載入程式

- 載入程式 (Loader)之功能 —

  原始程式經由編譯程式或組合程式處理後，所產生的目的程式中，將會含有目的程式要如何載入記憶體的相關資訊。載入程式就根據這些資訊，將目的程式載入至記憶體中的適當位址，準備執行程式。

載入程式主要的四項功能爲：

- (1) 配置 (Allocation)：配置目的程式於記憶體中所需之空間及位置。
- (2) 連結 (Linking)：將數個目的程式結合在一起，並且處理目的程式間符號參考的問題。
- (3) 重定位 (Relocation)：調整目的程式中所有與位置有關的目的碼，使得目的程式能夠載入與原位址不盡相同之記憶體位置中。
- (4) 載入 (Loading)：將目的程式 (機器指令與資料) 真實的搬移至所配置的記憶體中準備執行程式。

載入程式的處理過程：

- (1) 輸入：

  a. 目的程式。

  b. 由編譯程式或組合程式所提供有關於重定位的資訊。

- (2) 處理過程：

  a. 讀入表頭記錄。

  b. 確認程式的名稱與長度。

  c. 讀入每一列本文記錄，直至讀到結束記錄。

  1. 若該列目的碼爲字元型態，則將之轉換成機器內部表示型態。

  2. 將該列目的碼載入指定的記憶體位址中。

  d. 讀取結束記錄之後所緊接著的程式起始位址置，並跳至記憶體中該位址開始執行程式。

載入程式之類型

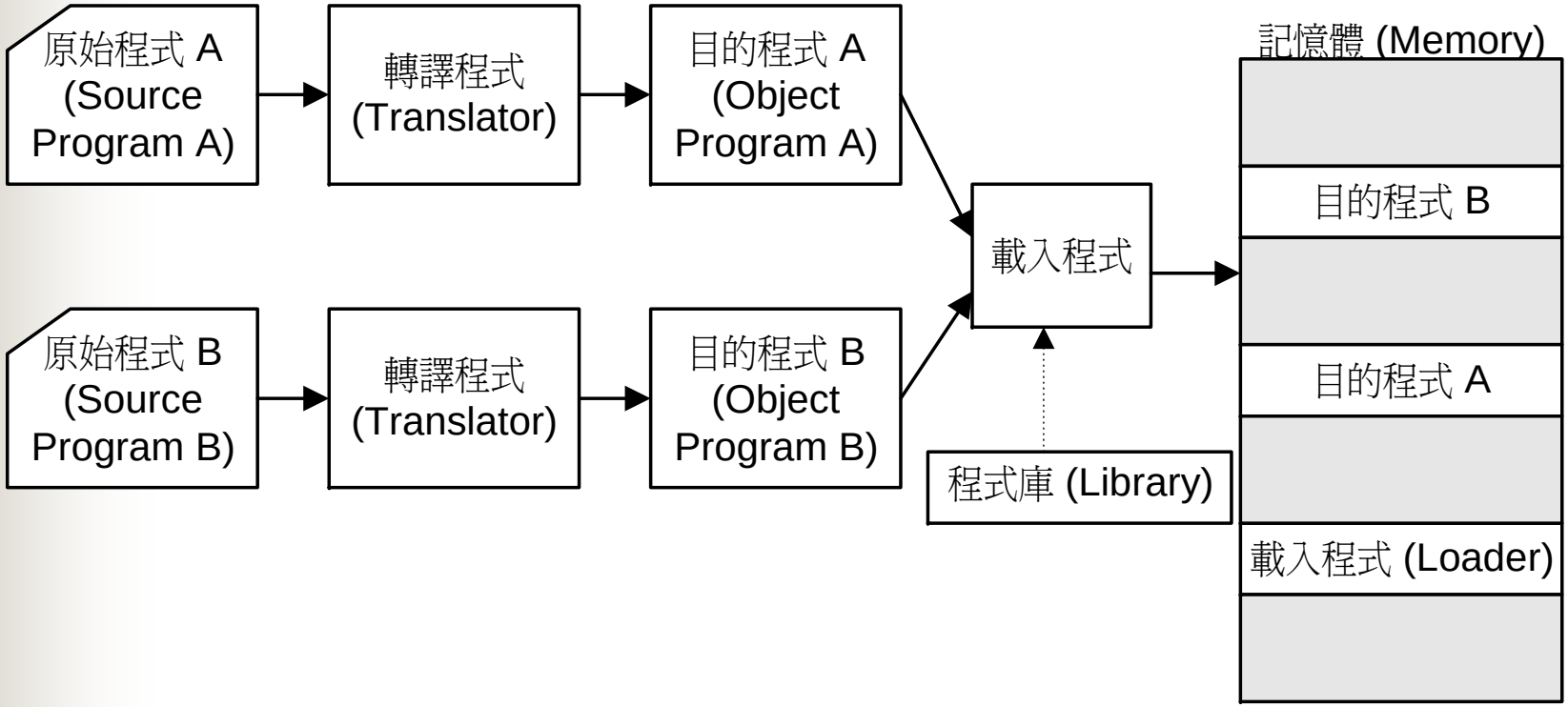
載入程式的種類可歸類如下：

- 1. 一般式載入程式
- 2. 組譯並載入式載入程式或編譯並載入式載入程式
- 3. 絕對位址式載入程式
- 4. 重定位址式載入程式或稱相對位址式載入程式：
  - (1) 二元符號式載入程式
  - (2) 直接連結式載入程式

1. 一般式載入程式 —

- (1) 處理方式：

  將經由轉譯程式轉譯後所產生的目的程式儲存於次儲存體 中 (如：磁碟，磁帶)，當要執行程式時，再藉由載入程式將目的程式載入記憶體中。


原始程式 A
(Source
Program A)
轉譯程式
(Translator)
目的程式 A
(Object
Program A)
原始程式 B
(Source
Program B)
轉譯程式
(Translator)
目的程式 B
(Object
Program B)
載入程式
程式庫 (Library)
記憶體 (Memory)
目的程式 B
目的程式 A
載入程式 (Loader)

- (2) 優點：

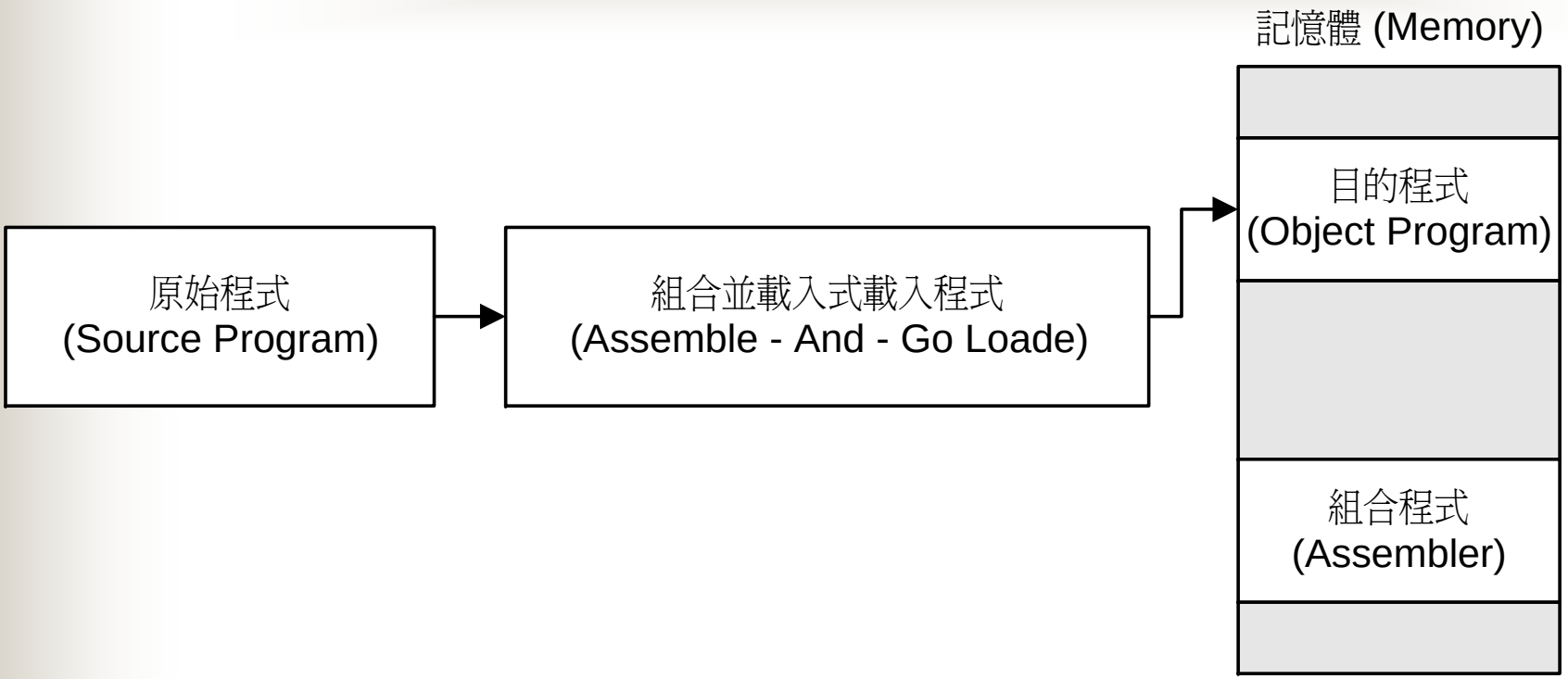
  a. 節省記憶體空間：因爲載入程式所佔用的記憶體空間比轉譯程式小得許多，所以可剩餘較大的記憶體空間供使用者利用。

  b. 節省時間：使用者要執行程式時，無須將原始程式再次的轉譯，只要將儲存於次儲存體內的目的程式載入記憶體即可。

2. 組合並載入式載入程式 —

- (1) 處理方式：

  此種載入程式在作業時，組合程式會佔用記憶體的一部份。當原始程式輸入後，組合程式直接將原始程式組譯成爲機器指令及資料置於指定的記憶體位址上。當組合程式完成原始程式的組譯工作後，隨即將控制權轉移至目的程式之起始位址，開始執行程式。


記憶體 (Memory)
原始程式
(Source Program)
組合並載入式載入程式
(Assemble - And - Go Loade)
目的程式
(Object Program)
組合程式
(Assembler)

- (2) 優點：

  由於被組譯出的機器指令與資料是直接置入記憶體內。所以，載入程式僅是一個控制權轉移指令，施行起來非常簡易。

- (3) 缺點：

  a.由於組合程式必須存放於記憶體內，佔用一部份的記憶體空間。因此，目的程式可使用的記憶體空間相對的減少了。

  b.每次要執行程式時，皆需對原始程式重新組譯，浪費時間。

  c.程式模組無法分次的個別組譯。使用者在撰寫程式時無法拆成數個模組分別撰寫、組譯與測試，必須以單一模組的方式撰寫會較不方便；即在程式設計上較缺乏彈性。

3. 絕對位址式載入程式 —

- (1) 處理方式：

  程式設計者在撰寫程式時必須在原始程式中指定目的程式要載入記憶體中的位址。若在設計程式時程式設計者必須自行負責主、副程式間的連結動作。

  組合程式將原始程式組譯，並予以適當的重定位。組譯後所產生的目的程式將被儲存於次儲存體內，等待要執行程式時，載入程式便直接將目的程式載入組合程式所指定的記憶體位址上執行。

  此種載入程式避免將組合程式存放於記憶體中，避免佔用過多記憶體的缺點。

- (2) 功能：

  在絕對位址式載入程式中，記憶體的配置與連結是由程式設計者自行負責，重定位則由組合程式負責，而載入程式僅負責將目的程式由次儲存體中載入主記憶體中執行。

- (3) 優點：

  a. 設計簡單。

  b. 組合程式不必置於記憶體內，故不佔用記憶體空間。

  c. 每次執行程式時無需再對原始程式重新做組譯的動作，可直接用目的程式執行。

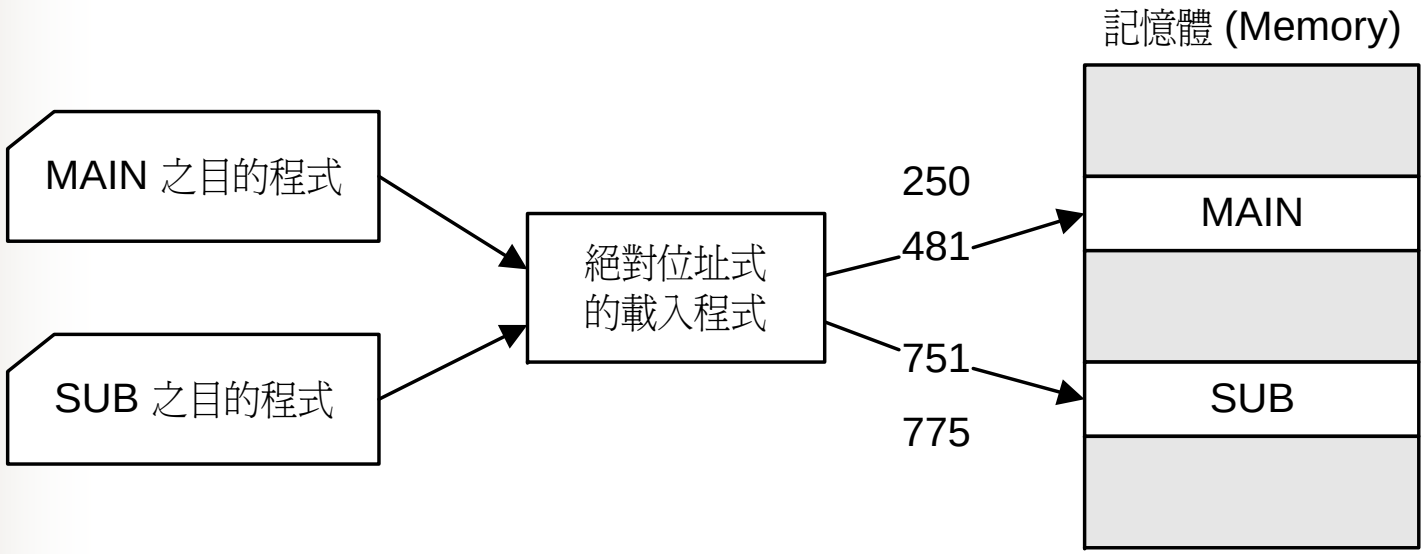
- (4) 缺點：

  a. 程式設計者於程式撰寫時，必須自行設定目的程式於主記憶體之絕對位址。

  b. 除了主程式外，尚有其它的副程式，程式設計者必須小心的安排所有程式的絕對位址，以避免位置重疊，並且要自行負責程式間的連結。

範例：

```
          (主程式)
MAIN      START     250          絕對位址值
          BLAR      12 , 4
          USING     MAIN + 2 , 12
          ⋮
          L         15 , ASUB    呼叫 SUB 副程式
          BLAR      14 , 15
          ⋮
ASUB      DC        F '751'      SUB 副程式位址 (做連結的動作)
          END
          (副程式)
SUB       STAR      751          絕對位址值
          USING     * , 15
          ⋮
          BR        14           返回主程式的位址值
          END
```


記憶體 (Memory)
MAIN 之目的程式
SUB 之目的程式
絕對位址式
的載入程式
250
481
751
775
MAIN
SUB

- 4. 重位址式載入程式 —

  此種載入程式會負責處理記憶體配置 及連結的工作，無需再由程式設計者自行負責。而且當某一副程式有所更改時，不必將整個程式重新組譯，僅需對該更動的副程式重新做組譯即可。

- (1) 二元符號式載入程式 (BSS Loader) —

  a. BSS Loader 能自行負責記憶體配置 ，連結，重定位以及載入四項工作。一般 BSS 載入程式適用於指令長度固定且指令格式爲直接定址的機器上。BSS 載入程式允許有多個程式段，但僅允許單一個資料段，此資料段即是共用資料區。

b. 重定位組合程式對每個程式段做組譯，並且必須提供下列的資訊供 BSS載入程式使用：

- 組合程式傳送給BSS載入程式之目的程式前端附加有“轉移向量”，在轉移向量內儲存著此目的程式所有會參考到的副程式名稱。
- 原始程式經組譯後產生的機器碼。
- 重定位位元 ：用以決定位元組 (每 2 個位元組爲一個單位) 是否需要做重定位。該位元爲 1 時，表示該 2 個位元組需做重定位；該位元爲 0 時，則表示爲絕對位址無需再做重定位。
- 程式的長度。
- 轉移向量的長度。

c. 處理方式：

- BSS 載入程式將目的程式載入記憶體中，並逐一檢查重定位位元，將所有必須做重定位的位元組予以重新定位。再將轉移向量內所被參考到的副程式載入記憶體內，並逐一檢查副程式中的重定位位元，將所有該重定位的位元組予以重新定位。再依照各副程式載入記憶體中之位址，將儲存副程式名稱的轉移向量更改爲跳躍至副程式所在位址的跳躍指令。

d. BSS 載入程式的功能：

- 以重定位位元解決重定位的問題。
- 以轉移向量解決連結的問題。
- 以程式的長度解決記憶體配置的問題。

e. BSS 載入程式的缺點：

- 轉移向量僅當作位址轉移之用 (即連結副程式)，對於如何載入外界的資料和儲存外界的資料並沒有任何幫助。
- 轉移向量增加目的程式佔用的記憶體空間。
- BSS 載入程式的資料段只允許各副程式共用，沒有個別擁有資料段的能力。

- (2) 直接連結式載入程式DLL) —

  DLL 亦是一種重定位式載入程式，是目前較常使用的一種載入程式。它解決了 BSS 載入程式的缺點，允許使用數個程式段 與數個資料段，程式可以隨意的參考其它程式段中的符號與資料。每個程式段也可以被個別的組譯。

a. 組合程式必須產生各個程式段與資料段的相關資訊供 DLL 使用，這些資訊如下：

- 程式段的長度。
- 列出讓其它程式段參考的符號，及這些參考符號在本程式段中相對於程式起始點的位址。
- 列出不在本程式段中定義，但被本程式段中被參考到的符號。
- 描述在本程式段中那些位置含有位址常數，以及這些位址常數要如何被修正的相關資訊。
- 原始程式被組譯後產生的機器碼，以及給定的相對位址。

b. 爲達到上述五點，組合程式必須產生四種型態之目的疊卡供 DLL 使用：

- 外部符號目錄卡(ESD)：
  包含所有在本程式段中定義而被其它程式段參考到的符號，以及在其它程式段中定義但是在本程式段中被參考到的符號。
- ESD 卡又包含下列三種疊卡：
- (1) 段落定義卡 (Segment Definition Cards,SD)：
  經由 START 或 CSECT 指令所產生。
- (2) 區域定義卡 (Local Definition Cards,LD)：
  經由ENTRY指令所產生。
- (3) 外部定義卡 (External Reference Cards,ER)：
  經由EXTRN指令所產生。

- 本文卡 (TXT)：
  爲目的程式之主體。包括原始程式被組譯後所產生的目的碼及其被設定之相對位址。
- 重定位及連結目錄表 (RLD)：
  含有程式中所有需要做重定位的相對位址，並提供如何做重定位的方法。
- 結束卡 (END)：
  用以指示整個目的程式的結束。但如果是主程式，則在 END 卡內含有程式開始執行之位址。

c. 在目的疊卡中，該四種卡有其一定的放置順序，依序爲 ESD 卡，TXT 卡，RLD 卡與 END 卡。

d. DLL 之優點：

- 程式設計者可以使用數個程式段及數個資料段 。
- 程式可以隨意的參考其它程式段的符號與資料。
- 各個程式段可以單獨的組譯。

e. DLL 之缺點：

- 每次要執行程式時，皆要對主程式及所參考到的副程式做記憶體配置，重定位，連結以及載入的工作，相當費時。
- DLL 所佔用的記憶體空間雖然比組合程式小，但也會佔用了相當大的空間。

範例：

	(原始程式)			(組譯後之程式)		
卡片編號				相對位址		
1.	MAIN	START				
2.		ENTRY	FINAL			
3.		ESTRN	TOTAL			
4.		BALR	12,0	0	BALR	12,0
5.		USING	*,12			
6.		ST	14,BUFFER	2	ST	14,54 (0,12)
7.		L	1,INDEX	6	L	1,46 (0,12)
8.		L	15,ATOTAL	10	L	15,58 (0,12)
9.		BALR	14,15	14	BALR	14,15
10.		ST	1,FINAL	16	ST	1,50 (0,12)

11.		L	14,BUFFER	20 26	L —	14,54 (0,12)
12.		BR	14	24	BCR	15,14
13.	LIST	DC	F '1,7,9,10,3'	28 32 36 40 44	1 7 9 10 3	
14.	INDEX	DC	A (LIST)	48	28	
15.	FINAL	DS	F	52	—	
16.	BUFFER	DS	F	56	—	
17.	ATOTAL	DC	A (TOTAL)	60	?	
18.		END		64		

- 則其所產生之疊卡如下：

ESD卡

參考卡片之編號	符號	型態	相對位址	長度
1	**MAIN**	**SD**	**0**	**64**
2	**FINAL**	**LD**	**52**	**—**
3	**TOTAL**	**ER**	**—**	**—**

TXT卡

參考卡片之編號	相對位置		目的碼
4	0	BALR	12,0
6	2	ST	14,54(0,12)
7	6	L	1,46(0,12)
8	10	L	15,58(0,12)
9	14	BALR	14,15
10	16	ST	1,50(0,12)
11	20	L	14,54(0,12)
12	24	BCR	15,14
13	28	1	
13	32	7	
13	36	9	
13	40	10	
13	44	3	
14	48	28	
17	60	0	

RLD卡

參考卡片之編號	符號	旗標	長度	相對位址
14	**MAIN**	＋	**4**	**48**
17	**TOTAL**	＋	**4**	**60**