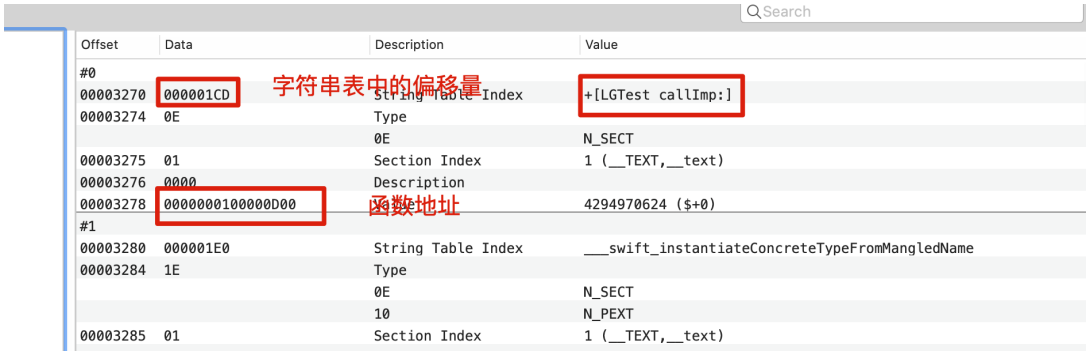
Swift第五节课：Enum & Optional

上节课遗留问题

- VM Address : Virtual Memory Address, 段的虚拟内存地址，在内存中的位置
- VM Size : Virtual Memory Size, 段的虚拟内存大小, 占用多少内存
- File Offset : 段在文件中的偏移量
- File Size : 段在文件中的大小
- Address Space Layout Random ， 地址空间布局随机化，是一种针对缓冲区溢出的安全保护技术，通过对堆、栈、共享库映射等线性区布局的随机化，通过增加攻击者预测目的地址的难度，防止攻击者指针定位攻击代码位置，达到阻止溢出攻击的一种技术



Enum

枚举的基本用法

`swift` 中通过 `enum` 关键字来声明一个枚举

```
enum LGEnum{
    case test_one
    case test_two
    case test_three
}
```

大家都知道在 `C` 或者 `OC` 中默认受整数支持，也就意味着下面的例子中： A， B， C分别默认代表 0， 1，2

```
 typedef NS_ENUM(NSUInteger, LGEnum) {
    A,
    B,
    C,
};
```

Swift 中的枚举则更加灵活，并且不需给枚举中的每一个成员都提供值。如果一个值（所谓“原始”值）要被提供给每一个枚举成员，那么这个值可以是字符串、字符、任意的整数值，或者是浮点类型。

```
 enum Color : String {
    case red = "Red"
    case amber = "Amber"
    case green = "Green"
}

enum LGEnum: Double {
    case a = 10.0
    case b= 20.0
    case c = 30.0
    case d = 40.0
}
```

`隐士 RawValue 分配` 是建立在 `Swift 的类型推断机制上的`

```
 enum DayOfWeek: Int {
    mon, tue, wed, thu, fri = 10, sat, sun
}
```

关联值

```
enum Shape{
    case circle(Double)
    case rectangle(Int, Int)
}
```

模式匹配

```
enum Weak: String
{
    case MONDAY
    case TUEDAY
    case WEDDAY
    case THUDAY
    case FRIDAY
    case SATDAY
    case SUNDAY
}


let currentWeak: Weak

switch currentWeak{
    case .MONDAY: print(Weak.MONDAY.rawValue)
    case .TUEDAY: print(Weak.TUEDAY.rawValue)
```

```
    case .WEDDAY: print(Weak.WEDDAY.rawValue)
    case .THUDAY: print(Weak.THUDAY.rawValue)
    case .FRIDAY: print(Weak.FRIDAY.rawValue)
    case .SUNDAY: print(Weak.SUNDAY.rawValue)
    case .SATDAY: print(Weak.SUNDAY.rawValue)
}
```

如果不想匹配所有的 `case` ，使用 `defalut` 关键字

```
enum Weak: String
{
    case MONDAY
    case TUEDAY
    case WEDDAY
    case THUDAY
    case FRIDAY
    case SATDAY
    case SUNDAY
}


let currentWeak: Weak = Weak.MONDAY

switch currentWeak{
    case .SATDAY, .SUNDAY: print("Happy Day")
    default : print("SAD DAY")
}
```

如果我们要匹配关联值的话

```
enum Shape{
    case circle(radious: Double)
    case rectangle(width: Int, height: Int)
}

let shape = Shape.circle(radious: 10.0)

switch shape{
    case let .circle(radious):
        print("Circle radious:\(radious)")
    case let .rectangle(width, height):
        print("rectangle width:\(width),height\(height)")
}
```

还可以这么写

```
enum Shape{
    case circle(radious: Double)
    case rectangle(width: Int, height: Int)
}
```

```
let shape = Shape.circle(radious: 10.0)

switch shape{
    case .circle(let radious):
        print("Circle radious:\(radious)")
    case .rectangle(let width, var height):
        print("rectangle width:\(width),height\(height)")
}
```

枚举的大小

接下来我们来讨论一下枚举占用的内存大小，这里我们区分几种不同的情况，首先第一种就是 `No-payload enums` 。

```
enum Week{
    case MONDAY
    case TUEDAY
    case WEDDAY
    case THUDAY
    case FRIDAY
    case SATDAY
    case SUNDAY
}
```

大家可以看到这种枚举类型类似我们在 `C` 语言中的枚举，当前类型默认是 `Int` 类型，那么对于这一类的枚举在内存中是如何布局？以及在内存中占用的大小是多少那？这里我们就可以直接使用 `MemoryLayout` 来测量一下当前枚举

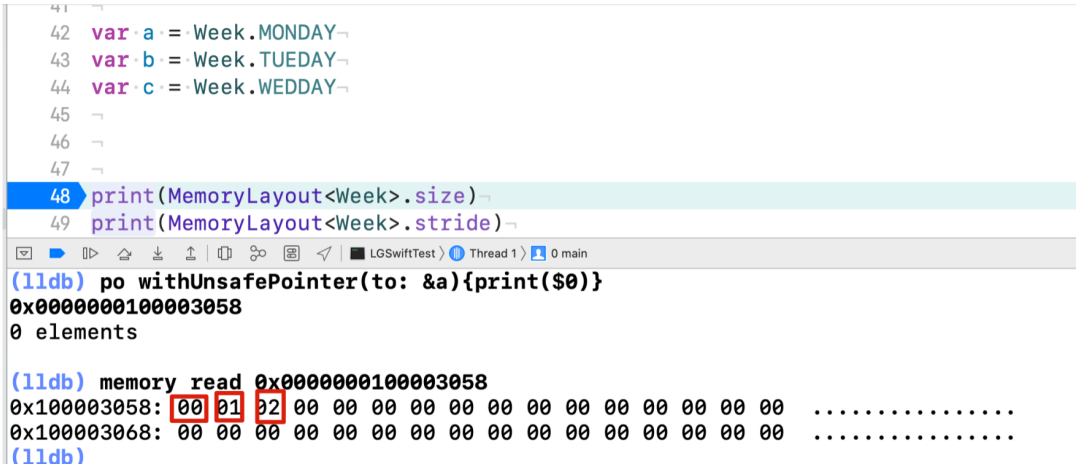
```
enum Week{
    case MONDAY
    case TUEDAY
    case WEDDAY
    case THUDAY
    case FRIDAY
    case SATDAY
    case SUNDAY
}

print(MemoryLayout<Week>.size)
print(MemoryLayout<Week>.stride)
```

```
1
1
```

可以看到这里我们测试出来的不管是 `size` 还是 `stride` 都是 `1` ，这个地方我们也很好理解，当前的 `enum` 有几个 `case` ？是不是 8 个啊，在 `Swift` 中进行枚举布局的时候一直是尝

试使用最少的空间来存储 `enum` ，对于当前的 `case` 数量来说， `UInt8` 能够表示 256 cases ，也就意味着如果一个默认枚举类型且没有关联值的 `case` 少于 `256` ，当前枚举类型的大小都是 1 字节。

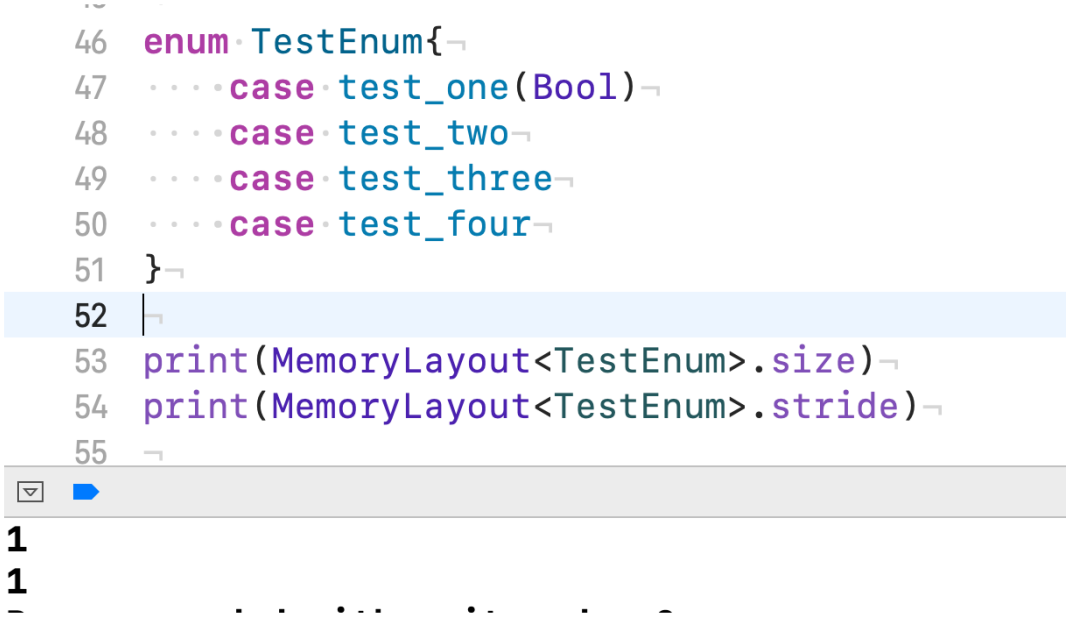


通过上面的打印我们可以直观的看到，当前变量 `a` ， `b` ， `c` 这三个变量存储的内容分别是 00， 01， 02 这和我们上面说的布局理解是一致的。

`No-payload enums` 的布局比较简单，我们也比较好理解，接下来我们来理解一下 `Single-payload enums` 的内存布局， 字面意思就是有一个负载的 `enum` 比如下面这个例子

```
enum LGEnum{
    case test_one(Bool)
    case test_two
    case test_three
    case test_four
}
```

大家猜一下当前的这个案例， `enum` 在内存中的大小是多少?



如果我把当前的案例换一下，换成如下的案例，那么当前 `enum` 占用的大小是多少?

```
enum LGEnum{
    case test_one(Int)
```

```
    case test_two
    case test_three
    case test_four
}
```

```
enum LGEnum{
    case test_one(Bool)
    case test_two
    case test_three
    case test_four
}

//var a = LGEnum.test_one(false)
//var b = LGEnum.test_one(true)
//var c = LGEnum.test_two
//var d = LGEnum.test_three
//var e = LGEnum.test_four

print(MemoryLayout<LGEnum>.size)
print(MemoryLayout<LGEnum>.stride)
```

这里我们就产生了疑问了，为什么都是单个负载，但是当前占用的大小却不一致?

注意， `Swift` 中的 `enum` 中的 `Single-payload enums` 会使用负载类型中的额外空间来记录没有负载的 `case` 值。这句话该怎么理解? 首先 `Bool` 类型是 1字节，也就是 `UInt8` ，所以当前能表达 256 个 case的情况，对于 布尔类型来说，只需要使用低位的 `0, 1` 这两种情况，其他剩余的空间就可以用来表示没有负载的 `case` 值。

```
enum LGEnum{
    case test_one(Bool)
    case test_two
    case test_three
    case test_four
}

var a = LGEnum.test_one(false)
var b = LGEnum.test_one(true)
var c = LGEnum.test_two
var d = LGEnum.test_three
var e = LGEnum.test_four

LGSwiftTest 〉 Thread 1 〉 0 main
(lldb) po withUnsafePointer(to: &a){print($0)}
0x0000000100003028
0 elements

(lldb) memory read 0x0000000100003028
0x100003028: 00 01 02 03 04 00 00 00 00 00 00 00 00 00 00 00  ................
0x100003038: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
(lldb)
```

可以看到，不同的 `case` 值确实是按照我们在开始得出来的那个结论进行布局的。

对于 `Int` 类型的负载来说，其实系统是没有办法推算当前的负载所要使用的位数，也就意味着当前 `Int` 类型的负载是没有额外的剩余空间的，这个时候我们就需要额外开辟内存空间来去存储我们的 `case` 值，也就是 8 + 1 = 9 字节。

```
46 enum LGEnum{
47     case test_one(Int)
48     case test_two
49     case test_three
50     case test_four
51 }
52
53 var a = LGEnum.test_one(10)
54 var b = LGEnum.test_one(20)
55 var c = LGEnum.test_two
56 var d = LGEnum.test_three
57 var e = LGEnum.test_four
58
59
60 print(MemoryLayout<LGEnum>.size)
61 print(MemoryLayout<LGEnum>.stride)
62
```


```
(lldb) po withUnsafePointer(to: &a){print($0)}
0x0000000100002028
0 elements

(lldb) x/8g 0x0000000100002028
0x100002028: 0x000000000000000a 0x0000000000000000
0x100002038: 0x0000000000000014 0x0000000000000000
0x100002048: 0x0000000000000000 0x0000000000000001
0x100002058: 0x0000000000000001 0x0000000000000001
(lldb) memory read 0x0000000100002028
0x100002028: 0a 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
0x100002038: 14 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

上面说完了 `Single-payload enums`，接下来我们说第三种情况 `Mutil-payload enums`，有多个负载的情况产生时，当前的 `enum` 是如何进行布局的那?

```
enum LGEnum{
    case test_one(Bool) //
    case test_two(Bool)
    case test_three
    case test_four
}
```

上面这个例子中，我们有两个 `Bool` 类型的负载，这个时候我们打印当前的 `enum` 大小发现其大小仍然为 1，这个时候我们来看一下内存当中的存储情况

```
70 print(MemoryLayout<LGEnum>.size)
71 print(MemoryLayout<LGEnum>.stride)
72
```

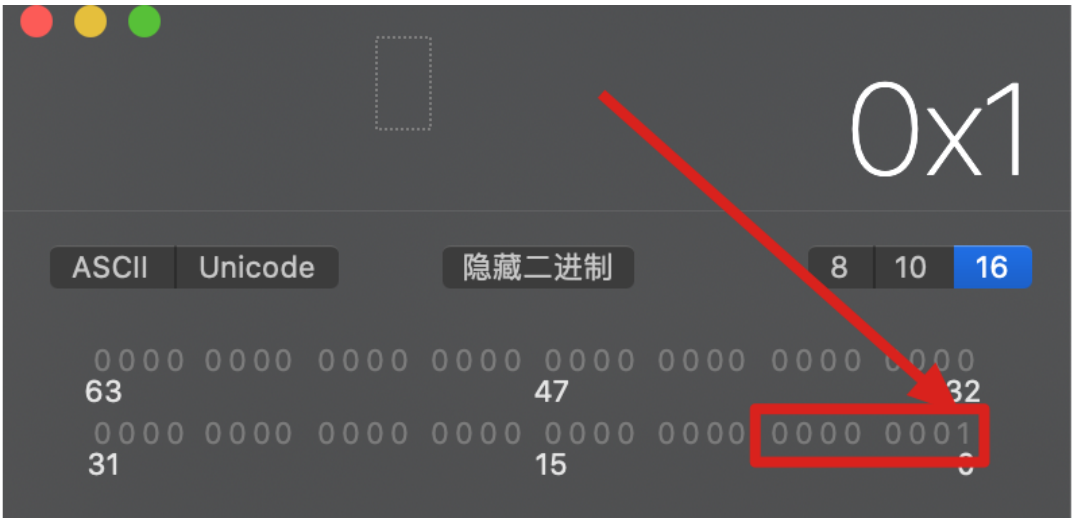
```
1
1
```

```
61 ¬
62 var a = LGEnum.test_one(false)¬
63 var b = LGEnum.test_one(true)¬
64 var c = LGEnum.test_two(false)¬
65 var d = LGEnum.test_two(true)¬
66 var e = LGEnum.test_three¬
67 var f = LGEnum.test_four¬
68 ¬
```

```
LGSwiftTest 〉 Thread 1 〉 0 main
(lldb) po withUnsafePointer(to: &a){print($0)}
0x0000000100003028
0 elements

(lldb) memory read 0x0000000100003028
0x100003028: 00 01 40 41 80 81 00 00 00 00 00 00 00 00 00 00  ..@A............
0x100003038: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
(lldb)
```

这里我们可以看到当前内存存储的分别是 `00, 01, 40, 41, 80, 81` ，这里在存储当前的 `case` 时候会使用到 `common spare bits` ，什么意思？其实在上一个案例我们也讲过了，首先 `bool` 类型需要 1 字节，也就是 8 位。



对于 `bool` 类型来说，我们存储的无非就是 `0` 或 `1` ，只需要用到 1 位，所以剩余的 7 位这里我们都统称为 `common spare bits` ，对于当前的 `case` 数量来说我们完全可以把所有的情况放到 `common spare bits` 中，所以这里我们只需要 1 字节就可以存储所有的内容了。

接下来我们来看一下 `00, 01, 40, 41, 80, 81` 分别代表的是什么？首先 `0, 4, 8` 这里我们叫做 `tag value` ， `0, 1` 这里我们就做 `tag index` ，至于这个 `tag value` 怎么来的，我目前在源码中还没有找到验证，如果感兴趣的通过可以去阅读一下源码中的 `Enum.cpp` 和 `GenEnum.cpp` 这两个文件。

当前一般来说，我们有多个负载的枚举时，当前枚举类型的大小取决于当前最大关联值的大小。我们来看一个例子

```
enum LGEnum{
    case test_one(Bool) //
    case test_two(Int)
    case test_three
    case test_four
}
```

当前 `LGEnum` 的大小就等于 `sizeof(Int) + sizeof(rawVlaue)` = 9，在比如

```
enum LGEnum{
    case test_one(Bool) //
```

```
    case test_two(Int, Int, Int)
    case test_three
    case test_four
}
```

当前大小就是 `sizeof(Int) * 3 + sizeof(rawVlaue)` = 25。

最后这里有几个特殊情况我们需要理解一下，我们来看下面的案例

```
enum LGEnum{
    case test_one
}
```

对于当前的 `LGEnum` 只有一个 `case` ,我们不需要用任何东西来去区分当前的 `case` ,所以当我们打印当前的 `LGEnum` 大小你会发现时 0。

Optional

认识可选值

之前我们在写代码的过程中早就接触过可选值，比如我们在代码这样定义：

```
class LGTeacher{
    var age: Int?
}
```

当前的 `age` 我们就称之为可选值，当然可选值的写法这两者是等同的

```
 var age: Int? = var age: Optional<Int>
```

那对于 `Optional` 的本质是什么？我们直接跳转到源码，打开 `Optional.swift` 文件

```
@frozen
public enum Optional<Wrapped>: ExpressibleByNilLiteral {
  case none
  case some(Wrapped)
}
```

既然 `Optional` 的本质是枚举，那么我们也可以仿照系统的实现制作一个自己的 `Optional`

```
enum MyOptional<Value> {
  case some(Value)
  case none
}
```

比如给定任意一个自然数，如果当前自然数是偶数返回，否则为 nil，我们应该怎么表达这个案例

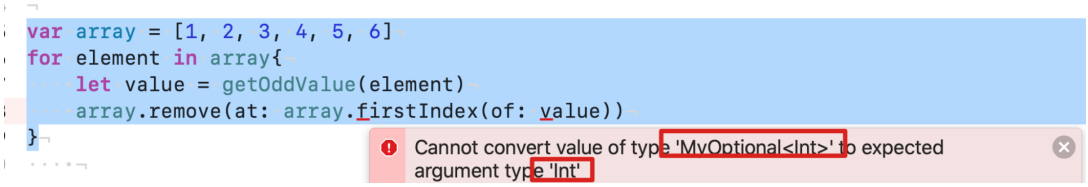
```
enum MyOptional<Value> {
  case some(Value)
  case none
```

```
}


func getOddValue(_ value: Int) -> MyOptional<Int> {
    if value % 2 == 0 {
        return .some(value)
    }
    else{
        return .none
    }
}
```

这个时候给定一个数组，我们想删除数组中所有的偶数

```
var array = [1, 2, 3, 4, 5, 6]
for element in array{
    let value = getOddValue(element)
    array.remove(at: array.firstIndex(of: value))
}
```



这个时候编译器就会检查我们当前的 value 会发现他的类型和系统编译器期望的类型不符，这个时候我们就能使用 `MyOptional` 来限制语法的安全性。

于此同时我们通过 `enum` 的模式匹配来取出对应的值

```
for element in array{
    let value = getOddValue(element)
    switch value {
    case .some(let value):
        array.remove(at: array.firstIndex(of: value)!)
    case .none:
        print("vlaue not exist")
    }
}
```

如果我们把上述的返回值更换一下，其实就和系统的 `Optional` 使用无疑

```
func getOddValue(_ value: Int) -> Int? {
    if value % 2 == 0 {
        return .some(value)
    }
    else{
        return .none
    }
}
```

这样我们其实是利用当前编译器的类型检查来达到语法书写层面的安全性。

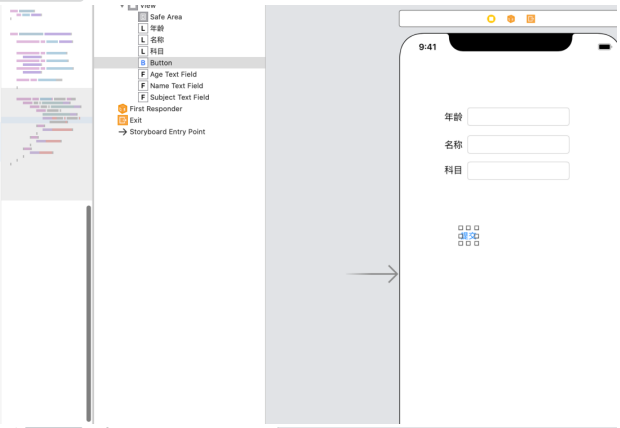
当然如果每一个可选值都用模式匹配的方式来获取值在代码书写上就比较繁琐，我们还可以使用 `if let` 的方式来进行可选值绑定

```
if let value = value{
    array.remove(at: array.firstIndex(of: value)!)
}
```

除了使用 `if let` 来处理可选值之外，我们还可以使用 `gurad let` 来简化我们的代码，我们来看一个具体的案例

`guard let` 和 `if let` 刚好相反，
`guard let` 守护一定有值。如果没有，直接返回。
通常判断是否有值之后，会做具体的逻辑实现，通常代码多
如果用 `if let` 凭空多了一层分支， `guard let` 是降低分支层次的办法

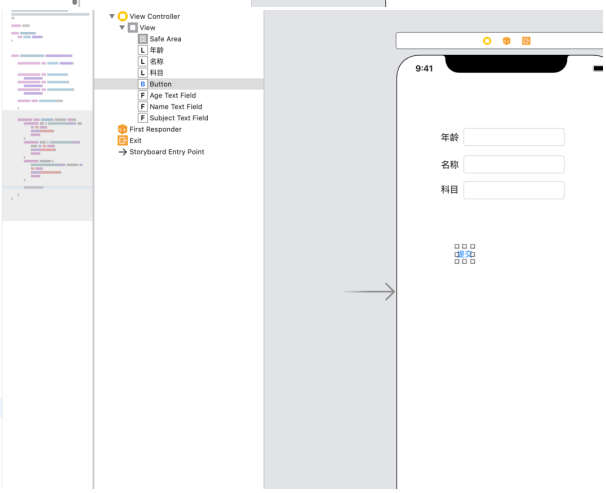
```
@IBAction func submit(_ sender: Any) {
    if let age = ageTextField.text{
        if let name = nameTextField.text{
            if let subject =
                subjectTextField.text{
                print("\(age) + \(name) +
                    \(subject)")
            }else{
                print("subject 为空")
            }
        }else{
            print("name 为空")
        }
    }else{
        print("age 为空")
    }
}
```




```
@IBAction func submit(_ sender: Any) {
    guard let age = ageTextField.text, age
        != "" else{
        print("age 为空")
        return
    }
    guard let name = nameTextField.text,
        name != "" else{
        print("name 为空")
        return
    }
    guard let subject =
        subjectTextField.text, subject !=
        "" else{
        print("subject 为空")
        return
    }

    //to do else

}
}
```



可选链

我们都知道再 `OC` 中我们给一个 `nil` 对象发送消息什么也不会发生， `Swift` 中我们是没有办法向一个 `nil` 对象直接发送消息，但是借助可选链可以达到类似的效果。我们看下面两段代码

```
let str: String? = "abc"
let upperStr = str?.uppercased() // Optional<"ABC">

var str: String?
let upperStr = str?.uppercased() // nil
```

我们再来看下面这段代码输出什么

```
let str: String? = "kody"
let upperStr = str?.uppercased().lowercased()
```

同样的可选链对于下标和函数调用也适用

```
var closure: ((Int) -> ())?
closure?(1) // closure 为 nil 不执行

let dict = ["one": 1, "two": 2]
dict?["one"] // Optional(1)
dict?["three"] // nil
```

?? 运算符 （空合并运算符）

(`a ?? b`) 将对可选类型 `a` 进行空判断，如果 `a` 包含一个值就进行解包，否则就返回一个默认值 `b` .

- 表达式 `a` 必须是 `Optional` 类型
- 默认值 `b` 的类型必须要和 `a` 存储值的类型保持一致

运算符重载

在源码中我们可以看到除了重载了 `??` 运算符， `Optional` 类型还重载了 `==` ， `?=` 等等运算符，实际开发中我们可以通过重载运算符简化我们的表达式。

比如在开发中我们定义了一个二维向量，这个时候我们想对两个向量进行基本的操作，那么我们就可以通过重载运算符来达到我们的目的

```
struct Vector {
  let x: Int
  let y: Int
}

extension Vector {
  static func + (fistVector: Vector, secondVector: Vector) -> Vector {
    return Vector(x:  fistVector.x + secondVector.x, y: fistVector.y + secor
  }

  static prefix func - (vector: Vector) -> Vector {
    return Vector(x: -vector.x, y: -vector.y)
  }

  static func - (fistVector: Vector, secondVector: Vector) -> Vector {
```

```
    return fistVector + -secondVector
  }
}
```

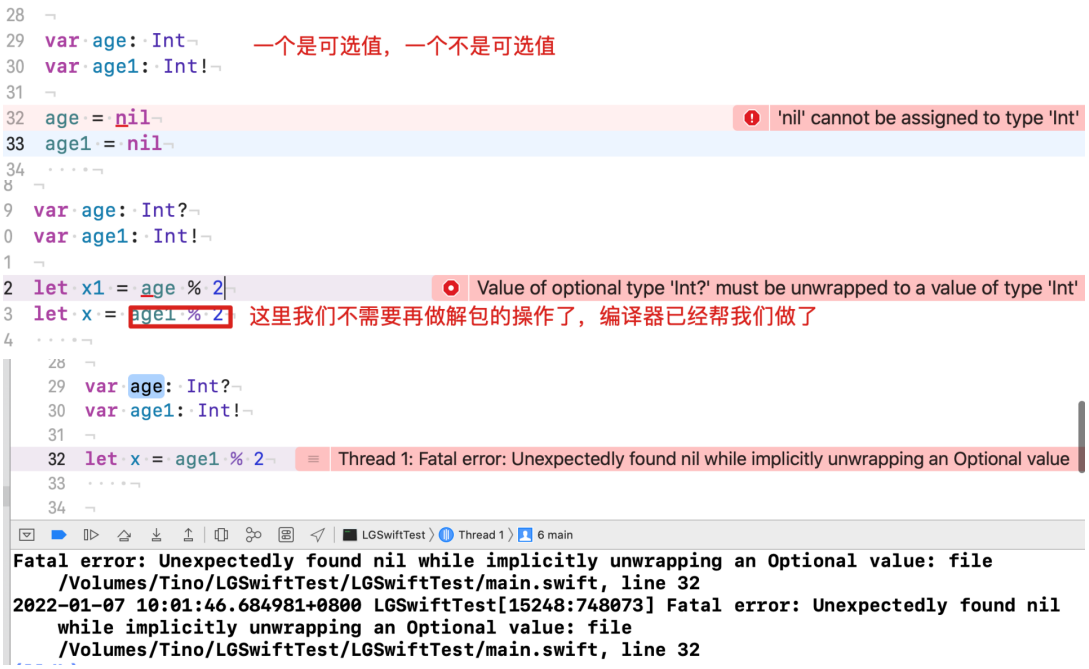
自定义运算符

隐士解析可选类型

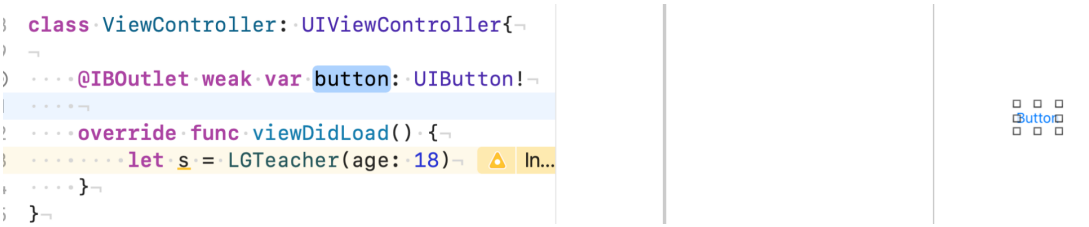
隐式解析可选类型是可选类型的一种，使用的过程中和非可选类型无异。它们之间唯一的区别是，隐式解析可选类型是你告诉对 Swift 编译器，我在运行时访问时，值不会为 nil。

```
var age: Int?
var age1: Int!

age = nil
age1 = nil
```



其实日常开发中我们比较常见这种隐士解析可选类型



IBOutlet类型是Xcode强制为可选类型的，因为它不是在初始化时赋值的，而是在加载视图的时候。你可以把它设置为普通可选类型，但是如果这个视图加载正确，它是不会为空的。

与可选值有关的高阶函数

- map： 这个方法接受一个闭包，如果可选值有内容则调用这个闭包进行转换

```
var dict = ["one": "1", "two": "2"]
let result = dict["one"].map{ Int($0) }
// Optional(Optional(1))
```

上面的代码中我们从字典中取出字符串”1”，并将其转换为Int类型，但因为String转换成Int不一定能成功，所以返回的是Int?类型，而且字典通过键不一定能取得到值，所以map返回的也是一个Optional，所以最后上述代码result的类型为Int??类型。

那么如何把我们的双重可选展平开来，这个时候我们就需要使用到

- flatMap: 可以把结果展平成为单个可选值

```
var dict = ["one": "1", "two": "2"]
let result = dict["one"].flatMap{ Int($0) }
// Optional(1)
```

`

- 注意，这个方法是作用在OptioanI的方法，而不是作用在Sequence上的
- 作用在Sequence上的flatMap方法在Swift4.1中被更名为compactMap，该方法可以将序列中的nil过滤出去

```
let array = ["1", "2", "3", nil]
let result = array.compactMap{ $0 } // ["1", "2", "3"]

let array = ["1", "2", "3", "four"]
let result = array.compactMap{ Int($0) } // [1, 2, 3]
```

元类型、AnyClass、Self （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