Static and Dynamic Libraries

Linking Libraries

The act of linking libraries is a form of code dependency management. When any app is run, its executable code is loaded into memory. Additionally, any code libraries that it depends on are also loaded into memory. There are two type of linking: static, and dynamic. Both offer different benefits to the developer and should be used according to these benefits. This blog post will cover the benefits offered by each and then explain the basics of how to create and link your own libraries on OS X and iOS.

Dynamic Linking

Dynamic linking is most commonly used on OS X and iOS. When dynamic libraries are linked, none of the library's code is included directly into the linked target. Instead, the libraries are loaded into memory at runtime prior to having symbols getting resolved. Because the code isn't statically linked into the executable binary, there are some benefits from loading at runtime. Mainly, the libraries can be updated with new features or bug-fixes without having to recompile and relink executable. In addition, being loaded at runtime means that individual code libraries can have their own initializers and clean up after their own tasks before being unloaded from memory. For more information on overview and design, see Apple's Dynamic Library Programming Topics.

• Libraries

Dynamic libraries are a type of Mach-O binary[1] that is loaded at launch or runtime of an application. Since the executable code in a dynamic library isn't statically linked into target executable, this affords some benefits when needing to reuse the same code. For example, if you have an application and a daemon or extension that needs to make use of the same code, that code only has to exist in a single location -- the dynamic library, rather than in both the executable's binary and the daemon's binary. Since dynamic libraries are loaded at runtime, the library is responsible for telling the linker what additional code is needed. This removes the burden of managing what all of the code that you use needs to operate.

• Frameworks

Dynamic frameworks are similar to dynamic libraries. Both are dynamically linkable libraries, except a dynamic framework is a dynamic library embedded in a bundle. This allows for versioning of a dynamic library and sorting additional assets that are used by the library's code.

• Building

This is a walk-through of the steps required to build `libfoo_dynamic.dylib`

bar.h

```
#ifndef __foo__bar__
#define __foo__bar__

#include <stdio.h>

int fizz();

#endif /* defined(__foo__bar__) */
```

bar.c

```
#include "bar.h"
#include <CoreFoundation/CoreFoundation.h>

int fizz() {
    CFShow(CFSTR("buzz"));

    return 0;
}
```

Starting out with the files `bar.h` and `bar.c`. The header file defines the function `fizz()`, which returns an integer value. The implementation file imports the CoreFoundation framework and implements the function `fizz` to print the strings "buzz" before returning `0`.

Compiling:

```
$ clang -c bar.c -o bar.o
```

This creates the object file[2], Mach-O binary with type `MH_OBJECT`, named "bar". One of these will be generated for each of the files compiled in the library.

Creating Library:

```
$ libtool -dynamic bar.o -o libfoo_dynamic.dylib -framework CoreFoundation -lSystem
```

This creates the dylib (dynamic library) and links against `libSystem` and `CoreFoundation.framework`. The dylib is a Mach-O binary file with a type `MH_DYLIB`. This will be loaded dynamically at launch time by dyld as a dependency of another binary.

• Linking

main.c

```
#include "bar.h"

int main() {
    return fizz();
}
```

In this example, importing the "bar.h" header for the dynamic library, and calling `fizz()` directly.

Compiling:

```
$ clang -c main.c -o main.o
```

This will generate the object file for main.

Linking:

```
$ ld main.o -lSystem -L. -lfoo_dynamic -o test_dynamic
```

This will generate a binary executable from the main object file, also passing

- `-lSystem` for `dyld_stub_binder`
- `-lfoo_dynamic` for linking against `libfoo_dynamic.dylib`

and finally, outputting a binary named `test_dynamic`.

Running:

```
$ ./test_dynamic
buzz
```

Symbols:

```
$ nm test_dynamic
0000000000001000 A __mh_execute_header
                 U _fizz
0000000000001fa0 T _main
                 U dyld_stub_binder
```

This lists all of the symbols in the main binary. Both the symbol `main` and `fizz` are listed here. The symbol `fizz` does not have an address, because it does not exist inside of the main binary, it exists in the dynamic library that was created. This symbol will be resolved at launch time, after all the referenced dependencies are loaded into memory.

References:

```
$ otool -L test_dynamic
test_dynamic:
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1197.1.1)
    libfoo_dynamic.dylib (compatibility version 0.0.0, current version 0.0.0)
```

The resulting binary only links against libSystem and the dylib that was created. The library `foo_dynamic` is responsible for linking against any additional libraries it needs. This is resolved at launch time, dynamically. In this case, the search path for `libfoo_dynamic.dylib` is going to be the same as the search location as the main executable.

Dynamic libraries and frameworks are loaded at launch time by the dynamic linker. They have associated search paths to help the linker find where they are located on the file system and load them.

Static Linking

Unlike dynamic, linking static libraries includes the object file code from the library into the target's binary. This results in a larger size on disk and slower launch times. Because the library's code is added directly to the linked target's binary, it means that to update any code in the library, the linked target would also have to be rebuilt.

Up until iOS 8, statically linked libraries were the de-facto way to ship and include any third-party code in an application.

Note: this is not to be confused with statically linking binaries.

• Libraries

A static library is a container for a set of object files. Static libraries use the file extension ".a", which comes from the (ar)chive file[3] type. An archive file was designed to contain a collection of files. This is ideal for the transport and use of many object files that comprise a single code library. However the linker can only use object files of a single architecture, so there are two different container formats for static libraries based on if they support single or multiple architectures.

All object files of the same architecture are stored in a single archive file. This is the type of container file that the linker expects per architecture. The object files are packaged by the utility ar, which stores the contents of each object file. OS X uses an implementation of `ar` that is similar to the BSD variant; the task of organizing the symbol lookup and table creation to a tool called `ranlib`. On OS X, this is an alias for libtool. This utility is responsible for mapping the symbols stored in the object files and will warn if there are mismatching architectures used. This will generate an archive file that can be examined and operated on using the `ar` utility.

Since a single archive file can only support a single architecture, a separate file format is used to act as a single container for multiple libraries. The file format chosen for this was the fat Mach-O binary. Due to this change in file type, `ar` can no longer operate on the static library. A fat Mach-O binary is a very simple container format that can house multiple files of different architectures.

```
// This is at the very beginning of the file
struct fat_header {
    uint32_t magic;     // This indicates the endianness of the binary file
    uint32_t nfat_arch; // This indicates how many architecture headers are defined for the file
};

// This is the architecture header definition, these definitions immediately follow the 'fat_header'
struct fat_arch {
    cpu_type_t cputype;         // This defines the CPU family type: "Intel", "ARM", "PPC"
    cpu_subtype_t cpusubtype;   // This defines the CPU variant for the family type: "i386", "x86_64", "armv7", "armv7s", "ppc64"
    uint32_t offset;            // Offset in the file where the architecture specific data starts
    uint32_t size;              // Length of the architecture specific data in the file
    uint32_t align;             // Power of 2 alignment data for the architecture type
};
```

While this is a Mach-O binary file type, it strictly acts as a safe container for multiple architectures. This format is used to store a copy of the library for each desired architecture type. To modify a static library that uses the fat Mach-O binary file type, the command lipo must be used. This can also extract a copy of the static library based on a specific architecture.

• Frameworks

A static framework is a bundle containing a static library file. These frameworks are just a convenient way to publish a static library that uses external assets; such as images, fonts, or language files. In addition, static frameworks behave exactly like static libraries. They are statically linked into the executable binary, not loaded at runtime.

• Building

This is a walk-through of building `libfoo_static.a`. This uses the same files used in the dynamic library example.

bar.h

```
#ifndef __foo__bar__
#define __foo__bar__

#include <stdio.h>

int fizz();

#endif /* defined(__foo__bar__) */
```

bar.c

```
#include "bar.h"
#include <CoreFoundation/CoreFoundation.h>

int fizz() {
    CFShow(CFSTR("buzz"));

    return 0;
}
```

Compiling:

```
$ clang -c bar.c -o bar.o
```

This creates the object file named "bar". Again, one of these will be generated for each of the files compiled in the library.

Creating Library:

```
$ ar -rcs libfoo_static.a bar.o
or
$ libtool -static bar.o -o libfoo_static.a
```

Unlike the dynamic library, when creating the static library there are no other libraries that are linked against it. This is because the (ar)chive file is just a container for the object files that need to be built. By running either `ar` or `libtool` on the set of object files generated by the compiler, they will be packaged up into an archive that can contain multiple sets of architecture and symbol definitions.

Running `ar` directly will produce a single archive file with just the object files that were passed. It will then call `ranlib` on this archive file it creates to sort the object files and also resolve any duplicate symbol names contained in the archive. Using `libtool` instead results in the same behavior and output, the code path for it changes slightly to call against `libstuff` instead of the tool `ar`.

Due to the fact this is not an executable binary file, static libraries do not retain any linkage they might need. This pushes the burden of tracking which dependencies to use onto the linked target executable file rather than on the static library itself. Luckily, Apple has implemented a load command for handling this, `LC_LINKER_OPTION`. This appears in a target's build settings in Xcode under the name "Link Frameworks Automatically". Enabling this option will append new load commands to each object file that specify linker flags that should be used with each object file. These flags can be displayed by using the following command:

```
$ otool -l <static library> | grep LC_LINKER_OPTION -A 4
```

• Linking

main.c

```
#include "bar.h"

int main() {
    return fizz();
}
```

In this example, importing the "bar.h" header for the static library, and calling `fizz()` directly.

Compiling:

```
$ clang -c main.c -o main.o
```

This will generate the object file for main.

Linking:

```
$ ld main.o -framework CoreFoundation -lSystem -L. -lfoo_static -o test_static
```

This will generate a binary executable from the main object file, also passing

- `-lSystem` for `dyld_stub_binder`
- `-framework CoreFoundation` for linking against `CoreFoundation.framework`
- `-lfoo_static` for linking against `libfoo_static.a`

and finally, outputting a binary named `test_static`.

Running:

```
$ ./test_static
buzz
```

Symbols:

```
$ nm test_static
                 U _CFShow
                 U ___CFConstantStringClassReference
0000000000001000 A __mh_execute_header
0000000000001f90 T _fizz
0000000000001f70 T _main
                 U dyld_stub_binder
```

Here we see that the symbol `fizz`, which was part of the static library, has an address associated with it. This is because the executable code that was associated with calling the function `fizz()` is now stored inside the main binary executable. Additionally there are references to `CFShow` and `CFConstantStringClassReference`, which exist as part of the CoreFoundation framework.

References:

```
$ otool -L test_static
test_static:
    /System/Library/Frameworks/CoreFoundation.framework/Versions/A/CoreFoundation (compatibility version 150.0.0, current version 855.17.0)
    /usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 1197.1.1)
```

When checking the list of linked libraries via otool, the main binary only links against libsystem. This is because now that the symbols from `libfoo_static` have been added to the main binary file. Since the code from `libfoo_static` depended on being linked against CoreFoundation, there is a dependency reference to that in the main binary.

Further Reading

- Overview of Dynamic Libraries
- Dynamic Library Programming Topics
- Mach-O Programming Topics
- Mach-O File Format ABI
- Object File
- UNIX (ar)chive
- OS X ABI Dynamic Loader Reference
- cctools source code

---

1. See "Further Reading" for "Mach-O Programming Topics" ↩
2. See "Further Reading" for "Object File" ↩
3. See "Further Reading" for "UNIX (ar)chive" ↩