Swift进阶第四节课：内存管理 & error & 元类型 & Mirror

遗留知识点补充

- withMemoryRebound ： 临时更改内存绑定类型
- bindMemory(to: Capacity:)：更改内存绑定的类型，如果之前没有绑定，那么就是首次绑定；如果绑定过了，会被重新绑定为该类型。
- assumingMemoryBound: 假定内存绑定，这里是告诉编译器：哥们我就是这种类型，你不要检查我了。

Swift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。

```
class LGTeacher{
    var age: Int = 18
    var name: String = "Kody"
}

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

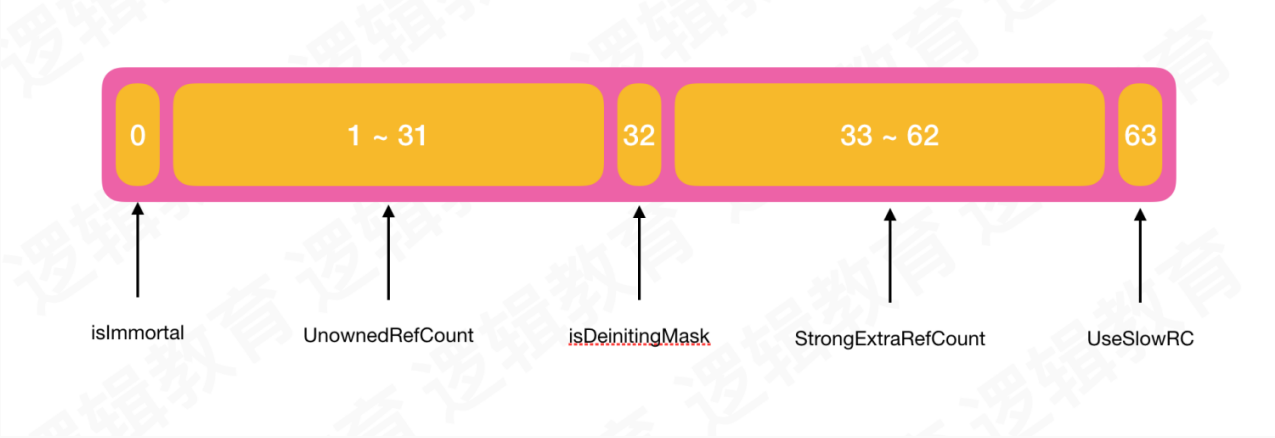
接下来我们通过 `LLDB` 指令来查看当前的引用计数：

```
(lldb) po t
<LGTeacher: 0x1005bdc50>

(lldb) x/8g 0x1005bdc50
0x1005bdc50: 0x0000000100003178 0x0000000200000002
0x1005bdc60: 0x0000000000000012 0x0000000079646f4b
0x1005bdc70: 0xe400000000000000 0x0000000000000000
0x1005bdc80: 0x0000000000000000 0x0000000000000000
(lldb)
```

这里好像和我们平常的不一样，我们来来到源码里面看一下，这个 `refCounted` 到底是什么？

源码分析参考视频内容的讲解~

最终我们得出这样一个结论：



弱引用

```
weak var t = LGTeacher()
```

弱引用声明的变量是一个可选值~因为在程序运行过程中是允许将当前变量设置为 `nil` 的。我们可以借助下面的代码理解一下：

```
class LGTeacher{
    var age: Int = 18
    var name: String = "Kody"
```

```
4
5       deinit {
6           print("LGTeacher deinit")
7       }
8   }
9
10  var t = LGTeacher()
11
12  t = nil  //这句代码会报错
```

所以也就意味着当前的 `weak` 必须是一个可选类型，才能让允许被设置为 `nil`

同样的我们来分析一下 `weak` 关键字在底层到底做了一件什么样的事情，参考视频讲解。

```
1  HeapObject{
2      InlineRefCountBits{ strong count  + unowned count}
3
4    HeapObjectSideTableEntry{
5      HeapObjec *object
6      xxx
7      strong count + unonwned count (uint64_t)
8      weak count (uint32_t)
9    }
10 }
```

循环引用

首先我们需要知道：**闭包一般默认捕获我们外部的变量**，我们来看下面这段代码

```
1  var age = 10
2
3  let closure = {
4      age += 1
5  }
6
```

```
closure() //打印为 11
```

从输出结果来看： **闭包内部对变量的修改将会改变外部原始变量的值**

```
class LGTeacher{
    var age = 18

    deinit{
        print("LGTeacher deinit")
    }
}

var t = LGTeacher()

let closure = {
    t.age += 1
}

closure() //打印为 11
```

如果我们把上面的例子修改一下：

```
class LGTeacher{
    var age = 18

    var complectionBack: (() -> ())?

    deinit{
        print("LGTeacher deinit")
    }
}

func test(){
    var t = LGTeacher()

    let closure = {
```

```
        t.complectionBack = {
            t.age += 1
        }
    }

    closure()
}

test()
```

这里是不是就产生了我们常见的循环引用啊，而且当前 `LGTeacher` 的反初始化器 `deinit` 也没有打印。

那么这里我们如何解决这里的循环引用哪?

```
class LGTeacher{
    var age = 18

    var complateCallBack:(()->())?

    deinit {
        print("LGTeacher deinit")
    }
}


func test() {
    let t = LGTeacher()

  //    t.complateCallBack = { [weak t] in
//        t?.age += 1
//    }

    t.complateCallBack = { [unowned t] in
        t.age += 1
    }

    print("end")
}
```

```
25
26 test()
```

上面的语法方式大家可能比较陌生，在 `Swift` 中叫做 **捕获列表**

定义在参数列表之前，捕获列表被写为用逗号括起来的表达式列表，并用方括号括起来。如果使用捕获列表，则即使省略参数名称，参数类型和返回类型，也必须使用in关键字

```
1 var age = 0
2
3 var height = 0.0
4
5 let closure = { [age] in
6     print(age)
7     print(height)
8 }
9
10 age = 10
11
12 height = 1.85
13
14 closure() //🤔：这里的输出结果是什么?
```

总结：

对于捕获列表中的每个常量，闭包会利用周围范围内具有相同名称的常量或变量，来初始化捕获列表中定义的常量。

Swift Runtime探索

我们用下面这段代码来测试一下：

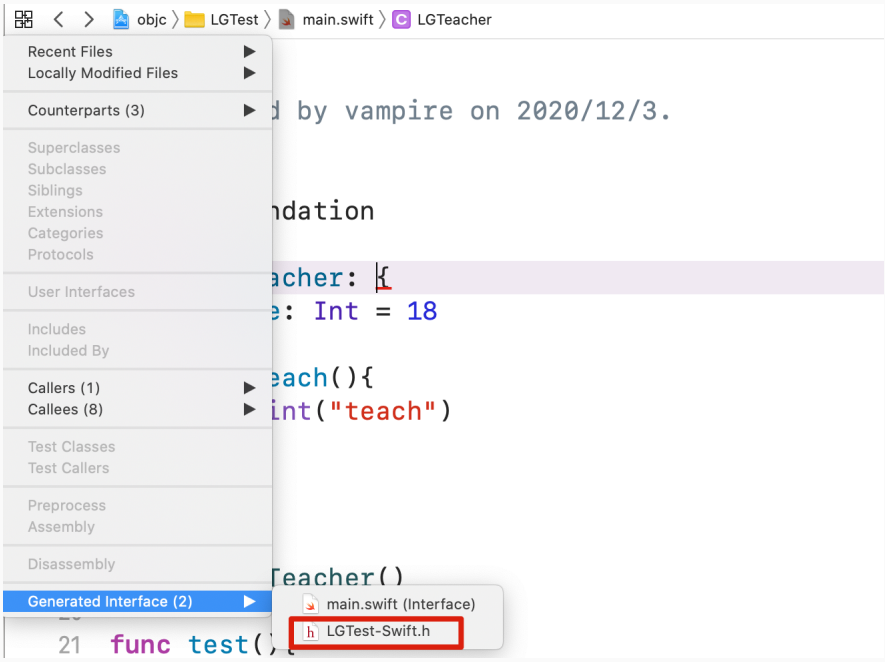
```
1 class LGTeacher{
2     var age: Int = 18
3
```

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

let t = LGTeacher()

func test(){
    var methodCount:UInt32 = 0
    let methodlist = class_copyMethodList(LGTeacher.self, &methodCount)
    for  i in 0..<numericCast(methodCount) {
        if let method = methodlist?[i]{
            let methodName = method_getName(method);
            print("方法列表:\(String(describing: methodName))")
        }else{
            print("not found method");
        }
    }

    var count:UInt32 = 0
    let proList = class_copyPropertyList(LGTeacher.self, &count)
    for  i in 0..<numericCast(count) {
        if let property = proList?[i]{
            let propertyName = property_getName(property);
            print("属性成员属性:\(String(utf8String: propertyName)!)")
        }else{
            print("没有找到你要的属性");
        }
    }
}
```

运行这段代码你会发现，当前不管是我们的方法列表还是我们的属性列表，此次此刻都是为空的。

上节课，我们学过 @objc 的标识，如果这个时候我们将我们当前的方法和属性添加上，会发生什么？

此刻代码会输出我们当前的 teach 方法和 age 属性。但是此刻对于我们的 OC 来说是没有办法使用的：



当前的 LGTest-Swift.h 这个文件里面没有任何关于 LGTeacher 这个信息。

那如果我们让 LGTeacher 继承自 NSObject ，同时去掉 @objc 的标识，重新运行一次上面的代码，发现了什么？当前打印的方法列表和属性列表都是空。

这时我们重新在我们当前的属性和方法前面修饰我们当前的 @objc 关键字，这时我们会发现不管是 Method 和 Property 的打印都符合了我们当前预期结果。

所以这里我们得出来这样一个结论：

- 对于纯 Swift 类来说，没有 动态特性。方法和属性不加任何修饰符的情况下。这个时候其实已经不具备我们所谓的 Runtime 特性了，这和我们在上一节课的方法调度（V-Table调度）是不谋而合的。dynamic（动态特性）
- 对于纯 Swift 类，方法和属性添加 @objc 标识的情况下，当前我们可以通过 Runtime API 拿到，但是在我们的 OC 中是没法进行调度的。
- 对于继承自 NSObject 类来说，如果我们想要动态的获取当前的属性和方法，必须在其声明前添加 @objc 关键字，方法交换： dynamic的标识。否则也是没有办法通过 Runtime API 获取的。

反射

反射就是可以动态获取类型、成员信息，在运行时可以调用方法、属性等行为的特性。上面我们分析过了，对于一个纯 Swift 类来说，并不支持我们直接像 OC 那样操作；但是 Swift 标准库依然提供了反射机制让我们访问成员信息，反射的用法非常简单，我们一起来熟悉一下：

```swift
let mirror = Mirror(reflecting: LGTeacher.self)
for pro in mirror.children{
    print("\(pro.label):\(pro.value)")
}
```

运行这段代码之后，就能在控制台看到打印的效果了。

那这个时候我们能用 Mirror 做些什么事情那？首先想到的应该就是 JSON 解析了：

```swift
func test(_ obj: Any) -> Any{
    let mirror = Mirror(reflecting: obj)

    guard !mirror.children.isEmpty else {return obj}

    var keyValue: [String: Any] = [:]

    for children in mirror.children{
        if let keyName = children.label {
            keyValue[keyName] = test(children.value)
        }else{
            print("children.label 为空")
        }
    }
    return keyValue
}
```

上述代码中我们虽然完成了一个简单的 `JSON` 解析的Demo ，但是很多错误都是输出，如何在 `Swift` 中专业的表达错误那？

错误处理

`Swift`提供`Error`协议来标识当前应用程序发生错误的情况，`Error`的定义如下：

```swift
public protocol Error{
}
```

所以不管是我们的 `struct` 、 `Class` 、 `enum` 我们都可以通过遵循这个协议来表示一个错误。这里我们选择 `enum` 。

```
enum JSONMapError: Error{
    case emptyKey
    case notConformProtocol
}
```

接下来我们的代码里关于 `print` 的输出修改成对应错误的枚举值了~

```
enum JSONMapError: Error{
    case emptyKey
    case notConformProtocol
}

protocol CustomJSONMap{
    func jsosnMap() -> Any
}

extension CustomJSONMap{
    func jsonMap() -> Any{
        let mirror = Mirror(reflecting: self)

        guard !mirror.children.isEmpty else {return self}

        var keyValue: [String: Any] = [:]

        for children in mirror.children{
            if let value = children.value as? CustomJSONMap{
                if let keyName = children.label {
                    keyValue[keyName] = value.jsonMap()
                }else{
                    return JSONMapError.emptyKey
                }
            }else{
                return JSONMapError.notConformProtocol
            }
```

```
        }
        return keyValue
    }
}
```

但是这里我们使用 return 关键字直接接收了一个 `Any` 的结果，如何抛出错误那，正确的方式是使用 `throw` 关键字。

于此同时，编译器会告诉我们当前的我们的 `function` 并没有声明成 `throws` ，所以修改代码之后就能得出这样的结果了:

```
enum JSONMapError: Error{
    case emptyKey
    case notConformProtocol
}

protocol CustomJSONMap{
    func jsosnMap() -> Any
}

extension CustomJSONMap{
    func jsonMap() throws -> Any{
        let mirror = Mirror(reflecting: self)

        guard !mirror.children.isEmpty else {return self}

        var keyValue: [String: Any] = [:]

        for children in mirror.children{
            if let value = children.value as? CustomJSONMap{
                if let keyName = children.label {
                    keyValue[keyName] = value.jsonMap()
                }else{
                    throw JSONMapError.emptyKey
                }
            }else{
                throw JSONMapError.notConformProtocol
            }
        }
```

```
        return keyValue
    }
}
```

这个时候会有一个问题，那就是当前的 `value` 也会默认调用 `jsonMap` 的方法，意味着也会有错误抛出，这里我们先根据编译器的提示，修改代码如下：

```
enum JSONMapError: Error{
    case emptyKey
    case notConformProtocol
}

protocol CustomJSONMap{
    func jsosnMap() throws -> Any
}

extension CustomJSONMap{
    func jsonMap() throws -> Any{
        let mirror = Mirror(reflecting: self)

        guard !mirror.children.isEmpty else {return self}

        var keyValue: [String: Any] = [:]

        for children in mirror.children{
            if let value = children.value as? CustomJSONMap{
                if let keyName = children.label {
                    keyValue[keyName] = try value.jsonMap()
                }else{
                    throw JSONMapError.emptyKey
                }
            }else{
                throw JSONMapError.notConformProtocol
            }
        }
        return keyValue
    }
}
```

到这里我们就完成了一个地道的 `swift` 错误表达方式了。

我们来使用一下我们当前编写完成的代码，会发现编译器要求我们使用 `try` 关键字来处理错误。接下来我们就来说一说 `Swift` 中错误处理的几种方式：

- 使用 `try` 关键字，是最简便的，也是我们最喜欢的：甩锅
  - 使用 `try` 关键字有两个注意点：一个还是 `try?` ，一个是 `try!`
  - `try?`:返回的是一个可选类型，这里的结果就是两类，一类是成功，返回具体的字典值；一类就错误，但是具体哪一类错误我们不关系，统一返回了一个nil
  - `try!` 表示你对这段代码有绝对的自信，这行代码绝对不会发生错误！

- 第二种方式就是使用 `do...catch`

如何你觉得仅仅使用 `Error` 并不能达到你想要详尽表达错误信息的方式，可以使用`LocalError` 协议，定义如下：

```
1  public protocol LocalizedError : Error {
2
3      /// A localized message describing what error occurred.
4      var errorDescription: String? { get }
5
6      /// A localized message describing the reason for the failur
   e.
7      var failureReason: String? { get }
8
9      /// A localized message describing how one might recover from
   the failure.
10     var recoverySuggestion: String? { get }
11
12     /// A localized message providing "help" text if the user req
   uests help.
13     var helpAnchor: String? { get }
14 }
```

`CustomError`有三个默认属性：

- a static errorDomain
- an errorCode integer
- an errorUserInfo dictionary

元类型、AnyClass、Self

我们来看一下下面这段代码的区别：

```
var t = LGTeacher()

//此时代表的就是当前 LGTeacher 的实例对象
var t1: AnyObject = t

//此时代表的就是 LGTeacher 这个类的类型
var t2: AnyObject = LGTeacher.self
```

- AnyObject:代表任意类的 instance，类的类型，仅类遵守的协议。
- Any: 代表任意类型，包括 `funcation` 类型或者 `Optional` 类型
- AnyClass 代表任意实例的类型: AnyObject.Type
- T.self，如果 T是实例对象，返回的就是它本身； T 是类，那么返回的是 `Metadata`
- T.Type：一种类型， T.self 是 T.Type 类型
- type(of:) : 用来获取一个值的动态类型