COMSM1302 電腦架構概論
第十週完整教材：函式呼叫與堆積記憶體

擴充 Hack VM · 呼叫框架 · 函式的實作 · malloc 與 free

依據 John Lapinskas（University of Bristol）課程講義編寫並延伸

2026 年 7 月

本教材的使用方式：本講義整合了第十週四份投影片（10-1 Extending Hack VM、10-2 Functions in general、10-3 Functions in Hack VM: Syntax and implementation、10-4 Heap memory allocation: malloc and free）的全部內容，並補充了 x86-64 呼叫慣例、真實世界 malloc（dlmalloc）、記憶體安全等延伸知識。每章結尾附有「本章重點」整理；第 5 章為綜合練習題，附完整詳解。上週我們學了 Hack VM 的堆疊運算、分支與記憶體段；本週補上最後一塊拼圖：**函式呼叫**與**執行期記憶體配置**。

目錄

1　擴充 Hack VM（Extending Hack VM）

1.1　本週目標

本週我們要擴充 Hack VM，加入：

- **函式呼叫**；
- 正式的**編譯期記憶體配置**（與函式呼叫同時解決！）；
- **多檔案編譯**以支援函式庫（相較之下容易）。

我們也會討論**執行期記憶體配置**（即 `malloc`）。下週就進入高階語言——Jack。

1.2　函式應有的行為：案例研究

回想 Programming in C 那個（糟糕的！）遞迴費波那契演算法：

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2},$$

$$\text{fib}(n) = \begin{cases} 0 & \text{若} n = 0, \\ 1 & \text{若} n = 1, \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{若} n \geq 2. \end{cases}$$

投影片的實作加了兩個 static 變數：`times_called`（總呼叫次數）與 `layers_deep`（目前遞迴深度），用來觀察函式呼叫的行為。

期望行為總結

每次**呼叫** `fibonacci` 時：

- 程式流程跳到函式開頭；
- 區域變數 `x`、`y` 被**清空**；
- 引數變數 `n` 由該次呼叫**設定**；
- static 變數 `times_called`、`layers_deep` **不變**。

每次**返回**時：

- 程式流程回到原呼叫的**下一行**；
- 區域變數 `x`、`y` **恢復舊值**；
- 引數變數 `n` 恢復舊值；
- static 變數不變。

這一切必須對**任意深度**的巢狀呼叫（記憶體允許的範圍內）都穩健成立——**包括遞迴呼叫**。

1.3　多檔案編譯

VM 翻譯器收到一個**資料夾**時，應該：

- 把資料夾內所有 `.vm` 檔翻譯成**一個**組語檔；
- 所有 VM 程式碼都必須在**函式內**；
- 編譯出的程式先照常把 `SP` 設為 256，然後呼叫 `Sys.vm` 裡的函式 `Sys.init` ——類比 C 的 `main`（`Sys.init` 由測試碼提供，因此各檔案的翻譯**順序無關緊要**）；
- 檔案 `abc.vm` 內所有函式的名稱必須以「`abc.`」開頭（如 `abc.print`）——防止檔案間名稱衝突；Jack 編譯器會強制執行：檔案 `abc` 中名為 `xyz` 的 Jack 函式編譯成 VM 函式 `abc.xyz`；
- 不同檔案中相同的 static 位址或標籤必須編譯到 **不同**位址——你的 VM 翻譯器已經做到了（`Foo.vm` 的 `static 5` → 組語變數 `Foo.5`，上週的伏筆在此回收!）。

1.4　Jack 與「作業系統」

Jack 附帶 nand2tetris 樂觀地稱為「作業系統」的東西——實際上是**八個用 Jack 寫的標準函式庫**。（平心而論，OS 的核心就是這個：在此基礎上擴充 Sys 函式庫、把行程視為函式呼叫，就能蓋出 DOS 那樣的單行程 OS。）

函式庫	提供內容
`Sys.vm`	`Sys.init`；停機、當機、等待毫秒數
`Memory.vm`	記憶體配置（見第 4 章!）
`Array.vm`	陣列資料型別
`String.vm`	字串資料型別
`Keyboard.vm`、`Screen.vm`	直接輸入與輸出
`Output.vm`	文字顯示與編輯
`Math.vm`	乘法、除法、最小值、最大值、平方根

下週寫 Jack 編譯器時會拿到這些函式庫的 Hack VM 版本。細節見 Nisan and Schocken 附錄 6。**皆不列入考試!**

1.5　自舉（Bootstrapping）

nand2tetris 中這些函式庫其實都是用 Jack 寫的。用 Jack 程式碼寫 Jack 編譯器是「作弊」嗎? **不是!** 讓編譯器能編譯自己叫做**自舉**，標準做法是：

1. 寫一個**暫代**的 Jack-to-Hack 編譯器 A——直接用 Hack VM ／組語寫，或用完全不同的系統寫（例如在 x86-64 上用 C 寫，即**交叉編譯**）；
2. 用 Jack 本身寫一個**品質更好**的編譯器 B；
3. 用 A 把 B 編譯成 Hack 機器碼；

4. 用 B **重新編譯** B 自己；

5. 把 A 扔掉，從此使用 B。

即使不交叉編譯，目標也是**盡量少寫低階程式碼**。「OS」函式大多直截了當，除了 `Memory.vm` 沒有新的架構概念——但想練組語／ VM 的話它們是好習題！

本章重點

- 函式呼叫需要：跳轉 + 清空 local + 設定 argument + static 不動；返回需要：跳回 + 恢復一切；遞迴也要成立。
- 多檔案編譯：全部翻成一個組語檔、程式碼都在函式內、從 `Sys.init` 開始、函式名帶檔名前綴、static 各檔案分開。
- 自舉：暫代編譯器 A → 用新語言寫更好的 B → A 編 B → B 自編 → 丟掉 A。

2　一般的函式實作：呼叫框架（Frames）

本章討論**任何**語言（例如 C）如何實作函式，下一章再回到 Hack VM 的具體做法。

2.1　四個子目標

實作函式的檢查清單

- **目標 1：程式流程**。呼叫時跳到函式開頭；返回時跳回來。
- **目標 2：記憶體配置**。呼叫時為新的 local ／ argument 變數配置記憶體；返回時釋放。
- **目標 3：程式狀態**。呼叫時把既有的 local ／ argument 變數與多數暫存器值**擱置**、換上新的；返回時**原封不動**地取回。
- **目標 4**：static 變數完全不受呼叫與返回影響。

2.2　堆疊無所不在

在任何語言、（幾乎）任何架構上，達成這些目標的最佳方式都用**堆疊**。Hack 的特殊之處在於堆疊只存在於 VM 層、不存在於組語層。這很不尋常！ARM 與 x86-64 都有：

- 擔任堆疊指標 `SP` 角色的暫存器（如 `ESP`）；
- `push` ／ `pop` **組語指令**；
- `call`、`ret`、`enter`、`leave` 指令，直接完成本章討論的大部分工作。

所以這些操作在 Hack 組語又慢又笨重（一句 `push local 5` 展開成十多條指令），在現代 CPU 上卻非常快。（MIPS 沒有原生 push/pop，但有 `jal` 與 `jr` 指令分擔函式呼叫的大部分負擔。）

2.3　目標 1：程式流程

假設我們有堆疊可用。

- **呼叫時**：把**返回位址**推上堆疊，然後跳到函式開頭；
- **返回時**：從堆疊**彈出**返回位址，跳過去。

要跳去哪裡的位址怎麼知道？**交給組譯器！**標籤是任何組語都有的功能，這問題它們早就解決了。

程式流程範例：`main` → `foo` → `bar` → `baz`

組語骨架（示意）：每個呼叫點宣告一個返回標籤並推上堆疊，再跳到函式標籤；每個函式結尾彈出返回位址並跳過去。

```
// main 的程式碼
@label0
[把位址 (label0) 推上堆疊]
```

```
[跳到 (foo)]
(label0)
// 停機

(foo)
@label1
[把位址 (label1) 推上堆疊]
[跳到 (bar)]
(label1)
// 更多程式碼
[彈出返回位址並跳過去]

(bar) ... (baz) ...   // 同樣模式
```

呼叫到最深處（baz 內）時，堆疊由底至頂是 (label0)、(label1)、(label2)——**每一層的返回位址按呼叫順序疊著**；每次返回就彈掉最上面那個，剛好回到正確的地方。堆疊的 LIFO 性質與「最後呼叫的函式最先返回」完美對應——這就是為什麼函式呼叫非用堆疊不可。

2.4　目標 2：記憶體配置

在 C 中，**每一個變數的大小都能在編譯期算出**。看起來長度在執行期才決定的東西（字串、陣列）其實是 **指標**——永遠 64 位元長。它們指向的記憶體要嘛編譯期就固定大小（`char *out = "Hello world!";`、`int myArray[100];`），要嘛由程式設計師用 `malloc` ／ `free` 明確配置。

因此，一個在語意分析階段（即解析之後）為函式的區域變數建符號表的 C 編譯器，會**提前確切知道**每次呼叫該函式要配多少空間。引數同理。所以編譯器需要：

- 每次函式被呼叫時，為**已知數量、已知大小**的 local ／ argument 變數找空間；
- 實作 `malloc` 與 `free`——這種記憶體與函式呼叫無關，不需額外處理（見第 4 章!）。

2.5　合併目標 2 與 3：用堆疊配置一切

假設呼叫函式 f 需要 10 個字的區域變數、7 個字的引數、5 個字的狀態（含相關暫存器與返回位址），這些資訊存在符號表裡。

呼叫、執行、返回三部曲

呼叫時：

- 把目前的堆疊指標存起來，稱為 `OSP` （Old Stack Pointer）；
- `SP` 加 10，為區域變數留空間；
- 把程式狀態與引數推上堆疊（`SP` 再加 12）；
- 跳到函式標籤。

執行中：（假設引數放最底、然後區域變數、最後程式狀態）

- 第 i 個引數 $\to RAM[\texttt{OSP}+i]$；
- 第 i 個區域變數 $\to RAM[\texttt{OSP}+7+i]$；
- 每個變數的位移存在符號表（順便解決非一字長的變數）。

返回時：

- （可選）存一個返回值；
- 把 `SP` 重設回 `OSP`——**一條指令就釋放了**舊的區域變數、引數與程式狀態；
- 把舊的程式狀態複製回暫存器；
- 跳到返回位址（在堆疊上）；
- （可選）處理返回值。

投影片以 `main` → `threemin` → `min` 的 C 程式為例（求三數最小值）：編譯器為每個函式存一張符號表（`threemin` 的表：引數 `a,b,c` 位移 0–2、區域變數 `x,y,z` 位移 3–5），執行時堆疊隨呼叫長高（`main` 的 `x,y,z` 在底部，每次呼叫疊上「引數＋舊狀態」），隨返回縮回——`SP` 與 `OSP` 一路記錄了「目前這層」的範圍。注意這只是 C 函式呼叫的**一種可能**實作；實際的編譯器會為了最佳化把堆疊排得稍有不同（甚至把整個函式呼叫最佳化掉）。

2.6　術語

呼叫框架與工作堆疊

- 需要推上堆疊的程式狀態（返回位址、舊 `OSP`、舊暫存器值）稱為呼叫者的**呼叫框架（call frame）**或**框架（frame）**——上面例子裡的「舊狀態」。
- 函式內部仍可用堆疊**最頂端**做算術運算的工作空間，這個子堆疊稱為**工作堆疊（working stack）**；整個堆疊（含所有歷史呼叫框架）稱為 **全域堆疊（global stack）**。函式呼叫時工作堆疊會連同舊狀態一起被保留（但它**不屬於**呼叫框架）。
- `malloc` 配置的記憶體稱為配置在 **堆積（heap）** 上，以區別於堆疊。驚人的是這跟 heap 資料結構**毫無關係**！只是個沒有更深含義的慣用語。

2.7　目標 4：static 變數

這個簡單——給每個 static 變數一塊**獨立於堆疊**的記憶體（Hack VM 就是 `static` 段），用符號表記錄對應關係，然後在呼叫／返回流程中**別去動它**。同理，堆疊上的工作儲存（算術暫存值）也不需特別處理——呼叫時自然保留、返回時自然恢復。

非 C 語言表面上常模糊堆疊與堆積的界線（允許變長陣列、自動釋放記憶體），但底層運作大致與 C 相同——一些變數放堆疊、另一些放堆積，並呼叫 `malloc` ／ `free` 的類似物。

延伸補充：x86-64 的真實呼叫慣例（System V ABI）。現代 x86-64 上這套機制長這樣：堆疊由高位址往**低**位址長；`call` 指令自動把返回位址推上堆疊、`ret` 自動彈出跳回。前六個整數引數不走堆疊、直接放暫存器（`rdi, rsi, rdx, rcx, r8, r9`），多的才推堆疊。函式開頭的**序言（prologue）**保存呼叫者的框架指標 `rbp` 並建立自己的框架；結尾的**尾聲（epilogue）**恢復並返回——正是本章「呼叫三部曲」的硬體加速版。最佳化編譯器常省略 `rbp`（改用 `rsp` 定位，省兩條指令、多一個通用暫存器），葉函式還能直接使用 `rsp` 下方 128 位元組的 **紅區（red zone）**而完全不動堆疊指標。對照 Hack：概念一模一樣，差別只在硬體支援的多寡。

本章重點

- 四目標：程式流程、記憶體配置、程式狀態、static 不動。
- 返回位址推上堆疊 ⇒ LIFO 恰好匹配「後呼叫先返回」；位址交給組譯器的標籤機制。
- C 的變數大小全部編譯期可知（動態的東西是指標）；所以每次呼叫配多少空間是常數。
- 呼叫框架 = 返回位址 + 舊 OSP + 舊暫存器；變數存取 = OSP + 符號表位移；返回 = `SP` 拉回 `OSP` 一次全釋放。
- 堆積與 heap 資料結構無關；static 放獨立區域即可。

3 Hack VM 的函式：語法與實作

3.1 函式語法

上一章重度依賴語意分析階段建立的符號表。**Hack VM 的語法設計完全避開了符號表的需要！**

函式語法

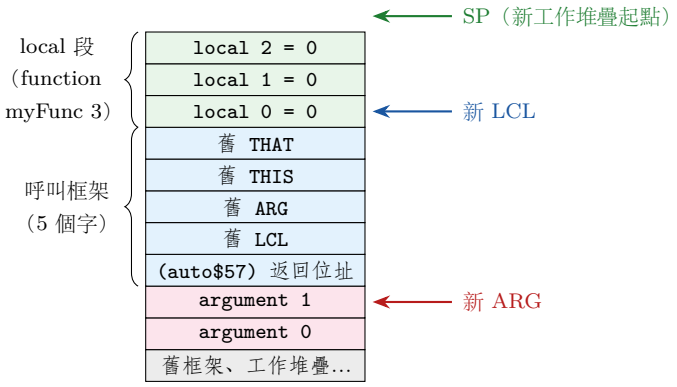
- `function name x`：定義函式。`name` 是函式名，`x` 是該函式 `local` 段的 **大小**（通常 = 用到的區域變數個數）。函式定義的結尾不用大括號或縮排，而是 **下一個 `function` 指令或檔案結尾**（所以函式外執行的程式碼必須放在檔案最上方）。
- `call name x`：呼叫函式。`x` 是引數個數。這會**彈出**堆疊頂的 `x` 個值（當作引數）、呼叫函式、並把返回值**推上**堆疊。
- `return`：返回，回傳值 = 堆疊頂的值。

從呼叫者的角度看，`call` 就像一個「吃掉 x 個值、吐回一個值」的超級運算子——與 `add` 吃 2 吐 1 同構。這讓函式呼叫能無縫嵌進運算式編譯（RPN）中。

3.2 實作 `call`：搭建新框架

假設 VM 翻譯器看到 `call myFunc 2`（呼叫前，堆疊頂已有兩個引數值）。產生的組語必須依序：

1. 產生一個**返回標籤**（例如 `auto$57`，不可與其他標籤衝突），把該位址推上堆疊，供之後的 `return` 跳回；
2. 把 `LCL`、`ARG`、`THIS`、`THAT` 推上堆疊，**保存**目前的值；
3. 把 `ARG` 設到（舊）堆疊頂往下數 2 個值的位置——那兩個引數**原地變成**新函式的 `argument` 段；
4. 把 `LCL` 設為新的堆疊頂——這將是新函式 `local` 段的起點；
5. 跳到函式標籤（由 `function` 定義處產生）。



`call myFunc 2` + `function myFunc 3` 完成後的堆疊（往上長）

注意步驟 1 推返回位址時**還不必更新 SP**（之後一起做更容易）。

3.3　實作 `function`：接手

VM 翻譯器看到 `function myFunc 3` 時，產生的組語必須：

1. 以一個**標籤**開頭，每次呼叫都跳到這裡。為免使用符號表，這個標籤必須**只從函式名就能導出**（這樣 `call` 端也能生成同一個標籤）；
2. 把 `SP` 設為 `LCL` + 3（local 段長度來自 `function` 指令）；
3. 把 `local 0`、`local 1`、`local 2` **初始化為零**；
4. 繼續執行函式本體的第一行。

為什麼要清零？（投影片註腳）這是 Hack VM 規格的一部分。官方沒解釋原因，但講師推測是**安全性**：即使在 DOS 這種單行程 OS，不同函式呼叫也可能屬於不同「行程」，防止它們看見彼此**殘留的堆疊記憶體**是合理的。它們本來就無法透過 `this` ／ `that` 看到——VM 模擬器只允許 `this` 用於堆積記憶體、`that` 用於堆積、SCREEN 與 KBD。（對照現實：C 不清零區域變數，未初始化就讀是未定義行為——這是無數真實漏洞的來源。）

3.4　實作 `return`：拆除框架

函式執行到 `return` 時，堆疊頂是返回值，下面可能還有工作堆疊殘值、local 段、呼叫框架、argument 段。產生的組語必須：

1. 把**返回位址**暫存起來（例如存到 `R13`）——它在 $RAM[\texttt{LCL} - 5]$；
2. 把**返回值**複製到新工作堆疊的位置，即目前 `ARG` 指的位址——**覆寫 argument 0，返回值恰好出現在呼叫者期望的堆疊頂**；
3. 把 `SP` 設為 `ARG` + 1（新工作堆疊的頂端再上一格）；

4. 從目前 LCL **往下數**，依序恢復舊的 THAT（LCL − 1）、THIS（LCL − 2）、ARG（LCL − 3）、LCL（LCL − 4）；

5. 跳到返回位址——SP 以上的一切等同被丟棄。

妙處：這段程式碼**從頭到尾不需要知道**呼叫的是哪個函式、從哪裡呼叫的！一切資訊都藏在堆疊上相對於 LCL、ARG 的固定位置。這正是遞迴「免費」成立的原因——每一層呼叫的框架結構完全相同，機器只是機械地搭建與拆除。

為什麼返回位址在 LCL − 5？ 看 3.2 節的堆疊圖：call 的推入順序是返回位址、舊 LCL、舊 ARG、舊 THIS、舊 THAT，而 LCL 指向框架結束後的第一格（local 0）。所以從 LCL 往下數：−1 是舊 THAT、−2 舊 THIS、−3 舊 ARG、−4 舊 LCL、−5 就是返回位址。必須**先**把返回位址存到 R13 再覆寫 argument 0，因為當函式**沒有引數**時（call f 0），argument 0 與返回位址的距離只有 5 格，而返回值一寫入 ARG、SP 一縮，框架區隨時可能被後續操作覆寫。

3.5　初始化

程式一開始 LCL 和 ARG 怎麼設？記住多檔案 Hack VM 程式以呼叫 Sys.init 開始——呼叫**之前**它們的值根本不重要，呼叫**之後**它們就照一般機制被正確設定了。官方 Sys.init 的預設行為：呼叫其他函式庫的初始化函式 → 呼叫 Main.main （編譯自 Jack 時它就是 C 的 main 的對應物）→ 進入無窮迴圈。

本章重點

- 語法：function name k（local 段大小 k）、call name n（吃 n 個引數、吐一個返回值）、return（回傳堆疊頂）。
- call：推返回位址 → 推 LCL/ARG/THIS/THAT → ARG = 引數起點 → LCL = 堆疊頂 → 跳。
- function：標籤（從函式名導出）→ SP = LCL + k → local 清零。
- return：R13 = RAM[LCL − 5] → 返回值寫入 ARG → SP = ARG+1 → 從 LCL 往下恢復四個暫存器 → 跳 R13。
- 整套機制與「是哪個函式」無關 ⇒ 遞迴自動成立。
- 開機流程：SP=256 → call Sys.init → Main.main → 無窮迴圈。

4　堆積記憶體配置：malloc 與 free

4.1　兩個關鍵函式

堆疊配置威力強大，但只適用於**編譯期已知大小**的變數。要以 C 的方式處理**執行期**記憶體配置，需要寫兩個函式（C 叫 `malloc` ／ `free`，Jack 叫 `Memory.alloc` ／ `Memory.deAlloc`）：

本章的規格

- `int alloc(int size)`：在堆積記憶體（位址 `0x800`–`0x3FFF`）配置 `size` 個字的記憶體段並回傳其基底位址；配不出來回傳 -1。
- `void deAlloc(int base)`：釋放基底位址為 `base` 的段（`base` 須為 `alloc` 回傳過的值）。

正確性要求：`base = alloc(size)` 之後，程式可任意寫入 $RAM[\texttt{base}]$ 到 $RAM[\texttt{base}+\texttt{size}-1]$；所以後續的 `alloc(size2)` 回傳的區間**不得**與之相交——至少直到 `deAlloc(base)` 被呼叫為止。程式若寫出界（$RAM[\texttt{base}-1]$、$RAM[\texttt{base}+\texttt{size}]$）或釋放後再寫，我們**不管**（現代系統中 OS 會回以 segfault）。

以下用高階語言的視角逐步檢視**四個漸進改良的演算法**，著重演算法本身而非程式碼。

4.2　嘗試 1：只進不出（bump allocator）

- $RAM[\texttt{0x800}]$ 存一個指標，指向**第一個未配置**的位址；
- `alloc(size)`：回傳 $RAM[\texttt{0x800}]$，然後把它加上 `size`；
- `deAlloc`：**什麼都不做**。

這是 Nisan and Schocken 12.1.3 的「basic」演算法。問題顯而易見：記憶體**永遠不會回收**——長時間執行的程式遲早耗盡堆積。但它極快（一次加法），實務上真的用於生命週期短暫的場合（如編譯器單趟處理的 arena allocator）。

4.3　嘗試 2：把資訊存進段裡（單向自由串列）

我們顯然需要記錄「有哪些自由／已用的段、多長」。但那是一個大小未知的串列……這不正是我們要解決的問題嗎？**點子：把這些資訊存在段自己裡面！**

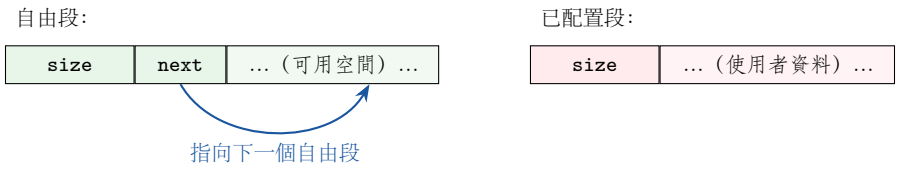
嘗試 2 的資料結構

- 每個段的**第一個字**存它的**可用大小**；
- **自由段**串成**單向連結串列**：指標存在每個自由段的**第二個字**；$RAM[\texttt{0x800}]$ 存串列第一個段的指標；
- 注意 [可用大小] = [實際大小] -1——我們損失一點空間當**管理負擔（overhead）**。

`alloc(size)`：

- 沿自由串列找一個**夠大**的段 S（first-fit）；
- 把 S 移出串列、把 S 的長度改成 `size`；
- 在 S 尾端**切出**一個新自由段加入串列（除非 S 的可用大小恰好等於 `size`）；
- 回傳**可用區**的基底位址（即跳過放大小的那格）。

`deAlloc(base)`：把新自由段插入串列**開頭**（指標寫進 RAM[`base`]）。



投影片追蹤了一長串 `alloc` ／ `deAlloc`：從一整塊 `0x37FE` 字的自由段開始，三次 `alloc(0xDFF)` 依序切出三段，釋放再配置時 first-fit 會重用夠大的自由段並再切割。一切運作正常……直到出現這個場景：

致命問題：相鄰的兩個自由段**永遠不會合併**。釋放兩個相鄰的 `0x6FF` 字段之後，串列裡有兩個相鄰的小自由段——但 `alloc(0xE00)` 仍然**失敗**，儘管總自由空間綽綽有餘！段一旦被切小，就永遠回不去。

4.4　嘗試 3：釋放時合併（coalescing）

我們想在釋放時把段與**記憶體上相鄰**的自由段合併（這個過程叫**合併／聚結（coalescing）**）。但自由串列**不是照位址排序**的，順著串列找鄰居太慢——需要存更多資訊讓它變快。

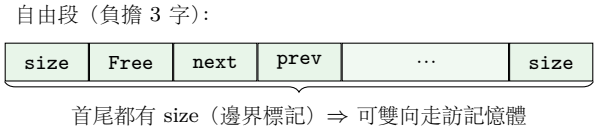
嘗試 3 的資料結構

- 每個段的**第二個字**存它的**自由／已用狀態**；
- 每個段的**第一個字與最後一個字**都存可用大小（這讓我們能**沿記憶體順序**快速走訪所有段——往後跳 size、往前讀前一段尾端的 size，這對字稱為**邊界標記，boundary tags**）；
- 自由段串成**雙向連結串列**：指標存在每個自由段的第三、四個字（這讓我們能 $O(1)$ **刪除任意段**）；RAM[`0x800`] 指向串列開頭；
- 現在 [可用大小] = [實際大小] -3。

`alloc(size)`：跟之前一樣（first-fit、切割），另外把段的狀態改成**已用**。
`deAlloc(base)`：

- 把 `base` 的狀態設為**自由**；
- 檢查記憶體中**緊鄰其前**的段：若自由，與之合併並更新大小；
- 檢查記憶體中**緊鄰其後**的段：若自由，與之合併、更新大小、並把它從自由串列中刪除；

- 若兩者皆非自由，把 base 插入自由串列。



現在釋放兩個相鄰段時它們自動合併回大段，嘗試 2 的問題解決了。

4.5　碎片化（Fragmentation）

即便有合併，**錯誤順序**的 alloc ／ deAlloc 仍會出問題：配置一大堆小段、再**隨機釋放其中一半**，就會得到「自由、已用、自由、已用……」相間的棋盤格局——**一半的記憶體是自由的，卻連稍大一點的段都配不出來！**這個問題叫**碎片化**，同時影響記憶體與檔案系統。

某種意義上有個簡單解法：把每個已用段按位址遞增順序 deAlloc 再重新 alloc，順便把資料搬到新位置。用演算法 3 或 4，最後自由空間必然合併成一段。這叫**重組（defragmentation）**，慢一點但保證有效。

嚴重問題：我們**不能**在 alloc ／ deAlloc 內部做重組。記住每次 alloc 只是回傳一個**指標**——我們不知道呼叫端把指標拿去做了什麼、改掉它會發生什麼事。身為程式設計師只能自己面對——不只 Hack，**C 也一樣！**這是 Java（下學期）這類語言的一大優勢：它們不給程式設計師裸指標，改用**參考（reference）**——行為相似但多一層間接、不含真實記憶體位址，所以執行環境可以在背後安全地搬移物件（壓縮式垃圾回收）。

alloc 內部能做的，是靠**慎選回傳值**來延緩碎片化：

- **first-fit**（目前用的）：回傳**第一個**夠大的自由段。快，但容易碎片化。
- **best-fit**：掃過**整條**自由串列，回傳大小**最接近**需求的段。碎片化較少，但非常慢。

能不能讓 best-fit（或接近它的東西）變快?

4.6　嘗試 4：分箱（Bins）

不存一條自由段的雙向串列，而是存**十條**，稱為**箱**！

嘗試 4 的資料結構

- RAM[0x800]：指向第一個長度 ≤ 0x1C 字的自由段；
- RAM[0x801]：長度 0x1D–0x38；
- RAM[0x802]：長度 0x39–0x70；……（每箱容量上限翻倍）
- RAM[0x809]：長度 0x1C01–0x3800。

alloc ／ deAlloc 幾乎照舊，僅兩點不同：

- `alloc(size)` 找自由段時，從**會包含長度 `size`** 的那個箱開始掃；找不到就掃包含 $2\times$`size` 的箱，依此類推（另一種變體：直接從 $2\times$`size` 的箱開始掃、原箱當最後手段——碎片化多一點但平均快很多）；
- 合併／切割段時，要檢查結果該進**哪個箱**。

想更講究，可以把雙向串列換成**平衡二元搜尋樹**（如 2-3-4 樹），快速選出 best-fit 的段！（現代的 malloc 實作確實會把箱排序——值得這個負擔……）

延伸補充：真實世界的 malloc 與記憶體安全。 Doug Lea 的 **dlmalloc**（glibc `malloc` 的前身）就是本章路線的工業級版本：邊界標記 + **128 個**按大小排序的箱 + 立即合併；小段另設 LIFO 的 fastbins 以加速高頻小配置。本章的規格也解釋了 C 最惡名昭彰的幾類 bug：寫出界（`RAM[base+size]`）會踩壞**下一段的邊界標記**——這就是**堆積緩衝區溢位**；釋放後再用指標是 **use-after-free**；同一段釋放兩次（double free）會把自由串列接成環。攻擊者正是靠覆寫 size ／ next ／ prev 這些管理欄位奪取程式控制權——理解本章，你就理解了這整類漏洞的原理。

本章重點

- 規格：`alloc(size)` 回傳不相交段的基底（失敗 -1）、`deAlloc(base)` 釋放；出界與釋放後使用不設防。
- 嘗試 1：指標一直往前推、不回收——快但耗盡即死。
- 嘗試 2：大小存段首、自由段單向串列、first-fit + 切割——相鄰自由段永不合併。
- 嘗試 3：狀態字 + 首尾 size（邊界標記）+ 雙向串列 $\Rightarrow$ 釋放時 $O(1)$ 合併前後鄰居。
- 碎片化：自由空間夠但不連續；重組有效但不能在 alloc/deAlloc 內做（指標已交出去）。
- first-fit 快而碎、best-fit 省而慢；嘗試 4 用分箱逼近 best-fit 的品質與 first-fit 的速度。

5　綜合練習題（附詳解）

練習 1：呼叫時的堆疊

執行 `call f 2` 前，$SP = 310$（堆疊頂兩個值是引數）。(a) `call` 的組語做完五個步驟後，SP、ARG、LCL 各是多少？(b) 接著執行 `function f 4` 之後 SP 是多少？

解答

(a) `call` 推了 5 個字（返回位址＋舊 LCL/ARG/THIS/THAT），所以 $SP = 310 + 5 = \mathbf{315}$。引數在舊堆疊頂往下 2 格：$ARG = 310 - 2 = \mathbf{308}$。`LCL` 設為新堆疊頂：$LCL = \mathbf{315}$。（通式：$ARG = SP_{舊} - n$、$LCL = SP_{舊} + 5$。）

(b) `function f 4` 把 SP 設為 $LCL + 4 = 315 + 4 = \mathbf{319}$，並把 $RAM[315..318]$（local 0–3）清零。

練習 2：返回時的堆疊

延續練習 1：函式執行完，堆疊頂（$RAM[SP-1]$，設 $SP = 320$）是返回值 99。寫出 `return` 的每一步對 SP、ARG、LCL 與相關 RAM 的影響。

解答

執行前：$LCL = 315$、$ARG = 308$。

1. 返回位址在 $RAM[LCL - 5] = RAM[310]$，存入 `R13`；
2. 返回值 99 寫入 $RAM[ARG] = RAM[308]$——覆寫 argument 0；
3. $SP = ARG + 1 = \mathbf{309}$——呼叫者看到的效果：兩個引數被吃掉、換成一個返回值；
4. 恢復：$THAT \leftarrow RAM[314]$、$THIS \leftarrow RAM[313]$、$ARG \leftarrow RAM[312]$、$LCL \leftarrow RAM[311]$（即 $LCL - 1$ 到 $LCL - 4$）；
5. 跳到 `R13` 中的返回位址。

淨效果與 `add` 彈 2 推 1 一樣的「介面」：呼叫前堆疊頂 2 個引數，呼叫後堆疊頂 1 個返回值。

練習 3：寫一個 VM 函式

Hack VM 沒有乘法指令。寫函式 `Math.mult`，接收兩個引數 a、b（$b \geq 0$），用重複加法回傳 $a \times b$。提示：local 0 當累加器、local 1 當計數器。

解答

```
function Math.mult 2      // local 0 = 累加, local 1 = 計數
label LOOP
push local 1
push argument 1
```

```
lt                      // 計數 < b ?
not
if-goto END             // 否則離開
push local 0
push argument 0
add
pop local 0             // 累加 += a
push local 1
push constant 1
add
pop local 1             // 計數++
goto LOOP
label END
push local 0            // 返回值上堆疊
return
```

呼叫方式：`push` 兩個值後 `call Math.mult 2`，返回後堆疊頂就是乘積。注意 local 不必手動清零——`function` 的實作保證了初值 0。

練習 4：遞迴為什麼可行

(a) `return` 的組語裡，哪些資訊讓它不需要知道「我是誰、誰呼叫我」？(b) 遞迴呼叫 `fib(4)` 時（`fib` 定義同 1.2 節），最深時全域堆疊上同時有幾個 `fib` 的呼叫框架？

解答

(a) 一切都以 `LCL` 與 `ARG` 為**相對**基準：返回位址固定在 $LCL-5$、舊暫存器在 $LCL-1$ 到 $LCL-4$、返回值該放的位置就是 `ARG`。每層呼叫的框架**結構**相同、只是位置不同，而位置由暫存器攜帶——所以同一段 `return` 碼對任何呼叫、任何深度都正確。

(b) 呼叫樹：$\text{fib}(4) \to \text{fib}(3) \to \text{fib}(2) \to \text{fib}(1)$。最深路徑上同時活著的框架是 $\text{fib}(4), \text{fib}(3), \text{fib}(2), \text{fib}(1)$ ——**4 個**。（$\text{fib}(3)$ 的第二個子呼叫 $\text{fib}(1)$ 發生時，$\text{fib}(2)$ 那支已經返回、框架已拆除——堆疊深度 = 呼叫樹的**深度**，不是節點總數。）

練習 5：多檔案規則

(a) 為什麼 `Foo.vm` 與 `Bar.vm` 裡的 `static 0` 必須映射到不同 RAM 位址？(b) 為什麼所有函式名必須帶檔名前綴？(c) 為什麼從 `Sys.init` 開始執行讓「檔案翻譯順序無關」？

解答

(a) 兩個檔案彼此**獨立編寫**（可能來自不同函式庫作者），各自的 `static 0` 是各自的「全域變數」；若共用位址，一個檔案會默默改壞另一個檔案的資料。翻譯成組語變數 `Foo.0` 與 `Bar.0` 讓組譯器自動分配不同位址。

(b) 同理防止**函式**名衝突：兩個檔案都想定義 `print` 時，強制前綴使它們成為 `Foo.print` 與 `Bar.print`——組語層的標籤因此不撞名。
(c) 因為**唯**一的進入點是開頭那句 `call Sys.init`——之後所有執行順序由 `call` ／ `goto` 決定，與各函式的機器碼**實際排列位置**無關。組譯器的標籤機制會解析所有位址，所以檔案先翻後翻只影響指令在 ROM 的擺放、不影響語意。

練習 6：局部變數為何清零

(a) Hack VM 規定 `function` 要把 local 段清零，講師推測的理由是什麼？(b) C **不**清零區域變數。寫出一段 C 程式，其行為因此不可預測。

解答

(a) **安全性**：不同函式呼叫可能屬於不同「行程」（即使在單行程 OS 中把行程視為函式呼叫），清零防止新呼叫**讀到前一個呼叫殘留在堆疊上的資料**（返回位址、變數值等敏感內容）。
(b)

```c
int f(void) {
    int x;          // 未初始化：值 = 堆疊上的殘留垃圾
    return x + 1;   // 未定義行為
}
```

`x` 的值取決於**之前哪個函式用過這塊堆疊**——換編譯器、換最佳化等級、甚至換呼叫順序都會改變結果。真實世界中未初始化變數可能洩漏前一次呼叫留下的密碼或指標。

練習 7：嘗試 1 與嘗試 2

堆積為 $RAM[\texttt{0x800}]$–$RAM[\texttt{0x3FFF}]$。(a) 用**嘗試 1**：初始指標 `0x800`，依序執行 `alloc(0x100)`、`alloc(0x200)`、`deAlloc(0x800)`、`alloc(0x100)`。寫出每次的回傳值與指標變化。(b) 用**嘗試 2**（負擔 1 字）：初始一整塊自由段在 `0x800`（可用大小 `0x37FE`）。執行 `alloc(0x100)` 後，自由串列長什麼樣？

解答

(a) `alloc(0x100)` 回傳 `0x800`，指標 → `0x900`；`alloc(0x200)` 回傳 `0x900`，指標 → `0xB00`；`deAlloc(0x800)` **什麼都不做**；`alloc(0x100)` 回傳 `0xB00`，指標 → `0xC00`。已釋放的 `0x800`–`0x8FF` **永遠不會被重用**。
(b) first-fit 找到唯一的自由段（夠大），切割：

- 已配置段：基底 `0x800`、$RAM[\texttt{0x800}] = \texttt{0x100}$（大小），可用區 `0x801`–`0x900`，**回傳** `0x801`；
- 新自由段：從 `0x901` 開始，$RAM[\texttt{0x901}] = \texttt{0x37FE} - \texttt{0x100} - 1 = \texttt{0x36FD}$（切割又花掉 1 字負擔），$RAM[\texttt{0x902}] = \text{Null}$；
- 串列頭 RAM 指標指向 `0x901`。

練習 8：嘗試 2 的致命缺陷

用嘗試 2，堆積中相鄰兩段：A（基底 `0x800`，可用 `0x6FF`）與 B（基底 `0xF00`，可用 `0x6FF`），兩者都已配置，其餘記憶體都已用完。依序執行 `deAlloc(A)`、`deAlloc(B)`、`alloc(0x800)`。結果如何？為什麼？

解答

兩次 `deAlloc` 之後，自由串列有兩個段：B（頭插法在前）→ A，各可用 `0x6FF` 字。它們在記憶體上**緊鄰**（A 佔 `0x800`–`0xEFF`、B 佔 `0xF00`–`0x15FF`），合計 `0xE00` 字的連續空間。但 `alloc(0x800)` **失敗回傳** -1：嘗試 2 從不合併，串列裡只有兩個 `0x6FF` 的段，沒有一個 $\geq$ `0x800`。這正是引入嘗試 3（合併）的動機。

練習 9：嘗試 3 的合併

用嘗試 3（負擔 3 字：首 size、狀態、尾 size；自由段另用第三、四字存 next/prev）。記憶體中依序有三段：P（已用）、Q（已用）、R（自由）。現在呼叫 `deAlloc(Q)`。(a) 演算法如何**找到** P 與 R？(b) 合併後自由串列發生什麼事？(c) 為什麼自由串列要**雙向**？

解答

(a) R（後鄰）：Q 的基底加上 Q 的大小（存在 Q 段首）再加負擔，就是 R 的段首。P（前鄰）：讀 Q 段首**再往前一格**——那是 P 的**尾 size**（邊界標記），由此算出 P 的段首位置。沒有尾 size 就得從堆積開頭線性掃描才能找到前鄰。

(b) P 已用、R 自由：Q 與 R 合併成一段（新大小 $= Q + R +$ 負擔），R 要**從串列中刪除**、合併後的大段插入串列。

(c) 刪除 R 時我們是「直接跳到」R（經由位址計算），不是沿串列走到它。單向串列刪除節點需要知道**前驅**，只能從頭走訪 $O(n)$；雙向串列存了 prev 指標，刪除是 $O(1)$。

練習 10：碎片化

堆積共 `0x1000` 字（忽略負擔）。程式配置了 16 段、每段 `0x100` 字，然後釋放**第 1、3、5、⋯、15** 段（隔一個放一個）。(a) 現在自由空間共多少？最大的單次 `alloc` 能要多少？(b) 為什麼「重組」能解決卻不能由 `deAlloc` 自動執行？(c) first-fit 與 best-fit 在這個格局下有差別嗎？

解答

(a) 釋放了 8 段 $\times$ `0x100` $=$ `0x800` 字——**一半的堆積是自由的**。但自由段彼此被已用段隔開、無法合併，最大可配置只有 `0x100` 字。這就是碎片化的教科書場景。

(b) 重組要**搬移已配置的資料**並更新指向它們的指標。但 `alloc` 只回傳了裸指標，呼叫端可能已把它複製到任何地方（變數、結構、另一段堆積記憶體）——配置器**無從知道也無從更新**。Java 等語言用參考（多一層間接）就是為了讓執行環境能安全搬移物件。

(c) 沒有差別：所有自由段大小相同（`0x100`），first-fit 與 best-fit 會做出等價的選擇。兩者的差異要在自由段**大小不一**時才顯現——best-fit 挑最接近的、避免把大段切碎。

練習 11：分箱

用嘗試 4 的箱配置（4.6 節的範圍表）。(a) 可用大小 `0x30` 的自由段屬於哪個箱？(b) `alloc(0x20)` 從哪個箱開始掃？若空手而歸，下一步？(c) 分箱如何同時逼近 first-fit 的速度與 best-fit 的品質？

解答

(a) `0x30` 落在 `0x1D`–`0x38` $\Rightarrow$ $RAM[\texttt{0x801}]$ 那箱（第二箱）。

(b) `0x20` 也屬於第二箱（`0x1D`–`0x38`），從它開始掃。掃完沒找到，就掃包含 $2 \times \texttt{0x20} = \texttt{0x40}$ 的箱（第三箱，`0x39`–`0x70`），依此類推往大箱走。

(c) **速度**：不必掃整條自由串列，直接跳到大小相近的候選群——每次掃的串列短得多。**品質**：從「剛好夠大」的箱開始找，回傳的段大小自然接近需求，切剩的餘料少——效果接近 best-fit，卻不用付出全串列掃描的代價。若箱內再排序（平衡 BST），就能拿到真正的 best-fit。

練習 12：綜合追蹤

下列 VM 程式從 `Sys.init` 開始執行（bootstrap 已設 $SP = 256$）：

```
function Sys.init 0
push constant 6
push constant 7
call Foo.addOne 2
label HALT
goto HALT

function Foo.addOne 1
push argument 0
push argument 1
add
push constant 1
add
return
```

(a) `call Sys.init` 完成、進入 `Sys.init` 本體時 SP、LCL、ARG 是多少？(b) `Foo.addOne` 的 `return` 執行後，堆疊頂是什麼值、SP 是多少？

解答

(a) bootstrap 的 `call Sys.init 0` 推 5 個字的框架（返回位址 + 四個舊暫存器，值無意義但佔位）：$SP = 256 + 5 = \mathbf{261}$；$ARG = 256 - 0 = \mathbf{256}$（零個引數）；$LCL = \mathbf{261}$（`function Sys.init 0` 不再推 local）。

(b) 進入 `call Foo.addOne 2` 前，堆疊頂是 6、7（$RAM[261] = 6$、$RAM[262] = 7$、$SP = 263$）。`call` 後：$ARG = 261$、$LCL = 263 + 5 = 268$、`function Foo.addOne 1` 後 $SP = 269$（local $0 = 0$）。函式本體算出 $6 + 7 + 1 = 14$ 在堆疊頂。`return`：14 寫入 $RAM[ARG] = RAM[261]$、$SP = ARG + 1 = \mathbf{262}$、恢復暫存器、跳回。**堆疊頂是 14**——兩個引數被換成了返回值，接著程式進入 `HALT` 無窮迴圈。

A 附錄：速查表

A.1 函式指令總表

指令	語意
`function name k`	定義函式；local 段大小 k（進入時清零）。定義止於下一個 `function` 或 EOF
`call name n`	彈出 n 個引數、呼叫、把返回值推回堆疊
`return`	回傳堆疊頂的值、拆框架、跳回呼叫點

A.2 call ／ function ／ return 的實作配方

指令	產生的組語要做的事
`call f n`	推返回標籤位址 → 推 LCL、ARG、THIS、THAT → $ARG = SP - n$（推完後即 $SP - 5 - n$）→ $LCL = SP$ → 跳 `(f)` → 放返回標籤
`function f k`	放標籤 `(f)`（僅由函式名導出）→ $SP = LCL + k$ → local $0..k{-}1$ 清零
`return`	$R13 = RAM[LCL - 5]$（返回位址）→ $RAM[ARG]$ = 返回值 → $SP = ARG + 1$ → $THAT, THIS, ARG, LCL \leftarrow RAM[LCL - 1..LCL - 4]$ → 跳 $R13$

框架內相對 `LCL` 的位置：$LCL - 1$ 舊 THAT、$LCL - 2$ 舊 THIS、$LCL - 3$ 舊 ARG、$LCL - 4$ 舊 LCL、$LCL - 5$ 返回位址。

A.3 四種堆積配置演算法對照

演算法	負擔	資料結構	特性
1 只進不出	0	一個指標	極快；`deAlloc` 無效，記憶體耗盡即死
2 單向串列	1 字	段首 size ＋自由段單向串列	可回收；相鄰自由段永不合併
3 合併	3 字	＋狀態字＋尾 size（邊界標記）＋雙向串列	釋放時 $O(1)$ 合併前後鄰居
4 分箱	3 字	＋ 10 條按大小分箱的串列	接近 best-fit 的品質、接近 first-fit 的速度

A.4 名詞中英對照

英文	中文	英文	中文
call frame / frame	呼叫框架	coalescing	合併（聚結）
working stack	工作堆疊	boundary tag	邊界標記
global stack	全域堆疊	first-fit / best-fit	先合／最合策略
return address	返回位址	fragmentation	碎片化
bootstrapping	自舉	defragmentation	重組
cross-compiling	交叉編譯	bins	箱
heap	堆積	overhead	（管理）負擔
prologue / epilogue	序言／尾聲	use-after-free	釋放後使用
red zone	紅區	buffer overflow	緩衝區溢位

A.5 參考資料

- John Lapinskas, *Extending Hack VM ／ Functions in general ／ Functions in Hack VM: Syntax and implementation ／ Heap memory allocation: malloc and free*（10-1 至 10-4），COMSM1302, University of Bristol.
- Nisan & Schocken, *The Elements of Computing Systems*（nand2tetris），Ch. 8（Virtual Machine II）、Ch. 12.1（Memory Management）、附錄 6.
- System V Application Binary Interface, AMD64 Architecture Processor Supplement（x86-64 呼叫慣例、紅區）.
- Doug Lea, *A Memory Allocator*（dlmalloc 設計文件：邊界標記、128 個箱、立即合併）.
- Eli Bendersky, *Stack frame layout on x86-64*（框架指標省略與紅區的實務討論）.