笑談軟體工程
例外處理設計的逆襲

Teddy Chen

這份文件是《笑談軟體工程：例外處理設計的逆襲》的原稿。本書由悅知出版社於2014年出版，書中內容大多來自作者Teddy的博士論文研究與部落格搞笑談軟工文章，經過Teddy改寫與出版社編輯後出書。

由於本書已經絕版，因此Teddy將書本原稿公開，提供給鄉民參考。

Teddy Chen

泰迪軟體

2020/04/20

目 錄

致謝

首先感謝「生活壓力」讓 Teddy 下定決心撰寫這本書。不是為了賺版稅，因為版稅真的少得可憐。而是希望藉由本書的出版，可以和更多人分享 Teddy 對於**「透過妥善的例外處理開發出高品質軟體」**的經驗，增加日後開課與顧問服務的生意。

例外處理對於絕大多數軟體開發人員而言，是一個非常傷腦筋的難題。Teddy 在涉入這個領域（2005 年）之前，每當遇到例外處理的問題也是被搞得非常頭痛。感謝指導教授鄭有進老師與謝金雲老師，當年在「半推半就」的情況下，讓 Teddy「多做」了這個題目，使得自己有機會打破一個多年來存在自己心中未解的謎團。也感謝北科大資工系軟體系統實驗室的學弟、學妹們，多年來持續耕耘著例外處理設計這個冷門題目，不但拓展了 Teddy 的學習領域，也讓自己免於獨學而無友的窘境。

本書許多靈感與內容受到多位前輩所撰寫的書籍與文章的啟發，族繁不及備載，在此一併謝過。也要感謝本書的編輯與悅知文化幕後工作人員的辛勞，讓本書得以順利出版。還有感謝報名參加「例外處理設計與重構實作班」的學員們，你們對於課程的回饋意見，成為 Teddy 修正本書內容的參考。

最後，也是最重要的，感謝陪伴 Teddy 多年另一半 Kay，持續忍受 Teddy 長時間在電腦前面寫稿與耕耘「搞笑談軟工」部落格。還有 Teddy 的母親，沒有她也就沒有本書的作者，也就不可能有本書的誕生。

序言

這是一本例外處理的書，也是一本軟體設計的書

是的，身為一位專業的軟體開發人員，從小到大你可能學過各式各樣的軟體設計技術與方法。從最基礎的程式語言（C、C++、Java、C#、VB.NET、Objective-C、JavaScript、Ruby、Python等）、資料結構與演算法，到物件導向分析與設計、設計模式、軟體架構，以及各種敏捷開發實務做法，包含自動化測試、測試驅動開發、行為驅動開發、持續整合、敏捷設計原則等。以上，所有的「大師」費盡心力，都在告訴你一件事：**「如何設計軟體的光明面」**。

什麼！「軟體的光明面」？是的，軟體跟宇宙萬物一樣，有光明面也有黑暗面。軟體的光明面就是**「正常行為」（normal behavior）**，而黑暗面則是**「異常行為」（abnormal behavior）**。就好像一般民眾看到「黑道」與「壞人」，人人莫不避而遠之。大部分的開發人員對於軟體的「黑暗面」與「異常行為」，都感到很頭大，在內心中抱持著「閉上眼睛就以為看不見」的心態：「反正程式看起來可以動就好了」，您說是嗎？

軟體設計包含著正常行為與異常行為，兩者相互影響，忽略任何一方，都可能讓原本精心規劃的設計變得不堪一擊。既然異常行為的設計那麼重要，與正常行為相較，為什麼很少人談論異常行為？

因為很難

對付異常行為很難，因為它「躲在正常行為的影子下」，很容易被有意、無意地被忽略。其次，它與正常行為「水乳交融」，「我泥中有你，你泥中有我」。套句軟體開發的術語，它是一種「橫切關注點」（cross-cutting concern），需要具備怪醫黑傑克與庖丁解牛的能力，同時關注多個面向，才有可能讓你在對「病人」（程式）進行「外科手術」（處理例外）的時候，避免不小心切斷動脈，導致「失血過多」（越改越亂）而讓病人提早上天堂的窘境。

俗話說「夜路走多了，自然會碰到鬼」，本書累積了 Teddy「碰到鬼」的經驗，從例外處理設計出發，專門探討軟體設計黑暗面的問題。希望補足開發人員長久以來所欠缺的「維他命」。

***

這是一本寫給開發人員的書，也是一本寫給管理階層的書

看到書名你可能會以為這是一本寫給開發人員的書，錯！開發人員的錢要賺，管理階層的錢也要賺。嗯嗯，Teddy 的意思是說，例外處理做得好，可以提升軟體的強健度，提高客戶滿意度，幫公司賺大錢，減少程式錯誤，早點下班回家陪家人，順便顧肝。

本書分為六部，其中的「第一部 例外處理的現況」、「第二部 例外處理基本觀念」、「第四部 為什麼例外處理那麼難？例外處理的 4+1 觀點」、「第五部 強健度等級與例外處理策略」等內容，技術含量相對較低，對於不想知道太多例外處理實作細節的管理階層可以只看這幾個部分。

另外像是〈Column A. | 帶賽的人很適合當神秘客〉、〈Column C. | 網路又斷了〉、〈Column D. | 這是你的問題，不是我的問題〉、〈Column E. | 你如何評價成功〉、〈Column G. | 升級、降級，傻傻分不清楚〉、〈Column I. | 客戶滿意，老闆賺錢，你護肝〉也很適合管理階層當做小說來看。

至於開發人員，當然是要「怒看」整本書的內容才夠「給力」。

***

用什麼程式語言有差嗎？

例外處理設計的重點不在程式語言和語法，而是在於背後的設計原理。雖然本書範例以 Java 語言為主，但也一併討論了許多 Java 語言和 C#語言（支援與不支援 checked exception 的雙方代表）在例外處理設計精神與作法上的差異。書中所介紹的例外處理設計觀念、名詞、原則、方法、與重構，只要是物件導向語言，不管是 Java、C#、VB.NET、Python、Ruby 等，大都可一體適用。

至於採用 Java 語言作為例子的原因很簡單：「因為它是當代流行的商業語言裡面，例外處理機制最困難（或是說最「討厭」）的語言。」搞懂了 Java 的例外處理，再應用到其他物件導向語言，就好像喝開水一樣，變得非常容易。

以上翻成白話文就是說：不管你是使用什麼語言，都應該要買一本帶回家。

書中用語和排版

以一本中文書而言，本書的英文稍微多了一些。有些英文術語翻成中文之後，閱讀起來總是少了點 fu（味道），所以 Teddy 會選擇性地保留這些英文，例如 checked exception、unchecked exception（若是翻成「受檢例外」、「非受檢例外」，念起來總覺得怪怪的）。但英文名詞第一次

出現的時候，在它的「前後」或「附近」一定會有相對應的中文翻譯或說明。

為了區分特別的名詞，本書的程式碼採用 `Courier New` 字體，書中提到模式名稱（pattern name）、壞味道（bad smell）、重構（refactoring），會用大寫開頭的英文斜體字表示。例如 *Command*、*Dummy Handler*。對於本書中所提出的例外處理重構，除了以斜體字表示以外，還會在名稱之後加上介紹該重構的章節頁碼，例如 *Replace Error Code with Exception（240）*。

一些軟體領域常用的英文名詞書中會直接用中文表示，例如用 function 與 method 翻譯成「函數」（有些中文書會使用「函式」或「方法」）。Class、object、instance、type 用「類別」、「物件」、「實例」、「型別」表示，Thread、process 用「執行緒」、「行程」稱呼。Application programming interface（API，應用程式介面）則直接使用 API。

最後，重要的觀念會以**粗體字**加以強調。

***

本書許多內容最初來自於 Teddy 的博士論文《Java Exception Handling: Models, Refactorings, and Patterns》以及工作上的實踐經驗，經改寫後從 2007 年起陸續發表在「搞笑談軟工」部落格上，最後集結成冊再加上強烈月光照射之後變身成為大家現在看到的這本書。書中內容經過 Teddy 多次修改與審校，但刻意保留原本在部落格中以「鄉民」稱呼讀者的方式。此為特意為之，並非 Teddy 偷懶，特此說明。

鄉民們若對本書內容有任何問題、心得、稱讚、發現錯誤、或是看到不爽想翻桌的時候，歡迎直接寫信到 teddy.chen.tw@gmail.com 給 Teddy，或是加入 Facebook 上的「搞笑談軟工社團」https://www.facebook.com/groups/268206229913141/，一起討論。

Teddy Chen
2014 年 1 月 21 日

第一部 例外處理的現況

1 你對例外處理有多了解？

本書第一章先幫鄉民們做個簡單的 YES/NO 測驗。以下有 20 題是非題，回答 YES 的數目越多，代表鄉民們對於例外處理的了解越深入。請直覺性的回答以下問題，準備好筆，計時 10 分鐘，開始作答。

- ☐ Q1 你曾經讀過任何一本例外處理的書籍嗎？
- ☐ Q2 你是否了解不良的例外處理，輕則造成使用者不便，重則造成生命財產損失？
- ☐ Q3 你知道「缺陷」（fault）、「錯誤」（error）、「失效」（failure）、「例外」（exception）的差別嗎？
- ☐ Q4 你知道區分「設計缺陷」（design fault）和「元件缺陷」（component fault）對於例外處理有何幫助嗎？
- ☐ Q5 你聽過物件導向語言例外處理機制的 10 項設計因素嗎？
- ☐ Q6 你能說出 Ruby 和 Java/C#的例外處理機制最大的差別在哪裡嗎？
- ☐ Q7 你知道例外處理設計的決策有哪些脈絡（context）可以參考嗎？
- ☐ Q8 你能夠區分「例外處理」和「容錯設計」的差別嗎？
- ☐ Q9 你清楚處理 `NullPointerException` 或 `NullReferenceException` 的作法嗎？
- ☐ Q10 你知道 `try-catch-finally` 各負擔何種責任嗎？
- ☐ Q11 你能說出 Java 的「受檢例外」（checked exception）與「非受檢例外」（unchecked exception）有何不同嗎？
- ☐ Q12 你知道哪些 Java 例外類別套用「例外階層」（*Exception Hierarchy*）設計模式，哪些又套用「聰明例外」（*Smart Exception*）設計模式嗎？
- ☐ Q13 你知道在敏捷方法中，要如何讓例外處理設計逐步成長的方法嗎？
- ☐ Q14 你能夠判斷一個程式是否存在「例外處理壞味道」（exception handling bad smell）嗎？
- ☐ Q15 你會寫「重試」（retry）的例外處理程式嗎？
- ☐ Q16 你知道一個函數如果只有一個 `try` 敘述有什麼好處嗎？
- ☐ Q17 你知道 C#和 Objective-C 的例外類別為什麼要實作 *Variable State* 設計模式嗎？
- ☐ Q18 你知道在資料庫裡面找不到資料要傳回 null 還是丟出 exception 嗎？
- ☐ Q19 你知道例外處理程式執行失敗該怎麼辦嗎？
- ☐ Q20 你知道花小錢但卻可大幅提升軟體強健度的方法嗎？

***

友藏內心獨白：看完本書之後就有能力可以全部回答 YES。

2 例外處理的重要性

智慧型手機的啟示

2007 年 3 月 Teddy 買了一支具有 GPS 導航功能的智慧型手機 Mio A701。在新鮮感還沒消失之前，Teddy 在網路上到處尋找關於智慧型手機的資料，有人提到執行 Windows Mobile 作業系統的智慧型手機，過一陣子就要幫它「插屁屁」（重新開機）。不知是應用軟體沒寫好，還是作業系統不穩，這些智慧型手機相對於個人電腦更容易當機，需要經常性的重開機。既然大家都有重新開機的「需求」，很多機種乾脆做了一個重開機（reset）按鈕，並將其設計在機體的下方。為了避免使用者不小心按到重開機按鈕，所以重開機按鈕都設計成凹進去的，需要用觸控筆尖尖的那一端來碰觸重開機按鈕，看起來就好像在幫手機「插屁屁」一樣[1]。

Mio A701 的重置按鈕（照片右方印有 RESET 字樣）

Teddy 用了智慧型手機沒多久之後，強烈感受到「插屁屁」這個「功能」真的是太重要了。為什麼智慧型手機這麼容易當機？好像時空倒轉，回到當年那 Windows 3.0/3.1 動不動就出現藍底白字的時代，需要不斷的按下電腦上面的重新開機按鈕。

軟體出問題導致電腦或是電子設備當機的原因很多，其中一個常見的原因，就是**不良的例外處理**。一個軟體系統的行為，可以區分為**正常行為（normal behavior）**與**異常行為（abnormal 或 exceptional behavior）**這兩大類。在產品開發過程中，由於專案時程緊迫、人力有限、經驗不足等因素，開發人員能夠在上市之前把產品在正常狀況之下所應具備的功能做出來，就已經要偷笑了，哪還有閒功夫去考慮例外狀況。因此，雖然妥善處理程式中所發生的例外，可以增

[1] 現在的智慧型手機，只要長按電源按鍵，就可以重新開機，不需要再「插屁屁」了。

強軟體的**強健度（robustness）**，也就是讓軟體比較不容易當機，但礙於現實狀況，很多軟體的例外處理都處於嚴重不足的情況。

鄉民甲：例外處理是什麼，可以吃嗎？

耶，那麼當程式遇到例外時該怎麼辦？鄉民們可能會想：「 嘿嘿嘿，很簡單的啦，只要把例外捕捉（catch）起來然後直接忽略不就好的啦。雖然有點窩藏犯人的嫌疑，反正眼不見為淨，寫程式的是我，我不講老闆也不會知道的啦。」也有一些開發人員對於程式執行可能會遭遇例外狀況這件事「渾然不知」，抱持著「反正程式當掉客戶自然會出聲，到時候再來解 bug 就好了，搞不好還可以因此爭取到一些打混摸魚的時間。」稍微負責一點的開發人員可能會捕捉例外並將其寫到日誌檔、直接列印出來、或是交給程式最外圍來處理（通常是啟動該程式的函數，例如 `main` 函數或是某個執行緒）。至於那種會思考具體例外處理策略的開發人員，算是「人間極品」，可遇而不可求。

好吧，就算鄉民們從現在開始立志要成為這種「人間極品」，打算大展身手好好修理一下這些在窩藏在程式中，不請自來造成系統不穩定的不速之客，但是，要怎麼做？這時候你靜下心來用力回想一下，距今 N 年前的學生時代，有什麼課程曾經教過例外處理設計？嗯，好像沒有。沒關係，身為一位有理想、有抱負的開發人員，褲襠裡隨時藏著幾本程式語言的書也是很正常的事情。先翻翻 VB 這一本，再看看 Java 這一本，還有 C++、 C#、Objective C、Ruby、Python、JavaScript 等。奇怪，怎麼好像除了把例外這個燙手山芋往外丟（rethrow），或是直接輸出到畫面上，難道就沒什麼其他更好的處理方法了嗎？此時，你內心吶喊著：「萬能的天神，請賜予我神奇的力量，讓我好好地處理掉這些該死的例外吧！」

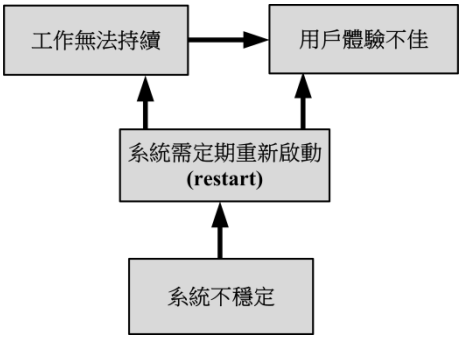
***

重開機不就好了

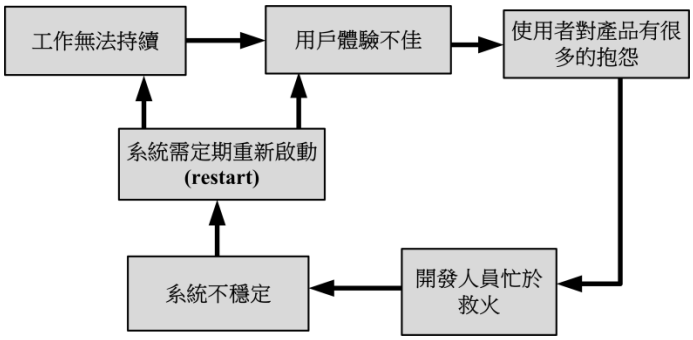
有些鄉民們可能會想：「有那麼嚴重嗎？」系統不穩重新開機一下就好了啊。現在讓 Teddy 在「公堂之上假設一下」，看看系統不穩會造成什麼問題。

系統不穩導致重新開機，因此：

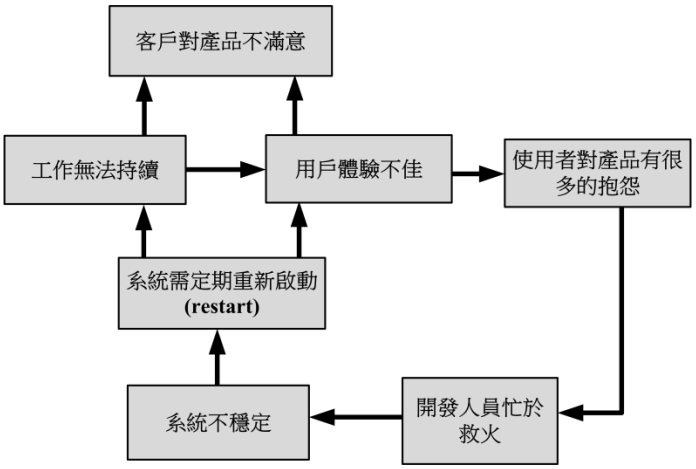
- 工作無法持續：假設你正在用電腦轉換多媒體檔案，或是執行複雜的科學模擬運算，工作執行到一半軟體當機。重新開機之後所有工作從頭開始，之前花費的時間全部浪費。此外，工作無法持續也會進一步導致不良的用戶體驗。
- 用戶體驗不佳：想像你正在用手機跟重要客戶通電話，電話講到一半手機突然重新開機。或是在一個陌生城市裡使用導航軟體，開車過程中導航軟體突然當機。以上狀況，輕則損失金錢，重則造成生命危險。



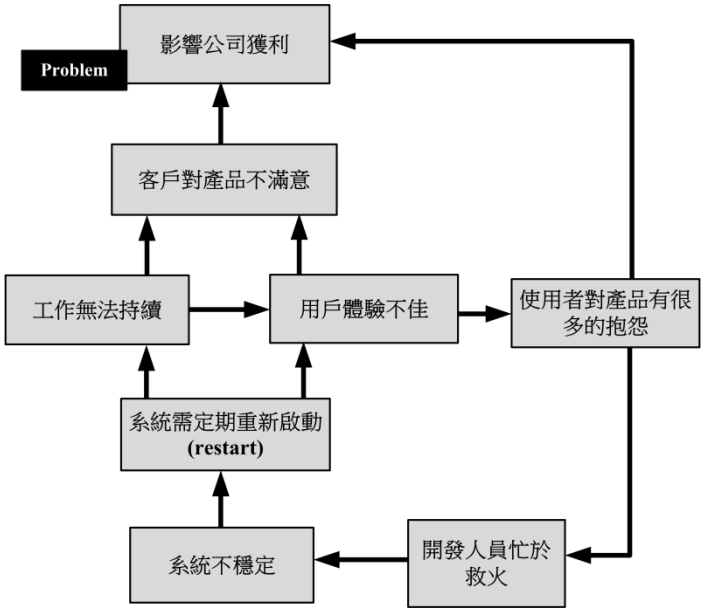
工作無法持續與用戶體驗不佳，進一步導致使用者對產品不滿，不斷地透過各種管道抱怨。公司收到如雪片飛來的客訴，因此催促開發人員趕緊處理。開發人員在壓力之下忙於到處救火，在匆促修改之下，系統穩定度時好時壞。有時修了一個 bug，產生其他更多 bug，導致惡性循環，每天加班爆肝，永世不得超生，一直到離職或是公司倒閉為止。



人的忍耐是有限度的，使用者經過一番抱怨與等待之後，產品穩定度不見提升，因此對產品失去信心。



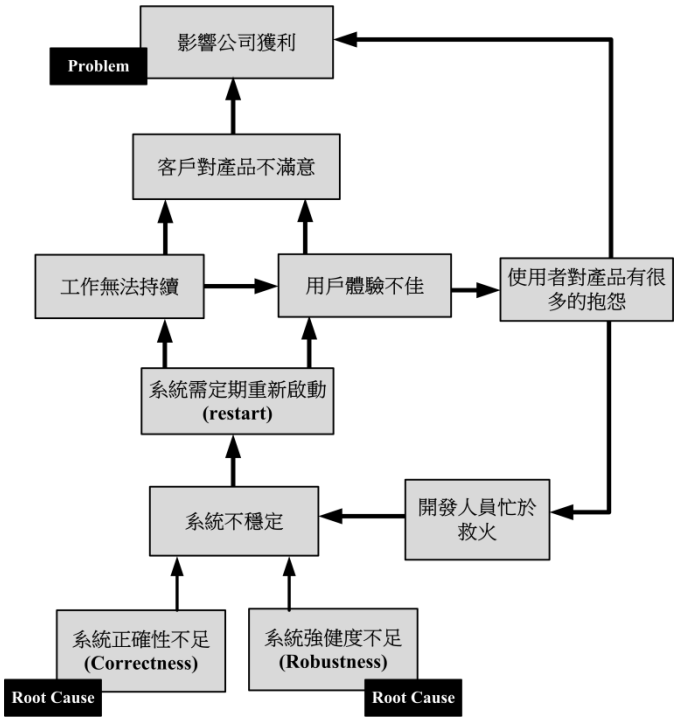
客戶對產品不滿意，日後便很可能不會購買公司的任何產品。同時，客戶對於產品的抱怨，也會導致產品口碑不佳，傷害公司形象與商譽。這兩者所造成的傷害，都會**影響公司獲利**。這就是系統不穩定對公司所造成的**問題**，絕非要求使用者重新開機，或是反過頭來跟使用者嗆聲：「自由世界，你喜歡退你就退」，就可以推個一乾二淨，當做沒事發生。



***

為什麼系統不穩定

認識到系統不穩定所造成的問題，接下來要思考是什麼原因造成系統不穩定？依據 Meyer[2]的看法，原因只有兩點：就是「**正確性不足**」與「**強健度不足**」。



如下圖所示，正確性（correctness）和強健度（robustness）是兩個不同但彼此互補的概念。**如果一個函數有依據規格來實作，我們說這個函數是正確的。**例如，一個計算兩個整數相加的函數 add，傳入 1 和 2，計算之後傳回 3。依據加法規格，這個結果是正確的。如果 1+2 傳回 5，這個函數就是不正確的。正確性是建構一個穩定系統的基本條件，不正確的函數，就算是在正常的狀況之下，也不可能表現出穩定的系統行為。

強健度是指系統對於異常狀況的回應能力。例如，一個儲存資料的 `save` 函數，規格書只規定接受一個檔案名稱與一個字串，然後將這個字串存入指定的檔案，並覆蓋原本檔案中的內容。考慮到異常狀況，如果發生儲存空間不足、系統忙碌、無法存取儲存媒體、儲存到一半停電等

[2] B. Myer, *Object-Oriented Software Construction, 2nd ed.*, Prentice Hall, 1997.

狀況，在規格書中並沒有特別描述。如果 `save` 函數在這些異常狀況之下，依然可以讓系統處在一個正常的狀態，則 save 函數就具有較高的強健度。人類如果身強體壯，當未知的流感病毒入侵時，就比較不容易感冒生病。同樣地，軟體如果身強體壯，也就不容易因為異常情況而輕易當機。

***

長期以來，不論是在學校或是在業界，關於軟體開發的技術訓練，絕大部分的焦點都著重在正確性上面。舉凡程式語言、資料結構、演算法，物件導向分析與設計、設計模式、測試、軟體重構等，這方面的資料已經很多，本書就不再錦上添花。

關於如何透過例外處理來增強軟體系統的強健度，相關資料並不多，就算有也是零零散散的局部資料，或是過於理論的研究論文，不易直接應用於工作之上。本書抱持著「雪中送炭」的精神，耕耘這一塊「不正常的軟體行為領域」。期望讓開發人員可以在輕鬆又不失嚴肅的情境之下，了解例外處理設計的必備知識，做出更好、更穩定的軟體，讓公司賺大錢，自己升官發財。

以上，就是購買本書回家珍藏的原因。

***

友藏內心獨白：花小錢，賺大錢，可能嗎？繼續看下去就知道。

3 我真是猜不透你啊：種花篇

講到軟體的強健度不足，會造成公司喪失生意機會或損害商譽。鄉民們可能會懷疑：「我們身邊的資訊系統，體質真的有那麼纖弱？」請看以下故事。

故事一

時間在 2011 年 7 月的某一天，Teddy 看到一則報導，「種花[3]」表示到 2020 年的時候，要家家戶戶都有 1G 的網際網路連線可以用。但是，根據網路上看到的消息，日本現在已經有 1G 的網路了，而且好像是雙向 1G，而韓國則是在 2012 年就要提供 1G 的網路連線。蝦米，Teddy 有沒有看錯，2020 年 ？！「種花」要等到 2020 年才要提供 1G 的網路服務，落後別人這麼多年還敢拿出來講。

看到這裡，讓 Teddy 想到「種花」最近強打 50M 光世代網路，別人都已經在用，或是準備要用 1000「妹」了（哇，1000 個妹，家裡都坐不下了），「種花」的 50「妹」相形之下就弱掉了。什麼，你說要那麼快地速度幹麼？你家住海邊啊，管那麼寬。

想一想 Teddy 還在用 8M/512K 的 ADSL，不禁悲從中來。好吧，50M 還是大於 8M，這麼簡單的算術問題相信大家都能了解。反正最近閒著沒事，想說去申請一下「種花」的 50M/5M，結果卻是令 Teddy 感到吐血。

首先，Teddy 在網路上填寫申請資料，填到最後一步的時候，畫面下方居然出現以下四點備註：

1. 光世代若須附掛市內電話，則客戶名稱、裝機地址須與附掛之市內電話相同，如不相同，請先辦理一退一租、移機或另申請一線電話。
2. 有關光世代進一步資訊，請參考光世代服務說明。
3. 光世代網路品質為「Best effort」模式，不提供頻寬保證。
4. **請於網路申請成功後十個工作天內攜帶證件至各地營運處窗口辦理驗證及預約裝機時間，逾期未辦理，本公司將註銷該申請。**

前三點 Teddy 都能接受，這第 4 點是怎樣？！都已經在網路申請了，還需要攜帶證件至各地營運處窗口辦理驗證及預約裝機時間。Teddy 都已經是「種花」ADSL 超過 10 年的客戶，還要驗證個什麼東東？還有，預約裝機時間不是打通電話來確認一下就好了嗎，誰有時間還跑去各地營運處窗口去辦理？**令人懷疑網路申請的目的到底在哪裡？**

[3] 中華電信在江湖上的別名。

算了，誰叫他是「種花」呢，不能以常人的眼光來看待，Teddy 認了。填完資料按下確定，請看下面畫面：

圖 3-1

現在是怎樣，生意太好不想接單，還是流量太大灌爆系統，填個小小的申請單居然還能夠發生系統程式錯誤，還建議 Teddy 「立即撥打服務專線」，這真是**「電子化比不上一通電話」**的最佳範例。

結論：想要 50 妹，沒這個命啊。

***

故事二

「案件」發生在 2012 年 4 月。前幾天 Teddy 把用了五年的 Mio A701 換成 HTC One X。為了安裝中華電信贈送的導航王軟體，Teddy 必須連到中華電信網站去註冊一組帳號。經過一番奮鬥（等一下會提到），好不容易註冊完畢變成一般會員。此時 Teddy 看到網站上面提到只要是中華電信的市話、手機、ADSL 用戶，經過註冊之後可以變成白金會員。雖然搞不清楚白金會員可以幹嘛，但是想說隨手順便註冊一下好了。沒想到這個「隨手順便註冊一下」的想法，讓 Teddy 又再度見識到中華電信網站系統的奇妙之處。

先說明一下註冊一般會員時 Teddy 所遭遇的問題。圖 3-2 是中華電信註冊一般會員資料的畫面，很標準的一個表格，一分鐘之內就可以填完，不是嗎？嘿嘿嘿，當然不是，如果那麼簡單就讓 Teddy 搞定，這就不叫做中華電信了。

圖 3-2：註冊一般會員畫面

填完資料之後，按下確定送出，看到圖 3-3 的錯誤訊息。

圖 3-3：錯誤訊息畫面

奇怪，e-mail 格式錯誤？應該沒有打錯啊。好吧，再試一次，應該說，再試一百次結果還是一樣，系統依舊認為 Teddy 的 e-mail 格式不正確。很顯然是系統判斷 e-mail 格式的程式有問題，因為 Teddy 的 e-mail 是 teddy.chen.tw@gmail.com，可能是系統無法接受@符號前面有「.」吧。這明明是一個合法的 e-mail 帳號啊，怎麼辦？沒辦法，只好用另外一個 Teddy 很少使用的 e-mail 來註冊會員資料了。

***

「天將降大任於斯人也，必先苦其心志，勞其筋骨，餓其體膚，空乏其身，行拂亂其所為，所以動心忍性，增益其所不能。」接下來還有一連串的挑戰等著 Teddy。如圖 3-4 所示，要註冊成白金會員，可以選擇：HiNet 用戶、手機用戶、市話用戶。

圖 3-4：註冊白金會員

既然是因為手機而註冊會員資料，那就選手機用戶吧。中華電信會送出一則簡訊認證碼到手機上面，拿到認證碼之後要輸入到如圖 3-5 的畫面中。

圖 3-5：手機用戶註冊，輸入認證碼

拿到簡訊認證碼之後，按下確定送出，鄉民們您猜發生什麼事？當然還是錯誤訊息伺候，請參考圖 3-6。

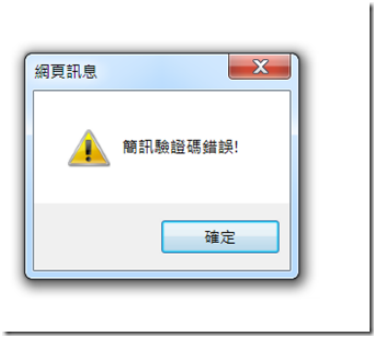


圖 3-6：認證碼錯誤

難道 Teddy 手拙打錯認證碼？再試一次，應該說，再試一百次也一樣，還是錯誤。那再按一下畫面上的「取得簡訊認證碼」，換一組新的認證碼試看看好了。結果哩，還是錯誤，只不過換了新的錯誤訊息，如圖 3-7 所示。

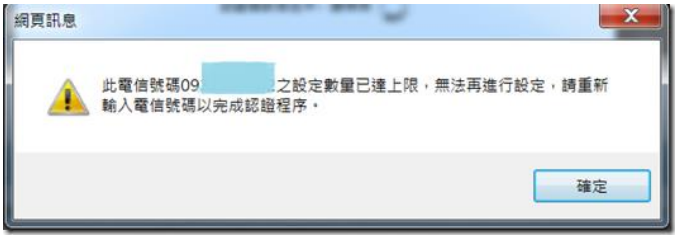


圖 3-7：取得簡訊認證碼錯誤

Hello…有人在家嗎？Teddy 才按了一次「取得簡訊認證碼」就已達上限？如果只能按一次，按完之後這個按鈕應該要被停用（disable）吧！這下子 Teddy 真的沒輒了，很顯然是系統有問題。還好，還有另外兩種註冊白金會員的方法。這次改選市話用戶好了，市話用戶應該比較容易…吧？請參考圖 3-8。

圖 3-8：市話用戶註冊畫面

輸入完畢所需資料之後按下確定送出，如圖 3-9 所示，顯示一個 0800 電話號碼以及一組認證密碼。

圖 3-9：顯示市話用戶認證密碼

Teddy 立刻撥打上面的 0800 電話，接電話的是電腦語音系統。按照語音系統指示輸入上面的認證密碼，此時 Teddy 只聽到語音系統快速的講了幾句話，講完之後電話就立刻被切斷了。來不及記住語音系統回覆的全部內容，大致上是說：「認證密碼錯誤」還是「資料庫中找不到認證密碼」之類的。

當天是四月一號還是農曆七月半嗎？怎麼會碰到這麼多靈異現象。

***

什麼，你問 Teddy 有沒有嘗試第三種註冊方法？拜託，俗話說：「一顆蘋果只要咬一口就知道是不是爛蘋果」，Teddy 已經傻傻地咬了兩口了耶，鄉民們忍心看 Teddy 去咬第三口嗎？

奇怪，為何 Teddy 至今還是中華電信的客戶哩？

***

友藏內心獨白：真的是「恁爸閒閒」才會去撥打這個 0800 的電話。

4 HTC One X 之發現一個 Bug

左手邊為 Mio A701，右手邊為 HTC One X。

話說 Teddy 去中華電信預購了一隻白色的 HTC One X，前幾天收到通知說 4 月 11 號可以領，但是從 2012 年 4 月初 HTC One X 上市以來，在網路上看到不少災情，讓 Teddy 考慮了好久，最後才「鼓起勇氣」在 4 月 13 號下午去把手機領回家。

拿到手機之後發現一件很囧的事，因為 Teddy 的 SIM 卡還是侏羅紀時代的 2G SIM 卡，雖然當場中華電信給了 Teddy 一張 3G SIM 卡，但是中華電信的 SIM 卡開通作業一個禮拜只有一次，要等到下禮拜一才會開通。這幾天 Teddy 只能暫時當做拿到一支 4.7 吋的平板。

圖 4-1：沒有電話訊號的手機，只能當做平板使用

先接上家裡的 WiFi 上網並設定資料，玩著玩著雖然有發現幾個小問題，但是基本上感覺還不錯，不過其中有一個和錯誤處理有關的 bug 卡住了 Teddy 好幾分鐘。這個 bug 發生在設定聯絡

人照片的時候，Teddy 先將手機透過 USB 接上 Mac Book Air 筆電，然後把聯絡人的照片複製到 HTC One X 的照片目錄中。接著選擇修改聯絡人，然後選擇新增照片：

圖 4-2：聯絡人新增照片

此時 HTC One X 出現選擇照片目錄的畫面：

圖 4-3：選擇照片目錄

但是當 Teddy 選擇之後，手機畫面上卻沒有出現任何的照片，也沒有顯示錯誤訊息。

圖 4-4：按下選擇之後沒有出現任何照片

試了好幾次都不行，也看不出什麼原因。後來 Teddy 靈機一動，想說用拍照功能來新增照片總可以了吧。於是切換到相機功能，此時畫面上出現了一個錯誤訊息：「內容無法使用，因為正在使用手機作為 USB 儲存裝置。請中斷 USB 連接以存取內容。」

圖 4-5：相機功能顯示的錯誤訊息

原來當手機連到電腦時，如果是以 USB 模式連接，手機的內容便無法被存取。不知道這是所有 Android 手機的限制，還是只有 HTC One X 會這樣？但是重點是，這個錯誤訊息怎麼只有在相機功能才出現勒？聯絡人讀取照片時也要出現啊，看起來很像某個例外被程式設計師給吞掉了。

結論：HTC 的測試案例設計還有待加強，應該要多設計一些使用情境測試案例（scenario-based test case）。

***

友藏內心獨白：使用 3C 產品一定要搞得好像在訓練客戶都成為「柯南」嗎？

5 HTC One X 之我只是想打一通電話呀

在〈CH4：HTC One X 之發現一個 Bug〉提及設定聯絡人照片時所遭遇到的一點小麻煩，後來將問題排除之後也很順利的把聯絡人設定好(其實 Teddy 經常聯絡的人用一隻手就可以數完)。之後 Teddy 裝了 Viber 和 Line 試著打網路電話，沒甚麼大問題。實際跟 Kay 通話之後，發現 Viber 的通話品質與聲音延遲都比 Line 來的好，但是 Line 的訊息功能有比較多的小圖示可用。難道要講網路電話的時候用 Viber，然後傳訊息用 Line？

以上都不是重點，本章的重點是，Teddy 的 3G SIM 卡終於開通了，於是 Teddy 打算用手機撥個電話試看看通話品質。選擇了如圖 5-1 所示的「電話→群組」功能，結果哩？

圖 5-1：撥號群組畫面

圖 5-1 這個畫面停留大概不到一秒鐘，程式就當掉了，出現了圖 5-2 所顯示的訊息：「很抱歉！android.process.acore 未正常終止。是否傳送錯誤報告至 HTC？這將會協助我們改善產品」。喂，好不容易建立好聯絡人群組，想打個電話卻不讓人家選擇聯絡人。試了好多次都一樣：當、當、當、當，當、當、當、當。看到「android.process.acore 未正常終止」氣得 Teddy 想舉起心中的中指。好吧，反正當 3C 用品的白老鼠也不是第一次了，開始將腦袋切換到工程師的 debug（除錯）模式，當起「柯南」來玩一下推理的遊戲。

圖 5-2：撥號錯誤畫面

會不會是裝了什麼軟體去給它「衝到」，試著把 Viber 與 Line 移除之後外加重新開機，還是一樣。 這個問題 Teddy 會遇到，想必廣大的鄉民也都會遇到，google 了一下果然發現了很多人也遇到同樣的問題。後來看到屬名為「小布隆咚」的作者在 2010 年 9 月 26 日所發表的文章中提到，這個問題可能是因為「聯絡人資料同步異常」所造成的。拜託，2010 年的問題到現在居然還沒排除？暫時解決方法就是到手機的設定功能（圖 5-3），先取消「帳號與同步處理」的「自動同步」。

圖 5-3：帳號同步處理畫面

然後如圖 5-4 所示，在應用程式的手機設定功能中，選擇「應用程式」→「全部」，按下「搜尋」，輸入聯絡人作為搜尋條件便可找到下列幾支程式。

圖 5-4：手機設定功能

逐一點選這些程式，如圖 5-5 所示，按下「清除資料」。全部都完成之後再把手機「帳號與同步處理」的「自動同步」功能打開就這樣可以了。

圖 5-5：清除聯絡人資料畫面

***

雖然選擇群組功能程式不會再當掉了，但是之前 Teddy 建好的群組資料卻因為執行清除聯絡人資料的動作而遺失了，只好再手動建一次。手機用到這邊突然覺得有種當年在用 Windows 3.X 版的感覺，常常會遇到程式執行到一半出現「異常終止」的問題。

雖然聯絡人群組的問題「暫時」解決了，但是另一個更討厭的問題應該是無解。什麼問題？請看圖 5-6。

圖 5-6：HTC Sense 正在載入畫面

據說看到這個訊息就表示 HTC Sense 程式當掉之後正在「重啟（restart）」，但是 HTC 比微軟聰明也貼心太多了，怕嚇到消費者可能導致血壓升高，因此不會直接告訴你「HTC Sense 程式異常終止，請聯絡系統管理員」，而是婉轉的以「正在載入...」一筆帶過。「正在載入、正在載入」，哪來那麼多東西讓你一天到晚都正在載入，有需要那麼忙嗎？難道四核心就是拿來加速「正在載入」的速度之用嗎，重啟程式需要用到多核心與多執行緒嗎？HTC Sense 的體質未免太嬌弱了一點吧，一天到晚重啟的次數已經多到不想數了，是不是該餵它吃點維骨力和維他命 B 群了。

謎之音：這是功能，不是 bug。

***

手機三不五時就會出現「很抱歉！android.process.acore 未正常終止。是否傳送錯誤報告至 HTC？這將會協助我們改善產品」，或是「正在載入...」，這樣的「智慧手機」只能說一句「很難了解它的明白」。**難道付錢買 HTC 手機的目的是為了協助 HTC 改善產品？**難怪有人要說，Android 系統是給「專業人士」使用的，Teddy 這種業餘人士還是改用飛鴿傳書算了。

這些大公司不是都有很多所謂的「用戶體驗設計師」嗎？該不會他們只設計「正常情境（normal scenario）」的使用者經驗，而「異常情境（exceptional scenario）」的使用者經驗就不管啦。這

也沒關係，但是如果使用的過程中「異常情境」的頻率高到快變成「正常情境」，那就…誰理你啊，請頭上自備三條線外加一隻烏鴉伺候[4]。

用戶體驗設計師：此為正常現象，請安心服用。

鄉民甲：誰說這是手機，這是一台跑著 Linux 的行動電腦，只是剛好有撥打電話的功能而已。

***

友藏內心獨白：「噴電」的問題還沒說哩。

[4] 雖然 Teddy 的 HTC One X 隨著「新一（HTC New One）」上市而成了「舊衣」，還好 HTC 幫「舊衣」免費乾洗了好幾次（更新系統），目前使用狀況相對於剛上市而言已經穩定許多，避免了「舊衣」成為「就醫」的慘劇發生。在此「呼籲」鄉民們有機會還是要多支持一下國貨，政府官員都開口了，台灣的出口成長率就靠 HTC 了。

6 例外處理之錯誤訊息描述：HTC One X 拍照篇

Teddy 前幾天用 HTC One X 照相的時候，出現了類似「SD 卡[5]存取錯誤」的訊息。詳細錯誤敘述 Teddy 已經忘記了，也沒辦法把手機畫面抓下來…因為…SD 卡存取錯誤，沒辦法把畫面存起來啊。

看到這個訊息之後 Teddy 心想：「耶，手機買不到一年，怎麼這麼快 SD 卡就壞掉了？！」。江湖上不是傳言說：「HTC 品質，堅若磐石」嗎…啊，不對，這是另外一家公司的口號。可能是個人電腦用太久了，Teddy 想說把手機重新開機看看問題會不會自動消失。重開之後再試一次拍照功能，還是一樣出現同樣的錯誤訊息（李組長眉頭一皺，發現案情並不單純）。Teddy 觀察到，拍照時照片好像都有拍成功，但是最後卻沒有被存起來。於是 Teddy 猜想，會不會是 SD 卡的空間已滿？刪掉幾張舊照片之後，果然錯誤訊息不見了，照片也可能成功保存下來。

***

看完上面這則故事不知道鄉民們心中是否有著跟 Teddy 一樣疑問：為什麼 HTC One X 的拍照軟體不直接告訴使用者：「SD 卡空間已滿，若要繼續拍照請先清除 SD 卡空間」，而是顯示「SD 卡存取錯誤」這種很容易讓人誤會的訊息？Teddy 在〈CH4：HTC One X 之發現一個 Bug〉也提到過類似錯誤訊息表達不清楚的問題。本篇提到「SD 卡存取錯誤」這個錯誤訊息，Teddy 在「公堂之上大膽假設一下」，可能是因為拍照軟體將捕捉到的例外未加處理就直接顯示給使用者，請參考圖 6-1。

[5] SD 卡的全名是 Secure Digital Memory Card，中文翻譯為安全數位卡。是一種廣泛使用在數位相機與多媒體播放器上的記憶卡。

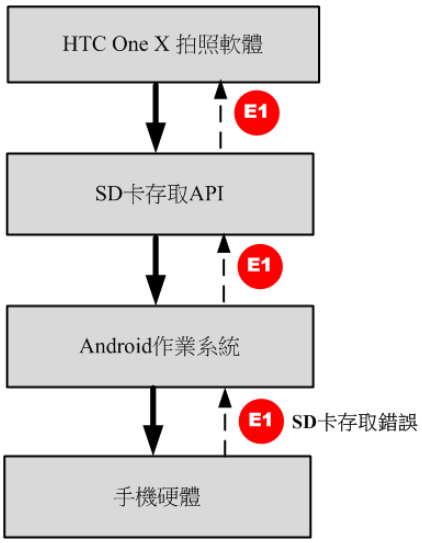


圖 6-1：Teddy 推測的 HTC One X 拍照軟體架構圖

在 SD 卡空間已滿的情況下，拍照軟體還要存入照片，於是手機硬體發出「SD 卡存取錯誤」的例外 `E1`，而拍照軟體可能有著類似下列 Java 程式結構：

```
1: function 拍照() {
2:    try{
3:         Picture p = takePicture();
4:         savePicture(p);
5:     } catch(Exception e){
6:         showErrMessage(e.toString());
7:    }
8: }
```

列表 6-1：Teddy 推測的 HTC One X 拍照軟體之例外處理程式虛擬碼

列表 6-1 第 5 行捕捉了 `Exception` 這個通用的例外型別，除了 `Error` 型別以外的例外不管是阿貓還是阿狗都會被第 5 行捕捉住。而這種類型的例外處理方式，最簡單的方法就是把捕捉到的例外直接顯示在畫面上，如第 6 行所示。

***

問題出在那裡

眼尖的鄉民們可能會說：「前面這段程式這樣寫並沒有錯啊，錯的是 Android 作業系統，為什麼不針對 SD 空間不足傳回一個 NotEnoughSDSpaceException，而卻傳回 SDAccessErrorException 這種這麼一般性的例外？」

沒錯，如果作業系統或是底層的 API 可以直接針對 SD 卡的操作傳回非常詳細的例外類別，API 的使用者就比較輕鬆，只要直接把錯誤訊息顯示在畫面上就可以了。但是由於存取 SD 卡的錯誤原因很多，如果要針對每一個原因都設計一個例外類別，系統中便會有太多的例外類別（例外類別的種類太多也是一個問題，需要更多的記憶體空間來存放這些例外）。像是 JDK（Java Development Kit）裡面，IOException 底下的子類別，也只有一個 FileNotFoundException 和檔案處理比較有關，其餘的檔案操作例外，大多直接使用 IOException 這個通用的例外類別來表示。

***

怎麼辦

身為一位繁忙的開發人員，正常功能都快沒時間寫了，哪有時間去管例外處理的問題。所以鄉民們當然可以正大光明地選擇馬照跑、舞照跳，維持現狀 50 年不變。如果真的要做到 Teddy 心中期待的例外處理方式，至少要顯示類似「SD 卡空間已滿，若要繼續拍照請先清除 SD 卡空間」這種對使用者而言比較有意義的訊息，程式可以改成如下所示（5~11 行為修改內容）：

```
1: function 拍照() {
2:     try{
3:           Picture p = takePicture();
4:           savePicture(p);
5:     } catch(SDAccessException e){
6:          String msg = null;
7:          if (SD is full)
8:              msg = "SD 卡空間已滿，若要繼續拍照請先清除 SD 卡空間";
9:          else
10:             msg = e.toString();
11:         showErrMessage(msg);
12:     } catch(Exception e){
13:         showErrMessage(e.toString());
```

```
14:     }
15: }
```

列表 6-2：讓 HTC One X 拍照軟體顯示 SD 卡空間已滿的錯誤訊息

***

很多時候，使用者認為的 bug，不一定真的是程式邏輯的錯誤，可能只是因為使用者不清楚程式內部運作的情境、邏輯、與前置條件。例如，雖然大家都知道要拍照一定要有足夠的儲存空間，否則拍完之後無法存檔，但使用者在拍照的時候並不一定會隨時注意到儲存空間是否足夠。在這種情況之下，**如果程式可以從使用者的使用情境來思考，當錯誤發生時儘量顯示對使用者有意義的訊息，讓他們有機會自行排除大部分的問題。**

***

友藏內心獨白：以上程式結構純屬紙上談兵。

7 可靠性可以是一門生意

Teddy 當年還在念研究所的時候，寫了幾篇例外處理的論文，在論文的序言章節，經常會有類似的開場白：

隨著電腦、網際網路、行動計算的普及應用，軟體系統已經深入人們的生活中。因此，軟體的可靠度也越來越獲得重視。軟體失效所引起的問題，輕則可能造成時間與金錢上的損失，重則可能危害生命安全。

可能是因為身受其害的緣故（請參考〈CH3：我真是猜不透你啊：種花篇〉、〈CH4：HTC One X 之發現一個 Bug〉、〈CH5：HTC One X 之我只是想打一通電話呀〉），Teddy 最近對於軟體可靠度重要性的感受越來越深刻。買了超酷超炫的智慧機手機或平板電腦，好高興啊。但是真正實際使用的時候，卻總是出現一些大大小小不等的問題。有些問題可能和作業系統與手機製造商有關，有些可能是第三方業者所提供的 APP 不穩因而「帶賽」整個系統。

這一陣子 Teddy 在思考是否要購買車用導航機。前一陣子 Kay 買了 iPhone 版的導航王，才幾百塊很便宜，但是實際上路使用，卻發現軟體反應很慢，常常車子已經開到某個路口，導航軟體卻顯示離路口還有一段距離。好吧，你可以怪 iPhone 4 的 CPU 太慢，記憶體太小，問題是這個導航軟體明明就宣稱可以在 iPhone 4 上面執行啊（當時 iPhone 4S 還沒上市，所以 iPhone 4 是當時 Apple 最高階的手機），總不能說：「只要軟體可以啟動就宣稱支援某個平台或設備吧？」至少功能要正常啊。導航軟體應該算是一種 soft real-time system[6]，對於 soft real-time system 而言，**時效**是很重要的。等過了某個路口之後導航軟體才告訴使用者要右轉，就已經來不及了。

[6] Real-time system（即時系統）是一種對於程式執行時序有著特殊要求的系統。這種系統可以分成 hard

後來買了 HTC One X 附贈一套導航王軟體，Teddy 想說 HTC One X 採用最新的四核處理器，應該可以跑得動導航王。好吧，就再給它一次機會。上禮拜天搭公車去老梅的時候，在公車上測試了一下，真的有比 iPhone 4 上面執行的版本要準確多了耶。當公車經過某個路口，導航軟體顯示馬路左邊有一家便利商店，Teddy 把頭往左一看，果真看到一家便利商店。看起來「好像」可以不用買專門的車用導航機了。

事情絕對不是 Teddy 想的那麼簡單，剛剛 Kay 在家裡試用了一下 Android 版本的導航王，用不到幾分鐘程式就卡在下面這個畫面一動也不動，按下手機的 Home 與上一頁按鈕也完全沒有反應（可是還可以擷取螢幕畫面）。此時 Teddy 心裡在想：「萬一在開車上路的時候用導航軟體用到一半程式當掉，一時緊張之下豈不是很危險？」

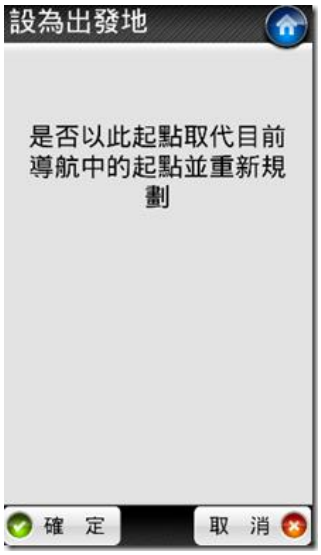


圖 7-1：導航王卡住畫面（Android 版本）

但是，購買車用導航機情況會比較好嗎？上面也是有執行導航軟體啊，很想拜託台灣的廠商可以把軟體的可靠度大大提升一下嗎？在寫這篇文章的時候 Teddy 想到之前指導教授在論文中所寫的一段話：

real-time 與 soft real-time。在前者中，程式錯過規定執行期限是不可容忍的，可能導致生命或財產的嚴重損失。例如，武器控制系統或核電廠控制系統。在後者中，程式執行錯過規定時效是可以被容忍的，但是會造成系統服務品質降低，或是產生沒有意義的執行結果。

Users generally demand good software quality in the long term. Occasional failures may be tolerated by the users while novelty lasts, inadequate exception handling design can eventually turn them away to competitors.

長期而言，使用者期待高品質的軟體。當新鮮感還在的時候，使用者可能會容忍偶然的軟體失效，但是不足的例外處理設計（導致軟體不可靠）終將把使用者推向競爭者的懷抱。

Teddy 覺得上面這段英文句子寫得真好，但是只有一個問題：「如果大家一起擺爛，每一家產品問題都很多，那不就沒得選啦？」

所以，有錢的大公司做產品要長長久久，不能只看到「功能面需求」。時至今日，當各家產品的很多功能都是「你有我也有」的時候，像可靠性、強健度這種非功能需求就很有可能可以讓自己的產品脫穎而出。

***

友藏內心獨白：有時候真的很懷疑，這些公司的老闆真的有在用自己的產品嗎？

Column A. | 帶賽的人很適合當神秘客

北京故宮

不知道是因為八字太輕還是天生「帶賽」，Teddy 經常遇到一些別人遇不到的倒楣事情。例如，2013 年 10 月到北京故宮參觀，Teddy 和 Kay 各借了一台語音導覽器，Teddy 拿到的這個導覽器用到一半就沒電了，後面的景點介紹都沒聽到。離開故宮之前退還語音導覽器的時候，跟工作人員反應語音導覽器有問題，故意問對方…

Teddy：語音導覽器沒電了，我有一半的景點都沒聽到，可以退費嗎？

服務人員 A 內心獨白：這種要求我這輩子從來沒聽過。

服務人員 A：（苦笑）你要去問一下我右手邊小房間裡面的人。

Teddy：（來到小房間前面敲門）。

服務人員 B：（正在屋內吃飯）走出小房間。

Teddy：請問這個語音導覽器在我聽到一半的時候就沒電了，可以退費嗎？

服務人員 B：（檢查一下語音導覽器）在某某處與某某處有服務櫃台，你可以去換一個導覽器。

Teddy：導覽器沒電的時候，我離這兩個服務櫃檯都很遠。我現在已經參觀完畢，要準備離開了，也不可能再拿去退換。既然租了這個導覽器只聽了一半的景點介紹，是不是可以退費？

服務人員 B：（苦笑）…

***

其實 Teddy 不是真的想去跟對方要回導覽器的租金（租一次 20 元人民幣，當然如果對方願意退還一半費用也是很好），主要希望對方可以注意到導覽器可能會沒電的問題。這個導覽器設計的還不錯，但就是沒有剩餘電量顯示功能。**只能說 Teddy 太幸運了，剛好拿到「機王」**。

***

台北客服

回台北之後的第一天（2013/10/21），打了三通不同的客服電話。首先是中華電信，家裡的網路在 Teddy 出國之後的 2~3 天就斷線了，至今也已過了 6~7 天了。一整天一共打了三通 0800-080-128 的電話給中華電信的客服中心，第一通是在凌晨 12 點之後打的，對方說會請查修人員回電。睡醒之後等到下午 1 點左右都沒人打電話來，又撥了一次 0800-080-128，對方在 Teddy **抗議**之下表示他會主動幫 Teddy 跟查修人員詢問，然後再回電給 Teddy。又等了兩個小時，根本沒人回電。這次真的很火大，第三次撥打 0800-080-128，對方在 Teddy 的**嚴重抗議**之下表示會把 Teddy 的問題反應給「**二線人員**」。

沒過多久這次終於有「二線人員」打電話過來，感覺好像只有「二線人員」是中華電信的員工，客服中心的人不知道是不是外包人員，除了接電話、記錄問題、向上回報、挨罵以外，幾乎沒什麼功能。

為了報修網路並得知維修的時間，從頭到尾跟四個不同的人通電話，幾乎搞了一整個工作天，Teddy 從和顏悅色講到破口大罵，最後才問到中華電信隔天會派人來查看的回答。

結果隔天維修人員來到 Teddy 家裡，發現在 Teddy 出國的時候，中華電信剛好「自動」幫 Teddy 的網路升速，應該是設定沒弄好所以網路不通。當場立刻通知機房人員重新設定，前後不到 30 秒網路就通了。看到這個情況 Teddy 更是火大，因為之前在電話中 Teddy 就已經跟客服中心的人提到，會不會是因為網路自動升速設定的問題導致斷線，但客服就是沒反應，堅持問題一定要交給查修人員到現場處理。

***

第二通電話是撥給中國信託，兩週前 Teddy 用中國信託的 AE 卡（美國運通卡）在 Google Play 上面買了一個軟體，可是不知為什麼卻購買失敗，而且還收到一封 29 元交易記錄的通知信。

消費日	消費金額	商店名稱
2013/10/09 18:34	$29 元	暫無商店資訊

後來又嘗試用 AE 卡在 Paypal 付款買一個短期的 VPN 服務，也是交易失敗。於是打電話去詢問這張 AE 卡除了在 Costco 以外的地方到底能不能用，中國信託的客服人員跟鬼打牆一樣，一直重複說：「**這張 AE 卡只要商家有支援就可以刷**」，這不是廢話嗎，就是不知道中國信託除了 Costco 以外有沒有跟其他商家談好可以使用這張 AE 卡，才打電話去詢問。對方完全無視於 Teddy 跟他們反應的問題（Google Play 與 Paypal 雙雙刷卡失敗）。Teddy 最後只好說：「**我了解了，我會把你們的卡剪掉，去辦另外一家銀行的卡**」。

***

最後，打去花旗銀行反應他們的信用卡機場接送服務問題。花旗銀行換了新的機場接送服務廠商之後，新廠商的司機根本不熟台北的道路（對方司機自爆自己公司是台中的廠商），而且車

輛非常老舊。**從北京回台灣當天晚上來桃園機場接送的司機，希望 Teddy 能夠跟他聊天，以免他睡著**。司機表示：「因為被公司安排接送太多客人，沒時間休息，非常想睡覺。」接著司機又說：「這輛車應該要去保養了，但是因為生意太好公司一直沒有安排保養的時間（前車門還撞凹了一塊）」。Teddy 聽了真的是...**忍不住舉起了心中的中指**。自己都快累死了，但為了自己和 Kay 的生命安全，還是強打起精神一路陪著司機聊天回台北，等到安全回到家裡才鬆了一口氣。

雖然 Teddy 對於花旗銀行為了減省成本而換了一家很爛的機場接送公司非常不滿，**這根本是拿卡友的性命開玩笑**，但花旗銀行客服人員的訓練與應對就比中華電信與中國信託好很多。雖然客服人員不可能當場給出什麼承諾，但在電話中客服人員有彙整 Teddy 跟他們抱怨的問題，然後表示會跟活動組同仁反應這個問題，並一再致歉。至少不像其他家的客服，**像是在念經一樣一直重複相同的制式說詞**。難道因為是外商所以客服比較有制度嗎？

***

結論就是：

- Teddy 真的很「帶賽」。
- 中華電信的客服制度很差，可能是為了省成本把整個產品與服務的價值鏈（value chain）切割成好幾段，然後外包給不同的廠商。每一段單獨的服務人員並沒有足夠的權限來處理問題，也失去了服務熱情。要不是因為是獨占企業，應該早就倒閉了。
- 中國信託客服人員可能有接受過繞口令的訓練，講不過他！
- 希望下次出國的時候，花旗銀行的信用卡機場免費接送可以不要再讓客戶有「玩命」的感覺，不然到時候又要再寫一篇文章，也是挺累人的。

***

友藏內心獨白：這麼大的公司，每年賺那麼多錢，服務做得好一點，很難嗎？

第二部 例外處理基本觀念

8 強健性大戰首部曲：威脅潛伏

尤達大師與黑武士

Teddy 非常喜歡「星際大戰」系列的電影與影集。還記得在《星際大戰首部曲：威脅潛伏》電影裡面，黑暗勢力想藉著那卜星（Naboo）的貿易糾紛製造紛端，企圖推翻共和國。絕地議會派出絕地大師魁剛金和歐比王前往調停，經過一番風風雨雨之後，最後黑暗勢力失敗，西斯武士達斯魔被歐比王殺死，那卜星重新恢復自由。

此次事件之後，尤達大師與絕地議會雖然意識到西斯大帝的存在，但在往後幾集裡面，終究沒有發現原來西斯大帝偽裝成議員白卜庭，最後並當選共和國議長。當上議長之後，權力更大，一天到晚想東想西，從中喬事情、關說、暗地搞破壞，最後導致民國，嗯嗯…應該是共和國滅亡，而絕地武士也因此幾乎消失殆盡，剩下小貓兩三隻。如果絕地武士們可以早點發現威脅共和國安全的根本原因，將其揪出並加以消滅，也許就可以避免這些慘劇（迷之音：傻孩子，那就沒有續集可以拍了啊！）

***

鏡頭回到軟體開發，想要讓系統達到「萬歲、萬歲、萬萬歲」的強健度，就必須要知道潛伏在系統之中，威脅系統強健度的這些「壞蛋」的身分（西斯大帝、西斯武士、黑武士）與可能的分身（共和國議長、天行者）。這一章的目的就是要把這些影響強健度的「壞蛋」揪出來，這些壞蛋包含：**fault（缺陷）、 error （錯誤）、failure（失效、失敗）和 exception（例外、異常）**，都是在討論軟體強健度與例外處理時經常會看到的幾個名詞。這些名詞看起來很像，而

且有時也被當成同義字交互使用。然而事實上它們卻有著不同的涵義。本章主要參考 Lee 與 Anderson 的著作[7]以及 Avizienis 等人的論文[8]，經整理、吸收、消化、反芻、再整理之後，所呈現的結果。

本章內容非常重要，瞭解這些名詞的定義可為日後例外處理設計打下堅實的基礎，在後續章節中也會經常使用這些名詞，請鄉民們花點心思仔細閱讀（否則下場可能會跟絕地武士一樣）。

當一個軟體元件（function、method、或 service）所提供的服務與其**功能規格 (functional specification)**相符，我們說這個元件提供**正確服務（correct service）**。如果軟體元件所提供的服務偏離正確服務，我們稱為**服務失效（service failure）**，簡稱**失效（failure）**。例如：

- `int add(int first, int second)` 這個函數，傳入 1 和 5 這兩個參數。在正常情況下應該要回傳 6，如果傳回 6 以外的任何答案，都算是這個函數失效。
- 一個網路購物系統的線上扣款動作必須在 60 秒之內完成，如果超過 60 秒尚未完成則視為扣款功能失效。
- 一個客戶關係管理系統要連到後端資料庫讀取使用者的登入帳號與密碼，但是網路不穩造成連線斷斷續續，最終導致使用者登入功能失效。

Error 指的是軟體元件內部處於錯誤狀態（erroneous state），此狀態可能會使得該元件執行失敗，因此導致 failure。反之，如果軟體元件可以將錯誤狀態加以修復，回到一個沒有錯誤的正確狀態，則有可能避免發生 failure。

Fault 是假設、或是經過判斷之後所確認導致 error 發生的原因。Fault 依據其**存在時間長短**可分為以下三類：

- 暫態缺陷（transient fault）：出現一次之後就消失，如果重新執行一次失效的操作則會正常執行。例如，有一隻「憤怒鳥」飛過兩棟大樓中間，阻擋了大樓採用雷射傳送的網路訊號。
- 間歇缺陷（intermittent fault）：出現之後消失，然後又重複出現。這種情況經常發生在鬆脫的接頭或是老化的硬體設備，導致訊號或功能時好時壞。也有可能是軟體元件產生資源

[7] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice, 2nd ed.*, Springer, 1990.
[8] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, 2004.

洩漏（resource leak），問題發生之後只要將應用程式重新啟動或是系統重開機便消失，但是系統使用一陣子之後問題又會重新出現。

- 永久缺陷（permanent fault）：發生之後除非將有問題的元件換掉，否則元件會一直處在 failure 狀態。例如，燒斷的保險絲、被漁船拉斷的海底光纖電纜、軟體的 bug。

***

Fault 依據**產生原因**可區分為兩大類：

- **設計缺陷（design fault）**：又稱為 **defect、bug、programming error**，是人們在軟體開發過程中所犯的問題。例如，誤解使用者需求、類別介面設計錯誤、忘記初始化物件、除數為零、條件判斷式中的大於寫成了大於或等於、演算法設計錯誤等。Design fault 屬於永久缺陷，除非將 fault 排除，否則系統將無法正常運作。例如，函數中的邏輯錯誤，把大於等於寫成大於，就屬於永久缺陷。除非有開發人員去修改這個函數，或是改用另一個沒有問題的函數來取代，否則這個 design fault 不會自己消失。就算是重複執行 82 億次這個有問題的函數，design fault 還是存在。
- **元件缺陷（component fault）**：原本是指硬體元件可能因為老化或受使用環境影響等因素所產生的問題，在此 Teddy 將其引申為因為硬體元件失效所產生的錯誤，或是軟體元件與軟體元件、軟體元件與執行環境、軟體元件與硬體環境，彼此互動時所產生的不正常情況。例如，一個網路線的接頭突然鬆脫、儲存資料時發現硬碟空間不足、列印檔案時印表機未開啟、無法存取遠端信用卡交易服務等，都屬於此類。Component fault 有可能是暫態缺陷、間歇缺陷、或永久缺陷，因此針對由 component fault 所造成的系統 failure，也存在不同的處理策略。針對暫態缺陷，只要重試原本的操作，系統即可恢復正常。如果是間歇缺陷與永久缺陷，就要先釐清缺陷發生原因，單純重試失效的操作並無法徹底解決問題。

在 Avizienis 等人的論文中，則將 fault 依據其發生原因分成另外三大類：

- 開發缺陷（development fault）：在開發過程中所注入在產品裡面的缺陷，例如需求錯誤、介面設計錯誤、程式撰寫錯誤等。
- 實體缺陷（physical fault）：因硬體設備而受影響的缺陷，例如自然因素（打雷、高溫、低溫、自然老化）造成的缺陷、在製造生產過程中所產生的缺陷。

- 互動缺陷（interaction fault）：由外部互動所引起的問題，例如使用者操作錯誤（輸入不正確的資料或惡意破壞）、錯誤的系統設定（例如網路設定錯誤導致無法上網）。

Fault 分類本身就是一門學問，好比疾病一樣，能夠區分不同類型的疾病，藥廠才可以針對這些疾病開發藥品，而醫生也才能夠對症下藥。知道系統可能會以怎樣的原因（fault）導致失效，就有可能在設計階段加以考慮並且預防。例如，如果系統因為邏輯錯誤（design fault）導致當機，就應該要提高開發人員的素質，或是採用程式碼檢閱（code review）、撰寫單元測試、導入自動程式碼檢查等手段來移除這些邏輯錯誤。如果是因為使用者輸入不合法的資料導致系統出問題，就應該在使用者介面元件設計嚴謹的資料檢查邏輯，以防止不正確的資料進入系統。

針對 fault 發生原因，本書採取區分 design fault 與 component fault 的觀點，前者相當於 development fault，後者包含了 physical fault 與 interaction fault。更簡單一點的想法：design fault 就是程式設計不良，有 bug 需要開發人員動手修正；component fault 就是個別元件的設計與實作都沒有問題（不存在 design fault），但是元件互動之後的結果產生了問題，可以在程式中透過例外處理方式來加以修復。

***

就好像火山有分活火山與休眠火山一樣，fault 也有分**活躍缺陷（active fault）**與**休眠缺陷（dormant fault）**。前者會造成 error，後者則因為沒有被引爆所以暫時不會造成 error。Fault 也好比地雷，沒被踩到就不會爆炸。在開發軟體的時候除非妥善處理 fault，否則一旦引爆地雷，最終將導致 error，進一步使得軟體的執行產生 failure 而無法滿足使用者的需求。這個 failure 又會成為呼叫該服務的人的 fault，一直循環下去一直到系統外部的使用者觀察到該系統無法提供正常功能或是當機為止。請參考圖 8-1。

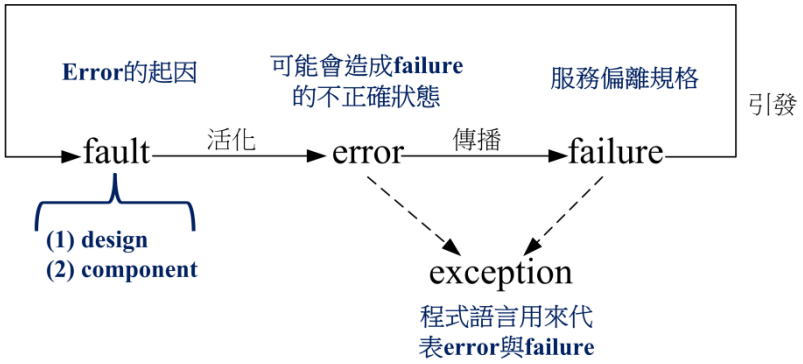


圖 8-1：Fault、error、failure 與 exception 的關係

***

Exception 是程式語言中用來表達 error 與 failure 的一種概念。參考圖 8-2，假設鄉民們寫了一個 `writeFile` 函數，在實作該函數的時候偵測到可能會發生 `IOException`。此時的 `IOException` 表示一個 error，也就是說，`writeFile` 可能正處於某種錯誤狀態。如果鄉民們不作任何處理，最後導致 `IOException` 傳遞給 `writeFile` 的呼叫者，從呼叫者的角度來看，此時的 `IOException` 就代表了呼叫 `writeFile` 的結果為 failure。因為這個 failure，而導致呼叫者產生 error。**請注意，同樣一個 `IOException`，在 `writeFile` 內部與外部分別代表著 error 與 faulire 兩種不同的意義。**

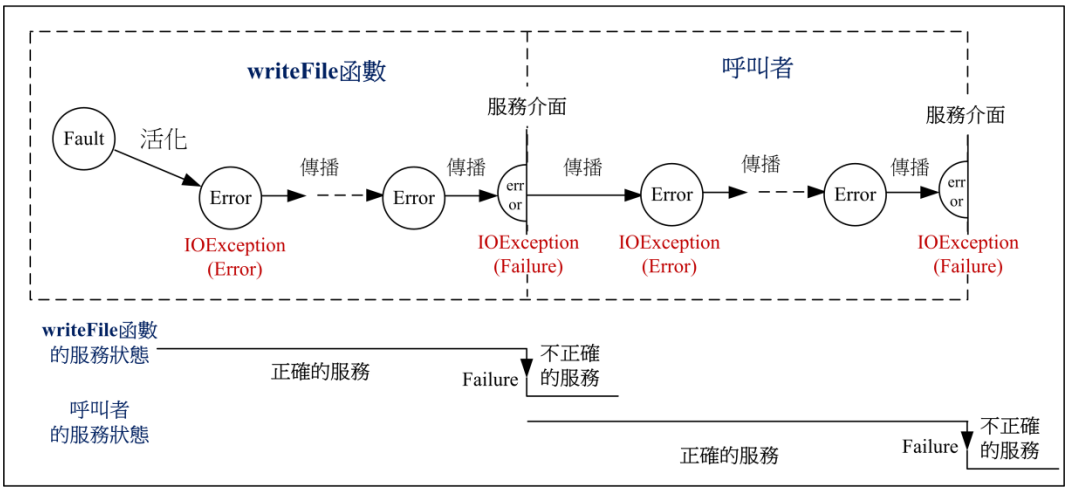


圖 8-2：Error 與 failure 的傳遞關係

***

友藏內心獨白：Fault、Error、Failure，怎麼覺得好像在繞口令啊。

Column B. | 找不到資料要傳回 Null 還是丟出 Exception？

2013 年 11 月某一天 Teddy 到某大學資工系演講，講題是「例外處理設計與重構」。演講結束之後有一位同學問了 Teddy 一個很常見也很有趣的問題。

同學：你剛剛建議我們不要用回傳碼（return code）來代表例外狀況，所以如果我要設計一個依據學號到資料庫中查詢學生資料的函數，當找不到符合條件的學生資料的時候，是不是應該要丟出例外？

Teddy：嗯，不一定，要看你的需求與設計。假設你的函數長成這樣：

```
public List<Student> queryStudents(String ID)
```

只要傳回一個大小為 0 的 List 就可以代表找不到學生資料，不需要丟出例外。

同學：那如果我的函數只會回傳一個學生物件呢？

Teddy：假設你的函數長成這樣：

```
public Student queryStudent(String ID)
```

只傳回一個物件，那麼你就要問自己：**「依據 ID 來尋找學生資料，但最後找不到任何一筆符合條件的資料，這種情況算是正常狀況還是異常狀況？」**

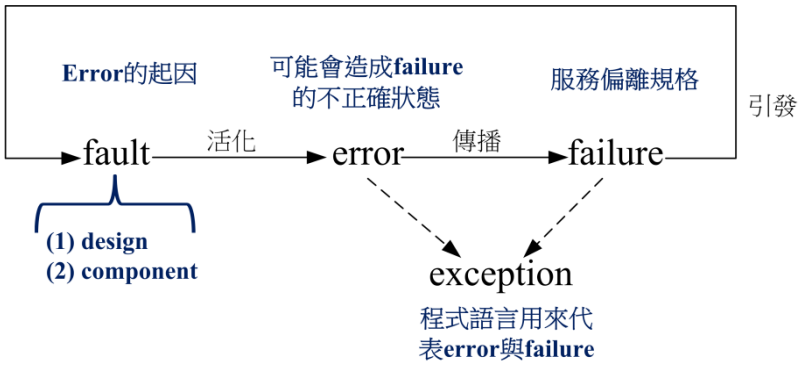
同學：…

Teddy：如果你問我的看法，我會覺得這是正常狀況。在日常生活中，依據某些條件去找資料，最後找不到任何符合條件的資料，算是一種很常見的狀況，所以我不會針對這種情況丟出例外。

你可以傳回 null 用來代表找不到任何資料，如果不喜歡 null，也可以套用 *Null Object*[9]設計模式，傳回一個 *Null Object* 來表示找不到任何一筆符合條件的資料。以上這兩種做法都比傳回例外要來的好。

***

同學所問的問題，可以〈CH8：強健性大戰首部曲：威脅潛伏〉的圖 8-1 來解釋（重繪如下）。**Exception 在程式語言中用來代表 error 與 failure，分別表示「目前可能處在不正確的狀態」與「被呼叫的函數辦事不力」。**「找不到資料」並非狀態錯誤，也不是尋找資料的函數辦事不力。事實上，尋找資料的函數正確執行完畢，只不過沒有找到符合查詢條件的資料罷了。這種狀況是在規格中所允許的狀況，和例外處理無關，只要在設計函數介面的時候規定好傳回某種特殊值（null 或 *Null Object*）代表找不到資料這樣就可以了。



***

友藏內心獨白：所以說了解 fault、error、failure 的定義對於例外處理設計是很有幫助的。

[9] 可參考維基百科的說明 http://en.wikipedia.org/wiki/Null_Object_pattern，或 R. C. Martin, D. Riehle, and F. Buschmann, *Pattern Languages of Program Design 3*, Addison-Wesley Professional, 1997.

9 例外處理的四種脈絡

地圖與現在位置，對於旅行者而言就是一種 context

Context 這個字，在軟體開發這個領域中用得非常多。根據它在文章中出現的用法，大致可以翻譯成「脈絡」、「情境」、「上下文」、「環境」等。本章要介紹例外處理的四種不同 context：**例外脈絡（exception context）**、**局部脈絡（local context）**、**物件脈絡（object context）**、**架構脈絡（architecture context）**。

***

Exception Context

Exception context 是當例外發生的時候，**例外處理程序（exception handler，簡稱 handler）**可以從**例外物件**所獲得的相關資訊。這些資訊可以透過程式語言執行環境傳入，例如產生例外的函數名稱、產生例外的程式碼行號、例外產生當時的函數呼叫堆疊軌跡（stack trace）。

請參考列表 9-1 的 Java 程式範例，執行 `main` 函數會讓 Java 虛擬機器丟出一個 `RuntimeException`。

```
 1: package tw.teddysoft.exception.context;
 2: public class ContextExample {
 3:     public void methodA(){
 4:         methodB();
 5:     }
 6:     public void methodB(){
 7:         methodC();
 8:     }
 9:     public void methodC(){
10:         throw new RuntimeException();
11:     }
12:     public static void main(String [] args){
13:         try{
14:             ContextExample example = new ContextExample();
15:             example.methodA();
16:         } catch(Exception e){
17:             e.printStackTrace();
18:         }
19:     }
20: }
```

列表 9-1：例外脈絡範例

第 17 行 `e.printStackTrace()` 會印出由 Java 虛擬機器所準備好的 exception context，如圖 0-1 所示，可從中得知 `main` 呼叫了 `methodA`，然後 `methodA` 呼叫了 `methodB`，然後 `methodB` 呼叫了 `methodC`。最後，由 `methodC` 丟出了一個 `RuntimeException`。

```
java.lang.RuntimeException
        at tw.teddysoft.exception.context.ContextExample.methodC(ContextExample.java:10)
        at tw.teddysoft.exception.context.ContextExample.methodB(ContextExample.java:7)
        at tw.teddysoft.exception.context.ContextExample.methodA(ContextExample.java:4)
        at tw.teddysoft.exception.context.ContextExample.main(ContextExample.java:15)
```

圖 0-1：執行列表 9-1 的結果，印出例外發生時的堆疊軌跡

***

Exception context 也可以由**例外產生者（signaler）**傳入，例如在 Java 語言中可以透過**例外串接（exception chaining）**的方法，把產生某個例外的原因（cause，另外一個例外物件）一併傳給例外處理程序。如列表 9-2 程式碼所示，第 11~13 行的例外處理程式，當捕捉到 `IOException`

之後，產生一個新的 NotEnoughDiskSpaceException，並且把捕捉到的 IOException 當成參數傳給 NotEnoughDiskSpaceException 的建構函數（constructor）。

```
 1: package tw.teddysoft.exception.context;
 2: import java.io.IOException;
 3: public class ExceptionChaining {
 4:     public void methodA() throws IOException {
 5:         throw new IOException();
 6:     }
 7:     public void methodB() throws NotEnoughDiskSpaceException {
 8:         try{
 9:             methodA();
10:         }
11:         catch(IOException e){
12:             throw new NotEnoughDiskSpaceException(e);
13:         }
14:     }
15:     public static void main(String [] args){
16:         try{
17:             ExceptionChaining ec = new ExceptionChaining();
18:             ec.methodB();
19:         }
20:         catch(NotEnoughDiskSpaceException e){
21:             e.printStackTrace();
22:         }
23:     }
24: }
```

列表 9-2：例外串接範例

執行 main 函數的結果如圖 9-2 所示，可以看到堆疊軌跡多了一個 Caused by 的資訊，顯示造成 NotEnoughDiskSpaceException 的原因是 IOException，而最早產生 IOException 的地方是在程式第 5 行。

```
tw.teddysoft.exception.context.NotEnoughDiskSpaceException: java.io.IOException
        at tw.teddysoft.exception.context.ExceptionChaining.methodB(ExceptionChaining.java:12)
        at tw.teddysoft.exception.context.ExceptionChaining.main(ExceptionChaining.java:18)
Caused by: java.io.IOException
        at tw.teddysoft.exception.context.ExceptionChaining.methodA(ExceptionChaining.java:5)
        at tw.teddysoft.exception.context.ExceptionChaining.methodB(ExceptionChaining.java:9)
        ... 1 more
```

圖 9-2：執行列表 9-2 的結果，印出例外發生時的堆疊軌跡。

***

Java 語言預設的 `Throwable` 類別是所有例外類別的父類別，看一下 `Throwable` 的建構函數：

- `Throwable()`
- `Throwable(String message)`
- `Throwable(String message, Throwable cause)`
- `Throwable(Throwable cause)`

最多只允許例外產生者自行傳入一個字串和一個例外物件。**也就是說，在 Java 語言中除非設計自己的例外物件，否則在預設的情況下例外產生者可以透過 exception context 傳遞給例外處理程序的資訊很有限。**

微軟.NET 框架的例外類別因為支援了 *Variable State* 這個實作模式，使得例外產生者可以透過 exception context 傳遞任意資料給例外處理程序[10]。

***

Object Context

在物件導向程式語言中，object context 表示在一個物件裡面可以存取到的資訊，包含了物件的資料成員（data member）以及全域變數。

在介紹 exception context 時提到，例外處理程式可以透過 exception context 來獲得一些例外處理的資訊。同樣地，也可以從 object context 獲得例外處理的資訊。請參考列表 9-3，`ObjectContext` 物件有六個資料成員，這些資料都有可能在例外處理程式中被使用。另外，第 28~30 行的例外處理程序，可以把 `SQLException` 發生的時候，使用者所使用的資料庫種類、帳號、資料庫名稱、資料庫連線的連接埠號碼（port number）等資料寫到日誌檔裡面。

[10] 請參考 Teddy 的部落格文章：〈Implementation Patterns: Variable State〉，http://teddy-chen-tw.blogspot.tw/2013/10/implementation-patterns-variable-state.html。

```
1: package tw.teddysoft.exception.context;
2: import java.sql.Connection;
3: import java.sql.DriverManager;
4: import java.sql.SQLException;
5: import java.util.Properties;
6:
7: public class ObjectContext {
8:    private Connection conn;
9:    private String userName;
10:    private String password;
11:    private String dbms;
12:    private String serverName;
13:    private int portNumber;
14:    public void connect() throws SQLException {
15:        Properties connectionProps = new Properties();
16:        connectionProps.put("user", userName);
17:        connectionProps.put("password", password);
18:        conn = DriverManager.getConnection(
19:                   "jdbc:" + this.dbms + "://" +
20:                   this.serverName +
21:                   ":" + this.portNumber + "/",
22:                   connectionProps);
23:    }
24:    public void readData(){
25:        try {
26:           connect();
27:           // do something
28:        } catch (SQLException e) {
29:          // handle the exception may need the object context
30:        }
31:        finally{
32:           close(conn);
33:        }
34:    }
35:    public ObjectContext(String dbms, String userName, String password){
36:        this.dbms = dbms;
37:        this.userName = userName;
```

```
38:         this.password = password;
39:     }
40:     private void close(Connection conn){
41:         if(null != conn)
42:             try {
43:                 conn.close();
44:             } catch (Exception e) {
45:                 // log the exception
46:             }
47:     }
48: }
```

列表 9-3：Object context 範例

***

Local Context

廣義的來講，任何程式區塊所形成的一個範圍（scope）便可算是一個 local context。在本書中以一個函數當做一個 local context 的範圍（因為很多程式語言的主要例外處理程序都是位於函數裡面）。

以列表 9-4 程式片段為例，`readMessageBody` 函數所傳入的兩個參數、函數裡面的區域變數、以及實作程式碼就形成了例外處理所關心的 local context。

```
1: public byte [] readMessageBody(int aLength, DataInputStream aIS)
2:         throws InvalidPacketException {
3:     try {
4:         byte[] messageBody = new byte[aLength];
5:         aIS.readFully(messageBody); // may throw EOFException
6:         return messageBody;
7:     } catch (EOFException e) {
8:         throw new InvalidPacketException("Data Underflow.", e);
9:     } catch (IOException e) {
10:         throw new InvalidPacketException("Read data body error.", e);
11:     }
12: }
```

列表 9-4：Local context 範例

再看列表 9-5 的例子，為什麼 dumpFileContent 函數和列表 9-4 的 readMessageBody 函數都遇到 EOFException，但是兩者的處理方式卻不同？如果光是從 exception context，鄉民們只知道捕捉到 EOFException 這種型別的例外，以及該例外的堆疊軌跡等資料，對於判斷這個例外要如何處理沒什麼幫助。因此必須再往外尋找，看看在 local context 或是 object context 裡面，能不能找到如何處理例外的蛛絲馬跡。

```
1: public void dumpFileContent(String aFileName) throws IOException{
2:     try (DataInputStream input = new DataInputStream
3:             (new FileInputStream(aFileName))) {
4:       while (true) {
5:         System.out.print(input.readChar());
6:       }
7:     }
8:     catch (EOFException e) {
9:       // 忽略, 在這個 context 之下，此為正常現象
10:    }
11: }
```

列表 9-5：相同的例外，因為 local context 不同形成不同的處理方式

從 dumpFileContent 函數和 readMessageBody 函數的實作程式碼，可以發現在 dumpFileContent 函數中，EOFException 被作為一種**狀態通知**的用途，表示已經讀到檔案結尾的這種狀態。因為 dumpFileContent 函數是用來印出檔案的內容，因此讀到檔案結尾之後丟出 EOFException 實屬正常現象，請安心服用，不須特別處理。

但是 readMessageBody 函數就不一樣，第 5 行 aIS.readFully(messageBody) 預期從 aIS 物件讀取一定長度的資料，如果此時發生 EOFException，則表示**資料長度不足**，因此在第 8 行丟出一個 InvalidPacketException("Data Underflow") 來表示資料格式錯誤這個例外情況。

以上兩個不同的 local context 例子，也引導出兩種不同的例外處理設計方法（什麼事都不用作，以及丟出另外一個錯誤語意比較清楚的 InvalidPacketException）。

***

Architecture Context

有時候就算是依據 exception context、object context、local context，還是無法決定例外處理的設計方法。這時候就需要擴大範圍，參考 architecture context[11]，從軟體架構的角度，來評估例外處理策略。

***

Teddy ：請問收到一個 `IOException` 要如何處理？

鄉民甲：不知道，這種沒頭沒尾的問題要怎麼回答？

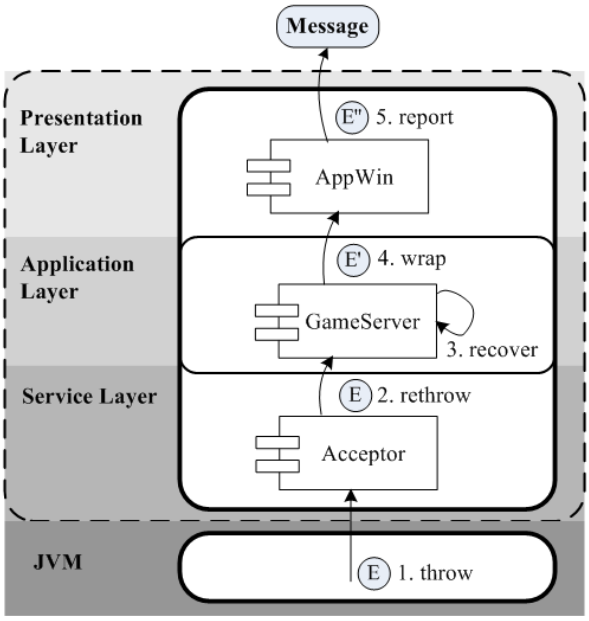


圖 9-3：參考 architecture context 決定例外處理策略。

沒頭沒尾的問題的確無法回答，請參考圖 9-3，這是一個超級簡化版的網路遊戲軟體架構，其中有三個主要的元件：

- `Acceptor`：程式啟動之後 Acceptor 會監聽某一個 TCP port，負責接受來自於網路的連線。

[11] 又稱為 application context，應用程式脈絡。

- GameServer：負責遊戲的邏輯。
- AppWin：負責顯示遊戲畫面。

假設當 Acceptor 要監聽某個 port 但卻發現這個 port 已經被其他人給使用了，因此接收到一個 IOException，請問 Acceptor 該如何處理這個例外？從 exception context，可以知道這個 IOException 代表某個 TCP port 被使用，再從 local context 與 object context，可以知道 Acceptor 所負擔的責任就是要去監聽某一個 TCP port。

***

鄉民甲：我知道了，如果原本的 port 被別人使用，那 Acceptor 就使用其他的 port 就好了，這樣上層的 GameServer 就不會被這個例外所影響，程式還是可以正常運作。

Teddy：讓 Acceptor 自行嘗試其他的 port 的確是一種方法，但是如果原先的 port 是對外公開的號碼，如果隨便自行修改別人可能會無法和你溝通那怎麼辦？又或者，原本的 port 被占用可能代表還有另外一個相同應用程式還在執行當中，如果 Acceptor 自行挑選其他的 port 然後假裝沒有任何例外發生，那麼使用者開始玩遊戲之後可能會遭遇到一些奇怪的現象，例如有些封包被另外一個還在執行的應用程式給收走。

鄉民甲：那，你的意思是說 Acceptor 不管遇到什麼例外都一律往上丟？

Teddy：也不是這個意思，從 architecture context 的角度來看，Acceptor 位於應用程式的底層。**通常越底層的元件，被重複使用的機會相對越高。如果這樣的元件當初設計的時候就針對某一個特定的應用情境做了例外處理的最佳化，則該元件在其他情境裡面被重複使用的機會就會減低。**換句話說，Acceptor 的使用會被綁死在這個網路遊戲軟體裡面。

Teddy：再者，**越底層的元件通常沒有足夠的 architecture context，不知道自己將來會被用在那些應用情境之中。**因此，通常也沒有足夠的全域知識來做好例外處理的設計。

鄉民甲：所以說，如果底層所擁有的 architecture context 不足夠，那麼就應該把例外狀況向外回報，讓上層的人有機會可以處理？

Teddy：基本上的精神是這樣的。就好像基層員工遇到例外狀況無法應付，就應該向上層主管回報，讓擁有更高權限的主管來處理。

鄉民甲：那在這個例子裡面，這個 `IOException` 應該交給 `GameServer` 還是 `AppWin` 來處理？

Teddy：這還是要看軟體架構設計與各層責任分工來決定，以這個例子來看，`GameServer` 應該是擁有足夠的資訊可以來做處理。例如 `GameServer` 可以嘗試先檢查是否已經有一個相同應用程式正在執行，如果有則嘗試終止它，或是顯示警告訊息給使用者。又或者，`GameServer` 可以改變系統配置，然後重新呼叫 `Acceptor` 透過其他連接埠來接受網路連線。

***

例外處理和打仗很類似，打仗需要知道戰場的地形、地物、天候、敵人戰鬥力、裝備與兵力部署等資料（作戰的 context），而例外處理也必須借用 exception context、object context、local context，以及 architecture context 等資訊來設計合適的例外處理程式。

***

友藏內心獨白：「能力越強，責任愈大」，擁有越多資訊的人，理應處理更棘手的例外。

10 物件導向語言的例外處理機制

在學習與使用一個程式語言來進行例外處理設計與實作的時候，無論這個語言是常見的商用語言像是 Java、C#、Objective-C、C++、VB.NET、Delphi、Python、Ruby、PHP，或是曾經在教科書上看到但卻很少有機會碰觸到的 Modula-3、CLU、Ada、BETA、Smalltalk、Eiffle，鄉民們內心深處是否存在一個疑問：「**到底怎樣才算是把這個語言的例外處理方法給學完整了？**」以 Java 為例，`try-catch-finally`、`throw`、`throws` 這幾個關鍵字的意義和語法很快就懂了，但總覺得光是知道語法好像還是有一種很空虛、很冷的感覺，不太確定是否已經掌握了這個語言對於例外處理所提供的支援。

到底設計一個程式語言的例外處理方法，要考慮那些因素？這些因素的內涵代表什麼意義？這些問題提供一個清楚的框架，除了能引導學習方向，同時也扮演著「**驗收條件**」的角色，可以用來評估自己對於一個程式語言例外處理機制的理解程度。此外，也可以依據這些因素來比較不同程式語言之間例外處理方式的差別，以便在一個新的程式語言裡面，重複使用以往在不同程式語言所累積的例外處理經驗，縮短學習時間並且減少犯錯的機會。這就是為什麼需要了解例外處理機制的原因。

本章介紹物件導向語言中，設計例外處理機制所需考量的十個因素。這些分類因素由 Garcia 與其同事所發表的〈A comparative study of exception handling mechanisms for building dependable object-oriented software〉[12]（建立可信賴物件導向系統之例外處理機制的比較性研究）一文中所提出，包括：

1. Representation：如何**表達**一個例外？
2. Declaration：往外傳遞的例外是否需要**宣告**？
3. Signaling：如何**產生**一個例外的實例（instance）？
4. Propagation：如何**傳遞**一個例外？
5. Attachment：例外處理程序可**綁定**到何種程式區塊？
6. Resolution：針對一個例外，如何**找到**可以處理它的例外處理程序？

[12] 參考資料：Garcia, C. M. F. Rubira, A. Romanovsky, and J. Xu, "A comparative study of exception handling mechanisms for building dependable object-oriented software," Journal of Systems and Software, vol. 59, no. 2: 197-222, 2001.

7. Continuation：例外發生之後，程式的**控制流程**該如何進行？
8. Cleanup：無論是否發生例外，如何讓程式**清理資源**以便保持在正確的狀態？
9. Reliability Check：因為例外處理機制所造成問題，程式語言提供何種**檢查**？
10. Concurrency：程式語言對於**並行處理**程式提供多少例外處理的支援？

本章將針對以上幾點逐一說明。

***

Representation

程式語言**表達例外**的方法有三種，分別是：

1. 符號（symbol）：用字串或是整數來代表一種例外狀況，Eiffel 語言的例外就是屬於這一種。為什麼要用字串或整數來代表錯誤？答案很簡單，就是想要降低用物件表達錯誤所造成的系統效能減低的可能性。
2. 資料物件（data object）：用物件來表示與儲存錯誤訊息，這是最常見的表達例外方法，Java、C++、C#的例外都屬於這一種。基本上這類的例外表達方式，例外物件身上沒有什麼行為，單純以物件的形式做為資料使用。
3. 完整物件（full object）：除了用物件來表示與儲存錯誤訊息，連如何產生一個例外實例、傳遞例外、例外發生之後的程式控制流程等，全部都定義在例外類別之中。BETA 的例外就是屬於這一種。

***

Declaration

函數向外傳遞的例外，是否需要宣告在該函數介面之上。有以下幾種做法：

1. 強制式：所有往外傳遞的例外一定要宣告在函數介面上。如圖 10-1 所示，Java 的 checked exception 採用強制式宣告，違反規則的函數視為編譯錯誤。C++的例外規範（exception

specification）也是一種強制式宣告，違反規則的函數視為執行錯誤，C++執行環境會呼叫內建的 unexpected 函數終止程式執行。

```
16    public void declareCheckedException() throws IOException{
17        throw new IOException();
18    }
19
20    public void doesNotDeclareCheckedException() {
21        throw new IOException();
22    }
```

圖 10-1：強制式宣告例外程式範例

2. 選擇式：宣不宣告都可以，例如 Java 的 unchecked exception 採取選擇式宣告。如列表 10-1 所示，IllegalArgumentException 屬於 unchecked exception，鄉民們可以使用 throws 關鍵字把它宣告在函數介面，也可以不宣告。

```
1: public void declareUncheckedException(int x)
2:                         throws IllegalArgumentException {
3:     if (x < 0) throw new IllegalArgumentException();
4: }
5: public void doesNotDeclareUncheckedException(int x) {
6:     if (x < 0) throw new IllegalArgumentException();
7: }
```

列表 10-1：Java 的 unchecked exception 可選擇性宣告在函數介面之上

3. 不支援：根本不支援在介面上宣告例外，例如 C#語言，沒有關鍵字來支援例外宣告，開發人員只能用註解或是 XML 標籤來說明函數所可能丟出的例外。這種說明對於編譯器或是執行環境沒有任何的強制性。
4. 混和式：同時支援上述一種以上的方式。廣義的來說， Java 對於例外是否需要宣告的作法是屬於混和式的做法，因為只有 checked exception 要強制宣告，unchecked exception 則採用選擇性的方式。

鄉民們可以自己練習一下，看看自己曾經學過的程式語言，其對於例外宣告的做法是屬於上述的哪一種。

***

Signaling

把例外類別的實例（instance）傳遞給例外接收者的這個動作叫做 **signaling**、**throwing**、**raising** 或 **triggering**。以 Java 為例，請參考列表 10-2，第 3 行與第 9 行使用 `throw` 關鍵字拋出例外。

```
1: public void handle(){
2:     try{
3:         throw new IOException("Function failure");  // 拋出例外
4:     } catch(IOException e){
5:         // 處理例外
6:     }
7: }
8: public void declare() throws IOException{
9:     throw new IOException("Function failure");  // 拋出例外
10: }
```

列表 10-2：Java 語言使用 `throw` 關鍵字來拋出例外

從 signaling 的角度來看，例外產生的方式可分為兩種形式：

- 同步例外（synchronous exception）：因為程式中所呼叫的指令或函數執行失敗所產生的例外。
- 非同步例外（asynchronous exception）：由執行環境（例如 JVM）所主動丟出的例外，和目前程式所執行的指令或函數沒有直接的關係。例如，執行環境偵測到內部錯誤或是記憶體不足而丟出一個例外，像是 Java 的 `OutOfMemoryError` 就屬於非同步例外[13]。

大部分的例外處理設計所談論的議題在於要如何來處理同步例外，當非同步例外發生，通常表示執行環境已經產生了嚴重的錯誤，此時最好的辦法就是結束程式執行，避免一錯再錯。

另外，從例外是否可傳遞出軟體元件的介面，又可將例外分成：

- 內部例外（internal exception）：在軟體元件內部所引發的例外，主要的目的是要將程式執行控制權跳到一個內部定義的例外處理程序。內部例外無法傳遞出軟體元件的外部。

[13] Java 的非同步例外可參考線上文件：http://docs.oracle.com/javase/specs/jls/se7/html/jls-11.html。

- 外部例外（external exception）：在軟體元件內部無法處理的例外，而需要將此例外往外傳遞。

如果一個程式語言有區分內部例外與外部例外，它可能會用不同的指令來產生例外實例。例如，使用 `raise` 來產生內部例外，而使用 `signal` 來產生外部例外。Garcia 與其同事認為區分內部與外部例外可以設計出更可靠的軟體系統，然而就 Teddy 所知，目前並沒有任何一個主流的程式語言區分內部與外部例外。以 Java 為例，如果例外被一個函數內部的例外處理程序（`catch` block）給捕捉住而沒有往外傳，這個例外就屬於內部例外。反之，便屬於外部例外。但無論是何者，都適用 `throw` 這個指令來丟出例外，也就是說 Java 並沒有特意區別內部例外與外部例外。

***

Propagation

Propagation 討論例外傳遞的問題，例如 `A` 函數呼叫 `B` 函數，在 `B` 函數裡面產生了一個例外，但是這個例外卻沒有被 `B` 函數給處理（捕捉住），則該例外可能會被傳遞到 `A` 函數身上。

例外傳遞的方式分成兩種：

- 外顯式（explicit）：接收到例外的人，如果不想處理這個例外，則必須要明白的指出要把這個未被處理的例外往外丟。Java 的 checked exception 就屬於這種類型，如果某個函數遇到一個 checked exception 但卻不想處理它，則必須要把這個 checked exception 宣告在函數介面。雖然在程式裡面不需要先捕捉這個 checked exception 然後再把它重丟出去，但是廣義的來看，要求把不想處理的例外宣告在介面上面，這樣也算是一種 explicit propagation。

列表 10-3 的 Java 程式片段則展示另外一種情況，捕捉例外加以處理之後轉成另外一種客戶端容易理解的例外類別並且將其宣告在函數介面上。第 5 行 `is.readFully(messageBody)` 會丟出 `EOFException` 與 `IOException`，`readMessageBody` 函數處理過後把 `IOException` 轉成 `InvalidPacketException`，然後把它宣告在函數介面上，讓呼叫 `readMessageBody` 函數的人有機會可以來處理 `InvalidPacketException` 的異常狀況。

```
1: public byte [] readMessageBody(int aLength, DataInputStream is)
```

```
2:         throws InvalidPacketException {
3:     try {
4:         byte[] messageBody = new byte[aLength];
5:         is.readFully(messageBody);
6:         return messageBody;
7:     } catch (EOFException e) {
8:         throw new InvalidPacketException("Data underflow.", e);
9:     } catch (IOException e) {
10:        throw new InvalidPacketException("Read data body error.", e);
11:    }
12: }
```

列表 10-3：Java 的 checked exception 採取外顯式例外傳遞方式

- 內隱式（implicit）：又稱為自動傳遞方式。接收到例外的人，如果不想處理這個例外，在預設的情況之下，例外會直接往外傳遞，不需要特意指定要把這個未被處理的例外往外丟。Java 的 unchecked exception，以及 C#所有 exception 的傳遞方式，就屬於內隱式。

列表 10-4 的 C#程式範例，其中第 4、5 兩行程式都可能產生例外，但是因為 C#的例外傳遞方式是採取內隱式傳遞，所以如果 `writeFile` 不想處理這些例外，也不需要特別指定要把它們往外丟，在預設的情況下這些沒有被捕捉的例外就會自動往上傳遞。

```
1: public void writeFile(String filePrefix, String data){
2:    TextWriter writer = null;
3:    try {
4:        writer = new StreamWriter(filePrefix + "_" + data + ".txt");
5:        writer.Write(data);
6:    } finally {
7:      // cleanup code
8:    }
9: }
```

列表 10-4：C#例外採取內隱式傳遞方式

***

Attachment

例外處理程序可以附加到哪些受保護的程式區域，常見的做法有：

- 敘述（statement）或區塊（block）：將例外處理程序與程式敘述或是區塊綁定在一起，這也是大部分主流物件導向程式語言所提供的做法，像是 C++、Java、C#，提供 `try-catch` 的關鍵字，讓藉由 `catch` 所定義的例外處理程序與 `try` 所定義的程式敘述綁定在一起，如列表 10-5 所示。

```
1: public void handle(){
2:     try{
3:         throw new IOException("Function failure");
4:     } catch(IOException e){
5:         // 我是例外處理程序
6:     }
7: }
```

列表 10-5：Java 的 例外處理程序（`catch` block）綁定到一個 `try` 敘述

- 函數（method）：將例外處理程序與整個函數綁定在一起，Eiffle 語言的例外處理屬於這一類型。
- 物件（object）：例外處理程序可以綁定到某一個特別的物件身上，支援的語言有 BETA，一般常用的程式語言比較少支援這種類型。
- 類別（class）：例外處理程序可以綁定到某一個類別身上，支援的語言有 Ada 95、Eiffle、Smalltalk、BETA。
- 例外（exception）：例外處理程序直接綁定到例外類別身上，當程式執行期間發生例外但卻找不到相對應的例外處理程序的時候，系統會呼叫定義在例外類別裡面的預設處理程序來處理例外（也就是例外物件自己處理自己）。支援的語言有 BETA。

***

Resolution

Exception resolution 又稱為 handler binding，探討如何找到一個例外處理程序的過程。尋找 handler 的方式，可以透過：

- 靜態範圍（static scoping）：只要光看程式碼的結構，就可以決定例外要丟給誰來處理。以列表 10-6 程式碼為例，在 `try` block 所丟出的 `IOException`，直接被相對應的 `catch(IOException)` block 所捕捉，所以只要依賴靜態範圍便可找到例外處理程序。

```
1: public void handle(){
2:     try{
3:         throw new IOException("Function failure");
4:     } catch(IOException e){
5:         // 處理例外
6:     }
7: }
```

列表 10-6：透過靜態範圍尋找例外處理程序

- 動態範圍（dynamic scoping）：需要在程式執行期間透過呼叫鏈(call chain)的關係來決定如何找到合適的例外處理程序，例如列表 10-3 中 `readMessageBody` 函數所丟出的 `InvalidPacketException` 將會由誰來處理？不知道，因為要看程式執行之後的呼叫關係，才知道如果 `readMessageBody` 函數丟出例外，會被 call chain 上面的那一個函數所處理。
- 半動態範圍（semi-dynamic scoping）：純粹採用靜態或動態範圍來決定如何尋找例外處理程序的程式語言並不多，大部分的程式語言都是同時採用這兩種做法。當在本地端可以找到例外處理程序時，就採用靜態範圍。找不到合適的本地端例外處理程序，就透過動態範圍來尋找。

***

如果是採用動態的方式來尋找例外處理程序，則尋找的方式又可以細分成兩種：

- 堆疊展開（stack unwinding）：程式執行的時候，函數呼叫的順序形成了一個 call chain，而這個 call chain 通常被執行環境保存在一個 stack 裡面。stack unwinding 的意思是說，沿著這個 stack 一層、一層往外找，直到找到一個合適的例外處理程序為止。
- 堆疊切割（stack cutting）：另外一種尋找的方式則是把例外處理程序放到一個 list（串列或是陣列）裡面，當例外發生的時候，到這個 list 去尋找是否有合適的例外處理程序。

2013 年 9 月所發生的「監聽門」事件，剛好可以用來解釋以上兩種不同的方法。當檢察總長發現到疑似司法案件關說的「~~常態~~ 例外」情況，如果採用 stack unwinding 的方式，檢察總長應該是逐層向上回報，如圖 10-2 所示。

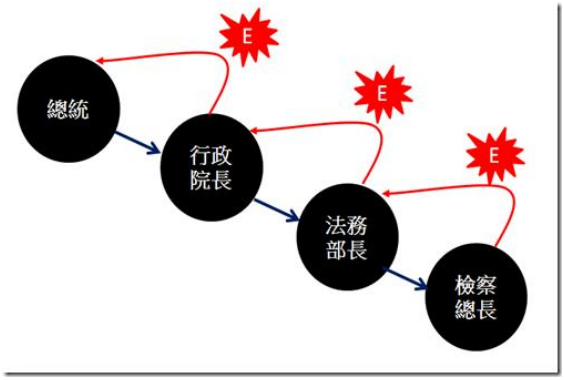


圖 10-2：stack unwinding 示意圖

但是，可能是因為這個「例外」牽涉到「法務部長」，於是檢察總長便改採 stack cutting 的方式來尋找他心目中合適的例外處理程序。假設行政院長與總統都曾經向檢察總長表示過「遇到問題可以來找他」，在「監聽門」的這個例外情況之下，經過判斷檢察總長覺得「總統」這個例外處理程序比較「給力」，於是選擇了把這個例外丟給總統來處理，如圖 10-3 所示。

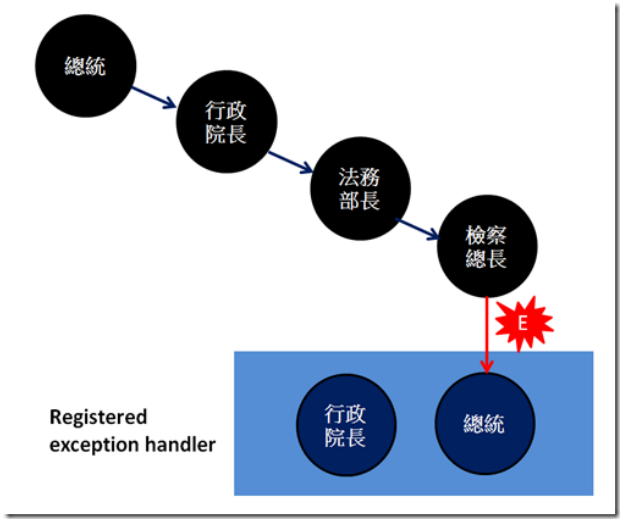


圖 10-3：stack cutting 示意圖

***

尋找合適的例外處理程序，以及例外處理程序本身是否能夠妥善處理例外，都是很重要的議題。試想一下檢察總長把「例外」丟給總統這個「例外處理程序」的下場，不但一開始 exception resolution 的過程就有可議之處，「例外處理程序」本身的實作方式也不好。這個故事告訴大家，例外處理真的很重要啊。

***

Continuation

Exception continuation 又稱為 exception model（例外模型），規範當例外處理程序執行完畢返回控制權之後程式的執行流程，常見的 exception model 包含以下三種：

- 終止模型（Termination model）：當例外處理程序裡面的程式碼執行完畢之後，程式控制權並不會返回原本發生例外的那一行指令，而是直接終止原本指令的執行。C++、Java、C# 這些程式語言都是採用 termination model。以列表 10- 為例，第 5 行程式 `is.readFully(messageBody)` 如果發生 `EOFException`，則程式流程會跑到第 7 行 `catch(EOFException e)` 這個例外處理程序身上。當例外處理程序執行完畢之後，控制權並不會返回第 5 行程式碼 `is.readFully(messageBody)`。
- 恢復模型（resumption model）：執行完例外處理程序之後，會繼續執行原本例外發生的那一行指令。Visual Basic 就支援這種模式，當例外處理完畢之後，可以用 resume 或是 resume next 指令，讓程式控制權返回例外發生的那一行（resume）或是下一行（resume next）程式。
- 重試模型（retry model）：結合 termination mode 與 resumption model 產生了 retry model。執行完例外處理程序之後，會先終止原本例外發生的那一個區塊，然後再重新執行一次整個區塊。圖 10-4 是一段 Eiffel 程式範例[14]，第 27 行的 `retry` 指令會重新執行一次 `get_integer` 函數的內容。除了 Eiffel 以外，近幾年很流行的 Ruby 語言，它的例外處理也參考 Eiffel 的設計，支援 retry model。

[14] 節錄自 B. Meyer, *Object-Oriented Software Construction, 2nd*, Prentice-Hall, 1997.

```
01  Maximum_attemps: INTEGER is 5
02     -- Number of attempts before giving up
03     --      getting an integer
04
05  get_integer is
06     -- Attempt to read integer in at most
07     --      Maximum_attempts attempts.
08     -- Set value of integer_was_read to record
09     --      whether successful.
10     -- If successful, make integer available
11     --       in last_integer_read.
12
13    local
14       attempts: INTEGER
15          -- default value of attempts is 0
16    do
17       if attempts < Maximum_attempts then
18           print("Please enter an integer:")
19           read_one_integer
21           integer_was_read := True
22       else
23           integer_was_read := False
24       end
25     rescue
26        attempts := attempts + 1
27        retry
28     end
```

圖 10-4：Eiffel 例外處理範例

***

一個程式語言可以支援一種以上的 exception model，但是為了設計上的簡化起見，目前主流的程式語言大多只支援 termination model。因為有人證明過，可以在 termination model 模擬出另外兩種 model。但是，如果鄉民們所採用的程式語言預設只支援 termination model，那麼在設計例外處理的時候，像是 retry 這種很有用的技巧就比較容易被忽略。反之，在 Eiffel 或 Ruby 這種直接支援 retry model 的程式語言裡面，採用 retry 這種例外處理策略則是家常便飯。

***

Cleanup

無論是否發生例外，一個軟體元件都應該要保持在正確的狀態，如此整個系統才可以繼續執行。因此，軟體元件要執行清理的動作以便確保系統狀態的正確性。清理的動作包含在錯誤發生之後讓系統恢復到一個可以接受的一致性狀態或是釋放資源以避免資源耗盡。

物件導向程式語言對於清理動作的支援，可以分為：

- 外顯傳遞（explicit propagation）：當例外往外傳遞之前，系統會呼叫資源清理程式碼。在C++語言中，如果使用了「資源取得便是初始化（resource acquisition is initialization；RAII[15]）」技巧，系統在傳遞例外之前便會自動呼叫解構函數（destructor）執行清理資源的工作。
- 特定構造（specific construct）：採用特定構造確保無論是否有例外發生，資源清理的程式碼都會被執行。Java 與 C#的 `finally` block 就屬於這一種。
- 自動清理（automatic cleanup）：系統自動清理所有資源，程式設計師完全不需要管資源釋放的問題。目前似乎沒有主流的程式語言可以做到這一點，Java 的 try-with-resources 與 C# 的 `using`，勉強算是比較接近自動清理的一種作法[16]。

***

Reliability Check

例外處理也是一種程式設計，因此可能會因為使用程式語言的例外處理機制而發生錯誤。可靠性查驗代表程式語言對於不正確使用例外處理機制的檢查支援，可分為以下兩種：

- 靜態檢查（static check）：在程式編譯的時候做檢查。以 Java 為例，靜態檢查包含往外傳遞的 checked exception 是否有宣告在函數介面之上、用 `throw` 所丟出的例外物件是否繼承自 `Throwable` 類別、`try` statement 的使用是否正確等。
- 動態檢查（dynamic check）：在程式執行期間所做的檢查。以 C++語言為例，C++支援例外規範，要求例外必須要宣告在函數介面，如下面 `foo` 函數宣告，`throw (int)` 表示 `foo` 函數只允許丟出型別為 `int` 的例外。C++對於函數所宣告的例外並沒有在編譯期間執行靜態檢查，而是在執行期間動態檢查。違反例外規範會引發 C++執行環境呼叫內建的 `unexpected` 函數，其預設行為將終止程式執行。

```
double foo (char arg) throw (int);
```

大部分的程式語言，像是 Java、C#、C++，同時都支援靜態與動態的可靠性檢查。

***

[15] 關於 RAII 的說明可參考維基百科 http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization 或 C++之父 B. Stroustrup 的著作：*The Design and Evolution of C++*, Addison Wesley, 1994, p. 389.

[16] 嚴格講起來只能算是半自動，因為開發人員還是要自己撰寫釋放資源的程式，系統只是自動幫忙呼叫這些釋放資源的程式而已。

Concurrency

並行或是平行處理算是程式語言裡面比較進階的項目，如果程式語言的例外處理有考慮到平行處理，則同時間可以有數個例外一起被產生，而例外處理機制也要考慮如何來處理這些同時產生的例外狀況。程式語言對於並行處理的例外支援，可分為以下三種：

- 不支援（unsupported）：老牌的程式語言像是 C++，沒有內建的並行處理支援，更談不上在並行處理程式中的例外處理機制。
- 有限支援（limited）：可以在多執行緒的程式中丟出例外，但是例外處理程序的解析需要依靠開發人員的規劃，系統也不支援「不可分割動作」（atomic action）。
- 完整支援（complete）：，對於並行處理程式的例外處理具有最完整（最理想）的支援，程式語言可以指定那些函數是屬於不可分割動作，當執行失敗的時候例外處理機制會確保這些不可分割動作的狀態完整性。基本上只要是與並行處理程式特性相關的例外狀況都需要支援。

想當然爾，流行的商業語言中，Java 和 C#屬於部分支援。以 Java 為例，因為語言內建支援執行緒，因此例外可以由個別不同的執行緒所產生，但是執行緒所產生的例外並不會傳遞到其他執行緒中。Java 的執行緒有一個 `setDefaultUncaughtExceptionHandler` 函數，可以在執行緒身上設定一個預設的例外處理程序，用來捕捉傳遞到執行緒身上的例外。這些都算是對於並行處理程式的有限例外處理支援。

至於完整支援並行程式例外處理機制的商用程式語言，可能還在等待高人發明之中。

***

友藏內心獨白：分類一定要湊滿整數就是了。

11 你的汽車有多耐撞？談談例外安全性

在購買新車的時候，除了考慮車的外型、顏色，性能以外，另外一個最重要的因素，應該就是安全性。翻成白話文就是：這輛車耐不耐撞？會不會車尾輕輕被 A 到一下，整個車屁股就解體了？一般而言，大家普遍的印象都是，歐洲車的板金比較厚，比較耐撞。日系車種為了表現出漂亮的油耗，車子都比較輕，也比較不耐撞。

北美和歐洲都有所謂的汽車耐撞測試，滿分五分，分數越高代表車輛在撞擊之後對於人員的安全性具有越高的保護。

那程式呢？有沒有程式在發生例外之後的安全性標準？

有的，有些研究學者可以用數學的方式來研究例外安全性。太理論的東西大部分的鄉民應該都「嘸呷意(不喜歡)」，這一章介紹一個比較沒那麼理論的標準：**例外安全性(exception safety[17])**。

***

例外安全性原本是 C++社群在討論 C++標準類別庫的實作，應該要達到怎樣的標準才可以稱為例外安全。雖然這個觀念起始於 C++語言，但是同樣可以適用於其他支援例外處理的程式語言，像是 Java、C#、Python、Ruby 等。

例外安全性可區分為四個等級：

- 不保證（no guarantee）：一個函數丟出例外的時候，對於系統的狀態正確性沒有任何的保證。系統可能恰巧處於正確狀態，也極有可能處於錯誤狀態。
- 基本保證（basic guarantee）：又稱為無洩漏保證 (no-leak guarantee)，發生例外的函數可能會造成一些副作用，但系統基本上處於正確狀態，而且沒有資源洩漏發生。例如，假設有一個 `drawPicture` 函數負責在螢幕上重複輪流顯示圖 11-1(a)的大眼怪以及圖 11-1(b)的尤達大師。假設 `drawPicture` 正在將尤達大師顯示在畫面上的時候發生例外，如果 `drawPicture` 具備基本保證，則它會保證例外發生之後系統對於照片的資源使用沒有發生資源洩漏的問題，而且畫面上一定會有照片被顯示（系統狀態是對的）。但是，`drawPicture` 無法保證畫面上所顯示的是尤達大師，還是大眼怪。

[17] 可參考維基百科與其所列出之參考資料：http://en.wikipedia.org/wiki/Exception_safety，或是以下書籍：H. Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 1999. S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd, Addison-Wesley, 2005.

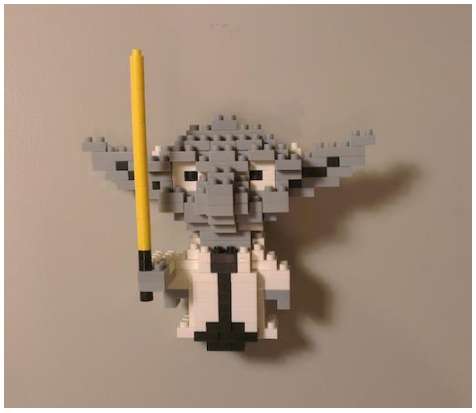



圖 11-1：(a) 大眼怪，(b) 尤達大師

- 強烈保證（strong guarantee）：具備認可或復原（commit-or-rollback）或是全部或沒有（all-or-nothing）的語意，就好像資料庫交易處理一樣，一個函數執行，要就成功，否則就失敗，沒有模糊的空間，不會產生什麼副作用，也不會有資源洩漏的問題。假設 `drawPicture` 具備強烈保證，它目前正將尤達大師的照片替換掉大眼怪的照片。如果沒有例外發生，則 `drawPicture` 執行結束之後會保證螢幕上一定是顯示尤達大師的照片。如果 `drawPicture` 的執行發生例外，則它會保證螢幕上顯示的依然是大眼怪的照片。
- 不丟擲保證（no-throw guarantee）：又稱為失效透明（failure transparency），翻成白話文的意思是說函數的執行不會失敗，當然也不會丟出任何例外。不丟擲保證並不是說函數在執行的時候不會遭遇例外，而是說即使遭遇到例外，也要在內部將例外狀況處理好，然後達成原本該函數被賦予的任務。所以從外部呼叫者的角度來看，這個函數保證一定會成功。

要做到不丟擲保證其實是一件很困難的事情，因為程式要出錯的可能性實在是太多、太多了，而要讓一個函數保證永不出錯，實在是有點強人所難。但是，如果沒有依靠少數具備不丟擲保證的函數來「撐場面」，就不可能達到強烈保證與基本保證。在 C++的標準類別庫中，只要求少數操作，例如基本型別、陣列、指標的指定、解構函數、`swap` 函數需要具備不丟擲保證，其餘函數可以基於這些操作來實作自己的例外安全性。

一個函數符合**例外安全（exception safe）**表示這個函數具備了基本保證、強烈保證、或不丟擲保證三者其中之一。

***

看到這邊請鄉民們回想一下：以前寫程式的時候，是否有留意到自己的程式達到哪一個例外安全性等級？如果大部分的函數都是不保證，那就真的要多買幾包綠色乖乖，放到電腦旁邊，求神明保佑。在這種情況下，系統不出錯才是異常，出錯則反而是正常。

例外安全性的觀念，可以做為開發強健度系統的一種評估與驗收標準。在專案規畫階段，可以依據專案的特性，來判斷與規範程式所需要達到的例外安全性。例如，如果這只是網頁查詢資料系統，而且時程很趕，客戶付的錢又很少，也許大部分的函數只要做到基本保證就可以了。如果是屬於系統共用或是核心元件，則應該儘量做到強烈保證。如果是為了趕著讓業務明天可以展示給客戶看的雛型，則也許根本不需要考慮例外安全性的問題，不保證就 OK 了。

***

友藏內心獨白：總不能因為怕被撞，就開戰車上路吧。

12 例外處理 PK 容錯設計

Teddy：考考你。

鄉民甲：儘管考。

Teddy：例外處理就是要在程式裡面處理遭遇到的例外，對不對？

鄉民甲：是啊（這是陷阱題嗎？！）。

Teddy：請問遇到 NullPointerException 要怎麼處理？

鄉民甲：嗯，NullPointerException 通常是變數忘了初始化，如果要處理的話，程式可以這樣寫：

```
1: public class NullPointerExceptionApp {
2:     List<Integer> list = null;
3:     public int foo(int arg){
4:         try{
5:             list.add(arg);
6:         }catch(NullPointerException e){
7:             list = new ArrayList<>();
8:             list.add(arg);
9:         }
10:        return list.size();
11:    }
12: }
```

列表 12-1：Java 程式範例，展示如何處理 NullPointerException

鄉民甲：如果 list 這個資料成員忘了初始化，在 foo 函數裡面可以捕捉 NullPointerException，然後初始化 list，這樣 foo 就可以正常執行了。你看我的例外處理學的還不錯吧。

Teddy：決定要不要誇你之前再問你一個問題，「你確定列表 12-1 的作法是屬於例外處理」？

鄉民甲：（果然有陷阱…看我的水母神功…）try、catch、NullPointerException 都派上用場了，**這不是例外處理，什麼才是例外處理？**

***

請鄉民們回憶一下〈CH8：強健性大戰首部曲：威脅潛伏〉這一章裡面關於 fault、error、failure、exception 的關係圖，重繪如下。

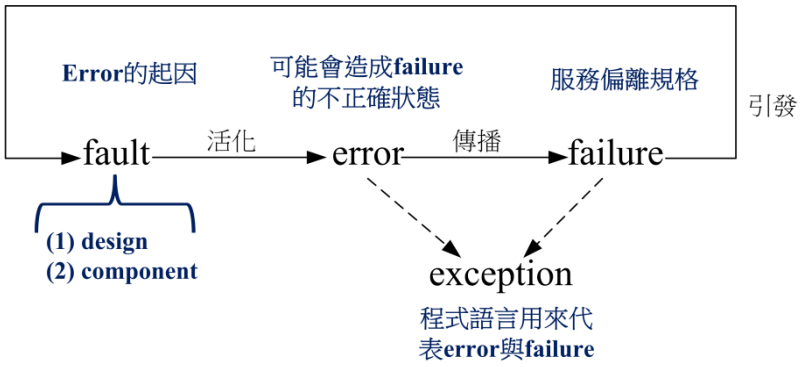


圖 12-1：Fault、error、failure 與 exception 的關係

當一個軟體元件發生 error，必須加以處理以免 error 變成 failure。根據 Knudsen[18]的看法，錯誤處理（error handling）可以分成兩類：

- 例外處理（exception handling）：負責處理 component fault（可預期的錯誤狀況）。
- 容錯設計（fault-tolerant programming）：負責處理 design fault（不可預期的錯誤狀況）。

由於程式語言只提供了例外處理機制，並沒有特別幫容錯設計安排額外的機制，因此無論是上述分類中的「exception handling」或「fault-tolerant programming」，在實作層面都可以採用程式語言的例外處理機制來達成。**這也是造成混淆不清的地方，因為當我們在談「例外處理」或是「錯誤處理」的時候，有可能是指「exception handling」，也可能是指「fault-tolerant programming」。**

鄉民甲：那又怎麼樣，反正都是錯誤處理啊？

當然不一樣，因為兩者的成本不同。回到列表 12-1 的程式碼，在程式中所捕捉的 `NullPointerException` 是一種 design fault，表示程式中存在一個忘了初始化 `list` 物件的 bug。如果真的要在程式裡面用例外處理的方式來處理 design fault 所造成的錯誤，此時就已經從「exception handling」提升到「fault-tolerant programming」。

[18] J.L. Knudsen. “Exception Handling versus Fault Tolerance”, in ECOOP’2000 Workshop W2. 該文章可在此處下載：http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/e1.pdf

兩種錯誤處理的方式，何者成本較高？當然是「fault-tolerant programming」。以 NullPointerException 為例，試想一下，程式中幾乎每一個使用到物件的敘述都有可能發生 NullPointerException。如果真的要在程式中考慮使用例外處理的方式來復原 NullPointerException 所造成的程式錯誤，這是一件多個可怕的工程？

容錯、容錯，顧名思義就是「要容忍程式裡面的錯誤，就算是程式有錯系統也必須要正常執行」。容錯設計通常是使用在需要非常高強健度等級的應用系統，例如飛機控制系統、軍事用途系統、衛星或太空梭控制系統等。由於對於可靠性的要求非常高，具備容錯功能的系統開發成本也比一般商用系統高出很多。

***

房仲：您想要尋找怎樣的房子？

鄉民：三房兩廳，地點要在天龍國市中心，距離捷運站 300 公尺以內，屋齡在五年之內的電梯大廈高樓層。室內可使用坪數最少 30 坪，還要附一個平面汽車車位。

房仲：我幫您查一下資料…有了，這裡有一戶屋齡三年的捷運共構房屋，符合您的需求。請透過我們獨家的 3D 虛擬實境線上看屋系統參觀一下這個物件。

鄉民：嗯，看起來不錯，前屋主的裝潢也很新。就是它了。

房仲：這戶屋主開價 5000 萬，請問您自備款有多少？

鄉民：5000 萬啊，剛好耶，我符合條件，我有 50 萬的自備款。

房仲：什麼，您只有房價 1%的自備款？

鄉民：新聞不是都有報導，只要準備 1%自備款就可以買豪宅了。

房仲：這…，我看您還是租屋先吧。

***

相信大部分的人都想要住在交通便利、治安良好、空間寬敞、學區好的豪宅，就好像每個人都想要使用穩定度高又可靠的軟體系統一樣，人類對於追求好東西的慾望是無止境的。但問題是，你願意付出多少代價來獲得這些品質？

身為一位軟體開發人員，你的客戶可能曾經告訴過你：我們公司生意做很大，一秒鐘幾十萬上下，這個系統你們要好好做，**不允許有當機的情況發生**。看來你的客戶心中要求的是一個具備高強健度等級與容錯功能的系統，很可惜他們為這個系統所準備的經費，只「請得起猴子」。當下次再聽到客戶對於強健度要求的時候，鄉民們要豎起耳朵，注意分辨客戶要求的是「exception handling」還是「fault-tolerant programming」，後者所需的經費與時間絕非前者可比擬。

話說回來，一般人連「exception handling」都沒時間做了，哪可能做到「fault-tolerant programming」。不要想這麼多，還是洗洗睡吧。

***

友藏內心獨白：記得當時年紀小，也曾經好傻好天真，想要在程式裡面處理`NullPointerException`。

Column C. | 網路又斷了

Teddy 家裡的光世代網路又斷線了，此為犯罪現場

2010 年 10 月和 Kay 到京都、大阪、奈良旅遊，Kay 特別規劃了一天去爬奈良的春日原始林。走著、走著繞了一大圈來到若草山。當時若草山某個區域正在施工，當 Kay 和 Teddy 要下若草山的時候，經過了施工的區域，有一台小山貓在挖土。這時候 Teddy 看到一位「歐吉桑」站在路旁指揮，這位歐吉桑看到 Kay 和 Teddy 經過，口中唸唸有詞，好像是在說「施工造成不便很抱歉」之類的話（迷之音：不懂日文怎麼會知道歐吉桑說什麼？）。同時，歐吉桑還示意操作小山貓的工人暫停工作，等 Kay 和 Teddy 通過之後才繼續啟動小山貓。

當時 Teddy 心裡就在想：「這些日本人也太小心了吧，根據 Teddy 目測結果，小山貓應該不會對經過的遊客造成什麼傷害才對，為什麼還要特意暫停等遊客經過呢？」。

***

最近兩個禮拜之內 Teddy 家中的光世代網路連續斷線三次。第一次是中華電信機房人員剛好在 Teddy 出國的時候「好心」把光世代給免費升速，但可能是機房設定有問題，所以 Teddy 的網路就在此次善意行動之後就斷線了。當時人在國外，只好等回國之後再打電話去報修。後兩次很可能是因為這幾天台北市政府「好心」的更新路口的人行道，網路應該是被施工包商連續挖斷兩次（由於沒有檢調介入協助調查，實際斷線原因不得而知）。因為 Teddy 這一陣子都在家裡工作，每次斷線都無法上網，真的很不方便。還好 Teddy 的手機可以無線上網，只好先用慢速的 3G 網路頂著先，才不至於影響到工作的進行。

***

以上兩個故事剛好是下面這兩句話的對比：

- Better safe than sorry（有備無患）：日本歐吉桑的做事方法。
- No safe, then saying sorry（撞到有聲）：台灣人行道施工包商的做事方法。

「事先不思考如何提高品質，事後再來販賣對不起」，這種案例在台灣實在是太多了。這種心態，跟寫程式但卻不願意寫測試案例或是不做例外處理一樣，都是「賭一把」的心態。「噯呀，寫測試、做好例外處理，好麻煩啊。乾脆什麼都不做，然後看看會有什麼事情發生。如果什麼事都沒有那就賺到了，反正有事的話客戶自然會告訴我們，到時候再來解 bug 就好了。」

什麼是好的服務？**當你不覺得自己被服務的服務，就是好的服務。**鄉民們不會每天感謝自來水公司和台電提供乾淨的飲用水和穩定的電力，也不會感謝老天爺賜給你乾淨的空氣。可是一但沒水、沒電、空氣汙染，那種不方便和痛苦，只有親身經歷過的人才能夠體會。

***

缺陷處理有四種方法[19]：

- 缺陷避免（fault prevention）：從做事方法與流程上面防範於未然，從根本上避免產生缺陷。這就是若草山的日本歐吉桑所採取的策略。
- 缺陷容忍（fault tolerance）：透過**錯誤偵測**與**系統回復**的手段，來避免服務失效。假設中華電信光世代支援缺陷容忍，當線路被人行道施工包商給挖斷的時候，系統會自動偵測到這個錯誤，然後切換到備援線路繼續提供民眾上網服務。
- 缺陷移除（fault removal）：在開發階段透過**驗證與確認（verification & validation；V&V）**手段移除注入系統中的缺陷，以及在使用階段透過**修正性維護（corrective maintenance）**來移除使用者回報的缺陷。這是中華電信採取的策略，當客戶跟中華電信回報問題之後，由中華電信派出檢修人員來移除以人行道施工包商所製造的缺陷。在使用階段的缺陷移除，比較好的方式是採用不中斷服務的方式來移除系統中的缺陷。如果是採用離線方式（offline）來移除缺陷，會造成**服務中斷（service outage）**。

[19] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, 2004.

- 缺陷預測（fault forecasting）：用定性或定量的方式評估因為缺陷導致系統失效的模式與機率。例如，網路經常斷線的原因有哪些？多久發生一次？

以上四種方法彼此之間並不是互斥的，同時採用可以更加增強系統或服務的可靠性。台灣人做事情，似乎少了點風險意識，**錯誤發生的時候，被要求要容忍的總是受害的客戶**（迷之音：你喜歡退，你就退）。想想 2013 年 11 月初的高鐵跳電事件，幾萬名旅客被影響，社會付出的成本何止千萬計。而高鐵呢？繼續漲他的票價，缺陷避免的工作有沒有加強，只能憑廠商自己的良心了。

一個國家、社會、公司、團隊乃至於個人是否進步，只要從他們對於缺陷預防與處理的態度，便可見一斑。

***

友藏內心獨白：人行道施工，倒楣的是住戶和中華電信。我就是要挖，你咬我啊！

第三部 Java 語言的例外處理機制

13 Java 的 Try、Catch、Finally

這一章介紹 Java 對於例外處理的支援。鄉民們可能會想:「Java 的例外處理不就是 try、catch、finally 這三個關鍵字就搞定了嗎，有什麼好講的？」看下去就知道。

Java SE 7 之前

在 Java Standard Edition（SE） 7 之前，Java 的例外處理支援主要依靠 try、catch、finally 來達成。鄉民們可以使用 try、catch，try、finally、或是 try、catch、finally 這三種不同的方式來捕捉與處理例外，以及釋放使用完畢的資源。這三者的用途分別是：

- try：想要在函數裡面捕捉例外，要將可能發生例外的程式碼放到 try block 裡面，否則例外只能往外傳遞。
- catch：catch block 一定會跟在 try block 後面，如果想捕捉 try block 所發生的例外，就把這個例外的型別寫在 catch 子句裡面，例如 catch(IOException e)就可以捕捉到 IOException 這種型別的例外。如果想要捕捉多種例外類別，就必須要撰寫多個 catch block，例如：

```
1: public void try_multiple_catch(){
2:     try{
3:         throwIOException();
4:         throwSQLException();
5:     }
6:     catch(IOException e){
7:     }
8:     catch(SQLException e){
9:     }
10:    catch(Exception e){
11:      // blanket catch
12:    }
13: }
```

列表 13-1：捕捉不同類別的例外

列表 13-1 程式範例的 `try` block 呼叫 `throwIOException` 與 `throwSQLException` 這兩個函數，他們分別會丟出 `IOException` 與 `SQLException`。為了捕捉這兩個例外，需要寫兩個 `catch` block（6~9 行）。最後 `catch(Exception e)`的這個 `catch` block 有一個術語，叫做 **blanket catch**，用來捕捉除了 `Error` 型態以外的全部例外。這種寫法是因為有時候鄉民們不希望自己的函數丟出例外，因此用一個 blanket catch 把所有例外都捕捉起來。由於 `Error` 例外在 Java 語言中代表嚴重的錯誤，通常是由 Java 虛擬機器（Java Virtual Machine；JVM）所發出，例如 `OutOfMemoryError`，而且一旦發生大部分的情況會導致程式終止。因此 blanket catch 通常只捕捉 `Exception` 而非 `Throwable`。但如果鄉民們在某種特殊情況下想要連 `Error` 都一起捕捉，則可以用 `catch(Throwable e)`的方式來實作 blanket catch。

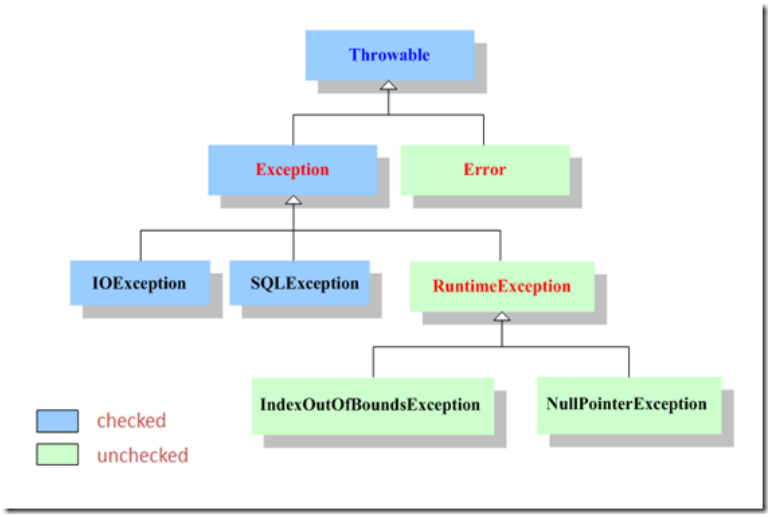


圖 13-1：Java 例外類別繼承架構（部分）

當多個 `catch` block 同時出現的時候，這些 `catch` block 出現在程式中的順序就很重要。如列表 13-2 所示，如果把 `catch(Exception e)`寫在其他 `catch` block 前面，所有的例外都會被第一個 `catch(Exception e)`給捕捉到，後面那兩個 `catch` block（9~12 行）也就沒有機會被執行到。

```
1: public void incorrect_try_multiple_catch(){
2:     try{
3:         throwIOException();
4:         throwSQLException();
5:     }
6:     catch(Exception e){
7:         // blanket catch
```

```
 8:     }
 9:     catch(IOException e){
10:     }
11:     catch(SQLException e){
12:     }
13: }
```

列表 13-2：捕捉多個例外時，越抽象的例外類別不應出現在 `catch` block 前面

- `finally`：最後這個 `finally` block 是用來釋放在 `try` block 裡面所使用的資源。不管 `try` block 中是否發生例外，只要寫了 `finally` block，程式的執行順序最後一定會跑到這裡面。就算是直接在 `try` 或是 `catch` 裡面使用 `return` 指令，`finally` block 也還是會被執行。請看列表 13-3 程式範例：

```
 1: public static int try_finally_with_return() {
 2:        try {
 3:           System.out.println("execute try block");
 4:           return 1;
 5:        }
 6:        finally{
 7:           System.out.println("execute finally block");
 8:           return 0;
 9:        }
10: }
11:
12: public static void main( String[] args )
13: {
14:     System.out.println("call try_finally_with_return : "
15:                        + try_finally_with_return());
16: }
```

列表 13-3：無論是否發生例外，`finally` block 都會被執行

鄉民們猜一下`main`呼叫`try_finally_with_return`之後會傳回1還是0？答案是傳回0，因為 `finally` block 一定會被執行，以下為列表 13-3 程式的執行結果：

```
execute try block
execute finally block
call try_finally_with_return : 0
```

世事無絕對，規則都有例外情況。當 JVM 執行到 `try` block 或是 `catch` block 裡面的程式碼，但是 JVM 自己本身卻被終止執行時，`finally` block 便不會被執行到。請參考列表 13-4 範例：

```
1: public static void main( String[] args )
2: {
3:     try {
4:         System.out.println("execute try block");
5:         System.exit(0);
6:     }
7:     finally{
8:         System.out.println("execute finally block");
9:     }
10: }
```

列表 13-4：`finally` block 沒被執行的情況

因為在 `try` block 裡面直接呼叫 `System.exit(0)`，終止整個程式的執行，所以 `finally` block 並不會被執行到。執行結果如下列所示：

```
execute try block
```

***

Java SE 7 之後

Java SE 7 之後多了幾個新的功能，先看第一個新功能叫做 **multi-catch exceptions**，可以在一個 `catch` block 裡面同時捕捉多個例外，如列表 13-5 第 10 行所示。Multi-catch exceptions 的好處，就是當不同的例外型態，都適用相同的處理方式的時候，就可以避免在不同的 `catch` block 出現重複程式碼的問題。

```
1: public void try_multi_catch(){
2:     try{
3:         URL url = new URL("http://teddysoft.tw/");
4:           BufferedReader reader = new BufferedReader(
```

```
5:              new InputStreamReader(url.openStream()));
6:          String line = reader.readLine();
7:          SimpleDateFormat format = new SimpleDateFormat("MM/DD/YY");
8:          Date date = format.parse(line);
9:      }
10:     catch(ParseException | IOException e){
11:         // handle the exception
12:     }
13: }
```

列表 13-5：Multi-catch exceptions 範例

***

第二個新功能叫做 **try-with-resources**，這個功能應該是 Java 從 C#那裡「借用」過來的。在 Java SE 7 之前，釋放資源的程式碼要寫在 `finally` block 裡面。如列表 13-6 所示，`try` block 裡面用到 `fis`、`reader` 這兩個 I/O 物件，因此在 `finally` block 需要做 cleanup 的動作。為了歸還這兩個物件所代表的資源，`finally` block 變得有點小複雜（10~24 行）。

```
1: public void try_finally() throws FileNotFoundException{
2:
3:     FileInputStream fis = null;
4:     BufferedReader reader = null;
5:     try{
6:         fis = new FileInputStream(new File("FileName"));
7:         reader = new BufferedReader(new InputStreamReader(fis));
8:         // do something
9:     }
10:    finally{
11:        if (null != fis)
12:            try {
13:                fis.close();
14:            } catch (IOException e) {
15:                // log the exception
16:            }
17:
18:        if (null != reader)
19:            try {
```

```
20:                 reader.close();
21:             } catch (IOException e) {
22:                 // log the exception
23:             }
24:     }
25: }
```

列表 13-6：Java SE 7 以前在 `finally` block 釋放資源的方式

Java SE 7 之後，可以把產生資源的程式碼寫在 `try(...)` 敘述裡面，如列表 13-7 第 2~4 行程式碼所示。如此一來，在離開整個 `try` statement 之前，JVM 會自動呼叫 `fis` 與 `reader` 物件的 `close` 函數以執行 cleanup。

```
1: public void try_with_resources() throws FileNotFoundException, IOException{
2:     try (FileInputStream fis = new FileInputStream(new File("FileName"));
3:             BufferedReader reader = new BufferedReader(
4:                     new InputStreamReader(fis)) ) {
5:         //do the normal processing logic
6:     }
7: }
```

列表 13-7：Java SE 7 支援 try-with-resources 的自動清除資源方式

看到這裡鄉民們應該會有以下疑問：「**是不是不管什麼型別的物件，只要它的 instance 是在 `try(...)` 裡面產生的，就可以自動被 JVM 清理回收？JVM 怎麼知道離開 `try` statement 之前要如何執行 cleanup 的動作？**」答案很簡單，如果鄉民們希望自己的類別要具備上述這種離開 `try` statement 之後自動 cleanup 的能力，就必須要讓該類別實作 **`AutoCloseable`** 這個 Java SE 7 之後才新增的介面。該介面很簡單，只有一個函數：

```
void close() throws Exception
```

***

最後要介紹一個一般人可能不太會注意到也不太會用到的特色，關於**重丟例外（rethrow）**的改變。列表 13-8 程式在 Java SE 7 之前是無法編譯。在 `rethrow` 函數的 `try` block 裡面，丟出一個 `IOException`。這個 `IOException` 被第 7 行的 `catch(Exception e)` 給捕捉住，然後

經過一番處理之後，被第 11 行的程式碼將捕捉下來的例外**原封不動往外丟**。由於例外被 `catch(Exception e)`給捕捉住，Java SE 7 之前的編譯器會認為這個被捕捉到的例外型別屬於 `Exception`。既然要把它往外丟，就應該在 `rethrow` 函數介面上宣告 `throws Exception` 而非 `throws IOException`。

```
 1: public class Java6InvalidRethrow  {
 2:     public static void rethrow() throws IOException {
 3:         try {
 4:             // some statements that throw an IOException
 5:             throw new IOException("Error");
 6:         }
 7:         catch (Exception e) {
 8:             /*
 9:              * Handle the exception and then rethrow it.
10:              */
11:             throw e;
12:         }
13:     }
14: }
```

列表 13-8：此程式在 Java SE 6 無法編譯成功

Java SE 7 之後的編譯器變得比較聰明一些，會自動判斷 `catch(Exception e)`所捕捉到的例外類別，其實是 `try` block 裡面所丟出來的 `IOException`，所以列表 13-8 的程式在 Java SE 7 之後是可以通過編譯的。

***

友藏內心獨白：基本知識介紹完畢，可以準備進入中階班。

14 我的例外被 Finally Block 蓋台了

在〈CH13：Java 的 Try、Catch、Finally〉介紹了 Java SE 7 之前與之後的例外處理特性，有了這些基本觀念之後，接下來可以談一個比較有趣的問題：「**finally block 丟出的例外會覆蓋掉 try block 與 catch block 所丟出的例外。**」

***

消失的例外

先看列表 14-1 程式範例，假設在 try block 中使用 MyOutputStream 會丟出一個 IOException（第 11 行），該例外將直接往外傳，因此沒有相對應的 catch block 去捕捉它。但由於 MyOutputStream 是一個 IO 物件，離開 try statement 之前要釋放它所使用的資源，因此在 finally block 中呼叫它的 close。但是 close 也可能發生錯誤而丟出一個 IOException，如果這個 IOException 也被往外丟（第 15 行），則呼叫 finallyOverridesExceptionThrownByTry 的人將只會收到 finally block 所丟出的 IOException，而原本由 try block 第 11 行所丟出的那一個 IOException 則被「**蓋台**」而消失不見了。

```
1: package tw.teddysoft.exception.ehm.cleanup;
2: import java.io.IOException;
3: import tw.teddysoft.exception.ehm.MyOutputStream;
4: public class Java7FinallyExceptionExample {
5:
6:       public static void finallyOverridesExceptionThrownByTry()
7:                                           throws IOException{
8:          MyOutputStream mos = null;
9:          try{
10:             mos = new MyOutputStream();
11:             throw new IOException("Function Failure");
12:
13:         } finally{
14:             try {
```

```
15:                 mos.close();
16:             } catch (IOException e) {
17:                 throw e; // Is this a good idea to do so?
18:             }
19:         }
20:     }
21:
22:     public static void main (String[] args) {
23:         try {
24:             finallyOverridesExceptionThrownByTry();
25:         } catch (Exception e) {
26:             e.printStackTrace();
27:             Throwable [] ta = e.getSuppressed();
28:             System.err.println("Suppressed exception size = " + ta.length);
29:         }
30:     }
31: }
```

列表 14-1：`finally` block 丟出的例外覆蓋 `try` block 所丟出的例外

接下來在 `main` 函數呼叫 `finallyOverridesExceptionThrownByTry`，stack trace 告訴我們，這個例外是由第 15 行的程式碼所丟出，也就是 `finally` block 裡面的 `mos.close()` 這一行所丟出的 `IOException` 蓋掉了原本 `try` block 所丟出的 `IOException`。

```
java.io.IOException
    at tw.teddysoft.exception.ehm.MyOutputStream.close(MyOutputStream.java:9)
    at
tw.teddysoft.exception.ehm.cleanup.Java7FinallyExceptionExample.finallyOverridesExc
eptionThrownByTry(Java7FinallyExceptionExample.java:15)
    at
tw.teddysoft.exception.ehm.cleanup.Java7FinallyExceptionExample.main(Java7FinallyEx
ceptionExample.java:24)
Suppressed exception size = 0
```

***

語意不同

看到這邊鄉民們可能會覺得這有什麼了不起的，被蓋掉就被蓋掉啊，反正眼不見為淨一向是程式設計師奉行不渝的最高指導原則。**雖然最後丟出的都是 `IOException`，但從例外處理的角度來看，這兩個 `IOException` 所代表的意義不同：**

- `try` block 所丟出的 `IOException`：表示該函數無法執行它被賦予的「正常功能」，也就是說呼叫該函數的結果是失效或是失敗的（failure）。
- `finally` block 所丟出的 `IOException`：表示 cleanup 失敗，系統可能會有資源釋放不乾淨的問題。

換句話說，這兩個 `IOException` 的語意不同，應該用不同的例外處理策略來應付：

- 功能失效（function failure）：造成 failure 的原因很可能是 component fault，常見的例外處理方式有 retry—重新執行一次原本的函數，或是考慮使用替代方案—呼叫另一個函數來取代原本失敗的函數。如果是 design fault，則需要修改程式。
- 清理失效（cleanup failure）：如果是資源釋放失敗，則需進一步研究其原因。這種情況很有可能是由於程式撰寫錯誤所造成的 design fault，需要修改程式內容。如果沒有辦法取得被呼叫元件的原始碼加以修正缺陷，則可能需要改用其他的元件來替代。

了解了這兩個 `IOException` 的語意之後，再請問各位看倌一個問題：**「當兩者（function failure 與 cleanup failure）同時發生的時候，哪一個例外比較重要，應該被保留下來？」**

理想上兩者都應該被保留，但因為只有一個例外可以往外傳遞，所以這時候就要從**呼叫者**的角度來思考那一個例外應該被保留。如果保留 cleanup failure，呼叫者會以為執行該函數的正常功能成功，只是 cleanup 失敗。但實際上卻是兩者都失敗，所以 cleanup failure 的語意就模糊不清。

但反過來，如果保留 function failure，呼叫者也無法分辨是否有 cleanup failure 的情況。兩害相權取其輕，實務上，cleanup 失敗的機率比較低，而且對呼叫者而言，最主要關心的議題是呼叫某個函數的結果是否成功，因為 cleanup 就算是失敗了，系統通常還是可以再執行一陣子，只要能夠把這個問題記錄下來即可，並不一定立刻需要在程式執行的當下去修復它。但是如果呼叫的函數失敗，就要立刻處理，不能假裝沒看到，否則整個系統的狀態可能都會錯誤。

***

怎麼辦

這個問題，不只在 Java 語言會發生，在 C#語言也有。這兩個主流的語言都建議：在 `finally` block 裡面不要丟出例外。所以在 `finally` block 如果會產生例外請把這個例外記錄到日誌檔案中，請參考列表 14-2 程式片段。

```
1: public static void preventFinallyOverridesExceptionThrownByTry()
2:                                       throws IOException{
3:     MyOutputStream mos = null;
4:     try{
5:         mos = new MyOutputStream();
6:         throw new IOException("Error");
7:     }
8:     finally{
9:         try {
10:            mos.close();
11:        } catch (IOException e) {
12:            // log the exception and do not throw it.
13:        }
14:    }
15: }
```

列表 14-2：將 `finally` block 所產生的例外寫入日誌檔

事情如果這麼簡單就不好玩了，還記得 Java SE 7 之後多了 **try-with-resources** 的功能嗎？列表 14-3 使用 try-with-resources 所重新改寫過的程式碼和列表 14-2 執行相同的功能。**但是現在問題來了，因為執行 cleanup 的程式碼現在改由 JVM 來呼叫，不需要鄉民們自己寫在 `finally` block 裡面。那如果 cleanup 失敗，哪怎麼辦？現在連把 cleanup failure 寫在日誌檔的機會都沒有了啊。**

```
1: public static void Java7TryWithResource()
2:                     throws IOException, SQLException{
3:     try( MyOutputStream mos = new MyOutputStream()){
4:         throw new IOException("Function failure");
5:     }
6: }
```

列表 14-3：採用用 try-with-resources 釋放資源，如果發生例外該怎麼辦？

這個問題比較複雜一點，請參考下一章〈CH15：Suppressed Exception：搶救例外大作戰〉。

***

友藏內心獨白：隨隨便便蓋掉別人的訊號是不對的行為喔。

15 Suppressed Exception：搶救例外大作戰

在〈CH14：我的例外被 Finally Block 蓋台了〉提到 Java SE 7 之後可以使用 try-with-resources 讓 JVM 自動幫忙執行 cleanup 的工作，但是新的 try-with-resources 功能引發一個的問題：「**如果 JVM 執行 cleanup 發生例外，這些 cleanup 例外要怎麼處理？**」Java SE 7 的作法是，在 `Throwable` 類別身上多了一個 `getSuppressed` 函數：

```
Throwable[] getSuppressed()
```

Suppressed 是壓抑、抑制、打壓的意思。顧名思義，`getSuppressed` 函數就是傳回「被抑制的例外」，也就是發生在執行 cleanup 動作所產生的例外（其他情況也會產生 suppressed exception，目前先專注在 cleanup 失敗的情況）。

請參考列表 15-1 程式片段，首先看到 `MyOutputStream` 類別，為了讓 JVM 可以自動關閉 `MyOutputStream` 所使用的資源，它實作了 `AutoCloseable` 介面。在此實作的 `close` 函數將直接丟出 `IOException`。

```
1: public class MyOutputStream implements AutoCloseable {
2:     @Override
3:     public void close() throws IOException {
4:             throw new IOException();
5:     }
6: }
```

列表 15-1：實作 `AutoCloseable` 介面，故意讓 `close` 函數丟出例外

接著採用 try-with-resources 來使用 `MyOutputStream` 類別，請參考列表 15-2 第 7 行。

```
1: package tw.teddysoft.exception.ehm;
2: import java.io.IOException;
3: import java.sql.SQLException;
4:
5: public class Java7TryWithOneResource {
```

```
 6:       public static void Java7TryWithResource()
 7:                               throws IOException, SQLException{
 8:           try( MyOutputStream mos = new MyOutputStream()){
 9:               throw new IOException("Function failure");
10:           }
11:       }
12:
13:       public static void main (String[] args) {
14:           try {
15:               Java7TryWithResource();
16:           } catch (Exception e) {
17:               e.printStackTrace();
18:               Throwable [] ta = e.getSuppressed();
19:               System.err.println("Suppressed exception size = " + ta.length);
20:           }
21:       }
22: }
```

列表 15-2：以 try-with-resources 使用 MyOutputStream 類別

在main測試一下Java7TryWithResource函數，因為在第9行丟出了一個IOException，因此呼叫Java7TryWithResource函數一定會產生例外。從下列測試結果可以看出來捕捉到的IOException是由Java7TryWithResource函數所丟出來的，而且是因為呼叫了MyOutputStream的close函數所導致例外發生。

```
java.io.IOException: Function failure
    at
tw.teddysoft.exception.ehm.Java7TryWithOneResource.Java7TryWithResource(Java7TryWit
hOneResource.java:9)
    at
tw.teddysoft.exception.ehm.Java7TryWithOneResource.main(Java7TryWithOneResource.jav
a:15)
    Suppressed: java.io.IOException
        at tw.teddysoft.exception.ehm.MyOutputStream.close(MyOutputStream.java:8)
        at
tw.teddysoft.exception.ehm.Java7TryWithOneResource.Java7TryWithResource(Java7TryWit
hOneResource.java:10)
        ... 1 more
```

```
Suppressed exception size = 1
```

本章一開始提到的問題:「如果 JVM 執行 cleanup 時發生例外,這些 cleanup 例外要怎麼處理?」看完上面的程式範例,答案揭曉:呼叫者(`main` 函數)所捕捉到的例外是由 `Java7TryWithResource` 函數所丟出來的 `IOException`(代表 function failure)。至於 `MyOutputStream` 的 `close` 函數所丟出來的 `IOException`(代表 cleanup failure)則變成了 function failure 例外(第一個 `IOException`)的 suppressed exception。

***

橋歸橋,路歸路(理論上)

這下子例外的語意就清楚很多了,一個函數在使用 try-with-resources 的情況下丟出例外,這個例外代表 function failure。如果該 function failure 例外還附帶有 suppressed exception,則表示 cleanup 動作也一併失敗[20]。捕捉到例外的人,除了要處理 function failure,如果也想處理 suppressed exception,則可以呼叫例外物件的 `getSuppressed` 函數來逐一讀出每一個 suppressed exception。

現在還剩下一個問題:如果 `try` block 或是 `catch` block 沒有丟出例外,而只有 JVM 在採用 try-with-resources 的情況下執行 cleanup 動作發生多個例外,那麼這些 cleanup 例外是由誰來壓制誰?也就是說呼叫者最後會收到哪一個 cleanup 例外?請參考列表 15-3 程式片段:`MyOutputStream`、`MyConnection`、`MyInputStream` 的 `close` 函數分別會丟出 `IOException`、`SQLException`、`FileNotFoundException`。

```
1: package tw.teddysoft.exception.ehm;
2: import java.io.IOException;
3: import java.sql.SQLException;
4:
5: public class Java7TryWithMultipleResources {
6:       public static void Java7TryWithResources()
7:                           throws IOException, SQLException{
```

[20] Function failure exception 夾帶著 suppressed exception 並不一定都表示 cleanup 失敗,也有可能是 catch block 執行錯誤狀態回復動作失敗,在本章只討論 cleanup 失敗的狀況。

```
 8:          try( MyOutputStream mos = new MyOutputStream();
 9:               MyConnection mc = new MyConnection();
10:               MyInputStream mis = new MyInputStream()){
11:          }
12:      }
13:
14:      public static void main (String[] args) {
15:          try {
16:              Java7TryWithResources();
17:          } catch (Exception e) {
18:              e.printStackTrace();
19:              Throwable [] ta = e.getSuppressed();
20:              System.err.println("Suppressed exception size = " + ta.length);
21:          }
22:      }
23: }
```

列表 15-3：try-with-resources 使用多個資源物件，各自的 close 函數丟出不同的例外

在 main 函數測試一下 Java7TryWithResources 函數，結果發現，被丟出來的例外是 FileNotFoundException，而 suppressed exception 則是 SQLException 與 IOException。也就是說，看起來 JVM 是依據使用資源的順序，反向釋放這些資源（把資源物件依據宣告的順序，逐一放入堆疊之中，最後要執行 cleanup 動作再從堆疊裡面逐一拿出物件，並且呼叫這些物件的 close 函數）。

```
java.io.FileNotFoundException
    at tw.teddysoft.exception.ehm.MyInputStream.close(MyInputStream.java:9)
    at
tw.teddysoft.exception.ehm.Java7TryWithMultipleResources.Java7TryWithResources(Java
7TryWithMultipleResources.java:11)
    at
tw.teddysoft.exception.ehm.Java7TryWithMultipleResources.main(Java7TryWithMultipleR
esources.java:16)
    Suppressed: java.sql.SQLException
        at tw.teddysoft.exception.ehm.MyConnection.close(MyConnection.java:9)
        ... 2 more
    Suppressed: java.io.IOException
        at tw.teddysoft.exception.ehm.MyOutputStream.close(MyOutputStream.java:8)
```

```
        ... 2 more
Suppressed exception size = 2
```

***

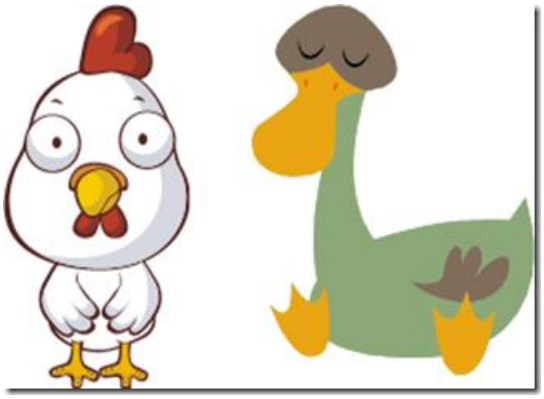
細心一點的鄉民，看到這裡應該會發現一個問題：**有了 suppressed exception 之後，雖然 `try` block 與 `catch` block 所丟出來的例外不會被 cleanup 例外蓋台，但是例外的語意對捕捉到例外的人來講還是不夠清楚。**如列表 15-3 所示，一個擁有 suppressed exception 的例外，可能是捕捉到 `try` block 或是 `catch` block 所丟出的 function failure 例外，也可能是多個 cleanup 例外所形成的結果（代表 cleanup failure）。

怎麼辦？這是一個**例外類別設計**的問題，請參考〈CH17：自己製作 Suppressed Exception〉。

***

友藏內心獨白：仔細讀，會讀出一點心得。

16 表達清楚的 Cleanup Failure 語意



語意不清，彷彿雞同鴨講。

在〈CH15：Suppressed Exception：搶救例外大作戰〉提到 cleanup failure 的語意在 Java SE 7 新增 try-with-resources 敘述之後依舊還是不太清楚。為了解決這個問題，Teddy 提出一個建議作法給鄉民們參考一下。

***

定義 Cleanup Failure 語意

這個問題的根本原因就是 Java 語言內建的例外類別沒有一個是用來表示 cleanup failure，因此請鄉民們自行定義一個 `CleanupException` 用來代表 cleanup failure，如列表 16-1 所示。

```
1: package tw.teddysoft.exception.ehm.cleanup;
2:
3: public class CleanupException extends RuntimeException {
4:     public CleanupException(Exception e) {
5:         super(e);
6:     }
7: }
```

列表 16-1：定義 `CleanupException`

CleanupException 繼承自 RuntimeException，是一個 unchecked exception。為什麼不是 checked exception？原因有二：

- 因為任何需要釋放資源的地方都可能會丟出 CleanupException，Teddy 在之前曾經提到，從例外處理程序的角度來看，主要關注的是 function failure。如果把 CleanupException 設計成 checked exception，則會強迫例外處理程序隨時都要關注這個比較次要的例外狀況，最後可能會導致這個例外反射性地被忽略，產生 *Ignored Checked Exception* 這個壞味道（請參考〈CH34：例外處理壞味道〉），反而降低系統強健度並且增加日後除錯的困難度。
- 再者，cleanup 會失敗，可能是 design fault 也可能是 component fault。理論上，cleanup 的動作應該要儘量可靠，否則系統三不五時就會產生資源不足的問題，變得十分不可靠。實務上，Teddy 在呼叫 Java 系統內建資源類別的 close 函數也鮮少發生例外。既然發生 cleanup failure 的機率很低，把 CleanupException 設計成 runtime exception，姑且先將其視為 design fault 來看待。當應用程式真正被測試或使用的時候，如果真的有發生 CleanupException，這個例外會被系統最上層的主程式寫到日誌檔，或是立即顯示讓使用者得知。透過日誌檔記錄或是使用者的問題回報，開發人員有一個比較具體的系統操作情境，可以協助研究如何重複產生以及排除 CleanupException 的再度發生。從整個軟體生命週期的角度來看，如此也可以提升系統強健度。

***

修改 close 函數所丟出的例外

有了 CleanupException 之後，接下來的事情就好辦了。原本的 MyConnection、MyInputStream、MyOutputStream 這三個類別的 close 函數，分別丟出 SQLException、FileNotFoundException、IOException，把這些不同型別的例外全部改成 CleanupException 就大事底定。

```
1: package tw.teddysoft.exception.ehm.cleanup;
2: import java.sql.SQLException;
3:
4: public class MyConnection implements AutoCloseable {
5:
6:     @Override
7:     public void close() throws CleanupException {
```

```
 8:         try{
 9:             throw new SQLException();
10:         } catch(Exception e){
11:             throw new CleanupException(e);
12:         }
13:     }
14: }
```

列表 16-2：MyConnection 丟出 CleanupException

```
 1: package tw.teddysoft.exception.ehm.cleanup;
 2: import java.io.FileNotFoundException;
 3:
 4: public class MyInputStream implements AutoCloseable {
 5:     @Override
 6:     public void close() throws CleanupException {
 7:         try{
 8:             throw new FileNotFoundException();
 9:         } catch(Exception e){
10:             throw new CleanupException(e);
11:         }
12:     }
13: }
```

列表 16-3：MyInputStream 丟出 CleanupException

```
 1: package tw.teddysoft.exception.ehm.cleanup;
 2: import java.io.IOException;
 3:
 4: public class MyOutputStream implements AutoCloseable {
 5:     @Override
 6:     public void close() throws CleanupException {
 7:         try{
 8:             throw new IOException();
 9:         } catch(Exception e){
10:             throw new CleanupException(e);
11:         }
12:     }
13: }
```

列表 16-4：MyOutputStream 丟出 CleanupException

如列表 16-2、列表 16-3、列表 16-4 所示，雖然三個不同資源類別的 close 函數都是丟出 CleanupException，但是原始的例外物件：SQLException、FileNotFoundException、IOException，還是被串接在 CleanupException 身上。也就是說，例外處理程式還是可以從 CleanupException 身上知道當初引發 CleanupException 的原因。

***

Cleanup Failure 的語意清楚了嗎？

接下來看兩個例子就可以知道在丟出 CleanupException 之後，cleanup failure 語意是否清楚。首先參考列表 16-5，在 try-with-resources 裡面使用 MyOutputStream、MyConnection、MyInputStream 這三個資源物件。然後在 try block 故意丟出一個 IOException。

```
1: package tw.teddysoft.exception.ehm.cleanup;
2: import java.io.IOException;
3:
4: public class MultipleCleanupFailuresExample {
5:
6:       public static void tryWithResourcesThrowsCleanupFailure()
7:                                    throws IOException{
8:
9:          try( MyOutputStream mos = new MyOutputStream();
10:              MyConnection mc = new MyConnection();
11:              MyInputStream mis = new MyInputStream()){
12:            throw new IOException("Function Failure");
13:         }
14:      }
15:
16:      public static void main (String[] args) {
17:         try {
18:            tryWithResourcesThrowsCleanupFailure();
19:         } catch (Exception e) {
20:            e.printStackTrace();
```

```
21:                 Throwable [] ta = e.getSuppressed();
22:                 System.err.println("Suppressed exception size = " + ta.length);
23:             }
24:         }
25: }
```

列表 16-5：在 try block 丟出例外以測試 cleanup failure 語意

在 main 程式中呼叫 tryWithResourcesThrowsCleanupFailure 函數，被捕捉到的是由 try block 所丟出的 failure exception，並且夾帶著三個 suppressed exception，型態都是 CleanupException。正如剛剛的設計，而這三個 CleanupException 分別由 FileNotFoundException、SQLException、IOException 所引起。請參考下面這段有點像是「亂碼」的執行結果：

```
java.io.IOException: Function Failure
    at
tw.teddysoft.exception.ehm.cleanup.MultipleCleanupFailuresExample.tryWithResourcesT
hrowsCleanupFailure(MultipleCleanupFailuresExample.java:12)
    at
tw.teddysoft.exception.ehm.cleanup.MultipleCleanupFailuresExample.main(MultipleClea
nupFailuresExample.java:18)
    Suppressed: tw.teddysoft.exception.ehm.cleanup.CleanupException:
java.io.FileNotFoundException
        at
tw.teddysoft.exception.ehm.cleanup.MyInputStream.close(MyInputStream.java:10)
        at
tw.teddysoft.exception.ehm.cleanup.MultipleCleanupFailuresExample.tryWithResourcesT
hrowsCleanupFailure(MultipleCleanupFailuresExample.java:13)
        ... 1 more
    Caused by: java.io.FileNotFoundException
        at
tw.teddysoft.exception.ehm.cleanup.MyInputStream.close(MyInputStream.java:8)
        ... 2 more
    Suppressed: tw.teddysoft.exception.ehm.cleanup.CleanupException:
java.sql.SQLException
        at
tw.teddysoft.exception.ehm.cleanup.MyConnection.close(MyConnection.java:11)
        at
```

```
tw.teddysoft.exception.ehm.cleanup.MultipleCleanupFailuresExample.tryWithResourcesT
hrowsCleanupFailure(MultipleCleanupFailuresExample.java:13)
        ... 1 more
    Caused by: java.sql.SQLException
        at
tw.teddysoft.exception.ehm.cleanup.MyConnection.close(MyConnection.java:9)
        ... 2 more
    Suppressed: tw.teddysoft.exception.ehm.cleanup.CleanupException:
java.io.IOException
        at
tw.teddysoft.exception.ehm.cleanup.MyOutputStream.close(MyOutputStream.java:10)
        at
tw.teddysoft.exception.ehm.cleanup.MultipleCleanupFailuresExample.tryWithResourcesT
hrowsCleanupFailure(MultipleCleanupFailuresExample.java:13)
        ... 1 more
    Caused by: java.io.IOException
        at
tw.teddysoft.exception.ehm.cleanup.MyOutputStream.close(MyOutputStream.java:8)
        ... 2 more

Suppressed exception size = 3
```

***

接著再看原本會引起 cleanup failure 語意不清的情況，就是 `try` block 裡面**沒有**丟出例外，但是 JVM 呼叫 `close` 函數時會發生例外的例子，請參考列表 16-6。

```
 1: package tw.teddysoft.exception.ehm.cleanup;
 2: import java.io.IOException;
 3:
 4: public class MultipleCleanupFailuresExample2 {
 5:
 6:       public static void tryWithResourcesThrowsCleanupFailure()
 7:                                    throws IOException{
 8:
 9:          try( MyOutputStream mos = new MyOutputStream();
10:               MyConnection mc = new MyConnection();
11:               MyInputStream mis = new MyInputStream()){
```

```
12:             }
13:         }
14:
15:         public static void main (String[] args) {
16:             try {
17:                 tryWithResourcesThrowsCleanupFailure();
18:             } catch (Exception e) {
19:                 e.printStackTrace();
20:                 Throwable [] ta = e.getSuppressed();
21:                 System.err.println("Suppressed exception size = " + ta.length);
22:             }
23:         }
24: }
```

列表 16-6：在 close 函數丟出例外以測試 cleanup failure 語意

在 main 函數中呼叫 tryWithResourcesThrowsCleanupFailure 函數，結果完全符合設計的預期，被捕捉到的例外是 CleanupException，它夾帶另外兩個 suppressed exception，型態都是 CleanupException。而這三個 CleanupException 分別由 FileNotFoundException、SQLException、IOException 所引起：

```
tw.teddysoft.exception.ehm.cleanup.CleanupException: java.io.FileNotFoundException
    at
tw.teddysoft.exception.ehm.cleanup.MyInputStream.close(MyInputStream.java:10)
    at
tw.teddysoft.exception.ehm.cleanup.MultipleCleanupFailuresExample2.tryWithResources
ThrowsCleanupFailure(MultipleCleanupFailuresExample2.java:12)
    at
tw.teddysoft.exception.ehm.cleanup.MultipleCleanupFailuresExample2.main(MultipleCle
anupFailuresExample2.java:17)
    Suppressed: tw.teddysoft.exception.ehm.cleanup.CleanupException:
java.sql.SQLException
        at
tw.teddysoft.exception.ehm.cleanup.MyConnection.close(MyConnection.java:11)
        ... 2 more
    Caused by: java.sql.SQLException
        at
```

```
tw.teddysoft.exception.ehm.cleanup.MyConnection.close(MyConnection.java:9)
        ... 2 more
    Suppressed: tw.teddysoft.exception.ehm.cleanup.CleanupException:
java.io.IOException
        at
tw.teddysoft.exception.ehm.cleanup.MyOutputStream.close(MyOutputStream.java:10)
        ... 2 more
    Caused by: java.io.IOException
        at
tw.teddysoft.exception.ehm.cleanup.MyOutputStream.close(MyOutputStream.java:8)
        ... 2 more
Caused by: java.io.FileNotFoundException
    at tw.teddysoft.exception.ehm.cleanup.MyInputStream.close(MyInputStream.java:8)
    ... 2 more

Suppressed exception size = 2
```

***

看到這裡如果鄉民們還沒把本書拿去網拍的話，應該可以很清楚的區分 function failure 和 cleanup failure 的語意。這樣子不但有助於例外處理程式的設計，對於日後加速除錯的進行也很有幫助。

但是，眼尖的人應該還是會發現一個問題：「**可是 Java 語言內建的資源物件，它們實作 `AutoCloseable` 介面 `close` 函數的做法，丟出來的都不是 `CleanupException` 啊？**」Teddy 認為這的確是一個 Java 例外處理設計可以加強的地方。有沒有解？請繼續讀下去。

***

友藏內心獨白：這裡面還是有點學問滴。

17 自己製作 Suppressed Exception

最近食安問題多，自己做三餐比較安全。

複習

從〈CH13：Java 的 Try、Catch、Finally〉寫到現在，這幾篇的文章內容充滿程式碼，還沒把本書拿去網拍的鄉民們差不多也看到頭暈了。再繼續往下之前，Teddy 先幫各位整理一下之前提談過的 Java SE 7 的 `try` statement 關於清除資源的特性：

1. 用 try-with-resources 來使用資源物件，在離開 `try` statement 之前，JVM 會自動幫忙關閉這些資源。
2. 如果 `try` block 或 `catch` block 丟出例外，且 JVM 在關閉資源時也產生例外，JVM 會把這些 cleanup 例外加到之前產生的例外之中，這些被附加上去的例外就稱之為 suppressed exception。
3. Try-with-resources 的功能雖然解決了 cleanup 例外蓋掉 `try` block 或 `catch` block 所產生例外地這個問題，但是最終所丟出去的例外，到底是代表 function failure 或 cleanup failure，其語意還是模糊不清。
4. 藉由讓 `AutoCloseable` 介面的 `close` 丟出 `CleanupException` 這個使用者自訂的 unchecked exception，可以明確區分 try-with-resources 的 function failure 與 cleanup failure 語意。但是，可惜 Java 內建的資源物件，其 `close` 並不會丟出 `CleanupException`，所以在現況之下 cleanup failure 例外還是語意不清。

5. 第 2 點所提到的功能，只有在 try-with-resources 的情況下 JVM 會幫忙產生 suppressed exception，如果是傳統的寫法，finally block 所產生的例外，在 Java SE 7 之後的行為依然沒變，還是會把 try block 或是 catch block 所產生的例外給「蓋台」。

複習完畢，本章的目的想要解決第 4 點與第 5 點所提到的問題。首先寫段程式確定一下第 5 點所描述的現象。請參考列表 17-1 的 Java SE 7 程式碼，在 finally block 中直接丟出 IOException（第 17 行）。

```
 1: package tw.teddysoft.exception.ehm.cleanup;
 2: import java.io.IOException;
 3: import tw.teddysoft.exception.ehm.MyOutputStream;
 4:
 5: public class Java7FinallyExceptionExample {
 6:
 7:       public static void finallyOverridesExceptionThrownByTry()
 8:                                           throws IOException{
 9:           MyOutputStream mos = null;
10:           try{
11:               mos = new MyOutputStream();
12:               throw new IOException("Function Failure");
13:           } finally{
14:               try {
15:                   mos.close();
16:               } catch (IOException e) {
17:                   throw e; // Is this a good idea to do so?
18:               }
19:           }
20:       }
21:
22:       public static void main (String[] args) {
23:           try {
24:               finallyOverridesExceptionThrownByTry();
25:           } catch (Exception e) {
26:               e.printStackTrace();
27:               Throwable [] ta = e.getSuppressed();
28:               System.err.println("Suppressed exception size = " + ta.length);
29:           }
```

```
30:         }
31: }
```

列表 17-1：在 finally block 直接丟出例外（使用 Java SE 7）

在 main 函數測試一下 finallyOverridesExceptionThrownByTry 函數，結果發現，沒有任何 suppressed exception，捕捉到的例外是由 finally block 所丟出來的 IOException，原本 try block 所丟出的 IOException 已經被蓋台了。Java SE 7 的 finally block 的行為模式和之前版本並無不同。

```
java.io.IOException
    at tw.teddysoft.exception.ehm.MyOutputStream.close(MyOutputStream.java:8)
    at
tw.teddysoft.exception.ehm.cleanup.Java7FinallyExceptionExample.finallyOverridesExc
eptionThrownByTry(Java7FinallyExceptionExample.java:15)
    at
tw.teddysoft.exception.ehm.cleanup.Java7FinallyExceptionExample.main(Java7FinallyEx
ceptionExample.java:24)
Suppressed exception size = 0
```

***

老師傅手工打造

接下來將會使用到 MyConnection、MyInputStream、MyOutputStream 這三個類別，由於要模擬 Java 內建的資源物件，因此這三個類別的 close 函數不再丟出 Teddy 在前一章所自訂的 CleanupException，而是丟出 Java 內建的 SQLException、FileNotFoundException、IOException。

```
1: package tw.teddysoft.exception.ehm.cleanup.hmse;
2:
3: import java.sql.SQLException;
4:
5: public class MyConnection implements AutoCloseable {
6:
```

```
 7:     @Override
 8:     public void close() throws SQLException {
 9:         throw new SQLException();
10:     }
11: }
```

列表 17-2：MyConnection 丟出 SQLException

```
 1: package tw.teddysoft.exception.ehm.cleanup.hmse;
 2:
 3: import java.io.FileNotFoundException;
 4:
 5: public class MyInputStream implements AutoCloseable {
 6:
 7:     @Override
 8:     public void close() throws FileNotFoundException {
 9:         throw new FileNotFoundException();
10:     }
11: }
```

列表 17-3：MyInputStream 丟出 FileNotFoundException

```
 1: package tw.teddysoft.exception.ehm.cleanup.hmse;
 2:
 3: import java.io.IOException;
 4:
 5: public class MyOutputStream implements AutoCloseable {
 6:
 7:     @Override
 8:     public void close() throws IOException {
 9:         throw new IOException();
10:     }
11: }
```

列表 17-4：MyOutputStream 丟出 IOException

接下來先看 function failure 的例子，之前已經說明過 try-with-resources 無法清楚的區別 function failure 與 cleanup failure 兩者的語意，因此只好用傳統的 `finally` block 加上一些程式設計技巧來嘗試解決這個問題。請先看列表 17-5 程式碼。

```
1: package tw.teddysoft.exception.ehm.cleanup.hmse;
2: import java.io.IOException;
3:
4: public class FunctionFailureExample {
5:
6:       public static void useCleanerWithFailureException() throws IOException {
7:          MyOutputStream mos = null;
8:          MyConnection mc = null;
9:          MyInputStream mis = null;
10:         Cleaner cln = Cleaner.newInstance();
11:         try{
12:            mos = cln.push(new MyOutputStream());
13:            mc =  cln.push(new MyConnection());
14:            mis = cln.push(new MyInputStream());
15:            throw new IOException("Function Failure");
16:         } catch(Exception e){
17:            cln.setLeadException(e);
18:            throw e;
19:         }
20:         finally{
21:            cln.clear();
22:         }
23:      }
24:
25:      public static void main (String[] args) {
26:         try {
27:            useCleanerWithFailureException();
28:         } catch (Exception e) {
29:            e.printStackTrace();
30:            Throwable [] ta = e.getSuppressed();
31:            System.err.println("Suppressed exception size = " + ta.length);
32:         }
33:      }
34: }
```

列表 17-5：用手動方式模擬 try-with-resources，在 `try` block 丟出例外

整個設計與使用概念其實很簡單，只有三個步驟：

1. Teddy 設計了一個 Cleaner 類別，利用這個類別來模擬 JVM 把資源物件 push 到一個堆疊的動作（第 12~14 行）。
2. 由於 try block 與 catch block 都可能會丟出例外（在此稱之為 lead exception），為了有機會可以把 cleanup exception 加入到 lead exception 裡面，因此第 16~19 行用一個 blanket catch 捕捉除了 Error 以外的全部例外類別。然後，呼叫 Cleaner 類別的 setLeadException 函數，把捕捉到的 lead exception 先存起來。等一下如果在 finally block 發生 cleanup 例外，才有辦法把 cleanup 例外加到 lead exception 裡面。
3. 最後，在 finally block 呼叫 Cleaner 類別的 clear 函數，執行資源釋放的動作。

接下來看一下執行結果，的確是捕捉到了代表 function failure 的 IOException，而三個型別屬於 CleanupException 的 suppressed exception 也都被正確加入。

```
java.io.IOException: Function Failure
    at
tw.teddysoft.exception.ehm.cleanup.hmse.FunctionFailureExample.useCleanerWithFailur
eException(FunctionFailureExample.java:15)
    at
tw.teddysoft.exception.ehm.cleanup.hmse.FunctionFailureExample.main(FunctionFailure
Example.java:27)
    Suppressed: tw.teddysoft.exception.ehm.cleanup.hmse.CleanupException:
java.io.FileNotFoundException
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:36)
        at
tw.teddysoft.exception.ehm.cleanup.hmse.FunctionFailureExample.useCleanerWithFailur
eException(FunctionFailureExample.java:21)
        ... 1 more
    Caused by: java.io.FileNotFoundException
        at
tw.teddysoft.exception.ehm.cleanup.hmse.MyInputStream.close(MyInputStream.java:9)
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:32)
        ... 2 more
    Suppressed: tw.teddysoft.exception.ehm.cleanup.hmse.CleanupException:
java.sql.SQLException
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:36)
        at
tw.teddysoft.exception.ehm.cleanup.hmse.FunctionFailureExample.useCleanerWithFailur
```

```
eException(FunctionFailureExample.java:21)
        ... 1 more
    Caused by: java.sql.SQLException
        at
tw.teddysoft.exception.ehm.cleanup.hmse.MyConnection.close(MyConnection.java:9)
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:32)
        ... 2 more
    Suppressed: tw.teddysoft.exception.ehm.cleanup.hmse.CleanupException:
java.io.IOException
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:36)
        at
tw.teddysoft.exception.ehm.cleanup.hmse.FunctionFailureExample.useCleanerWithFailur
eException(FunctionFailureExample.java:21)
        ... 1 more
    Caused by: java.io.IOException
        at
tw.teddysoft.exception.ehm.cleanup.hmse.MyOutputStream.close(MyOutputStream.java:9)
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:32)
        ... 2 more

Suppressed exception size = 3
```

***

Function failure 的語意正確了，接下來看一下 cleanup failure 的語意是否正確，請參考列表 17-6，程式內容和列表 17-5 的 function failure 例子差不多，只是現在不讓 `try` block 丟出例外，只有 `finally` block 會丟出例外。

```
 1: package tw.teddysoft.exception.ehm.cleanup.hmse;
 2: import java.io.IOException;
 3:
 4: public class CleanupFailureExample {
 5:
 6:       public static void useCleanerWithCleanupException() throws IOException {
 7:           MyOutputStream mos = null;
 8:           MyConnection mc = null;
 9:           MyInputStream mis = null;
10:           Cleaner cln = Cleaner.newInstance();
```

```
11:           try{
12:               mos = cln.push(new MyOutputStream());
13:               mc =  cln.push(new MyConnection());
14:               mis = cln.push(new MyInputStream());
NA:               throw new IOException("Function Failure");
15:           } catch(Exception e){
16:               cln.setLeadException(e);
17:               throw e;
18:           }
19:           finally{
20:               cln.clear(); // 現在只有這一行會丟出例外
21:           }
22:       }
23:
24:       public static void main (String[] args) {
25:           try {
26:               useCleanerWithCleanupException();
27:           } catch (Exception e) {
28:               e.printStackTrace();
29:               Throwable [] ta = e.getSuppressed();
30:               System.err.println("Suppressed exception size = " + ta.length);
31:           }
32:       }
33: }
```

列表 17-6：用手動方式模擬 try-with-resources，只有 `finally` block 丟出例外

執行結果，的確是捕捉到代表 cleanup failure 的 `CleanupException`，而且 suppressed exception 現在只剩二個，也都被正確加入。

```
tw.teddysoft.exception.ehm.cleanup.hmse.CleanupException:
java.io.FileNotFoundException
    at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:39)
    at
tw.teddysoft.exception.ehm.cleanup.hmse.CleanupFailureExample.useCleanerWithCleanup
Exception(CleanupFailureExample.java:20)
    at
tw.teddysoft.exception.ehm.cleanup.hmse.CleanupFailureExample.main(CleanupFailureEx
ample.java:26)
```

```
    Suppressed: tw.teddysoft.exception.ehm.cleanup.hmse.CleanupException:
java.sql.SQLException
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:42)
        ... 2 more
    Caused by: java.sql.SQLException
        at
tw.teddysoft.exception.ehm.cleanup.hmse.MyConnection.close(MyConnection.java:9)
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:32)
        ... 2 more
    Suppressed: tw.teddysoft.exception.ehm.cleanup.hmse.CleanupException:
java.io.IOException
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:42)
        ... 2 more
    Caused by: java.io.IOException
        at
tw.teddysoft.exception.ehm.cleanup.hmse.MyOutputStream.close(MyOutputStream.java:9)
        at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:32)
        ... 2 more
Caused by: java.io.FileNotFoundException
    at
tw.teddysoft.exception.ehm.cleanup.hmse.MyInputStream.close(MyInputStream.java:9)
    at tw.teddysoft.exception.ehm.cleanup.hmse.Cleaner.clear(Cleaner.java:32)
    ... 2 more
Suppressed exception size = 2
```

***

最後看一下

```
 1: package tw.teddysoft.exception.ehm.cleanup.hmse;
 2: import java.util.Stack;
 3:
 4: public class Cleaner {
 5:     private Stack<AutoCloseable> stack;
 6:     private Throwable leadException;
 7:     private CleanupException outgoingCleanupException;
 8:
 9:     private Cleaner(){
10:         stack = new Stack<AutoCloseable>();
11:         leadException = null;
```

```
12:         outgoingCleanupException = null;
13:     }
14:
15:     static public Cleaner newInstance(){
16:         return new Cleaner();
17:     }
18:
19:     public <T extends AutoCloseable> T push(T obj){
20:         return (T) stack.push(obj);
21:     }
22:
23:     public void setLeadException(Throwable lead){
24:         leadException = lead;
25:     }
26:
27:     public void clear(){
28:         while(!stack.isEmpty()){
29:             AutoCloseable auto = stack.pop();
30:             try{
31:                 if(null != auto)
32:                     auto.close();
33:             }
34:             catch(Exception e){
35:                 if (hasLeadException()) {
36:                     leadException.addSuppressed(new CleanupException(e));
37:                 }
38:                 else if (!hasOutgoingCleanupException()) {
39:                     outgoingCleanupException = new CleanupException(e);
40:                 }
41:                 else{
42:                     outgoingCleanupException.
43:                                 addSuppressed(new CleanupException(e));
44:                 }
45:             }
46:         }
47:
48:         if (hasOutgoingCleanupException())
49:             throw outgoingCleanupException;
```

```
50:     }
51:
52:     public boolean hasLeadException(){
53:         return (null != leadException) ? true : false;
54:     }
55:
56:     public boolean hasOutgoingCleanupException(){
57:         return (null != outgoingCleanupException) ? true : false;
58:     }
59: }
```

列表 17-7 的 Cleaner 類別程式碼，運作原理剛剛已經說明過了，程式也很簡單，不再贅述。

```
1: package tw.teddysoft.exception.ehm.cleanup.hmse;
2: import java.util.Stack;
3:
4: public class Cleaner {
5:     private Stack<AutoCloseable> stack;
6:     private Throwable leadException;
7:     private CleanupException outgoingCleanupException;
8:
9:     private Cleaner(){
10:         stack = new Stack<AutoCloseable>();
11:         leadException = null;
12:         outgoingCleanupException = null;
13:     }
14:
15:     static public Cleaner newInstance(){
16:         return new Cleaner();
17:     }
18:
19:     public <T extends AutoCloseable> T push(T obj){
20:         return (T) stack.push(obj);
21:     }
22:
23:     public void setLeadException(Throwable lead){
24:         leadException = lead;
25:     }
26:
```

```
27:     public void clear(){
28:         while(!stack.isEmpty()){
29:             AutoCloseable auto = stack.pop();
30:             try{
31:                 if(null != auto)
32:                     auto.close();
33:             }
34:             catch(Exception e){
35:                 if (hasLeadException()) {
36:                     leadException.addSuppressed(new CleanupException(e));
37:                 }
38:                 else if (!hasOutgoingCleanupException()) {
39:                     outgoingCleanupException = new CleanupException(e);
40:                 }
41:                 else{
42:                     outgoingCleanupException.
43:                                 addSuppressed(new CleanupException(e));
44:                 }
45:             }
46:         }
47:
48:         if (hasOutgoingCleanupException())
49:             throw outgoingCleanupException;
50:     }
51:
52:     public boolean hasLeadException(){
53:         return (null != leadException) ? true : false;
54:     }
55:
56:     public boolean hasOutgoingCleanupException(){
57:         return (null != outgoingCleanupException) ? true : false;
58:     }
59: }
```

列表 17-7：Cleaner 類別程式碼

***

例子看到這邊，鄉民們可能會覺得：有必要為了區分 function failure 與 cleanup failure 把程式寫成這個樣子嗎？直接用 try-with-resources 不是簡單又方便嗎？這個問題其實 Teddy 也沒有什麼「正確答案」。以前念書的時候有一陣子在研究例外處理，發現例外處理很複雜，其中原因很多，而程式語言沒有清楚的區分 function failure 與 cleanup failure 就是其中的原因之一。

請回想一下自己寫過的 Java 或 C#程式，請問大家都怎麼處理 close 函數所丟出的例外？大部分的人不是忽略它，就是反射性的印出例外訊息然後這個訊息就被掩沒在其他更多的訊息之中。到底呼叫 close 函數發生例外的機率高不高？依據 Teddy 的經驗，不高。但是一旦發生卻又被有意無意忽略，則這樣的問題是很難被找出來的。

好幾年前 Teddy 曾經用過一個第三方廠商所寫的免費 JDBC 驅動程式，用來連接某個資料庫。在那個年代，有些 JDBC 驅動程式是需要付費的，所以 Teddy 找了這個免費的驅動程式來使用。剛開始一切功能都很正常，一直到系統上線之後，發現程式跑了一陣子就會把資料庫的連線數目全部使用完畢，造成後續的連線要求全部失敗，導致需要重新啟動資料庫。

發生問題之後，Teddy 以為是因為釋放資源的程式碼沒有寫好，後來搞了好久，才發現在 `finally` block 呼叫 `Connection` 物件的 `close` 函數丟出了例外，但這個例外直接被忽略，因此一直都沒被注意到。經一再測試後發現很可能是這個免費的 JDBC 驅動程式存在 design fault，最後只好花錢換另一個 JDBC 驅動程式才解決這個問題。

可能是因為有這個經驗，所以 Teddy 對於 cleanup failure 的重要性才有著自己一點小小的堅持。鄉民們即使不打算像本章中所介紹的方法，嚴格區分 function failure 與 cleanup failure，在讀完這幾章的內容之後，至少也應該知道現有 Java 與 C#語言的 `try-catch-finally` 例外處理機制在例外語意上存在模糊不清的限制，儘量避免造成 cleanup failure 覆蓋 function failure 的情況。

***

友藏內心獨白：程式要不要這樣寫都可以再討論，但解題思考模式可以參考一下。

18 Try、Catch、Finally 的責任分擔

這一章來談一個基本的問題：try、catch、finally 各要負擔怎樣的責任？物件導向設計要解決最基本的問題之一就是**責任分派（responsibility assignment）**，以往責任分派所探討的重點都在架構、模組、類別與函數等層級上面。要做好例外處理設計，應該要重新審視一下常見的例外處理的最小單位：try、catch、finally 這三個 block 各自應該負擔那些工作。

鄉民甲：這還要問嗎？Java 語言不是已經告訴我們，如果程式碼會丟出例外，而且你想處理這些例外，就要把程式碼寫在 try block 裡面，然後用 catch block 來捕捉例外並且加以處理。最後，在 finally block 裡面把使用到的資源給釋放掉。

Teddy：你說的沒錯，這也是一般人所認知的「例外處理」。但請仔細想一下，這樣的解釋是否有助於開發人員真正著手撰寫例外處理程式？就拿 catch block 所負擔的責任來講：「捕捉例外並且加以處理」。前面這個「捕捉例外」比較沒爭議，但是後面的「加以處理」就有問題了。所謂「加以處理」到底要怎樣處理？

鄉民甲：這個你也不知道喔，很簡單啊，「加以處理」就是呼叫例外物件的 printStackTrace 函數把例外給印出來，或是把捕捉到的例外轉成另外一種例外型別，然後再往外丟。也可能在 catch block 裡面提供替代方案，取代原本 try block 的實作。

Teddy：還有嗎？

鄉民甲：大致上就這樣了，難道還能生出一朵花出來？

***

關於這個問題，Teddy 在學生時代研究例外處理的當下，也是苦思良久。思考的心路歷程就省略，直接說明結論。Teddy 認為 try、catch、finally 各應負擔的責任有：

- Try

- 實作函數所被賦予對外提供的服務。實作的方式可能只有一種，也可以包含多個替代方案，以便在發生例外時輪流使用各種不同的方案。
- 為狀態回復做準備，例如開始一個資料庫交易（begin transaction）或是製作檢查點（check point）。
- 回報系統錯誤。如果發生錯誤導致不能提供正確的服務，且該錯誤無法在函數內部處理，則可以在此直接回報服務失效。。

- Catch
  - 進行錯誤處理（error handling）與缺陷處理（fault handling）。
  - 回報錯誤狀況。服務失效除了可以在 `try` block 回報，也可能由 `catch` block 回報，端看錯誤偵測發生在哪裡而決定。
  - 控制重試流程。如果例外發生之後要採取重試策略，可以在此控制重試流程，以決定是否繼續重試或準備離開函數執行。
- Finally
  - 釋放資源並且回報釋放資源失敗的狀況。
  - 如果在 `try` block 製作檢查點，則需要在此丟棄不再使用的檢查點以免佔用系統資源與儲存空間。

***

Try Block

先來看一下 `try` block 的責任，第一點實作函數所被賦予對外提供的服務，就是俗稱的實作需求。這一點很簡單，就是把實作某個功能的正常邏輯寫在 `try` block 裡面。正常邏輯可以有一種以上的實作，除了預設實作方式（primary），也可以包含替代方案（alternative）。

第二點為狀態回復做準備，鄉民們可能比較少會寫出這樣的程式。請參考列表 18-1，`try` block 裡面的實作會改變物件或是系統的狀態，為了希望例外發生時可以將系統回復到正確的狀態，因此一進入 `try` block 立刻產生一個檢查點。當例外發生的時候，可以在 `catch` block 裡面透過這個檢查點把系統狀態回復到正常狀態。

```
1: public void foo() throws FailureException{
2:     CheckPoint cp = new CheckPoint(/* 參數 */);
3:     try{
```

```
 4:         cp.establish();
 5:         /* 可能會改變物件或系統狀態的程式碼 */
 6:     } catch(Exception e){
 7:         cp.restore();
 8:         throw new FailureException(e);
 9:     }
10:     finally{
11:         cp.drop();
12:     }
13: }
```

列表 18-1：在 try block 產生查核點

在 try block 為狀態回復做準備，然後在 catch block 回復錯誤狀態（error handling）也不是什麼新鮮事。有寫過 JDBC 程式的鄉民應該老早就有這樣的經驗，請參考列表 18-2 範例第 2 行，con.setAutoCommit(false)相當於告訴 Connection 物件準備一個檢查點，接下來對資料庫的操作先不要直接提交（commit）。如果發生資料庫交易錯誤，在 catch block 裡面的 con.rollback()這一行就是用來回復錯誤狀態。

```
 1: try {
 2:         con.setAutoCommit(false);
 3:         updateSales = con.prepareStatement(updateString);
 4:         updateTotal = con.prepareStatement(updateStatement);
 5:
 6:         for (Map.Entry<String, Integer> e : salesForWeek.entrySet()) {
 7:             updateSales.setInt(1, e.getValue().intValue());
 8:             updateSales.setString(2, e.getKey());
 9:             updateSales.executeUpdate();
10:             updateTotal.setInt(1, e.getValue().intValue());
11:             updateTotal.setString(2, e.getKey());
12:             updateTotal.executeUpdate();
13:             con.commit();
14:         }
15:     } catch (SQLException e ) {
16:         JDBCTutorialUtilities.printSQLException(e);
17:         if (con != null) {
18:             try {
19:                 System.err.print("Transaction is being rolled back");
```

```
20:                 con.rollback();
21:             } catch(SQLException excep) {
22:                 JDBCTutorialUtilities.printSQLException(excep);
23:             }
24:         }
25:     } finally {
26:         if (updateSales != null) {
27:             updateSales.close();
28:         }
29:         if (updateTotal != null) {
30:             updateTotal.close();
31:         }
32:         con.setAutoCommit(true);
33:     }
```

列表 18-2：資料庫交易在 `try`、`catch`、`finally` block 所代表的操作

最後，`try` block 也可以回報函數失效，請參考列表 18-3 程式片段。

```
1: public void deposit(int value) throws ATMTTransactionException {
2:     try{
3:         if (value < 0)
4:             throw new ATMTTransactionException("提款金額不得為負數");
5:
6:         if (value >= getBalance()) // 可能會產生 ATMBrokerException
7:             throw new ATMTTransactionException("提款金額不得超過或等於存款餘額");
8:
9:         // 執行交易,可能會丟出 ATMBrokerException
10:     }
11:     catch(ATMBrokerException e){
12:         // 處理交易錯誤
13:     }
14: }
```

列表 18-3：在 `try` block 回報函數執行失效

Catch Block

接著討論 catch block 的三個責任，第一點執行錯誤與缺陷處理（perform error handling and fault handling）。Error handling 和 fault handling 是兩個不同的操作，要分別討論。首先 error handling 的目的在於如果系統因為例外發生而處於一種錯誤狀態，則需要修正這種錯誤狀態，讓系統回復到可繼續運行的正確狀態。Error handling 的例子在解釋 try block 的時候已經說明，包含列表 18-1 第 7 行 cp.restore()，以及列表 18-2 第 20 行 con.rollback()。在〈CH29：強健度等級 2：狀態回復的實作策略〉對於 error handling 方法有更詳盡的說明。

Fault handling 則是要嘗試排除造成錯誤產生的根本原因，因為如果光是修正系統狀態而沒有排除造成錯誤的原因，系統繼續執行下去還是很有可能重複發生相同的錯誤。由程式自動來做 fault handling 是屬於比較困難的工作，因為要事先預知發生 fault 的原因，才有可能在設計階段把排除 fault 的方法設計到程式裡面。所以，實務上很多 fault handling 是交給聰明的「人類」來處理。例如，假設使用者正在使用文書處理軟體，在儲存檔案時因為同時間有另外一隻程式也開啟了相同的檔案而儲存失敗。這時候文書處理軟體會提示使用者：「該檔案已被其他軟體使用中，請先關閉其他程式。」接著使用者可以選擇「重試」或「取消」存檔的動作。文書處理軟體把「請先關閉其他程式」這個 fault handling 的責任交給使用者來執行。關於 fault handling 的更多說明可參考〈CH30：強健度等級 3：行為回復的實作策略〉

catch block 的第二個責任是回報錯誤狀況，舉凡像是丟出例外、在螢幕上印出錯誤訊息、把錯誤訊息記錄到日誌檔中，都屬於回報錯誤狀況的做法。類似的例子之前已經出現過很多次，例如列表 18-1 第 8 行 throw new FailureException(e) 與列表 18-2 第 16 行 JDBCTutorialUtilities.printSQLException(e)。

catch block 的最後一個責任就是控制重試流程，很多人把 catch block 當做 try block 的「備胎」，如果 try block 執行失敗，則在 catch block 裡面提供一個替代方案用來取代 try block 的實作。把 catch block 當做 try block 的備胎是很常見的例外處理方法，請參考下列程式片段[21]，第 4~8 行的 catch block 相當於 try block 的備胎。

```
1: Object value = new Integer(8);
2: try{
3:      attributeObj.replaceValue("Age", value);
```

[21] 節錄自 K. Arnold, J. Gosling, and D. Holmes,*The Java Programming Language, 4th ed*, Addison-Wesley, 2005, p.287.

```
4: } catch (NoSuchAttributeException e)
5:     // shouldn't happen, but recover if it does
6:     Attr attr = new Attr(e.attrName, value);
7:     attributeObj.add(addr);
8: }
```

列表 18-4：將 `catch` block 當成 `try` block 的備胎

Teddy 在介紹 `try` block 的責任時有提到，提供預設實作方式與替代方案的責任都屬於 `try` block，所以不應在 `catch` block 提供替代方案。上面這種寫法 Teddy 認為是一種稱之為 *Spare Handler* 的**例外處理壞味道（bad smell）**，並不建議採用這種寫法。詳細說明請參考＜CH34：例外處理壞味道＞。

為了讓例外發生之後 `try` block 可以有機會執行替代方案，就必須要在 Java 裡面模擬出重試。下列程式片段就是模擬重試的做法，鄉民們可以先不用管 `Retry` 類別的實作，總之在 `catch` block 裡面的 `retry.rescue()`（第 15 行程式碼）就是用來扮演**控制重試流程**的責任。

```
1: public boolean login(String name, String pwd)
2:                         throws AuthenticationException {
3:     Thrower<AuthenticationException> thrower = new Thrower<>();
4:     Retry retry = new Retry();
5:     retry.maxAttempt(5);
6:     boolean result = false;
7:     do {
8:        try{
9:           if (retry.attempt() < 2)
10:              result = readFromDB(name, pwd);
11:          else
12:              result = readFromLDAP(name, pwd);
13:       } catch(IOException | SQLException e){
14:          thrower.setException(new AuthenticationException(e));
15:          retry.resue();
16:       }
17:    } while(retry.isRecurrent());
18:    thrower.signalIFFailure();
19:    return result;
20: }
```

列表 18-5：在 `catch` block 控制重試流程

Finally Block

最後討論 `finally` block 的兩個責任，第一是釋放資源並且回報釋放資源失敗的狀況，關於這一點之前章節已經討論很多了，在此不再重複。直接說明第二點：清除檢查點。請回想一下 `try` block 的第二點責任是準備狀態回復資料，因此可能會在 `try` block 裡面產生一個檢查點。如果發生例外，這個檢查點會在 `catch` block 裡面被用來恢復狀態。但是不管有沒有例外發生，都必須在 `finally` block 裡面把檢查點丟掉，否則累積太多檢查點也會發生系統資源不足的問題（這種情況就好像備份太多次資料但卻沒把舊資料刪除，最後導致備份硬碟空間不足的的情況）。

請參考列表 18-1 第 11 行，`cp.drop()` 就是在 `finally` block 裡面用來丟棄檢查點的程式碼。

***

以上 `try`、`catch`、`finally` 的責任，是 Teddy「遍覽群書」之後整理出來的看法，並非程式語言的要求，鄉民們可以參考一下是否合用。Teddy 自己的經驗是，有了這樣的責任分派觀念之後，在做例外處理設計與程式撰寫時，會清楚很多，也比較不容易寫出混亂、易錯、不易看懂的程式。

***

友藏內心獨白：責任不可以亂扛。

Column D. | 這是你的問題，不是我的問題

當程式執行時發生了例外，首要之務就是要找到例外發生的根本原因（root cause），其次，就是要釐清「誰該負責」來處理這個例外狀況。本篇要談的是**誰該負責**的問題。所謂冤有頭，債有主，如果找到不該負責的人來處理例外，那麼程式很可能會變得不易理解也不容易維護，弄不好也可能會導致更多的錯誤發生。

舉個例子，假設你要設計一個讓使用者透過網頁輸入個人資料然後將資料儲存到資料庫的功能，其中有一個欄位是輸入「年齡」。一般正常的人類的年齡應該是不可能超過 150 歲（彭祖和偷吃了長生不老藥的嫦娥除外），所以你在資料庫中將年齡這個欄位的長度設為 unsigned short (0-255)。除非科學家發明了長生不老藥，否則 255 應該很夠用了。

你為這個功能設計了四個元件：

- UI：顯示網頁的程式碼。
- UserBean：用來將使用者輸入的資料由 Web UI 傳送到資料庫。
- Servlet：呼叫 UserDAO 將由 UI 端收到的 UserBean 存到資料庫中 。
- UserDAO (Data Access Object)：負責透過 JDBC 或是 OR-Mapping（物件與關聯式資料庫對應）工具將 UserBean 存到資料庫中。如果儲存資料發生錯誤，將丟出例外。

程式開發完畢，很幸運的你找到彭祖幫你作測試，彭祖在年齡這個欄位輸入 888，按下確定送出之後，畫面上看到由資料庫所發出的例外：「整數資料超出範圍（integer out of range）」。好了，這個例外，是誰的問題？

- UI：老闆，請聽我說，這不是我的問題。誰叫你資料庫欄位設計的太小，改成 unsigned int (0-65535) 不就好了。
- UserBean：老闆，請聽我說，很明顯的這不是我的問題，因為我的 `setAge` 函數和 `getAge` 函數接受參數和傳回值的都是 int，範圍遠大於 888。
- Servlet：老闆，請聽我說，這絕對不是我的問題，我只是負責把接收到的 UserBean 傳給 UserDAO。
- UserDAO：老闆，請聽我說，這肯定不是我的問題。依照規格，年齡欄位最大值就是 255。有人要輸入超過這個數值的資料，當然一定會發生例外，如果沒有丟出例外才是我的問題。

和以上例子類似的狀況在開發軟體的時候屢見不鮮，只要有寫過資料庫程式的鄉民們應該都會遇到。一般來說 UI 應該依據資料庫可接受值的範圍來做檢查，以避免將不正確或無法接受的資料存到資料庫。問題是，第一個丟出例外的人是 UserDAO，所以，如果你是設計 UserDAO 的人，你如何決定要自行處理這個例外（例如，用一個預設的數值來代替），或是往上丟交給別人處理？

***

設計軟體元件的目的就是要提供某種服務，因此當程式在執行的時候，軟體元件之間的互動關係，便可簡單區分為以下兩者：

- 客戶（client）：呼叫其他元件以便獲得某種服務。
- 供應商（supplier）：提供服務的元件。

有了客戶和供應商的概念之後，要幫軟體系統釐清責任的方法就變得很簡單了，只要幫元件介面撰寫「合約」（contract）就可以了。合約有兩種 (其實不只兩種，但這裡先看兩種就足夠了)，分別是：

- 前置條件（precondition）：在執行軟體元件**之前**必須滿足的條件。當客戶要呼叫供應商之前，必須要保證定義在該供應商的前置條件被滿足，才可以呼叫它（也就是說客戶必須滿足執行供應商所需的環境）。
- 後置條件（postcondition）：在執行軟體元件**之後**必須滿足的條件。當客戶呼叫供應商之後，供應商必須要保證定義在該供應商的後置條件被滿足（也就是說供應商提供了他所宣稱的服務）。

當程式執行時，如果合約被違反了，系統將會丟出例外。違反前置條件表示客戶有 bug，而違反後置條件則表示供應商有 bug。

故事講到這邊，再回到前面儲存使用者資料的例子，有了合約的概念，UserDAO 的合約大概長成這樣子：

```
public void saveUser (UserBean aBean)
```

```
Precondition:
    require aBean.getAge() >= 0 && aBean.getAge() <= 255
Postcondition:
    ensure DBUtil.getUser(aBean.getName()).equals(aBean)
```

有了這個合約，如果使用者在 UI 的年齡欄位輸入 888 而 UI 又沒有做檢查的情況下，Servlet 呼叫 `UserDAO.saveUser` 函數便會出現違反前置條件例外。有了合約之後，這是誰的問題就很清楚了。由於 UserDAO 已經明白表示「如果要呼叫我，age 的值一定要介於 0-255」，所以根本也不用考慮是否需要修改資料庫欄位來接受大於 255 的數值（因為違反前置條件是客戶，而非供應商的問題）。

以此類推，如果 Servlet 和 UserBean 都有類似的合約，那麼當使用者在畫面上的年齡欄位輸入 888，就可以立刻知道這是 UI 的問題，不是別人的問題了。那麼，UI 的合約要怎麼寫？

```
Precondition:
   require true
Postcondition:
   ensure  A valid UserBean is created and saved in the session
```

UI 畫面是要讓使用者輸入資料的，所以只要使用者執行這個功能就可以用，因此前置條件永遠成立（當然你也可以寫成 require a user is logged on 之類的條件）。既然 UI 的目的是要收集使用者資料，所以在畫面結束之後（假設使用者按下確定送出），要保證能夠產生一個合法的 UserBean 物件並將它存在 session。因此，為了滿足這個後置條件，很顯然的 UI 的實作就必須要檢查使用者所打的欄位是否正確。也就是說，使用者輸入資料的驗證要做在 UI 端。

***

友藏內心獨白：這就是依合約設計（Design by Contract）的觀念，高工局應該要學一下。

19 例外處理失敗怎麼辦？

在「例外處理設計與重構實作班」課程中，有一位學員問了 Teddy 一個問題…

學員：你上課的時候提到 `catch` block 的責任之一是負責「**錯誤處理（error handling）**」，也就是說如果系統狀態不對了，要想辦法將系統恢復到一個正確的狀態。

Teddy：對啊。

學員：我們之前有遇到一個錯誤處理的問題，團隊成員和主管討論了很久都沒有結論。

Teddy：什麼問題？

學員：**如果錯誤處理的程式也發生錯誤，那該怎麼辦？**

Teddy：很簡單，三秒鐘就可以回答你。如果錯誤處理的程式也發生錯誤，無法將系統回復到正確的狀態，就丟出一個 `ErrorHandlingException` 來代表這樣的異常狀況。

***

上述問題可以用下面的例子來思考。有遊客在海邊溺水發出求救訊號（丟出例外），這時候「救生員」（例外處理程式）要去搶救這位溺水的遊客。如果救生員搶救失敗，自己也溺水了，那怎麼辦？

1. 假裝什麼事都沒發生，把求救訊號（例外）吃掉。
2. 繼續向外發出求救訊號。

現在答案就很明顯，應該是要選 2，繼續向外發出求救訊號。但是接下來還是有一個問題需要考慮，那就是「例外蓋台」的問題：「如果救生員直接發出 `ErrorHandlingException`，這個例外會覆蓋掉原本遊客溺水的例外，產生「例外蓋台」的問題。」

所以說，這個 ErrorHandlingException 應該成為一個 suppressed exception，被加入到原本遊客溺水的例外身上（代表遊客溺水的例外在這裡稱為 function failure）。

***

圖 19-1 說明例外處理失敗（包含 cleanup 失敗）後，try、catch、finally block 的行為。

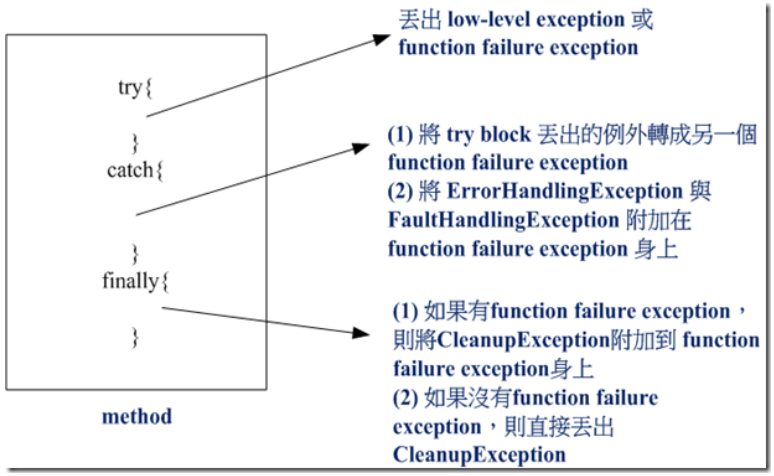


圖 19-1：例外處理失敗之後，try、catch、finally block 的行為

簡而言之，鄉民們需要：

- 自己定義 ErrorHandlingException、FaultHandlingException、CleanupException 這三種 unchecked exception。
- 如果 catch block 需要做 error handling 或是 fault handling 但卻失敗了，則產生 ErrorHandlingException 與 FaultHandlingException 來表示這種例外狀況。
- 如果 finally block 需要做 cleanup 的工作但卻失敗了，則產生 CleanupException 來表示這種例外狀況。

看到這裡鄉民們可能會想：「難道只要丟出 ErrorHandlingException、FaultHandlingException、CleanupException 就沒事了嗎？是否需要在呼叫者的程式碼中去處理這些例外？」這個問題要看鄉民們對於程式強健度的要求有多高，以及這些例外產生的機率。雖然「理論上」程式運行的時候有可能會發生 ErrorHandlingException、FaultHandlingException、CleanupException，但「實際上」這些例外發生的機率與頻率有多高，有時候在設計階段並不容易預測。如果針對每一個函數都要去思考該如何處理它

們「可能」會遇到的所有例外，很可能會導致不必要的過度設計，浪費太多資源。在這種情況下，開發人員只要確定：

1. 所有的例外狀況都有被回報。
2. 回報**語意**清楚的例外，以便協助開發團隊縮短事後除錯時間。

這樣的例外處理策略足夠應付大部分情況，也可避免過早且過度的例外處理設計。

***

友藏內心獨白：例外處理的例外，還是例外。

20 Checked 與 Unchecked 例外的語意與問題

Java 語言將例外分成兩種：checked exception 與 unchecked exception。鄉民們可能會對這兩個分類的命名覺得有點奇怪，為什麼要叫做 checked（被檢查、受檢）與 unchecked（不被檢查、非受檢）？顧名思義，要了解 checked 與 unchecked exception，就要先揪出到底「是誰」在背後做檢查，然後再去探討「檢查的對象」，以及「如何檢查」，還有「檢查的結果」又是什麼。

檢查的人就是**Java 編譯器**，也就是說 checked 或是 unchecked 是從 Java 編譯器的角度來看例外。請參考列表 20-1。

```
1: public void foo() {
2:    throw new MyException();
3: }
```

列表 20-1：`MyException` 的類別將決定 foo 函數該如何處理它

如果 `foo` 函數丟出的 `MyException` 屬於 checked exception，Java 編譯器會很雞婆的幫鄉民們檢查下面這件事：

如果 `foo` 沒有捕捉 `MyException`，則要將它宣告在 `foo` 的簽名（signature）中。

以上就是 Java 編譯器針對 checked exception 所執行的檢查規則，稱之為**處理或宣告原則（handle-or-declare rule）**。在程式中如果有人丟出 checked exception，Java 編譯器會檢查處理或宣告原則有沒有被遵守，如果沒有，則視為**語法錯誤**，程式無法編譯成功。對於 unchecked exception，Java 編譯器則不會理它，隨便鄉民們愛怎麼丟，就怎麼丟。

假設 `MyException` 是一個 checked exception，列表 20-1 的程式要修改寫成列表 20-2 或列表 20-3 才可以被 Java 編譯器成功編譯。

```
1: // 符合 handle rule
2: public void foo() {
3:  try {
4:      throw new MyException();
5:  }
```

```
6:   catch (MyException e) {
7:      // handle the exception
8:   }
9: }
```

列表 20-2：捕捉 checked exception

或

```
1: // 符合 declare 的要求
2: public void foo() throws MyException{
3:      throw new MyException();
4: }
```

列表 20-3：宣告 checked exception

***

現在問題來了，Java 吃飽沒事幹為什麼要把例外分成這兩種類型，然後又要求 Java 編譯器雞婆地幫大家檢查使用 checked exception 有沒有「乖乖的遵守交通規則」？Java 是被廣泛使用的程式語言中第一個採用 checked exception 的語言[22]，**Java 設計者認為，unchecked exception 很容易被程式設計師給忽略，導致例外沒有被處理而降低系統的強健度。**因此在 Java 語言裡面設計了 checked exception 並且讓編譯器主動幫程式設計師檢查，如果程式中發生 checked exception，請提高警覺賦予這個 checked exception 足夠多「關愛的眼神」，千萬不要不理它啊。所謂「不要不理它」，最基本的門檻就是程式碼至少要滿足處理或宣告原則。

在《Effective Java, 2nd》[23]這本書第九章裡面，提到了 Java 語言對於 checked 與 unchecked exception 的**語意**：

- 使用 checked exception 來表示可回復狀況（using checked exceptions for recoverable conditions）。

[22] Java 的許多設計參考自 C++，C++的例外處理設計支援例外規格（exception specification），開發人員可以將函數可能丟出的例外宣告在函數介面上。例如，void foo() throw(…) 表示 foo 可以丟出任何型別的例外，類似 Java 的 void foo() throws Exception。與 Java 不同，C++編譯器並不會在編譯期間檢查程式碼是否有符合例外規格，而是在程式執行時間檢查。

[23] J. Bloch, *Effective Java Programming Language Guide, 2nd*, Addison Wesley, 2008.

- 使用 unchecked exception 來表示程式錯誤（using runtime exceptions for programming errors）。

上面這兩點非常重要，在 Java 程式中做例外處理，不是只會寫程式就好了，還要和 Java 語言設計者「心心相印」。Java 的設計者認為，checked exception 本質上代表著一種**可以被回復的狀況**。因此，當鄉民們遇到 checked exception，理當嘗試去「回復」它，東西壞掉總不能不修理就直接丟掉換新的吧。

Unchecked exception 表示**程式錯誤**，翻成白話文就是程式有 bug。那麼遇到 unchecked exception 要如何處理呢？把這個問題換成後面這個問句，答案就很清楚了：「程式有 bug 要如何處理？」答案很簡單，有 bug 就改程式把 bug 修掉啊。所以針對 unchecked exception「理論上不應該在程式中去處理它[24]」，因為它代表 bug。當 unchecked exception 發生時只需要在主程式（main program）中捕捉並且以使用者看得懂的方式回報給使用者，或是記錄詳細的錯誤資訊，然後回報給開發人員以便協助後續除錯工作的進行。

***

雖然Java的設計者希望鄉民們能夠了解他們的想法，但是也有很多人是不買帳的。理想歸理想，實際上寫程式的時候還是會遇到很多問題，例如：

- 雖說 Java 建議「使用 checked exception 來表示可回復的狀況」，但是為什麼偏偏當程式中遇到一個 checked exception 的時候鄉民們卻打死也不知道要如何去回復（recover）任何東西哩？舉個最常見的 checked exception—`IOException` 為例，處理檔案、輸入、輸出串流物件或網路 socket 物件，都會附贈 `IOException`。收到 JDK 送來的這份大禮之後該怎麼辦？因為僅僅依靠例外物件本身不足以提供足夠的 context 來協助例外處理設計，因此沒人知道要如何處理。
- 既然 checked exception 要符合處理或宣告原則，但是很多鄉民既不知道如何處理，也不明白如何宣告。例外這麼多，到底要怎麼處理？那就亂處理啊！所以程式中經常可以看到「捕捉然後忽略」（例外被捕捉之後忽略不處理），或是 「盲目宣告」（盲目地將所有發生的 checked exception 都宣告在函數介面之上）。
- Checked exception 要宣告在函數介面之上，所以如果一個函數會丟出去的 checked exception 數目或是型別改變了，則函數的介面也跟著改變。介面改變這件事可不是好玩的，會造成軟體元件不相容的問題。關於這個問題的討論請參考〈CH21：介面演進〉。

[24] 關於是否應該用例外處理機制來修正程式裡面的 bug，請參考〈CH12:例外處理 PK 容錯設計〉。

- 有些鄉民們「自視甚高、天生反骨、不遵教化」，打死都不想用 checked exception。問題是這些武功高強的鄉民們又很有本事地寫出很多好用的開源軟體，當我們使用這些開源軟體的時候，對於例外處理的腦神經開始錯亂了起來。原本 Java 告訴大家，unchecked exception 代表 bug，不應該在程式裡面處理，而是要靠程式設計師去修正。但是不甩這一套的人卻一律使用 unchecked exception 來代表可回復狀況與程式錯誤。在開發程式的時候同時遇到兩派人馬所丟出的例外，把例外處理的問題更加複雜化了。這種不一致的作法，套一句古書上的話：「民安手措其手足」？

  舉個例子，在 Standard Widget Toolkit（SWT, Eclipse 的跨平台 Java 使用者介面類別庫）中，`SWTException` 用來表示可回復的 SWT 錯誤（recoverable SWT error），而 `SWTError` 則用來表示不可回復的錯誤。這兩個例外都是 unchecked exception，換句話說，SWT 壓根就不喜歡 checked exception，而是用 `SWTException` 與 `SWTError` 這兩種不同類別的 unchecked exception 來分別代表可回復與不可回復的例外狀況[25]。

  類似的例子還有 Spring 與 Hibernate 這兩個廣泛被使用的開源軟體，它們也是不喜歡 checked exception 而鼓勵只使用 unchecked exception。

***

例外處理設計包含著多個面向，雖然例外類別設計者當初也許對於不同的例外賦予不同的語意，但實際上在面對例外處理設計問題的時候，光是依靠例外類別的語意，還是不足以提供足夠的資訊，當做設計決策參考。欲知例外處理設計的各個面向，請參考本書「第四部為什麼例外處理那麼難？例外處理的 4+1 觀點」。

***

友藏內心獨白：例外處理和「媽寶」、「公主」一樣，真的很難搞定啊。

[25] 例外的可回復性（recoverability）與是否支援 checked exception 沒有關係，只要在設計例外類別時，清楚定義例外類別的語意即可。

21 介面演進

在〈CH20：Checked 與 Unchecked 例外的語意與問題〉提到因為 checked exception 屬於函數介面的一部分，因此如果函數丟出 checked exception 的數目或是型別改變，則等於函數的介面也改變了。上述現象有一個術語來稱呼它，叫做**「由 checked exception 所引起的介面演進（interface evolution，以下簡稱介面演進）」**。很多 Java 開發人員認為介面演進是一件很討厭的事情，因為一旦某個函數的介面改變，所有呼叫到它的客戶端程式都會因為介面不相容而被影響。請看列表 21-1 範例程式。

```
1: public void callee () throws IOException {
2:     // code that throws IOException
3: }
4: public void caller () {
5:     try {
6:         callee();
7:     } catch (IOException e) {
8:         // do something...
9:     }
10: }
```

列表 21-1：callee 函數的原始介面

如果列表 21-1 的 callee 函數額外丟出了一個新的 SQLException，則 caller 函數被迫改成列表 21-2 這個樣子（還記得 Java 的「處理或宣告原則」嗎？）：

```
1: public void callee () throws IOException, SQLException {
2:     // code that throws IOException and SQLException
3: }
4: public void caller () {
5:     try {
6:         callee();
7:     } catch (IOException e) {
8:         // do something...
9:     } catch (SQLException e){
10:         // do something...
11:     }
12: }
```

列表 21-2：callee 函數介面增加 SQLException 的宣告

只因為程式中用到的某個函數多丟出一個 checked exception，原本可以正常執行的程式變成無法通過編譯，這樣不是很討厭嗎？如果將 SQLException 改用 unchecked exception 取代，那該多好。請看下面這個改用 unchecked exception 的例子：

```
1: public void callee () throws IOException {
2:     // code that throws IOException
3:
4:     // code that throws RuntimeException;
5: }
6: public void caller () {
7:     try {
8:         callee();
9:     } catch (IOException e) {
10:        // do something...
11:    }
12: }
```

列表 21-3：callee 函數丟出 RuntimeException

好棒喔，因為用了 RuntimeException，所以原本的 caller 就不需要修改也不會有編譯錯誤。這樣子一來，就可以徹底避免介面演進的問題了…嗎？

***

事情當然沒有鄉民們想得那麼簡單，請思考下面幾點：

- 如果 callee 函數在第一版（列表 21-1）完成之後發現自己可能因為其他原因導致執行失敗，所以需要通知 caller 函數讓它有機會可以做相對應的因應措施。此時 callee 函數如果採用 unchecked exception 來代表一個「可回復例外」，那麼關於 callee 函數的修正版會丟出一個新的例外這件事情，caller 函數是「無從得知」的（無法從編譯器那裡得到通知）。假設有 N 個客戶端程式都呼叫了 callee 函數，這 N 個客戶端程式在編譯期間「看起來」不會因為 callee 的介面演進而造成任何影響，但事實上這些客戶端程式在執行期間的行為（runtime behavior）已經因為 callee 丟出一個新的例外這件事情而被改變了。
- 也就是說，從廣義的角度來看，checked exception 與 unchecked exception 都會造成介面演進，只不過前者會造成**外顯式介面演進（explicit interface evolution）**，而後者（如果是拿來當成可回復例外使用）會造成**內隱式介面演進（implicit interface evolution）**。

- 當然，如果 unchecked exception 被當成 programming error（bug），就不會造成內隱式介面演進的問題。

無論介面演進的起因是否為例外所造成，介面演進雖然討厭（尤其是當改變介面的那個函數已經被很多人使用的時候），但卻是軟體發展歷程中無法避免的問題。為什麼？舉個例子，在結婚前身份證上面的配偶欄是空白的，那麼結婚之後配偶欄就會登記配偶的姓名（介面演進，從單身狀態變成已婚狀態，所以要宣告「結婚」這個「功能」無法再被使用）。如果沒有這個介面演進，很多人就可能在「無意之中」變成了別人的「小三」（小三內心吶喊：怎麼會這樣，我老早就檢查過了，老王的身份證配偶欄明明是空白的啊！）這下子知道介面演進有多重要了吧。

所以，如果函數的行為已經改變，其變化之大已經和原本的行為不相容了，就應該要改變原先的介面，或是創造一個新版本的介面，讓客戶端知道該函數已非昔日阿蒙了。

***

既然介面演進不可避免，可是一旦發生又會造成開發人員的困擾，在設計介面的時候有沒有什麼原則可以降低介面演進所造成的傷害？有的，Folwer[26]建議了以下四點原則：

- 區分公開介面與發佈介面（separating a public interface from a published interface）：一般的程式語言並沒有區分公開（public）與 published（發佈）介面，所有公開介面（例如 Java 與 C#裡面的 public 類別與函數）都可以被客戶端程式所存取。這會造成一個麻煩，有時候某些元件的介面還沒有很成熟，隨時可能會改變，但是這些介面因為需要被程式裡面其他元件所存取，因此必須設計成公開介面。但是開發者又不希望這些公開介面被其他外部使用者給存取，以免改版之後造成其他使用者的困擾。在這種情況之下，開發人員必須自己區分那些是因為程式互動而必須被設計成公開介面，那些是已經成熟可以被外部使用者所直接呼叫的發佈介面。例如，Eclipse 的設計，利用 Java 語言的 package 來區分公開介面與發佈介面。放在 internal package 裡面的類別，就屬於尚未發佈的介面，隨時可能會改變，因此雖然放在 internal package 的類別可以直接被使用者呼叫，但是請不要這麼做，否則日後介面改變造成客戶端程式無法執行的後果，請自行負責。再舉個例子，假設 Teddy 開發了一套例外處理函式庫，準備對外釋出 1.0 版。Teddy 把發佈給使用者可直接呼叫使用的類別都放在 `teddysoft.tw.exception.design` 這個 package 裡面。但是有一些類別還不太成熟，其實作與介面可能會經常改變。這些發展中的類別會被已發佈的類別使用到，因此還是得一併釋出給使用者。這時候可以把這些發展中的類別放入

[26] M. Fowler, “Public versus Published Interfaces,” IEEE Software, vol. 19, no. 2, Mar/Apr, 2002, pp.18-19.

`teddysoft.tw.exception.design.internal` package，雖然放在這裡面的類別屬於公開介面，但卻不是發佈介面。

也有人利用軟體版本號來區分公開介面與發佈介面，例如單數版本，例如 1.1、1.3、1.5，屬於「搶鮮版」，介面還不成熟，在正式釋出版中可能會異動。雙數版本，例如 1.0、1.2、1.4，屬於正式版，裡面的類別屬於正式發佈介面，除非遇到大改版，否則開發者會儘量維持日後介面相容性。

- 重構非發佈介面（refactoring unpublished interfaces）：遵循了第一條原則，接下來的事情就好辦了。只要是非發佈介面，開發人員就可以放心大膽的重構，不用擔心會影響到外部使用者。
- 謹慎宣告發佈介面（declaring published interfaces with circumspection）：針對發佈介面，就等於是開發人員對於外部使用者的承諾，因此在設計與釋出發佈介面的時候，應該要謹慎小心。關於如何避免因為任意宣告 checked exception 造成介面改變的方法，請參考〈CH31：例外類別設計與使用技巧〉。
- 不要改變發佈介面（making published interfaces immutable）：一旦介面發佈之後，就不要修改。這一點聽起來很奇怪，萬一發佈之後發現有問題，又不能改，那怎麼辦？不要修改的意思是說，原本發佈的介面不要修改，但是可以有「版本」的概念。例如，一個已經發佈的介面 `Runnable`，經過一段時間之後發現原本介面裡面的函數沒有考慮到傳回值，因此打算在下一版的介面增加傳回值的功能。因為 `Runnable` 已經發佈而且有非常多的客戶端程式使用這個介面，如果直接修改介面會導致很多現有程式無法執行。因此可以把新版的 `Runnable` 取名為 `Runnable2` 或 `RunnableV2`，甚至是直接換名字為 `Callable` 也是一種方法。

***

軟體開發的問題非常複雜，經常無法僅僅依靠「單一點」的知識來判斷設計或決策的好壞，有時需要往外尋求更大的 context，才得以一窺全貌。從本章中，鄉民們應該可以感受到這種「逐步向外尋找線索的推理過程」。如果單從**程式語法**的角度來看，checked exception 的確是會造成介面演進的問題，更進一步從**程式行為**的角度來看，其實 checked 與 unchecked exception 都會造成介面演進，只不過前者造成外顯式介面演進，後者造成內隱式介面演進。到底是外顯式比較好，還是內隱式比較好，這是程式語言設計領域傳統的「靜待型別檢查」與「動態型別檢查」之爭，很難拼出個高下。就好像有人喜歡 C/C++/Java/C#這種編譯式語言，也有人喜歡 JavaScript、Python、Ruby 這種直譯式語言，沒什麼對錯，就是知道彼此之間的強處與限制，然後挑一個自己覺得合適的來使用。

最後，再把 context 往外擴展一層，從「版本控制」的角度來看介面演進問題。只要能夠區分公

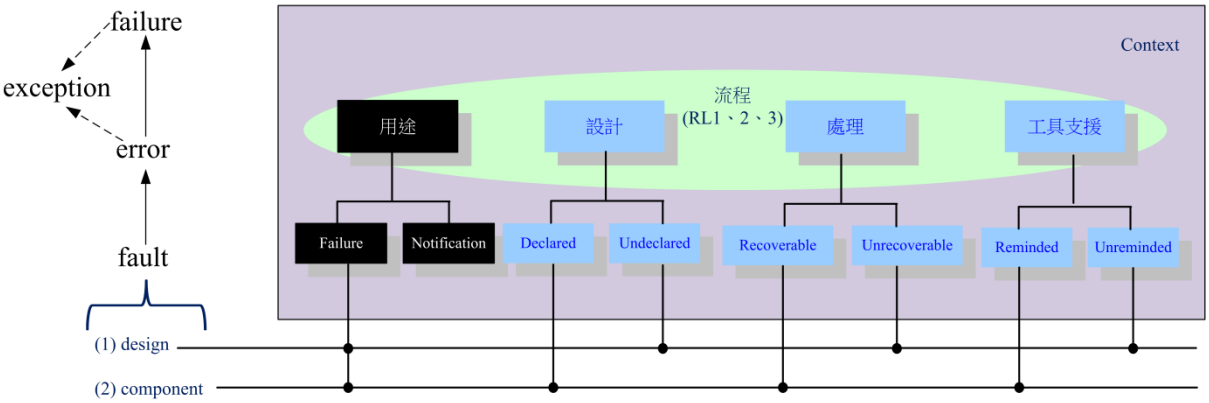
開與發佈介面，便可以自由重構公開介面以符合軟體開發所需。針對發佈介面則需要多花心思，依據軟體版本的生命週期，儘量保持發佈介面的向下相容性。如此便可讓介面演進對軟體開發所造成的傷害，減少到一個可以控制與接受的範圍之內。

***

友藏內心獨白：改變是無法改變的事實，就好像台灣的地震一樣，一定會發生。擁抱改變並接受它，就會想出好辦法。

第四部 為什麼例外處理那麼難？例外處理的 4+1 觀點

22 用途觀點



有寫過程式的鄉民們大概都會同意，例外處理是一件很困難的工作。但是，有沒有人想過，為什麼例外處理會這麼難？到底難在何處？**唯有先了解問題，才有機會找出合適的解決方案。**

其實說破了道理也很簡單：因為「例外處理」牽扯到好幾個相關但卻不相同的觀點或是面向，例外處理要做得好，這些觀點都必須要被關注到。Teddy 認為例外處理至少涵蓋了以下五個觀點：

- Usage View（用途觀點）
- Design View（設計觀點）
- Handling View（處理觀點）
- Tool-Support View（工具支援觀點）
- Process View（流程觀點）

這一章先談一下例外處理的用途觀點。

***

例外不就是例外嗎，還能有什麼其他用途？如果鄉民們這樣想就弱掉了。雖然理論上例外是用來表示 failure（請參考〈CH8：強健性大戰首部曲：威脅潛伏〉），但是事實上例外還有可能被用來當做 result classification 與 monitoring。當例外被當做 result classification 與 monitoring 時，並不是代表一種異常狀況，而是用來表達一種**狀態通知**，因此通常不需要加以特別處理。

以上翻成白話文就是說：**「寫程式遇到別人丟出例外時，請先判斷這個例外是用來代表 failure，還是用來表示狀態通知（result classification 與 monitoring）」**。看列表 22-1 的例子大家會比較清楚一點：「請問 Java 語言裡面的 `InterruptedException` 要如何處理？」

```
1: public class InterruptedExceptionUsage {
2:
3:     public void sleepMillisecond(int ms){
4:         try {
5:             Thread.sleep(ms);
6:         } catch (InterruptedException e) {
7:             // 如何 "處理" 這個例外？
8:         }
9:     }
10: }
```

列表 22-1：遇到 `InterruptedException` 要如何處理？

這個問題多年前當 Teddy 第一次用 Java 撰寫多執行緒程式時困擾了許久：「`InterruptedException` 明明是一個例外，可是為什麼捕捉之後卻不需要特別處理？」多年之後，Teddy 陰錯陽差投入例外處理研究之後才慢慢弄清楚，原來 `InterruptedException` 在這裡（列表 22-1）並不是用來表達 failure（某人辦事不力），而只是一種狀態通知，用來告訴等待或是執行中的執行緒有其他人把你給中斷了，請識相一點不要硬撐著不下台，趕快拍拍屁股做完收工閃人了。這也是為什麼上面這個例子在捕捉到 `InterruptedException` 之後什麼都沒做的原因，在這裡如果硬要設計什麼「例外處理程式碼」，想要「拯救」被中斷的執行緒，被中斷之後繼續死賴著不結束，反倒會造成程式錯誤。

***

以上內容如果鄉民們能夠理解，恭喜你踏入了例外處理設計的大門。但是世間的事情如果都那麼簡單，黑白分明，事情就好辦了。要特別將「例外用途」提出來當成一個觀點來討論的原因，**就是因為開發人員通常無法直接透過例外物件本身來判斷這個例外是屬於哪種用途，還必須搭配其他 context，例如 local context 與 application context，才能決定例外的用途〔請參考〈CH0：2013 年 11 月某一天 Teddy 到某大學資工系演講，講題是「例外處理設計與重構」。演講結束之後有一位同學問了 Teddy 一個很常見也很有趣的問題。**

同學：你剛剛建議我們不要用回傳碼（return code）來代表例外狀況，所以如果我要設計一個依據學號到資料庫中查詢學生資料的函數，當找不到符合條件的學生資料的時候，是不是應該要丟出例外？

Teddy：嗯，不一定，要看你的需求與設計。假設你的函數長成這樣：

```
public List<Student> queryStudents(String ID)
```

只要傳回一個大小為 0 的 List 就可以代表找不到學生資料，不需要丟出例外。

同學：那如果我的函數只會回傳一個學生物件呢？

Teddy：假設你的函數長成這樣：

```
public Student queryStudent(String ID)
```

只傳回一個物件，那麼你就要問自己：「依據 ID 來尋找學生資料，但最後找不到任何一筆符合條件的資料，這種情況算是正常狀況還是異常狀況？」

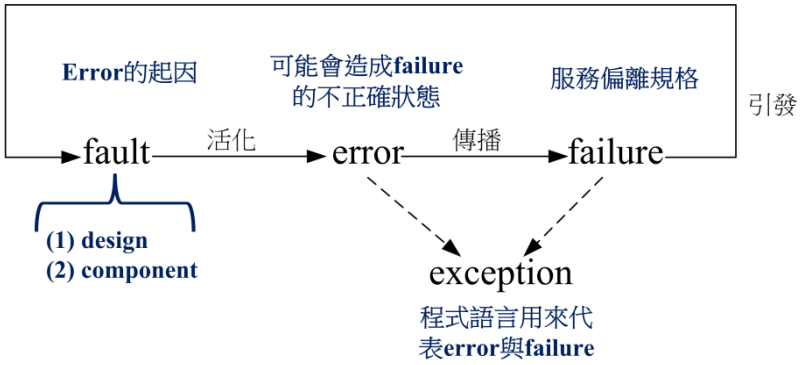
同學：…

Teddy：如果你問我的看法，我會覺得這是正常狀況。在日常生活中，依據某些條件去找資料，最後找不到任何符合條件的資料，算是一種很常見的狀況，所以我不會針對這種情況丟出例外。你可以傳回 null 用來代表找不到任何資料，如果不喜歡 null，也可以套用 Null *Object 設計模式，傳回一個 Null Object 來表示找不到任何一筆符合條件的資料。以上這兩種做法都比傳回例外要來的好。*

***

同學所問的問題，可以〈CH8：強健性大戰首部曲：威脅潛伏〉的圖 8-1 來解釋（重繪如下）。Exception 在程式語言中用來代表 error 與 failure，分別表示「目前可能處在不正確的狀態」

與「被呼叫的函數辦事不力」。「找不到資料」並非狀態錯誤，也不是尋找資料的函數辦事不力。事實上，尋找資料的函數正確執行完畢，只不過沒有找到符合查詢條件的資料罷了。這種狀況是在規格中所允許的狀況，和例外處理無關，只要在設計函數介面的時候規定好傳回某種特殊值（null 或 Null *Object*）*代表找不到資料這樣就可以了*。



***

友藏內心獨白：所以說了解 fault、error、failure 的定義對於例外處理設計是很有幫助的。

例外處理的四種脈絡）。

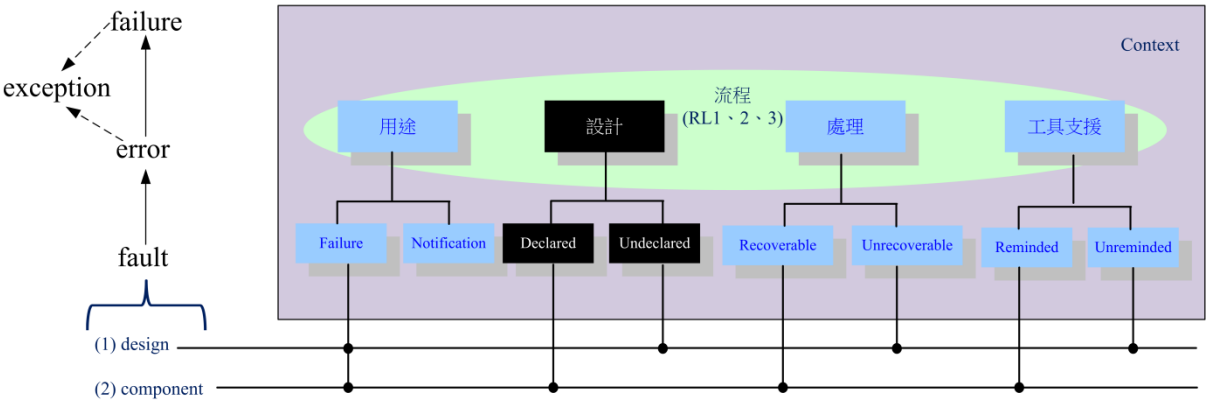
舉個例子，假設鄉民們正在開發藍光光碟撥放器軟體，這個軟體有一個自我測試的功能，用來判斷使用者的電腦夠不夠力播放高畫質藍光電影。在自我測試模式，應用程式如果持續接收到影像畫面（video frame）遺失的 `FrameLostException`，應該將這個例外當成 failure（電腦系統不夠力），並且把這個情況回報給使用者知道。

如果播放軟體處於正常模式，而且硬體效能也足夠，但是正在播放電影時因為防毒軟體突然在背景模式執行掃描工作，導致電腦忙碌而產生 `FrameLostException`。此時就應該把這個例外當成是一種狀態通知，直接忽略或是把例外記錄到日誌檔中，不須立即回報給使用者。除非開發人員在設計階段就可以明確得知在怎樣的電腦負載程度之下會直接造成 `FrameLostException`，並且當這種情況發生的時候，捕捉 `FrameLostException` 並將其轉成使用者看得懂的錯誤訊息，例如：「目前電腦負載過重導致播放效能不佳，請確定您的電腦是否有程式在背景模式之下執行大量的工作。」反之，如果直接將不斷產生的多個 `FrameLostException` 顯示在畫面上，則使用者在看電影的時候會一直被中斷，強迫按下「確定」或「取消」按鈕才可以繼續。這樣的播放軟體所提供的影片觀賞體驗，應該沒有人會想要持續使用吧。

***

友藏內心獨白：生病就該看醫生，但沒生病一直亂吃藥也是不對的。

23 設計觀點



2013 年 4 月 29 日看到一則新聞，大意是說對岸駭客可能會將入侵目標轉向台灣，對台灣國防、政治、經濟安全造成危害。記者電話訪問了某警察單位…

記者：對於對岸駭客可能會入侵台灣一事警察單位有什麼因應之道？

警察：我們主要還是要看有沒有人報案，有人報案我們就會處理。

Teddy 內心獨白：靠…過來一點。「有人報案我們就會處理」這是哪招，會不會太消極了一點？

***

專案經理：對於我們產品品質不佳，bug 很多一事團隊有何因應之道？

程式設計師：我們主要還是要看有沒有人回報 bug，有人回報我們就會處理。

Teddy 內心獨白：好熟悉的對話內容，好像在那裡聽過。

***

以上所述和例外處理有何關係？對於例外處理的看法，其實開發人員和警察杯杯的想法很像：「只要有人報案，我們就會處理」。在程式中，不同的時間點有著不同的報案方法。如果是消極的等到軟體上市之後，由使用者發現問題之後才來「報案」，使用者會留下軟體品質不良的印象，此時再來處理已經稍嫌太晚，而且成本也高出很多。

要提升使用者的滿意度，減低「辦案成本」，最好能夠把發現問題的時間點往前移到設計階段（design phase），讓程式能夠以例外的方式來「報案（回報問題）」，使得開發人員能夠在程式中處理這些異常狀況，提高產品的品質。從設計的角度來看，例外的回報有以下兩種形式：

- 公告例外（declared exception）：將例外宣告在元件的介面規範中，又稱為 anticipated 或 expected exception（預期例外），用來代表 component fault。
- 非公告例外（undeclared exception）：例外沒有被宣告在元件的介面規範中，又稱為 unanticipated 或 unexpected exception（不預期例外），代表 design fault。

看一個 Java 程式例子，`IllegalArgumentException` 在列表 23-1 中是屬於 undeclared exception，因為它沒有被宣告在 `deposit` 函數的介面上。

```
1: public void deposit(int value) {
2:     if (value < 0 ) {
3:         throw new IllegalArgumentException("存款金額不得為負數.");
4:     }
5:     // doing normal deposit logic
6: }
```

列表 23-1：非公告例外範例

反之，列表 23-2 程式碼的 `IllegalArgumentException` 則屬於 declared exception，因為它被宣告在 `deposit` 函數的介面上。

```
1: public void deposit(int value) throws IllegalArgumentException {
2:     if (value < 0 ) {
3:         throw new IllegalArgumentException("存款金額不得為負數.");
4:     }
5:     // doing normal deposit logic
6: }
```

列表 23-2：公告例外範例

***

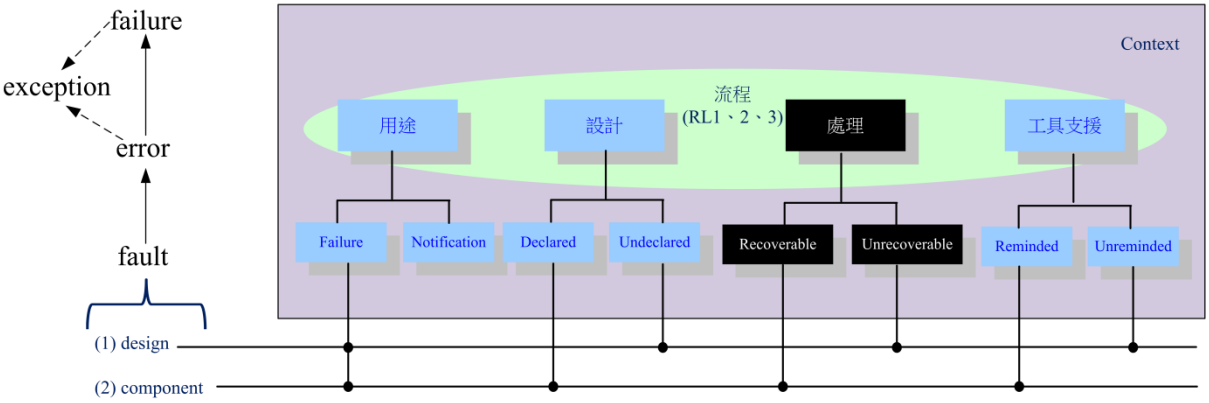
就好像沒有人報案，則警察不知道有犯罪案件發生是一樣的道理，唯有將例外宣告在介面上，或是以某種形式存在程式碼或文件當中（例如微軟公司所提供的 MSDN 線上文件），在設計階段開發人員才有機會知道要如何預備與因應可能會遭遇到的異常狀況。實事上，要求開發人員去處理 undeclared exception，已非例外處理的範疇，而是屬於**容錯設計**的議題，要實作容錯設計所花費的工夫比起例外處理要困難許多，成本也高出許多（請參考〈CH12：例外處理 PK 容錯設計〉）。

下次，如果客戶提出「undeclared exception 發生時系統也不能當機」的要求，這時候客戶對於系統可靠度的要求，已經提升到容錯的等級。記得提醒公司，至少在報價金額的尾數多加一個零。

***

友藏內心獨白：做任何事都要有成本觀念，區分例外處理和容錯設計是有意義的。

24 處理觀點



從〈CH23：設計觀點〉得知要做例外處理，首先必須先知道一個元件可能會丟出那些例外，接著以〈CH22：用途觀點〉來判斷這個例外是否真的代表 failure 或只是一種狀態通知，兩者的例外處理方法各不相同。本章處理觀點則是從**實作例外處理程式**角度來探討例外處理的問題。因為關係到要如何實作例外處理的細節，處理觀點是「例外處理的 4+1 觀點」裡面最複雜的觀點，需考慮到下列兩個因素：

- 可回復性 (recoverability)：針對例外所造成的錯誤狀態，捕捉例外的人是否有能力可以將其回復（recoverable）或是沒有能力回復（unrecoverable，又稱為 irrecoverable）。在預設情況下，recoverable exception 隱含著代表 component fault 的語意，而 unrecoverable exception 則代表 design fault。但實際上，很多時候僅僅依據例外類別是不足以判斷例外狀況是否可以回復，必須同時考慮到〈CH0：2013 年 11 月某一天 Teddy 到某大學資工系演講，講題是「例外處理設計與重構」。演講結束之後有一位同學問了 Teddy 一個很常見也很有趣的問題。

同學：你剛剛建議我們不要用回傳碼（return code）來代表例外狀況，所以如果我要設計一個依據學號到資料庫中查詢學生資料的函數，當找不到符合條件的學生資料的時候，是不是應該要丟出例外？

Teddy：嗯，不一定，要看你的需求與設計。假設你的函數長成這樣：

```
public List<Student> queryStudents(String ID)
```

只要傳回一個大小為 0 的 List 就可以代表找不到學生資料，不需要丟出例外。

同學：那如果我的函數只會回傳一個學生物件呢？

Teddy：假設你的函數長成這樣：

```
public Student queryStudent(String ID)
```

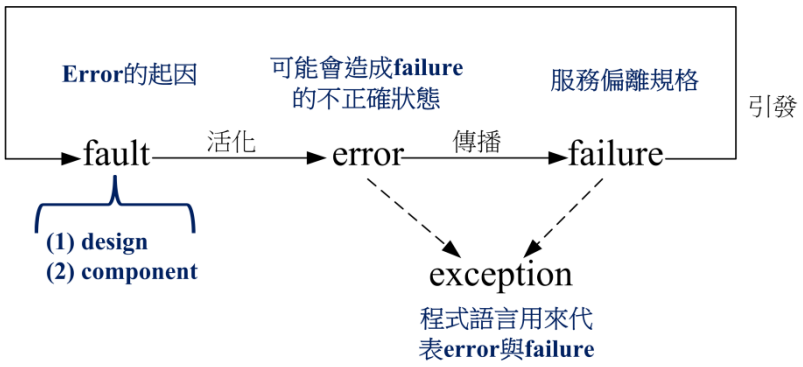
只傳回一個物件，那麼你就要問自己：「依據 ID 來尋找學生資料，但最後找不到任何一筆符合條件的資料，這種情況算是正常狀況還是異常狀況？」

同學：…

Teddy：如果你問我的看法，我會覺得這是正常狀況。在日常生活中，依據某些條件去找資料，最後找不到任何符合條件的資料，算是一種很常見的狀況，所以我不會針對這種情況丟出例外。你可以傳回 null 用來代表找不到任何資料，如果不喜歡 null，也可以套用 Null *Object 設計模式，傳回一個 Null Object 來表示找不到任何一筆符合條件的資料。以上這兩種做法都比傳回例外要來的好。*

***

同學所問的問題，可以〈CH8：強健性大戰首部曲：*威脅潛伏*〉的圖 8-1 來解釋（重繪如下）。Exception 在程式語言中用來代表 error 與 failure，分別表示「目前可能處在不正確的狀態」與「被呼叫的函數辦事不力」。「找不到資料」並非狀態錯誤，也不是尋找資料的函數辦事不力。事實上，尋找資料的函數正確執行完畢，只不過沒有找到符合查詢條件的資料罷了。這種狀況是在規格中所允許的狀況，和例外處理無關，只要在設計函數介面的時候規定好傳回某種特殊值（null 或 Null *Object）代表找不到資料這樣就可以了。*



***

友藏內心獨白：所以說了解 fault、error、failure 的定義對於例外處理設計是很有幫助的。

- 例外處理的四種脈絡〉所介紹的其他 context 以及「例外處理的 4+1 觀點」裡面的其他觀點。
- 例外處理實作：有哪些例外處理策略可以使用？例如，是否需要將例外往外傳遞、如何執行狀態恢復動作、如何確保函數在例外發生時能夠繼續提供服務？在所選擇的程式語言中，要如何實作這些策略？例如，假設採用 Java 語言，則要如何將程式的正常與例外處理邏輯，妥善地分配到 `try`、`catch`、`finally` 這三個結構之中。另外，還要考慮是否有足夠的輔助工具或函式庫，例如日誌檔（logging）機制、統一的錯誤回報框架、自動程式更新機制等，可以用來支援例外處理實作。

先談談可恢復性。一個例外狀況可不可修、要不要修，要同時考量**被呼叫者（callee）**與**呼叫者（caller）**的情況。**首先看看被呼叫者在拋出例外之後是否依然處於正確的狀態，再判斷呼叫者是否有足夠的 context 來處理這個例外。**理想上，被呼叫者應該確保例外發生之後自己並沒有造成系統狀態錯誤，否則會增加呼叫者例外處理的負擔；呼叫者除了要擦自己的屁股（維持正確狀態），還要幫被呼叫者擦屁股。在真實世界當中，幫別人擦屁股是一件非常不愉快的工作，在程式當中也是如此。尤其是當被呼叫者會改變系統的全域狀態或是外部狀態的時候，要幫它擦屁股不但是一件非常困難的工作，有時候甚至無法做到。

舉個例子，如圖 24-1 所示，假設你用了某個 Java 函數，它同時會更新五個資料庫表格，但是這個函數在更新資料的時候並沒有做交易處理（transaction）。一旦更新過程中發生 `SQLException`，最後資料庫處在資料不一致的狀態。身為呼叫者的你，要如何挽救這種狀況（考慮例外的可回復性）。

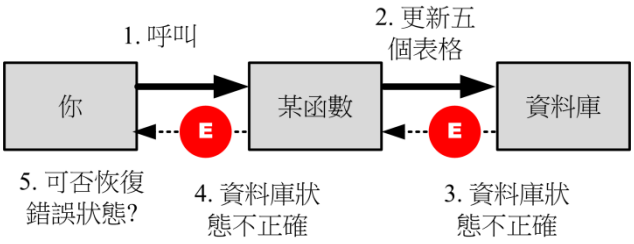


圖 24-1：分析例外可回復性

從 Java 語言的角度來看，`SQLException` 是一種可回復例外（recoverable exception，請參考〈CH20：Checked 與 Unchecked 例外的語意與問題〉）。最不花腦筋的作法是先將資料庫複製一份，當例外發生的時候再用之前的備份將其復原。這種作法不但耗時，而且在同時有很多人操作資料庫的情況下，也不切實際。也就是說，如果某函數丟出例外的時候系統的狀態已經不

正確，就算這個例外本身的語意是屬於可恢復性例外，接受的例外的人，很可能愛莫能助，沒有能力將系統狀態復原。這也是 Teddy 在本書中一再強調的觀念：**「無法單獨依據例外類別來判斷例外處理方式。」**

以上問題的癥結在於，該函數發生例外之後並沒有讓整個系統保持在正確的狀態，如果它可以負責任一點，確保自己執行失敗之後資料庫維持在正常狀態，那麼你所要考慮的例外處理工作就輕鬆許多，只需考慮恢復自己對系統造成的狀態修改即可。

例外發生的時候，唯有當系統依然處於正確狀態下，或是先執行狀態回復的動作，讓系統狀態返回正常，才可以繼續思考接下來要怎麼做。一個元件在發生例外之後沒有造成系統狀態錯誤，在本書中稱這個元件達到「強健度等級 2：狀態回復」。關於軟體元件強健度等級分類的詳細說明，請參考〈CH27：例外處理設計的第一步：決定強健度等級〉。一旦接收者確定系統狀態正確，接著便可依據〈CH0：2013 年 11 月某一天 Teddy 到某大學資工系演講，講題是「例外處理設計與重構」。演講結束之後有一位同學問了 Teddy 一個很常見也很有趣的問題。

同學：你剛剛建議我們不要用回傳碼（return code）來代表例外狀況，所以如果我要設計一個依據學號到資料庫中查詢學生資料的函數，當找不到符合條件的學生資料的時候，是不是應該要丟出例外？

Teddy：嗯，不一定，要看你的需求與設計。假設你的函數長成這樣：

```
public List<Student> queryStudents(String ID)
```

只要傳回一個大小為 0 的 List 就可以代表找不到學生資料，不需要丟出例外。

同學：那如果我的函數只會回傳一個學生物件呢？

Teddy：假設你的函數長成這樣：

```
public Student queryStudent(String ID)
```

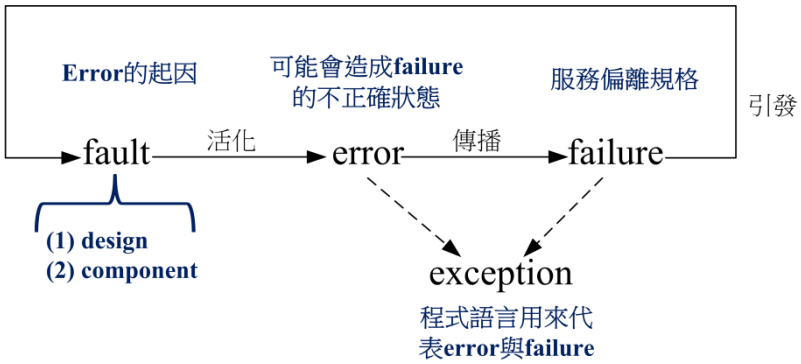
只傳回一個物件，那麼你就要問自己：「依據 ID 來尋找學生資料，但最後找不到任何一筆符合條件的資料，這種情況算是正常狀況還是異常狀況？」

同學：…

Teddy：如果你問我的看法，我會覺得這是正常狀況。在日常生活中，依據某些條件去找資料，最後找不到任何符合條件的資料，算是一種很常見的狀況，所以我不會針對這種情況丟出例外。你可以傳回 null 用來代表找不到任何資料，如果不喜歡 null，也可以套用 Null *Object 設計模式，傳回一個 Null Object 來表示找不到任何一筆符合條件的資料。以上這兩種做法都比傳回例外要來的好。*

***

同學所問的問題，可以〈CH8：強健性大戰首部曲：威脅潛伏〉的圖 8-1 來解釋（重繪如下）。Exception 在程式語言中用來代表 error 與 failure，分別表示「目前可能處在不正確的狀態」與「被呼叫的函數辦事不力」。「找不到資料」並非狀態錯誤，也不是尋找資料的函數辦事不力。事實上，尋找資料的函數正確執行完畢，只不過沒有找到符合查詢條件的資料罷了。這種狀況是在規格中所允許的狀況，和例外處理無關，只要在設計函數介面的時候規定好傳回某種特殊值（null 或 Null *Object）代表找不到資料這樣就可以了。*



***

友藏內心獨白：所以說了解 fault、error、failure 的定義對於例外處理設計是很有幫助的。

例外處理的四種脈絡〉，以及「第四部 為什麼例外處理那麼難？例外處理的 4+1 觀點」，來決定例外處理方式。

***

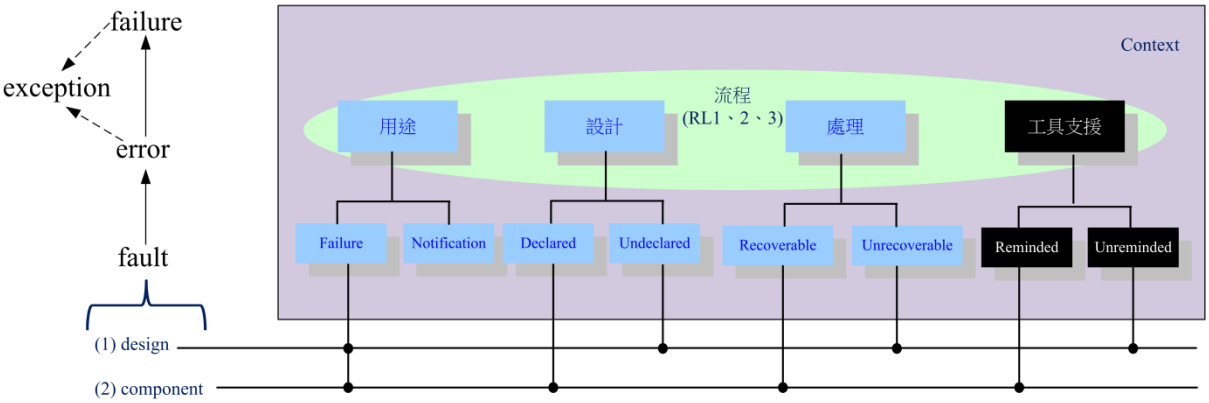
接下來討論例外處理實作的因素。有學者曾經做過研究，許多軟體系統的 bug，出自於不良的例外處理。任何系統設計的再好，如果例外處理實作錯誤也是枉然。關於例外處理的實作方式，在本書有其他章節專門討論，可參考：

- 例外處理策略：「第五部 強健度等級與例外處理策略」的章節內容。。
- 如何實作這些策略：〈CH13：Java 的 Try、Catch、Finally〉，以及「第六部 例外處理壞味道與重構」的章節內容。

***

友藏內心獨白：把設計不良當成駭客入侵，那就糗大了。

25 工具支援觀點



工具支援觀點是從開發環境提供例外處理支援的角度來探討為什麼例外處理會這麼困難。舉凡編譯器、整合開發環境(Integrated Development Environment；IDE，像是 Eclipse 或 Microsoft Visual Studio）與外掛程式（plug-in 或 add-on）、第三方廠商所開發的工具程式等都屬於開發環境的範疇。

在這個觀點之中，例外分成兩大類：reminded（提醒）與 unreminded（不提醒）。屬於前者的例外，開發環境會幫忙檢查每一個函數呼叫是否會拋出 reminded 類型的例外。檢查的結果將以某種方式通知開發人員，例如將結果顯示在 IDE 畫面上或產出報表，以協助開發人員設計例外處理。

從工具支援觀點來看，Java 的 checked exception 其實就是一種程式語言內建的 reminded exception，負責執行檢查與提醒的工具不是 IDE，也不是外掛程式或第三方軟體，而是 Java 編譯器。檢查的規則稱為「處理或宣告原則」，沒有滿足這條規則的程式直接被 Java 編譯器判定為語法錯誤。

看到這裡鄉民們可能會想:「Java 的 unchecked exception 不會被編譯器檢查，所以屬於 unreminded exeption？還有像是 C#這類只支援 runtime exception（unchecked exception）的語言，全部的例外也都屬於 unreminded exception？」

答案是：不一定，要看情況而定。程式語言的編譯器只是一種工具，鄉民們還是可以透過像是 IDE 外掛程式或第三方廠商所提供的工具，自訂檢查規則以區分 reminded 與 unreminded exception。舉個例子，SWT[27]是用 Java 語言開發的圖形介面工具，但是它並不喜歡 Java 的 checked exception，因此 SWT 自己設計了繼承自 `RuntimeException` 與 `Error` 的兩種類外類別：`SWTException` 與 `SWTError`，分別用來代表可回復（recoverable）與不可回復（unrecoverable）的異常狀況（也就是 component fault 與 design fault）。`RuntimeException` 與 `Error` 都屬於 unchecked exception，所以「從 Java 編譯器」的角度來看，它們都是 unreminded exception。

但是，鄉民們可以在 Eclipse 裡面自行開發一個外掛程式，來檢查採用 SWT 所開發的應用程式在那些地方會丟出 `SWTException`，並藉此提醒使用者去處理這些例外狀況。如果有了這個外掛程式，則 `SWTException` 就是一種 reminded exception。

C#語言雖然只支援 unchecked exception，但是在 Visual Studio 開發環境當中，當你將滑鼠移到某個函數，Visual Studio 會顯示該函數可能拋出的例外，如圖 25-1 所示。

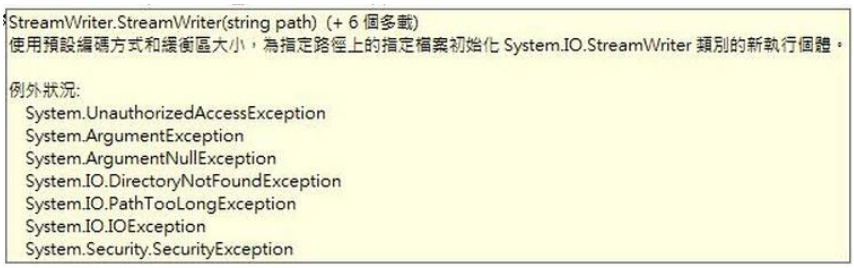


圖 25-1：Visual Studio 顯示函數拋出的例外

C#編譯器會去分析寫在程式中的<exception cref="member">description</exception>[28]這個特殊格式的文件註解，以便顯示這些例外訊息，如列表 25-1 所示。

```
1: /// <exception cref="System.IO.IOException">無法讀取檔案</exception>
2: public void readFile()
3: {
4:     // 讀檔案
5: }
```

列表 25-1：C#函數丟出例外的文件式註解範例

[27] S. Northover and M. Wilson, *SWT: The Standard Widget Toolkit, Volume 1*, Addison-Wesley, 2004.
[28] 請參考 MSDN 線上文件的說明：http://msdn.microsoft.com/en-us/library/w1htk11d.aspx

從這個角度來看，所有寫在<exception >標籤裡面的例外也可被視為 reminded exception。但是這個標籤的目的，只是單純用來標註一個函數所有可能拋出的例外，並不像 Java 的 checked 和 unchecked exception，除了 Java 編譯器是否會檢查的差別以外，同時還有「語意上」的不同。也就是 checked exception 用來代表可回復例外或 component fault，uncheckekd exception 用來代表不可回復例外或 design fault。

如果鄉民們在 C#語言想要透過例外類別來區分可回復與不可回復的例外，比較接近的作法是，將 Exception 與其子類別（不包含 SystemException）視為可回復例外，將 SystemException 與其子類別視為不可回復例外[29]。如果想用 C#語言模仿 Java 的「處理或宣告原則」規則，特別提醒開發人員程式中有那些未被處理或宣告的 Exception 與其子類別（不包含 SystemException），鄉民們也可以自行開發 Visual Studio 外掛程式來達到這個目的。

***

為了提高軟體的強健度，開發人員需要一個**提醒機制**，告知那些操作有可能產生例外，否則開發人員很容易忽略例外處理，只能等程式執行期間發生錯誤再回頭修補（還需要遇到有良心的開發人員才會回頭修補）。至於這個提醒機制，像是 Java 語言的做法，強制由編譯器將例外區分為 checked 與 unchecked，只是一種實作的方式。很多人不喜歡 Java「強迫中獎」的提醒方式，但無論如何提醒機制的存在，對於提升開發人員對於系統強健度的關注與警覺心，還是很有幫助的做法。如果開發環境對於「那些函數有可能產生例外」的提醒支援不足，將造成開發人員輕忽例外處理的必要性，以及增加例外處理設計的負擔。

比較列表 25-2 的 Java 程式與列表 25-3 的 C#程式，兩段程式的功能相同。列表 25-2 的程式，Java 編譯器會顯示第 4 與第 5 行可能會拋出 IOException，要求開發人員處理。

```
1: public void writeFile(String filePrefix, String data) {
2:    Writer writer = null;
3:    try {
4:       writer = new FileWriter(filePrefix+"_"+data+".txt");
```

[29] 關於.NET 例外類別階層設計，請參考：http://msdn.microsoft.com/en-us/library/z4c5tckx(v=vs.110).aspx.

```
5:       writer.write(data);
6:     }
7:     finally {  // code for cleanup  }
8: }
```

列表 25-2：第 4~5 行的底線代表 Java 編譯器提醒程式會丟出 checked exception

```
1: public void writeFile(String filePrefix, String data) {
2:   TextWriter writer = null;
3:   try {
4:      writer = new StreamWriter(filePrefix+"_"+data+".txt");
5:      writer.Write(data);
6:   }
7:   finally {   // code for cleanup   }
8: }
```

列表 25-3：C#編譯器不會提醒程式遭遇到例外

反之，列表 25-3 的程式，C#編譯器並沒有提供這樣的「服務」[30]。開發人員需要自行查閱文件，才知道原來第 4 行的程式會丟出為數驚人的例外：`UnauthorizedAccessException`、`ArgumentException`、`ArgumentNullException`、`DirectoryNotFoundException`、`PathTooLongException`、`IOException`、`SecurityException`。再查一下第 5 行程式，則可能會丟出 `ObjectDisposedException` 和 `IOException` 這兩個例外。

真的是「不查不知道，查了嚇一跳」，短短的兩行 C#程式居然可能會丟出 9 個例外[31]，比列表 25-3 的總程式碼行數還要多。Teddy 曾經有一年用 C#開發一個檔案同步軟體，由於檔案（數位資料）對使用者而言非常重要，如果檔案同步軟體沒寫好，資料同步出錯導致檔案損毀或遺失，這樣的軟體沒有人敢用。因此，在開發檔案同步軟體的時候更要特別關注它的正確性與強健度。

由多年使用 Java 的經驗得知，處理檔案的時候一定會遇到很多例外狀況。剛開始寫 C#程式的時候還真是不習慣，因為 C#編譯器不會提醒開發人員那些函數會丟出例外，全靠自己查 MSDN

[30] 必須要在 Visual Studio 的開發環境中，將滑鼠移到函數身上，才可以看到該函數所丟出的例外。如果不使用 Visual Studio 開發軟體，寫好程式之後直接呼叫 C#編譯器來編譯，就無法直接看到這些資訊。

[31] 請鄉民們判斷一下，這 9 個例外，有哪些是代表 design fault，也就是 bug，不需要用例外處理做法來應付。有哪些是代表 component fault，應該要思考如何處理這些例外狀況。關於 component fault 與 design fault 的說明，請參考〈CH8：強健性大戰首部曲：威脅潛伏〉。

線上文件。有許多從 C#轉戰 Java 的開發人員也經歷過類似相反的情境：為什麼到處都是 checked exception，好煩啊。

好幾年前關於 checked exception 是否有必要，曾經有過一番爭辯。以下兩篇文章，分別訪問 Java 之父 James Gosling 與 C#之父 Anders Hejlsberg，堪為經典，值得一讀。

- B. Venners, “Failure and Exceptions: A Conversation with James Gosling, Part II,” http://www.artima.com/intv/solidP.html, 2003.
- B. Venners and B. Eckel, “The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II,” http://www.artima.com/intv/handcuffsP.html, 2003.

在著名的 Stack Overflow 網站上，也有不少關於 checked exception 的討論，鄉民們也可參考，例如：

- http://stackoverflow.com/questions/613954/the-case-against-checked-exceptions
- http://stackoverflow.com/questions/tagged/checked-exceptions

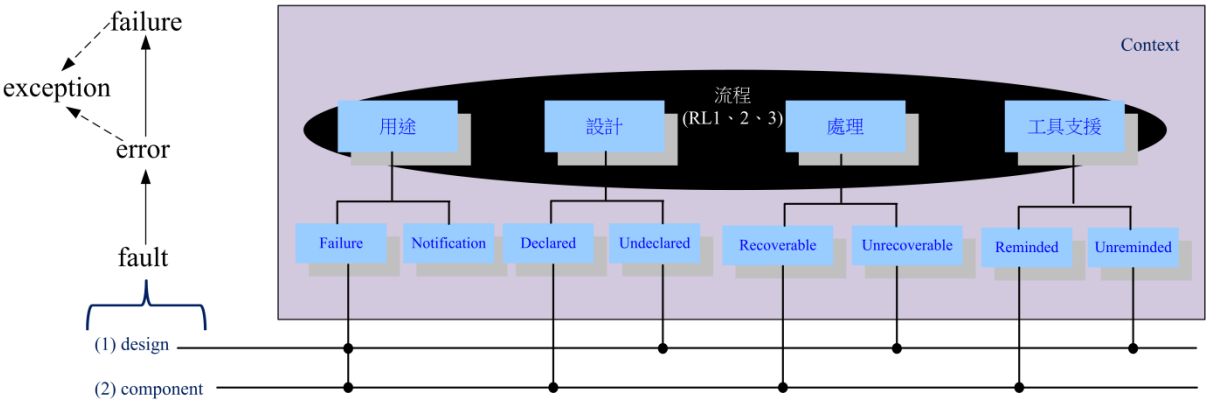
***

了解工具支援觀點後，現在鄉民們可以從另外一個面向來看待 checked exception：「它只是 Java 語言所提供的一種工具支援罷了」。無論鄉民們使用哪種程式語言，開發環境如果可以幫忙驗證例外的使用情況，對於開發強健性軟體會有很大幫助。

***

友藏內心獨白：查文件太累，眼不見為淨，乾脆假裝沒看到例外好了。

26 流程觀點



流程觀點串起了用途、設計、處理、工具支援這四個觀點。這個觀點探討的不是例外本身的含意或是用途，而是從軟體生命週期的角度，來討論例外處理活動所扮演的角色，以及遭遇到的困難。

例外處理設計也是軟體開發的一個環節，要能夠被落實就必須要考慮如何把這個環節妥善的「卡入」軟體開發流程當中。無論是瀑布式（waterfall）流程也好，敏捷與精實開發方法也罷，當程式設計師遇到例外的時候，內心裡總是反覆迴響著一句小小的吶喊聲音：

I will handle this exception when I have time（當我有空的時候我會來處理這個例外）

很遺憾，身為開發人員的你只要一「有空」老闆就會丟更多的工作給你，所以你永遠都不會「有空」，因此例外也一直被忽略（迷之音：好慘啊！）。看到這邊有些鄉民們可能會想：「不對啊，在敏捷方法中，不是可以用循序漸進的方式，先把一個使用者故事（user story，簡稱 story）區分為正常情境（normal scenario）與異常情境（exceptional scenario），然後在這個 sprint（開發週期）中先做正常劇本，等到下個 sprint 再安排時間來對付異常劇本，這樣就搞定了啊。」

所以說，敏捷開發讓例外處理變得好簡單啊。

才怪!

***

上述說法聽起來很有道理，但實務上問題出在：**「處理正常劇本的時候遇到例外怎麼辦？」**。以下為一個採用 Scrum[32]的開發團隊，在某次 sprint planning meeting 進行中的對話：

Product Owner：這個 story 我們要實作透過自訂的網路封包傳遞資料的功能。

ScrumMaster：如果大家都清楚這個功能，請出牌。

ScrumMaster：請翻牌。嗯，看來差異不小，有沒有人要發表一下自己的看法。

開發人員甲：我出 3，因為封包格式老早就已經設計好了，我們只需要實作傳遞封包和解封包的程式就好了，應該蠻簡單的。

開發人員乙：我出 13，我覺得沒有那麼簡單耶。這個功能最後是要給公司其他團隊的人使用。如果他們沒注意到封包格式的正確性，系統很容易因為格式錯誤而當機，或是存在安全性的疑慮。這些問題也應該一併加以考慮進去吧。

開發人員甲：你說的沒錯，可是這個 sprint 我們還有很多其他 story 要完成，我們現在就需要考慮這些例外狀況嗎？

Product Owner：我看這個 sprint 先不用管格式錯誤的問題好了，假設封包格式一定是正確的，之後我們有時間再來安排格式錯誤的處理。

***

[32] Scrum 是一種敏捷開發方法，規範了角色（Product Owner、ScrumMaster、Team、Stakeholder）、活動（sprint planning meeting、daily Scrum、sprint review、retrospective、product backlog refinement workshop）、產出物（product backlog、product backlog item、sprint backlog、task board、burndown chart、running software）。關於 Scrum 的說明可參考 Teddy 的第一本書：《笑談軟體工程：敏捷開發法的逆襲》，悅知出版，2012。

如果現在是在演童話故事或是偶像劇，以上劇情演完就可以殺青收工。雖然 Product Owner 同意先不需要考慮封包格式錯誤的問題，但並不會因為 Product Owner 的一句話就讓開發人員在處理正常劇本的情況下不遭遇到例外。

Sprint planning meeting 結束之後，開發團隊開始實作 story。假設開發人員甲寫了一個 readMessageV1 函數來實作讀取封包的功能，如列表 25-1 所示，該函數從 DataInputStream 得到一個位元陣列，並且依據應用系統規格所定義的網路封包格式，把這個位元陣列傳給 Message 類別的建構函數，然後傳回產生好的 Message 物件。

```
1: public Message readMessageV1(DataInputStream aIS) throws IOException {
2:     int length = aIS.readInt();  //可能會丟出 IOException
3:     byte[] messageBody= new byte[length];
4:     aIS.readFully(messageBody); //可能會丟出 EOFException 和 IOException
5:     return new Message(length, new String(messageBody));
6: }
```

列表 25-1：readMessageV1 函數

奉 Product Owner 指示，開發人員甲現在只想處理正常狀況，但是在寫程式的時候還是會遇到 IOException 與 EOFException，此時程式設計師可能會：

- 既然 Product Owner 說先不處理異常狀況，所以就先直接往外丟好了，把 IOException 宣告在 readMessageV1 函數介面上。
- 雖然 Product Owner 說先不處理異常狀況，但是宣告在 readMessageV1 函數介面上只是把例外處理的問題丟給它的呼叫者去煩惱而已，還是應該把例外捕捉下來然後先暫時忽略它比較好。所以把程式寫成如列表 25-2 所示的內容比較好：

```
1: public Message readMessageV2(DataInputStream aIS) {
2:     Message result = null;
3:     try{
4:         int length = aIS.readInt();
5:         byte[] messageBody= new byte[length];
6:         aIS.readFully(messageBody);
7:         result = new Message(length, new String(messageBody));
8:     } catch(IOException e){
9:         // TODO
10:    }
```

```
11:     return result;
12: }
```

列表 25-2：readMessageV2 函數

- 噯呀，不行啦，忽略例外是一種例外處理壞味道，如果不想處理 checked exception 又不想把它宣告在介面上面，應該改丟出一個 runtime exception，請看列表 25-3：

```
 1: public Message readMessageV3(DataInputStream aIS) {
 2:     Message result = null;
 3:     try{
 4:         int length = aIS.readInt();
 5:         byte[] messageBody= new byte[length];
 6:         aIS.readFully(messageBody);
 7:         result = new Message(length, new String(messageBody));
 8:     } catch(IOException e){
 9:         throw new RuntimeException("Unhandled exception.", e);
10:     }
11:     return result;
12: }
```

列表 25-3：readMessageV3 函數

***

同樣是「先不處理例外」，一個簡單的函數就有至少三種不同的寫法。如果開發團隊沒有取得共識，整個系統關於例外處理的程式風格會十分混亂。日後就算是真的擠出時間想要加強例外處理，也會變得十分的困難。

看到這邊鄉民們可能會想：「Teddy 你舉的例子都是 Java 的例子，因為 Java 有 checked exception 才會變得這麼麻煩啊。如果上面這段讀取封包的程式是用 C#開發，就不會有這些問題了」。

好，應鄉民們要求，讓我們「在公堂之上假設一下」，有一個語言叫做 Java--（Java 減減），語言的全部特性都和 Java 一模一樣，唯一的差別就是 Java--不支援 checked exception。列表 25-4 程式採用 Java--所撰寫，內容和之前的 readMessageV1 函數一模一樣，不過所有的例外都是 unchecked exception，所以不需要在介面上宣告 throws IOException，未被捕捉的例外會自動往外傳遞：

```
1: public Message readMessageV4(DataInputStream aIS) throws IOException {
2:     int length = aIS.readInt();  //可能會丟出 IOException
3:     byte[] messageBody= new byte[length];
4:     aIS.readFully(messageBody); //可能會丟出 EOFException 和 IOException
5:     return new Message(length, new String(messageBody));
6: }
```

列表 25-4：readMessageV4 函數

鄉民甲：你看吧，就告訴你要用 unchecked exception。這樣不是很棒嗎，只有一個版本。不像剛剛使用 Java，會產生三種不同的寫法。

一個函數如果不想捕捉例外，只想把所有發生的例外往外傳遞，Teddy 把這種處理方式稱之為錯誤回報（error-reporting）。採用這種方式的函數，具備有「強健度等級 1」的能力。關於強健度等級的詳細說明請參考〈CH27：例外處理設計的第一步：決定強健度等級〉。回到 Java--的程式範例上面，拿掉 checked exception 之後，在預設狀況之下，不需要捕捉任何例外，每一個函數便達到Product Owner所要求的先不處理異常狀況，這是採用unchecked exception的優點。但是，再思考一下，萬一日後想要回頭處理封包格式不正確的異常狀況，要如何著手？兩個禮拜、一個月、三個月、半年之後，誰還記得當初 readMessageV4 函數沒有處理封包格式異常的問題？

鄉民甲：很簡單啊，寫個註解不就好了，請看列表 25-5。

```
1: // TODO 有空要處理封包格式不正確的例外狀況
2: public Message readMessageV5(DataInputStream aIS)  {
3:     int length = aIS.readInt();
4:     byte[] messageBody= new byte[length];
5:     aIS.readFully(messageBody);
6:     return new Message(length, new String(messageBody));
7: }
```

列表 25-5：readMessageV5 函數

在程式裡面寫類似的註解，就跟沿路丟麵包屑來指路的效果差不多。不敢說完全沒用，但缺少強制力，而且很容易被遺忘。

***

就好像需求會改變的道理一樣，程式對於強健度的要求，也會隨著時間而改變。理論上系統強健度應該是要隨著時間演進而逐漸增強，但若開發團隊沒有一套可行的演進式例外處理設計方法，而是依據開發人員的個人喜好來決定處理方式，則敏捷開發法不但不會讓例外處理變得更簡單，反而增加困難度。

***

友藏內心獨白：軟體開發的問題，一定要放到流程裡面來討論。

Column E. | 你如何評價成功

2014 年 1 月中旬收到 YA 先生的來信：

Dear Teddy

看到下面這樣的消息...
Teddy 哥，你真的覺得內在美（軟工、Exception Handling）重要嗎 ？
我開玩笑的啦...我當然知道很重要...
只是被你嫌棄的「XXX」，也能被估值 OO 億。
這世界上，大家都是看表面的，是嗎 ？

YA 先生

「XXX」是台灣某個新創公司，Teddy 曾經用過他們的產品，一開始覺得 XXX 提供的服務很好，幫 Teddy 解決了一個問題。經過好幾個月的使用，在這段期間內，XXX 系統雖然經過多次升級，但系統的穩定度卻沒有隨著升級而提升，bug 還是很多，多到 Teddy 一度想衝到對方公司去教他們如何除錯與開發軟體。經過多次與對方反應但仍未改善，後來 Teddy 決定放棄，不再使用 XXX 的產品。

最近得知 XXX 成功募資，而且號稱目前價值 OO 億，YA 先生和 Teddy 很熟，故意寫信來「酸」Teddy—「你看吧，產品品質好又怎樣，人家品質差一樣募到很多資金啊。」

***

成功有很多面向，有的人很會造勢，舉辦各式各樣的活動，和媒體、政府搞好關係，成功獲得知名創投或大公司的資金。只要一切合法，這種做法並沒有不好，反正一個願打一個願挨，這也是人家的本事與厲害的地方，應該值得幫對方拍拍手。換成是 Teddy，就沒有這種本事可以去要到那麼多的錢。

Teddy 只是就事論事，從一個使用者，同時也剛好是軟體開發者的角度，XXX 的系統品質之前真的不行（有一陣子沒使用了，不知道現在品質有沒有好一點），這個「事實」和對方是否成功募資本來就是兩回事。**希望 XXX 有充足的資金之後，可以拿來改善他們產品的品質**，品質變好了，Teddy 也會多一種選擇（因為 XXX 競爭對手的品質現在也快不行，Teddy 快沒產品可用了…Orz）。

如果「市值」就代表成功，那「日月光」、「味全」、「統一」都是非常成功的企業，鄉民們也不用氣著去追究他們「產品品質」的好壞、是否偷偷排放廢水。

***

還記得 N 年前第一波.COM 風潮的時候，當初也是跟著自己公司的老闆朱先生拜訪了很多創投與其他公司老闆。有一次朱先生經朋友介紹，到 F 公司拜訪對方的董事長。據說這家公司當年在中國就已經非常成功，有多厲害就有多厲害，而且公司業務與 Teddy 的公司互補性很高，應該有很大的合作機會。

見到對方董事長之後，Teddy 覺得對方給人一種很奇怪的感覺，也說不上來為什麼，**就是覺得不太像是「一般的生意人」**。朱先生很努力地跟對方說明公司的業務與對方其實有很多可以互相合作的地方，但對方似乎一點都沒有興趣。**那種無所謂的感覺，讓人覺得對方好像對於他自己公司的業務與未來發展也沒有任何興趣，非常奇怪的感覺。**

那次見面之後沒有什麼結果，後來回公司，Teddy 才想起來，有一次去投標一個政府機關製作網頁案子，這個案子前一年度是由 Teddy 的公司得標，第二年度的標案，得標的公司就是 F 公司。當初投標的時候是 Teddy 去投標的，F 公司也有去投標，但是他們以低於底價 60%的金額得標。換句話說，一個底價 100 萬的案子，對方用 40 萬就得標了。因為低於底價太多，導致 F 公司還要先繳交一筆保證金，才可以跟政府簽約。當初開標之後，F 公司派來的代表還跟 Teddy 以及另一家投標公司的人說：「早知道你們價錢都寫這樣，大家應該先溝通一下。」

以上都不是重點，約略過了 1~2 年左右，有一天 Teddy 看到一則新聞：「F 公司負責人 YYY 夫婦，涉嫌在台灣、美國販賣未上市股票詐財 OO 億元。」

天啊，原來 F 公司的董事長是靠詐欺發財，難怪他對於和公司「名目上的業務項目」，會一點都不感興趣啊。

***

有些事，看看就好，自己該做什麼，就去做什麼。

***

友藏內心獨白：是誰說過他出生就是要來做公益的。

第五部 強健度等級與例外處理策略

27 例外處理設計的第一步：決定強健度等級

這一章 Teddy 想談一下例外處理設計 (exception handling design) 這個問題。曾經有學者研究過，軟體中大概有 2/3 的程式都是用來做例外處理或是錯誤處理。既然例外處理佔了軟體系統那麼大的篇幅，理當受到開發人員極大的重視才對。錯！在實務上，相較於討論如何設計正常功能的文獻（書籍、論文、文章、程式範例），關於如何做例外處理設計的討論就少了很多。鄉民們可以回想一下，在求學的過程中，在程式語言的課程或是軟體設計課程中，有多少老師曾經教過你如何設計例外處理？在出社會之後，你自己、你同事、你老闆，又有多少人和你討論過例外處理該如何設計？

如果有，在此先恭喜老爺，賀喜夫人。如果沒有，也沒關係，看看本篇就是一個好的開始。

***

例外處理在軟體開發中面臨的困難

從**軟體開發生命週期**的角度來看，要把例外處理做好是件蠻難的工作。為什麼難，簡單可歸類下面三個原因：

- 例外處理屬於非功能性需求，很容易被忽略：大體上軟體開發的順序，是先從功能性需求 (functional requirement) 開始做起，然後再考慮像是可用性、安全性、效能、強健性、易用性等非功能性需求 (non-functional requirement，或稱為 quality attribute)。在軟體開發專案普遍面臨開發時間與人力不足的問題，因此例外處理這類非功能性需求就成為被忽略與犧牲的對象。開發人員常常催眠自己與同伴：「讓我把功能做出來，先過了這一關再說，以後有時間再來處理例外。」可是，通常等功能寫好之後，這些暫時被列為 TO DO（代辦事項）的例外，就永遠被世人給遺忘了。
- 有些例外從需求面看不到，要等到實作才會出現：寫過使用案例（use case）的鄉民們都知到，使用案例有所謂的失敗情境（failure scenario），因此在撰寫需求時分析師便可規劃對於 failure scenario 所遭遇到的異常情況要如何處理。**但是，有很多例外是和實作方法或是選擇的軟體元件有關，因此這種與實作相關的例外就沒有被列在需求分析中，而需要依靠開發人員的良心來辦事。**至於開發人員有沒有良心，套句檢察官的口頭禪：「全案已進入司法審查階段，不便對外說明」。不過，Teddy 可以確定的是，就算是很有良心的開發人員，也不見得有能力把例外處理做好（請看下一點）。

- 真的不知道要怎麼處理：寫過 Java 程式的人都知道，呼叫與 I/O（輸入、輸出）相關的函數時，會丟出 `IOException` 這個 checked exception（請參考〈CH20：Checked 與 Unchecked 例外的語意與問題〉）。但是有沒有什麼通用的設計準則告訴開發人員，當你收到 Java 贈送的 `IOException` 這份「禮物」要如何處理？好像沒有（有的話請好心地通知 Teddy）。因此，要如何處理就只能任憑開發人員各自的判斷。有人會反射性地把例外捕捉之後直接忽略，然後假裝沒事繼續做其他功能；有人把例外往外丟，將這份「禮物」轉送給下一個人；有人會把例外記錄到日誌檔中然後認為這樣就算是把例外處理好了；有的人捕捉這個例外然後丟出一個新型別的例外；有的人捕捉例外之後，嘗試修復例外造成的問題，但是通常越補越大洞導致問題越改越多。簡單地說，在大部分情況下，由於不知道例外要如何處理，不同的開發人員依據自己的喜好隨機做出決定，而這樣的決定通常會降低程式的強健度。

***

定義例外處理等級

Teddy 建議的方法很簡單，就是請開發團隊定義軟體的「強健度等級」，請參考表：27-1。強健度等級就是例外處理的需求或是驗收條件，有了它開發人員遇到例外就知道要如何處理才能滿足這個需求。有點玄，Teddy 先解釋一下這四個強健度等級的含意，再舉例子說明。

表：27-1[33]

項目	等級 0	等級 1	等級 2	等級 3
名稱	未定義（Undefined）	錯誤回報（Error-reporting）	狀態回復（State-recovery）	行為回復（Behavior-recovery）
服務	隱含或外顯的失敗	外顯的失敗	外顯的失敗	持續提供
狀態	未知或不正確	未知或不正確	正確	正確
生命週期	終止或繼續	終止	繼續	繼續
如何達成	NA	1. 傳遞所有未被處理的例外 2. 在主程式中回報所	1. error handling 2. cleanup	1. fault handling 2. retry, and/or 3. design diversity,

[33] 表：27-1 與圖 27-1 修改自 C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, I-L. Wu, “Exception handling refactorings: Directed by goals and driven by bug fixing,” Journal of Systems and Software, vol. 82, issue 2, 2009, pp. 333–345。

		有的例外		functional diversity, data diversity, temporal diversity
別名	NA	failing-fast	weakly tolerant	strongly tolerant

等級 0：未定義（Undefined）

如果鄉民們還沒有幫程式貼上強健度等級標籤，這些程式就屬於未定義這個等級。未定義表示當某個服務（service，可視為整個系統、軟體元件、函數呼叫、或是服務導向架構裡面的服務呼叫）發生錯誤的時候，可能會讓呼叫者知道錯誤發生，也有可能會假裝沒事 (failing implicitly or explicitly)。也就是說使用該服務的人無法確切得知它是否有成功達成任務。當錯誤發生的時候，服務處於不明或是錯誤狀態。例外發生時系統可能會終止也可能繼續執行。

等級 1：錯誤回報（Error-reporting）

錯誤回報表示當某個服務發生錯誤的時候，一定要讓呼叫者知道，絕對不能假裝沒事 (failing explicitly)。因此，使用該服務的人便可確切得知它是否有成功達成任務。當錯誤發生的時候，因為沒有執行錯誤回復動作，系統可能處於不明或是錯誤狀態。例外發生時系統要終止執行，因為此時狀態已經不明，所以繼續執行下去可能會讓整個系統錯得更離譜。

要達到錯誤回報強健度等級很簡單，就是把所有的例外都往外丟，然後在主程式（整個系統最外層的那個程式）捕捉所有的例外並回報給使用者或開發人員知道。錯誤回報又稱為「早死早投胎」（failing-fast），〈CH28：強健度等級 1：錯誤回報的實作策略〉對於錯誤回報的實作方法有詳細的說明。

等級 2：狀態回復（State-recovery）

和錯誤回報一樣，狀態回復要求當某個服務失敗的時候，一定要讓呼叫者知道，絕對不能假裝沒事。和錯誤回報不同，狀態回復要求當錯誤發生之後，服務必須保證系統還是處於正確狀態。由於整個系統的狀態還是正確的，因此例外發生之後系統可以繼續執行。

要達到狀態回復強健度等級要多做兩件事情。首先是**錯誤處理（error handling）**，讓系統回復到一個正確的狀態。假設有一個服務修改了資料庫內容，當例外發生時就要執行復原（rollback）動作。其次是**釋放資源（cleanup）**。例如，把要來的記憶體、檔案、資料庫連線等資源釋放。狀態回復又稱為弱容錯（weakly tolerant），〈CH **錯誤! 找不到參照來源。**：強健度等級 2：狀態回復的實作策略〉對於狀態回復方法有更進一步的介紹。

等級 3：行為回復（Behavior-recovery）

從強健度等級 3 的名字鄉民們應該可以猜到這個等級的軟體是很有責任感的，要求使命必達。因此，當某個服務執行失敗的時候，要想辦法排除困難，總之就是要達成原本被賦予的任務。和狀態回復相同，行為回復要求當錯誤發生之後，服務必須保證還是處於正確的狀態。由於整個系統的狀態還是正確的，因此例外發生時系統可以繼續執行。

要達到強健度等級 3 除了要做到強健度等級 2 的 error handling 和 cleanup 之外，還需要「想其他方法達成原本的任務」。為了不讓下次執行再度失敗，需要先啟動**缺陷處理（fault handling）**以排除造成失敗的原因。造成錯誤的缺陷排除之後，便可套用**重試（retry）**與**設計多樣性（design diversity）**、**功能多樣性（functional diversity）**、**資料多樣性（data diversity）**、**時序多樣性（temporal diversity）**等設計技巧來嘗試持續提供服務。行為回復又稱為強容錯（strongly tolerant），〈CH30：強健度等級 3：行為回復的實作策略〉對於上述行為回復設計方法有詳細的說明。

鄉民甲：等一下，萬一遇到 311 日本大地震這種超大災難，一個強健度等級 3 的服務使出渾身解數之後還是沒辦法達成使命，那怎麼辦？

Teddy：請參考圖 27-1，此時強健度等級 3 降級為強健度等級 2，若再失敗則降級為強健度等級 1。

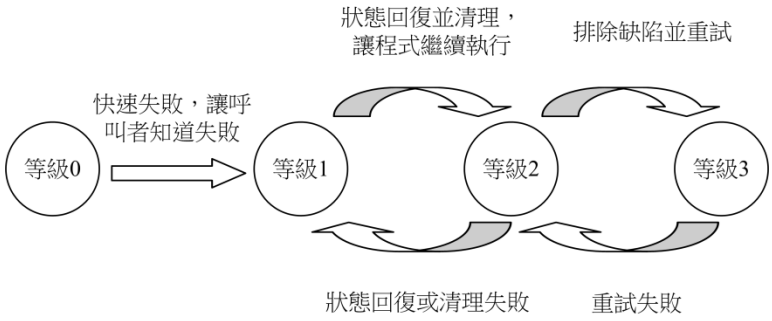


圖 27-1：強健度等級的升級與降級

***

要怎麼用

重點來了，這些強健度等級要如何使用？以下是 Teddy 實際在團隊中推導此作法的步驟：

1. 花 1~2 小時對團隊成員宣導強健度等級觀念。沒錯，只需要 1~2 小時，因為觀念本身很簡單，不需長時間訓練便可了解。
2. 在沒有特殊情況下，預設所有函數一定要達到強健度等級 1。如此一來，在開發階段經由各種人工或自動化測試手段，團隊便可盡量找出應該處理而沒有被處理的問題。在做這個規定之前，有不少例外被開發人員有意或無意地忽略，因此從使用者介面上看不到這些例外發生時所產生的錯誤訊息，導致後續除錯變得非常困難。乍看之下強健度等級 1 好像很不負責地把所有例外都往外丟，但是如此一來反而可在**開發階段**及時發現問題並加以修復。將問題暴露之後，已經有足夠的情境可決定該例外的處理方式是否必須由強健度等級 1 提昇至更高的等級，因此整體而言能提昇軟體的強健度。
3. 對於特定函數，例如資料庫處理，一旦資料庫資料錯誤會造成使用者很大的困擾，加上資料庫處理有交易（transaction）功能可以使用，因此預設應該達到強健度等級 2（除非所開發的系統只是為了展示用的雛型）。
4. 除非客戶特別要求，或是不達到強健度等級 3 系統會變得很難用（例如網路資料傳遞，如果不能自動保證資料可以完整無錯誤地傳遞到遠端，系統將會變得很難用），否則不會特別要求函數要做到行為回復。

Teddy 多年來採用此方法從實務經驗得知還蠻可行的，**強健度等級這個觀念不但簡單易懂，而且也很符合敏捷開發精神**。為什麼？因為強健度等級秉持「階段性，逐步改善例外處理設計」的精神。在許多情況下，正常功能還沒有全部完成時，是不太容易決定例外處理到底應該做在整個系統的那一層。此時過早、過於精細的例外處理實作不見得有用，反而可能造成時間上的浪費。舉個例子，假設你採用 Scrum 開發軟體，而有某個功能（例如，權限管理）需要 2 到 4 個 sprint 才可以把全部的正常功能做完。當然在做此功能的第一個 story 就有可能會遇到很多例外，如果此時對於要如何處理這些例外還沒有很清楚的解法，可以規定這個 story 只要滿足強健度等級 1 或 2 即可。等這個功能組的全部 story 都做完，或是等到做完足夠多的 story，讓該功能組的例外處理變得可以決定之後，再增加一個 story 來提昇現有 story 的強健度。如此一來，因為客戶已經知道每一個已完成 story 的強健度，所以可以讓客戶在「增加新功能」與「提昇強健度等級」這兩種不同類型的 story 之間做取捨。

***

友藏內心獨白：如果有人「自願」把自家的稻米..嗯嗯...產品降級，算不算是一種「自甘墮落」的行為？

28 強健度等級 1：錯誤回報的實作策略

錯誤回報的基本精神就是不可以忽略例外，在 C#這種只支援 unchecked exception 的語言，例外傳遞採取自動傳遞（implicit）方式，如果開發人員只想先撰寫正常邏輯，暫時不想處理例外，在程式中只要不捕捉例外即可。但是，在 Java 語言中，因為 checked exception 必須要符合「處理或宣告原則」規則，不想處理的 checked exception 要特別將例外宣告在函數介面才能夠往外傳遞。但是宣告在函數介面上的例外如果日後有異動，又會造成介面改變，影響所有呼叫該函數的客戶端程式，處理不好的話會降低系統的穩定度。

本章介紹達到錯誤回報等級的幾點實作技巧，可適用於 Java 或是 C#這種只支援 unchecked exception 的程式語言。

不可忽略例外

在絕大部分的情況之下，不可以用一個空的 `catch` block 捕捉例外並忽略它。

```
1:     catch (IOException e) {
2:         // 忽略例外是不好的錯誤示範
3:     }
```

就算是捕捉例外之後將其印出，其效果也幾乎等同忽略例外，因為在程式執行的時候，使用者或是開發人員很難觀察到這些錯誤訊息。

```
1:     catch (IOException e) {
2:         e.printStackTrace();  // 印出例外效果幾乎等同於忽略例外
3:     }
```

只有在某些特殊狀況才可以在不實際處理例外的情況下，捕捉例外。例如，在 `finally` block 裡面呼叫的函數不應該丟出例外（原因請參考〈CH14：我的例外被 Finally Block 蓋台了〉）。以下的 `close` 函數將被使用在 `finally` block 中用來關閉資源物件，在這種情況下捕捉例外並把錯誤訊息寫入日誌檔中而非將例外往外傳遞，是可接受的作法。

```
1: public void close(AutoCloseable res){
2:     try{
```

```
3:         res.close();
4:     }
5:     catch(Exception e){
6:         logger.error("關閉資源錯誤", e);
7:     }
8: }
```

另一個常見的情況請參考本章的最後一個技巧：「建立安全網」。

自動傳遞 Unchecked Exception

在程式中不要捕捉 unchecked exception，讓它自動往外傳遞。下列 C#程式的第 4、5 兩行可能會產生 IOException、ArgumentException、SecurityException、PathTooLongException、DirectoryNotFoundException、UnauthorizedAccessException、ObjectDisposedException、ArgumentNullException、以及 NotSupportedException 這麼多個 unchecked exception。在預設狀況下這些例外都會自動往外傳遞，因此 writeFile 函數不需要處理這些例外即可達到強健度等級 1 的要求。

```
1: public void writeFile(String filePrefix, String data){
2:    TextWriter writer = null;
3:    try {
4:        writer = new StreamWriter(filePrefix + "_" + data + ".txt");
5:        writer.Write(data);
6:    } finally {
7:      // cleanup code
8:    }
9: }
```

使用 UnhandledException 傳遞 Checked Exception

針對 checked exception 的傳遞，定義一個 RuntimeException 的子類別 UnhandledException，將捕捉到的 checked exception 串接在 UnhandledException 身上，然後丟出這個 UnhandledException 代表目前該例外還沒有被處理。

```
1: catch (IOException e) {
```

```
2:     throw new UnhandledException(e);
3: }
```

不要寫入日誌檔之後再丟出例外

有些開發人員為了達到「徹底」的錯誤回報，會想辦法先捕捉所有的例外，將其寫入日誌檔，然後再把這個例外原封不動的重複丟出（rethrow）。這種作法其實沒有意義，不但會傷害程式的效能，因為相同一個例外被記錄了好幾筆，也會造成日後除錯的困擾。

```
1: catch (FileNotFoundException e) {
2:     logger.error("找不到系統主設定檔", e);
3:     throw new UnhandledException(e); //不要寫入日誌檔之後又丟出例外
4: }
```

建立安全網

如果一味的將例外往外丟，都沒有人捕捉，最後將導致程式異常終止，丟出使用者看不懂的錯誤訊息。因此，需要在程式的最外面（主程式以及每一個執行緒）統一用一個 `try` statement 來建立安全網，捕捉所有的例外以避免程式異常終止。此時除了可以將例外轉成使用者比較容易理解的錯誤訊息，還可以統一將例外狀況寫入日誌檔中。如此一來，便不需要在所有的函數裡面重複捕捉、寫日誌檔、重新丟出相同的例外。

```
1: static public void main(String [] args){
2:     try{
3:         /*
4:          * 做一大堆事情的主程式
5:          */
6:     }
7:     catch (Throwable e){
8:         /*
9:          *  顯示錯誤訊息並且記錄到日誌檔中
10:         */
11:    }
12: }
```

要達到強健度等級 1 其實並不需要花費太多額外的功夫，成本低廉但卻可為日後增強系統強健度打下良好的基礎。話雖如此但實際上因為很多開發人員缺乏例外處理的知識，未達強健度等級 1 的程式還是可以很容易看到。看完本章之後，請鄉民們回去看看自己所寫的程式，是否有符合強健度等級 1 的要求。

***

友藏內心獨白：如果一直作假，就不可能有改善的機會。

29 強健度等級 2：狀態回復的實作策略

狀態回復需要做到 error handling 與 cleanup 這兩個操作，兩者都是為了讓系統保持在正常狀態，其差異可以簡單的如下區分：**Error handling 關注於系統內部的狀態**，例如資料結構、物件以及全域變數的狀態，還有系統所使用的檔案或是資料庫內容是否正確。**Cleanup 強調「外部資源（作業系統或是執行環境）」的清理**，以避免資源洩漏，例如記憶體洩漏（memory leak）、檔案處置器（file handler）或資料庫連線用盡等問題。

在〈CH18：Try、Catch、Finally 的責任分擔〉從實作的角度說明在 Java 程式語言（C#也適用）中如何將 error handling 與 cleanup 操作對應到程式語言的例外處理結構。本章說明幾項常見的 error handling 與 cleanup 技巧。

Error Handling

Error handling 的目的在於**移除錯誤所造成的不正確狀態，將系統回復到一個沒有錯誤的狀態（error-free state）**。Error handling 的方法在容錯系統中著墨很多，本章參考 Lee 與 Anderson[34]、L. L. Pullum[35]與 Avizienis 等人[36]的著作來介紹這些觀念。Error handling 可以分成兩大種類：**向後回復（backward recovery）與向前回復（forward recovery）**，本章介紹 backward recovery，至於 forward recovery 已經屬於容錯設計的領域，可以達到強健度等級 3，將在下一章介紹。

Backward recovery 是指當錯誤發生的時候，將系統回復至錯誤發生之前所保留的狀態，所以稱為向後回復（從時間點的角度來看，回復到一個比較舊的狀態，也就是錯誤發生之後所做的改變等於沒有發生）。Backward recovery 假設系統發生錯誤之前處在正確狀態，這個假設需要成立，否則保留這個狀態就沒有意義。Backward recovery 需借助不會被系統失效所影響的**穩定儲存裝置（stable storage）**，例如硬碟，來保留狀態。我們在例外處理中所說的 error handling 就是指採用 backward recovery，依據其實作方法不同，又可以細分為以下兩種：

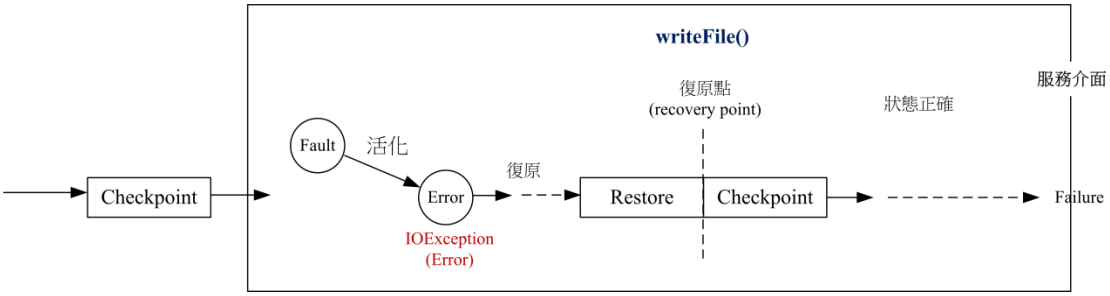
- 基於狀態的（state-based）：儲存一個完整的正確狀態，待錯誤發生之後丟棄整個錯誤狀態，以所儲存的正確狀態來取代。例如查核點（checkpointing）、快照（snapshot）、時光機（time machine）等都屬於這一類。

[34] P. A. Lee and T. Anderson, *Fault Tolerance: Principles and Practice, 2nd ed.*, Springer, 1990.
[35] L. L. Pullum, *Software Fault Tolerance Techniques and Implementation*, Artech House, 2001.
[36] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, 2004.

- 基於操作的（operation-based）：不儲存完整的正確狀態，而是儲存改變狀態的事件或是操作。當錯誤發生之後，利用這些儲存下來的狀態改變記錄，反向取消原本的操作以達到狀態復原的效果。例如稽核軌跡（audit trail）、日誌（logging）等。

基於狀態的向後回復（State-Based Backward Recovery）

下圖修改自〈CH8：強健性大戰首部曲：威脅潛伏〉中圖 8-2 的範例，現在考慮幫 `writeFile` 函數增加 backward recovery。在執行 `writeFile` 函數之前，先製作一個查核點。執行 `writeFile` 函數的過程中因為活化了某個 fault 導致 error 產生，此時執行狀態回復（restore）動作，使用剛剛製作的查核點來恢復系統狀態。`writeFile` 無法達到其原先規格所設定的功能，因此其執行結果為 failure，但是因為做了 backward recovery，所以最後系統狀態處於正確狀態，符合強健度等級 2 的規範（假設 `writeFile` 沒有發生資源洩漏的問題）。



***

基於操作的向後回復（Operation-Based Backward Recovery）

以操作為基礎的向後回復方法，採取復原（undo）原本操作的方式，來復原系統的狀態。舉個例子，假設某線上遊戲被駭客入侵，有些玩家的寶物被盜，此時系統處在一個不正確的狀態。在這種情況下，因為整個線上遊戲世界一直在改變，因此系統並沒有存在一個合適的查核點（正確狀態的備份）可以用來執行狀態復原。如果系統平常針對寶物買賣的交易與玩家狀態改變都有妥善記錄在日誌檔中，則可以利用日誌檔的資料來反向回推出每一位受害玩家的正確狀態。

要用這種方法來復原狀態，系統必須記錄足夠的額外資訊。常見的方式有使用**冗餘資料**與**冗餘函數**來達成狀態復原。以線上遊戲為例，每一筆寶物交易資料或是玩家的狀態改變，都必須要寫入到日誌檔（冗餘資料），日後才可以利用這個日誌檔來恢復資料。另一個常見的程式設計

技巧就是套用 *Command* 設計模式[37]，將每一個操作包裝成一個獨立的類別，在此類別中提供 do 和 undo（冗餘函數）兩個動作相反的函數，如此便可透過 undo 函數來執行狀態復原。

***

向後回復的優點

Backward recovery 有很多優點，首先只要錯誤沒有對 backward recovery 機制發生影響，它便可復原由 fault 所造成的各種不預期錯誤。以 `writeFile` 函數為例，不管造成錯誤的原因是 `NullPointException` 也好，是檔案讀寫錯誤也罷，都可以用相同的 backward recovery 方法來復原系統狀態。

其次，backward recovery 是一種通用狀態回復方法，它與應用程式或應用領域無關。再者，實作 backward recovery 不需要特別去研究錯誤發生的原因，反正不管是什麼原因，只要狀態有錯就復原至原本保存的狀態就對了。就好像鄉民們使用作業系統提供的「快照」功能，製作快照之後，不管是電腦中毒或是檔案被誤刪，都可以用之前準備好的快照將系統回復到一個正確的狀態。

最後，backward recovery 很適合用來復原因為暫態故障（transient fault，又稱為瞬時故障）所造成的錯誤。暫態故障顧名思義就是只會出現一瞬間的故障，例如電力系統突然不穩定造成瞬間斷電或是電壓不穩。利用 backward recovery 來復原由暫態故障所造成的系統狀態錯誤，在狀態復原之後只要重新執行一次原本的操作，就有很高的機會可以成功繼續執行。例如，使用文書處理器的時候資料打到一半突然停電導致電腦關機。電來了之後重新開機打開文書處理器，如果文書處理器有實作 backward recovery 機制，便可以自動幫我們把資料復原到停電之前的狀態，讓工作繼續下去（除非真的很幸運，重開機之後又再度停電…）。

向後回復的限制

Backward recovery 也存在一些先天性的限制，首先它很**耗費資源**。製作與復原查核點需要電腦計算資源、時間、儲存空間。舉個例子，下圖是 Teddy 的 Mac Book Air 筆電正在執行製作系統快照的畫面，離上次備份約兩個禮拜，本次備份需要一個小時以及 2.53GB 硬碟空間。

[37] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

其次，在保存與復原系統狀態的當下，backward recovery 方法可能會要求系統暫時停止（halt）運作、降低系統的服務等級或是拉長系統的反應時間。最後，在分散式系統或是採用巢狀 backward recovery 的應用中，當一個函數或是行程（process）復原到之前的狀態，可能會引發骨牌效應，觸發其他函數或是行程的 backward recovery 動作。

***

Cleanup

由作業系統或是執行環境（例如 JVM 或是.NET Runtime）所管理的外部資源，像是記憶體、硬碟空間、資料庫連線，一般來講都是有限的共用資源。所謂「有借有還，再借不難。有借無還，再借免談。」雖然作業系統或是執行環境不至於立即做到「有借無還，再借免談」這麼冷酷無情的地步，但是使用完畢的資源如果遲遲沒有歸還，最後會導致資源耗盡，系統也無法再繼續運作，通常只得重新啟動程式或是重新開機。

Error handling 的實作策略，除了像是資料庫交易處理這種特定的應用與程式語言和使用的函式庫或 API 有關以外，其他像是查核點、快照、稽核追蹤等策略，大多與程式語言無關。相較於 error handling，cleanup 的實作則和所使用的程式語言緊密相關，大致可分成兩個議題：

- 記憶體管理：採用自動回收或是交由程式開發人員自行管理？
- 資源釋放：資源釋放的機制為何？

有些鄉民以為在 Java 與 C#這種支援記憶體自動回收（garbage collection）的程式語言裡面，就不會發生記憶體洩漏的問題，這是不正確的。雖然相對於 C++這種需要自己管理記憶體的程式語言而言，支援記憶體自動回收的程式語言發生記憶體洩漏的機會大幅減低，但也不是可以完全避免。Bloch 在《Effective Java, 2nd》[38]中舉出了三種可能發生記憶體洩漏的狀況：**自己管理記憶體、製作快取（cache）、使用監聽器（listener）或回呼函數（callback function）**，很值得一讀。C++管理記憶體的技巧，則可參考《Effective C++, 3rd》[39]、《More Effective C++》[40]、《Exceptional C++》[41]。

關於支援 `try-finally` 的程式語言，例如 Java 與 C#，釋放記憶體以外的外部資源作法，請參考〈CH13：Java 的 Try、Catch、Finally〉。

***

友藏內心獨白：要讓程式「行得正坐得穩」，保持狀態正確，也是要下一番苦工。

[38] J. Bloch, *Effective Java, 2nd ed.*, Addison-Wesley, 2008, p. 24-26.
[39] S. Meyers, *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, 3rd ed.*, Addison-Wesley, 2005.
[40] S. Meyers, *More Effective C++: 35 New Ways to Improve Your Programs and Designs*, Addison-Wesley, 1996.
[41] H. Sutter, *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*, Addison-Wesley, 1999.

30 強健度等級 3：行為回復的實作策略

行為回復代表例外發生之後軟體元件還可以繼續提供正常的服務，對外部使用者來說，就好像這個例外根本沒有發生過一樣。從例外處理的角度，要做到行為回復，需要考慮 fault handling 與 retry 這兩個操作。從容錯設計的角度，如果採行 forward recovery 的狀態回復方式，也可以達到強健度等級 3。接下來逐一介紹這三個操作。

Fault Handling

系統失效的原因是因為程式的執行引爆了某個 fault，導致 error，最後形成 failure。強健度等級 2 修正了 error 狀態，但是如果沒有進一步的把系統中的 fault 給排除，系統繼續執行下去還是有很高的機會再度引爆同一個 fault，結果還是產生 failure。Fault handling 就是探討如何排除 fault 的方法。嚴格來講，fault handling 比較偏向容錯處理的領域，但是在某些特定的狀況之下，例外處理還是可以應用 fault handling 的技巧來提升系統的強健度。

依據 Avizienis 等人[42]的分類，fault handling 包含以下四個操作：

- 診斷（diagnosis）：鑑定與記錄錯誤發生的原因，包含錯誤發生的位置與種類。
- 隔離（isolation）：將失效的元件從系統中隔離，以防止後續的操作繼續使用到這個失效的元件。
- 重新組態（reconfiguration）：如果系統有冗餘或是備用的元件，則將系統重新組態以便使用這些冗餘或備用元件。或是重新指派工作，讓沒有發生問題的元件來取代被隔離的元件。
- 重新初始化（reinitialization）：確認重新組態之後的系統是可運作的之後，重新設定系統以便採用這個新的組態。

Fault handling 的四個操作在日常生活中也經常會套用。例如，鄉民們連續三天使用電腦打報告或是寫程式，最後搞到自己的右手很痛，幾乎無法使用滑鼠（發生 failure）。經過醫生或是自己自行檢查之後（診斷），判斷是手腕出了問題，要休息一個禮拜不能使用滑鼠（隔離）。但

[42] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, “Basic Concepts and Taxonomy of Dependable and Secure Computing,” IEEE Transactions on Dependable and Secure Computing, vol. 1, no. 1, 2004.

是公司並不會因此允許你一個禮拜不用上班，你還是要想辦法讓自己在強健度等級 3。因此你重新調整工作模式，改用左手來使用滑鼠（重新組態）。改變之後發現雖然工作速度變慢了一些，但是至少還可以繼續幫公司賣命。最後你決定重新安排桌面，把滑鼠和滑鼠墊移到左邊，改用左手來操控（重新初始化）。

具備 fault handling 的系統需要有足夠的**知識**與**資源**，才能夠診斷與隔離 fault，進而做到自我調適（重新組態與重新初始化）。

***

Retry

成功的執行 fault handling 之後，只要在新的系統配置之下重試原本的操作（此時原本操作的實作方式可能已經改變，例如採用備用元件取代原本失效的元件），系統就應該可以提供正確服務。

重試的結果能否成功很大程度受到 fault handling 的重新組態所影響。針對暫態缺陷或是間歇缺陷，重新組態的實作結果有可能並不需要改變任何系統的配置，僅透過**重試原本實作（retry with the pirmary）**便可以成功。如果是永久缺陷，或是想要徹底解決間歇缺陷，只是重試原本的實作顯然是行不通的。因此，重新組態階段便必須要使用替換的元件或是新的執行路徑，來取代有缺陷的元件。此時的重試就稱為**重試替代方案（retry with alternative）**。

製作與執行替代方案的技巧，可以分成四大類：

- 設計多樣性（design diversity）：將同一份規格交給不同團隊設計與實作出多個功能相同的版本。設計多樣性在軟體容錯設計中是一種很常見的作法，常見的方式有 **N 版本程式設計（N-Version Programming；NVP）**與**回復區塊（Recovery Block）**。N 版本程式設計的結構如圖 30-1 所示：在同一時間一個函數的多個版本同時執行，其執行結果交由決策器來判斷，最後選出一個正確的結果。例如，假設有一個加法函數 `add` 交由三個團隊，採用設計多樣性策略來達到軟體容錯。在執行階段，`add` 的三個實作版本，針對 1、2 這兩個輸入（準備執行 1+2 運算），分別傳回 3、3、-1，最後由決策器決定執行結果為 3。雖然

這三個實作版本有一個存在著 fault（很可能是 design fault），但因為採用設計多樣性的容錯策略，這個 fault 並沒有對系統服務造成影響。

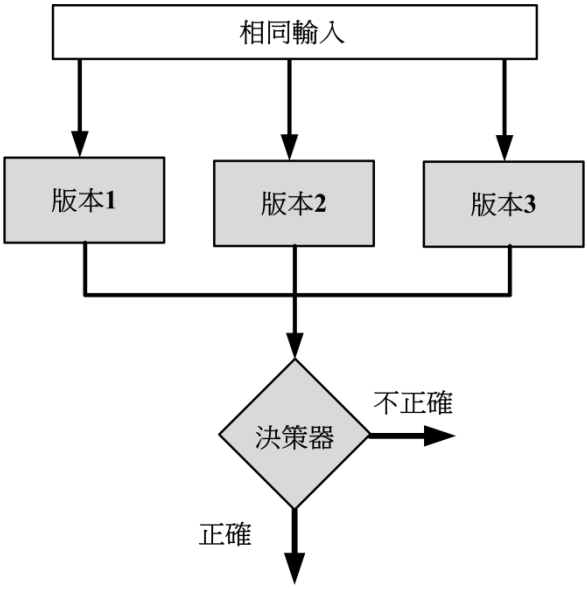


圖 30-1：N 版本程式設計結構圖

回復區塊的結構如圖 30-2 所示：首先產生查核點，接著執行主要方案。如果主要方案執行過程中沒有發生例外，且其結果通過驗收測試，則代表主要方案執行成功，丟棄查核點之後便可正常離開回復區塊。反之，若執行主要方案發生例外或是其執行結果沒有通過驗收測試，則使用查核點來回復系統狀態。接著判斷是否有替代方案可供選擇，而且尚未超過執行期限（例如最多執行三種不同替代方案，或是執行時間不能超過 30 秒）。如果條件都成立則重新執行替代方案，若不成立則代表回復區塊失效，丟出代表 failure 的例外。

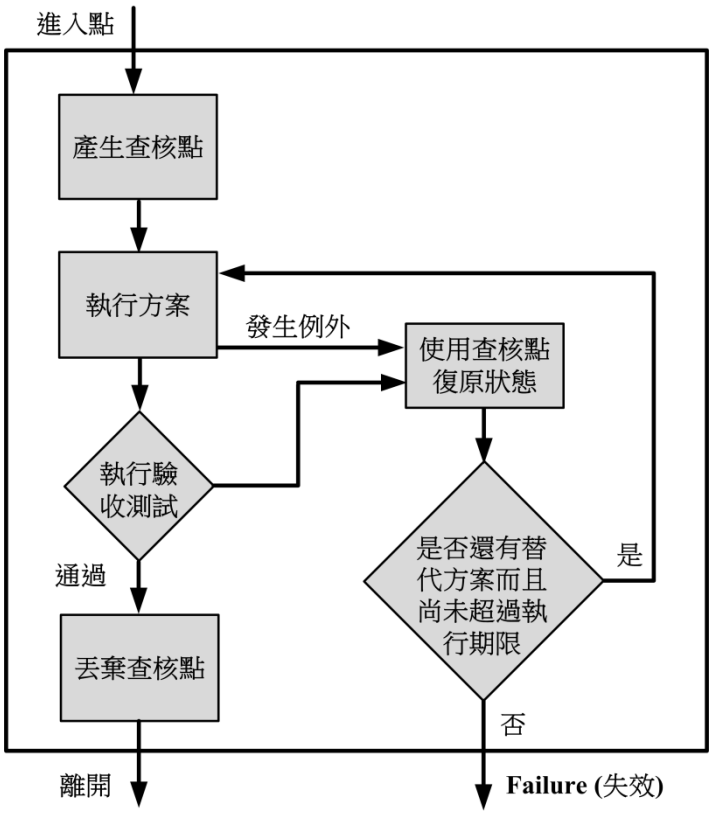


圖 30-2：回復區塊結構圖

因為設計多樣性的成本實在太高，尤其是 N 版本程式設計，因此實務上在例外處理領域較少有機會需要採用這個策略來達到強健度等級 3。比較接近一點的作法則是用函數來模擬回復區塊的結構，針對已知可能會失效且重要性較高的函數，提供一種或多種實作方式的替代方案。但不像容錯設計所要求的設計多樣性，需要由不同團隊來設計與實作個別版本，在例外處理的領域，替代方案通常由同一個團隊設計與實作。

- 功能多樣性（functional diversity）：廣義的來講，功能多樣性也屬於設計多樣性的一種，不同之處在於功能多樣性採用**不同的輸入**與**不同的設計、實作**方式，來達到相同的功能（設計多樣性則是採用相同的輸入，不同的設計、實作）。功能多樣性又稱為多才多藝的系統（resourceful system），這種系統可能需要具備人工智慧的能力，**系統知道如何透過不同的方式來完成相同的目標。**例如，假設鄉民們使用一個具備功能多樣性的高鐵訂票函數來預訂過年期間台北到高雄的來回車票。這個函數發現直達車票已經被訂完了，但是它有能力用分段的方式，先訂從台北到台中，然後再訂台中到高雄的車票。針對相同的目標（台北到高雄），具有多種功能可以完成此目標。

看到以上解釋，鄉民們心理應該有底了：在例外處理的領域，除非針對已知且非常重要的功能，否則不太會套用這麼「厚工（繁複）」的技巧。

- 資料多樣性（data diversity）：有時候函數執行失敗是因為輸入的參數有問題，在這種情況下提供多份不同的函數實作可能也無濟於事。因此，如果可以提供不同的資料，也有可能可以達到強健度等級 3。例如，使用者輸入的字型或是顏色資料錯誤，系統就自動選用預設的一組資料作為顯示之用，讓系統可以運作下去。
- 時序多樣性（temporal diversity）：有時候系統因為效能或是時序的問題，導致函數執行失敗。此時只要調整一下函數執行的時間點，就有可能可以解決問題。例如，假設有一個保全監控系統採用輪詢（polling）的方式，每隔 60 秒詢問一次客戶家中的感應器是否正常。在剛開始的時候，客戶只有 100 人，系統每次輪詢都會得到正確的結果。但隨著生意變好，同時間要監控的客戶量增加到 1000 個。因為客戶數量增加，導致每次詢問的失敗比例也增加，不是發生逾時（timeout），就是因為系統負載太重而無法在排定的時間點啟動輪詢。這時候，如果系統可以調整輪詢間隔（polling interval），就有可能降低輪詢失效的機率。

看到這邊不知道會不會有鄉民們覺得這些製作與執行替代方案的方法過於抽象或是太理論？其實並不會，以 Teddy 自身的經驗，除了 N 版本程式設計以外，上述方法都有實際運用於軟體專案的經驗，已確定其可行性，提供給鄉民們參考一下。

關於重試還有最後一件事情要提醒鄉民們，如果發生 fault 的原因沒有排除，無限制的一直重試是沒有意義的，而且很可能會被誤認為是「駭客入侵」報警抓人，搞到客戶人人自危，擔心自己手動在網頁上按個「重新整理」按鈕也被當成是嫌疑犯（迷之音：原來 82 億次駭客攻擊的元兇居然是…「伊踢膝 APP」。這種鍥而不捨、屢戰屢敗、屢敗屢戰，不斷重試的精神，可能是受到國父革命推翻滿清的啟發，抑或是蔣公小時候看魚兒逆流而上的鼓舞。）

***

Forward Recovery

Forward Recovery 是狀態回復的一種方式，但因為它讓系統「向前」回復到一個沒有錯誤的狀態，也等於說沒有發生 failure，因此達到強健度等級 3。

鄉民甲：明明發生 error，函數執行錯誤，為什麼可以「未卜先知」，有辦法讓系統「向前」來到一個沒有錯誤的狀態？不是應該只能「向後」回到函數執行之前的狀態嗎？

是的，Teddy 一開始接觸到 forward recovery 也是覺得很奇怪，都發生錯誤了應該是無法繼續執行，怎麼有可能「向前」來到未來的正確狀態呢？其實很簡單，剛剛介紹過的 N 版本程式設計其實就是一種 forward recovery 加上 design diversity 的容錯技巧[43]。請回頭參考圖 30-1，因為有多個版本同時執行（可能在一台電腦上，也有可能在不同的電腦上），所以即使有任何一個版本發生 error，整個系統還是可以「向前」來到一個沒有錯誤的狀態。

另一種 forward recovery 的做法需要搭配日誌檔，經常應用在資料庫系統。假設鄉民們在一個月前執行了一次資料庫備份，一個月後的某一天，因為不明原因資料庫檔案突然損毀，幸好資料庫的交易日誌檔還存在。因此你利用備份檔，先將資料庫狀態回復到一個月前的狀態，再利用交易日誌，讓資料庫的狀態「向前」來到系統剛剛損毀之前的正確狀態。

有些資料庫針對即時交易提供 forward recovery，當交易進行時會將所需的資料會先搬移到記憶體中，等待交易完成之後，再將交易記錄檔寫入磁碟機中，最後才把記憶體中的資料寫入資料庫。如果交易記錄已經成功寫入磁碟機，但是記憶體中的資料尚未寫入資料庫，此時如果資料庫系統當機，待下次啟動之後，資料庫會參考交易記錄的資料，自動將資料庫的內容向前回復到正確狀態。

***

友藏內心獨白：要做到使命必達還真是要付出不少心血。

[43] 回復區塊則是 backward recovery 加上 design diversity 的容錯技巧。

Column F. | VMWare 越獄之替代方案

虛擬化技術這幾年變得十分流行，記得早在 10 多年前 VMWare 剛出來的時候，Teddy 的同事發現了這個好東西，於是我們就在 VMWare Workstation 上面建了好幾個虛擬機器（virtual machine；VM），用來安裝不同版本的瀏覽器作為測試之用。2009 年之後 Teddy 改用免費的 VMWare Server 3.x 來作為虛擬化測試平台，它支援透過網頁介面來產生與管理虛擬機器，很方便也很好用。

VMWare Server 用了一陣子之後，Teddy 發現了幾個不方便的地方，於是再度改用 Oracle VirtualBox 3.2 版，並將原本 VMWare Server 所產生的虛擬機器轉移到 VirtualBox 上面。整個過程大致上還算順利，VirtualBox 可以直接讀取 VMWare 所製作的虛擬機器，只需要做一些設定之後便可使用，不過，重點來了，原本跑在 VMWare 虛擬機器裡面的測試程式，搬到 VirtualBox 上面居然會出錯！不是都說虛擬化就是可以把虛擬機器隨便移來移去的嗎，現在是什麼情況？

經過一番奮鬥之後，終於發現原來問題出在 VirtualBox 對於桌面管理介面（Desktop Management Interface；DMI）的模擬支援實在是太遜了，不像 VMWare 做的那麼好，而 Teddy 使用的測試程式剛好需要透過 DMI 讀取系統有關主機板的資訊，所以就槓龜了。

稍微解釋一下 DMI 是做什麼的，簡單的說，就是電腦 BIOS 會記錄著一些主機板相關的資料，這些資料被作業系統映射到某一塊特定的記憶體中。如果應用程式需要得知與硬體相關的資料，例如主機板製造商、型號、BIOS 版本、實體記憶體大小等，便可以讀取這些資料。讀取的方法有很多種，在 Linux 作業系統中有一個叫做 dmidecode 的程式可以直接看到這些資料，資料內容大概長成這個樣子：

```
root@Ubuntu:~# dmidecode
# dmidecode 2.9
SMBIOS 2.5 present.
30 structures occupying 1477 bytes.
Table at 0x0009F400.

Handle 0x0000, DMI type 0, 24 bytes
```

BIOS Information

Vendor: American Megatrends Inc.

Version: R01-C0L

Release Date: 06/17/2009

Address: 0xE0000

Runtime Size: 128 kB

ROM Size: 1024 kB

Characteristics:

ISA is supported

PCI is supported

PNP is supported

APM is supported

BIOS is upgradeable

BIOS shadowing is allowed

ESCD support is available

Boot from CD is supported

Selectable boot is supported

BIOS ROM is socketed

EDD is supported

5.25"/1.2 MB floppy services are supported (int 13h)

3.5"/720 KB floppy services are supported (int 13h)

3.5"/2.88 MB floppy services are supported (int 13h)

Print screen service is supported (int 5h)

8042 keyboard services are supported (int 9h)

Serial services are supported (int 14h)

Printer services are supported (int 17h)

CGA/mono video services are supported (int 10h)

ACPI is supported

USB legacy is supported

LS-120 boot is supported

ATAPI Zip drive boot is supported

BIOS boot specification is supported

Targeted content distribution is supported

BIOS Revision: 8.14

Handle 0x0001, DMI type 1, 27 bytes

System Information

Manufacturer: Acer

Product Name: Aspire M3203 <----- **這是 Teddy 花了 9999 所買的電腦型號**

Version:

Serial Number:

UUID: Not Present

Wake-up Type: Power Switch

SKU Number: To Be Filled By O.E.M.

Family: To Be Filled By O.E.M.

(族繁不及備載，以下省略)

***

講到這邊，以上所說和例外有何關聯？雖然這些 DMI 資料和硬體有關，但並不是所有的主機版製造商都會提供完整的資料，但是幾個常見的資料結構都還是會存在的。因此，Teddy 在撰寫讀取 DMI 程式的時候根本不會想到會出現「讀不到 DMI 資料」的狀況。在 VMWare 中，Teddy 所需要的 DMI 資料剛好都有被模擬到，因此測試程式在 VMWare 虛擬機器裡面執行都沒問題。但是，VirtualBox 模擬的 DMI 資料很少，程式搬到 VirtualBox 後就出問題了。

原本 Teddy 的程式達到強健度等級 2 狀態回復，但是現在為了解決這個問題，必須要把強健度等級提昇到 3，也就是要作到行為回復。

程式是用 Java 所開發，但是讀取 DMI 的程式則是透過呼叫由 C 語言所寫的原生程式（native code）來完成，因為這種方式執行速度最快。但是這段程式碼在 VirtualBox 就是無法正常執行，而且

不知道為什麼會讓整個 JVM 卡住。為了先避開這個問題，於是對於相同的工作，必須要提供另外的實作，也就是要準備替代方案。修改之後的程式有一個主要方案與一個替代方案：

- 主要方案：透過呼叫外部程式執行 Linux 的 dmidecode 程式來讀取 DMI 資料，如果這個方法失敗，則改用替代方案。因為 dmidecode 程式已經成為 Linux 內建的程式，因此原則上主要方案應該都會成功。但是考慮到也許不同的 Linux 版本的 dmidecode 輸出格式可能不同，以及萬一遇到很念舊的使用者把程式安裝在很古老的 Linux 版本沒有內建 dimdecode 程式，因此還是保留一個替代方案當做備胎。
- 替代方案：以 native code 的實作方式讀取 DMI 資料，雖然執行速度比較快，但考量到在 VirtualBox 會導致 JVM 卡住的問題，因此列為替代方案。

***

鄉民甲：為什麼不在開發之初就把程式設計成強健度等級 3 呢？

Teddy：如果把每一個函數都設計成強健度等級 3，則開發成本最少會變成原來的兩倍，因為要有一個主要方案和至少一個替代方案。如果考慮到測試與整合工作，成本會遠遠超過原來的兩倍。此外，在設計階段很可能根本沒辦法預期到會有哪些錯誤情況發生，也無從事先針對不同錯誤情況設計替代方案。**因此，採取演進式的策略，逐步提昇程式的強健度是比較符合敏捷開發的精神。**

***

友藏內心獨白：不須做過多的先期設計，強健度是可以依據需要而逐步提升的。

31 例外類別設計與使用技巧

前幾章談了達到不同強健度等級所採用的例外處理策略，這一章要談另一個也是很重要的議題：例外類別設計與使用技巧。這個議題算起來屬於〈CH28：強健度等級 1：錯誤回報的實作策略〉的範疇，因為在程式開發中主要是透過例外來回報錯誤。在〈CH28：強健度等級 1：錯誤回報的實作策略〉裡面主要從實作的角度來討論如何達到所要求的強健度等級，本章則是專門討論例外類別的命名、宣告、傳遞、設計等技巧。

本章前兩個技巧參考 Wirfs-Brock[44]的建議，其餘技巧參考 Haase[45]所採用的術語。Haase 採用模式（pattern）的格式來介紹這些技巧，在本書中將這些模式以斜體字標示。

用哪裡出了問題而不是用誰丟出例外來命名

「命名」一直是撰寫易讀、易懂程式的基本功夫，像 Beck 在《Implementation Patterns》，Martin 在《Clean Code》都提到命名的技巧。Wirfs-Brock 認為關於例外類別的命名，應該以「哪裡出了問題」來命名。例如一個自動提款機（ATM）類別的 `withdraw` 函數，負責實作提款功能。如果存款餘額小於提款金額，則執行 `withdraw` 函數會發生錯誤而丟出一個例外。假設鄉民們要設計一個新的例外類別來代表提款錯誤狀況，可能有以下幾個候選名稱：

1. `ATMException`
2. `WithdrawException`
3. `NotEnoughMoneyException`

前兩者的名稱比較接近「誰丟出了例外」，而 `NotEnoughMoneyException` 則是以「哪裡出了問題、出了什麼問題」來命名。如果鄉民覺得 `NotEnoughMoneyException` 太過具體，用這種觀點來宣告例外 ATM 類別可能會有太多例外物件，則可以改用比較一般性的 `TransactionException` 來代表所有交易例外，然後把「錢不夠用」當成 `TransactionException` 的錯誤內容。

---

[44] R. Wirfs-Brock, “Toward exception-handling best practices and patterns,” IEEE Software, 23 (5), 2006, pp. 11-13.

[45] A. Haase, “Java idioms: exception handling,” EuroPLoP’2002, 2002.

***

將低階層例外轉成高階層所理解的例外

這個實務做法建議鄉民們將低階層例外轉成高階層所理解的例外，不要將低階層的實作例外（implementation exception）直接跨層（layer）傳遞給上層的人，因為這樣一來收到例外的人會被底層的實作細節給綁住，造成不必要的相依性。假設鄉民們有一個儲存設定檔的函數 saveProperty，這個函數的第一版實作採用檔案來儲存設定檔，所以當存檔失敗之後會丟出 IOException：

```
public void saveProperty() throws IOException
```

如果有一天實作方法改變，改成用資料庫來儲存設定資料，則 saveProperty 函數執行失敗之後就不會丟出 IOException，而是改丟出 SQLException：

```
public void saveProperty() throws SQLException
```

上述兩種作法，都是將底層實作例外直接丟給上層的人，並不是好的做法。應該將例外轉成上層呼叫者看得懂的例外，在此套用「用哪裡出了問題而不是用誰丟出例外來命名」這個做法，改丟出 PropertyManipulationException：

```
public void saveProperty() throws PropertyManipulationException
```

這樣子不管 saveProperty 函數的實作方式如何修改，也不會影響到拋出例外的型別，而且有助於上層呼叫者的例外處理設計。

***

Homogeneous Exception（同質性例外）

程式語言採用 exception 來代表 failure，如果一個函數會因為很多種原因造成 failure，則該函數可能會丟出多種不同類型的 exception。這種現象在採用 unchecked exception 的情況下並不會造成很大的問題，因為 unchecked exception 不需要被宣告在介面上，所以例外數量與型別改變，都不會造成函數介面改變。

但如果是在採用 checked exception 的情況下，函數介面同時宣告了多個不同的 checked exception，則函數很可能會因為例外數量或是型別改變，造成介面的改變。為了避免這個問題，開發人員可以設計一個新的例外類別，然後用這個例外類別來代表同一個類別所有函數可能會丟出的例外。這種例外類別稱之為**同質性例外（homogeneous exception）**。例如，一個 ATM 類別的 `login`、`withdraw`、`deposit` 函數可能會遇到 `IOException` 和 `SQLException`：

```
login() throws IOException, SQLException
withdraw() throws IOException, SQLException
deposit() throws IOException, SQLException
```

鄉民們可能會想，上述作法違反了「將低階層例外轉成高階層所理解的例外」，直接暴露了實作細節。因此，把函數宣告改成：

```
login() throws loginException
withdraw() throws WithdrawException
deposit() throws DepositException
```

修改後的版本雖然符合了「將低階層例外轉成高階層所理解的例外」這條建議，但是卻可能產生過多例外類別。如果鄉民們並不希望一個 ATM 類別的不同函數各自丟出不一樣的例外，則可套用 *Homogeneous Exception* 模式來解決這個問題：

```
login() throws ATMOperationException
withdraw() throws ATMOperationException
deposit() throws ATMOperationException
```

然後將每一個函數產生例外的原因串接到 `ATMOperationException` 身上，以便接收到例外的人可以有足夠的脈絡資源進一步地處理。

Homogeneous Exception 可以緩解因為使用 checked exception 所造成的函數介面改變的問題。因為一個函數只宣告一個或少量的 *Homogeneous Exception*，所以可以避免函數直接將實作細節所遭遇的例外直接宣告在介面上往外傳遞，也有助於實踐「將低階層例外轉成高階層所理解的例外」這條規則。

***

Unhandled Exception（未處理的例外）

在實作函數功能的時候，你遇到了 checked exception，但是你當下只想要實作正常邏輯，並不想要處理例外行為。這時候，不可以用一個空的 `catch` block 捕捉例外並忽略它，而是應該先定義一個屬於 `RuntimeException` 的 `UnhandledException`，再將捕捉到的 checked exception 串接在 `UnhandledException` 身上，最後丟出這個 `UnhandledException`，用以代表該例外目前還沒有被處理，例如：

```
1: catch (FileNotFoundException e) {
2:     throw new UnhandledException(e);
3: }
```

***

Tunneling Exception（通道例外）

有時候一個函數不能宣告自己要丟出什麼 checked exception，例如你要實作別人定義的回呼函數（callback）或是 *Command* 設計模式，如果原本的回呼函數或是 *Command* 沒有宣告會丟出任何 checked exception，你的實作也就無法丟出任何的 checked exception。

因此，定義一個繼承自 `RuntimeException` 的 `TunnelingException`，丟出這個 `TunnelingException`，並且把原本想要丟出的 checked exception 串接到 `TunnelingException` 身上，例如，Java 的 `Runnable` 介面只包含 `run` 這個函數，並沒有

宣告會丟出任何 checked exception。如果實作 run 函數又想要丟出 checked exception，則可套用 *Tunneling Exception*，請參考以下範例：

```
1: static public void main(String args[]){
2:     try{
3:         CommandExecutor executor = new CommandExecutor();
4:         executor.execute(new Runnable(){
5:             public void run(){
6:                 System.out.println("Encounter SQLException");
7:                 // 模擬實作遇到 SQLException
8:                 throw new TunnelingException(new SQLException());
9:             }
10:        });
11:    }
12:    catch(TunnelingException e){
13:        if (e.getCause() instanceof SQLException){
14:            // handle the SQLException
15:        }
16:    }
17: }
```

CommandExecutor 沒做什麼事，只是直接呼叫 run 函數。

```
1: public class CommandExecutor {
2:     public void execute(Runnable runnable) {
3:         runnable.run();
4:     }
5: }
```

Tunneling Exception 雖然最常以丟出 RuntimeException 子類別來串接 checked exception，但並非固定都是如此。例如，Java 用來處理 XML 的 ContentHandler 介面，裡面有好幾個函數都宣告會丟出 SAXException（一個 checked exception），這個 SAXException 也是扮演 *Tunneling Exception* 的角色。也就是說，不管實作上可能會遭遇什麼例外，都可以透過這個 SAXException 來傳遞。

還有沒有其他例子？有，Java 的 `AutoCloseable` 介面就只有一個函數：

```
void close() throws Exception
```

這個 `Exception` 也是扮演著 *Tunneling Exception* 的角色。還有和 `Runnable` 用途很接近的 `Callable<V>`介面，它的 `call` 函數彌補了 `Runnable` 無法傳回值與丟出 checked exception 的限制：

```
V call() throws Exception
```

***

Exception Wrapping（例外包裝）

若一個函數宣告它會丟出 *Homogeneous Exception*，則將原本發生的例外串接在 *Homogeneous Exception* 身上（把一個例外串接到另外一個例外裡面，這個動作叫做 wrapping，包裝、包裹的意思）。讓客戶端的程式捕捉到 *Homogeneous Exception* 的時候，可以透過被打包的例外物件，知道例外發生的根本原因。

Exception Wrapping 是一種很常見的拋出例外技巧，像是 *Unhandled Exception* 與 *Tunneling Exception* 也都會套用這個技巧。

***

Smart Exception（聰明例外）

客戶端程式捕捉到例外之後，有時候需要依據例外發生原因而設計不同的例外處理方式。例如當你遇到 `LoginException` 這個例外，如果是帳號不存在，你會問使用者是否要新增帳號。如果是密碼錯誤，你會顯示重新登入或是寄發密碼的選項。要如何讓捕捉到例外的人可以進一步依據例外發生的原因，來設計不同的處理方式？

設計一個列舉類別（enumeration class）來代表錯誤發生的原因。在例外類別之中包含這個列舉類別，使得例外物件變「聰明」，只要用一個例外物件便可以代表很多種異常狀況。例如，Java 語言的 `SQLException` 就是一種 *Smart Exception*，它有一個 `public int getErrorCode` 函數，負責傳回資料庫廠商所定義的例外代碼（vendor-specific exception code）。客戶端程式可以直接透過以下方式來依據不同錯誤原因執行不同的處理方式：

```
1: catch (java.sql.SQLException e) {
2:     rollback(con);
3:     if (e.getErrorCode() == EXTERNAL_ROUTINE_EXCEPTION) {
4:         compensate();
5:     }
6:     throw new AggregationException(e, sql); // a failure exception
7: }
```

***

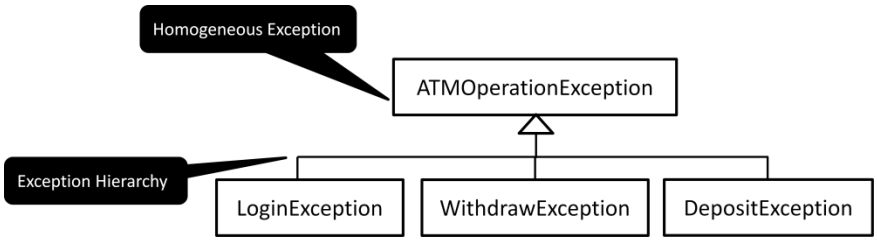
Exception Hierarchy（例外階層）

使用 *Homogeneous Exception* 的好處是客戶端程式可以用相同的方式來捕捉例外，在介紹 *Homogeneous Exception* 所提到的例子，無論是 `login`、`withdraw`、`deposit` 發生例外，只要捕捉 `ATMOperationException` 即可。

```
login() throws ATMOperationException
withdraw() throws ATMOperationException
deposit() throws ATMOperationException
```

但是，*Homogeneous Exception* 的優點也是它的缺點。假設客戶端程式在一個 `try` statement 裡面同時呼叫 `login`、`withdraw`、`deposit` 函數，如果此時想要用不同的例外處理方式分別對付這三個函數執行失敗的情況，但是它們都丟出相同的 `ATMOperationException`，那該怎麼辦？

因此，建立例外類別繼承架構，讓更精確的例外類別繼承 *Homogeneous Exception*。在介面宣告上函數會丟出 *Homogeneous Exception*，在實作上則是丟出例外子類別。如此客戶端程式便可自由的選擇要捕捉 *Homogeneous Exception*，或是其子類別。



Exception Hierarchy 和 *Smart Exception* 都是一種用來提供詳細錯誤資料的方法，前者透過繼承，後者則是透過包含一個錯誤資料列舉物件。*Exception Hierarchy* 可以讓客戶端直接用不同的 `catch` block 來捕捉例外，省卻了 *Smart Exception* 只用一個 `catch` block，但卻需要在 `catch` block 裡面用 `if` 條件式依據列舉物件的值來判斷錯誤原因的作法。

通常，如果錯誤原因不是很多，或是不同錯誤原因的處理方式各不相同，則可能會考慮採用 *Exception Hierarchy* 的方法。例如，Java 的 `IOException` 就是一種 *Homogeneous Exception*，而其子類別有：

```
ChangedCharSetException, CharacterCodingException,
CharConversionException, ClosedChannelException, EOFException,
FileLockInterruptionException, FileNotFoundException, FilerException,
FileSystemException, HttpRetryException, IIOException,
InterruptedByTimeoutException, InterruptedIOException,
InvalidPropertiesFormatException, JMXProviderException,
JMXServerErrorException, MalformedURLException, ObjectStreamException,
ProtocolException, RemoteException, SaslException, SocketException,
SSLException, SyncFailedException, UnknownHostException,
UnknownServiceException, UnsupportedDataTypeException,
UnsupportedEncodingException, UserPrincipalNotFoundException,
UTFDataFormatException, ZipException
```

各自代表不同的錯誤狀況，處理的方式也都不相同，這時候套用 *Exception Hierarchy* 就比較合適。如果改用 *Smart Exception* 則每個 `catch (IOException e)` 的區塊都還要用 `if` 條件式來判斷詳細錯誤原因，很像傳統程序導向的寫法。

Java 的 `SQLException` 除了是 *Smart Exception*，同時也是 *Homogeneous Exception*。在處理資料庫的時候可能遭遇多種不同的錯誤原因，例如刪除的資料不存在、鍵值重複、表格不存在、資料表被鎖住、沒有操作權限等。這些錯誤可能都是因為「執行 SQL 指令所發生的錯誤」，只是產生錯誤的原因不同，再加上很多存取 SQL 資料庫的介面都是用錯誤碼來代表錯誤原因，因此用 *Smart Exception* 比較合適。如果要套用 *Exception Hierarchy*，針對一種錯誤原因就要設計一種例外類別，則需要產生數十到上百個例外類別，數量相當驚人。

其實 `SQLException` 也套用了 *Exception Hierarchy*，有如下數個例外子類別。和 `SQLException` 本身扮演 *Smart Exception* 所代表的錯誤狀況不同，以下這些例外，各用來表示應與 `SQLException` 採用不同處理方法的例外狀況。

```
BatchUpdateException, RowSetWarning, SerialException,
SQLClientInfoException, SQLNonTransientException,
SQLRecoverableException, SQLTransientException, SQLWarning,
SyncFactoryException, SyncProviderException
```

***

例外類別的設計與使用，是例外處理領域比較少討論到的議題。本章介紹的這幾個技巧，相信足以應付大部分的情況。

***

友藏內心獨白：如何使用與設計例外類別還真的是有很多「眉角」啊。

32 終止或繼續

終止或繼續，這是每個人遇到困難都必須要做出的抉擇：

宅男：玩遊戲卡關，要休息，還是要熬夜拼下去？
情侶：對方劈腿，要分手，還是選擇原諒繼續交往？
觀眾：連續劇越演越扯，要轉台，還是繼續看下去？
學生：功課沒寫，要蹺課，還是勇敢的上學去？
員工：公司太爛，要離職，還是繼續苦撐？
老闆：錢燒光了，要關門大吉，還是借錢繼續營運下去？
工程師：期限到了，要無視為數眾多的 bug 把產品推出，還是要無限期除錯下去？

寫程式遇到例外狀況，開發人員同樣要做出終止或繼續的決定。

- 終止：結束目前執行中的函數並丟出例外讓呼叫者處理。
- 繼續：繼續目前執行中的函數，對於執行過程中發生的例外，先暫時寫到日誌檔中，或是以收集參數（*Collecting Parameter*）的方式回傳給呼叫者。

例外處理設計，除了要決定前幾篇文章中所提到的強健度等級以外，還要決定遇到例外的時候程式是要採取終止或繼續的策略。在 C++、Java、C#這些語言，由於採用終止模式（termination model），所以當例外發生的時候預設是採取終止策略，開發人員很自然地就接受這種方法。但有時候例外發生之後終止函數執行並不太合適，例如：

- 執行批次工作：有一個函數要連續新增 100 位使用者資料，新增到第 5 位使用者資料發生錯誤。在終止模式下，函數停止執行並丟出例外，後面 95 位使用者資料就沒有辦法繼續新增。
- 執行週期性或重複性工作：有一個執行緒每隔 5 秒要檢查 CPU 負載並且將檢查結果記錄下來。如果某次檢查因為不明原因導致例外發生，如果採取終止模式而結束執行緒，就違反了週期性執行的需求。

雖然支援例外處理的程式語言告訴廣大群眾：「程式發生錯誤要丟出例外來通知呼叫者」，但是如果你的程式採取的是繼續執行策略，此時就不應該丟出例外了。那要怎麼做？

執行批次工作

接下來看一個執行批次工作的函數，使用 *Collecting Parameter* 收集例外的 Java 範例：

```
1: public void runBatchJob(List<Job> aList, ErrorCollector collector){
2:     for (Job job : aList) {
3:         try {
4:             job.execute();
5:             } catch(Exception e) {
6:                 collector.addError(e);
7:         }
8:     }
9: }
```

`ErrorCollector` 就是 *Collecting Parameter* 的實作，它本身是由呼叫者所傳入的參數，在此負責收集錯誤資料，所以叫做 *Collecting*（收集）*Parameter*（參數），其參考介面如下。鄉民們可以自行擴充 `ErrorCollector` 介面，例如可以增加輸出 `ErrorCollector` 所有錯誤內容的函數，或是包裝加入 `ErrorCollector` 的內容，將 `Exception` 加入 `ErrorCollector` 之外，還可以增加錯誤描述、錯誤等級等資料。

```
1: public interface ErrorCollector {
2:     List<Exception> getErrors();
3:     void addError(Exception error);
4:     void clear();
5:     int size();
6: }
```

套用 *Collecting Parameter* 之後，`runBatchJob` 函數就可以透過它來收集批次執行過程中可能產生的多個例外資訊。

***

執行週期性或重複性工作

週期性或重複性工作通常執行於背景模式，如果執行過程中遇到例外，最常見的方法就是把例外寫入日誌檔。請看下面這個 Java 範例：

```
1: public void run() {
2:   try {
3:       while (true) {
4:           Task task = queue.take(); // blocking call
5:           task.execute();
6:       }
7:   } catch (InterruptedException ex) {
8:       Thread.currentThread().interrupt();
9:   } catch(Exception e) {
10:     logger.error(e);
11:   }
12: }
```

除非發生 `InterruptedException` 通知執行緒結束執行，否則 `run` 函數即使發生例外也會一直執行下去。

***

例外發生時決定終止或繼續將會影響例外處理的實作方法。例如，符合強健度等級 1 的函數若是採用終止模式則必須丟出一個例外（通常是 unchecked exception）。同樣是達到強健度等級 1 等級的函數，例外發生之後若是採用繼續模式，則依據執行的工作屬於批次或是週期性/重複性工作，而可以採用 collecting parameter 或是透過日誌檔來回報錯誤。

***

友藏內心獨白：有些工作的特性就是不適合直接傳回例外並終止執行。

33 自動化更新

有一陣子 Teddy 家裡一堆電子產品好像串通好似的，陸續壞掉，讓 Teddy 想起了一個例外處理設計方法：自動化更新。先談一下 Teddy 維修的慘痛經驗。

維修奇美 42” 液晶電視

2010 年 10 月，Teddy 買 2 年多的奇美電視壞了。當初會買奇美，主要是因為價錢便宜，和日系相同尺寸的電視相比，大約只需 1/3~1/2 的價錢便可帶回家，而且有三年保固。現在電子產品的生命週期都那麼短，與其買一台很貴的日系電視，還不如買國產電視擋一陣子先，萬一 3 年後真的壞了，液晶電視到時候價錢一定降很多，再買一台新的也划算。

沒想到用不到三年就真的壞了，電視完全無法開機。聯絡了原廠的維修師父，服務還不錯，當天晚上就帶著一個電路板直接到 Teddy 家裡來更換，當場就修好了，免費。相信有類似經驗的鄉民們應該不少（維修過程與結果是好是壞就不一定了），以上故事告訴我們：

- 即時上市（time-to-market）搶商機很重要，很多電子產品的品質在上市之初可能只要求做到堪用的程度。
- 消費者為了嘗鮮，心裡也都預期到新的東西可能 bug 很多。

消費者為什麼要去搶買有問題的新產品？如果公司的產品新穎（有賣點）、價錢有競爭力（C/P 值高）、加上售後服務好，就算產品有問題，甚至有點貴，消費者還是很有可能會買單的。但是，如果售後服務很爛，消費者被騙一次之後就不太可能持續購買同一家的產品。

***

要推出一個成功的產品，至少存在著兩個互相衝突的力量（conflicting force）：

- 及時上市
- 品質

以消費性電子產品生命週期越來越短的趨勢，及時上市變得越來越重要，先推出產品把錢騙進來再說。那麼品質有問題該怎麼辦？聰明的生意人就想出了到府維修、到府收送、快速維修甚至終身保固（迷之音：是誰的終身？產品？消費者？還是廠商？），以降低消費者的戒心：「反正早買早享受，有問題隨時送修很方便。」

***

自動化更新是一種安全網

鏡頭回到軟體開發。軟體例外處理的最高境界，就是程式中所有可能發生的例外都被妥善的處理，如此一來在產品使用時期就不會因為有未被處理的例外而導致程式當機或是系統出現不預期的行為。這個理想很好，但是實施起來卻很難，因為：

- 屬於非功能性需求的例外處理很容易被忽略（時間不夠）。
- 有些例外是需求面看不到的，要到實作時才會出現。
- 真的不知道要怎麼處理（實作知識不足）。

就算以上三點都不成問題（團隊時間很多、實作所遭遇的例外有人幫忙分析、開發人員清楚各種例外處理的實作技巧…地表上應該不太容易找到這種團隊），實務上也幾乎不可能把所有的例外都處理好。為什麼？很簡單，因為程式中可能會發生例外的地方與情況太多了，而軟體專案除非被取消了，否則總是有上市時間。因此就算是一個資源很充裕的專案團隊，在軟體上市之前也很難把所有的例外都處理好。

想像一下，原本你開發的軟體可以順利執行在 Windows 2008 64-bit，但是有一天使用者安裝了 SP2（service pack 2）更新程式，你的軟體就無法再執行。這種「因為執行環境改變而導致系統無法順利執行」的情況實在太多了，當然你可以辯解說「我們的程式出生的時候，SP2 都還沒釋出，我們怎麼可能預測未來？這是 SP2 的問題，不是我們的問題。」無論你如何解釋，對於使用者來說，更新 SP2 之後 Windows 跑得好好的，只有你的程式有問題，這不是你的問題，什麼才是你的問題？

假設兩週後產品就要推出，環境相容性測試還沒做好，程式還有一些 bug 還沒處理，怎麼辦？

別擔心，Teddy 介紹你吃這一帖，保證藥到病除，就是：**自動化更新**。

對，就是自動化更新，明知軟體產品有問題還是要硬著頭皮推出，還好有自動化更新這個法寶當做最後的安全網。這就好像廠商告訴消費者：「我們提供到府維修、到府收送、快速維修與終身保固。」這麼好的服務，就放心給它敗家下去吧（小朋友陸續出走中...）。

這時候，消費者早就已經忘了，「為什麼以前映像管的電視看 20 年都不會壞，現在的電視看兩年就壞了。」總之消費者與業者各取所需。

消費者：先買先享受（口袋又薄了）。

業者：先賺先贏（股價又漲了）。

***

結論：軟體功能一開始做得爛不要緊張，自動化更新一定要做好。多學學軟體大廠，多麼認真的更新他們的產品啊，而且更新完畢還會幫你重新開機喔…（路人甲：我跑了三天三夜的實驗數據啊！）

***

友藏內心獨白：對廠商而言，過保固期之後會立即故障的產品，就是好的產品！

Column G. | 升級、降級，傻傻分不清楚


使用 KKTIX 遇到問題？

歡迎光臨 KKTIX 客服中心找解答、問問題、給建議。

或者您也可以寄信到 contact@kktix.com，與我們分享辦活動、報名活動的建議、想法、合作提案等。

與 KKTIX 在社群網站相遇

KKTIX 也在社群網站活躍！常態放送最有趣的活動消息、小技巧、答客問等。

1. Facebook
2. Twitter

客服電話勒，找不到人開罵。

話說 Teddy 將課程的報名系統換到 Registrano 沒幾個月，Registrano 就被 KKBox 併購了，改名為 KKTIX。2014 年 1 月 5 日突然收到 KKTIX 寄來的一封信：

您好，
感謝您對 KKTIX 服務的支持。
您在 KKTIX 上曾經購買過 10 個點數，目前還剩餘 4 點。因為新版網站直接提供單場活動升級，以及組織年費方案，既有點數機制將不再繼續使用。您原本的點數我們會發送 4 張單場活動升級兌換券給您。
您的兌換券號碼為：
.... (以下省略)

這封信來的有點無俚頭，Teddy 完全不知道信中所言代表什麼意思，也沒時間去理它。直到 1 月 6 日，Teddy 想要在 KKTIX 上新增 3 月份「Design Patterns 這樣學就會了：入門實作班」報名網頁，才發現原來 KKTIX 系統改版了…**什麼，系統改版了**，心中隱隱有著一種不祥的預感。重新登入 KKTIX 之後果然沒錯，這次改版改得有點大，介面沒有舊版來的直覺，使用上覺得挺不方便的。

好吧，這可能是 Teddy 個人偏見，也許習慣之後就好了。**先撇開「個人觀感」不談，為什怎麼原本存在舊版的很多功能，在新版本中反而不見了？！**像是 Teddy 經常使用的「手動修改繳費狀態」的功能，就找不到，而且還發現好幾個新版程式的 bug。正想打電話去客服反應，什麼，

算你狠，KKTIX 在本文撰寫時根本沒有提供客服電話，只能透過網路（e-mail、facebook、twitter）跟他們聯繫。

系統更新出錯的時候，需要靠客服人員來宣洩顧客的不滿。**當客戶正在氣頭上的時候，沒有一通可以直接打去「幹譙」的電話，這種「有氣沒地方發洩的用戶體驗」，真的會讓人氣到得內傷。**

好吧，先透過 facebook 留言反應，但是等不到即時的回答，只好厚著臉皮直接去找 KKTIX 的負責人（因為他剛好在 Teddy 的好友名單之中，只是之前沒有直接與他本人聯繫過）。

和負責人反應問題之後，對方解釋了為什麼拿掉某些功能的原因，也很「阿沙力」的表示願意在日後將「手動設定繳費狀態」這個功能加回來，還把「地下版客服電話（他的手機 ）」告訴 Teddy。Teddy 也順便利用這個機會，向對方回報目前所發現的幾個 bug。

***

軟體升級之用戶體驗守則第一條：不應在未溝通的情況下違反與用戶所約定的合約

對於一個線上報名系統，在規劃每次升級時，從「用戶體驗」的角度來看，Teddy 覺得最重要的因素只有一點，那就是**應該儘量遵守之前和用戶所約定的合約**，通俗一點的說法就是：「要維持向下相容性」。

軟體設計領域有一種設計的方法，叫做**依合約設計（Design by Contract；DBC）**。在 DBC 的精神中，子類別（subclass）不可以破壞超類別（superclass）所訂定的合約，也就是超類別的介面所規範的前置條件（precondition）、後置條件（postcondition）、類別不變量（class invariant）。簡而言之子類別的前置條件只能相等或是更弱於超類別的前置條件（也就是說，子類別對呼叫者的要求只能更少或相等於超類別，不可更多），子類別的後置條件只能相等或更強於超類別的後置條件（也就是說，子類別對呼叫者只能提供相等或更多的服務，不可偷工減料）。子類別的類別不變量則是需要加上超類別的類別不變量。

上面這一堆有的沒的像是繞口令的東西，翻成白話文就是說：**「新版軟體（相等於子類別）不可以破壞舊版軟體（相當於超類別）和客戶所簽訂的合約。」**那些合約？例如：

- 原本的收費與服務條件，在已經預付費用的情況下，只可以更優惠，不能更嚴苛（子類別對於前置條件的要求不可強過於超類別的要求。對於後置條件所承諾提供的服務，不可以少於超類別的承諾）。至於新的收費條件，則屬於另訂合約的範疇，子類別可另行約定。
- 原本存在的功能不應該任意消失，如果真的逼不得已要移除，也應事先通知與溝通，事後列出異動記錄（change log），讓用戶知道每一個版本功能的異動狀況。**不然每次改版找不到原本的功能，Teddy 都搞不清楚這到底是 bug，還是「功能調整」，真的令人非常困擾。**
- 原本可以動的功能，不應該在改版之後出問題。系統的穩定度，絕對是影響用戶體驗的重大因素，沒有人喜歡不穩定的升級或改版。**千萬不要為了趕著釋出，而提供低品質的昇級版產品。**

以上三點做得到，說真的就已經很了不起了。很遺憾，以 Teddy 親身使用的經驗，目前台灣市面上兩個比較知名的活動報名系統，針對以上三點都還有努力的空間。

***

也許規劃新版軟體的人會覺得：「手動修改繳費狀況這個功能應該沒有人會用，那就從下一版移除吧。不是都說要簡單設計、減法原則、少即是多嗎…」

可是這個功能 Teddy 經常使用，而且實務上有很「正當」的使用理由。什麼理由？因為以下兩點原因：

- KKTIX 系統設定五日之內沒繳費就會被自動取消報名。因為參加 Teddy 活動的人有些是由公司付費，雖然學員為了幫公司省錢提早報名早鳥票，但是公司的行政流程卻要等到開課前幾天（甚至是開課後拿到發票）才方便轉帳付款，因此幾乎不可能在報名之後的五日之內完成繳費。因為系統沒有支援上述的狀況，所以 Teddy 只好手動將這些人的繳費記錄改成「已繳費」或是「等待繳費」，以免他們的報名記錄被刪除，然後再請對方公司的會計「有空的時候」直接把費用轉到泰迪軟體。
- 第二個情況是出現在公司團報，因為 KKTIX 沒有支援團報功能，假設同公司四個人一起團報，必須要填寫四份獨立的報名資料，然後分別轉帳四次。有很多公司的會計會打電話問 Teddy：「分多次轉帳很麻煩，能不能一次轉帳就好了？」在這種情況之下，Teddy 也只能先將報名者的繳費狀態改成「已繳費」或是「等待繳費」，然後請對方公司會計直接把費用一次轉帳到泰迪軟體。

系統不支援的功能，只好靠人工方式解決。但是現在系統要把人工解決的途徑也拿掉，就真的變得很不方便。

***

在規劃系統升級的時候，應該要稍微站在使用者的角度考慮一下使用情境，以免好意的升級活動對使用者而言變成降級活動。

這樣應該不算是罵人吧。

***

友藏內心獨白：這本書出版後可能沒人想把服務和產品賣給 Teddy 了。

第六部 例外處理壞味道與重構

34 例外處理壞味道

有了強健度等級的觀念（請參考〈CH27：例外處理設計的第一步：決定強健度等級〉），接下來 Teddy 要談一下妨礙程式沒有達到各個強健度等級的常見原因。Teddy 把這些造成例外處理程式不良的原因稱為**例外處理壞味道（exception handling bad smell）**，藉由辨識與消除這些壞味道，可以提升程式的強健度等級。

回傳碼（Return Code）

使用 *Return Code* 來代表例外狀況是由來已久的一種程式設計習慣，尤其是學過 C 語言的鄉民們，因為 C 語言沒有支援例外，因此慣用特殊的函數回傳值來表達錯誤狀況。下列程式片段顯示用 *Return Code* 來表達錯誤狀況的例子，如果 `withdraw` 函數的回傳值為-1，則表示提款錯誤。

```
1: public int withdraw(int amount) {
2:
3:      if (amount > this.balance)
4:            return -1;
5:      else {
6:            this.balance = this.balance - amount;
7:             return this.balance;
8:    }
9: }
```

使用 *Return Code* 來表達錯誤狀況有兩個主要的問題，首先處理正常邏輯的程式和處理異常邏輯的程式交錯在一起，導致程式不易閱讀與修改。其次，開發人員很可能會忽略對於 *Return Code* 的檢查，因此導致錯誤被忽略，降低系統的強健度並且增加除錯的困難度。

下列程式片段顯示一個呼叫 `withdraw` 函數的例子，第 4~6 行程式碼用來檢查錯誤狀況，第 7~9 行用來處理正常狀況。因為使用了 *Return Code* 來代表錯誤，因此正常與錯誤程式碼穿插在一起，無法清楚區分。

```
1: public void useWithdraw(){
```

```
2:
3:     int balance = withdraw(500);
4:     if (-1 == balance) {
5:         // 錯誤處理
6:     }
7:     else{
8:         // 執行正常邏輯
9:     }
10: }
```

***

忽略受檢例外（Ignored Checked Exception）

忽略例外等於隱藏潛在問題，會降低系統強健度並且增加除錯困難度。*Ignored Checked Exception* 是專屬 Java 語言的壞味道，開發人員為了逃避 Java 語言對於 checked exception 所規範的「處理或宣告原則」(請參考〈CH20：Checked 與 Unchecked 例外的語意與問題〉)，因此很容易寫出捕捉 checked exception 並將其忽略的程式，如下列程式片段 6~8 行所示，捕捉並忽略 `FileNotFoundException` 以便通過 Java 編譯器的語法檢查。

```
1: public void doIt(){
2:     FileInputStream fis = null;
3:     try {
4:         fis = new FileInputStream(new File("aFileName"));
5:     }
6:     catch (FileNotFoundException e) {
7:         // 忽略它
8:     }
9:     finally{
10:        close(fis);
11:    }
12: }
```

***

忽略例外（Ignored Exception）

Ignored Exception 是和 *Ignored Checked Exception* 形式相近，但產生原因不同並且和 Java 語言無關的壞味道。有些開發人員因為不知道如何處理例外，深怕自己所寫的程式一旦丟出例外，可能會被其他團隊成員質疑。或是有意、無意之間為了隱藏自己所寫的程式可能會產出的問題，因此寫出如下列程式片段的程式碼，將自己所寫的函數用一個 `try` statement 包起來，然後搭配一個 `catch(Throwable e)` 或 `catch(Exception e)` 捕捉並忽略所有例外。

```
1: public void noExceptionThrown(){
2:     try{
3:         // 可能會丟出例外的正常程式邏輯
4:     }
5:     catch(Throwable e){
6:         // 捕捉並忽略所有例外，這樣別人就不知道我幹了什麼蠢事了，YA
7:     }
8: }
```

還是那句老話：忽略例外等於隱藏潛在問題，會降低系統強健度並且增加除錯困難度。

***

未被保護的主程式（Unprotected Main Program）

未被捕捉的例外一直往上傳遞，最終傳到主程式或執行緒身上。如果主程式或執行緒沒有捕捉由下傳遞至自己身上的例外，則主程式或執行緒會不預期地終止執行。如下列程式片段所示，Teddy 稱其為 *Unprotected Main Program*，使用者會認為這種不預期的程式終止是軟體品質不佳的表現。

```
1: static public void main(String[] args) {
2:       MyApp myapp = new MyApp();
3:       myapp.start();
4: }
```

***

虛設的例外處理程序（Dummy Handler）

Dummy Handler 是一種常見但不良的例外處理方式，如下所示：

```
1: catch (FileNotFoundException e) {
2:     e.printStackTrace();  // 我是 Dummy Handler
3: } catch (IOException e) {
4:     System.out.println(e.getMessage()); // 我也是 Dummy Handler
5: }
```

這種方式看起來比捕捉然後直接忽略例外要來的好，至少第 2 行與第 4 行程式把例外輸出到主控台（console）。但在現今圖形化、網頁化、行動化、分散式的應用系統中，使用者或開發人員並不容易直接觀看到輸出至主控台的例外訊息。此外，這類型的例外處理程式並沒有考慮修復程式狀態的問題，可能導致程式狀態不正確而無法繼續運作下去。

***Dummy Handler* 造成一種例外已經被妥善處理的假象。**與 *Ignored Checked Exception* 和 *Ignored Exception* 類似，*Dummy Handler* 會降低系統強健度並且增加除錯困難度。

***

巢狀 Try 敘述（Nested Try Statement）

無限制地使用巢狀敘述，像是條件式（`if-then-else`）與迴圈（`for` 與 `while`），很容易產生複雜的程式結構，使得程式難以閱讀、測試、維護。因為例外處理而衍生的 *Nested Try Statement* 一樣會產生相同的問題。例如下面程式片段，第 8 行程式呼叫 `close` 函數釋放資源，該呼叫可能產生 `IOException`，所以需要在 `finally` block 裡面撰寫另外一個 `try` statement（第 6~12 行）來捕捉這個 `IOException`。

```
1: try {
2:     in = new FileInputStream("aFile");
3: } catch (IOException e){
4:       // 處理例外
5: } finally {
6:     try {    // 我是巢狀 Try 敘述
```

```
7:          if (in != null)
8:              in.close();
9:     }
10:    catch(IOException e) {
11:        // 將例外寫入日誌檔中
12:    }
13: }
```

***

當成備胎的例外處理程序（Spare Handler）

Spare Handler 就是把 `catch` block 當成 `try` block 的「備胎」，當 `try` block 裡面的實作發生例外，則執行 `catch` block 裡面的替代方案，其結構如下面程式片段所示。

```
1: try{
2:     // 主要實作
3: }
4: catch(Exception e){
5:     // 替代方案
6: }
```

這種例外處理方法相當於「採用替代方案重試」(retry with alternative)，該方法本身沒有問題，**但將替代方案寫在 `catch` block 裡面，會導致只能重試一次。如果想要多次重試，勢必需要在 `catch` block 裡面再放置多個 `try` statement，形成 *Nested Try Statement* 壞味道。**

此外，原本 `catch` block 應該負責其相對應 `try` block 的 error handling 與 fault handling 的工作，如果把 `catch` block 當成備胎使用，則其中的程式碼將混和 error handling、fault handling 與正常邏輯替代方案實作，這樣子將使得整個程式的結構更加難以理解與維護。

***

粗心的資源釋放（Careless Cleanup）

Careless Cleanup 表示資源沒有被正確的釋放，會導致資源耗盡並降低系統穩定度。在 Java 與

C#語言中，釋放資源的程式碼應該要寫在 `finally` block 裡面，但是大家還是經常發現如下程式片段中第 3 行的寫法，把釋放資源的程式碼放在 `try` block 的最後一行。

```
1: try{
2:     FileInputStream fis = new FileInputStream(new File(aFileName));
3:     fis.close();   // 難道關閉資源錯了嗎？
4: }
5: catch(IOException e){
6:     // 處理例外
7: }
```

或是放在 `catch` block 裡面，如下程式片段所示。

```
1: try{
2:     fis = new FileInputStream(new File(aFileName));
3: }
4: catch(IOException e){
5:     fis.close();    // 難道關閉資源錯了嗎？
6: }
```

或是如下面程式片段所示，在 `finally` block 中的第 9 行和第 10 行連續關閉兩個資源，但卻沒有考慮到第 9 行如果發生例外，則第 10 行的程式將不會被執行。

```
1: public void carelessCleanup(String aFileName) throws IOException {
2:     FileInputStream fis = null;
3:     DataInputStream dis = null;
4:     try{
5:         fis = new FileInputStream(new File(aFileName));
6:         dis = new DataInputStream(fis);
7:     }
8:     finally{
9:         fis.close();
10:        dis.close();    // 如果上一行發生例外，我會沒被執行到喔
11:    }
12: }
```

***

友藏內心獨白：趕快看一下自己的程式有沒有這些壞味道。

Column H. | 仙人打鼓有時錯：談談 Clean Code 的例外處理

收到一本博碩文化贈閱的中文版《Clean Code》（中文版書名：《無暇的程式碼》）。原本博碩文化的陳技術主編邀請 Teddy 幫忙寫推薦序，但這本書的英文版 Teddy 一直沒時間全部讀完，加上中文版出版在即，因此無緣幫本書寫推薦序。但陳主編還是很好心的送了一本中文版《Clean Code》給 Teddy，在這個 Column 介紹一下這書中關於例外處理的內容。

***

錯誤處理

拿起書快速翻了一下，看到第 7 章在講錯誤處理，這剛好是 Teddy 的「法定專長」，這一章介紹幾個錯誤處理技巧：

1. 使用例外取代回傳碼（return code）。
2. 在開頭寫下 `try-catch-finally` 敘述。
3. 使用非受檢例外（unchecked exception）。
4. 提供例外發生相關資訊。
5. 從呼叫者角度定義例外類別。
6. 定義正常程式流程。
7. 不要回傳 null。
8. 不要傳遞 null。

寫到這邊有點快寫不下去，因為本來是想要推薦《Clean Code》這本書，但結果可能變成在「吐槽」書中內容。《Clean Code》提到有關錯誤處理的這 8 點技巧，Teddy 對於第 2 點與第 3 點有點小意見。先談談第 3 點，《Clean Code》認為 Java 語言的 checked exception 是一種「失敗的實驗」，建議開發人員使用 unchecked exception 而不要使用 checked exception。

關於這個問題，在本書中已經討論過很多次了，眼尖的鄉民們可能看的出來 Teddy 比較偏好 checked exception，但不可否認的確有很多人不喜歡 checked exception，而且大部分的程式語言也都沒有支援它。所以 Teddy 認為這個問題應該要跳脫 Java 語言對於 checked 或是 unchecked exception 的語法層次，從其他較高層次的面向來討論。例如，〈CH20：Checked 與 Unchecked 例外的語意與問題〉、〈CH21：介面演進〉、「第四部 為什麼例外處理那麼難？例外處理的 4+1 觀點」、以及〈CH27：例外處理設計的第一步：決定強健度等級〉。

***

接下來討論第 2 點：在開頭寫下 `try-catch-finally` 敘述，這個建議就比較有趣了。首先，在函數開頭就寫下 `try-catch-finally` 有時候是辦不到的。請看以下範例，`try_catch_finally_V1` 函數的實作使用一個 `try-catch-finally` 敘述包起來。

```
1: public void try_catch_finally_V1(String aFileName) {
2:     try {
3:         FileInputStream fis = new FileInputStream(new File(aFileName));
4:     } catch(IOException e){
5:         throw new RuntimeException(e);
6:     } finally {
7:     }
8: }
```

上面程式看起來很好，沒問題。等一下，`FileInputStream` 用完之後不是應該要關閉嗎？《Clean Code》書中的例子採用類似以下程式碼第 4 行的作法來關閉串流：

```
1: public void try_catch_finally_V2(String aFileName) {
2:     try {
3:         FileInputStream fis = new FileInputStream(new File(aFileName));
4:         fis.close();
5:     } catch(IOException e) {
6:         throw new RuntimeException(e);
7:     } finally {
8:     }
9: }
```

第 4 行把釋放資源的 `fis.close()` 程式碼寫在 `try` block 裡面，這種寫法是 *Careless Cleanup* 例外處理壞味道（嚴格講起來是 bug）。為什麼？因為 cleanup 程式碼要寫在 `finally` 裡面，請參考下列程式片段：

```
1: public void try_catch_finally_V3(String aFileName) {
2:     try {
3:         FileInputStream fis = new FileInputStream(new File(aFileName));
4:     } catch(IOException e) {
5:         throw new RuntimeException(e);
6:     } finally {
7:         //fis.close(); 要寫在這裡
8:     }
9: }
```

上面這段程式如果把第 7 行的註解拿掉將無法通過編譯，因為 `fis` 被宣告在 `try` block 裡面，在 `finally` block 存取不到它。所以程式只好改成如下所示：

```
1: public void try_catch_finally_V4(String aFileName) {
2:     FileInputStream fis = null;
3:     try {
4:         fis = new FileInputStream(new File(aFileName));
5:     } catch(IOException e) {
6:         throw new RuntimeException(e);
7:     } finally {
8:         try {
9:             if (null != fis)
10:                fis.close();
11:        } catch (IOException e) {
12:            // 寫入日誌檔
13:        }
14:    }
15: }
```

也就是要把宣告 `fis` 的敘述放在 `try` statement 外面，如此一來也就違反了「在開頭寫下 `try-catch-finally` 敘述」的這條建議。除非鄉民們要把程式寫成下面這個樣子，但如此一來程式結構又會造成 *Nested Try Statement* 這個例外處理壞味道，感覺也不太好。

```
1: public void try_catch_finally_V5(String aFileName) {
2:     try {
3:         FileInputStream fis = null;
4:         try {
5:             fis = new FileInputStream(new File(aFileName));
6:         } catch(IOException e) {
7:             throw new RuntimeException(e);
8:         } finally {
9:             try {
10:                if (null != fis)
11:                    fis.close();
12:            } catch (IOException e) {
13:                // 寫入日誌檔
14:            }
15:        }
16:    }
17:    finally {
18:    }
19: }
```

***

請 Java SE 7 來幫忙

參考《Clean Code》書上的範例，如果要做到「在開頭寫下 `try-catch-finally` 敘述」，就必須使用 Java 7 的新功能。

```
1: public void try_catch_finally_V6(String aFileName) {
2:     try (FileInputStream fis = new FileInputStream(new File(aFileName))) {
3:
4:     } catch(IOException e){
```

```
5:         throw new RuntimeException(e);
6:     }
7: }
```

請參考上面第 2 行程式，把需要關閉的資源寫在 `try (...)` 裡面，這樣子當離開 `try` statement 的時候，所使用到的資源會自動被關閉，程式設計師也就不需要在 `finally` 中去手動關閉資源。

***

沒那麼嚴格

再仔細讀一下「在開頭寫下 try-catch-finally 敘述」這條規則的定義，《Clean Code》書中提到：「如果你的程式可能會拋出例外，請養成讓 `try-catch-finally` 成為開頭敘述的好習慣。」如果從寬解釋，`try_catch_finally_V4` 函數這個版本也符合這條規則，因為宣告 `fis` 這一行不會丟出例外，所以寫在 `try` statement 之外是可以接受的。

講了這個多有的沒的，Teddy 想提醒的重點是，可能是書中採用測試驅動開發（test-driven development；TDD）方式來舉例子，所以程式碼出現了類似下面程式片段第 3 行的壞味道（因為程式還沒有完成重構所以遺留了壞味道…這算是幫「權貴」開脫罪刑嗎…）。再強調一次，下面程式第 3 行 `fis.close()` 要放在 `finally` block 裡面，而非 `try` block 的結尾。

```
1:     try {
2:         FileInputStream fis = new FileInputStream(new File(aFileName));
3:         fis.close();
4:     }
```

***

友藏內心獨白：你這是在推薦還是在翻桌啊。

35 用例外代替錯誤碼[46]（Replace Error Code with Exception）

一個函數用回傳值來代表錯誤狀況。

拋出一個例外來代替。

```
1: public synchronized int withdraw(int amount) {
2:      if (amount > this.balance)
3:            return -1;
4:      else {
5:            this.balance = this.balance - amount;
6:             return this.balance;
7:    }
8: }
```

```
1: public synchronized int withdraw(int amount) throws NotEnoughMoneyException {
2:      if (amount > this.balance)
3:            throw new NotEnoughMoneyException();
4:
5:       this.balance = this.balance - amount;
6:       return this.balance;
7: }
```

動機

以回傳值來代表錯誤狀況，會讓軟體元件連「強健度等級 1：錯誤回報」的標準都達不到，因

[46] Matrin Fowler 在 *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 2012 這本書中介紹了 *Replace Error Code with Exception* 與 *Replace Exception with Test* 這兩個和例外處理有關的重構。只有前者與本書中介紹的例外處理壞味道有關，因此在本章中介紹。

為呼叫者通常傾向忽略回傳值的檢查，因此也忽略了錯誤狀況。

因此，採用例外來取代錯誤碼，可讓軟體元件達到強健度等級 1 的要求。

套用步驟

1. 決定使用哪種例外，判斷錯誤狀況屬於 design fault 或是 component fault。
   - 如果是 design fault，在 Java 語言中用 unchecked exception。
   - 如果是 component fault，在 Java 語言中使用 checked exception。
   - 如果使用非 Java 語言，例如 C#或 VB.NET，或是不想採用 Java 語言的 checked exception，則需要自己定義用來代表 design fault 或 component fault 的例外類別。例如，鄉民們可以定義 `RecoverableException` 類別用來代表 design fault，定義 `ProgrammingErrorException` 代表 design fault。
2. 修改所有呼叫舊函數的地方，加上處理例外的程式碼。
3. 修改舊函數介面，用來反應使用新的例外物件。
4. 編譯與測試。

範例

`Account` 類別有一個用來提款的 `withdraw` 函數：

```
1: public synchronized int withdraw(int amount) {
2:       if (amount > this.balance)
3:             return -1;
4:       else {
5:             this.balance = this.balance - amount;
6:              return this.balance;
7:     }
8: }
```

當提款金額 `amount` 大於存款金額 `balance`，則程式回傳-1 用來代表存款餘額不足（程式第 2~3 行）。現在你要用例外來取代回傳-1 的作法，首先要考慮的問題是，這個例外是 design fault 還是 component fault？

贊成是design fault的人認為，呼叫withdraw之前應該先檢查一下提款金額是否大於存款餘額，如果想要提領的錢比自己的存款還多，就不應該允許提領的動作。如下面程式片段所示，這樣存款餘額不足的例外狀況就不會發生在 withdraw 中。

```
1: if (account.getBalance() > amount)
2:     account.withdraw(amount);
```

換句話說，會發生存款餘額不足單純是「程式設計錯誤」的問題，應該丟出一個 unchecked exception，或是 ProgrammingErrorException 來代表 design fault。

但是仔細想一下，就算是在呼叫 withdraw 之前已經確定存款餘額是足夠的，但還是有可能因為其他人剛好也執行扣款的動作（例如銀行在此時扣了一筆自動繳款的水電費），而導致執行 account.withdraw(amount) 發生存款餘額不足的問題。所以這個例外應該不是程式設計的問題，而是一種 component fault。因此決定用一個 checked exception，或是繼承自 RecoverableException 的 NotEnoughMoneyException 來代表此異常狀況：

```
1: public class NotEnoughMoneyException extends Exception {
2:     public NotEnoughMoneyException(){
3:         super();
4:     }
5: }
```

接著修改呼叫 withdraw 函數的地方，使其變成類似下面程式片段的結構：

```
1: try{
2:     account.withdraw(amount);
3:     doNormalLogic();
4: }
5: catch(NotEnoughMoneyException e){
6:     handleOverdrawn();
7: }
```

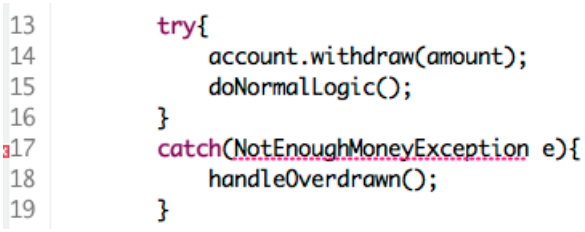
然後修改 withdraw 函數：

```
1: public synchronized int withdraw(int amount) throws NotEnoughMoneyException {
2:       if (amount > this.balance)
3:             throw new NotEnoughMoneyException();
4:
5:        this.balance = this.balance - amount;
6:        return this.balance;
7: }
```

最後，編譯並測試。

***

上述修改流程在步驟二直接修改所有原本呼叫 withdraw 的程式碼，如果程式中有很多地方都呼叫 withdraw，但是要等到步驟四才執行編譯、測試，因此一次修改這麼多地方比較容易在修改過程中發生錯誤。尤其當 NotEnoughMoneyException 是採用 checked exception，此時 withdraw 還沒有宣告丟出 NotEnoughMoneyException（這個動作是步驟三才會執行），此時步驟二所修改的程式碼是無法通過 Java 編譯器的檢查，如下圖所示：



Fowler 在《Refactoring》一書中建議了另一個「殺傷力」較小的方法。首先新增一個使用 NotEnoughMoneyException 的 newWithdraw：

```
1: public synchronized int newWithdraw(int amount) throws NotEnoughMoneyException {
2:       if (amount > this.balance)
3:             throw new NotEnoughMoneyException();
4:
5:        this.balance = this.balance - amount;
6:        return this.balance;
7: }
```

然後修改原本的 withdraw，讓它呼叫 newWithdraw，如下所示：

```
1: public synchronized int withdraw(int amount) {
2:     try{
3:         return newWithdraw(amount);
4:     }
5:     catch(NotEnoughMoneyException e){
6:         return -1;
7:     }
8: }
```

此時便可拿 withdraw 當做 newWithdraw 的使用者，來針對 newWithdraw 做測試。也就是說，如果原本呼叫 withdraw 的程式都還是正常的，則因為 withdraw 的實作已經改成透過呼叫 newWithdraw 完成，那麼 newWithdraw 的實作也是正確的。

接下來便可逐一修改原本呼叫 withdraw 的地方，替換成呼叫 newWithdraw：

```
1: try{
2:     account.newWithdraw(amount);
3:     doNormalLogic();
4: }
5: catch(NotEnoughMoneyException e){
6:     handleOverdrawn();
7: }
```

因為 withdraw 和 newWithdraw 同時都存在，因此每次只要改完一個地方就可以立刻編譯與測試一次程式。等全部使用 withdraw 的地方都換成了呼叫 newWithdraw，就可以把 withdraw 刪除。最後使用 *Rename Method* 這個重構技巧，把 newWithdraw 改名為 withdraw 就大功告成了。

***

友藏內心獨白：重構就是每次以極小的修改步驟來改善設計。

36 以非受檢例外取代忽略受檢例外（Replace Ignored Checked Exception with Unchecked Exception）

你捕捉了 checked exception 但卻什麼事都沒做。

將所捕捉的 checked exception 轉成事先定義好的 unchecked exception — `UnhandledException`，然後再將其丟出。

```
1:    catch (IOException e) {
2:        /* ignoring the exception */
3:    }
```

```
1:    catch (IOException e) {
2:        throw new UnhandledException(e);
3:    }
```

動機

當你呼叫一個會丟出 checked exception 的函數，你就被 Java 編譯器盯上，它會檢查你是否捕捉這個 checked exception，或是把它宣告在函數介面上。採取後者會改變函數介面，產生介面演進問題（請參考〈CH21：介面演進〉），讓呼叫你的人也受到這個 checked exception 影響而沒有好日子可過。兩害相權取其輕，大部分的人都選擇前者，捕捉這個 checked exception。

但是，捕捉例外之後卻不知道要如何處理，這下子糗大了。你目前只想專心於撰寫正常的程式邏輯，沒有多餘時間去處理異常情況。或是目前你並沒有足夠的 context 來判斷要如何處理例外，

必須等到整個功能都串起來之後，例外處理的策略才會比較清楚。怎麼辦？最簡單的方法就是忽略它：

```
catch (IOException e) {  /* TODO */ }
```

雖然寫了 TODO 註解來提醒自己要「抽空」回來再續這段「孽緣」，但是註解本身「僅供參考」的成分較大，沒有強制約束力，就算程式執行過程中真的引發了例外，也沒有人會發現。也就是說忽略 checked exception 會讓軟體元件達不到「強健度等級 1：錯誤回報」，系統的強健度因此受到損害。

因此，與其捕捉 checked exception 並且忽略它，不如在捕捉後將它轉成 `UnhandledException` 丟出。`UnhandledException` 是一個 unchecked exception，所以不需要宣告在函數介面上也可以直接向外傳遞。這時候你就可以專心處理正常邏輯，而暫時不被 checked exception 干擾。由於原本發生例外的原因保存在 `UnhandledException` 身上，你只需要套用 *Avoid Unexpected Termination with Big Outer Try Block*（253）便可在最上層程式中印出錯誤訊息，作為除錯之用。

套用步驟

1. 定義一個 unchecked exception — `UnhandledException` 類別，用來代表有一個 checked exception 尚未被處理。
2. 丟出 `UnhandledException` 物件來取代被忽略的 checked exception。
   - 產生一個 `UnhandledException` 實例（instance）。
   - 把捕捉到的 checked exception 當成建構函數的參數傳給 `UnhandledException`。
   - 如果要特別說明忽略例外的原因或其他注意事項，可再傳入一個字串給 `UnhandledException` 的建構函數。
   - 丟出 `UnhandledException`。
   - 刪除原有的註解（如果有的話）。
3. 編譯與測試。

範例

`FileUtility` 類別有一個將字串寫到檔案的 `writeFile` 函數：

```
1: public void writeFile(String fileName, String data) {
2:     try (Writer writer = new FileWriter(fileName)) {
3:         writer.write(data);  // 可能會丟出 IOException
4:     }
5:     catch (IOException e) {
6:         // 忽略例外
7:     }
8: }
```

第 3 行程式會丟出 IOException，你現在要將被捕捉且忽略的 IOException（第 5~7 行）改以丟出 unchecked exception 來取代。

首先，新增一個 UnhandledException，讓它繼承自 RuntimeException：

```
1: public class UnhandledException extends RuntimeException {
2:     public UnhandledException() {
3:         super();
4:     }
5:     public UnhandledException(Throwable e) {
6:         super(e);
7:     }
8:     public UnhandledException(String msg) {
9:         super(msg);
10:    }
11:    public UnhandledException(String msg, Throwable e){
12:        super(msg, e);
13:    }
14: }
```

接著修改 writeFile 函數，使其丟出 UnhandledException：

```
1: public void writeFile(String fileName, String data) {
2:     try (Writer writer = new FileWriter(fileName)) {
3:         writer.write(data);  // 可能會丟出 IOException
4:     }
```

```
5:     catch (IOException e) {
6:         throw new UnhandledException(e);
7:     }
8: }
```

最後，編譯並測試。

***

友藏內心獨白：註解不會制裁粗心大意的程式設計師啊。

37 以非受檢例外取代虛設的例外處理程序（Replace Dummy Handler with Rethrow）

你採用 *Dummy Handler* 來處理例外。

移除 *Dummy Handler*。將捕捉到的例外轉成事先定義好的 unchecked exception — `UnhandledException`，然後再將其丟出。

```
1: catch (IOException e) {
2:     e.printStackTrace();
3: }
```

```
1:     catch (IOException e) {
2:         throw new UnhandledException(e);
3:     }
```

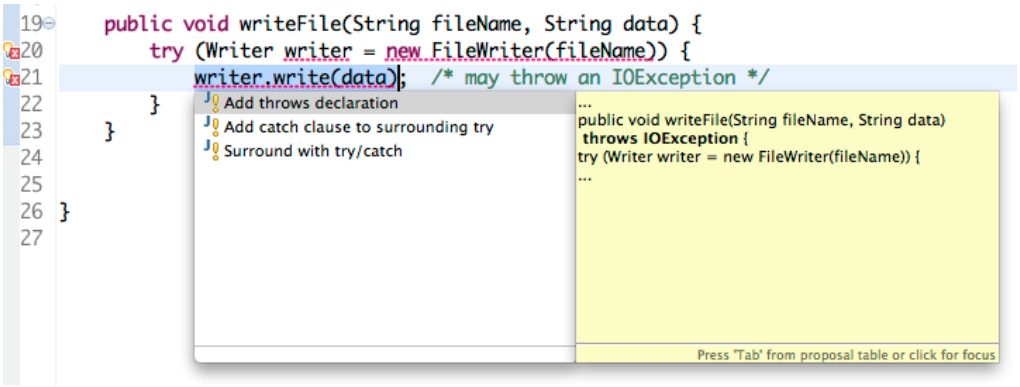
動機

和 *Ignored Checked Exception* 壞味道很像，*Dummy Handler* 經常被用來當做逃避 Java 編譯器對於 checked exception 的「處理或宣告原則」的檢查。雖然捕捉並直接忽略 checked exception 可以滿足 Java 編譯器，但這麼明顯的作弊方法實在是犯了例外處理的兵家大忌，很容易被同事或主管拆穿，降低自己在他們面前「無瑕」的崇高地位。你心裡想著，多多少少還是做點事，那就把例外當做錯誤訊息給印出來吧：

```
e.printStackTrace();
```

有些開發工具甚至助長 *Dummy Handler* 的產生速度，Eclipse 就是其中的代表作。Eclipse 提供「快速修正」（Quick Fix）功能，可以幫助開發人員排除一些語法錯誤的問題。如下圖所示， 針

對 `write.write(data)` 這一行程式會丟出 `IOException` 但卻沒有被處理所造成的語法錯誤，Eclipse 提供三個建議。很多開發人員會選擇第二點「Add catch clause to surrounding try」（將 `catch` 子句加到鄰近的 `try`）來捕捉例外。



一個「完美」的 *Dummy Handler* 就此產生：

```
19    public void writeFile(String fileName, String data) {
20        try (Writer writer = new FileWriter(fileName)) {
21            writer.write(data);  /* may throw an IOException */
22        } catch (IOException e) {
23            // TODO Auto-generated catch block
24            e.printStackTrace();
25        }
26    }
```

「Eclipse 好棒啊，按兩次滑鼠按鈕就幫我把例外處理好了。」抱持這種想法的開發人員，對於例外處理的能力，猶如一張未被汙染的白紙，殊不知案情並沒有這麼單純啊。

在 C#這種只支援 unchecked exception 的語言，也可以見到 *Dummy Handler* 的蹤影，它經常被拿來作為一種避免 *Ignored Exception* 的手段。在這類的語言中，例外不須捕捉便可直接往外傳遞，但很多時候開發人員因為怕自己所寫的程式丟出例外，導致系統終止而被長官或同事責罵，因此突發奇想將所有程式中的例外全部捕捉下來。同樣的心態又出現了，例外不應該被忽略，但捕捉之後又不知道要做什麼，所以就把它印出來吧。

這種作法，同樣也是太天真了。

Dummy Handler 雖然將例外列印出來，並沒有直接忽略例外，但程式真的跑起來之後，這些輸出至主控台的例外是沒有人會去注意它的。也就是說 *Dummy Handler* 和 *Return Code* 與 *Ignored Checked Exception* 一樣，會讓軟體元件達不到強健度等級 1 的最基本要求。

因此，與其盲目不加思索地將例外印出造成 *Dummy Handler*，請將捕捉到的例外轉成 `UnhandledException` 丟出。身為一個 unchecked exception，`UnhandledException` 不需要被宣告在函數介面，也不會影響 call chain 上的其他人。最後，只需要套用 *Avoid Unexpected Termination with Big Outer Try Block（253）*便可在最上層程式中印出錯誤訊息，作為除錯之用。

套用步驟

1. 移除造成 *Dummy Handle*r 的程式碼。
2. 套用 *Replace Ignored Checked Exception with Unchecked Exception（245）*。
3. 編譯與測試。

範例

`FileUtility` 類別有一個將字串寫到檔案的 `writeFile` 函數：

```
1: public void writeFile(String fileName, String data) {
2:     try (Writer writer = new FileWriter(fileName)) {
3:         writer.write(data);  /* may throw an IOException */
4:     } catch (IOException e) {
5:         // TODO Auto-generated catch block
6:         e.printStackTrace();
7:     }
8: }
```

第 3 行程式會丟出 `IOException`，你現在要將 *Dummy Handler*（第 5~6 行）改以丟出 unchecked exception 的方式來取代。

首先，新增一個 `UnhandledException`，讓它繼承自 `RuntimeException`：

```
1: public class UnhandledException extends RuntimeException {
```

```
    public UnhandledException() {
        super();
    }
    public UnhandledException(Throwable e) {
        super(e);
    }
    public UnhandledException(String msg) {
        super(msg);
    }
    public UnhandledException(String msg, Throwable e){
        super(msg, e);
    }
}
```

接著修改使用 `writeFile` 的程式碼，讓它丟出 `UnhandledException`：

```
public void writeFile(String fileName, String data) {
    try (Writer writer = new FileWriter(fileName)) {
        writer.write(data);  /* may throw an IOException */
    }
    catch (IOException e) {
        throw new UnhandledException(e);
    }
}
```

最後，編譯並測試。

***

友藏內心獨白：不想處理就往外丟。

38 使用最外層 Try 敘述避免意外終止（Avoid Unexpected Termination with Big Outer Try Statement）

未被捕捉的例外最終傳遞到最外層的元件，造成應用程式意外終止。

將最外層元件以一個 `try` statement 包住，捕捉所有例外，然後將其顯示或/且記錄到日誌檔中。

```
1: static public void main(String [] args){
2:     /*
3:      * 做一大堆事情的主程式
4:      */
5: }
```

```
1: static public void main(String [] args){
2:     try{
3:         /*
4:          * 做一大堆事情的主程式
5:          */
6:     }
7:     catch (Throwable e){
8:         /*
9:          *  顯示錯誤訊息並且記錄到日誌檔中
10:         */
11:    }
12: }
```

動機

看完 *Replace Ignored Checked Exception with Unchecked Exception（245）*和 *Replace Dummy Handler with Rethrow（249）*，你可能會有一個疑問：「大家都拚命把例外往外丟，最後例外要交給誰去處理啊？」

如果拋出的例外都沒人處理，所有未被捕捉的例外最終都會傳遞到主程式（或執行緒）身上。要是主程式也沒有捕捉這些例外，整個應用程式就會被迫終止，這就是大家非常熟悉的老朋友—程式當掉。輕則造成使用者觀感不佳，重則造成生命或財產損失。

在程式異常終止的情況下，雖然作業系統會顯示錯誤訊息，但是這些訊息的說明通常很模糊。就算使用者回報這些訊息，對於事後的除錯幫助也有限。

因此，為了避免應用程式不預期終止，將最外層程式碼以一個 `try` statement 包住，捕捉所有類別的例外，在畫面上顯示清楚且容易理解的錯誤訊息，並視需要記錄詳細除錯資訊到日誌檔中，最後結束應用程式的執行，讓它「死相好看一點」。

在單一執行緒（single-threaded）應用程式中，很容易找到「主程式」並在其中套用此重構方法。在多執行緒（multi-threaded）應用程式中，因為個別執行緒所產生的例外，可能不會傳遞到主執行緒身上，因此必須將保護的對象擴及到每一個執行緒，以防止其不預期終止。

套用步驟

1. 找到應用程式的主程式。針對多執行緒應用程式，找出每一個執行緒的進入點。
2. 把主程式與執行緒進入點新增一個 `try-catch` statement，在 `catch` block 捕捉所有例外。
3. 把原本的程式碼移入 `try` block。
4. 在 `catch` block 裡面，需要記錄詳細除錯資訊到日誌檔中，然後顯示錯誤訊息。
5. 結束應用程式的執行。
6. 編譯與測試。

範例

你有一個未受保護的 `main` 函數：

```
1: static public void main(String [] args){
2:     /*
3:      * 做一大堆事情的主程式
4:      */
5: }
```

現在用一個 *Big Outer Try Statement* 來保護它：

```
1: static public void main(String [] args){
2:     try{
3:     }
4:     catch (Throwable e){
5:
6:     }
7:
8:     /*
9:      * 做一大堆事情的主程式
10:     */
11: }
```

接下來把原本主程式或執行緒中的程式碼移到 `try` block 裡面：

```
1: static public void main(String [] args){
2:     try{
3:         /*
4:          * 做一大堆事情的主程式
5:          */
6:     }
7:     catch (Throwable e){
8:
9:     }
10: }
```

在 `catch` block 裡面將錯誤訊息寫入日誌檔，並顯示錯誤訊息：

```
1: static public void main(String [] args){
```

```
 2:     try{
 3:         /*
 4:          * 做一大堆事情的主程式
 5:          */
 6:     }
 7:     catch (Throwable e){
 8:         logger.log(e);
 9:         dialog.show(Dialog.CRITICAL, e.getMessage());
10:     }
11: }
```

上述程式片段第 9 行，在顯示完錯誤訊息之後，程式就會結束執行。

最後，編譯並測試。

***

友藏內心獨白：寫日誌檔與顯示錯誤訊息可以做成公用程式。

39 以函數取代巢狀 Try 敘述（Replace Nested Try Statement with Method）

程式中存在著巢狀 `try` statement。

將巢狀 `try` statement 抽離到一個新的函數，讓該函數的名稱反映出它的目的。

```
1: finally {
2:     try {
3:         if (in != null) in.close();
4:     } catch(IOException e) {
5:         // log the exception
6:    }
7: }
```

```
1: finally {
2:     cleanup(in);
3: }
```

```
1: private void cleanup(Closeable cls){
2:     try {
3:         if (cls != null)  cls.close();
4:     } catch(IOException e) {
5:         // log the exception
6:    }
7: }
```

動機

在 Java 與 C#這類程式語言中，你可以在 `try` block、`catch` block、`finally` block 裡面加入另一個 `try` statement，形成巢狀的 `try` statement。這樣的巢狀結構深度不受限制，直到你寫出再也沒有人看得懂的程式為止。

有兩個常見的理由會產生巢狀 `try` statement：

1. 在 `catch` block 中提供替代方案：請看一段 JFreeChart[47] 1.0.14 版的程式碼片段，11~21 行的 `catch` block 為 5~10 行的 `try` block 提供了例外發生之後的替代方案。不幸的是，`catch` block 替代方案的程式碼，也會產生例外，所以只好在 `catch` block 裡面再用另一個 `try` statement（第 12~20 行）將替代方案程式碼包起來，因此產生巢狀 `try` statement。

```
 1: private RegularTimePeriod createInstance(Class periodClass,
 2:           Date millisecond, TimeZone zone, Locale locale) {
 3:
 4:     RegularTimePeriod result = null;
 5:     try {
 6:       Constructor c = periodClass.getDeclaredConstructor
 7:         (new Class[] {Date.class, TimeZone.class, Locale.class});
 8:       result = (RegularTimePeriod) c.newInstance
 9:                    (new Object[] {millisecond, zone, locale});
10:     }
11:     catch (Exception e) {
12:        try {
13:          Constructor c = periodClass.getDeclaredConstructor
14:                                      (new Class[] {Date.class});
15:          result = (RegularTimePeriod) c.newInstance
16:                                       (new Object[] {millisecond});
17:        }
18:        catch (Exception e2) {
19:              // do nothing
20:        }
21:     }
22:
23:     return result;
```

[47] JFreeChart 是一個以 Java 語言開發的開源軟體，可用來繪製許多複雜的圖表。官方網址為：http://www.jfree.org/jfreechart/

```
24: }
```

2. 在 `finally` block 關閉資源：很多關閉資源的函數都會丟出例外，用來代表釋放資源失敗。例如 Java 語言的 I/O 串流物件與連接資料庫的 `Connection`、`Statement` 物件等。因此，在 `finally` block 裡面很容易發現如下的巢狀 `try` statement 程式。

```
1: finally {
2:     try {
3:         if (in != null) in.close();
4:     } catch(IOException e) {
5:         // 將例外寫入日誌檔中
6:    }
7: }
```

不管是哪一種情況，都會讓程式結構變得複雜，而且很容易產生 *Long Method(過長函數)*[48]這個壞味道。

因此，套用 *Extract Method（提煉函數）*[49]來解決這個問題。新增一個函數，把它的名字取為原本巢狀 `try` statement 程式片段所代表的意圖，然後把巢狀 `try` statement 程式片段抽離到該函數身上，最後只要在程式中呼叫新增的函數取代原本的巢狀 `try` statement 即可。

套用步驟

1. 套用 *Extract Method* 來抽離每一個巢狀 `try` statement。
   - 建立一個新的函數，以它「做什麼（what）」，而非以它「怎麼做（how）」來命名。
   - 將巢狀 `try` statement 程式碼複製到新函數裡面。
   - 如果複製出來的程式碼使用原始函數裡面的區域變數，將這些區域變數當成參數傳入新函數。
   - 編譯，確定原本巢狀 `try` statement 的程式都已經成功抽離到新函數。
   - 刪除巢狀 `try` statement 程式碼，改呼叫新函數。
2. 編譯與測試。

[48] *Long Method* 是《Refactoring: Improving the Design of Existing Code》書中介紹的壞味道。
[49] *Extract Method* 是《Refactoring: Improving the Design of Existing Code》書中介紹的重構方法。

範例

FileUtility 類別有一個讀取設定檔的 readConfig 函數：

```
 1: public void readConfig(String fileName){
 2:     FileInputStream in = null;
 3:     try {
 4:         in = new FileInputStream(fileName);
 5:         // 正常邏輯
 6:     } catch (IOException e){
 7:         // 例外處理邏輯
 8:     }
 9:     finally {
10:         try {
11:             if (in != null) in.close();
12:         } catch(IOException e) {
13:             // 將例外寫入日誌檔中
14:        }
15:     }
16: }
```

第 9~15 行的 finally block 裡面出現了巢狀 try statement。你現在要想辦法移除它。

首先，新增一個 cleanup 函數，把巢狀 try statement 程式碼複製到裡面：

```
1: private void cleanup(){
2:     try {
3:         if (in != null) in.close();
4:     } catch(IOException e) {
5:         // 將例外寫入日誌檔中
6:    }
7: }
```

原本的巢狀 try statement 程式片段使用到型別為 FileInputStream 的 in 這個區域變數，因此需要將 in 當成參數傳入 cleanup：

```
1: private void cleanup(FileInputStream in){
2:     try {
3:         if (in != null) in.close();
4:     } catch(IOException e) {
5:         // 將例外寫入日誌檔中
6:    }
7: }
```

編譯程式，發現抽離到 cleanup 函數的程式語法正確。現在可以在 readConfig 函數呼叫 cleanup 函數來取代原本的巢狀 try statement：

```
1: public void readConfig(String fileName){
2:     FileInputStream in = null;
3:     try {
4:         in = new FileInputStream(fileName);
5:         // 正常邏輯
6:     }
7:      catch (IOException e){
8:         // 例外處理邏輯
9:      }
10:     finally {
11:         cleanup(in);
12:     }
13: }
```

最後，編譯並測試。

***

補充說明一點，Java SE 5 之後 I/O 類別實作了 Closeable 介面，因此可以將 cleanup(FileInputStream in)改成 cleanup(Closeable in)，如此便可進一步套用 *Move Method（搬移函數）*[50]將 cleanup 函數移到其他合適的工具類別（utility）裡面，以便日後可重複使用。

[50] *Move Method* 是《Refactoring: Improving the Design of Existing Code》書中介紹的重構方法。

***

友藏內心獨白：如果一個函數只有一個 `try` statement，那就太好了。

40 引入檢查點類別（Introduce Checkpoint Class）

你的 `try` block 改變了應用程式的狀態。萬一 `try` block 發生例外，你想要確保應用程式依然保持在正確的狀態。

新增一個 checkpoint 類別來管理應用程式狀態，該類別具有保存狀態、回復狀態、丟棄狀態等功能。

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     try{
3:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
4:     }
5:     finally{
6:         // 釋放資源
7:     }
8: }
```

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     FolderCheckpoint fscp = null;
3:     try{
4:         fscp = new FolderCheckpoint();
5:         fscp.establish(srcFolder);
6:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
7:     } catch(Exception e){
8:         fscp.restore();
9:         throw e;
10:    } finally{
11:        fscp.drop();
12:        // 釋放資源
13:    }
14: }
```

動機

在程式執行的過程中，系統狀態可能會被改變。如果應用程式想要達到「強健度等級 2」，也就是當應用程式在例外發生之後還可以正常地持續運作下去，就必須想辦法讓系統在例外發生之後仍然處於正確狀態。使用 checkpoint（檢查點）是一個達到狀態回復的有效方法。checkpoint 就是「快照（snapshot）」的觀念，在資料備份軟體裡面經常使用到的一種方法。為了怕系統損毀造成資料遺失，可定期或依據需要（例如對系統做出重大升級之前）幫系統建一份快照，日後萬一真的發生錯誤可以使用這份快照將系統回復到之前正確的狀態。

Checkpoint有三個基本函數，分別是用來產生checkpoint的`establish`，回復資料的`restore`，以及捨棄 checkpoint 的 `drop`。對應到程式語言的 `try-catch-finally` 例外處理結構，在 `try` block 尚未修改系統狀態之前要先執行的 `establish`，如果例外發生，則可在 `catch` block 裡面以 `restore` 來回復狀態。最後，不管是否有例外發生，在 `finally` block 裡面呼叫 `drop` 清除 checkpoint。

使用類別來封裝資料與函數是物件導向設計的基本原理，因此將保存資料、回復資料、丟棄保存資料等函數封裝在 checkpoint 類別裡面，就跟一位汽車維修員隨身攜帶榔頭、板手、「羅賴把」一樣，都是很合理的做法。不同的功能需要不同類別的 checkpoint 來執行狀態回復，引入 checkpoint 類別將可簡化例外處理程式，提高重複使用例外處理程式的程度、並且減少重複程式碼的發生。

套用步驟

1. 產生 checkpoint 類別。
   - 新增 `establish`、`restore`、`drop` 三個函數。
   - 依據所要保存的狀態，分別實作 `establish`、`restore`、`drop`。
   - 編譯與測試。
2. 使用 checkpoint 類別。
   - 在 `try` block 外面宣告並產生 checkpoint 類別的實例。
   - 在 `try` block 中，尚未改變所要保存的狀態之前，呼叫 checkpoint 的 `establish` 函數來保存資料。
   - 在 `catch` block 中，執行 checkpoint 的 `restore` 函數來回復資料。

- 在 finally block 中，執行 checkpoint 的 drop 函數來清理資料。

3. 編譯與測試。

範例

FileUtility 類別有一個移動檔案的 moveFiles 函數：

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     try{
3:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
4:     }
5:     finally{
6:         // 釋放資源
7:     }
8: }
```

moveFiles 目前只達到「強健度等級 1：錯誤回報」，你準備引入 checkpoint，提升 moveFiles 的強健度。首先，新增一個包含 establish、restore、drop 三個函數的 FolderCheckpoint 類別：

```
1: public class FolderCheckpoint {
2:     public void establish(String backupFolder) {
3:     }
4:     public void restore() {
5:     }
6:     public void drop() {
7:     }
8: }
```

接著逐一實作與測試每一個函數：

```
1: public class FolderCheckpoint {
2:     public void establish(String backupFolder) {
3:         // 備份 backupFolder 所包含的全部檔案，包含子目錄的檔案
4:     }
5:     public void restore() {
```

```
6:         // 將備份的資料復原
7:     }
8:     public void drop() {
9:         // 刪除所有備份資料
10:    }
11: }
```

然後在 moveFiles 宣告並產生 FolderCheckpoint 類別實例：

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     FolderCheckpoint fscp = new FolderCheckpoint();
3:     try{
4:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
5:     } finally{
6:         // 釋放資源
7:     }
8: }
```

在 try block 裡面呼叫 establish 函數：

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     FolderCheckpoint fscp = new FolderCheckpoint(); ;
3:     try{
4:         fscp.establish(srcFolder);
5:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
6:     } finally{
7:         // 釋放資源
8:     }
9: }
```

新增一個 blanket catch，在其中呼叫 restore 函數，然後再將捕捉到的例外往外丟：

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     FolderCheckpoint fscp = new FolderCheckpoint(); ;
3:     try{
4:         fscp.establish(srcFolder);
```

```
5:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
6:     } catch(Exception e){
7:         fscp.restore();
8:         throw e;
9:     } finally{
10:        // 釋放資源
11:    }
12: }
```

在 `finally` block 裡面呼叫 `drop` 函數：

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     FolderCheckpoint fscp = new FolderCheckpoint();
3:     try{
4:         fscp.establish(srcFolder);
5:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
6:     } catch(Exception e){
7:         fscp.restore();
8:         throw e;
9:     } finally{
10:        fscp.drop();
11:        // 釋放資源
12:    }
13: }
```

最後，編譯並測試。

***

看到這邊你可能會想：「如果使用 Java SE 7 的 try-with-resources，是否可以稍微簡化上述程式結構？」以 Java SE 7 為例，要先讓 `FolderCheckpoint` 類別實作 `Closeable` 介面：

```
1: public class FolderCheckpoint implements Closeable
```

接著實作 `Closeable` 所定義的 `close`，直接呼叫原本的 `drop`：

```
1: @Override
```

```
2: public void close() throws IOException {
3:     drop();
4: }
```

最後修改 moveFiles 函數，改用 try-with-resources，現在可以拿掉 finally block：

```
1: public void moveFiles(String srcFolder, String destFolder) throws IOException {
2:     try (FolderCheckpoint cp = new FolderCheckpoint()) {
3:         cp.establish(srcFolder);
4:         // 複製 srcFolder 所有檔案到 destFolder, 可能會發生 IOException
5:     } catch(Exception e){
6:         cp.restore();
7:         throw e;
8:     }
9: }
```

上面程式片段看起來簡短許多，但卻行不通。如下圖所示，因為採用 try-with-resources 所宣告的變數 cp，只有在 try block 存取的到，在 catch block 中便無法存取，當然也就不能夠呼叫 cp.restore 函數。

```
9    public void moveFiles(String srcFolder, String destFolder) throws IOException {
10       try (FolderCheckpoint cp = new FolderCheckpoint()) {
11           cp.establish(srcFolder);
12           // 從把 srcFolder 所有檔案複製到 destFolder, 可能會發生 IOException
13       } catch(Exception e){
14           cp.restore();
15           throw e;
16       }
17   }
```

***

友藏內心獨白：朋友還是老的好，還是乖乖用回老方法。

41 引入多才多藝的 Try 區塊（Introduce Resourceful Try Block）

你把 `catch` block 當做 `try` block 的備胎使用，在其中所提供的替代方案實作可能會失敗，進一步丟出例外。

把 `catch` block 裡面的替代方案移到 `try` block，修改程式結構，使其在例外發生之後可以重新執行 `try` block。

```
1: public User readUser(String name) throws ReadUserException{
2:     try{
3:         return readFromDatabase(name);
4:     } catch(Exception e){
5:         try{
6:             return readFromLDAP(name);
7:         } catch(IOException ex){
8:             throw new ReadUserException(ex);
9:         }
10:    }
11: }
```

```
1: public User readUser(String name) throws ReadUserException{
2:     final int maxAttempt = 3;
3:     int attempt = 1;
4:     while(true){
5:         try{
6:             if(attempt <= 2) return readFromDatabase(name);
7:             else             return readFromLDAP(name);
8:         } catch(Exception e){
9:             if (++attempt > maxAttempt)
10:                throw new ReadUserException(e);
```

```
11:         }
12:     }
13: }
```

動機

重試是一種很常見，也很有用的例外處理策略，尤其是針對系統負載過重，或是服務暫時瞬間失效的異常情況特別有用。例如，當瀏覽網頁的時候發生問題，大部分的人第一個反應一定是按下瀏覽器的「重新載入」按鈕，這是一個典型手動重試的例子。透過程式自動重試，可以將應用程式強健度提升到等級三的水準，也就是當發生例外之後，應用程式依舊可以提供正常服務，滿足使用者需求。有些程式語言，像是 Eiffel 與 Ruby，直接支援重試，因此在這種語言的例外處理程式中，採用重試策略便是一種非常普遍的方式。但像是 Java 與 C#這類的語言，因為採用 termination model，並不直接支援重試。在 Java 與 C#如果要模擬重試，需要採用 while 或是 do-while 結構，而且程式寫起來比較複雜，因此使用這種語言的開發人員也就比較少在例外處理程式中套用重試策略。總歸一句，人性本懶啊。

以下節錄一段 Jenkins[51]的 `Engine.java` 程式碼，可以看到採用 `while` 迴圈來模擬重試的方法。

```
1: while(true) {
2:     try {
3:         Socket s = new Socket(host, Integer.parseInt(port));
4:         s.setTcpNoDelay(true); // we'll do buffering by ourselves
5:         s.setSoTimeout(30*60*1000); // 30 mins. See PingThread for the ping interval
6:         return s;
7:     } catch (IOException e) {
8:         if(retry++>10)
9:             throw (IOException)new
10:                 IOException("Failed to connect to "+host+':'+port).initCause(e);
11:         Thread.sleep(1000*10);
12:         listener.status(msg+" (retrying:"+retry+")",e);
13:     }
14: }
```

[51] Jenkins 是一個以 Java 開發的開源持續整合系統，官方網址在此：http://jenkins-ci.org/。

並不是說 Java 與 C#開發人員都不使用重試，問題是該怎麼用比較好？最常見的用法並不是如上面程式片段所示的採用 `while` 迴圈，而是直接把程式寫成如下所示的 *Spare Handler*，將 `catch` block 當成 `try` block 的「備胎」，等同於在 `catch` block 中執行重試：

```
1: try{
2:     // primary implementation
3: }
4: catch(Exception e){
5:     // alternative implementation
6: }
```

這種做法的主要缺點在於**「只能重試一次」**，如果要多次重試，勢必會寫出具備 *Nested Try Statement* 的程式，這又是另外一個例外處理壞味道。

因此，重新思考 `try` block 與 `catch` block 所需擔負的責任（請參考〈CH18：Try、Catch、Finally 的責任分擔〉），讓 `try` block 負責實作程式正常邏輯（包含主要方案與替代方案），將「備胎（替代方案）」從 `catch` block 移到 `try` block，讓 `catch` block 負責控制重試終止條件與錯誤回報。在每次重試之前可套用 *Introduce Checkpoint Class（263）* 以確保程式處在正確狀態。

套用步驟

1. 決定最多重試次數。
   - 在函數內部定義區域變數，`final int maxAttempt = 3;`。
   - `maxAttempt=3` 表示 `try` block 最多會被執行 3 次。
2. 宣告一個型別為 `int` 的 `attempt` 區域變數，用來記錄重試次數，並將其初始值設為 1。
3. 用一個 `while(true)` 迴圈將 `try` statement 包起來。
4. 產生 resourceful try block：將 `catch` block 裡面的「備胎程式」移到 `try` block，然後用 `if` 條件式依據重試次數判斷執行主要方案與替代方案的時間點。
   - 替代方案可以有多個，選擇何種替代方案的程式邏輯可以依據實作需要以多個 `if-else` 條件式來完成。
5. 刪除 `catch` block 裡面的「備胎程式」。
6. 在 `catch` block 裡面將 `attempt` 變數加 1，用來代表執行過一次 `try` block。

7. 加入判斷 attempt 是否大於 maxAttempt 的條件判斷，當條件成立丟出例外。
8. 編譯與測試。

範例

ACLManager 類別有一個讀取使用者物件的 readUser 函數，在預設情況下它透過 readFromDatabase 函數到資料庫中讀取使用者資料。如果發生例外，則採取替代方案，改呼叫 readFromLDAP 函數從 LDAP 伺服器讀取使用者資料。

```
 1: public User readUser(String name) throws ReadUserException{
 2:     try{
 3:         return readFromDatabase(name); // 可能丟出 SQLException
 4:     } catch(Exception e){
 5:         try{
 6:             return readFromLDAP(name); // 可能丟出 IOException
 7:         } catch(IOException ex){
 8:             throw new ReadUserException(ex);
 9:         }
10:     }
11: }
```

雖然 readUser 函數已經達到「強健度等級 3：行為回復」，但是目前的實作方法當例外發生之後只有「重試 1 次」的機會，而且還存在 *Nested Try Statement* 壞味道。你準備套用 *Introduce Resourceful Try Block（269）*重構來解決以上兩個問題。

首先，在 readUser 函數內宣告變數決定最多重試次數，在此將執行 try block 3 次：

```
final int maxAttempt = 3;
```

接著宣告變數用來代表重試次數，並將其初始值設為 1。

```
int attempt = 1;
```

使用 while(true) 迴圈將 try statement 包起來：

```
final int maxAttempt = 3;
int attempt = 1;
while(true){
    try{
        return readFromDatabase(name);
    } catch(Exception e){
        try{
            return readFromLDAP(name);
        } catch(IOException ex){
            throw new ReadUserException(ex);
        }
    }
}
```

將 `catch` block 裡面的 `readFromLDAP` 函數移到 `try` block，然後加上 `if` 條件式，讓 `readFromDatabase` 函數最多執行 2 次，`readFromLDAP` 函數最多執行 1 次：

```
final int maxAttempt = 3;
int attempt = 1;
while(true){
    try{
        if(attempt <= 2) return readFromDatabase(name);
        else             return readFromLDAP(name);
    } catch(Exception e){
        try{
            return readFromLDAP(name);
        } catch(IOException ex){
            throw new ReadUserException(ex);
        }
    }
}
```

刪除 `catch` block 裡面的「備胎程式」。

```
final int maxAttempt = 3;
int attempt = 1;
```

```
 3: while(true){
 4:     try{
 5:         if(attempt <= 2) return readFromDatabase(name);
 6:         else             return readFromLDAP(name);
 7:     } catch(Exception e){
 8:         try{
 9:             return readFromLDAP(name);
10:         } catch(IOException ex){
11:             throw new ReadUserException(ex);
12:         }
13:     }
14: }
```

在 catch block 裡面將 attempt 加 1，當 attempt 超過最大重試次數時丟出 ReadUserException：

```
 1: public User readUser(String name) throws ReadUserException{
 2:     final int maxAttempt = 3;
 3:     int attempt = 1;
 4:     while(true){
 5:         try{
 6:             if(attempt <= 2) return readFromDatabase(name);
 7:             else             return readFromLDAP(name);
 8:         } catch(Exception e){
 9:             if (++attempt > maxAttempt)
10:                 throw new ReadUserException(e);
11:         }
12:     }
13: }
```

最後，編譯並測試。

***

你也可以練習將「主要方案」與「替代方案」設計成 *Command* 模式，或是採用「反射（reflection）」的方式來呼叫它們，看看上面的程式結構會有怎樣的變化。更進一步可以設計一個 RetryExecutor 類別，只要向它註冊「主要方案」、「替代方案」、「選擇函數（用來決定何時

執行主要方案或是替代方案的函數）」、重試失敗之後所要丟出的「例外類別」，以及設定最多重試次數，便可重複使用且減少很多用來實作重試策略的重複程式碼。

***

友藏內心獨白：例外處理程式也是需要經過設計的。

Column I. | 客戶滿意，老闆賺錢，你護肝

本書寫到這裡即將接近尾聲，不知鄉民們心中是否存在一個疑問：「書中介紹的這一大堆方法，到底有沒有用？」當年 Teddy 在學校做例外處理研究的時候也存在同樣的疑問，雖然自己很有信心，但如果可以找到第三方「公正單位」來驗證一下方法的可行性，以後講話可以比較大聲一點。後來有機會和一位在台灣 IBM 工作多年的學弟一起合作，在一個銀行業的案例中採用本書所提到的方法來改善系統強健度，得到不錯的成果[52]。

現況

為了取信於各位鄉民，在此簡介這個案例。如圖 I-1 所示，這個案例牽涉的是一個以 Java 語言所開發的「徵信系統」，銀行授信人員透過這個系統，將客戶（申請信用卡、房屋貸款、車貸等）的資料傳送到財團法人金融聯合徵信中心（簡稱聯徵中心），查詢客戶的信用狀況，以作為審核貸款的依據。

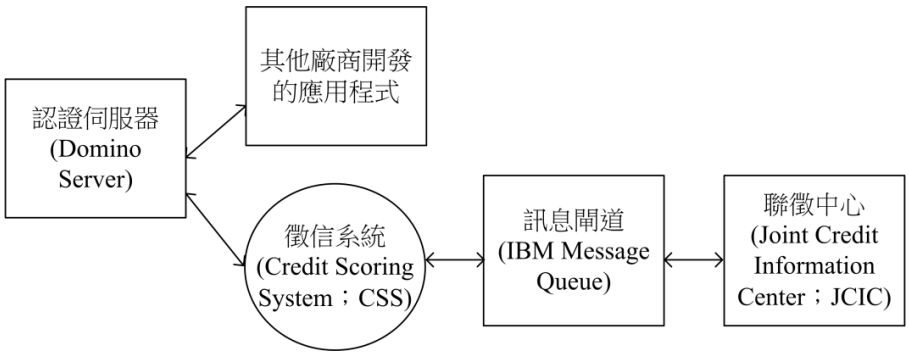


圖 I-1：徵信系統架構圖，圓形表示預計重構的系統，方形表示外部系統

銀行使用者登入徵信系統的時候，系統會連到銀行內部的認證伺服器（IBM Domino Server）確認使用者身分，然後透過訊息閘道（IBM Message Queue）作為與聯徵中心交換資料的管道。當然銀行內部系統不會只有這個徵信系統而已，還有其他廠商所開發的系統，也會使用相同的認證伺服器。

[52] C.-T. Chen, Y. C. Cheng, C.-Y. Hsieh, and I-L. Wu, "Exception Handling Refactorings: Directed by Goals and Driven by Bug Fixing," Journal of Systems and Software, volume 82, issue 2, 2009, pp. 333-345.

Teddy 跟學弟解釋了強健度等級、例外處理壞味道、例外處理重構等方法之後，學弟表示可以套用這些做法來增強徵信系統的強健度。當初在開發這個系統的時候，團隊成員並沒有特別受過任何例外處理的訓練，也沒有強健度等級的觀念。因此，該系統處於強健度等級 0—未定義：

- 程式中穿插使用回傳碼與例外。
- 有許多 checked exception 被捕捉且忽略了。
- 未被捕捉的 unchecked exception 導致 JSP（JavaServer Page）程式不預期終止。
- 函數執行失敗或是系統當機之後，系統狀態是否正確並沒有受到探討。

***

以修正 Bug 為動力的提升強健度等級策略

因為徵信系統已經上線運作，不可能跑去跟客戶說：「請把它交給我當做實驗的白老鼠吧」，和學弟討論之後決定採用修正 bug 的策略（bug-fixing strategy）來提升強健度等級。如果系統存在著例外處理壞味道，例如 *Careless Cleanup*、*Unprotected Main Program*，最終很可能因為資源洩漏導致系統無法持續運作，或是造成系統不預期終止。這些都是使用者可以觀察到的 failure，也就從客戶的角度來看，他們會打電話來抱怨系統有 bug。無論是從客戶或是公司（廠商）的角度，利用修正 bug 的同時來提升強健度等級，或是反過來說，利用提升強健度等級移除掉系統中例外處理壞味道，同時一併修正 bug，是一石二鳥的好策略。

不須先告訴客戶你要幫他提升強健度等級，只要讓他知道你會積極修正他們所抱怨的問題。

為了讓焦點放在因為例外處理不良所導致的客戶抱怨，因此學弟先收集與統計 2005 年客戶所回報的問題，並從中篩選出客訴次數最多的前三名，如表 I-1 所示。

表 I-1：2005 年客戶所回報可能與例外處理不良有關的問題統計

回報的失效內容	失效的操作/可能原因	回報次數	現有復原作法	復原時間
無法連線到認證伺服器	使用者登入/Domino 的連線資源用盡，很可能是使	125	（1）重開認證伺服器。 （2）重開徵信系統。	約 20 分鐘

	用後沒有正常釋放。			
訊息閘道連線中斷	送訊息至 JCIS/訊息閘道連線資源用盡，很可能是使用後沒有正常釋放。	66	（1）重開訊息閘道伺服器。 （2）重開徵信系統。	約 10 分鐘
無法送資料到訊息閘道	送訊息至 JCIS/有兩種可能的原因：（1）訊息佇列已滿，（2）訊息閘道因為維護或當機而離線。	10	（1）重開訊息閘道伺服器。 （2）重開徵信系統。	約 10 分鐘

使用者回報問題的第一名是「無法連線到認證伺服器」，一年之內高達 125 次，約 3 天就發生一次。如果扣除掉假日，大概 2 天就發生一次，頻率相當高。這個問題有點棘手，因為其他廠商開發的應用程式同時間也在使用同一台認證伺服器，所以很難判斷「無法連線到認證伺服器」是由那個應用程式所造成的。還好 IBM 公司有先見之明，當初賣認證伺服器給銀行的時候就考慮到備援的問題，一次賣了兩台認證伺服器給客戶。雖然這兩台認證伺服器的資料會自動同步，但是應用程式的開發者，幾乎都只使用到第一台認證伺服器。

現在要處理這個問題的策略變得很簡單了，如圖 I-2 所示：

- 以程式碼檢視（code review）的方式，檢查負責認證的程式碼是否存在例外處理壞味道，特別是 *Careless Cleanup*，確定程式是否有正確釋放認證伺服器連線資源。
- 套用例外處理重構技巧，移除找到的壞味道。
- 套用重試策略將強健度等級提升到三。
  - 在預設狀況下使用認證伺服器 1。
  - 當發生「無法連線到認證伺服器」的例外時，改用認證伺服器 2。

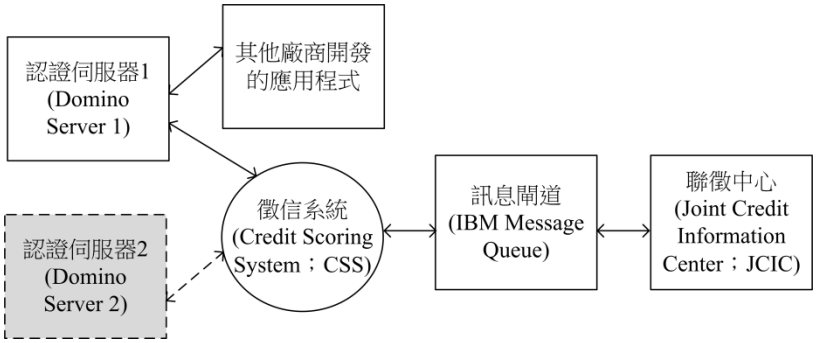


圖 I-2：修正後的徵信系統架構圖，虛線表示新增的例外處理路徑

列表I-1是負責認證動作的check函數，目前屬於強健度等級0。從程式中可以看到*Return Code*、*Dummy Handler*、*Nested Try Statement* 這幾個例外處理壞味道。看起來連線資源是有被正確釋放，可以初步排除資源洩漏的問題。

```
1: public boolean check(String userId, String password) {
2:     Session session = null;
3:     boolean result = false;
4:     try {
5:         session = NotesFactory.createSession(mServer, userId, password);
6:         result = true;
7:      } catch (AuthException e) {
8:         e.printStackTrace();
9:         result = false;
10:     } finally {
11:        if (session != null) {
12:           try {
13:               session.recycle();
14:           } catch (NotesException e) {
15:              log(e);
16:           }
17:        }
18:    }
19:    return result;
20: }
```

列表 I-1：重構前的認證函數，只有強健度等級 0。

套用 *Replace Error Code with Exception（240）*、*Replace Nested Try Statement with Method（257）*、*Introduce Resourceful Try Block（269）* 這三個重構方法後，將 check 函數由強健度等級 0 提升到強健度等級 3。重構後的程式碼如列表 I-2 所示。

```
1: public void check(String userId, String password)
2:                            throws AuthenticationException {
3:     Session session = null;
4:     int maxAttempt = 1;
```

```
 5:    int attempt = 0;
 6:    boolean retry = false;
 7:    do {
 8:       try{
 9:          retry = false;
10:          if (attempt == 0)
11:             session = NotesFactory.createSession(mPrimary, userId, password);
12:          else
13:             session = NotesFactory.createSession(mSecondary, userId, password);
14:       } catch (AuthException e) {
15:            attempt++;
16:            retry = true;
17:            if (attempt > maxAttempt)
18:               throw new AuthenticationException (e);
19:       } finally {
20:          recycle(session);
21:       }
22:     } while (attempt<= maxAttempt && retry);
23:  }
```

列表 I-2：重構後的認證函數，達到強健度等級 3

***

針對表 I-1 所列出的三個問題，將系統重構之後，在 2006 年的 9~12 月連續觀察四個月，發現這三個問題都已經消失，請參考表 I-2 所列出的重構前後使用者回報系統失效次數的比較。在重構之後的第一個月（2006 年 9 月），依然收到使用者回報高達 10 次的「無法連線到認證伺服器」的問題。經過調查之後發現這是因為其他廠商的應用程式所引起的問題，徵信系統本身並沒有再發生此問題。由於徵信系統的強健度等級提高，使用者觀察到原有問題消失所帶來的改善，因此也接受了學弟的建議，找其他廠商套用本方法一起來檢視與修正他們的系統。經過這樣的過程，2006 年 10 月這個問題發生的次數已經減少到 5 次，2006 年 11 月之後所有其他廠商所開發的應用程式都經過修正，因此相同的問題就沒有再發生過。

至於重構之後依舊發生零星的「訊息閘道連線中斷」與「無法送資料到訊息閘道」的問題，並非因為例外處理所導致，而是銀行客戶在當時將徵信系統導入到各個分行，因此一下子增加了數量可觀的使用者，導致訊息閘道無法負擔。後來訊息閘道經過系統效能調整之後，此問題就沒有再出現。

表 I-2：重構前後使用者回報系統失效次數比較表

回報的失效	2005 年 9 月到 12 月使用者回報失效次數（重構前）				2006 年 9 月到 12 月使用者回報失效次數（重構後）			
	9	**10**	**11**	**12**	**9**	**10**	**11**	**12**
無法連線到認證伺服器	13	5	9	14	10	5	0	0
訊息閘道連線中斷	6	1	4	8	3/0	0	1/0	0
無法送資料到訊息閘道	0	0	0	2	3/0	1/0	0	0

***

最後看一下此次實驗的一些數據。首先看到表 I-3，徵信系統不含註解的程式碼有 14,150 行，本次實驗只改了其中的 371 行，修改的比例為 2.63%。如果只看 `catch` block 區塊的程式碼，整個系統一共有 855 行，本次實驗改了 21 行，只佔 2.46%。也就是說，只需要針對例外處理程式的部分做小幅度修改，對於系統穩定度以及使用者滿意度的提升，就可以有很大的效益。

表 I-3：程式行數與修改比例

程式碼種類	全部/修改程式碼行數	修改比例
未包含註解的程式碼	14150 / 371	2.63%
Catch block 程式碼	855 / 21	2.46%

接著看到表 I-4，本次實驗一共花了 41 個小時，其中包含了 18 個小時用來收集與整理 2005 年使用者所回報的問題，7 個小時用來檢視程式碼與找出例外處理壞味道。接著花了 4 個小時來決定所修改的函數需要達到哪一個強健度等級，然後花了 3 小時來重構程式碼，9 小時執行測試與驗證。

表 I-4：時間花費統計

工作	人時	小計
收集與分析 2005 年使用者回報的問題	18 小時	18 小時（43.9%）
重構成本		

程式碼檢視	7 小時（30.4%）	
強健度等級分析	4 小時（17.4%）	
修改程式	3 小時（13%）	
測試	9 小時（39.2%）	
重構時數		23 小時（56.1%）
總時數		41 小時

最後看一下表 I-5，從「錢」的角度來看本次實驗的成效。假設一小時的人工成本是 3,000 元台幣，2005 年針對「無法連線到認證伺服器」、「訊息閘道連線中斷」、「無法送資料到訊息閘道」這三個問題，使用者一共回報了 201 次。假設每次公司派人去客戶端處理需要 2 個小時的時間，則 2005 年光是花在這三個問題的維護成本，就需要一百二十萬零六千元。

本次重構一共花了 41 小時，所需成本為十二萬三千元。簡單的計算之後可以發現，重構之後每年最少可以幫公司省下一百零八萬三千元，相當於 89.8%的維護成本。這些都還只是保守的估計，如果加上客戶因為系統當機而導致的損失，以及對廠商的不信任等因素，改善後的系統所能達到有形與無形的效益，絕對大於表 I-5 所列出來的金額。

表 I-5：成本效益分析

Cost element	**Cost calculation**
2005 年維護成本	201 (失效) * 2 (小時) * 3,000 (台幣) = 1,206,000 (台幣)
重構成本	41 (人-時) * 3,000 (台幣) = 123,000 (台幣)
Maintenance cost saving in the following year (money)	1,206,000 (台幣) - 123,000 (台幣) = 1,083,000 (台幣)
Maintenance cost saving in the following year (percentage)	1,083,000 (台幣) / 1,206,000 (台幣) * 100 = 89.8%

***

例外處理壞味道的辨識與重構技巧，有點像是針對癌症病患（就是各位手上要負責的那個經常當機的軟體系統）所設計的標靶藥物，可以瞄準癌細胞所在的範圍集中攻擊，盡量不去影響到

旁邊的正常細胞。如果沒有這樣的標靶藥物，漫無目的的全身化療，很有可能在殺死癌細胞的同時，連正常細胞也一併殺死。還有更慘的情況，就是癌細胞還沒殺死，就先殺死一堆正常細胞（大幅度的修改程式，結果越改問題越多，最後只好砍掉重練）。

各位鄉民們，針對工作中的系統，你準備好合適的標靶藥物了嗎？

***

友藏內心獨白：東西修一修還可以用，沒有計畫的砍掉重練，問題也不見得會消失。

42 一個函數只能有一個 Try 敘述

前一陣子讀了《Clean Code》[53]這本書，讀著、讀著突然想到一件事，當年 Teddy 還在唸書的時候，曾經提出一個在 Java 與 C#這類語言中使用 `try` statement 的原則：

一個函數只能有一個 try 敘述

（One method, one try statement）

這個想法是Teddy從Eiffel語言的例外處理做法中得到的靈感，Eiffel的例外處理是透過 `rescue`、`retry` 來達成，細節就不管它了，總之一個函數裡面只能有一個 `rescue` 敘述（可以想成 Java 的 `catch(Throwable e)`），用來捕捉例外。請參考圖 0-1 程式範例：

```
05 get_integer is
06    -- Attempt to read integer in at most
07    --     Maximum_attempts attempts.
08    -- Set value of integer_was_read to record
09    --     whether successful.
10    -- If successful, make integer available
11    --      in last_integer_read.
12
13   local
14      attempts: INTEGER
15        -- default value of attempts is 0
16   do
17      if attempts < Maximum_attempts then
18          print("Please enter an integer:")
19          read_one_integer
21          integer_was_read := True
22      else
23          integer_was_read := False
24      end
25    rescue
26       attempts := attempts + 1
27       retry
28    end
```

圖 0-1：Eiffel 例外處理程式範例

***

當年 Teddy 只是直覺認為在 Java 裡面如果也可以遵守一個函數裡面只有一個 `try` statement，應該可以讓函數正常邏輯與例外處理邏輯變得更簡單且易懂。有一點像是「單一責任原則」（single

[53] R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.

responsibility principle；SRP）[54]的道理，一個函數如果需要一個以上的 `try` statement，可能隱含這個函數負擔太多的責任。但是當時 Teddy 並沒有特別花時間去把這個想法做更進一步的整理。讀了《Clean Code》之後，突然想到把之前寫的例外處理範例程式拿出來整理一下。請看列表 0-1 的 `setupMessageV1` 函數，它接受一個 `DataInputStream` 物件，從中讀出一個訊息封包，並將此封包設定到定義 `setupMessageV1` 函數的物件屬性。這個函數並沒有做例外處理，而是把 `IOException` 直接往外丟。

```
1: public void setupMessageV1(DataInputStream aIS)
2:                                 throws IOException {
3:     int length = aIS.readInt();
4:     setMessageLength(length);
5:     byte[] messageBody= new byte[length];
6:     aIS.readFully(messageBody);
7:     setMessageBody(new String(messageBody));
8: }
```

列表 0-1：原始版本的 `setupMessageV1` 函數

***

經過分析之後，發現列表 0-1 的 `setupMessageV1` 函數做了兩件事，這兩件事都可能發生例外。首先，`setupMessageV1` 函數從 `DataInputStream` 物件讀出一個整數，這個整數代表資料封包的長度。列表 0-1 第 3 行 `aIS.readInt()` 程式可能會發生 `IOException`，表示讀取封包長度發生錯誤。

第二件事就是根據剛剛讀到的封包長度，連續讀取若干個位元組，把封包內容讀出來。列表 0-1 第 6 行 `aIS.readFully(messageBody)` 讀取封包內容的程式可能會發生兩個例外，分別是 `EOFException` 與 `IOException`。前者表示資料長度不夠，後者表示讀取資料內容發生錯誤。

知道 `setupMessageV1` 函數所負責的兩個責任之後，現在要想辦法來簡化它的責任。首先分別將實作這兩個責任的程式片段各用一個 `try` statement 包起來，如列表 0-2 所示。針對列表 0-2

[54] 一個類別或介面應該只有一個改變的理由，請參考 R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, Prentice Hall, 2002.

第 5 行 aIS.readInt()所丟出的 IOException，捕捉下來之後轉成另一個語意比較清楚的 InvalidPacketException，並且註明其發生原因為「讀取封包長度錯誤」（列表 0-2 第 8 行）。

接著，針對列表 0-2 第 12 行 aIS.readFully(messageBody)所丟出代表資料長度不夠以及讀取資料內容發生錯誤的例外，捕捉下來之後同樣轉成 InvalidPacketException，並分別註明其發生原因為「封包長度少於封包表頭資料所示」與「讀取封包內容錯誤」。

```
 1: public void setupMessageV2(DataInputStream aIS)
 2:              throws InvalidPacketException {
 3:     int length = 0;
 4:     try {
 5:         length = aIS.readInt();
 6:         setMessageLength(length);
 7:     } catch(IOException e){
 8:         throw new InvalidPacketException("讀取封包長度錯誤", e);
 9:     }
10:     try {
11:         byte [] messageBody = new byte[length];
12:         aIS.readFully(messageBody);
13:         setMessageBody(new String(messageBody));
14:     } catch (EOFException e) {
15:         throw new InvalidPacketException("封包長度少於封包表頭資料所示", e);
16:     } catch (IOException e) {
17:         throw new InvalidPacketException("讀取封包內容錯誤", e);
18:     }
19: }
```

列表 0-2：使用兩個 try 敘述區隔函數的兩種責任

由列表 0-2 可知，setupMessageV2 函數一共有三個狀況會發生例外，雖然對外都是丟出 InvalidPacketException，但是錯誤的原因各不相同。如果採用 setupMessageV1 的做法直接丟出 IOException，則客戶端的例外處理程序很難判斷錯誤發生的原因到底是以上三種狀況的哪一種（例外處理設計變得很困難）。

***

套用「一個函數裡面只能有一個 try 敘述」這個設計原則，以及簡單的 *Extract Method*，將列表 0-2 程式重構成列表 0-3 程式。首先新增一個 `readMessageLength` 函數，把第一個 `try` statement 的程式片段搬到裡面，接著在原本的函數裡面呼叫 `readMessageLength` 函數取代被抽離出去的程式碼。接著再新增 `readMessageBody` 函數，把第二個 `try` statement 的程式片段搬到裡面，然後在原本的函數裡面呼叫 `readMessageBody` 函數取代抽離出去的程式碼。

```
1: public void setupMessageV3(DataInputStream aIS)
2:                               throws InvalidPacketException {
3:     int length = readMessageLength(aIS);
4:     byte[] messageBody = readMessageBody(length, aIS);
5:     setMessageLength(length);
6:     setMessageBody(new String(messageBody));
7: }
8: private int readMessageLength(DataInputStream aIS)
9:               throws InvalidPacketException {
10:    try {
11:        int length = aIS.readInt();
12:        return length;
13:    } catch(IOException e){
14:        throw new InvalidPacketException("讀取封包長度錯誤", e);
15:    }
16: }
17: private byte [] readMessageBody(int aLength, DataInputStream aIS)
18:                          throws InvalidPacketException {
19:    try {
20:        byte[] messageBody = new byte[aLength];
21:        aIS.readFully(messageBody);
22:        return messageBody;
23:    } catch (EOFException e) {
24:        throw new InvalidPacketException("封包長度少於封包表頭資料所示", e);
25:    } catch (IOException e) {
26:        throw new InvalidPacketException("讀取封包內容錯誤", e);
27:    }
28: }
```

列表 0-3：套用「一個函數裡面只能有一個 try 敘述」原則來執行重構

重構之後，`readMessageLength` 函數與 `readMessageBody` 函數都只有一個 `try` statement，而且也只負責單一責任。原本的 `setupMessageV3` 函數，經過重構之後不但程式邏輯變得比較清楚，而且往外傳遞的例外由原本的 `IOException` 改為 `InvalidPacketException`，語意也清楚了很多。

***

《Clean Code》第三章提到：**「函數應該只做一件事。它們應該把這件事做好。它們應該只做這件事。（Function should do one thing. They should do it well. They should do it only.）」**道理很簡單，困難點在於如何界定何者屬於「一件事」的範圍。從例外處理的角度來看，套用「一個函數只能有一個 try 敘述」這個做法，可以協助開發人員思考一個函數是否只做一件事，還是負擔了太多責任。

***

友藏內心獨白：很多時候程式寫得很亂是因為責任分派不明。

Column J. | 視力測驗

本書至此已全部結束，最後請鄉民們做個簡單的自我檢查，看看大家在例外處理設計方面的「視力」是否有變好。以下一共有 20 題單選的選擇題，答對一題得 0.1 分，最佳「視力」為 2.0。1.4 以上屬於正常，低於 1.4 屬於近視，需要再花點時間好好消化書中各章的內容，補充「眼睛」所需的營養。

請準備好筆，計時 20 分鐘，開始作答。

1. 關於例外處理的敘述，何者是錯的？
   A. 例外處理是一種非功能需求（non-functional requirement）。
   B. 妥善的例外處理可以提高系統強健度（robustness）。
   C. 因為及時上市（time-to-market）的原因，有時候就算知道產品的例外處理設計不完善，還是只能釋出讓客戶先使用，事後再想辦法慢慢修正。
   D. 所有的例外狀況在需求分析時都可以被找出來。
   E. 以上皆非。

2. 以下那種狀況應該交由例外處理來對付，才不至於需要付出容錯設計的高額成本？
   A. `NullPointerException` 或 `NullReferenceException`。
   B. 陣列索引超出範圍。
   C. 硬碟空間不足。
   D. 以上皆是。
   E. 以上皆非。

3. 在 `finally` block 裡面發生例外，採取何種做法比較洽當？
   A. 把例外往外丟。
   B. 捕捉起來然後忽略它。
   C. 捕捉起來然後印在主控台（console）。
   D. 捕捉起來然後寫到日誌檔中。
   E. 以上皆可。

4. C#與 Objective-C 的例外類別因為實作了以下哪一個模式（pattern），可以讓例外物件夾帶不固定數量的使用者自訂資料？

   A. Collecting Parameter。

   B. Variable State。

   C. Composite。

   D. Strategy。

   E. State。

5. 敏捷開發讓例外處理設計變得…

   A. 更簡單，因為不需要做太多事前設計（up-front design），所以不用考慮太多情況就可以寫動手程式。

   B. 更難，因為不允許做太多事前設計，而且需求改來改去，造成例外處理更難設計。

   C. 沒差別，因為不管用什麼開發方法，我們從來都沒花時間去設計例外處理。

   D. 更難，因為每天都要開會討論例外處理的問題。

   E. 更簡單，因為開發人員會自我管理，所以不需要擔心例外處理設計的問題。

6. Fault、error、failure 在例外處理設計中經常會聽到的三個名詞。以下關於這三者的敘述何者為對？

   A. Failure 表示程式設計錯誤（programming error）。

   B. Error 表示一個函數無法如其規格所描述，提供正確的服務。

   C. Fault 表示系統處於不正確的狀態。

   D. 以上皆非。

   E. 以上皆是。

7. 例外發生之後，如果要讓程式繼續正常執行，例外處理程序（exception handler）至少必須確定以下哪一件事情？

   A. 系統的 bug 已經被移除。

   B. 使用者獲得錯誤訊息通知。

   C. 系統自動重新啟動。

   D. 電腦沒有中毒。

   E. 系統處於正確狀態且沒有資源遺漏（resource leak）。

8. 以下節錄自 ANT 的程式碼，存在哪一個例外處理壞味道？

```
private Properties getProperties(Resource r) {
    InputStream in = null;
    Properties props = new Properties();
    try {
        in = r.getInputStream();
        props.load(in);
    } catch (IOException ioe) {
        ioe.printStackTrace();
    } finally {
        FileUtils.close(in);
    }
    return props;
}
```

A. 粗心的資源釋放（Careless Cleanup）。

B. 忽略受檢例外（Ignored Checked Exception）。

C. 虛設的例外處理程序（Dummy Handler）。

D. 當成備胎的例外處理程序（Spare Handler）。

E. 沒有存在任何壞味道。

9. 以下關於 Java 的 checked exception 和 unchecked exception 的敘述，何者是錯的？

A. Checked exception 代表可回復錯誤狀況，unchecked exception 代表不可回復錯誤狀況。

B. Checked exception 如果要往函數外傳遞，必須要被宣告在函數介面上。反之，unchecked exception 不可以被宣告在函數介面上。

C. Java 編譯器會檢查 checked exception 的使用是否遵循「處理或宣告原則（handle-or-declare rule）」。

D. 在程式執行的時候（runtime），Java 虛擬機器不會檢查 checked exception 是否違反「處理或宣告原則（handle-or-declare rule）」。

E. 以上皆非。

10. Java 的 `EOFException` 屬於 checked exception。以下關於 EOFException 的敘述何者是對的？

    A. 有時候 `EOFException` 僅代表通知（notification），而非例外狀況。

    B. 因為 `EOFException` 屬於 checked exception，所以一定要把它宣告在介面上，否則會產生語法錯誤。

    C. 因為 `EOFException` 屬於 checked exception，所以一定是可回復例外（recoverable exception），捕捉下來之後要執行狀態回復動作。

    D. `EOFException` 是 `IOException` 的子類別，這種設計技巧稱為 *Smart Exception*（聰明例外）。

    E. 以上皆是。

11. 以下關於 C#例外處理機制，何者是錯的？

    A. C#預設採用 stack unwinding 來尋找例外處理程序（exception handler）。

    B. C#的例外傳遞是屬於內隱式（implicit）。

    C. C#使用資料物件（data object）來表達例外。

    D. C#的例外模型採用終止模型（termination model）。

    E. 因為 C#沒有區分 checked exception 與 unchecked exception，所以也沒有辦法區分可回復例外與不可回復例外。

12. 兩棟大樓之間的雷射資料傳輸，因為有鴿子飛過而導致資料傳輸失效。此種失效原因，可稱為？

    A. 間歇缺陷（intermittent fault）。

    B. 永久缺陷（permanent fault）。

    C. 週期性缺陷（periodic fault）。

    D. 安全性缺陷（safety fault）。

    E. 暫態缺陷（transient fault）。

13. Java 語言的非同步例外（asynchronous exception）由誰產生？

    A. 多執行緒（thread）。

    B. 多行程（process）。

    C. 非同步呼叫。

    D. Java 虛擬機器（JVM）。

    E. 以上皆是。

14. 以下節錄自 JFreeChart 的程式碼，**不存在**哪一個例外處理壞味道？

```
private RegularTimePeriod createInstance(Class periodClass,
          Date millisecond, TimeZone zone, Locale locale) {

    RegularTimePeriod result = null;
    try {
       Constructor c = periodClass.getDeclaredConstructor
         (new Class[] {Date.class, TimeZone.class, Locale.class});
       result = (RegularTimePeriod) c.newInstance
                    (new Object[] {millisecond, zone, locale});
    }
    catch (Exception e) {
        try {
          Constructor c = periodClass.getDeclaredConstructor
                                  (new Class[] {Date.class});
          result = (RegularTimePeriod) c.newInstance(
                        new Object[] {millisecond});
        }
        catch (Exception e2) {
              // do nothing
        }
     }
    return result;
}
```

A. 巢狀 Try 敘述（*Nested Try Statement*）。
B. 忽略受檢例外（*Ignored Checked Exception*）。
C. 虛設的例外處理程序（*Dummy Handler*）。
D. 當成備胎的例外處理程序（*Spare Handler*）。
E. 以上皆非。

15. Java 的 `Runnable` 介面定義了 `run` 函數，它不允許拋出任何 checked exception。如果你想在 `run` 函數丟出 checked exception，可以套用以下哪一個設計模式？
A. 同質性例外（*Homogeneous Exception*）
B. 未處理的例外（*Unhandled Exception*）
C. 聰明例外（*Smart Exception*）
D. 例外階層（*Exception Hierarchy*）

E. 通道例外（*Tunneling Exception*）

16. Forward recovery（向前回復）可以達到哪一個強健度等級？
    A. 強健度等級 0 ：未定義。
    B. 強健度等級 1 ：錯誤回報。
    C. 強健度等級 2 ：狀態回復。
    D. 強健度等級 3 ：行為回復。
    E. 以上皆非。

17. 以下何者屬於 backward recovery（向後回復）的限制？
    A. 不適合用來復原因為暫態缺陷（transient fault）所造成的錯誤。
    B. 需要特別研究錯誤發生原因才可以用來回復狀態。
    C. 很耗費資源。
    D. 不屬於通用的狀態回復方法。
    E. 以上皆是。

18. 以下哪一個 Java 例外類別套用了「聰明例外」（Smart Exception）設計模式？
    A. `IOException`。
    B. `SQLException`。
    C. `NullPointerException`。
    D. `OutOfMemoryException`。
    E. `IndexOutOfBoundsException`。

19. 在工具支援觀點中，例外被分成哪兩類？
    A. reminded（提醒）與 unreminded（不提醒）。
    B. checked（受檢）與 unchecked（非受檢）。
    C. reocverable（可回復）與 unrecoverable（不可回復）。
    D. declared（公告）與 undeclared（非公告）。
    E. failure（失效）與 notification（通知）。

20. 巢狀 Try 敘述（*Nested Try Statement*）壞味道可以用何種重構技巧來移除？
    A. 引入多才多藝的 TRY 區塊（*Introduce Resourceful Try Block*）。
    B. 提煉函數（*Extract Method*）。
    C. 使用最外層 TRY 敘述避免意外終止（*Avoid Unexpected Termination with Big Outer Try Statement*）。
    D. 引入檢查點類別（*Introduce Checkpoint Class*）。
    E. 用例外代替錯誤碼 （*Replace Error Code with Exception*）。

***

友藏內心獨白：參考答案在附錄 A。

附錄 A ：視力測驗參考答案

1. 關於例外處理的敘述，何者是錯的？
   A. 例外處理是一種非功能需求（non-functional requirement）。
   B. 妥善的例外處理可以提高系統強健度（robustness）。
   C. 因為及時上市(time-to-market)的原因，有時候就算知道產品的例外處理設計不完善，還是只能釋出讓客戶先使用，事後再想辦法慢慢修正。
   D. 所有的例外狀況在需求分析時都可以被找出來。
   E. 以上皆非。

答案：D，許多例外狀況在實作階段才會突顯出來。

2. 以下那種狀況應該交由例外處理來對付，才不至於需要付出容錯設計的高額成本？
   A. `NullPointerException` 或 `NullReferenceException`。
   B. 陣列索引超出範圍。
   C. 硬碟空間不足。
   D. 以上皆是。
   E. 以上皆非。

答案：C，硬碟空間不足屬於 component fault，可由例外處理機制處理。A、B 屬於 design fault，代表程式中有 bug，應該修改程式移除 bug。

3. 在 `finally` block 裡面發生例外，採取何種做法比較洽當？
   A. 把例外往外丟。
   B. 捕捉起來然後忽略它。
   C. 捕捉起來然後印在主控台（console）。
   D. 捕捉起來然後寫到日誌檔中。
   E. 以上皆可。

答案：D，請參考列表 14-2。

4. C#與 Objective-C 的例外類別因為實作了以下哪一個模式（pattern），可以讓例外物件夾帶不固定數量的使用者自訂資料？

   A. Collecting Parameter。

   B. Variable State。

   C. Composite。

   D. Strategy。

   E. State。

答案：B，請參考〈CH9：例外處理的四種脈絡〉的註釋 10。

5. 敏捷開發讓例外處理設計變得…

   A. 更簡單，因為不需要做太多事前設計（up-front design），所以不用考慮太多情況就可以寫動手程式。

   B. 更難，因為不允許做太多事前設計，而且需求改來改去，造成例外處理更難設計。

   C. 沒差別，因為不管用什麼開發方法，我們從來都沒花時間去設計例外處理。

   D. 更難，因為每天都要開會討論例外處理的問題。

   E. 更簡單，因為開發人員會自我管理，所以不需要擔心例外處理設計的問題。

答案：B，請參考〈CH26：流程觀點〉。

6. Fault、error、failure 在例外處理設計中經常會聽到的三個名詞。以下關於這三者的敘述何者為對？

   A. Failure 表示程式設計錯誤（programming error）。

   B. Error 表示一個函數無法如其規格所描述，提供正確的服務。

   C. Fault 表示系統處於不正確的狀態。

   D. 以上皆非。

   E. 以上皆是。

答案：D，請參考〈CH8：強健性大戰首部曲：威脅潛伏〉。

7. 例外發生之後，如果要讓程式繼續正常執行，例外處理程序（exception handler）至少必須確定以下哪一件事情？

   A. 系統的 bug 已經被移除。

   B. 使用者獲得錯誤訊息通知。

   C. 系統自動重新啟動。

   D. 電腦沒有中毒。

   E. 系統處於正確狀態且沒有資源遺漏（resource leak）。

答案：E，請參考〈CH27：例外處理設計的第一步：決定強健度等級〉。

8. 以下節錄自 ANT 的程式碼，存在哪一個例外處理壞味道？

```
private Properties getProperties(Resource r) {
    InputStream in = null;
    Properties props = new Properties();
    try {
        in = r.getInputStream();
        props.load(in);
    } catch (IOException ioe) {
        ioe.printStackTrace();
    } finally {
        FileUtils.close(in);
    }
    return props;
}
```

   F. 粗心的資源釋放（Careless Cleanup）。

   G. 忽略受檢例外（Ignored Checked Exception）。

   H. 虛設的例外處理程序（Dummy Handler）。

   I. 當成備胎的例外處理程序（Spare Handler）。

   J. 沒有存在任何壞味道。

答案：C，請參考〈CH34：例外處理壞味道〉。

9. 以下關於 Java 的 checked exception 和 unchecked exception 的敘述，何者是錯的？

   A. Checked exception 代表可回復錯誤狀況，unchecked exception 代表不可回復錯誤狀況。

   B. Checked exception 如果要往函數外傳遞，必須要被宣告在函數介面上。反之，unchecked exception 不可以被宣告在函數介面上。

   C. Java 編譯器會檢查 checked exception 的使用是否遵循「處理或宣告原則（handle-or-declare rule）」。

   D. 在程式執行的時候（runtime），Java 虛擬機器不會檢查 checked exception 是否違反「處理或宣告原則（handle-or-declare rule）」。

   E. 以上皆非。

答案：B，unchecked exception 也可以宣告在介面上，請參考第 69 頁，關於「選擇式」的內容說明。

10. Java 的 `EOFException` 屬於 checked exception。以下關於 EOFException 的敘述何者是對的？

    A. 有時候 `EOFException` 僅代表通知（notification），而非例外狀況。

    B. 因為 `EOFException` 屬於 checked exception，所以一定要把它宣告在介面上，否則會產生語法錯誤。

    C. 因為 `EOFException` 屬於 checked exception，所以一定是可回復例外（recoverable exception），捕捉下來之後要執行狀態回復動作。

    D. `EOFException` 是 `IOException` 的子類別，這種設計技巧稱為 *Smart Exception*（聰明例外）。

    E. 以上皆是。

答案：A，請參考列表 9-4 與列表 9-5。

11. 以下關於 C#例外處理機制，何者是錯的？

    A. C#預設採用 stack unwinding 來尋找例外處理程序（exception handler）。

    B. C#的例外傳遞是屬於內隱式（implicit）。

    C. C#使用資料物件（data object）來表達例外。

D. C#的例外模型採用終止模型（termination model）。

E. 因為 C#沒有區分 checked exception 與 unchecked exception，所以也沒有辦法區分可回復例外與不可回復例外。

答案：E，請參考〈CH20：Checked 與 Unchecked 例外的語意與問題〉。

12. 兩棟大樓之間的雷射資料傳輸，因為有鴿子飛過而導致資料傳輸失效。此種失效原因，可稱為？

A. 間歇缺陷（intermittent fault）。

B. 永久缺陷（permanent fault）。

C. 週期性缺陷（periodic fault）。

D. 安全性缺陷（safety fault）。

E. 暫態缺陷（transient fault）。

答案：E，請參考〈CH8：強健性大戰首部曲：威脅潛伏〉。

13. Java 語言的非同步例外（asynchronous exception）由誰產生？

A. 多執行緒（thread）。

B. 多行程（process）。

C. 非同步呼叫。

D. Java 虛擬機器（JVM）。

E. 以上皆是。

答案：D，請參考〈CH10：物件導向語言的例外處理機制〉。

14. 以下節錄自 JFreeChart 的程式碼，**不存在**哪一個例外處理壞味道？

```
private RegularTimePeriod createInstance(Class periodClass,
          Date millisecond, TimeZone zone, Locale locale) {

    RegularTimePeriod result = null;
    try {
      Constructor c = periodClass.getDeclaredConstructor
        (new Class[] {Date.class, TimeZone.class, Locale.class});
      result = (RegularTimePeriod) c.newInstance
```

```
                    (new Object[] {millisecond, zone, locale});
        }
        catch (Exception e) {
            try {
              Constructor c = periodClass.getDeclaredConstructor
                                        (new Class[] {Date.class});
              result = (RegularTimePeriod) c.newInstance(
                            new Object[] {millisecond});
            }
            catch (Exception e2) {
                  // do nothing
            }
         }
        return result;
    }
```

A. 巢狀 Try 敘述（*Nested Try Statement*）。
B. 忽略受檢例外（*Ignored Checked Exception*）。
C. 虛設的例外處理程序（*Dummy Handler*）。
D. 當成備胎的例外處理程序（*Spare Handler*）。
E. 以上皆非。

答案：C，請參考〈CH34：例外處理壞味道〉。

15. Java 的 `Runnable` 介面定義了 `run` 函數，它不允許拋出任何 checked exception。如果你想在 `run` 函數丟出 checked exception，可以套用以下哪一個設計模式？
    A. 同質性例外（*Homogeneous Exception*）
    B. 未處理的例外（*Unhandled Exception*）
    C. 聰明例外（*Smart Exception*）
    D. 例外階層（*Exception Hierarchy*）
    E. 通道例外（*Tunneling Exception*）

答案：E，請參考〈CH31：例外類別設計與使用技巧〉。

16. Forward recovery（向前回復）可以達到哪一個強健度等級？
    A. 強健度等級 0 ：未定義。
    B. 強健度等級 1 ：錯誤回報。

C. 強健度等級 2 ：狀態回復。
D. 強健度等級 3 ：行為回復。
E. 以上皆非。

答案：D，請參考〈CH30：強健度等級 3：行為回復的實作策略〉。

17. 以下何者屬於 backward recovery（向後回復）的限制？
A. 不適合用來復原因為暫態缺陷（transient fault）所造成的錯誤。
B. 需要特別研究錯誤發生原因才可以用來回復狀態。
C. 很耗費資源。
D. 不屬於通用的狀態回復方法。
E. 以上皆是。

答案：C，請參考〈CH29：強健度等級 2：狀態回復的實作策略〉。

18. 以下哪一個 Java 例外類別套用了「聰明例外」（Smart Exception）設計模式？
A. `IOException`。
B. `SQLException`。
C. `NullPointerException`。
D. `OutOfMemoryException`。
E. `IndexOutOfBoundsException`。

答案：B，請參考〈CH31：例外類別設計與使用技巧〉。

19. 在工具支援觀點中，例外被分成哪兩類？
A. reminded（提醒）與 unreminded（不提醒）。
B. checked（受檢）與 unchecked（非受檢）。
C. reocverable（可回復）與 unrecoverable（不可回復）。
D. declared（公告）與 undeclared（非公告）。
E. failure（失效）與 notification（通知）。

答案：A，請參考〈CH25：工具支援觀點〉。

20. 巢狀 Try 敘述（*Nested Try Statement*）壞味道可以用何種重構技巧來移除？
A. 引入多才多藝的 TRY 區塊（*Introduce Resourceful Try Block*）。
B. 提煉函數（*Extract Method*）。

C. 使用最外層 TRY 敘述避免意外終止（*Avoid Unexpected Termination with Big Outer Try Statement*）。
D. 引入檢查點類別（*Introduce Checkpoint Class*）。
E. 用例外代替錯誤碼 （*Replace Error Code with Exception*）。

答案：B，請參考〈CH39：以函數取代巢狀 Try 敘述〈Replace Nested Try Statement with Method〉〉。

***

友藏內心獨白：有人 20 題全都答對的嗎？

作者簡介

Teddy Chen，泰迪軟體創辦人，著有《笑談軟體工程：敏捷開發法的逆襲》與《笑談軟體工程：例外處理設計的逆襲》等書。

Teddy 畢業於台北科技大學機電科技研究所（資訊組）博士班，有二十年以上的軟體開發經驗。Teddy 曾擔任程式設計師、技術總監、專案經理、軟體架構師，目前為敏捷顧問與敏捷培訓課程講師，並在台北科技大學資工所擔任兼任助理教授，開設敏捷與精實軟體開發、軟體架構兩個課程。

聯絡 Teddy：
teddy@teddysoft.tw

部落格：
https://teddy-chen-tw.blogspot.com/

搞笑談軟工 FB 社群:　https://www.facebook.com/groups/teddy.tw/

This page is intentionally left blank