第5章 連結與載入

連結器是用來將許多目的檔連結成一個執行檔的程式，而載入器則是用來將執行檔載入到記憶體當中執行的程式。在本章中，我們會將焦點放在連結器、載入器與目的檔這三個主題上。

組合語言程式被組譯完成後所輸出的檔案，通常稱為目的檔。假如目的檔中沒有未連結的外部變數，而且已經指定了起始位址，那麼，這種目的檔通常被稱為執行檔，載入器可以將執行檔載入到記憶體當中執行。

但是，如果有外部變數尚未被連結，那麼，目的檔就必須被連結成執行檔後，才能被放入到記憶體中執行。連結器必須計算出區段與變數的位址，並利用修正記錄進行修正後，才能消除這些外部變數，以便將目的檔轉換成執行檔。

其實，對於作業系統而言，執行檔只是一種與目的檔很像的二進位式檔案，例如，當你啟動命令列視窗，輸入某個執行檔的名稱，像是 Windows 中的 test.exe 或 Linux 當中的 a.out 時，命令列會呼叫載入器，根據該執行檔的格式，將該執行檔載入到記憶體當中。然後，將程式計數器設定到該程式的啟始位置上，交由 CPU 開始執行。

在本章中，我們將討論有關連結器與載入器的設計方式，並且說明目的檔的格式，以及動態連結等進階主題。在最後一節當中，我們將說明 GNU 的連結工具的用法，並以 a.out 格式作為實務案例，以說明連結與載入的真實情況。

5.1 簡介

組譯器會將組合語言程式轉為目的檔，最簡單的目的檔，是不包含分段結構的純目的碼，這種目的碼的載入非常簡單，我們將在本節當中進行說明，首先，請讀者看看範例 5.1 當中的目的碼。

範例 5.1 是一個 CPU0 程式的組譯報表，該報表列出了每個指令的位址，組合語言與目的碼，這個程式的功能是計算陣列 a 當中元素的總和，並將結果存入到變數 sum 當中。

範例 5.1 組合語言程式及其目的碼 (加總功能)

位址	組合語言程式	目的碼
0000	LDI R1, 0	08100000
0004	LD R2, aptr	002F003C
0008	LDI R3, 3	08300003
000C	LDI R4, 4	08400004
0010	LDI R9, 1	08900001
0014	FOR:	
0014	LD R5, [R2]	00520000
0018	ADD R1,R1,R5	13115000
001C	ADD R2, R2, R4	13224000
0020	SUB R3, R3, R9	14339000
0024	CMP R3, R0	10309000
0028	JNE FOR	21FFFFE8
002C	ST R1, sum	011F0018
0030	LD R8, sum	008F0014
0034	RET	2C000000
0038	a: WORD 3, 7, 4	00000003000000070000004
0044	aptr:WORD a	00000038
0048	sum:WORD 0	00000000

在範例 5.1 當中，組譯器採用相對於程式計數器 PC 定址的方式編碼，因此 LD R2, aptr 才會組譯成 002F003C，也就是 LD R2, [R15+0x3C] 的編碼方式。同理，JNE FOR、ST R1, sum、LD R8, sum 等指令也都是採用 PC 作為相對定址的基底。

由於採用相對於 PC 的定址法，因此不管目的檔被載入到記憶體的哪個位址，都可以直接執行，而不需要進行任何修正，這種與位址無關的編碼方式被稱為 PIC (Position Independent Code) 目的碼。

圖 5.1 顯示了<範例 5.1>的目的檔，這個目的檔只有單一個區塊，而且其中全部都是程式或資料碼，並沒有分成任何的段落，也沒有加入特殊的資料結構，因此是一種純粹的目的碼檔案，其格式非常簡單。

```
08100000 002F003C 08300003 08400004
08900001 00520000 13115000 13224000
14339000 10309000 21FFFFE8 011F0018
008F0014 2C000000 00000003 00000007
00000004 00000038 00000000
```

圖 5.1 <範例 5.1>的目的檔

若要設計一種載入器以載入此種目的檔，也是非常簡單的，圖 5.2 顯示了這種載入器的演算法。在執行時，載入器只要將整個目的檔放入記憶體當中，接著設定好程式計數器 PC，剩下的就可以交給 CPU 去處理了。此時，CPU 就會開始提取該程式的第一個指令，以執行該程式。

```
Algorithm SimpleLoader
Input objFile
   memoryCode = loadFile(objFile);
   PC = memoryCode.startAddress;
End Algorithm
```

圖 5.2 簡單的載入器演算法

在圖 5.2 的演算法當中，假如 objFile 被載入到記憶體位址 0x1200 的地方，那麼，載入後的記憶體位址與內容將如圖 5.3 所示。

記憶體位址	記憶體內容
1200	08100000 002F003C 08300003 08400004
1210	08900001 00520000 13115000 13224000
1220	14339000 10309000 21FFFFE8 011F0018
1230	008F0014 2C000000 00000003 00000007
1240	00000004 00000038 00000000 XXXXXXXX

圖 5.3 <圖 5.1>的目的檔載入到記憶體後的結果

在圖 5.3 當中，記憶體的內容每列佔 16 個位元組，正好是 16 進位的 0x10，因此，記憶體位址從 0x1200 起開始算，每次遞增 0x10。由於目的碼的長度為 0x4C，因此，載入的範圍從 0x00 開始直到 0x4B 結束，最後的 XXXXXXXX 符號代表其內容未被指定，因此可能是任何數值。

圖 5.2 的載入器演算法相當簡單，其原因是目的檔格式很單純，沒有分段與任何特殊結構，純粹只是目的碼序列而已，因此載入器不需要解讀分段結構，其動作也就變得非常的單純。

真實的目的檔通常會有分段結構，將內文段、資料段、BSS 段分別儲存在不同的區塊當中，然後再使用一個檔頭結構儲存這些區塊的索引資訊，這使得目的檔的結構變得複雜許多。

一但目的檔變複雜了，載入器也就會跟著複雜起來，通常載入器必須解讀目的檔的檔頭資訊，然後分別載入每個區段。因此，我們必須能理解這種複雜的目的檔結構後，才能設計載入對應的載入器。我們將在 5.2 節中說明真實的目的檔結構、然後在 5.3 節中說明連結器的原理後，再回到載入器這個主題，以便理解真實的載入器之原理。

5.2 目的檔

到目前為止，我們所看到的組合語言程式，大都是可以完整進行組譯的單一程式。但是，真實的程式通常會使用到函式庫，這些函式庫中的函數，通常不會與主程式一同組譯，而是在主程式被組譯之前就儲存在函式庫當中了。

主程式必須與函式庫進行連結動作，以便處理函數呼叫與參數傳遞的連結問題，連結器會將函數呼叫等動作，轉換成真實的位址與編碼後，將所有的目的碼集合成一個執行檔，然後才能交給載入器執行。

在實務上，很少看到有單一程式的系統，大部分的程式都需要使用函式庫。但是，假如程式設計師每次都要重新編譯 (或組譯) 所使用到的函式庫，那麼編譯的動作必定非常緩慢。因此，編譯器 (或組譯器) 通常會允許某些外部函數的存在，這些外部函數通常儲存在另一些目的檔中，然後，在連結的時候，才去解決這些外部引用的情況。

為了說明這樣的過程，我們特別設計了三個 C 語言程式，以實作並測試堆疊的功能，如範例 5.2 所示，其中 StackType.c 是資料宣告部分，StackFunc.c 包含了堆疊的基本函數，而 StackMain.c 則是測試用的主程式，這三個程式將會被分開編譯。

範例 5.2 具有交互引用的 C 語言程式 (實作堆疊功能)

StackType.c	StackFunc.c	StackMain.c

int stack[128];
int top = 0;
	extern int stack[];
	extern int top;
	
	void push(int x) {
	  stack[top++] = x;
	}
	
	int pop() {
	  return stack[--top];
		extern void push(int x);
		extern int pop();
		
		int main() {
		  int x;
		  push(3);
		  x= pop();
		  return 0;
		}

	}	

如果我們將上述程式改寫為 CPU0 的組合語言，那麼，程式將會如範例 5.3 所示。讀者會發現這個程式多了許多假指令，像是 .bss, .global, .data, .text, .extern 等，這些假指令是與目的檔格式有關的組譯指引，我們將會詳細說明這些假指令的用途。

範例 5.3 具有交互引用的組合語言程式 (實作堆疊功能)

檔名：StackType.s

```
.bss
.global stack
stack: RESW 512
.data
.global top
top: WORD 0
```

檔名：StackFunc.s

```
.text
.extern stack
.extern top
.global push
push :
   POP R1          ; R1=x
   LD R2, top      ; R2=top
   LD R3, stack    ; R3 = stack
   LDI R4, 4       ; R4=4
   LDI R5, 1       ; R5=1
   MUL R6, R2, R4  ; R6=top+4
   STR R1, [R3+R6] ; stack[top]=x
   ADD R2, R2, R5  ; R2 ++
   ST R2, top      ; top=R2
   RET
.global pop
pop:
   LD R2, top      ; R2=top
   LD R3, stack    ; R3=stack
   LDI R4, 4       ; R4=4
   LDI R5, 1       ; R5 = 1
   MUL R6, R2, R4  ; R5 = top*4
   LDR R1, [R3+R6] ; R1=stack[top]
   SUB R2, R2, R5  ; R2=R2-1
   ST R2, top      ; top = R2
   RET
```

檔名：StackMain.s

```
.text
.extern push
.extern pop
.global main
main:
   LDI R1, 3     ; R1=3
   PUSH R1       ; 推入參數
   CALL push     ; push(3)
   CALL pop      ; pop()
   ST R1, x      ; x=pop()
   LDI R1, 0     ; ret 0
   RET
x: RESW 1
```

在範例 5.3 當中，使用了 .bss , .text, .data 這樣的分段指令，代表後續的指令必須被編入該段落中。其中的 .bss 是 Block Started by Symbol 的簡稱，是儲存未初

始化全域變數的區段。而 .data 則代表資料段，用來儲存已初始化的全域變數。假指令 .text 代表內文段 (或稱程式段)，用來儲存程式的指令碼。

假指令 .global 宣告某變數為全域變數，可以供外部的程式引用，像是範例 5.3 當中的 global stack, global top, global push, global pop, global main 等，都是全域變數的範例。

引用全域變數時必須使用 .extern 指令，以告訴組譯器這些變數是外部變數，才不會因為這些變數未被定義而產生錯誤訊息。像是範例 5.3 當中的 extern stack, extern top, extern push, extern pop 等，都是引用外部變數的範例。

在範例 5.3 中，變數 stack, top 與函數 push, pop, main 等，都是全域變數，這些變數可以供其他程式使用。因此，在定義這些變數的檔案中，我們會以 .global 指令宣告該變數為全域變數，然後再引用這些變數的檔案中，以 .extern 指令宣告該變數為外部變數。

範例 5.3 中的三個檔案，如果在分開的情況下，分別被組譯成目的碼，那麼，在組譯時，組譯器將無法計算出外部變數的位址，因此，必須進行特別的處理。

範例 5.4、範例 5.5、範例 5.6 分別顯示了 StackType.s、StackFunc.s 與 StackMain.s 等程式的組譯報表。在這些報表的目的碼欄位中，位址前面會加上分段代碼，像是 B (BSS 段), D (DATA 段), T (TEXT 段) 等，這是由於使用了分段機制的原因。

在這些範例中，由於具有分段與外部變數，使得其目的碼變得較為複雜。我們使用 <記錄類型>(欄位 1、欄位 2、…) 的語法表示目的碼。例如，S(B,0000,stack) 所代表的是符號記錄 S，其三個欄位的值分別為 (B, 0000, stack)。

範例 5.4 堆疊型態 StackType.s 及其目的碼

位址	組合語言檔： StackType.s	目的碼	說明
	.bss .global stack		BSS 段開始 stack 是全域變數
B 0000	stack RESW 512	B(0200),S(B,0000,stack)	保留 B 區段 512 byte stack 的位址為 B0000
	.data .global top		DATA 段開始 top 是全域變數
D 0000	top WORD 0	D(00000000),S(D,0000,top)	top 編碼為 D(00000000) top 的位址是 D0000

在範例 5.4 中，開頭的假指令 .bss 指示了該段為 BSS 段，這導致後續的 stack 變數被放入了 BSS 段當中，所以 stack RESW 512 的目的碼被編為 B(0200)，這代表 stack 變數會在 BSS 段中保留一塊區域，其長度為 512 (0x0200) bytes。

由於 stack 變數已經被 .global stack 指令宣告為全域變數，因此，stack RESW 512 指令還會輸出一筆 S(B,0000,stack) 的記錄，以便在記錄符號 stack 的位址是 BSS 分段的 0000。

範例 5.4 中的 .data 指令宣告了資料段的開始，而 top WORD 0 則宣告了 top 變數為 0，於是會輸出 D(00000000) 這樣一個資料段的記錄。而且，由於 top 被 .global top 指令宣告為全域變數，因此，接著會輸出一筆 S(D,0000,top)，以便在符號表中記錄符號 top 的位址是 DATA 分段的 0000。

在範例 5.5 中，假指令 .text 告訴組譯器這是內文段的開始。然後，指令 .extern stack, .extern top 宣告了 stack 與 top 為外部變數。接著，指令 .global push 與後續的 .global pop 宣告了 push 與 pop 為全域變數。當組譯器看到 push: 符號標記時，由於發現 push 是全域變數，因此，會輸出一筆 S(T,0000,push) 記錄，以記錄符號 push 的位址為 TEXT 分段的 0000。

範例 5.5 堆疊函數 StackFunc.s 及其目的碼

	組合語言檔： StackFunc.s	目的碼	說明
	.text		內文段開始
	.extern stack	S(U,,stack)	stack 為外部變數
	.extern top	S(U,,top)	top 為外部變數
	.global push		push 為全域變數
	push :	S(T,0000, push)	push 函數開始
T 0000	POP R1 // R1=x	T(31100000)	
T 0004	LD R2, top	T(002F0000), M(T,0004,top,pc)	修改記錄 +top
T 0008	LD R3, stack	T(003F0000), M(T,0008,stack,pc)	修改記錄 +stack
T 000C	LDI R4, 4	T(08400004)	
T 0010	LDI R5, 1	T(08500001)	
T 0014	MUL R5, R2, R4	T(15524000)	
T 0018	STR R1, [R3+R5]	T(05135000)	
T 001C	ADD R2, R2, R5	T(13225000)	
T 0020	ST R2, top	T(012F0000), M(T,0020,top,pc)	
T 0024	RET	T(2C000000)	

	.global pop		pop 為全域變數
	pop:	S(T,0028, pop)	pop 函數開始
T 0028	LD R2, top	T(002F0000), M(T,0028,top,pc)	修改記錄 +top
T 002C	LD R3, stack	T(003F0000), M(T,002C,stack,pc)	修改記錄 +stack
T 0030	LDI R4, 4	T(08400004)	
T 0034	LDI R5, 1	T(08500001)	
T 0038	MUL R5, R2, R4	T(15524000)	
T 003C	LDR R1, [R3+R5]	T(04135000)	
T 0040	SUB R2, R2, R5	T(14225000)	
T 0044	ST R2, top	T(012F0000),M(T,0044,top,pc)	修改記錄 +top
T 0048	RET	T(2C000000)	

接著，當組譯器看到 POP R1 指令時，由於先前的 .text 宣告了目前為內文段，因此，會輸出一筆 T(31100000) 的記錄，其中的 T 代表 text 段。

接著，由於 LD R2, top 當中的 top 是一個外部變數，因此無法直接確定 top 的位址，所以只能採用修正記錄的方式，以 M(T,0004,top,pc) 記錄對指令碼 T(002F0000) 進行修正。

在此有必要詳細說明指令 LD R2, top 的編碼方式，為何該指令會被編為目的碼 T(002F0000) 呢？這是由於 CPU0 的組譯器採用了相對於 PC (R15) 的定址方式，因此，指令 LD R2, top 被解讀為 LD R2, [R15+offset]，於是位移 offset 就被設定為 top-pc。但是由於 top 是外部變數，其位址無法得知，於是只能以 LD R2, [R15+0000] 的編碼方式，先將 offset 設定為 0，

於是 LD R2, top 指令被編碼為 T(002F0000)，其中的 F 就是 R15 的編碼。然後，再補上一筆修改記錄 M(T,0004,top,pc)，以處理外部變數的引用。這個修改記錄要求連結器必須對位於 T0004 的目的碼進行修正動作，以便在連結器取得 top 的位址後可以修改該目的碼，計算出 top 變數與定址基底 R15 之間的距離。

接著，讓我們繼續看範例 5.6 中 StackMain.s 的目的碼，一開始的假指令 .text 告訴組譯器這是內文段的開始。然後，指令 extern push, extern pop 等宣告了符號 push, pop 為外部函數。

接著，指令 .global main 宣告了 main 為全域變數，在 main: 標記定義時，由於全域符號 main 的位址已經確定，因此會輸出一筆 S(T,0000,main) 的記錄，告知連結器 main 符號的位址，以便給其他程式引用。

範例 5.6 堆疊主程式 StackMain.s 及其目的碼

	組合語言檔： StackMain.s	目的碼	
	.text	S(U,,push)	程式 (Text) 段開始
	.extern push	S(U,,push)	push 為外部變數
	.extern pop		pop 為外部變數
	.global main		main 為全域變數
	main:	S(T,0000,main)	main 程式開始
T 0000	LDI R1, 3	T(08100003)	
T 0004	PUSH R1	T(30100000)	
T 0008	CALL push	T(2BF00000), M(T,0008,push,pc)	修改記錄 +push
T 000C	CALL pop	T(2BF00000), M(T,000C,pop,pc)	修改記錄 +pop
T 0010	ST R1, x	T(01100008)	
T 0014	LDI R1, 0	T(08100000)	
T 0018	RET	T(2C000000)	
T 001C	x: RESW 1	T(00000000)	區域變數 x

範例 5.6 中最值得注意的是最後一行的 x: RESW 1，其目的碼為 T(00000000)，其中開頭的 T 代表內文段 .text。但是，這明明是一個資料宣告指令，為何被放入內文段呢？

其實，程式與資料都可以被放入內文段 .text 中，這是為何我們不將內文段稱為程式段 (.code) 的原因。範例 5.6 當中的 x 變數是區域變數，而非全域變數，放入到內文段後可以利用相對於 PC 的定址方式，直接對 ST R1, x 這個指令進行編碼，而不需要加上修改記錄，為了方便起見，我們直接將 x 變數放入到內文段當中。

我們已經看過了內文段 (.text, 以 T 開頭)、資料段 (.data，以 D 開頭) 與 BSS 段 (.bss, 以 B 開頭) 的目的碼。也看過 .global, .extern 所造成的符號表記錄 (S 記錄)、與修改記錄 (M 記錄) 等範例。接著，讓我們正式說明這些記錄的格式與用途。

圖 5.4 以顯示了目的檔的主要記錄類型，這些記錄會以結構化的方式，被儲存在目的檔當中。

記錄	程式範例	目的碼	說明
T	CALL push	T(2BF00000)	內文段的目的碼
D	top WORD 0	D(00000000)	資料段的目的碼

B	stack RESW 512	B(0200)	保留 BSS 段的空間
M	CALL push	M(T,0008,push,pc)	修改記錄 (或稱重定位記錄)：採用相對於 PC 的位址計算方式
S	.global pop	S(T,0028, pop)	符號記錄：記錄全域變數的位址
S	.extern stack	S(U,,stack)	符號記錄：記錄外部變數

圖 5.4 目的檔中的記錄與範例

T、D、B、M、S 等記錄是目的檔當中常見的五種記錄型態，由於這些記錄必須被輸出到檔案當中，因此，必須使用某種存檔格式。在本書中，我們會使用文字的方式表示這些目的碼，這只是為了讓讀者閱讀方便而採用的呈現方式。實際上，目的檔通常會直接以二進位的結構存檔，因為這樣在連結與載入時會更為簡單快速。

記錄的儲存格式

舉例而言，修改記錄 M 的結構可能如圖 5.5 所示，包含 section、offset、symbol 與 type 等欄位。其中 section 記錄了分段、offset 代表修改位址、symbol 為符號、而 type 則為修改類型。

```
#define TYPE_ABSOLUTE 0
#define TYPE_PC_RELATIVE 1
…
typedef struct
{
  int section;          // 段代號
  int offset;           // 待修改位址
  int symbol;           // 符號代碼 (用以取得修改值)
  int type;             // 修改記錄類型 (0. 絕對 1. 相對於 PC、2. …)
} RelocationRecord;     // 修改記錄 (M 記錄)
```

圖 5.5 重定位記錄 (M 記錄) 的表示法

如果我們用更精確的表示法，以表示修改記錄，那麼像 M(T,0008, push, pc) 這樣一個修改記錄，可以改寫為如下的表示法。

RelocationRecord { section = id(text), offset = 0x0008, symbol= id(push), type=1 }

其中的 type=1 代表 TYPE_PC_RELATIVE，也就是使用 PC 的相對定址法，因此在修改時必須採用相對於 PC 的計算方式。

細心的讀者可能已經注意到，我們在圖 5.5 的結構中只記錄符號的代碼 (int symbol)，而非記錄符號的名稱，這樣的記錄方式會使得目的檔更為精簡，使用上會更有效率。

RelocationRecord 中的 symbol 欄位會連結到符號表 (Symbol Table)，符號表是由符號記錄 (S 記錄) 所組成的表格，其中的 S 是符號 Symbol 的意思。S 記錄是由 .extern 與 .global 等指令所造成的。指令 .extern 會在符號表中建立一個未定義 (Undefined, 簡寫為 U) 的符號記錄，像是 (U,,stack)、(U,,top)、(U,,push)、(U,,pop) 等，而用 .global 指令所定義的全域變數，在變數位址確定後，會產生一個已知位址的符號記錄，像是 S(T,0028, pop)、S(T,0000, pop)、S(T,0000,main)、S(D,0000,top)、S(B,0000,stack) 等。

如果將 S 記錄表達為 C 語言，則其結構將會如圖 5.6 的 SymbolRecord 所示，其中包含 name, section, value, type 等四個欄位，name 代表符號名稱，section 則是該符號的分段，value 代表符號的值，而 type 則代表符號的類型。

```
#define SYM_TYPE_DATA 0
#define SYM_TYPE_FUNC 1
#define SYM_TYPE_SECT 2
#define SYM_TYPE_FILE 3
…
typedef struct
{
  int name;     // 符號在字串表中的位址
  int section; // 定義該符號的分段
  int value;    // 符號的值 (通常為一個位址)
} SymbolRecord; // 符號記錄 (S 記錄)
```

圖 5.6 符號記錄 (S 記錄) 的表示法

舉例而言，我們可以用下列的表示法代表 S(T,0028, pop) 這樣一筆記錄。

SymbolRecord { name=id(pop), section=id(text), value=0028 }

在上述表示法中，欄位 name 所記錄的是 pop 這個字串的代碼，分段 section 所記錄的是內文段 (.text) 的代碼，而 value 則儲存了 pop 的位址 0028。

當然，這種表示法相當冗長，但是可以較精確的反映其內部結構。我們可以從中

知道名稱欄位 name 所儲存的是一個代碼，而非直接儲存字串。而欄位 section 所儲存的是分段代碼，而非分段名稱。

在 RelocationRecord 與 SymbolRecord 當中，我們都盡可能的以代號儲存，而避免採用字串的方式，以節省儲存空間。因此，勢必會有一個字串表的存在，以便在必要的時候將代號轉換為字串。目的檔中的字串表通常格式相當簡單，我們可以將 C 語言當中以 \0 結尾的字串，直接串連後所形成的一個表格，圖 5.7 顯示了對應到範例 5.6 的字串表結構，我們特別將其中每個字的位址標示出來，以便讓讀者能更清楚的辨認字串表結構。

字串：".text\0.data\0.bss\0stack\0top\0push\0pop\0main\0"															
位址	0	1	2	3	4	5	6	7	9	9	A	B	C	D	E	F
0000	.	t	e	x	t	\0	.	d	a	t	a	\0	.	b	s	s
0010	\0	s	t	a	c	k	\0	t	o	p	\0	p	u	s	h	\0
0020	p	o	p	\0	m	a	i	n	\0							

圖 5.7 字串表的結構

目的檔結構的設計，對連結器與載入器的執行速度有相當大的影響，因此，真實的目的檔會盡可能的將各種記錄以最緊湊的格式儲存，以便節省儲存空間，並提升執行速度。

為了有效率的組織目的檔，通常會在目的檔的開頭，加入一個表頭記錄，以儲存各個分段的索引資訊。連結器與載入器會讀取這個表頭記錄，然後再讀出每一個分段的內容。

為了方便起見，在本章中我們盡可能採用簡寫的方式，像是 S(B,0000,stack)，而不採用像 SymbolRecord 這樣寫法，以簡化目的檔的呈現方式，讓讀者得以快速的閱讀這些目的碼。

當許多同類型的目的碼記錄被放在一起，形成記錄群時，我們會使用 <記錄名稱>{ 內容 } 的方式表達，例如 T{ 31100000 002F0000 003F0000 08400004….} 這樣的結構就可以表達整個 StackFunc.s 內文段的 T 記錄。但是，如果記錄的內容中還有子記錄，則我們會用 <記錄名稱>{ (子記錄 1) (子記錄 2) … } 的方式，舉例而言，我們會用 M{ (T,0004,top,pc) (0008,stack,pc) (0020,top,pc)… } 這個語句，代表 StackFunc.s 的所有修改記錄。

根據這種方式，StackType.s、StackFunc.s、StackMain.s 的目的碼，可以更簡潔的表達為圖 5.8 的表示法。請讀者參考範例 5.4、範例 5.5 與範例 5.6 以對照閱讀。

檔案	目的碼	說明
stackType.o	B { 0200 } D { 00000000 } S { (B,0000,stack), (D,0000, top) }	BSS 段 資料段 符號表

stackFunc.o	T {
	31100000 002F0000 003F0000 08400004
	08500001 15524000 05135000 13225000
	012F0000 2C000000 002F0000 003F0000
	08400004 08500001 15524000 04135000
	14225000 012F0000 2C000000}
	M {
	(T,0004,top,pc) (T,0008,stack,pc) (T,0020,top,pc)
	(T,0028,top,pc) (T,002C,stack,pc) (T,0044,top,pc)
	}
	S { (T,0000,push) (T,0028,pop) (U,,stack) (U,,top) }
		內文段
		
		重定位表
		
		符號表

stackMain.o	T {
	08100003 30100000 2BF00000 2BF00000
	01100008 08100000 2C000000 00000000
	}
	M { (T,0008,push,pc), (T,000C,pop,pc) }
	S { (U,,push), (U,,pop), (T,0000, main) }
		內文段
		
		重定位表
		符號表

圖 5.8 本書使用的目的檔表示法 – 以 StackFunc.o, StackType.o, StackMain.o 為範例

圖 5.8 中的這些目的檔，由於具有外部引用，因此，必須經過進一步的連結，才能成為執行檔，交由載入器執行。接著，我們將說明如何將目的檔連結成一個完整的執行檔。

5.3 連結器

連結器可以將許多個目的檔連結成一個可執行檔，也有可能是像函式庫 (Library) 或動態函式庫 (DLL) 這樣的半成品，圖 5.9 顯示了連結器的功能，

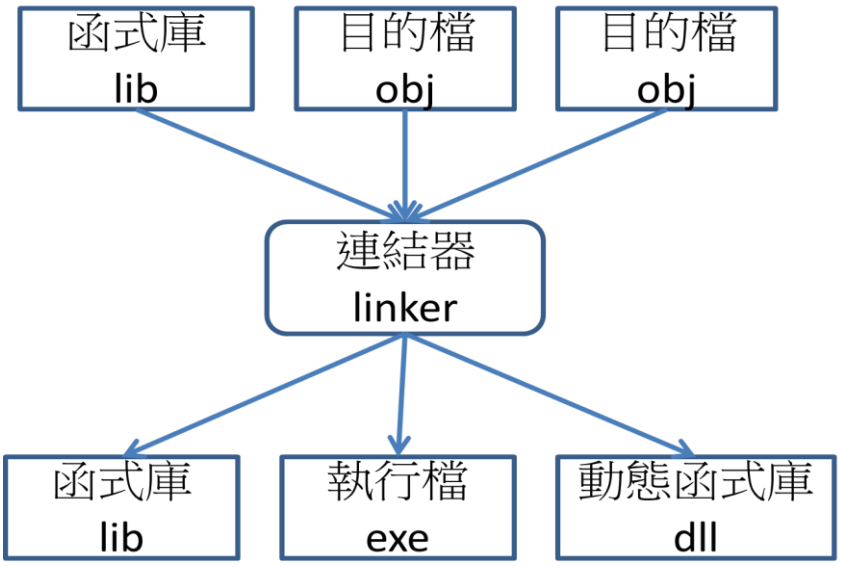

圖 5.9 連結器的輸入與輸出

在連結的過程中，連結器會盡可能的消除外部引用，確定外部變數的位址，讓程式盡可能的接近可執行狀態。另外，還必須進行區段合併的動作，將內文段 (.text)、資料段 (.data) 與 BSS (.bss) 段合併，並且修改符號的位址，更新符號表與修改記錄等，以下讓我們分別說明這些連結器的功能。

區段合併

要將一群目的檔連結成一個執行檔，首先會將相同性質的區段合併。例如，所有的內文段 (.text) 會被合併形成單一個內文段，所有的資料段會被合併形成單一個資料段，所有 BSS 段會被合併形成單一個 BSS 段。這個過程如圖 5.10 所示。

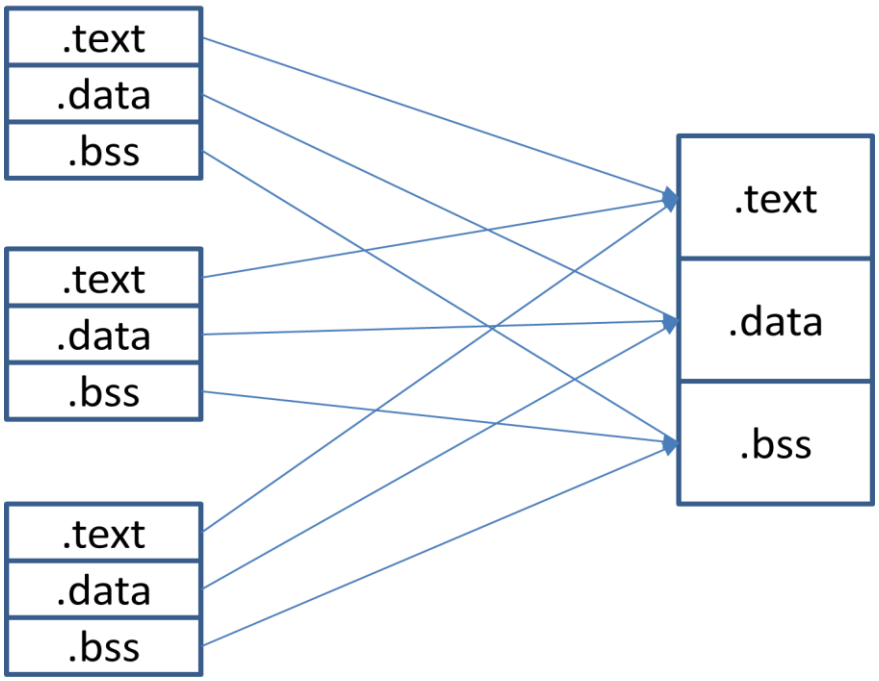

圖 5.10 連結器的功能 – 區段合併

連結器必須適當的安排所有區段的位址，將同類型的區段合併，然後修改符號表中每個符號的位址。

舉例而言，假如我們利用連結指令 ld -o stack.exe StackFunc.o StackType.o StackMain.o，將 StackFunc.o StackType.o StackMain.o 等三個檔案連結成執行檔 stack.exe。此時，圖 5.8 中的三個目的檔，會被連結器連結成圖 5.11 的執行檔。

指令：ld -o stack.exe StackFunc.o StackType.o StackMain.o 說明：連結 StackFunc.o StackType.o StackMain.o 三個檔案以形成執行檔 Stack.exe	

Stack.exe	T {
	// 來源：StackMain.o 的內文段 (T 記錄)
	08100003 30100000 2BF00014 2BF00028
	01100008 08100000 2C000000 00000000
	// 來源：StackFunc.o 的內文段 (T 記錄)
	31100000 002F0000 003F0000 08400004
	08500001 15524000 05135000 13225000
	012F0000 2C000000 002F0000 003F0000
	08400004 08500001 15524000 04135000
	14225000 012F0000 2C000000
	}
	B { 0200 }
	D { 00000000 }
	M {
	// 來源：StackMain.o 的修改記錄
	~~(T,0008,push,pc) (T,000C,pop,pc)~~
	// 來源：StackFunc.o 的修改記錄
	(T,0024,top,pc) (T,0028,stack,pc) (T,0040,top,pc) (T,0048,top,pc),
	(T,004C,stack,pc) (T,0064,top,pc)
	}
	S {
	// 來源：StackMain.o 的符號表
	(T,0000,main)
	// 來源：StackFunc.o 的符號表
	(T,0020,push) (T,0048,pop)
	// StackType.o
	(B,0000, stack) (D,0000, top)
	}

圖 5.11 連結器輸出的執行檔，以 Stack.exe 為例

在圖 5.11 中，符號表 S { (T,0000,main) (T,0020,push) (T,0048,pop) (B,0000, stack) (D,0000, top) } 當中的符號位址都已經確定，不再有 (U,,stack) (U,,top) (U,,push), (U,,pop) 這樣的未連結符號，這顯示在 Stack.exe 這個執行檔中，所有的變數位址都已經確定了。

那麼，這些位址是如何計算出來的呢？關於這點，我們可以參考如圖 5.12 的連結過程圖。從圖中我們可以看到，StackMain.o 的 T 記錄被放在 T,0000 的位址，然後，StackFunc.o 的 T 記錄被附加在其後，從 T,0020 開始。

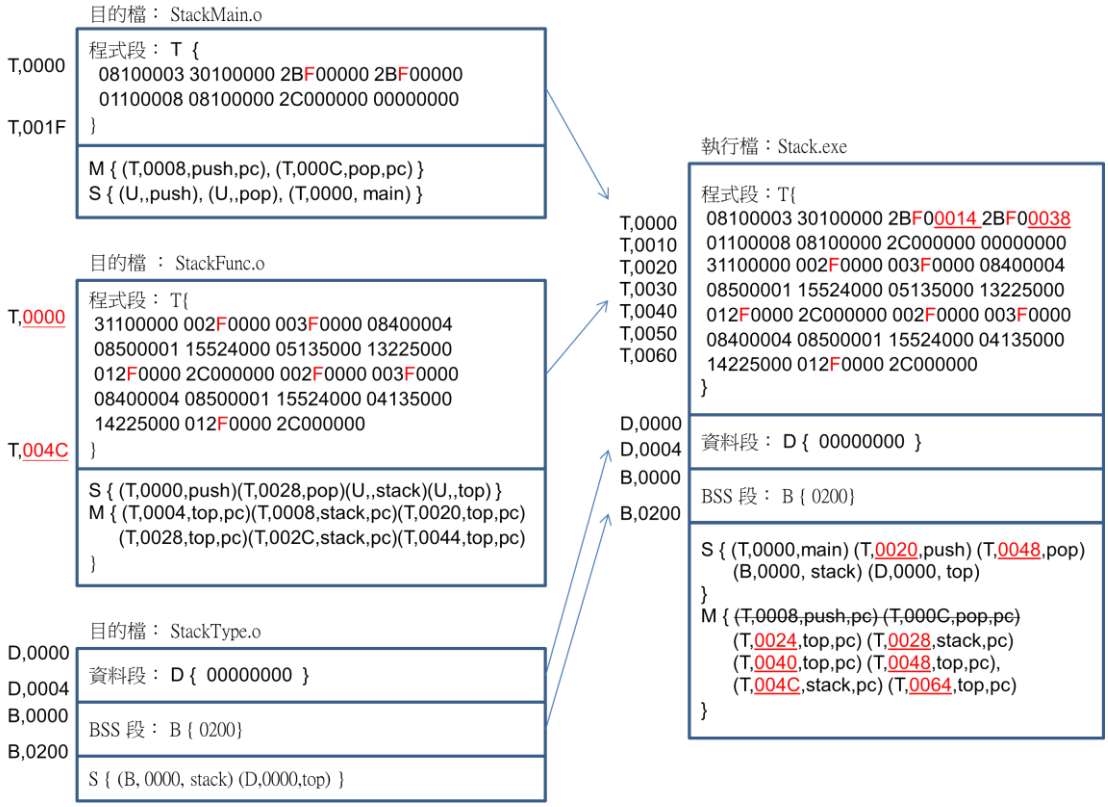

圖 5.12 連結過程圖

根據圖 5.12 的連結過程圖，我們可以知道整個目的檔的連結情況。由於 StackFunc.o 被附加在 StackMain.o 之後，因此，StackMain.o 中符號表的 S 記錄會被調整，其中，push 函數的位址從 T0000 移動到 T0020，所以符號記錄 S(T,0000,push) 也就被改為 S(T,0020,push)。同樣的，pop 函數的位址從 T0028 移動到 T0048，於是符號記錄 S(T,0028,pop) 被修改為 S(T,0048,pop)。

除了符號表需要修改以外，連結器還會更新重定位記錄 (M 記錄)。舉例而言，由於 StackFunc.o 的內文段從 T0000 移到了 T0020，這使得 StackFunc.o 中 M 記錄的位址全都被加上了 0x0020。

接著，由於符號 push, pop 的位址已經確定，於是連結器可以利用修改記錄 M 中的 (T,0008,push,pc) (T,000C,pop,pc) 對目的碼進行修改。舉例而言，位於 T,0008 的目的碼 2B000000 被改為 2BF00014 ，原因是 address(push)-PC (0008+4) = T:0020-T:000C=0014，而位於 T,000C 的 2B000000 則被修改為 2BF00038，原因是 address(pop)-PC (000C+4) =T:0048-T:0010=0038。在修改完畢之後，這些修改記錄

就可以被去除了。舉例而言，圖 5.12 當中的 (T,0008,push,pc) (T,000C,pop,pc) 兩筆修改記錄就被刪除掉了。

連結器建立了的完整的符號表，更新了修改記錄之後，就可以輸出執行檔，連結完成後的執行檔 Stack.exe 的內容如圖 5.11 所示。當然，連結器還必須建立新的表頭結構，這樣載入器才能根據表頭結構取得各分段，將各分段的目的碼載入到記憶體中執行。

圖 5.13 顯示了連結器的演算法，該演算法假設輸出一定是執行檔，而且所有的符號都應該要被定義。如此，就可以先讀取每一個目的檔中的符號表，並計算每個分段的大小，然後，再利用修改記錄修改目的碼，修改完後輸出到執行檔當中。

連結器的演算法	說明
Algorithm Linker	連結器的演算法
Input objFileList	輸入為一群目的檔
output exeFile	輸出為執行檔
secTable = new SectionTable(T,D,B,…);	建立分段表
foreach objFile in objFileList	對於每一個目的檔 objFile
foreach S_record s in objFile	對於每個符號記錄 s
if s.type is not 'U'	如果 s 不是未定義符號
s.address += secTable[s.name].size	更新符號 s 的位址
exeFile.writeS_Record(s)	輸出 s 記錄到執行檔中
foreach M_record m in objFile	對於每一筆修改記錄 m
m.address += secTable[m.section].size	修正 m 記錄的位址
exeFile.writeM_Record(m)	輸出 m 記錄到執行檔中
foreach section e in objFile	對於每一個分段 e
secTable[e.name].size += e.size	加入 objFile 的分段大小
exeFile.writeSection(e);	輸出分段到執行檔
exeFile.modifyHeader(secTable)	修正執行檔表頭
End Algorithm	

圖 5.13 連結器的演算法

5.4 載入器

組譯器所輸出的檔案稱為目的檔。在上一節當中，我們已經看過幾個目的檔的範例，並且看到了如圖 5.11 所示的執行檔。雖然執行檔中幾乎都是目的碼，但卻不能直接放入記憶體當中就執行。因為，其中仍然有分段資訊與修改記錄等結

構。

載入器必須讀取執行檔，組合其中的分段，放入到記憶體中，然後根據修改記錄修正記憶體中的指令區與資料區，最後才將程式計數器 PC 設定為起始指令的位址，開始執行該程式。

圖 5.14 (a) 顯示了一個執行中程式的記憶體配置情況，其中內文段與資料段乃是由目的檔中搬移過來的，BSS 段則只要保留空間大小即可，有些載入器會順便將 BSS 段清除為 0，但是有些載入器則不會做這樣的動作，因此，我們通常不能假設載入後 BSS 段的內容都會是 0。在圖 5.14(b) 當中，我們以 XXXXXXX 符號代表其內容值為不確定的情況。

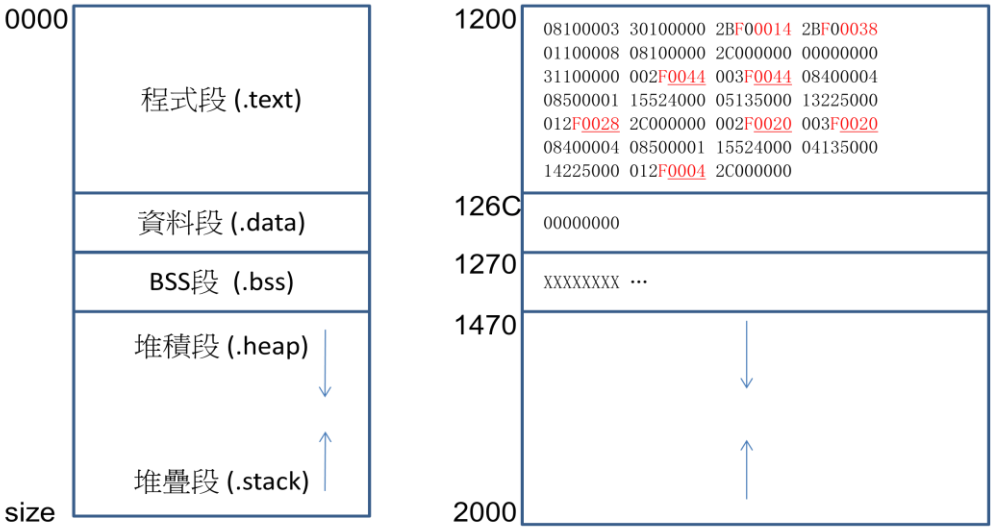

圖 5.14 執行檔 Stack.exe 的記憶體配置情況

除了目的檔中的區段之外，載入後的程式會多出兩個區段，那便是堆疊段 (Stack) 與堆積段 (Heap)。堆疊段用來儲存函數中的參數、區域變數與返回點等資料，如此才能正確進行函數呼叫，甚至執行遞迴函數。而堆積段則用來儲存動態分配的記憶體，像是 C 語言中用 malloc 函數所配置的記憶體，並且可以用 free 函數釋放後回收使用。

通常，堆疊段與堆積段會共用同一塊空間，但是其成長方向相反，這樣才能充分利用中間部分的空間，以避免浪費記憶體。

載入器除了載入內文段、資料段、保留 BSS 段的空間之外，還必須根據修正記錄 M，修改跨區段的變數引用。如此，程式才能正確無誤的執行。圖 5.15 顯示

了目的檔 Stack.exe 被載入到記憶體位址 0x1200 後的情況，其中連結器根據 M 記錄所修改的部分被加上了底線。

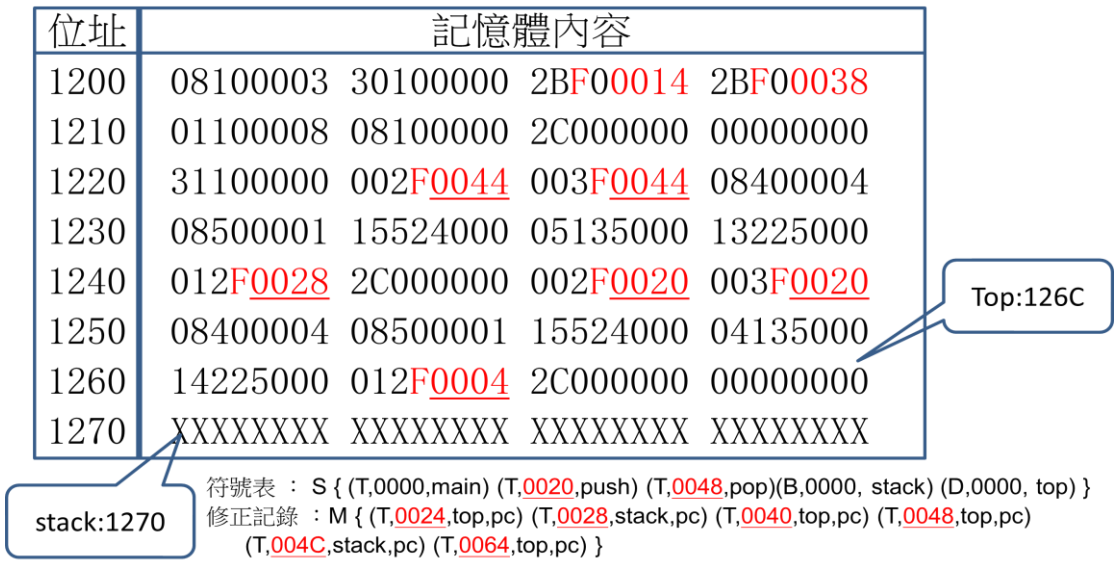

圖 5.15 執行檔 Stack.exe 載入記憶體後的情況

一但目的檔載入完成後，載入器就會將程式計數器 PC 設定為程式的進入點，讓 CPU 開始執行該程式。舉例而言，當 Stack.exe 被載入到 0x1200 後，載入器會將 PC 設定為 0x1200，於是，該程式便開始執行了。

載入器的演算法

現在，我們已經完整說明了載入器的原理，可以開始說明載入器的演算法了，圖 5.16 顯示了載入器的演算法。這個演算法的輸入是一個執行檔 exeFile。載入器首先讀取檔頭，以取得各個分段的結構資訊，包含分段的大小、種類、在 exe 檔中的位置等資訊。然後，分配各整個執行檔分段所需要的記憶體 (包含內文段、資料段、BSS 段、堆疊段與堆積段等)，接著將各個分段載入到記憶體中，並且根據修改記錄 M，修改記憶體中的目的碼，如此就完成了載入的動作。最後，將程式計數器設定為起始位址，開始執行程式。

	載入器的演算法	說明
1	Algorithm Loader	載入器
2	Input exeFile	輸入檔：執行檔
3	exeFile.readHeader();	讀取檔頭
4	memory = allocateMemory(exeFile);	分配執行空間
5	foreach section e in exeFile	對於每個分段 e
6	load e into memory	載入 e 到記憶體中

7	foreach M_record m in exeFile	對於每個修改記錄 m
8	using m to modify memory	根據 m 修改記憶體
9	PC = memory.startAddress	設定 PC 後開始執行
10	End Algorithm	

圖 5.16 載入器的演算法

5.5 動態連結

傳統的連結器會將所有程式連結成單一的執行檔，在執行時只需要該執行檔就能順利執行。但是，使用動態連結機制時，函式庫可以先不需要被連結進來，而是在需要的時候才透過動態連結器 (Dynamic Linker) 尋找並連結函式庫，這種方式可以不用載入全部的程式，因此可以節省記憶體。當很多執行中的程式共用到函式庫時，動態連結所節省的記憶體數量就相當可觀。

除了節省記憶體之外，動態連結技術還可以節省連結階段所花費的時間。這是因為動態連結函式庫 (Dynamic Linking Libraries: DLLs) 可以單獨被編譯、組譯與連結，程式設計師不需要在改了某個函式庫後就重新連結所有程式。因此，對程式開發人員而言，動態連結技術可以節省程式開發的時間，因為程式設計人員使用編譯、組譯與連結器的次數往往非常頻繁，有些人甚至不到一分鐘就會編譯一次。

除此之外，動態連結函式庫由於可以單獨重新編譯，因此，一但編譯完新版本後，就可以直接取代舊版本。這讓舊程式得以不需重新編譯就能連結到新函式庫，因此，只要將動態連結函式庫換掉，舊程式仍可順利執行該新版的函數，這讓動態函數成為一種可任意抽換的函式庫。

動態連結器的任務，就是在需要的時候才載入動態函式庫，並且進行連結 (linking) 與重新定位 (relocation) 的動作。然後再執行該函式庫中的函數。

當程式第一次執行到動態函數時，動態連結器會搜尋看看該函數是否已經在記憶體中，如果有則會跳到該函數執行，如果沒有則會呼叫載入器，動態的將該函式庫載入到記憶體，然後才執行該函數。

使用動態連結機制呼叫函數時，通常會利用間接跳轉的方式，先跳入一個稱為 Stub 的程式中。在第一次呼叫該函數時，Stub 會請求動態載入器載入該函數，而在第二次呼叫時，則會直接跳入該函數。

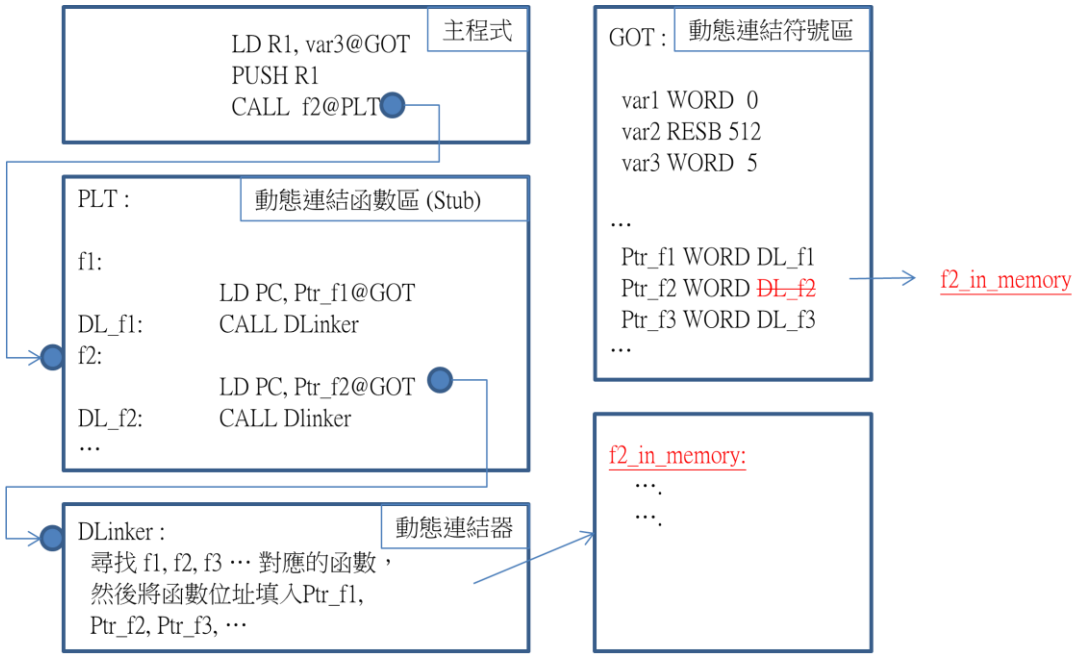

圖 5.17 動態連結機制的實作方式

圖 5.17 所顯示了這個動態跳轉機制，在程式剛載入之時，動態連結符號區 GOT 當中的 Ptr_f1,Ptr_f2 等變數，會被填入 DL_f1, DL_f2 等位址。當主程式執行 CALL f2@PLT 指令時，會跳到 Stub 區的 f2 標記。此時，會執行 LD PC, Ptr_f2@GOT 這個指令。但是由於 Ptr_f2 當中儲存的是 DL_f2 的位址，因此，該 LD 跳轉指令相當於沒有任何作用。於是會繼續執行 CALL DLinker 這個指令，該指令會呼叫 Dlinker 去載入 f2 對應的函數到記憶體中。

一但 f2 的函數內容 f2_in_memory 被載入後，Dlinker 會將載入 f2_in_memory 的位址填入到 Ptr_f2 當中。於是，當下一次主程式再呼叫 CALL f2@PLT 時，Stub 區中的 LD PC, Ptr_f2@GOT 就會直接跳入動態函數 f2_in_memory 中，而不會再透過載入器了。

透過上述的實作方法，就能達成動態連結的目的，讓函式庫動態的掛載到系統當中，等到執行時期才進行連結的動作。動態連結雖然不需要事先載入函式庫，但是在編譯時就已經知道動態函數的名稱與參數類型，因此，編譯器可以事先檢查函數型態的相容性。

動態連結函式庫通常是與位置無關的程式碼 (Position Independent Code)，其優點是可任意抽換函式庫，不需重新連結就能讓系統直接運行。但是，這也可能造成『動態連結地獄』 (DLL hell) 的困境，因為，如果新的函式庫有錯，或者與舊的程式不相容，那麼，原本執行正常的程式會突然出現錯誤，甚至無法使用。

雖然動態連結已經是相當常見的功能，但是在 UNIX/Linux 與 Windows 中卻有不同的稱呼，在 Windows 中直接稱為 DLLs (Dynamic Linking Libraries)，其附檔名通常為 .dll，而在 UNIX/Linux 中的動態連結函式庫則被稱為 Share Objects，其附檔名通常是 .so。

動態載入

一但有了『動態連結技術』之後，就能很容易的實作出『動態載入技術』。所謂的動態載入技術，是允許程式在執行時期，再決定要載入哪個函式庫的技術。這種技術比動態連結更具有彈性，靈活度也更高。其方法是讓程式可以呼叫動態載入器，以便載入程式，因此才被稱為動態載入技術。

舉例而言，我們可以讓使用者在程式中輸入某個函式庫名稱，然後立刻用『動態載入技術』載入該函式庫執行。這會使得程式具有較大的彈性，因為，我們可以在必要的時候呼叫動態載入器，讓使用者決定要載入哪些函式庫。

表格 5.1 顯示了 MS. Windows 與 Linux 兩個平台的動態載入函式庫對照表。在 Linux 當中，『libdl.so』 函式庫可支援動態載入功能，其引用檔為 dlfcn.h，我們可以用 dlopen() 載入動態函式庫，然後用 dlsym() 取得函數指標，最後用 dlclose() 函數關閉函式庫。

而在 MS. Windows 當中，動態函式庫直接內建在核心當中，其引用檔為 windows.h，動態連結的目的檔為 Kernel32.dll，可以使用 LoadLibrary() 與 LoadLibraryEx() 等函數載入動態函式庫，然後用 GetProcAddress 取得函數位址，最後用 FreeLibrary() 關閉動態函式庫。

表格 5.1 Linux 與 MS. Windows 中動態載入函式庫的對照比較表

	UNIX/Linux	Windows
引入檔	#include <dlfcn.h>	#include <windows.h>
函式庫檔	libdl.so	Kernel32.dll
載入功能	dlopen	LoadLibrary LoadLibraryEx
取得函數	dlsym	GetProcAddress
關閉功能	dlclose	FreeLibrary

動態載入技術可以說是動態連結的一種自然延伸，只要將作業系統中的動態連結技術獨立成一組函式庫，就能成為動態載入函式庫，這種函式庫對程式設計人員而言，是一種強大的系統函式庫，能夠充分展現出動態連結技術的力量。

5.6 實務案例(一)：GNU 連結工具

在本節中，我們將示範如何使用 GNU 的連結工具，這包含目的檔格式的觀察、函式庫的建立、程式的編譯與連結等。在本節中將以範例導向的方式說明這些連結工具的用法，有關這些工具的用法說明，請參閱本書的附錄 C。

首先，請讀者先撰寫下列三個程式，或者直接切換到本書範例 ch05 的資料夾中，您就會看到這些程式。

範例 5.7 具有交互引用的 C 語言程式 (實作堆疊功能)

StackType.c	StackFunc.c	StackMain.c

int stack[128];
int top = 0;
	extern int stack[];
	extern int top;
	
	void push(int x) {
	  stack[top++] = x;
	}
	
	int pop() {
	  return stack[--top];
	}
		extern void push(int x);
		extern int pop();
		
		int main() {
		  int x;
		  push(3);
		  x= pop();
		  return 0;
		}

接著，讓我們來練習編譯與連結動作，請讀者按照範例 5.8 的步驟進行操作。

範例 5.8 <範例 5.2>的編譯、連結執行

指令與執行結果	說明
C:\ch05>gcc -c StackType.c -o StackType.o C:\ch05>gcc -c StackFunc.c -o StackFunc.o C:\ch05>gcc -c StackMain.c -o StackMain.o C:\ch05>gcc StackMain.o StackFunc.o StackType.o -o stack	編譯 StackType.c 為目的檔 編譯 StackFunc.c 為目的檔 編譯 StackMain.c 為目的檔 連結成為執行檔

那麼，到底這些具有外部引用的 C 語言程式，會被編譯器編譯成甚麼樣子呢？讓我們用 gcc 中的 –S 參數，將這些檔案編譯成組合語言看看。

範例 5.9 將<範例 5.2>的 C 語言程式編譯為 IA32 組合語言

指令與執行結果	說明
C:\ch05>gcc -S StackType.c -o StackType.s	編譯 StackType.c，輸出 StackType.s
C:\ch05>gcc -S StackFunc.c -o StackFunc.s	編譯 StackFunc.c，輸出 StackFunc.s
C:\ch05>gcc -S StackMain.c -o StackMain.s	編譯 StackMain.c，輸出 StackFunc.s

範例 5.10 顯示了 StackType.s, StackFunc.s, StackMain.c 等三個組合語言檔的內容。

範例 5.10<範例 5.2>所對應的 IA32 組合語言檔

檔案：StackType.s

```
        .file  "StackType.c"
.globl _top
        .bss
        .align 4
_top:
        .space 4
        .comm     _stack, 512        # 512
```

檔案：StackFunc.s

```
        .file  "StackFunc.c"
        .text
.globl _push
        .def _push; .scl 2; .type 32; .endef
_push:
        pushl       %ebp
        movl        %esp, %ebp
        movl        _top, %eax
        movl        %eax, %edx
        movl        8(%ebp), %eax
        movl        %eax, _stack(,%edx,4)
        incl   _top
        popl %ebp
        ret
.globl _pop
        .def _pop;.scl 2; .type 32; .endef
_pop:
        pushl       %ebp
        movl        %esp, %ebp
        decl _top
```

檔案：StackMain.s

```
        .file  "StackMain.c"
        .def  ___main;.scl 2;.type 32;.endef
        .text
.globl _main
        .def  _main;     .scl   2;     .type
        32;   .endef
_main:
        pushl       %ebp
        movl        %esp, %ebp
        subl $24, %esp
        andl $-16, %esp
        movl        $0, %eax
        addl $15, %eax
        addl $15, %eax
        shrl  $4, %eax
        sall   $4, %eax
        movl        %eax, -8(%ebp)
        movl        -8(%ebp), %eax
        call   __alloca
        call   ___main
        movl        $3, (%esp)
        call   _push
        call   _pop
        movl        %eax, -4(%ebp)
        movl        $0, %eax
        leave
        ret
        .def _pop; .scl 3; .type 32; .endef
```

movl _top, %eax movl _stack(,%eax,4), %eax popl %ebp ret	.def _push; .scl 3; .type 32; .endef

對於程式 StackType.c 而言，其中的指令 int top=0; 被編譯為組合語言 .bss 段中的 _top .space 4，而 int stack[STACK_SIZE] 被編譯為 .comm _stack, 400，而 _top 被宣告為 .globl，代表全域變數 (global)。

對於程式 StackFunc.c 而言，其中的 push, pop 函數分別被編譯為標記 _push: 與 _pop，並且用.globl _push 與 .globl _pop 標記為全域變數。同樣的 StackMain.c 當中的主程式 main 也被編譯為標記 _main，並標上全域變數記號 .globl，放在內文段 .text 當中。

如果我們用 nm 指令，檢視目的檔 StackType.o 與 StackFunc.o 中的符號表，那麼，就能看到其中的分段與全域變數，範例 5.11 顯示 nm 指令所查看到的符號表。

範例 5.11 使用 nm 指令觀看目的檔的符號表

指令與執行結果	說明
C:\ch05>nm StackType.o	顯示 StackType.o 的符號表(map)
00000000 b .bss	bss 段 (b:未初始化變數)
00000000 d .data	data 段 (d:已初始化資料)
00000000 t .text	text 段 (t:內文段)
00000200 C _stack	int stack[] 的定義 (C:Common)
00000000 B _top	int top=0 的定義 (B:bss)
C:\ch05>nm StackFunc.o	顯示 StackFunc.o 的符號表(map)
00000000 b .bss	bss 段 (b:未初始化變數)
00000000 d .data	data 段 (d:已初始化資料)
00000000 t .text	text 段 (t:內文段)
0000001c T _pop	pop() 的定義 (T:內文段)
00000000 T _push	push() 的定義 (T:內文段)
U _stack	未定義 (U:_stack)
U _top	未定義 (U:_top)
C:\ch05>nm StackMain.o	顯示 StackMain.o 的符號表(map)
00000000 b .bss	bss 段 (b:未初始化變數)

00000000 d .data	data 段 (d:已初始化資料)
00000000 t .text	text 段 (t:內文段)
U ___main	___main() 未定義
U __alloca	__alloca() 未定義
00000000 T _main	_main() 的定義 (T:內文段)
U _pop	_pop() 未定義
U _push	_push() 未定義

在範例 5.11 當中，有一個奇怪的地方，int top=0 被放入到 BSS 段當中，這是因為 GNU 規定載入器要在載入時先將 BSS 段全部清除為 0，因此設定為零的變數時就可以編入 BSS 段，而不需要編入 DATA 段了。這種作法可以降低 DATA 段目的碼所的佔據的空間，因為 BSS 段只要記錄長度，但不需要將初始值編入到目的碼當中。

如果想知道每一個區段 (Section) 的大小，可以使用 size 指令，範例 5.12 顯示了 size 指令的使用結果。

範例 5.12 用 size 指令檢視目的檔中分段大小 (單位為 byte)

指令與執行結果	說明
C:₩ch05>size StackFunc.o text data bss dec hex filename 64 0 0 64 40 StackFunc.o C:₩ch05>size StackType.o text data bss dec hex filename 0 0 16 16 10 StackType.o C:₩ch05>size StackMain.o	StackFunc.o 主要為程式段 StackType.o 主要為 BSS 段 StackMain.o 主要為程式段 Stack.exe 各段均有

text data bss dec hex filename 80 0 0 80 50 StackMain.o C:₩ch05>size Stack.exe text data bss dec hex filename 2548 700 704 3952 f70 Stack.exe	

從範例 5.12 當中，您可以看到執行檔 Stack.exe 當中的各分段都比目的檔當中大，這是因為 gcc 編譯器會自動連結一些必要的函式庫，像是 ___main(), __alloca() 等，因此各個區段都明顯得比目的檔中更大。

製作靜態函式庫

要製作靜態函式庫，可以透過 ar (archive) 指令加上 –r 參數，將一群目的檔包裝為函式庫，例如，在範例 5.13 中，我們就用了 ar -r libstack.a StackFunc.o StackType.o 這樣一個指令，將 StackFunc.o StackType.o 這兩個目的檔包裝成 libstack.a 函式庫，放在 lib 資料夾下。然後，在連結的時候，再利用 gcc -o stack StackMain.c -lstack -L . 這樣的指令，直接連結函式庫。只要懂得利用函式庫，就不需要逐個檔案進行連結了。

您也可以用 ar 指令加上 –tv 參數，檢視函式庫中到底包含了哪些目的檔，甚至可以用 ar 指令加上 –x 參數，將目的檔從函式庫當中取出，

範例 5.13 使用 ar 指令建立函式庫

指令與執行結果	說明
C:\ch05>ar -r libstack.a StackFunc.o StackType.o	建立 libstack.a 函式庫
C:\ch05>gcc -o stack StackMain.c -lstack -L .	編譯 StackMain.c 並連結 libstack.a 函式庫
C:\ch05>ar -tv libstack.a	顯示函式庫 libstack.a 的內容
rw-rw-rw- 0/0 324 Apr 04 18:45 2010 StackType.o	包含 StackFunc.o
rw-rw-rw- 0/0 502 Apr 04 18:46 2010 StackFunc.o	包含 StackType.o

C:\ch05>ar -x libstack.a StackFunc.o	從 libstack.a 中取出 StackFunc.o

目的檔觀察工具 **- objdump**

Objdump 是 GNU 主要的目的檔觀察工具，附錄 C 中有其詳細的用法。您可以使用 objdump 觀察目的檔的表頭與內容，您可以用 –t 參數顯示符號表，-r 參數顯示修改記錄，範例 5.14 顯示了 objdump –tr StackFunc.o 的執行結果。

範例 5.14 使用 objdump 觀察目的檔

指令：objdump -tr StackFunc.o

```
C:₩ch05>objdump -tr StackFunc.o


StackFunc.o:     file format pe-i386


SYMBOL TABLE:
[  0](sec -2)(fl 0x00)(ty   0)(scl 103) (nx 1) 0x00000000 StackFunc.c
File
[  2](sec  1)(fl 0x00)(ty  20)(scl   2) (nx 1) 0x00000000 _push
AUX tagndx 0 ttlsiz 0x0 lnnos 0 next 0
[  4](sec  1)(fl 0x00)(ty  20)(scl   2) (nx 0) 0x0000001c _pop
[  5](sec  1)(fl 0x00)(ty   0)(scl   3) (nx 1) 0x00000000 .text
AUX scnlen 0x33 nreloc 6 nlnno 0
[  7](sec  2)(fl 0x00)(ty   0)(scl   3) (nx 1) 0x00000000 .data
```

```
AUX scnlen 0x0 nreloc 0 nlnno 0

[  9](sec  3)(fl 0x00)(ty   0)(scl   3) (nx 1) 0x00000000 .bss

AUX scnlen 0x0 nreloc 0 nlnno 0

[ 11](sec  0)(fl 0x00)(ty   0)(scl   2) (nx 0) 0x00000000 _top

[ 12](sec  0)(fl 0x00)(ty   0)(scl   2) (nx 0) 0x00000000 _stack



RELOCATION RECORDS FOR [.text]:

OFFSET   TYPE                VALUE

00000004 dir32               _top

00000010 dir32               _stack

00000016 dir32               _top

00000021 dir32               _top

00000026 dir32               _top

0000002d dir32               _stack
```

從範例 5.14 的傾印結果中，我們可以看到 StackFunc.o 目的檔的『SYMBOL TABLE』段落中含有符號表，包含了 _push, _pop, .text, .data, .bss, _top, _stack, 等符號，其中符號的 _push 的分段為 sec 1，代表 _push 是內文段，因為 .text 符號的段落也是 sec 1。而 _stack 與 _top 兩個符號的分段為 sec 0，代表這兩個符號尚未被定義。

在『RELOCATION RECORDS FOR [.text]』段落中，包含了許多內文段的修正記錄，舉例而言，像是 00000004 dir32 _top 代表在 .text 區段位址 0x04 的地方，

需要加上 _top 的以便修正，其修正方法為 dir32 (據筆者猜測，這應該是 32 bits direct addressing 的意思)。同理，00000010 dir32 _stack 這行修正記錄，則是說 0x10 的位址處，需要加上 _stack 的以便修正，其修正方法也是 dir32。

專案建置工具

當程式越來越多時，編譯、連結與測試的動作會變得相當繁瑣，此時就必須使用專案建置工具。GNU 的 make 是專案編譯上相當著名的典型工具，在此，我們將用 make 來學習大型專案開發所需的專案管理技巧。並且透過 make 觀察大型專案的開發過程，讓讀者得以學習到專業的系統程式開發流程。

專案編譯工具 make 是用來編譯專案的強有力工具，使用 make 工具可以有效的整合許多程式，並且進行快速的大型專案編譯動作。像是著名的 Linux 作業系統就是以 gcc 與 make 等 GNU 工具所建構出來的。因此，學習 GNU 工具更是進入 Linux 系統程式設計的捷徑。

對於初學者而言，可能會覺得 make 的語法相當怪異，然而，對於有經驗的程式設計人員而言，卻會覺得 make 專案管理工具相當方便。

在此，我們將利用上述的範例程式 (StackType.c, StackFunc.c, StackMain.c)，示範如何使用 make 工具。這些檔案在光碟中被放在範例的 ch05 資料夾中，其中包含一個名為 Makefile 的專案檔案，該檔案的內容如下：

範例 5.15 專案編譯的 Makefile 檔案。

```
CC = gcc
AR = ar
OBJS = StackType.o StackFunc.o
BIN = stack
RM = rm -f
INCS = -I .
LIBS = -L .
CFLAGS = $(INCS) $(LIBS)

all: $(BIN)

clean:
```

```
	${RM} *.o *.exe *.a

$(BIN): $(AR)
	$(CC) StackMain.c -lstack -o $(BIN) $(CFLAGS)

$(AR) : $(OBJS)
	$(AR) -r libstack.a $(OBJS)

StackFunc.o : StackFunc.c
	$(CC) -c StackFunc.c -o StackFunc.o $(CFLAGS)

StackType.o : StackType.c
	$(CC) -c StackType.c -o StackType.o $(CFLAGS)
```

接著，我們進行專案編譯的動作，其過程如範例 5.16 所示。您可以看到當專案編譯指令 make 執行時，一連串的動作被觸發。

範例 5.16 使用 make 工具的過程

指令與執行結果	說明
C:\ch05>make clean rm -f *.o *.exe *.a	清除上一次產生的檔案 使用 rm 清除*.o,*.exe,*.a 檔
C:\ch05>make gcc -c StackType.c -o StackType.o -I . -L . gcc -c StackFunc.c -o StackFunc.o -I . -L . ar -r libstack.a StackType.o StackFunc.o ar: creating libstack.a gcc StackMain.c -lstack -o stack -I . -L .	進行專案編譯 編譯 StackType 中… 編譯 StackFunc 中… 建立函式庫 libstack.a 中… 函式庫建立完成 編譯主程式，並連結函式庫 輸出執行檔 stack

在 make 的建置過程中，第一個目標 all 會先被觸發。接著，根據 all : $(BIN) 規則，其中 $(BIN) 指的是前面所定義的 BIN = stack，因此，$(BIN) 會被觸發。接著，由於 $(BIN): $(AR) 規則，於是目標 $(AR) 被觸發。根據這樣的連鎖反應規則，您可以看到如圖 5.18 的觸發樹與執行過程，請讀者仔細追蹤，應可理解 make 檔案的執行原理。

```
→ all
  → $(BIN)
```

→ $(AR)
→ $(OBJS)
→ StackType.o
StackType.o : gcc -c StackType.c -o StackType.o -I . -L .
→ StackFunc.o
StackFunc.o : gcc -c StackFunc.c -o StackFunc.o -I . -L .
$(AR) : ar -r libstack.a StackType.o StackFunc.o
$(AR) : ar: creating libstack.a
$(BIN) : gcc StackMain.c -lstack -o stack -I . -L .

圖 5.18 <範例 5.16>中 make 指令的觸發樹與執行過程

GNU 工具的指令不只這些，其參數的用法更是繁多，在本節中，我們介紹了有關連結 (ld)、函式庫 (ar) 與目的檔格式 (nm, objdump) 等相關的指令，有關 GNU 工具的的用法說明，請參考本書的附錄 C。

5.7 實務案例(二)：目的檔格式 – a.out

真正的目的檔，為了效能的緣故，通常不會儲存為文字格式，而是儲存為二進位格式。像是 DOS 當中的 .com 檔案，Windows 當中的 .exe 檔案，與 Linux 早期的 a.out 格式，還有近期的 ELF 格式，都是常見的目的檔格式。

為了讓讀者更清楚目的檔的格式，在本節中，我們將以 Linux 早期使用的 a.out 這個格式作為範例，以便詳細說明目的檔的格式。

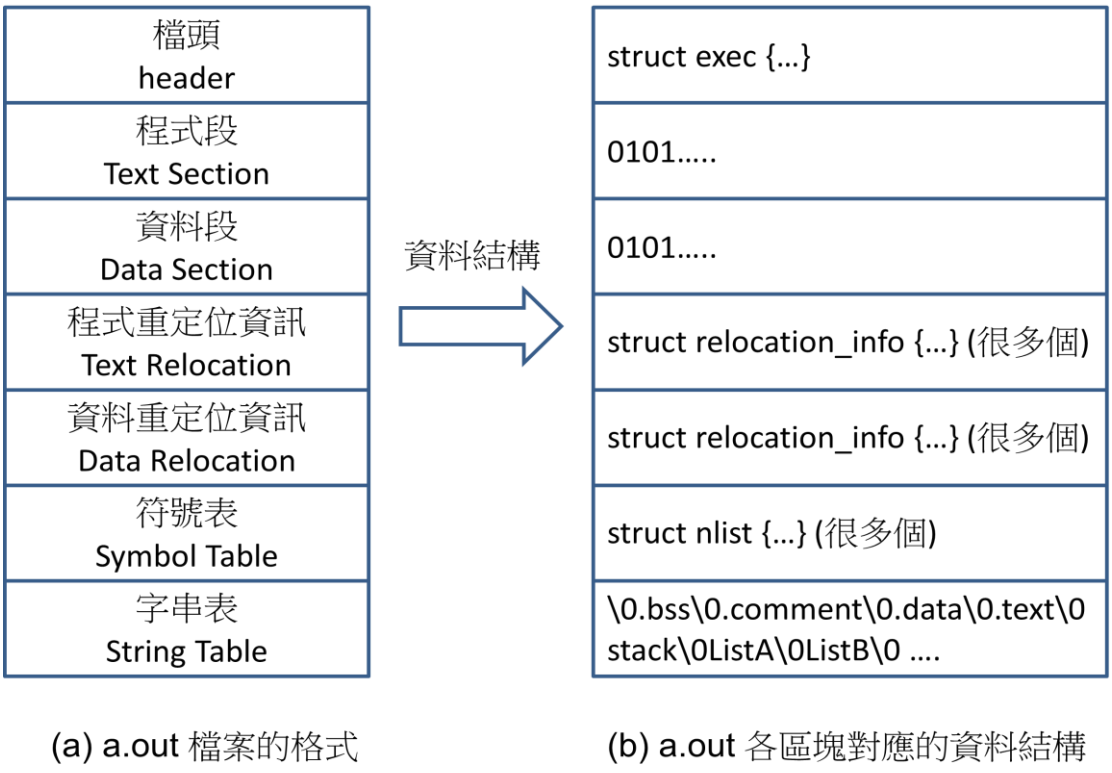

圖 5.19 目的檔 a.out 各區段所對應的資料結構

早期的 Linux 採用的是較簡單的目的檔格式 a.out。這是由於 UNIX 與 Linux 的預設編譯執行檔名稱為 a.out 的原因。圖 5.19**Error! Reference source not found.** 顯示了 a.out 檔案中的區段，以及各區段的資料結構。

目的檔 a.out 的格式相當簡單，總共分為 7 個段落，包含三種主要的資料結構，也就是 1. 檔頭結構 (exec)、2. 重定位結構 (relocation_info)、3. 字串表結構 (nlist)。這三個結構的定義如圖 5.20 所示。

```
struct exec {                   // a.out 的檔頭結構
   unsigned long a_magic;       // 執行檔魔數
                                //     OMAGIC:0407, NMAGIC:0410,ZMAGIC:0413
   unsigned a_text;             // 內文段長度
   unsigned a_data;             // 資料段長度
   unsigned a_bss;              // 檔案中的未初始化資料區長度
   unsigned a_syms;             // 檔案中的符號表長度
   unsigned a_entry;            // 執行起始位址
   unsigned a_trsize;           // 程式重定位資訊長度
   unsigned a_drsize;           // 資料重定位資訊長度
};
```

```
struct relocation_info {        // a.out 的重定位結構
   int r_address;               // 段內需要重定位的位址
   unsigned int r_symbolnum:24; // r_extern=1 時:符號的序號值,
                                // r_extern=0 時:段內需要重定位的位址
   unsigned int r_pcrel:1;      // PC 相關旗標？？是「相對」吧？？
   unsigned int r_length:2;     // 被重定位的欄位長度 (2 的次方)
                                // 2^0=1,2^1=2,2^2=4,2^3=8 bytes)。
   unsigned int r_extern:1;     // 外部引用旗標。      1-以外部符號重定位
                                //                     0-以段的地址重定位。
   unsigned int r_pad:4;        // 最後補滿的 4 個位元 (沒有使用到的部分)。
};

struct nlist {                  // a.out 的符號表結構
   union {                      //
      char *n_name;             // 字串指標，
      struct nlist *n_next;     // 或者是指向另一個符號項結構的指標，
      long n_strx;              // 或者是符號名稱在字串表中的位元組偏移值。
   } n_un;                      //
   unsigned char n_type;        // 符號類型 N_ABS/N_TEXT/N_DATA/N_BSS 等。
   char n_other;                // 通常不用
   short n_desc;                // 保留給除錯程式用
   unsigned long n_value;       // 含有符號的值，對於代碼、資料和 BSS 符號，
                                // 通常是一個位址。
};
```

圖 5.20 目的檔 a.out 的資料結構（以 C 語言定義）

請讀者對照圖 5.19**Error! Reference source not found.**與圖 5.20，應該能很容易的看出下列對應關係，『檔頭部分』使用的是 exec 結構。『程式碼部分』與『資料部分』則直接存放二進位目的碼，不需定義特殊的資料結構。而在『程式碼重定位的部分』與『資料重定位的部分』，使用的是 relocation_info 的結構。最後，在『符號表』的部分，使用的是 nlist 的結構。而『字串表』部分則直接使用了 C 語言以 '\0' 為結尾的字串結構 **Error! Reference source not found.**。

當 a.out 檔案被載入時，載入器首先會讀取檔頭部分，接著根據檔頭保留適當的大小的記憶體空間 (包含內文段、資料段、BSS 段、堆疊段、堆積段等)。然後，載入器會讀取內文段與資料段的目的碼，放入對應的記憶體中。接著，利用重定位資訊修改對應的記憶體資料。最後，才把程式計數器設定為 exec.a_entry 所對

應的位址，開始執行該程式，圖 5.21 顯示了 a.out 檔被載入的過程。

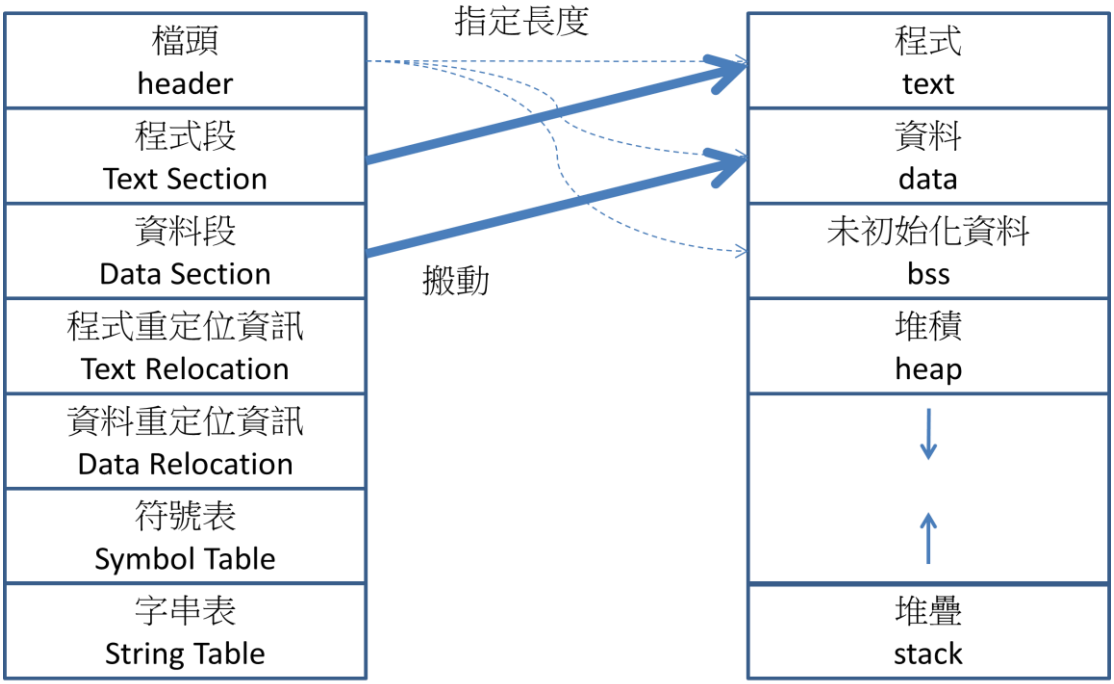

圖 5.21 目的檔 a.out 的載入過程

目的檔格式 a.out 是一種比較簡單而直接的格式，但其缺點是格式太過固定，因此無法支援較為進階的功能，像是動態連結與載入等。目前，UNIX/Linux 普遍都已改用 ELF 格式作為標準的目的檔格式，在 Linux 2.6 當中就支援了動態載入的功能。

由於 ELF 檔案的格式較為複雜，我們將不在本書當中進行說明，但在本書的網站上有詳細的論述，有興趣的讀者可以參考網站上的文章。

5.8 實務案例(三)：Linux 的動態載入技術

在 Linux 的 2.6 版當中支援了動態連結技術，因而也支援了動態載入函式庫。Linux 的動態載入函式庫儲存在 libdl.so 這個檔案中，您可以透過引用 <dlfcn.h> 這個檔案，使用 Linux 的動態載入函式庫。

範例 5.17 顯示了 Linux 動態載入的程式範例，該程式使用 dlopen 載入數學函式庫 libm.so，然後用 dlsym 取得 cos() 函數的指標，接著呼叫該函數印出 cos(2.0) 的值，最後用 dlclose() 關閉函式庫。

範例 5.17 Linux 動態載入函式庫的使用範例

程式：dlcall.c 編譯指令：gcc -o dl_call dl_call.c –ldl	說明

#include <dlfcn.h>
 	
int main(void) {
  void *handle = dlopen("libm.so", RTLD_LAZY);
  double (*cosine)(double);
  cosine = dlsym(handle, "cos");
  printf ("%f\n", (*cosine)(2.0));
  dlclose(handle);
  return 0;
}
	引用 dlfcn.h 動態函式庫
	
	開啟 shared library 'libm'
	宣告 cos()函數的變數
	找出 cos()函數的記憶體位址
	呼叫 cos()函數
	關閉函式庫

這種動態載入功能對系統程式設計師而言非常有用，舉例而言，為了降低所使用的記憶體數量，您可以利用動態載入功能，在需要時才載入某個函式庫，並且在用完後立即釋放。如此，就能在程式尚未結束前，就將函式庫所佔據的空間還給作業系統，因而降低的記憶體的用量。或者，您也可以在使用者輸入某些參數，像是函式庫名稱之後，才用程式動態的載入該函數庫，如此，就不需要事先將所有可能用到的函式庫全部引用進來，因而解決了記憶體不足的現象。

習題

5.1 請說明連結器的輸入、輸出與功能為何？
5.2 請說明載入器的功能為何？
5.3 請說明 CPU0 組合語言當中的 .text, .data 與 .bss 等假指令的用途為何？
5.4 請說明 CPU0 組合語言當中的 .global 與 .extern 等假指令的用途為何？
5.5 請說明 CPU0 目的檔中的 T, D, B, S, M 等記錄各有何用途？
5.6 請說明連結器是如何處理外部引用問題的？
5.7 請說明目的檔中符號表的用途？
5.8 請使用 gcc 加上-S -c 參數，分別編譯範例 5.2 中的三個程式，以分別產生組合語言檔。
5.9 繼續前一題，請使用 gcc 分別組譯前一題所產生的三個組合語言檔，產生目的檔。
5.10 繼續前一題，請使用 gcc 連結前一題所產生的三個目的檔，輸出執行檔。
5.11 繼續前一題，請使用 nm 指令分別觀看這三個目的檔與輸出的執行檔。
5.12 繼續前一題，請使用 objdump 指令分別觀看這三個目的檔與輸出的執行檔。

5.13 繼續前一題，請找出其中的符號表部分。