数据采集系列白皮书

# iOS 全埋点技术白皮书

## iOS Autotrack Technology Whitepaper

S.R 用户行为洞察研究院

# 目录

01 埋点概述 01

02 应用程序启动和退出 04

03 页面浏览事件 09

04 控件点击事件 13

05 UITableView 和 UICollectionView 点击事件 17

06 采集手势 22

07 标识用户 26

08 时间相关 33

09 数据存储 37

10 数据同步 41

11 采集崩溃 44

12 App 与 H5 打通 48

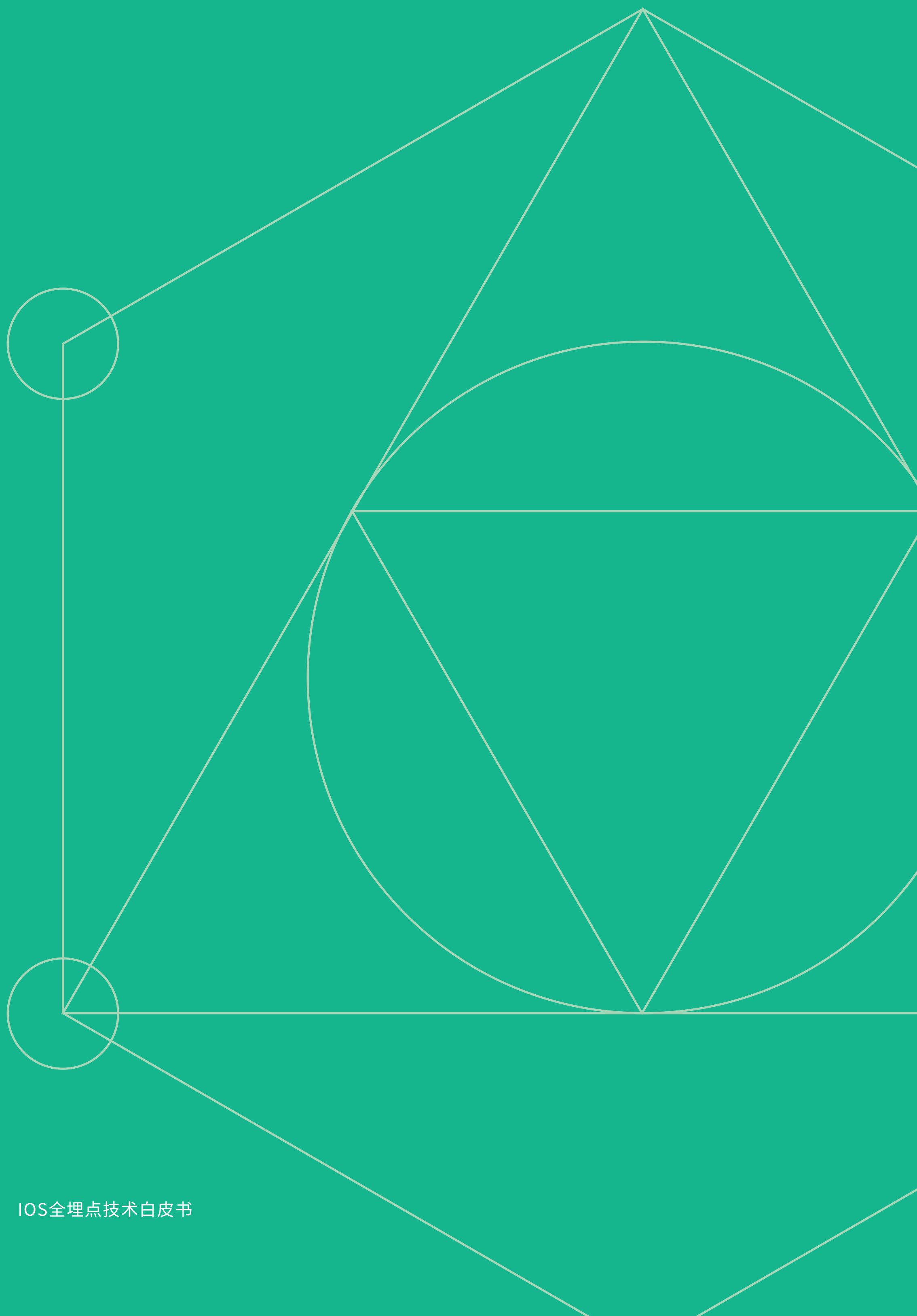
13 App Extension 54

14 React Native 全埋点 59

## 王灼洲

《Android 全埋点解决方案》一书作者，神策数据合肥研发中心负责人。有 10+ 年 Android & iOS 相关开发经验，是国内第一批从事 Android 研发工作，开发和维护国内第一个商用的开源 Android & iOS 数据埋点 SDK。

# 01
# 埋点概述



# 埋点概述

## 1、全埋点概述

数据分析的流程一般为：数据采集 → 数据传输 → 数据建模 → 数据统计 / 分析 / 挖掘 → 数据可视化 / 反馈，因此，数据采集是基本，是源头。

数据采集 SDK，一般需要包含两大基础功能：

- 通过埋点来采集数据
- 将采集的数据传输到指定的服务器端

不论是采集数据，还是传输数据，都会要求数据采集 SDK 要最大限度的保证数据的准确性、完整性和及时性，这就要求数据采集 SDK 要处理很多细节方面的问题，比如：用户标识、网络策略、缓存数据策略、同步数据策略、数据准确性和数据安全性等。

目前，业界主流的埋点方式，主要有如下三种：

- 代码埋点
- 全埋点
- 可视化埋点

代码埋点，指应用程序集成埋点 SDK 后，在启动的时候初始化埋点 SDK，然后在某个事件发生的时候调用埋点 SDK 提供的方法来触发事件。

代码埋点是“最原始”的埋点方式，同时也是“最万能”的埋点方式，这是因为它具有一系列的优点：

- 可以精准控制埋点的位置
- 可以更方便、更灵活地自定义事件和属性
- 可以采集更丰富的与业务相关的数据
- 可以满足更精细化的分析需求

当然，代码埋点也有相应的缺点：

- 前期埋点成本相对较高
- 若分析需求或事件设计发生变化，则需要应用程序修改埋点并发版

全埋点，也叫无埋点、无码埋点、无痕埋点、自动埋点等。全埋点，是指无需应用程序开发工程师写代码或者只写少量的代码，即可预先自动收集用户的所有或者绝大部分的行为数据，然后再根据实际的业务分析需求从中筛选出所需行为数据并进行分析。

全埋点目前可以采集的事件有：

- 应用程序启动事件（$AppStart）
- 应用程序退出事件（$AppEnd）
- 页面浏览事件（$AppViewScreen）
- 控件点击事件（$AppClick）
- 应用程序崩溃事件（$AppCrashed）

全埋点有如下几个优点：

- 前期埋点成本相对较低
- 若分析需求或事件设计发生变化，无需应用程序修改埋点并发版
- 可以有效地解决“历史数据回溯”问题

同时，全埋点也有一些缺点：

- 由于技术方面的原因，对于一些复杂的操作（比如缩放、滚动等），很难做到全面覆盖
- 无法自动采集和业务相关的数据
- 无法满足更精细化的分析需求

- 各种兼容性方面的问题（比如 Android 和 iOS 之间的兼容性，不同系统版本之间的兼容性，同一个系统版本不同 ROM 厂商之间的兼容性等）

可视化埋点，也叫圈选，是指通过可视化的方式进行埋点。

可视化埋点一般有两种应用场景：

- 默认情况下，不进行任何埋点，然后通过可视化的方式指定给哪些控件进行埋点（指定埋点）
- 默认情况下，全部进行埋点，然后通过可视化的方式指定哪些控件不进行埋点（排除埋点）

可视化埋点的优缺点，整体上与全埋点类似。

本白皮书主要以全埋点为核心进行介绍，部分内容也适用于代码埋点。

# 02
# 应用程序启动和退出

# 应用程序启动和退出

## 1、原理概述

iOS 应用程序常见的退出场景包括：

- 双击 Home 键切换到其它应用程序
- 按 Home 键让当前应用程序进入后台
- 双击 Home 键上滑强杀当前应用程序
- 当前应用程序发生崩溃导致应用程序退出

iOS 应用程序常见的启动场景包括：

- 冷启动：指应用程序被系统杀死后，在这种状态下启动的应用程序。
- 热启动：指应用程序没有被系统杀死，仍在后台运行，在这种状态下启动的应用程序。

在介绍应用程序退出（$AppEnd）和启动（$AppStart）事件的全埋点实现方案之前，我们先简单介绍一下 iOS 应用程序状态相关的内容。

（1）应用程序状态

大家都知道，对于一个标准的 iOS 应用程序来说，在不同的时期会有不同的状态，如下图 2-1 所示。

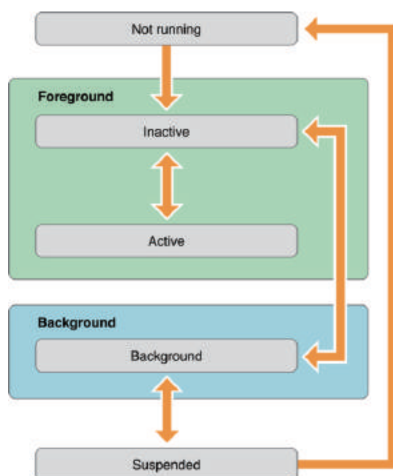



图 2-1 iOS 应用程序状态转换图

正常情况下，iOS 应用程序主要有五种常见的状态。

- Not running

非运行状态。指应用程序还没有被启动，或者已被系统终止。

- Inactive

前台非活动状态。指应用程序即将进入前台状态，但当前未接收到任何事件（它可能正在执行其它代码）。应用程序通常只在转换到其它状态时才会短暂地进入这个状态。

- Active

前台活跃状态。指应用程序正在前台运行，可接收事件并进行处理。这也是一个 iOS 应用程序处于前台的正常模式。

- Background

进入后台状态。指应用程序进入了后台并可执行代码。大多数应用程序在被挂起前都会短暂地进入这个状态。

- Suspended

挂起状态。指应用程序进入后台但没有执行任何代码，系统会自动的将应用程序转移到此状态，并且在执行此操作之前不会通知应用程序。挂起时，应用程序会保留在内存中，但不执行任何代码。当系统出现内存不足情况时，系统可能会在未通知应用程序的情况下清除被挂起的应用程序，为前台应用程序尽可能腾出更多的运行资源。

在应用程序不同状态转换的过程中，系统会回调实现了 UIApplicationDelegate 协议的类的一些方法，并发送相应的本地通知（系统会先回调相应的方法，待回调方法执行完后，再发送相应的通知），回调方法和本地通知的对应关系可参考如下表 2-1 所示。

| 回调方法 | 本地通知 |
| --- | --- |
| - applicationDidBecomeActive: | UIApplicationDidBecomeActiveNotification |
| - applicationDidEnterBackground: | UIApplicationDidEnterBackgroundNotification |
| - application:didFinishLaunchingWithOptions: | - application:didFinishLaunchingWithOptions: |
| - applicationWillEnterForeground: | - applicationWillEnterForeground: |
| - applicationWillResignActive: | - applicationWillResignActive: |
| - applicationWillTerminate: | - applicationWillTerminate: |

表 2-1 回调方法和本地通知对应关系

关于 iOS 应用程序状态更详细的内容，也可参考苹果官网文档介绍。

(2) 应用程序退出

通过上面介绍的内容可知，当一个 iOS 应用程序退出时，就意味着该应用程序进入了“后台”，即处于 Background 状态。因此，对于实现 $AppEnd 事件的全埋点，我们只需要注册监听 UIApplicationDidEnterBackgroundNotification 通知，然后在收到通知时触发 $AppEnd 事件，即可达到 $AppEnd 事件全埋点的效果。

(3) 应用程序启动

应用程序的启动，一般情况下，大致可以分为两类场景：

- 冷启动
- 热启动（从后台恢复）

不管是冷启动还是热启动，触发 $AppStart 事件的时机，都可以理解成是当“应用程序开始进入前台并处于活动状态”，也即前文介绍的 Active 状态。

因此，为了实现 $AppStart 事件的全埋点，我们可以注册监听 UIApplicationDidBecomeActiveNotification 本地通知，然后在其相应的回调方法里触发 $AppStart 事件。

通过测试可以发现，仍有以下几个特殊场景存在问题：

- 下拉通知栏并上滑，会触发 $AppStart 事件
- 上滑控制中心并下拉，会触发 $AppStart 事件
- 双击 Home 键进入切换应用程序页面，最后又选择当前应用程序，会触发 $AppStart 事件

以上几个场景均会触发 $AppStart 事件，明显与实际情况有所不符。

那这些现象是什么原因导致的呢?

我们继续分析可以发现以下几个现象：

- 下拉通知栏时，系统会发送 UIApplicationWillResignActiveNotification 通知；上滑通知栏时，系统会发送 UIApplicationDidBecomeActiveNotification 通知

• 上滑控制中心时，系统会发送 UIApplicationWillResignActiveNotification 通知；下拉控制中心时，系统会发送 UIApplicationDidBecomeActiveNotification 通知

• 双击 Home 键进入切换应用程序页面时，系统会发送 UIApplicationWillResignActiveNotification 通知，然后选择当前应用程序，系统会再发送 UIApplicationDidBecomeActiveNotification 通知

很容易总结出规律：在以上几个场景下，系统均是先发送 UIApplicationWillResignActiveNotification 通知，然后再发送 UIApplicationDidBecomeActiveNotification 通知。而我们又是通过注册监听

UIApplicationDidBecomeActiveNotification 通知来实现 $AppStart 事件全埋点，因此均会触发 $AppStart 事件。

那如何解决这个问题呢？

在解决这个问题之前，我们先看另一个现象：不管是冷启动还是热启动，系统均没有发送 UIApplicationWillResignActiveNotification 通知。

因此，只要在收到 UIApplicationDidBecomeActiveNotification 通知时，判断之前是否收到过 UIApplicationWillResignActiveNotification 通知，若没有收到，则触发 $AppStart 事件; 若已收到，则不触发 $AppStart 事件。这样即可解决上面的问题。

（4）被动启动

被动启动?什么意思?完全没有听说过!

你若有这些疑问，那就对了!因为这完全是我们神策数据自创的一个名词。

在 iOS 7 之后，苹果新增了后台应用程序刷新功能，这个功能允许操作系统在一定的时间间隔内（这个时间间隔根据用户不同的操作习惯而有所不同，可能是几个小时，也可能是几天），拉起应用程序并同时让其进入后台运行，以便应用程序可以获取最新的数据并更新相关内容，从而可以确保用户在打开应用程序的时候可以第一时间查看到最新的内容。例如新闻或者社交媒体类型的应用程序，可以使用这个功能在后台获取到最新的数据内容，在用户打开应用程序时可以缩短应用程序启动和获取内容展示的等待时间，最终提升产品的用户体验。

后台应用程序刷新，对于用户来说可以缩短等待时间；对于产品来说，可以提升用户体验；但对于数据采集 SDK 来说，可能会带来一系列的问题，比如：当系统拉起应用程序（会触发 $AppStart 事件）并同时让其进入后台运行时，应用程序的第一个页面（UIViewController）也会被加载，也即会触发一次页面浏览事件（$AppViewScreen 事件），这明显是不合理的，因为用户并没有打开应用程序，更没有浏览第一个页面。其实，整个后台应用程序刷新的过程，对于用户而言，完全是透明的、无感知的。

因此，在实际的数据采集过程中，我们需要避免这种情况的发生，以免影响到正常的数据分析。

我们把由 iOS 系统触发的应用程序自动进入后台运行的启动，称之为（应用程序的）被动启动，使用 $AppStartPassively 事件来表示。

后台应用程序刷新是最常见的造成被动启动的原因之一。而后台应用程序刷新只是其中一种后台运行模式，还有一些其它后台运行模式同样也会触发被动启动，我们下面会详细介绍。

使用 Xcode 创建新的应用程序，默认情况下后台刷新功能是关闭的，我们可以在 Capabilities 标签中开启 Background Modes，然后就可以勾选所需要的功能了，如下图 2-2 所示

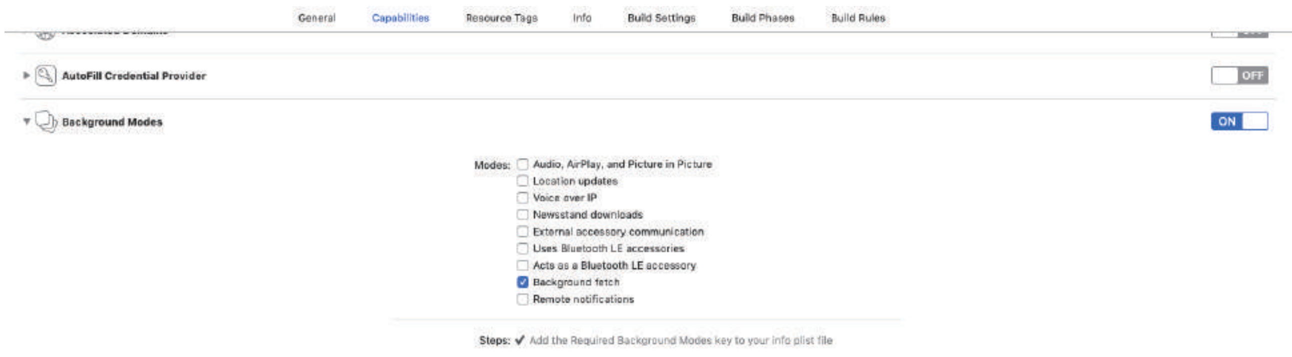



图 2-2 开启 Background Modes

通过上图 2-2 我们可以看到，还有如下几种后台运行模式，它们同样也会导致触发被动启动（$AppStartPassively 事件）：

- Location updates：此模式下，会由于地理位置变化而触发应用程序启动
- Newsstand downloads：这种模式只针对报刊杂志类应用程序，当有新的报刊可下载时，会触发应用程序启动
- External Accessory communication：此模式下，一些 MFi 外设通过蓝牙或者 Lightning 接头等方式与 iOS 设备连接，从而可在外设给应用程序发送消息时，触发对应的应用程序启动
- Uses Bluetooth LE accessories：此模式与 External Accessory communication 类似，只是无需限制 MFi 外设，而需要的是 Bluetooth LE 设备
- Acts as a Bluetooth LE accessory：此模式下，iPhone 作为一个蓝牙外设连接，可以触发应用程序启动
- Background fetch：此模式下，iOS 系统会在一定的时间间隔内触发应用程序启动，去获取应用程序数据
- Remote notifications：此模式是支持静默推送，当应用程序收到这种推送后，不会有任何界面提示，但会触发应用程序启动

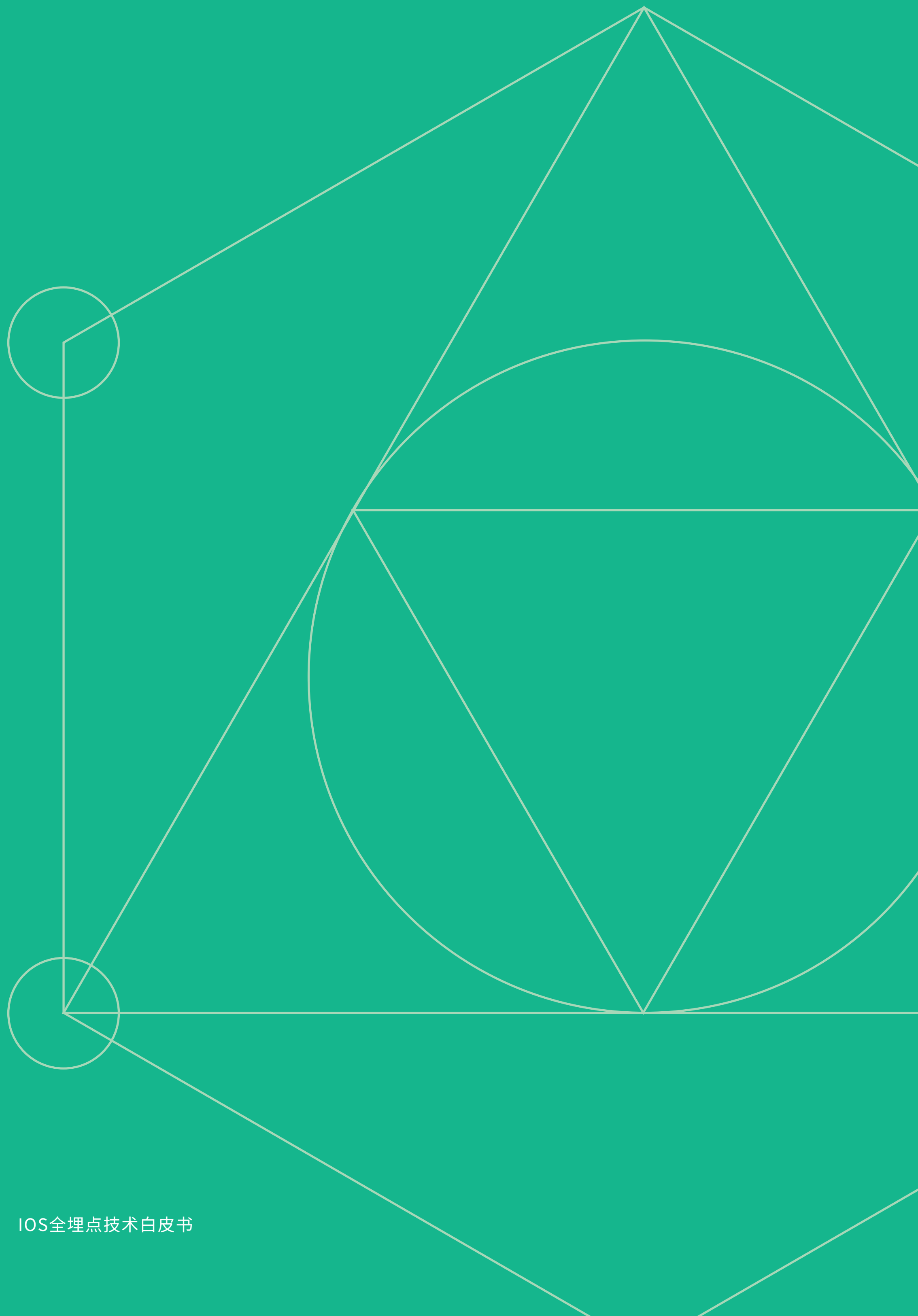
# 2、实现步骤

完整的项目源码后续会 release 给大家。

# 3、知识点

- UIApplicationDelegate
- Background App Refresh

# 03
# 页面浏览事件



# 页面浏览事件

## 1、原理概述

本章主要介绍页面浏览事件（$AppViewScreen）全埋点的实现原理。在介绍具体的实现原理之前，我们先介绍 UIViewController 生命周期相关的内容，然后再介绍 iOS 的“黑魔法”Method Swizzling。

（1）UIViewController 生命周期

众所周知，每一个 UIViewController 都管理着一个由多个视图组成的树形结构，其中根视图保存在 UIViewController 的 view 属性中。UIViewController 会懒加载它所管理的视图集，直到第一次访问 view 属性时，才会去加载或者创建 UIViewController 的视图集。

有以下几种常用的方式加载或者创建 UIViewController 的视图集：

•使用 Storyboard

•使用 Nib 文件

•使用代码，即重写 - loadView

以上这些方法，最终都会创建出合适的根视图并保存在 UIViewController 的 view 属性中，这是 UIViewController 生命周期的第一步。当 UIViewController 的根视图需要展示在页面上时，会调用 - viewDidLoad 方法。在这个方法中，我们可以做一些对象初始化相关的工作。

需要注意的是：此时，视图的 bounds 还没有确定。对于使用代码创建视图，- viewDidLoad 方法会在 -loadView 方法调用结束之后运行；如果使用的是 Stroyboard 或者 Nib 文件创建视图，- viewDidLoad 方法则会在 - awakeFromNib 方法之后调用。

当 UIViewController 的视图在屏幕上的显示状态发生变化时，UIViewController 会自动回调一些方法，确保子类能够响应到这些变化。如下图 3-1 所示，它展示了 UIViewController 在不同的显示状态时会回调不同的方法。

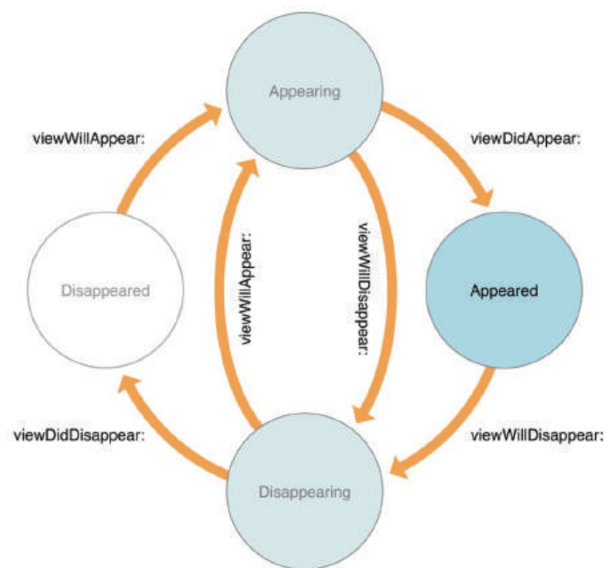



图 3-1 UIViewController 不同状态的方法调用

在 UIViewController 被销毁之前，还会回调 - dealloc 方法，我们一般通过重写这个方法来主动释放不能被 ARC 自动释放的资源。

我们现在对 UIViewController 的整个生命周期有了一些基本了解。那么，我们如何去实现页面浏览事件（$AppViewScreen 事件）的全埋点呢？

通过 UIViewController 的生命周期可知，当执行到 - viewDidAppear: 方法时，表示视图已经在屏幕上渲染完成，也即页面已经显示出来了，正等待用户进行下一步操作。因此，- viewDidAppear: 方法就是我们触发页面浏览事件的最佳时机。如果想要实现页面浏览事件的全埋点，需要使用 iOS 的“黑魔法”Method Swizzling 相关的技术。

下面，我们先介绍 Method Swizzling。

（2）Method Swizzling

Method Swizzling，顾名思义，就是交换两个方法的实现。简单的来说，就是利用 Objective-C runtime 的动态绑定特性，把一个方法的实现与另一个方法的实现进行交换。

在 Objective-C 的 runtime 中，一个类是用一个名为 objc_class 的结构体表示的，它的定义如下：

```
struct objc_class {
    Class _Nonnull isa  OBJC_ISA_AVAILABILITY;
 #if !__OBJC2__
    Class _Nullable super_class
OBJC2_UNAVAILABLE;
    const char * _Nonnull name
OBJC2_UNAVAILABLE;
    long version                                OBJC2_UNAVAIL-
ABLE;
    long info                                   OBJC2_UNAVAIL-
ABLE;
    long instance_size
OBJC2_UNAVAILABLE;
    struct objc_ivar_list * _Nullable ivars
OBJC2_UNAVAILABLE;
    struct objc_method_list * _Nullable * _Nullable
methodLists             OBJC2_UNAVAILABLE;
    struct objc_cache * _Nonnull cache
OBJC2_UNAVAILABLE;
    struct objc_protocol_list * _Nullable protocols
OBJC2_UNAVAILABLE;
#endif
 } OBJC2_UNAVAILABLE;
```

在上面的结构体中，虽然有很多字段在 OBJC2 中已经废弃了（OBJC2_UNAVAILABLE），但是了解这个结构体还是有助于我们理解 Method Swizzling 的底层原理。我们从上述结构体中可以发现，有一个 objc_method_list 指针，它保存着当前类的所有方法列表。同时，objc_method_list 也是一个结构体，它的定义如下：

```
struct objc_method_list {
    struct objc_method_list * _Nullable obsolete
OBJC2_UNAVAILABLE;


    int method_count
OBJC2_UNAVAILABLE;
#ifdef __LP64__
    int space                                   OBJC2_UNAVAIL-
ABLE;
#endif
    /* variable length structure */
    struct objc_method method_list[1]
 OBJC2_UNAVAILABLE;
}
```

在上面的结构体中，有一个 objc_method 字段，我们再来看看 objc_method 这个结构体：

```
struct objc_method {
    SEL _Nonnull method_name
 OBJC2_UNAVAILABLE;
    char * _Nullable method_types
OBJC2_UNAVAILABLE;
    IMP _Nonnull method_imp
OBJC2_UNAVAILABLE;
}
```

从上面的结构体中可以看出，一个方法由下面三个部分组成：

- method_name：方法名
- method_types：方法类型
- method_imp：方法实现

使用 Method Swizzling 交换方法，其实就是修改了 objc_method 结构体中的 method_imp，也即改变了 method_name 和 method_imp 的映射关系，如下图 3-2 所示。

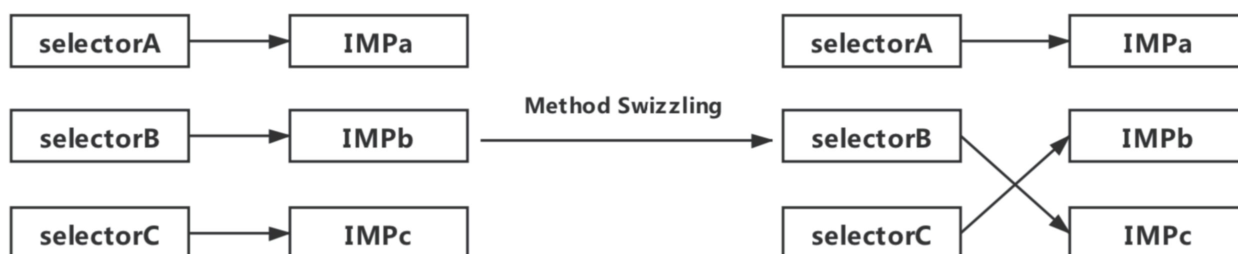



图 3-2 method_name 和 method_imp 映射关系

那我们如何改变 method_name 和 method_imp 的映射关系呢?在 Objective-C 的 runtime 中，提供了很多非常方便使用的函数，让我们可以很简单的就能实现 Method Swizzling，即改变 method_name 和 method_imp 的映射关系，从而达到交换方法的效果。

（3）实现页面浏览事件全埋点

通过对 UIViewController 生命周期和 Method Swizzling 的学习，我们就可以通过 Method Swizzling 来交换 UIViewController 的 - viewDidAppear: 方法，然后在交换的方法中触发 $AppViewScreen 事件，从而就可以实现页面浏览事件的全埋点了。

# 2、实现步骤

完整的项目源码后续会 release 给大家。

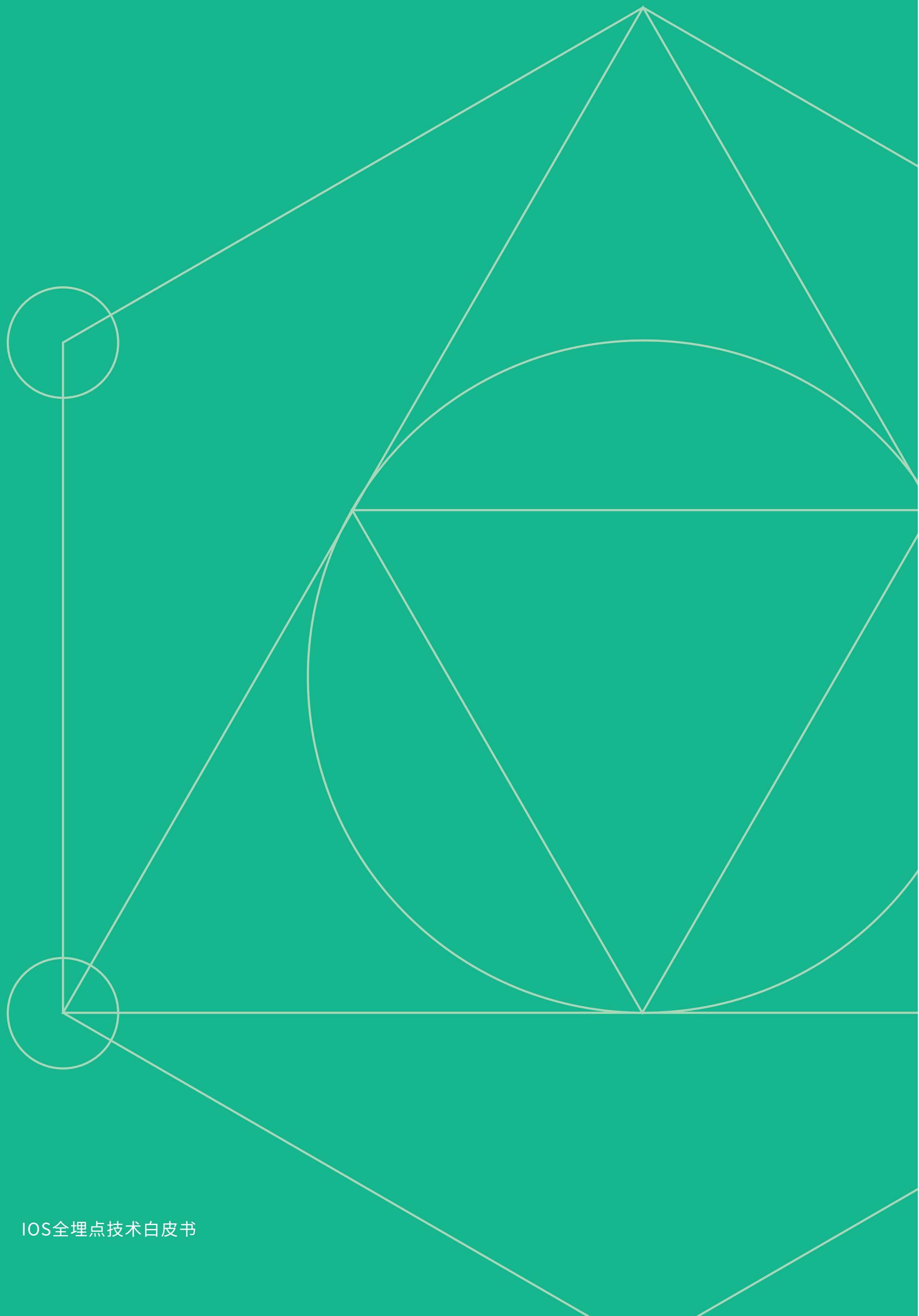
# 3、缺点

- 应用程序热启动时（从后台恢复），第一个页面没有触发 $AppViewScreen 事件。这是由于这个页面没有再次执行 - viewDidAppear: 方法导致的。
- 要求 UIViewController 的子类要么不重写 - viewDidAppear: 方法，一旦重写必须调用 [super viewDidAppear:animated]，否则也不会触发 $AppViewScreen 事件。这是由于我们是直接交换 UIViewController 的 - viewDidAppear: 方法导致的。

# 4、知识点

- UIViewController State Transitions
- Method Swizzling

# 04
# 控件点击事件



# 控件点击事件

## 1、原理概述

在本章，我们主要介绍如何实现控件点击事件（$AppClick）的全埋点。在介绍如何实现之前，我们需要先了解一下，在 UIKit 框架下，处理点击或拖动事件的 Target-Action 设计模式。

（1）Target-Action

Target-Action，也叫目标 - 动作模式，即当某个事件发生的时候，调用特定对象的特定方法。

比如，在 LoginViewController 页面，有一个按钮，点击按钮时，会调用 LoginViewController 里的 -loginBtnOnClick 方法，“特定对象”就是 Target，“特定方法”就是 Action。也即 Target 是 LoginViewController，Action 是 - loginBtnOnClick 方法。

Target-Action 设计模式主要包含两个部分：

- Target 对象：接收消息的对象
- Action 方法：用于表示需要调用的方法

Target 对象可以是任意类型的对象。但是在 iOS 应用程序中，通常情况下会是一个控制器，而触发事件的对象和 Target 对象一样，也可以是任意对象。例如，手势识别器 UIGestureRecognizer 就可以在识别到手势后，将消息发送给另一个对象。Target-Action 设计模式，最常见的应用场景还是在控件中。iOS 中的控件都是 UIControl 类或者其子类，当用户在操作这些控件时，会将消息发送到指定的对象（Target），而对应的 Action 方法必须符合以下几种形式之一：

```
- (void)doSomething;

- (void)doSomething:(id)sender;

- (void)doSomething:(id)sender forEvent:(UIEvent *)event;

- (IBAction)doSomething;

- (IBAction)doSomething:(id)sender;

- (IBAction)doSomething:(id)sender forEvent:(UIEvent *)event;
```

其中以 IBAction 作为返回值类型的形式，是为了让该方法能在 Interface Builder 中被看到；sender 参数就是触发事件的控件本身；第二个参数 event 是 UIEvent 的对象，封装了触摸事件的相关信息。

我们可以通过代码或者 Interface Builder 为一个控件添加一个 Target 对象以及相对应的 Action 方法。

若想使用代码方式添加 Target-Action（我们也会用 Target-Action 表示：一个 Target 对象以及相对应的 Action 方法），可以直接调用控件对象的如下方法：

```
- (void)addTarget:(nullable id)target action:(SEL)action forControlEvents:(UIControlEvents)controlEvents;
```

我们也可以多次调用 - addTarget:action:forControlEvents: 方法给控件添加多个 Target-Action，即使多次调用 - addTarget:action:forControlEvents: 添加相同的 Target 但是不同的 Action，也不会出现相互覆盖的问题。另外，在添加 Target-Action 的时候，Target 对象也可以为 nil（默认会先在 self 里查找 Action）。

当我们为一个控件添加 Target-Action 后，控件又是如何找到 Target 对象并执行对应的 Action 方法的呢？

在 UIControl 类中有一个方法：

```
- (void)sendAction:(SEL)action to:(nullable id)target forEvent:(nullable UIEvent *)event;
```

如果控件被用户操作（比如点击），首先会调用这个方法，并将事件转发给应用程序的 UIApplication 对象。

同时，在 UIApplication 类中也有一个类似的实例方法：

```
- (BOOL)sendAction:(SEL)action to:(nullable id)target from:(nullable id)sender forEvent:(nullable UIEvent *)event;
```

如果 Target 对象不为 nil，应用程序会让该 Target 对象调用对应的 Action 方法响应事件；如果 Target 对象为 nil，应用程序会在响应者链中搜索定义了该方法的对象，然后执行 Action 方法。

基于 Target-Action 设计模式，我们有两种方案可以实现 $AppClick 事件的全埋点。

下面，我们逐一进行介绍。

(2) 方案一

通过 Target-Action 执行模式可知，在执行 Action 方法之前，会先后通过控件和 UIApplication 对象发送事件相关的信息。因此，我们可以通过 Method Swizzling 交换 UIApplication 的 - sendAction:to:from:forEvent: 方法，然后在交换后的方法中触发 $AppClick 事件，并根据 target 和 sender 采集相关的属性，即可实现 $AppClick 事件的全埋点 。

对于 UIApplication 类中的 - sendAction:to:from:forEvent: 方法，我们以给 UIButton 设置 action 为例，详细介绍一下。

```
[button addTarget:person action:@selector(btnAction) forControlEvents:UIControlEventTouchUpInside];
```

参数：

- action：Action 方法对应的 selector，即示例中的 btnAction。
- target：Target 对象，即示例中的 person。如果 Target 为 nil，应用程序会将消息发送给第一个响应者，并从第一个响应者沿着响应链向上发送消息，直到消息被处理为止。
- ender：被用户点击或拖动的控件，即发送 Action 消息的对象，即示例中的 button。
- event：UIEvent 对象，它封装了触发事件的相关信息。

返回值：

如果有 responder 对象处理了此消息，返回 YES，否则返回 NO。

一般情况下，对于一个控件的点击事件，我们至少还需要采集如下信息（属性）：

- 控件类型（$element_type）
- 控件上显示的文本（$element_content）
- 控件所属页面，即 UIViewController（$screen_name）

基于目前的方案，我们来看如何实现采集以上三个属性。

获取控件类型

获取控件类型相对比较简单，我们可以直接使用控件的 class 名称来代表当前控件的类型，比如可通过如下方式获取控件的 class 名称：

```
NSString *elementType = NSStringFromClass([sender class])
```

获取显示文本

获取控件上的显示文本，我们只需要针对特定的控件，调用其相应的方法获取文本即可。

获取控件所属页面

如何知道一个 UIView 所属哪个 UIViewController 呢？

这就需要借助 UIResponder 了！

大家都知道，UIResponder 类是 iOS 应用程序中专门用来响应用户操作事件的，比如

- Touch Events：即触摸事件
- Motion Events：即运动事件
- Remote Control Events：即远程控制事件

UIApplication、UIViewController、UIView 类都是 UIResponder 的子类，所以它们都具有响应以上事件的能力。另外，自定义的 UIView 和自定义视图控制器也都可以响应以上事件。在 iOS 应用程序中，UIApplication、UIViewController、UIView 类的对象也都是一个

个响应者，这些响应者会形成一个响应者链。一个完整的响应者链传递规则（顺序）大概如下：UIView → UIViewController → RootViewController → Window → UIApplication → UIApplicationDelegate，可参考下图 4-1 所示（此图来源于苹果官方网站）。

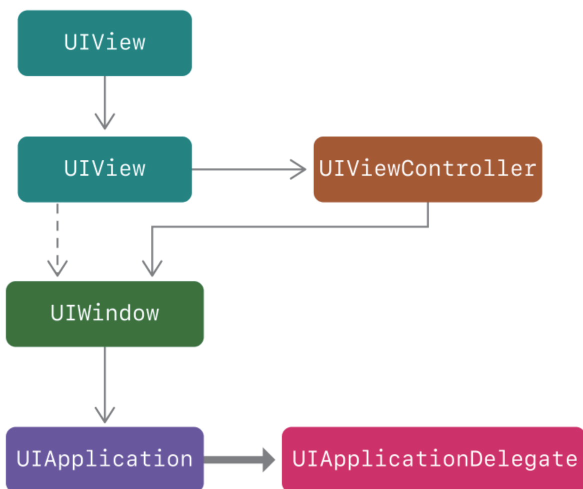



图 4-1 事件响应者链

注意：对于 iOS 应用程序里实现了 UIApplicationDelegate 协议的类（通常为 AppDelegate），如果它是继承自 UIResponder，那么也会参与响应者链的传递；如果不是继承自 UIResponder（例如 NSObject），那么它就不会参与响应者链的传递。

通过图 4-1 可以知道，对于任意一个视图来说，都能通过响应者链找到它所在的视图控制器，也就是其所属的页面，从而可以达到获取它所属页面信息的目的。

（3）方案二

当一个视图被添加到父视图上时，系统会自动调用它的 -didMoveToSuperview 方法。因此，我们可以通过 Method Swizzling 交换 UIView 的 - didMoveToSuperview 方法，然后在交换方法里，给控件添加一组UIControlEventTouchDown 类型的 Target-Action，并在 Action 里触发 $AppClick 事件，从而达到 $AppClick 事件全埋点的效果，这就是我们的方案二。

（4）方案总结

方案一和方案二，其实都是运用了 iOS 中的 Target-Action 模式。这两种方案，各有优缺点。

对于方案一：如果给一个控件添加了多个 Target-Action 方法，会导致多次触发 $AppClick 事件。

对于方案二：由于我们 SDK 为控件添加了一个默认触发类型的 Action 方法，因此，如果开发者在开发过程中使用 UIControl 类的 allTargets 或者 allControlEvents 属性进行一些逻辑判断，有可能会引入无法预料的问题。

因此，在选择方案的时候，可以根据自己的实际情况和需求，来确定最终的实现方案。

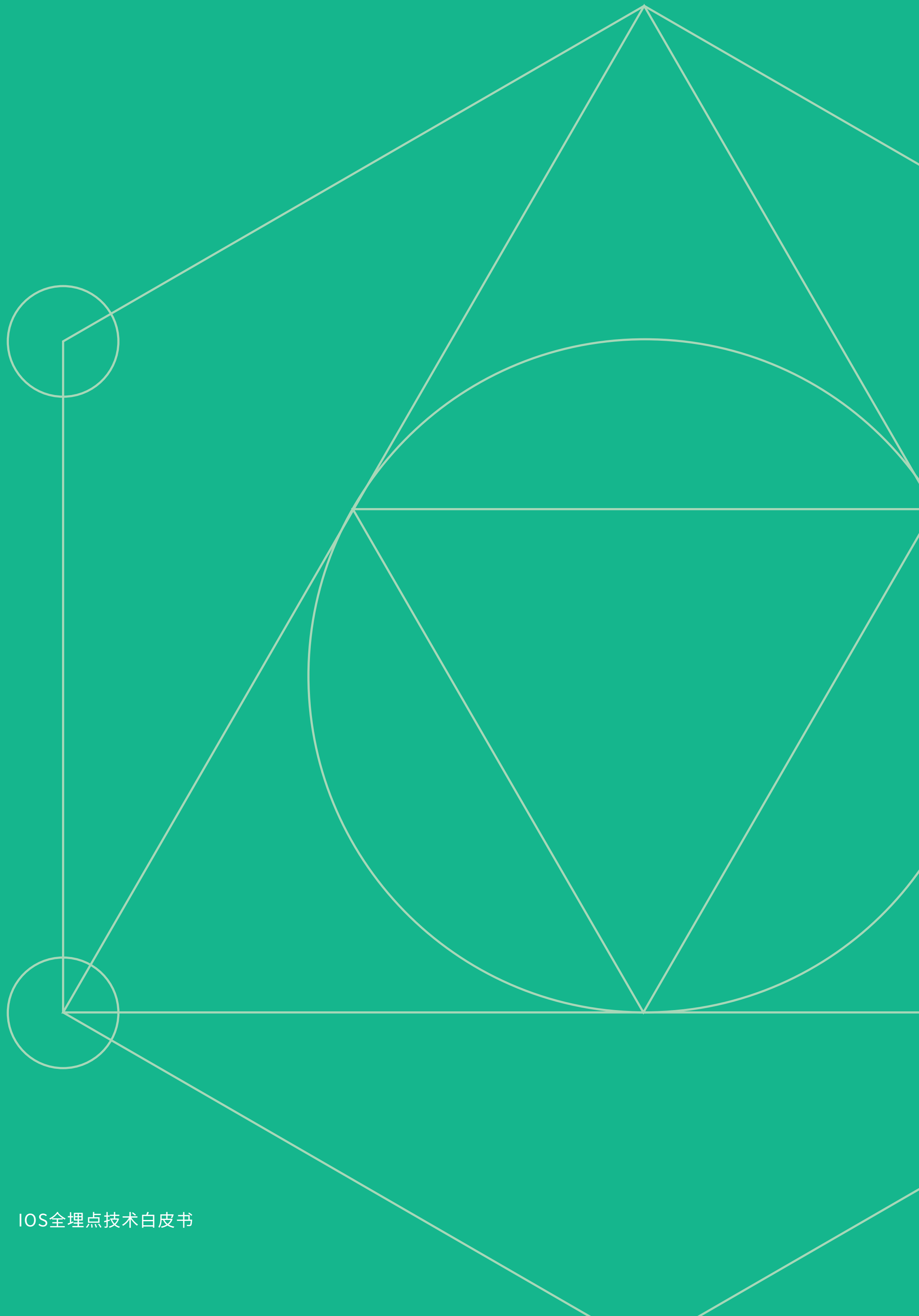
## 2、实现步骤

完整的项目源码后续会 release 给大家。

## 3、知识点

- Target-Action
- Responder Chain
- UIControlEvents

# 05
# UITableView 和 UICollectionView 点击事件



# UITableView 和 UICollectionView 点击事件

## 1、原理概述

在 $AppClick 事件采集中，还有两个比较特殊的控件：

- UITableView
- UICollectionView

这两个控件的点击事件，一般指的是点击 UITableViewCell 和 UICollectionViewCell。而 UITableViewCell 和 UICollectionViewCell 都是直接继承自 UIView 类，而不是 UIControl 类。因此，我们之前实现 $AppClick 事件全埋点的两个方案均不适用于 UITableView 和 UICollectionView。

关于实现 UITableView 和 UICollectionView $AppClick 事件的全埋点，常见的方案有三种：

- 方法交换
- 动态子类
- 消息转发

这三种方案，各有优缺点。

下面，我们以 UITableView 控件为例，来分别介绍如何使用这三种方案实现 $AppClick 事件的全埋点。

（1）方案一：方法交换

众所周知，如果需要处理 UITableView 的点击操作，需要先设置 UITableView 的 delegate 属性，并实现 UITableViewDelegate 协议的 - tableView:didSelectRowAtIndexPath: 方法。因此，我们也很容易就想到使用 Method Swizzling 来交换 - tableView:didSelectRowAtIndexPath: 方法来实现 UITableView $AppClick 事件的全埋点。

大概思路：首先，我们使用 Method Swizzling 交换 UITableView 的 - setDelegate: 方法，然后能获取到实现了 UITableViewDelegate 协议的 delegate 对象，在拿到 delegate 对象之后，就可以交换 delegate 对象的 - tableView:didSelectRowAtIndexPath: 方法，最后，在交换后的方法中触发 $AppClick 事件，从而达到全埋点的效果。

（2）方案二：动态子类

动态子类的方案，就是在运行时，给实现了 UITableViewDelegate 协议的 - tableView:didSelectRowAtIndexPath: 方法的类创建一个子类，让这个类的对象变成我们自己创建的子类的对象。同时，还需要在创建的子类中动态添加 - tableView:didSelectRowAtIndexPath: 方法。那么，当用户点击 UITableViewCell 时，就会先运行我们创建的子类中的 - tableView:didSelectRowAtIndexPath: 方法。然后，我们在实现这个方法的时候，先调用 delegate 原来的方法实现再触发 $AppClick 事件，即可达到全埋点的效果。

（3）方案三：消息转发

在介绍方案三之前，我们先介绍一下 NSProxy 这个类。

在 iOS 应用开发中，自定义一个类的时候，一般都需要继承自 NSObject 类或者 NSObject 的子类。但是 NSProxy 类却并不是继承自 NSObject 类或者 NSObject 的子类，NSProxy 是一个实现了 NSObject 协议的抽象基类。

```
/*  NSProxy.h
    Copyright (c) 1994-2019, Apple Inc. All rights reserved.
*/

#import <Foundation/NSObject.h>

@class NSMethodSignature, NSInvocation;

NS_ASSUME_NONNULL_BEGIN

NS_ROOT_CLASS
@interface NSProxy <NSObject> {
    Class   isa;
}

+ (id)alloc;
+ (id)allocWithZone:(nullable NSZone *)zone NS_AUTOMATED_REFCOUNT_UNAVAILABLE;
+ (Class)class;

- (void)forwardInvocation:(NSInvocation *)invocation;
- (nullable NSMethodSignature *)methodSignatureForSelector:(SEL)sel NS_SWIFT_UNAVAILABLE("NSInvocation
and related APIs not available");
- (void)dealloc;
- (void)finalize;
@property (readonly, copy) NSString *description;
@property (readonly, copy) NSString *debugDescription;
+ (BOOL)respondsToSelector:(SEL)aSelector;

- (BOOL)allowsWeakReference API_UNAVAILABLE(macos, ios, watchos, tvos);
- (BOOL)retainWeakReference API_UNAVAILABLE(macos, ios, watchos, tvos);

// - (id)forwardingTargetForSelector:(SEL)aSelector;

@end

NS_ASSUME_NONNULL_END
```

从 NSProxy 的名字可以看出，这个类的作用就是作为一个委托代理对象，将消息转发给一个真实的对象或者自己加载的对象。

当然，在大部分情况下，使用 NSObject 类也可以实现消息转发，实现方式与 NSProxy 类相同。

但是，大部分情况下使用 NSProxy 类更为合适，有以下三个理由：

a. NSProxy 类作为一个抽象基类，其自身能够处理的方法很少，仅是实现了 NSObject 协议中的方法，而 NSObject 类则更复杂

b. 通过 NSObject 类实现的代理类不会自动转发 NSObject 协议中的方法

c. 通过 NSObject 类实现的代理类不会自动转发 NSObject 类别中的方法

通过使用消息转发机制可以实现 UITableView 的 $AppClick 事件全埋点。

（4）三种方案的总结

对于 UITableView 的 $AppClick 事件全埋点的三种方案，它们各有优缺点，我们可以根据实际情况选择相应的方案。

方案一：方法交换

优点：简单、易理解；Method Swizzling 属于成熟技术，性能相对来说较高。

缺点：会对原始类有入侵，容易造成冲突。

方案二：动态子类

优点：没有对原始类入侵，不会修改原始类的方法，不会和第三方库冲突，是一种比较稳定的方案。

缺点：动态创建子类对性能和内存有比较大的消耗。

方案三：消息转发

优点：充分利用消息转发机制，对消息进行拦截，性能较好。

缺点：容易与一些同样使用消息转发进行拦截的第三方库冲突，例如 ReactiveCocoa。

（5）优化

在前面介绍的三种方案中，我们只是采集了与普通控件的 $AppClick 事件相同的一些简单属性。

在实际的业务分析需求中，针对 UITableView 的 $AppClick 事件，我们一般还需要采集如下属性：

- 被点击 UITableViewCell 控件上的显示内容（$element_content）
- 被点击 UITableViewCell 控件所在的位置（$element_position）

我们下面对于 UITableView 的 $AppClick 事件全埋点方案进行优化，使其支持获取 $element_content 和 $element_position 属性。

获取控件内容

要想获取 UITableViewCell 控件上显示的文本，首先必须要拿到被点击的那个 UITableViewCell 对象。

在 - tableView:didSelectRowAtIndexPath: 回调方法中有两个参数：

- tableView：UITableView
- indexPath：NSIndexPath

由于用户当时点击的 UITableViewCell 控件一定是显示在屏幕上的。因此，可以通过以下方式获取 UITableViewCell 控件对象。

```
UITableViewCell *cell = [tableView cellForRowAtIndexPath:indexPath];
```

拿到 UITableViewCell 对象之后，接下来就是获取上面显示的文本。但 UITableViewCell 控件其实不是一个简单的控件，它本身也并不显示任何内容，它更像一个容器，在其上面添加了一些子控件用于展示内容。我们可以递归遍历它所有的子控件，依次获取每一个子控件的显示内容，并按一定格式进行拼接，从而可以将拼接的内容作为 UITableViewCell 的显示内容。

获取 UITableViewCell 的位置

获取用户点击 UITableViewCell 控件所在的位置（$element_position），需要借助 -tableView:didSelectRowAtIndexPath: 方法的第二个参数 indexPath。从 indexPath 对象中，我们可以获取用户点击 UITableViewCell 的 Section 和 Row。

(6) 支持 UICollectionView

通过上面的章节，我们实现了 UITableView 的 $AppClick 事件全埋点。对于 UICollectionView 的 $AppClick 事件全埋点方案，整体上跟 UITableView 控件类似，同样可以使用以上三种方案去实现。

# 2、实现步骤

完整的项目源码后续会 release 给大家。

# 3、知识点

- UITableViewDelegate
- NSProxy

# 06
# 采集手势

# 采集手势

## 1、原理概述

（1）手势识别器

苹果为了降低开发者在手势事件处理方面的开发难度，定义了一个抽象类 UIGestureRecognizer 来协助开发者。UIGestureRecognizer 是具体手势识别器的抽象基类，它定义了一组可以为所有具体手势识别器配置的常见行为。它还可以通过设置委托（即实现了 UIGestureRecognizerDelegate 协议的对象），来支持对某些行为进行更细粒度的定制。

手势识别器必须被添加在一个特定的视图上（比如 UILabel、UIImageView 等控件），即需要通过调用 UIView 类中的 - addGestureRecognizer: 方法进行添加。手势识别器也是用了 Target-Action 设计模式。当我们为一个手势识别器添加一个或者多个 Target-Action 后，在视图上进行触摸操作时，一旦系统识别了该手势，就会向所有的 Target 对象发送消息，并执行 Action 方法。虽然手势识别器和 UIControl 类一样，都是使用了 Target-Action 设计模式，但是手势识别器并不会将消息交由 UIApplication 对象来进行发送。因此，我们无法使用与 UIControl 控件相同的处理方式，即通过响应者链的方式来实现对手势操作的全埋点。

由于 UIGestureRecognizer 是一个抽象基类，所以它并不会处理具体的手势。因此，对于轻拍（UITapGestureRecognizer）、长按（UILongPressGestureRecognizer）等具体的手势触摸事件，需要使用相应的子类即具体的手势识别器进行处理。

常见的具体手势识别器有：

- UITapGestureRecognizer：轻拍手势
- UILongPressGestureRecognizer：长按手势
- UIPinchGestureRecognizer：捏合（缩放）手势
- UIRotationGestureRecognizer：旋转手势
- UISwipeGestureRecognizer：轻扫手势
- UIPanGestureRecognizer：平移手势
- UIScreenEdgePanGestureRecognizer：屏幕边缘平移手势

给上面所有的具体手势识别器添加 Target-Action 的方法都是相同的，常见的主要是通过以下的两个方法进行添加。

- - initWithTarget:target action:
- - addTarget:action:

详细的定义参考如下：

```
/**
 指定初始化方法

 通过添加一个 Target-Action 进行初始化，
 当初始化的手势识别器对象，识别到触摸手势时，会向 Target 对象发送消息，即调用 Action 方法

 @param target 需要发送消息的 Target 对象
 @param action 向 Target 对象发送的消息，即方法名
 @return 初始化的对象
 */
- (instancetype)initWithTarget:(nullable id)target action:(nullable SEL)action NS_DESIGNATED_INITIALIZER;

/**
 向一个手势识别器添加一个 Target-Action

 可以多次调用此方法，给一个手势识别器对象添加多个 Target-Action 。
 如果已经添加了一个 Target-Action，再次添加相同的 Target-Action 时，会被忽略。

 @param target 需要发送消息的 Target 对象
 @param action 向 Target 对象发送的消息，即方法名
 */
- (void)addTarget:(id)target action:(SEL)action;
```

在实际的开发过程中，使用比较多的是 UITapGestureRecognizer 和 UILongPressGestureRecognizer 两个手势识别器，这两个手势识别器分别是处理轻拍手势和长按手势。

（2）手势全埋点

在数据采集中，一般只需要采集常见控件（UILabel、UIImageView）的轻拍和长按手势。

接下来，基于以上内容，我们分别介绍如何实现控件轻拍和长按手势的全埋点。

UITapGestureRecognizer 全埋点

为了采集控件的轻拍手势，我们可以通过 Method Swizzling 交换 UITapGestureRecognizer 类的添加 Target-Action 的方法，从而可以添加一个新的 Target-Action，并在新添加的 Action 方法中触发 $AppClick 事件，从而就可以达到采集控件轻拍手势全埋点的效果。

在 UITapGestureRecognizer 类中，用于添加 Target-Action 方法有两个：

- - initWithTarget:action:
- - addTarget:action:

因此，我们需要对这两个方法分别进行交换。

UILongPressGestureRecognizer 全埋点

对于 UILongPressGestureRecognizer 来说，其实现逻辑与 UITapGestureRecognizer 基本上是相同的。

（3）总结

至此，我们就实现了手势 $AppClick 事件的全埋点方案，包括了两种手势识别器：

- UITapGestureRecognizer
- UILongPressGestureRecognizer

对于其它手势识别器的 $AppClick 事件全埋点方案，实现思路也都是相同的。由于其它的手势在实际应用中使用的比较少，因此这里就不再赘述了。如果大家有实际的需求，可以按照 UITapGestureRecognizer 和 UILongPressGestureRecognizer 的实现方案自行扩展。

# 2、实现步骤

完整的项目源码后续会 release 给大家。

# 3、知识点

- UIGestureRecognizer

# 07
# 标识用户

# 标识用户

## 1、原理概述

分析用户行为，首先需要标识用户。选取合适的用户标识，对于提高用户行为分析的准确性有非常大的影响，尤其是对漏斗、留存、Session 等和用户相关的分析功能。

在事件中，我们可以新增一个 distinct_id 字段，来标识是哪个用户触发的事件，比如：

```
{
  "event": "$AppClick",
  "time": 1575337589670,
  "distinct_id": "D8E4354E-C18A-44BB-BC75-548BD67C56E5",
  "properties": {
    "$model": "x86_64",
    "$manufacturer": "Apple",
    "$element_type": "UIButton",
    "$lib_version": "1.0.0",
    "$os": "iOS",
    "$element_content": "Button",
    "$app_version": "1.0",
    "$screen_name": "ViewController",
    "$os_version": "13.2.2",
    "$lib": "iOS"
  }
}
```

提示：在数据分析里，用户是指事件发生的主体，不一定就是指使用终端的人，也可以是一个企业、商家，甚至是一辆汽车，需要根据具体的业务场景而定。

对于唯一标识一个用户，需要考虑两种场景：

- 用户登录之前如何标识？
- 用户登录之后如何标识？

下面，我们分别进行介绍。

（1）登录之前

对于登录之前的用户，我们可以努力去唯一标识 ta 当前正在使用的 iOS 设备。业界一般使用 iOS 设备的某个特定属性或者某几个特定属性组合的方式，来唯一标识一台 iOS 设备。此时的用户 ID，我们一般称之为设备 ID 或匿名 ID。对于究竟该如何去唯一标识一台 iOS 设备，目前业界也没有一个非常完美或普适的方案。同时，苹果为了维护整个生态系统的健康发展，也会极力阻止个人或者组织去唯一标识一台 iOS 设备。因此，对于如何唯一标识一台 iOS 设备，将会是一场持久战，更是一个多方博弈的过程。可能我们唯一能做的，就是在现有的条件及政策之下，尝试努力寻找一种最优的解决方案。

下面，我们介绍几个常见的可以考虑用来作为 iOS 设备 ID 的属性。

A.UDID

UDID 的英文全称是 Unique Device Identifier，是设备唯一标识符的缩写。从名称我们可以猜测到，UDID 是和设备相关的，而且只跟设备相关。它是一个由 40 位十六进制组成的序列。

在 iOS 5 之前，可以通过如下代码片段获取当前设备的 UDID

```
NSString *udid = [[UIDevice currentDevice] uniqueIdentifier];:
```

返回的 UDID 示例如下：

```
fc8c9322aeb3b4042c85fda3bc12953896da88f6
```

但从 iOS 5 开始，苹果为了保护用户隐私，就不再支持通过以上方式获取 UDID。

结论：由于从 iOS 5 开始，苹果禁止 iOS 应用程序通过代码获取 UDID，因此 UDID 不适合作为 iOS 设备 ID。

B．UUID

UUID 的英文全称是 Universally Unique Identifier，是通用唯一标识符的缩写。它是一个由 32 位十六进制组成的序列，使用小横线来连接，格式为：8-4-4-4-12（数字代表位数，加上 4 个横线，总共是 36 位），如下示例。

UUID 示例：

```
D8E4354E-C18A-44BB-BC75-548BD67C56E5
```

通过 UUID，能让你在任何一个时刻，在不借助任何服务器的情况下生成唯一标识符，也就是说，UUID 在某一特定的时空下是全球唯一的。

从 iOS 6 开始，iOS 应用程序可以通过 NSUUID 类来获取 UUID，如下代码片段。

```
NSString *uuid = [NSUUID UUID].UUIDString;
```

生成的 UUID，系统不会做持久化存储，因此每次调用的时候都会获得一个全新的 UUID。

如果需要兼容更老版本的 iOS 系统，也可以使用 CFUUID 类来获取 UUID。CFUUID 是 CoreFoundation 框架的一部分，因此 API 都是 C 语言风格，代码片段参考如下。

```
CFUUIDRef cfuuidRef = CFUUIDCreate(kCFAllocatorDefault);
NSString *uuid = (NSString*)CFBridgingRelease(CFUUIDCreateString(kCFAllocatorDefault, cfuuidRef));
```

提示：NSUUID 和 CFUUID 的功能完全一样，只不过 NSUUID 是 Objective-C 接口，而 CFUUID 是 C 语言风格的 API。

结论：由于每次获取 UUID 时，返回的都是一个全新的 UUID，如果用户删除应用程序并再次安装时，将无法做到唯一标识 iOS 设备。因此 UUID 也不适合作为 iOS 设备 ID。

C．MAC 地址

MAC 地址是用来标识互联网上的每一个站点，它是一个由 12 位十六进制组成的序列，如下示例。

```
C4:B3:01:BD:42:B1
```

凡是接入网络的设备都会有一个 MAC 地址，用来区分每个设备的唯一性。一个 iOS 设备（一般指 iPhone 手机）可能会有多个 MAC 地址，这是因为它可能会有多个设备接入网络，比如 Wi-Fi、SIM 卡等。一般情况下，我们只需要获取 Wi-Fi 的 MAC 地址即可，即 en0 的地址。

但从 iOS 7 开始，苹果禁止 iOS 应用程序获取 MAC 地址。如果 iOS 应用程序继续获取 MAC 地址，系统将会返回一个固定的 MAC 地址 02:00:00:00:00:00，这也是因为 MAC 地址和 UDID 一样，都属于隐私信息。

结论：从 iOS 7 开始，苹果禁止 iOS 应用程序获取 MAC 地址，因此 MAC 地址也不适合作为 iOS 设备 ID。

D．IDFA

IDFA 的英文全称是 Identifier For Advertising，是广告标识符的缩写，主要用于广告推广、换量等跨应用的设备追踪等。它也是一个由 32 位十六进制组成的序列，格式与 UUID 一致。在同一个 iOS 设备上，同一时刻，所有的应用程序获取到的 IDFA 都是相同的。

从 iOS 6 开始，我们可以利用 AdSupport.framework 库提供的方法来获取 IDFA，代码片段参考如下。

```
#import <AdSupport/AdSupport.h>
 NSString *idfa = [[[ASIdentifierManager sharedManager] advertisingIdentifier] UUIDString];
```

返回 IDFA 示例：

```
FB584D10-FFC4-40D8-A3AF-DDC77B60462B
```

但 IDFA 的值也并不是固定不变的。

目前，以下操作均会改变 IDFA 的值：

- 通过设置 → 通用 → 还原 → 抹掉所有内容和设置，如图 7-1 所示
- 通过 iTunes 还原设备
- 通过设置 → 隐私 → 广告 → 限制广告追踪，如图 7-2 所示

一旦用户限制了广告追踪，我们获取到的 IDFA 将是一个固定的 IDFA，即一连串的零：00000000-0000-0000-0000-000000000000。

因此，在获取 IDFA 之前，我们可以利用 AdSupport.framework 库提供的接口来判断用户是否限制了广告追踪。

```
BOOL isLimitAdTracking = [[ASIdentifierManager sharedManager] isAdvertisingTrackingEnabled];
```

一旦用户还原了广告标识符，系统将会生成一个全新的 IDFA，如下图 7-2 所示。

图 7-1 抹掉所有内容和设置

图 7-2 限制广告追踪和还原广告标识符

结论：虽然 IDFA 有一些限制条件，但对于这些操作，只会在特定的情况下才会出现，或者只有专业人士才有可能做这些操作。同时，IDFA 也能解决应用程序卸载重装唯一标识设备的问题。因此，IDFA 目前来说比较适合作为 iOS 设备 ID。

E.IDFV

IDFV 英文全称是 Identifier For Vendor，是应用开发商标识符的缩写，是给 Vendor 标识用户使用的，主要适用于分析用户在应用内的行为等。它也是一个由 32 位十六进制组成的序列，格式也与 UUID 一致。

每个 iOS 设备在所属同一个 Vendor 的应用里，获取到的 IDFV 是相同的。Vendor 是通过反转后的 BundleID 的前两部分进行匹配的，如果相同就属于同一个 Vendor。比如，对 于 com.apple.example1 和 com.apple.example2 这两个 BundleID 来说，它们就属于同一个 Vendor，将共享同一个 IDFV。和 IDFA 相比，IDFV 不会出现获取不到的场景。

但 IDFV 也有一个很大的缺点：如果用户将属于此 Vendor 的所有应用程序都卸载，则 IDFV 的值将会被系统重置。即使后面重装此 Vendor 的应用程序，获取到的也将是一个全新的 IDFV。

另外，以下操作也会重置 IDFV：

- 通过设置 → 通用 → 还原 → 抹掉所有内容和设置，如上图 7-1 所示
- 通过 iTunes 还原设备
- 卸载设备上某个开发者账号下的所有应用程序

在 iOS 应用程序内，可以通过 UIDevice 类来获取 IDFV，可参考如下代码片段。

```
NSString *idfv = [[[UIDevice currentDevice] identifierForVendor] UUIDString];
```

返回的 IDFV 示例如下：

18B4587A-0A6F-44A0-AD3C-3BB1C490C177

结论：和 IDFA 相比，特别是在解决应用程序卸载重装的问题上，IDFV 不太适合作为 iOS 设备 ID。

F. IMEI

IMEI 的 英 文 全 称 是 International Mobile Equipment Identity，

是国际移动设备身份码的缩写，它是由 15 位纯数字组成的串，并且是全球唯一的。任何一部手机，在其生产并组装完成之后，都会被写入一个全球唯一的 IMEI。我们可以通过设置 → 通用 → 关于本机查看本机 IMEI，如下图 7-3 所示。

图 7-3 查看本机 IMEI

结论：从 iOS 2 开始，苹果提供了相应的 API 来获取 IMEI。但后来为了保护用户隐私，从 iOS 5 开始，苹果就不再允许应用程序获取 IMEI。因此，IMEI 也不适合作为 iOS 设备 ID 。

G．最佳实践

上面介绍的属性，它们各有优缺点，都不是非常完美的方案。

它们总的来说有以下两个问题：

- 无法保证唯一性
- 会受到各种相关政策的限制

关于设备 ID，到底有没有一种完美的方案呢?目前确实没有!我们只能在现有的条件和限制之下，寻找一种相对比较完美的方案。

结合实际情况来看，对于常规数据分析中的设备 ID，可按照如下优先级顺序获取，基本上能满足我们的业务需求。

IDFA → IDFV → UUID

对于设备 ID，不管是使用 IDFA 还是 IDFV，用户限制广告追踪或应用程序卸载重装，都有可能导致发生变化。那我们是否还有更好的方案呢?

是有的，那就是 Keychain。

什么是 Keychain？

Keychain 是 OS X 和 iOS 都提供的一种安全存储敏感信息的工具。比如，我们可以在 Keychain 中存储用户名、密码等信息。Keychain 的安全机制是从系统层面保证了存储的这些敏感信息不会被非法读取或者窃取。

Keychain 的特点：

- 保存在 Keychain 中的数据，即使应用程序被卸载了，数据仍然存在；重新安装应用程序，可以从 Keychain 中读取到这些数据
- Keychain 中的数据可以通过 Group 的方式实现应用程序之间共享，只要应用程序具有相同的 TeamID 即可
- 保存在 Keychain 中的数据都是经过加密的，因此非常安全

关于 Keychain 的详细用法，我们在此处不再详细描述，详细说明可以参照苹果官方文档 https://developer.apple.com/documentation/security/keychain_services?language=objc。

（2）登录之后

对登录之后的用户进行标识，相对来说比较简单。因为用户一旦在你的产品中进行了注册或登录，那 ta 在你的用户系统里肯定就是唯一的了。此时的用户 ID 我们一般称之为登录 ID。登录 ID 通常是业务数据库里的主键或其它唯一标识。因此，登录 ID 相对来说会更精确、更稳定，同时，也具有唯一性。

我们可以提供一个 - login: 方法，当应用程序拿到用户的登录 ID 之后，通过调用 - login: 方法把登录 ID 传给 SDK，在此之后触发的事件，即可使用登录 ID 来标识用户。

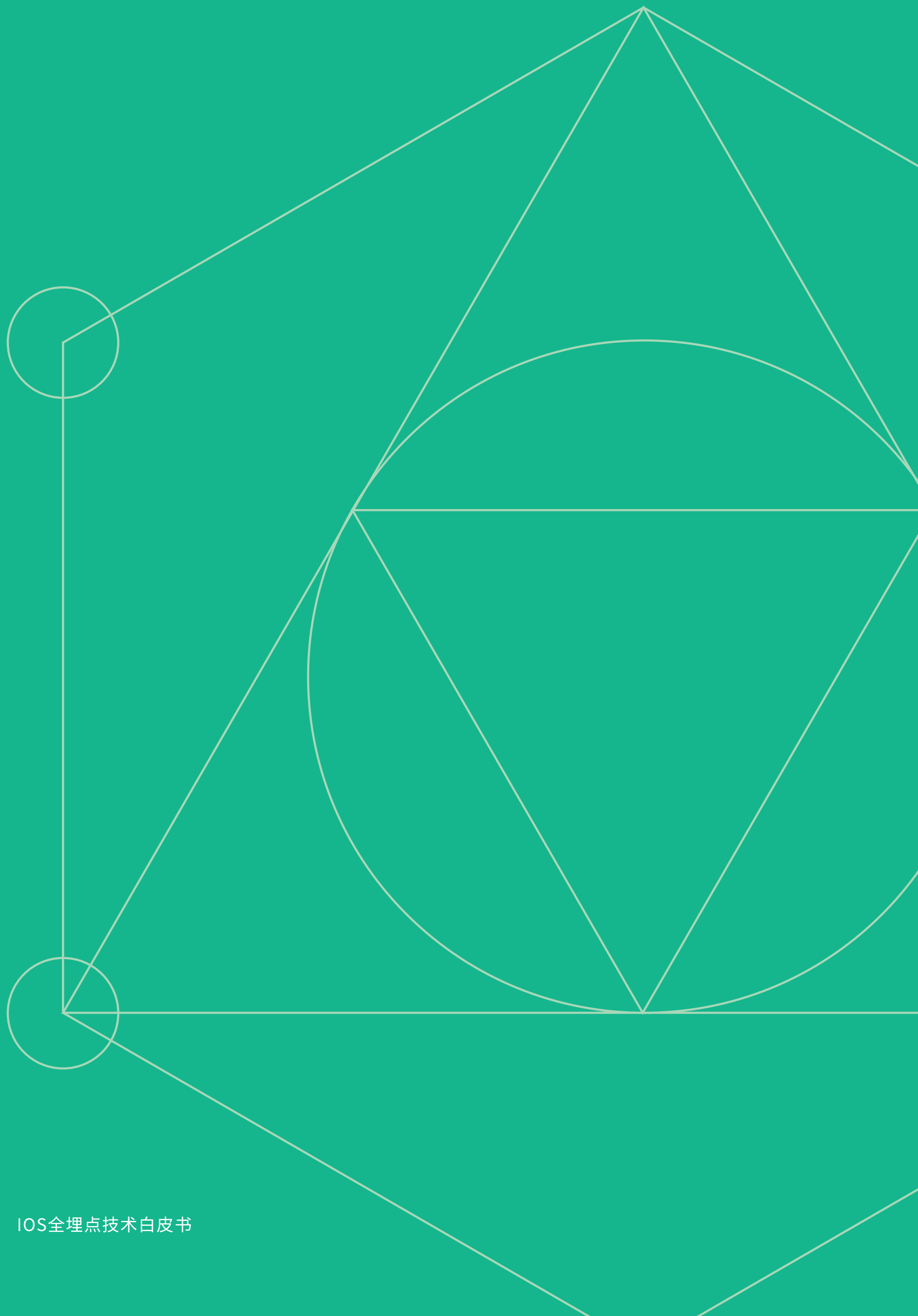
## 2、实现步骤

完整的项目源码后续会 release 给大家。

## 3、知识点

- IDFA
- IDFV
- UUID
- Keychain

# 08
# 时间相关



# 时间相关

## 1、原理概述

在介绍时间相关的内容之前，我们先介绍一下事件模型。

(1) 事件模型

在数据分析中，一般都是采用“事件模型”(Event 模型) 来描述用户在产品上的各种行为或者动作。简单来说，事件模型包括事件 (Event) 和用户 (User) 两个最核心的实体。事件模型，也是数据分析中最基本的数据模型。事件模型可以给我们提供足够的信息，让我们知道用户在使用我们的产品时具体都做了什么事情。事件模型给予我们更全面且更具体的信息，指导我们对业务、产品做出更好、更准确的决策。

一般来说，一个“事件”就是描述了：一个用户 (Who) 在某个时间点 (When)、某个地方 (Where)，以某种方式 (How) 完成了某个具体的事情 (What)。因此，一个完整的事件，主要包含以下五个关键因素：

Who：参与这个事件的用户是谁 (可参考第 8 章介绍的用户标识)。

When：事件发生的时间。

Where：事件发生的地点 (比如通过 IP 地址解析省市区、记录经度纬度等)。

How：用户触发这个事件的方式 (比如用户使用的设备信息、浏览器、应用程序版本号、操作系统版本号、渠道信息等)。

What：描述用户所做的这个事件的具体内容 (比如对于一个“搜索”类型的事件，则可能需要记录的字段有：搜索关键词、搜索类型等)。

在本章，我们重点介绍 When 这个因素，即时间。

在数据采集时，和时间相关的问题主要体现在以下两个方面：

- 事件发生的时间戳
- 统计事件持续的时长

(2) 事件发生的时间戳

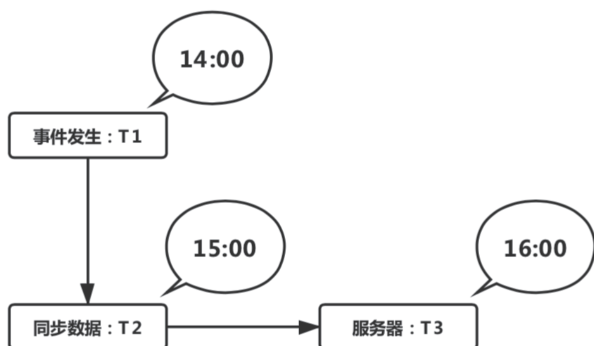
一般情况下，把用户手机设备的时间戳作为事件发生的 time，并没有什么太大的问题。但在一些比较特殊的情况或者场景下，用户手机设备的时间戳有可能是不准确的，从而就会导致我们采集到的 time 不符合实际情况，比如今天是 8 月 20 日，结果我们采集到了 8 月 28 日的事件，或者采集到了 8 月 10 日的事件。

关于时间戳的问题，也是一个老生常谈的话题。大家首先想到的，应该是同步或者校准用户手机的时间。但同步或者校准用户手机时间，不仅需要网络权限，更需要一个稳定的时间服务器，相对比较麻烦，也得不偿失，而且不一定就能解决我们在数据采集中面临的问题 (比如在无网环境下无法进行同步或者校准时间，导致触发事件的时间错误)。

大家可能也想到了另一个方案：使用服务端的时间戳作为事件发生的 time。其实，这个方案也不可行。这是因为事件触发后，事件信息并不会立即同步给服务端，从而导致无法及时获取服务端的时间戳。比如，数据采集 SDK 为了最大限度的减少对应用程序本身的性能影响，一般情况下，在本地都会有缓存机制 (比如 iOS 应用程序的 SQLite3 数据库)，即事件会先保存到本地缓存，当符合特定的同步策略时 (比如本地缓存了一定量的事件是、每隔一定的时间间隔、关键事件发生时等)，才会向服务端同步数据。

经过长时间的摸索和总结，我们目前采用的是“时间纠正”这个策略。

对，是“时间纠正”，不是“时间校准”，也不是“时间同步”。




那什么是“时间纠正”呢?

图 8-1 事件时间纠正策略

我们以上图 8-1 为例来介绍“时间纠正”策略。

用户在手机时间戳 T1 时触发了一个事件（比如登录事件），然后事件存入本地缓存。数据采集 SDK 在用户手机时间戳 T2 时开始同步事件数据（包含登录事件），然后服务端通过 HTTP(S) 的 Request Header Date 可以获取到数据采集 SDK 发起 Request 时用户手机时间戳 T2 。如果 T2 与当前服务端的时间戳 T3 的误差在一定的可接受范围之内（比如 30s，这是因为网络请求也需要一定的时间），就可以说明当前用户手机的时间戳是准确的（我们假设服务端时间戳是准确的，绝大部分情况下也都是准确的），同时说明登录事件发生时的时间戳 T1 也是准确的，服务端在接收到登录事件时无需做任何的特殊处理；如果 T2 与 T3 相差比较大（比如上图 8-1 中的 T2 为 15:00，T3 为 16:00），我们可以确定当前用户手机的时间戳 T2 是不准确的，即比服务端的时间戳晚了一个小时（16:00 - 15:00 = 一个小时），从而就可以假设登录事件发生时的时间戳 T1 也比服务端晚了一个小时，即 14:00 加上一个小时应该为 15:00。因此，服务端在接收到事件时会把登录事件的时间戳 T1 再加上一个小时，变成 15:00，从而就达到了“时间纠正”的效果。

当然，“时间纠正”策略，也无法确保可以 100% 解决所有和时间戳相关的问题。比如，还是以上图 8-1 为例，如果在时间戳 T1 和 T2 之间，用户手机的时间戳再次发生了人为变更，那么，该方案就无法正确的进行时间纠正了，这是因为时间戳 T2 和 T3 之间的误差额，与时间戳 T1 和当时服务端对应时间戳的差额就不相等了。但从实际情况来看，由于数据采集 SDK 一般都会以较短的固定时间间隔同步数据（比如 15s），所以时间戳 T1 和 T2 之间的时间间隔会比较短，本身发生时间戳变更的可能性就比较小，即使真正发生变更了，影响到的事件数量也比较少（15s 内可能只会触发几条事件）。因此，“时间纠正”策略可以解决 99% 以上的和事件时间戳相关的问题。

另外，由于“时间纠正”策略的实现逻辑，都是在服务端处理的，而且实现起来也比较简单，在此我们不再详细描述实现细节。

（3）统计事件持续时长

事件持续时长，是用来统计用户的某个行为或者动作持续了多长时间（比如观看某个视频）。统计事件持续时长，就像一个计时器，当用户某个行为或者动作发生时，我们就开始计时；当行为或者动作结束时，我们就停止计时。这个时间间隔（在事件中，我们用属性 $event_duration 来表示），就是用户发生这个行为或者动作的持续时长。

如果用户调整了手机时间，有可能会导致出现以下三个问题：

- 统计的 $event_duration 可能接近为零
- 统计的 $event_duration 可能非常大，比如超过一个月
- 统计的 $event_duration 可能为负数

这是什么原因导致的?

这是因为，我们目前是借助手机客户端时间来计算 $event_duration 的。一旦用户调整了手机时间，必然会影响到 $event_duration 的计算结果。

那我们有没有一种办法，在统计时长时不受手机时间的影响呢?

其实是有的！

下面我们引入 systemUpTime。

关于 systemUpTime，苹果官方文档解释为：

The amount of time the system has been awake since the last time it was restarted.

翻译成中文，即系统启动时间，也叫开机时间，是指设备开机后一共运行了多少秒（ 设备休眠不统计在内 ），并且不会受到系统时间更改的影响。如果我们使用 systemUpTime 来计算 $event_duration，就会 100% 准确了。

（4）全埋点事件时长

我们之前实现的全埋点事件中，有一些事件也是需要统计事件持续时长的，比如 $AppEnd 事件和 $AppViewScreen 事件。

下面我们分布介绍如何为 $AppEnd 事件和 $AppViewScreen 事件添加 $event_duration 属性。

A.$AppEnd 事件时长

对于 $AppEnd 事件的 $event_duration 属性来说，是指应用程序从进入前台处于活跃状态到进入后台的整个运行时间间隔。$AppEnd 事件时长基本上就代表了用户此次使用应用程序的时长。

实现统计 $AppEnd 事件时长相对来说比较简单：当收到 UIApplicationDidBecomeActiveNotification 本地通知时开始计时；收到 UIApplicationDidEnterBackgroundNotification 时结束计时。

B.$AppViewScreen 事件时长

对于 $AppViewScreen 事件，它不像 $AppClick 事件是一个短暂的动作，而是属于一个相当耗时的过程。因此，在实际业务分析需求中，我们更想知道用户浏览页面事件的持续时长。那我们如何才能采集到页面浏览事件的持续时长呢？

大家可能会想到，我们之前实现 $AppViewScreen 事件的全埋点时，是通过交换 UIViewController 的 -viewDidAppear: 方法实现的。我们从 UIViewController 的生命周期可知，当前页面离开时会调用 -viewWillDisappear: 方法。因此，我们可以再去交换 UIViewController 的 -viewWillDisappear: 方法，然后在 -viewDidAppear: 方法中开始计时，并在 viewWillDisappear: 中触发 $AppViewScreen 事件并统计时长。

这个方案实现起来也比较简单，但它会有两个问题：

- 如何计算最后一个页面的页面浏览事件持续时长？

因为在某个页面，用户可能随时会把应用程序强杀或者应用程序由于意外发生崩溃，甚至应用程序正常进入后台，都不会执行或来不及执行当前页面的 -viewWillDisappear: 方法。

- 如何处理嵌套了子页面的浏览事件时长？

比如 A 页面嵌套了 B 页面，如何计算 A 页面的页面浏览事件时长？

其实，在神策商用数据采集 SDK 中，我们并没有在 SDK 中实现采集页面浏览事件时长，而是使用 Session 来解决这个问题的。大家如果对 Session 感兴趣，可以查看神策官网上的一篇文章《如何应用 Sensors Analytics 进行 Session 分析》，网址为：https://www.sensorsdata.cn/blog/ru-he-ying-yong-sensors-analytics-jin-xing-session-fen-xi/。

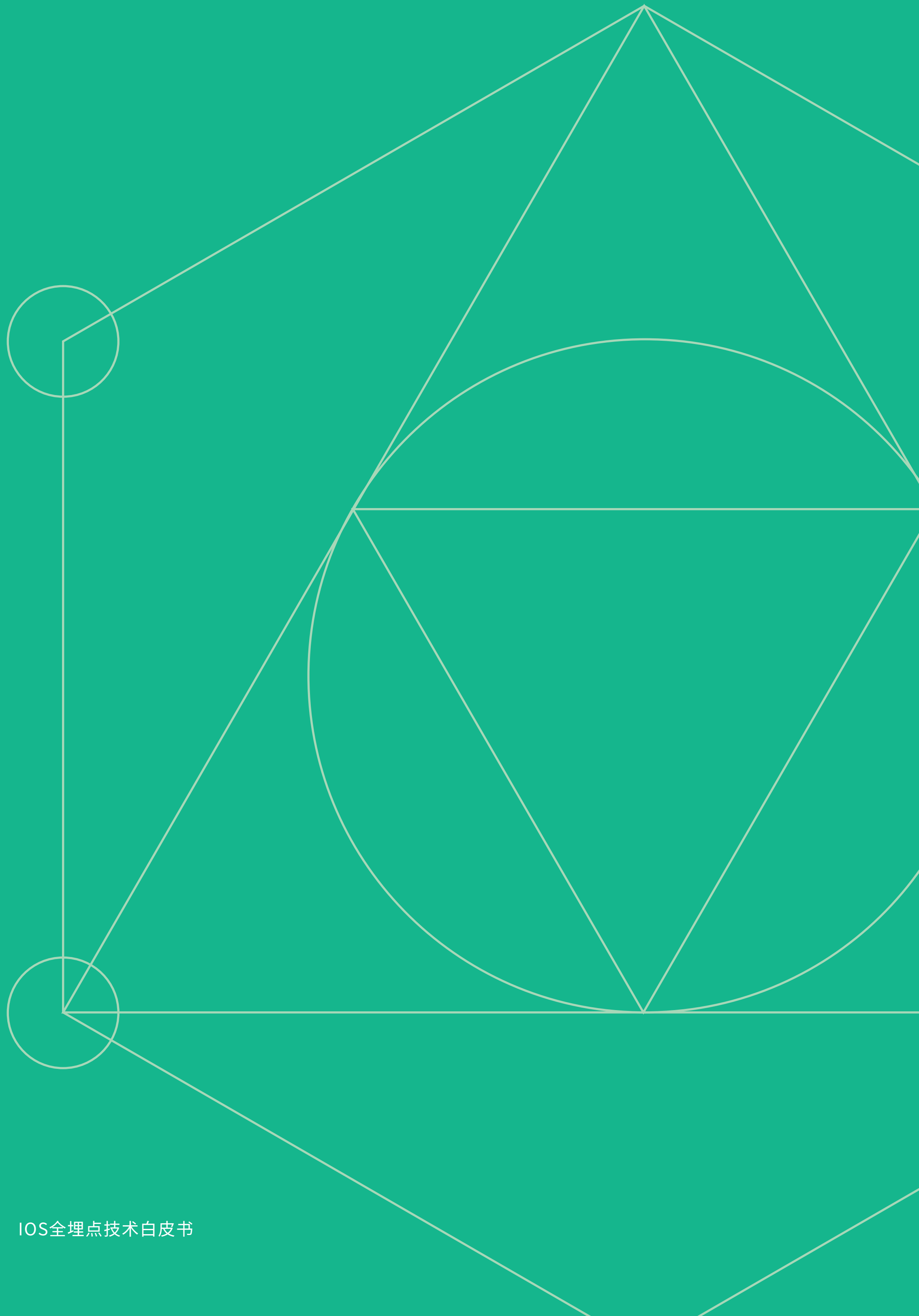
## 2、实现步骤

完整的项目源码后续会 release 给大家。

## 3、知识点

- 时间纠正
- systemUpTime
- Session

# 09
# 数据存储



# 数据存储

## 1、原理概述

为了最大限度的保证事件数据的准确性、完整性和及时性，数据采集 SDK 需要及时的将事件数据同步到服务端。但在某些特殊情况下，比如手机处于断网环境，或者根据实际需求只能在 Wi-Fi 环境中才能同步数据等，可能会导致事件数据同步失败或者无法进行同步。因此，数据采集 SDK 需要先把事件数据缓存在本地，待符合一定的策略（条件）之后，再去同步数据。

（1）数据存储策略

在 iOS 应用程序中，从“数据缓存在哪里”这个维度看的话，缓存一般分为两种类型：

- 内存缓存
- 磁盘缓存

内存缓存，是将数据缓存在内存中，供应用程序直接读取和使用。由于是读写内存，因此读写速度极快。但是，内存缓存的缺点也十分明显，由于内存资源比较有效，应用程序在系统中申请的内存，会随着应用程序生命周期的结束而被释放。这就意味着，如果应用程序在运行的过程中被用户强杀或者出现崩溃情况，都有可能导致内存中缓存的数据丢失。因此，将事件数据缓存在内存中不是一个最佳选择。

磁盘缓存，是将数据缓存在磁盘空间中，其特点正好与内存缓存相反。磁盘缓存容量大，但是读写速度相对于内存缓存来说要慢一些。不过磁盘缓存是持久化存储，不受应用程序生命周期的影响。一般情况下，一旦数据成功的保存在磁盘中，其丢失的风险就非常低。因此，即使磁盘缓存数据读写速度较慢，但在综合考虑下，磁盘缓存是我们缓存事件数据的最优选择。

在 iOS 应用程序中，一般通过以下两种方式使用磁盘缓存：

- 文件缓存
- 数据库缓存（一般是指在 SQLite 数据库）

这两种方式，都可以用来实现数据采集 SDK 的缓存机制。不管我们使用哪种方式，缓存策略都是相同的：当事件触发后，先讲事件存储在缓存中，待符合一定的策略之后，从缓存中读取事件并进行同步，同步成功后，再将已同步的事件从缓存中删除。

在写入性能方面，SQLite 数据库优于文件缓存；对于读取性能，情况稍微复杂一些。

我们先看一组如下表 9-1 的测试数据：

| 每页缓存大小 | 单条数据大小 | | | | | | |
|---|---|---|---|---|---|---|---|
| | 10k | 20k | 50k | 100k | 200k | 500k | 1m |
| 1024 | 1024 | 1024 | 1024 | 1024 | 1024 | 1024 | 1024 |
| 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 | 2048 |
| 4096 | 4096 | 4096 | 4096 | 4096 | 4096 | 4096 | 4096 |
| 8192 | 8192 | 8192 | 8192 | 8192 | 8192 | 8192 | 8192 |
| 16384 | 16384 | 16384 | 16384 | 16384 | 16384 | 16384 | 16384 |
| 32768 | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 | 32768 |
| 65536 | 65536 | 65536 | 65536 | 65536 | 65536 | 65536 | 65536 |

表 9-1 SQLite 读取性能

这个表格是 SQLite 官方提供的测试结果。对于每个测试用例，都是读取包含 100MB 大小、BLOB 类型的数据，每条数据的大小从 10KB 到 1MB，对应的数据条数从 1000 条到 100 条。另外，为了使内存中缓存的数据保持在 2MB，同时对数据做了分页处理，例如，将数据分成 2000 页，那么每页就有 1024B 大小的数据。表格中的测试结果，表示的是直接从文件中读取数据的时间除以从 SQLite 数据库中读取数据的时间。因此，如果值大于 1.0，则表示从 SQLite 数据库中读取速度更快；如果值小于 1.0，从文件中读取的速度更快。相关说明可参照 SQLite 官网中的说明文档 https://www.sqlite.org/intern-v-extern-blob.html。

通过上表 9-1 中的测试数据，我们可以得出如下结论：

- 当每页缓存大小在 8192 到 16384 之间时，数据库拥有最好的读写性能
- 如果单条数据小于 100KB 时，从 SQLite 数据库中读取数据速度更快；单条数据大于 100KB 时，从文件中读取速度更快

当然 SQLite 官方测试的环境是在 Linux 工作站上进行的，上述测试结果肯定会受到硬件和操作系统的影响。使用 iPhone 设备进行测试发现，单条事件数据的阈值在 20KB 左右。

因此，数据采集 SDK 一般都是使用 SQLite 数据库来缓存数据，这样可以拥有最佳的读写性能。如果希望采集更完整、更全面的信息，比如采集用户操作时当前截图的信息（一般会超过 100KB），文件缓存可能是最优的选择。

接下来我们分别介绍如何使用文件和数据库来缓存事件数据。

（2）文件缓存

使用文件缓存数据，实现起来相对比较简单，主要操作就是写文件和读取文件。可以使用 NSKeyedArchiver 类将字典对象进行归档并写入文件，也可以使用 NSJSONSerialization 类把字典对象转换成 JSON 格式字符串写入文件。

我们每次都是将所有的数据写入同一个文件，但其实性能并不差，因为前面我们也对比过，每次写入的数据量越大，文件操作相对来说拥有更优秀的性能。

当然，文件缓存是不够灵活的，我们很难使用更细的粒度去操作数据，比如，很难对其中的某一条数据进行读和写。

（3）数据库缓存

在 iOS 应用程序中，使用的数据库一般是 SQLite 数据库。SQLite 是一个轻量级的数据库，数据存储简单高效，使用也非常简单，只需要在项目中添加 libsqlite3.0 依赖，并在使用的时候引入 sqlite3.h 头文件即可。

数据库缓存事件数据，实现起来，相对文件缓存来说要复杂很多，并且需要一定的 SQL 基础，sqlite3 的 API 学习成本也很高。

但是相对于文件缓存来说，数据库缓存更加灵活，可以实现对单条数据的插入、查询和删除操作，同时调试也更容易。SQLite 数据库也有极高的性能，特别是对单条数据的操作，性能明显优于文件缓存。

（4）总结

文件缓存和数据库缓存各有优缺点，大家可以根据自己的需求进行选择。

在我们的数据采集 SDK 中，综合考虑了灵活性和存取性能，使用了 SQLite 数据库来缓存数据。

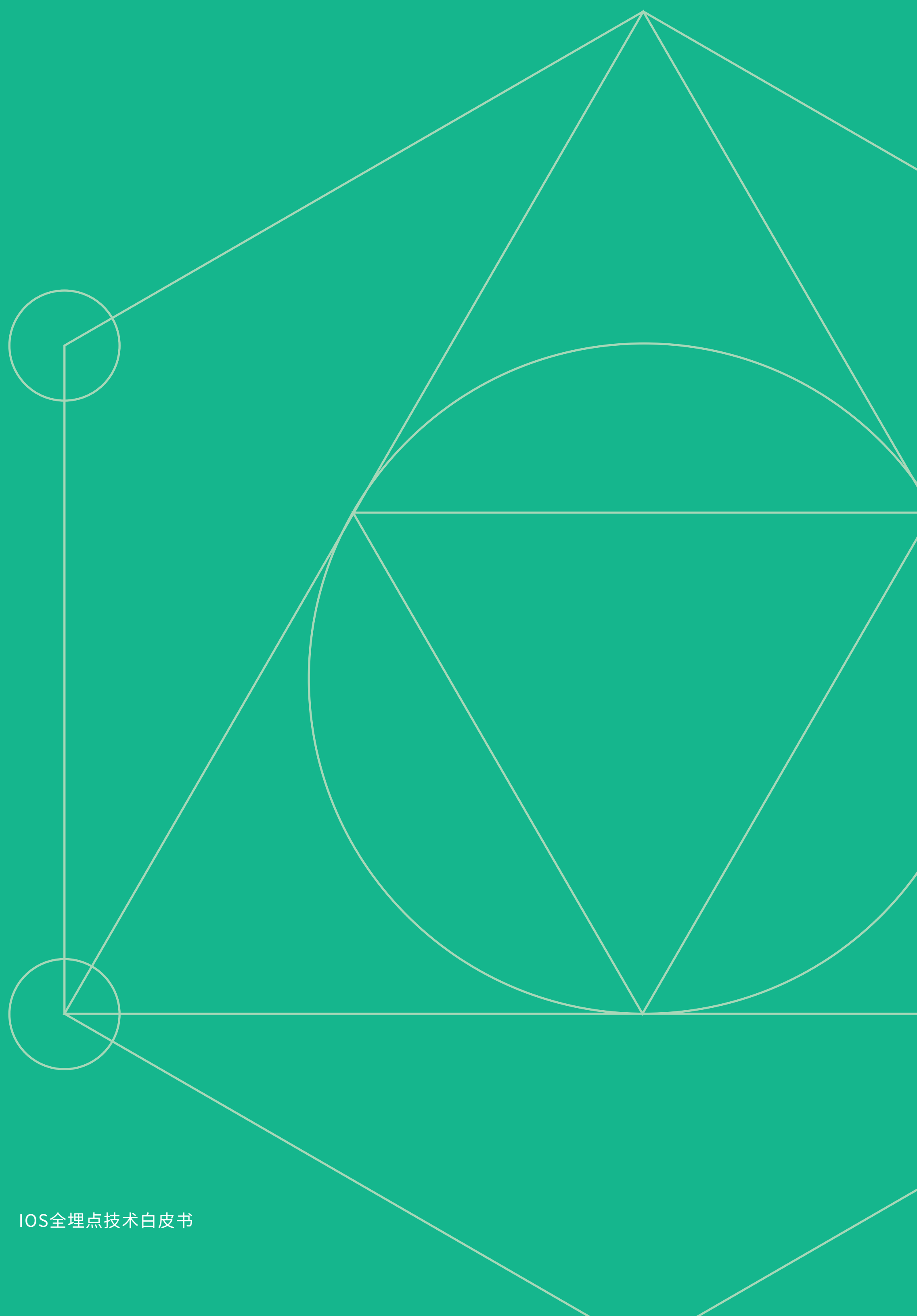
# 2、实现步骤

完整的项目源码后续会 release 给大家。

# 3、知识点

- 文件缓存
- SQLite

# 10 数据同步



# 数据同步

## 1、原理概述

在前面的章节中我们介绍了如何把事件数据缓存到客户端本地。如果事件数据一直缓存在本地，是没有什么意义的，我们还需要把数据同步到服务端，然后再经服务端的存储、抽取、分析和展现，才能充分发挥数据真正的价值。因此，本章我们主要介绍如何把缓存在本地的事件数据同步给服务端。

（1）同步数据

在 Foundation 框架中，苹果为我们提供了封装好的发送网络请求的 API。但在实际的 iOS 应用程序开发中，开发人员很少会直接基于 Foundation 框架进行开发，绝大部分情况下都会选择使用第三方库，比如 AFNetworking 等（AFNetworking 也是基于 Foundation 框架进行封装的）。作为通用的数据采集 SDK，要求（尽量）不能去依赖任何第三方库。因此，我们就需要基于 Foundation 框架进行网络相关的开发。另外，同步数据功能相对比较简单，我们直接使用 Foundation 框架中的 NSURLSession 类即可满足我们的需求。

（2）数据同步策略

作为一个标准的数据采集 SDK，必须包含一些自动同步数据的策略，一方面是为了降低用户使用 SDK 的难度和成本，另一方面更是为了确保数据的正确性、完整性和及时性。

A．基本原则

在这里，我们以最常用的三种数据同步策略为例来介绍：

- 策略一：客户端本地已缓存的事件超过一定条数时同步数据（比如 100 条）
- 策略二：客户端每隔一定的时间间隔同步一次（比如每隔 15s 就同步一次）
- 策略三：应用程序进入后台时尝试同步本地已缓存的所有数据

需要特别注意的是，事件和事件之间，是有先后顺序的。比如，需要先有浏览商品事件，然后才能有加入购物车的事件。因此，在同步数据的时候，我们需要严格按照事件触发的时间先后顺序同步数据。

B. 策略一

即客户端本地已缓存的数据超过一定条数时同步数据（比如 100 条）。

实现这个策略非常简单：每次事件触发并入库后，我们检查一下已缓存的事件条数是否超过了我们定义的阈值，如果已达到，则同步数据。

C. 策略二

即客户端每隔一定的时间间隔就同步一次数据（比如默认 15s）。

我们可以开启一个定时器（NSTimer），每隔一定时间间隔同步一次数据。

D. 策略三

即应用程序进入后台时尝试同步本地已缓存的所有数据。

对于 iPhone 用户来说，比较习惯通过按 Home 键或者上滑 HomeBar 让应用程序进入后台。如果此时我们不去尝试同步数据，就有可能造成一些数据丢失，因为用户把应用程序进入后台，ta 下次具体什么时候再启动应用程序就不得而知了，更有甚者，ta 下一秒可能就把应用程序卸载。因此，每当应用程序进入后台的时候，我们都要尝试同步数据，最大限度的保证数据的完整性。

我们都知道，苹果对应用程序的后台运行管控的非常严格，如果不做任何处理，应用程序进入后台后大概只有几秒钟的时间处理数据。在这么短的时间内，我们完成数据的同步，是完全不可控的，有可能会出现一些无法预料的后果。因此，我们需要更长的后台运行时间。此时，可以借助 UIApplication 类的 - beginBackgroundTaskWithExpirationHandler: 方法，这个方法可以让应用程序在后台最多有 3 分钟的时间来处理数据。在网络正常的情况下，这个时间对于我们同步数据而言足够了。

# 2、实现步骤

完整的项目源码后续会 release 给大家。

# 3、知识点

- NSURLSession
- 数据同步策略

# 11
# 采集崩溃

# 采集崩溃

## 1、原理概述

在 iOS 应用程序开发中，我们难免会碰到由于各种异常原因导致的崩溃情况。特别是对于像 Objective-C 这样的动态语言来说，一旦代码出了异常，一般都会导致应用程序崩溃。在开发的过程中，如果出现了崩溃，我们都可以根据本地崩溃信息快速定位、修改代码并修复。但对于线上版本发生的一些崩溃，我们只能通过收集崩溃信息来分析具体的原因。苹果也提供了崩溃信息上报的功能，但并不是所有的 iPhone 用户都开启了该功能。因此，对于数据采集 SDK 来说，采集崩溃信息并上报也是必不可少的一项功能。

采集应用程序的崩溃信息，主要分为以下两种场景：

- NSException 异常
- Unix 信号异常

下面，我们分别详细介绍如何实现采集崩溃信息的全埋点。

（1）NSException 异常

NSException 异常是 Objective-C 代码抛出的异常。在 iOS 应用程序中，最常见的就是通过 @throw 抛出的异常，比如常见的数组越界访问异常：

```
@throw [NSException exceptionWithName:@"NSRangeException" reason:@"index 2 beyond bounds [0 .. 1]" userInfo:nil];
```

运行程序就会出现如下异常信息：

```
Terminating app due to uncaught exception 'NSRangeException', reason: 'index 2 beyond bounds [0 .. 1]'
```

A．捕获 NSException 异常

要想捕获 NSException 类型的异常，我们可以通过 NSSetUncaughtExceptionHandler 函数来全局设置异常处理函数，然后再收集异常堆栈信息并触发相应的事件（$AppCrashed），即可达到采集 NSException 异常的全埋点效果。

B．传递 UncaughtExceptionHandler

在上一小节中，我们知道通过 NSSetUncaughtExceptionHandler 函数来全局设置异常处理函数，可以采集应用程序的崩溃事件（$AppCrashed）。但我们仍需考虑另一个问题，在应用程序实际的开发过程中，可能会集成多个 SDK，如果这些 SDK 都按照上面介绍的方法采集异常信息，总会有一些 SDK 采集不到异常信息。这是因为通过 NSSetUncaughtExceptionHandler 函数设置的是一个全局异常处理函数，后面设置的会自动覆盖前面设置的。

那如何解决这个问题呢？

目前，常见的做法是：在我们调用 NSSetUncaughtExceptionHandler 函数全局设置异常处理函数之前，先通过

NSGetUncaughtExceptionHandler 函数获取之前已设置的异常处理函数，并进行保存，在我们处理完异常信息采集后，再主动调用已备份的处理函数（让所有的异常处理函数形成链条），即可解决上面提到的覆盖问题。

通过这样处理之后，即可把所有的异常处理函数形成链条，确保在我们 SDK 之前设置异常处理函数的 SDK 也能采集到异常信息。不过，如果在我们后面设置异常处理函数的 SDK 没有进行有效的传递，可能也会导致我们也无法采集到异常信息。

（2）捕获信号

在 iOS 系统自动采集的崩溃日志中，我们可以看到如下的条目：

```
Exception Type:  EXC_BAD_ACCESS (SIGSEGV)

Exception Subtype: KERN_INVALID_ADDRESS at 0x0000000001000010

VM Region Info: 0x1000010 is not in any region.  Bytes before following region: 4283498480

    REGION TYPE START - END [ VSIZE] PRT/MAX SHRMOD  REGION DETAIL UNUSED SPACE AT START

--->  __TEXT 0000000100510000-0000000100514000 [   16K] r-x/r-x SM=COW  ....app/Ekuaibao


Termination Signal: Segmentation fault: 11

Termination Reason: Namespace SIGNAL, Code 0xb

Terminating Process: exc handler [21776]

Triggered by Thread:  9
```

这是一个很常见的异常崩溃信息，在 Exception Type 中有两个字段：EXC_BAD_ACCESS 和 SIGSEGV，它们分别是 Mach 异常和 Unix 信号。

那什么是 Mach 异常和 Unix 信号呢?

Mach 是 macOS 和 iOS 操作系统的微内核，Mach 异常就是最底层的内核级异常。在 iOS 系统中，每个 Thread、Task、Host 都有一个异常端口数据。通过 Mach 暴露的 API，开发者可以设置 Thread、Task、Host 的异常端口，从而捕获 Mach 异常。Mach 异常会被转换成相应的 Unix 信号，并传递给出错的线程。上面 Exception Type 的意思就是，Mach 层的异常 EXC_BAD_ACCESS 被转换成 SIGSEGV 信号并传递给出错的线程。在 Triggered by Thread 中也可以看到出错的线程编号。之所以会将 Mach 异常转换成 Unix 信号，是为了兼容 POSIX 标准 (SUS 规范 )，这样即使不了解 Mach 内核也可以通过 Unix 信号的方式来进行兼容开发。

Unix 信号的种类有很，在 iOS 应用程序中，常见的 Unix 信号有以下几种：

- SIGILL：程序非法指令信号，通常是因为可执行文件本身出现错误 , 或者试图执行数据段，堆栈溢出时也有可能产生这个信号。
- SIGABRT：程序中止命令中止信号，调用 abort 函数生成。
- SIGBUS：程序内存字节地址未对齐中止信号，比如访问一个四个字长的整数 , 但其地址不是 4 的倍数。
- SIGFPE：程序浮点异常信号，不仅包括浮点运算错误 , 还包括溢出及除数为 0 等其它所有的算术错误时都会产生这个信号。
- SIGKILL：程序结束接收中止信号，用来立即结束程序运行，不能被处理、阻塞以及忽略。
- SIGSEGV：程序无效内存中止信号，即试图访问未分配的内存 , 或往没有写权限的内存地址写数据。

- SIGPIPE：程序管道破裂信号，通常在进程间通信产生。
- SIGSTOP：程序进程中止信号，这个信号和 SIGKILL 一样不能被处理、阻塞以及忽略。

在 iOS 应用程序中，一般情况下，我们只需要采集 SIGILL、SIGABRT、SIGBUS、SIGFPE 和 SIGSEGV 这几个常见的信号，即能满足我们日常采集应用程序异常信息的需求。

（3）采集异常时的 $AppEnd 事件

在之前的章节中介绍过，我们通过监听应用程序的状态（UIApplicationDidEnterBackgroundNotification），实现了 $AppEnd 事件的全埋点。但是，一旦应用程序发生了异常，我们就采集不到 $AppEnd 事件了，从而会造成在用户的行为序列中，会出现 $AppStart 事件和 $AppEnd 事件不是成对的情况。因此，在应用程序发生崩溃时，我们需要补发 $AppEnd 事件。

这样处理之后，当应用程序发生异常时，我们不仅可以采集 $AppCrashed 事件，也能正常采集 $AppEnd 事件。

## 2、实现步骤

完整的项目源码后续会 release 给大家。

## 3、知识点

- NSException
- UncaughtExceptionHandler
- Mach 异常
- Unix 信号

# 12
# App 与 H5 打通

# App 与 H5 打通

## 1、原理概述

iOS 混合开发越来越流行，App 与 H5 的打通需求，也越来越迫切！

那什么是 App 与 H5 打通呢？

所谓打通，是指 H5 集成 JavaScript 数据采集 SDK 后，H5 触发的事件不是直接同步给服务端，而是先发给 App 端的数据采集 SDK，经 App 端数据采集 SDK 二次加工处理后入本地缓存再进行同步。

本章后面的内容，主要是回答以下两个问题：

- App 与 H5 为什么要打通？
- App 与 H5 如何打通？

App 为什么要与 H5 打通呢？

主要是从以下几个角度考虑：

- 数据丢失率

在业界，App 端采集数据的丢失率一般在 1% 左右，而 H5 采集数据的丢失率一般在 5% 左右（主要是因为缓存、网络或切换页面等原因）。因此，如果 App 与 H5 打通，H5 触发的所有事件，可以先发给 App 端数据采集 SDK，经过 App 端二次加工处理后并入本地缓存，在符合特定策略之后再进行同步数据，即可把数据丢失率由 5% 降到 1% 左右。

- 数据准确性

众所周知，H5 无法直接获取设备相关的信息，只能通过解析 UserAgent 值获取到有限的信息。而解析 UserAgent 值，至少会面临以下两个问题：

1）有些信息，通过解析 UserAgent 值根本获取不到，比如应用程序的版本号等

2）有些信息，通过解析 UserAgent 值可以获取到，但内容可能不正确

如果 App 与 H5 打通，由 App 端数据采集 SDK 补充这些信息，即可确保事件信息的准确性和完整性。

- 用户标识

如果用户在 App 端注册或登录之前使用我们的产品，我们一般都是使用匿名 ID 来标识用户。而 App 与 H5 标识匿名用户的规则不一样（iOS 一般使用 IDFA 或 IDFV，H5 一般使用 Cookie），从而就会导致一个用户使用了我们的产品，结果产生了两个匿名用户。如果 App 与 H5 打通，就可以将两个匿名 ID 做归一化处理（以 App 端匿名 ID 为准）。

App 与 H5 如何打通？

常见的打通方案有以下两种：

- 通过拦截 WebView 请求进行打通
- 通过 JavaScript 与 WebView 相互调用进行打通

（1）方案一：拦截请求

拦截请求，顾名思义就是拦截 WebView 发送的 URL 请求，即：如果请求是协定好的特定格式，我们进行拦截并获取事件数据；如果不是则让请求继续加载。此时，JavaScript SDK 就需要知道，当前 H5 是在 App 内显示还是在 Safari 浏览器内显示，只有在 App 内显示时，H5 触发事件后，JavaScript SDK 才能向 App 发送特定的 URL 请求进行打通；如果是在 Safari 浏览器内显示，JavaScript SDK 也去发送请求进行打通，会导致事件丢失。对于 iOS 来说，目前常用的方案是借助 UserAgent 来进行判断，即：当 H5 在 App 内显示时，我们通过在当前的 UserAgent 上追加一个特殊的标记（“ /sa-sdk-ios ”），用来告诉 JavaScript SDK 当前 H5 是在 App 内显示并需要进行打通。

A. 修改 UserAgent

在当前的 UserAgent 上追加一个特殊的标记（“ /sa-sdk-ios ”），就会涉及到修改 UserAgent。

在 iOS 应用程序中，WebView 控件有以下两种：

- UIWebView
- WKWebView

虽然苹果近年来一直在推动 WKWebView 控件来替代 UIWebView 控件，但实际上仍有大量的 iOS 应用程序在使用 UIWebView 控件来展示 H5 页面。因此，对于如何修改 UserAgent，我们仍需支持 UIWebView 控件和 WKWebView 控件。

对于 UIWebView，可以通过如下方式修改 UserAgent：

```
// 创建一个空的 UIWebView
UIWebView *webView = [[UIWebView alloc] initWithFrame:CGRectZero];
// 取出 UIWebView 的 UserAgent
NSString *userAgent = [webView stringByEvaluatingJavaScriptFromString:@"navigator.userAgent"];
// 给 UserAgent 中添加自己需要的内容
userAgent = [userAgent stringByAppendingString:@" /sa-sdk-ios "];
// 将 UserAgent 字典内容注册到 NSUserDefaults 中
[[NSUserDefaults standardUserDefaults] registerDefaults:@{@"UserAgent": userAgent}];
```

对于 WKWebView 控件，可以通过如下方式修改 UserAgent：

```
// 创建一个空的 UIWebView
UIWebView *webView = [[UIWebView alloc] initWithFrame:CGRectZero];
// 取出 UIWebView 的 UserAgent
NSString *userAgent = [webView stringByEvaluatingJavaScriptFromString:@"navigator.userAgent"];
// 给 UserAgent 中添加自己需要的内容
userAgent = [userAgent stringByAppendingString:@" /sa-sdk-ios "];
// 将 UserAgent 字典内容注册到 NSUserDefaults 中
[[NSUserDefaults standardUserDefaults] registerDefaults:@{@"UserAgent": userAgent}];
```

对于 WKWebView 控件，可以通过如下方式修改 UserAgent：

```
// 创建一个空的 WKWebView，由于 WKWebView 执行 JavaScript 代码是异步过程，所以需要强引用 WKWebView 对象
self.webView = [[WKWebView alloc] initWithFrame:CGRectZero];
// 创建一个 self 的弱引用，防止循环引用
__weak typeof(self) weakSelf = self;
// 执行 JavaScript 代码，获取 WKWebView 中的 UserAgent
[self.webView evaluateJavaScript:@"navigator.userAgent" completionHandler:^(id result, NSError *error) {
  // 创建强引用
  __strong typeof(weakSelf) strongSelf = weakSelf;
  // 执行结果 result 为获取到的 UserAgent 值
  NSString *userAgent = result;
  // 给 UserAgent 中追加自己需要的内容
  userAgent = [userAgent stringByAppendingString:@" /sa-sdk-ios "];
  // 将 UserAgent 字典内容注册到 NSUserDefaults 中
  [[NSUserDefaults standardUserDefaults] registerDefaults:@{@"UserAgent": userAgent}];
  // 释放 webView
  strongSelf.webView = nil;
}];
```

通过 WKWebView 控件修改 UserAgent 稍微复杂一点，这是因为 WKWebView 控件执行 JavaScript 代码是一个异步的过程。

修改 UserAgent，我们一般都是建议进行“追加”，比如示例中追加的是：/sa-sdk-ios 字符串，应该尽量避免直接覆盖。

对于标准的 UserAgent ，一般都有固定的格式，比如每段信息都会使用空格进行分割：

```
Mozilla/5.0 (iPhone; CPU iPhone OS 13_2_2 like Mac OS X) AppleWebKit/605.1.15 (KHTML, like Gecko) Mobile/15E148
```

因此，在修改 UserAgent 时需要注意符合格式规范，以免引起无法正确解析 UserAgent 值的情况。

B. 是否拦截

当 JavaScript SDK 发送一个特殊的 URL 请求后，App 端数据采集 SDK 需要判断是否要进行拦截。

那具体如何判断呢？

我们可以和 JavaScript SDK 协定好这个特殊请求的 URL 格式，比如：sensorsanalytics://trackEvent?event=xxxxx，其中后面的 event 参数代表的就是事件信息。因此，我们可以新增一个方法，用来判断当前发送的请求是否符合我们上面协定的格式。

C. 二次加工 H5 事件

拿到事件信息后，我们还需要对 H5 的事件进行二次加工处理，比如：

- 添加 App 端的预置属性，防止 H5 事件没有相应的预置属性或者预置属性内容不正确
- 添加 _hybrid_h5 事件属性，表明当前 H5 事件是经过打通处理并同步的
- 修改 distinct_id 字段，确保用户标识归一

我们还可以根据实际的需求添加其它二次加工处理的逻辑。

D. 拦截

UIWebView 控件和 WKWebView 控件拦截请求的方式有所差异，我们下面分别进行介绍。

UIWebView

UIWebView 控件有一个 delegate 属性，设置该属性时，需要这个对象的类实现 UIWebViewDelegate 协议。在这个协议中，有以下几个方法：

```
- (BOOL)webView:(UIWebView *)webView shouldStartLoadWithRequest:(NSURLRequest *)request navigationType:(UIWebViewNavigationType)navigationType;
- (void)webViewDidStartLoad:(UIWebView *)webView;
- (void)webViewDidFinishLoad:(UIWebView *)webView;
- (void)webView:(UIWebView *)webView didFailLoadWithError:(NSError *)error;
```

其中，UIWebView 控件每次加载请求时都会调用 - webView:shouldStartLoadWithRequest:navigationType: 方法，然后再根据这个方法的返回值来决定是否继续加载这个请求。因此，我们可以在这个方法中进行请求拦截。

WKWebView

WKWebView 控件中也有一个 navigationDelegate 属性，设置该属性的对象需要实现 WKNavigationDelegate 协议。在这个协议中也有很多方法，其中 - webView:decidePolicyForNavigationAction:decisionHandler: 方法与 - webView:shouldStartLoadWithRequest:navigationType: 方法的作用类似，我们也可以在这个方法中进行请求拦截。

(2) 方案二：JavaScript 与 WebView 相互调用

方案二相对于方案一来说，会容易理解很多，实现也比较简单。但 UIWebView 控件是一个很古老的控件，它与 H5 页面交互的能力非常有限，已无法满足 H5 迅猛发展的需求。即使在 iOS 7 推出了 JavaScriptCore 框架后，仍有诸多限制（例如，需要在页面加载完成之后才能获取到 JavaScript 的上下文对象）。因此，后面才会有 WKWebView 控件的诞生。针对

UIWebView 控件的诸多限制和不足，再加上苹果对 WKWebView 控件的大力推广和重视，我们的方案二暂且只支持 WKWebView 控件。

WKWebView 控件中有一个 WKWebViewConfiguration 类型的属性 configuration，它是 WKWebView 初始化时一些属性的集合封装。属性中有一个 WKUserContentController 类型的 userContentController 属性，通过调用它的 - addScriptMessageHandler:name: 方法可以让 JavaScript 向 WKWebView 发送信息。

综上，方法二的原理大概如下：在 WKWebView 控件初始化之后，我们通过调用 webView.configuration.userContentController 的 - addScriptMessageHandler:name: 方法注册回调，然后实现 WKScriptMessageHandler 协议中的 - userContentController:didReceiveScriptMessage: 方法，JavaScript SDK 即可通过 window.webkit.messageHandlers.<name>.postMessage(<messageBody>) 方式触发事件，我们就能在回调中接收到消息，然后从消息中解析出事件信息。

方案二实现起来相对比较简单，无需修改 UserAgent，但目前只支持 WKWebView 控件。因此，如果你的应用程序支持 iOS 8 以上，并在应用中未使用 UIWebView 控件时，可以选择这个方案。

# 2、实现步骤

完整的项目源码后续会 release 给大家。

# 3、知识点

- UIWebView
- WKWebView
- JavaScript

# 13
# App Extension

# App Extension

## 1、原理概述

App Extension 即应用程序扩展，是从 iOS 8 开始引入的一个非常重要的新特性。通过 App Extension，可以扩展应用程序的功能和内容，并允许用户在其它应用程序中或系统的某些特定地方使用，例如应用程序可以以小组件的形式出现在系统的 Today 页面中。

（1）App Extension 类型

在 iOS 系统中，支持应用程序扩展的区域称为 Extension Point（扩展点）。每个扩展点都定义了不同的使用策略及相应的接口，我们可以根据实际业务需求，选择一个合适扩展点。不同的扩展点对应了不同类型的应用程序扩展。在 iOS 系统中，主要有以下几种扩展。

Share

即分享扩展。可以使用户在不同的应用程序之间分享内容。如果你的应用程序是一个社交网络平台或者其它形式的分享平台，就可以提供一个分享扩展，让系统可以分享图片、视频、网站或者其它内容给你的用户。分享扩展可以在任意应用程序中被激活，但是开发者需要设置激活的规则，让分享扩展在合适的使用场景中出现。

Today

即 Today 扩展。可以让用户更快速、更方便的查看 App 最及时的信息。例如股票、天气、热搜等需要实时获取数据并更新展示的场景，都可以创建一个 Today 扩展。Today 扩展显示在通知中心的 Today 视图中，又被称为 Widget（小组件）。

Photo Editing

即图片编辑扩展。可以将你提供的滤镜或编辑工具嵌入到系统的照片和相机应用程序中，这样用户就可以很容易地将你的效果应用到图像和视频中了。因此，图片扩展只能在照片或相机应用的照片管理器中激活使用。

Custom Keyboard

在 iOS 8 以后，苹果允许开发者自定义键盘，提供不同的输入方式和布局，让用户在手机上安装和使用。不过自定义键盘需要用户在设置中进行配置，才能在文字输入时使用。

File Provider

如果你的应用程序能够提供一个存储文件位置，可以通过 File Provider 的功能让其它应用程序访问。其它应用程序在使用文档选择器视图控制器的时候，就可以打开存储在你的应用程序中的文件或者将文件存储到你的应用程序中。

Actions

动作扩展允许在 Action Sheet 中创建自定义动作按钮，例如允许用户为文档添加水印、向提醒事项中添加内容、将文本翻译成其它语言等等。动作扩展和分享扩展一样都可以在任意的应用程序中激活使用，同样也需要开发者进行相应的设置。

Document Provider

如果你的应用程序是给用户提供 iOS 文档的远程存储，可以创建一个 Document Provider，让用户可以直接在任何兼容的应用程序中上传和下载文档。

Audio

通过音频单元扩展，你可以提供音频效果、声音生成器和乐器，这些可以由音频单元宿主应用程序使用，并通过应用程序商店分发。

随着 iOS 系统功能的不断完善和丰富，应用程序扩展的种类也在不断增加。我们可以通过在 Xcode 的 File → New → Target 菜单中选择创建不同种类的应用程序扩展。

（2）App Extension 生命周期

应用程序扩展并不是一个独立的应用程序，它是包含在应用 Bundle 里一个独立的包，它的后缀名为 .appex。包含应用程序扩展的应用程序我们称为容器应用（Containing App），能够使用该扩展的应用被称为宿主应用（Host App）。例如，在 Safari 里使用微信的扩展，将一个网页分享到微信中，那 Safari 就是宿主应用，而微信就是容器应用。

当用户在手机中安装容器应用时，应用程序扩展也会随之一起被安装；如果容器应用被卸载，应用程序扩展也会一起被卸载。在宿主应用程序中定义了提供给扩展的上下文环境，并在响应用户操作发送请求时启动扩展。应用程序扩展通常在完成从宿主应用程序接收到的请求后不久终止。关于应用程序扩展的生命周期，可以参考下图 13-2 所示（此图摘自苹果官网）。

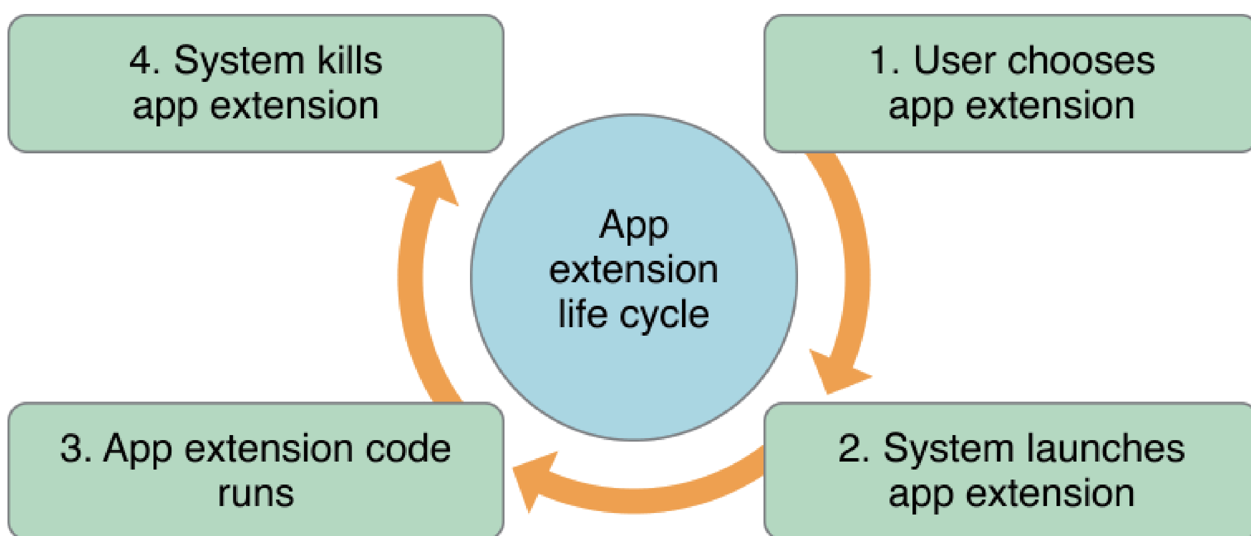



图 13-2 应用程序扩展的生命周期

关于应用程序扩展的生命周期，可简单的描述为：

- 用户选择需要使用的应用程序扩展
- 系统启动应用程序扩展
- 执行应用程序扩展的代码
- 系统终止应用程序扩展的执行

（3）App Extension 通信

在应用程序扩展启动和运行的过程中，应用程序扩展与宿主应用是如何通信的呢?在这个过程中，容器应用又是如何执行的呢?如下图 13-3 所示（此图摘自苹果官网），解释了应用程序扩展、容器应用和宿主应用之间是如何通信的。

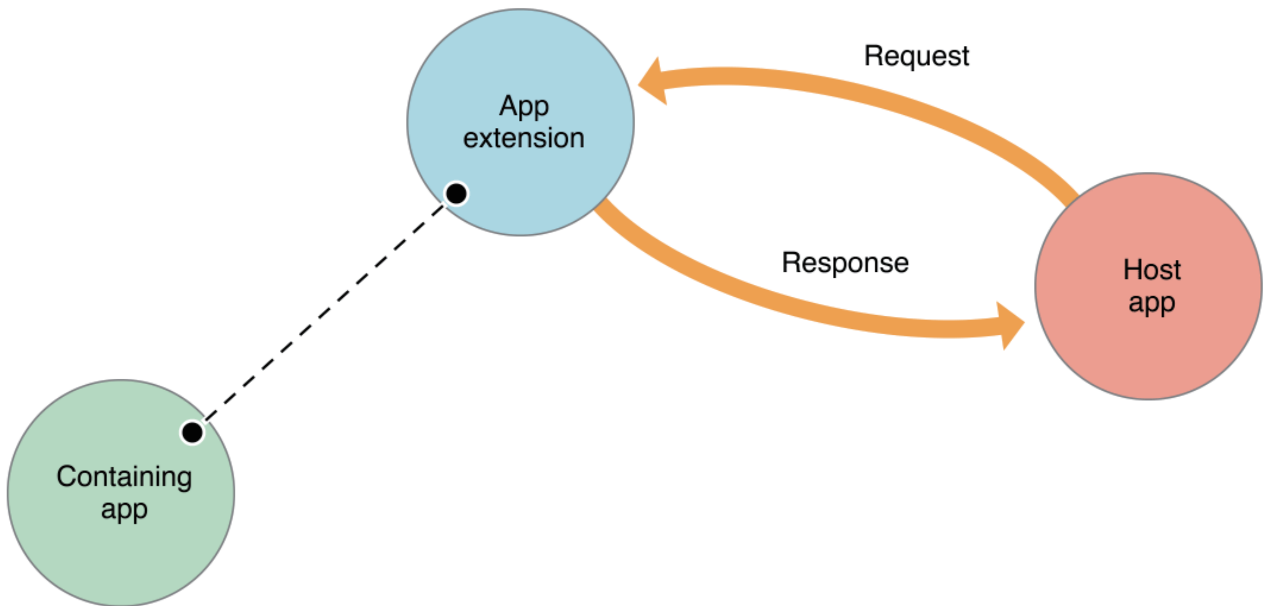



图 13-3 应用程序扩展、容器应用和宿主应用的通信

从图 13-3 可以看出，应用程序扩展和容器应用之间并没有直接的通信，一般情况下，容器应用甚至可能都不会运行，容器应用和宿主应用更是不会有任何关联。在宿主应用中打开一个应用程序扩展，宿主应用向应用程序扩展发送一个请求，即传递一些数据给应用程序扩展，应用程序扩展接收到数据后，展示应用程序扩展的界面并执行一些任务，当应用程序扩展任务执行完成，将数据处理的结果返回给宿主应用。

对于图 13-3 中的虚线部分，代表的是应用程序扩展与容器应用之间存在有限的交互方式。系统 Today 视图中的小组件，可以通过调用 NSExtensionContext 的 - openURL:completionHandler: 方法让系统打开容器应用，但这个方式只限 Today 视图中的小组件。对于任何应用程序扩展和它的容器应用，有一个私有的共享资源空间，它们都可以访问其中的文件。

应用程序扩展、容器应用及宿主应用的完整通信如下图 13-4 所示。

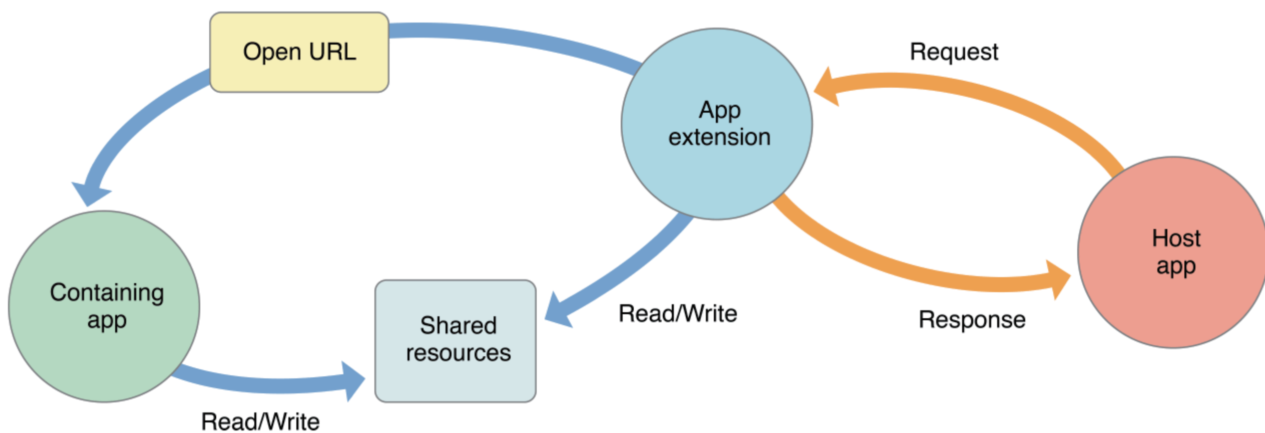



图 13-4 通信方式

(4) App Extension 埋点

通过上一节，我们了解了如何开发一个 App Extension，并通过 App Group Identifier 实现了应用程序扩展与容器应用共享数据。本节主要介绍如何在 App Extension 中进行埋点。

应用程序扩展运行的基本上是一些比较简单的任务，实现并不会特别复杂。一般情况下，还会要求应用程序扩展尽量做到简单易用。如何在应用程序扩展内采集各种事件呢?虽然我们也可以实现全埋点，但我们不建议这么做，这样不仅会增加应用程序扩展的逻辑复杂度，也会导致应用程序扩展的包体积增加。

因此，在 App Extension 中一般是采用代码埋点，然后将事件数据保存在共享资源空间里。容器应用每次启动的时候，都会尝试从共享资源空间里读取事件数据，然后进行二次加工并存入本地缓存，然后在合适的时机进行同步数据。

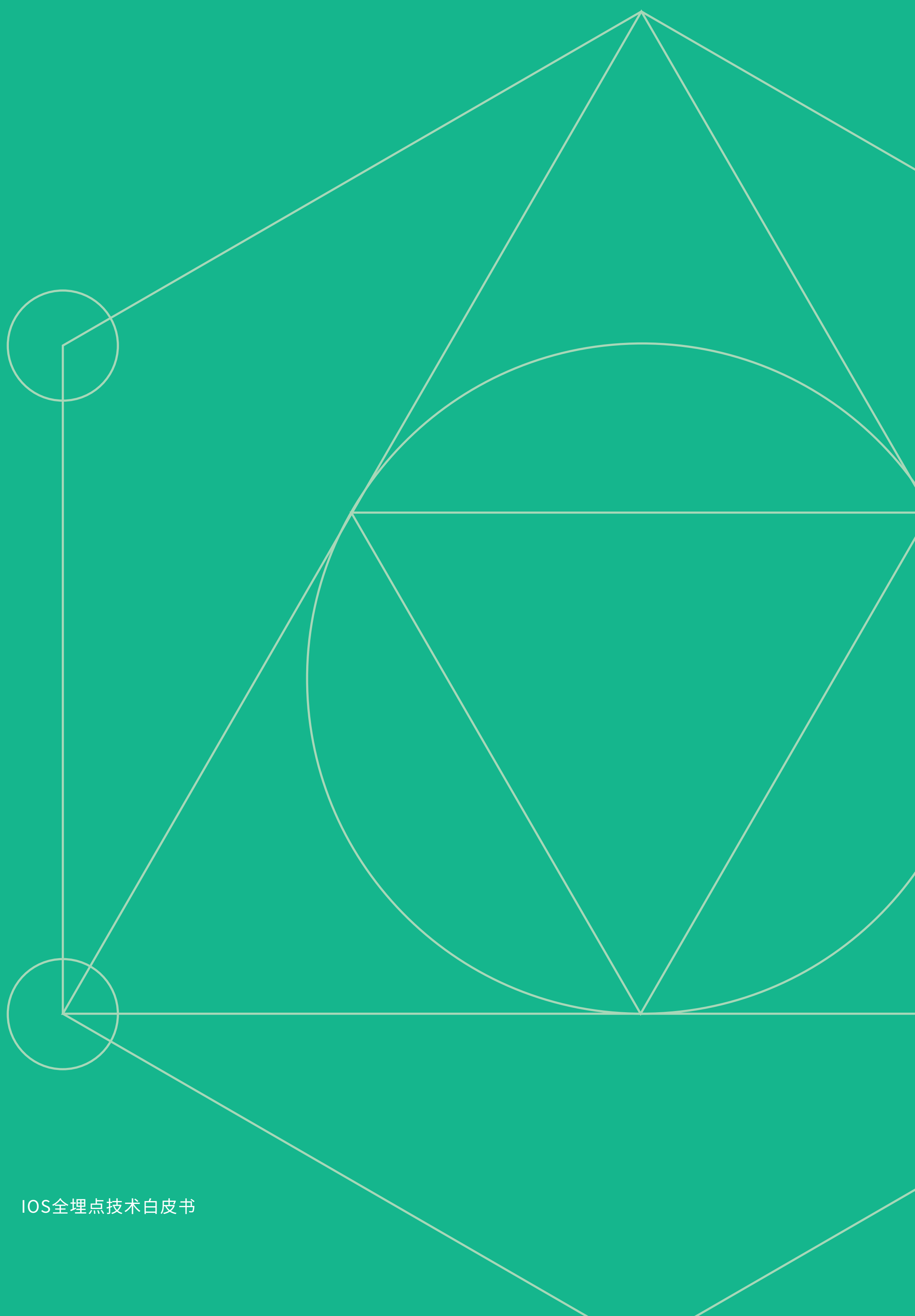
## 2、实现步骤

完整的项目源码后续会 release 给大家。

## 3、知识点

- App Extension

# 14
# React Native 全埋点



# React Native 全埋点

## 1、原理概述

本章主要介绍如何实现 React Native 的全埋点，主要是控件点击 $AppClick 事件。本章介绍的内容，会默认你有一定的 React Native 开发经验。

React Native 是由 Facebook 推出的移动应用开发框架，可以用来开发 iOS、Android、Web 等跨平台应用程序，官网为 https://facebook.github.io/react-native/。

React Native 和传统的 Hybrid 应用最大的区别就是它抛开了 WebView 控件。React Native 产出的并不是“网页应用”、“HTML5 应用”或者“混合应用”，而是一个真正的移动应用，从使用感受上和用 Objective-C 或 Java 编写的应用相比几乎是没有区分的。React Native 所使用的基础 UI 组件和原生应用完全一致。我们要做的就是把这些基础组件使用 JavaScript 和 React 的方式组合起来。React Native 是一个非常优秀的跨平台框架。

（1）React Native 全埋点

在实现 Button 控件的 $AppClick 事件全埋点之前，我们先简单的介绍一下 React Native 的事件响应机制。

（2）事件响应

在 React Native 中，触摸事件响应会涉及到 JavaScript 端和 Native 端，这里的 Native 端指的是 iOS 端，本章的内容暂不涉及 Android 部分。

查看 React Native 的源码，通过类名我们很容易找到两个与触摸事件相关的类：

- RCTTouchEvent
- RCTTouchHandler

RCTTouchEvent 类实现了 RCTEvent 协议。从触摸开始、移动到触摸结束或取消，都会创建一个 RCTTouchEvent 类的对象，用来描述触摸的各个不同阶段。在 Native 端，将触摸状态发送到 JavaScript 端的过程中，传递的也是 RCTTouchEvent 类的对象。其实，RCTTouchEvent 类的对象就是在 RCTTouchHandler 类中创建的。

RCTTouchHandler 类继承自 UIGestureRecognizer 类，也就是说 RCTTouchHandler 类其实就是一个手势识别器，它重写了触摸响应传递的以下几个方法。

```
- (void)touchesBegan:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event;
- (void)touchesMoved:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event;
- (void)touchesEnded:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event;
- (void)touchesCancelled:(NSSet<UITouch *> *)touches withEvent:(UIEvent *)event;
```

在这几个方法里，都会调用 - _updateAndDispatchTouches:eventName: 方法。在 - _updateAndDispatchTouches:eventName: 方法中，使用 RCTTouchEvent 类的对象来描述当前的触摸状态。由于 RCTTouchHandler 类也是一个手势识别器，因此需要将其添加到一个视图中才能响应触摸事件。

这个时候你会不会在想，如果我们交换了 RCTTouchHandler 类的 - _updateAndDispatchTouches:eventName: 方法就可以采集到控件的 $AppClick 事件了呢?虽然我们通过这种方法，能接收到所有的触摸事件，但是在这个方法中，我们无法知道在 JavaScript 端到底是哪个控件响应了触摸事件。因此，此种实现方案并不可取，不能满足我们实际的全埋点采集需求。

我们继续往下分析。

在 RCTTouchHandler 类的对象进行处理完成之后，会通过一系列方法将触摸事件发送到 JavaScript 端。在 JavaScript 端也实现了类似于 Native 端的触摸事件处理机制——手势响应系统。每个触摸事件都可以通过手势响应系统找到能够响应的组件，并执行响应事件。当触摸事件找到响应者时，会触发 ReactNativeGlobalResponderHandler.js 的 onChange 方法，相关代码片段如下。

```
// Module provided by RN:
var ReactNativeGlobalResponderHandler = {
 onChange: function(from, to, blockNativeResponder) {
  if (to !== null) {
   var tag = to.stateNode._nativeTag;
   ReactNativePrivateInterface.UIManager.setJSResponder(
    tag,
    blockNativeResponder
   );
  } else {
   ReactNativePrivateInterface.UIManager.clearJSResponder();
  }
 }
};
```

从上面的代码可以看出，当响应控件触摸事件的时候，JavaScript 端会调用 UIManager 中的 - setJSResponder: 方法，然后会调用 Native 端的 RCTUIManager 类中的 - setJSResponder:blockNativeResponder: 方法。这个方法的实现代码较少，参考如下。

```
/**
 * JS sets what *it* considers to be the responder. Later, scroll views can use
 * this in order to determine if scrolling is appropriate.
 */
RCT_EXPORT_METHOD(setJSResponder:(nonnull NSNumber *)reactTag
                  blockNativeResponder:(__unused BOOL)blockNativeResponder)
{
  [self addUIBlock:^(__unused RCTUIManager *uiManager, NSDictionary<NSNumber *, UIView *> *viewRegistry) {
    _jsResponder = viewRegistry[reactTag];
    if (!_jsResponder) {
      RCTLogWarn(@"Invalid view set to be the JS responder - tag %@", reactTag);
    }
  }];
}
```

这个方法有两个参数，通过第一个参数 reactTag 我们可以获取到响应者。

介绍到这里，我们已经有实现 React Native 中 Button 控件 $AppClick 事件的全埋点方案了，那就是交换 RCTUIManager 类中的 - setJSResponder:blockNativeResponder: 方法。

（3）$AppClick 事件

关于 React Native 的 $AppClick 事件，主要是通过一个交换函数来实现，该函数需要做三件事情：

- 调用原始的方法，保证 React Native 可以继续完成触摸事件的响应
- 获取触发事件响应的视图控件
- 触发 $AppClick 事件

# 2、实现步骤

完整的项目源码后续会 release 给大家。

# 3、知识点

- React Native

## 作者

王灼洲 合肥研发中心负责人
神策数据内容营销组

## 美术编辑

池亚茹 神策数据平面设计

## 联系我们

邮箱：contact@sensorsdata.cn

电话：400 650 9827

网址：www.sensorsdata.cn

关注神策数据

关注用户行为洞察研究院