第3章

核心数据类型你不知道的真相

本章将要介绍的内容是你在 iOS 开发中时时刻刻都在使用的几种基础的数据类型，但是对于其内部的本质，你却可能从来没有关注过。本章不介绍这些数据类型的用法，相信一个初级的 iOS 工程师都可以对这些数据类型的使用游刃有余，本章的核心是为你深入剖析这些看似平常的结构中更深层次的实现原理与设计思路，主要将涉及设计思路、内存布局、内部运行机制等。由于 Objective-C 语言与 Swift 语言在很多数据类型实现上都不相同，因此本章也会同时涉及 Objective-C 与 Swift 两种语言的相关知识。

在日常的产品需求开发中，我们可能并不需要过多地对数据类型内部的实现机制做了解。但是对这些内容的深入研究，可以帮助我们更加深刻地对语言的设计思想、程序内部的运行机制进行理解，也可以帮助我们解决疑难 BUG、优化程序性能。

在面试中，很多面试官会通过略微底层的问题考查应聘者的学习能力与钻研精神，掌握本章所介绍的内容，可以帮助你在面试时面对相对底层的问题时更加自信。

(1) 你了解 iOS 在运行时的内存分布情况吗?

(2) 堆和栈这两种数据结构分别在内存分布中担任着怎样的职责?

(3) NSString、NSNumber 等类簇相关的内容你之前是否关注过?

(4) 引用计数技术会管理所有的对象数据吗?

(5) 在 Swift 语言核心库中，String、Array 等数据类型的实现都是采用的结构体，结构体和类有何本质区别?

3.1 多变的 NSString 类

NSString 类是 Objective-C 语言 Foundation 库中核心的数据类型，在 iOS 应用开发中也是

使用最为广泛的一个类。NSString 用来创建字符串对象，NSMutableString 用来创建可变的字符串对象。实际上，在 Foundation 框架中，NSString 是采用类簇的模式实现的，即我们在学习设计模式时提到的抽象工厂设计模式，NSString 类与 NSMutableString 类是公共的抽象父类。本节我们就深入 NSString 的核心，对 Foundation 框架的巧妙设计思想进行学习与理解。

3.1.1 从 NSString 对象的引用计数说起

引用计数是 Objective-C 语言中进行内存管理的一种方式，当一个对象被强引用时，其引用计数会增加，当一个引用解除时，引用计数会减少。当对象的引用计数被减少到 0 时，此对象所使用的内存空间会被回收。随着 ARC（Automatic Reference Counting，自动引用计数）技术的应用，开发者很少再关心对象的内存管理问题，但是 ARC 技术不是万能的，关于对象的内存管理还有很多细节需要开发者了解，这些我们会专门在后面的章节进行介绍。本节我们以 NSString 对象的内存占用情况为引子，为大家展示 NSString 对象的特殊行为。

首先，使用 Xcode 开发工具创建一个新的 iOS 模板工程，在模板生成的 ViewController.m 文件中编写如下代码：

```
#import "ViewController.h"
@interface ViewController ()
@property (strong) NSString *string1;
@property (weak) NSString *string2;
@property (weak) NSString *string3;
@property (weak) NSString *string4;
@property (weak) NSString *string5;
@property (weak) NSString *string6;
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString *string1 = @"string1";
    NSString *string2 = @"string2";
    NSString *string3 = [[NSString alloc] initWithString:@"string3"];
    NSString *string4 = [[NSString alloc] initWithFormat:@"string4"];
    __weak NSString *string5 = [[NSString alloc]
initWithFormat:@"string5string5"];
    __weak NSString *string6 = [NSString
stringWithFormat:@"string6string6string6"];
    self.string1 = string1;
    self.string2 = string2;
    self.string3 = string3;
    self.string4 = string4;
    self.string5 = string5;
    self.string6 = string6;
```

```
    NSLog(@"%@, %@", string1, string1.class);
    NSLog(@"%@, %@", string2, string2.class);
    NSLog(@"%@, %@", string3, string3.class);
    NSLog(@"%@, %@", string4, string4.class);
    NSLog(@"%@, %@", string5, string5.class);
    NSLog(@"%@, %@", string6, string6.class);
}
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event
{
    NSLog(@"%@, %@", self.string1, self.string1.class);
    NSLog(@"%@, %@", self.string2, self.string2.class);
    NSLog(@"%@, %@", self.string3, self.string3.class);
    NSLog(@"%@, %@", self.string4, self.string4.class);
    NSLog(@"%@, %@", self.string5, self.string5.class);
    NSLog(@"%@, %@", self.string6, self.string6.class);
}
@end
```

运行代码，不做任何操作，控制台打印效果如下：

```
string1, __NSCFConstantString
string2, __NSCFConstantString
string3, __NSCFConstantString
string4, NSTaggedPointerString
(null), (null)
string6string6string6, __NSCFString
```

通过打印信息可以看出，使用不同方式创建的字符串其真实类型并不是 NSString，而是__NSCFConstantString、NSTaggedPointerString 和__NSCFString 类。在 Objective-C 语言中，字符串实际上分为 3 类：__NSCFConstantString 指静态字符串；NSTaggedPointerString 指标签地址字符串；__NSCFString 是对象字符串。关于这 3 种字符串的特点，后面我们会一一分析。点击运行起来后的应用页面，控制台会重新打印出一组信息，具体如下：

```
string1, __NSCFConstantString
string2, __NSCFConstantString
string3, __NSCFConstantString
string4, NSTaggedPointerString
(null), (null)
(null), (null)
```

通过与前面的打印信息进行比较可以发现一些有趣的事情，其中除了属性 string5 引用的对象被释放外，其他都没有被释放。

首先， string1 属性使用 strong 关键字进行修饰，对它进行赋值的对象会被强引用，根据引用计数的原理，其引用的对象不会被释放。

string2 属性虽然采用了弱引用的方式对对象进行引用，但是其赋值的字符串为__NSCFConstantString 常量类型，常量类型不会受引用计数的影响，因此其不会被释放。

string3 属性引用对象没有释放的原因与 string2 属性相同，使用 NSString 类的 initWithString 进行字符串对象的创建时，当传入的参数为字符串常量时，其构造出来的也是字符串常量。

string4 属性引用的字符串也没有被释放，那是因为其所构造出来的字符串为 NSTaggedPointerString 类型。这是一种特殊的指针类型，其指针本身就存储着数据信息，也不会受引用计数的影响。

string5 属性所引用的字符串为正常的字符串对象类型，根据引用计数的规则，弱引用不会增加引用计数，对象被释放。

在上面的打印信息中，string6 属性的行为最为奇怪，在 viewDidLoad 方法中其对象没有被释放，但是后续用户交互的方法中，这个属性引用的对象却被释放了，这是由于使用类方法创建的字符串对象会受自动释放池的影响。

相信现在你已经感受到了，NSString 类并没有我们想象的那么简单，后面我们会逐一解释其中缘由。

3.1.2 iOS 程序的内存分布

在 iOS 程序的运行中，内存根据作用会分为 5 个大的区域，分别为代码区、常量区、全局静态区、堆区和栈区。

（1）代码区用来存放程序的二进制代码，是只读属性，防止程序在运行时代码被修改。

（2）常量区用来存放常量数据，当整个应用程序结束后，其资源会由系统进行释放。在上一小节提到过，通过字面量创建的字符串就是常量，其会被存放到常量区中。例如：

```
NSLog(@"%p", @"Hello");
```

运行结果如下：

```
0x10524e080
```

可以看出，常量区在内存中的位置较低。

（3）全局静态区用来存放全局的静态数据，包括两个区域：存放未初始化的全局静态数据的区域被称为 BSS 区，存放已经初始化的全局静态数据的区域被称为数据区。例如：

```
static int data = 1;
static int bss;
static int data2 = 1;
static int bss2;
NSLog(@"data:%p, bss:%p", &data, &bss);
NSLog(@"data2:%p, bss2:%p", &data2, &bss2);
```

运行代码，打印效果如下：

```
data:0x10481adb8, bss:0x10481ae80
data2:0x10481adbc, bss2:0x10481ae84
```

可以看到，我们对变量的声明顺序是 data、bss、data2、bss2。从内存地址的分布可以看到存放 data 和 data2 的内存地址是连续的（int 类型在 64 位设备上大小为 4 个字节），bss 和 bss2 所在的内存地址是连续的，它们虽然都在全局静态区，但是对于已初始化和未初始化的数据是分开存放的。从打印的地址信息可以看出，全局静态区的地址要比常量区的地址高。

（4）堆区是日常开发中开发者需要最多关注的一个区域，其内存的使用需要开发者手动申请，内存的释放也需要开发者自己进行管理。堆区的地址并不连续，并且会向高地址进行扩展。在 Objective-C 语言中，引用计数管理的就是这部分的内存。

（5）栈区由系统进行分配，通常用来存放函数参数、局部变量等数据。在实际开发中，指针变量本身存放在栈中，指向的对象数据会存放在堆中，例如：

```
NSObject *object = [[NSObject alloc] init];
NSObject *object2 = [[NSObject alloc] init];
NSLog(@"%p, %p", object, &object);
NSLog(@"%p, %p", object2, &object2);
```

运行代码，打印效果如下：

```
0x600001c87670, 0x7ffee0b67908
0x600001c87660, 0x7ffee0b67900
```

理解了 iOS 程序运行中的内存分布情况和各个内存区域的功能，后面我们就能更加容易地理解内存管理的机制和数据的访问原理了。

3.1.3　NSString 类簇

本节我们就具体看一下 NSString 类簇中的 3 个类：__NSCFConstantString、NSTaggedPointerString 和__NSCFString。

前面有提到，__NSCFConstantString 实际上是字符串常量类型。所谓常量，即其不能被修改，一旦创建，在应用程序整个运行过程中就不会被回收。在 Objective-C 中，__NSCFConstantString 常量字符串有一个特点，即相同的字符串不会重复消耗内存，例如：

```
#import "ViewController.h"
@interface ViewController ()
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString *string1 = @"Hello";
    NSString *string2 = @"Hello";
    NSLog(@"%p, %p", string1, string2);
}
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event {
    NSLog(@"%p", @"Hello");
}
```

```
@end
```

运行代码，打印效果如下：

```
0x107eff068, 0x107eff068
0x107eff068
```

从打印的地址信息可以看出，所有相同的字面量字符串的内存地址都是一样的，因此无论程序中我们使用了多少个相同的字符串常量，都不会消耗额外的内存空间。

NSTaggedPointerString 是更加特殊的一种类型，表示标签指针。一般情况下，指针存储的内容是一个地址，需要根据地址指向的内存空间来获取真正的数据，但是在 64 位架构的系统上，这种数据存储方式有时会非常浪费空间。很多时候我们要存储的数据本身就很少，64 位的指针空间就可以容纳下来。在 Objective-C 中，NSNumber 和 NSString 这类对象就采用了标签指针这种优化方式，当我们使用 stringWithFormat 这类方法创建字符串对象时，如果字符串长度较短，系统就会默认将其创建为 NSTaggedPointerString 类型的字符串，这样在读取时通过指针本身就可以解读出内容，节省内存空间的同时也大大地加快了数据的访问速度，示例代码如下：

```
#import "ViewController.h"
@interface ViewController ()
{
    void *p;
}
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString * string3 = [NSString stringWithFormat:@"Hello"];
    p = (__bridge void *)(string3);
    NSLog(@"%@, %p", string3.class, string3);
}
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event {
    NSLog(@"%p, %@", p, p);
}
@end
```

运行代码，打印信息如下：

```
NSTaggedPointerString, 0xc8f25fbe582161b3
0xc8f25fbe582161b3, Hello
```

从打印信息可以看出，在 touchesBegan 方法中通过直接访问地址的方式依然可以获取到“Hello”字符串数据，这是 NSTaggedPointerString 非常有趣的地方，由于其内容本身就存放在指针中，因此只要指针数据是正确的，就可以访问到真实的数据。

__NSCFString 是传统意义上的字符串变量，当我们创建的字符串不是常量且长度较长时，其就会自动以__NSCFString 类型进行存储，此时字符串对象是受引用计数所影响的，当引用计数为 0 时，内存会被回收掉。示例代码如下：

```
#import "ViewController.h"
@interface ViewController ()
{
    void *p;
}
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];
    NSString * string5 = [NSString stringWithFormat:@"HelloHelloHello"];
    p = (__bridge void *)(string5);
    NSLog(@"%@, %p", string5.class, string5);
    NSLog(@"%p, %@", p, p);
}
-(void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event {
    NSLog(@"%p, %@", p, p);
}
@end
```

运行代码，你可能会发现一个神奇的现象，程序启动没有问题，无论是直接访问 string5 变量还是通过 p 指针访问字符串数据都可以正常访问，但是当点击屏幕时，系统有时会崩溃，有时又可以正常运行，这就是开发中经常遇到的野指针异常。当出了 viewDidLoad 方法后，局部变量 string5 对字符串对象的引用消失，对象所占内存被回收，此时指向这个地址的指针将变成野指针，访问野指针指向的内存是非常不安全的，此时此块空间可能已经存储了别的数据。

3.1.4　NSString 复制相关的方法

在 iOS 开发的相关职位面试中，经常会考查关于 copy 方法的问题，其中深复制与浅复制的区别常常会使应聘者困惑。对于 NSString 类型的数据，复制操作更加复杂。本节我们就来一探 NSString 复制操作的真相。

首先，深复制与浅复制的定义并不复杂。浅复制通常是指对指针进行赋值，即创建一个新的指针变量，让其指向被浅复制的对象。在 Objective-C 中，浅复制真正的意义就是增加了对象的引用计数，当真正的对象所在内存被回收时，所有浅复制出的指针都需要置为空。深复制则是指真正的数据内容复制，其会开辟新的内存空间，复制出的对象与原对象是完全不同的两个对象，不会影响原对象的引用计数。

在 Objective-C 中，与复制相关的方法有 copy 方法与 mutableCopy 方法，从字面意思上理解，copy 为普通复制方法，mutableCopy 为可变复制方法，但是其调用的对象不同，行为又会有一些差异。

对于 NSString 对象，无论创建出的对象真实的类是什么，使用 copy 方法都将进行浅复制操作，即不会开辟新的空间存储新的对象，而是将新的指针指向原对象。如果使用 mutableCopy 方法则会进行深复制，会创建出 NSMutableString（实际上是__NSCFString）对象，示例代码如下：

```
#import "ViewController.h"
@interface ViewController ()
@end
@implementation ViewController
- (void)viewDidLoad {
    [super viewDidLoad];

    NSString *string1 = @"Hello";
    NSString *string2 = [NSString stringWithFormat:@"Hello"];
    NSString *string3 = [NSString stringWithFormat:@"HelloWorld"];

    NSLog(@"%@, %@, %@", string1.class, string2.class, string3.class);
    NSLog(@"%p, %p, %p", string1, string2, string3);

    NSString *string11 = [string1 copy];
    NSString *string22 = [string2 copy];
    NSString *string33 = [string3 copy];

    NSLog(@"%@, %@, %@", string11.class, string22.class, string33.class);
    NSLog(@"%p, %p, %p", string11, string22, string33);

    NSMutableString *string111 = [string1 mutableCopy];
    NSMutableString *string222 = [string2 mutableCopy];
    NSMutableString *string333 = [string3 mutableCopy];

    NSLog(@"%@, %@, %@", string111.class, string222.class, string333.class);
    NSLog(@"%p, %p, %p", string111, string222, string333);

    [string111 appendString:@"Hi"];
    NSLog(@"%@", string111);
}
@end
```

运行上面的代码，打印效果如下：

```
__NSCFConstantString, NSTaggedPointerString, __NSCFString
0x10f416068, 0x80d0026d2df230be, 0x6000022b2200
__NSCFConstantString, NSTaggedPointerString, __NSCFString
```

```
0x10f416068, 0x80d0026d2df230be, 0x6000022b2200
__NSCFString, __NSCFString, __NSCFString
0x600002cad2f0, 0x600002cad650, 0x600002cad170
HelloHi
```

从打印信息可以看到，对 NSString 对象调用 copy 方法后，其类型和地址都没有任何变化，调用 mutableCopy 方法后，从对象地址可以看出全部创建了新的对象，新创建出的对象一律为 __NSCFString 类型。

对于 NSMutableString 类型的对象，无论调用 copy 方法还是 mutableCopy 方法，其都会进行深考虑，不同的是，调用 copy 方法会重新创建出一个不可变 NSString 对象，使用 mutableCopy 会重新创建出一个可变的 NSMutableString 对象，示例代码如下：

```
NSMutableString *mString1 = [NSMutableString stringWithString:@"HelloWorld"];
NSLog(@"%@, %@, %p", mString1, mString1.class, mString1);
NSString *mString11 = [mString1 copy];
NSLog(@"%@, %@, %p", mString11, mString11.class, mString11);
NSMutableString *mString111 = [mString1 mutableCopy];
NSLog(@"%@, %@, %p", mString111, mString111.class, mString111);
[mString111 appendString:@"Hi"];
NSLog(@"%@, %@", mString1, mString111);
```

控制台的打印信息如下：

```
HelloWorld, __NSCFString, 0x6000014bdbc0
HelloWorld, __NSCFString, 0x600001a84e40
HelloWorld, __NSCFString, 0x60000148cab0
HelloWorld, HelloWorldHi
```

下面我们来做一个简单的总结：

（1）对于 NSString 对象，copy 方法进行浅复制，mutableCopy 方法进行深复制并创建出 NSMutableString 对象。

（2）对于 NSMutableString 对象，copy 方法和 mutableCopy 方法都进行深复制，copy 方法创建出不可变的 NSString 对象，mutableCopy 方法创建出可变的 NSMutableString 对象。

在面试时，只要把握住上面两条原则，关于字符串的复制问题就不会难倒你。

3.2　深入理解 NSArray 类

通过上一小节的学习，我们知道 NSString 的实现采用了类簇的方式，其实 NSArray 也是这样，通过类簇的方式在不同的场景下生成不同类型的数组实例，这种实现极大地优化了内存的存储和使用效率。本节我们就一起来探索 NSArray 内部平时容易忽略的深入内容。