Swift第二节课：类与结构体（下）

一，异变方法

上一节课我们了解到，Swift 中 class 和 struct 都能定义方法。但是有一点区别的是默认情况下，值类型属性不能被自身的实例方法修改。

```
 struct Point {
    var x = 0.0, y = 0.0
    func moveBy(x deltaX: Double, y deltaY: Double) {
        //self
        x += deltaX
        y += deltaY
    }
}
```

通过 `SIL` 来对比一下，不添加 `mutating` 访问和添加 `mutating` 两者有什么本质的区别

```
 struct Point {
    var x = 0.0, y = 0.0

    func test(){
        let tmp = self.x
    }

    mutating func moveBy(x deltaX: Double, y deltaY: Double) {
        x += deltaX
        y += deltaY
    }
}
```

```
sil hidden [ossa] @$s4main5PointV4testyyF : $@convention(method) (Point) ->

debug_value %0 : $Point, let, name "self", argno 1 // id: %1
```

```
sil hidden [ossa] @$s4main5PointV6moveBy1x1yySd_SdtF : $@convention(method)

@inout Point

```

```
debug_value_addr %2 : $*Point, var, name "self", argno 3 // id: %5

let self = Point
var self = &Point

```

SIL 文档的解释
An @inout parameter is indirect. The address must be of an initialized object.（当前参数类型是间接的，传递的是已经初始化过的地址）

异变方法的本质：**对于变异方法, 传入的 `self` 被标记为 `inout` 参数。无论在 `mutating` 方法内部发生什么，都会影响外部依赖类型的一切。**

输入输出参数：如果我们想函数能够修改一个形式参数的值，而且希望这些改变在函数结束之后依然生效，那么就需要将形式参数定义为 `输入输出形式参数` 。在形式参数定义开始的时候在前边添加一个 inout关键字可以定义一个输入输出形式参数

二、方法调度

objc_mgsend

我们先来看一下 `Swift` 中的方法调度

```
class LGTeacher{
    func teach(){
        print("teach")
    }
}

var t = LGTeacher()
t.teach()
```

teach函数的调用过程：找到 `Metadata` ，确定函数地址（metadata + 偏移量）， 执行函数

基于函数表的调度

之前我们在第一节课讲到了 `Metdata` 的数据结构，那么 `V-Table` 是存放在什么地方那?
我们先来回顾一下当前的数据结构

```
 struct Metadata{
    var kind: Int
    var superClass: Any.Type
    var cacheData: (Int, Int)
    var data: Int
```

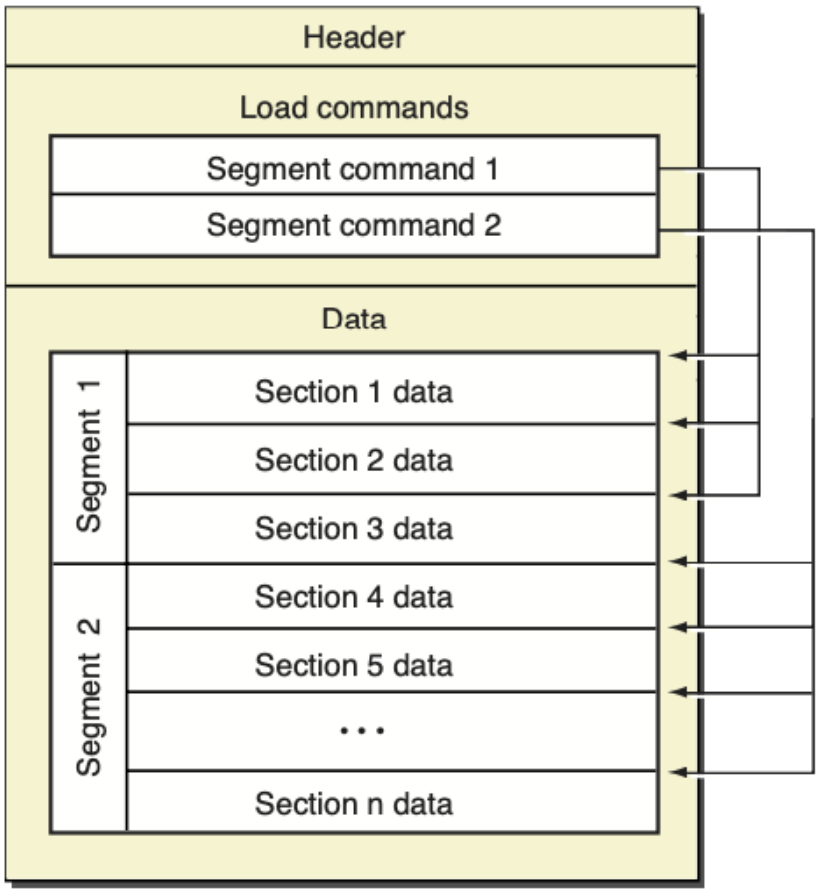
```
    var classFlags: Int32
    var instanceAddressPoint: UInt32
    var instanceSize: UInt32
    var instanceAlignmentMask: UInt16
    var reserved: UInt16
    var classSize: UInt32
    var classAddressPoint: UInt32
    var typeDescriptor: UnsafeMutableRawPointer
    var iVarDestroyer: UnsafeRawPointer
}
```

这里我们有一个东西需要关注 `typeDescriptor` ，不管是 `Class` ， `Struct` ， `Enum` 都有自己的 `Descriptor` ，就是对类的一个详细描述

```
struct TargetClassDescriptor{
    var flags: UInt32
    var parent: UInt32
    var name: Int32
    var accessFunctionPointer: Int32
    var fieldDescriptor: Int32
    var superClassType: Int32
    var metadataNegativeSizeInWords: UInt32
    var metadataPositiveSizeInWords: UInt32
    var numImmediateMembers: UInt32
    var numFields: UInt32
    var fieldOffsetVectorOffset: UInt32
    var Offset: UInt32
    var size: UInt32
    //V-Table
}
```

Mahco: Mach-O 其实是Mach Object文件格式的缩写，是 mac 以及 iOS 上可执行文件的格式， 类似于 windows 上的 PE 格式 (Portable Executable), linux 上的 elf 格式 (Executable and Linking Format）。常见的 .o，.a .dylib Framework，dyld .dsym。

Mahoc文件格式：



- 首先是文件头，表明该文件是 Mach-O 格式，指定目标架构，还有一些其他的文件属性信息，文件头信息影响后续的文件结构安排
- Load commands是一张包含很多内容的表。内容包括区域的位置、符号表、动态符号表等。

LC_SEGMENT_64	将文件中（32位或64位）的段映射到进程地址空间中
LC_DYLD_INFO_ONLY	动态链接相关信息
LC_SYMTAB	符号地址
LC_DYSYMTAB	动态符号表地址
LC_LOAD_DYLINKER	dyld加载
LC_UUID	文件的UUID
LC_VERSION_MIN_MACOSX	支持最低的操作系统版本
LC_SOURCE_VERSION	源代码版本
LC_MAIN	设置程序主线程的入口地址和栈大小
LC_LOAD_DYLIB	依赖库的路径，包含三方库
LC_FUNCTION_STARTS	函数起始地址表
LC_CODE_SIGNATURE	代码签名

- Data 区主要就是负责代码和数据记录的。Mach-O 是以 Segment 这种结构来组织数据的，一个 Segment 可以包含 0 个或多个 Section。根据 Segment 是映射的哪一个 Load Command，Segment 中 section 就可以被解读为是是代码，常量或者一些其他的数据类型。在装载在内存中时，也是根据 Segment 做内存映射的。

方法调度方式总结：

类型	调度方式	extension
值类型	静态派发	静态派发
类	函数表派发	静态派发
NSObject子类	函数表派发	静态派发

三、影响函数派发方式

- final： 添加了 final 关键字的函数无法被重写，使用静态派发，不会在 vtable 中出现，且对 objc 运行时不可见。
- dynamic: 函数均可添加 dynamic 关键字，为非objc类和值类型的函数赋予动态性，但派发方式还是函数表派发。
- @objc： 该关键字可以将Swift函数暴露给Objc运行时，依旧是函数表派发。
- @objc + dynamic： 消息派发的方式

四、函数内联

函数内联 是一种编译器优化技术，它通过使用方法的内容替换直接调用该方法，从而优化性能。

- 将确保有时内联函数。这是默认行为，我们无需执行任何操作. Swift 编译器可能会自动内联函数作为优化。
- always – 将确保始终内联函数。通过在函数前添加 @inline(__always) 来实现此行为
- never – 将确保永远不会内联函数。这可以通过在函数前添加 @inline(never) 来实现。
- 如果函数很长并且想避免增加代码段大小，请使用@inline(never)（使用@inline(never)）

实际开发过程中属性，方法，类不需要被重载，001–