國立政治大學資訊科學系
Department of Computer Science
National Chengchi University

碩士論文

Master's Thesis

AspectW：剖面導向之例外處理與重試機制

AspectW: an Aspect-Oriented Catch and Retry Mechanism

研 究 生：高沛功

指導教授：陳　恭

中華民國一百零三年六月

June 2014

AspectW：剖面導向之例外處理與重試機制

AspectW: an Aspect-Oriented Catch and Retry Mechanism

研 究 生：高沛功　　Student：Pei-Gong Kao

指導教授：陳　恭　　Advisor：Kung Chen

國立政治大學

資訊科學系

碩士論文

A Thesis

submitted to Department of Computer Science

National Chengchi University

in partial fulfillment of the Requirements

for the degree of

Master

in

Computer Science

中華民國一百零三年六月

June 2014

AspectW：剖面導向之例外處理與重試機制

摘要

雖然絕大多數的現代程式語言都有嚐試與補獲（try-and-catch）的例外處理機制，提供開發人員撰寫模組性高的例外處理程式碼，既可以立即處理例外狀況，也可以將例外傳遞（propagate）到系統其他模組。但是很多例外情況是暫態的（transient），發生後，應用程式是有可能從例外狀況恢復（recovery）過來，而不必啟動例外處理的程序。例如：分散式系統在進行網路連線時，可能因為網路一時不穩定而失敗，但稍停幾秒後再進行連線就可以了。於是就有學者倡議擴充例外處理機制，增加補獲與重試（catch-and-retry）的功能。本研究採用剖面導向（aspect-oriented）的觀點，參考 AspectF 的方法來實作一個輕量級的補獲與重試模組，AspectW，讓開發人員可以“流利（fluent）介面”的方式輕鬆撰寫例外捕獲與重試的程式碼，達到了讓開發人員不必更動主功能邏輯程式碼就能簡單地加入補獲與重試的功能。

關鍵詞—Exception, Exception handling, Try-and-Catch, AOP, Catch-and-Retry

AspectW: an Aspect-Oriented Catch and Retry Mechanism

Abstract

Most modern programming languages support try-and-catch mechanism that enables developers to write modular exception handling code which can not only process exceptions immediately but also propagate them to the other modules in the system. However, many exceptions are transient, when occurred, the application is likely to recover from the exception, thereby rendering it unnecessary to start up the exception handling code. For example: during a network connection in a distributed system, you may fail due to network instability moment, suffice to wait for a few seconds and then re-connect it. Thus, some scholars initiate to expand the exception handling mechanism, and proposed catch-and-retry mechanism. This thesis takes an aspect-oriented point of view, and adapts the AspectF library to implement a lightweight level catch-and-retry library, AspectW, so that developers can use the "fluent interface" approach to easily write exceptions capture and retry code. As a result, developers do not have to modify the main logic code and could simply add catch-and-retry code to handle exceptions well.

Keywords—Exception, Exception handling, Try-and-Catch, AOP, Catch-and-Retry

誌謝

首先誠摯的感謝我的指導教授陳恭老師，從提案開始就嚴格細心的指導，不時的討論並指點我正確的方向，研究其間數次的進度報告也不斷指正，使我在這二年中獲益匪淺。

兩年的日子裡也要感謝同學們的支援。因為是在職專班相處時間不算長，但透過網路科技可以明顯感受到這真是一群嘰嘰喳喳的同學們，在嘰喳有餘中課程作業的相互支援、互相玩笑。也要感謝學長們的幫助與學習心得的分享。

感謝我親愛的家人，感謝父母養育我、同理我、關心我，在我人生低點時給予我溫馨的打氣。感謝兄長看重我、鼓勵我、支持我，使我仍能保有前進的動力。女朋友在背後的默默支持更是我前進的動力，沒有的體諒、包容，相信這兩年的生活將是很不一樣的光景。

最後，謹以此文獻給我摯愛的雙親。

高沛功 謹誌

國立政治大學　資訊科學研究所在職專班

中華民國 １０３年3月１８日。

目錄

表目錄

圖目錄

第 1 章

緒論

1.1 前言

補獲與重試(Catch-And-Retry)機制並非是個新穎的觀念。對於處理通訊網路上暫態的(transient)故障，在雲端運算(cloud computing)時代以前是較不重視的，大都當作是偶發的例外。而量變造成質變，網際網路社交應用掘起，如：Facebook、Twitter、WhatsApp 等的大規模分散式系統(large-scale distributed systems)，每日傳遞的訊息量已達十億次數等級了。在百萬次通訊等級偶發的例外故障，到了每日十億次通訊等級的規模後就不是偶發而是“常常”了，對於提供服務的廠商若不處理這“常常”發生的故障，對本身的產品品質形象的質變是會有危害的，當客戶們從網路等媒體開始流傳品質不良的傳言是會影響產品行銷的。

透過補獲與重試機制並無法根治大規模分散式系統在長程通訊上暫態的例外故障，通訊對象可能位在地球另一端，訊號路徑可能長達數千公里且路程還有無數個交換設備。不過補獲與重試機制可以降低通訊故障的發生機率讓品質不良的質變效應不要出現。在技術上套用補獲與重試機制對於開發人員來說常常是困擾人的，尤其是當系統已上線一段時間在功能調整、需求增加與系統維護後程式碼已變成了交互糾纏(tangled)的狀態。本研究採用以 AOP(aspect-oriented programming)觀念為基礎，設計了 AspectW 模組讓開發人員可以方便、簡易又彈性的加入補獲與重試機制。

1.2 研究動機

例外處理(exception handling)的研究一直是程式語言研究發展的重要議題。在今天的程式語言中，例外(exception)為開發人員提供了當發生一些故障或其他特殊情況時引發和傳遞出訊息的機制，此機制稱為嚐試與補獲(try-and-catch)。而這機制並不包括進行相應的反應，並從這些故障和其他條件中恢復(recovery)，因為這些復甦過程通常是依各應用程式狀況而定。然而，我們相信在大規模分散式系統有一套通用的復甦機制存在，因為我們觀察到一個現象。特別是在大規模的網絡服務，以及其他分散式系統中，雖有多種的故障與錯誤要對抗，而我們發現對抗的方式就是什麼都不做僅僅稍停數秒再重新執行剛剛出錯的指令即可。在瀏覽網站與寄送電子郵件(email)時偶有這樣同樣的體驗。把這樣的體驗轉換成一個機制，就是補獲與重試(catch-and-retry)機制。

補獲與重試機制其實是軟體在通訊應用時品質提昇的一個機制，在單機系統上討論這一項是沒有什麼意義的，在討論系統架構時常是被忽略的。它在大規模分散式系統架構下才會顯得出意義。在討論軟體系統演算法或是建構系統的主要架構功能時通常也不會把它放在心上。在建構系統的雛型階段這項考量在大部份況狀是不須要的。問題就出在後續的實作到上線的過程，它就真的被忽略了以致系統上線後問題才開始。對於接手維護的開發人員來說，這些長期維護的系統程式碼只有糾纏不清。常為了加入一項機制而必需牽動許多的程式碼。這又讓程式碼糾纏的狀態更加的嚴重，整體狀況變成了負向循環，也讓程式碼設計意圖的可讀性與理解度不斷的下降。

要讓程式碼在長期維護過程中不會互相糾結纏繞是有方法的就是導入 AOP[1]。AOP 的研究背景原因之一就是要對抗糾纏不清的程式碼，也有了相當多的研究基礎與應用。我們研究了數種 AOP 實作方案，發現了一個輕量、簡單又實用的 AOP 技術，在考量評估後決定引進。我們引進了 AspectF[9]的 AOP 實作方法，在觀點上它提高對縫合(weaving)的重視，在技術面上導入 “流利介面(fluent interface)”[10]的設計，讓開發人員

使用時可以更順手也更有彈性，可以在流程程式碼中依需要把剖面功能縫合到流程邏輯之中並成為一體。

在此研究我們結合了 AOP 觀念與 AspectF 的實作方法再加入為補獲與重試機制設計的剖面功能指令，設計並實作出 AspectW，可以用來防止程式碼慢慢地變成糾纏程式碼(tangled code)狀態，也可以同時簡單地把補獲與重試機制加入到程式邏輯裡而不必牽動過多程式碼。

1.3 研究目的

實現補獲與重試機制可以從三個層面下手： 1)在系統初始設計階段就從結構面最小化通訊故障帶來的衝擊。2)執行平台支援，使用具有補獲與重試能力的執行環境。3)應用程式特別制定。本研究著重在應用程式特別制定這一層。

在組織流程程式碼時又可拆分成主功能邏輯程式碼與非主功能機制程式碼，這一點剛好與 AOP 觀念相呼應，其中非主功能機制可以對照成剖面功能。補獲與重試是屬於非主功能機制，也就是輔助的，所以常在開發過程被忽略總是在事後回頭來加入，而在這之前若未導入 AOP 的話通常要牽動許多已組織好的程式碼，因而會有意想不到的狀況發生。

研究 AOP 在商務領域的應用。我們可以想像如圖 1.1 OOP 程式碼模型圖，一個網路

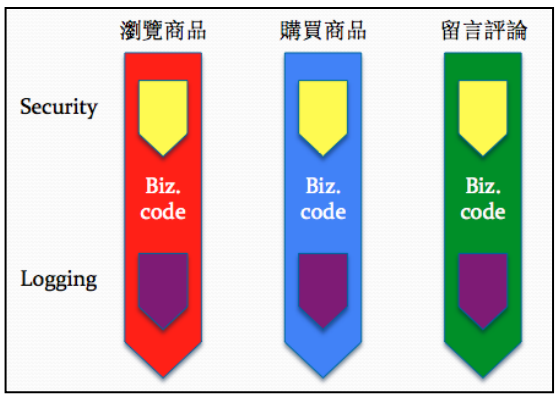


圖 1.2 OOP 程式碼模型圖

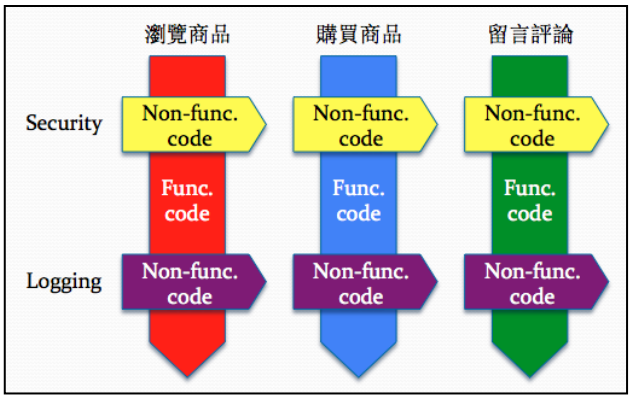


圖 1.1 AOP 程式碼模型圖

購物平台系統透過 OOP(Object-oriented programming)方法把各作業功能切割後，比如：瀏覽商品、購買商品、留言評論，各個子功能在理想上是完全獨立不相依的。可仔細查看內部每道完整的作業流程中都有“security”程式碼段落檢查是否依然有效的登入認證，還有“logging”記錄使用行為的程式碼。這相同局部的程式碼幾乎完全一樣，所以會做成共用元件，只要共用元件一改變全部流程都改變了，也就是說各業務功能在實際上並非完全不相依。

改用 AOP 的觀念來看同樣的內容。請參考圖 1.2 AOP 程式碼模型圖，圖中“Func. code”代表主功能邏輯程式碼；“Non-func. code”代表非主功能機制程式碼。AOP 的研究告訴我們有些程式碼就是無法被明白的封裝起來，這部份程式碼就定義為剖面(aspect)。這些非主功能機制程式碼基本上都是剖面。這些剖面普遍性散佈在各作業功能的局部，有些目的相同、邏輯也相當所以是可以共用的，可以把共同的機制做成可重複使用的元件，在使用上差異的部份用參數化來處理。透過縫合(weaving)把剖面也就是就非主功能機制與主功能邏輯組合成完整的流程。在縫合的過程中主功能邏輯程式碼不必為了配合而做邏輯調整，也就是不會有像前面所提牽一髮動全身的狀況了，只需針對性的更動局部就行了。

重新整理這份研究的目標，主要有：

一、 具有 AOP 的優勢。

二、 提供容易使用的語言機制(language mechanism)，讓開發人員可以簡易地在系統內加入補獲與重試機制。

三、 其中補獲與重試機制並具有兩項特性：隔離(isolation)、復原(recovery)。

在補獲與重試機制部份，它不只是一個指令而已，也不是終端使用者會提出的需求，它是一個產品品質議題。研究此機制所真實面對的故障情境，與開發人員經常使用的應對動作，並考量整體運作的行為模式再轉換成指令設計方向，整理如下：

一、 Retry，補獲到故障後重新執行出錯的指令，過程中可能會適當的變更參數。

二、 Restore，補獲到故障後立即進行應用程式狀態(application state)的還原。

三、 Ignore，忽略補獲到的故障。有時有些故障是可以忽略的，比如：按讚。可能有重試過幾次可最終就是失敗，這時忽略若不影響大局就忽略掉吧。

四、 Handle Fail，用於進階的操作。補獲到故障無法以常用的方法應對時，就必需再依當時狀況制定應對措施。

以上這四類指令設計於呼應補獲故障時通常的行為模式：是否要重試？可以忽略不管嗎？故障使得應用程式狀態不正常嗎？要復原狀態嗎？常用的方法無效必須依個案加入特別的措施嗎？

1.4 研究成果

本研究基於前述研究動機、研究目的並蒐集相關文獻與技術探討。實作文獻中提到的可行方案，綜合了研究成果設計了一個模組，一個具有使用上高彈性的類別，取名為“AspectW”，此名稱中的“W”代表著強調縫合(weaving)的能力。這是一個小型模組也可以用小工具來形容，它極輕量。AspectW 除了提供一些已被判定為剖面功能的基本指令外，最重要的是特別為補獲與重試機制設計了一組動作指令。依照前一節研究目的提到的四個應對例外故障的動作指令，這些指令也依面對的狀況差異提供了一些多載(overloading)版本，讓開發人員可以輕鬆的使用。

此研究主要的成果有：

一、 引入一種極輕量的 AOP 實作技術，AspectF。應用 lambda expression 的原理特性即可實作出來，不必複雜的 dynamic proxy、IL(Intermediate Language)等技術。

二、 重構 AspectF[9]的核心，設計了 AspectW，剖面功能改以補獲與重試機制為主要應用。

三、 整理出了補獲與重試機制的四個動作指令設計方向：retry、restore、ignore、handle-failure。其中 retry 指令再分成 basic retry 與 parameterized retry。這些指令可以在不影響主功能邏輯程式碼下可簡單輕鬆的加入或移除。

1.5 論文大綱

本論文主要分成五個章節，第一章為緒論，主要在介紹整個研究的源起與概要特點，包括了前言、研究動機、研究目的、研究成果及各章節概述。第二章是相關研究與技術背景，補獲與重試機制的來龍去脈介紹。AOP 的研究心得。以 AOP 實現補獲與重試機制要具備的規格與特徵。也研究 AOP 實作的領導者之一 AspectJ 的原理與使用方法。研究過程中的幾種其它實現方案與 AspectJ 大同小異就不再特別討論。也特別要介紹 AspectF 這個評估後決定採用的 AOP 解決方案，它提出了對剖面(Aspect)的新看法與實作原理。第三章詳細說明設計理念、設計考量、為何決定打造 AspectW 與設計原理。第四章嘗試以真實的案例來展示此系統。最後第五章提出結論與未來可能的發展。

第 2 章

相關研究與技術背景

這一章節將介紹此研究的背景與主要歷程，過程中一些不打算採用的部份則忽略不記。在章節 2.1 Catch And Retry 介紹源由； 章節 2.2 Aspect-Oriented Programming 介紹 AOP 的觀念;章節 2.3 Implementing Retry 介紹達成的規格與特徵;章節 2.4 AspectJ 介紹 AOP 實作的領頭者；章節 2.5 AspectF 介紹本研究決定採用的實作技術。

2.1 Catch And Retry

Catch And Retry[3]是一個程式語言編碼的機制(mechanism)，適合應用在多層架構的分散式應用系統。這個系統可能是以大規模伺服器端架構(large-scale server-side infrastructure)為主的應用，有資料中心(data center)、它的用戶端(client-side)可能是以 JavaScript 編製的 web application，如：Facebook、Google+、Twitter 等類似的社交應用系統。

在網際網路盛行的現代，資訊系統架構分分合合又分分，從大型主機架構[5]為主演進成主從式架構(如圖 2.1)近幾年有變成行動計算架構為主(如圖 2.2)。在行動計算架構下系統的離散程序又更勝以往。大型主機架構或主從式架構大概是一台伺服器端服務好幾台客戶端。到了行動計算架構就變成可能好幾台伺服器同時服務非常多台的客戶端。而資料訊息在此複雜的網路通訊路徑傳遞過程中出現例外性錯誤的機率將大大的提昇。

在今天的程式語言中，例外(exceptions)為開發人員提供一項機制用以觸發和傳遞已發生的一些錯誤或其他特殊情況的訊息(signal)。一般來說這些機制都沒有產生對應

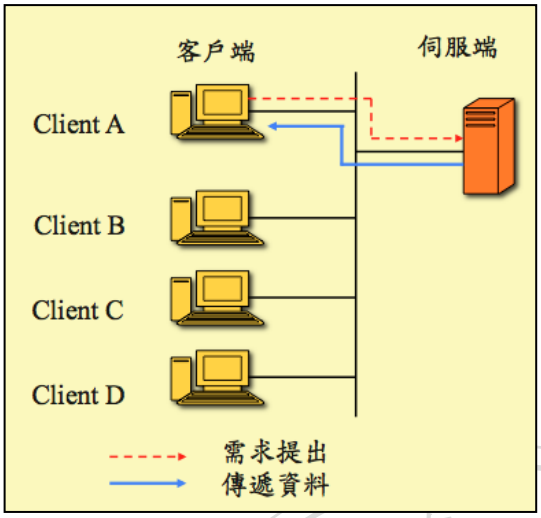


圖 2.2 主從式架構圖

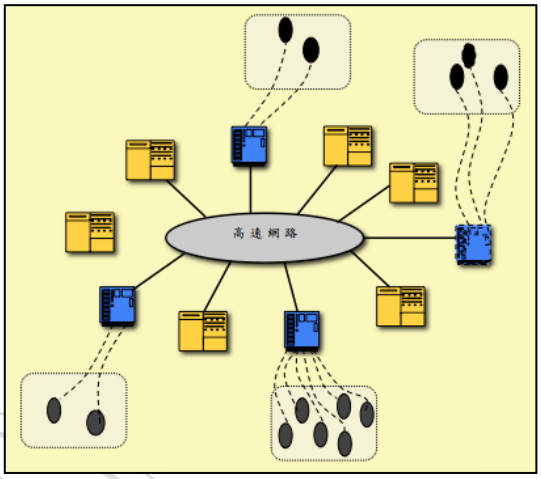


圖 2.1 行動計算架構圖

的反應，並從這些錯誤訊息和其他條件進行復原，因為這樣的復原反應通常是由應用程式另外再特別的加工來支援。

在今天的程式語言中，例外(exceptions)為開發人員提供一項機制用以觸發和傳遞已發生的一些錯誤或其他特殊情況的訊息(然而，我們相信存在一套復原機制(recovery mechanism)可以套用在分散式系統中。

特別是大規模的網絡服務以及其他分散式系統中，必須對抗多種多樣性常見的故障和錯誤，這些錯誤可以統括分類成二類：

- 資料性錯誤(Data errors): 維持資料的新鮮度和一致性在分散式系統中的代價是昂貴的，甚至在某些狀況下是不可能辦到的。這樣的資料錯誤可能包括讀取過時(reading stale)的值或讀取來自多個不同源頭的資料庫。

- 可用性錯誤(Availability errors): 也或許是因為網路分區(network partitions)，硬體故障(hardware failures)，性能跳針(performance stutters)，或軟體缺陷(software bug)的關係，當遠端的服務已失效或反應緩慢時，分散式系統的編碼工作中必須對這些狀況都有對應的準備與處理機制。

當然 Data errors 並非完全與 Availability errors 完全無關，有些 Data errors，比如：資料遺失，可能是因為某部份的儲存體可能失效造成的。這些在網際網路服務和其他許多分散式系統的錯誤通常可以在各個設計階層進行處理： 1）在系統初始設計結合節點和數據複製以盡量減少這些誤差的影響； 2）在系統基礎架構層級檢測和修復相關的故障（例如，通過重新啟動、重新映像失敗的節點）； 3）在應用程式編程階段處理使用者介面與系統性能部份。

整理了這些方法，可以實際運用的復原機制包括：

- Re-execution of failed operations：重新執行出錯的指令。
- Scheduling control：重新排程執行出錯的指令，指定該指令在何時(when)、何處(where)、是否(whether)執行。

這兩個機制可以混合同時使用，也是大部份已被常用的技術原則。Catch And Retry 是建造一個程式碼區塊(block)的復原機制。其結果是一個易於使用的語言機制，可以大大簡化復原程式碼的撰寫。然而只是重新嘗試是有缺陷的，比如無限的重新嘗試錯誤的指令。解決的方法之一是簡單的指定重試次數以避免無限循環的危險，處理多重的例外時也可以變更參數後再重新執行原指令，這些參數也可提供在重試排程中。

Catch And Retry 重新嘗試在語意上的關鍵字整理起來只須幾個： try, catch, retry, fail, finally。在功能面部份要能滿足幾項需求：

1）指定重新嘗試次數，以避免無限次迴圈的危險。

2）可以處理多重的例外，同一指令可能有多種可能的錯誤出現。

3）在重新嘗試時可以被參數化，不同的參數但滿足相同的目的，或許品質可能降低一些但目的不變。 下面將以案例來說明這些。

4) 對指令設定排程參數，當該指令出錯時依排程參數重新嘗試。

2.1.1. Basic Retry

當程式執行過程中一但由例外機制發出了資料性錯誤或可用性錯誤，在程式語言階層上使用 retry 關鍵字表示要重新嘗試 try 區塊程式碼。範例將以 C#語法說明，類似的語法在不同的語言如 Java、JavaScript 一樣有效用。一個選擇性的整數參數指定可重新嘗試 try 區堆幾次，如：“retry 3;”。表示可以最多可以重新嘗試 3 次且這參數必需是常數。若沒有指定重試次數那就假設與“retry 1;”同意。限制允許的重試次數可以防止可能發生無限的重試。圖 2.3 是一個範例。

儘管試圖依多次的重新嘗試特定的 try 區塊，還是會有這些重新嘗試的動作最終還是失敗的狀況。為了處理這種情況，每個 try 區塊可以選擇性的伴隨一個 fail 區塊用來處理指定的重新嘗試次數後依然還是失敗的狀況。在 fail 區塊若希望把例外送往更外層處理，那就簡單的再“throw;”不必指定參數。需注意一點，若在多重例外的狀況下將以第一個遇到的 fail 區塊為主。

```
try {
   Console.WriteLine("reading file...");
   // code to read a file from the network
   ...
} catch( StalenessException e ) {
   // wait 5 seconds before retrying
   Thread.Sleep(TimeSpan.FromSeconds(5))
   Console.WriteLine("about to retry...");
   retry; // retry once
   Console.WriteLine("retried the read");
} fail {
   Console.WriteLine("retry failed");
   throw; // re-throw it, when we wish.
}
```

圖 2.3 Basic Retry

2.1.2. Parameterized Retry

愛因斯坦定義過何謂精神錯亂：“同樣的事情做了一遍又一遍，並期待有不同的結果”。但在在分散式系統中，重新嘗試相同指令常可導致不同的結果，例如：當故障(fault)只是暫態的(transient)或者該系統具有不確定性(non-deterministic)。

在 Parameterized Retry 的語法中把參數明列放在 try 後面，表示這是在重新嘗試時為可修改的。若該參數沒指定新值就使用原值。在 try 區塊中不可以變更這些明列在

try 後的參數。要注意一點，在 try 區塊中是沒有交易屬性的。換句話說，重新嘗試 try 區塊之前可能會已有副作用(side-effects)了，這是在上一次的執行 try 區塊時的例外往外丟時留下的。

```
// retry parameter
string protocol="HTTPS";
try(protocol) {
   SendMessage(friend, message, protocol);
} catch( AvailablityException e ) {
   // wait 5 seconds before retrying
   Thread.Sleep(TimeSpan.FromSeconds(5))
   retry(protocol="HTTP");
}
```

圖 2.4 Parameterized Retry

在本文提供的圖 2.4 Parameterized Retry 範例中，主要流程目的是 SendMessage，目的是採用 HTTPS 協定傳送訊息給朋友，若失敗時把協定換成 HTTP 再重新傳送訊息給朋友，使用不同的方法，但目的是一樣的。

2.1.3. Recovery From Multiple Exception

一個指令執行時可能有多種的失敗狀況，這是常見的分散式系統狀況之一，故可以處理多重失敗是 Catch-And-Retry 機制必要的功能之一。

例如本文中的範例，如圖 2.5，從網絡讀取資料時若網路連線出現故障時會先稍為稍停一小段時間再重新嘗試，稍停一小段時間是因為在分散式系統的網路錯誤常只是短暫的。在重試了五次還是連不上就認定網路已確實斷線了。在此同時，若遇到的資料讀取過時的狀況這時就先刷新資料後再重新嘗試讀取資料程序。Catch-And-Retry 會自動分派正確的例外處理的 catch 區塊。

```
try {
   ReadData();
} catch(NetworkConnectivityException e) {
   Thread.Sleep(TimeSpan.FromSeconds(5));
   retry 5;
} fail {
   Console.Writeline("network failure");
} catch(StaleDataException e) {
   RefreshData();
   retry 10;
}
```

圖 2.5 Recovery From Multiple Exceptions

2.1.4. Scheduling Control Over Retries

雖然發生故障時立即重新嘗試往往是有用的。也有很多時候，我們知道立即重新嘗試機制是不可能成功的，除非有些額外的條件先滿足才行。比如，若遠端的主機需先完成重新開機的程序；比如，有些遠端的中繼資料或 cache 尚未完成刷新程序等。

為了支援此情況，必需擴展 catch 區塊的功能讓它可以接受有明確指定重新嘗試的操作函式。以本文範例圖 2.6 中的 r 函式，此函式抓取（capture）原所屬的 try 區塊的功能，這個函式 r 同時也有封裝性(closure)的特質。要注意，此機制的一個重要的問題是可能會破壞原先設計好的流程，要注意流程結構不要因為重新排程而破壞。也有些功能邏輯是無流程性的或是可有可無的，比如 Facebook 中的按讚功能。

```
try{
 //do something
 ...
}catch(AvailabilityException e,
      RetryFunction r) {
 //allow the application to schedule
 // the try block asynchronously
 scheduleForLater(r);
}
```

圖 2.6 Scheduling Control Over Retries

2.2 Aspect-Oriented Programming

在這一段將說明 Aspect-Oriented Programming[1](AOP, 剖面導向程式設計[2])的觀念。AOP 是在 1997 年由 Xerox Palo Alto 實驗室所提出，此概念一提出立即引起程式語言與軟體工程方面學者與專家的重視與迴響，之後相關的各種研究陸續發佈出來。其中又以該實驗室以 Java 為基礎的 AOP 程式語言 AspectJ 最盛行。

AOP 的研究背景是基於發現使用 OOP(object oriented programming)方法來開發系統的許多問題。OOP 無法足夠有效的清楚的把程式碼中的各局部的內容用意清楚表達，最終程式碼總是會變成複雜的糾纏程式碼(tangled code)狀態，即程式碼交互纏繞難以理解該段代碼的邏輯與用意，不但維護難度增加，若要加入新的機制也是難上加難且還常有意想不到的例外狀況。

軟體設計流程(software design process)和程式語言(programming language)存在著一個相互支援的互助合作關係。在軟體設計流程中會把系統向下分割成較小的單元。程式語言提供機制讓開發人員(programmer)把這些抽象的系統子單元各別實作出來，然後這些子單位再依不同的方式組合，最後產生整個系統。若程式語言對這些抽象

過的子單位本身就有合適的語法機制支援，且設計流程分割出來的子單元也適合程式語言實作的話，它們的合作就會相當的良好，程式碼也能清楚的表達各局部的設計用意。可實際上設計流程與程式語言之間的鴻溝是不窄的，比如：商業程序、認證機制、參數驗證機制、交易機制、記錄機制等，同時組合在一整串的處理流程。

用相對的角度來看，一個軟體系統可以區分成可被封裝的元件(component)與無法被封裝的剖面(aspect)。而這兩部份可以被“縫合(weave)”成一體。

在真實的應用系統開發上其程式碼有三種狀況：容易理解但效率差，有效率但不易理解，還有以剖面基礎(AOP-based)實作可以同時有效率與容易理解。換句話說，AOP 的目的就是要讓程式程式碼具有高度的可讀性，同時也保有高度的執行效率。

2.2.1 AOP 術語

有關於 AOP 的許多名詞術語都過於抽象不易理解，單從字面上並不容易理解其名詞意義，這邊將配合圖形來說明。

一、Cross-cutting concern

在幾個不同的業務流程程式碼中，記錄程式碼常以橫切(cross-cutting)型式加入。其它如安全(Security)檢查、交易(Transaction)等系統層面的服務(Service)，在一些應用程式之中也常被安插至各個業務物件的處理流程之中，這些動作在 AOP 的術語中被稱之為 Cross-cutting concerns。

以圖形來說明 Cross-cutting concerns 的意涵，例如圖 2.7，原來的業務流程是很單純的。現在為了要加入記錄(Logging)與安全(Security)

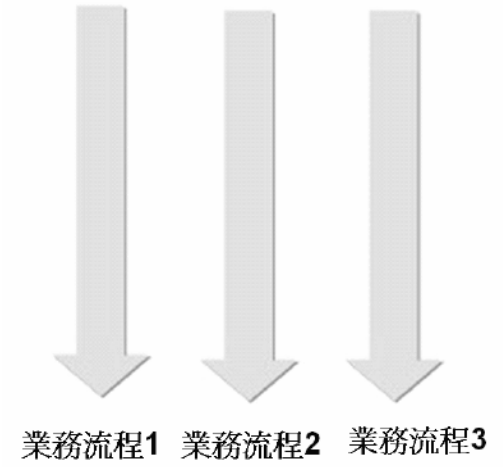


圖 2.7 AOP 概念圖解之一

檢查等服務，業務物件的程式碼中若被硬生生的寫入相關的 Logging、Security 程式片段，則可使用以下圖 2.8 圖解表示出 Cross-cutting 與 Cross-cutting concerns 的概念。

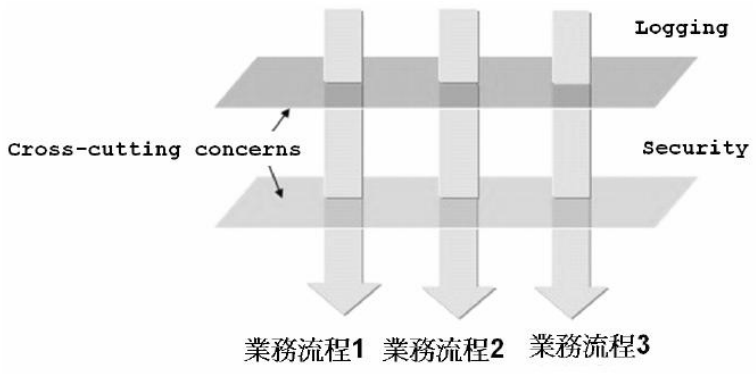


圖 2.8 AOP 概念圖解之二

Cross-cutting concerns 若直接撰寫在負責某業務物件的流程中，會使得維護程式的成本增高，例如若您今天要將物件中的記錄功能修改或是移除該服務，則必須修改所有撰寫曾記錄服務的程式碼，然後重新編譯，另一方面，Cross-cutting concerns 混雜於業務邏輯之中，使得業務物件本身的邏輯或程式的撰寫更為複雜。

二、Aspect

將散落於各個業務物件之中的 Cross-cutting concerns 收集起來，設計各個獨立可重用的物件，這些物件稱之為 Aspect。例如在登錄的動作設計為一個 LogHandler 類別，LogHandler 類別在 AOP 的術語就是 Aspect 的一個具體實例，在 AOP 中著重於 Aspect 的辨認，將之從業務流程中獨立出來，在需要該服務的時候，縫合(Weave)至應用程式之上，不需要服務的時候，也可以馬上從應用程式中脫離，應用程式中的可重用元件不用作任何的修改。另一方面，對於應用程式中可重用的元件來說，以 AOP 的設計方式，

它不用知道處理提供服務的物件之存在，具體的說，與服務相關的 API 不會出現在可重用的應用程式元件之中，因而可提高這些元件的重用性，您可以將這些元件應用至其它的應用程式之中，而不會因為目前加入了某些服務而與目前的應用程式框架發生耦合。

三、Advice

Aspect 的具體實作稱之為 Advice，以記錄的動作而言，Advice 中會包括真正的記錄程式碼是如何實作的，像之前提到的 LogHandler 類別就是 Advice 的一個具體實例，Advice 中包括了 Cross-cutting concerns 的行為或所要提供的服務。

四、Joint Point

Aspect 在應用程式執行時加入業務流程的點或時機稱之為 Join Point，具體來說，就是 Advice 在應用程式中被呼叫執行的時機，這個時機可能是某個方法被呼叫之前或之後（或兩者都有），或是某個例外發生的時候。

五、Point Cut

Point Cut 是一個定義，藉由這個定義您可以指定某個 Aspect 在哪些 Join Point 時被應用至應用程式之上。具體的說，您可以在某個定義檔中撰寫 Point Cut，當中說明了哪些 Aspect 要應用至應用程式中的哪些 Join Point。

六、Target

一個 Advice 被應用的對象或目標物件。

七、Introduction

對於一個現存的類別，Introduction 可以為其增加行為，而不用修改該類別的程式，具體的說，您可以為某個已撰寫、編譯完成的類別，在執行時期動態加入一些方法或行為，而不用修改或新增任何一行主功能邏輯程式碼。

八、Weave

Advice 被應用至物件之上的過程稱之為縫合(Weave)，在 AOP 中縫合的方式有幾個時間點：編譯時期(Compile time)、類別載入時期(Class loading time)、執行時期(Runtime)。

結合以上介紹過的 AOP 相關名詞具體的使用圖 2.9 來加以說明，有助於對每一個名

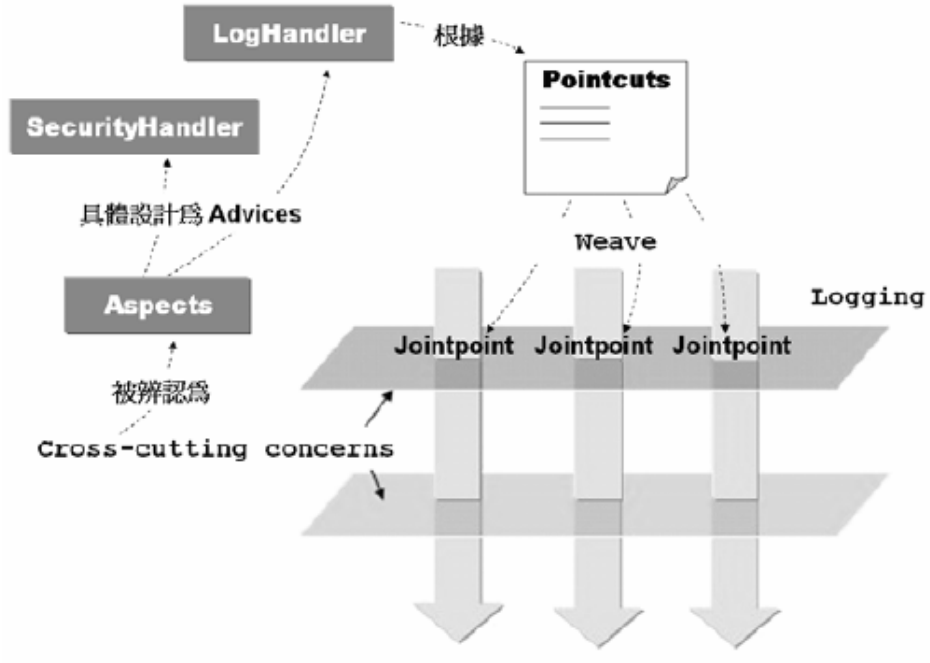
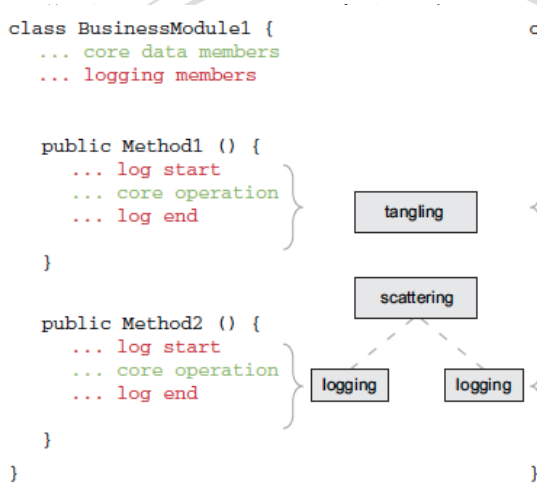


圖 2.9 AOP 術語示圖

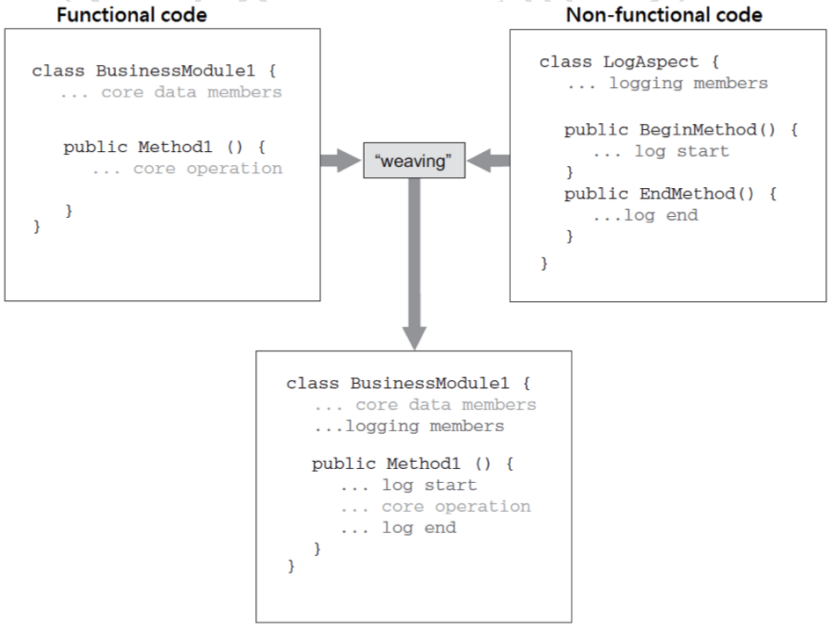
詞的理解與認識：

2.2.2 How AOP Works: Weaving

未使用 AOP 開發的軟體系統，開發過程中主功能邏輯與非主功能機制的程式碼是混合在一起的。當上線維護一段時間後，需求或規格的調整，程式碼也不段的增加或移除，原本就混合的程式碼再混入更多的程式碼。在這過程中，常會使用使用一個技巧，「複製、貼上」，這個技巧可以稱為“scattering”，因為這個技巧讓同一用意的程式碼分散在多處。慢慢地經過一段時間不斷的再增加或移除程式碼，這情形就像是「溫水煮青蛙」謎思一樣，程式碼會越來越讓人難以理解，通常在上線三、五年後的程式碼就會變成糾纏的(tangled



以下將以 (紅色)部份是非主功能機制 ng'的程式碼是分散的，雖 比如：購買商品，在購買前 實作。在之後客人可能買了 ” 位置實作。



在導入 AOP 時，這些‘logging’部份的程式碼可以移動到新的 class，而原來‘core’的程式碼原封不動。接下來使用 AOP 工具把‘core’與‘logging’程式碼依照指定好的“point cut”位置進行組合。這個組合的程序就稱為縫合(Weaving)。可以用圖 2.11 導入 AOP 圖例來說明。圖左“Functional code”是主功能邏輯程式碼 class；圖右“Non-functional code”是非主功能機制程式碼 class。在經過縫合程序後產生新的 class，這個 class 把主功能邏輯程式碼與非主功能機制程式碼組合成一體。

2.2.1 AOP Benefits

這一小節將說明使用 AOP 的好處，整理後簡述如下：

- Clean up spaghetti code

程式碼不再像義大利麵一樣交錯複雜，變得比較乾淨比較清析，也就會比較容易閱讀，也就比較容易理解。可以一眼就知道主功能程式碼在那，非主功能程式碼在那。

圖 2.11 導入 AOP 前

- Reduce repeating

使用 AOP 後，一個普遍常用的編碼技巧「複製、貼上」就不需要了。也就是程式碼重複的部份也大大減少。所謂良好編碼的一個重要指標 DRY(Don’t Repeat Yourself) 也可達到。

- Encapsulation

系統的封裝程度也會提昇，程式碼重複引用程度也獲得提昇。把邏輯程序中 Cross-cutting concern 的程式碼抽取出來移至新的類別，更改程式碼時也不用處處更改只需

圖 2.10 導入 AOP

更動一處即可。

2.3 Implementing Retry - Featuring AOP

每天都有經驗告訴我們，有些錯誤是非常短暫的，單單只須要重試(“retrying”)就能處理掉這些錯誤。比如，網路傳輸路徑上的電子訊號干擾可能讓遠端的設備在一個短暫的時間週期裡讓訊號無法送達。比如，一個遠端應用服務突然在一小段時間的存取衝到高峰，而讓部份請求服務被拒絕。在很多的情形下，也不需要提供任何種類的復原機制就能讓系統正常的持續運作去下，我們只須要重新執行這個失敗的指令即可。

可惜的是，重試指令在很多執行平台或是程式語言是沒有明確的支援的。且重試也不是萬能的不該盲目的使用在所有的不正常狀況。

一些程式語言，如：C#、Java 或有支援重試指令的程式語言，如：Smalltalk 和 Eiffel 其錯誤處理都是透過例外機制“exception”來溝通此錯號訊號。而且並沒有一個簡單的方法在企圖重試前去清除前一回合執行失敗時留下的產物。開發人員必須明確的提供指令程序且有時很複雜又傾向出錯的程式碼(error-prone code)，來修複程式執行產出的產物或還原到原先執行前的狀態。

在這份文件裡，我們打算以 AOP 技術實作“retrying”。此方法可以讓開發人員不必再特地去撰寫狀態清除程式碼(“state-clean” code)去清除指令錯誤時的不正常產出。

重試一個已出錯的指令是常被我們使用的一個修復策略。儘管如此，此策略只限制在幾個要點上才有效果，這些出錯的指令是不是“等冪的(idempotent)” --- 意指同一指令不管執行幾次的結果都是一樣的，且開發人員原本就打算讓一些指令或函數本身就具有可以重新執行的特性。

考慮當今的軟體資訊系統與網路或網際網路(WWW)大多有強烈的連結，那麼使用者或者應用程式對於這些極短暫性、臨時性的跳針般的錯誤現象要有通用性的應對方法。例如：使用瀏覽器去瀏覽某個網址時，第一次執行時出現了該網頁不存在的訊息回報，可稍停個二、三秒再重新瀏覽同個一網址卻成功了。同樣的，例如寫好了一封 E-mail，在第一次按下送出指令到 STMP 伺服器時失敗了，可是第二次再做同樣的按下送出指令到 SMTP 伺服器卻成功了。其發生的原因並非總是相同的可能是多種變化的，像是伺服器可能太忙著另一個或數個先到的服務要求；或可能是網路在一小段時間失效了，一個不預期的某項資源臨時性的沒有作用了。然而，我們也確實觀察到一個實際的現象，不必去做任何種類的狀態復原或修復動作，只要重新嘗試執行失敗的指令就能使得系統在正常的狀態再持續運作下去。

“retrying”在有時也或許不是首選的方法，有時它必須預先稍停一小段或幾秒定額的時間後才開始再度重試前次失敗的指令；有時它或許要重試多次以上才會成功；有時它或許在重試前必須把程式狀態或來龍去脈回復到與執行前一樣；也或許這些動作都必須同時運用。

然而，對於任一的案例，一位老練的開發人員總是有辦法輕易的就能應付這些繁瑣俗事。一個比較大的議題是：如何用一個簡單的風格撰寫出等冪指令(“How to write idempotent operations in a simple manner?”)。儘管如此，有些運算指令也許是難以達到此目標的，尤其是數量較大的較謹慎設計完成的運算指令。此外，這類應用程式內的工作中本身已有提供了方法來自動修復應運算指令錯誤造成的應用程式不正常狀態。

有些程式語言，例如：Smalltalk 和 Eiffel ，有實作出“retry” 指令的結構，但大部份的開發人員認為並不容易使用，因為開發人員自己必須返回前次運算失敗的不正常產出。這等同強烈暗示著所引用的運算指令在存取使用時是否具備“等冪”的特性。一

般來說這是相當困難的，因為光是呼叫函數的堆疊方法就可能有數種。此外為了代入這些返回前次運算的額外程式碼程序，可能迫使開發人員必須打破已設計完好的程式封裝結構 --- 此封裝結構正也是所謂好的應用程式設計的關鍵點之一。

2.3.1 Related Works For Catch And Retry

關於"retrying"的語法結構組成與內容，以下將以 C# 語法形式來表達，當然相同的設計精神也能應用在不同的程式語言。

如下圖 2.12 所示，在設計好的 try 區塊是具有交易性質(transactional)的。因此，在這個受保衛的區塊內被叫用的運算指令都算是交易的一部份。當執行緒進入 try 區塊時交易就開始啟動，若區塊內完整的執行過程沒有引發任何的例外狀況的話，在離開區塊時會有提交(commit)動作。若有例外被引發，那交易就立即中止，並跳到對應的 catch 指令區塊的起點，在這一點上能夠執行必要的操作，以嘗試解決該問題，並允許剛剛失敗的運算區塊再重新嘗試執行。

```
try {
    // the transaction is initiated here

    // try to send a datagram
    socket.send(dp);

    // the transaction commits when
    // leaving the try{} block
} catch (IOException ce) {
    // the transaction is aborted and
    // its effects are eliminated when
    // the handler begins executing. A
    // new transaction is initiated.

    // try to reconnect the socket
    socket = new DatagramSocket();

    // the programmer requests the re-
    // execution of the failed block
    // if some condition is verified
    if (...)
        retry();
    // Note: if retry is not executed the
    // handler transaction commits here
}
```

圖 2.12 "retrying" 語法結構

一個重點是,在應用程式中運作當中的 heap 與 stack 記憶體變數的滾返(rollback)以使得應用程式的狀態是乾淨正常的。執行恢復操作後，重新執行受保衛的區塊可以由開發人員要求指定與否。

這例子雖然簡單，但以指出了“重新嘗試”語意結構的基本設計原則。其中“retry”指令在幾個程式語言是有效用的，像是 Smalltalk、Ruby、Eiffel。這些程式語言的對 retry 關鍵字的運用就像是 goto 關鍵字一樣，遇到例外後跳入對應的處理程式碼區塊，再進行恢復、還原等運算指令，然後再判定是否重新執行。很明顯的，若故障導致的問題沒有得到解決的話，那將導致一個無限的循環。為 retry 指令指定一個次數上限是個很簡單的解法。

即使程式語言不支援“retrying”的語法結構，我們仍可以使用其它的語法組合來獲得相同的效果，例如 C#，可以合併 for/while 迴圈語法與 try-catch 語法來組合出“retrying” 語法結構。

右圖 2.13 為一範例。考慮到 SQLException 的異常原因可能是因為服務尚未啟動完成。這是合理的考量，故先稍停十秒後再重新開啟連線，除非成功連線了或重試次數達到上限。

```
SqlConnection conn = new Connection();
conn.ConnectionString = ...;
for (int i=0; i<max; i++) {
    try {
        conn.Open();
        break;
    } catch (SqlException ex) {
        Thread.Sleep(10000);
    }
}
```

圖 2.13 以 C# 組織有次數上限的 “retrying”

使用 AOP 機制，相同的結果可以在更有組織的方式和橫向來撰寫程序程式碼。如果我們考慮異常處理的程序的橫切面，其中引發異常的位置是理想的點(join point)為處理異常發生插入(weaving)程式碼(advice)。

例如在 Spring.NET 可以宣告一些規則(point cuts)去縫合例外處理的程序(advice) ：在 Spring 1.2 版提供 AfterThrows 介面，到了 Spring 2.0 支援了標籤(annotation)語法

```
RetryTemplate template =
   new RetryTemplate();

template.setRetryPolicy(
    new TimeoutRetryPolicy(30000L));

Foo foo =
    template.execute(
        new RetryCallback<Foo>() {
          public Foo doWithRetry(
            RetryContext context) {
            // business logic here
              return result;
          },
        new RecoveryCallback<Foo>() {
          Foo recover(RetryContext context)
            throws Exception {
            // recover logic here
          }
});
```

圖 2.14 Spring Batch 的 retry 範例

@ThrowAdvice。不管採用那一種語法結構，確定的是當有例外被引發時，都要有對應的 advice 來接續處理。通常情況下，每個 advice 會有一個引數參數指向所引發的例外類別(exception class)。

2.3.2 Architecture And Programming Model

例外的處理模型，以現代的程式語來說（如：C++，C#，Java）共同使用 try 區塊來"防衛(guard)"在執行期可能引發例外的運算指令。在我們所設計的程式模型裡，程式中的 try 區塊是具有交易性質的。它可以滾返(rollback)失敗的運算的不正常產出，且允許重新嘗試在使用時可以驗證，程式碼變得更乾淨、更容易維護，而開發人員也不必花費精力去恢復應用程序的不正常狀態的這件事解脫出來。事實上這是我們和傳統的異常處理模型的方法之間的主要區別。這也是整個方法的關鍵點。也可以說是此系統的交易類型，實際上已符合了 STM(Software Transactional Memory)框架，使得 try 區塊變成 atomic。

整理了以上的討論，可以歸納出重新執行 try 區塊時至少必須保證二個要件：

一、 應用程式狀態(variables, heap, stack, data structures, etc.)就是在那當下，在那第一次進入 try 區塊的當下。

二、 該 try 區塊必須是隔離的(isolated)，這意味著如果執行外部的 I/O 操作，它必須能夠撤消他們或者他們必須是等冪的(idempotent)。

此兩個要件將在後續討論。

Restoring Application State

復原應用程式狀態是至關重要的，其用以保證開發人員的設計意圖得以保持。例如，如果一個開發人員在一個 try 區塊內累加變數“total”，如果該區塊由於例外而被重新執行，在完成區塊內的運算離開後，該變數只能被累加一次。

有幾種可行的方法可以用來恢復應用程式的不正常狀態。典型的方法包括：

法一、執行緒進入受保護的區塊中，為所有與其接觸的物件建立一個副本。此副本是用來在運算時更新的。如果該區塊正常的完成離開，原先所接觸的物件再由副本來更新取代原值。如果發生了重試程序，那副本就會被丟棄，這等同保存了原先的應用程式狀態。

法二、不建立副本。直接更新其所接觸的物件，但所有的更新都會留下紀錄。該程式碼區塊若正常的離開其更新紀錄就可以丟棄。 如果發生了重試程序，那剛剛的更新紀錄就用來還原所接觸的物件，等同也還原了原先應用程式的狀態。

這些方法的實作細節都有各自棘手的部份。在物件導向的平台，第一種方法感覺更自然，儘管它比第二個方法需要更多的記憶體和嚴重依賴垃圾收集器(garbage collector)的執行。關於交易模式(transactional model)在我們的系統是不可或缺的的部份，也已形成了一個更一般化的議題：STM(software transactional memory)。STM 在不同的執行平台也都有一些實作，也有用於網際網路的離散式交易的研究，但到現在為止效果還是不命人滿意。所以決定此研究不使用 STM 工具或模組，改提供一些輔助的功能函數達到相同的效果，同時也減化開發人員在例外發生時欲還原應用程式狀態的需求。

Isolation in try blocks

正如前面所提，然而只是保留應用程式的狀態是不夠的。例如有些運算是具有嚴峻挑戰的，例如：我們不希望存入相同的金額到銀行帳戶超過一次；我們不希望寫入相同

的數據到磁碟多於一次等等。若出現這類情形則表示著這個交易運算不具隔離性(isolation)。

此外，有一些 I/O 操作是不能撤銷的，就像“發射導彈(firing a missile)”一樣。然而，仍然必須允許同時存在交易性程式碼與非交易性程式碼在同一段程式程式碼中。因此，在執行時期(runtime) 去區別出運算類別(class)是否交易性是需要的。一個簡單的方法是直接在開發時期明確指定所有的交易性物件類別都要實現 Transactional 介面，稱做 Transactional-aware class。此介面至少要提供三個 callback 函數：一個告知已進入交易程序了；一個告知交易運算正常完成可以提交(commit)了；一個告知交易要撤銷。而且這交易行為的提交與撤銷在巢狀結構下也要有效用。採用這方法顯然有時會太過複雜或麻煩且似乎也不實際。其實也只有執行外部的 I/O 運算時才須要小心的以這個方法標記。實現這類 callback 函數的複雜度各不相同，用語意來整理說明可以清楚的表示了二個方向：

一、若 I/O 指令已叫用可以先暫時緩存它，直到提交(commit)訊號到達才真的寫入。

二、在某些情況下有些 I/O 運算可以設計成等冪的運算，這就代表著它可以被重複執行多次但結果都一樣。

最後一點，這點與現行資料庫系統是一樣的，比如：有一個交易中先前寫入的數據被讀取了，而此交易尚未提交，那就會有資料衝突發生。在外部 I/O 操作上此類問題還沒有通用的解決方案，只能小心的防止它發生。大部份的情況下都是簡單的提供掛鉤(hook)來撤消或檢測衝突。在一般的開發平台上，直接參與外部 I/O 內部運算是極其有限的。在任何狀況下，若證明交易太難以達成那就選擇讓它變成非交易性的運算吧。

2.4 AspectJ

AspectJ 是由 Java 語言延伸的 AOP 語言。每一個有效的 Java 程式就是一個有效的 AspectJ 程式。AspectJ 的編譯產生出符合 Java byte-code 規範，可執行在 Java virtual machine(JVM)。

在 AspectJ 經由編譯縫合非主功能的實作稱為橫跨(cross-cutting)。AspectJ 定義了兩種橫跨：靜態橫跨(Static Crosscutting)和動態橫跨(Dynamic Crosscutting)。

動態橫跨是縫合非主要功能至主要功能的執行流程之中。大多數 AspectJ 橫跨都是動態的。動態橫跨的擴展，甚至取代了核心程序執行流程從而改變系統的行為。

靜態橫跨是修改系統靜態結構（如：類別、介面和剖面）的縫合。其本身不改變系統的執行行為。靜態橫跨最重要的功能是支援動態橫跨的實作。例如：想要新增成員變數成員函數至類別或介面中，為了確定特定類別的狀態和行為。這可被使用在動態橫跨的行為上。另外，靜態橫跨可以被使用在宣告編譯時的警告和錯誤的橫跨多個模組。

以下以案例來說明 如何使用 AspectJ：

AspectJ 範例一：Hello & World

使用 AspectJ Compiler 開發 AspectJ 程式，“Hello” 與“World” 分別由主功能類別與剖面類別各自輸出。準備主類別 Hello 程式用一般 Java 寫法即可。

```
class Hello{
   public static void main(String[] args){
      System.out.println("Hello");
   }
}
```

圖 2.15 AspectJ 範例一之 Hello

再寫剖面類別 World，寫法有些不同其目的在縫合 Advice。

```
aspect World{
      after() : call(* *.*(..)) && !within(World){
      System.out.println(" World");
   }
}
```

圖 2.16 AOP 範例一之 World

編譯與執行後畫面上出現 ==> Hello World

注意到了嗎，“Hello”與“World”並非在同一字串一起輸出，在主程式中只有印出“Hello”而已。然而透過剖面縫合程式碼，在此範例上，在呼叫完 main()函式後會接續執行印出“ World”。

在此簡單的範例可以看出基本的 AspectJ 結構，其組成分成三大部份：原主功能類別(此例為 Hello)、剖面類別(此例為 World)與非主功能類別。其中非主功類別就是在剖面類別中與 point-cut 逢合的部份，此例為“System.out”。

以下再看另一個剖面類別的案例，這個案例是接近實際應用案例的一個簡化，如圖 2.17 所示。

```
/* aspect declaration */
aspect SubjectObserverProtocol {

   /* pointcut declaration */
   pointcut stateChanges(Subject s):
      target(s) && call(void Subject.click());

   /* advice declaration */
   after(Subject s): stateChanges(s) {
      for (int i = 0; i < s.getObservers().size(); i++) {
         ((Observer)s.getObservers().elementAt(i)).update();
      }
   }

   /* introduction declarations # Subject # */
   private  Vector   Subject.observers = new Vector();
   public   Vector   Subject.getObservers()     { return observers; }
}
```

圖 2.17 Aspect 類別範例

此案例中定義了一個 point-cut 名為“stateChanges” ，它會將映射到主功能類別程式碼中類別名稱是“Subject”，成員函式名為“click()”的位置。也定義一個 advice，當跑完 stageChanges 關節後，也就是“Subject.click()”跑完後，對原主功能類別“Subject”內部屬性進行更新。在 introduction 段落裡，甚至也能為主功能類別添加新的成員變數與成員函式“observers”和“getObservers()”。

AspectJ 定義的 Joint Points 大概可以涵概了 class 所有的節點，可以分成四類：object creation、method invocation、field access、error handling。在 Advice 執行時間點可以是 before、after 或 around。在圖表 2.18 明列。

Join Points in AspectJ:

AspectJ' s join point model differentiates four kinds of actions and identifies the following join points associated with them:

- object creation: constructor calls, constructor executions, and initialization
- method invocation: method calls and method executions
- field access: field assignment and field reference
- error handling: error handler executions

圖 2.18 Join Points in AspectJ

也有人研究用 UML 來表達剖面，設計一套 Aspect-Oriented Design Notation[13], 可在設計階段能視覺化呈現 AspectJ 剖面部份。圖 2-19 是一個範例。

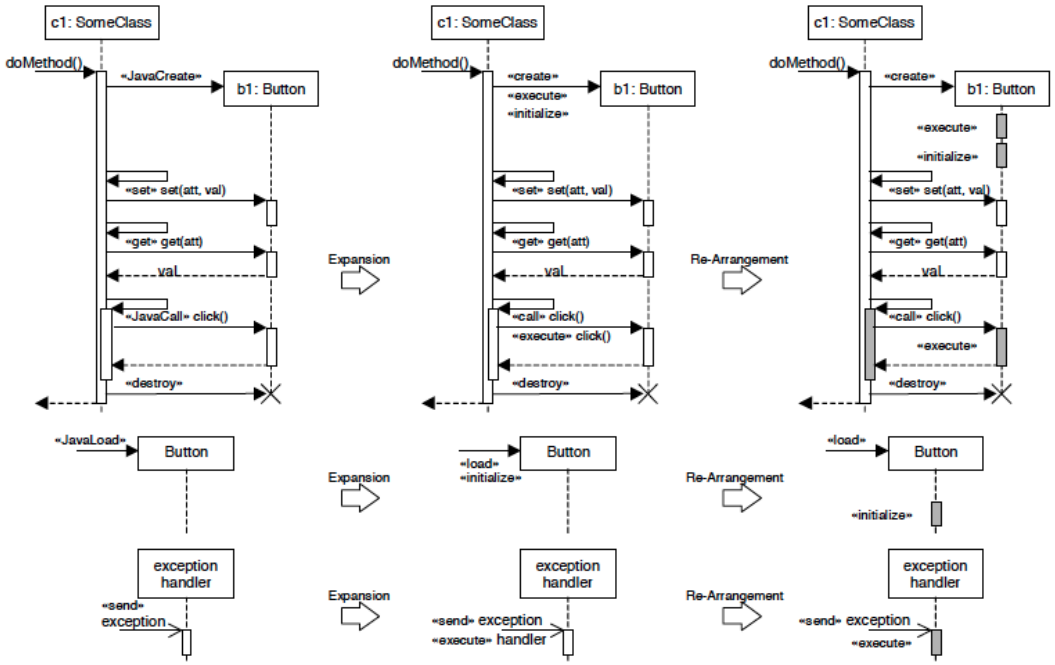


圖 2.19 AspectJ Notation 轉換說明

由圖左標準的 UML interaction 圖形轉變到圖右的 AspectJ Design Notation，圖中是轉變的中繼圖形。由圖左中的«JavaCreate» stereotype 由一個轉化成三個«create»、«execute»和«initialize»；«JavaCall» stereotype 由一個轉化成二個«call»和«execute»；«JavaLoad» stereotype 也是由一個轉化成二個 «load» 和«initialize»。這些 stereotype 都能再有三重 advice，before、after、around。我們從外部角度來看此設計，可以發現此設計等同擴張了系統的對外接觸面。AspectJ 的擴張能力已強大到甚至可以改變主功能流程的行為，從而可以使得 Aspect Class 甚至由輔助變成主要，這個強大若用的好是好，若用不好的話麻煩就大了。

2.5 AspectF

在研究了數種 AOP 的實作方法我們認為 AspectF[9] 是最好的解決方案。現在已知只在 C#/.NET Framework 有實作。其它的方案以 AspectJ 為領頭為學習對象。AspectJ 的功

能強大,對 AOP 的實現也最為完整,但我們實際使用的評價是太複雜、不易使用。AspectJ 功能最強大完成度也最高但太複雜易用性低，而 AspectF 功能較精簡但好用。在技術上以 lambda expression 為核心再進行應用。這個方法超乎想像的簡單，卻也能讓程式碼撰寫更乾淨清晰、可維護性也更高。

2.5.1 AOP 的一般案例

剖面(Aspect)是在撰寫程式程式碼的過程中，出現在不同的運算程序中的局部相同區段中有著相同的功能片段，例如它可能是一種處理異常的方法，也可以是呼叫方法時的紀錄(logging)意圖，或者是呼叫函式的執行花費時間記錄(time execution)，這些部分很可能出現在程式的不同局部地方且有一些是重複使用(reuse)多次的。如果在編碼的過程中沒有使用任何的 AOP 框架的話，那麼將重複的撰寫這些相同的程式碼，這將使得你的程式碼難以維護。例如你的業務邏輯層需要進行日誌記錄，錯誤處理，執行時間需要記錄。

以下舉一個一般的情境案例，如圖 2.20 所示。此案例中撰寫了一些實際的程式碼，這個程式片段實現了客戶資料的插入操作，同時程式中還有日誌記錄、異常處理和時間記錄的操作，由於這種多機制多設計意圖的程式碼混合，在閱讀的邏輯上就顯得混亂可讀性也變差。但是設想一下，如果要在業務邏輯中添加驗證或其它剖面的時候，那麼隨著程式碼的增加業務邏輯將會變得更亂，真是要多亂有多亂，也就是變成了糾纏程式碼(tangled code)狀態。而且當增加業務邏輯規則時，常需要不斷的「複製、貼上」程式碼,再微調每個業務邏輯層的部份。例如:你需要在業務邏輯中增加一個 UpdateCustomer 的成員函式，你就必須再「複製、貼上」程式碼一遍再微調，這真是很苦悶的一件事。

```
public bool InsertCustomer(string firstName, string lastName, int age)
{
   Logger.Writer.WriteLine("Inserting customer data...");
   DateTime start = DateTime.Now;
   try
   {
      CustomerData data = new CustomerData();
      bool result = data.Insert(firstName, lastName, age, attributes);
      if (result == true)
      {
         Logger.Writer.Write("Successfully inserted customer data in "
            + (DateTime.Now-start).TotalSeconds + " seconds");
      }
      return result;
   }
   catch (Exception x)
   {
      Debug.WriteLine(x.StackTrace);
      Logger.Writer.WriteLine(x.StackTrace);
      return false;
   }
}
```

圖 2.20 AOP 的一般案例

再想像一下，對於業務邏輯變更來說，如果我們的項目需要大範圍的修改例外處理的方式，那你就必須把所有業務層的方法一個一個的進行修改。然後如果新的需求又來了，要修改時間統計的方式...這過程真是苦不堪言啊。

剖面導向程式設計解決了上述問題，如圖 2.21 所示，其功能邏輯與圖 2.20 是一樣的，看起來是不是很簡潔呢？

在 AOP 中，例如把 logging ，time execution，validation 這些剖面從業務程流程式碼中進行分離。如上例所示，你可以通過 Attribute（在 java 稱為 annotation）解決，這樣的程式碼乾淨又清析，這裡每一個 Attribute 代表一個剖面。例如：通過增加 Logging 剖面你可以進行日誌記錄。總之，無論你用了什麼 AOP 框架，它都保證了剖面在運行時被編入了你的程式碼中。

```
[Log]
[TimeExecution]
public void InsertCustomer(string firstName, string lastName, int age)
{
    CustomerData data = new CustomerData();
    data.Insert(firstName, lastName, age, attributes);
}
```

圖 2.2 1 AOP 的一般案例 2

2.5.2 一個簡單的 AOP 框架

使用 AspectF 來達成與上一節所提需求有相同效果，如圖 2.22 所示。在應用語法上 AspectF 不是使用 Attribute 或是操作反射(reflection)來實現，只是簡單的對目標指令區塊指定剖面功能就行了。使用這種方式的程序具備了很高的重用性和可維護性，而最重要的是它很輕量級，還能使得縫合(weaving)變得極有彈性，極適合多變的業務規則變化。

```
public void InsertCustomerTheEasyWay(string firstName, string lastName, int age)
{
   AspectF.Define
      .Log(Logger, "Inserting customer the easy way")
      .HowLong(Logger, "Starting customer insert", "Inserted customer in {1} sec.")
      .Do(() =>
      {
         CustomerData data = new CustomerData();
         data.Insert(firstName, lastName, age);
      });
}
```

圖 2.2 2 AspectF 一般案例

AspectF 相當的輕量，在結構上只有一個類別(class)名稱也就是“AspectF”。而它的剖面可以簡單的再自訂新增以應付多變的需求。

如上述案例所示的程式碼，AspectF 語法不太傳統，用簡化的 EBNF 表示法來簡單說明基礎語法：

```
<AspectF-Syntax> ::= "AspectF.Define"
                        {"."<Aspect-Function>}
                        ".Do(()=>{"
                          <Logic-Function-Block>
                        "});"
```

圖 2.2 3 AspectF 語法

將 AspectF 語法與範例比較就能很清楚其結構。AspectF 的語法結構分成三個部份。首先是"Aspect.Define"，表示即將包裹(wrapping)其下的一塊業務邏輯程式碼區塊，並將為它添增一些剖面功能函式。第二部份是"Do()"函式，把商業邏輯包裹住，不必有其它的修改。要注意的是別忘了加上大括號"{"與"}"，因為 Do()函數的參數型別其實是個 lambda expression。在後面的設計原理再詳細介紹，此段重點放在如何使用與為何好用、易用。第三部份就是依需求在"Aspect.Define"與"Do()"中間加入<Aspect Function>，圖 2.22 的例子中有 Log 與 HowLong 兩個。

2.5.3 與一般 AOP 方案比較

通過 AspectF 我們就可以在業務邏輯的外部直接指定剖面功能，同時使業務流程主功能邏輯程式碼與非主功能機制程式碼分離但又鄰近在旁。可以輕易理解完整的業流程同時又不會交互糾纏。。

讓我們看一下它與通常的 AOP 解決方案在特徵上有何不同：

(1) 不在類別的成員函數的外面定義剖面，而是在的內部直接定義。

(2) 以函數實現剖面，取代以類別實作剖面的方式。

再以 AspectF 的特性來看，它有以下優勢：

(1) 使得業務流程的剖面更清晰。

(2) 可以不用過度考慮性能的損失，因為它只是一個輕量級的類別。在內部運作原理其實是以 lambda expression 為核心。

(3) 剖面功能函數可以傳遞參數。有些 AOP 方案是不允許這麼做的。

(4) 甚至不能稱為一個框架，因為它只是一個叫做 AspectF 的類別而已。

(5) 可以在流程程式碼的任何位置設置剖面也能用於迴圈內與巢狀結構，使用彈性極大。

第 3 章

系統設計與架構

在建置大規模分散式的網際網路應用服務，例如：搜尋引擎、電子郵件服務、社交網路應用等等，開發人員必需花費不少的精力處理資料性錯誤(Data errors)與可用性錯誤(Availability errors)，目的之一是讓系統不因一些例外故障而停止運作或讓使用者感觀上的認為“又當機了”。而這些例外故障中有一部份都是暫態的(transient)，非常短暫且無法預期，像是跳針一樣，完全不做任何處理在幾秒後重新執行出錯的指令卻又成功了且不影響系統的運作狀態。本系統將以補獲與重試(catch-and-retry)機制來處理這些例外故障，在技術上採用我們設計的 AspectW，一個極輕量的 AOP 實現方案，可以簡易地又不衝擊主流程狀況下加入補獲與重試機制。

在這章節將會對系統設計原理進行介紹，包含簡介系統設計架構、系統設計方法及如何使用 AOP 特點來達成本系統的設計理念。

3.1 設計理念

3.1.1 緣由

網際網路近年來的發展已是超越一般化的遍及了，從早期主機到工作站，數年前開始普及到家庭個人電腦，幾年前又盛行到個人行動智慧裝置，手機、平板等。個人使用一個以上裝置同時上網已是常態。在主機伺服器方面，在 Google 引領下開始拋棄超大型主機為主的架構轉向叢集架構為主。幾年前開始雲端運算，主機伺服器虛擬化形成主流。這些硬體部署演進與軟體應用演進是互相影響著。不管是採用或混用那一類雲端技

術 IaaS、PaaS 或 SaaS 都只會讓系統部署更分散且是動態的分散。我們可以用一個簡化的模型來描述此狀況，就先稱作“雲中雲網路”模型吧。

也不過在數年前，想要描述網路的複雜度只需一朵雲就夠了。而現在，尤其是網路、主機都虛擬化後用一朵雲來代表網路的複雜度是不夠的，而是雲中有雲，網路中有網路，實體網路中有虛擬網路。

如圖 3.1 所描繪，我們可以想像一個現代的網際網路應用系統，它有自己的商業邏輯，用“Biz Srv”表示；可也引用了第三方的元件服務，用“Widget Srv”表示，比如 Google Map 等等；在認證時又採用另一個第三方認證服務，用“Authz”表示，比如 Facebook 等。現代的社群網站、照片分享等應用服務大多符合此模型。

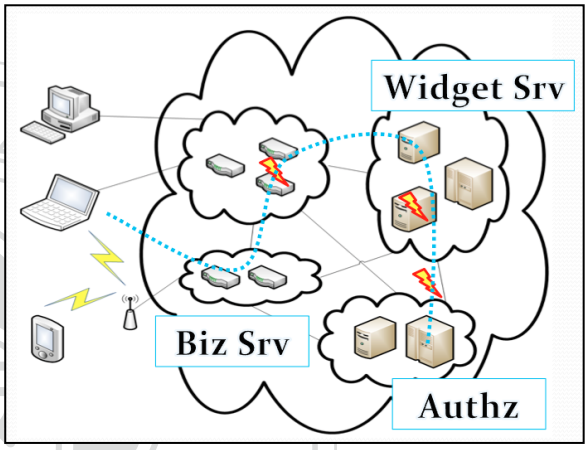


圖 3.1 雲中樹網路模型圖

在如此複雜的網路中，訊號的實際傳遞路線可能有上千公里半個地球之遠是家常便飯。跑了上千公里又通過了無數個裝置與設備，訊號偶有跳針也是可以理解。這些錯誤發生的原因可能是硬體真的故障了，可能是軟體有缺陷，可能是性能到達瓶頸遲頓了一下，可能是雜訊讓訊號變化。這些網際網路上的故障原因來源是多樣也多變的難以捉摸清楚，但我們也發現了一些現象，有些故障只需重新執行就會成功了。第一次執行失敗，第二次或第三次執行同樣的指令卻成功了，這樣的經驗偶而在瀏覽網站時發生，本來是 404/504 重新刷新卻成功了；偶而在寄出電子郵件時發生，第一次交寄失敗第二次卻成功；偶而在 IM(Instant messaging)也發生剛送出的訊息系統回覆傳送失敗，重新輸入同樣的訊息再送出對方卻收到二次同樣的訊息，明顯的內有重試機制的設計。

了解這些存在的現象，在系統的設計上是否也能成為常規程序的一部份。讓系統對這類暫態性的例外故障可以自動的應對。

3.1.2 理念

補獲與重試機制在大規模分散式系統已是普遍的需求，在被大量研究過後也認為是可以一般化的，套入 AOP 來實現是普遍認為是好的解決方案。比較幾種有支援補獲與重試的技術並不是那麼另人滿意，我們認為可以更好一些。Ruby、Spring.NET、AspectJ 各提供了不同類型的解法。Ruby 直接提供 retry 關鍵字與語法但不符 AOP 精神，程式碼容易變成糾纏狀態。Spring.NET 透過以 template class 並套用 strategy pattern 來處理，優勢是物件化，各物件各執其所，缺點是分得太細難以理解整體流程與設計意圖。AspectJ 是由 AOP 提暢者所在的 Xerox Palo Alto 實驗室設計出來的作品，也是最接近 AOP 原創精神的成品。可就我們的應用目標，我們發現由 Omar Al Zabir 提出的 AspectF 的 AOP 實現方法是更適合我們的。最主要的原因是“剛剛好的好用”，也正是近期重視的使用者體驗(UX, user experience)。這裡的使用者非終端的產品使用者，而是「開發人員」。對於系統的設計階段來說 AspectJ 是比較好的解法，這時的使用者是設計師；到了系統開發階段也就是撰寫程式碼的階段，使用者變成開發人員，我們認為 AspectF 是比較好的解法方案。關鍵不在功能強大，在剛好夠用也易用。

3.1.3 功能規劃

在規劃一套補獲與重試機制工具該有的功能時，在此前題，我們認為把使用者體驗(UX)做為第一考量是重要的，當然該有的功能是必需的，就是在分散式系統下對於通訊指令執行失敗的重試與復原。

以應用情境為基準來決定設計考量，首先先來重溫一下目的。此設計是應用在大規模分散式系統，在單機上並不適用。另一個關鍵點在等冪(idempotent)特性，同樣的指令不管執行幾次結果都是相同的。愛因斯坦曾定義過何謂精神錯亂：同樣的事情做了一遍又一遍卻期待有不同的結果。這狀況在單機上是合理的，除非該指令設計本身是非決定性的(non-deterministic)，如：亂數產生器。不過在網際網路上就不一樣了，同樣的指令送出後回傳的結果卻有一定的小機率會不同。還好的是，這些不同結果基本上都是錯誤訊息,不是似是而非的含糊狀態。也發現了一些在網際網路應用的一些現象，像是瀏覽網站時有時會出現 404、504 的失敗訊息，再刷新同樣的網址卻又正常了。寄出電子郵件時偶而會無效但再下達寄出指令二、三次卻又成功了。使用 IM 跟朋友聊天時，送出的談話訊息回覆說失敗，再送出同樣的談話訊息，對方卻回應為何同樣的談話訊息要重覆送多次。

把這些觀察到的現象整理出一般化的應對流程模式，也就是補獲與重試機制的流程模式：

Step 1：補獲失敗訊息。

Step 2：稍停一小段額定週期，一般是幾秒即可。

Step 3：還原應用程式的不正常狀態。

Step 4：再重新執行剛剛失敗的指令。

以上也是功能上要能滿足的部份，非功能部份希望做到二點：

一、 流程程式碼不要因此而散開，以避免整體流程的設計意圖難以識別。

二、 加入補獲與重試機制的指令要儘可能簡潔，鷹架碼越少越好。

3.1.4 為何決定打造 AspectW

研究 AspectF 與其它 AOP 的實現方案比較，我們發現它使用一些近幾年才出現的程式語言技術，使得在使用上有二個特有的優勢：

一、簡單的宣告就能開始縫合剖面功能(aspect function)，因它導入了“fluent interface”[10]的設計。

二、剖面功能可簡單的擴充，因它透過“partial class”來擴展。

我們也用幾個不同角度來評估，從使用方法、從應用情境、從邏輯推理等。

從使用方法來看

以使用的方式來比較，整理起來有二類：template class 與 tag。

舉例來說：Spring（請參考圖 2.12、圖 2.13、圖 2.14）、AspectJ 等在比較早的版本的都是繼承或引用事先做好的樣板類別，再依各案例需要自訂加入剖面功能。在較新的版本也都可以用貼標籤的方式來使用。這種貼標籤方式在 C#的技術名稱為 Attribute；Java 的技術名稱叫 Annotation。這些標籤可以參數化。

```
@Component
public class RetryOnFailureTest
{
   @RetryOnFailure(attempts = 3, delay = 2000)
   public void testRetry()
   {
      System.out.println("Entered .....");
      throw new RuntimeException("Forcing an exception");
   }
}
```

圖 3.2 AspectJ 重試標籤簡化範例

圖 3.2 是用 AspectJ 實作了一個重試標籤的簡化範例。此圖例中只秀出如何使用。在圖例 3.2 中，標籤“@RetryOnFailure(attempts = 3,delay = 2000)”代表當例外發生時先稍停 2 秒再重試，最多重試 3 次。這個標籤是自訂的。若要實作要完成二件主要工作，一、宣告一個介面“@interface RetryOnFailure”，二、再實作一個 Aspect Class “RetryOnFailureAspect”。詳細實作可參考這兩篇參考文章[14][15]。

實作過程要引入 AspectJ package，只要遵循 AspectJ 定義好的開發規則與順序還滿容易實作出來的。若使用從 AspectF 改良過的 AspectW 同樣的邏輯程式碼大概如圖 3.3 所示。

```
public class RetryOnFailureTest
{
   public void testRetry()
   {
      AsepctW.Define
         .Retry(2000, 3)
         .Do(()=>
         {
            System.out.println("Entered .....");
            throw new RuntimeException("Forcing an exception");
            });
         }
}
```

圖 3.3 AspectW 重試範例

AspectW 的語法結構延用 AspectF 的設計，並改良剖面功能(aspect function)的部份，讓它更通用於各種狀況。AspectW 的語法可參考圖 3.10 的說明。先用 “AspectW.Define” 來表示將包裹一塊程式碼區塊，接下來串接剖面功能指令，此例是 “Retry”，第三段的 Do()函式負責將目標程式碼區塊包裹起來。

比較 AspectW 的用法，它不屬於 template class 也不是 tag，而是一串函式鏈。這個鏈長度是動態的，函式串接起來修飾共同的目標程式碼區塊 --- 由 Do()函式負責包裹。若用 GoF design pattern 來分析的話，最接近的應該是 Decorator pattern。Do()函式的參數其實是個 lambda expression。

從使用面來說，AspectW 似乎比較麻煩一些些。我們再仔細看一下程式碼。圖 3-2 的標籤是貼在函式宣告位置，也就是標籤跟函式是綁在一起的在定義時期就固定了。而傳統的縫合過程必須另外撰寫 Aspect Class 負責縫合，在主流程是看不到是否被逢合，優點是物件化後重複可用性可以提高的，但它的優點也是缺點，整體流程的可讀性因而降低。再比較圖 3.3，AspectW 的語法不像貼標籤與函式宣告綁在一起，而是直接在主流程上宣示並縫合，不再另外建立 Aspect Class。直接把主功能邏輯與非主功能機制程式碼組合起來，優點是整體流程是清楚的。也有人會認為這樣重複可用性不就無法提高。這論點在系統設計階段決定各子單元責任時是合理的，不過到了實作階段撰寫程式碼時，再過度拆解出小物件反到弄巧成拙讓整體流程的可讀性降低。

從應用情境案例來看

容易又簡短的案例是不值得提的，因為太簡單根本比不出價值的差異，太長或細節太多又會冗餘。花了一些時間找一個不會太短又細節不會冗餘的例子。以下舉出一個購物車系統在實務的案例，整個購物車系統在流程結構一般至少拆成四個主要階段步驟：

〔挑選商品〕=>〔選擇配送方式〕=>〔選擇付款方式〕=>〔結帳〕

在設計流程結構階段時，導入物件化分段設計是好的。到了更細部的撰寫程式碼階段時，比如：在〔結帳〕階段步驟中的出結帳單這一段落，要有一段不算短的處理程序。如下描述購物車的結帳單可能的處理程序共有七道，如圖 3-4 所示。

1.認證檢查，是否有效的登入認證。
2.自購物車取出選購貨品數量、單價等資訊。
3.去庫存再度確認貨品數量並鎖定。
4.各種折扣活動或贈品活動或紅利抵扣計算。
5.新增紅利累計預算。
6.總應付金額、總折舊金額計算。
7.依計算結果出結帳單。

圖 3.4 購物車結帳單計算程序

為簡化過程，假設這七道順序全部都寫成子函式。也就是說，出結帳單業務流程由這七道順序組成，這是主功能邏輯的部份，可實際上還會有其它非主要功能機制的部份，比如 logging，在每道子順序都要加上在出錯時追查或偵錯使用。比如第 1 道要看

是否登入認證仍有效，驗證輸入參數是否合法。第 2 道、第 3 道要加入 retry 以應對在讀取庫存時網路的暫態性失效問題。

使用 AspectJ 加入這些非主功能機制的話，在主流程的程式碼是完全看不到的，因為分散到 Aspect Class 了。只是單獨閱讀 Aspect Class 程式碼是看不出任何的程序關聯，因為都是關節點(concern)。若改用 annotation 貼標籤也是在 class 定義端，在主流程引用時還是看不到的。

使用 AspectJ 的好處是，主流程的程式碼是完全不須任何更動的。因為 Aspect Class 是另外再建立的。AspectJ 支援萬用字元與分群設定，在初期可少寫一些縫合設定。然而，整體流程其實包函了主功能邏輯與非主功能機制兩部份，在上線初期有降偶合的好處。但只要上線維護時間一久，溫水煮青蛙效應，使用者業務需求不斷調整，主功能邏輯程式碼也會配合不斷微加微減，非主功能機制有時也要更動，而這整體流程的分離將造成設計意圖隨著時間可讀性與理解度不斷降低。初期以萬用字元或分群的設定，容易造成加了不該加的或少了不該少的非主功能機制，又因分散不易查覺。

```
AspectF.Define.Log().AuthzChecking()
        .Do(()=>{ 1.認證程序，是否有效的登入認證 });
AspectF.Define.Log().Retry()
        .Do(()=>{ 2.自購物車取出選購貨品數量、單價等資訊 });
AspectF.Define.Log().Retry()
        .Do(()=>{ 3.去庫存再度確認貨品數量並鎖定 });
AspectF.Define.Log()
        .Do(()=>{ 4.各種折扣活動或贈品活動或紅利抵扣計算 });
AspectF.Define.Log()
        .Do(()=>{ 5.新增紅利累計預算 });
AspectF.Define.Log()
        .Do(()=>{ 6.總應付金額、總折舊金額計算 });
AspectF.Define.Log()
        .Do(()=>{ 7.依計算結果出結帳單 });
```

圖 3.5 購物車結帳單計算程序 with AspectF

使用 AspectF 的話，大概變成如圖 3.5 所示。主功能邏輯與非主功能機制是放在鄰近的，在縫合區可隨意增減剖面功能，也就是非主功能機制，同時也能明確看出主功能邏輯的順序。就算維護時間拉長整體流程的設計意圖理解度與可讀性不會降低。

在具規模的軟體系統，負責主功能邏輯的開發人員與負責非主功能機制也就 Aspect Class 的開發人員可能是不同人。那溝通成本肯定要增加了，否則根本無法知道剖面功能是否正確的放上。之後更換開發人員維護後將更麻煩。又若在程序中的一小局部的剖面功能要調整，這時還找得到是那個剖面嗎。縫合設定支援萬用字元代表可以有多個標的被設成目標，那麼在調整這類組態設定時會不會不小心漏了不該漏的，或無意加了不該加的。這種過度細緻的物件化反而帶來更多長期維護的問題。

採用 AspectF/AspectW 的方法就不會有過度細化的維護一致性問題存在。使用 AspectF/AspectW 在整體流程的順序上，可以為每道順序各別標上各道需要的剖面功能。主功能與剖面功能在縫合區可以一眼看出。主功能順序依然清楚。剖面功能的程式碼也不會跟主功能程式碼混合造成程式碼互面糾纏的狀態。一般的反對想法是重複使用性不就降低了嗎？若已化成剖面功能的部份，其重複使用性並未下降。重複使用性下降的只有縫合區。在實務經驗上，使用者的業務需求經常會微調，主功能邏輯要配合調整，有時剖面功能要改變順序或參數，而剖面功能本身極少變化可以共用。也就是主功能與縫合區在實務上以 1 對 1 的關系存在最好。圖 3.6 主功能區與縫合區聯聯比較圖，說明結構上的差異。

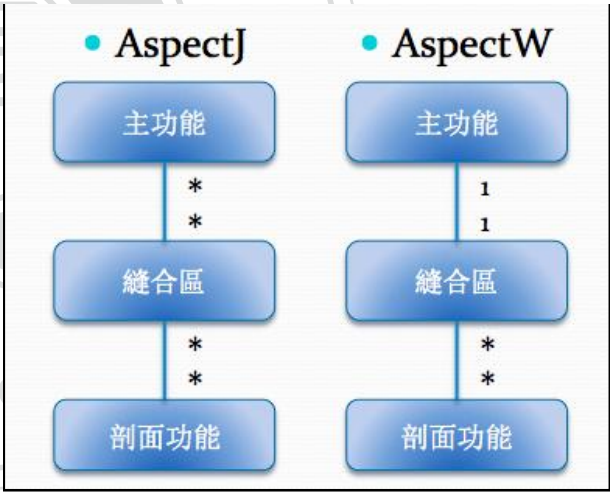


圖 3.6 主功能區與縫合區關聯比較

從邏輯推理來看

整理了研究心得，業務處理的整體流程除了主功能邏輯還有非主功能機制與一些鷹架碼，其中鷹架碼是相依存在且量不大可以忽略。可以用數學式字寫下：

整體流程 = 主功能 + 非主功能 ----------（公式 1）

有些非主功能機制在某些執行平台或開發平台有直接支援，只須引用平台指定的使用規則即可。有些非主能機制可以元件化或模組化包裝成剖面功能，在傳統上一般被稱做小工具(miscellaneous)，但總是被視為可有可無，若小工具設計的好那就大受歡迎，若設計的不好也沒人在乎的樣子，其實是有人在意的只是不明顯，這部份讓人在意的"小雜事"通常就由當事的開發人員認份的吸收了。

AOP 的組成也可以化成三個部份：主功能、剖面功能、縫合區，用數學描述如下：

AOP ∈ { 主功能，剖面功能，縫合區 } ---------- (公式 2)

然而，在整體流程中的非主功能若是有重複使用性的，最典型的就是 log，就可以轉化成元件或共用函式來使用，也就是剖面功能。寫成數學式子如下：

非主功能 → 剖面功能 ----------（公式 3）

要注意一點，非主功能並非就是剖面功能，而是轉化成剖面功能。剖面功能通常不能單獨存在，需要縫合時補入一些參數才能運作，就算是 log 也要參數指明要記錄那個點。當然有些資訊可以由系統環境取得，但畢竟不是自己本身產出。所以再重寫一下數學式子如下：

非主功能 = 縫合區 + 剖面功能 ----------（公式 4）

把公式 1 與公式 4 聯結起來就變成了：

整體流程 = 主功能 + 非主功能

= 主功能 + 縫合區 + 剖面功能

→ 主功能 + 縫合區 ----------(公式 5)

整體流程從主功能與非主功能的組成轉化成主功能與縫合區的組成。剖面功能就像一般共用的元件放到後端的共用元件庫。換句話說，開發人員只需像使用共用元件一樣引用隱身在後端的剖面功能即可，開發人員只需撰寫主功能邏輯與縫合區程式碼。

下面表 3.1 比較 AspectJ 與 AspectF，其名稱相像但內函完全不同。這表格是這幾個月來研究 AspectJ 與 AspectF 的比較心得。在主要特性方面若非得用最簡單的區分的話，AspectJ 是物件導向的設計，相對的 AspectF 則是程序導向，只是相對的，因為 AspectF 本身可是物件。縫合面的話，AspectJ 的確強大，這在前面〈章節 2-4 AspectJ〉有介紹。功能強大的縫合導致組織縫合的工作最終也變得相對複雜。反而 AspectF 的縫合功能簡單，但也相對的容易使用多了。在開始使用 AspectJ 前必須先引入其函式庫。AspectF 也是要引入函式庫，不過兩者相比 AspectF 只有一個類別。在使用 AspectJ 前須先學習一些前置知識才能開始使用。AspectF 的學習曲線相對的低多了。在縫合部份 AspectJ 要另外建立 Aspect Class 組織主功能類別與非主功能類別的縫合關係，可以套用萬用字元機制一次與多個目標建立縫合關係。一個主功能類別也能接受多個剖面類別的縫合。也就是 AspectJ 的主功能類別與負責縫合的剖面類別是多對多關係。而 AspectF 的主功能區與縫合區只有簡單的一對一關係，不過使用上很有彈性可以很流利的增減剖面功能。另一個 AspectF 最跳躍的特性是它的主功能區與縫合區是合成一體的。在擴充的比較 AspectJ 的手續也相對的比 AspectF 多，而 AspectF 只是寫個函式即可。在長期維護上，以經驗來分析，因使用者需求會不斷調整，微調參數加入新特徵等等，就像是溫水煮青蛙，長期下來可以推測因為整體流程的主功能與非主功能分離的關係，維護難度將隨時間上昇。而 AspectF 完全相反，主功能與非主功能一體，維護難度可以持平。主要技術面 AspectJ 使用已研究多年的 Spring，內有多種機制有 dynamic-proxy、

byte-code weaving、load time weaving 等等複雜的技術。而另人訝異的是 AspectF 的主要原理是 lambda expression。以下整理成比較表格。

表 3.1 AspectJ 與 AspectF 比較表

	AspectJ	AspectF
主要特性	物件導向	程序導向
應用範圍	可普遍應用各領域，尤其是 plug-in。	較適合商務領域，即靠近終端使用者的商務流程。
使用前準備	引入AspectJ函式庫	一個AspectF類別
縫合功能比較	縫合功能強大。 支援萬用字元可分群設定。 主功能與縫合區可有多對多的縫合關係。	主功能與縫合區只能有一對一的縫合關係。
縫合方法比較	1) aspect class 組織縫合參數。物件化，但也造成流程分散。 2) annotation 縫合在定義時期就綁定。	weaving 與流程縫合成一體可隨需要流利的(fluent)增減。
擴充上比較	advice class realize annotation	aspect function
長期維護	因為分散，長期需求變動後可讀性反而降低，並導致越來越難維護。	因為集中，長期可讀性可維持，維護難度持平。
主要技術	Spring dynamic-proxy byte-code weaving	lambda expression
學習曲線	相對較高	相對低很多

3.2 設計原理

這一章節將討論 AspectW 設計的原理。AspectW 延用了 AspectF 原有的核心碼，一開始計劃再精鍊但仔細檢查並測試後，認為延用是最好的策略。曾嘗試過幾次核心碼的調整，事後認為沒有原本的好，還不如就保有原創也向原作者致敬。

然而在剖面功能指令部份，仔細試用一陣子後，認為是原作者為當時專案需要特製，不易一般化的應用。AspectF 本身已內附了 retry 功能，在試用好幾次後發現若使用原有設計代入補獲與重試機制會有些冗餘也有些不足。在多日考量下決定重製，延用原有核心碼而剖面功能(aspect function)部份以補獲與重試應用為主。

3.2.1 AspectW 核心原理

在開始前先從一個使用範例開始，先了解語法結構，主功能邏輯、剖面功能與縫合區組織成一體後的模樣。如圖 3-7，此例的程式語言是 C#，沒有使用 Attribute 也不用為了縫合另外建立剖面類別，可以直接把目標程式碼區塊包裹住，跟它縫合的剖面功能也能傳入參數。

```
public void InsertCustomerTheEasyWay(string firstName, string lastName, int age
   , Dictionary<string, string> attributes)
{
   AspectW.Define
      .TraceBefore(() => InsertLog("Inserting customer the easy way"))
      .TraceHowLong(() => InsertLog("Starting customer insert"),
                        (ts) => InsertLog("Inserted customer in {0} seconds", ts))
      .RetryOnce(1000)
      .Do(() =>
      {
         // business logic
         CustomerData data = new CustomerData();
         data.Insert(firstName, lastName, attributes);
      });
}
```

圖 3.7 一個 AspectW 使用範例

此例子一下子就能看出主功能邏輯只有二行，建立一筆客戶資料並新增到資料庫裡，由 Do 函式負責包裹。其外包裹了三項非主功能機制，1)在執行前先記錄一下要做什麼動作。2)記錄執行花費了多久時間。3)若執行失敗的話，稍停 1 秒再重試一次。

整理一下在使用上與一般 AOP 方案的差異：

- 取代在函式外部定義剖面，改成在函式內部直接定義。
- 取代用類別來實作剖面功能，改成只需寫個函式即可。

也再看一下優勢有那些：

1. 沒有複雜的技術，沒有 Attributes, ContextBoundObject, Post build event, IL Manipulation, Dynamic Proxy。
2. 效能幾乎沒有損失。
3. 可以為縫合的剖面功能排順序，比圖 3.7 的例子：你可以只 trace 一次而不管 retry 幾次，也可以每次 retry 都 trace，只需改變它們的順序把 retry 上移 trace 下移即可。
4. 也可以傳遞參數到剖面功能裡，這在有些 AOP 方案是沒有的。
5. 它沒有引用什麼函式庫或平台，只是一個類別。
6. 可以在程式碼的任何位置把剖面功能縫合上去。可以放在 for 迴圈裡，也支援巢狀結構，在包裹的程式碼區塊內可以再包裹程式碼區塊。

接下來展示圖 3.7 範例中的 TraceBefore 與 RetryOnce 是怎麼做出來的。圖 3.8 是剖面功能 TraceBefore 的原始碼，挑選它是因為它是最簡單的。其原始碼中有個函數 Combine 這是最重要的核心之一，它負責縫合剖面功能全部串成一個鏈(chain)，後面再對它詳細介紹。介紹一下參數 work 它代表了被包裹住的程式碼區塊。傳入的二個參數

中的 aspect 是在使用時由“AspectW.Define”所建構出來的，它是動態產生的物件每 Define 一次就建構一個實體出來。另一個參數 traceBefore，它的型別是 Action，以程式語言 C 來看就是函式指標(function pointer)， JavaScript 來看是函式物件(function object)。Action 是.NET Framework 裡較新的技術，它是個特別的類別，對它最快的理解就是個 delegate，更好的比喻是 lambda expression。在例子裡 traceBefore 參數可以看做是指向“InsertLog("Inserting customer the easy way")”函式的指標。把這些資訊組合起來，TraceBefore 的功能就是在執行被包裹的程式碼區塊的程式前，先去調用(invoke) traceBefore 函式，也就是“InsertLog("Inserting customer the easy way")”。

```
[DebuggerStepThrough]
public static AspectW TraceBefore(this AspectW aspect, Action traceBefore)
{
   return aspect.Combine((work) =>
   {
      traceBefore();
      work();
   });
}
```

圖 3.8 剖面功能 TraceBefore 原始碼

```
[DebuggerStepThrough]
public static AspectW RetryOnce(this AspectW aspect, int retryDuration)
{
   return aspect.Combine((work) =>  // around
   {
      try
      {         // before
         work();
      }         // after
      catch
      {
         Thread.Sleep(retryDuration);
         work();    // exception
      }
   });
}
```

圖 3.9 剖面功能 RetryOnce 原始碼

以剖面功能 TraceBefore 原始碼為基礎開始，再來閱讀另一個剖面功能函式 RetryOnce 原始碼應該就快速多了。請參考圖 3.9 剖面功能 RetryOnce 原始碼。

其原始碼中有 try-catch 語法結構，整體語意是這樣的：執行被包裹的程式碼區塊 work，若失敗的話稍停一段時間後，此例是 1000 豪秒(1 秒)，再重新執行一次被包裹的程式碼區塊。選取此例的因另一個原因是此例可以一眼清楚的看出在 joint point，也就是 work，的“before”、“after”、“around”還有“exception”的位置點，可以依需要把程式碼加上。

接下來可以讀取最核心的部份之一 Combine 函式，它負責縫合的工作。請看圖 3.10，閱讀原始碼可以看到 AspectW 有個成員變數 Chain，這個 Chain 存放串聯好的剖面功能。Combine 函式的功能是依順序串聯所有的剖面功能，換句話說，Combine 函式就是縫合所有欲修飾或新增特徵或加入非主功能機制到被包裹的程式碼區塊。

```
[DebuggerStepThrough]
internal AspectW Combine(Action<Action> newAspectDelegate)
{
   if (this.Chain == null)
   {
      this.Chain = newAspectDelegate;
   }
   else
   {
      Action<Action> existingChain = this.Chain;
      Action<Action> callAnother = (work) => existingChain(() => newAspectDelegate(work));
      this.Chain = callAnother;
   }
   return this;
}
```

圖 3.10 核心原始碼之一 Combine 函式原始碼

再看另一個核心 Do 函式，請看圖 3.11 另一核心 Do 函式原始碼。Do 函式負責執行經由 Combine 函式縫合好的剖面功能鏈。在細讀 Do 函式的原始碼時，先細看一下其中有個名為 Chain 的屬性，它也是核心之一，其型態是：

“Action<Action>”

當中，這個 Action<T>型別是個函數的泛型，於.Net Framework 2.0 版以後開始發展，不過在.Net Framework 3.5 版以後才成熟。Action 也可以說就是 Delegate，若用 C 語言的角度來看則是一個函式指標(function pointer)。在理解上，可以用 lambda expression 來理解，可以看作在 FL(function language)的 f(g(x))描述句。

在這個“Action<Action>”結構下 Chain 就可以不斷的串聯 Action 。而這裡的 Action 其實就是前面提到多次的剖面功能(aspect function)，也就是 Combine 函式可以縫合剖面功能的原理。理解了 Chain 結構後，再來閱讀 Do 函式就快多了。從技術面來閱讀：若 Chain 沒有任何串聯 Action 那就直接調用 work 函式；否則就依序由外向內一層層調用 Action 最後才調用 work 函式。再注意一點，此 work 其實也是一個 Action。改以使用面來閱讀：若沒有縫合剖面功能則執行被包裹的主功能程式碼，否則依序執行剖面功能最後再執行被包裹的主功能程式碼。請注意 work 函式也就是主功能程式碼。

```
/// <summary>
/// Chain of aspects to invoke
/// </summary>
internal Action<Action> Chain = null;

[DebuggerStepThrough]
public void Do(Action work)
{
   Debug.WriteLine("ON : Do");
   if (this.Chain == null)
   {
      work();
   }
   else
   {
      // will invoke action layer by layer
      this.Chain(work);
   }
}
```

圖 3.11 核心原始碼之一 Do 函式原始碼

整理一下，AspectW 最主要的核心有三個：Chain、Combine、Do，而串聯這三個核心的基本原理是 FL 的 lambda expression。其它重要部份，如“AspectW.Define”其中的 Define 有語法包裝用途，內容是建構式，其效果在語法上有宣示的意味，宣示在其下將要包裹一段程式碼區塊為其加入一個或數個剖面功能。

AspectW 語法

接下來，用 EBNF 格式重新寫下 AspectW 的使用語法，如圖 3.12 AspectW 基礎法。其實 AspectW 的語法與 AspectF 是一樣的因為核心基礎是一致的。

首先用“AspectW.Define”進行宣示，再來可以先寫下 Do 敘述先包裹好目標程式碼區塊，也就是主功能邏輯，然後再一一加入需要的剖面功能，也就是非主要功能機制。

其中<Aspect-Function>是剖面功能，實現非主功能機制，其實體是 Action。Action 是個特別的類別可以理解成“函式指標”，在 C#裡 class method 也是 Action，Delegate 也是 Action。在 Do 函式的符號“()=>”表示這是一個 Delegate，用來接連一個 lambda expression 敘述句。<Logic-Function-Block>放的就是主功能邏輯，其實體就是程式語言敘述句的集合。

```
<AspectW-Syntax> ::= “AspectW.Define”
                        {“.”<Aspect-Function>}
                        “.Do(()=>{”
                           <Logic-Function-Block>
                        “});”

<Aspect-Function> ::= Action

<Logic-Function-Block> ::= {<Language-Statement>}
```

圖 3.12 AspectW 基礎語法

AspectW 的核心碼還有另一個 Return 函式，這個函式是 Do 函式的延伸，目的是當被包裹的程式碼區塊有回傳(return)的情境時使用。在使用上把 Do 置換成 Return 並對應好回傳的泛型類別即可。在此就不再細述。

3.2.2 Catch-And-Retry 指令設計

在指令的設計上，依據前面一些文獻的探討還有開發經驗與一些測試。這此指令是應用在網際網路上大規模分散式系統。目標程式語言為 C#。在應用上須滿足幾項情境：

- catch exception

  補獲故障。這項需求在現代的程式語言多有支援。

- retry

  重試，重新執行失敗的指令。在 C#未支援要做一些加工。

- skip/ignore

  忽視例外故障。不重要，不嚴重，影響不大，可承受的故障，如：“按讚”，可能重試過幾次最終還是失敗，那就忽略掉吧。

- restore state

  要能恢復運算到半途的不正常狀態。這情境並不算少若能支援就能減少一些恢復的開發工作。

- re-schedule，重新排程。這需要平台支援且可能造成邏輯順序問題，故在此階段不實作。且指令須有等冪(idempotent)特性才能重新排程。

在語法結構上依文獻探討要有像這樣的五個段落：

Try - Catch - Retry - Fail - Finally

不過真的試著做了後，發現要十個段落才能滿足又好用：

Enter - Try - Catch - Restore - Wait - Recovery - Retry - Fail - Ignore - Leave

這十個段落一一說明如下：

- Enter，進入點。進入被包裹的程式碼區塊時執行。
- Try，即被包裹的程式碼區塊。也就是主功能邏輯程式碼區塊。在 AspectW 就是由 Do 函式負責包裹。
- Catch，補獲例外故障。
- Restore，還原狀態，當補獲故障後立即還原本地端的應用程式狀態。
- Wait，補獲故障後先稍停一小段額定的時間。
- Recovery，復原操作，與 Restore 不同的是需自製復原操作程式碼。
- Retry，重試，重新執行剛剛失敗的指令並指定重試次數以免無限循環。
- Fail，進階操作，若多次 Retry 最終還是失敗可在此決定要如何處理。
- Ignore，忽視例外故障。
- Leave，離開點，離開被包裹的程式碼區塊時執行。與關鍵字 finally 用意相同。在命名上為與 Enter 相呼應而改取。

在實作指令時又依使用頻率與情境狀況再次調整，分成五個主要指令型，再依應用細節有多載(overloading)變型，以下只說明這五個主要指令型：

一、Enter-Leave

指令宣告：OnEnterLeave(Action *enterAction*, Action *leaveAction*)

以 event 的型式來對待，在進入被包裹的程式碼時調用 enterAction 離開時調用 leaveAction。

二、Try-Catch-Wait-[Recovery]-Retry

指令宣告：Retry<TException>(int *retryDuration*, int *retryCount*)

Retry<TException>(int *retryDuration*, int *retryCount* ,
Action<TException> *recoveryAction* ,
Action<TException> *failAction*)

這是最主要的指令，因應對多種狀況有多個變型，這裡說明其中的兩個，一個基本的與一個進階的。基本款假設使用情境是當發生故障時不必特別的處置只須稍停一段額定時間後再重試就能成功。此情境應該在大部份狀況下都符合。在基本款部份參數只有兩個，其中 retryDuration 指定當故障後先稍停多少豪秒，然後再重新執行由 Do 包裹的主功能邏輯程式碼區塊（之後簡稱為：主程式碼）。retryCount 參數限制重試次數上限。進階款多了二個參數，recoveryAction 指示在重新執行前，先調用以復原狀態後再重試，而若最終還是失敗就調用 failAction 做處理。預估 failAction 應該很少會用到不過還是加入設計。

三、Restore

指令宣告：Restore(params RestoreWhenFail[] *restoreArgs*)

設計來復原本機端的狀態，此機制相當複雜在下面〈章節 3.2.3 Restore 設計原理說明〉再介紹細節。以應用程式狀態的復原來說，最好的解決方案是用 STM(software transactional memory)，不過這項技術並不成熟，故改採比較原始但可信任的機制：備份與還原。

使用上把想保護的狀態物件當作特製類別 RestoreWhenFail 的建構參數用以備份(backup)，當補獲到故障時會馬上被還原(restore)。

四、Fail

指令宣告：HandleFail(Func<Exception, HandleFailMethod> *failHandler*)

設計來處理進階的狀況，也就是當使用 Retry 指令無法滿足時才派上用場。

先說明參數型別 Func，它與 Action 都可視為 Delegate 差別在它有回傳值而 Action 沒有。以此例子“Func<Exception, HandleFailMethod>”來說，是表示函式特徵(Function Signatures)有一個輸入參數型態是 Exception 且也有回傳值型態是 HandleFailMethod 的函式。比如像下面這樣的故障處理函式：

```
HandleFailMethod FooFailHandler(Exception ex);
```

此故障處理函式可能做一些比較特別的復原動作。不管如何復原，在完成離開前必須回傳給一個訊息告知 AspectW 這個故障是要“ThrowOut”向外通告，或是“Retry”重新執行，或是“Ignore”忽略不管。圖 3.13 說明 HandleFailMethod 的資料結構程式碼。

```
/// <summary>
/// 定義補獲到的例外(故障)的處理方法
/// </summary>
public enum HandleFailMethod
{
   ThrowOut = 0, // default
   Retry,
   Ignore
}
```

圖 3.13 指定故障的應對方式

五、Ignore

指令宣告：Ignore()

Ignore<TException>()

如同其名，設計來忽略故障。也可透過泛型參數指定只忽略特定的例外故障。有時有些不是很重要的例外故障是可以忽略的。

3.2.3 Restore 設計原理說明

應用程式狀態(application state)的保留與恢復一個另外獨立的研究主題，這個研究的解決方案以 STM(software transactional memory)為主。STM 已有一些研究成果但還不成熟。本系統以 C# 程式語言為應用目標，所以以.NET Framework 平台的 STM 研究為主。試著找了幾天，有看到二個官方的研究案：SXM 與 STM.NET，不過都沒正式釋出。其中 STM.NET 在.NetFx40 beta 有出試用版，但沒有後續的釋出。現在.NetFx4.5 已釋出但沒有 STM.NET 功能，查了一些原因原來是因為有人找到了一些嚴重的漏洞所以就被取消釋出了。也有找到一些 C# open source 的 STM 但不打算採用，因為品質難以保證。親自去研究測試應該要花上不少時間，在整體進程考量下，決定尋找別的可行方案。

以保存與還原應用程式狀態的方法有兩類：

一、建立副本以故障時復原，也就是備份與還原機制。

二、為所有異動留下紀錄，要復原時再一一撤消(undo)。

選取第一種備份與還原機制因為有效又簡單，在實作上又包括了兩個主要的項目：

一、備份：深層複製(deep clone)一份副本。

二、還原：將副本指定(assign)回來。

這兩件工作看似單純實作起來立馬就發現問題。備份的處理就是複製一份，在想法上是這麼的直覺，可技術上複製又有分所謂淺層複製與深層複製，在高度物件化的程式語言，都只做到淺層複製。在尋找多日才找到方法，可以使用序列化(serialization)技術來處理。序列化可以理解成就是存檔與讀檔。一般存檔都是存到磁碟機，透過序列化可以導向記憶體完整地儲存一份副本，在需要還原時再反序列化回來。

還原的部份直覺上也是認為是簡單的，但一開始實作時馬上就遇到一個問題，因為是必需透過 AspectW 來復原所以必需取得狀態物件的指標才行,這問題就來了因為 C# 這

類的較高階的程式語言基於安全性考量是不提供指標的。在尋找了數天才找到一個解法，一個類指標類別，原理是間接透過委派(delegate)也就是函式指標，再搭配 lambda expression 技巧就可以仿製指標行為。

參考圖 3.14,REF<T>類指標類別，它是一個泛型類別，透過這個類別可模擬指標的特性，並非代理(proxy)而是相似指標(pointer)的特性。以下面一個簡單使用範例來說明，請參考圖 3.15，宣告一個整數變數“a”並給予初始值 33 。再透過 REF<T>建構出類指標物件，取名為“ra”。到此時，並未對“ra”設值但可以看到其值與“a”一樣都是 33。接下來改對“ra”指定新值為 22 ，但“a”不更動。此時再查看其值可以發現它們的值一起變成了 22 。雖然在使用上並不直覺，不過這是現階段找到的解法中比較容易又能一般化的解了。

```
internal sealed class REF<T>
{
   private Func<T> getter;
   private Action<T> setter;

   public REF(Func<T> getter
              ,Action<T> setter)
   {
      this.setter = setter;
      this.getter = getter;
   }
   public T Value
   {
      get { return getter(); }
      set { setter(value); }
   }
}
```

圖 3.14 REF<T>類指標類別

真正應用到 AspectW 內部時並未直接使用 REF<T>，因希望使用上可以有更直覺的感受故又調整了設計，以同樣的原理設計了 RestoreWhenFail 類別，當然因應我們的應用目的再做了相應調整。

```
int a = 33;
REF<int> ra = new REF<int>(() => a, (v) => a = v); // 模擬ref
Console.WriteLine(String.Format("a = {0}", a)); // 33
Console.WriteLine(String.Format("ra = {0}", ra.Value)); // 33

ra.Value = 22;
Console.WriteLine(String.Format("a = {0}", a)); // 22
Console.WriteLine(String.Format("ra = {0}", ra.Value)); // 22
```

圖 3.15 REF<T>使用範例

3.2.4 常見例外故障情境與指令下法

在 AspectW 的指令設計過程中，最早的想法是直接使用 AspectF 已設計好的指令，然而試用了幾次在文獻探討中的案例就發現功能雖有達到但不是很容易使用，仍然要有特定的客製化工作，沒有一個文獻探論中的案例是可以直接套用 AspectF 指令的。既然一定要有自訂的加工工作，那就重新設計吧。

在幾天的努力下，起初第一版只是就功能來做，一個一個指令做完再拚湊著使用，目標是不需任何的再加工就能套入文獻探討中的案例。在經過數次調整指令的下法就形成了在〈章節 3.2.2 Catch-And-Retry 指令設計〉所述的基本設計方針。接下來就幾種認為頻率最高的使用形式介紹如下：

（一）指令基本順序

在本質上，剖面功能是沒有順序的因為每個都完全獨立。在實際上，多個剖面功能修飾同一段原始碼時，到了調用剖面功能時還是存在著執行順序。首先回顧一下在〈章節 3.2.2 Catch-And-Retry 指令設計〉所提，Catch-And-Retry 機制實務上執行段落共分成十段，在指令設計上有五個指令型。用這五個指令把十段執行段落組織出來。

這十段落簡單複習一下：

Enter - Try - Catch - Restore - Wait - Recovery - Retry - Fail - Ignore - Leave

若用設計好的 AspectW 來描寫這段機制，就如圖 3.16 所述。這個指令順序第一次看應該會覺得太複雜。若認為一次縫合五個剖面功能就覺得複雜難懂，那麼完全不使用的話，肯定會比這更複雜更難以閱讀。

開始前先理解一點，AspectW 的剖面功能順序由上往下(或由外向內)包裹機制，而例外由內向外傳遞。在這例子則包裹了五層，第六層是由 Do 函式包裹主功能邏輯程式碼區塊。

```
BarClass bar = new BarClass();
AspectW.Define
   .OnEnterLeave(FooEnterAction, FooLeaveAction)
   .Ignore<FooException>()
   .HandleFail(FooFailHandler)
   .Retry(3000, 2, (ex) => FooRecoveryAction())
   .Restore(new RestoreWhenFail(bar, (v)=>bar=(BarClass)v))
   .Do(() =>
   {
      bar.DoSomeChange();
   });
```

圖 3.16 AspectW 用在 Catch-And-Retry 機制的指令下法

這一段執行順序解讀如下：

- OnEnterLeave，當進入之前先調用 FooEnterAction，在離開之前調用 FooLeaveAction。
- Ignore，若後來發生了像 FooException 這類特定的故障狀況，就忽略不管。
- HandleFail，有些特殊的例外狀況必須調用自訂的 FooFailHandler 來決定如何處置。
- Retry，若發生了故障就先稍停 3 秒(3000 豪秒)，然後調用 FooRecoveryAction 做一些進階的自訂恢復操作，然後再重試但上限為 2 次。
- Restore，當出現故障時立即還原物件本機端物件 “bar” 的狀態。

這個例子並非是常用的案例，是混合了基本與進階的狀況。基本狀況預計常用 Restore、Retry，當無法滿足時再用 HandleFail 進階處理。而 Ignore 也算常使用，像社交網站應用中像「按讚」這類不是很重要的訊息就可以忽視。OnEnterLeave 大多用來做 UI 的介面管制，如：滑鼠、按鈕的管制，在按下按鈕後運算期間讓按鈕失能以免重複下達多餘的指令產出不必要的意外。

（二）基本重試Basic Retry

依據文獻探討來推論，圖 3.17 應是頻率最高的使用情境，有時應該連 Restore 都不需要。我們觀察到有些情境：在網際網路上有些錯誤的應對方式是只要單純的稍停一小段時間再重新執行出錯的指令甚至連復原動作都不必做。圖 3.17 描述的狀況就是如此，若不需要 Restore 只要把它移除即可。

```
BarClass bar = new BarClass();
AspectW.Define
   .Retry(3000, 2)
   .Restore(new RestoreWhenFail(bar, (v)=>bar=(BarClass)v))
   .Do(() =>
   {
      bar.DoSomeChange();
   });
```

圖 3.17 基本應用

在此程式碼範例內容表示，當執行主功能邏輯“bar.DoSomeChange()”出現故障時立刻回復狀態，然後稍停 3 秒再重試，重試的上限次數 2 次。

（三）多重例外故障，Multiple Exception

```
BarClass bar1 = new BarClass();
BarClass bar2 = new BarClass();
AspectW.Define
   .Retry<InvalidDataException>(3000, 2)
   .Retry<ApplicationException>(3000, 2)
   .Restore(new RestoreWhenFail(bar1, (v) => bar1 = (BarClass)v)
            ,new RestoreWhenFail(bar2, (v) => bar2 = (BarClass)v))
   .Do(() =>
   {
      bar1.DoSomeChange();
      bar2.DoSomeChange();
   });
```

圖 3.18 多重例外狀況處理

多重的故障、例外也是常見的狀況。圖 3.18 多重例外處理，其關鍵點是透過泛型機制在指令 Retry 指定針對要處理的例外故障。Retry 可以連續下達多次。Restore 也

支援多個想保護的狀態物件。要注意 Restore 參數的下法，它是一個 RestoreWhenFail 陣列的結構。

（四）參數化重試，Parameterized Retry

有時候也許換個參數再執行也通，比如傳送一些簡短的聊天訊息或通知短訊。也許一開始用比較嚴謹的協定像是“HTTPS”，但有些特殊的關卡過不了。也許改用較不嚴謹的協定像是“HTTP”就會過關了。若還不行或許改用替代方案用簡訊“SMS”做最後的備用方案。如果連簡訊還失敗就忽略此錯誤吧。

```
BarClass bar4 = new BarClass();
string protocol = "HTTPS";
AspectW.Define
   .Ignore()
   .RetryParam(3000, () => protocol = "HTTP", () => protocol = "SMS")
   .Do(() =>
   {
      FooSendMessage(bar4, protocol);
   });
```

圖 3.19 Parameterized Retry & Ignore

（五）發射導彈(firing a missile)

有些網際網路的指令就像是發設導彈一樣是無法收回的，這時候只能靠像“愛國者飛彈”的設計一樣，再送出另一個取消指令請求廢除或無效化剛發出的指令。

```
BarClass bar3 = new BarClass();
RestoreWhenFail res3 = new RestoreWhenFail(bar3, (v) => bar3 = (BarClass)v);
AspectW.Define
   .Retry(3000, 3, (ex) => {
        FooFireCancelMissile(bar3); // firing a Patriot.
        res3.Restore(); }) // restore state
   .Do(() =>
   {
      bar3.DoSomeChange();
      FooFireUpdateMissile(bar3); // firing a missile.
   });
```

圖 3.20 發射導彈(firing a missile)

如圖 3.20 發射導彈就是一個這樣的案例。主功能邏輯產出一些訊息存並更新所屬狀態物件“bar3” 隨後再由 FooFireUpdateMissile 送出到目的地伺服器，可過程出錯了，這時就需要再用另一個指令 FooFireCancelMissile 去請求目的地伺服器廢除或無效化剛剛送出的更新指令。可像這類的“導彈型”指令都必需特別設計 undo 機制才行。它並不是想要就有的。

3.3 設計過程

在完成文獻探討階段後，開始尋找技術上的解決方案。目標程式語言是 C# 並以 AOP 方法實現，在尋找過程中有 PostSharp、Spring.NET、Castle's DynamicProxy for .NET 等等，還有一些程式語言的網路社群關於重試(retry)與 AOP 的實作討論，之後發現了 AspectF 這個與眾不同、特別輕量的 AOP 實作方案。在反覆的互相比較這些實作方案，最後決定選用 AspectF。AspectF 本身也有提供重試能力，在代入文獻探討中的幾個補獲與重試的典型情境案例後，認為可用性並不高，其重試的指令總是有些冗餘也有些不足。在多日考量與評估後決定自製補獲與重試指令。

3.3.1 AspectW 核心碼開發過程

決定自製補獲與重試指令後，第一個考量是以 AspectF 為基礎再延伸補獲與重試指令；或是全部重新打造。開始分析 AspectF 的指令設計，可以推論其剖面功能指令應該都是為當時所負責的專案而臨場設計，尤其是補獲與重試情境的套用，常是不足又冗餘。因此決定全部重新打造，在剖面功能指令設計部份以補獲與重試機制為主，以呼應本研究的中心。在取名時為凸顯“縫合”概念故取名為“AspectW”，其中“W”代表縫合(weaving)。

接下來開始開發核心碼，原本計劃以 AspectF 原本的原始碼再精鍊，嘗試了二個版本後回頭比較發現並不比原本的更好，再精鍊也只能做到換湯不換藥，故決定延用原有的核心原始碼，也算是向原作者致敬。

3.3.2 AspectW 補獲與重試指令設計過程

補獲與重試指令的設計過程並非一帆風順，總共可概分為三個階段。第一個階段是先求有再求好，以技術上的實驗為主。首先確認在目標程式語言 C#，實作重試 retry 與還原 restore 是可行的。先研究重試，相關的解法還滿多的不久就完成了。再研究應用程式狀態還原，其在學術研究上以 STM(Software Transactional Memory)為主，研究相關網路文章與社群討論時發現並不成熟，故改成以「備份-還原」機制來實現。

到了第二階段指令試用。以文獻探討中以五個執行段落為原則設計指令，為呼應非功能需求之加入補獲與重試機制的指令要儘可能簡潔，曾設計一個指令就對應一個常用情境，在基本重試情境下還滿順手的，到了參數化、多重例外、狀態還原同時作用時，相關參數可能多達十個以上。在這階段的 Retry 指令的功能亦包含了追蹤 Trace 與還原 Restore 的能力。在試用時因參數太多，以至連設計指令的我們都會被這眾多的參數打亂心思，把參數下的不知所云。

到了第三階段時期。確認一個指令對應一個情境的設計是不可行的，必須以情境的使用頻率狀況來切割指令功能。同時也確定了重試指令 Retry 不再追蹤也不提供還原功能。也就是重試 Retry、還原 Restore、追蹤 Trace 這三項功能各自獨立，在需要時再串接組合成一體。同時也再度交叉參考文獻與之前的經驗演進，這時才發展出十個執行段落的觀點。更多說明請參考〈章節 3.2.2 Catch-And-Retry 指令設計〉。以下複習一下這十個段落：

Enter - Try - Catch - Restore - Wait - Recovery - Retry - Fail - Ignore - Leave

接下來的指令設計過程也進行了程序變更，原來是「情境」→「指令」，從應用情境演化出指令規格。變成「情境」→「執行段落組成」→「指令」，一樣從應用情境開始，然後改以執行段落組合出對應的執行情形，例如：無還原的基本重試→ Try-Catch-Wait-Retry；有還原的基本重試→Try-Catch-Restore-Wait-Retry；有多重例外重試若最終失敗則忽略→Try-Catch-Retry1-Retry2-Retry3-Ignore 等等。如此返復設計，為減少指令的數量開始進行收斂，最後收斂成五個主要指令型；同時因應實際細節的多樣性，把指令多載化以應對多樣化的事件狀況。

在此階段中指令 Ignore 原本不在這五個主要指令型中，因為它太單調了，原本計劃在 Retry 指令中以參數指定的形式處理。之後推論 Ignore 在意圖上是獨立的且頻率不算低，在此考量下決定把 Ignore 抽離出來成為另一個主要指令型。

3.3.3 與 AspectF 的重試指令比較

指令設計比較部份在此研究只比補獲與重試部份。AspectF 的剖面功能指令與補獲與重試相關的只有一個「Retry 指令」，但共有 5 個多載，下面介紹完整的版本：

```
public static void Retry(int retryDuration, int retryCount,
                         Action<Exception> errorHandler,
                         Action<IEnumerable<Exception>> retryFailed,
                         Action work,
                         ILogger logger)
```

以下一一說明各參數：

retryDuration：重試前的稍停時間。
retryCount：重試上限次數。
errorHandler：出現例外時自訂的操作程序。相當於前面所定義的*recoveryAction*。
retryFailed：最終依然失敗時的操作程序。相當於前面所定義的*failAction*。
work：被包裹的主邏輯程式碼。
logger：追蹤器。

這個指令的能力相當於在〈章節 3.2.2 Catch-And-Retry 指令設計〉中 Retry 指令的進階版。

第一個差異是追蹤器 logger 部份。AspectF 要求必須提供追蹤器，可是有些應用可能就是不想被追蹤，這就產生了冗餘。其中追蹤器必須實作 AspectF 定義的 ILogger 介面，這等於多了一些不必要的工作。提供了追蹤器後，記錄點的位置由 AspectF 決定，不能彈性的變更，這就產生了不足。我們設計的方式是把追蹤與重試指令解構。在 AspectW 裡的 Retry 指令只聚焦重試；追蹤則由 Trace 指令專門處理。

第二個差異是功能完整性部份。重試的應對行為在之前的分析，我們歸納出有四種：*retry*、*restore*、*ignore*、*handle-failure*。其中 restore 也就是應用程式狀態還原的指令在 AspectF 並未提供，開發人員必需另外處理此狀態還原的問題。

第三個差異是 AspectF 未提供參數化重試能力。

第四個差異是設計意圖表達的部份。在功能上 *retry*、*ignore*、*handle-failure*，這三項在 AspectF 是可以達到的，只是 AspectF 只有一個「Retry 指令」，只能明顯的表達出 *retry* 這項意圖而已。若真正的意圖是 ignore 或 handle-failure 是不明顯的必須加入註解說明。

第五個差異是鷹架碼部份。延續前一個差異，因為 AspectF 未直接支援 *ignore* 與 *handle-failure*，所以實現上必須加入額外的程式碼修飾，這就使得程式碼變得不簡潔。

基於這幾項差異，在套入幾個典型的應用情境，如：基本重試、多重例外故障、參數化重試、發射導彈等等，都有冗餘與不足的狀況。

第 4 章

系統實作與展示

這一章說明實作過程使用的平台與工具，還有成果展示。

4.1 實作語言與工具

自系統設計階段開始，一開始選取的目標程式語言就是 C# 。執行平台想當然爾以微軟的 Windows 作業系統與.NET Framework 為首選，以下一一介紹：

- 程式語言：C#

  C# [16] 是微軟推出的一種基於.NET 框架的、物件導向的高階程式語言。C# 由 C 語言和 C++衍生而來，繼承了其強大的效能，同時又以.NET 框架類別庫作為基礎，擁有類似 Visual Basic 的快速開發能力。C#由安德斯·海爾斯伯格 (Anders Hejlsberg) 主持開發，微軟在 2000 年發行了這種語言。

- 作業系統： Windows 7

  Windows 7 [17]是微軟公司推出的電腦作業系統，供個人、家庭及商業使用，一般安裝於筆記型電腦、平板電腦、多媒體中心等，於 2009 年 10 月發行。

- 開發平台：Microsoft Visual Studio 2010

  Visual Studio 2010 [18]，代號為"Hawaii"，已於 2010 年 4 月 12 日上市。微軟稱 Visual Studio 2010 整合式開發環境(IDE)的介面被重新設計和組織，變得更加清晰和簡單。新的 IDE 更好的支援了多文件窗口以及浮動工具窗，並且對於多顯示器的支援也有所增強。IDE 的外殼(shell)使用 WPF 重寫，內部使

用 MEF 重新設計，以提供比先前版本更好的擴充功能性。新的多重編程範式 ML 變體 F# 語言將會成為 Visual Studio 2010 的一部分，同時增加的還有文字模型化語言 M，以及視覺化模型設計器 Quadrant，這些都是微軟 Oslo 發展的一部分。

- .NET Framework 4

.NET Framework [19] 是由微軟開發，一個致力於敏捷軟體開發(Agile software development)、快速應用開發(Rapid application development)、平臺獨立性和網路透明化的軟體執行平臺。.NET Framework 包含許多有助於網際網路和內部網應用迅捷開發的技術。.NET Framework 是以一種採用系統虛擬機運行的編程平臺，以通用語言運行庫(Common Language Runtime, CLR)為基礎，支援多種語言（C# 、VB.NET 、C++ 、Python 等）的開發。比較大的更版幅度在 3.5 版，開始 LINQ 的支援，包括 LINQ to Object 、LINQ to ADO.NET 以及 LINQ to XML 。運算式目錄樹(Expression Tree)，用於為 Lambda expression 提供支援。現已更新到 4.5 版。

4.2 系統實作展示

在開發完成後，理想上的展示應該部署到目標系統上。而問題正在此處，補獲與重試機制是設計運行在網際網路上的大規模分散式系統，這是無法在實驗室也部署一套的。另外一點，例外故障的發生頻率就是小的，至少要以百萬分之一為單位，也就是百萬次的網際網路存取可能只出現一次故障。

這樣看起來補獲與重試機制似乎無意義。再看一下現在的 Google 搜尋引擎每天有數億次的搜尋，世界有名氣的社群網站 Facebook、twitter、WhatsApp 等，每天都有數億或數十億次的網際網路存取。若一百萬只出現 1 次的錯誤，在一千萬就變成 10 次，一億就變成 100 次，到了十億就有了 1000 次。這不是只發生一天，是天天都發生。若天天都發生 1000 次失敗的存取，那麼要花人力來處理吧。這不打緊，最讓人擔心的是客戶對產品的品質印象不是看機率而是零和，這類負面的印象也會經由人際關係或網路媒體傳開。然而這種大規模又同時分散全球各地的系統就算機器設備都正常的狀況下，也無法完全保證絕對不會故障。不過我們可以降低機率，把出現故障機率從一百萬一次變成一千萬才一次，也就是讓每天 1000 次故障降成 100 次，那麼負面的印象就有可能不會散播，也可以省下不必要的故障處理人力。

4.2.1 三個基本應用模式

因補獲與重試機制應用的目標是大規模分散式系統，這是展示時測試時都是難以實現的，又若完全憑空想像也不對。考量後，發現文獻探討時有提過一些真實的案例模型，那就以這些模型為應用目標。而比較的方式是有採用 AspectW 的程式碼與沒有採用但邏輯上完全相同且可編譯執行的程式碼。

這些案例模式有：一、Basic Retry；二、Parameterized Retry；三、Recovery From Multiple Exception；比較的重點放在程式碼糾纏程度與可讀性。

案例模式一：Basic Retry

為了讓比較上可以方便，把這兩份比較的原始碼放在一起。圖 4.1 有使用 AspectW。圖 4.2 沒有使用任何 AOP 機制。這兩份程式碼邏輯是一模一樣的。在圖 4.1 的程式碼相當精簡，且並沒有下達註解的狀況下依然可以清楚的判定出各段落的設計意圖。那一部份是主功能邏輯，非主功能機制又有那幾個被加入。這個案例裡有三個非主功能機制，Trace、Retry、WaitCorsor（管制 UI 以防呆）。在圖 4.2 的原始碼中，若沒有加註的話，還看得出來共加入幾項非主功能機制嗎，尤其是 Retry 機制的部份。就算有加註在未來的維護也容易混亂。

```
AspectW.Define
    .WaitCursor(this, btnBasicRetry)
    .Retry(3000, 2)
    .Trace(() => this.MyTrace("BEGIN : Basic Retry"),
           () => this.MyTrace("END\r\n"),
           (ex) => this.MyTrace("CATCH : {0}", ex.Message))
    .Do(() =>
    {
        _svc.SendMessage("John", "hello, long time no see.");
    });
```

圖 4.1 Basic Retry with AspectW

```
try
{
    // OnEnter
    this.Cursor = Cursors.WaitCursor; //@ UI control, mouse
    btnBasicRetry.Enabled = false;

    int retryCount = 2;
    for (; ; ) //@ for retry mechanism.
    {
        try
        {
            // before
            this.MyTrace("BEGIN : Basic Retry");
            // Biz-Logic
            _svc.SendMessage("John", "hello, long time no see.");
            // after
            this.MyTrace("END\r\n");
            break; // success & leave
        }
        catch (Exception ex)
        {
            // exception tracing
            this.MyTrace("CATCH : {0}", ex.Message);
            //@ for retry mechanism.
            if (retryCount-- > 0)
            {
                Thread.Sleep(3000);
                continue; // retry
            }
            throw; // fail & leave
        }
    }
}
finally
{
    // OnLeave
    this.Cursor = Cursors.Default; //@ UI control, mouse
    btnBasicRetry.Enabled = true;
}
```

圖 4.2 Basic Retry without AOP mechanism

案例模式二：Parameterized Retry

第二案例也是把比較的兩份程式碼擠到同一頁以方便比較。在圖 4.3 中的程式碼依然精簡且清晰，或許會有像 RetryParam 這類不直覺的指令出現，但只須一些說明即可。RetryParam 的功能是參數化重試，第一個參數代表稍停多少毫秒，第二個參數以後可以一直串接，變更參數的值後再重試。這個案例在執行時先使用較嚴謹的通訊協定 HTTPS，當失敗時改用較不嚴謹的 HTTP，還是失敗的話就 SMS 傳簡訊做後備。若不用 AspectW 的話程式碼就像圖 4.4 所示，其設計意圖若註解沒清楚標明是難以理解的。

```
string protocol = "HTTPS";
AspectW.Define
    .WaitCursor(this, btnParamRetry)
    .RetryParam(3000,()=> protocol="HTTP",()=> protocol="SMS")
    .Trace(() => this.MyTrace("BEGIN : Parameterized Retry"),
           () => this.MyTrace("END\r\n"),
           (ex) => this.MyTrace("CATCH : {0}", ex.Message))
    .Do(() =>
    {
        _svc.SendMessage("John", "see you again.", protocol);
    });
```

圖 4.3 Parameterized Retry with AspectW

```
try
{
    // OnEnter
    this.Cursor = Cursors.WaitCursor; //@ UI control, mouse
    btnParamRetry.Enabled = false;
    // parameters for doing retry
    string[] retryParamAry = new string[]{"HTTPS","HTTP","SMS"};
    int retryParamIndex = 0;
    string protocol = retryParamAry[retryParamIndex++];
    for (; ; ) //@ for retry mechanism.
    {
        try
        {
            // before
            this.MyTrace("BEGIN : Parameterized Retry");
            // Biz-Logic
            _svc.SendMessage("John", "see you again.", protocol);
            // after
            this.MyTrace("END\r\n");
            break; // success & leave
        }
        catch (Exception ex)
        {
            // exception tracing
            this.MyTrace("CATCH : {0}", ex.Message);
            //@ for retry mechanism.
            if (retryParamIndex < retryParamAry.Length) {
                Thread.Sleep(3000);
                protocol = retryParamAry[retryParamIndex++];
                continue; // retry
            }
            throw; // fail & leave
        }
    }
}
finally
{
    // OnLeave
    this.Cursor = Cursors.Default; //@ UI control, mouse
    btnParamRetry.Enabled = true;
}
```

圖 4.4 Parameterized Retry without AOP mechanism

案例模式三：Recovery From Multiple Exception

第三個案例算是比較符合真實的狀況，向遠端發出一個讀取數據的需求，這時可能會有兩種故障是只要重試就行了。數據上的不一致錯誤（用 DataException）模擬，或是網路的問題（用 IOException）模擬。這兩種故障用不一樣的應對方式。數據上不一致的錯誤就要求遠端伺服器先刷新數據（refresh data）後再重新讀取數據。網路上的錯誤就只能稍停一小段時間避開短暫的干擾或爆量延遲等狀況，然後再重新讀取數據。有套用 AspectW 的程式碼就像圖 4.5 所示。若完全不導入 AOP 機制那就像圖 4.6 所示，因為程式碼太長實在無法擠到同一頁所以分成兩段放在兩頁。

```
AspectW.Define
    .WaitCursor(this, btnMultiExRetry)
    .Retry<DataException>(0, 10
        ,(ex) => _svc.RefreshData()) // recovery action
    .Retry<IOException>(3000, 2
        ,null // recovery action
        ,(ex) => txtMessage.AppendText("network failure!\r\n"))
    .Trace(() => this.MyTrace("BEGIN : Recovery From Multiple Exception")
        ,() => this.MyTrace("END\r\n")
        ,(ex) => this.MyTrace("CATCH<{0}>", ex.GetType().Name))
    .Do(() =>
    {
        _svc.ReadData();
    });
```

圖 4.5 Recovery From Multiple Exception with AspectW

```
try
{
    // OnEnter
    this.Cursor = Cursors.WaitCursor; //@ UI control, mouse
    btnMultiExRetry.Enabled = false;

    int retryCount1 = 2;
    int retryCount2 = 10;
    for (; ; ) //@ for retry mechanism.
    {
        try
        {
            // before
            this.MyTrace("BEGIN : Recovery From Multiple Exception");
            // Biz-Logic
            _svc.ReadData();
            // after
            this.MyTrace("END\r\n");
            // success & leave
            break;
        }
```

圖 4.6 Recovery From Multiple Exception with AspectW - part1

```
        catch (IOException ex)
        {
            // exception tracing
            this.MyTrace("CATCH : {0}", ex.Message);

            //@ for retry mechanism.
            if (retryCount1-- > 0)
            {
                Thread.Sleep(3000);
                continue; // retry
            }
            else
            {
                // Handle Fail!
                txtMessage.AppendText("network failure!\r\n");
                throw; // fail & leave
            }
        }
        catch (DataException ex)
        {
            // exception tracing
            this.MyTrace("CATCH : {0}", ex.Message);

            //@ for retry mechanism.
            if(retryCount2-- > 0)
            {
                // recovery action
                _svc.RefreshData();

                continue; // retry
            }
            else // fail!
            {
                throw; // fail & leave
            }
        }
    }
}
finally
{
    // OnLeave
    this.Cursor = Cursors.Default; //@ UI control, mouse
    btnMultiExRetry.Enabled = false;
}
```

圖 4.6 Recovery From Multiple Exception with AspectW - part2

在這三個案例裡，光是可以少寫很多程式碼就能達到相同邏輯這一點，就足以說服我們自己了。仔細的觀看圖 4.2、圖 4.4、圖 4.6 就可以初步體會程式碼的糾纏(tangled)狀態是什麼模樣。尤其是像重試(retry)這類在主功能邏輯程式碼區塊前面、後面與例

外都散佈一些程式碼的機制只要同時來二個，程式碼狀態就會開始糾纏不清，可讀性當然也瞬間下降。

使用 AspectW 另外一個重要的理由是彈性，剖面功能也就是非主要功能機制的加入與移除是非常快速與方便的。剖面功能指令再配合著參數也能加大應用範圍，這在某些 AOP 解決方案是沒有的能力。使用 AspectW 另一個主要彈性是順序變換。想像一下，若要變更順序以圖 4.1 的例子來說只要把 Retry()指令與 Trace()指令上下移動就行了；而相同邏輯圖 4.2 只要搬動二行 MyTrace()的程式碼就行了，但實際上這類簡單的例子其實不多。案例模式三算是比較接近真實的一般狀況，若想變動順序，用 AspectW 來處理是小事一樁只須上下調換數秒之內就可解決；若是像圖 4.6 的程式碼肯定不是數秒之內就能解決的了。

4.2.2 二個 Facebook 案例套用

接下來再引入二個文獻探討 CatchAndRetry[3]中提到的案例，這個案例來自 Facebook 系統，雖也只能模擬業務流程上相同需求邏輯，不過也更接近真實的應用狀況，也是希望能以更接近真實的狀況下採用 AspectW。這兩個應用是：

一、Facebook Status Updates：取得使用者他所有朋友們最新的狀態並更新到自己臉書的首頁。

二、Organizing a Facebook Event：發出一個邀請函給他住的城市或地區的朋友們來參加一場派對。

第一個 Facebook 案例情境如圖 4.7 所示。有位名為“Alen”的人士他有四位好友。此例只以四位好友為例，不過真實世界數百位才是常有的情境。他打算取出他所有好友新更新的訊息。在執行中當然也要留下紀錄，事後可用來分析使用者行為。同時取得眾多朋友的訊息過程中，若有一、兩位朋友的訊息取得失敗就忽略不管吧。

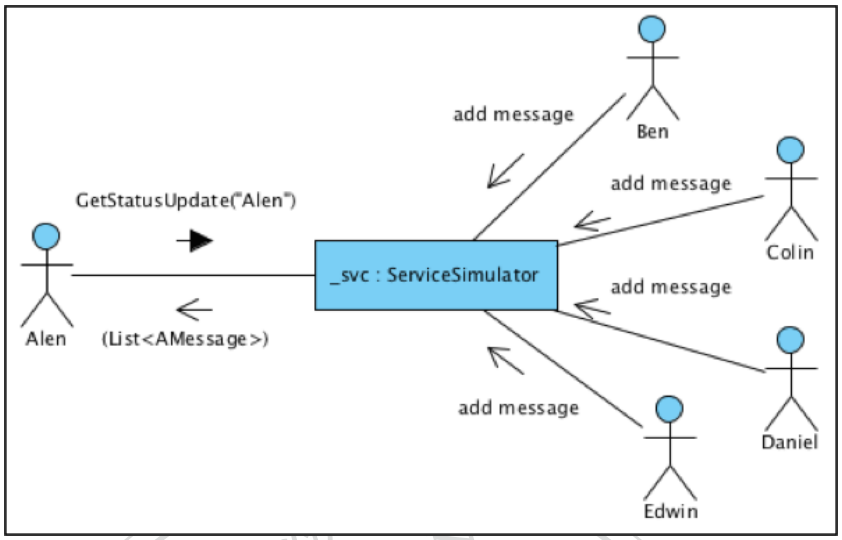


圖 4.7 Facebook Status Updates - scenario

```
/// main flow
AspectW.Define
    .WaitCursor(this, btnSkipFault)
    .Trace(() => this.MyTrace("BEGIN : GetStatusUpdate"),
           () => this.MyTrace("END\r\n"))
    .Do(() =>
    {
        List<AMessage> result = _svc.GetStatusUpdate("Alan");

        txtMessage.AppendText("SHOW RESULT : \r\n");
        foreach (AMessage m in result)
            txtMessage.AppendText("{0} : {1}\r\n", m.sender.Name, m.message);
    });
//-----------------------------------------------
/// to get someone's new status messages that all of his friends updated.
public List<AMessage> GetStatusUpdate(string name)
{
    List<AMessage> result = new List<AMessage>();
    Person[] f = friends.Get(name);
    foreach (Person p in f)
    {
        AspectW.Define
            .Ignore<IOException>()
            .TraceException<IOException>(
                    (ex)=> _logger.WriteLine("IGNORE : " + ex.Message))
            .Do(()=>
                result.Add(statusUpdates.Get(p)) //get a friend's new message
             );
    }
    return result;
}
```

圖 4.8 Case Study : Facebook Status Updates with AspectW

在圖 4-8 是截取了最關鍵的程式碼片段。在上半段的主流程程式碼中，有使用 AspectW 可以很快看出有加入非主功能機制 Trace 做使用紀錄。主功能是呼叫 “GetStatusUpdate(“Alen”)”函式取得“Alen”好友們的新訊息然後輸出到畫面。在下半段也有使用 AspectW。這一段碼先用“friends.Get(name)”把“Alen”所有好友找出來再用指令“statusUpdates.Get(p)”一一取得每人更新的訊息後再回傳。若在取得個人訊息失敗的話就用“TraceExcepiton”記錄下來但不處理直接“Ignore”忽略掉。這在社交網站應用上是合理的作法，因為大部份社交類的訊息是影響不大。且幾百位好友的訊息漏掉一位其實使用者也看不出來。在整體訊息傳遞的機制上，這些失敗未送達的狀態訊息可以在使用者後來再送出的請求中再重送即可。這事後處理機制是可以被接受的，因為社交應用的訊息是不需要有即時性的。

轉到技術面，再看一下圖 4-8 下半斷的程式碼可以看出 AspectW 是用在 foreach 迴圈裡面。此特性在一些 AOP 的方案中是沒有的能力。也是 AspectW 的賣點之一。

第二個案例情境是這樣的：“Alen” 打算明天舉辦一場派對邀請 Facebook 好友們來同歡，但因地理限制不可能把全世界的好友都請來，故只邀請跟他的住在同一個城市的好友們。圖 4-9 顯示這樣的情境，“Alen”送出了邀請不過只有其中兩位住在台北“city = Taipei”的好友有收到邀請訊息，另兩位住在紐約的好友就被過濾掉了。這一段碼在圖 4.10 一樣的只取其中關鍵的程式碼。一樣分成上半段與下半段。在上半段的程式碼中也有使用 AspectW 用來紀錄使用者使用經驗。這一段碼謹是模擬行為流程，在細節上先不看。程式碼程序中先用 “GetPerson” 取得 “Alen” 個人簡介資料 “me”。再用 “SendEventInvite” 送出邀請函。其中一個參數 “isValid” 其實是個條件審查函式，檢查住址是台北的話回傳 true 。

在圖 4-10 下半段的程式碼中，是個更複雜的程式碼組織結構，包函了巢狀結構，在 AspectW 的 Do 函式內的主功能程式碼區塊，還內函另一個 AspectW 陳述句。

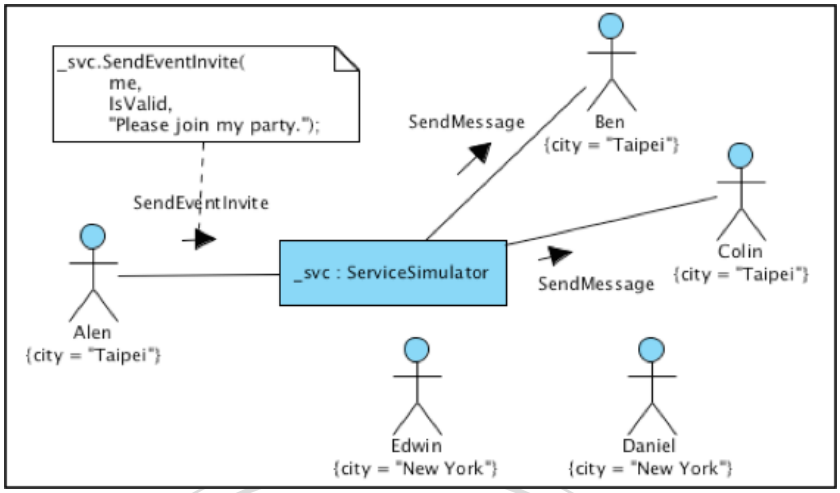


圖 4.9 Organizing a Facebook Event – scenario

```
AspectW.Define
    .WaitCursor(this, btnComposite)
    .Trace(() => this.MyTrace("BEGIN : SendEventInvite"),
           () => this.MyTrace("END\r\n"))
    .Do(() =>
    {
        Person me = _svc.GetPerson("Alan");
        _svc.SendEventInvite(me, IsValid, "Please join my party."); // template method pattern
    });
//-------------------------------------------------
public void SendEventInvite(Person me, Func<string, bool> isValid, string message)
{
    foreach(Person friend in friends.Get(me))
    {
        AspectW.Define
            .OnLeave(() => _logger.WriteLine("Done"))
            .Retry<DataException>(3000, 5,
                (ex)=> RefreshAddress(friend), // recovery action
                (ex)=> _logger.WriteLine("5 retries failed")) // fail action
            .TraceException((ex) =>
        _logger.WriteLine("CATCH@L1<{0}> : {1}" ,ex.GetType().Name, ex.Message)
            .Do(()=>
            {
                string address = friend.GetAddress();
                string protocol = "HTTPS";
                AspectW.Define
                    .WhenTrue(isValid(address))
                    .RetryParam<ApplicationException>(3000, ()=> protocol="HTTP" )
                    .TraceException((ex) =>
        _logger.WriteLine("CATCH@L2<{0}> : {1}",ex.GetType().Name, ex.Message))
                    .Do(()=>
                        SendMessage(friend, message, protocol)
                    );
            });
    }
}
```

圖 4.10 Case Study : Organizing a Facebook Event with AspectW

這個例子也是前面數個單獨機制的綜合應用，前面三個基本機制 Basic Retry、Parameterized Retry、Recovery From Multiple Exception 同時使用。在“SendEventInvite”函式中，加入好幾項非主功能機制，在一般的程式碼撰寫方法想找出主功能邏輯程式碼的位置與邏輯順序通常變成了開發人員的一個挑戰。在圖 4.10 程式碼中，可以很快的在 Do()函式找出主功能邏輯。非主功能機制也可以在縫合區一條目一條目的看清楚。

在圖 4.10 程式碼中，上半段可以很快的找出主功能邏輯只有二行，“_svc.GetPerson('Alen')”取得個人簡介資料“me”，再用指令“_svc.SendEventInvite()”送出邀請訊息。

下半段程式碼，“foreach(Person friend in friends.Get(me))”取得“me”也就是“Alen”的好友清單，再一一向好友“friend” 用指令 “SendMessage(friend, message, protocol)” 送出邀請訊息。其中的參數“message”放的就是邀請訊息。

其它部份都是非主功能機制，在前面的例子多已說明過。在下半段巢狀結構內的 AspectW 陳述句內有個指令“WhenTrue”這個指令的功能是當參數為“true”時才執行由 Do()包裹的程式碼區塊。仔細看一下這一句程式碼“WhenTrue(isValid(address))”，其意指當調用函式“isValid(address)”回傳值為“true”也就是說：當“Alen”好友的住址是“Taipei”時才成立，也才送出邀請參加派對的訊息。

這是個複合性的案例，也是個較接近真實狀況案例。雖是以 Facebook 為案例標的，其實在其它各類的社交應用系統都有類似的操作行為。這些訊息或指令在網路傳遞的程式碼大概都有一個特性，就是非主功能機制比主功能邏輯還多，會有這現象主要是對品質的要求。補獲與重試機制並不是治根的方案，但可讓故障機率下降。

第 5 章

結論與建議

5.1 結論

補獲與重試(catch and retry)與 AOP 搭配應用的研究並非新穎的研究項目，但我們重新思考 AOP 的源由與價值，也引用新的方法與新技術設計實作了“AspectW”。在語法設計上我們採用“流利介面(fluent interface)”，實作原理主要是 lambda expression (非新觀念，但近幾年才又被重視並實作到 C#)。

我們分析了補獲與重試機制的應用行為，並設計了相應的五個指令型。我們重新思考與評估我們所有找得到的 AOP 實作方案。我們發現了“AspectF”，一個特別的 AOP 實現方案，使用上可用“流利(fluent)”來形容。我們花了些時間去分析它的設計，以觀念來看它更注重「縫合」這一觀點。我們以它為基礎設計了 AspectW。硬是要用一句話來道出它的特徵那就是“程序導向”，這只是相對的因為它本身可是物件。

AspectW 延用了 AspectF 的核心碼在剖面功能剖份以補獲與重試應用重新打造。在試用多次後發覺一件事，在套用 AspectW 開發程式碼過程變成「二維式」了。在這之前寫程式就是由上往下寫，這一個方向的開發過程可以用一個維度來形容。比較有套用 AspectW 就不一樣了，撰寫程式碼時由上往下只須考慮主功能邏輯即可，而非主功能機制須橫切(cross-cutting)進去與主功能邏輯縫合成一體，這過程主要由左向右撰寫程式碼，在縫合過程兩個方向的邏輯並非有關聯，但也不是沒有關係不然就不會組合成一體，這樣組織整體流程程式碼的過程感覺像有二個維度。

AOP 的觀念啟發了我們對軟體系統開發與設計的新看法，有些事物是無法向下切割成子元件的。補獲與重試機制或說是現象，對於現在或至少是近未來的雲端運算系統的品質觀點也有了新的啟發。

5.2 未來發展

在以 AspectW 支援補獲與重試機制的幾個規格項目中，其中一項“交易(transaction)”未達成改以還原(restore)代替。這裡的“交易”是指「應用程式狀態」的交易機制，其實這已是另一個研究主題以 STM(Software Transactional Memory)為主。因 STM 技術尚未成熟故決定不採用，未來若成熟後就可以引入。另外在試用了多次後，認為未來研究可以加入幾個項目：

一、引入 STM。若未來技術成熟的話。

二、主功能與非主功能狀態交換。這一點在一般的 AOP 上是沒有的，不過試用多次套用 AspectW 縫合主功能與剖面功能時發現似乎是需要的，這一點還要再詳細檢驗看看。而若要導入的話應該可以套用 GoF 的 mediator pattern 來設計。

三、程式語言直接支援縫合的關鍵字與語法。若程式語言直接支援讓編譯器在編譯時就進行主功能與剖面功能縫合是最好的解了。

參考文獻

【1】 Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin. (June 1997). Aspect-Oriented Programming. *Published in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.* http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf, May 2014

【2】 陳恭，「剖面導向程式設計(AOP/AOSD)簡介」，政大資科系，Last revised: May 14, 2007 http://www.cs.nccu.edu.tw/~chenk/AOP-intro.pdf, May 2014

【3】 Emre Kıcıman, Benjamin Livshits, Madanlal Musuvathi (October 2009). CatchAndRetry: Extending Exceptions to Handle Distributed System Failures and Recovery, *Microsoft Research. PLOS '09, October 11, 2009, Big Sky, Montana, USA. Copyright 2009 ACM 978-1-60558-844-5/09/10* http://research.microsoft.com/en-us/um/people/livshits/papers/pdf/plos09.pdf, May 2014

【4】 Bruno Cabral, Paulo Marques (Sep. 2009). Implementing Retry - Featuring AOP. *4th Latin-American Symposium on Dependable Computing (LADC'09), September 2009* http://pmarques.dei.uc.pt/papers/bcabral_pmarques_ladc09.pdf, May 2014

【5】 系統架構設計 第 14 章，和春技術學院 http://el.fotech.edu.tw/localuser/kjyang/992/SA/ch14.ppt, May 2014

【6】 AOP 入門，Spring 技術手冊第四章 http://www.dotspace.idv.tw/Book/chap4.pdf, May 2014

【7】 Matthew D. Groves (June 2013). AOP in .NET - Practical Aspect-Oriented Programming - chapter 1,2, *Copyright 2013 Manning Publications. ISBN: 9781617291142* http://www.manning.com/groves/, May 2014

【8】Aspect-oriented programming, *Wikipedia, modified on 1 Feb. 2014.* http://en.wikipedia.org/wiki/Aspect-oriented_programming, May 2014

【9】 Omar Al Zabir (11 Jun 2011). AspectF Fluent Way to Add Aspects

for Cleaner Maintainable Code.
http://www.codeproject.com/Articles/42474/AspectF-Fluent-Way-to-Add-Aspects-for-Cleaner-Main, May 2014

【10】 Fluent interface, wikipedia
http://en.wikipedia.org/wiki/Fluent_interface , May 2014

【11】 如何使用 AspectJ Compiler 開發 AspectJ 程式 (2006-05-10)
https://sites.google.com/site/swankyhsiao/aspectj-with-ajc, May 2014

【12】 Dominik Stein (2002). An Aspect-Oriented Design Model Based on AspectJ and UML – chapter 3: AspectJ. *submitted to the Department of Business Arts, Economics, and Management Information Systems, University of Essen, Germany.*
http://www-stud.uni-essen.de/~sw0136/wissensArbeiten/DiplomarbeitDIIDominikStein.pdf, May 2014

【13】 Dominik Stein, Stefan Hanenberg, and Rainer Unland (2002). An UML-based Aspect-Oriented Design Notation For AspectJ. *Institute for Computer Science, University of Essen, Germany. AOSD 2002, Enschede, The Netherlands. Copyright 2002 ACM*

1-58113-469-X/02/0004.

http://www.dawis.wiwi.uni-due.de/uploads/tx_itochairt3/publications/StHaUn_AspectOrientedDesignNotation_AOSD_2002.pdf, May 2014

【14】 Mohsen (February 23, 2008). Using AspectJ and Java Annotations to Try Again.

http://zoftware.blogspot.tw/2008/02/using-aspectj-and-java-annotations-to_23.html, May 2014

【15】 Arulkumaran Kumaraswamipillai (July 2007). Creating Java custom annotations with Spring aspectj AOP

http://java-success.blogspot.tw/2013/07/creating-java-custom-annotations-with.html?utm_source=tuicool, May 2014

【16】 C#程式語言，維基百科。

http://zh.wikipedia.org/wiki/C♯ , May 2014

【17】 Windows 7，維基百科。

http://zh.wikipedia.org/wiki/Windows7, May 2014

【18】 Microsoft Visual Studio，維基百科。

http://zh.wikipedia.org/wiki/Visual_Studio_2010, May 2014

【19】 .NET Framework，維基百科

http://zh.wikipedia.org/wiki/.NET 框架, May 2014

102
碩士論文

剖面導向之例外處理與重試機制

AspectW：

政治大學資訊科學系

高沛功