登录 | 注册

黄文臣的专栏

脚踏实地, 不忘初心





黄文臣

关注 发私信



等级:

访问: 1085220次

积分: 13476

排名: 第1014名

BLOG > 7

原创: 284篇

译文: 0篇 评论: 211条

友情链接

我的Github

转载: 0篇

我的StackOverflow

博客专栏



iOS项目实战

文章: 1篇

阅读: 1412



iOS开发详解

文章: 67篇 阅读: 259968



校招面试

文章: 2篇

阅读: 4247

Swift下的GCD详解

文章: 5篇



异步赠书: 10月Python畅销书升级 618取胜之道、质量保障、技术解密) 【线路图】人工智能到底学什么?!

程序员9月书讯

每周荐书(京东篇):

Swift进阶之内存模型和方法调度

标签: Swift 内存模型 方法调度 优化 进阶

2016-11-13 16:08

4559人阅读

评论(1)

Ⅲ 分类:

Swift (22) -

▮ 版权声明:本文为博主原创文章,如需转载请注明出处

目录(?) [+]

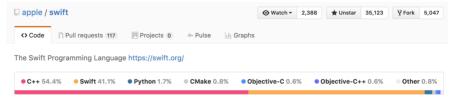
前言

Apple今年推出了Swift3.0,较2.3来说,3.0是一次重大的升级。关于这次更新,在这里都可以找 到,最主要的还是提高了Swift的性能,优化了Swift API的设计(命名)规范。

前段时间对之前写的一个项目ImageMaskTransition做了简单迁移,先保证能在3.0下正常运行,只 用了不到30分钟。总的来说,这次迁移还是非常轻松的。但是,有一点要注意: 3.0的API设计规范 较2.3有了质变,建议做迁移的开发者先看下WWDC的Swift API Design Guidelines。后面有时间 了,我有可能也会总结下。

内存分配

通过查看Github上Swift的源代码语言分布



可以看到

- Swift语言是用C++写的
- Swift的核心Library是用Swift自身写的。

对于C++来说,内存区间如下

- 堆区
- 栈区



阅读: 49016



Swift实用技术

文章: 14篇 阅读: 68931



由浅至深学习Linux

文章: 14篇 阅读: 87282

Coca and Casa Nucle Bild sells JUL compler Opticitàr and Accessive AMC entering insurpress. Same author as Object C. learning swift with hwc

文章: 18篇 阅读: 47505

文章分类

自己的iOS库 (11)

iOS进阶 (31)

Swift (23)

React Native (7)

React (0)

数据持久化+CoreData (16)

iOS基础 (29)

Swift入门教程 (1.0) (30)

CoreAnimation (16)

Foundation (14)

UlKit (23)

iOS 网络编程 (8)

iOS 多线程 (6)

C++ (14)

Linux (17)

小技巧 (10)

综合 (11)

设计模式 (4)

吐槽 (1)

文章存档

2017年09月 (1)

2017年08月 (1)

2017年07月 (2)

2017年06月 (3)

2017年05月 (1)

展开

阅读排行

React Native开发之IDE(Ato... (42443)
Linux Sed命令详解+如何替... (34859)
RxSwift使用教程 (23027)
为React Native开发写的JS和... (21001)
Linux时间同步+国内常用的... (19268)
React Native开发之动画(Ani... (18495)

完整详解GCD系列(一)disp...

代码区

全局静态区

Swift的内存区间和C++类似。也有存储代码和全局变量的区间,这两种区间比较简单,本文更多专注于以下两个内存区间。

- Stack(栈),存储值类型的临时变量,函数调用栈,引用类型的临时变量指针
- Heap(堆),存储引用类型的实例

栈

举个很简单的例子: 当一个递归函数, 陷入死循环, 那么最后函数调用栈会溢出。

例如,一个没有引用类型Struct的临时变量都是在栈上存储的

```
struct Point{
    var x:Double // 8 Bytes
    var y:Double // 8 bytes

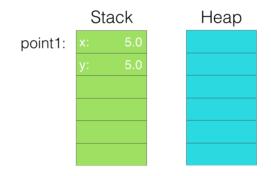
let size = MemoryLayout<Point>.size
    print(size) // 16

let point1 = Point(x:5.0,y:5.0)

let instanceSize = MemoryLayout<Point>.size(ofValue: point1)

print(instanceSize) //16
```

那么,这个内存结构如图



Tips: 图中的每一格都是一个Word大小,在64位处理器上,是8个字节

堆

在堆上可以动态的按需分配内存,每次在堆上分配内存的时候,需要查找堆上能提供相应大小的位置,然后返回对应位置,标记指定位置大小内存被占用。

在堆上能够动态的分配所需大小的内存,但是由于每次要查找,并且要考虑到多线程之间的线程安全问题,所以性能较栈来说低很多。

比如, 我们把上文的 struct 改成 class,

(16499)

React Native的Navigator详解 (16199) 完整详解GCD系列(二)disp... (15968) iOS NSAttributedString所有... (13531)

评论排行	
React Native开发之IDE(Ato	(19)
完整详解GCD系列(一)disp	(13)
优雅的开发Swift和Objective	(12)
Estimote的蓝牙数据包	(11)
React Native的Navigator详解	(8)
iOS 单元测试之XCTest详解	(7)
React Native入门一实战解析	(6)
iOS编译过程的原理和应用	(5)
iOS SDK详解之视频播放(A	(5)
Swift App状态恢复-State R	(5)

Swift进阶之内存模型和方法调度 - 黄文臣的专栏 - CSDN博客

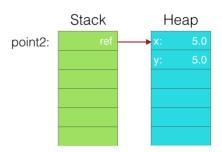
```
class PointClass{
   var x:Double = 0.0
   var y:Double = 0.0

let size2 = MemoryLayout<PointClass>.size
   print(size2) //8

let point2 = Point(x:5.0,y:5.0)

let instanceSize = MemoryLayout<Point>.size(ofValue: point2)
   print(instanceSize) //8
```

这时候的内存结构如图



Tips: 图中的每一格都是一个Word大小,在64位处理器上,是8个字节

Memory Alignment (内存对齐)

和C/C++/OC类似,Swift也有Memory Alignment的概念。举个直观的例子 我们定义这样两个Struct

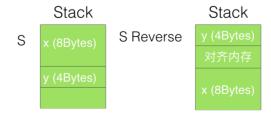
```
1  struct S{
2     var x:Int64
3     var y:Int32
4  }
5  struct SReverse{
6     var y:Int32
7     var x:Int64
8  }
```

然后,用MemoryLayout来获取两个结构体的大小

```
1 let sSize = MemoryLayout<S>.size //12
2 let sReverseSize = MemoryLayout<SReverse>.size //16
```

可以看到,只不过调整了结构体中的声明顺序,其占用的内存大小就改变了,这就是内存对齐。

我们来看看,内存对齐后的内存空间分布:



内存对齐的原因是,

现代CPU每次读数据的时候,都是读取一个word(32位处理器上是4个字节,64位处理器上是8个字节)。

内存对齐的优点很多

保证对一个成员的访问在一个Transition中,提高了访问速度,同时还能保证一次操作的原子性。除了这些,内存对齐还有很多优点,可以看看这个SO答案

自动引用计数(ARC)

提到ARC,不得不先讲讲Swift的两种基本类型:

- 值类型, 在赋值的时候, 会进行值拷贝
- 引用类型,在赋值的时候,只会进行引用(指针)拷贝

比如, 如下代码

```
1 struct Point{ //Swift中, struct是值类型
2
       var x,y:Double
3 }
 4 class Person{//Swift中, class是引用类型
      var name:String
 5
      var age:Int
 6
      init(name:String,age:Int){
 7
          self.name = name
 8
9
           self.age = age
10
11 }
12 var point1 = Point(x: 10.0, y: 10.0)
13 var point2 = point1
14 point2.x = 9.0
15 print(point1.x) //10.0
17 | var person1 = Person(name: "Leo", age: 24)
18 var person2 = person1
19 person2.age = 25
20 print(person1.age)//9.0
```

我们先看看对应内存的使用 值类型有很多优点,其中主要的优点有两个 - 线程安全,每次都是获得一个copy,不存在同时修改一块内存 - 不可变状态,使用值类型,不需要考虑别处的代码可能会对当前代码有影响。也就没有side effect。 ARC是相对于引用类型的。 > ARC是一个内存管理机制。当一个引用类型的对象的reference count(引用计数)为0的时候,那么这个对象会被释放掉。 我们利用XCode 8和iOS开发,来直观的查看下一个值类型变量的引用计数变化。 新建一个iOS单页面工程,语言选择Swift,然后编写如下代码![这里写图片描述]

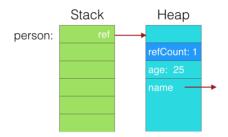
(http://img.blog.csdn.net/20161113111539024) 然后, 当断点停在24行处的时候, Person的引用



Thread 0 Thread_2673...ple.main-thread (serial)

计数如下 这里,底部 2233

是主线程堆Person对象的持有,是iOS系统默认添加。所以,`var leo = Person(nar 25)`这一行后,准确的说是引用计数加一,并不是引用计数为一。当然,这些系统 动销毁,我们无须考虑。 可以看到,person唯一的引用就是来自`VM:Stack threa 上。 因为引用计数的存在,Class在堆上需要额外多分配一个Word来存储引用计数:



当栈上代码执行完毕,栈会断掉对Person的引用,引用计数也就减一,系统会断掉自动创建的引用。这时候,person的引用计数位0,内存释放。

方法调度(method dispatch)

Swift的方法调度分为两种

- 静态调度 static dispatch. 静态调度在执行的时候,会直接跳到方法的实现,静态调度可以进行inline和其他编译期优化。
- 动态调度 dynamic dispatch. 动态调度在执行的时候,会根据运行时(Runtime),采用table的方式,找到方法的执行体,然后执行。动态调度也就没有办法像静态那样,进行编译期优化。

Struct

对于Struct来说,方法调度是静态的。

```
1 struct Point{
2    var x:Double // 8 Bytes
3    var y:Double // 8 bytes
4    func draw(){
5        print("Draw point at\(x,y)")
6    }
7 }
8 let point1 = Point(x: 5.0, y: 5.0)
```

≐

```
point1.draw()
print(MemoryLayout<Point>.size) //16
```

可以看到,由于是Static Dispatch,在编译期就能够知道方法的执行体。所以,在Runtime也就不需要额外的空间来存储方法信息。编译后,方法的调用,直接就是变量地址的传入,存在了代码区中

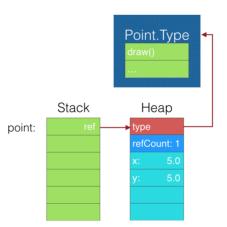
如果开启了编译器优化,那么上述代码被优化成Inline后,

```
1 let point1 = Point(x: 5.0, y: 5.0)
2 print("Draw point at\(point1.x,point1.y)")
3 print(MemoryLayout<Point>.size) //16
```

Class

Class是Dynamic Dispatch的,所以在添加方法之后,Class本身在栈上分配的仍然是一个word 堆上,需要额外的一个word来存储Class的Type信息,在Class的Type信息中,存储着virtual table(V-Table)。根据V-Table就可以找到对应的方法执行体。

```
1 class Point{
       var x:Double // 8 Bytes
2
       var y:Double // 8 bytes
 3
       init(x:Double,y:Double) {
 4
 5
            self.x = x
 6
            self.y = y
 7
       }
 8
       func draw(){
            print("Draw point at\(x,y)")
9
10
11
12 let point1 = Point(x: 5.0, y: 5.0)
13 point1.draw()
14 print(MemoryLayout<Point>.size) //8
```



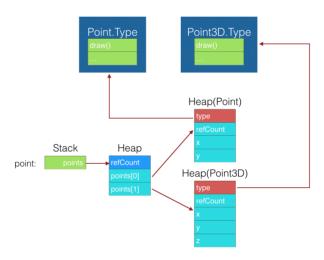
继承

因为Class的实体会存储额外的Type信息,所以继承理解起来十分容易。子类只需要存储子类的 Type信息即可。

例如

```
class Point{
  var x:Double // 8 Bytes
  var y:Double // 8 bytes
  init(x:Double,y:Double) {
```

```
self.x = x
5
 6
            self.y = y
 7
       }
 8
        func draw(){
9
            print("Draw point at\(x,y)")
10
11
   class Point3D:Point{
12
       var z:Double // 8 Bytes
13
       init(x:Double,y:Double,z:Double) {
14
            self.z = z
15
            super.init(x: x, y: y)
16
17
       }
18
       override func draw(){
           print("Draw point at\(x,y,z)")
19
20
21 }
22 let point1 = Point(x: 5.0, y: 5.0)
23 let point2 = Point3D(x: 1.0, y: 2.0, z: 3.0)
24 let points = [point1,point2]
25 points.forEach { (p) in
26
       p.draw()
27
28 //Draw point at(5.0, 5.0)
29 //Draw point at(1.0, 2.0, 3.0)
```



协议

我们首先看一段代码

```
struct Point:Drawable{
 1
       var x:Double // 8 Bytes
2
       var y:Double // 8 bytes
3
        func draw(){
 4
 5
            print("Draw point at\(x,y)")
6
 7
 8
   struct Line:Drawable{
9
       var x1:Double // 8 Bytes
10
       var y1:Double // 8 bytes
       var x2:Double // 8 Bytes
11
       var y2:Double // 8 bytes
12
       func draw(){
13
            print("Draw line from (x1,y1) to (x2,y2)")
14
15
16 }
17 let point = Point(x: 1.0, y: 2.0)
18 let memoryAsPoint = MemoryLayout<Point>.size(ofValue: point)
19 let memoryOfDrawable = MemoryLayout<Drawable>.size(ofValue: point)
20 print(memoryAsPoint)
```

Swift进阶之内存模型和方法调度 - 黄文臣的专栏 - CSDN博客

```
print(memoryOfDrawable)

22

23 let line = Line(x1: 1.0, y1: 1.0, x2: 2.0, y2: 2.0)

24 let memoryAsLine = MemoryLayout<Line>.size(ofValue: line)

25 let memoryOfDrawable2 = MemoryLayout<Drawable>.size(ofValue: line)

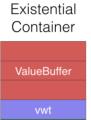
26 print(memoryAsLine)

27 print(memoryOfDrawable2)
```

可以看到,输出

```
1 16 //point as Point
2 40 //point as Drawable
3 32 //line as Line
4 40 //line as Drawable
```

16和32不难理解,Point含有两个Double属性,Line含有四个Double属性。对应的的。那么,两个40是怎么回事呢?而且,对于Point来说,40-16=24,多出了24个字来说,只多出了40-32=8个字节。 这是因为Swift对于协议类型的采用如下的内存模Existential Container。



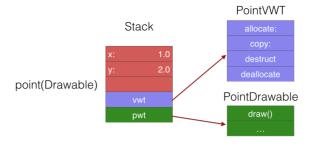
pwt

Existential Container包括以下三个部分:

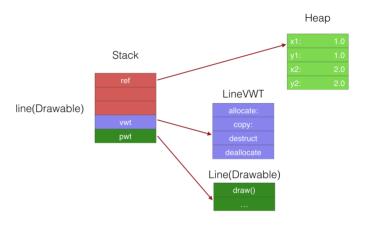
- 前三个word: Value buffer。用来存储Inline的值,如果word数大于3,则采用指针的方式, 在堆上分配对应需要大小的内存
- 第四个word: Value Witness Table(VWT)。每个类型都对应这样一个表,用来存储值的创建,释放,拷贝等操作函数。
- 第五个word: Protocol Witness Table(PWT), 用来存储协议的函数。

那么, 内存结构图, 如下

ne



[point]



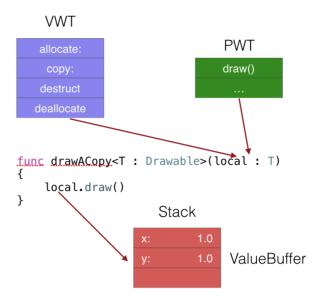
[line]

范型

范型让代码支持<mark>静态多态</mark>。比如:

```
func drawACopy<T : Drawable>(local : T) {
  local.draw()
}
drawACopy(Point(...))
func drawACopy(Point(...))
```

那么,范型在使用的时候,如何调用方法和存储值呢?



[范型]

范型并不采用Existential Container, 但是原理类似。

- 1. VWT和PWT作为隐形参数,传递到范型方法里。
- 2. 临时变量仍然按照ValueBuffer的逻辑存储 分配3个word,如果存储数据大小超过3个word,则在堆上开辟内存存储。

范型的编译器优化

1. 为每种类合成具体的方法

比如

```
func drawACopy<T : Drawable>(local : T) {
local.draw()
}
```

在编译过后,实际会有两个方法

```
func drawACopyOfALine(local : Line) {
  local.draw()

func drawACopyOfAPoint(local : Point) {
  local.draw()
}
```

然后,

```
drawACopy(local: Point(x: 1.0, y: 1.0))
```

会被编译成为

```
func drawACopyOfAPoint(local : Point(x: 1.0, y: 1.0))
```

Swift的编译器优化还会做更多的事情,上述优化虽然代码变多,但是编译器还会对代码进行压缩。 所以,实际上,并不会对二进制包大小有什么影响。

参考资料

- WWDC 2016 Understanding Swift Performance
- WWDC 2015 Optimizing Swift Performance
- Does Swift guarantee the storage order of fields in classes and structs?

T页

上一篇 记第一次跳槽

下一篇 iOS编译过程的原理和应用

相关文章推荐

- 对象存储系统Swift技术详解: 综述与概念
- 机器学习之数学基础系列--Al100
- Swift iOS 9通讯录访问
- 使用Keras快速构造自己的深度学习模型--谢梁
- iOS友盟消息推送总是推送失败或者token无效
- 跳过Java开发的各种坑
- swift 中"?"和"!"区别以及相关用法
- Android自动化刷量、作弊与防作弊

- Java进阶之内存模型&并发编程
- Retrofit 从入门封装到源码解析
- 使用Pandas与Matplotlib分析科比职业生涯数据
- Swift 对象内存模型探究(一)
- Swift 学习进阶
- swift进阶学习
- swift入门进阶

查看评论



扬起帆前行

您好, 您上面 讲解 arc 那, 是 堆里面的代码 执行完后, 栈内的res 会断开吧?

1楼 前天 16:05发表

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点,不代表CSDN网站的观点或立场

公司简介 | 招贤纳士 | 广告服务 | 联系方式 | 版权声明 | 法律顾问 | 问题报告 | 合作伙伴 | 论坛反馈

网站客服 杂志客服 微博客服 webmaster@csdn.net 400-660-0108 | 北京创新乐知信息技术有限公司 版权所有 | 江苏知之为计算机有限公司 |

江苏乐知网络技术有限公司

京 ICP 证 09002463 号 | Copyright © 1999-2017, CSDN.NET, All Rights Reserved

