Adrian

+

Netguru

Wrapping C libraries in Swift

Agenda

- Setting up the project
- Adding the dependency
- Decorating headers using apinotes
- Decorating headers using clang attributes
- Writing a custom wrapper

Setting up the project

Setting up the project

macOS, iOS, tvOS, watchOS

1. Create a new Xcode project
2. Add an Objective-C framework target with unit tests

Setting up the project

Linux

1. ¯_(ツ)_/¯

Adding the dependency

Adding the dependency

- Pre-built library (macOS, iOS, tvOS, watchOS)
- Swift Package Manager (macOS, Linux)
- Carthage (macOS, iOS, tvOS, watchOS)

Pre-built static library

- Usually comes with one `.a` archive and a bunch of `.h` headers
- Can only be linked statically

Pre-built static library

1. Download pre-built files
2. Link framework with pre-built library
3. Add headers to the framework target and make sure they’re public

Pre-built static library

Pros:

- No need to build C library from source

Cons:

- No version control

Swift Package Manager

- Requires C library to contain an umbrella header in `include` folder
- Produces static libraries

Swift Package Manager

1. Add the dependency to `Package.swift`
2. Run `swift build`

Swift Package Manager

Pros:

- Truly cross-platform
- Takes care of compiling and linking

Cons:

- No iOS, tvOS, watchOS support

Carthage

Contains `xcodeproj`:

- Takes care of everything and produces a pre-built framework

No `xcodeproj`:

- Just resolves version and fetches the source code

Carthage

1. Add the dependency to `Cartfile`
2. Run `carthage update`

If there is no `xcodeproj`:

3. Add sources from `Checkouts` to the target
4. Make sure header files are public

Carthage

Pros:

- It just works™
- Doesn't require special files

Cons:

- Requires to build the C library from source
- No Linux support

Adding a modulemap

- `module.modulemap` contains description of where a library’s headers are and what are its submodules

```
// When included in a custom library

module CLibFoo {
    umbrella header "LibFoo.h"
    export *
}

// When importing a system library

module CCommonCrypto [system] {
    header "/usr/include/CommonCrypto/CommonCrypto.h"
    export *
}
```

Decorating using apinotes

```yaml
Functions: # Symbol category
  - Name: XYZFooCreate # Name of C symbol
    SwiftName: Foo.init(bar:) # Name of Swift symbol

Globals:
  - Name: XYZBazQux
    SwiftName: Baz.qux
```

apinotes

- Just plain YAML files
- Contain mappings of symbols
- Included in the framework target
- Used extensively by Apple when “swiftifying” APIs for SDK frameworks

```
# CoreGraphics.apinotes

Functions:
  - Name: CGRectMake
    SwiftName: CGRect.init(x:y:width:height:)
  - Name: CGRectIsNull
    SwiftName: getter:CGRect.isNull(self:)
  - Name: CGContextFillRect
    SwiftName: CGContext.fill(self:_:)

Enumerators:
  - Name: kCGRenderingIntentDefault
    SwiftName: CGColorRenderingIntent.defaultIntent
```

apinotes

Pros

- No need to edit original header files

Cons

- Error-prone, easy to forget about symbols
- Hard to maintain
- Not documented

Decorating using clang attributes

```
// With Foundation.framework
NS_SWIFT_NAME("Foo.bar()")

// Without Foundation.framework
__attribute__(swift_name("Foo.bar()"))
```

clang attributes

- Decorate symbols in header files
- Widely used in SDK and 3rd-party frameworks

```
XYZFoo * XYZFooCreate(XYZBar *bar)
NS_SWIFT_NAME("XYZFoo.init(bar:)");

XYZBar * XYZFooGetBar(XYZFoo *foo)
NS_SWIFT_NAME("getter:XYZFoo.bar(self:)");

void XYZFooSetBar(XYZFoo *foo, XYZBar *bar)
NS_SWIFT_NAME("setter:XYZFoo.bar(self:newValue:)");

void XYZFooDoSomething(XYZFoo *foo)
NS_SWIFT_NAME("XYZFoo.doSomething(self:)");

XYZBar * XYZFooDefaultBar
NS_SWIFT_NAME("XYZFoo.defaultBar");
```

clang attributes

Pros

- Well maintainable
- Play well with nullability
- Documented and widely adopted

Cons

- Need to edit the original header files

Writing a custom wrapper

Writing a custom wrapper

Pros

- Reduces friction when interacting with C
- Can use all powerful features of Swift

Cons

- Harder to maintain when breaking changes occur

Writing a custom wrapper

- Architecture of a Swift wrapper depends on architecture of the wrapped C library
- “Swift and C Interop” by Chris Eidhof on Mobile Warsaw Edition #1
- “Swift API Design Guidelines” session on WWDC16

Questions?

@akashivskyy

Twitter, GitHub

@adrian

Mobile Warsaw Slack