

# OC 和 Swift 运行时简介



扫码试看/订阅

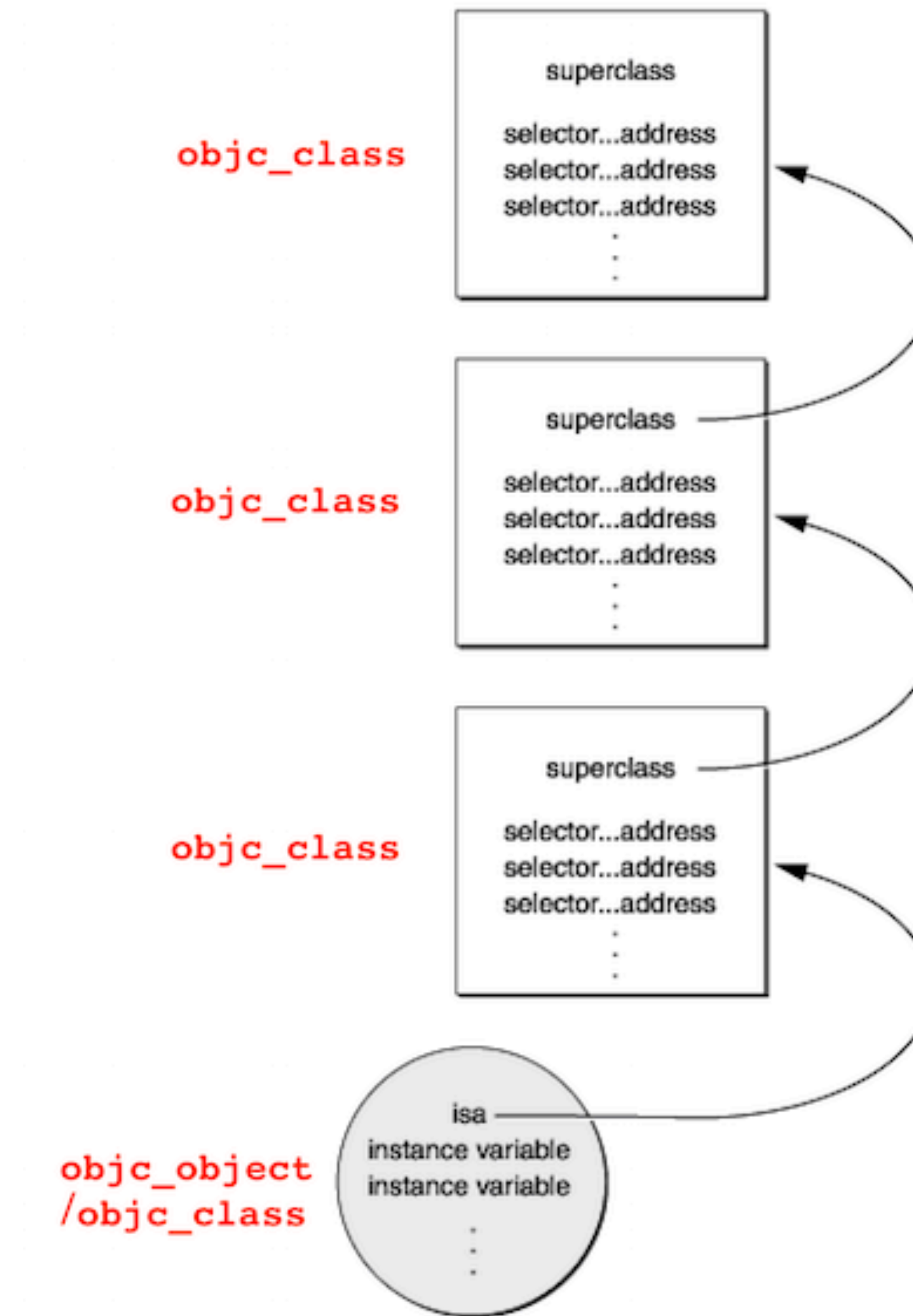
《Swift核心技术与实战》视频课程

# Objective-C 运行时

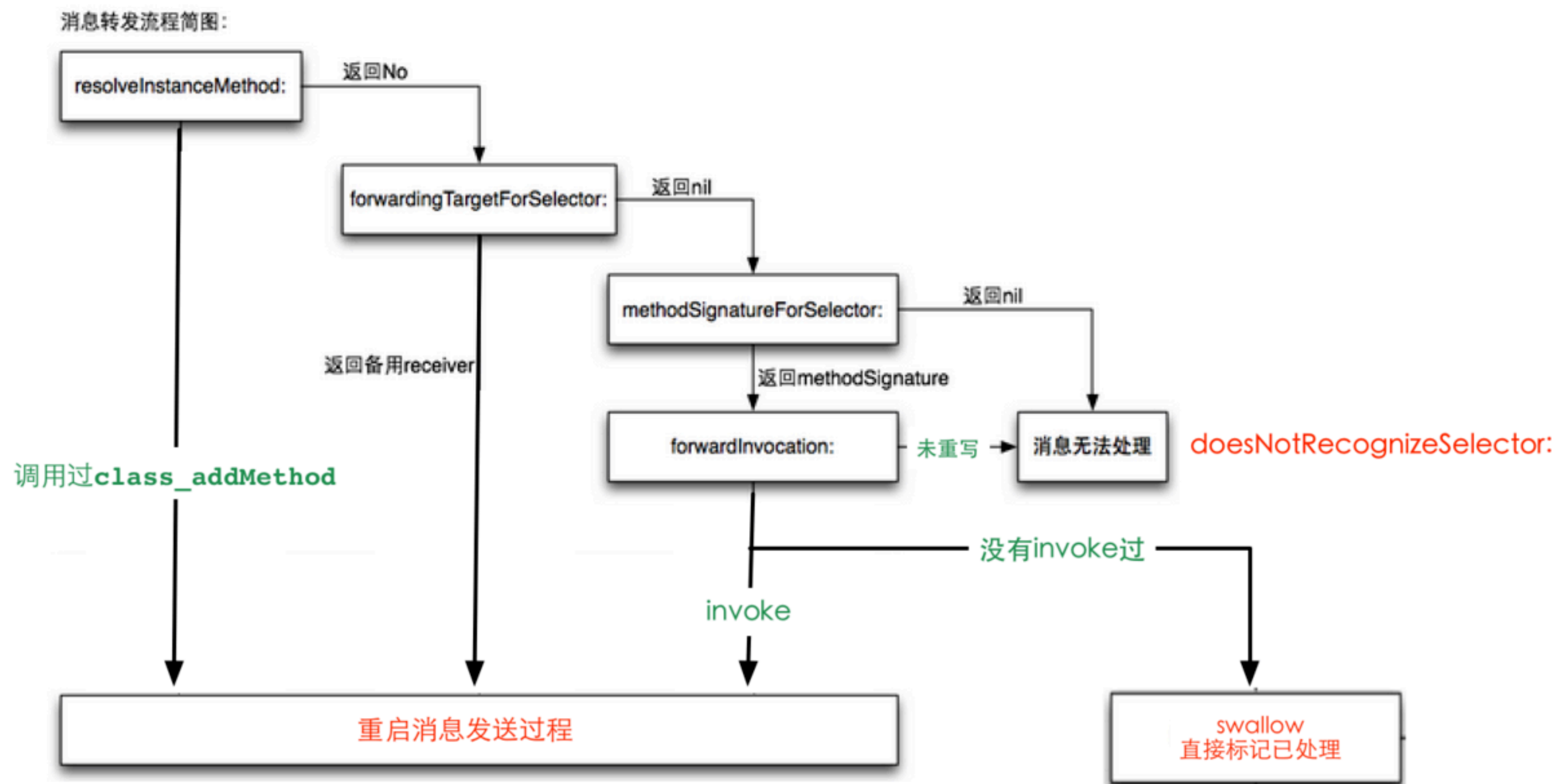
- 动态类型 (dynamic typing)
- 动态绑定 (dynamic binding)
- 动态装载 (dynamic loading)

# Objective-C 运行时

- [object methodA];



# Objective-C 运行时



# 派发方式

- 直接派发 (Direct Dispatch)
- 函数表派发 (Table Dispatch )
- 消息机制派发 (Message Dispatch )

# 直接派发

- 直接派发是最快的, 不止是因为需要调用的指令集会更少, 并且编译器还能够有很大的优化空间, 例如函数内联等, 直接派发也有人称为静态调用。
- 然而, 对于编程来说直接调用也是最大的局限, 而且因为缺乏动态性所以没办法支持继承和多态。

# 函数表派发

- 函数表派发是编译型语言实现动态行为最常见的实现方式. 函数表使用了一个数组来存储类声明的每一个函数的指针. 大部分语言把这个称为 “virtual table” (虚函数表), Swift 里称为 “witness table”. 每一个类都会维护一个函数表, 里面记录着类所有的函数, 如果父类函数被 override 的话, 表里面只会保存被 override 之后的函数. 一个子类新添加的函数, 都会被插入到这个数组的最后. 运行时会根据这一个表去决定实际要被调用的函数.



# 函数表派发

```
class ParentClass {  
  func method1() {}  
  func method2() {}  
}  
class ChildClass: ParentClass {  
  override func method2() {}  
  func method3() {}  
}
```

Offset

0
1
2

0xA00	ParentClass
0x121	method1
0x122	method2

0xB00	ChildClass
0x121	method1
0x222	method2
0x223	method3

```
let obj = ChildClass()  
obj.method2()
```

# 函数表派发

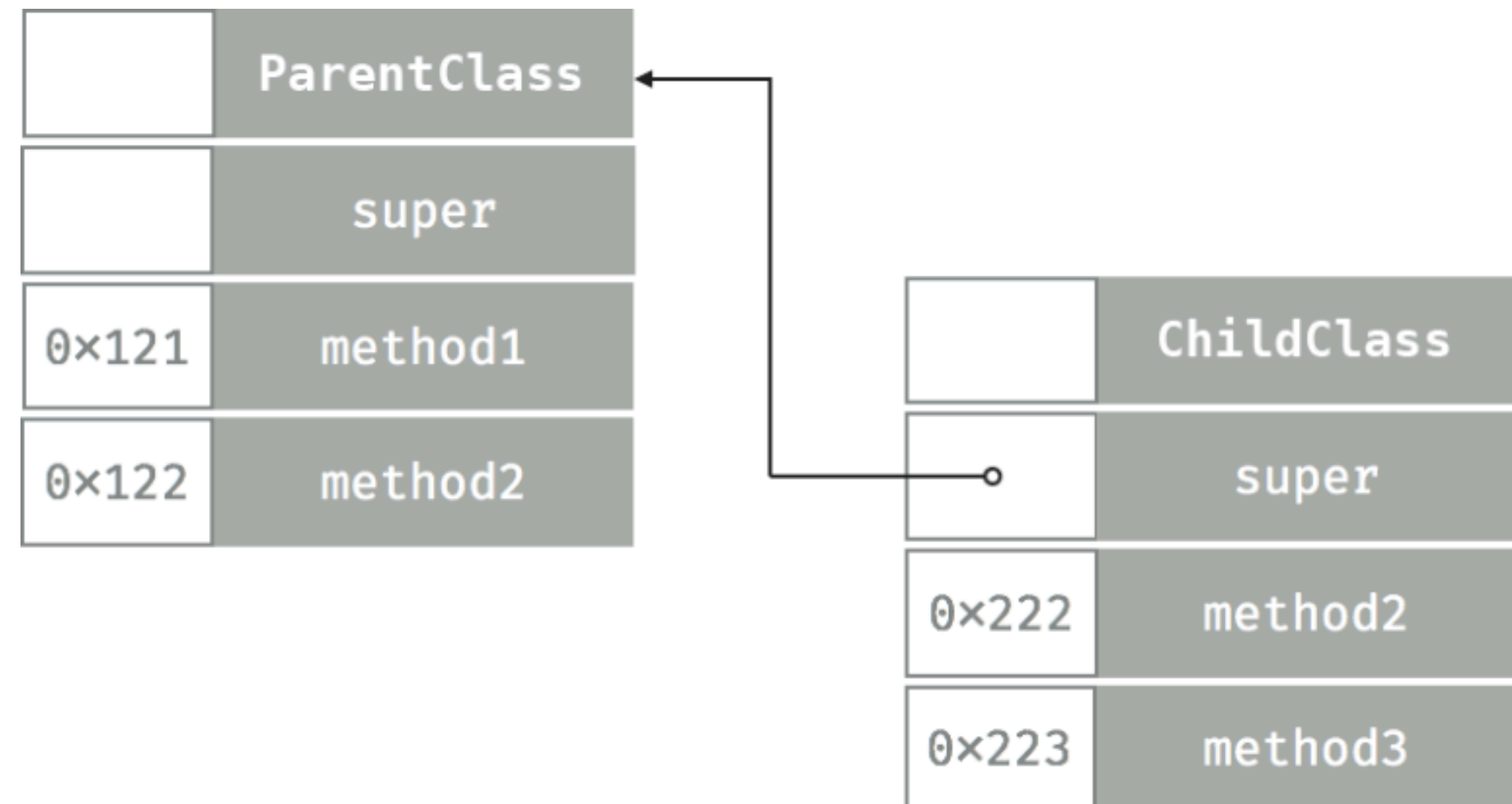
- 查表是一种简单, 易实现, 而且性能可预知的方式. 然而, 这种派发方式比起直接派发还是慢一点. 从字节码角度来看, 多了两次读和一次跳转, 由此带来了性能的损耗. 另一个慢的原因在于编译器可能会由于函数内执行的任务导致无法优化. (如果函数带有副作用的话)
- 这种基于数组的实现, 缺陷在于函数表无法拓展. 子类会在虚数函数表的最后插入新的函数, 没有位置可以让 extension 安全地插入函数.

# 消息机制派发

- 消息机制是调用函数最动态的方式. 也是 Cocoa 的基石, 这样的机制催生了 KVO, UIAppearance 和 CoreData 等功能. 这种运作方式的关键在于开发者可以在运行时改变函数的行为. 不止可以通过 swizzling 来改变, 甚至可以用 isa-swizzling 修改对象的继承关系, 可以在面向对象的基础上实现自定义派发.

# 消息机制派发

```
class ParentClass {  
    dynamic func method1() {}  
    dynamic func method2() {}  
}  
class ChildClass: ParentClass {  
    override func method2() {}  
    dynamic func method3() {}  
}
```



# Swift 运行时

- 纯 Swift 类的函数调用已经不再是 Objective-c 的运行时发消息，而是类似 C++ 的 vtable，在编译时就确定了调用哪个函数，所以没法通过 runtime 获取方法、属性。
- 而 Swift 为了兼容 Objective-C，凡是继承自 NSObject 的类都会保留其动态性，所以我们能通过 runtime 拿到他的方法。这里有一点说明：老版本的 Swift（如 2.2）是编译期隐式的自动帮你加上了 @objc，而 4.0 以后版本的 Swift 编译期去掉了隐式特性，必须使用显式添加。
- 不管是纯 Swift 类还是继承自 NSObject 的类只要在属性和方法前面添加 @objc 关键字就可以使用 runtime。

# Swift 运行时

	原始定义	扩展
值类型	直接派发	直接派发
协议	函数表派发	直接派发
类	函数表派发	直接派发
继承自NSObject的类	函数表派发	消息机制派发

# Swift 运行时

- 值类型总是会使用直接派发, 简单易懂
- 而协议和类的 extension 都会使用直接派发
- NSObject 的 extension 会使用消息机制进行派发
- NSObject 声明作用域里的函数都会使用函数表进行派发.
- 协议里声明的, 并且带有默认实现的函数会使用函数表进行派发

# Swift 运行时

<b>final</b>	直接派发
<b>dynamic</b>	消息机制派发
<b>@objc &amp; @nonobjc</b>	改变在OC里的可见性
<b>@inline</b>	告诉编译器可以直接派发



## Swift 运行时-final @objc

- 可以在标记为 final 的同时, 也使用 @objc 来让函数可以使用消息机制派发. 这么做的结果就是, 调用函数的时候会使用直接派发, 但也会在 Objective-C 的运行时里注册相应的 selector. 函数可以响应 perform(selector:) 以及别的 Objective-C 特性, 但在直接调用时又可以有直接派发的性能.

# Swift 运行时

```
class PureSwiftClass {  
    var boolValue: Bool = false  
    var age: Int = 0  
    var height: Float = 0  
    var name: String?  
    var exName: String?  
    func testPureAction() {  
        print("PureSwiftClass.testPureAction")  
    }  
}
```

```
class PureSwiftClass {  
    @objc var boolValue: Bool = false  
    @objc var age: Int = 0  
    @objc var height: Float = 0  
    @objc var name: String?  
    @objc var exName: String?  
    @objc func testPureAction() {  
        print("PureSwiftClass.testPureAction")  
    }  
}
```

# Swift 运行时

```
class MuixSwiftClass: UIViewController {
    var boolValue: Bool = false
    var age: Int = 0
    var height: Float = 0
    var name: String?
    var exName: String?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
    }

    func createSubView(view : UIView) {
        print("MuixSwiftClass.createSubView")
    }
}
```

```
class MuixSwiftClass: UIViewController {
    @objc var boolValue: Bool = false
    @objc var age: Int = 0
    @objc var height: Float = 0
    @objc var name: String?
    @objc var exName: String?

    override func viewDidLoad() {
        super.viewDidLoad()
    }

    override func viewWillAppear(_ animated: Bool) {
        super.viewWillAppear(animated)
    }

    @objc func createSubView(view : UIView) {
        print("MuixSwiftClass.createSubView")
    }
}
```

# Swift 运行时

- <https://github.com/apple/swift/blob/f4db1dd7a4abba2685247e1a7415d4fcb91f640d/stdlib/public/runtime/SwiftObject.h>
- SwiftObject

```
-----  
#endif  
SWIFT_RUNTIME_EXPORT @interface SwiftObject<NSObject> {  
    @private  
    Class isa;  
    SWIFT_HEAPOBJECT_NON_OBJC_MEMBERS;  
}  
  
- (BOOL)isEqual:(id)object;  
- (NSUInteger)hash;  
  
- (Class)superclass;  
- (Class)class;  
- (instancetype)self;  
- (struct _NSZone *)zone;  
  
- (id)performSelector:(SEL)aSelector;  
- (id)performSelector:(SEL)aSelector withObject:(id)object;  
- (id)performSelector:(SEL)aSelector withObject:(id)object1 withObject:(id)object2;  
  
- (BOOL)isProxy;  
  
+ (BOOL)isSubclassOfClass:(Class)aClass;  
- (BOOL)isKindOfClass:(Class)aClass;  
- (BOOL)isMemberOfClass:(Class)aClass;  
- (BOOL)conformsToProtocol:(Protocol *)aProtocol;  
  
- (BOOL)respondsToSelector:(SEL)aSelector;  
+ (BOOL)instancesRespondToSelector:(SEL)aSelector;  
- (IMP)methodForSelector:(SEL)aSelector;  
+ (IMP)instanceMethodForSelector:(SEL)aSelector;  
  
- (instancetype)retain;  
- (oneway void)release;  
- (instancetype)autorelease;  
- (NSUInteger)retainCount;
```

# 桥接

# 桥接

Choose options for your new file:

Class:

Subclass of:  ▼


☐ Also create XIB file

Language:  ⌵



# 桥接

Choose options



**Would you like to configure an Objective-C bridging header?**

Adding this file to swiftAndOC will create a mixed Swift and Objective-C target. Would you like Xcode to automatically configure a bridging header to enable classes to be accessed by both languages?

Cancel Don't Create Create Bridging Header

Class:

Subclass of:  ▼

☐ Also create XIB file


Language:  ⌵

Cancel Previous Finish


# 桥接

<

>



swiftAndOC



swiftAndOC

◇

General

Signing & Capabilities

Resource Tags

Info

Build Settings

Build Phases

Build Rules

Basic

Customized

All

Combined

Levels

+


Q

swift

×

▼ Packaging

Setting



swiftAndOC

Info.plist File

swiftAndOC/Info.plist

Product Bundle Identifier


com.geektime.swiftAndOC

Product Name

swiftAndOC

▼ Swift Compiler - Code Generation

Setting



swiftAndOC

▼ Optimization Level

<Multiple values> ◇

Debug


No Optimization [-Onone] ◇

Release

Optimize for Speed [-O] ◇

▼ Swift Compiler - General

Setting




swiftAndOC

Objective-C Bridging Header

swiftAndOC/swiftAndOC-Bridging-Header.h

▼ Swift Compiler - Language

Setting



swiftAndOC

Swift Language Version

Swift 5 ◇



# 桥接

A screenshot of a macOS Finder window titled "DerivedSources". The window features a standard macOS title bar with a blue folder icon and the text "DerivedSources". Below the title bar is a toolbar containing icons for view modes (icon view, compare view, web view, gallery view), a search field with the placeholder text "Search", and action buttons for share, print, and delete. The main content area displays a table of files with the following columns: "Name", "Date Modified", "Size", and "Kind". Three files are listed: "Entitlements-Simulated.plist" (384 bytes, Property List), "Entitlements.plist" (243 bytes, Property List), and "swiftAndOC-Swift.h" (10 KB, C Header Source File). The third file, "swiftAndOC-Swift.h", is selected and highlighted in blue. The rest of the window shows empty rows in the table.

# 相互调用

# Swift 调用 OC

```
< > swiftAndOC > swiftAndOC > swiftAndOC-Bridging-Header.h > No Selection
1 //
2 // Use this file to import your target's public hea
3 //
4
5 #import "MyViewController.h"
6
```

```
import UIKit

class ViewController: UIViewController {

    override func viewDidLoad() {
        super.viewDidLoad()
        title = "Swift 页面"
        view.backgroundColor = .white
        let button = UIButton(frame: CGRect(x: 100, y: 100, width: 100, height: 40))
        button.setTitle("点击跳转", for: .normal)
        button.setTitleColor(.blue, for: .normal)
        view.addSubview(button)
        button.addTarget(self, action: #selector(didClickButton), for: .touchUpInside)
    }

    @objc func didClickButton() {
        let myVC = MyViewController()
        navigationController?.pushViewController(myVC, animated: true)
    }
}
```

# OC 调用 Swift

```
import Foundation

class Person: NSObject {
    @objc var name: String
    @objc var age: Int

    @objc init(name: String, age: Int) {
        self.name = name
        self.age = age
        super.init()
    }
}
```

```
#import "MyViewController.h"
#import "swiftAndOC-Swift.h"

@interface MyViewController ()

@end

@implementation MyViewController

- (void)viewDidLoad {
    [super viewDidLoad];
    self.title = @"OC 页面";
    self.view.backgroundColor = UIColor.whiteColor;

    UILabel *label = [[UILabel alloc] initWithFrame:CGRectMake(100, 100, 100, 50)];
    label.text = @"OC 写的页面";
    label.textColor = UIColor.redColor;
    [self.view addSubview:label];

    Person *p = [[Person alloc] initWithName:@"zhangsan" age:20];
    NSLog(@"%@", p.name);
}
```

# OC 调用 Swift

```
swiftAndOC-Swift.h > No Selection
216
217 SWIFT_CLASS("_TtC10swiftAndOC6Person")
218 @interface Person : NSObject
219 @property (nonatomic, copy) NSString * _Nonnull name;
220 @property (nonatomic) NSInteger age;
221 - (nonnull instancetype)initWithName:(NSString * _Nonnull)name age:(NSInteger)age
    OBJC_DESIGNATED_INITIALIZER;
222 - (nonnull instancetype)init SWIFT_UNAVAILABLE;
223 + (nonnull instancetype)new SWIFT_UNAVAILABLE_MSG("-init is unavailable");
224 @end
225
226 @class UIWindow;
```

# NS\_SWIFT\_NAME

- 在 Objective-C 中，重新命名在swift中的名称。

# NS\_SWIFT\_UNAVAILABLE

- 在 Swift 中不可见，不能使用。



# 采坑指南



# Subclass

- 对于自定义的类而言，Objective-C 的类，不能继承自 Swift 的类，即要混编的 OC 类不能是 Swift 类的子类。反过来，需要混编的 Swift 类可以继承自 OC 的类。

# 宏

- 定义一个常量值，后面可以方便使用；如 `#define TOOLBAR_HEIGHT 44;`
- 定义一个不变化的常用值，或者一个较长的对象属性；如 `#define SCREEN_WIDTH ([[UIScreen mainScreen] bounds].size.width);`
- 定义一个会变化的常用变量值，或者一个较长的对象属性；如： `#define STATUS_BAR_HEIGHT ([UIApplication sharedApplication].statusBarFrame.size.height);`
- 定义一个带参数的宏，类似于一个函数；如 `#define RGB_COLOR(r,g,b) [UIColor colorWithRed:r/255.f green:g/255.f blue:b/255.f alpha:1.0]`

# 宏

- 第一种的话就比较简单，可以直接使用`let TOOLBAR_HEIGHT:CGFloat = 44`来替换就可以了；
- 第二种因为后面的值永远不会改变，也可以使用`let`来替换；可以用`let SCREEN_WIDTH = UIScreen.mainScreen().bounds.size.width`；
- 第三种情况，也就是后面的值会发生改变，如状态栏高度，就不能够使用`let`来替换了，因为`let`是定义的常量，如果使用`let`，将会导致不能够获取正确的值；这里可以使用函数来获取：`func STATUSBAR_HEIGHT() -> CGFloat { return UIApplication.sharedApplication().statusBarFrame.size.height }`；使用时通过函数`STATUSBAR_HEIGHT()`获取状态栏高度；
- 第四种，因为有输入参数，所以也只能使用函数来替换；如：`func RGB_COLOR(r:CGFloat, g:CGFloat, b:CGFloat) -> UIColor {return UIColor(red: r, green: g, blue: b, alpha: 1.0)}`；

# Swift 独有特性

- Swift 中有许多 OC 没有的特性，比如，Swift 有元组、为一等公民的函数、还有特有的枚举类型。所以，要使用的混编文件要注意 Swift 独有属性问题。

# NS\_REFINED\_FOR\_SWIFT

- Objective-C 的 API 和 Swift 的风格相差比较大，Swift 调用 Objective-C 的 API 时可能由于数据类型等不一致导致无法达到预期（比如，Objective-C 里的方法采用了 C 语言风格的多参数类型；或者 Objective-C 方法返回 `NSNotFound`，在 Swift 中期望返回 `nil`）。这时候就要 `NS_REFINED_FOR_SWIFT` 了。

```
@interface MyClass : NSObject  
  
- (NSUInteger)indexOfString:(NSString *)aString NS_REFINED_FOR_SWIFT;  
  
@end
```

```
extension MyClass {  
    func indexOfString(aString: String!) -> Int? {  
        let index = Int(__index(of: aString))  
        if (index == NSNotFound) {  
            return nil  
        }  
        return index  
    }  
}
```



扫码试看/订阅

《Swift核心技术与实战》视频课程