目录

iOS逆向：Swift逆向

- 最新版本：`v0.4`
- 更新时间：`20240220`

简介

整理iOS逆向期间，关于Swift逆向相关的各种知识。包括通用逻辑和各种常见类型，尤其是各种类型的内存布局。

源码+浏览+下载

本书的各种源码、在线浏览地址、多种格式文件下载如下：

HonKit源码

- crifan/ios_re_swift_reverse: iOS逆向：Swift逆向

如何使用此HonKit源码去生成发布为电子书

详见：crifan/honkit_template: demo how to use crifan honkit template and demo

在线浏览

- iOS逆向：Swift逆向 book.crifan.org
- iOS逆向：Swift逆向 crifan.github.io

离线下载阅读

- iOS逆向：Swift逆向 PDF
- iOS逆向：Swift逆向 ePub
- iOS逆向：Swift逆向 Mobi

版权和用途说明

鸣谢

感谢我的老婆**陈雪**的包容理解和悉心照料，才使得我 `crifan` 有更多精力去专注技术专研和整理归纳出这些电子书和技术教程，特此鸣谢。

其他

作者的其他电子书

本人 `crifan` 还写了其他 `150+` 本电子书教程，感兴趣可移步至：

crifan/crifan_ebook_readme: Crifan的电子书的使用说明

关于作者

关于作者更多介绍，详见：

关于CrifanLi李茂 – 在路上

Swift逆向概览

iOS逆向期间，有些app二进制代码内部是：`ObjC`和Swift混合的。

所以涉及到：Swift逆向。

Swift基础知识

Swift逆向相关

Swift逆向之静态分析

导出Swift头文件

TODO：把导出ObjC和Swift混淆时会报错的问题，和解决办法（最新版的class-dump），整理过来

IDA分析Swift

TODO：把已加到IDA中的，各种类型的定义，整理过来

Swift逆向之动态调试

调试时，辅助用IDA打开Swift的各种库文件，比如：

- `libswiftCore.dylib`
- `libswiftFoundation.dylib`
- 等等

方便调试内容。

这样，通过动态调试和查看IDA中的分析，函数的伪代码实现等等，就可以有更深入的理解很多函数和变量了。

TODO：把下面帖子中有用的内容，整理过来：

- 【未解决】iOS逆向Swift：swift_getAssociatedTypeWitness
- 【未解决】iOS逆向Swift：Data.withUnsafeMutableBytes(_:)
- 【已解决】iOS逆向Swift：Data.withUnsafeBytes(_:)
- 【未解决】iOS逆向WhatsApp：swiftPodCopy_10039A1B4
- 【未解决】iOS逆向Swift：Set.init(_:)
-
- 【记录】iOS逆向Swift：IDA静态分析libswiftFoundation.dylib
  - 。。。
-
- 【已解决】iOS逆向Swift：库文件libswiftCore.dylib
  - 【已解决】iOS逆向WhatsApp：ArrayAdoptStorage_2ToW1_CB2C
    - static Swift.Array.*adoptStorage(*: __owned Swift._ContiguousArrayStorage<τ_0_0>, count: Swift.Int) -> (Swift.Array<τ_0_0>, Swift.UnsafeMutablePointer<τ_0_0>)
  - 【已解决】iOS逆向：dyld_stub_binder
  - 【已解决】iOS逆向Swift：String的WitnessTable详情
    - destroy value witness for Swift.String
  - 【已解决】iOS逆向Swift：Float对应的Builtin.Int32的VWT的具体值
    - type metadata for Swift.Float
  - 【未解决】iOS逆向Swift：_NativeSet._unsafeInsertNew
    - Swift.*NativeSet._unsafeInsertNew(*: __owned τ_0_0, at: Swift._HashTable.Bucket) -> ()
  - 【已解决】iOS逆向Swift：Optional的VWT=ValueWitnessTable
  - 【未解决】iOS逆向Swift：_SetStorage.allocate
    - static Swift._SetStorage.allocate(capacity: Swift.Int) -> Swift._SetStorage<τ_0_0>
  - 【已解决】iOS逆向Swift：Xcode中给Swift函数_NativeSet.init加断点
    - Swift._NativeSet.init(capacity: Swift.Int) -> Swift._NativeSet<τ_0_0>
  - 【未解决】iOS逆向Swift：NativeSet
  - Collection
    - 【未解决】iOS逆向WhatsApp：Collection.map(_:)(void)
    - 【未解决】iOS逆向Swift：swift_getAssociatedTypeWitness
      - Swift.Collection.isEmpty.getter : Swift.Bool
    - 【基本解决】iOS逆向Swift：protocol requirements base descriptor的含义
      - protocol requirements base descriptor for Swift.Collection
  - 【已解决】iOS逆向：Swift的pod_copy
    - pod_copy(swift::OpaqueValue*, swift::OpaqueValue*, swift::TargetMetadata<_inprocess class="copyright">const*)
  - 【已解决】iOS逆向：Swift._ContiguousArrayStorage
    - type metadata accessor for Swift._ContiguousArrayStorage
  - 【已解决】iOS逆向：swift_getTypeByMangledNameInContext

- 【未解决】iOS逆向：__swift_instantiateConcreteTypeFromMangledName
- 【已解决】iOS逆向WhatsApp：Swift函数String.append(_:)的字符串拼接的实现逻辑
  - Swift.String.append(Swift.String) -> ()

Swift通用逻辑

- 其他
  - ValueBuffer
  - Exitential Container?
    - Box?
  - Opaque ?

TypeMetadata

ValueMetadata

`VWT = ValueWitnessTable`

ClassMetadata

内存布局

图

文字

Swift源码

IDA定义

举例

ValueBuffer

常用类型

String字符串

空间占用=大小

- 空间占用：`0x10` 字节 = `16` 字节 = **2**个 `int64`

分类=类型

概述

- Swift中String
  - `Small String`
  - `Large String`
    - 三类
      - `Native`
        - 真正地址

```
realStrAddr = objectAddr + 0x20
            = objectAddr + 32
            = objectAddr + nativeBias
```

      - `Shared`
        - 真正地址

```
realStrAddr = objectAddr
```

        - 说明
          - 对于Bridge的字符串：objectAddr == NSString的地址
            - 内存布局保存的数据，是NSString中的数据
      - `Foreign`
        - 真正地址

```
realStrAddr = objectAddr
```

详解

- Swift中String
  - `Small String`
  - `Large String`
    - 分3类
      - native
        - Native strings have tail-allocated storage, which begins at an offset of `nativeBias` from the storage object's address. String literals, which reside in the constant section, are encoded as their start address minus `nativeBias`, unifying code paths for both literals ("immortal native") and native strings. Native Strings are always managed by the Swift runtime.
      - shared
        - Shared strings do not have tail-allocated storage, but can provide access upon query to contiguous UTF-8 code units. Lazily-bridged NSStrings capable of providing access to contiguous ASCII/UTF-8 set the ObjC bit. Accessing shared string's pointer should always be behind a resilience barrier, permitting future evolution.
      - foreign
        - Foreign strings cannot provide access to contiguous UTF-8. Currently, this only encompasses lazily-

bridged NSStrings that cannot be treated as "shared". Such strings may provide access to contiguous UTF-16, or may be discontiguous in storage. Accessing foreign strings should remain behind a resilience barrier for future evolution. Other foreign forms are reserved for the future.

- 其他说明
  - 对于Shared和foreign
    - always created and accessed behind a resilience barrier, providing flexibility for the future.

内存布局

图

文字

概述

- Swift中Large String
  - Native
    - 真正字符串的地址=objectAddr+0x20
  - Shared/Foreign
    - 真正字符串的地址=objectAddr

详解

- Swift的String(64位)
  - 通用（Small String和Large String都有）
    - discriminator
      - 关于名称和
        - 位置
          - == leading nibble == top nibble = 64bit的最顶部的4个bit：b63:b60 == [bit63 ~ bit60]
            - 注：nibble=半个字节 = 4bit
        - 名称
          - （此处有个专门的名字）discriminator = 辨别器：用于区分字符串的具体类型
            - 其实叫做：String type 更加容易理解
        - 所以代码和注释中下面的名称是一个意思：
          - b63:b60 = discriminator = leading nibble == top nibble
      - 具体字段和含义
        - b63 = isImmortal
          - Should the Swift runtime skip ARC
            - Small strings are just values, always immortal
            - Large strings can sometimes be immortal, e.g. literals
        - b62 = (large) isBridged / (small) isASCII
          - For large strings, this means lazily-bridged NSString: perform ObjC ARC
          - Small strings repurpose this as a dedicated bit to remember ASCII-ness
        - b61 = isSmall
          - Dedicated bit to denote small strings
        - b60 = isForeign = isSlow
          - cannot provide access to contiguous UTF-8
      - 完整的映射表 = 不同字段的组合，表示不同类型字符串

Form	b63	b62	b61	b60
Immortal, Small	1	ASCII	1	0
Immortal, Large	1	0	0	0
Native	0	0	0	0
Shared	x	0	0	0
Shared, Bridged	0	1	0	0
Foreign	x	0	0	1
Foreign, Bridged	0	1	0	1

- Small String
  - [+0x00-0x0E] = 8+7个字节 =15个字节 = 64+56位 = 120位 : realString
  - [+0xF] = 第16个字节=共8位: discriminator + count
    - byte15的[b4:b7] = discriminator
    - byte15的[b0:b3] = count
- Large String
  - [+0x00-0x07] = 8个字节 = 64位 : flag
    - b63 = ASCII = isASCII
    - b62 = NFC = isNFC
    - b61 = native = isNativelyStored
    - b60 = tail = isTailAllocated
    - b59 = UTF8 = isForeignUTF8
    - b58:48 = reserved
    - b47:0 = count
  - [+0x08-0x0F] = 8个字节 = 64位 : discriminator + objectAddr
    - b63:b60 = discriminator
      - 详见上述解释：b63 = isImmortal、b62 = (large) isBridged / (small) isASCII、b61 = isSmall、b60 = isForeign = isSlow
    - b60:b0 = objectAddr
      - realStrAddr = objectAddr + nativeBias = objectAddr + 32 = objectAddr + 0x20

Swift源码

StringObject.swift

- StringObject.swift
  - swift/stdlib/public/core/StringObject.swift at main · apple/swift (github.com)

```
extension _StringObject {
  @inlinable @inline(__always)
  internal init(_ small: _SmallString) {
    // Small strings are encoded as _StringObjects in reverse byte order
    // on big-endian platforms. This is to match the discriminator to the
    // spare bits (the most significant nibble) in a pointer.
    let word1 = small.rawBits.0.littleEndian
    let word2 = small.rawBits.1.littleEndian
#if _pointerBitWidth(_64)
    // On 64-bit, we copy the raw bits (to host byte order).
    self.init(rawValue: (word1, word2))
#elseif _pointerBitWidth(_32)
    // On 32-bit, we need to unpack the small string.
    let smallStringDiscriminatorAndCount: UInt64 = 0xFF00_0000_0000_0000


    let leadingFour = Int(truncatingIfNeeded: word1)
    let nextFour = UInt(truncatingIfNeeded: word1 &>> 32)
    let smallDiscriminatorAndCount = word2 & smallStringDiscriminatorAndCount
    let trailingTwo = UInt16(truncatingIfNeeded: word2)
    self.init(
      count: leadingFour,
      variant: .immortal(nextFour),
      discriminator: smallDiscriminatorAndCount,
      flags: trailingTwo)
#else
#error("Unknown platform")
#endif
    _internalInvariant(isSmall)
  }


  @inlinable
  internal static func getSmallCount(fromRaw x: UInt64) -> Int {
#if os(Android) && arch(arm64)
    return Int(truncatingIfNeeded: (x & 0x000F_0000_0000_0000) &>> 48)
#else
    return Int(truncatingIfNeeded: (x & 0x0F00_0000_0000_0000) &>> 56)
#endif
  }


  @inlinable @inline(__always)
  internal var smallCount: Int {
    _internalInvariant(isSmall)
    return _StringObject.getSmallCount(fromRaw: discriminatedObjectRawBits)
  }


  @inlinable
  internal static func getSmallIsASCII(fromRaw x: UInt64) -> Bool {
#if os(Android) && arch(arm64)
    return x & 0x0040_0000_0000_0000 != 0
#else
    return x & 0x4000_0000_0000_0000 != 0
#endif
  }
  @inlinable @inline(__always)
  internal var smallIsASCII: Bool {
    _internalInvariant(isSmall)
    return _StringObject.getSmallIsASCII(fromRaw: discriminatedObjectRawBits)
  }
```

```
  @inlinable @inline(__always)
  internal init(empty: ()) {
    // Canonical empty pattern: small zero-length string
#if _pointerBitWidth(_64)
    self._countAndFlagsBits = 0
    self._object = Builtin.valueToBridgeObject(Nibbles.emptyString._value)
#elseif _pointerBitWidth(_32)
    self.init(
      count: 0,
      variant: .immortal(0),
      discriminator: Nibbles.emptyString,
      flags: 0)
#else
#error("Unknown platform")
#endif
    _internalInvariant(self.smallCount == 0)
    _invariantCheck()
  }
}
```

StringBridge.swift

`swift/stdlib/public/core/StringBridge.swift`

```
extension String {
  @_spi(Foundation)
  public init(_cocoaString: AnyObject) {
    self._guts = _bridgeCocoaString(_cocoaString)
  }
}
```

->

```
@usableFromInline
@_effects(releasenone) // @opaque
internal func _bridgeCocoaString(_ cocoaString: _CocoaString) -> _StringGuts {
  switch _KnownCocoaString(cocoaString) {
  case .storage:
    return _unsafeUncheckedDowncast(
      cocoaString, to: __StringStorage.self).asString._guts
  case .shared:
    return _unsafeUncheckedDowncast(
      cocoaString, to: __SharedStringStorage.self).asString._guts
#if _pointerBitWidth(_64)
  case .tagged:
    // Foundation should be taking care of tagged pointer strings before they
    // reach here, so the only ones reaching this point should be back deployed,
    // which will never have tagged pointer strings that aren't small, hence
    // the force unwrap here.
    return _StringGuts(_SmallString(taggedCocoa: cocoaString)!)
#if arch(arm64)
  case .constantTagged:
    let taggedContents = getConstantTaggedCocoaContents(cocoaString)
    return _StringGuts(
      cocoa: taggedContents.untaggedCocoa,
      providesFastUTF8: false, //TODO: if contentsPtr is UTF8 compatible, use it
      isASCII: true,
      length: taggedContents.utf16Length
    )
#endif
#endif
  case .cocoa:
    // "Copy" it into a value to be sure nobody will modify behind
    // our backs. In practice, when value is already immutable, this
    // just does a retain.
    //
    // TODO: Only in certain circumstances should we emit this call:
    //   1) If it's immutable, just retain it.
    //   2) If it's mutable with no associated information, then a copy must
    //      happen; might as well eagerly bridge it in.
    //   3) If it's mutable with associated information, must make the call
```

```
    let immutableCopy
      = _stdlib_binary_CFStringCreateCopy(cocoaString)


#if _pointerBitWidth(_64)
    if _isObjCTaggedPointer(immutableCopy) {
      // Copying a tagged pointer can produce a tagged pointer, but only if it's
      // small enough to definitely fit in a _SmallString
      return _StringGuts(
        _SmallString(taggedCocoa: immutableCopy).unsafelyUnwrapped
      )
    }
#endif


    let (fastUTF8, isASCII): (Bool, Bool)
    switch _getCocoaStringPointer(immutableCopy) {
    case .ascii(_): (fastUTF8, isASCII) = (true, true)
    case .utf8(_): (fastUTF8, isASCII) = (true, false)
    default: (fastUTF8, isASCII) = (false, false)
    }
    let length = _stdlib_binary_CFStringGetLength(immutableCopy)


    return _StringGuts(
      cocoa: immutableCopy,
      providesFastUTF8: fastUTF8,
      isASCII: isASCII,
      length: length)
  }
}
```

IDA定义

加到IDA中的定义

举例

Small String

"control"

寄存器中：

```
(lldb) reg r x0 x1 sp
      x0 = 0x006c6f72746e6f63
      x1 = 0xe700000000000000
```

内存中：

```
(lldb) po 0x0000000281a4cea0
MainAppLibrary.OfflineAB.BucketInfo
(lldb) x 0xg 0x0000000281a4cea0
0x281a4cea0: 0x0000000010415b460 0x0000000000000003
0x281a4ceb0: 0x006c6f72746e6f63 0xe700000000000000
0x281a4cec0: 0x0000000000002710 0x0000000281a4c930
(lldb) p c 0x006c6f72746e6f63
(long) control\0
```

->字符串本身 = realString：

```
0xe700000000000000 0x006c6f72746e6f63
```

->

```
(lldb) p c 0x006c6f72746e6f63
(unsigned long) control\0
```

->字符串是："control"

第15个字节=8位，分2部分：

- 最高4位=discriminator
  - 0xe
    - = 0b 1110
      - b63 = isImmortal -> 1
        - isImmortal=True
      - b62 = (large) isBridged / (small) isASCII -> 1
        - Small String：ASCII=True
      - b61 = isSmall -> 1
        - 是Small string
      - b60 = isForeign = isSlow -> 0
        - isForeign=False
- 最低4位=count
  - 0x7
    - 字符串长度是：7
      - = 7个字符
        - 对应着：control字符串的长度=7

Large String

Large String - Native

"ios_prod_latam_tos_reg_universe"

```
(lldb) po 0x00000002800f5e00
MainAppLibrary.OfflineAB.UniverseInfo

(lldb) x 6gx 0x00000002800f5e00
0x2800f5e00: 0x0000000105d3c398 0x0000000200000003
0x2800f5e10: 0xd00000000000001f 0x8000000105110930
0x2800f5e20: 0x6469725f72657375 0xe800000000000000
```

根据定义：

- MainAppLibrary.OfflineAB.UniverseInfo
  - [+0x10~0x18] = name
    - 类型：Swift的Large String（真正字符串地址是：name值+0x20）

此处值：

- [+0x10~0x18] = name
  - 0x2800f5e10: 0xd00000000000001f 0x8000000105110930

-》真正字符串地址是：

- 0x8000000105110930 + 0x20 = 0x8000000105110950

```
(lldb) x s 0x8000000105110950
0x105110950: "ios_prod_latam_tos_reg_universe"
```

额外说明

（1）如果不加0x20，则看到的字符串是别的（错位后的）值：

```
(lldb) x s 0x8000000105110930
0x105110930: "os_reg_experiment"
```

（2）去掉最开始的0x8，也是可以用x/s看到字符串的:

```
(lldb) x s 0x0000000105110930
0x105110930: "os_reg_experiment"
(lldb) x s 0x0000000105110950
0x105110950: "ios_prod_latam_tos_reg_universe"
```

类似的：去掉0x8前缀，可以用po查看出字符串的值：

```
(lldb) po (char *)0x0000000105110950
"ios_prod_latam_tos_reg_universe"
```

（3）不论是否加0x20，直接po查看，则都是（异常的，不是我们要的）时间类型的值：

```
(lldb) po 0x8000000105110930
2001-01-01 00:00:00 +0000
(lldb) po 0x8000000105110950
2001-01-01 00:00:00 +0000

(lldb) po (char *)0x8000000105110950
2001-01-01 00:00:00 +0000
```

后记：

另外某次去po，却连异常的时间类型的值都看不到，而是：就是数字：

```
(lldb) reg r x0 x1
      x0 = 0x000000000000007c
      x1 = 0xe100000000000000
(lldb) reg r x20 sp
     x20 = 0x000000016d658c20
      sp = 0x000000016d658c20
(lldb) x 8gx 0x000000016d658c20
0x16d658c20: 0xd000000000000021 0x80000001051946f0
0x16d658c30: 0x0000000000000002 0x00000002833c1140
0x16d658c40: 0x0000000105da1628 0x0000000105dbbcd0
0x16d658c50: 0x00000002833c1140 0x00000002828d0780
(lldb) po 0x80000001051946f0
9223372041235285744
```

而换x/s字符串查看，是可以看出：

（虽然是错误的，但是是）字符串的值的

```
(lldb) x s 0x80000001051946f0
0x1051946f0: "dummy_aa_offline_user_rid_ios"
```

当然，真正地址+0x20的字符串：

```
(lldb) p x 0x80000001051946f0 + 0x20
(unsigned long) 0x8000000105194710
```

->但是po也还是看不出：

```
(lldb) po 0x8000000105194710
9223372041235285776
```

只能用字符串查看：

```
(lldb) x s 0x8000000105194710
0x105194710: "dummy_aa_offline_rid_universe_ios"
```

Large String - Shared

"2.23.25.85"

此处：

```
0xc00000000000000a 0x400000028143c6c0
```

其中：

- Large String
  - 0xc00000000000000a
    - 0xC = 12 = 0b 1100
      - b63 = ASCII = isASCII
        - True
      - b62 = NFC = isNFC
        - True
      - b61 = native = isNativelyStored
        - False
      - b60 = tail = isTailAllocated
        - False
      - b59 = UTF8 = isForeignUTF8

            - False
        - 0xa = 10： 字符串长度是10
    - 0x4000000028143c6c0
        - discriminator = 0x4 = 0b 0100
            - b63 = isImmortal
                - False
            - b62 = (large) isBridged / (small) isASCII
                - True
            - b61 = isSmall
                - False
            - b60 = isForeign = isSlow
                - False
        - objectAddr = 0x28143c6c0

->

- Large String
    - length = 10
    - flag
        - isASCII = True
        - isNFC = True
        - isNativelyStored = False
        - isTailAllocated = False
        - isForeignUTF8 = False
    - discriminator
        - isImmortal = False
        - isBridged = True
        - isSmall = False
        - isForeign == isSlow = False
    - objectAddr = 0x28143c6c0

->

- Swift中的字符串
    - 是bridged桥接的
    - 字符串地址是：0x28143c6c0
    - 其他细节
        - 是ASCII的
        - 是NFC（Normal Form C）的
        - 不是尾部分配的TailAllocated

-> 此处对应着：从 `ObjC` 的 `NSString`： `"2.23.25.85"`

```
(lldb) reg r x0
      x0 = 0x4000000028143c6c0
(lldb) po $x0
2.23.25.85
(lldb) po [$x0 class]
__NSCFString
```

调用

- `libswiftFoundation.dylib`
    - `static Swift.String._unconditionallyBridgeFromObjectiveC(Swift.Optional<__C.NSString>) -> Swift.String`

而Bridge桥接过来的

->

此处：想要查看出字符串的值：

- （方式1）加上NSString强制类型转换去打印字符串

```
(lldb) po (NSString )0x000000028143c6c0
2.23.25.85
```

- （方式2）自己查看ObjC的NSString的内存值，手动打印出字符串

此处（NSString类型的字符串的）内存值是：

```
(lldb) x 8xg 0x00000028143c6c0
0x28143c6c0: 0x000021a1efdfa941 0x000000030000078c
0x28143c6d0: 0x35322e33322e320a 0x000000000035382e
0x28143c6e0: 0x00004fcde1ddc6e0 0x0000000000000041
0x28143c6f0: 0x0000000000000000 0x0000000000000000
```

其中核心数据是：

- `0x28143c6d0: 0x35322e33322e320a 0x000000000035382e`

->可以调试查看到具体字符串的值和长度：

```
(lldb) p c 0x35322e33322e320a
(long) \n2.23.25
(lldb) p c 0x000000000035382e
(int) .85\0

(lldb) x 8xb 0x28143c6d0
0x28143c6d0: 0x0a 0x32 0x2e 0x32 0x33 0x2e 0x32 0x35
(lldb) x 8xb 0x28143c6d8
0x28143c6d8: 0x2e 0x38 0x35 0x00 0x00 0x00 0x00 0x00
(lldb) x s 0x28143c6d0
0x28143c6d0: "\n2.23.25.85"
(lldb) x s 0x28143c6d8
0x28143c6d8: ".85"

(lldb) p d 0xa
(int) 10
(lldb) x s 0x28143c6d1
0x28143c6d1: "2.23.25.85"
```

即：

- 字符串：
  - 长度：`10`
  - 值：`2.23.25.85`

Array数组

内存布局

图

文字

概述

详解

Swift源码

IDA定义

举例

Set集合

内存布局

图

文字

概述

详解

Swift源码

IDA定义

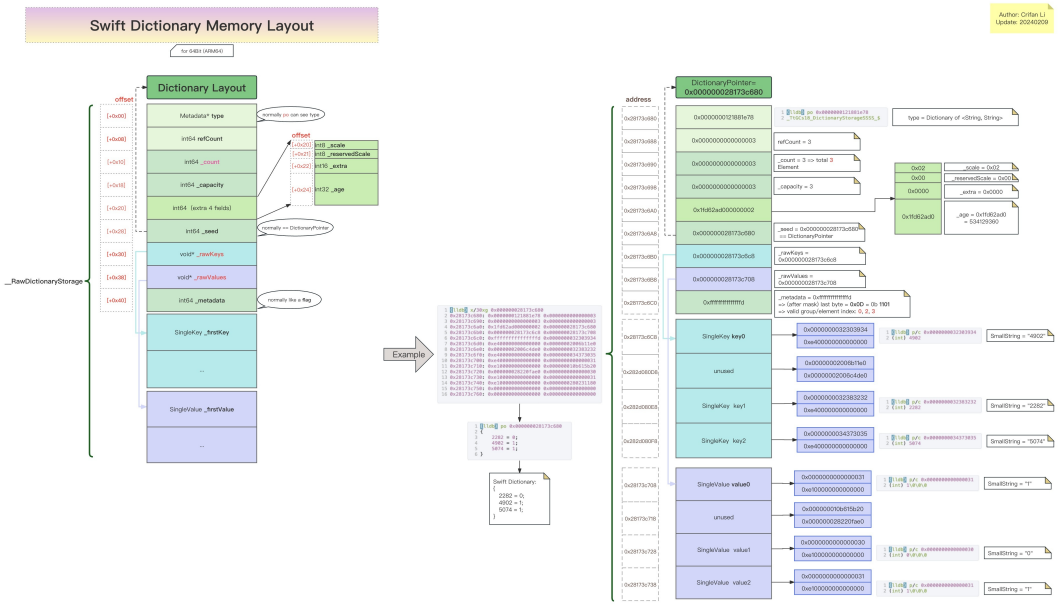
举例

Dictionary字典

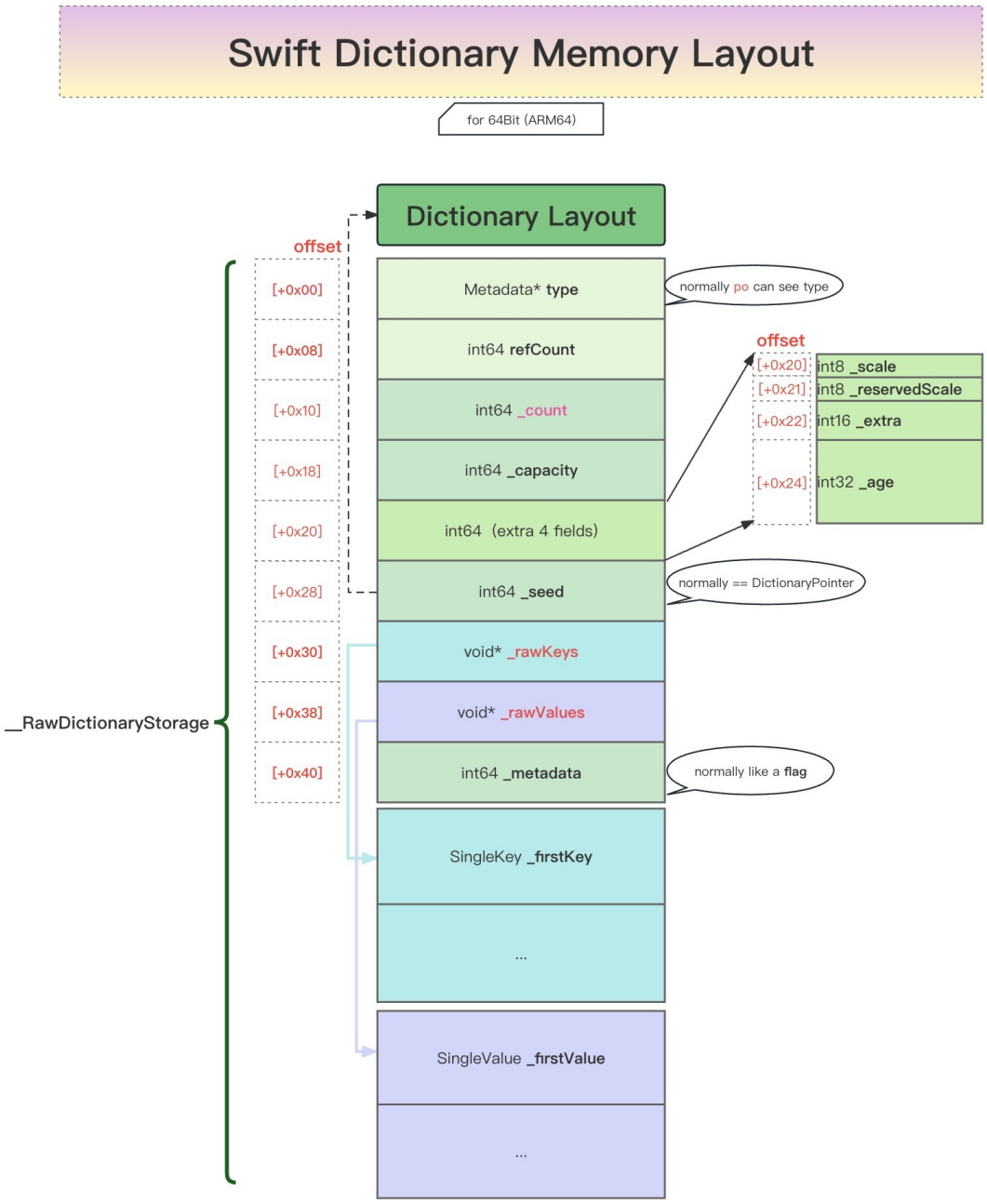
Swift的Dictionary内存布局

Swift的Dictionary内存布局结构图

- Swift的Dictionary内存布局结构图
  - 在线浏览
    - Swift的Dictionary内存布局结构图| ProcessOn免费在线作图,在线流程图,在线思维导图
  - 本地查看
    - 完整图



    - 核心内容



Swift的Dictionary内存布局 文字

- Swift的Dictionary内存布局 = 字段 = 属性
  - `[+0x00]` = `Metadata* type`
  - `[+0x08]` = `Int refCount`
  - `__RawDictionaryStorage`
    - `[+0x10]` = `var _count: Int`
    - `[+0x18]` = `var _capacity: Int`
    - `[+0x20]` = `Int`
      - `[+0x20~0x20]` = `var _scale: Int8`
      - `[+0x21~0x21]` = `var _reservedScale: Int8`
      - `[+0x22~0x23]` = `var _extra: Int16`
      - `[+0x24~0x27]` = `var _age: Int32`
    - `[+0x28]` = `var _seed: Int`
    - `[+0x30]` = `var _rawKeys: UnsafeMutableRawPointer`
      - 指向 `Key` = `键` 的数组列表
    - `[+0x38]` = `var _rawValues: UnsafeMutableRawPointer`
      - 指向 `Value` = `值` 的数组列表
    - `[+0x38]` = `int _metadata`
      - 指定了**哪几个**index元素是**有效数据**

Swift源码

IDA定义

举例

举例1：简单dict

Swift的Dict值：

```
{
    2282 = 0;
    4902 = 1;
    5074 = 1;
}
```

对应的例子，已经放到了内存布局图中了，详见：

Dictionary字典的内存布局图

此处只额外贴出：

相关调试内容

```
(lldb) po 0x000000028173c680
{
    2282 = 0;
    4902 = 1;
    5074 = 1;
}
```

->

```
(lldb) x 8xg 0x000000028173c680
0x28173c680  0x0000000121881e78 0x0000000000000003
0x28173c690  0x0000000000000003 0x0000000000000003
0x28173c6a0  0x1fd62ad000000002 0x000000028173c680
0x28173c6b0  0x000000028173c6c8 0x000000028173c708
(lldb) x 20xg 0x000000028173c680
0x28173c680  0x0000000121881e78 0x0000000000000003
0x28173c690  0x0000000000000003 0x0000000000000003
0x28173c6a0  0x1fd62ad000000002 0x000000028173c680
0x28173c6b0  0x000000028173c6c8 0x000000028173c708
0x28173c6c0  0xfffffffffffffffd 0x0000000032303934
0x28173c6d0  0xe400000000000000 0x00000002006b11e0
0x28173c6e0  0x00000002006c4de0 0x0000000032383232
0x28173c6f0  0xe400000000000000 0x0000000034373035
0x28173c700  0xe400000000000000 0x0000000000000031
0x28173c710  0xe100000000000000 0x000000010b615b20
```

可以继续了：

- 0xd = 0b 1101
  - 有效index
    - 0
    - 2
    - 3

->

```
(lldb) x 40xg 0x000000028173c680
0x28173c680  0x0000000121881e78 0x0000000000000003
0x28173c690  0x0000000000000003 0x0000000000000003
0x28173c6a0  0x1fd62ad000000002 0x000000028173c680
0x28173c6b0  0x000000028173c6c8 0x000000028173c708
```

```
0x28173c6c0  0xfffffffffffffffd 0x0000000032303934
0x28173c6d0  0xe400000000000000 0x00000002006b11e0
0x28173c6e0  0x00000002006c4de0 0x0000000032383232
0x28173c6f0  0xe400000000000000 0x0000000034373035
0x28173c700  0xe400000000000000 0x0000000000000031
0x28173c710  0xe100000000000000 0x000000010b615b20
0x28173c720  0x000000028220fae0 0x0000000000000030
0x28173c730  0xe100000000000000 0x0000000000000031
0x28173c740  0xe100000000000000 0x0000000280231180
0x28173c750  0x0000000000000000 0x0000000000000000
0x28173c760  0x0000000000000000 0x0000000000000000
0x28173c770  0x0000000000000000 0x0000000000000000
0x28173c780  0x0000000000000000 0x0000000000000000
0x28173c790  0x0000000000000000 0x0000000000000000
0x28173c7a0  0x0000000000000000 0x0000000000000000
0x28173c7b0  0x0000000000000000 0x0000000000000000
(lldb)

(lldb) x 30xg 0x000000028173c680
0x28173c680  0x0000000121881e78 0x0000000000000003
0x28173c690  0x0000000000000003 0x0000000000000003
0x28173c6a0  0x1fd62ad000000002 0x000000028173c680
0x28173c6b0  0x000000028173c6c8 0x000000028173c708
0x28173c6c0  0xfffffffffffffffd 0x0000000032303934
0x28173c6d0  0xe400000000000000 0x00000002006b11e0
0x28173c6e0  0x00000002006c4de0 0x0000000032383232
0x28173c6f0  0xe400000000000000 0x0000000034373035
0x28173c700  0xe400000000000000 0x0000000000000031
0x28173c710  0xe100000000000000 0x000000010b615b20
0x28173c720  0x000000028220fae0 0x0000000000000030
0x28173c730  0xe100000000000000 0x0000000000000031
0x28173c740  0xe100000000000000 0x0000000280231180
0x28173c750  0x0000000000000000 0x0000000000000000
0x28173c760  0x0000000000000000 0x0000000000000000
```

->

```
(lldb) po 0x0000000121881e78
_TtGCs18_DictionaryStorageSSSS_$

(lldb) p c 0x0000000032303934
(int) 4902
(lldb) p c 0x0000000032383232
(int) 2282
(lldb) p c 0x0000000034373035
(int) 5074
(lldb) p c 0x0000000000000031
(int) 1\0\0\0
(lldb) p c 0x0000000000000030
(int) 0\0\0\0
(lldb) x 2xw 0x28173c6a0
0x28173c6a0  0x00000002 0x1fd62ad0
(lldb) x 4xh 0x28173c6a0
0x28173c6a0  0x0002 0x0000 0x2ad0 0x1fd6
(lldb) x 8xb 0x28173c6a0
0x28173c6a0  0x02 0x00 0x00 0x00 0xd0 0x2a 0xd6 0x1f
(lldb) p d 0x1fd62ad0
(int) 534129360
```

举例2：复杂dict - key是String，value是AnyPrimitive

调试到复杂dict：

```
(lldb) reg r x0 x1 x2 x8
      x0 = 0x0000000011100b200
      x1 = 0x00000001031145b0  WhatsApp`$s10LeafFilterVN
      x2 = 0x0000000103114658  WhatsApp`LeafFilter_related_1030B4658
      x8 = 0x00000001006409a8  WhatsApp`sub_1005E09A8
(lldb) po 0x0000000011100b200
4580225536
(lldb) x 6xg 0x0000000011100b200
0x11100b200  0x000000010fe175c8 0x0000000000000003
```

```
0x11100b210  0x0000000000000007 0x000000000000000c
0x11100b220  0xf3473a6200000004 0x000000011100b200
(lldb) po 0x000000010fe175c8
_TtGCs18_DictionaryStorageSSOVO14MainAppLibrary9OfflineAB8UserInfoP10$102e672c012AnyPrimitive_$
```

->

此处是个Dictionary，value元素是

- AnyPrimitive
  - MainAppLibrary.OfflineAB.UserInfo 的 AnyPrimitive

查看内存值：

```
(lldb) x 80xg 0x000000011100b200
0x11100b200  0x000000010fe175c8 0x0000000000000003
0x11100b210  0x0000000000000007 0x000000000000000c
0x11100b220  0xf3473a6200000004 0x000000011100b200
0x11100b230  0x000000011100b248 0x000000011100b348
0x11100b240  0xffffffffffff0299 0x6d726f6674616c70
0x11100b250  0xe800000000000000 0x00000001e84fab78
0x11100b260  0x00000001e84fab78 0x00000001e84fab78
0x11100b270  0x00000001e84fab78 0x695f656369766564
0x11100b280  0xe900000000000064 0x69737265765f736f
0x11100b290  0xea00000000006e6f 0x00000001e84fab78
0x11100b2a0  0x00000001e84fab78 0x00000001e84fab78
0x11100b2b0  0x00000001e84fab78 0x6c6975625f707061
0x11100b2c0  0xe900000000000064 0x00000001e84fab78
0x11100b2d0  0x00000001e84fab78 0x6e5f656369766564
0x11100b2e0  0xeb00000000656d61 0x00000001e84fab78
0x11100b2f0  0x00000001e84fab78 0x00000001e84fab78
0x11100b300  0x00000001e84fab78 0x5f6573616656c572
0x11100b310  0xef6c656e6e616863 0x00000001e84fab78
0x11100b320  0x00000001e84fab78 0x00000001e84fab78
0x11100b330  0x00000001e84fab78 0x737265765f707061
0x11100b340  0xeb000000006e6f69 0x0000656e6f687069
0x11100b350  0xe000000000000000 0x000000010fe17468
0x11100b360  0x000000010fe16950 0x000000010fe169f8
0x11100b370  0x00000001e84fab01 0x00000001e84fab78
0x11100b380  0x00000001e84fab78 0x00000001e84fab78
0x11100b390  0x00000001e84fab78 0x00000001e84fab78
0x11100b3a0  0x00000001e84fab78 0x00000001e84fab78
0x11100b3b0  0x00000001e84fab78 0x00000001e84fab78
0x11100b3c0  0x00000001e84fab78 0x00000001e84fab78
0x11100b3d0  0x00000001e84fab78 0xc000000000000024
0x11100b3e0  0x40000002820700c0 0x00000002820700c0
0x11100b3f0  0x000000010fe16950 0x000000010fe169f8
0x11100b400  0x00000001e84fab01 0x0000000014164cccd
0x11100b410  0xc000000000000a0a 0x40000002835413a0
0x11100b420  0x000000010fe17798 0x000000010fe17840
0x11100b430  0x00000001e84fab00 0x00000001e84fab78
0x11100b440  0x00000001e84fab78 0x00000001e84fab78
0x11100b450  0x00000001e84fab78 0x00000001e84fab78
0x11100b460  0x00000001e84fab78 0x00000001e84fab78
0x11100b470  0x00000001e84fab78 0x00000001e84fab78
```

->

```
(lldb) p c 0x6d726f6674616c70
(long) platform

(lldb) p c 0x695f656369766564
(long) device_i
(lldb) p c 0xe900000000000064
(unsigned long) d\0\0\0\0\0\0\xe9

(lldb) p c 0x69737265765f736f
(long) os_versi
(lldb) p c 0xea00000000006e6f
(unsigned long) on\0\0\0\0\0\xea

(lldb) p c 0x6c6975625f707061
(long) app_buil
```

```
(lldb) p c 0x64
(int) d\0\0\0

(lldb) p c 0x6e5f656369766564
(long) device_n
(lldb) p c 0x0000000065 6d61
(int) ame\0

(lldb) p c 0x5f657361656c6572
(long) release_
(lldb) p c 0xef6c656e6e616863
(unsigned long) channel\xef

(lldb) p c 0x737265765f707061
(long) app_vers
(lldb) p c 0xeb0000000069 6f69
(unsigned long) ion\0\0\0\0\xeb
```

keys:

- platform
- device_id
- os_version
- app_build
- device_name
- release_channel
- app_version
- iphone

和：

```
(lldb) p c 0x0000656e6f687069
(long) iphone\0\0

(lldb) x s "0x40000028207 00c0 + 0x20"
0x2820700e0: "5E8-8164-1AEA05E45CD9"
(lldb) po 0x00000028207 00c0
2DE85147 8D62 45E8 8164 1AEA05E45CD9
'''
```

values:

- iphone
- 2DE85147-8D62-45E8-8164-1AEA05E45CD9
  - 注：Swift的Shared=Bridged的String
- ...

Struct结构体

内存布局

图

文字

概述

详解

Swift源码

IDA定义

举例

Bool布尔

Enum枚举

Tuple元祖

Int整型

附录

下面列出相关参考资料。

参考资料

- 【已解决】iOS逆向Swift：Dictionary字典的内存布局结构
- 【已解决】iOS逆向Swift：找Swift的Dictionary字典的测试数据
- 【整理】iOS逆向：去画Swift的Dictionary字典的内存布局结构图
- 【未解决】iOS逆向WhatsApp：genWAABOfflineAssignSha256_10039E294
- 【已解决】iOS逆向：Swift中String字符串的内部结构和逻辑
- 【已解决】iOS逆向Swift：研究_unconditionallyBridgeFromObjectiveC后的SwiftString的值的逻辑和含义
- 【未解决】iOS逆向Swift：String._unconditionallyBridgeFromObjectiveC
-
- swift-evolution/proposals/0247-contiguous-strings.md at main · apple/swift-evolution (github.com)
-