

作者 大石头布 (/u/d61ef5da631a)

2016.10.20 20:53 字数 1570 阅读 392 评论 2 喜欢 16

(/u/d61ef5da631a)



swift

Swift 3 出来也有一阵子了，也对公司的项目做了升级，但是暂时还没有赶放上线，依旧用着2.3。相较于Swift 2.2，Swift 3做了很大的改动，逐渐脱离OC的影子。语法上很多对象去掉了NS开头，去掉了繁琐的命名。如 `UIColor.redColor()` 改为 `UIColor.red`，变成了属性，还有方法的第一个参数如果不指定 `_` 调用的时候也要写参数名等等...

本文主要讨论Swift 3中的一些坑和使用过程中的一些小技巧，排名无理由~~

@discardableResult 消除返回值警告

在Swift 3中，如果方法的返回值没有处理xCode会报一个警告，如果在方法前加上 `@discardableResult` 不处理的时候就不会有警告了。也可以用 `_ = xxx()` 来消除警告。

浮点数取余数和除法

在Swift 3中，如果你声明一个 `let m = 12.0` 默认m是 `Double`，`Double` 是不能和 `Float` 做运算的。`CGFloat` 在32位设备上 是 `Float32` 在64位设备上 是 `Float64`，所以如果一个 `Double` 和一个 `Float` 做运算时先要转换类型的

```
let m = 12.0
let n:CGFloat = 19.0

let x = CGFloat(m)/n
let k = m.multiplied(by: Double(n)) // 乘法
let y = m.divided(by: Double(n))    // 除法
```

但是取余算法是不能作用于浮点型的，如果这样就会报错 `CGFloat(m)%n`
正确的做法是：

```
let z = CGFloat(m).truncatingRemainder(dividingBy: n) //取余 12.0
```

这个改动导致项目很多地方都要随着改，而且大多数库也做了改变，如Alamofire的参数从 [String:AnyObject]? 变成 [String:Any]?

值得一提的是 Any 不可以代表任何可空类型，不用指定 Any?

栗子：

```
let str:String? = "xwwa"
var param:[String:Any] = ["x":1,"code":str]
// ["code": Optional("xwwa"), "x": 1]
```

str 是一个 Optional 类型的，输出出来也是 Optional。因为我们以前的请求是需要在 header中带参数的json机密，换成 Any 怎么都过不去，后来发现有 Optional 值。

这里写了个方法转化了下

```
func absArray(param:[String:Any])->[String:Any]{
    let res = param.map { (key,value) -> (String,Any?) in
        let newValue = Mirror(reflecting: value)
        if newValue.displayStyle == Mirror.DisplayStyle.optional{
            if let v = newValue.children.first?.value{
                return (key,v)
            }else{
                return (key,nil)
            }
        }
        return (key,value)
    }
    var newParam:[String:Any] = [:]
    res.forEach { (key,v) in
        newParam[key] = v
    }
    return newParam
}
print(absArray(param:param))    // ["code": "xwwa", "x": 1]
```

用了反射判断如果值是optional就取出他实际的值。

Swift 3中 Notification 使用方法

```
extension Notification.Name {
    static let kNoticeDemo = Notification.Name("xx.xx.ww.ss")
}

class DE{
    func test(){
        NotificationCenter.default.post(name: Notification.Name.kNoticeDemo , object: nil)
        NotificationCenter.default.addObserver(self, selector: #selector(demo), name: Notification.Name.kNoticeDemo, object: nil)

        NotificationCenter.default.removeObserver(self, name: Notification.Name.kNoticeDemo, object: nil)
    }
    @objc func demo(){

    }
}
```

```
precedence 130  
}
```

现在在Swift 3中这样的话会报警告 `Operator should no longer be declared with body;use a precedence group instead`

```
// 自定义操作符 别名类型  
infix operator >>> : ATPrecedence  
precedencegroup ATPrecedence {  
    associativity: left  
    higherThan: AdditionPrecedence  
    lowerThan: MultiplicationPrecedence  
}
```

直接指定操作符的类型，对这个类型进行定义， `associativity: left` 表示左结合
`higherThan` 优先级高于 `AdditionPrecedence` 这个是加法的类型
`lowerThan` 优先级低于 `MultiplicationPrecedence` 乘除

这里给出常用类型对应的group

- `infix operator || : LogicalDisjunctionPrecedence`
- `infix operator && : LogicalConjunctionPrecedence`
- `infix operator < : ComparisonPrecedence`
- `infix operator <= : ComparisonPrecedence`
- `infix operator > : ComparisonPrecedence`
- `infix operator >= : ComparisonPrecedence`
- `infix operator == : ComparisonPrecedence`
- `infix operator != : ComparisonPrecedence`
- `infix operator === : ComparisonPrecedence`
- `infix operator !== : ComparisonPrecedence`
- `infix operator ~= : ComparisonPrecedence`
- `infix operator ?? : NilCoalescingPrecedence`
- `infix operator + : AdditionPrecedence`
- `infix operator - : AdditionPrecedence`
- `infix operator &+ : AdditionPrecedence`
- `infix operator &- : AdditionPrecedence`
- `infix operator | : AdditionPrecedence`
- `infix operator ^ : AdditionPrecedence`
- `infix operator * : MultiplicationPrecedence`
- `infix operator / : MultiplicationPrecedence`

- infix operator >> : BitwiseShiftPrecedence
 - infix operator ..< : RangeFormationPrecedence
 - infix operator ... : RangeFormationPrecedence
 - infix operator *= : AssignmentPrecedence
 - infix operator /= : AssignmentPrecedence
 - infix operator %= : AssignmentPrecedence
 - infix operator += : AssignmentPrecedence
 - infix operator -= : AssignmentPrecedence
 - infix operator <=: AssignmentPrecedence
 - infix operator >>= : AssignmentPrecedence
 - infix operator &= : AssignmentPrecedence
 - infix operator ^= : AssignmentPrecedence
 - infix operator |= : AssignmentPrecedence
-

合理的使用异常处理，提高代码质量

在日常开发中，可能遇到很多特殊情况，使得程序不能继续执行下去。有的来自系统语法方面，有的是来自业务方面的。这时候可以使用自定义异常，在底层代码中不断throw在最后一层中去处理。

```
struct ZError : Error {
    let domain: String
    let code: Int
}

func canThrow() throws{
    let age = 10
    if a < 18{
        let error = ZError(domain: "xxx", code: 990)
        throw error
    }
}

do {
    try canThrow()
} catch let error as ZError {
    print("Error: \(error.code) - \(error.domain)") // Error: 990 - xxx
}
```

是时候放弃前缀的扩展了

以前我们要给 UIView 扩展是这样的

```
}  
}  
}
```

这样在自己写的属性前面加一个前缀。但是Swift 3出来后更多的选择应该是这样的
view.zz.height 。 以前 kingfisher 是 imageView.kf_setImage
现在变成 imageView.kf.setImage 。 SnapKit 也改变成了 make.snp.left 之类的语法

那么怎么写这样的扩展呢？

这里看了 KingFisher 的代码，给出他的解决方案。比如我们想写一个UIView的扩展。

```
// 写一个协议 定义一个只读的类型  
public protocol UIViewCompatible {  
    associatedtype CompatibleType  
    var zz: CompatibleType { get }  
}  
  
public extension UIViewCompatible {  
    // 指定泛型类型为自身，自身是协议 谁实现了此协议就是谁了  
    public var zz: Auto<Self> {  
        get { return Auto(self) } // 初始化 传入自己  
        set { }  
    }  
}  
  
// Auto是一个接受一个泛型类型的结构体  
public struct Auto<Base> {  
    // 定义该泛型类型属性  
    public let base: Base  
    public init(_ base: Base) {  
        self.base = base  
    }  
}  
  
// 写一个Auto的扩展 指定泛型类型是UIView 或者其子类  
public extension Auto where Base:UIView {  
  
    var height:CGFloat{  
        set(v){  
            self.base.frame.size.height = v  
        }  
        get{  
            return self.base.frame.size.height  
        }  
    }  
}  
  
// 扩展 UIView 实现 UIViewCompatible 协议，就拥有了zz属性 zz又是Auto类型 Auto是用泛型实例  
extension UIView : UIViewCompatible{  
  
}  
  
// 使用  
  
view.zz.height
```

上面的注释已经尽量详细的解释了这段代码，hope you can understand !

GCD 的改变

```
DispatchQueue.main.async {
    print("这里在主线程执行")
}
```

优先级

- DISPATCH_QUEUE_PRIORITY_HIGH: .userInitiated
- DISPATCH_QUEUE_PRIORITY_DEFAULT: .default
- DISPATCH_QUEUE_PRIORITY_LOW: .utility
- DISPATCH_QUEUE_PRIORITY_BACKGROUND: .background

```
//global 中设置优先级 不设置默认是 default
DispatchQueue.global(qos: .userInitiated).async {
    print("设置优先级")
}
```

创建一个队列

```
let queue = DispatchQueue(label: "im.demo.test")
```

也可以指定优先级和队列

```
let highQueue = DispatchQueue(label: "high.demo.test.queue", qos: DispatchQoS.background)

highQueue.async {
    print("ceshi")
}
```

3秒后执行

```
DispatchQueue.main.asyncAfter(deadline: DispatchTime.now() + 3.0) {
    print("after")
}
```

根据View查找VC

如果你在一个 UITableViewCell 或者 cell上自定义的一个view上想使用这个view所在的vc怎么办？代理？层层引用？NO！一个扩展解决。
一个UIView的扩展

```
    }
    next = next?.superview
  }
  return (responder as! UIViewController)
}
```

记得写在扩展中哦，加上前面的技巧。不论你在哪个view中。只需要这样 `let vc = view.zz.responderViewController()` 就能拿到所处的vc了。

View中的第一响应者


又是一个View的扩展也很好用，上代码

```
func findFirstResponder()->UIView?{
    if self.isFirstResponder{
        return self
    }
    for subView in self.subviews{
        let view = subView.findFirstResponder()
        if view != nil {
            return view
        }
    }
    return nil
}
```

用法同上，这个东西能干啥呢？

利用这个可以在 `NSNotification.Name.UIKeyboardWillShow` 通知通知中拿到第一响应者，如果第一响应者是UITextField，可以算出局底下的距离，给挡墙view做变换。避免被覆盖。而且这个东西可以写在协议的默认实现中或者写在一个基类中公用。本文为杂谈不细说了，点到为止咯！

暂时写这些吧，后面有时间再补。。。

 iOS 技术总结 (/nb/1955256)

© 著作权归作者所有



更多分享

(<http://cwb.assets.jianshu.io/note>)

