# 类型系统

- 值类型 value type
  - 基础数据类型
  - 结构 struct
  - 枚举 enum
  - 元组 tuple
  - 特殊类型
    - String
    - Array
    - Dictionary
    - Set
- 引用类型 refrence type
  - 类 class
  - 闭包 closure

- 类型服饰
  - 协议 protocol
  - 扩展 extentsion
  - 泛型 generics

# 1. OC与Swift的异同

- swift和OC的共同点：
  1. OC出现过的绝大多数概念，比如引用计数、ARC（自动引用计数）、属性、协议、接口、初始化、扩展类、命名参数、匿名函数等，在Swift中继续有效（可能最多换个术语）。
  2. Swift和Objective-C共用一套运行时环境，Swift的类型可以桥接到Objective-C（下面我简称OC），反之亦然

- swift的优点：
  1. swift注重安全，OC注重灵活
  2. swift注重面向协议编程、函数式编程、面向对象编程，OC注重面向对象编程
  3. swift注重值类型，OC注重指针和引用
  4. swift是静态类型语言，OC是动态类型语言
  5. swift容易阅读，文件结构和大部分语法简易化，只有.swift文件，结尾不需要分号

6. swift中的可选类型，是用于所有数据类型，而不仅仅局限于类。相比于OC中的nil更加安全和简明
7. swift中的泛型类型更加方便和通用，而非OC中只能为集合类型添加泛型
8. swift中各种方便快捷的高阶函数（函数式编程） (Swift的标准数组支持三个高阶函数：map，filter和reduce,以及map的扩展flatMap)
9. swift新增了两种权限，细化权限。open > public > internal(默认) > fileprivate > private
10. swift中独有的元组类型(tuples)，把多个值组合成复合值。元组内的值可以是任何类型，并不要求是相同类型的。
11. swift中函数可以当作参数或返回值，OC中不可以。

来一次有侧重点的区分Swift与Objective-C

# 2. 类(class) 和 结构体(struct) 有什么区别? 类(class) 和 结构体(struct) 比较,优缺点?

二者的本质区别：

- struct是深拷贝，拷贝的是内容；
- class是浅拷贝，拷贝的是指针。
- struct是值类型，class是引用类型。

（值类型的变量直接包含它们的数据，对于值类型都有它们自己的数据副本，因此对一个变量操作不可能影响另一个变量。 引用类型的变量存储对他们的数据引用，因此后者称为对象，因此对一个变量操作可能影响另一个变量所引用的对象。）

相同点：

1. 都可以定义以下成员：属性、方法、下标、初始化器
2. 都支持类型扩展、协议

不同点：

1. 类支持继承和多态，结构体不支持
2. 类必须自己定义初始化器，结构体会有默认的按成员初始化器
3. 类支持析构器（deinit），结构体不支持
4. 类的实例在堆上，由ARC负责释放；结构体的实例在栈上，栈结束自动释放，不参与ARC管理
5. 变量赋值方式不同：struct是值拷贝；class是引用拷贝
6. immutable变量：swift的可变内容和不可变内容用var和let来甄别，如果初始为let的变量再去修改会发生编译错误。struct遵循这一特性；class不存在这样的问题
7. mutating function： struct 和 class 的差别是 struct 的 function 要去改变 property 的值的时候要加上 mutating，而 class 不用。
8. 类支持引用相等比较（===于!==）,结构体不支持

枚举、结构体和类

# 3. swift 中的枚举,关联值 和 原始值的区分?

1. 关联值--有时会将枚举的成员值跟其他类型的变量关联存储在一起，会非常有用

```
1. // 关联值
   enum Date {
     case digit(year: Int, month: Int, day: Int)
     case string(String)
   }
```

2. 原始值--枚举成员可以使用相同类型的默认值预先关联，这个默认值叫做:原始值

```
// 原始值
enum Grade: String {
  case perfect = "A"
  case great = "B"
  case good = "C"
  case bad = "D"
}
```

# 4. swift中, 存储属性和计算属性的区别?

Swift中跟实例对象相关的属性可以分为2大类

1. 存储属性(Stored Property):
   - 类似于成员变量这个概念
   - 存储在实例对象的内存中
   - 结构体、类可以定义存储属性
   - 枚举不可以定义存储属性
2. 计算属性(Computed Property):
   - 本质就是方法(函数)
   - 不占用实例对象的内存

- 枚举、结构体、类都可以定义计算属性

```
struct Circle {
    // 存储属性
    var radius: Double
    // 计算属性
    var diameter: Double {
        set {
            radius = newValue / 2
        }
        get {
            return radius * 2
        }
    }
}
```

# 5. 什么是延迟存储属性(Lazy Stored Property)?

使用lazy可以定义一个延迟存储属性，在第一次用到属性的时候才会进行初始化(类似OC中的懒加载) lazy属性必须是var，不能是let let必须在实例对象的初始化方法完成之前就拥有值 如果多条线程同时第一次访问lazy属性 无法保证属性只被初始化1次

```
class PhotoView { // 延迟存储属性
  lazy var image: Image = {
    let url = "https://...x.png"
    let data = Data(url: url)
    return Image(data: data) }()
}
```

# 6. 运算符重载(Operator Overload)?

类、结构体、枚举可以为现有的运算符提供自定义的实现，这个操作叫做:运算符重载

```
struct Point {
    var x: Int
    var y: Int

    // 重载运算符
    static func + (p1: Point, p2: Point) -> Point   {
        return Point(x: p1.x + p2.x, y: p1.y + p2.y)
    }
}

var p1 = Point(x: 10, y: 10)
var p2 = Point(x: 20, y: 20)
var p3 = p1 + p2
```

# 7. 什么是可选型(Optional)，Optional（可选型） 是用什么实现的

1. 在 Swift 中,可选型是为了表达一个变量为空的情况,当一个变量为空,他的值就是 nil 在类型名称后面加个问号? 来定义一个可选型 值类型或者引用类型都可以是可选型变量

2. Optional 是一个泛型枚举 大致定义如下:

```
enum Optional<Wrapped> {
  case none
  case some(Wrapped)
}
```

除了使用 let someValue: Int? = nil 之外, 还可以使用let optional1: Optional = nil 来定义

# 8. 定义静态方法时关键字 static 和 class 有什么区别

static 定义的方法不可以被子类继承, class 则可以

```
class AnotherClass {
    static func staticMethod(){}
    class func classMethod(){}
}
class ChildOfAnotherClass: AnotherClass {
    override class func classMethod(){}
    //override static func staticMethod(){}// error
```

# 9. swift中,关键字 guard 和 defer 的用法

- guard也是基于一个表达式的布尔值去判断一段代码是否该被执行。与if语句不同的是，guard只有在条件不满足的时候才会执行这段代码。`guard let name = self.text else { return }
- defer的用法是，这条语句并不会马上执行，而是被推入栈中，直到函数结束时才再次被调用。`defer { //函数结束才调用 }`

# 10. swift 中的下标是什么?

使用subscript可以给任意类型(枚举、结构体、类)增加下标功能，有些地方也翻译为:下标脚本 subscript的语法类似于实例方法、计算属性，本质就是方法(函数)

```
class Point {
    var x = 0.0, y = 0.0
    subscript(index: Int) -> Double {
        set {
            if index == 0 {
                x = newValue
            } else if index == 1 {
                y = newValue }
        }
        get {
            if index == 0 {
                return x
            } else if index == 1 {
                return y
            }
            return 0
        }
```

```
        }
    }

    var p = Point()
    // 下标赋值
    p[0] = 11.1
    p[1] = 22.2
    // 下标访问
    print(p.x) // 11.1
    print(p.y) // 22.2
```

# 11. 简要说明Swift中的初始化器?

类、结构体、枚举都可以定义初始化器 类有2种初始化器: 指定初始化器(designated initializer)、便捷初始化器(convenience initializer)

```
    // 指定初始化器
    init(parameters) {
        statements
    }
    // 便捷初始化器
    convenience init(parameters) {
        statements
    }
```

规则:

- 每个类至少有一个指定初始化器，指定初始化器是类的主要初始化器
- 默认初始化器总是类的指定初始化器
- 类偏向于少量指定初始化器，一个类通常只有一个指定初始化器
- 初始化器的相互调用规则
- 指定初始化器必须从它的直系父类调用指定初始化器
- 便捷初始化器必须从相同的类里调用另一个初始化器
- 便捷初始化器最终必须调用一个指定初始化器

# 12. 比较Swift 和OC中的初始化方法 (init) 有什么不同?

swift 的初始化方法,更加严格和准确, swift初始化方法需要保证所有的非optional的成员变量都完成初始化, 同时 swfit 新增了convenience和 required两个修饰初始化器的关键字

convenience只提供一种方便的初始化器,必须通过一个指定初始化器来完成初始化 required是强制子类重写父类中所修饰的初始化方法

# 13. Swift 是面向对象还是函数式的编程语言?

Swift 既是面向对象的，又是函数式的编程语言。 说 Swift 是面向对象的语言，是因为 Swift 支持类的封装、继承、和多态，从这点上来看与 Java 这类纯面向对象的语言几乎毫无差别。 说 Swift 是函数式编程语言，是因为 Swift 支持 map, reduce, filter, flatmap 这类去除中间状态、数学函数式的方法，更加强调运算结果而不是中间过程。

# 14. 什么是泛型， swift哪些地方使用了泛型?

泛型（generic）可以使我们在程序代码中定义一些可变的部分，在运行的时候指定。使用泛型可以最大限度地重用代码、保护类型的安全以及提高性能。 泛型可以将类型参数化，提高代码复用率，减少代码量。

例如 optional 中的 map、flatMap 、?? (泛型加逃逸闭包的方式，做三目运算)

Swift 泛型

# 15. swift 语法糖？ ！ 的本质（实现原理)

语法糖 是指计算机语言中添加的某种特殊语法，这种语法对语言本身的功能没有什么影响，但更方便编程者使用，使用语法糖能够增加程序的可读性、优化代码，从而减少出错。

swift当中主要是：

1. Selector
2. Then

3. if let和 guard
4. ？？
5. ？ 和 ！

？ 为optional的语法糖 optional 是一个包含了nil 和普通类型的枚举，确保使用者在变量为nil的情况下处理

！ 为optional 强制解包的语法糖

# 16. 什么是高阶函数

一个函数如果可以以某一个函数作为参数, 或者是返回值, 那么这个函数就称之为高阶函数, 如 map, reduce, filter

# 17. 如何解决引用循环

转换为值类型, 只有类会存在引用循环, 所以如果能不用类, 是可以解引用循环的, delegate 使用 weak 属性. 闭包中, 对有可能发生循环引用的对象, 使用 weak 或者 unowned, 修饰

# 18. 请说明并比较以下关键词：Open, Public, Internal, File-private, Private

Swift 有五个级别的访问控制权限，从高到底依次为比如 Open, Public, Internal, File-private, Private。

他们遵循的基本原则是：高级别的变量不允许被定义为低级别变量的成员变量。比如一个 private 的 class 中不能含有 public 的 String。反之，低级别的变量却可以定义在高级别的变量中。比如 public 的 class 中可以含有 private 的 Int。

1. Open 具备最高的访问权限。其修饰的类和方法可以在任意 Module 中被访问和重写；它是 Swift 3 中新添加的访问权限。
2. Public 的权限仅次于 Open。与 Open 唯一的区别在于它修饰的对象可以在任意 Module 中被访问，但不能重写。
3. Internal 是默认的权限。它表示只能在当前定义的 Module 中访问和重写，它可以被一个 Module 中的多个文件访问，但不可以被其他的 Module 中被访问。
4. File-private 也是 Swift 3 新添加的权限。其被修饰的对象只能在当前文件中被使用。例如它可以被一个文件中的 class，extension，struct 共同使用。

5. Private 是最低的访问权限。它的对象只能在定义的作用域内使用。离开了这个作用域，即使是同一个文件中的其他作用域，也无法访问。

# 19. 关键字:Strong,Weak,Unowned 区别?

Swift 的内存管理机制同OC一致,都是ARC管理机制; Strong,和 Weak用法同OC一样

Unowned(无主引用), 不会产生强引用，实例销毁后仍然存储着实例的内存地址(类似于OC中的unsafe_unretained), 试图在实例销毁后访问无主引用，会产生运行时错误(野指针)

# 20. 如何理解copy-on-write?

值类型(比如:struct),在复制时,复制对象与原对象实际上在内存中指向同一个对象,当且仅当修改复制的对象时,才会在内存中创建一个新的对象,

为了提升性能，Struct, String、Array、Dictionary、Set采取了Copy On Write的技术 比如仅当有“写”操作时，才会真正执行拷贝操作 对于标准库值类型的赋值操作，Swift 能确保最佳性能，所有没必要为了保证最佳性能来避免赋值

```
举例:
var str1 = "hi"
var str2 = str1

print(str1)
print(str2)
/*
str1和str2指针地址相同
打印结果:
hi
hi
*/

str1.appendContentsOf("xixi")
print(str1)
print(str2)

/*
str1和str2指针地址不相同
打印结果:
```

```
hixixi
hi
*/
```

# 21. swift 为什么将String,Array,Dictionary设计为值类型?

值类型和引用类型相比,最大优势可以高效的使用内存。 值类型在栈上操作,引用类型在堆上操作。 栈上操作仅仅是单个指针的移动,而堆上操作牵涉到合并,位移,重链接。 Swift 这样设计减少了堆上内存分配和回收次数,使用 copy-on-write将值传递与复制开销降到最低。

# 22. 什么是属性观察?

属性观察是指在当前类型内对特性属性进行监测,并作出响应,属性观察是 swift 中的特性,具有2种, willset 和 didset

```
var title: String {
    willSet {
        print("willSet", newValue)

    }
    didSet {
        print("didSet", oldValue, title)
    }

}

willSet会传递新值，默认叫newValue
didSet会传递旧值，默认叫oldValue
在初始化器中设置属性值不会触发willSet和didSet
```

# 23. 如何将Swift 中的协议(protocol)中的部分方法设计为可选(optional)?

1. 在协议和方法前面添加 @objc,然后在方法前面添加 optional关键字,改方式实际上是将协议转为了OC的方式

```
@objc protocol someProtocol {
  @objc  optional func test()
}
```

2. 使用扩展(extension),来规定可选方法,在 swift 中,协议扩展可以定义部分方法的默认实现

```
protocol someProtocol {
    func test()
}

extension someProtocol{
    func test() {
        print("test")
    }
}
```

# 24. 比较 Swift和OC中的 protocol 有什么不同?

Swift 和OC中的 protocol相同点在于: 两者都可以被用作代理; 不同点: Swift中的 protocol还可以对接口进行抽象,可以实现面向协议,从而大大提高编程效率,Swift中的protocol可以用于值类型,结构体,枚举;

# 25 Swift 和OC 中的自省 有什么区别?

自省在OC中就是判断某一对象是否属于某一个类的操作,有以下2中方式

`[obj iskinOfClass:[SomeClass class]]` `[obj isMemberOfClass:[SomeClass class]]`

在 Swift 中由于很多 class 并非继承自 NSObject, 故而 Swift 使用 is 来判断是否属于某一类型, is 不仅可以作用于class, 还是作用于enum和struct

# 26. 什么是函数重载? swift 支不支持函数重载?

函数重载是指: 函数名称相同,函数的参数个数不同, 或者参数类型不同,或参数标签不同, 返回值类型与函数重载无关 swift 支持函数重载

# 27. Swift 中的闭包结构是什么样子的?什么是尾随闭包?什么是逃逸闭包?什么是自动闭包?

```
{
    (参数列表) -> 返回值类型 in 函数体代码
}
```

2. 如果你需要将一个很长的闭包表达式作为最后一个参数传递给函数，将这个闭包替换成为尾随闭包的形式很有用。尾随闭包是一个书写在函数圆括号之后的闭包表达式，函数支持将其作为最后一个参数调用。

```
// fn 就是一个尾随闭包参数
func exec(v1: Int, v2: Int, fn: (Int, Int) -> Int) {
    print(fn(v1, v2))
}

// 调用
exec(v1: 10, v2: 20) {
    $0 + $1
}
```

3. 当一个闭包作为参数传到一个函数中，但是这个闭包在函数返回之后才被执行，我们称该闭包从函数中逃逸。当你定义接受闭包作为参数的函数时，你可以在参数名之前标注 @escaping，用来指明这个闭包是允许“逃逸”出这个函数的。 非逃逸闭包、逃逸闭包，一般都是当做参数传递给函数 非逃逸闭包:闭包调用发生在函数结束前，闭包调用在函数作用域内 逃逸闭包:闭包有可能在函数结束后调用，闭包调用逃离了函数的作用域，需要通过@escaping声明

```
// 定义一个数组用于存储闭包类型
var completionHandlers: [() -> Void] = []

//  在方法中将闭包当做实际参数,存储到外部变量中
func someFunctionWithEscapingClosure(completionHandler: @escaping () ->
Void) {
    completionHandlers.append(completionHandler)
}
```

someFunctionWithEscapingClosure(_:) 函数接受一个闭包作为参数，该闭包被添加到一个函数外定义的数组中。如果你不将这个参数标记为 @escaping，就会得到一个编译错误。

4. 自动闭包不接受任何参数，延迟求值，只有在被调用时才会返回被包装在其中的表达式的值。 在我们调用函数A获取函数结果作为参数传递给另一个函数B时，无论这个结果在函数B中是否使用，函数A都会被执行，如下第二次调用goodAfternoon(afternoon: false, who: getName())方法后并没有使用getName的值但是函数getName还是执行了

```
func getName() -> String{
    print(#function)
    return "DKJone"
}
func goodAfternoon(afternoon:Bool ,who:String){
    if afternoon {
        print("Good afternoon, (who)")
    }
}
print("------goodAfternoon(afternoon: true, who: getName())-------")
goodAfternoon(afternoon: true, who: getName())
print("------goodAfternoon(afternoon: false, who: getName())-------")
goodAfternoon(afternoon: false, who: getName())
/*log:
    ------goodAfternoon(afternoon: true, who: getName())-------
    getName()
    Good afternoon, DKJone
    ------goodAfternoon(afternoon: false, who: getName())-------
    getName()
*/
```

当我们在第二个参数添加 @autoclosure 关键字后，第二个参数中的代码会在函数执行时自动生成一个闭包，只有闭包真正执行时第二个参数种类代码才会调用所以下面的goodMooning(morning: false, who: getName())并没有调用getName方法

```
//@autoclosure
func goodMooning(morning:Bool ,who:@autoclosure() -> String){
    if morning {
```

```
            print("Good morning, (who())")
        }
    }
    print("------goodMooning(morning: true, who: getName())-------")
    goodMooning(morning: true, who: getName())
    print("------goodMooning(morning: false, who: getName())-------")
    goodMooning(morning: false, who: getName())
    /* log:
    ------goodMooning(morning: true, who: getName())-------
    getName()
    Good morning, DKJone
    ------goodMooning(morning: false, who: getName())-------
    */
```

为了避免与期望冲突，使用了@autoclosure的地方最好明确注释清楚:这个值会被推迟执行

@autoclosure 会自动将 20 封装成闭包 { 20 }

@autoclosure 只支持 () -> T 格式的参数

@autoclosure 并非只支持最后1个参数

有@autoclosure、无@autoclosure，构成了函数重载

如果你想要自动闭包允许逃逸，就同时使用 @autoclosure 和 @escaping 标志。

# 28. Swift 中如何使用单例模式?

可以通过类型属性+let+private 来写单例; 代码如下如下:

```
 public class FileManager {
    public static let shared = {
        // ....
        // ....
        return FileManager()
}()
    private init() { }
}
```

# 29. 什么可选链?

可选链是一个调用和查询可选属性、方法和下标的过程，它可能为 nil 。如果可选项包含值，属性、方法或者下标的调用成功；如果可选项是 nil ，属性、方法或者下标的调用会返回 nil 。多个查询可以链接在一起，如果链中任何一个节点是 nil ，那么整个链就会得体地失败。

多个?可以链接在一起 如果链中任何一个节点是nil，那么整个链就会调用失败

# 30. OC和Swift中的扩展（Extension）区别

OC中有分类和扩展 swift中只有扩展（更类似OC中的分类）

swift中扩展(Extensions)的说明:

扩展就是向一个已有的类、结构体、枚举类型或者协议类型添加新功能（functionality）。 这包括在没有权限获取原始源代码的情况下扩展类型的能力（即逆向建模）。 扩展和 Objective-C 中的分类（categories）类似。（不过更加强大，而且与Objective-C 不同的是，Swift 的扩展没有名字。）

tip： 与OC不同的是，在Swift 的 extension 中 不可以直接添加属性。编译会报错。 和oc一样，我们可以用关联方法来添加属性。

类别（Category）和扩展（Extension）-- OC和Swift中的区别