前言

閱讀本書可讓你的程式設計技能更上一層樓。因為你一定會從本書提供的實務知識中受益，所以這還不賴。若你有豐富的 C 程式設計經驗，將從書裡學到優質設計決策的細節及其優缺點。若你是 C 程式設計的新鮮人，將從書中獲得設計決策的相關指引，理解這些決策如何被逐步運用到示例中，進而建置大型程式。

本書解決某些問題，譬如：如何對 C 程式結構化、如何做錯誤處理、如何設計有彈性的介面。當你對 C 程式設計有進一步了解時，往往會乍現一些問題，例如：

- 應該回傳程式碼的錯誤資訊嗎？
- 應該使用全域變數 `errno` 作業嗎？
- 應該實作少量的「多參數的函式」，還是多個「少量參數的函式」呢？
- 要如何建置有彈性的介面呢？
- 要如何建置像迭代器這樣的基本項目呢？

就物件導向的語言來說，Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides 在《*Design Patterns: Elements of Reusable Object-Oriented Software*》（Prentice Hall，1997）這本四人幫（Gang of Four）書中已解答上述的大部分問題。設計模式為程式設計師提供以下議題的最佳做法：物件應該如何互動，以及哪個物件擁有另外哪些種類的物件。此外，設計模式也會解釋如何將這些物件按分類湊在一起。

然而，針對像 C 這樣的程序式程式語言來說，這些設計模式大部分都無法用四人幫所描述的方式實作。C 中並無原生的物件導向機制。可用 C 程式語言模擬繼承或多型，不過因為這樣的模擬會讓慣用 C 的程式設計師（但不常以像 C++ 這樣的物件導向程式語言編程，以及運用如繼承與多型之類的概念者）感到陌生，所以這可能不是首選。這類程

式設計師可能執意採取他們慣用的 C 程式設計原生風格。然而，對於 C 程式設計的原生風格而言，並非所有物件導向設計模式的指引都適用，或者至少不會特別為非物件導向的程式語言，提供設計模式中所呈現之概念的具體實作。

而這就是本書的立意：我們希望以 C 編程，但無法直接利用現有設計模式中記載的大部分知識。本書說明如何彌補這個落差，並實作 C 程式語言的實務設計知識。

本書的撰寫動機

讓我來告訴你，為什麼本書蒐集的知識對我而言相當重要，為什麼難以發現這些知識。

在學校裡，我所學的第一個程式語言是 C 語言。就像每位 C 程式設計的新鮮人一樣，我想知道為什麼陣列的索引從 0 開始，以及最初相當隨意地嘗試怎樣擺放 `*`、`&` 運算元，最終才能讓 C 指標神奇的運作。

大學時我學到 C 語法的實際運作方式，以及將其轉成硬體上的位元和位元組。有了這些知識，我就能寫出結果相當不錯的小程式。然而，我仍然難以理解為什麼較長的程式碼看起來是那樣，當然也提不出類似以下的解決方案：

```
typedef struct INTERNAL_DRIVER_STRUCT* DRIVER_HANDLE;
typedef void (*DriverSend_FP)(char byte);
typedef char (*DriverReceive_FP)();
typedef void (*DriverIOCTL_FP)(int ioctl, void* context);

struct DriverFunctions
{
  DriverSend_FP fpSend;
  DriverReceive_FP fpReceive;
  DriverIOCTL_FP fpIOCTL;
};

DRIVER_HANDLE driverCreate(void* initArg, struct DriverFunctions f);
void driverDestroy(DRIVER_HANDLE h);
void sendByte(DRIVER_HANDLE h, char byte);
char receiveByte(DRIVER_HANDLE h);
void driverIOCTL(DRIVER_HANDLE h, int ioctl, void* context);
```

檢視上述程式碼而產生下列的諸多問題：

- 為何在 `struct` 中有函式指標？
- 為何函式需要 `DRIVER_HANDLE`？

- IOCTL 是什麼，為什麼不改用個別的函式實作？
- 為何要明確地建立和銷毀函式？

當我開始編寫工業應用程式時，這些問題就出現了。經常遇到的情況是，我體認到自己沒有 C 程式設計知識，例如，決定如何實作迭代器或如何進行函式中的錯誤處理。我發覺，雖然知道 C 語法，但卻不知道如何應用之。我試著實現一些目標，但僅是以笨拙的方式完成，或者根本做不來。我需要如何使用 C 語言完成特定任務的最佳做法。例如，需要知道類似如下的內容：

- 如何用簡單方式獲取及釋放資源
- 使用 `goto` 做錯誤處理，是好主意嗎？
- 應該把介面設計得有彈性嗎？或應該在有需要時直接變更介面嗎？
- 應該使用 `assert` 述句，還是應該回傳錯誤碼？
- C 的迭代器是如何實作的？

讓我覺得非常有趣的是，雖然經驗資深的同事對這些問題有多種答案，但是沒有人可以幫我指引這些設計決策及其優缺點的記載內容。

因此，接著我轉向網際網路，然而又驚訝地發覺，即使 C 程式語言已存在幾十年，還是很難找到這些問題的合理解答。我發現，雖然有很多 C 程式語言基礎及其語法的相關文獻，但對於 C 程式設計進階主題，或者如何撰寫出適合工業應用的優美 C 程式，有著墨的文獻並不多。

而這正是本書出現的原因。本書教你如何提升程式設計技能，從編寫基礎的 C 程式轉而撰寫大型的 C 程式，其中會考量錯誤處理，並讓需求與設計中的某些未來變化具有彈性。本書採用設計模式的概念，為你逐步介紹設計決策及其優缺點。這些設計模式會運用於示例中，告訴你像範例起初那樣的程式碼是如何演變的，以及為何範例最終看起來是這副模樣。

本書的模式可運用於各個領域的 C 程式設計。由於我涉及的是多執行緒即時環境的嵌入式程式設計領域，因此某些模式會偏向該領域。無論如何，你會看到這些模式的一般概念可用在其他領域的 C 程式設計，甚至超越 C 程式設計的應用範疇。

模式基礎

本書以模式的形式提供設計指引。用模式呈現知識和最佳做法的概念，源自建築師 Christopher Alexander 的《*The Timeless Way of Building*》（Oxford University Press，1979）一書。他以歷經證實的小解決方案解決在他專業領域中的大問題：如何設計與建造城市。軟體開發領域採取模式運用的做法，在專門舉行的會議中，例如 Pattern Languages of Programs（PLoP）會議，得以擴展模式知識本體。尤其四人幫的《*Design Patterns: Elements of Reusable Object-Oriented Software*》（Prentice Hall，1997）一書帶來重大影響，讓設計模式的概念廣為軟體開發者所知。

但究竟什麼是模式呢？諸多定義不勝枚舉，若你對這個主題深感興趣，則 Frank Buschmann 等人所著的《*Pattern-Oriented Software Architecture: On Patterns and Pattern Languages*》（Wiley，2007）一書可以為你提供準確的描述和細節。就本書的目的而言，模式乃為實際的問題提供歷經證實的解決方案。本書模式的章節呈現如表 P-1 所示的結構。

表 P-1　本書模式的分節呈現方式

小節標題	小節內容
模式	此為模式的名稱（你應該能輕鬆記住）。目的是讓該名稱能被程式設計師用於日常用語中（如同使用四人幫的模式一般，你會聽到程式設計師說：「用 Abstract Factory 建立該物件」）。本書的模式名稱是大寫的。
情境	情境小節會設置該模式的情景。告訴你在何種情況下可以運用此模式。
問題	問題小節提供你要處理的議題相關資訊。首段以粗體字描寫主要問題的陳述，隨後的段落描述難以解決該問題的原因細節。就其他的模式格式來說，這些原因細節會被安排在名為「作用力」（force）的單獨一節中。
解決方案	本節就如何處理問題而提供指引。首段以粗體字撰寫解決方案的主要概念，隨後的段落則為解決方案的細節。另外還有一個範例程式，提供相當具體的指引。
結果	本節列出所述解決方案的運用優缺點。運用模式時，你應該一直確認作業所生的結果是沒有問題的。
已知應用	已知應用小節為你提供證據，證明所提出的解決方案是不錯的，是在實際應用程式中確實有效的。其中還會呈現具體的範例，協助你了解如何運用模式。

以模式的形式呈現設計指引，其主要好處是這些模式可一個接著一個的被套用。若你有一個大型設計問題，很難找到一個指引文件及一個解決方案能處理這個問題，則可以將這個大型而相當特定的問題，視為多個小型而較為一般的問題，你可以一個接著一個的運用多個模式，逐一解決這些問題。你只要確認這些模式的問題描述，並運用適合的模式（該模式可符合你的問題，能產生合你意的結果）。這些結果可能會導致另一個問題，而你可以套用另一個模式解決該問題。這樣就會逐步設計出你的程式碼，而不是在寫第一行程式碼之前，就得試著提出完整的前期設計。

本書的閱讀方式

你應該已經具備 C 程式設計的基礎。知道 C 語法及其運作方式——例如，本書不會教你何謂指標或如何用指標。本書旨在提供進階主題的建議與指引。

書中各章獨立。你可以按任意順序閱讀各章，可以僅挑選感興趣的主題。下一節會列出本書所有模式的概觀，你可以透過該節的概述內容跳至感興趣的模式章節。所以，若你確切地知道要找的是什麼，就可以從該節開始閱讀。

若你不是要尋找某個特定模式，而是想得知 C 程式可能的設計抉擇概觀，則可以閱讀本書的第一部分。每一章都有特定的主題，從基本的主題（譬如錯誤處理和記憶體管理）開始，接著轉到進階的特定主題（例如介面設計或與平台無關的程式碼）。每一章會介紹與該章主題相關的模式，以及每一章會有一個示例，用於說明如何逐一運用這些模式。

本書第二部分介紹兩個大型案例，其中會運用第一部分的諸多模式。就此，你可以了解如何透過模式的應用，逐步建置大型軟體。

模式概觀

你可以從表 P-2 ～表 P-10 找到本書所有模式的概述。這些表格以簡短形式呈現模式內容，其中只包含問題一節的核心（首段）內容，接著是關鍵字「因此」，最後是解決方案一節的核心（首段）內容等簡述。

表 P-2　錯誤處理的模式

模式	摘要
「Function Split」（第 6 頁）	此函式有多個職責，讓人難以閱讀與維護。因此，將該函式分解。取出其中看似有用的部分，為這些內容建立新函式，並呼叫此新函式。
「Guard Clause」（第 9 頁）	因為該函式將前置條件檢查與主程式邏輯混在一起，所以會不好閱讀與維護。因此，檢查是否有強制的前置條件，若不符合這些前置條件，則此函式會立即回返。
「Samurai Principle」（第 12 頁）	在回傳錯誤資訊時，你假設呼叫者會檢查此資訊。然而，呼叫者可能完全忽略該檢查，而該錯誤也許就會被忽視。因此，無論結果成功與否，函式皆會回返。若有發生已知錯誤無法被處理的情況，則中止程式執行。
「Goto Error Handling」（第 16 頁）	若要在某函式內多處獲取與清理多個資源程式，則會造成難以閱讀與維護的程式碼。因此，將所有資源的清理與錯誤處理置於函式的結尾。若無法獲取某資源，則使用 `goto` 述句跳至資源清理的程式碼。
「Cleanup Record」（第 19 頁）	若某段程式碼會獲取並清理多個資源，而那些資源又相依時，則會讓人難以閱讀與維護該程式碼。因此，呼叫資源獲取函式，只要函式運作成功，就將其列為待清理的函式（儲存起來）。根據這些儲存值，呼叫對應的清理函式。
「Object-Based Error Handling」（第 22 頁）	一個函式負有多個職責（譬如資源獲取、資源清理與資源運用），其程式碼會讓人難以實作、閱讀、維護、測試。因此，將初始化與清理作業置於個別的函式中，類似於物件導向程式設計中建構式與解構式的概念。

表 P-3　回傳錯誤資訊的模式

模式	摘要
「Return Status Codes」（第 32 頁）	你想要有個機制，能夠將狀態資訊回傳給呼叫者，讓呼叫者可以對該狀態有所反應。你希望這個機制簡單好用，而呼叫者應該能夠明確區分可能發生的各種錯誤情況。因此，使用函式的 Return Value 回傳狀態資訊。以回傳的值表示特定狀態。被呼叫者與呼叫者雙方必須對值的含意有共同認知。
「Return Relevant Errors」（第 39 頁）	一方面，呼叫者應能對錯誤做出反應；另一方面，回傳的錯誤資訊越多，被呼叫者的程式碼與呼叫者的程式碼需要的錯誤處理就越多，如此會讓程式碼較為冗長。冗長程式碼的閱讀與維護並不容易，進而招致額外錯誤的風險。因此，若錯誤資訊和呼叫者相關的話，才將該錯誤資訊回傳給呼叫者。若呼叫者可以對這個資訊有所反應，即表示該錯誤資訊與呼叫者相關。

模式	摘要
「Special Return Values」（第 45 頁）	你想要回傳錯誤資訊，但不想明確選用 Return Status Codes，理由是這表示你無法使用函式的 Return Value 回傳其他資料。你得透過 Out-Parameters 回傳該資料，不過如此會使得呼叫函式更加困難。因此，使用函式的 Return Value 回傳函式算出的資料。保留一個或多個特殊值，以供發生錯誤時對應回傳之用。
「Log Errors」（第 49 頁）	你要確保在發生錯誤時可以輕易找出個中原因。可是不希望錯誤處理程式碼因而變得複雜。因此，使用不同管道提供相關的錯誤資訊（「與呼叫程式碼有關的」以及「與開發者有關的」）。例如，將除錯的錯誤資訊寫入記錄檔，而不會把除錯用的詳細錯誤資訊回傳給呼叫者。

表 P-4　記憶體管理的模式

模式	摘要
「Stack First」（第 62 頁）	選擇變數儲存處的種類和記憶體區段（堆疊、堆積……），是每位程式計師得經常做的決定。若每個變數的所有利弊折衷，都必須仔細考量的話，那會讓人筋疲力盡。因此，預設的情況是直接將變數置於堆疊中，以取得堆疊變數的自動清理優勢。
「Eternal Memory」（第 66 頁）	保留大量資料，而於函式的呼叫之間傳輸這些資料，並非易事，理由是你必須確保儲存資料的記憶體夠大，其生命期可以跨函式的呼叫而延續不絕。因此，將資料放在程式整個生命期間皆可供使用的記憶體中。
「Lazy Cleanup」（第 69 頁）	若你需要大量記憶體，但事先不曉得所需的大小為何，則可用動態記憶體。然而，處理動態記憶體的清理作業很棘手，況且是許多程式設計錯誤的根源。因此，配置動態記憶體，讓作業系統於你的程式結束時執行該記憶體的釋放作業。
「Dedicated Ownership」（第 73 頁）	動態記憶體的強大功能伴隨著必須妥善清理記憶體的重責大任。在大型程式中，難以確保能夠妥善清理所有的動態記憶體。因此，當你實作記憶體配置時，可以明確定義和記載待清理之處以及負責該作業的執行者。
「Allocation Wrapper」（第 76 頁）	動態記憶體的配置並非每次都能成功，你應該檢查程式碼中的配置，如實因應。因為你的程式碼中有諸多之處要做這樣的檢查，所以不好處理。因此，包裝配置和釋放的函式呼叫，並於這些外包函式中實作錯誤處理或另外的記憶體管理組織。

第一章

錯誤處理

錯誤處理（error handling）是編寫軟體的重要環節，若處理不當，軟體就會變得難以擴展和維護。像 C++、Java 這樣的程式語言提供「異常情況」（exception）、「解構式」（destructor），以利錯誤處理。C 語言並無類似的機制，而針對 C 語言錯誤處理的妥善做法，其相關文獻資料散落在網際網路各處。

本章基於 C 的錯誤處理模式，集結妥善錯誤處理的知識，以及用一個示例呈現這些模式的運用。這些模式具有妥善做法的設計決策，並詳細說明應用時機與其賦予的結果。就程式設計師而言，這些模式可減少許多細膩決策的負擔。而程式設計師能以這些模式中所呈現的知識，作為寫出優質程式碼的出發點。

圖 1-1 概略呈現本章探討的模式以及這些模式彼此的關係，而表 1-1 列出這些模式的摘要。

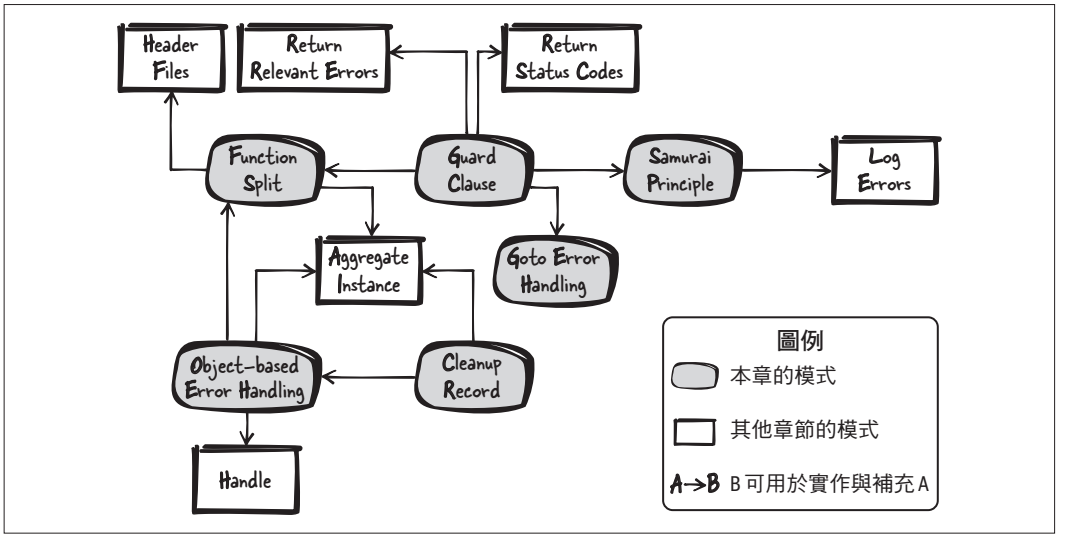

圖 1-1　錯誤處理的模式概觀

表 1-1　錯誤處理的模式

模式	摘要
Function Split	此函式有多個職責，讓人難以閱讀與維護。因此，將該函式分解。取出其中看似有用的部分，為這些內容建立新函式，並呼叫此新函式。
Guard Clause	因為該函式將前置條件檢查與主程式邏輯混在一起，所以會不好閱讀與維護。因此，檢查是否有強制的前置條件，若不符合這些前置條件，則此函式會立即回返。
Samurai Principle	在回傳錯誤資訊時，你假設呼叫者（caller）會檢查此資訊。然而，呼叫者可能完全忽略該檢查，而該錯誤也許就會被忽視。因此，無論結果成功與否，函式皆會回返。若有發生已知錯誤無法被處理的情況，則中止程式執行。
Goto Error Handling	若要在某函式內多處獲取與清理多個資源程式，則會造成難以閱讀與維護的程式碼。因此，將所有資源的清理與錯誤處理置於函式的結尾。若無法獲取某資源，則使用 `goto` 述句跳至資源清理的程式碼。
Cleanup Record	若某段程式碼會獲取並清理多個資源，而那些資源又相依時，則會讓人難以閱讀與維護該程式碼。因此，呼叫資源獲取函式，只要函式運作成功，就將其列為待清理的函式（儲存起來）。根據這些儲存值，呼叫對應的清理函式。
Object-Based Error Handling	一個函式負有多個職責（譬如資源獲取、資源清理與資源運用），其程式碼會讓人難以實作、閱讀、維護、測試。因此，將初始化（initialization）與清理作業置於個別的函式中，類似於物件導向（object oriented）程式設計中建構式與解構式的概念。

示例

你想要實作一個函式，以剖析檔案的特定關鍵字，回傳這些關鍵字所在的相關資訊。

就指出錯誤情況而言，C 語言的標準做法是，以函式回傳值表示此資訊。為了提供另外的錯誤資訊，傳統的 C 函式通常將 `errno` 變數（請參閱 *errno.h*）設為特定的錯誤碼。而呼叫者可以檢查 `errno`，得知該錯誤的相關資訊。

不過，以下的程式碼不需要詳細錯誤資訊，因此只需使用回傳值（不用 `errno`）。你可以從下列的程式碼初始片段開始實作：

```
int parseFile(char* file_name)
{
  int return_value = ERROR;
  FILE* file_pointer = 0;
  char* buffer = 0;

  if(file_name!=NULL)
  {
    if(file_pointer=fopen(file_name, "r"))
    {
      if(buffer=malloc(BUFFER_SIZE))
      {
        /* 剖析檔案內容 */
        return_value = NO_KEYWORD_FOUND;
        while(fgets(buffer, BUFFER_SIZE, file_pointer)!=NULL)
        {
          if(strcmp("KEYWORD_ONE\n", buffer)==0)
          {
            return_value = KEYWORD_ONE_FOUND_FIRST;
            break;
          }
          if(strcmp("KEYWORD_TWO\n", buffer)==0)
          {
            return_value = KEYWORD_TWO_FOUND_FIRST;
            break;
          }
        }
        free(buffer);
      }
      fclose(file_pointer);
    }
  }
  return return_value;
}
```

在上述程式碼中，你必須檢查各個函式呼叫的回傳值，確認是否有錯誤發生，因此終究會在程式碼中用深層的巢狀 `if` 述句。進而出現下列的問題：

- 此示例函式把錯誤處理、初始化、資源清理、主功能等程式碼混在一塊，而顯得冗長。如此將造成難以維護的程式碼。
- 讀取和解譯檔案資料的主程式碼位在深層的巢狀 `if` 述句中，因而難以理解其中的程式邏輯。
- 清理作業的函式與初始化作業的函式相距甚遠，因此容易忘記某些清理作業。若此示例函式有多個 `return` 述句，特別容易發生這個問題。

若要得到較好的結果，首先要用 Function Split。

Function Split

情境

你有個函式會執行多個動作。例如：配置資源、使用資源、清理資源——如動態記憶體（dynamic memory）或檔案的 handle 等資源。

問題

此函式有多個職責，讓人難以閱讀與維護。

這樣的函式可能會負責配置資源、處理資源以及清理資源。其中的清理作業甚至會分散在此函式各處，而且有些地方還會重複作業。尤其對於資源配置失敗的錯誤處理，將造就難以閱讀的函式，往往最終會出現巢狀的 `if` 述句。

以一個函式處理多個資源的配置、清理、運用，很容易忘了某資源的清理，尤其是在之後有程式碼變更的情況下。例如，若程式碼中間加入一個 `return` 述句，則往往會忘記清理函式中該述句之前已配置的資源。

解決方案

將該函式分解。取出其中看似有用的部分，為這些內容建立新函式，並呼叫此新函式。

要找出函式可獨立的部分，只需檢查是否可以為這些內容指定有意義的名稱，以及是否可以分出獨立的職責。例如，這樣的做法可能的結果是：一個函式只有功能程式碼，另一個函式僅有錯誤處理程式碼。

某函式是否要被分解，有個不錯的參考依據：有無在該函式的多處清理同一個資源。若有的話，最好將程式碼拆成兩個函式，一個用於配置和清理資源，另一個則會使用這些資源。而（用到這些資源的）被呼叫的函式可以直接具有多個 `return` 述句，無須在每個 `return` 述句之前清理資源，清理作業乃是另一個函式所為的。如下列程式碼所示：

```
void someFunction()
{
  char* buffer = malloc(LARGE_SIZE);
  if(buffer)
  {
    mainFunctionality(buffer);
  }
  free(buffer);
}

void mainFunctionality()
{
  // 主功能實作處
}
```

此時會有兩個函式（呼叫的函式與被呼叫的函式），而非原來的一個。當然，如此表示此呼叫的函式不再是自立的，而會依賴另一個函式。你必須定義另一個函式的擺放之處。第一步是將另一個函式與此呼叫的函式直接放在同一個檔案中，不過若兩個函式並非緊密耦合（closely coupled），則可以考慮將該被呼叫的函式置於單獨的實作檔中，並引入該（被呼叫的）函式的 Header File 宣告。

結果

因為相較於一個冗長的函式來說，兩個簡短函式更容易閱讀、維護，所以你對原程式碼做了改良。例如，因為清理作業函式更接近需要清理的函式，以及資源配置和清理作業沒有與主程式邏輯混在一起，所以程式碼易於閱讀。如此讓主程式邏輯更好維護，以及對往後的功能擴充更加容易。

此時被呼叫的函式，因為不必在每個 `return` 述句之前負責資源的清理，所以可以輕易加入多個 `return` 述句。清理作業乃由呼叫的函式單獨完成。

若被呼叫的函式用了許多資源，則必須另外將這些資源傳給該函式。函式的參數量過多會衍生出不易閱讀的程式碼，而在呼叫該函式時不小心將參數順序調換可能會造成程式設計錯誤。為了避免這種情況，就此你可以採用 Aggregate Instance。

已知應用

下列是此模式的應用範例：

- 幾乎所有 C 程式皆包含「運用此模式的部分」以及「未用此模式的部分」，因此難以維護。根據 Robert C. Martin 所著的《*Clean Code: A Handbook of Agile Software Craftsmanship*》（Prentice Hall，2008）書籍內容，每個函式應該只有一個職責，即單一職責原則（single-responsibility principle），因此資源處理和其他程式邏輯應一律以個別函式實作。
- Portland Pattern Repository 中將此模式稱為 Function Wrapper。
- 物件導向程式設計的 Template Method 模式也有將內容分解，以讓程式碼結構化的方法描述。
- Martin Fowler 在《*Refactoring: Improving the Design of Existing Code*》（Addison-Wesley，1999）一書中以 Extract Method 模式描述函式分解所在與分解時機的準則。
- 電玩 NetHack 將該模式用於其 `read_config_file` 函式中，該函式會處理資源以及呼叫 `parse_conf_file` 函式運用這些資源。
- OpenWrt 程式會針對緩衝區（buffer）處理，於多處使用此模式。例如，負責 MD5 計算的程式碼會配置緩衝區，將該緩衝區傳給另一個函式以供處置，再清理這個緩衝區。

運用於示例中

你的程式看起來已好很多了。目前有兩個各司其職的大函式（而非原本的一個龐大函式）。其中一個函式負責獲取與釋放資源，另一個函式負責搜尋關鍵字，如下列程式碼所示：

```
int searchFileForKeywords(char* buffer, FILE* file_pointer)
{
  while(fgets(buffer, BUFFER_SIZE, file_pointer)!=NULL)
  {
    if(strcmp("KEYWORD_ONE\n", buffer)==0)
    {
      return KEYWORD_ONE_FOUND_FIRST;
```

```
    }
    if(strcmp("KEYWORD_TWO\n", buffer)==0)
    {
      return KEYWORD_TWO_FOUND_FIRST;
    }
  }
  return NO_KEYWORD_FOUND;
}

int parseFile(char* file_name)
{
  int return_value = ERROR;
  FILE* file_pointer = 0;
  char* buffer = 0;

  if(file_name!=NULL)
  {
    if(file_pointer=fopen(file_name, "r"))
    {
      if(buffer=malloc(BUFFER_SIZE))
      {
        return_value = searchFileForKeywords(buffer, file_pointer);
        free(buffer);
      }
      fclose(file_pointer);
    }
  }
  return return_value;
}
```

`if` 串接（cascade）的深度降低，而 `parseFile` 函式仍有三個 `if` 述句用於確認資源配置是否有誤，此數量也不算少。你可以實作 Guard Clause，讓該函式無瑕。

Guard Clause

情境

你有個函式用於執行：僅在特定條件下（例如有效的輸入參數）才能成功完成的任務。

問題

因為該函式將前置條件檢查與主程式邏輯混在一起，所以會不好閱讀與維護。

配置的資源最後都要被清理。若你配置某資源，後來發現未符合函式的另一個前置條件，則也得清理該資源。

若函式裡遍布多個前置條件檢查（尤其是這些檢查內容以巢狀的 `if` 述句實作時），則會令人難以理解該程式流程。當多處都有這樣的檢查時，函式會變得非常冗長，而這本身就是一種程式碼壞味道（code smell）。

程式碼壞味道

若程式碼結構不佳或程式設計方式造成不好維護的程式碼，則代表程式碼「有壞味道」。程式碼壞味道的例子是相當冗長的函式或重複的程式碼。Martin Fowler 在《*Refactoring: Improving the Design of Existing Code*》（Addison-Wesley，1999）一書中對於程式碼壞味道的範例與對策，提及更多的內容。

解決方案

檢查是否有強制的前置條件，若不符合這些前置條件，則此函式會立即回返。

例如，檢查輸入參數是否有效，或確認程式是否為可執行該函式其餘內容的狀態。你可仔細想想，呼叫函式時要設立哪些前置條件。一方面，若你容許的函式輸入相當嚴格，這可讓你較容易實作函式內容，而另一方面，若你對可能的輸入較為寬鬆，則如此會讓函式的呼叫者較為輕鬆（如 Postel 定律所述：「對自己所做的要保守，對他人提供的要開放」）。

若你有許多前置條件的檢查，則可以呼叫單獨的函式執行這些檢查。無論如何，在所有資源配置完成前，先做這些檢查，由於該函式不用清理資源，所以可直接回返。

在函式介面中清楚描述函式的前置條件。記載該行為的最佳之處是在函式宣告的標頭檔中。

若呼叫者必須知道不符合的前置條件為何，則你可以提供錯誤資訊給呼叫者。例如，可以用 Return Status Codes，不過務必只用 Return Relevant Errors 即可。下列是未回傳錯誤資訊的範例程式：

someFile.h

```
/* 此函式會處理「user_input」，其內容不能是 NULL */
void someFunction(char* user_input);
```

someFile.c

```
void someFunction(char* user_input)
{
  if(user_input == NULL)
  {
    return;
  }
  operateOnData(user_input);
}
```

結果

相較於巢狀的 `if` 構件，當不符合前置條件時立即回返的做法，使得程式碼較好閱讀。程式碼清楚表明，若不符合前置條件，則該函式不會繼續執行。如此讓前置條件與其他程式碼完全分開。

然而，某些編程準則嚴禁於函式中間回返。例如，對於必須形式化驗證的程式碼而言，通常只允許在函式的結尾有 `return` 述句。若你想要有個錯誤處理集中處，就此可用 Cleanup Record 處置，這也是比較好的選擇。

已知應用

下列是此模式的應用範例：

- Portland Pattern Repository 有描述 Guard Clause。
- Klaus Renzel 在〈Error Detection〉（EuroPLoP 第二屆會議論文集，1997）一文中描述相當類似的模式——Error Detection，其中建議採用前置條件與後置條件的檢查。
- 電玩 NetHack 在程式多處使用此模式，例如：在 `placebc` 函式中，為 NetHack 主角加上一條鎖鏈，拖慢移動速度作為懲罰。若無可用的鎖鏈物件，則此函式會立即回返。
- OpenSSL 程式使用這個模式。例如，`SSL_new` 函式針對無效的輸入參數會立即回返。
- Wireshark 的 `capture_stats` 程式碼，負責在監聽網路封包時收集統計資料，首先檢查其輸入參數是否有效，並在參數無效時立即回返。

運用於示例中

下列的程式碼是 parseFile 函式運用 Guard Clause 檢查該函式的前置條件：

```
int parseFile(char* file_name)
{
  int return_value = ERROR;
  FILE* file_pointer = 0;
  char* buffer = 0;

  if(file_name==NULL) ❶
  {
    return ERROR;
  }
  if(file_pointer=fopen(file_name, "r"))
  {
    if(buffer=malloc(BUFFER_SIZE))
    {
      return_value = searchFileForKeywords(buffer, file_pointer);
      free(buffer);
    }
    fclose(file_pointer);
  }
  return return_value;
}
```

❶ 若提供的參數內容無效，函式將立即回返，而此時因為尚未獲取任何資源，所以不需要清理資源。

此程式以 Return Status Codes 實作 Guard Clause。其中對於 NULL 參數這種特殊情況，會回傳常數 ERROR。呼叫者此時可以檢查 Return Value，確認是否對該函式提供無效的 NULL 參數。不過這種無效參數通常被視為程式設計錯誤，而檢查程式設計錯誤，並於程式碼中傳播此資訊，並非好的做法。就此較簡單的方法是僅運用 Samurai Principle。

Samurai Principle

情境

你的某些程式內有複雜的錯誤處理（其中一些錯誤是很嚴重的）。你的系統並未執行安全關鍵作業，也無高度可用性的需求。

問題

在回傳錯誤資訊時，你假設呼叫者會檢查此資訊。然而，呼叫者可能完全忽略該檢查，而該錯誤也許就會被忽視。

在 C 語言中，並無強制要求檢查被呼叫的函式的回傳值，而呼叫者可能完全忽略該函式的回傳值。若在函式中發生的錯誤，其嚴重程度無法由呼叫者妥善處理，則你不會希望呼叫者決定是否處理錯誤以及如何處理錯誤。你反而會想要確保明確採取了某個動作。

就算呼叫者處理某錯誤情況，往往程式依然會當掉，或者仍會出錯。該錯誤可能只在某處出現——可能是呼叫者的呼叫程式碼中（可能在無法妥善處理錯誤情況之際）。就此，處理該錯誤時會掩蓋這個錯誤，而讓除錯（找出問題根源）更難。

程式的某些錯誤可能較少出現。針對此種情況用 Return Status Codes，而於呼叫者的程式碼中處理，如此會讓該程式碼的可讀性降低，主程式邏輯會失焦以及干擾呼叫者程式碼的實際功能。呼叫者為處理極少發生的錯誤情況，可能必須編寫多行程式碼。

回傳此類錯誤資訊也會造成如何確切回傳資訊的問題。函式中使用 Return Value 或 Out-Parameters 回傳錯誤資訊，將讓函式的簽名式（signature）變得更加複雜，而讓人難以理解該程式碼。就此而言，你不會想要針對只回傳錯誤資訊的函式添加另外的參數。

解決方案

無論結果成功與否，函式皆會回返（Samurai Principle）。若有發生已知錯誤無法被處理的情況，則中止程式執行。

不要使用 Out-Parameters 或 Return Value 回傳錯誤資訊。錯誤資訊的所有內容在此，所以可以立即處理錯誤。若發生錯誤，直接讓程式當掉。使用 `assert` 述句，以結構化方式中止程式執行。此外，還可以使用 `assert` 述句提供除錯資訊，如下列程式碼所示：

```
void someFunction()
{
  assert(checkPreconditions() && "Preconditions are not met");
  mainFunctionality();
}
```

此段程式碼會檢查 assert 述句中的條件，其結果若不為真（true），則 assert 述句連帶右邊的字串，會顯示在 stderr，並中止程式執行。在不檢查 NULL 指標以及存取這樣的指標之下，以不太結構化的方式中止程式執行，這是可行的。只要確保程式在發生錯誤之處會當掉即可。

通常，Guard Clauses 是錯誤發生時中止程式執行的不錯選項。例如，若你知道發生了編程錯誤（如呼叫者提供 NULL 指標），則中止程式執行，記錄除錯資訊（而非將錯誤資訊回傳給呼叫者）。然而，不要因為每種錯誤皆中止程式執行。例如，使用者輸入無效內容這類的的執行期（runtime）錯誤，當然不該讓程式中止執行。

呼叫者必須清楚知道你的函式行為，因此你必須在函式的 API 中記載：此函式會在什麼情況中止該程式的執行。例如，若提供給函式的參數為 NULL 指標，則函式描述文件必須說明程式是否會因此當掉。

當然，Samurai Principle 並非所有錯誤或所有應用領域皆適用。就某些未預期的使用者輸入情況而言，你不會想要讓程式當掉。然而，以程式設計錯誤來說，迅速呈現失敗而讓程式當掉，是貼切的做法。這讓程式設計師在找錯誤時輕而易舉。

不過，如此死當的做法不一定要對使用者呈現。若你的程式只是某個大型應用程式的非重要部分，則你可能還是想要讓你的程式以當掉處置。但是在整個應用程式的環境中，你的程式執行失敗時可以默默地表現，以免影響使用者或該應用程式的其餘部分。

Release 版執行檔中的 assert

當使用 assert 述句時，要提出探討的是：僅於 debug（除錯）版執行檔中運作，或在 release（釋出）版執行檔中也可運作。在你的程式碼中引入 *assert.h* 之前定義 NDEBUG 巨集，或直接在你的工具鏈（toolchain）中定義該巨集，就可以取消 assert 述句的作用。停用 release 版執行檔的 assert 述句，其主要論點是：在測試 debug 版執行檔時，你已使用 assert 找到程式設計錯誤，因此，在 release 版執行檔中，不需要因 assert 而有中止程式執行的風險。在 release 版執行檔中還保有 assert 述句的主要論點是：你無論如何都以這些述句處理無法妥善處置的嚴重錯誤，而這些錯誤絕不應該被忽視，甚至客戶使用的 release 版執行檔中也是如此。

結果

因為該錯誤在發生之處即被處理，所以不會被忽視。呼叫者不必擔負檢查這個錯誤的責任，所以呼叫者的程式碼會顯得更加簡單。然而，呼叫者此時無法對該錯誤有所反應。

在某些情況下，中止應用程式執行是可行的，理由是迅速當掉總比之後發生不可預期的行為要好。不過，你必須考量應該如何將這樣的錯誤呈現給使用者。也許使用者在螢幕上看到時會將其視為中止的陳述。然而，對於使用感應器、控制器與環境互動的嵌入式應用程式來說，你必須更加小心，要考量中止程式執行時對環境的影響，以及結果是否能夠接受。在這樣的諸多情況下，應用程式可能要更為穩健，而直接中止應用程式的運作，並非是可行的做法。

要中止程式的執行，並在錯誤發生之處立即用 Log Errors，因為不掩飾錯誤，所以更容易找到錯誤並修正錯誤。因此，就長遠而論，應用這個模式，最終會造就穩健而無誤的軟體。

已知應用

下列是此模式的應用範例：

- 有個類似的模式——Assertion Context，該模式建議增加一個除錯資訊字串至 `assert` 述句中，Adam Tornhill 在《*Patterns in C*》（Leanpub，2014）一書中有論述該模式。
- Wireshark 網路監聽工具的整個程式皆有應用此模式。例如，`register_capture_dissector` 函式使用 `assert` 檢查解析器的註冊是否重複。
- Git 專案的原始碼使用 `assert` 述句。例如，儲存 SHA1 雜湊值的函式使用 `assert`，就儲存的雜湊值，檢查其所在的檔案路徑是否正確。
- 負責處理大數字的 OpenWrt 程式使用 `assert` 述句，檢查其函式中的前置條件。
- Pekka Alho 和 Jari Rauhamäki 在〈Patterns for Light-Weight Fault Tolerance and Decoupled Design in Distributed Control Systems〉（*https://oreil.ly/x0tQW*）一文中介紹類似的模式——Let It Crash。該模式聚焦於分散式控制系統，其中建議讓單獨失效安全的程序（single fail-safe process）當掉，再迅速重啟。
- C 標準函式庫的 `strcpy` 函式不會檢查使用者輸入是否有效。若你將 `NULL` 指標供給該函式，會是當掉的結果。