跨平台 Common Lisp 程式設計 (試讀本)

Release 1.0.0

Michelle Chen

Apr 10, 2022

目錄

版權聲明 (Copyright)

免責聲明 (Disclaimer)

本書的內容 (文字、圖片、電腦程式等) 僅為一般性質的資訊，而非正式的技術文件。我們致力於保持本書的內容是即時和正確的，但我們無法保證本書內容的完整性、即時性、正確性、可靠性。本書的內容仍可能因人為錯誤、技術性問題等因素造成錯誤。

此外，我們也無法擔保本書使用者因使用本書或直接或間接受到本書的內容所致的任何損失或傷害。本書使用者應自行評估、判斷本書的內容對自己的風險。

本書使用者可以透過本書的超連結前往外部網站，但我們無法控制外部網站的性質、內容和可得性。我們在本網站中加入這些連結不代表我們推薦這些連結的內容或為這些連結的內容背書。我們無法擔保這些連結的內容。

當你使用本書時，表示你同意本書的聲明，會自行評估、判斷使用本書所導致的風險。

版本演進 (History)

- 1.0.0
  - 首次發佈

本書所使用的記述方式

本書會對 Common Lisp 程式碼加上語法高亮：

```
;; Simulate a main function.
(defun main ()
  (write-line "Hello World")
  ;; Trick for Clozure CL.
  #+ccl (finish-output)
  (quit))

;; Call the main function.
(main)
```

根據 SBCL 的慣例，使用 * 表示 SBCL 的 REPL 環境的提示符號 (prompt)。該環境回饋的部分則沒有 *：

```
* (+ 3 4)
7
```

相對來說，Clozure CL 則是使用 ? 表示其 REPL 環境的提示符號。同理，該環境回饋的部分沒有 ?：

```
? (+ 3 4)
7
```

根據 Unix 的慣例，使用 $ 表示命令列環境的提示符號：

```
$ cd path/to/project
```

若使用 root 帳號，則改以 # 表示命令列環境的提示符號：

```
# apt install sbcl
```

該情境表示上述指令需要 root 帳號才能順利執行。替代的方式是用 sudo(1)。

根據 Windows 的慣例，使用 > 表示命令列環境的提示符號。為了簡化，不列出工作目錄：

```
> cd path\to\project
```

Windows 無法從終端機畫面區分一般使用者和系統管理員。當需要系統管理員權限時會另行說明。

介紹

歷久彌新的古典語言

Lisp 是資訊界上古三大神兵 (Fortran、Lisp、COBOL) 之一，世界第二古老的高階程式語言。這個語言在長期的演進過程中出現過許多方言 (dialect)，包括本書要介紹的 Common Lisp。直到現在仍然有一些軟體專案使用 Lisp 家族語言來實作。

雖然 Lisp 是老語言，在當時實作出許多開創性的 (innovative) 程式語言概念，這些概念由後來的程式語言所吸收和利用。像是動態型態 (dynamic typing)、高階函式 (higher-order function)、遞迴 (recursion)、垃圾回收 (garbage collection)、REPL 環境等特性最早就是實作在 Lisp 上。

但早期電腦的運算資源相對受限，所以 Lisp 沒有受到廣泛的採用，只有在人工智慧 (artificial intelligence) 等少數研究領域會用到。近年來電腦的運算資源改善許多，所以 Lisp 有復甦的跡象。此外，也出現 Clojure 等新興 Lisp 方言，為 Lisp 帶來新氣象。

由於 Lisp 算是相對冷門的語言，學習 Lisp 並不是為了就學、就業等實際層面的考量。而是藉由學習 Lisp 體驗不同的範式 (paradigms)，以拓展對程式設計的視野。

Lisp 的主要方言 (Dialect)

Lisp 的方言很多，初學者會覺得眼花瞭亂。比較好的方式是先專注在其中一種 Lisp 方言上。以下是幾個主要的 Lisp 方言：

- Common Lisp：主流的 Lisp 方言，支援多種範式 (paradigms)
- Scheme：另一個主流的 Lisp 方言，以精簡的 (minimal) 特性著名於世
- Racket：基於 Scheme 的 Lisp 方言，但不完全相容於 Scheme
- Emacs Lisp：Emacs 編輯器所用的 Lisp 方言

- Clojure：運行在 Java 平台的 Lisp 方言

由於每種 Lisp 方言間不相容，建議一次先專心學一種，直到熟練後才換學另一種。雖然我們會以方言來稱呼不同的 Lisp，最好還是把不同 Lisp 方言視為不同的程式語言來學習比較好。

本書所要學習的 Lisp 方言是 Common Lisp。

Common Lisp 的歷史

Common Lisp 是一套語言標準，其目的是為了整合數個不相容的 Lisp 實作品，包括 Lisp Machine Lisp、MacLisp、InterLisp 等。所以 Common Lisp 沒有官方實作品，而是由不同開發團隊提供 Common Lisp 的實作品。

整合標準本來就是一個困難的過程。Common Lisp 的目標是提供可攜 (portable)、一致 (consistent)、有效率 (efficient)、具有表達力的 (expressive) Lisp 方言，而且儘可能和先前的 Lisp 方言相容。但不同 Common Lisp 實作品間仍有一些細微的差異。不過，只要小心地處理不相容的部分，Common Lisp 程式碼仍然是可攜的。

有關 Common Lisp 的迷思 (Myth)

由於 Common Lisp 使用者少、社群資源少、學習資源缺乏，的確會讓程式設計者不太想學。本節整理出一些對 Common Lisp 的迷思，讓我們重新看待這個語言。

Common Lisp 已經沒人在用了

現在 Common Lisp 的確算是利基語言 (niche programming language)，在幾個主要的程式語言排名皆排不進主流語言之列。但 Common Lisp 並非過時的 (obsolete) 語言，現在仍然有一些軟體專案[1]以 Common Lisp 來實作，其中不乏商業公司[2]的專案。

Common Lisp 只是電腦玩家使用的語言

網路上的確有數個免費的 Common Lisp 實作品，但也有一些商用方案：

- Clozure Associates：提供 Common Lisp 相關的開發或諮詢服務
- LispWorks：商用 Common Lisp 整合式開發環境
- Allergo Common Lisp：另一個商用 Common Lisp 整合式開發環境

[1] https://lisp-lang.org/success/
[2] https://common-lisp.net/lisp-companies

所以 Common Lisp 並不僅是技客 (geek) 或玩家 (power user) 所用的程式語言。

Common Lisp 是運行緩慢的純直譯語言

早期的 Lisp 的確是直譯語言，但 Common Lisp 中有數個實作品都可以把 Lisp 程式碼編譯成機械碼。雖然 Common Lisp 編譯器所編譯出來的機械碼不太會比 C 或 C++ 編譯器所編譯出來的機械碼快，但至少會比 Python 或 Ruby 等採用直譯的程式來得有效率。

筆者先前有撰文說明程式語言對能源運用的影響[3]，在該篇文章的原引用論文所介紹的語言中，Common Lisp *(註)* 表現算是中上的。

(註) 使用 SBCL (Steel Bank Common Lisp)。

Common Lisp 是動態型態語言，無法寫出穩固的程式

Common Lisp 的確是動態型態語言，但可藉由 declare[4]、check-type[5] 等指令在程式中加入型態宣告。在加入型態宣告後，Common Lisp 編譯器會協助程式設計者抓出程式中型態錯誤的部分，實際上的效果和靜態型態語言相差無幾。

更棒的是，由於 `declare`、`check-type` 等指令是選擇性的，Common Lisp 的型態是可動可靜的。程式設計者可依據自己的需求選擇最適合的撰碼方式。

Common Lisp 程式碼只是充斥著中括號的無意義符號

S 表達式 (s-expression) 是 Lisp 家族語言的一大特色，但也給人 Lisp 難以撰寫的感覺。有些沒寫過 Lisp 家族語言的程式設計者會以為撰寫 Lisp 程式碼時要自行配對中括號，實際上不會這樣做。

如果我們仍然像個苦行僧，使用完全沒有輔助功能的 Notepad 來撰寫程式，每種程式語言看起來都難度倍增。反之，若我們搭配合適的程式編輯器，這些編輯器會自動協助我們檢查中括號是否成對，撰寫 Lisp 就不會那麼困難。

時間就是金錢。學習用合適的工具來改善開發流程，甚至自己寫工具來改善工作效率，也是學習程式設計的一環。

[3] https://opensourcedoc.com/blog/the-greenest-programming-language/
[4] http://www.lispworks.com/documentation/HyperSpec/Body/s_declar.htm
[5] http://www.lispworks.com/documentation/HyperSpec/Body/m_check_.htm

Common Lisp 不支援中日韓等多國文字

其實很多 Common Lisp 實作品都支援統一碼 (unicode)，所以 Common Lisp 的確能處理多國文字。像是以下小程式可以在終端機正確地印出中日韓文字：

```
(write-line "你好，世界")
(write-line "こんにちは世界")
(write-line "안녕 세상")
```

Common Lisp 無法用來做圖形介面程式

實際上是可以的，常見的 binding 有 Tk、Qt、GTK、IUP、Nuklear 等。以下小程式用 GTK 3 建立簡單的視窗：

```
(ql:quickload 'cl-cffi-gtk :silent t)

(in-package :gtk)

      ;; Create a toplevel window.
(let ((window (make-instance 'gtk-window
                             :type :toplevel
                             :title "Hello World"
                             :default-width 300
                             :default-height 300)))
  ;; Signal handler for the window to handle the signal "destroy".
  (g-signal-connect window "destroy"
                    (lambda (widget)
                      (declare (ignore widget))
                      (leave-gtk-main)))
  ;; Show the window.
  (gtk-widget-show-all window)
  ;; Invoke the main event loop of a GTK program.
  (gtk-main))
```

Common Lisp 的跨平台議題

Common Lisp 的跨平台議題分為跨越多個平台和跨越多個實作品兩部分。若沒有要為 Common Lisp 寫函式庫，只要選定單一 Common Lisp 實作品即可。只要該實作品可在多個平台使用，手頭的 Common Lisp 程式碼就算跨平台了。反之，若要為 Common Lisp 寫函式庫，則要寫出能在多個 Common Lisp 實作品間有一致運算結果的 Common Lisp 程式碼。

有些網路上的 Common Lisp 程式碼針對十多種 Common Lisp 實作品撰寫。但部分 Common Lisp 實作品的使用者甚少，這樣子的程式碼實質意義不大。只要針對幾種常用的 Common

Lisp 實作品來寫撰寫程式碼即可。詳見下一節說明。

常見的 Common Lisp 實作品

Common Lisp 本身是語言標準 (language standard)。並沒有所謂的官方實作品，而是透過不同實作品來實踐該標準的特性。

以下是免費且常見的 Common Lisp 實作品：

- SBCL (Steel Bank Common Lisp)：高效率的 Common Lisp 實作品
- Clozure CL：商用軟體，但免費且開放原始碼
- Embeddable CL：Common Lisp 轉 C 的轉譯器
- ABCL (Armed Bear Common Lisp)：運行在 Java 平台的 Common Lisp 實作品
- Clasp：和 C++ 整合的 Common Lisp 實作品
- CLISP：GNU 計畫項目之一

以下是商業的 Common Lisp 實作品。商業的 Common Lisp 實作品會加上整合式開發環境：

- Allegro CL
- LispWorks

由於不同 Common Lisp 實作品間會有一些細微的差異，最好選定實作品後就持續使用，才不用一直修改程式碼。

選擇 Common Lisp 實作品

Common Lisp 的實作品眾多，要如何選擇呢？根據 Reddit 上的非正式調查 (survey)，最多人使用的 Common Lisp 實作品是 SBCL，次多的是 Clozure CL (出處[6])。

使用較多人使用的 Common Lisp 實作品主要的好處在於第三方函式庫。這些函式庫的作者在測試函式庫時，通常會優先測試較普遍的 Common Lisp 實作品。當我們使用普遍的 Common Lisp 實作品時，支援度會比較好。

有些 Common Lisp 程式設計者會同時安裝多套 Common Lisp 實作品，這樣的目的是測試 Common Lisp 程式碼的相容性。不過，除非是要寫 Common Lisp 函式庫，這個作法不是必要的。

本書的 Common Lisp 程式碼會分別以 SBCL 和 Clozure CL 測試。對於不相容的部分，則會分別列出等效的程式碼。

[6] https://docs.google.com/forms/d/e/1FAIpQLSfg7UJRKrkI3OjOHWL4xI-murE4LpQjIxsiAhFdPEmtyLX3kg/viewanalytics

基本概念

前言

由於 Lisp 家族語言和主流語言差異較大，在本文中，我們會介紹 Lisp 和 Common Lisp 的基本概念，做為撰寫 Common Lisp 程式的準備。

Common Lisp 程式碼所用的副檔名

Common Lisp 原始碼使用 *.lisp* 或 *.lsp* 為副檔名。而編譯後的位元組碼 (bytecode) 使用 *.fasl* (記為 fast load) 為副檔名。

有許多 Common Lisp 實作品可編譯出機械碼 (machine code)，即原生執行檔 (native executable)。執行檔所用的副檔名依系統而異。Unix 的執行檔不使用副檔名，而 Windows 的執行檔使用 *.exe* 為副檔名。

Hello World 程式

由於 Common Lisp 沒有嚴格規定要有主函式，所以最精簡的 Hello World 程式只需一行：

```
(write-line "Hello World")
```

因 `write-line` 是內建函式，不需引入函式庫。

但在編譯 Common Lisp 程式碼時，則需要自訂主函式。詳見本書附錄。

編譯 (Compilation) 和直譯 (Interpretation)

根據實作品的特性，我們可以把程式語言區分為編譯語言和直譯語言。前者所產生的程式會比後者來得有效率。此外，把原始碼編譯成機械碼後，無法還原成原始碼，只能恢復成等效的組語程式碼。所以，程式設計者可以藉由編譯保護原始碼。

在 Common Lisp 中，這樣的區分沒有意義。因為許多 Common Lisp 實作品兼具編譯器和直譯器的功能。視實作品的特性，可以編譯出機械碼或位元碼。

S 表達式 (S-Expression)

S 表達式是 Lisp 家族語言的特色之一。寫慣 Algol 家族語言 *(註)* 的程式設計者看到 Lisp 程式碼往往不太習慣。其實，只要花一點時寫閱讀和寫 Lisp 程式碼，就會習慣這種表達方式。現在有許多編輯器可輔助撰寫 Lisp 程式碼的任務，寫起來不會太難。

(註) 即 C 家族語言。

Lisp 的 S 表達式的虛擬碼如下：

```
(sym a b c ...)
```

S 表達式本身是列表 (list)。在列表中的第一個項目 `sym` (symbol) 代表對列表所做的指令 (command)。列表的第二個以後的項目是該指令的參數 (parameter)。參數的個數由該指令決定。

Lisp 程式碼並不完全等於 S 表達式，因 Lisp 程式碼結合 S 表達式和前綴表示法 (prefix notation) *(註)* 。我們不需要知道非 Lisp 的 S 表達式怎麼寫，只要知道 Lisp 程式碼的寫法即可。

(註) 又稱為波蘭表示法 (Polish notation)。

例如，以 Lisp 撰寫 `4 + 3` 如下：

```
(+ 4 3)
```

由於 Lisp 的 `+` 沒有限制參數數目，所以可以使用多個參數。以 Lisp 撰寫 `1 + 2 + 3 + 4 + 5` 如下：

```
(+ 1 2 3 4 5)
```

這裡可以看出 Lisp 家族語言和 Algol 家族語言的差異。

S 表達式可以嵌套。像是以 Lisp 寫 `(4 + 3) * (5 - 1)` 如下：

```
(* (+ 4 3) (- 5 1))
```

說實在的，這樣的程式碼剛開始會不太容易閱讀。只要持續寫一段時間 Lisp 程式後就能適應。

雖然 Lisp 程式碼乍看不易閱讀，但 Lisp 程式的好處是不用記憶指令的優先順序。當我們寫出 Lisp 程式碼時，指令的執行順序就已經決定了。因為 Lisp 程式碼和程式語言的抽象語法樹 (abstract syntax tree) 在概念上是相通的。

Lisp 程式的組成

Lisp 程式碼由 form 組成。此處的 form 是 Lisp 的專有名詞，代表 Lisp 程式的最小單位。Form 可能為下列三者之一：

- 資料 (data)：像是布林 (boolean)、數字 (number)、字元 (character)、字串 (string) 等
- 符號 (symbol)：即為變數 (variable)
- 列表 (conses 或 list)：Lisp 的核心資料結構。可再細分為
  - 語法 (special form)
  - 函式 (function)
  - 巨集 (macro)

Common Lisp 的核心資料結構是列表，該結構可同時用來撰寫程式或資料結構。

`3.14159` 是資料，不會轉換成其他的形式。`(+ 4 3)` 是列表。該列表包括符號 `+` 和資料 `4`、`3`。

由於 `+` 是 Common Lisp 內建的函式，不需要使用者自行宣告即能使用。

大小寫敏感性 (Case Sensitivity)

Common Lisp 的程式碼不區分大小寫。按照 Common Lisp 社群的慣例，會使用小寫，不刻意使用縮寫，以 `kebab-case` 來撰寫識別字 (identifier)。

空白 (Space)、縮進 (Indentation)、換行 (End of Line)

在 Lisp 程式中，空白用來區隔列表的元素。元素可能為符號或資料，視程式碼而定。

但 Lisp 程式不嚴格規範縮進和換行。利用縮進和換行排列程式碼的目的是讓程式碼美觀易讀。容易閱讀的程式碼也會好維護。

註解 (Comment)

註解的用途是在程式碼中加入說明文字。由於註解會被編譯器或直譯器忽略，不需以程式語言來撰寫，以自然語言 (natural language) 來撰寫即可。

Common Lisp 的註解方式如下：

- 單行註解：; 之後的單行文字
- 多行註解：一對 #| 和 |# 所包覆的文字。可跨越多行

在 Common Lisp 社群中，主流的方式是使用單行註解。只有需要跨行註解或行內註解時，才使用多行註解。

由於程式碼本身即可表達程式的行為，註解應用來說明程式設計者的意圖或想法。有時候註解會用來解釋罕見的演算法，或是標註特定的引用。對於教學用的程式碼來說，註解可用來補充說明程式的運作。

表達式 (Expression) 和敘述 (Statement)

表達式是指會回傳值的一段程式，該段程式由值、變數、運算子、函式呼叫等事物組成。而敘述代表獨立的一段指令。表達式也可以做為敘述使用，但反之則否。

Lisp 程式的 form 視為表達式，所以會自動取得值。例如，以下的 `cond` 指令會依 `a` 和 `b` 的關係回傳 `1`、`0` 或 `-1` 三者之一：

```
(defun cmp (a b)
  (cond ((> a b) 1)
        ((< a b) -1)
        (t 0)))
```

上述 Common Lisp 函式相似於以下 C 函式：

```
int cmp (int a, int b)
{
    if (a > b) {
        return 1;
    }
    else if (a < b) {
        return -1;
    }
    else {
        return 0;
    }
}
```

Lisp 的 `cond` 指令和 C 的 `if` 指令的差別在於前者是表達式，後者是敘述。

運算子 (Operator)、函式 (Function)、巨集 (Macro)

在 Algol 家族語言中，會明確地區分運算子、函式、巨集等指令。因不同指令可能會分別採用前綴表示法 (prefix notation)、中綴表示法 (infix notation) 或後綴表示法 (postfix notation) 三者之一。此外，還要考慮各種指令的優先順序。

但在 Lisp 家族語言中，運算子、函式、巨集的界限變得模糊，因為所有的指令都用前綴表示法，而且不需考慮指令的優先順序。所以，Lisp 程式乍看難寫，但寫起來反而是最單純的。

主函式 (Main Function)

許多程式語言具有主函式，做為應用程式的起始點。Common Lisp 沒有主函式的概念，程式碼直接寫在命令稿頂層。

但筆者會建議在每個 Common Lisp 應用程式中加入自訂的主函式，並將主程式實作在主函式中。當我們把 Common Lisp 命令稿編譯成執行檔時，就會有設置主函式的項目，代表 Common Lisp 認定程式中存在著主函式。

以下是最精簡的範例程式：

```
;; Simulate a main function.
(defun main ()
  ;; Implement your code here.

  ;; Run the script in batch mode.
  (quit))

;; Run the main function.
(main)
```

defun[7] (記為 define function) 巨集用來宣告函式。本範例程式將主函式命名為 `main`。由於本範例程式僅是用來展示主函式的架構，其內部是空的 (dummy)。

接著，執行 `main` 指令，即呼叫主函式。在主函式的尾端會執行 `quit` 指令，以離開此程式。

Common Lisp 沒有規範主函式的名稱，筆者建議一律使用 `main` 來命名主函式。由於 `main` 恰巧是多種程式語言的主函式使用的名稱，易於閱讀和撰寫。

[7] http://www.lispworks.com/documentation/HyperSpec/Body/m_defun.htm

離開狀態 (Exit Status)

離開狀態是程式終止時回傳給系統的代碼，該代碼為整數型態資料。

在程式設計的慣例中，回傳 `0` 代表程式正常結束，回傳 `1` 或其他非零數字代表程式異常結束。除了 `0` 和 `1` 以外，程式設計者對回傳值沒有共識。所以，不要設計複雜的回傳值，因為這類設計無法跨平台。

由於 Common Lisp 實作品間用來回傳離開狀態的指令不一致，我們把這個議題留在可攜性的章節來處理。

將 Common Lisp 命令稿編譯成執行檔 (Executable)

Common Lisp 實作品可以編譯 Common Lisp 原始碼。但 Common Lisp 標準沒有明確規範編譯的目標格式，把這個部分留給各個實作品來決定。在常見的 Common Lisp 實作品中，編譯的目標可能為

- *fasl* (fast load) 檔，為一種位元碼
- 直接轉成機械碼
- 先轉譯為 C 程式碼，再由 C 編譯器編譯成機械碼

Common Lisp 無法直接在命令列環境輸入指令來編譯程式碼，必需要在命令稿中額外寫出指令才能編譯。但不同 Common Lisp 實作品用於編譯程式碼的指令不一致。我們在可攜性的章節會說明如何處理此議題。

附記

本章所介紹的工具函式收錄在 cl-portable[8] 和 cl-yautils[9] 專案。

[8] https://github.com/cwchentw/cl-portable
[9] https://github.com/cwchentw/cl-yautils