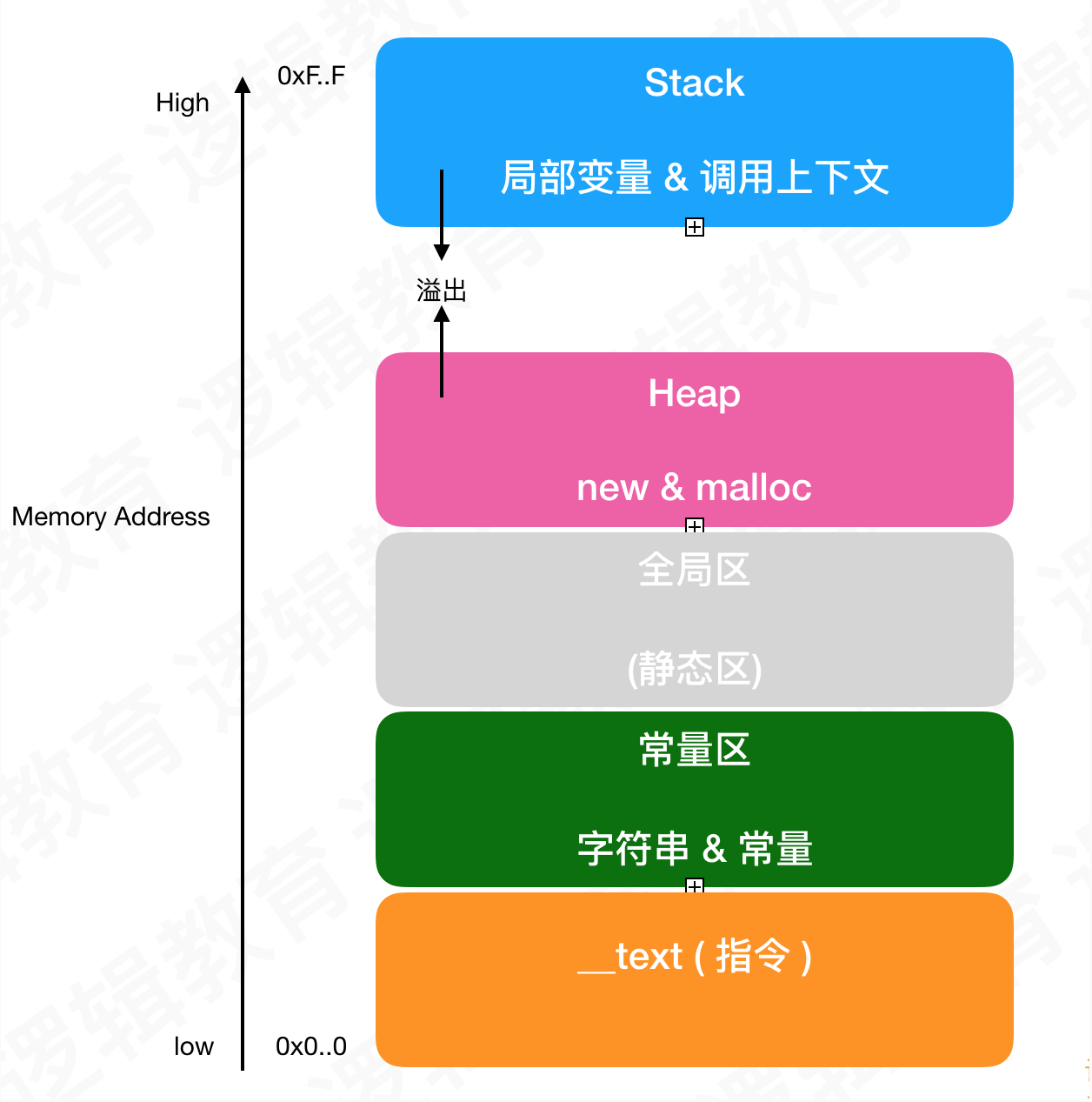
Swift 进阶第三节课：指针 & 内存管理

iOS内存分区

上节课我们看到了这样一张图：



其中栈区就是简单存放我们当前的：局部变量和函数运行过程中的上下文，比如我们举一个例子：

```
1 //test是不是一个函数
2 func test(){
3     //我们在函数内部声明的age变量就是一个局部变量
4     var age: Int = 10
5     print(age)
6 }
```

接下来我们使用命令来查看我们当前的 age 的内存地址：

```
po withUnsafePointer(to: &age){print($0)}
```

同样的，对于堆区的内存来说，就是通过 new & malloc 关键字来申请的内存空间，不连续，类似链表的结构，最直观的就是我们的对象。

```
class LGTeacher{
}


func test(){
    var t = LGTeacher()

    print(t)
}

test()
```

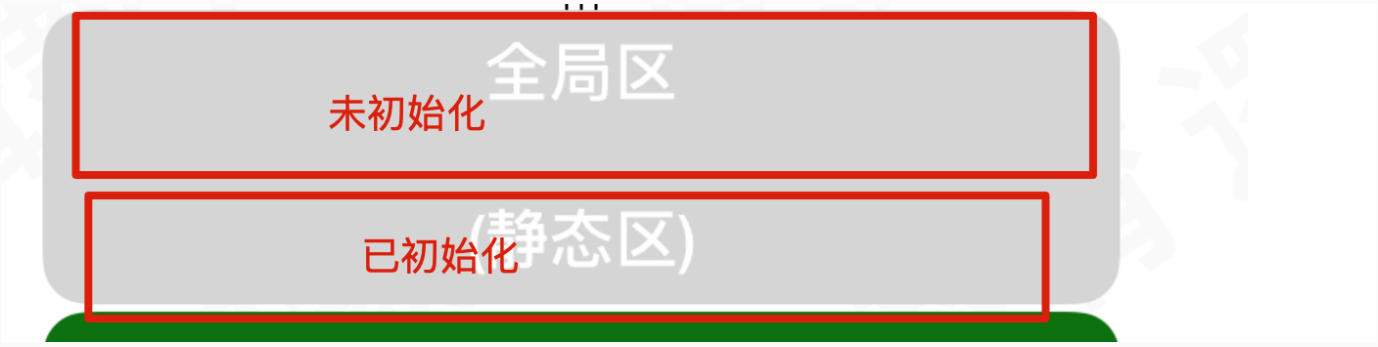
思考下面几个 int 值和 char 存放在哪里？并用工具验证：

```
int a = 10;

int age;

static int age2 = 30;

int main(int argc, const char * argv[]) {
    char *p = "LGTeacher";
    // insert code here...
    printf("%d",a);
    printf("%d",age2);
    return 0;
}
```



方法调度

对于结构体中的方法都是静态调用（直接调用），那对于类比的 class 中的方法那？我们类中声明的方法是通过 `V-table` 来进行调度的。

`V-Table` 在 `SIL` 中的表示是这样的：

```
1 decl ::= sil-vtable
2 sil-vtable ::= 'sil_vtable' identifier '{' sil-vtable-entry* '}'
3 sil-vtable-entry ::= sil-decl-ref ':' sil-linkage? sil-function-na
  me
```

这里我们通过一个简单的源文件来看一下：

```
class LGTeacher {
  func teach()
  func teach2()
  func teach3()
  func teach4()
  @objc deinit
  init()
}
```

这里是我们当前头文件的声明

文件拉到最后：(LGTeacher函数表)

```
sil_vtable LGTeacher {
  #LGTeacher.teach!1: (LGTeacher) -> () -> () : @main.LGTeacher.teach() -> () // LGTeacher.teach()
  #LGTeacher.teach2!1: (LGTeacher) -> () -> () : @main.LGTeacher.teach2() -> () // LGTeacher.teach2()
  #LGTeacher.teach3!1: (LGTeacher) -> () -> () : @main.LGTeacher.teach3() -> () // LGTeacher.teach3()
  #LGTeacher.teach4!1: (LGTeacher) -> () -> () : @main.LGTeacher.teach4() -> () // LGTeacher.teach4()
  #LGTeacher.init!allocator.1: (LGTeacher.Type) -> () -> LGTeacher : @main.LGTeacher.__allocating_init() -> main.LGTeacher
  #LGTeacher.deinit!deallocator.1: @main.LGTeacher.__deallocating_deinit  // LGTeacher.__deallocating_deinit
}
```

首先是 `sil_vtable` 的关键字，然后是 `LGTeacher` 表明当前是 `LGTeacher` class 的函数表

其次就是当前方法的声明对应着方法的名称
这张表的本质其实就类似我们理解的数组，声明在 `class` 内部的方法在不加任何关键字修饰的过程中，连续存放在我们当前的地址空间中。

接下来我们通过断点来直观的看一下，首先我们需要明确几个指令：

```
ARM64汇编指令

blr  ; 带返回的跳转指令，跳转到指令后边跟随寄存器中保存的地址

mov: 将某一寄存器的值复制到另一寄存器(只能用于寄存器与寄存器或者寄存器 与常量之间传值，不能用于内存地址)，如:
          mov x1, x0 将寄存器 x0 的值复制到寄存器 x1 中

ldr: 将内存中的值读取到寄存器中，如:
         ldr x0, [x1, x2] 将寄存器 x1 和寄存器 x2 相加作为地址，取该内存地址的值放入寄存器 x0 中

str : 将寄存器中的值写入到内存中，如:
            str x0, [x0, x8] ,  将寄存器 x0的值保存到内存[x0 + x8]处

bl: 跳转到某地址
```

函数声明位置的不同也会导致派发方式的不同。如果我们在类的扩展中声明的函数，这里就是一个直接调用。
🤔 ：为什么会 teach 是直接调用，能不能给出你解释?

```
class LGTeacher{
```

```
    var age: Int = 10
}

extension{
  func teach () {
    print("teach")
  }
}

var t = LGTeacher()
```

final 关键字

```
class LGTeacher{
    var age: Int = 10

  final func teach () {
    print("teach")
  }
}
```

@objc：OC

```
class LGTeacher{
    var age: Int = 10

  //标记暴露给我们的OC
  @objc func teach () {
    print("teach")
  }
}
```

dynamic

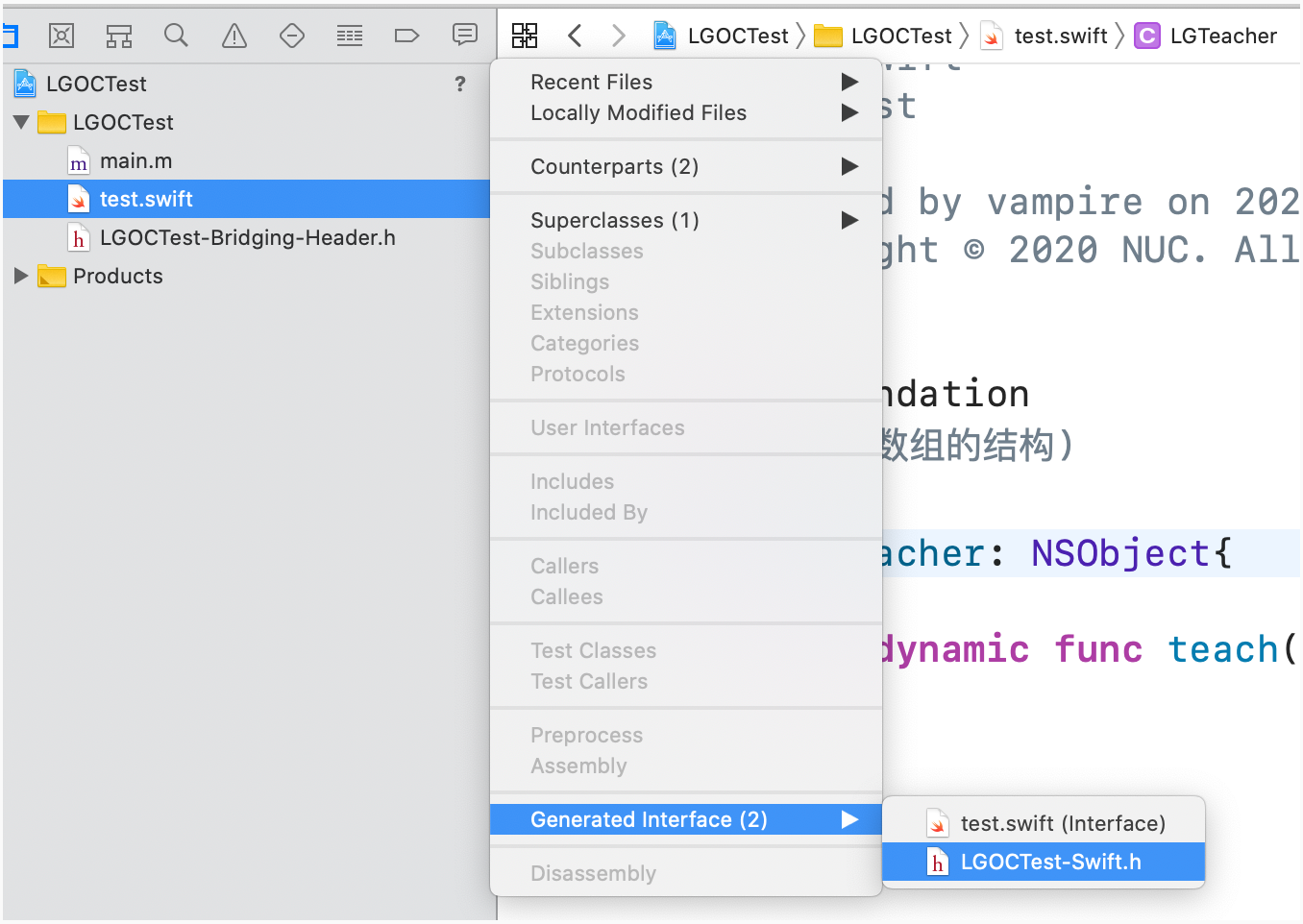
```
class LGTeacher{
    var age: Int = 10

  //动态的特性
  dynamic func teach () {
    print("teach")
  }
}
```

@objc + dynamic

```
class LGTeacher{
    var age: Int = 10

  @objc dynamic func teach () {
    print("teach")
  }
}
```




```swift
class LGTeacher{
    var age: Int = 10

    func teach(){
      print("teach")
  }
}

extension LGTeacher{
//    @_dynamicReplacement(for: teach)
    func teach1()  {
        print("teach1")
    }
}
```

指针

Swift中的指针分为两类，`typed pointer` 指定数据类型指针，`raw pointer` 未指定数据类型的指针(原生指针)

`raw pointer` 在 Swift 中的表示是 `UnsafeRawPointer`

`tyepd pointer` 在 Swift 中的表示是 `UnsafePointer<T>`

我们来看一下 `Swift` 中的指针和 `OC` 中指针的对应关系：

Swift	Object-C	说明
unsafePointer<T>	const T *	指针及所指向的内容都不可变
unsafeMutablePointer	T *	指针及其所指向的内存内容均可变
unsafeRawPointer	const void *	指针指向未知类型
unsafeMutableRawPointer	void *	指针指向未知类型

原始指针的使用

假设我们想在内存中存储连续 4 个整形的数据，我们如何使用 `Raw Pointer` 来做。

```
/**
 RawPionter的使用
 */
//1、分配32字节的内存空间大小
let p = UnsafeMutableRawPointer.allocate(byteCount: 32, alignment: 8)

//2、advanced代表当前 p 前进的步长，对于 RawPointer 来说，我们需要移动的是当前存储值得内存大小即，MemoryLayout.stride
//3、storeBytes: 这里就是存储我们当前的数据，这里需要制定我们当前数据的类型
for i in 0..<4{
    p.advanced(by: i * 8).storeBytes(of: i + 1, as: Int.self)
}

```

```
//4、load顾明思义是加载，fromBytesOffe：是相对于我们当前 p 的首地址的偏移
for i in 0..<4{
    let value = p.load(fromByteOffset: i * 8, as: Int.self)
    print("index\(i),value:\(value)")
}

p.deallocate()
```

Type Pointer

```
//e.g:如何获取 age 变量的地址
var age = 10

//1、通过Swift提供的简写的API，这里注意当前尾随闭包的写法
let p = withUnsafePointer(to: &age){$0}
print(p.pointee)
```

```
//2、withUnsafePointer的返回值是 unSafePointer，意味着我们不能直接修改值，此刻b的值就是 22
var b = withUnsafePointer(to: &age){ prt in
    prt.pointee += 12
}
```

```
//3、如果我们想要直接修改当前Pointer.pointee的值，那么使用withUnsafeMutablePointer
withUnsafeMutablePointer(to: &age){ptr  in
    ptr.pointee += 10
}
```

```
/**
 另一种创建Type Pointer的方式
 */
var age = 10

```

```swift
//1、capacity:容量大小，当前的大小为 1 * 8字节
let ptr = UnsafeMutablePointer<Int>.allocate(capacity: 1)

//2、初始化当前的UnsafeMutablePointer<Int> 指针
ptr.initialize(to: age)

//3、下面两个成对调用，管理内存
ptr.deinitialize(count: 1)
ptr.deallocate()
```

案例小练习

```swift
struct HeapObject {
    var kind: UnsafeRawPointer
    var strongref: UInt32
    var unownedRef: UInt32
}

struct lg_swift_class {
    var kind: UnsafeRawPointer
    var superClass: UnsafeRawPointer
    var cachedata1: UnsafeRawPointer
    var cachedata2: UnsafeRawPointer
    var data: UnsafeRawPointer
    var flags: UInt32
    var instanceAddressOffset: UInt32
    var instanceSize: UInt32
    var flinstanceAlignMask: UInt16
    var reserved: UInt16
    var classSize: UInt32
    var classAddressOffset: UInt32
    var description: UnsafeRawPointer
}

class LGTeacher{
    var age = 18
}
```

```

//实例变量的内存地址
var t = LGTeacher()

//Unmanagedpass.Unretained(t as AnyObject).toOpaque()
//OC 和 CF 交互的方式， __brige ，所有权的转换
let ptr = Unmanaged.passUnretained(t as AnyObject).toOpaque()

let hepObject = ptr.bindMemory(to: HeapObject.self, capacity: 1)

let metaPtr = hepObject.pointee.kind.bindMemory(to: lg_swift_class.self, capacity: 1)

print(metaPtr.pointee)
```

```
/**
 使用 assmuingMemoryBound 案例 1： 元组指针类型转换
 */

var tul = (10, 20)

withUnsafePointer(to: &tul) { (tulPtr: UnsafePointer<(Int, Int)>) in
    testPointer(UnsafeRawPointer(tulPtr).assumingMemoryBound(to: Int.self))
}

//UnsafePointer<T>
func testPointer(_ p : UnsafePointer<Int>){
    print(p)
    print("end")
}
```

```
/**
```

```
 使用 assmuingMemoryBound 案例 1:  元组指针类型转换
 */

var tul = (10, 20)

withUnsafePointer(to: &tul) { (tulPtr: UnsafePointer<(Int, Int)>)
in
    testPointer(UnsafeRawPointer(tulPtr).assumingMemoryBound(to:
Int.self))
}

//UnsafePointer<T>
func testPointer(_ p : UnsafePointer<Int>){
    print(p)
    print("end")
}
```

```
/**
 使用 assmuingMemoryBound 案例 2:  如何获取结构体的属性的指针
 */
struct HeapObject {
    var strongref = 10
    var unownedRef = 20
}

var t = HeapObject()
withUnsafePointer(to: &t) { (ptr:UnsafePointer<HeapObject>)  in
    //思考这里是否需要通过withUnsafePointer来获取?
    //1、withUnsafePointer(to: &ptr.pointee.strongref, <#T##body:
(UnsafePointer<T>) throws -> Result##(UnsafePointer<T>) throws ->
Result#>)?
    //2、ptr.advanced(by: <#T##Int#>)?
    //3、是不是通过原生指针 + 偏移量
    let strongRefPtr = UnsafeRawPointer(ptr) + MemoryLayout<HeapO
bject>.offset(of: \HeapObject.strongref)!
    testPointer(strongRefPtr.assumingMemoryBound(to: Int.self))
}
```

```swift
var t = LGTeacher()

let ptr = UnsafeMutablePointer<LGTeacher>.allocate(capacity: 2)

ptr.initialize(to: LGTeacher())

 注意这里的advanced 其实就是当前要移动是i * 类型大小中的i
ptr.advanced(by: 1).initialize(to: <#T##LGTeacher#>)


print(MemoryLayout<LGTeacher>.size)
print(MemoryLayout<LGTeacher>.stride)
print(ptr[0])
print(ptr[1])

print(ptr.pointee)
//print((ptr + 1).pointee)
print(ptr.successor().pointee)



ptr.deinitialize(count: 2)
ptr.deallocate()
```

内存管理

`Swift`中使用自动引用计数(ARC)机制来追踪和管理内存。

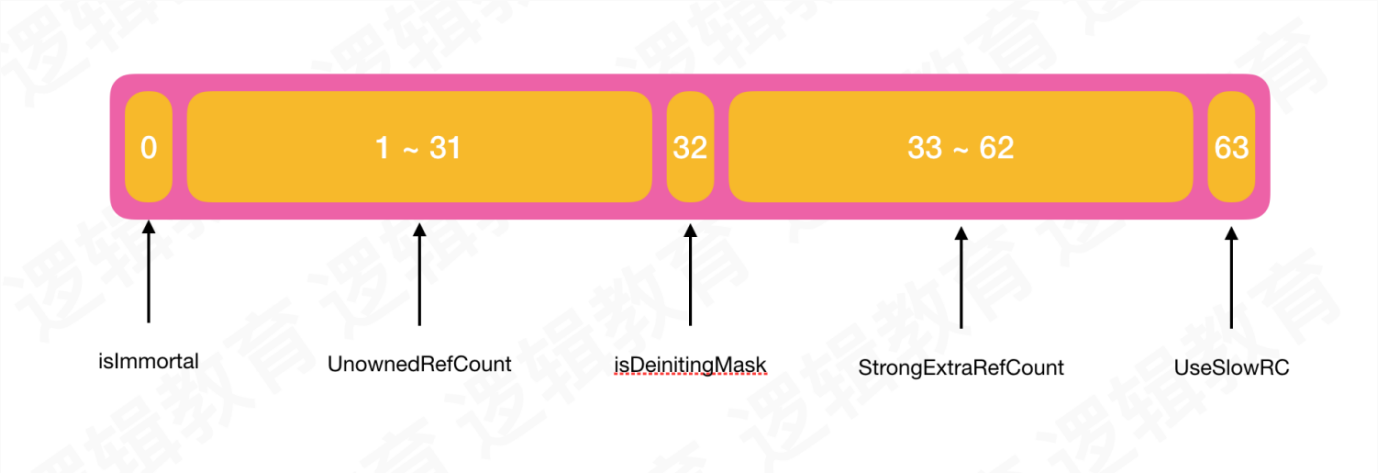
强引用

```swift
//思考一下这里的引用计数是几?
class LGTeacher{
    var age: Int = 18
    var name: String = "Kody"
}

```

```
var t = LGTeacher()
var t1 = t
var t2 = t
```

通过阅读源码可以得到如下 64 位的分配图：



弱引用

```
class LGTeacher{
    var age = 18
}

weak var t = LGTeacher()
```

- 弱引用创建出来的变量是可选的
- 弱引用设置为 nil 的时候不会触发属性观察者

无主引用

```
class LGTeacher{
    var age = 18
}

unowned var t = LGTeacher()
```

- 无主引用和弱引用类似，不会 retain 当前实例对象的引用，非可选，当前 t 被 unowned 标识之后，假定永远有值，非可选类型
- 既然是非可选类型，也就意味着存在访问 crash 风险

🤔：什么时候使用无主引用，什么时候使用 Weak?

闭包的循环引用

```
class LGTeacher{
    var age = 18

    deinit {
        print("LGTeacher deinit")
    }
}

func testARC() {
    let t = LGTeacher()

    let closure = {
        t.age += 1
        print("closure")
    }

    closure()

    print("end")
}

testARC()
```