Dynamic Library Standard Setup for Apps

This thread has been locked by a moderator.

144

If you haven't already read Dynamic Library Identification, read that before reading this.

If you have questions or comments about this, start a new thread here on DevForums. Tag it with *Linker* so that I see it.

Share and Enjoy
—
Quinn "The Eskimo!" @ Developer Technical Support @ Apple
`let myEmail = "eskimo" + "1" + "@" + "apple.com"`

Dynamic Library Standard Setup for Apps

When building a dynamic library, you must choose an install name. I recommend that you use an rpath-relative path for this, because that allows for a clean separation of concerns:

- You, as the library developer, don't have to worry about where the library is going to be installed. Your library can use the same install name no matter where it ends up on disk.
- Someone using your library has the freedom to place it wherever they want, adding an `LC_RPATH` load command so that the dynamic linker can find the library at runtime.

Newly created Xcode projects use this technique. This post walks you through an example of how such a project is set up, and how it works at runtime.

App with Framework Example

Imagine you're building an iOS app with an embedded framework. In this case the app is called WaffleOMatic, and has the WaffleVarnish framework embedded within it. On creating this project with Xcode, using the standard iOS > App project template and the iOS > Framework target template, you'll see the following:

- The framework's Dynamic Library Install Name build setting is set to `$(DYLIB_INSTALL_NAME_BASE:standardizepath)/$(EXECUTABLE_PATH)`. This resolves to `@rpath/WaffleVarnish.framework/WaffleVarnish`.
- The framework's Runpath Search Paths build setting is set to `$(inherited) @executable_path/Frameworks @loader_path/Frameworks`. The `$(inherited)` value is empty, so this resolves to `@executable_path/Frameworks @loader_path/Frameworks`.
- The app's Runpath Search Paths build setting is set to `$(inherited) @executable_path/Frameworks`. The `$(inherited)` value is empty, so this resolves to `@executable_path/Frameworks`.

When you build this app these settings get reflected in its Mach-O images:

```
% otool -l WaffleOMatic.app/Frameworks/WaffleVarnish.framework/WaffleVarnish | grep -A 2 LC_ID_DYLIB
          cmd LC_ID_DYLIB
      cmdsize 72
         name @rpath/WaffleVarnish.framework/WaffleVarnish …
% otool -l WaffleOMatic.app/Frameworks/WaffleVarnish.framework/WaffleVarnish | grep -A 2 LC_RPATH
          cmd LC_RPATH
      cmdsize 40
         path @executable_path/Frameworks …
--
          cmd LC_RPATH
      cmdsize 40
         path @loader_path/Frameworks …
% otool -l WaffleOMatic.app/WaffleOMatic | grep -A 2 LC_RPATH
          cmd LC_RPATH
      cmdsize 40
         path @executable_path/Frameworks …
% otool -l WaffleOMatic.app/WaffleOMatic | grep -A 2 LC_LOAD_DYLIB
          cmd LC_LOAD_DYLIB
      cmdsize 72
         name @rpath/WaffleVarnish.framework/WaffleVarnish …
…
```

Now let's look at how the dynamic linker loads the app:

1. It starts by loading the app's main executable, `WaffleOMatic.app/WaffleOMatic`.
2. This has an `LC_RPATH` load command for `@executable_path/Frameworks`, so it adds that directory to the rpath list.
3. It then finds that the app has a `LC_LOAD_DYLIB` load command that imports `@rpath/WaffleVarnish.framework/WaffleVarnish`, so it attempts to find that library.
4. This is an rpath-relative install name, so it looks in each directory in the rpath list. The first and only entry is `@executable_path/Frameworks`, so it looks for `@executable_path/Frameworks/WaffleVarnish.framework/WaffleVarnish`. It finds a dynamic library there.
5. It looks at that library's `LC_ID_DYLIB` value. That matches the library it's looking for, and so it starts loading that library.

Neat-o!

Note The above bundle structure applies to iOS and its child platforms. macOS uses a different structure, and you have to adjust the paths accordingly. See Placing Content in a Bundle for details on the macOS layout.

Swift System Libraries

Modern systems have the Swift system libraries built-in. If your app supports older systems, Xcode embeds a copy of these libraries within your app. It uses rpath magic to ensure that your app uses the built-in system libraries if they're available, falling back to the embedded ones if they're not.

To demonstrates how this works, change the deployment target for the WaffleOMatic app and the WaffleVarnish framework to iOS 12. Also add some trivial Swift code to the framework. The app now includes a copy of the Swift system libraries:

```
% ls -l WaffleOMatic.app/Frameworks
total 79932
drwxr-xr-x  … WaffleVarnish.framework
-rwxr-xr-x  … libswiftCore.dylib
…
```

Each library has an rpath-relative install name:

```
% otool -l WaffleOMatic.app/Frameworks/libswiftCore.dylib | grep -A 2 LC_ID_DYLIB
          cmd LC_ID_DYLIB
      cmdsize 52
         name @rpath/libswiftCore.dylib …
```

The app also has a new `LC_RPATH` load command for `/usr/lib/swift`:

```
% otool -l WaffleOMatic.app/WaffleOMatic | grep -A 2 LC_RPATH
          cmd LC_RPATH
      cmdsize 32
         path /usr/lib/swift …
--
          cmd LC_RPATH
      cmdsize 40
         path @executable_path/Frameworks …
```

The placement of this load command is critical. By placing it first in the list, the dynamic linker will use the built-in Swift system libraries in preference to the embedded ones.

IMPORTANT Some developers encounter a crash whose backtrace like this:

```
Thread 0 name:   Dispatch queue: com.apple.main-thread
Thread 0 Crashed:
0   libsystem_kernel.dylib  … __pthread_kill …
1   libsystem_pthread.dylib … pthread_kill …
2   libsystem_c.dylib       … abort …
3   libswiftCore.dylib      … swift::fatalError…
4   libswiftCore.dylib      … checkVersion() …
5   dyld                    … invocation function for block in dyld4::Loader::findAndRunAllInitializers…
…
```

This suggests that your app has a problem with its `LC_RPATH` load commands. Frames 4 and 3 shows that the Swift runtime has noticed a version disparity and trapped. Check the Runpath Search Paths build setting for each of your targets, making sure it matches the default value you get when you create a new target from the appropriate built-in template.

Linker

Reply

Posted 1 month ago by eskimo

Add a Comment