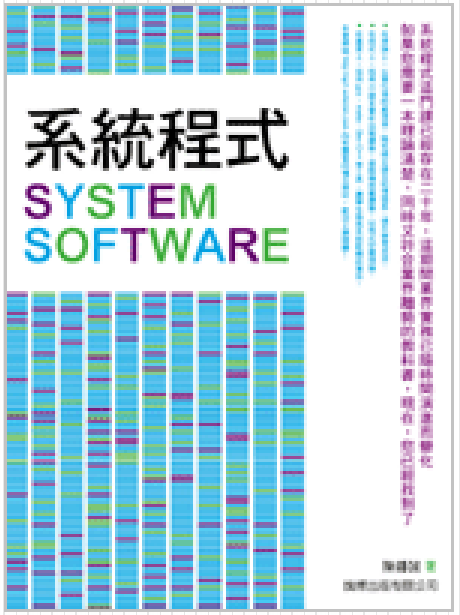
第 5 章、連結與載入

作者：陳鍾誠

旗標出版社



第 5 章、連結與載入

- 5.1　簡介
- 5.2　目的檔
- 5.3　連結器
- 5.4　載入器
- 5.5　動態連結
- 5.6　實務案例 (一)：GNU 連結工具
- 5.7　實務案例 (二)：目的檔格式 – a.out
- 5.8　實務案例 (三)：Linux 的動態載入技術

5.1 簡介

- 目的檔
  - 一種由程式機器碼與資料碼所組成的格式化檔案結構
  - 組譯時若有外部標記
    - 則必須保留這些標記以代連結時進行處理。
    - 這種保留外部標記的機器碼檔，就稱為目的檔。
- 執行檔
  - 目的檔連結完成後，就會輸出可執行檔。
- 連結器
  - 將許多目的檔連結成一個執行檔的程式
- 載入器
  - 將可執行檔載入到記憶體當中執行的程式。

執行檔

- Windows
  - 像是 test.exe，csc.exe，…
- Linux
  - 像是 a.out，test.o 等。

組譯報表

- 範例 5.1 的程式功能
  - 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	000000030000000700000004
0044	aptr:	WORD	a	00000038
0048	sum:	WORD	0	00000000

映像檔：最簡單的目的檔

- PIC (Position Independent Code) 目的碼
  - 圖 5.1 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。

圖 5.1 範例 5.1 的目的檔

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

簡單的載入器演算法

- 步驟：
  - 1：將目的檔載入到記憶體
  - 2：將程式計數器設為載入起始點

- 演算法

圖 5.2 簡單的載入器演算法

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

檔載入到記憶體後的結果

- 對於上述的範例而言
  - 直接將整塊目的檔搬入記憶體即可。

- 載入結果如圖 5.3 所示

▶圖 5.3 圖 5.1 的目的檔載入到記憶體後的結果

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

5.2 目的檔

具有交互引用的 C 語言程式

- 在範例 5.2 中
  - stack, top 等變數有外部引用的情況。
  - push(), pop() 等函數也有外部引用的情況。

▶範例 5.2 具有交互引用的 C 語言程式 (實作堆疊功能)

StackType.c

```
int stack[128];
int top = 0;
```

StackFunc.c

```
extern int stack[];
extern int top;

void push(int x) {
  stack[top++] = x;
}

int pop() {
  return stack[--top];
}
```

StackMain.c

```
extern void push(int x);
extern int pop();

int main() {
  int x;
  push(3);
  x= pop();
  return 0;
}
```

具有交互引用的組合語言

範例 5.3 具有交互引用的組合語言(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
```

分段

- .bss：
  - 是 Block Started by Symbol 的簡稱
  - 是儲存未初始化全域變數的區段。
- .text
  - 內文段 (或稱程式段), 用來儲存程式的指令碼。
- .data
  - 資料段, 用來儲存已初始化的全域變數。
- 說明：通常在目的檔中也會分成這些段落。

標記

- .global
  - 全域標記, 可以供外部的程式引用
  - 範例：
    - 變數：global stack, global top
    - 函數：global push, global pop, global main

- .extern
  - 外部標記, 引用其他程式的標記時使用
  - 範例：
    - 變數： extern stack, extern top
    - 函數： extern push, extern pop

連結器的功能

- 將許多目的檔連結成一個檔案
- 處理外部引用，進行重定位

StackType 的組合語言及目的碼

- 本文目的碼中的簡寫
  - B (BSS 段)
  - D (DATA 段)
  - T (TEXT 段)

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		BSS 段開始
	.global stack		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 的位址爲 B0000
	.data		DATA 段開始
	.global top		top 是全域變數
D 0000	top WORD 0	D(00000000),S(D,0000,top)	top 編碼爲 D(00000000)
			top 的位址是 D0000

StackFunc 的組合語言及目的碼 (1)

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 爲外部變數
	.extern top	S(U,,top)	top 爲外部變數
	.global push		push 爲全域變數
	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)	

StackMain 的組合語言及目的碼

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 爲外部變數
	.extern pop		pop 爲外部變數
	.global main		main 爲全域變數
	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

StackFunc 的組合語言及目的碼 (2)

	.global pop		pop 爲全域變數
	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)	

目的碼檔中的各類記錄

- T：Text (內文段、程式段)
- D：Data (資料段)
- B：BSS (位初始化資料段)
- M：Modification (重定位記錄)
- S：Symbol (符號記錄)

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

記錄的儲存格式

- 每種段落有不同的儲存格式
  - T：Text (內文段、程式段)
  - D：Data (資料段)
  - B：BSS (位初始化資料段)
  - M：Modification (重定位記錄)
  - S：Symbol (符號記錄)

- 盡可能用代號，而不是用名稱 (字串)
  - 字串名稱儲存在字串表中，在記錄內使用字串代號

重定位記錄 (M 記錄)

- C 語言的結構定義

圖 5.5 重定位記錄 (M 記錄) 的表示法

```
#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)
```

- M 記錄的範例

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

符號記錄 (S 記錄)

- C 語言的結構定義

圖 5.6 符號記錄 (S 記錄) 的表示法

```
#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)
```

- S 記錄的範例

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

字串表

- 在目的檔中，會盡可能儲存記錄代號，而非字串
- 因此, 通常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。

圖 5.7 字串表的結構

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							.

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 爲範例。

5.3 連結器

- 將許多個目的檔連結成
  - 一個可執行檔
  - 函式庫 (Library)
  - 動態函式庫 (DLL)

- 連結器的動作
  - 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
  - 進行區段合併的動作
    - 內文段 (.text)、資料段 (.data) 與 BSS (.bss) 段合併
  - 更新符號表與修改記錄

連結器的功能

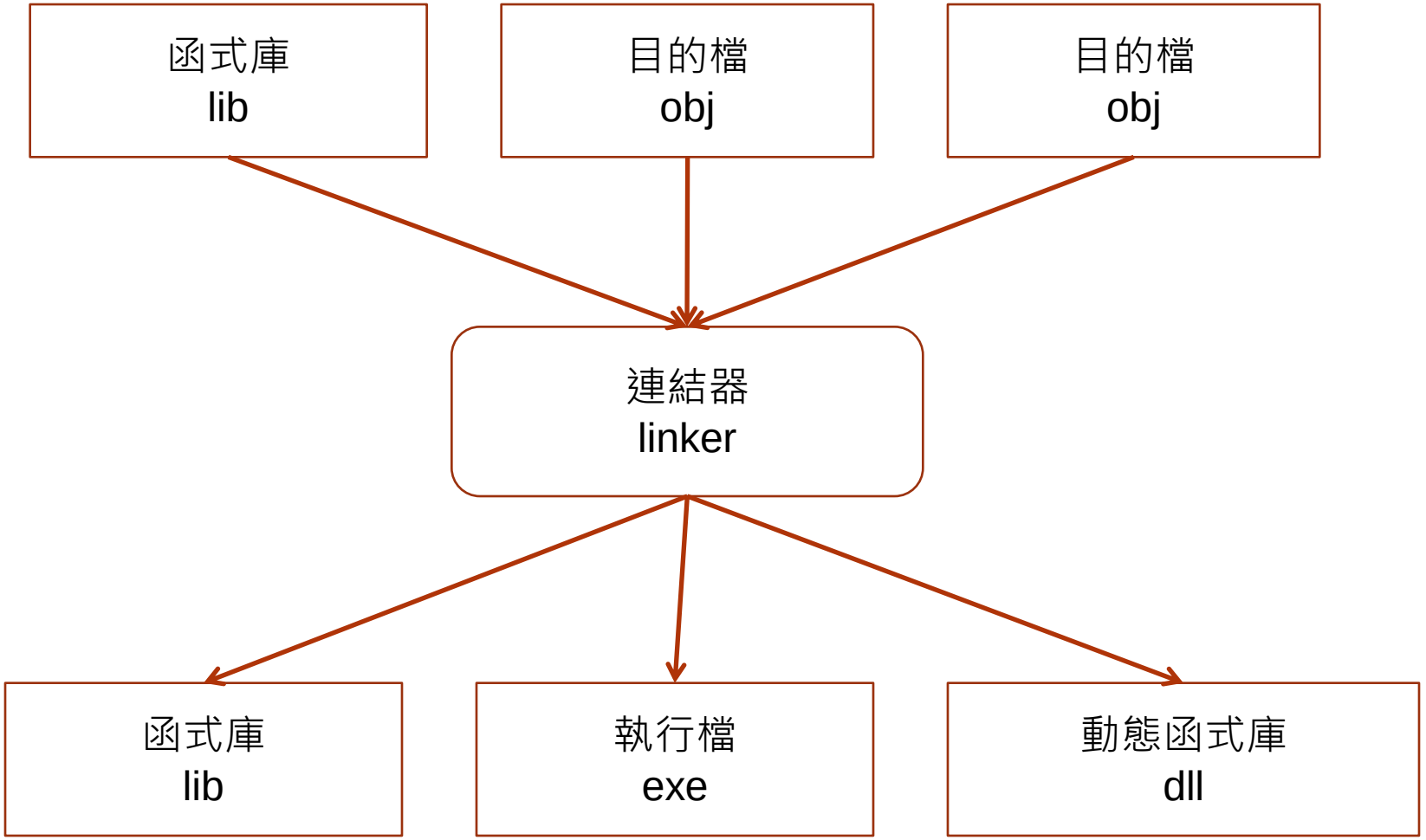


圖 5.9 連結器的輸入與輸出

區段合併

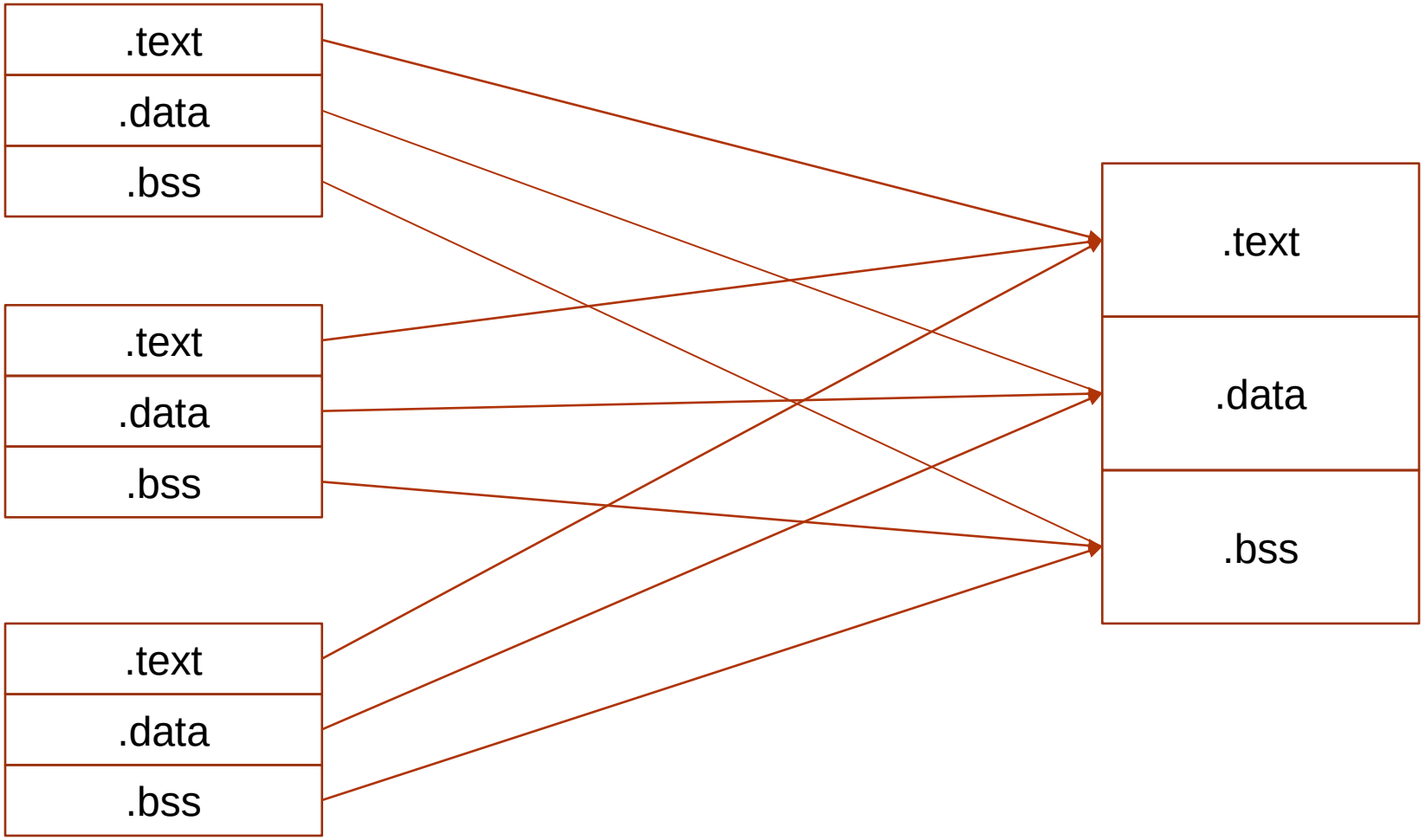


圖 5.10 連結器的功能 – 區段合併

連結的範例

1. 消除外部引用,確定外部變數的位址
2. 進行區段合併
3. 更新符號表與修改記錄

指令：ld -o stack.exe StackFunc.o StackType.o StackMain.o

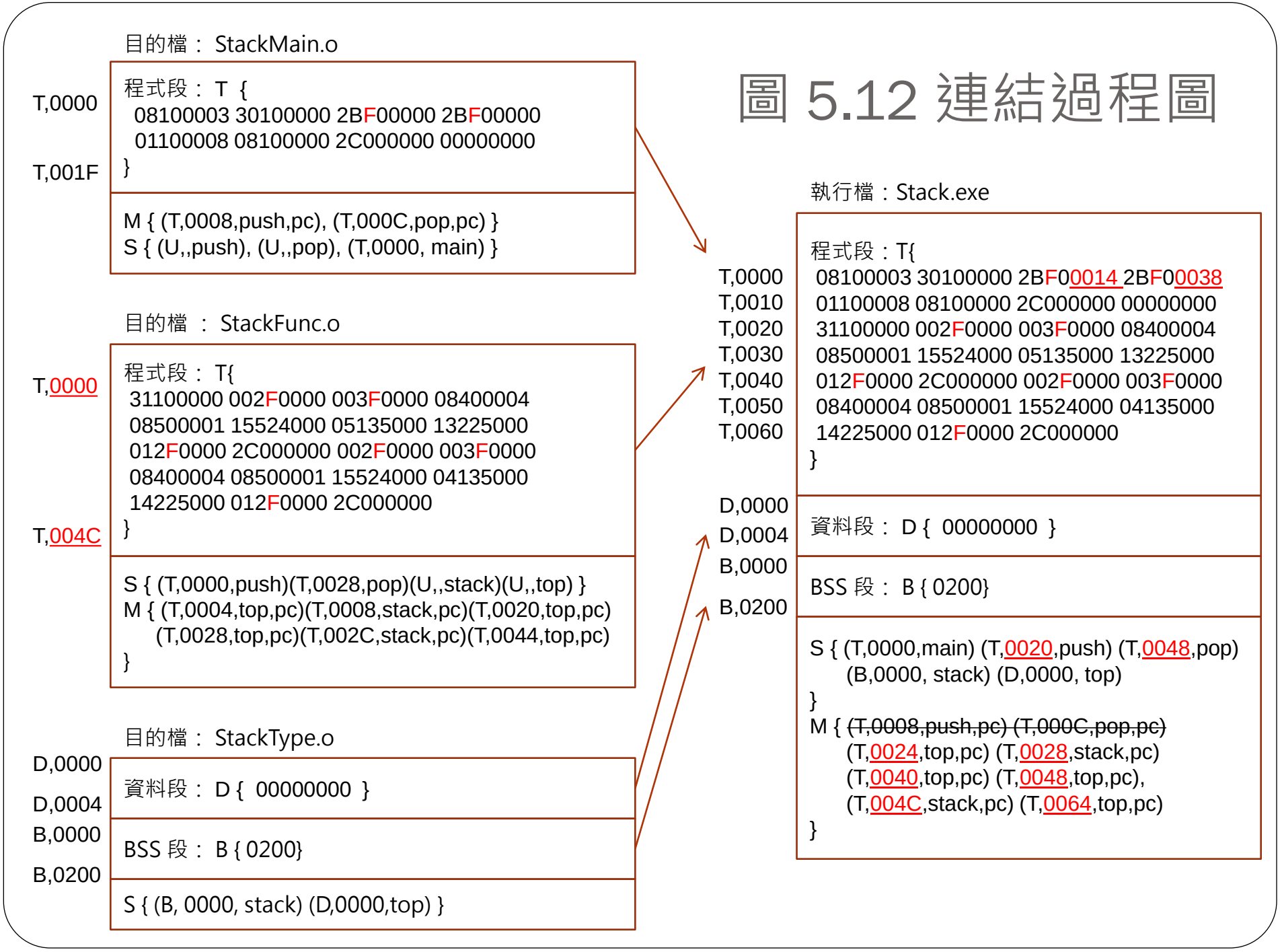
說明：連結 StackFunc.o StackType.o StackMain.o 三個檔案以形成執行檔 Stack.exe

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

圖 5.12 連結過程圖


目的檔： StackMain.o
T,0000
程式段： T {
08100003 30100000 2BF00000 2BF00000
01100008 08100000 2C000000 00000000
T,001F
}
M { (T,0008,push,pc), (T,000C,pop,pc) }
S { (U,,push), (U,,pop), (T,0000, main) }
目的檔 ： StackFunc.o
T,0000
程式段： T{
31100000 002F0000 003F0000 08400004
08500001 15524000 05135000 13225000
012F0000 2C000000 002F0000 003F0000
08400004 08500001 15524000 04135000
14225000 012F0000 2C000000
T,004C
}
S { (T,0000,push)(T,0028,pop)(U,,stack)(U,,top) }
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)
}
目的檔： StackType.o
D,0000
資料段： D { 00000000 }
D,0004
B,0000
BSS 段： B { 0200}
B,0200
S { (B, 0000, stack) (D,0000,top) }
執行檔：Stack.exe
T,0000
T,0010
T,0020
T,0030
T,0040
T,0050
T,0060
程式段：T{
08100003 30100000 2BF00014 2BF00038
01100008 08100000 2C000000 00000000
31100000 002F0000 003F0000 08400004
08500001 15524000 05135000 13225000
012F0000 2C000000 002F0000 003F0000
08400004 08500001 15524000 04135000
14225000 012F0000 2C000000
}
D,0000
D,0004
資料段： D { 00000000 }
B,0000
B,0200
BSS 段： B { 0200}
S { (T,0000,main) (T,0020,push) (T,0048,pop)
(B,0000, stack) (D,0000, top)
}
M { (T,0008,push,pc) (T,000C,pop,pc)
(T,0024,top,pc) (T,0028,stack,pc)
(T,0040,top,pc) (T,0048,top,pc),
(T,004C,stack,pc) (T,0064,top,pc)
}

連結器的演算法

▶圖 5.13 連結器的演算法

連結器的演算法

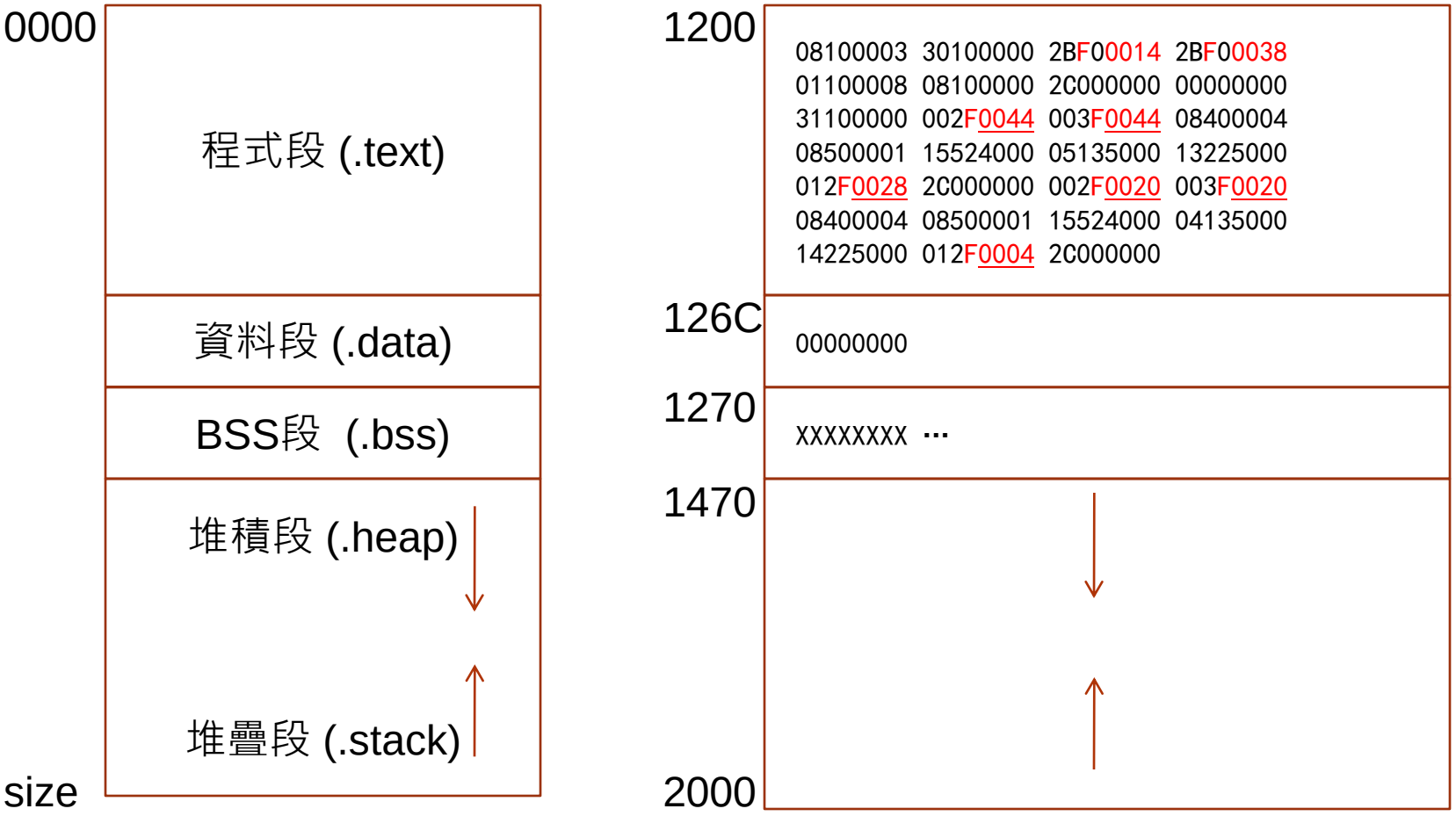
```
Algorithm Linker
Input objFileList
Output exeFile
  secTable = new SectionTable(T, D, B, …);
  foreach objFile in objFileList
    foreach S_record s in objFile
      if s.type is not 'U'
        s.address += secTable[s.name].size
        exeFile.writeS_Record(s)
    foreach M_record m in objFile
      m.address += secTable[m.section].size
      exeFile.writeM_Record(m)
    foreach section e in objFile
      secTable[e.name].size += e.size
      exeFile.writeSection(e);
  exeFile.modifyHeader(secTable)
End Algorithm
```

說明

```
連結器的演算法
輸入為一群目的檔
輸出為執行檔
建立分段表
對於每一個目的檔 objFile
    對於每個符號記錄 s
        如果 s 不是未定義符號
            更新符號 s 的位址
            輸出 s 記錄到執行檔中
    對於每一筆修改記錄 m
        修正 m 記錄的位址
        輸出 m 記錄到執行檔中
    對於每一個分段 e
        加入 objFile 的分段大小
        輸出分段到執行檔
修正執行檔表頭
```

5.4　載入器

執行檔 的記憶體配置情況



(a) 載入到記憶體後的分段情況　　(b) 執行檔 Stack.exe 被載入記憶體後的情況

圖 5.14 執行檔Stack.exe 的記憶體配置情況

執行檔 載入記憶體後的情況

圖 5.15 執行檔Stack.exe載入記憶體後的情況

執行檔 載入記憶體後的情況

位址	記憶體內容			
1200	08100003	30100000	2BF00014	2BF00038
1210	01100008	08100000	2C000000	00000000
1220	31100000	002F0044	003F0044	08400004
1230	08500001	15524000	05135000	13225000
1240	012F0028	2C000000	002F0020	003F0020
1250	08400004	08500001	15524000	04135000
1260	14225000	012F0004	2C000000	00000000
1270	XXXXXXXX	XXXXXXXX	XXXXXXXX	XXXXXXXX

Top:126C

stack:1270

符號表 ： S { (T,0000,main) (T,0020,push) (T,0048,pop)(B,0000, stack) (D,0000, top) }
修正記錄 ： M { (T,0024,top,pc) (T,0028,stack,pc) (T,0040,top,pc) (T,0048,top,pc)
(T,004C,stack,pc) (T,0064,top,pc) }

圖 5.15 執行檔Stack.exe載入記憶體後的情況

載入器的演算法

- 主要功能
  - 將各分段載入到記憶體
  - 利用修正記錄修改記憶體內容
  - 設定程式計數器，開始執行

▶圖 5.16 載入器的演算法

行號	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	

5.5 動態連結

- 靜態連結
  - 連結器必須將所有使用到的函式庫連結到執行檔中

- 動態連結
  - 函式庫可以先不需要被連結進來
  - 而是在執行到某函數時，才透過動態連結器尋找並連結函式庫
  - 可以不用載入全部的函式庫, 以節省記憶體。

動態連結器的任務

- 在需要的時候才載入動態函式庫
- 動態連結 (linking)
- 動態重定位 (relocation)

動態連結的優缺點

- 特性
  - 通常是與位置無關的程式碼 (Position Independent Code)
- 優點
  - 節省記憶體
  - 節省連結時間
  - 可以抽換函式庫
- 缺點
  - 可能造成『動態連結地獄』(DLL hell) 的困境
    - 假如新的函式庫有錯, 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。

動態連結的實作

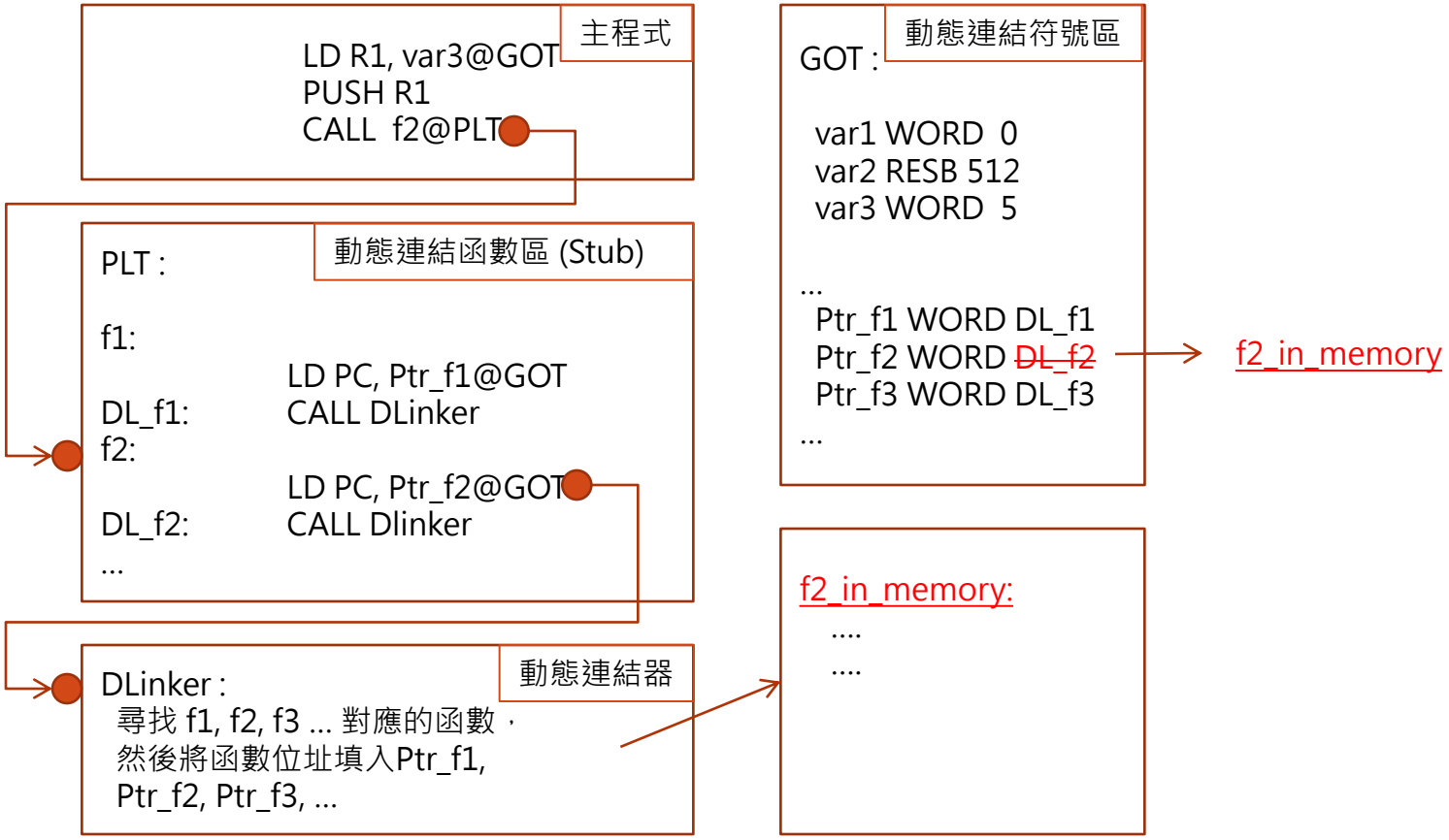


圖 5.17 動態連結機制的實作方式

動態連結：Linux v.s. Windows

- Windows
  - 動態連結檔被稱為 DLLs (Dynamic Linking Libraries)
  - 其附檔名通常為 .dll
- UNIX / Linux
  - 動態連結檔被稱為 Share Objects
  - 其附檔名通常是 .so

動態載入

- 說明
  - 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。
- 範例
  - 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。

動態載入：Linux v.s. Windows

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

5.6　實務案例 (一)：GNU 連結工具

- GNU 的連結工具
  - 主要為 ld (也可用 gcc 代替，gcc 會自動呼叫 ld)

- GNU 的目的檔工具
  - objdump：觀察目的檔
  - objcopy：目的檔複製修改
  - nm : 符號表列印

GNU 連結工具

表格 C.2 GNU 的工具與用法

工具	工具類型	說明
gcc	C 語言編譯器 GNU C Compiler	範例：gcc hello.c -o hello.o
as[6]	組譯器 Assembler	範例：as hello.s -o hello.o 說明：將 hello.s 組譯為 hello.o
ld[7]	連結器 Linker	範例：ld -o abc.o a.o b.o c.o 說明：將 a.o, b.o, c.o 連結成執行檔 abc.o
ar	函式庫製作 Archive	範例：ar -r libabc.a a.o b.o c.o 說明：將 a.o, b.o, c.o 包裝成函數庫 libabc.a
nm	name mangling[8] 目標檔中的符號	範例：nm hello.o 說明：看 hello.o 目標檔的符號表
objdump	Object File Dump 目標檔傾印	範例：objdump -x hello.o 說明：查看目標檔資訊
objcopy	Object File Copy 複製/轉換目標檔	範例：objcopy -O binary hello.elf hello.bin 說明：將 elf 檔轉換為 binary 檔
strip	Strip 去除除錯資訊	範例：strip a.o 說明：把 a.o 當中的符號表與除錯資訊去除
strings	觀看字串表	範例：strings a.o 說明：觀看 a.o 檔中的字串表, 會顯示符號名稱與分段名稱
ltrace	追蹤函數呼叫路徑	範例：ltrace a.o 說明：追蹤函數呼叫路徑 (在 Cygwin 中沒有)

C 語言的連結範例

- 三個 C 語言程式

▶範例 5.7 具有交互引用的 C 語言程式 (實作堆疊功能)

StackType.c

```
int stack[128];
int top = 0;
```

StackFunc.c

```
extern int stack[];
extern int top;

void push(int x) {
  stack[top++] = x;
}

int pop() {
  return stack[--top];
}
```

StackMain.c

```
extern void push(int x);
extern int pop();

int main() {
  int x;
  push(3);
  x= pop();
  return 0;
}
```

- 連結過程

▶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	編譯 StackType.c 為目的檔
C:\ch05>gcc -c StackFunc.c -o StackFunc.o	編譯 StackFunc.c 為目的檔
C:\ch05>gcc -c StackMain.c -o StackMain.o	編譯 StackMain.c 為目的檔
C:\ch05>gcc StackMain.o StackFunc.o StackType.o -o stack	連結成為執行檔

將 C 轉為組合語言以便觀察

▶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

三個組合語言程式

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
  movl _top, %eax
  movl _stack(,%eax, 4), %eax
  popl %ebp
  ret
```

檔案：StackMain.s

```
  .file "StackMain.c"
  .def  ___main;.scl 2;.type 32;.endef
  .text
.globl _main
  .def  _main; .scl 2;.typ 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
 .def_push; .scl 3;.type 32;.endef
```

使用 nm 指令觀察符號表

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

使用 size 指令觀察分段大小

▶範例 5.12 用 size 指令檢視目的檔中分段大小

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	StackFunc.o 主要為程式段
C:\ch05>size StackType.o text data bss dec hex filename 0 0 16 16 10 StackType.o	StackType.o 主要為 BSS 段
C:\ch05>size StackMain.o text data bss dec hex filename 80 0 0 80 50 StackMain.o	StackMain.o 主要為程式段
C:\ch05>size Stack.exe text data bss dec hex filename 2548 700 704 3952 f70 Stack.exe	Stack.exe 各段均有

書籍內文有誤，在此修正

製作靜態函式庫

▶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 rw-rw-rw- 0/0 324 Apr 04 18:45 2010 StackType.o rw-rw-rw- 0/0 502 Apr 04 18:46 2010 StackFunc.o	顯示函式庫 libstack.a 的內容 包含 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

範例 5.14 使用 objdump 觀察目的檔

```
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
```

專案建置工具

專案建置檔 - Makefile

定義：
類似組合語言的 EQU
或 C 語言的 #define

動作：
從第一個標記開始，
以觸發驅動的方式
展開執行

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)
```

專案建置的指令與過程

▶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

Makefile 觸發的過程

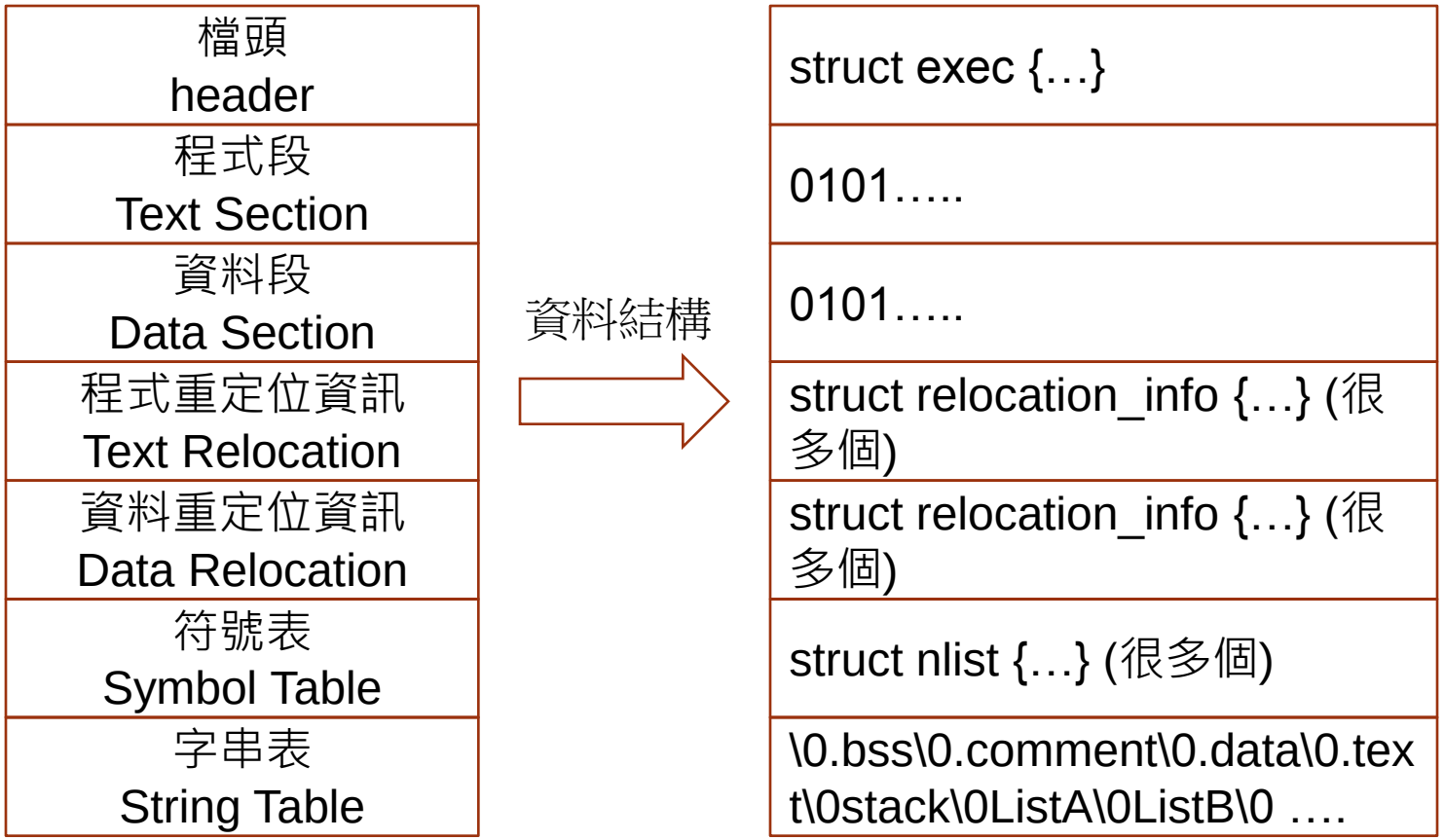
圖 5.18 範例 5.16 中 make 指令的觸發樹與執行過程

```
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 .
```

5.7　實務案例 (二)：目的檔格式 – a.out

- Linux 的目的檔
  - 早期：a.out 格式
  - 現在：ELF 格式
- Windows 的目的檔
  - 早期：.com 檔案
  - 現在：PE/COFF 格式

a.out 檔案格式



(a) a.out 檔案的格式

(b) a.out 各區塊對應的資料結構

圖 5.19 目的檔a.out各區段所對應的資料結構

載入的過程

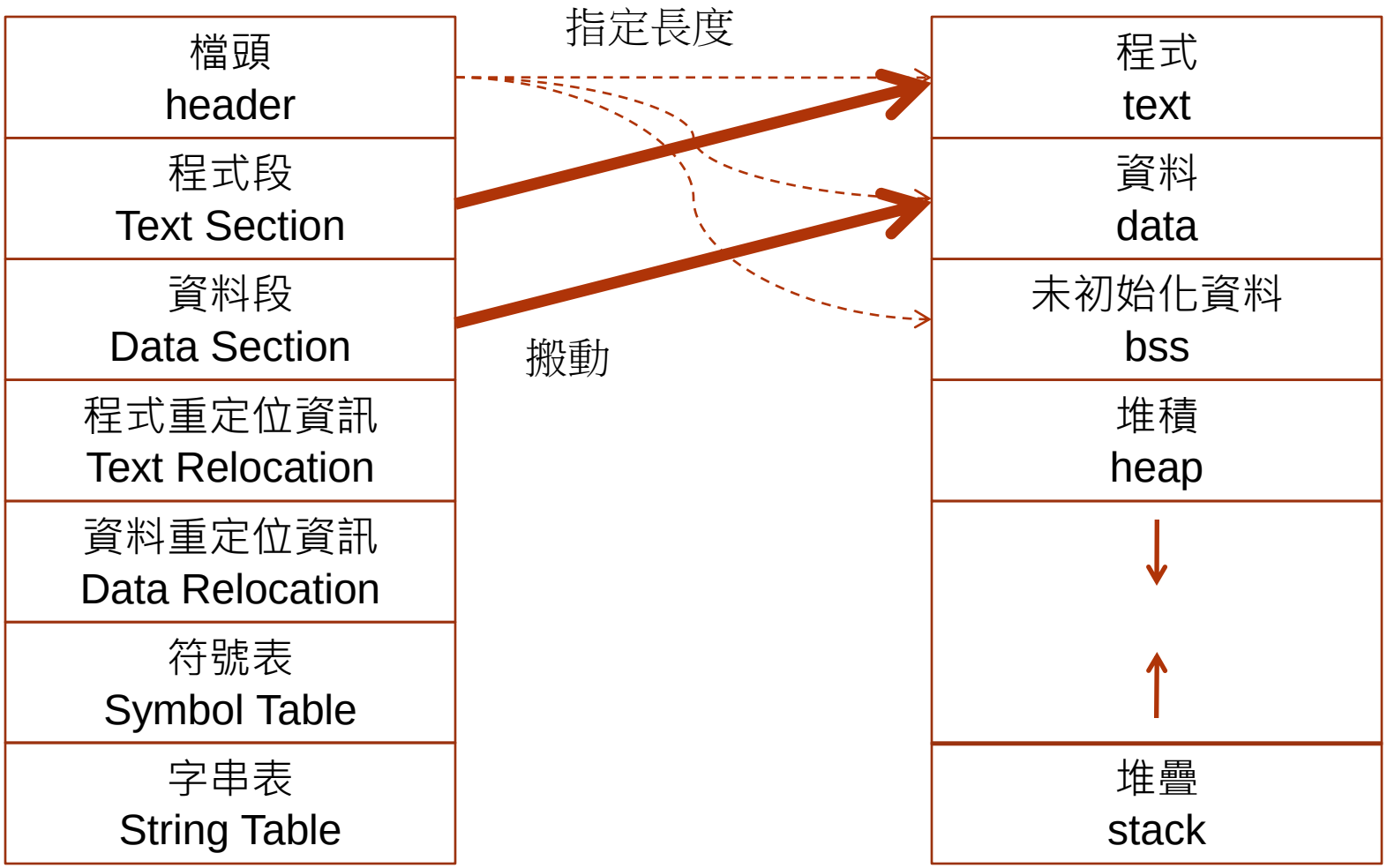


圖 5.21 目的檔 a.out 的載入過程

a.out 的檔案結構

圖 5.20 目的檔 a.out 的資料結構 (以 C 語言定義)

```
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 相對旗標
 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
 } n_un;                            // 位元組偏移值
 unsigned char n_type;              // 符號類型N_ABS/N_TEXT/N_DATA/
 char n_other;                      // N_BSS等通常不用
 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。
};
```

5.8 實務案例 (三)：Linux 的動態載入技術

- 說明
  - 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。

- Windows 與 Linux 的動態載入技術

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

動態載入的範例

▶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

```
#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;
}
```

說明

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

習題

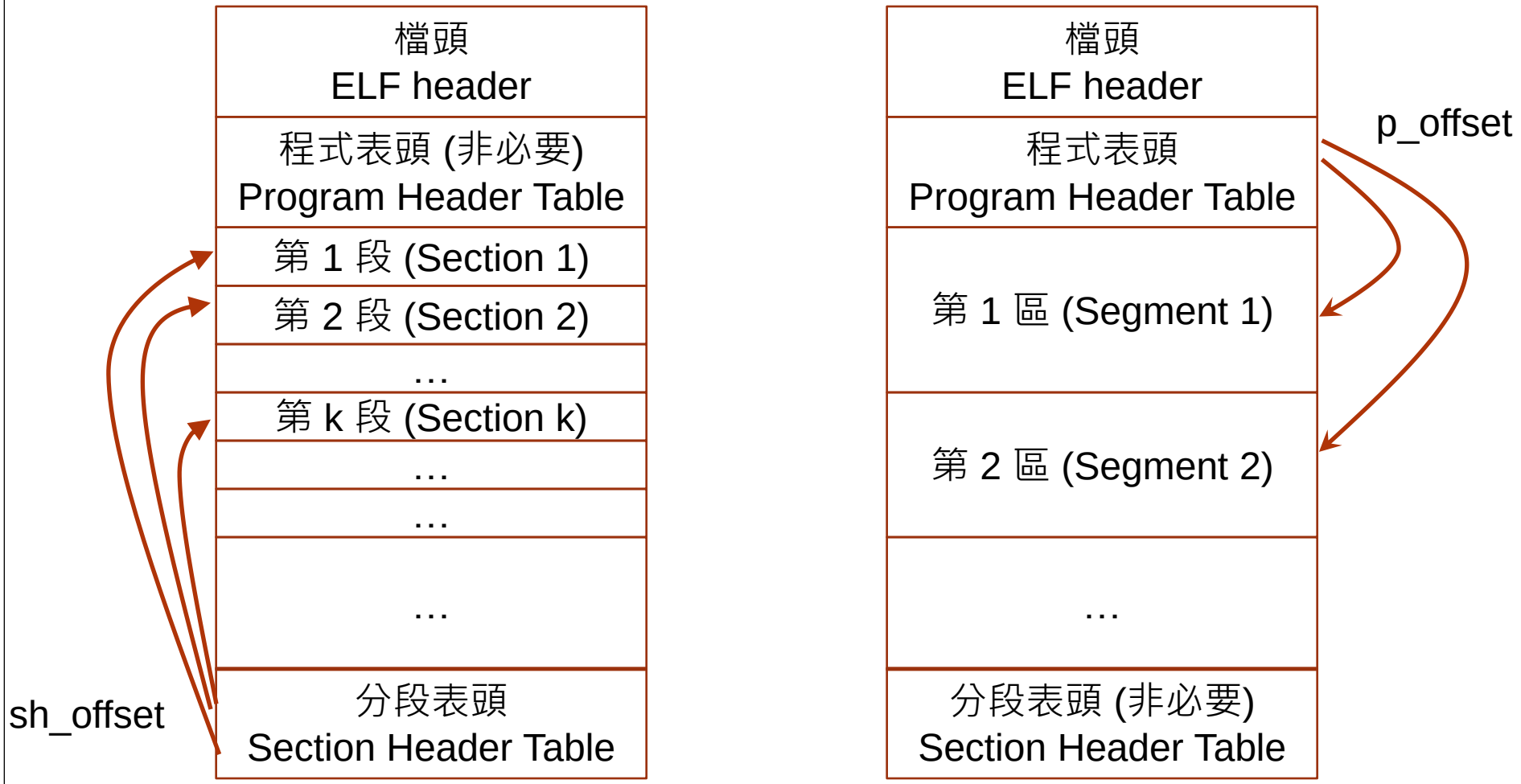
- 5.1 請說明連結器的輸入、輸出與功能為何？
- 5.2 請說明載入器的功能為何？
- 5.3 請說明 CPU0 組合語言當中的 .text, .data 與 .bss 等假指令的用途為何？
- 5.4 請說明 CPU0 組合語言當中的 .global 與 .extern 等假指令的用途為何？
- 5.5 請說明 CPU0 目的檔中的 T, D, B, S, M 等記錄各有何用途？
- 5.6 請說明連結器是如何處理外部引用問題的？
- 5.7 請說明目的檔中符號表的用途？

習題 (續)

- 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。

補充教材

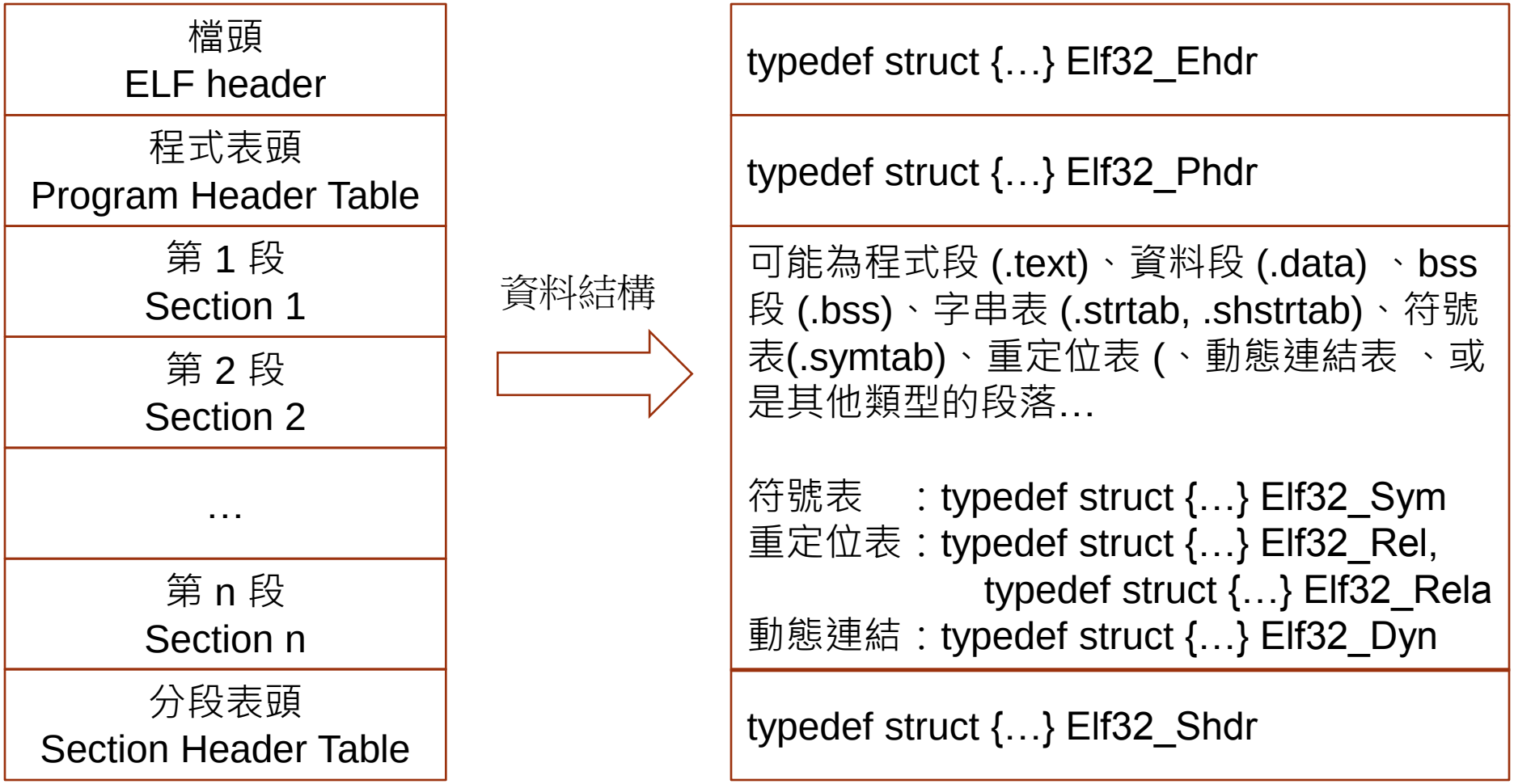
目的檔 ELF 的兩種不同觀點



(a) 連結時期觀點 (Linking View)

(b) 執行時期觀點 (Execution View)

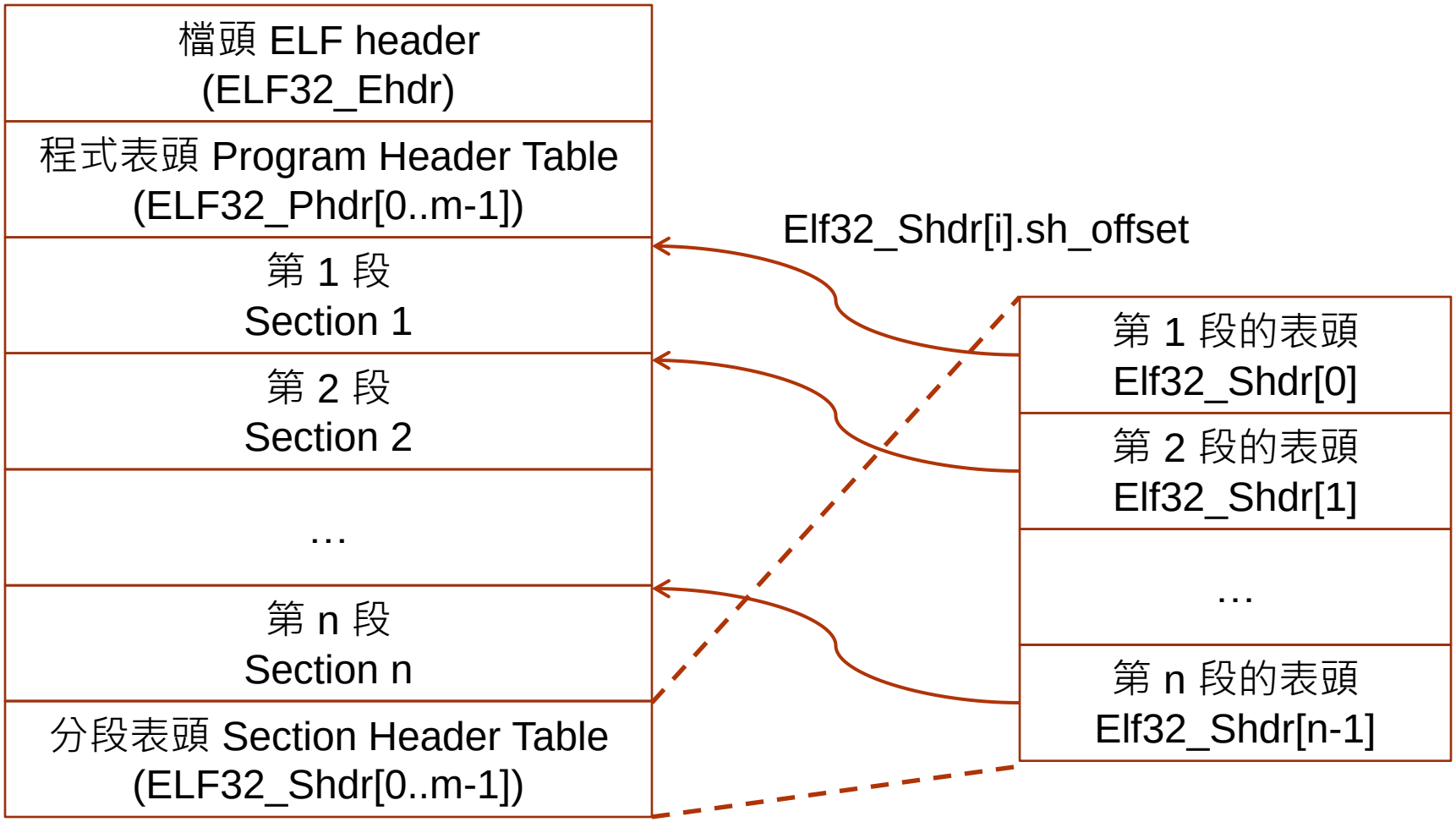
目的檔ELF的資料結構



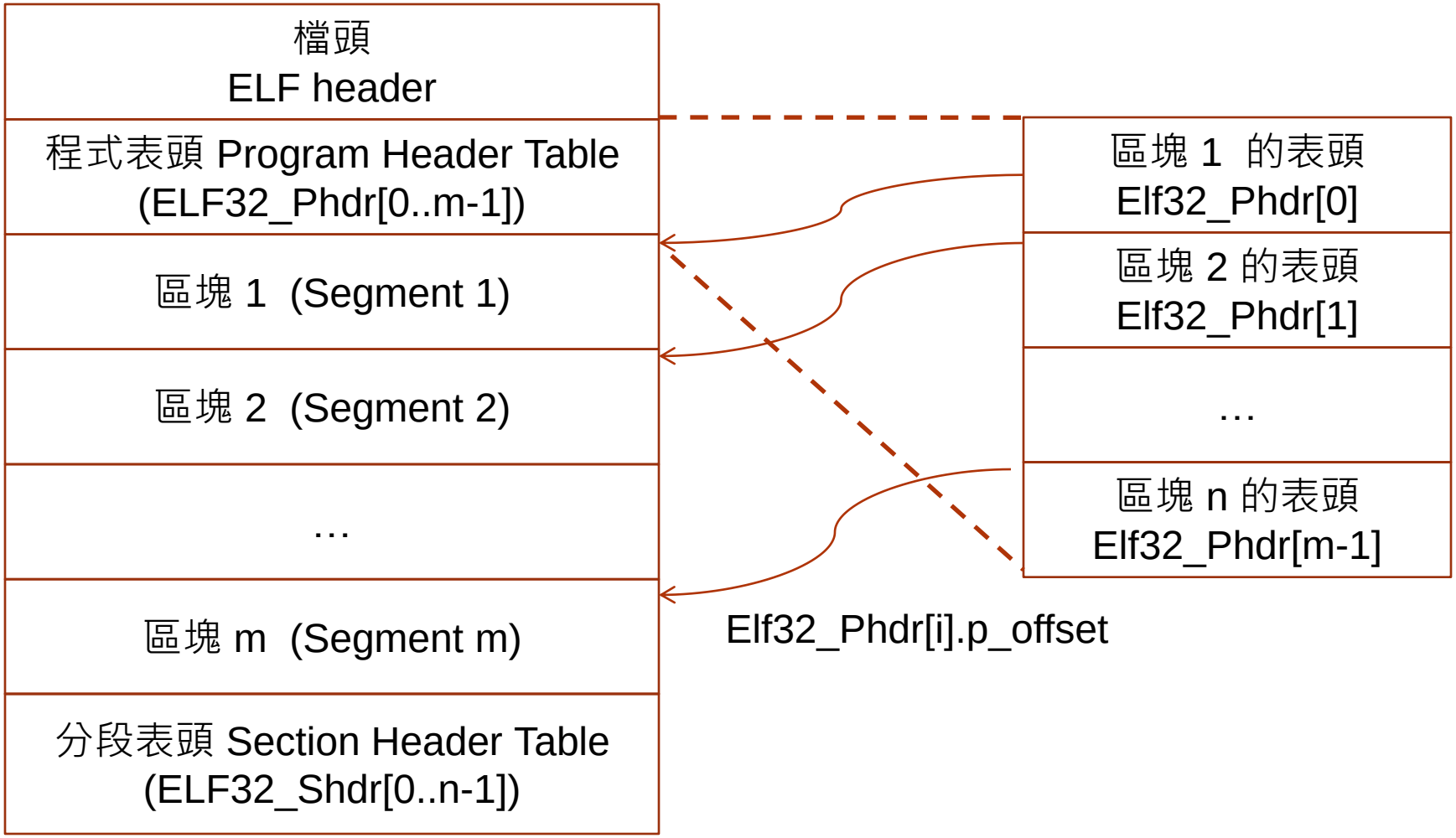
(a) ELF 的檔案結構

(b) ELF 各區塊對應的資料結構

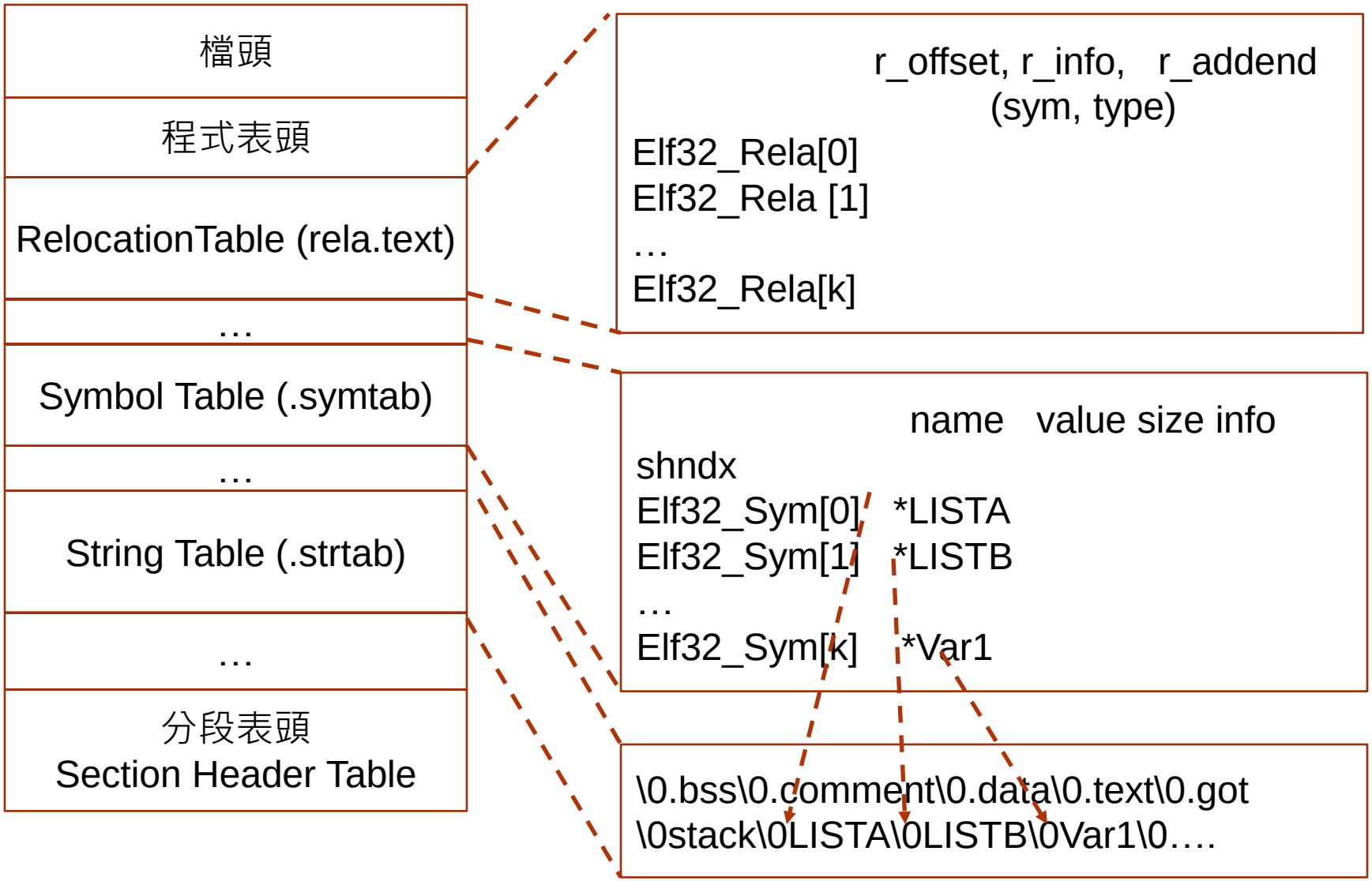
目的檔ELF的分段表頭



目的檔ELF的程式表頭



目的檔ELF中的重定位表、符號表與字串表的關連性



目的檔a.out 的格式與範例

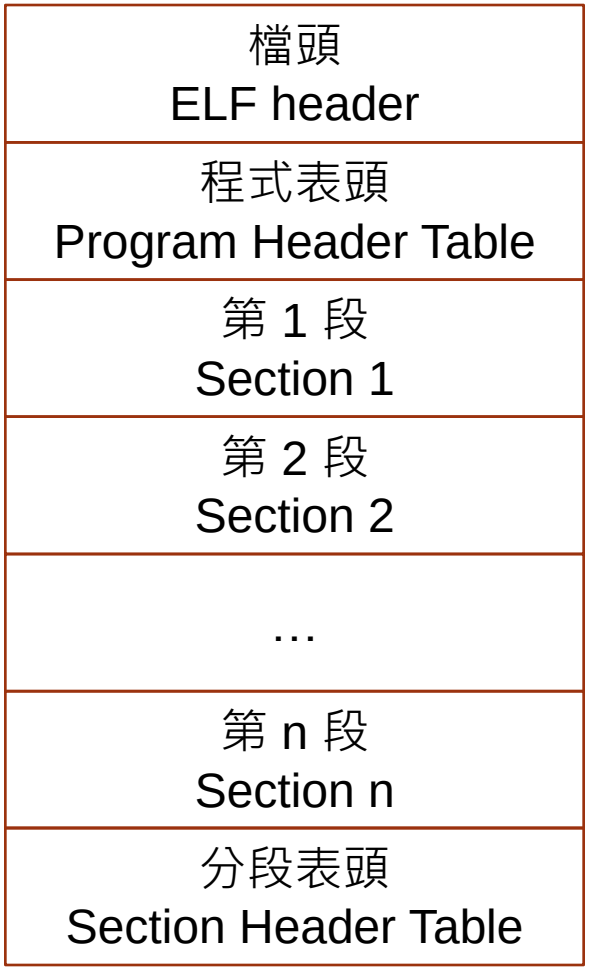
檔頭 (H 記錄)	H(StackFunc.s, 各段的長度)
程式段 (T 記錄)	T{31100000 00200000 00300000 08400004 … }
資料段 (D 記錄)	
程式重定位資訊 (M 記錄)	M { (0004,top,pc) (0008,stack,pc) (0020,top,pc) …}
資料重定位資訊 (M 記錄)	
符號表 (S 記錄)	S { (T,0000,push) (T,0028,pop) (U,,stack) (U,,top) }
字串表 String Table	.text\0.data\0.bss\0top\0stack\0push\0pop\0

(a) a.out 檔案的格式

(b) a.out 的檔案範例 (StackFunc.o)

兩種目的檔的格式 – a.out 與 ELF

(a) a.out 檔案的格式



(b) ELF 檔案的格式