統架構。

系統的架構是整個系統的骨幹、支柱。它表現出系統最突出的特徵。知道系統架構究竟屬於那一種類型，通常大大有益於了解系統的個別組成之間的靜態及動態關係。

有些系統因為所用的技術或框架的關係，決定了最上層的架構。例如，採用Java Servlet/JSP技術的應用系統，最外層的架構便是以J2EE（或起碼J2EE中的Web Container）為根本。

使用Java Servlet/JSP技術時，決定了某些組成之間的關係。例如，Web Container依據web.xml的內容載入所有的Servlets、Listeners、以及Filters。每當Context發生事件（例如初始化）時，它便會通知Listener類別。每當它收到來自客戶端的請求時，便會依循設定的所有Filter Chain，讓每個Filter都有機會檢查並處理此一請求，最後再將請求導至用來處理該請求的Servlet。

當我們明白某個系統採用這樣的架構時，便可以很容易地知道各個組成之間的關係。即使我們還不知道究竟有多少Servlets，但我們會知道，每當收到一個請求時，總是會有個相對應的Servlet來處理它。當想要關注某個請求如何處理時，我應該去找出這個請求對應的Servlet。

了解架構，必須要加上層次感

同樣的，以Java寫成的Web應用程式中，也許會應用諸如Struts之類的MVC框架，以及像Hibernate這樣的資料存取框架。它們都可以視為最主要的架構下的較次級架構。而各個應用系統，甚至有可能在Struts及Hibernate之下，建立自有的更次級的架構。

也就是說，當我們談到「架構」這樣的觀念時，必須要有層次感。而不論是那一層級的架構，都會定義出各自的角色，以及角色間的關係。對閱讀者來說，相較於直接切入最細微的單一角色行為，不如了解某個特定的架構中，究竟存在多少角色，以及這些角色之間的互動模式，比較能夠幫助我們了解整個系統的運作方式。

這是一個很重要的關鍵，當你試著進到最細節處之前，應該先試著找出參與的角色，及他們之間的關係。例如，對事件驅動式的架構而言，有3個很重要的角色。一個是事件處理的分派器（Event Dispatcher）、一個是事件產生者（Event Generator）、另一個則是事件處理器（Event Handler）。

事件產生器產生事件，並送至事件分派器，而事件分派器負責找出各事件相對應的事件處理器，並且轉交該事件，並命令事件處理器加以處理。像Windows的GUI應用程式，便是採用事件驅動式的架構。

當你知道此類的應用程式皆為事件驅動式的架構時，你便可以進一步得知，在這樣的架構下會有3種主要的角色。雖然也許還不清楚整個系統中，究竟會需要處理多少事件的類型，但對你而言，已經建立了對系統全貌最概觀的認識。

雖然你還不清楚所有的細節，但諸如確切會有那些事件類型之類的資訊，在此刻還不重要——不要忘了，我們採取的是由上而下的方式，要先摸清楚主建築結構，至於壁紙的花色怎麼處理，那是到了尾聲時才會做的事。

探索架構的第一件事：找出系統如何初始化

有經驗的程式人，對於時常被運用的架構都很熟悉。常常只需要瞧上幾眼，就能明白一個系統所用的架構，自然就能夠直接聯想到其中會存在的角色，以及角色間的關係。

然而，並不是每個系統所用的架構，都是大眾所熟悉，或是一眼能夠望穿的。這時候，你需要探索。目標同樣要放在界定其中的角色、以及角色間的靜態、動態關係。

不論某個系統所採用的架構是否為大部分人所熟知的，在試著探索一個系統的長相時，我們應該找出來幾個答案，了解在它所用的架構下，下列這件事是如何被完成的：一、系統如何初始化，二、與這個系統相接的其他系統（或使用者）有那些，而相接的介面又是什麼；三、系統如何反應各種事件，四、系統如何處理各種異常及錯誤。

系統如何初始化是很重要的一件事，因為初始化是為了接下來的所有事物而做的準備。從初始化的方式、內容，能知道系統做了什麼準備，對於系統會有什麼行為展現，也就能得窺一二了。

之所以要了解與系統相接的其他系統（或使用者），為的是要界定出系統的邊界。其他的系統可能會提供輸入給我們所探索的系統，也可能接收來自這系統的輸出，了解這邊界所在，才能確定系統的外觀。

而系統所反應的事件類型、以及如何反應，基本上就代表著系統本身的主要行為模式。最後，我們必須了解系統處理異常及錯誤的方式，這同樣也是系統的重要行為，但容易被忽略。

之前，我們提到必須先具備一個系統的語言基礎，才能夠進一步加以閱讀，而在本文中，我們的重點放在：要了解一個系統，最好是採取由上至下的方式。先試著捕捉系統架構性的觀念，不要過早鑽進細節，因為那通常對於你了解全貌，沒有多大的幫助。

閱讀他人的程式碼 (3) 優質工具在手，讀懂程式非難事

系統的複雜度往往超過人腦的負荷。閱讀程式碼的時候，你會需要更多工具提供協助。使用好的整合式開發環境（IDE）或文字編輯器，就能提供最基本的幫助。

先建立系統的架構性認識，然後透過名稱及命名慣例，就可以推測出各組件的作用。例如：當Winamp嘗試著初始化一個Plug-In時，它會呼叫這個結構中的init函式，以便讓每個Plug-In程式有機會初始化自己。當Winamp打算結束自己或結束某個Plug-In的執行時，便會呼叫quit函式。

在閱讀程式碼的細節之前，我們應先試著捕捉系統的運作情境。在採取由上至下的方式時，系統性的架構是最頂端的層次，而系統的運作情境，則是在它之下的另一個層次。

好的說明文件難求，拼湊故事的能力很重要

有些系統提供良善的說明文件，也許還利用UML充分描述系統的運作情境。那麼對於閱讀者來說，從系統的分析及設計文件著手，便是快速了解系統運作情境的一個途徑。

但是，並不是每個軟體專案都伴隨著良好的系統文件，而許多極具價值的開放原始碼專案，也時常不具備此類的文件。對此，閱讀者必須嘗試自行捕捉，並適度地記錄捕捉到的運作情境。

我喜歡將系統的運作情境，比擬成系統會上演的故事情節。在閱讀細節性質的程式碼前，先知道系統究竟會發生那些故事，是必備的基本功課。你可以利用熟悉或者自己發明的表示工具，描述你所找到的情境。甚至可以只利用簡單的列表，直接將它們列出。只要能夠達到記錄的目的，對程式碼閱讀來說，都能夠提供幫助。或者，你也可以利用UML中的類別圖、合作圖、循序圖之類的表示方法，做出更詳細的描述。

當你能夠列出系統可能會有的情境，表示你對系統所具備的功能，以及在各種情況下的反應，都具備概括性的認識。以此為基礎，便可在任何需要的時候，鑽進細節處深入了解。

探索架構的第一步──找到程式的入口

在之前，我們在一個開發專案中，曾經需要將系統所得到的MP3音訊檔，放至iPod這個極受歡迎的播放設備中。

雖然iPod本身也可以做為可移動式的儲存設備，但並不是單純地將MP3檔案放到iPod中，就可以讓iPod的播放器認得這個檔案，甚至能夠加以播放。

這是因為iPod利用一個特殊的檔案結構（iTunes DB），記錄播放器中可供播放的樂曲、播放清單以及樂曲資訊（例如專輯名稱、樂曲長度、演唱者等）。為了了解並且試著重複使用既有的程式碼，我們找到了一個Winamp的iPod外掛程式（Plug-In）。

Winamp是個人電腦上極受歡迎的播放軟體，而我們找到的外掛程式，能讓Winamp直接顯示連接至電腦的iPod中的歌曲資訊，並且允許Winamp直接播放。

我們追蹤與閱讀這個外掛程式的思路及步驟如下，首先，我們要先了解外掛程式的系統架構。很明顯的，大概瀏覽過原始碼後，我們注意到它依循著WinAmp為Plug-In程式所制定的規範，也就是說，它是實作成Windows上的DLL，並且透過一個叫做winampGetMediaLibraryPlugin的DLL函式，提供一個名為winampMediaLibraryPlugin的結構。

當我們不清楚系統的架構究竟為何時，我們會試著探索，而第一步，便是找到程式的入口。如何找到呢？這會依程式的性質不同而有所差別。

對一個本身就是可獨立執行的程式來說，我們會找啟動程式的主要函式，例如對C/C++來說就是main()，而對Java來說，便是static void main()。在找到入口後，再逐一追蹤，摸索出系統的架構。

但有時，我們所欲閱讀的程式碼是類別庫或函式庫，它只是用來提供多個類別或函式供用戶端程式（Client Program）使用，本身並不具單一入口，此類的程式碼具有多重的入口——每個允許用戶端程式呼叫的函式或類別，都是它可能的入口。

例如，對WinAmp的iPod Plug-In來說，它是一個DLL形式的函式庫，所以當我們想了解它的架構時，必須要先找出它對外提供的函式，而對Windows DLL來說，對外提供的函式，皆會以dllexport這個關鍵字來修飾。所以，不論是利用grep或gtags之類的工具，我們可以很快從原始碼中，找到它只有一個DLL函式（這對我們而言，真是一個好消息），而這個函式便是上述的winampGetMediaLibraryPlugin。

系統多會採用相同的架構處理Plug-In程式

如果經驗不夠的話，也許無法直接猜出這個函式的作用。

不過，如果你是個有經驗的程式人，多半能從函式所回傳的結構，猜出這個函式實際的用途。而事實上，當你已經知道它是一個Plug-In程式時，就應該要明白，它可能採用的，就是許多系統都採用的相同架構處理Plug-In程式。

當一個系統採用所謂Plug-In形式的架構時，它通常不會知道它的Plug-In究竟會怎麼實作、實作什麼功能。它只會規範Plug-In程式需要滿足某個特定介面。當系統初始化時，所有的Plug-In都可以依循相同的方式，向系統註冊，合法宣示自己的存在。

雖然系統並不確切知道Plug-In會有什麼行為展現，但是因為它制定了一個標準的介面，所以系統仍然可以預期每個Plug-In能夠處理的動作類型。這些動作具體上怎麼執行，對系統來說並不重要。這也正是物件導向程式設計中的「多型」觀念。

隨著實務經驗，歸納常見的架構模式

我想表達的重點，是當你「涉世越深」之後，所接觸的架構越多，就越能觸類旁通。只需要瞧上幾眼，就能明白系統所用的架構，自然就能夠直接聯想到其中可能存在的角色，以及角色間的關係。

像上述的Plug-In程式手法，時常可以在許多允許「外掛」程式碼的系統中看到。所以，有經驗的閱讀者，多半能夠立即反應，知道像Winamp這樣的系統，應該是讓每個Plug-In程式，都寫成DLL函式庫。

而每個Plug-In的DLL函式庫中，都必須提供winampGetMediaLibraryPlugin()這個函式（如果你熟悉Windows的程式設計，你會知道這是利用LoadLibrary()和GetProcAddress()來達成的一種多型手法）。如果你熟悉設計模式，你更會知道這是Simple Factory Method這個設計模式的運用。

winampGetMediaLibraryPlugin()所回傳的winampMediaLibraryPlugin結構，正好就描述了每個Winamp Plug-In的實作內容。

善用名稱可加速了解

利用gtags這個工具，我們立即發現，這個Plug-In它所定義的init、quit、PluginMessageProc這三個名稱，都是函式名稱。這暗示在多型的作用下，它們都是在某些時間點，會由Winamp核心本體呼叫的函式。

名稱及命名慣例是很重要的。看到「init」，我們會知道它的作用多半是進行初始化的動作，而「quit」大概就是結束時處理函式，而PluginMessageProc多半就是各種訊息的處理常式（Proc通常是procedure的簡寫，所以PluginMessageProc意指Plugin Message Procedure）了。

「望文生義」很重要，我們看到函式的名稱，就可以猜想到它所代表的作用，例如：當Winamp嘗試著初始化一個Plug-In時，它會呼叫這個結構中的init函式，以便讓每個Plug-In程式有機會初始化自己；當Winamp打算結束自己或結束某個Plug-In的執行時，便會呼叫quit函式。當Winamp要和Plug-In程式溝通時，它會發送各種不同的訊息至Plug-In，而Plug-In程式必須對此做出回應。

我們甚至不需要檢視這幾個函式的內容，就可以做出推測，而這樣的假設，事實上也是正確的。

閱讀他人的程式碼(5)找到程式入口，再由上而下抽絲剝繭

根據需要決定展開的層數，或展開特定節點，並記錄樹狀結構，然後適度忽略不需要了解的細節—這是一個很重要的態度。因為你不會一次就需要所有的細節，閱讀都是有目的的，每次的閱讀也許都在探索程式中不同的區域。

探索系統架構的第一步，就是找到程式的入口點。找到入口點後，多半採取由上而下（Top-Down）的方式，由最外層的結構，一層一層逐漸探索越來越多的細節。

我們的開發團隊曾針對Winamp的iPod plug-in進行閱讀及探索，不僅找到入口點，也找出、並理解它最根本的基礎架構。從這個入口點，可以往下再展開一層，分別找到三個重要的組成及其意義：

- **init()：**初始化動作
- **quit()：**終止化動作
- **PluginMessageProc()：**以訊息的方式處理程式所必須處理的各種事件

展開的同時，隨手記錄樹狀結構

當我們從一個入口點找到三個分支後，可以順著每個分支再展開一層，所以分別繼續閱讀init、quit、以及PluginMessageProc的內容，並試著再展開一層。閱讀的同時，你可以在文件中試著記錄展開的樹狀結構。

- **init()：**初始化動作
- **itunesdb_init_cc()：**建立存取iTunes database的同步物件
- 初始化資料結構
- 初始化GUI元素
- 載入設定
- 建立log檔
- **autoDetectIpod()：**偵測iPod插入的執行緒
- **quit()：**終止化動作

● **itunesdb_del_cc()：**終止存取iTunes database的同步物件

● 關閉log檔

● 終止化GUI元素

● **PluginMessageProc()：**以訊息的方式處理程式所必須面臨的各種事件

● 執行所連接之iPod的MessageProc()

這部分必須要留意幾個重點。首先，應該一邊閱讀，一邊記錄文件。因為人的記憶力通常有限，對於陌生的事物更是容易遺忘，因此邊閱讀邊記錄，是很好的輔助。

再者，因為我們採取由上而下的方式，從一個點再分支出去成為多個點，因此，通常也會以樹狀的方式記錄。除此之外，每次只試著往下探索一層。從init()來看你便會明白。以下試著摘要init()的內容：

```
int init() {
itunesdb_init_cc();
currentiPod=NULL;
iPods = new C_ItemList;
…略
conf_file=(char*)SendMessage(plugin.hwndWinampParent,WM_WA_IPC,0,IPC_GETINIFILE);
m_treeview = GetDlgItem(plugin.hwnd LibraryParent,0x3fd);
//this number is actually magic :)
…略
g_detectAll = GetPrivateProfileInt("ml_ipod", "detectAll",0,conf_file)!=0;
```

…略

g_log=GetPrivateProfileInt("ml_ipod","log",0,conf_file)!=0;

…略

g_logfile=fopen(g_logfilepath,"a");

…略

autoDetectIpod();

return 0;

}

因為我們只試著多探索一層，而目的是希望發掘出下一層的子動作。所以在init()中看到像「itunesdb_init_cc();」這樣的函式呼叫動作時，我們知道它是在init()之下的一個獨立子動作，所以可以直接將它列入。但是當看到如下的程式行：

currentiPod=NULL;

iPods = new C_ItemList;

我們並不會將它視為init()下的一個獨立的子動作。因為好幾行程式，才構成一個具有獨立抽象意義的子動作。例如以上這兩行構成了一個獨立的抽象意義，也就是初始化所需的資料結構。

理論上，原來的程式撰寫者，有可能撰寫一個叫做init_data_structure()的函式，包含這兩行程式碼。這樣做可讀性更高，然而基於種種理由，原作者並沒有這麼做。身為閱讀者，必須自行解讀，將這幾行合併成單一個子動作，並賦予它一個獨立的意義──初始化資料結構。

無法望文生義的函式，先試著預看一層

對於某些不明作用的函式叫用，不是望其文便能生其義的。當我們看到「itunesdb_init_cc()」這個名稱時，我們或許能從「itunesdb_init」的字眼意識到這個函式和iPod所採用的iTunes database的初始化有關，但「cc」卻實在令人費解。為了理解這一層某個子動作的真實意義，有時免不了要往前多看一層。

原來它是用來初始化同步化機制用的物件。作用在於這程式一定是用了某個內部的資料結構來儲存iTunes database，而這資料結構有可能被多執行緒存取，所以必須以同步物件（此處是Windows的Critical Section）加以保護。

所以說，當我們試著以樹狀的方式，逐一展開每個動作的子動作時，有時必須多看一層，才能真正了解子動作的意義。因為有了這樣的動作，我們可以在展開樹狀結構中，為itunesdb_init_cc()附上補充說明：建立存取itunes database的同步物件。這麼一來，當我們在檢視自己所寫下的樹狀結構時，就能輕易一目了然的理解每個子動作的真正作用。

根據需要了解的粒度，決定展開的層數

我們究竟需要展開多少層呢？這個問題和閱讀程式碼時所需的「粒度（Granularity）」有關。如果我們只是需要概括性的了解，那麼也許展開兩層或三層，就能夠對程式有基礎的認識。倘若需要更深入的了解，就會需要展開更多的層次才行。

有時候，你並不是一視同仁地針對每個動作，都展開到相同深度的層次。也許，你會基於特殊的需求，專門針對特定的動作展開至深層。例如，我們閱讀Winamp iPod plug-in的程

式目錄，其實是想從中了解究竟應該如何存取iPod上的iTunes DB，使我們能夠將MP3歌曲或播放清單加至此DB中，並於iPod中播放。

當我們層層探索與分解之後，找到了parseIpodDb()，從函式名稱判斷它是我們想要的。因為它代表的正是parse iPod DB，正是我們此次閱讀的重點，也就達成閱讀這程式碼的目的。

我們強調一種不同的做法：在閱讀程式碼時，多半採取由上而下的方式；而本文建議了一種記錄閱讀的方式，就是試著記錄探索追蹤時層層展開的樹狀結構。你可以視自己需要，了解的深入程度，再決定要展開的層數。你更可以依據特殊的需要，只展開某個特定的節點，以探索特定的細目。

適度地忽略不需要了解的細節，是一個很重要的態度，因為你不會一次就需要所有的細節，閱讀都是有目的的。每次的閱讀也許都在探索程式中不同的區域；而每次探索時，你都可以增補樹狀結構中的某個子結構。漸漸地，你就會對這個程式更加的了解。

閱讀他人的程式碼(6)閱讀的樂趣：透過程式碼認識作者

即便每個人的寫作模式多半受到他人的影響，程式人通常還是會融合多種風格，而成為自己獨有的特色，如果你知道作者程式設計的偏好，閱讀他的程式碼就更得心應手。

閱讀程式碼時，多半會採取由上而下、抽絲剝繭的方式。透過記錄層層展開的樹狀結構，程式人可以逐步地建立起對系統的架構觀，而且可以依照需要的粒度（Granularity），決定展開的層次及精緻程度。

建立架構觀點的認識是最重要的事情。雖然這一系列的文章前提為「閱讀他人的程式碼」，但我們真正想做的工作，並不在於徹底地詳讀每一行程式碼的細節，而是想要透過重點式的程式碼「摘讀」，達到對系統所需程度的了解。每個人在閱讀程式碼的動機不盡相同，需要了解的程度也就有深淺的分別。只有極為少數的情況下，你才會需要細讀每一行程式碼。

閱讀程式碼是新時代程式人必備的重要技能

這一系列的文章至此已近尾聲，回顧曾探討的主題，我們首先研究了閱讀程式碼的動機。尤其在開放原始碼的風氣如此之盛的情況下，妥善利用開放原始碼所提供的資源，不僅能夠更快學習到新的技術，同時在原始碼版權合適時，還可以直接利用現成的程式碼，大幅地提高開發階段的生產力。所以，閱讀程式碼儼然成為了新時代程式人必備的重要技能之一。

接著，我們提到了閱讀程式碼前的必要準備，包括了對程式語言、命名慣例的了解等等。在此之後，我們反覆提起了「由上而下」的閱讀方向的重要性。

由上而下的閱讀方式，是因為我們重視架構更勝於細節。從最外層的架構逐一向內探索，每往內探索一層，我們了解系統的粒度就增加了一個等級。當你識別出系統所用的架構時，便能夠輕易了解在這個架構下會有的角色，以及它們之間的動態及靜態的關係。如此一來，許多資訊便不言可喻，毋需額外花費力氣，便能夠快速理解。

好的名稱能夠摘要性地點出實體的作用

追蹤原始碼時，固然可以用本來的方式，利用編輯器開啟所需的檔案，然後利用編輯器提供的機制閱讀，但是倘若能夠善用工具，閱讀程式碼的效率及品質都能大大提升。在本系列文章中，我們介紹了一些工具，或許你還可以在坊間找到其他更有用的工具。

我在這一系列的文章中，實際帶著大家閱讀、追蹤了一個名為ml_pod的開放原始碼專案。它是一個Winamp的iPod plug-in程式。在追蹤的過程中，我們試著印證這一系列文中所提到的觀念及方法。我們採用逐漸開展的樹狀結構來記錄追蹤的過程，並藉以建立起對系統的概觀認識。

就原始碼的閱讀來說，之前的討論涉及了工具面及技巧面。但還有一些主題不在這兩個範疇之內，例如，善用名稱賦予你的提示。名稱做為隱喻（Metaphor）的作用很大，好的名稱能夠摘要性地點出實體的作用，例如我們看到autoDetectIpod()，自然而然能夠想像它的作用在於自動（Auto）偵測（Detect）iPod的存在。

我們在展開樹狀結構時，有時候需要預看一層，有時卻不需要這麼做，便可得到印證。程式人都會有慣用的名稱以及組合名稱的方法，倘若能夠從名稱上理解，便毋需鑽進細節，可以省去相當多的時間。例如，當我們看到parseIpodDb()時，便可以輕易了解它是剖析（Parse）iPod的資料庫（DB），因此便不需要立即鑽進parseIpodDb()中查看底細。

儘管如此，能否理解程式人命名的用意，和自身的經驗以及是否了解原作者的文化背景，是息息相關的。

命名本身就是一種文化產物。不同的程式人文化，就會衍生出不同的命名文化。當你自己的經驗豐富，看過及接觸過的程式碼也多時，對於名稱的感受及聯想的能力自然會有不同。

這種感受和聯想的能力，究竟應該如何精進，很難具體描述。就我個人的經驗，多觀察不同命名體系的差異，並且嘗試歸納彼此之間的異同，有助於更快地提升對名稱的感受及聯想力。

轉換立場，理解作者的思考方式

除了工具及技巧之外，「想要閱讀程式碼，得先試著閱讀寫這個程式碼的程式人的心。」這句話說來十分抽象，或許也令人難以理解。

當你在閱讀一段程式碼時，或許可以試著轉換自己的立場，從旁觀者的角度轉換成為寫作者的心態，揣摩原作者的心理及處境。當你試著設身處地站在他的立場，透過他的思考方式來閱讀、追蹤他所寫下的程式碼，將會感覺更加流暢。

許多軟體專案，都不是由單一程式人所獨力完成。因此，在這樣的專案中，便有可能呈現多種不同的風格。

許多專案會由架構師決定主體的架構及運作，有既定實施的命名慣例，及程式設計需要遵守方針。在多人開發的模式下，越是好的軟體專案，越看不出某程式碼片段究竟是由誰所寫下的。

不過，有些開放原始碼的專案，往往又整合了其他開放原始碼的專案。有的時候，也很難求風格的統一，便會出現混雜的情況。好比之前提到的ml_pod專案，因為程式碼中混合了不同的來源，而呈現風格不一致的情況。

我在閱讀非自己所寫的程式碼時，會觀察原作者寫作的習慣，藉以對應到腦中所記憶的多種寫作模型。在閱讀的過程中，讀完幾行程式碼，我會試著猜想原作者在寫下這段程式碼時的心境。他寫下這段程式碼的用意是什麼？為什麼他會採取這樣的寫法？順著原作者的思考理路閱讀，自己的思考才能更貼近對方寫作當時的想法。

當你短暫化身為原作者時，才能更輕易的理解他所寫下的程式碼。

如果你能知道原作者的背景，程式設計時的偏好，閱讀他的程式碼，就更能得心應手了。

從程式碼著手認識作者獨有的風格，進而見賢思齊

我在閱讀別人寫下的程式碼時，我會試著猜想，原作者究竟是屬於那一種「流派」呢？每個人都有自己獨特的寫作模式，即便每個人的寫作模式多半受到他人的影響──不論是書籍的作者、學習過程中的指導者，或一同參與專案的同儕，但每個程式人通常會融合多種風格，而成為自己獨有的風格。

物件導向的基本教義派，總是會以他心中覺得最優雅的物件導向方式來撰寫程式。而閱讀慣用、善用設計模式的程式人所寫下的程式碼時，不難推想出他會在各種常見的應用情境下，套用哪些模式。

有些時候，在閱讀之初，你並不知道原作者的習性跟喜好，甚至你也不知道他的功力。但是，在閱讀之後，你會慢慢地從一個程式人所寫下的程式碼，開始認識他。

你或許會在閱讀他人的程式碼時，發現令人拍案叫絕的技巧或設計。你也有可能在閱讀的同時，發現原作者所留下的缺失或寫作時的缺點，而暗自警惕於心。這也算是閱讀他人程式碼時的一項樂趣。

當你從視閱讀他人的程式碼為畏途，轉變成為可以從中獲取樂趣的時候，我想，你又進到了另一個境界。