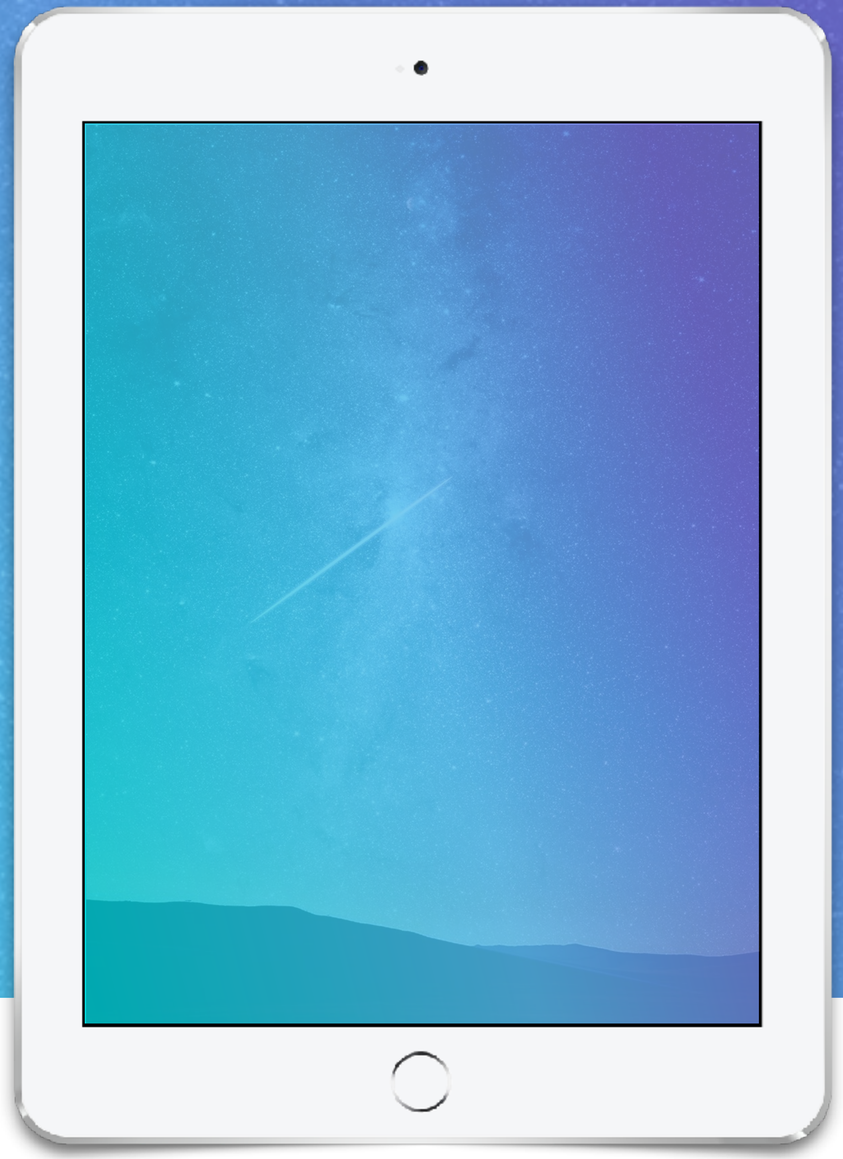
iOS 黑魔法课程

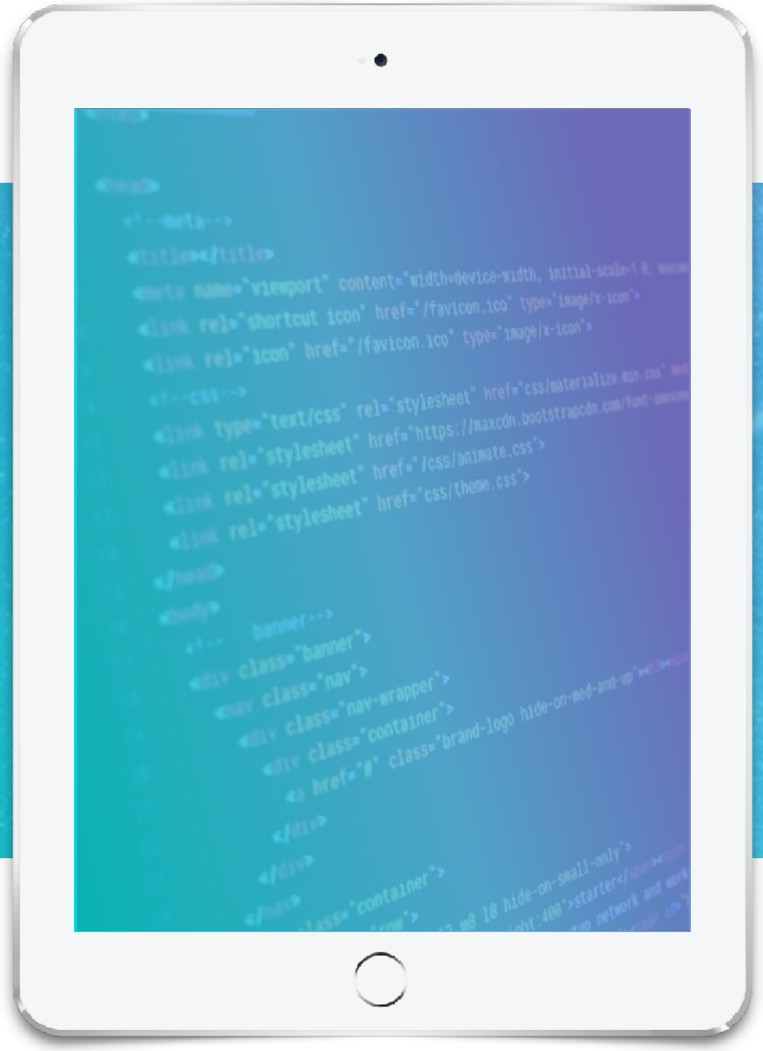
第四课

Swift 语言魔法



StuQueue

一个新的学习方式



本课内容

- Swift vs OC
- Swift OOP 详解
- Swift 自身特点

StuQ

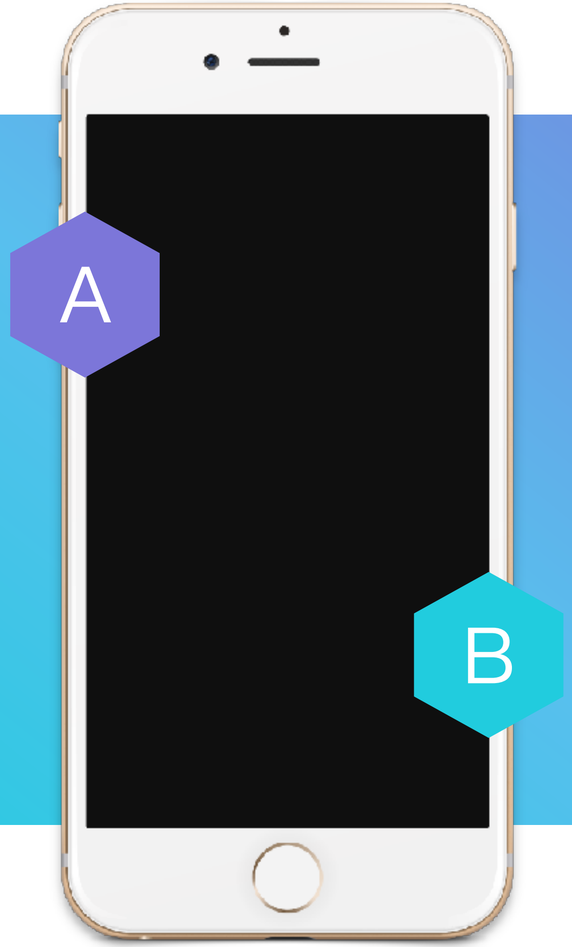
Swift VS OC

StuQ

SWIFT VS OC

类型系统

静态类型
编译器决定类型结构
编译期绑定方法调用



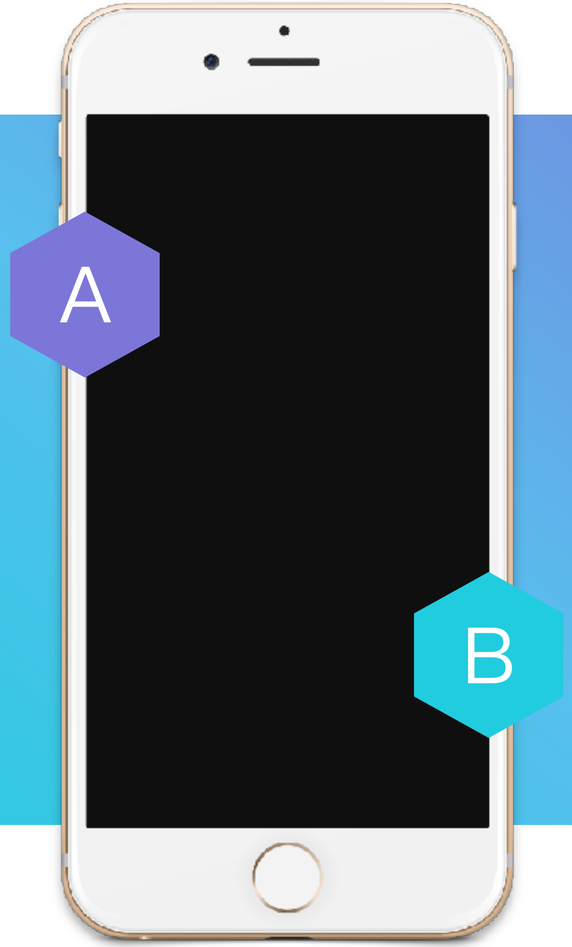
动态类型
运行时决定类型结构
运行时动态寻找方法

StuQ

SWIFT VS OC

泛型

通过虚函数表派发函数调用
开启优化后可以实现泛型特化



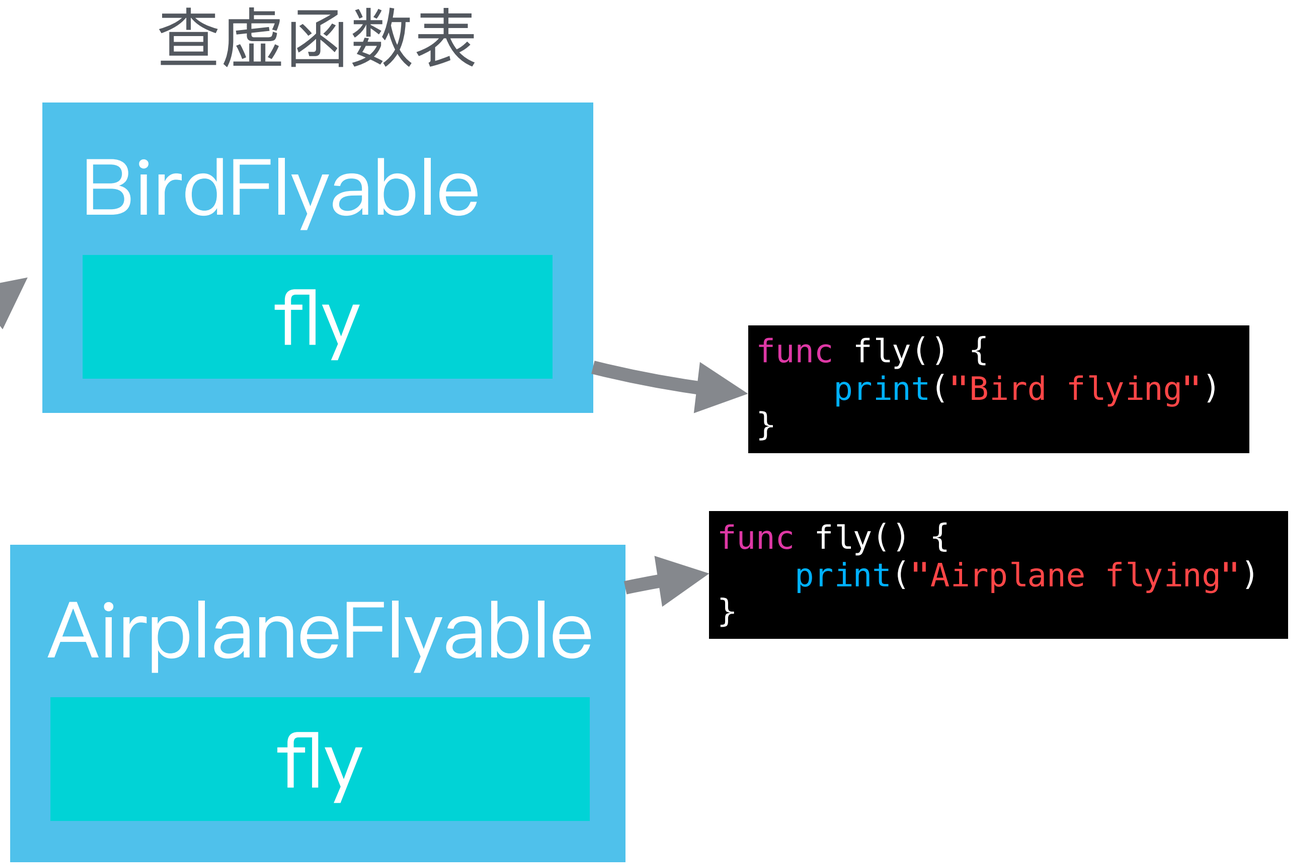
只用于编译器类型检查
运行时不包含任何泛型信息
方法调用全部动态派发

StuQ

SWIFT VS OC

通过虚函数表派发函数调用

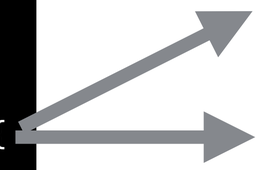
```swift
protocol Flyable {
    func fly()
}

struct Bird : Flyable {
    func fly() {
        print("Bird flying")
    }
}
struct Airplane : Flyable {
    func fly() {
        print("Airplane flying")
    }
}

func somethingFly<F: Flyable>(_ something: F) {
    something.fly()
}

somethingFly(Bird())
somethingFly(Airplane())
```



SWIFT VS OC

泛型特化 (Swift 2.0+ 编译优化) (C++ 模板)

```swift
protocol Flyable {
    func fly()
}

struct Bird : Flyable {
    func fly() {
        print("Bird flying")
    }
}
struct Airplane : Flyable {
    func fly() {
        print("Airplane flying")
    }
}

func somethingFly<F: Flyable>(_ something: F) {
    something.fly()
}

somethingFly(Bird())
somethingFly(Airplane())
```

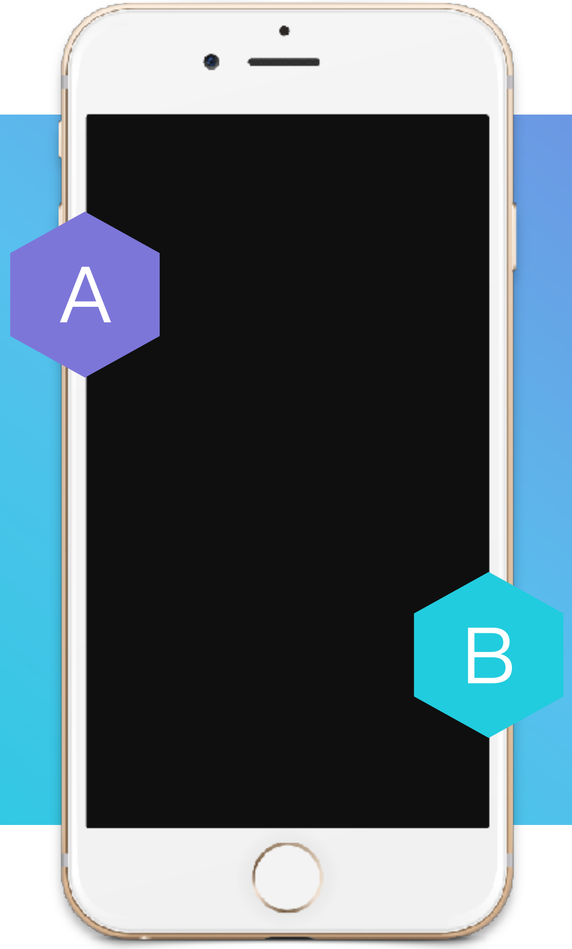


```swift
struct Bird : Flyable {
    func fly() {
        print("Bird flying")
    }
}
struct Airplane : Flyable {
    func fly() {
        print("Airplane flying")
    }
}

func somethingFlyOfABird(_ something: Bird) {
    something.fly()
}

func somethingFlyOfAnAirplane(_ something: Airplane) {
    something.fly()
}

somethingFlyOfABird(Bird())
somethingFlyOfAnAirplane(Airplane())
```

StuQ

SWIFT VS OC

类型统一性



类型统一

Int

基于 C 语言，类型不统一

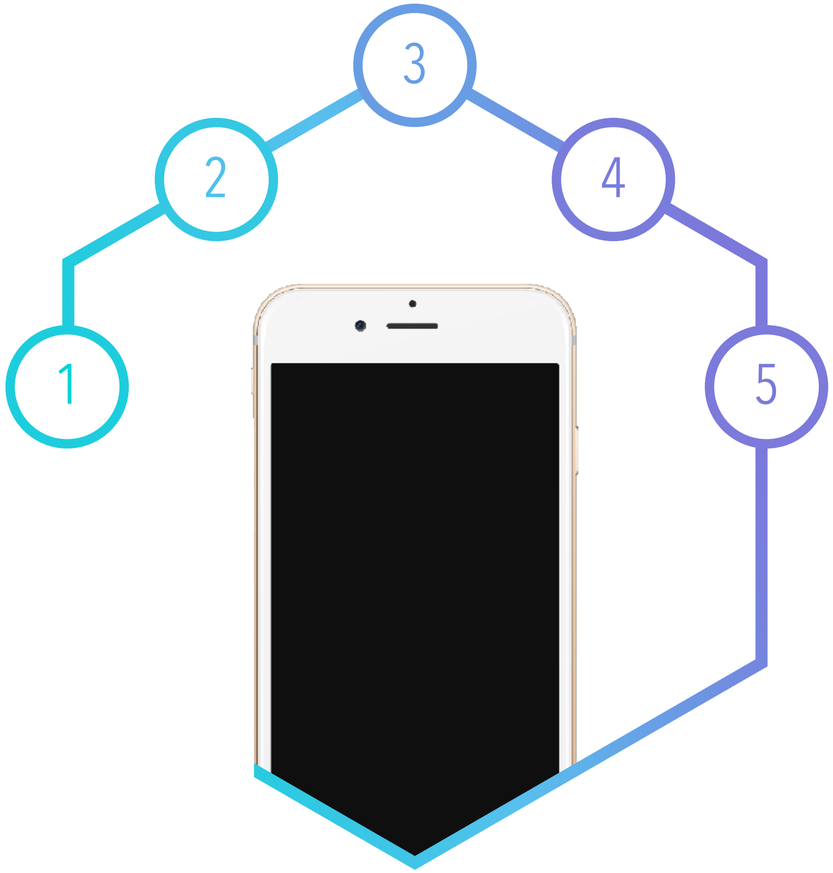
int, double, NSNumber

StuQ

SWIFT VS OC

类型统一的好处

- 可以扩展基本类型
- 不用处理像 OC 的 int, NSNumber 之间的转换



StuQ

SWIFT VS OC

扩展基本类型

```
extension Int {
    func times(block: () -> Void) {
        for i in 0..<self {
            block()
        }
    }
}

3.times {
    print("hello")
}
```

StuQ

SWIFT VS OC

闭包默认捕获引用

```swift
func makeIncrementer(forIncrement amount: Int) -> () -> Int {
    var runningTotal = 0

    return { () -> Int in
        runningTotal += amount
        return runningTotal
    }
}

let incrementer = makeIncrementer(forIncrement: 10)
incrementer()       // 10
incrementer()       // 20
incrementer()       // 30
```

需要明确指明闭包捕获引用

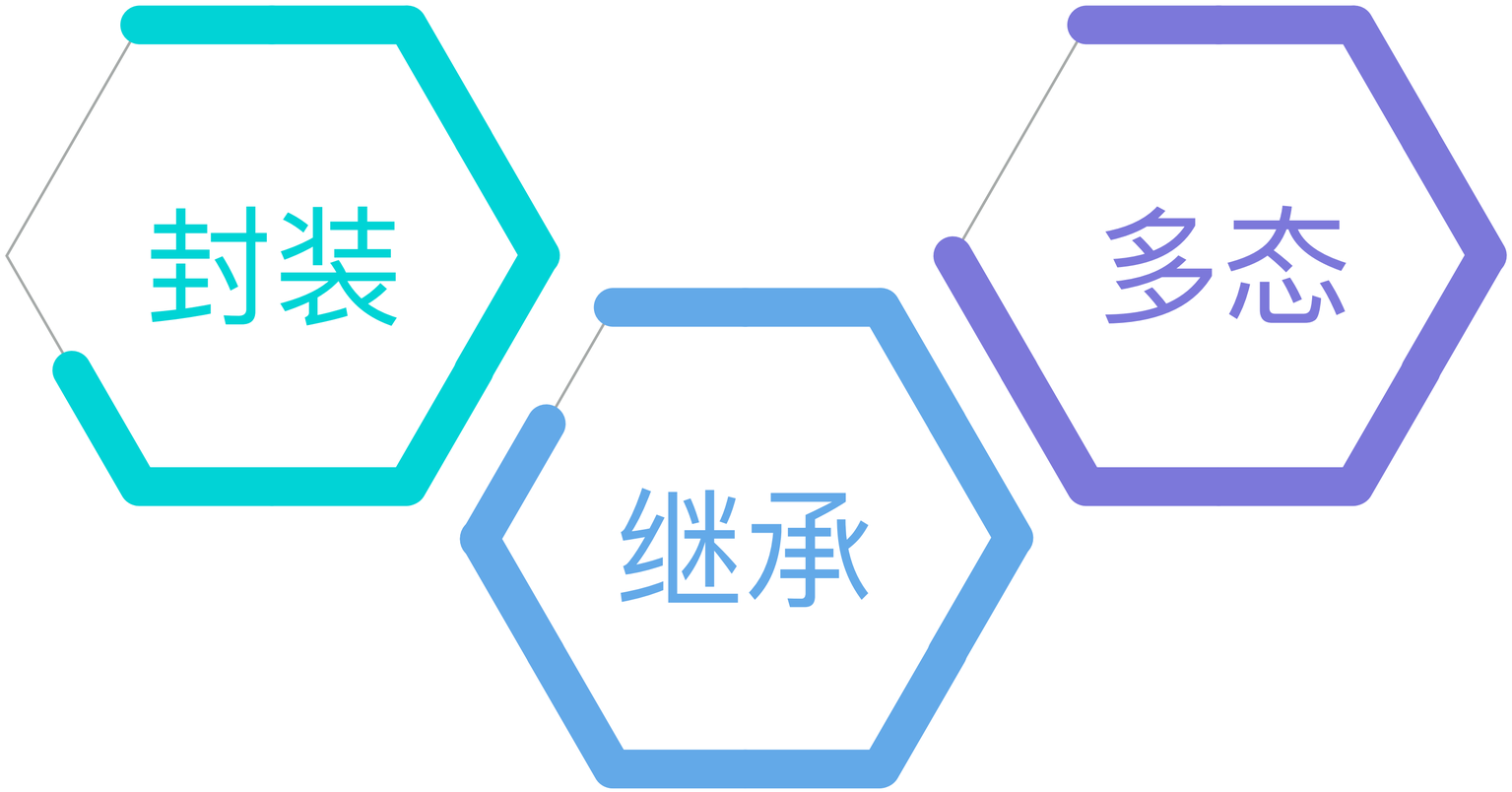
```objc
typedef int(^IncrementerType)();

IncrementerType makeIncrementr(int amount) {
    __block int runningTotal = 0;

    return ^(){
        runningTotal += amount;
        return runningTotal;
    };
}

int main(int argc, const char * argv[]) {
    @autoreleasepool {
        IncrementerType incrementer = makeIncrementr(10);
        NSLog(@"%d", incrementer());        // 10
        NSLog(@"%d", incrementer());        // 20
        NSLog(@"%d", incrementer());        // 30
    }
    return 0;
}
```

StuQ 课程

Swift OOP 详解

StuQ ueue

SWIFT OOP 详解



StuQ

SWIFT OOP 详解

封装

1. 属性的公开与隐藏
2. 方法的公开与隐藏
3. 类的公开与隐藏

StuQ

SWIFT OOP 详解

封装

- 属性、函数、类、结构体、枚举 都可以设置访问级别

StuQ

SWIFT OOP 详解

访问级别

限制增多

- **public:** 可以在模块内外任意使用
- **internal:** 只能在同一模块内部使用
- **fileprivate:** 只能在定义的文件内部访问
- **private:** 只能在所定义的作用域内使用

StuQ

SWIFT OOP 详解

访问级别

```
public class SomePublicClass {                      // 显式公开类
    public var somePublicProperty = 0               // 显式公开类成员
    var someInternalProperty = 0                    // 隐式内部类成员
    fileprivate func someFilePrivateMethod() {}     // 显式文件私有类成员
    private func somePrivateMethod() {}             // 显式私有类成员
}

class SomeInternalClass {                           // 隐式内部类
    var someInternalProperty = 0                    // 隐式内部类成员
    fileprivate func someFilePrivateMethod() {}     // 显式文件私有类成员
    private func somePrivateMethod() {}             // 显式私有类成员
}

fileprivate class SomeFilePrivateClass {            // 显式文件私有类
    func someFilePrivateMethod() {}                 // 隐式文件私有类成员
    private func somePrivateMethod() {}             // 显式私有类成员
}

private class SomePrivateClass {                    // 显式私有类
    func somePrivateMethod() {}                     // 隐式私有类成员
}
```

StuQ

SWIFT OOP 详解

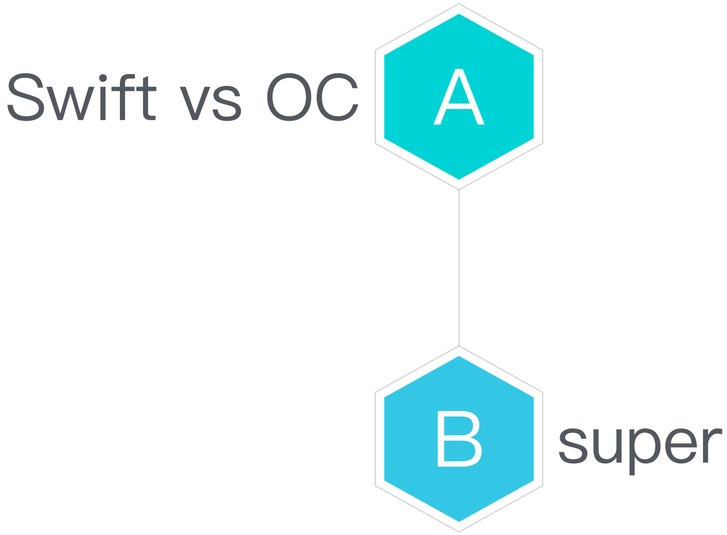
Swift 封装破解说明

- 声明私有方法为公有方法　　编译不通过 ❌
- 使用运行时调用私有方法　　没有运行时 ❌
- 直接从符号表寻找函数地址　私有函数在符号表不存在 ❌

StuQ

SWIFT OOP 详解

继承

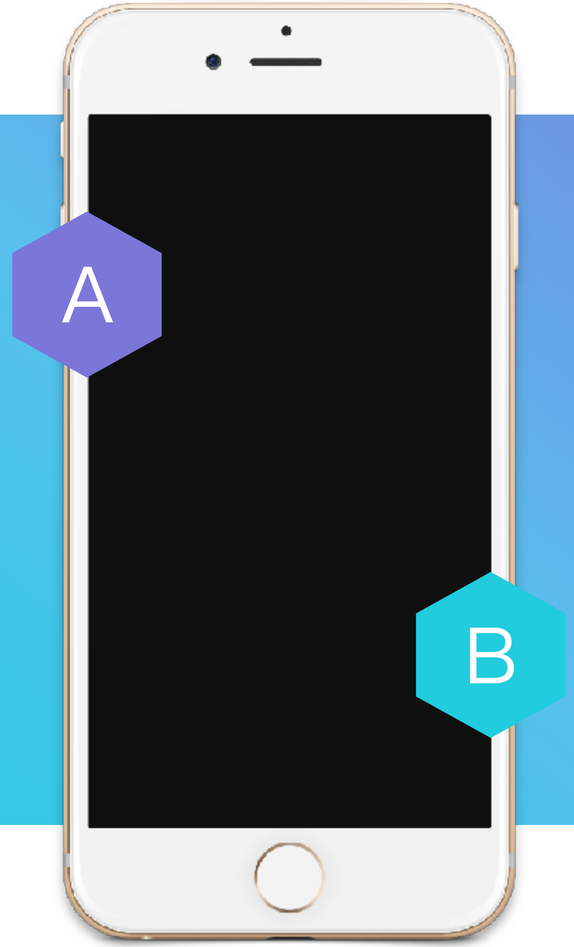


StuQ

SWIFT OOP 详解

Swift vs OC

- 编译器实现，编译后不存在实体类
- 编译器静态实现 super



- 通过保存 superclass 指针指向父类
- 通过 objc_super 结构动态实现 super

StuQ

SWIFT OOP 详解

super 的实现

```swift
class MyClass {
    func hello() {
        print("hello from MyClass")
    }
}

class MySubClass : MyClass {
    override func hello() {
        super.hello()

        print("hello from MySubClass")
    }
}

let a = MySubClass()
a.hello()
```

编译期确定实际调用的函数，并直接调用

StuQ

SWIFT OOP 详解

多态

协议

泛型

StuQ

SWIFT OOP 详解

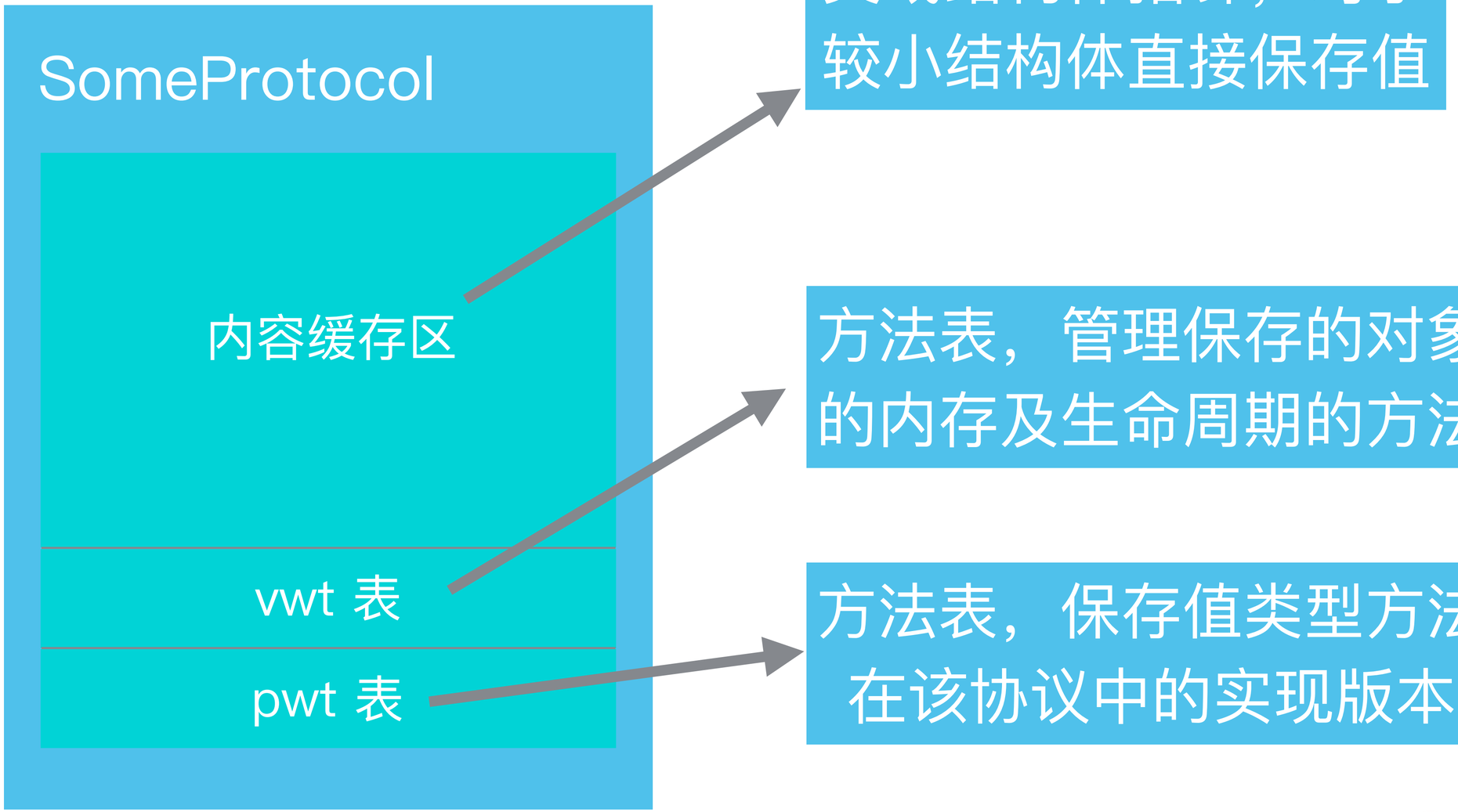
协议

```
protocol SomeProtocol {}
```

SWIFT OOP 详解

协议

```
protocol SomeProtocol {}
```



StuQ

Swift 自身特点

StuQ

SWIFT 自身特点

1. 运算符重载
2. 协议扩展
3. FP 支持

StuQ

SWIFT 自身特点

运算符重载

语法糖

StuQ

SWIFT 自身特点

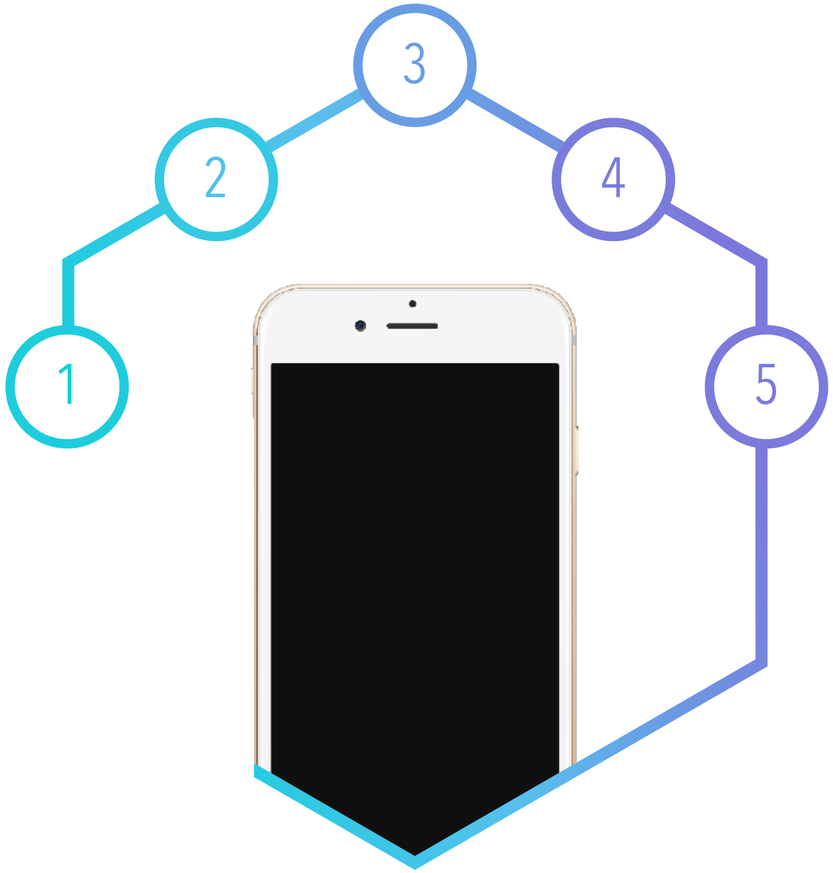
运算符重载实现语法糖

```
precedencegroup Curry { associativity: left }
infix operator <| : Curry

func <|<A, B, C, D, E, T>(f: @escaping (A, B, C, D, E) -> T , a: A) -> (B, C, D, E) -> T {
    return { f(a, $0, $1, $2, $3) }
}

func <|<A, B, C, D, T>(f: @escaping (A, B, C, D) -> T , a: A) -> (B, C, D) -> T {
    return { f(a, $0, $1, $2) }
}

func <|<A, B, C, T>(f: @escaping (A, B, C) -> T , a: A) -> (B, C) -> T {
    return { f(a, $0, $1) }
}

func <|<A, B, T>(f: @escaping (A, B) -> T , a: A) -> (B) -> T {
    return { f(a, $0) }
}

func <|<A, T>(f: @escaping (A) -> T , a: A) -> T {
    return f(a)
}

func test(_ a: Int, _ b: Int, _ c: Int, _ d: Int, _ e: Int) -> Int {
    return a + b + c + d + e
}

let func1 = test <| 5 <| 4  // (Int, Int, Int) -> Int
print(func1(3, 2, 1))       // 15

let result = func1 <| 3 <| 2 <| 1
print(result)               // 15
```

SWIFT 自身特点

协议扩展

- 组合优于继承
- 迅速给一个新的类添加一系列易用的方法



StuQ

SWIFT 自身特点

协议扩展: Enumerable 例子

在 Ruby 中，只要一个类实现了 Enumerable 的 each 方法，就可以轻松的给这个类加入大量易用的方法，这些方法都是通过调用 each 方法实现的，不需要每个方法再手动实现一遍

```
.all?
.any?
.collect
.count
.drop
.drop_while
.each_with_index
.each_with_object
.find
.find_all
.first
.include?
.map
.max
```

```
.min
.none?
.one?
.reduce
.reject
.reverse_each
.select
.sort
.sort_by
.sum
.take
.take_while
.uniq
.zip
```

StuQ queue

SWIFT 自身特点

协议扩展: Enumerable 例子

用 Swift 来实现这一功能，首先定义一个

Enumerable 协议，遵守者需要实现一个 each 方法

```
protocol Enumerable {
    associatedtype Element
    func each(block: (Self.Element) -> Void)
}
```

StuQ

SWIFT 自身特点

协议扩展: Enumerable 例子

通过协议扩展给协议的遵守者加入大量的默认方法，

每个方法都是通过调用 each 实现

```
extension Enumerable {
    func dropIf(predicate: (Self.Element) -> Bool) -> [Self.Element] {
        var result = [Element]()
        each { item in if !predicate(item) { result.append(item) } }
        return result
    }

    func dropWhile(predicate: (Self.Element) -> Bool) -> [Self.Element] {
        ...
    }

    func findAll(predicate: (Self.Element) -> Bool) -> [Self.Element] {
        ...
    }

    ... /* many more methods here */ ...
}
```

StuQ

SWIFT 自身特点

协议扩展: Enumerable 例子

现在我们定义一个我们自己的类型

```
struct Family: Enumerable {
    var name = "Smith"
    var father = "Bob"
    var mother = "Alice"
    var child = "Carol"
}
```

StuQ

SWIFT 自身特点

协议扩展: Enumerable 例子

给它实现 Enumerable 协议

```
struct Family: Enumerable {
    var name = "Smith"
    var father = "Bob"
    var mother = "Alice"
    var child = "Carol"

    func each(block: (String) -> Void) {
        for i in 0...2 {
            switch i {
            case 0: block("\(father) \(name)")
            case 1: block("\(mother) \(name)")
            case 2: block("\(child) \(name)")
            default: break
            }
        }
    }

}
```

StuQ

SWIFT 自身特点

协议扩展: Enumerable 例子

现在，我们获得了诸多好用的方法，例如

```
let f = Family()
let withoutBob = f.dropIf { p in p.hasPrefix("Bob") }

// ["Alice Smith", "Carol Smith"]
```

StuQ

SWIFT 自身特点

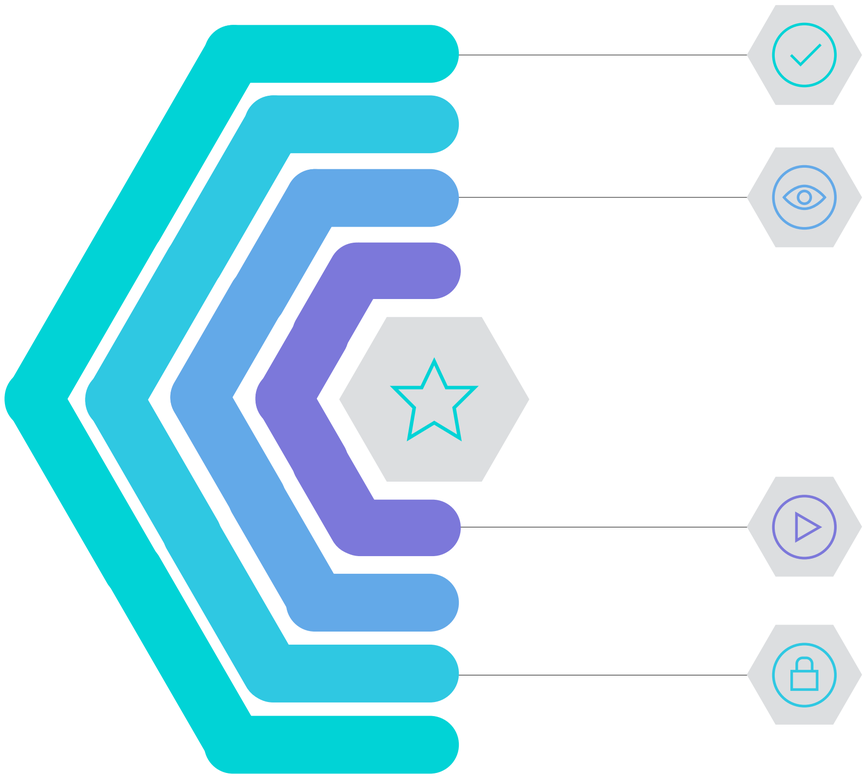
面向协议编程

先用协议抽象世界，再实现具体代码

StuQ

SWIFT 自身特点

FP 支持



- 函数是一类对象
- 高阶函数
- 模式匹配
- 不可变类型

StuQ

SWIFT 自身特点

函数是一类对象

函数是基本类型

函数可以赋值给变量

函数可以接受其他函数作为参数

函数可以返回一个函数

（高阶函数）

StuQ

SWIFT 自身特点

高阶函数

写出更加优雅可读的代码

```
.filter
.map
.reduce
```

StuQ

SWIFT 自身特点

高阶函数

```
let items = [-5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5]

let result = items.filter { $0 > 0 }
    .map { $0 * 2 }
    .reduce(0) { $0 + $1 }

print(result) // 30
```

StuQ

SWIFT 自身特点

模式匹配

没有模式匹配的代码（复杂的条件判断）

```
enum Limit: String {
    case Admin = "管理员"
    case Guest = "游客"
}

func login(limit: Limit, userName: String, passwordDigest: Int) {
    if limit == .Admin {
        if userName == "xiaoming" && passwordDigest == "abc123".hashValue {
            print("管理员登录成功")
        } else {
            print("帐号密码错误")
        }
    } else {
        print("游客身份登录")
    }
}

login(limit: .Admin, userName: "xiaoming", passwordDigest: "abc123".hashValue) // 管理员登录成功
```

StuQ

SWIFT 自身特点

模式匹配

使用模式匹配实现

```
enum Limit: String {
    case Admin = "管理员"
    case Guest = "游客"
}

func login(limit: Limit, userName: String, passwordDigest: Int) {
    switch (limit, userName, passwordDigest) {
    case (.Admin, "xiaoming", let passHash) where passHash == "abc123".hashValue:
        print("管理员登录成功")
    case (.Admin, _, _):
        print("帐号密码错误")
    case (.Guest, _, _):
        print("游客身份登录")
    }
}

login(limit: .Admin, userName: "xiaoming", passwordDigest: "abc123".hashValue) // 管理员登录成功
```

StuQ neun

SWIFT 自身特点

模式匹配

```
if limit == .Admin {
    if userName == "xiaoming" && passwordDigest == "abc123".hashValue {
        print("管理员登录成功")
    } else {
        print("帐号密码错误")
    }
} else {
    print("游客身份登录")
}
```

VS

```
switch (limit, userName, passwordDigest) {
case (.Admin, "xiaoming", let passHash) where passHash == "abc123".hashValue:
    print("管理员登录成功")
case (.Admin, _, _):
    print("帐号密码错误")
case (.Guest, _, _):
    print("游客身份登录")
}
```

StuQ

SWIFT 自身特点

不可变类型

可以用 let 将变量声明为不可变，结构体为值类型，可以实现整个程序使用的类型都为不可变类型，实现纯函数式编程

StuQ

SWIFT 自身特点

associatype & typealias

关联类型为协议中的某个类型提供了一个占位名，

其代表的实际类型在协议被采纳时才会被指定

StuQ

SWIFT 自身特点

associatype & typealias

本质上是泛型协议

```
protocol Container {
    associatedtype ItemType
    mutating func append(_ item: ItemType)
    var count: Int { get }
    subscript(i: Int) -> ItemType { get }
}
```

```
struct IntContainer: Container {
    typealias ItemType = Int

    var items = [Int]()
    mutating func append(_ item: Int) {
        self.items.append(item)
    }
    var count: Int {
        return items.count
    }
    subscript(i: Int) -> Int {
        return items[i]
    }
}
```

StuQ

SWIFT 自身特点

枚举的关联值：更加优雅的数据抽象

```
enum Result<Value> {
    case success(Value)
    case failure(Error)
}

let result: Result<Int> = .success(100)

switch result {
case .success(let value):
    print(value)
case .failure(let error):
    // deal with error
    print(error)
}

// will print 100
```

事实上，Optional 也是枚举的关联值实现的

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

StuQ

Q&A

感谢聆听