iOS 动态调用实例方法的 5 种方式及场景选择

我将以清晰的结构把 “iOS 动态调用实例方法的 5 种方式” 整理成内容，包含核心原理、示例代码和场景建议，方便你后续导出为 PDF。以下是完整内容：

iOS 动态调用实例方法的 5 种方式

在 iOS 开发中，动态调用实例方法指不依赖编译期静态类型检查，通过 Objective-C Runtime 消息机制在运行时触发方法调用。以下是 5 种核心实现方式，适用于不同场景需求。

一、NSObject 自带的 performSelector: 系列方法（最简单）

核心原理

`NSObject` 封装了 Runtime 消息发送逻辑，提供简化的 API，无需手动处理底层细节，是最简单的动态调用方式。

常用 API

- `performSelector:`：调用无参实例方法
- `performSelector:withObject:`：调用带 1 个 `id` 类型参数的方法
- `performSelector:withObject:withObject:`：调用带 2 个 `id` 类型参数的方法
- `performSelector:withObject:afterDelay:`：延迟指定时间调用方法

示例代码（Objective-C）

```
// 假设已有实例对象 instance，需调用其 "handleClick:title:" 方法
SEL clickSEL = NSSelectorFromString(@"handleClick:title:");
// 先检查方法是否存在，避免崩溃
if ([instance respondsToSelector:clickSEL]) {
    // 调用带2个参数的方法（参数需为 id 类型，基本类型需包装为 NSNumber）
```

```
    [instance performSelector:clickSEL withObject:@"btn1" withObject:@"确认"];
}
```

特点

- **优势**：使用简单，无需导入 Runtime 头文件，安全性高（可提前检查方法存在性）。
- **局限**：最多支持 2 个参数，且参数必须是 `id` 类型（如 `int` 需包装为 `@(10)`）。

二、直接使用 Runtime 的 objc_msgSend 函数（最底层）

核心原理

`objc_msgSend` 是 Objective-C 消息发送的核心函数，直接向实例发送消息，触发方法调用，是所有动态调用的底层基础。

前置条件

需导入 Runtime 头文件：`#import <objc/runtime.h>`

示例代码（Objective-C）

```
// 调用实例方法 "calculateSum:a:b:c:"（返回 int，参数为3个 int 类型）
SEL sumSEL = NSSelectorFromString(@"calculateSum:a:b:c:");
// 强制类型转换：匹配方法签名（返回值+参数类型），避免编译器警告
int (*sumFunc)(id, SEL, int, int, int) = (int (*)(id, SEL, int, int, int))objc_msgSend;
// 调用方法并获取返回值
int result = sumFunc(instance, sumSEL, 10, 20, 30);
NSLog(@"计算结果：%d", result);
```

特点

- **优势**：支持任意参数类型和数量，灵活性最高，可直接控制底层消息逻辑。

- **局限**：需手动处理类型转换，参数不匹配会直接导致崩溃，代码可读性较差。

三、通过 NSInvocation 调用（最灵活）

核心原理

`NSInvocation` 是专门用于消息调用的封装类，可构建复杂的方法调用（支持任意参数类型和数量），适合参数较多或类型非 `id` 的场景。

示例代码（Objective-C）

```
// 1. 定义要调用的方法（示例：setUserInfo:age:isVip:，参数含字符串、int、BOOL）
SEL userSEL = NSSelectorFromString(@"setUserInfo:age:isVip:");
// 2. 获取方法签名（需确保实例和 SEL 有效）
NSMethodSignature *signature = [instance methodSignatureForSelector:userSEL];
if (!signature) {
    NSLog(@"方法签名无效");
    return;
}
// 3. 创建 NSInvocation 实例
NSInvocation *invocation = [NSInvocation invocationWithMethodSignature:signature];
invocation.target = instance;  // 指定调用目标实例
invocation.selector = userSEL;  // 指定调用的方法
// 4. 设置参数（注意：index 从 2 开始，0=instance，1=SEL）
NSString *info = @"iOS 开发者";
int age = 28;
BOOL isVip = YES;
[invocation setArgument:&info atIndex:2];  // 字符串参数
[invocation setArgument:&age atIndex:3];  // int 参数
[invocation setArgument:&isVip atIndex:4]; // BOOL 参数
// 5. 触发方法调用
[invocation invoke];
// 6. （可选）获取返回值（若方法有返回值）
NSString *returnStr;
[invocation getReturnValue:&returnStr];
NSLog(@"返回结果：%@", returnStr);
```

特点

- **优势**：支持任意参数类型（如 `int`、`float`、结构体）和数量，兼容性最强。
- **局限**：步骤繁琐，代码量较大，简单场景下效率较低。

四、通过 method_getImplementation 获取实现并调用（高性能）

核心原理

先通过 Runtime 函数 `class_getInstanceMethod` 获取方法对象（`Method`），再通过 `method_getImplementation` 拿到方法的函数指针（`IMP`），直接调用函数指针，减少消息发送的中间开销。

前置条件

需导入 Runtime 头文件：`#import <objc/runtime.h>`

示例代码（Objective-C）

```
// 1. 定义要调用的方法（示例：processData:，参数为 NSData，无返回值）
SEL dataSEL = NSSelectorFromString(@"processData:");
// 2. 获取方法对象（参数1：实例的类，参数2：方法 SEL）
Method dataMethod = class_getInstanceMethod([instance class], dataSEL);
if (!dataMethod) {
    NSLog(@"方法不存在");
    return;
}
// 3. 获取方法的函数指针（IMP）
IMP dataImp = method_getImplementation(dataMethod);
// 4. 定义函数指针类型并调用（匹配方法签名）
void (*processFunc)(id, SEL, NSData*) = (void (*)(id, SEL, NSData*))dataImp;
// 准备参数并调用
NSData *data = [@"test data" dataUsingEncoding:NSUTF8StringEncoding];
processFunc(instance, dataSEL, data);
```

特点

- **优势**：直接调用函数指针，避免消息发送的动态查找过程，性能略高于其他方式。
- **局限**：需严格匹配方法签名（返回值、参数类型），不匹配会导致内存错误。

五、Swift 中的动态调用（针对 Objective-C 兼容类）

核心原理

Swift 中需通过 `NSObject` 桥接，调用 Objective-C 类的实例方法（前提：Objective-C 类已标记 `@objc`，或 Swift 类继承 `NSObject` 并标记 `@objc` 方法）。

示例代码（Swift）

```
// 1. 动态获取 Objective-C 类并创建实例（假设类名为 "UserManager"）
guard let userManagerClass = NSClassFromString("UserManager") as? NSObject.Type else {
    print("类不存在或非 NSObject 子类")
    return
}
let userManager = userManagerClass.init()
// 2. 调用无参方法（示例："refreshUserInfo"）
let refreshSEL = NSSelectorFromString("refreshUserInfo")
if userManager.responds(to: refreshSEL) {
    userManager.perform(refreshSEL)
}
// 3. 调用带参数并返回值的方法（示例："getUserName:userId:"）
let getNameSEL = NSSelectorFromString("getUserName:userId:")
let userId: NSString = "123456"
// 调用方法并获取返回值（takeUnretainedValue() 避免内存泄漏）
if let userName = userManager.perform(getNameSEL, with: userId)?.takeUnretainedValue() as
? String {
    print("用户名称：\(userName)")
}
```

特点

- **优势**：适配 Swift 语法，无需手动处理 Runtime 细节，可直接调用 Objective-C 类方法。
- **局限**：对纯 Swift 类（未继承 `NSObject` 或未标记 `@objc`）不适用，因纯 Swift 方法默认不暴露给 Runtime。

六、场景选择建议

适用场景	推荐方式	核心优势
简单调用（≤2 个 `id` 类型参数）	`performSelector:` 系列	简洁、安全，无需依赖 Runtime 头文件
多参数 / 复杂类型（如结构体）	`NSInvocation`	支持任意参数，兼容性最强
性能敏感场景（如高频调用）	`method_getImplementation`	直接调用函数指针，效率最高
底层定制需求（如动态 hook）	`objc_msgSend`	最底层接口，灵活度最高
Swift 调用 Objective-C 类	`perform` 方法	适配 Swift 语法，桥接成本低

注意事项

1. **方法签名必须匹配**：所有动态调用方式均需已知方法名（`SEL`）和完整参数签名（返回值类型、参数类型 / 数量），否则会导致崩溃。
2. **提前检查方法存在性**：建议通过 `respondsToSelector:` 检查实例是否支持目标方法，减少崩溃风险。
3. **内存安全**：Swift 中调用时，需通过 `takeUnretainedValue()` 处理返回值，避免循环引用或内存泄漏。

你可以直接复制上述内容到 Word 或 Pages 中，再导出为 PDF 文件。如果需要我帮你整理成更贴合 PDF 排版的格式（如调整字体、增加目录），可以告诉我你的具体排版需求，我会进一步优化内容。

> （注：文档部分内容可能由 AI 生成）