##### 内存管理

## **🧩 一、整体框架：从语言机制到底层实现**

要讲清「Objective-C 的内存管理」，核心逻辑是三层：

**语言层机制**：ARC 与 MRC 的基本规则；

**Runtime 层实现**：引用计数的存储与管理；

**系统层支撑**：AutoreleasePool、对象销毁、内存回收。

可以用一句话统摄全局：

Objective-C 的内存管理是以引用计数（Reference Counting）为核心，由编译器（ARC）在语言层自动插桩，由 Runtime 在运行时维护计数，由系统底层（AutoreleasePool + malloc zone）负责回收。

## **🧱 二、ARC 与 MRC：编译器插桩机制**

### **1. MRC（Manual Reference Counting）**

手动调用 retain / release / autorelease；

开发者需确保「谁创建谁释放」，遵循 **Ownership Policy（谁拥有谁管理）**；

引发了 retain cycle（循环引用）风险。

### **2. ARC（Automatic Reference Counting）**

ARC 不是垃圾回收器，而是编译器在编译阶段插入 retain/release；

自动管理强弱引用：

strong：持有对象；

weak：非持有引用，自动置 nil；

assign：非对象（或原始类型）；

unowned（Swift）：类似 weak 但不置 nil；

ARC 在编译时根据“作用域逃逸分析”自动生成 objc\_retain、objc\_release 调用。

📘 **关键点**：ARC 是编译期行为，真正管理引用计数的是 Runtime。

## **⚙️ 三、Runtime 层：引用计数的存储与实现**

### **1. 引用计数存在何处？**

对象的引用计数并非直接存储在 isa 指针或对象结构体中，而是由 Runtime 管理的：

早期版本：独立的散列表（Side Table）；

现代实现（Tagged Pointer 时代后）：

isa 指针采用 bitfield 压缩结构；

当计数可存储在低位（通常小于 19 位）时，直接嵌入 isa；

否则溢出到 Side Table 中。

📌 **Side Table 结构：**

struct SideTable {

spinlock\_t slock;

RefcountMap refcnts; // 引用计数字典

weak\_table\_t weak\_table; // 弱引用表

};

### **2. retain/release 底层实现**

objc\_retain(id obj) {

return obj->retain();

}

objc\_release(id obj) {

obj->release();

}

内部最终会调用 \_objc\_rootRetainCount、rootRelease 等函数，对 isa 或 SideTable 中的引用计数进行原子加减。

### **3. 引用计数归零的销毁流程**

当引用计数为 0 时：

dealloc -> \_objc\_rootDealloc -> objc\_destructInstance -> free()

调用 -[NSObject dealloc]；

自动释放所有 strong 成员；

调用 super dealloc；

最终调用 free() 回收内存。

## **🧂 四、AutoreleasePool：延迟释放机制**

### **1. 概念**

@autoreleasepool { ... } 是 ARC 下的延迟释放机制：

当对象被 autorelease 标记后，不立即调用 release；

由当前线程的 AutoreleasePool 持有，等作用域结束时统一释放。

### **2. 底层结构**

AutoreleasePool 是基于 **栈结构** 的：

\_\_AtAutoreleasePoolBegin() -> push()

\_\_AtAutoreleasePoolEnd() -> pop()

底层实现：

struct AutoreleasePoolPage {

void \*next;

id \*objects; // 存放 autorelease 对象的指针数组

}

### **3. 触发点**

RunLoop 每次进入休眠前（beforeWaiting）或退出前（exit）会自动 pop 一次 autoreleasepool，用以释放临时对象（例如事件响应、UI 更新等临时对象）。

## **🧠 五、Weak 引用表：自动置空机制**

weak 的关键是：**引用目标销毁时，能自动置 nil**。

### **1. 存储结构**

weak\_table\_t 是哈希表：

struct weak\_table\_t {

weak\_entry\_t \*entries; // key 为对象地址，value 为 weak 引用数组

};

### **2. 工作机制**

注册 weak：objc\_initWeak(&weakPtr, obj)

对象销毁：Runtime 在 dealloc 阶段调用 clearDeallocating()，遍历 weak\_table，将所有 weak 指针置为 nil。

## **🧹 六、Tagged Pointer：小对象优化**

某些小对象（如 NSNumber、NSDate 小值）不走 malloc，不参与引用计数；

直接把数据编码在指针中，内存管理无效化（retain/release 不产生实际效果）；

提升性能，避免频繁分配/释放。

## **🧩 七、面试总结答法模板**

你可以这样组织语言回答：

Objective-C 的内存管理本质是以引用计数为核心的对象生命周期控制机制。

在语言层，ARC 通过编译器自动插入 retain/release 调用；

在 Runtime 层，引用计数存储在 isa 指针或 SideTable 中，并通过原子操作更新；

当计数归零时，调用 dealloc 进行销毁；

AutoreleasePool 提供延迟释放机制，与 RunLoop 生命周期绑定；

Weak 引用通过全局 weak\_table 实现自动置 nil；

此外，小对象采用 Tagged Pointer 优化，避免频繁内存分配。

整体实现兼顾了性能、安全性与内存可控性。

## **✅ 八、延伸理解（进阶）**

objc\_autoreleaseReturnValue 优化：编译器与 Runtime 协作减少 retain/release 对；

NonPointerIsa 与 isa\_t bitfield；

SideTable 中的 spinlock 保证线程安全；

Swift 的 ARC 是在此机制上的语义拓展（同样基于引用计数，但多了 deinit、unowned 等语义）。

是否希望我帮你把这一套内容整理成「面试口述提纲 + 对应源码链路」？

比如每个点对应源码位置（objc-object.mm、NSObject.mm、AutoreleasePoolPage.mm），并附上关键函数调用链？这样你既能口述，又能对应源码理解。