(19)中华人民共和国国家知识产权局

(12)发明专利申请

(10)申请公布号 CN 112230903 A
(43)申请公布日 2021.01.15

(21)申请号 202011084482.5

(22)申请日 2020.10.12

(71)申请人 上海赛可出行科技服务有限公司
地址 200131 上海市浦东新区自由贸易试验区杨高北路2001号1幢4部位三层333室

(72)发明人 金小俊 赵化 李卫丽

(51)Int.Cl.
G06F 8/30(2018.01)
G06F 11/36(2006.01)

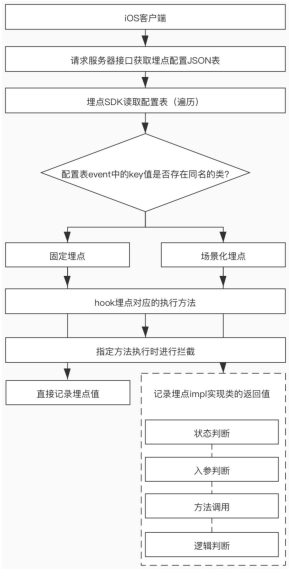
权利要求书2页 说明书17页 附图4页

(54)发明名称

一种基于iOS应用的轻量级解耦式埋点方法及装置

(57)摘要

本发明公开了一种基于iOS应用的轻量级解耦式埋点方法及装置,涉及数据采集分析技术领域,方法包括:通过维护一个JSON文件来指定埋点所在的类和方法,继而利用AOP的方式在对应的类和方法执行时动态嵌入埋点代码。本发明通过维护一个JSON文件来指定埋点所在的类和方法,继而利用AOP的方式在对应的类和方法执行时动态嵌入埋点代码;对于需要逻辑判断来确定埋点值的场景,提供hook方法的入参,以及所在类的属性值读取,根据相应的状态值设置不同的埋点;具有以下优点:支持动态下发埋点配置;物理隔离埋点代码和业务代码;插件式的埋点功能实现。



CN 112230903 A

1.一种基于iOS应用的轻量级解耦式埋点方法及装置,其特征在于,所述装置包括埋点配置JSON表和埋点SDK,具体埋点方法如下:

一、配置埋点信息

埋点配置JSON表中包含需要hook的类名class和具体的事件event信息,event中包括hook的方法和对应的埋点值;通过AOP的方式物理隔离埋点代码和业务代码,避免埋点的逻辑侵入污染业务逻辑;埋点包括固定埋点和需要逻辑判断的场景化埋点,对于场景化埋点,需要提供一个impl类来提供相应的逻辑判断;

二、固定埋点场景:

对于固定的埋点,只需要在对应的方法执行时直接记录埋点,具体方法为利用iOS系统的runtime来实现AOP,通过MethodSwizzling机制来hook相应的类和方法;为了便于检测无效的埋点,还需对hook的类和方法进行匹配校验,若类中没有对应的方法,则抛出断言;

三、场景化埋点:

场景化埋点主要为同一事件但是在多种状态或逻辑下不同埋点的情况,本发明通过提供一个protocol由埋点impl类来实现,根据不同的逻辑判断,返回对应的埋点值;埋点实现类的类名需要与埋点配置JSON中的event里的key保持一致;

四、状态判断:

根据状态量来确定埋点值;根据订单种类和订单状态来返回对应的埋点值,首先定义JSON表中同名的impl类,并遵循RJEventTracking协议;

五、入参判断:

需要根据JSON中设置的所hook方法的入参来确定埋点名称的情况,比如在订单列表中点击全部、进行中、待支付、待评价、已完成菜单项时分别埋点;被hook的方法为tripLabClickWithLabKey:其参数为UILabel,原先代码中通过Label的tag判断是点击的哪个子项,同样,可以获取到Label的入参然后据此判断,由于参数只有一个,所以可以直接取arguments第一个值;通过AOP来hook方法时,可以获取到当前hook方法所对应的实例对象和入参,在调用协议方法时,直接传给协议实现类;

六、方法调用:

和读取属性值类似,也是在不同场景下同一事件不同埋点名称的情况,但获取的状态量不是当前实例对象的,而是某个方法的返回值,这种情况下可以通过埋点SDK提供的方法调用函数来实现;

七、逻辑判断:

对于需要额外添加逻辑判断的场景,比如在订单详情页需要统计用户进入页面的查看行为,但是详情页的类型需要在网络请求后才能获取,而且该网络请求会定时触发,所以埋点hook的方法会走多次,该情况下,需要添加一个属性用来标记是否已记录埋点,故而埋点SDK需要提供动态添加属性的功能;在埋点实现impl类里面,添加额外的属性来标记是否已记录过埋点;使用addExtraProperty:defaultValue:来给当前实例动态添加属性,而extraProperty:方法则用来获取实例的某个额外属性,如果isRecorded返回YES代表已经记录过该埋点,返回nil值来忽略该次埋点;

八、动态下发埋点配置:

埋点JSON配置表可以由服务器提供接口,客户端在每次启动时通过接口获取最新埋点

配置表,从而达到动态下发的目的,客户端拿到JSON后,读取埋点信息并生效;读取埋点配置的逻辑为遍历埋点中的类和hook的方法,并检测是固定埋点还是场景化埋点,对于场景化埋点的情况查询是否有对应的埋点impl实现类,当然,还需检测JSON配置表的合法性,每个类和其中的方法是否匹配。

一种基于iOS应用的轻量级解耦式埋点方法及装置

技术领域

本发明涉及数据采集分析技术领域,特别涉及一种基于iOS应用的轻量级解耦式埋点方法及装置。

背景技术

在发展日新月异的移动互联网时代,数据扮演着极其重要的角色。埋点作为一种最简单最直接的用户行为统计方式,能够全面精确的采集用户的使用习惯以及各功能点的迭代反馈等等,有了这些数据才能更好的驱动产品的决策设计和新业务场景的规划。

目前业内最常用的埋点方法是代码埋点,代码埋点的特点是灵活,可以在相应的业务代码处根据业务逻辑和变量值判断实现场景化的埋点值记录,但其缺点也尤其明显,将埋点和业务代码耦合在了一起,造成了埋点代码对业务代码的入侵,且埋点逻辑极易对业务逻辑造成污染。另外,对于埋点代码的移除只能通过手动删除代码的方式来实现。

发明内容

本发明要解决的技术问题是克服现有技术的缺陷,提供一种基于iOS应用的轻量级解耦式埋点方法及装置,本发明通过维护一个JSON文件来指定埋点所在的类和方法,继而利用AOP的方式在对应的类和方法执行时动态嵌入埋点代码;对于需要逻辑判断来确定埋点值的场景,提供hook方法的入参,以及所在类的属性值读取,根据相应的状态值设置不同的埋点;具有以下优点:支持动态下发埋点配置;物理隔离埋点代码和业务代码;插件式的埋点功能实现。

为了解决上述技术问题,本发明提供了如下的技术方案:

本发明一种基于iOS应用的轻量级解耦式埋点方法及装置,所述装置包括埋点配置JSON表和埋点SDK,具体埋点方法如下:

一、配置埋点信息

埋点配置JSON表中包含需要hook的类名class和具体的事件event信息,event中包括hook的方法和对应的埋点值;通过AOP的方式物理隔离埋点代码和业务代码,避免埋点的逻辑侵入污染业务逻辑;埋点包括固定埋点和需要逻辑判断的场景化埋点,对于场景化埋点,需要提供一个impl类来提供相应的逻辑判断;

二、固定埋点场景:

对于固定的埋点,只需要在对应的方法执行时直接记录埋点,具体方法为利用iOS系统的runtime来实现AOP,通过MethodSwizzling机制来hook相应的类和方法;为了便于检测无效的埋点,还需对hook的类和方法进行匹配校验,若类中没有对应的方法,则抛出断言;

三、场景化埋点:

场景化埋点主要为同一事件但是在多种状态或逻辑下不同埋点的情况,本发明通过提供一个protocol由埋点impl类来实现,根据不同的逻辑判断,返回对应的埋点值;埋点实现类的类名需要与埋点配置JSON中的event里的key保持一致;

四、状态判断：

根据状态量来确定埋点值;根据订单种类和订单状态来返回对应的埋点值,首先定义JSON表中同名的impl类,并遵循RJEventTracking协议;

五、入参判断：

需要根据JSON中设置的所hook方法的入参来确定埋点名称的情况,比如在订单列表中点击全部、进行中、待支付、待评价、已完成菜单项时分别埋点;被hook的方法为tripLabClickWithLabKey:其参数为UILabel,原先代码中通过Label的tag判断是点击的哪个子项,同样,可以获取到Label的入参然后据此判断,由于参数只有一个,所以可以直接取arguments第一个值;通过AOP来hook方法时,可以获取到当前hook方法所对应的实例对象和入参,在调用协议方法时,直接传给协议实现类;

六、方法调用：

和读取属性值类似,也是在不同场景下同一事件不同埋点名称的情况,但获取的状态量不是当前实例对象的,而是某个方法的返回值,这种情况下可以通过埋点SDK提供的方法调用函数来实现;

七、逻辑判断：

对于需要额外添加逻辑判断的场景,比如在订单详情页需要统计用户进入页面的查看行为,但是详情页的类型需要在网络请求后才能获取,而且该网络请求会定时触发,所以埋点hook的方法会走多次,该情况下,需要添加一个属性用来标记是否已记录埋点,故而埋点SDK需要提供动态添加属性的功能;在埋点实现impl类里面,添加额外的属性来标记是否已记录过埋点;使用addExtraProperty:defaultValue:来给当前实例动态添加属性,而extraProperty:方法则用来获取实例的某个额外属性,如果isRecorded返回YES代表已经记录过该埋点,返回nil值来忽略该次埋点;

八、动态下发埋点配置：

埋点JSON配置表可以由服务器提供接口,客户端在每次启动时通过接口获取最新埋点配置表,从而达到动态下发的目的,客户端拿到JSON后,读取埋点信息并生效;读取埋点配置的逻辑为遍历埋点中的类和hook的方法,并检测是固定埋点还是场景化埋点,对于场景化埋点的情况查询是否有对应的埋点impl实现类,当然,还需检测JSON配置表的合法性,每个类和其中的方法是否匹配。

与现有技术相比,本发明的有益效果如下：

本发明通过维护一个JSON文件来指定埋点所在的类和方法,继而利用AOP的方式在对应的类和方法执行时动态嵌入埋点代码;对于需要逻辑判断来确定埋点值的场景,提供hook方法的入参,以及所在类的属性值读取,根据相应的状态值设置不同的埋点;具有以下优点:支持动态下发埋点配置;物理隔离埋点代码和业务代码;插件式的埋点功能实现。

附图说明

附图用来提供对本发明的进一步理解,并且构成说明书的一部分,与本发明的实施例一起用于解释本发明,并不构成对本发明的限制。在附图中：

图1是本发明的配置信息获取和无侵入-物理隔离-插件化-解耦化示意图;

图2是本发明的埋点方法总体流程图。

图3是本发明的协议编程图；

图4是本发明的impl对应多个类中的多个method方法编程图；

具体实施方式

以下结合附图对本发明的优选实施例进行说明，应当理解，此处所描述的优选实施例仅用于说明和解释本发明，并不用于限定本发明。

实施例1

如图1-4所示，本发明提供一种基于iOS应用的轻量级解耦式埋点方法及装置，装置包括埋点配置JSON表和埋点SDK，具体埋点方法如下：

一、配置埋点信息

埋点配置JSON表中包含需要hook的类名class和具体的事件event信息，event中包括hook的方法和对应的埋点值，如下所示：

```
{
    "version": "0.1.0",
    "tracking": [
        {
            "class": "RJMainViewController",
            "event": {
```

```
                    "rj_main_tracking": [
                        "tripTypeViewChangedWithIndex:",
                        "tripLabClickWithLabKey:"
                    ],
                    "user_fp_slide_click": "clickNavLeftBtn",
                    "user_fp_reflocate_click": "clickLocationBtn"
                }
            },
            {
                "class": "RJTripHistoryViewModel",
                "event": {
                    "user_mytrip_show":
"tableView:didSelectRowAtIndexPath:"
                }
            },
            {
                "class": "RJTripViewController",
                "event": {
                    "rj_trip_tracking": "callServiceEvent"
                }
            }
        ]
    }
```

简单来说就是本来埋点需要手动在该方法写入埋点代码来记录埋点值，现在通过AOP的方式物理隔离埋点代码和业务代码，避免埋点的逻辑侵入污染业务逻辑。埋点包括固定埋点和需要逻辑判断的场景化埋点，固定埋点如下所示：

```
        {
            "class": "RJTripHistoryViewModel",
            "event": {
                "user_mytrip_show":
    "tableView:didSelectRowAtIndexPath:"
            }
        }
```

RJTripHistoryViewModel为类名,tableView:didSelectRowAtIndexPath:为需要hook的该类中的方法,而user_mytrip_show则是具体的埋点值,也就是当RJTripHistoryViewModel中的tableView:didSelectRowAtIndexPath:方法执行的时候记录埋点值user_mytrip_show;

对于场景化埋点,则需要提供一个impl类来提供相应的逻辑判断。比如:

```
    {
        "class": "RJTripViewController",
        "event": {
            "rj_trip_tracking": "callServiceEvent"
        }
    }
```

上述配置表中的rj_trip_tracking为场景埋点的实现类,在该类中根据状态量返回对应的埋点值,即当callServiceEvent方法执行时会去找rj_trip_tracking这个埋点impl同名类,取该类返回的埋点值记录埋点。需要注意到是event中的key值既可以作为埋点值也可以作为impl的类名,埋点SDK会首先判断是否存在对应的类,存在即认为是impl实现类,从该类中取具体的埋点值。反之,则认为是固定埋点值。

配置表中的类名和方法名需要对应,在hook的时候会去匹配,如果发现类中不存在对应的方法,则会自动触发断言;

二、固定埋点场景:

对于固定的埋点,只需要在对应的方法执行时直接记录埋点,具体方法为利用iOS系统的runtime来实现AOP,通过Method Swizzling机制来hook相应的类和方法(实际开发中通常使用业内广泛使用的第三方框架-Aspects)

为了便于检测无效的埋点,还需对hook的类和方法进行匹配校验,若类中没有对应的方法,则抛出断言;

三、场景化埋点:

场景化埋点主要为同一事件但是在多种状态或逻辑下不同埋点的情况,比如同是联系客服的操作,在各种订单类型以及订单状态下所设置的埋点是不同的。这个情况下,本发明通过提供一个protocol(协议)由埋点impl(实现类)类来实现,根据不同的逻辑判断,返回对应的埋点值:

```
@protocol RJEventTracking<NSObject>
-(NSString*)trackingMethod:(NSString*)method instance:(id)instance arguments:(NSArray*)arguments;
@end
```

比如上文的rj_trip_tracking类需要遵循RJEventTracking协议,并根据相关逻辑判断返回对应的埋点值。

埋点实现类的类名需要与埋点配置JSON中的event里的key保持一致,因为埋点SDK会通过检测是否有同名的类来实现插件式的埋点规则。另外,一个impl可以对应多个method方法。

四、状态判断:

根据状态量来确定埋点值。还是联系客服埋点的例子,根据订单种类和订单状态来返回对应的埋点值,首先定义JSON表中同名的impl类,并遵循RJEventTracking协议:

```
#import"RJEventTracking.h"
@interface rj_trip_tracking:NSObject<RJEventTracking>
@end
```

在埋点的impl类中实现自定义埋点的协议方法

```
trackingMethod:instance:arguments:

    #import "rj_trip_tracking.h"


    @implementation rj_trip_tracking


    - (NSString *)trackingMethod:(NSString *)method
instance:(id)instance arguments:(NSArray *)arguments {
        id dataManager         = [instance property:@"dataManager"];
        NSInteger orderStatus = [[dataManager
property:@"orderStatus"] integerValue];
        NSInteger orderType    = [[dataManager property:@"orderType"]
integerValue];
        if ([method isEqualToString:@"callServiceEvent"]) {
            if (orderType == 1) {
```

```
                if (orderStatus == 1) {
                    return @"user_inbook_psgservice_click";
                } else if (orderStatus == 2) {
                    return @"user_finishbook_psgservice_click";
                }
            } else {
                return @"user_psgservice_click";
            }
        }
        return nil;
    }


    @end
```

在协议方法中,可以获取当前的实例(在这个示例下为RJTripViewController)和入参数组。订单的类型和状态是存储在RJTripViewController中的dataManager属性中的,所以可以通过埋点SDK封装好的property:方法来获取属性值,并根据属性值返回对应的埋点名称。

使用AOP方式hook类中的方法时,在截获方法运行时会同时得到方法所在类的实例对象以及方法的入参和出参,获取该实例对象的状态值(变量值)可通过KVC的方式实现。KVC(Key-value coding)键值编码,就是指iOS的开发中,可以允许开发者通过Key名直接访问对象的属性,或者给对象的属性赋值。而不需要调用明确的存取方法。这样就可以在运行时动态地访问和修改对象的属性。而不是在编译时确定。

五、入参判断:

需要根据JSON中设置的所hook方法的入参来确定埋点名称的情况。比如在订单列表中点击全部,进行中,待支付,待评价,已完成等菜单项时分别埋点。被hook的方法为tripLabClickWithLabKey:其参数为UILabel,原先代码中通过Label的tag判断是点击的哪个子项,同样,也可以获取到Label的入参然后据此判断。由于参数只有一个,所以可以直接取arguments第一个值。

```
#import "rj_main_tracking.h"
#import <UIKit/UIKit.h>

static NSString *order_types[5] = { @"user_order_all_click",
@"user_order_ongoing_click", @"user_order_unpay_click",
@"user_order_unmark_click", @"user_order_finish_click" };
@implementation rj_main_tracking

- (NSString *)trackingMethod:(NSString *)method
instance:(id)instance arguments:(NSArray *)arguments {
    if ([method isEqualToString:@"tripLabClickWithLabKey:"]) {
        UILabel *label = arguments[0];
        if (!label || label.tag > 4) {
            return nil;
        }
        return order_types[label.tag];
    } else if ([method
isEqualToString:@"tripTypeViewChangedWithIndex:"]) {
        return @"xx_ryan_jin";
    }
}

@end
```

通过AOP来hook方法时,可以获取到当前hook方法所对应的实例对象和入参(开源库Aspects亦提供此功能),在调用协议方法时,直接传给协议实现类。

六、方法调用:

和读取属性值类似,也是在不同场景下同一事件不同埋点名称的情况,但获取的状态量不是当前实例对象的,而是某个方法的返回值,这种情况下可以通过埋点SDK提供的方法调用函数来实现:

```
@interface NSObject(RJEventTracking)
-(id)performSelector:(NSString*)selector arguments:(nullable NSArray*)
arguments;
@end
```

比如获取某个页面的视图类型,而这个视图类型存储于单例对象中:

```
[RJViewTypeModel sharedInstance].viewType
```

该场景下则根据viewType的类型,来返回相应的埋点名称:

```
- (NSString *)trackingMethod:(NSString *)method
instance:(id)instance arguments:(NSArray *)arguments {
        NSString *labKey    = [instance property:@"labKey"];
        id viewTypeModel    = [NSClassFromString(@"RJViewTypeModel")
performSelector:@"sharedInstance" arguments:nil];
        NSInteger viewType = [[viewTypeModel property:@"viewType"]
```

```
integerValue];
        if (viewType == 0) {
            if ([labKey isEqualToString:@"rj_view_begin_add"]) {
                return @"user_fp_book_on_click";
            }
            if ([labKey isEqualToString:@"rj_view_end_add"]) {
                return @"user_fp_book_off_click";
            }
        }
        if (viewType == 1) {
            if ([labKey isEqualToString:@"rj_view_begin_add"]) {
                return @"user_fr_on_click";
            }
            if ([labKey isEqualToString:@"rj_view_end_add"]) {
                return @"user_fr_off_click";
            }
        }
        return nil;
    }
```

方法调用的原理是利用iOS系统的performSelector:arguments:方法,可以通过传入方法名的字符串调用任意实例的指定方法。另外,iOS支持根据字符串换取对应的类对象。

七、逻辑判断:

需要额外添加逻辑判断的场景,比如在订单详情页需要统计用户进入页面的查看行为,但是详情页的类型需要在网络请求后才能获取,而且该网络请求会定时触发,所以埋点hook的方法会走多次,该情况下,需要添加一个属性用来标记是否已记录埋点。故而埋点SDK需要提供动态添加属性的功能。

```
@interface NSObject(RJEventTracking)
-(id)extraProperty:(NSString*)property;
-(void)addExtraProperty:(NSString*)propertydefaultValue:(id)value;
@end
```

[0038] 在埋点实现impl类里面,添加额外的属性来标记是否已记录过埋点:

```
@implementation user_orderdetail_show


- (NSString *)trackingMethod:(NSString *)method
instance:(id)instance arguments:(NSArray *)arguments {
    if ([instance extraProperty:@"isRecorded"]) {
        return nil;
    }
    [instance addExtraProperty:@"isRecorded"
defaultValue:@(YES)];
    return @"user_orderdetail_show";
}


@end
```

使用addExtraProperty:defaultValue:来给当前实例动态添加属性,而extraProperty:方法则用来获取实例的某个额外属性。如果isRecorded返回YES代表已经记录过该埋点,返回nil值来忽略该次埋点。

上面示例中添加的isRecorded属性是因为埋点的需求,和业务逻辑无关,所以比较合理的方式是在埋点的插件impl类中添加,避免影响业务代码。

埋点库动态添加属性的原理也很简单,利用runtime的objc_setAssociatedObject和objc_getAssociatedObject方法来绑定属性到实例对象。

八、动态下发埋点配置:

埋点JSON配置表可以由服务器提供接口,客户端在每次启动时通过接口获取最新埋点配置表,从而达到动态下发的目的,客户端拿到JSON后,读取埋点信息并生效。

读取埋点配置的逻辑为遍历埋点中的类和hook的方法,并检测是固定埋点还是场景化埋点,对于场景化埋点的情况查询是否有对应的埋点impl实现类。当然,还需检测JSON配置表的合法性,每个类和其中的方法是否匹配。

具体,触发断言是在开发(DEBUG)模式下,APP启动后会对JSON中列出的各个类中的方法进行hook,若发现某个类中没有对应的方法名(即JSON中指定的方法名和类中实际的方法名不一致了,这种情况可能发生在开发人员将埋点的类和方法名在JSON中配置完成后,在后续的开发过程中将方法名进行了修改,但是又忘记更新JSON文件了),会触发断言,并打印出JSON中的哪个类中的哪个方法不存在,开发人员看到断言提示后对方法名进行检查

并对JSON文件进行更新。其实就是一个提示的功能,在开发环境下没有问题后(即JSON中的类名和方法名都存在且匹配),再递交给测试人员进行埋点测试。线上(Release)环境下,即使有断言也不会被触发(因为断言只在DEBUG下生效),其实理论上触发断言的类名和方法名错误的问题在线上情况下是不会存在的(除非开发人员在开发的时候触发了断言但是不去修改,但是不修改的话断言会一直触发,断言一触发程序就无法执行,所以开发人员只能修改而不能无视)。

协议名是规定好的,协议的实现是自定义的,用来返回场景化状态下的埋点值,执行方法是协议的一个参数。详细请参见附图3;

对于附图3,协议名是规定好的,只有一个。协议的实现是自定义的,用来返回场景化状态下的埋点值,执行(hook的)方法是协议的其中一个参数;

比如,在附图3中的JSON中,RJMainViewController为类名,tripTypeViewChangedWithIndex:和tripLabClickWithLabKey:,表示需要hook RJMainViewController这个类中的tripTypeViewChangedWithIndex:和tripLabClickWithLabKey这两个方法,当这两个方法执行时判断rj_main_tracking是不是一个存在的类(代码中是否有rj_main_tracking.h和rj_main_tracking.m这两个文件),如果没有这个类,则rj_main_tracking就是埋点值,当执行这两个方法的时候直接记录[rj_main_tracking]这个字符串值作为埋点值;如果存在这个类,需要判断是否是一个有效的埋点impl类,判断是否有效,即判断它是否实现(遵循)了RJEventTracking协议,所有的埋点impl类都必须实现RJEventTracking协议,协议的方法声明为:

```
@protocol RJEventTracking<NSObject>
-(NSString*)trackingMethod:(NSString*)method instance:(id)instance arguments:(NSArray*)arguments;
@end
```

这个协议的入参为方法名,当前类的实例对象,以及参数,impl类根据这些参数,做相应的逻辑场景判断,最后返回对应的埋点值(返回值)。比如当前这个例子,method会是tripTypeViewChangedWithIndex:或者tripLabClickWithLabKey:,instance是RJMainViewController的实例对象,arguments是tripTypeViewChangedWithIndex:或者tripLabClickWithLabKey:的参数值。

还是上面的例子,比如创建rj_main_tracking这个类

```
#import"RJEventTracking.h"
@interface rj_main_tracking:NSObject<RJEventTracking>
@end
```

在.h头文件中,标记遵循RJEventTracking协议,然后在.m中实现

```
-(NSString*)trackingMethod:(NSString*)method instance:(id)instance arguments:(NSArray*)arguments;
```

这个方法。具体代码如下:

```
#import "rj_main_tracking.h"
#import <UIKit/UIKit.h>

static NSString *order_types[5] = { @"user_order_all_click",
@"user_order_ongoing_click", @"user_order_unpay_click",
@"user_order_unmark_click", @"user_order_finish_click" };

@implementation rj_main_tracking

- (NSString *)trackingMethod:(NSString *)method
instance:(id)instance arguments:(NSArray *)arguments {
    if ([method isEqualToString:@"tripLabClickWithLabKey:"]) {
        UILabel *label = arguments[0];
        if (!label || label.tag > 4) {
            return nil;
        }
        return order_types[label.tag];
    } else if ([method
isEqualToString:@"tripTypeViewChangedWithIndex:"]) {
        return @"xx_ryan_jin";
    }
}

@end
```

在协议方法中,根据method判断当前是哪个方法,tripTypeViewChangedWithIndex:还是tripLabClickWithLabKey:,以及这些方法的入参;根据instance可以拿到RJMainViewController的成员变量在运行的值。最终利用这些信息值结合相应的逻辑判

断,返回埋点值。

一个impl可以对应多个类中的多个method方法。作用是为了减少埋点代码文件的数量。具体参考附图4,在附图4中严格的说法其实是一个impl可以对应多个类中的多个method方法。作用是为了减少埋点代码文件的数量。

比如,可以在JSON指定RJMainViewController和RJConfirmViewController这两个类中的tripTypeViewChangedWithIndex:和tripLabClickWithLabKey:方法,以及clickConfirmBtn和confirmViewWillAppear方法,对应的埋点impl类都是rj_main_tracking,这样rj_main_tracking在实现协议方法的时候,就需要判断当前是哪个类的哪个方法,然后再做相应的逻辑判断

```
@implementation rj_main_tracking


- (NSString *)trackingMethod:(NSString *)method
instance:(id)instance arguments:(NSArray *)arguments{
    NSString * classStr = NSStringFromClass([instance class]);
    if ([classStr isEqualToString:@"RJMainViewController"]) {
        if ([method
isEqualToString:@"tripTypeViewChangedWithIndex:"]) {
            return @"xxx";
        }
        if ([method isEqualToString:@"tripLabClickWithLabKey:"])
{
            return @"xxx";
        }
    }
    if ([classStr isEqualToString:@"RJConfirmViewController"]) {
        if ([method isEqualToString:@"clickConfirmBtn"]) {
            return @"xxx";
        }
        if ([method isEqualToString:@"confirmViewWillAppear"]) {
            return @"xxx";
        }
    }
    return @"";
}
```

当然也可以不同的类指定不同的impl类,这样的话impl的实现类数量就会相对比较多,建议的做法是将一组具有关联功能的类的impl都指定成同一个类

补充异常情况处理逻辑:

1.JSON的中的方法名和代码中的方法名不匹配;

比如由于笔误在JSON中写入了错误的方法名。实际项目中JSON中的方法名会很多，如果单纯依靠人工检查会非常耗费精力。本发明的解决方法是在APP启动时HOOK JSON中对应的类和方法，在HOOK之前依次检测JSON中指定的类中是否包含相应的方法名，如果类中没有对应的方法名，则触发断言，并上报类名和方法名。判断的方法是通过字符串名生成类：

Class c＝NSClassFromString(classname)；

之后检测类中是否有对应的类方法和示例方法两种情况，如果都没有，则认为类中不存在JSON中指定的方法

BOOL respond＝[c respondsToSelector:sel]||[c instancesRespondToSelector:sel];

NSString*err＝[NSString stringWithFormat:@"<SCReportTracking>-no specified method:%@found on class:%@,please check",method,class];

NSAssert(respond,err);//触发断言

2.对于固定埋点(非场景化)有遗漏或错误

直接更新(添加/修改/删除)JSON中的相应类名，方法名或埋点值后再重新下发至客户端

3.对于引起严重问题的场景化埋点内容

通过在JSON中删除对应的内容后再下发至客户端，以实现对应的容灾和容错。

与现有技术相比，本发明的有益效果如下：

本发明通过维护一个JSON文件来指定埋点所在的类和方法，继而利用AOP的方式在对应的类和方法执行时动态嵌入埋点代码；对于需要逻辑判断来确定埋点值的场景，提供hook方法的入参，以及所在类的属性值读取，根据相应的状态值设置不同的埋点；具有以下优点：支持动态下发埋点配置；物理隔离埋点代码和业务代码；插件式的埋点功能实现。

最后应说明的是：以上所述仅为本发明的优选实施例而已，并不用于限制本发明，尽管参照前述实施例对本发明进行了详细的说明，对于本领域的技术人员来说，其依然可以对前述各实施例所记载的技术方案进行修改，或者对其中部分技术特征进行等同替换。凡在本发明的精神和原则之内，所作的任何修改、等同替换、改进等，均应包含在本发明的保护范围之内。

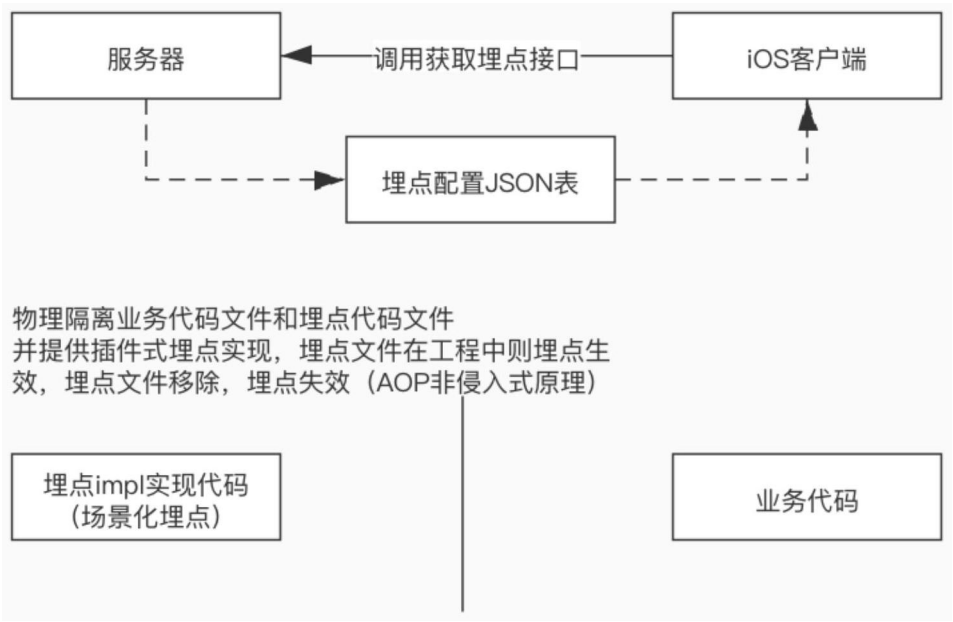

服务器
调用获取埋点接口
iOS客户端
埋点配置JSON表
物理隔离业务代码文件和埋点代码文件
并提供插件式埋点实现，埋点文件在工程中则埋点生效，埋点文件移除，埋点失效（AOP非侵入式原理）
埋点impl实现代码
（场景化埋点）
业务代码

图1

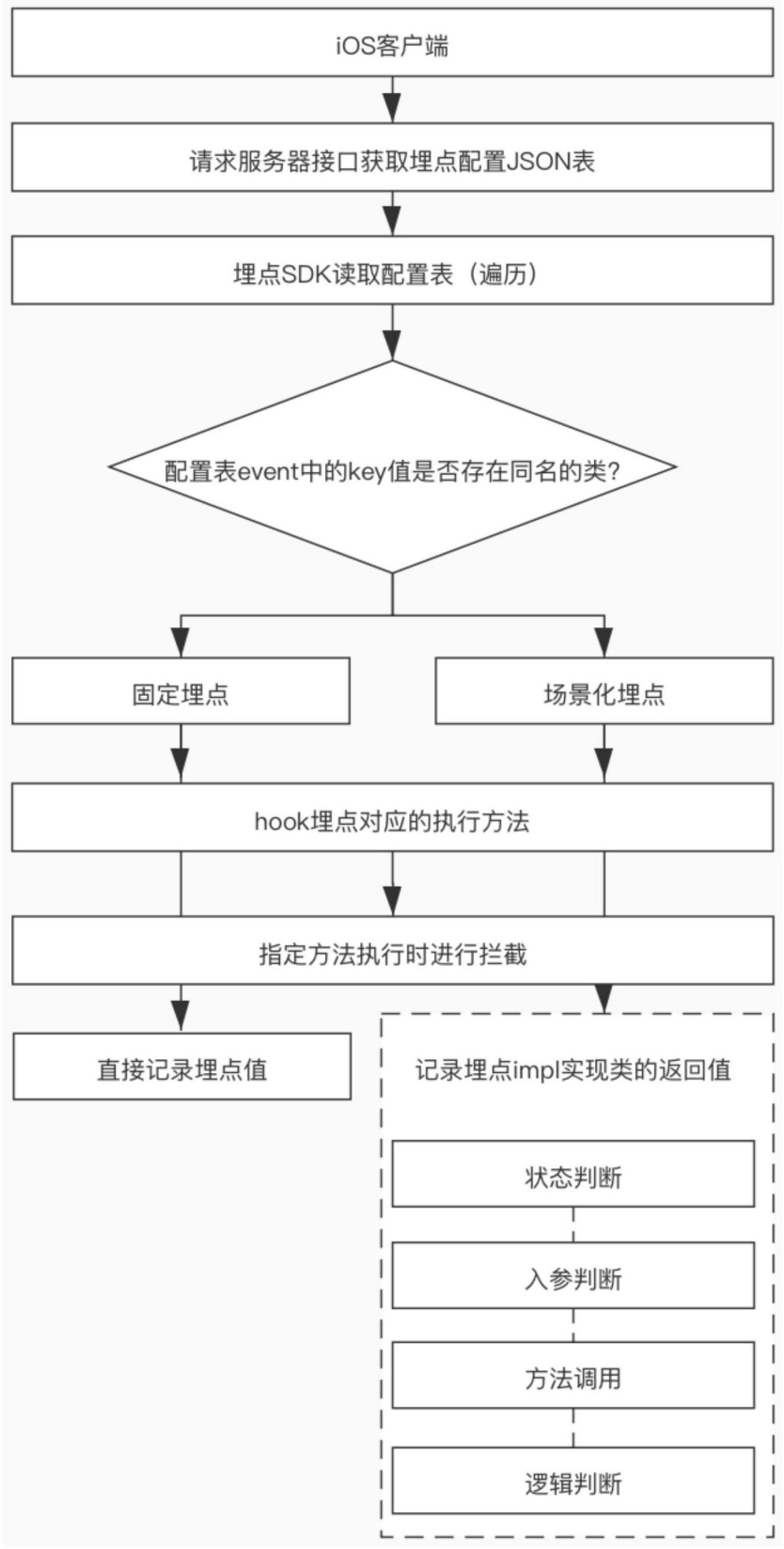

iOS客户端
请求服务器接口获取埋点配置JSON表
埋点SDK读取配置表（遍历）
配置表event中的key值是否存在同名的类?
固定埋点
场景化埋点
hook埋点对应的执行方法
指定方法执行时进行拦截
直接记录埋点值
记录埋点impl实现类的返回值
状态判断
入参判断
方法调用
逻辑判断

图2


{
"version": "0.1.0",
"tracking": [
{
"class": "RJMainViewController",
类名
"event": {
"rj_main_tracking": [
"tripTypeViewChangedWithIndex:",
"tripLabClickWithLabKey:"
方法名
],
"user_fp_slide_click": "clickNavLeftBtn",
"user_fp_reflocate_click": "clickLocationBtn"
}
},
{
"class": "RJTripHistoryViewModel",
"event": {
"user_mytrip_show": "tableView:didSelectRowAtIndexPath:"
}
},
{
"class": "RJTripViewController",
"event": {
"rj_trip_tracking": "callServiceEvent"
}
}
]
}

图3


{
"version": "0.1.0",
"tracking": [
{
"class": "RJMainViewController",
"event": {
"rj_main_tracking": [
"tripTypeViewChangedWithIndex:",
"tripLabClickWithLabKey:",
],
"user_fp_msg_click": "SaicMainViewClickNavRightBtn",
"user_fp_show": "mainViewViewWillAppear",
}
},
{
"class": "RJConfirmViewController",
"event": {
"rj_main_tracking": [
"clickConfirmBtn",
"confirmViewWillAppear",
]
}
}]
}

图4