



Bibliothèques logicielles

Chouki TIBERMACHINE

Chouki.Tibermachine@umontpellier.fr



Bibliothèques logicielles

Qu'est-ce que c'est ?

- Ensemble de routines, qui peuvent être déjà compilées et prêtes à être utilisées par des programmes ;
- Enregistrées dans des fichiers semblables aux fichiers de programmes, ou sous la forme d'un ensemble de fichiers de code objet rassemblés ;
- Apparues dans les années 1950 et devenues un outil incontournable dans les langages de programmation ;
- Opérations fréquentes en prog. : manipulation des interfaces utilisateur, des bases de données ou calculs maths ;
- Manipulées par l'éditeur de lien (si compilées) et le système d'exploitation (qu'elles soient statiques ou partagées).

Problématique

Plusieurs questions

- Comment retrouver une routine dans une bibliothèque ?
- Comment écrire des bibliothèques ?

Les réponses dépendent

- Du système (Unix, Windows, ou Mac OS X) ;
- Si le langage utilisé est un langage compilé ou non ;
- Si la bibliothèque est statique ou partagée.

Un peu de vocabulaire

Bibliothèque statique

- destinée à être copiée dans les programmes qui l'utilisent

Bibliothèque partagée (dynamique)

- destinée à être associée aux programmes à l'exécution ;
- même copie de la bib. peut être utilisée par +ieurs programmes.

Éditeur de liens

- le programme qui combine différents fichiers de code objet (après compilation), parmi lesquels des bibliothèques, pour en faire un programme exécutable.

Un peu de vocabulaire

Table de symboles

- C'est une structure manipulée par les compilateurs et les éditeurs de liens, qui contient, entre autres :
 1. les noms d'éléments du programme (routines, variables, constantes),
 2. leur adresses, c'est-à-dire leur emplacement dans le programme.
- La table de symboles produite par le compilateur est utilisée ensuite par l'éditeur de liens pour déterminer l'emplacement des routines qui se trouvent dans les bibliothèques.

Résolution de noms

Ce que l'on veut

- Pour qu'un programme utilise une routine, il est nécessaire que la provenance de la routine soit connue ;
- La provenance est déterminée conjointement par le compilateur, l'éditeur de liens et le système d'exploitation, dans un processus de résolution qui recherche dans les bibliothèques ;
- Le processus diffère selon que la ou les bibliothèques utilisées sont statiques ou partagées.

Résolution de noms

Processus préalable

- Lors de la traduction d'un fichier de code source en code objet, le compilateur ajoute à la table de symboles les noms des routines utilisées dans ce fichier source ainsi que leurs adresses ;
- L'adresse est laissée vide si la routine n'a pas été trouvée dans le fichier de code source ;
- Puis l'éditeur de liens recherchera alors à quoi correspond chaque routine dont l'adresse est laissée vide par le compilateur.

Résolution de noms

Dans le cas d'une bibliothèque statique

- L'éditeur de liens copie l'intégralité de la bibliothèque dans le programme (qui utilise des routines de la bibliothèque);
- Le fichier de bibliothèque (qui a été copié) n'est alors plus nécessaire à l'exécution du programme;
- La bibliothèque est ainsi copiée dans chaque programme qui l'utilise.

Résolution de noms

Dans le cas d'une bibliothèque partagée

- Si la routine provient d'une bibliothèque partagée alors l'éditeur de liens ne la copie pas dans le programme ;
- Au lieu de cela le système d'exploitation placera la bibliothèque en mémoire en même temps que le programme avant son exécution ;
- Une seule copie de la bibliothèque est enregistrée, et tous les programmes utilisent la même copie de la bibliothèque, ce qui permet d'économiser de la place par rapport à une bibliothèque statique.

Résolution de noms

Difficultés avec les bibliothèques partagées

- Bibliothèque utilisée par un programme non trouvée au moment où le programme en a besoin, entraînant l'échec du programme ;
- Interface de programmation (API) de la bibliothèque trouvée ne correspondant pas à celle dont le programme a besoin, entraînant un crash de ce dernier ;
- Très sensible aux changements de versions (parfois l'API n'a pas changé mais l'implantation oui, et cela peut entraîner des crashes) ;
- Problèmes de concurrence si la bibliothèque est exécutée simultanément par plusieurs programmes (les variables globales peuvent potentiellement être modifiées simultanément).

Bibliothèques logicielles selon les systèmes

Unix

- Fichiers « .so » (partagées), « .a » (statiques);
- Répertoires /lib ou /usr/lib (à l'installation de l'OS) et /usr/local/lib ou /etc/lib (à l'installation d'un logiciel);
- Variable d'environnement LD_LIBRARY_PATH.

Windows

- Fichiers « .dll » (partagées), « .lib » (statiques);
- Autres extensions : « .ocx », « .drv » ou « .cpl » (dynamiques);
- Répertoires \Windows ou \Windows\System.

Bibliothèques logicielles selon les systèmes

Mac OS X

- Système de « frameworks » ;
- « Framework » : répertoire dans lequel se trouve la/les bibliothèque(s) ainsi que la documentation et les headers (description des routines dans un langage de programmation) ;
- Fichiers « .a » (statiques) « .dylib » (partagées) ;
- Répertoire /System/Library/Frameworks.

Bibliothèques : cas du langage C

Un exemple : un fichier de code C

```
// main.c
#include <stdio.h>
void helper () {
    puts("helper"); // Écrit le mot helper
}
int main () {
    helper();
}
```

Bibliothèques : cas du langage C

Compilation

```
$ gcc -c main.c
```

Produit un fichier main.o (qui contient le code objet)

Affichage de la table de symboles

```
$ nm main.o
                 ...
0000000000000000 T helper
0000000000000013 T main
                 U puts
```

U veut dire (symbole) Undefined (sera lié plus tard)

T veut dire (qu'il apparaît dans la section de) Text (code)

Bibliothèques : cas du langage C

Édition des liens

```
$ gcc -o main main.o
```

Produit un fichier main (le fichier exécutable)

Exécution du programme

```
$ ./main
```

Utilisation des bibliothèques partagées et statiques

Compilation avec bibliothèques partagées

- Dépendance de l'exécutable vis-à-vis de bibliothèques dynamiques;
- Bibliothèques partagées chargées une seule fois en mémoire;
- Exemple :

```
$ gcc -o main main.c (compilation + édition des liens)
$ ldd main (pour lister les dépendances dynamiques)
    linux-vdso.so.1 => (0x00007ffe9b949000)
    libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0
        x00007f8d526af000)
    /lib64/ld-linux-x86-64.so.2 (0x000055b86d571000)
```


Utilisation des bibliothèques partagées et statiques

Compilation avec bibliothèques statiques

- Exécutable « standalone » (aucune dépendance);
- Exécutable plus « lourd » (bibliothèques intégrées);
- Exemple :

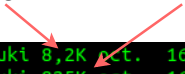
```
$ gcc -static -o main main.c
```

```
$ ldd main
```

n'est pas un exécutable dynamique

Exécutable avec
bibliothèque partagée

Exécutable avec
bibliothèque statique



```
-rwxr-xr-x 1 chouki chouki 8,2K oct. 16 09:34 maind  
-rwxr-xr-x 1 chouki chouki 825K oct. 16 09:34 mains
```

Création des bibliothèques partagées et statiques

Code de la bibliothèque (l'entête – *header* : « helper.h »)

```
// helper.h  
void helper();
```

Code de la bibliothèque (le programme : « helper.c »)

```
// helper.c  
#include <stdio.h>  
#include "helper.h"  
void helper () {  
    puts("helper");  
}
```

Création des bibliothèques partagées et statiques

Création de la bibliothèque partagée

- Tout se fait avec gcc :

```
$ gcc -c -fPIC helper.c -o helper.o  
$ gcc -shared -Wl,-soname,libhelper.so.1 \  
    -o libhelper.so.1.0.1 helper.o
```

Création de la bibliothèque statique

- Compilation normale, puis utilisation de l'« archiver » (« ar ») :

```
$ gcc -c helper.c -o helper.o  
$ ar rcs libhelper.a helper.o
```

Création des bibliothèques partagées et statiques

Programme utilisant la bibliothèque (« main.c »)

```
#include "helper.h"

int main () {
    helper();
}
```

Création des bibliothèques partagées et statiques

Édition de liens et exécution en partagé

```
$ gcc main.c -o dynamically_linked -L. -lhelper  
$ export LD_LIBRARY_PATH=.  
$ ./dynamically_linked
```

Édition de liens et exécution en statique

```
$ gcc -static main.c -L. -lhelper -o statically_linked  
$ ./statically_linked
```

Bibliothèques : cas de Python

Système de gestion de paquets (« pip »)

```
$ sudo apt-get install python-pip
$ sudo apt-get install python3-pip
$ pip search smbus2 (ou bien : python -m pip search smbus2)
$ sudo pip install smbus2
$ pip show smbus2
$ pip install smbus2 --upgrade
$ pip freeze
$ sudo pip uninstall smbus2
```

Publier ses propres bibliothèques (sur Pypi)

- Écrire sa bibliothèque (on va le voir) ;
- Utiliser « setuptools » (on ne fera pas) ;
- Mettre à jour le fichier « setup.py » (on ne fera pas).

Importer des fonctions d'autres fichiers

Fichier « func.py » (module)

```
#!/usr/bin/python2.7
#-*- coding: utf-8 -*-

def ajoute_un(v):
    return v + 1
```

- Bibliothèque = Module = fichier Python contenant des définitions et des instructions réutilisables
- Chaque module a sa propre table de symboles (définitions globales)

Importer des fonctions d'autres fichiers

Fichier « fiche.py »

```
#!/usr/bin/python2.7
#-*- coding: utf-8 -*-

import func

age = input("Quel est votre âge? : ")
print "Vous avez %d ans" % age
age_plus_un = func.ajoute_un(age)
print "Dans un an vous aurez %d ans" % age_plus_un
```

Le nom du module est inséré dans la table de symboles du code objet de fiche.py (fonctions du module accessibles avec `< nom_module > . < fonction >`)

Variante d'importation de fonctions

Fichier « fiche.py »

```
#!/usr/bin/python2.7
#-*- coding: utf-8 -*-

from func import *

age = input("Quel est votre âge? : ")
print "Vous avez %d ans" % age
age_plus_un = ajoute_un(age)
print "Dans un an vous aurez %d ans" % age_plus_un
```

Les noms des fonctions du module sont insérés dans la table de symboles du code objet de fiche.py

Créer des packages

Étapes

- Un package = ensemble de modules (et de packages)
- Créer un répertoire qui sera le nom du package ;
- Dans ce répertoire, on y met :
 - Un fichier « `__init__.py` » (qui peut être vide) ;
 - Des modules (fichiers Python classiques) et des packages.

Exemple

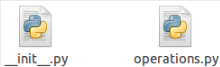
- On crée un package « `utils` » ;
- On ajoute un module « `operations.py` ».

Créer des packages

Dossier du projet



Dossier « utils »



Fichier « operations.py »

```
#!/usr/bin/python2.7
#-*- coding: utf-8 -*-

def ajoute_deux(v):
    return v + 2
```

Créer des packages

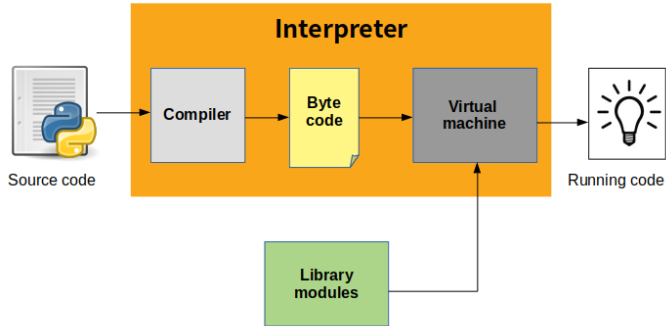
Fichier « fiche.py »

```
#!/usr/bin/python2.7
#-*- coding: utf-8 -*-

from func import *
from utils.operations import ajoute_deux

age = input("Quel est votre âge? : ")
print "Vous avez %d ans" % age
age_plus_un = ajoute_un(age)
print "Dans un an vous aurez %d ans" % age_plus_un
age_plus_deux = ajoute_deux(age)
print "Dans un an vous aurez %d ans" % age_plus_deux
```

Pré-compilation de scripts Python



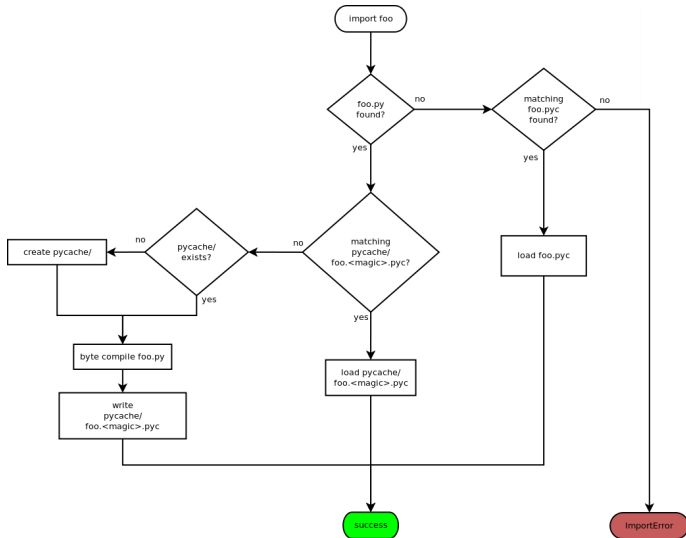
<https://indianpythonista.wordpress.com/2018/01/05/demystifying-pyc-files/>

Pré-compilation de scripts Python

Une vraie compilation ? Non !

- Fonctionnalité permettant de mettre en cache la phase de compilation d'un fichier source ;
- Création de fichiers avec l'extension « .pyc », placés au même niveau que les fichiers « .py » correspondants (dans rép. `__pycache__` / récemment) ;
- Compilation bytecode (interprète toujours nécessaire), qui permet d'améliorer la vitesse de chargement d'un module lors de sa prochaine invocation (mais non sa vitesse d'exécution) ;
- Effectué automatiquement pour les fichiers sources composant un module lors de son chargement (ne compile que si .py est plus récent que .pyc).

Pré-compilation de scripts Python



Compilation à la main de programmes Python

Utiliser le module *py_compile*

```
$ python  
>>> import py_compile  
>>> py_compile.compile('abc.py')
```

génère un fichier abc.pyc

Utiliser la commande *compileall*

```
python -m compileall .
```

compile tous les fichiers dans le répertoire courant (.)

Affichage du bytecode (code objet pour l'interprète Python)

```
>>> def hello():  
...     print "hello!"  
>>> import dis  
>>> dis.dis(hello)  
2           0 LOAD_CONST          1 ('hello!')  
           3 PRINT_ITEM  
           4 PRINT_NEWLINE  
           5 LOAD_CONST          0 (None)  
           8 RETURN_VALUE
```

Recherche de modules/packages Python

- Lorsqu'un module est importé dans un programme Python, il est recherché (dans l'ordre) dans :
 1. le répertoire courant
 2. la liste de répertoires indiquée dans la variable d'environnement *PYTHONPATH* (éq. à *PATH*)
 3. la liste de répertoires fixée à l'installation de Python (*/usr/lib/python2.7/, ...*)
- La variable *sys.path* contient la liste de répertoires où l'interprète recherche les modules/packages importés
- La variable (une liste) peut être modifiée (*sys.path.append(< un_chemin >)*)

Remerciements et références

Remerciements

David Delahaye, professeur à la FDS (mon prédécesseur)

Références bibliographiques

- Andrew Tanenbaum. Architecture de l'ordinateur, 5ème édition. Pearson Education, 2009
Andrew Tanenbaum. *Structured Computer Organization*, 6th edition. Pearson, 2012.
- Andrew Tanenbaum. Systèmes d'exploitation 3ème Ed. Pearson, 2008.
Andrew Tanenbaum & Herbert Bos. Modern Operating Systems. Pearson 2016.