hoping_sir Lv1

runtime之ivar内存布局篇

随着`runtime`越来越常用，`iOSer`对`runtime`的理解要求也越来越高，大家都热衷于`runtime`源码理解，这篇我带领大家理解下关于`Ivar`的内容。

1.内存对齐

在分析`Ivar`之前，我们要了解下**内存对齐**的概念。 每个特定平台上的编译器都有自己的默认“对齐系数”，而64位中`iOS`里这个参数是8。我们测试一下：

```
@interface Dog : NSObject
{
    int age;          //4个字节
    BOOL sex;         //1个字节
    NSString* name;   //8个字节的指针地址
    short lifeTime;       //2个字节
    NSString* style;    //8个字节的指针地址
}
@end
```

这是我们新建的类`Dog`，里面有各种各样的成员变量，如果不存在内存对齐的话，会是一段连续的地址。我们打印下成员变量的地址偏移：

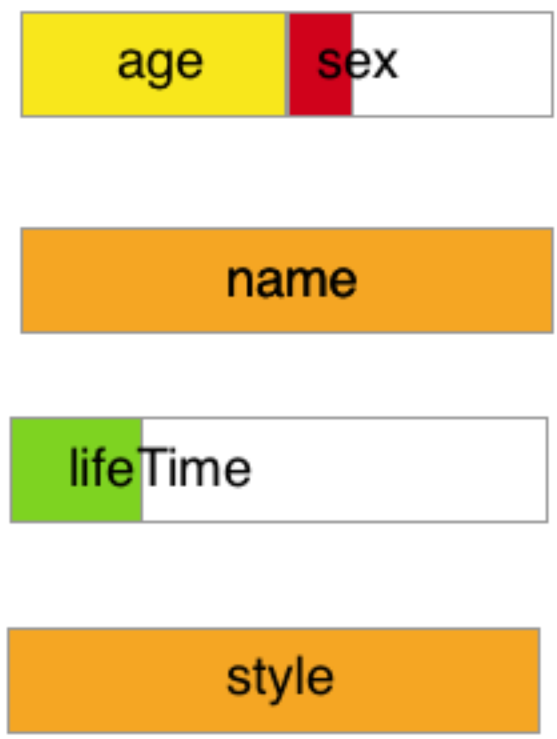
```
    Class class = objc_getClass("Dog");
    NSLog(@"内存地址：%p",class);
    unsigned int count;
    Ivar* ivars =class_copyIvarList(objc_getClass("Dog"), &count);
    for (unsigned int i = 0; i < count; i++) {
        Ivar ivar = ivars[i];
        ptrdiff_t offset = ivar_getOffset(ivar);
        NSLog(@"%s = %td",ivar_getName(ivar),offset);
```

```
    }
    free(ivars);
    NSLog(@"Dog总字节 = %lu",class_getInstanceSize(objc_getClass("Dog")));
```

运行结果：

```
2019-03-02 14:26:53.613593+0800 Runtime-Ivar[39894:1319445] 内存地址：0x1060d6f28
2019-03-02 14:26:53.613780+0800 Runtime-Ivar[39894:1319445] age = 8
2019-03-02 14:26:53.613867+0800 Runtime-Ivar[39894:1319445] sex = 12
2019-03-02 14:26:53.613954+0800 Runtime-Ivar[39894:1319445] name = 16
2019-03-02 14:26:53.614038+0800 Runtime-Ivar[39894:1319445] lifeTime = 24
2019-03-02 14:26:53.614123+0800 Runtime-Ivar[39894:1319445] style = 32
2019-03-02 14:26:53.614234+0800 Runtime-Ivar[39894:1319445] Dog总字节 = 40
```

根据打印结果，sex是bool类型，应该只占1个字节，但是却好像占了4个字节，其实这里并不是占了4个字节，而是因为内存对齐，其中3个字节是没用的。我们画下内存结构图：



我们可以看到内存并不是全部占满的，这是由于CPU并不是以字节为单位存取数据的，以单字节为单位会导致效率变差，开销变大，所以 CPU 一般会以 2/4/8/16/32 字节为单位来进行存取操作。而这里，会以8个字节为单位存取。

2.ivar的内存分布

这一部分我们从这4个方面去看 `ivar` 的分布情况。

- 属性与变量的分布

```
@interface Cat : NSObject
{
    NSString* c1;
    NSString* c4;
}

@property(nonatomic, copy)NSString* c2;
@property(nonatomic, copy)NSString* c3;
@end
```

```
unsigned int count;
    Ivar* ivars =class_copyIvarList(objc_getClass("Cat"), &count);
    for (unsigned int i = 0; i < count; i++) {
        Ivar ivar = ivars[i];
        ptrdiff_t offset = ivar_getOffset(ivar);
        NSLog(@"%s = %td",ivar_getName(ivar),offset);
    }
    free(ivars);
    NSLog(@"Cat总字节 = %lu",class_getInstanceSize(objc_getClass("Cat")));
```

运行结果:

```
2019-03-01 17:19:27.926009+0800 Runtime-Ivar[10017:6532014] c1 = 8
2019-03-01 17:19:27.926046+0800 Runtime-Ivar[10017:6532014] c4 = 16
2019-03-01 17:19:27.926056+0800 Runtime-Ivar[10017:6532014] _c2 = 24
2019-03-01 17:19:27.926065+0800 Runtime-Ivar[10017:6532014] _c3 = 32
2019-03-01 17:19:27.926097+0800 Runtime-Ivar[10017:6532014] Cat总字节 = 40
```

我们可以看到先是**成员变量**后是**属性**。

- 对象类型与基本类型的分布 先看下属性吧:

```
@interface Cat : NSObject
```

```
@property(nonatomic, copy)NSString* c1;
@property(nonatomic, assign)int c3;
@property(nonatomic, copy)NSString* c2;
@property(nonatomic, assign)int c4;
@end
```

打印ivar的方法和上面一致，运行后：

```
2019-03-01 17:11:58.167160+0800 Runtime-Ivar[9888:6528420] _c3 = 8
2019-03-01 17:11:58.167202+0800 Runtime-Ivar[9888:6528420] _c4 = 12
2019-03-01 17:11:58.167214+0800 Runtime-Ivar[9888:6528420] _c1 = 16
2019-03-01 17:11:58.167224+0800 Runtime-Ivar[9888:6528420] _c2 = 24
2019-03-01 17:11:58.167264+0800 Runtime-Ivar[9888:6528420] Cat总字节 = 32
```

我们可以看到属性的话先是**基本类型**后是**对象类型**。 再看下成员变量吧：

```
@interface Cat : NSObject
{
    NSString* c1;
    int c3;
    NSString* c2;
    int c4;
}
@end
```

运行结果：

```
2019-03-01 17:13:05.937474+0800 Runtime-Ivar[9909:6529050] c1 = 8
2019-03-01 17:13:05.937515+0800 Runtime-Ivar[9909:6529050] c3 = 16
2019-03-01 17:13:05.937526+0800 Runtime-Ivar[9909:6529050] c2 = 24
2019-03-01 17:13:05.937534+0800 Runtime-Ivar[9909:6529050] c4 = 32
2019-03-01 17:13:05.937567+0800 Runtime-Ivar[9909:6529050] Cat总字节 = 40
```

我们可以看到成员变量的话没有先后之分。

- `m` 文件与 `h` 文件的分布 先看属性吧：

```
Cat.h
```

```
@interface Cat : NSObject

@property(nonatomic, copy)NSString* c1;
@property(nonatomic, copy)NSString* c3;

@end
Cat.m
#import "Cat.h"
@interface Cat()
@property(nonatomic, copy)NSString* c2;
@property(nonatomic, copy)NSString* c4;
@end
@implementation Cat
@end
```

运行结果：

```
2019-03-01 17:16:16.989271+0800 Runtime-Ivar[9962:6530367] _c1 = 8
2019-03-01 17:16:16.989309+0800 Runtime-Ivar[9962:6530367] _c3 = 16
2019-03-01 17:16:16.989319+0800 Runtime-Ivar[9962:6530367] _c2 = 24
2019-03-01 17:16:16.989328+0800 Runtime-Ivar[9962:6530367] _c4 = 32
2019-03-01 17:16:16.989360+0800 Runtime-Ivar[9962:6530367] Cat总字节 = 40
```

我们可以看到先是**h文件**后是**m文件**。 再看看成员变量：

```
Cat.h
@interface Cat : NSObject
{
    NSString* c1;
    NSString* c3;
}
@end
Cat.m
#import "Cat.h"
@interface Cat()
{
    NSString* c2;
    NSString* c4;
}
@end
@implementation Cat
@end
```

运行结果：

```
2019-03-01 17:18:05.865890+0800 Runtime-Ivar[9992:6531268] c1 = 8
2019-03-01 17:18:05.865942+0800 Runtime-Ivar[9992:6531268] c3 = 16
2019-03-01 17:18:05.865952+0800 Runtime-Ivar[9992:6531268] c2 = 24
2019-03-01 17:18:05.865960+0800 Runtime-Ivar[9992:6531268] c4 = 32
2019-03-01 17:18:06.866000+0800 Runtime-Ivar[9992:6531268] Cat总字节 = 40
```

和上面一样显示 `h` 文件后是 `m` 文件。

那我们综合以上几种情况：

```
Cat.h
@interface Cat : Animal
{
    NSString* string_h_ivar;
     int int_h_ivar;

}

@property(nonatomic, copy)NSString* string_h_property;
@property(nonatomic, assign)int int_h_property;

@end
Cat.m
#import "Cat.h"
@interface Cat()
{
    NSString* string_m_ivar;
     int int_m_ivar;

}
@property(nonatomic, assign)int int_m_property;
@property(nonatomic, copy)NSString* string_m_property;
@end

@implementation Cat

@end
```

运行结果为：

```
2019-03-01 16:43:56.295851+0800 Runtime-Ivar[9412:6514675] string_h_ivar = 24
2019-03-01 16:43:56.295907+0800 Runtime-Ivar[9412:6514675] int_h_ivar = 32
2019-03-01 16:43:56.295917+0800 Runtime-Ivar[9412:6514675] string_m_ivar = 40
2019-03-01 16:43:56.295926+0800 Runtime-Ivar[9412:6514675] int_m_ivar = 48
2019-03-01 16:43:56.295934+0800 Runtime-Ivar[9412:6514675] _int_h_property = 52
2019-03-01 16:43:56.295942+0800 Runtime-Ivar[9412:6514675] _int_m_property = 56
2019-03-01 16:43:56.295950+0800 Runtime-Ivar[9412:6514675] _string_h_property = 64
2019-03-01 16:43:56.295960+0800 Runtime-Ivar[9412:6514675] _string_m_property = 72
2019-03-01 16:43:56.296001+0800 Runtime-Ivar[9412:6514675] Cat总字节 = 80
```

分析可得顺序为**h文件的ivar**->**m文件的ivar**->**h文件的property基本类型**->**m文件的property对象类型**

3.分析ivarlayout源码

在 `runtime.h` 里面关于 `IvarLayout` 的几个方法。

```
const uint8_t * _Nullable
class_getIvarLayout(Class _Nullable cls)
OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);

const uint8_t * _Nullable
class_getWeakIvarLayout(Class _Nullable cls)
OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);

void
class_setIvarLayout(Class _Nullable cls, const uint8_t * _Nullable layout)
OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);

void
class_setWeakIvarLayout(Class _Nullable cls, const uint8_t * _Nullable layout)
OBJC_AVAILABLE(10.5, 2.0, 9.0, 1.0, 2.0);
```

我们试用下，我们创建 `Person` 类：

```
@interface Person : NSObject
{
    int int1;
    bool bool1;
    __strong NSString* strong1;
```

```
    __weak NSString* weak1;
    char char1;
    __weak NSString* weak2;
     __strong NSString* strong2;
     __strong NSString* strong3;
    char char2;
    __weak NSString* weak3;
    char char3;
    int  int2;
    __weak NSString* weak4;
    __weak NSString* weak5;
}
```

然后我们使用下`class_getIvarLayout`和`class_getWeakIvarLayout`：

```
-(void)getIvarLayout {
    const uint8_t *strongLayout =   class_getIvarLayout(objc_getClass("Person"));
    if (!strongLayout) {
        return;
    }
    uint8_t byte;
    while ((byte = *strongLayout++)) {
        printf("strongLayout = #%02x\n",byte);
    }

    const uint8_t *weakLayout =   class_getWeakIvarLayout(objc_getClass("Person"));
    if (!weakLayout) {
        return;
    }
    while ((byte = *weakLayout++)) {
        printf("weakLayout = #%02x\n",byte);
    }
}
```

打印结果：

```
strongLayout = #11
strongLayout = #32
weakLayout = #21
weakLayout = #11
weakLayout = #31
weakLayout = #12
```

粗一看看不出什么，文档里面并没有详细说明 `layout` 的含义，我们要探究 `IvarLayout` 的话，还是要在源码找线索。

```
void fixupCopiedIvars(id newObject, id oldObject)
{
    for (Class cls = oldObject->ISA(); cls; cls = cls->superclass) {
        if (cls->hasAutomaticIvars()) {
            // Use alignedInstanceStart() because unaligned bytes at the start
            // of this class's ivars are not represented in the layout bitmap.
            size_t instanceStart = cls->alignedInstanceStart();

            const uint8_t *strongLayout = class_getIvarLayout(cls);
            if (strongLayout) {
                id *newPtr = (id *)((char*)newObject + instanceStart);
                unsigned char byte;
                while ((byte = *strongLayout++)) {
                    unsigned skips = (byte >> 4);
                    unsigned scans = (byte & 0x0F);
                    newPtr += skips;
                    while (scans--) {
                        // ensure strong references are properly retained.
                        id value = *newPtr++;
                        if (value) objc_retain(value);
                    }
                }
            }

            const uint8_t *weakLayout = class_getWeakIvarLayout(cls);
            // fix up weak references if any.
            if (weakLayout) {
                id *newPtr = (id *)((char*)newObject + instanceStart), *oldPtr = (id *)(
                unsigned char byte;
                while ((byte = *weakLayout++)) {
                    unsigned skips = (byte >> 4);
                    unsigned weaks = (byte & 0x0F);
                    newPtr += skips, oldPtr += skips;
                    while (weaks--) {
                        objc_copyWeak(newPtr, oldPtr);
                        ++newPtr, ++oldPtr;
                    }
                }
            }
        }
    }
}
```

这一段源码是 `runtime` 如何使用 `strongLayout` 和 `weakLayout` 。 下面，我们仔仔细细的分析这段源码，我们取出其中关键的一部分，先看关于 `strongLayout`：

```
//获得strongLayout的数组，数组元素类型为uint8_t，uint8_t为2位16进制数
const uint8_t *strongLayout = class_getIvarLayout(cls);
if (strongLayout) {
    //newPtr为ivar的初始地址
    id *newPtr = (id *)((char*)newObject + instanceStart);
    unsigned char byte;
    //遍历strongLayout，并且将内容赋值给byte
    while ((byte = *strongLayout++)) {
        //取出byte的左边一位
        unsigned skips = (byte >> 4);
        //取出byte的右边一位
        unsigned scans = (byte & 0x0F);
        //地址跳过skips位
        newPtr += skips;
        //循环scans次
        while (scans--) {
            // ensure strong references are properly retained.
            //取出地址里的内容，并且地址+1
            id value = *newPtr++;
            if (value) objc_retain(value);
        }
    }
}
```

从这一段源码，我们可以看到 `scans` 的地址值存放的是 `strong` 的成员变量，而 `skips` 是无效值，同样我们也可以分析 `weakLayout` 的那一段源码。为了能更加清晰的看到 `ivar` 的布局，我们通过 `ivar_getOffset` 方法获得ivar的内存布局。

```
-(void)getOffset {
    unsigned int count;
    Ivar* ivars =class_copyIvarList(objc_getClass("Person"), &count);
    for (unsigned int i = 0; i < count; i++) {
        Ivar ivar = ivars[i];
        ptrdiff_t offset = ivar_getOffset(ivar);
        NSLog(@"%s = %td",ivar_getName(ivar),offset);
    }
    free(ivars);
```

```
    NSLog(@"Person总字节 = %lu",class_getInstanceSize(objc_getClass("Person")));
}
```

运行结果:

```
2019-03-04 10:27:23.147813+0800 Runtime-Ivar[32952:841600] int1 = 8
2019-03-04 10:27:23.147842+0800 Runtime-Ivar[32952:841600] bool1 = 12
2019-03-04 10:27:23.147853+0800 Runtime-Ivar[32952:841600] strong1 = 16
2019-03-04 10:27:23.147863+0800 Runtime-Ivar[32952:841600] weak1 = 24
2019-03-04 10:27:23.147875+0800 Runtime-Ivar[32952:841600] char1 = 32
2019-03-04 10:27:23.147884+0800 Runtime-Ivar[32952:841600] weak2 = 40
2019-03-04 10:27:23.147894+0800 Runtime-Ivar[32952:841600] strong2 = 48
2019-03-04 10:27:23.147904+0800 Runtime-Ivar[32952:841600] strong3 = 56
2019-03-04 10:27:23.147913+0800 Runtime-Ivar[32952:841600] char2 = 64
2019-03-04 10:27:23.147922+0800 Runtime-Ivar[32952:841600] weak3 = 72
2019-03-04 10:27:23.147932+0800 Runtime-Ivar[32952:841600] char3 = 80
2019-03-04 10:27:23.147941+0800 Runtime-Ivar[32952:841600] int2 = 84
2019-03-04 10:27:23.147950+0800 Runtime-Ivar[32952:841600] weak4 = 88
2019-03-04 10:27:23.147959+0800 Runtime-Ivar[32952:841600] weak5 = 96
2019-03-04 10:27:23.147992+0800 Runtime-Ivar[32952:841600] Cat总字节 = 104
```

通过这个我们可以画出内存布局图:

这张图可以清晰的看到内存布局，然后我们再把 `strongLayout` 和 `weakLayout` 加到上面去：

因为无论是`strong`还是`weak`都是对象类型的变量，存的都是指针地址，所以占8位。所以源码

中的 `scan` 地址，其实就是存的 `strong` 或者 `weak` 的指针地址。我们可以看到在 `strongLayout` 中，高位x8代表非 `strong` 类型所占的内存地址，低位代表 `strong` 类型的个数，在 `weakLayout` 中，高位x8代表非 `weak` 类型所占的内存地址，低位代表 `weak` 类的个数。

> **关注下面的标签，发现更多相似文章**

iOS

评论

JungHsu Lv1 iOS开发...

呃，CPU是否以单字节存取和效率有关么？字节对齐是按当前类型所占字节数的倍数来做地址编排的

2月前 回复

欧阳大哥2013 Lv3 乐于分...

内存对齐是8个字节的结论不完全正确，确切的说是和CPU的字长有关，或者和操作系统有关。

3月前 2 回复

hoping_sir Lv1 (作者)iOS工程...

回复 欧阳大哥2013 Lv3: 谢大佬指点，那我严谨的说是不是应该是arm64下iOS 的内存对齐是8个字节

3月前

欧阳大哥2013 Lv3 乐于分...

回复 欧阳大哥2013 Lv3: 应该是64位的系统是8字节，32位的系统4字节。

3月前

加载更多

相关推荐

专栏Yuqi · 17小时前 · 掘金翻译计划 / Flutter

[译] Flutter 布局备忘录

18

专栏孙杰同志 · 23小时前 · iOS

J_Knight_ iOS 高级面试题 基础题解答

8

专栏QiShare · 9小时前 · iOS

iOS 常用布局方式之Masonry

4

专栏咆哮的蜗牛 · 11小时前 · iOS

对 iOS 中 GPU 编程的高度优化的框架 Metal

2

专栏ben123 · 20小时前 · iOS

localStorage关于ios10以下隐私模式不可用的问题

3 1

专栏一线搬砖工人 · 4天前 · WWDC / iOS

【WWDC2019 Session】Xcode 11新特性

37 5

专栏FreeBSFree · 23小时前 · iOS

YYClassInfo源码解析

1

专栏卡布骑诺 · 23小时前 · iOS

iOS-如何跳转至WhatsApp指定联系人对话界面

专栏portandbridge · 15小时前 · 机器学习 / iOS

[译] 用于 iOS 的 ML Kit 教程：识别图像中的文字

2

阴明 · 7天前 · Apple / Mac / macOS

苹果 WWDC 2019 最全记录：iPad 支持鼠标！系统独立，最强电脑亮相

34 3