Swift进阶第三节课：属性

上节课遗留

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)）

如果对象只在声明的文件中可见，可以用 private 或 fileprivate 进行修饰。编译器会对 private 或 fileprivate 对象进行检查，确保没有其他继承关系的情形下，自动打上 final 标记，进而使得对象获得静态派发的特性（fileprivate： 只允许在定义的源文件中访问，private ： 定义的声明中访问）

```
class LGPerson{
    private var sex: Bool

    private func unpdateSex(){
        self.sex = !self.sex
    }

    init(sex innerSex: Bool) {
        self.sex = innerSex
    }
}
```

一、存储属性

存储属性是一个作为特定类和结构体实例一部分的常量或变量。存储属性要么是变量存储属性（由 var 关键字引入）要么是常量存储属性（由 let 关键字引入）。存储属性这里没有什么特别要强调的，因为随处可见

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

比如这里的 `age` 和 `name` 就是我们所说的存储属性，这里我们需要加以区分的是 `let` 和 `var` 两者的区别：从定义上： `let` 用来声明常量，常量的值一旦设置好便不能再被更改；`var` 用来声明变量，变量的值可以在将来设置为不同的值。

这里我们来看几个案例：

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

struct LGStudent{
    let age: Int
    var name: String
}

let t = LGTeacher(age: 18, name: "Hello")
t.age = 20
t.name = "Logic"
t = LGTeacher(age: 30, name: "Kody")

var t1 = LGTeacher(age: 18, name: "Hello")
t.age = 20
t.name = "Logic"
t = LGTeacher(age: 30, name: "Kody")

let s = LGStudent()
s.age = 25
s.name = "Doman"
s = LGStudent()

var s1 = LGStudent()
s.age = 25
s.name = "Doman"
s = LGStudent()
```

let 和 var 的区别：

- 从汇编的角度
- 从 SIL的角度

二、计算属性

存储的属性是最常见的，除了存储属性，类、结构体和枚举也能够定义计算属性，计算属性并不存储值，他们提供 `getter` 和 `setter` 来修改和获取值。对于存储属性来说可以是常量或变

量，但计算属性必须定义为变量。于此同时我们书写计算属性时候必须包含类型，因为编译器需要知道期望返回值是什么。

```
struct square{
    var width: Double

    var area: Double{
        get{
          return width * hegith
        }
        set{
            self.width = newValue
        }
    }
}
```

三、属性观察者

属性观察者会观察用来观察属性值的变化，一个 `willSet` 当属性将被改变调用，即使这个值与原有的值相同，而 `didSet` 在属性已经改变之后调用。它们的语法类似于 getter 和 setter。

```
 class SubjectName{
    var subjectName: String = ""{
        willSet{
            print("subjectName will set value \(newValue)")
        }
        didSet{
            print("subjectName has been changed \(oldValue)")
        }
    }
}
```

这里我们在使用属性观察器的时候，需要注意的一点是在初始化期间设置属性时不会调用 `willSet` 和 `didSet` 观察者；只有在为完全初始化的实例分配新值时才会调用它们。运行下面这段代码，你会发现当前并不会有任何的输出。

```
class SubjectName{
    var subjectName: String = "[unnamed]"{
        willSet{
            print("subjectName will set value \(newValue)")
        }
        didSet{
            print("subjectName has been changed \(oldValue)")
        }
    }

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

```
    }
}

let s = SubjectName(subjectName: "Swift进阶")
```

上面的属性观察者只是对存储属性起作用，如果我们想对计算属性起作用怎么办？很简单，只需将相关代码添加到属性的 setter。我们先来看这段代码

```
class Square{
    var width: Double

    var area: Double{
        get{
          return width * width
        }
        set{
          self.width = sqrt(newValue)
        }
        willSet{
            print("area will set value \(newValue)")
        }
        didSet{
            print("area has been changed \(oldValue)")
        }
    }

    init(width: Double) {
        self.width = width
    }
}
```

三、延迟存储属性

- 延迟存储属性的初始值在其第一次使用时才进行计算。
- 用关键字 `lazy` 来标识一个延迟存储属性

四、类型属性

- 类型属性其实就是一个全局变量
- 类型属性只会被初始化一次

```objc
#import "LGThread.h"

@implementation LGThread

+ (instancetype)sharedInstance {
    static LGThread *sharedInstance = nil;
    static dispatch_once_t onceToken;

    dispatch_once(&onceToken, ^{
        sharedInstance = [[LGThread alloc] init];
    });
    return sharedInstance;
}

@end
```

之前的版本直接翻译OC 代码可以这么写，现在

Swift已经不允许我们这么做了

```swift
class Subject{
    class var sharedInstance: Subject {
        struct Static {
            static var onceToken: dispatch_once_t = 0
            static var instance: Subject? = nil
        }
        dispatch_once(&Static.onceToken) {
            Static.instance = Subject()
        }
        return Static.instance!
    }

```

五、属性在Mahco文件的位置信息

在第一节课的过程中我们讲到了 `Metadata` 的元数据结构，我们回顾一下

```swift
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
}
```

上一节课讲到方法调度的过程中我们认识了 `typeDescriptor` ，这里面记录了 `V-Table` 的相关信息，接下来我们需要认识一下 `typeDescriptor` 中的 `fieldDescripto`

```swift
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
}
```

`fieldDescriptor` 记录了当前的属性信息，其中 `fieldDescriptor` 在源码中的结构如下：

```
struct FieldDescriptor  {
    MangledTypeName int32
    Superclass      int32
    Kind            uint16
    FieldRecordSize uint16
    NumFields       uint32
    FieldRecords    [FieldRecord]
}
```

其中 `NumFields` 代表当前有多少个属性， `FieldRecords` 记录了每个属性的信息，`FieldRecords` 的结构体如下：

```
struct FieldRecord{
    Flags           uint32
    MangledTypeName int32
    FieldName       int32
}
```