Swift第一节课：类与结构体（上）

一、初识类与结构体

我们先来看一段代码

```
struct/class LGTeacher{
    var age: Int
    var name: String

    init(age: Int, name: String) {
        self.age = age
        self.name = name
    }

    deinit{

    }
}
```

结构体和类的主要相同点有：

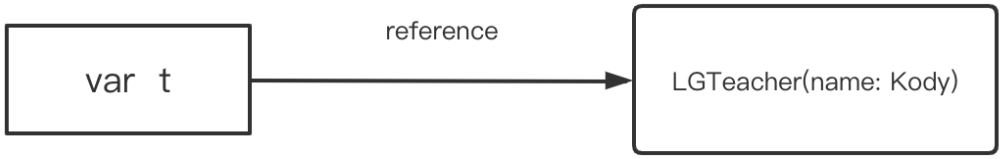
- 定义存储值的属性
- 定义方法
- 定义下标以使用下标语法提供对其值的访问
- 定义初始化器
- 使用 extension 来拓展功能
- 遵循协议来提供某种功能

主要的不同点有：

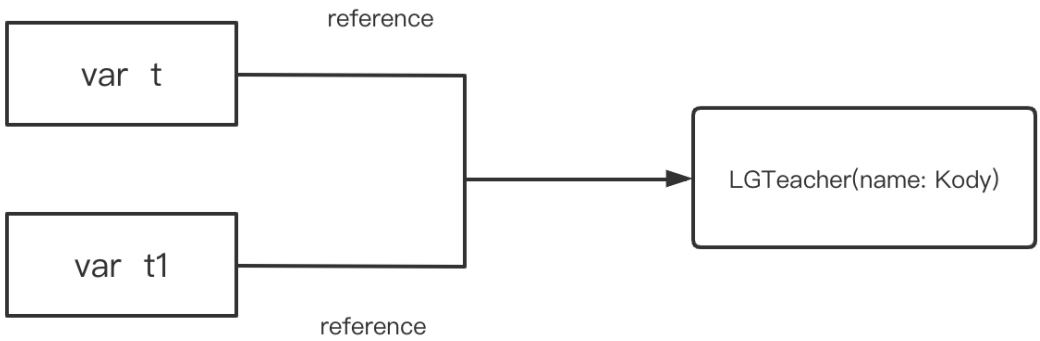
- 类有继承的特性，而结构体没有
- 类型转换使您能够在运行时检查和解释类实例的类型
- 类有析构函数用来释放其分配的资源
- 引用计数允许对一个类实例有多个引用

对于类与结构体我们需要区分的第一件事就是：

类是引用类型。也就意味着一个类类型的变量并不直接存储具体的实例对象，是对当前存储具体实例内存地址的引用。




```
var t1 = t
```

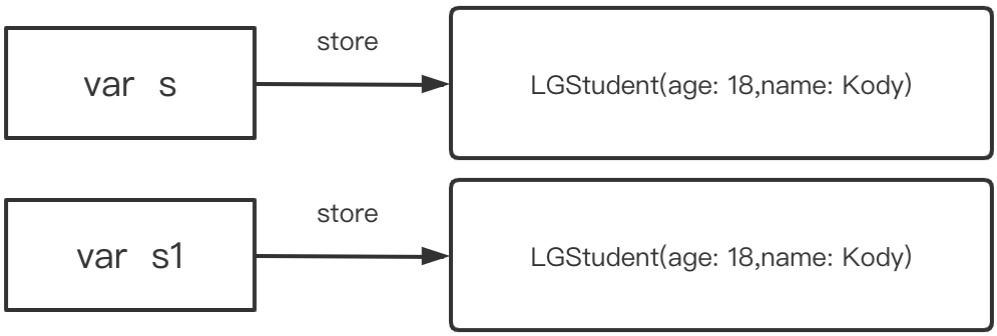


这里我们借助两个指令来查看当前变量的内存结构

```
po ： p 和 po 的区别在于使用 po 只会输出对应的值，而 p 则会返回值的类型以及命令结果的引用名。

x/8g: 读取内存中的值(8g: 8字节格式输出)
```

`swift` 中有引用类型，就有值类型，最典型的就是 Struct ,结构体的定义也非常简单，相比较类类型的变量中存储的是地址，那么值类型存储的就是具体的实例（或者说具体的值）。

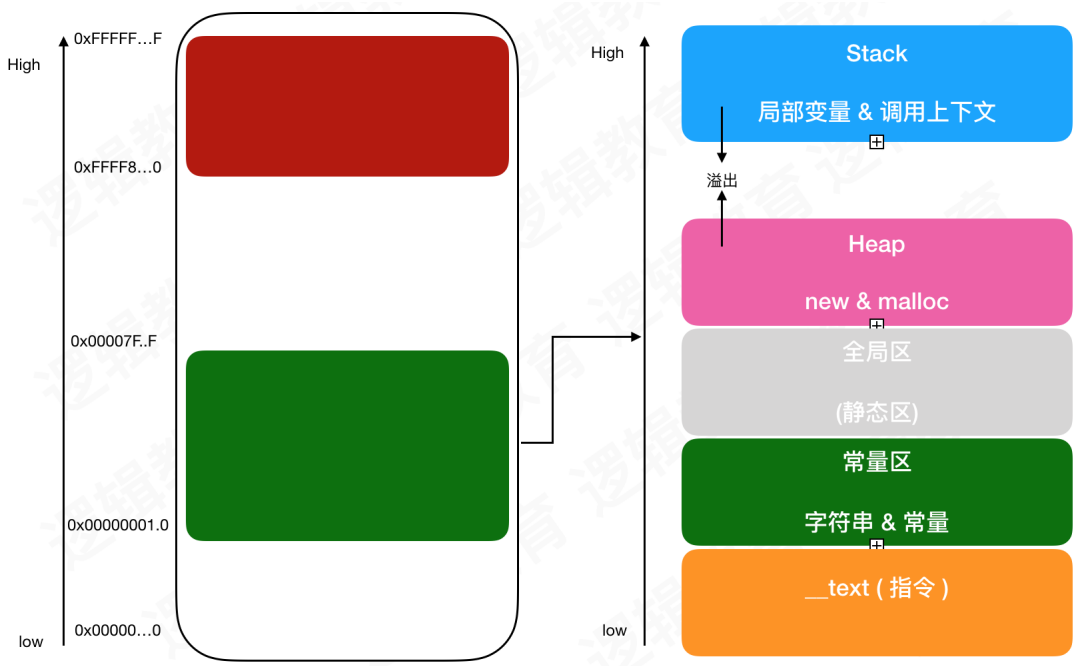
```
struct LGStudent{
    var age: Int
    var name: String
}
var s = LGStudent(age:18, name:kody)
var s1 = s
```



其实引用类型就相当于在线的 `Excel` ，当我们把这个链接共享给别人的时候，别人的修改我们是能够看到的；值类型就相当于本地的 `Excel` ,当我们把本地的 Excel 传递给别人的时候，就相当于重新复制了一份给别人，至于他们对于内容的修改我们是无法感知的。

另外引用类型和值类型还有一个最直观的区别就是存储的位置不同：一般情况，值类型存储的在栈上，引用类型存储在堆上。

首先我们对内存区域来一个基本概念的认知，大家看下面这张图



栈区（stack）：局部变量和函数运行过程中的上下文

```
//test是不是一个函数
func test(){
    //我们在函数内部声明的age变量是不是就是一个局部变量
    var age: Int = 10
    print(age)
}
```

Heap：存储所有对象

Global: 存储全局变量；常量；代码区

Segment & Section: `Mach-O` 文件有多个段（`Segment`），每个段有不同的功能。然后每个段又分为很多小的 `Section`

TEXT.text ： 机器码
TEXT.cstring ： 硬编码的字符串
TEXT.const： 初始化过的常量
DATA.data： 初始化过的可变的（静态/全局）数据
DATA.const： 没有初始化过的常量
DATA.bss： 没有初始化的（静态/全局）变量

DATA.common： 没有初始化过的符号声明

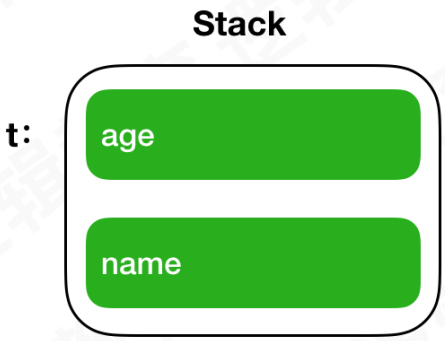
我们来看例子

```
struct LGTeacher{
    var age = 18
    var name = "Kody"
}

func test(){
    var t = LGTeacher()
    print("end")
}

test()
```

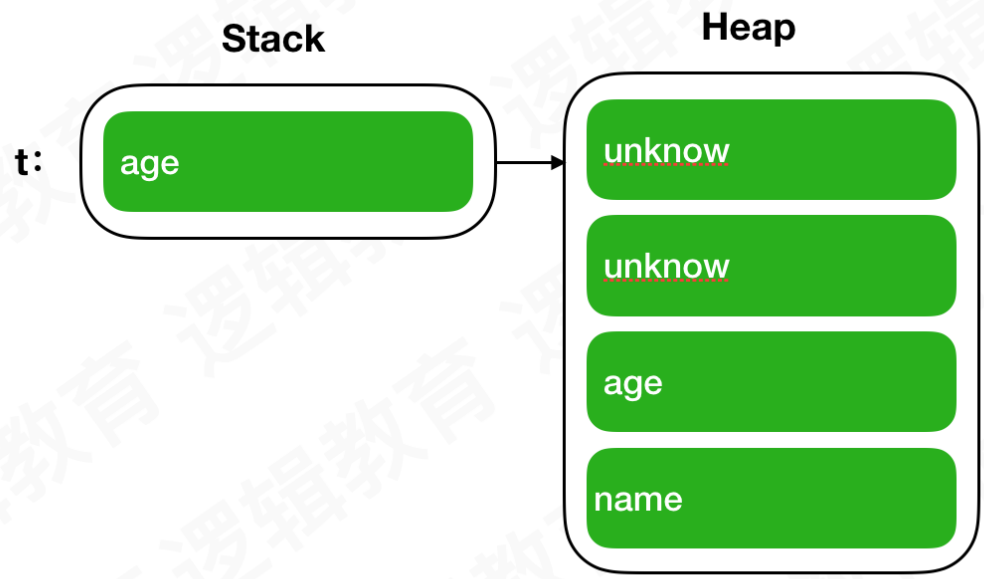
接下来使用命令

```
frame varibale -L xxx
```

当前结构体在内存当中的分布示意图：



如果我们把其他条件不变，将 `strcut` 修改成 `class` 的情况我们来看一下：



这里我们也可以通过github上StructVsClassPerformance这个案例来直观的测试当前结构体和类的时间分配。

我们来看两个官方案例

```
enum Color { case blue, green, gray }
enum Orientation { case left, right }
enum Tail { case none, tail, bubble }

var cache = [String : UIImage]()

func makeBalloon(_ balloon: Ballon) -> UIImage {
   if let image = cache[balloon] {
      return image
   }
   ...
}

struct Balloon: Hashable{
  var color: Color
  var orientation: Orientation
  var tail: Tail
}
```

```
struct Attachment {
   let fileURL: URL
   let uuid: UUID
   let mineType: MimeType

  init?(fileURL: URL, uuid: String, mimeType: String) {
      guard mineType.isMineType
      else { return nil }

      self.fileURL = fileURL
      self.uuid = uuid
      self.mineType = mimeType
} }

enum MimeType: String{
    case jpeg = "image/jpeg"
  ....
}
```

二、类的初始化器

需要注意的一点是：当前的类编译器默认不会自动提供成员初始化器，但是对于结构体来说编译器会提供默认的初始化方法（前提是我们自己没有指定初始化器）！

```
struct LGTeacher{
  var age: Int
  var name: String
}
```

Swift 中创建类和结构体的实例时必须为所有的存储属性设置一个合适的初始值。所以类 LGPerson 必须要提供对应的指定初始化器，同时我们也可以为当前的类提供便捷初始化器(注意：便捷初始化器必须从相同的类里调用另一个初始化器。)

```
class LGPerson{
    var age: Int
    var name: String
    init(_ age: Int, _ name: String) {
        self.age = age
        self.name = name
    }

    convenience init() {
        self.init(age: 18, name:"Kody")
    }
}
```

当我们派生出一个子类 LGTeacher ,并指定一个指定初始化器之后会出现什么问题

```
class LGPerson{
    var age: Int
    var name: String
    init(_ age: Int, _ name: String) {
        self.age = age
        self.name = name
    }

    convenience init() {
        self.init(age: 18, name:"Kody")
    }
}

class LGTeacher: LGPerson{
  var subjectName: String

  init(subjectName: String){
    self.subjectName = subjectName
  }
```

```
20 }
```

这里我们记住：

- 指定初始化器必须保证在向上委托给父类初始化器之前，其所在类引入的所有属性都要初始化完成。
- 指定初始化器必须先向上委托父类初始化器，然后才能为继承的属性设置新值。如果不这样做，指定初始化器赋予的新值将被父类中的初始化器所覆盖
- 便捷初始化器必须先委托同类中的其它初始化器，然后再为任意属性赋新值（包括同类里定义的属性）。如果没这么做，便捷构初始化器赋予的新值将被自己类中其它指定初始化器所覆盖。
- 初始化器在第一阶段初始化完成之前，不能调用任何实例方法、不能读取任何实例属性的值，也不能引用 self 作为值。

可失败初始化器： 这个也非常好理解，也就意味着当前因为参数的不合法或者外部条件的不满足，存在初始化失败的情况。这种 Swift 中可失败初始化器写 return nil 语句，来表明可失败初始化器在何种情况下会触发初始化失败。写法也非常简单:

```
class LGPerson{
    var age: Int
    var name: String
    init?(age: Int, name: String) {
        if age < 18 {return nil}
        self.age = age
        self.name = name
    }

    convenience init?() {
        self.init(age: 18, name: "Kody")
    }
}
```

比如这里我们定义了如果小于 18 就不是一个合法的成年人

必要初始化器：在类的初始化器前添加 required 修饰符来表明所有该类的子类都必须实现该初始化器

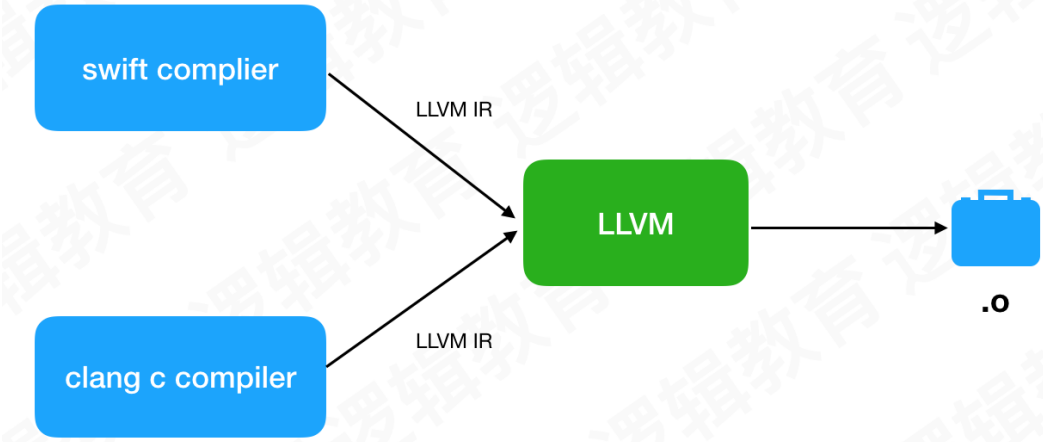
```
class LGPerson{
    var age: Int
    var name: String
    required init(age: Int, name: String) {
        self.age = age
        self.name = name
    }

    convenience init() {
        self.init(age: 18, name: "Kody")
    }
}

class LGTeacher: LGPerson{
    var subjectName: String

    init(subjectName: String) {
        self.subjectName = subjectName
        super.init(age: 18, name: "Kody")
    }

    convenience init() {
        self.init(subjectName: "[Unnamed]")
    }
}
```

如果在子类没有提供就会报错

'required' initializer 'init(age:name:)' must be provided by subclass of 'LGPerson'

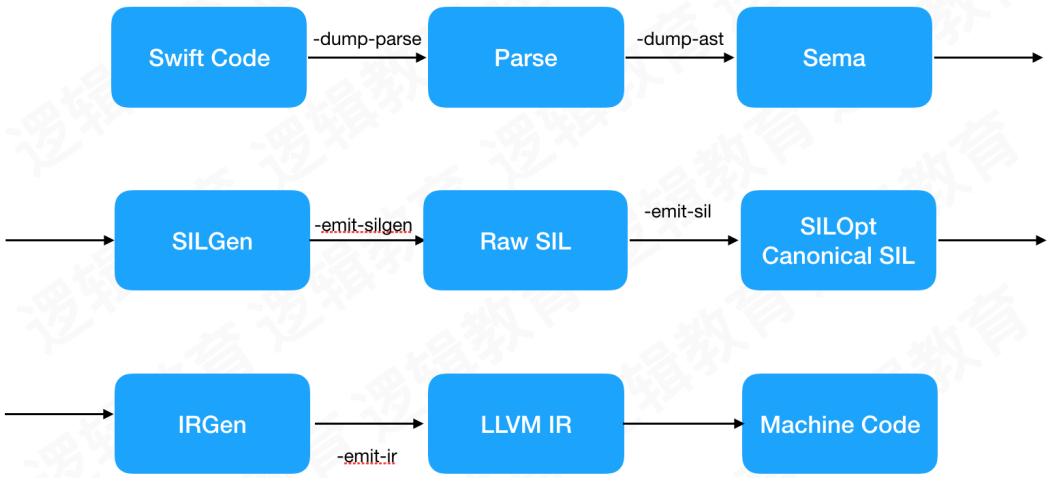
三、类的生命周期

iOS开发的语言不管是OC还是Swift后端都是通过LLVM进行编译的，如下图所示：



OC 通过 clang 编译器，编译成 IR，然后再生成可执行文件 .o(这里也就是我们的机器码)

Swift 则是通过 Swift 编译器编译成 IR，然后在生成可执行文件。



// 分析输出AST

```
// 分析输出AST
swiftc main.swift -dump-parse

// 分析并且检查类型输出AST
swiftc main.swift -dump-ast

// 生成中间体语言（SIL），未优化
swiftc main.swift -emit-silgen

// 生成中间体语言（SIL），优化后的
swiftc main.swift -emit-sil

```

```
// 生成LLVM中间体语言 （.ll文件）
swiftc main.swift -emit-ir

// 生成LLVM中间体语言 （.bc文件）
swiftc main.swift -emit-bc

// 生成汇编
swiftc main.swift -emit-assembly

// 编译生成可执行.out文件
swiftc -o main.o main.swift
```

@main： 入口函数， @
%0： 寄存器，虚拟的，真的寄存器
xcrun swift-demangle

Swift 对象内存分配：

- `__allocating_init` -----> `swift_allocObject` -----> `_swift_allocObject_` -----> `swift_slowAlloc` -----> `Malloc`
- Swift 对象的内存结构 HeapObject (OC objc_object)，有两个属性： 一个是 Metadata ，一个是 RefCount ，默认占用 16 字节大小。

objc_object{

isa

}

源码中 `kind` 种类

name	Value
Class	0x0
Struct	0x200
Enum	0x201
Optional	0x202
ForeignClass	0x203
ForeignClass	0x203
Opaque	0x300
Tuple	0x301
Function	0x302
Existential	0x303
Metatype	0x304
ObjCClassWrapper	0x305
ExistentialMetatype	0x306
HeapLocalVariable	0x400
HeapGenericLocalVariable	0x500
ErrorObject	0x501
LastEnumerated	0x7FF

类的结构：objc_class

经过源码分析我们不难得出 `swift` 类的数据结构

```
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
}
```