

转id等等)

源代码下载

下载地址：苹果公开的源代码在这里可以下载，opensource.apple.com/tarballs/

例如，其中，有两个比较常见需要学习源码的下载地址：

- runtime的源代码在[opensource.apple.com/tarballs/ob...](https://opensource.apple.com/tarballs/runtime/)
- CoreFoudation(包括runloop)的源代码在[opensource.apple.com/tarballs/CF...](https://opensource.apple.com/tarballs/CoreFoundation/)

当然，如果你想在github上在线查看源代码，可以点这里：[runtime](#)，[CoreFoudation](#)

为什么需要了解 引用转换？

例如，在用到runtime的关联对象API的时候，可能见到过这种代码：

```
static NSString * const kCMkvoClassPrefix_for_Block =  
@"CMObserver_";  
  
NSMutableArray * observers = objc_getAssociatedObject(self,  
(__bridge void *)kCMkvoAssociateObserver_for_Block);
```

需要通过 `(__bridge void *)` 转换 id 和 void *。为什么转换？这是因为 `objc_getAssociatedObject` 的参数要求的。先看一下它的API：

```
objc_getAssociatedObject(id _Nonnull object, const void * _Nonnull  
key)
```

对比一下两个参数：

- `const void * _Nonnull key`
- `static NSString * const kCMkvoClassPrefix_for_Block =
@"CMObserver_";`

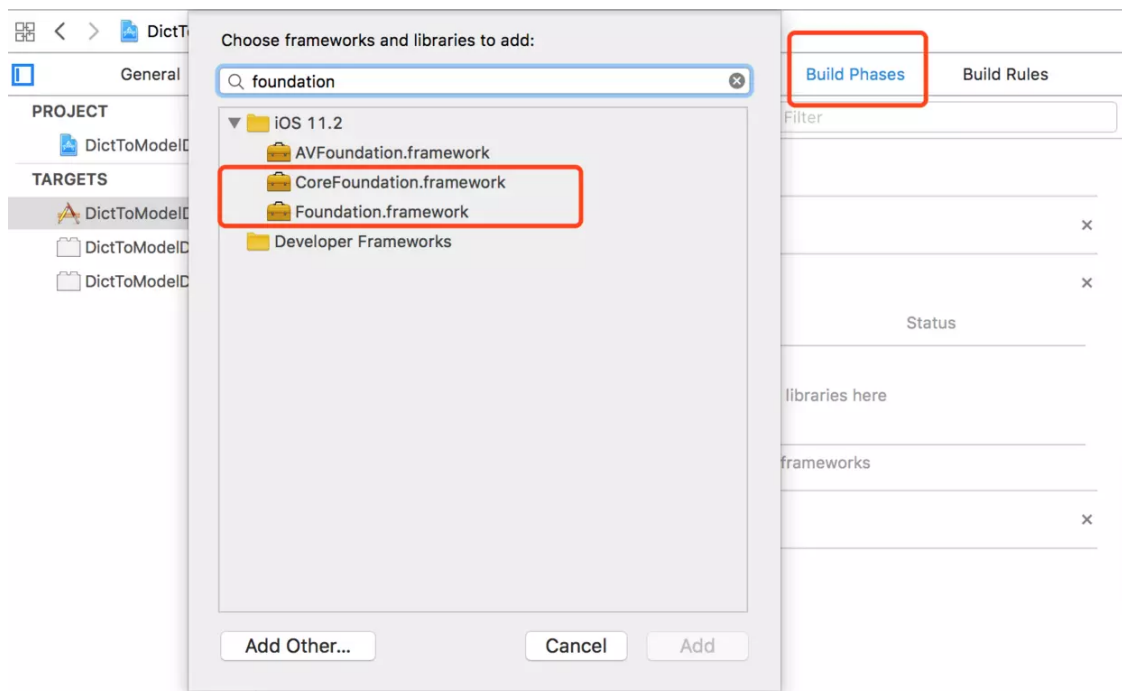
那么，想把 `NSString` 的字符串转成 `void *` 类型参数，必须进行引用转换。那么转换的是什么？OC中经常要对两个框架的对象进行转换：Foundation与Core Foundation对象。

至于上面的代码，完整的功能可查阅 [iOS开发·KVO用法](#)，[原理与底层实现: runtime 模拟实现KVO监听机制](#)

1. 两个框架的基本知识

1.1 Foundation

框架名是 `Foundation.framework`，在Xcode新建工程时可以选择导入（其实会默认自动依赖好）。Foundation框架允许使用一些基本对象,如数字和字符串,以及一些对象集合,如数组,字典和集合,其他功能包括处理日期和时间、内存管理、处理文件系统、存储(或归档)对象、处理几何数据结构(如点和长方形)。这个框架中的类都是一些最基础的类。来自于这个框架的类名以 `NS` 开头。



Foundation框架提供了非常多好用的类, 比如：

```
NSString : 字符串
NSArray  : 数组
NSDictionary : 字典
```

NSDate : 日期
NSData : 数据
NSNumber : 数字

1.2 Core Foundation

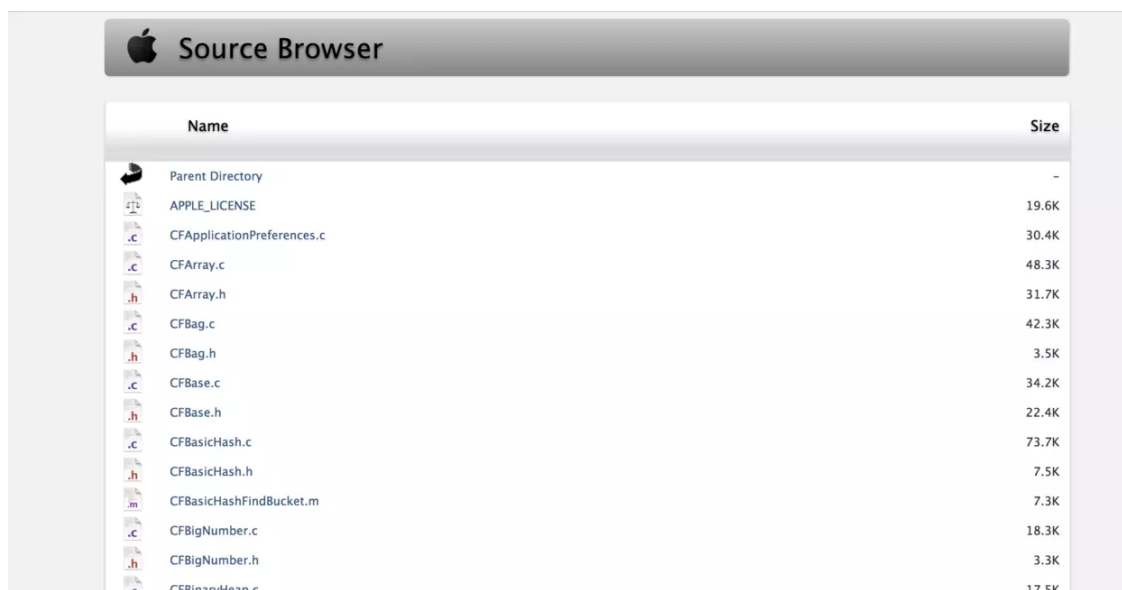
Core Foundation 对象主要使用在用C语言编写的Core Foundation 框架中,并使用引用计数的对象。在ARC无效时,Core Foundation 框架中的retain/release 分别是CFRetain /CFRelease。

框架 CoreFoundation.framework 是一组C语言接口, 它们为iOS应用程序提供基本数据管理和服务功能。下面列举该框架支持进行管理的数据以及可提供的服务。查阅Core Foundation的完整API [点这里](#)。

- CF的引用定义: CFStringRef 与 CFArrayRef 查阅CFArrayRef 的定义 [点这里](#) 查阅CFStringRef 的定义 [点这里](#)

```
typedef const struct __CFString * CFStringRef;  
typedef const struct __CFArray * CFArrayRef;
```

- CF的源代码: __CFString 与 __CFArray 查阅CF中结构体的源代码 [点这里](#)。



The screenshot shows the Apple Source Browser interface. At the top is a header bar with the Apple logo and the text "Source Browser". Below this is a table with two columns: "Name" and "Size". The table lists various files and directories in the Core Foundation framework, including "Parent Directory", "APPLE_LICENSE", "CFApplicationPreferences.c", "CFArray.c", "CFArray.h", "CFBag.c", "CFBag.h", "CFBase.c", "CFBase.h", "CFBasicHash.c", "CFBasicHash.h", "CFBasicHashFindBucket.m", "CFBigNumber.c", "CFBigNumber.h", and "CFBinaryHeap.c". Each row has a small icon to the left of the file name, representing its type (e.g., folder, source file, header file, module).

Name	Size
Parent Directory	-
APPLE_LICENSE	19.6K
CFApplicationPreferences.c	30.4K
CFArray.c	48.3K
CFArray.h	31.7K
CFBag.c	42.3K
CFBag.h	3.5K
CFBase.c	34.2K
CFBase.h	22.4K
CFBasicHash.c	73.7K
CFBasicHash.h	7.5K
CFBasicHashFindBucket.m	7.3K
CFBigNumber.c	18.3K
CFBigNumber.h	3.3K
CFBinaryHeap.c	17.5K

- 这些结构体的定义如下:

CFArray.c

```
struct __CFArray {
    CFRuntimeBase _base;
    CFIndex _count;      /* number of objects */
    CFIndex _mutations;
    int32_t _mutInProgress;
    __strong void *_store;      /* can be NULL when
MutableDeque */
};
```

CFString.c

```
struct __CFString {
    CFRuntimeBase base;
    union { // In many cases the allocated structs are smaller than
these
        struct __inline1 {
            CFIndex length;
        } inline1;      // Bytes
follow the length
        struct __notInlineImmutable1 {
            void *buffer;      // Note that
the buffer is in the same place for all non-inline variants of
CFString
            CFIndex length;
            CFAllocatorRef contentsDeallocator;    // Optional; just
the dealloc func is used
        } notInlineImmutable1;      // This is the
usual not-inline immutable CFString
        struct __notInlineImmutable2 {
            void *buffer;
            CFAllocatorRef contentsDeallocator;    // Optional; just
the dealloc func is used
        } notInlineImmutable2;      // This is the
not-inline immutable CFString when length is stored with the
contents (first byte)
        struct __notInlineMutable notInlineMutable;
    } variants;
};
```

1.3 两者关系

Core Foundation 框架和 Foundation 框架紧密相关，它们为相同功能提供接口，但 Foundation 框架提供 Objective-C 接口。Foundation 对象 和 Core Foundation 对象间的转换，俗称为桥接。如果您将 Foundation 对象和 Core Foundation 类型掺杂使用，则可利用两个框架之间的“[Toll Free Bridging](#)”。所谓的 Toll-free bridging 是说您可以在某个框架的方法或函数同时使用 Core Foundation 和 Foundation 框架中的某些类型。

很多数据类型支持这一特性，其中包括群体和字符串数据类型。每个框架的类和类型描述都会对某个对象是否为 Toll-free bridged，应和什么对象桥接进行说明。如需进一步信息，请阅读 [Core Foundation 框架参考](#)。

2. Objective-C 指针与 CoreFoundation 指针之间的转换

2.1 MRC 下的转换

- CF-->OC 强制转换符：(CFStringRef)
- OC-->CF 强制转换符：(NSString *)
- 例子

```
-(void)bridgeInMRC {
    // 将Foundation对象转换为Core Foundation对象，直接强制类型转换即可
    NSString *strOC1 = [NSString stringWithFormat:@"%xxxxxx"];
    CFStringRef strC1 = (CFStringRef)strOC1;
    NSLog(@"%@ %@", strOC1, strC1);
    [strOC1 release];
    CFRelease(strC1);

    // 将Core Foundation对象转换为Foundation对象，直接强制类型转换即可
    CFStringRef strC2 =
    CFStringCreateWithCString(CFAllocatorGetDefault(), "12345678",
    kCFStringEncodingASCII);
    NSString *strOC2 = (NSString *)strC2;
    NSLog(@"%@ %@", strOC2, strC2);
    [strOC2 release];
    CFRelease(strC2);
}
```

2.2 ARC下的转换

ARC仅管理Objective-C指针（retain、release、autorelease），不管理CoreFoundation指针。CF指针由人工管理，手动的CFRetain和CFRelease来管理。

在ARC中，CF和OC之间的转化桥梁是 __bridge，有3种方式：

- `__bridge` 只做类型转换，不改变对象所有权，是我们最常用的转换符。
- `__bridge_transfer`：ARC接管 管理内存
- `__bridge_retained`：ARC释放 内存管理

2.3 简单互相转换： `__bridge`

① 从OC转CF，ARC管理内存：

- `(__bridge CFStringRef)`
- 需要人工CFRetain，否则，Cocoa指针释放后，传出去的指针则无效。
- 例子

```
- (void)viewDidLoad
{
    [super viewDidLoad];

    NSString *aNSString = [[NSString alloc] initWithFormat:@"%test"];
    CFStringRef aCFString = (__bridge CFStringRef)aNSString;

    (void)aCFString;
}
```

上面只是单纯地执行了类型转换，没有进行所有权的转移，也就是说，当aNSString对象被ARC释放的时候，aCFString也不能被使用了。

② 从CF转OC，需要开发者手动释放，不归ARC管：

- `(__bridge NSString *)`
- 需要人工CFRelease，否则，OC对象的指针释放后，对象引用计数仍为1，不会被销毁。
- 例子

```

- (void)viewDidLoad
{
    [super viewDidLoad];

    CFStringRef aCFString = CFStringCreateWithCString(NULL, "test",
kCFStringEncodingASCII);
    NSString *aNSString = (__bridge NSString *)aCFString;

    (void)aNSString;

    CFRelease(aCFString);
}

```

3. ARC下内存管理发生改变的转换

3.1 CF-->OC: `__bridge_transfer`

- 例子

```

- (void)viewDidLoad {
    [super viewDidLoad];

    NSString *aNSString = [[NSString alloc] initWithFormat:@"test"];
    CFStringRef aCFString = (__bridge_retained CFStringRef)
aNSString;
    aNSString = (__bridge_transfer NSString *)aCFString;
}

```

3.2 OC-->CF: `__bridge_retained`

- 例子

```

- (void)viewDidLoad {
    [super viewDidLoad];

    NSString *aNSString = [[NSString alloc] initWithFormat:@"test"];
    CFStringRef aCFString = (__bridge_retained CFStringRef)
aNSString;
}

```

```
(void)aCFString;

// 这时候, 即使开启ARC, 也需要手动执行CFRelease
CFRelease(aCFString);
}
```

3.3 怎么区分记忆?

因为ARC无法管理CF对象的指针, 所以, 无论是CF转OC还是OC转CF, 我们只需关心CF对象的引用需要加1还是减1即可。

- CF转OC: CFRef必须减1

这样原来的CF对象就被释放, 所以, 以后也不用手动释放。

```
NSString *c = (__bridge_transfer NSString*)my_cfref; // -1 on the
CFRef
```

- OC转CF: CFRef 必须加1

这样新的CF对象就不会被释放, 所以, 以后用完必须手动释放。

```
CFStringRef d = (__bridge_retained CFStringRef)my_id; // returned
CFRef is +1
```

```
// 这时候, 即使开启ARC, CF对象用完后也需要手动执行CFRelease
CFRelease(aCFString);
```

3.4 转换相关的宏

- CFBridgingRetain

```
NS_INLINE CTypeRef CFBridgingRetain(id X) {
```



```
    return (__bridge_retain CTypeRef)X;
}
```

- CFBridgingRelease

```
NS_INLINE id CFBridgingRelease(CTypeRef X) {
    return (__bridge_transfer id)X;
}
```

例1

下面两个等效

```
CFStringRef cfStr = (__bridge_retained CFStringRef)ocStr;
```

```
CFStringRef cfStr = CFBridgingRetain(ocStr);
```

例2

下面两个等效

```
NSString *ocStr = (__bridge_transfer NSString*)cfStr;
```

```
NSString *ocStr = CFBridgingRelease(cfStr);
```

3.5 总结

- CF转化为OC时，并且对象的所有者发生改变，则使用 CFBridgingRelease() 或 __bridge_transfer 。
- OC转化为CF时，并且对象的所有者发生改变，则使

用 `CFBridgingRetain()` 或 `__bridge_retained`