### runtime

1. 概述

runtime是OC语言的运行时核心支撑库，编译器和runtime的配合，让OC这门语言具备了强大的运行时动态性。

1. runtime有五大功能模块：对象模型，消息发送机制，动态特性，内存管理，系统交互
2. 对象模型：

基本数据结构：

struct objc\_object {

isa\_t isa;

}

struct objc\_class:objc\_object {

class superClass;

cache\_t cache;

class\_data\_bits\_t bits;

}

struct class\_rw\_t {

const class\_ro\_t \*ro;

method\_array\_t methods;

property\_array\_t properties;

protocol\_array\_t protocols;

}

struct class\_ro\_t {

method\_list\_t \*baseMethodList;

protocol\_list\_t \*baseProtocols;

const ivar\_list\_t \*ivars;

property\_list\_t \*baseProperties;

}

union isa\_t {

class cls;

unsigned long bits;

struct {

has\_assoc;

has\_cxx\_dtor;

shiftcls;

weakly\_referenced;

has\_sidetable\_rc;

extra\_rc;

}

}

1. 实例对象、类对象、元类对象

实例对象：程序运行时通过 alloc/init 等方式创建，存放isa和实例变量（ivars），isa指向其类对象；

类对象：由 runtime 在初始化阶段根据 Mach-O 中的类定义动态分配堆空间创建，一个类只有一个类对象。类对象通过其 class\_rw\_t/class\_ro\_t 记录了实例方法、属性、协议、实例变量等信息。

元类对象：每个类也有唯一的元类对象，存储类方法、类属性等。类对象的 isa 指向元类对象。

1. isa、superclass链条
2. isa 链：

实例对象的 isa → 类对象

类对象的 isa → 元类对象

元类对象的 isa → 根类 NSObject 的元类对象

根类 NSObject 的元类对象 isa → 根类 NSObject 的类对象（自环闭合）

1. superclass 链：

类对象的 superclass → 父类的类对象 → … → nil

元类对象的 superclass → 父类的元类对象 → … → 根类 NSObject 的元类对象 → 根类 NSObject 的类对象

1. isa存储的信息：

isa是个共用体，不同位域存储不同信息，主要存储的有：是否有关联对象，是否 有c++析构函数，类对象的内存地址，是否有弱引用，是否用sidetable存储引用 计数，引用计数值。引用计数先用extra\_rc存储，当不够用时额外的部分用sidetable存储。

1. class\_ro\_t和class\_rw\_t

类对象和元类对象都是objc\_class，其内部有个data指针指向class\_rw\_t，而class\_rw\_t内部有指针ro指向class\_ro\_t。

class\_rw\_t可读可写，由runtime初始化类对象和元类对象时创建，存放在堆上，用于保存动态扩展的信息，包括合并后的方法列表，协议列表，属性列表，其数据有两个来源：编译期分类文件中定义、程序运行时通过runtime api动态添加。

class\_ro\_t只读，由编译器在编译期生成，存放在macho文件\_\_data段的\_\_objc\_const区域，用于保存类的原始定义信息，包括类名，方法，协议，属性和实例变量列表，只读不可修改。

1. 消息机制
   1. 入口函数：obje\_msgsend(obj,selector,parameters)
   2. 查找流程：
      1. 根据对象类型，在cache\_t方法缓存中查找。实例对象则通过isa找到类对象，在类对象cache\_t中查找，类对象则去元类对象的cache\_t中查找
      2. 缓存未命中，则在类对象/元类对象的method\_list中查找
      3. 未找到，则沿着superclass链条向上查找，直到根类
      4. 一旦找到，就会写入cache\_t，加快后续调用速度。
   3. 动态方法解析：如果上一步没有找到，进入动态解析，实例方法调用+resolveInstanceMethod:,类方法调用+resolveClassMethod:。会去消息接收者的元类对象及其父类的元类对象的方法列表中查找resolveInstanceMethod或resolveClassMethod，如果未实现或者返回NO，则直接进入快速转发，如果实现了并返回true，则表示动态补充了方法的实现，然后重新进入消息查找流程，寻找目标方法。如果还是没找到，则进入快速转发。
   4. 快速转发：如果动态方法解析失败，则调用-forwardingTargetForSelector:，将消息发送给别的对象
   5. 标准转发：如果快速转发失败，则进入完整转发：
      1. 调用-methodSignatureForSelector:生成方法签名
      2. -forwardInvocation: 将消息封装为NSInvocation，交给开发者处理
   6. 抛异常，提示unreconginzed selector sended to xxxxxx
2. 动态特性
3. 内存管理
4. 系统交互

## **✅ 5. 动态特性（Dynamic Features）**

### **一句话定义**

Runtime 提供一套 C API，让我们在运行时动态修改类与对象的结构和行为。

### **典型例子**

**方法交换（Method Swizzling）**

核心 API：method\_exchangeImplementations

使用场景：AOP埋点、统计页面曝光、拦截系统方法。

示例：“我们可以交换 UIViewController 的 viewDidAppear 和自定义方法，实现自动打点。”

**动态添加方法**

核心 API：class\_addMethod

使用场景：消息转发补救，或者运行时扩展系统类。

示例：“在 resolveInstanceMethod 中为未实现的方法动态添加实现。”

**关联对象（Associated Object）**

核心 API：objc\_setAssociatedObject

场景：分类中模拟添加属性、状态缓存。

**KVO 实现原理**

动态创建一个子类，修改 isa 指针指向它。

在子类中重写 setter 并通知观察者。

### **延伸原理**

动态特性的底层都基于 class\_rw\_t 可写结构和 runtime 的方法表操作。

👉 **简短总结句**：

“Runtime 的动态特性本质上是操作类的可写结构体，通过修改方法、isa、关联表等方式实现运行时行为注入。”

## **✅ 6. 内存管理（Memory Management）**

### **一句话定义**

Runtime 负责 OC 内存管理的底层实现，包括引用计数、weak 表、AutoreleasePool。

### **典型例子**

**引用计数**

引用计数存储在对象的 isa.extra\_rc 或 SideTable.RefcountMap。

retain/release 操作是编译器插入、runtime 执行的。

**weak 弱引用表**

weak\_table\_t 存储对象 → 弱引用指针集合。

对象销毁时统一清零。

**AutoreleasePool**

基于 AutoreleasePage 实现。

由 RunLoop 在 beforeWaiting / afterWaiting 时 push/pop。

### **延伸原理**

ARC 只是编译器的语法糖，runtime 才是实际内存管理执行者。

所以 ARC 与 runtime 的关系是“编译期生成调用 → 运行时执行内存操作”。

👉 **简短总结句**：

“ARC 只负责插 retain/release，真正执行对象引用计数、weak 清理和 autoreleasepool 的，是 runtime。”

## **✅ 7. 系统交互（System Integration）**

### **一句话定义**

Runtime 不孤立存在，它与编译器（clang）、加载器（dyld）、RunLoop 紧密协作，完成 OC 程序从编译到运行的整个链路。

### **典型例子**

**与编译器 (clang)**

clang 把 [obj method] 翻译为 objc\_msgSend。

ARC 插入 objc\_retain / objc\_release。

**与 dyld**

Mach-O 中的 \_\_objc\_classlist、\_\_objc\_catlist。

dyld 加载可执行文件时，调用 runtime 注册所有类、分类、协议。

**与 RunLoop**

AutoreleasePool 的 push/pop 由 RunLoop 驱动。

### **延伸原理**

runtime 在 App 启动时由 dyld 触发初始化，读取 Mach-O 段数据构建类结构。

所以 OC 程序启动顺序是：clang 编译 → dyld 加载 → runtime 初始化 → main 函数执行。

👉 **简短总结句**：

“runtime 是连接编译器、dyld、RunLoop 的桥梁，支撑了整个 OC 世界的动态运行环境。”

1. 编译器在编译期将OC的类、分类、属性、方法，函数调用等符号按照runtime的格式编译为runtime能加载的结构，存放在macho文件的objc\_classlist等数据段。APP启动时，dylb将APP可执行文件加载到内存后，让runtime解析可执行文件，并在内存中建立类的运行时树状结构，至于函数调用，runtime提供了objc\_msgsend、objc\_msgsendsuper等函数，在运行时根据预设的状态值以及运行时外界临时改变的状态确定最终调用的函数是哪个。
2. 消息发送的流程
   1. 消息解析
   2. 消息转发
3. 反射机制，利用runtime的api，程序员可以获取类和对象的内存布局
4. 动态特性，利用runtime的api，程序员可以在运行时动态修改系统库或三方库的函数实现等，进行runtime层面的hook操作
5. kvo，系统库利用runtime的api，通过动态创建子类，修改原对象的类标记等方式，实现观察者模式
6. 在内存管理中，引用计数的保存是runtime预设的数据结构维护的，弱指针表也是runtime在维护，当然内存管理还要配合runloop来实现
7. 关联对象表的维护，由runtime负责，可以给已有的类附加自定义的属性
8. 为什么给nil发消息，程序部会崩溃，不会报错unrecongnized selector send to xxxxxx
9. OC方法的runtime内存结构如下，OC方法的三要素：选择子，方法签名，方法实现。选择子是全局唯一字符串，也就是方法名和冒号，方法签名是字符串，其信息是方法返回值和参数的类型，impl是方法实现指针，指向（macho text段的代码地址）。method\_t结构将selector和impl绑定，方法查找的时候，selector作为查找的凭证。

struct method\_t {

SEL name;

const char \*types;

IMP imp;

}