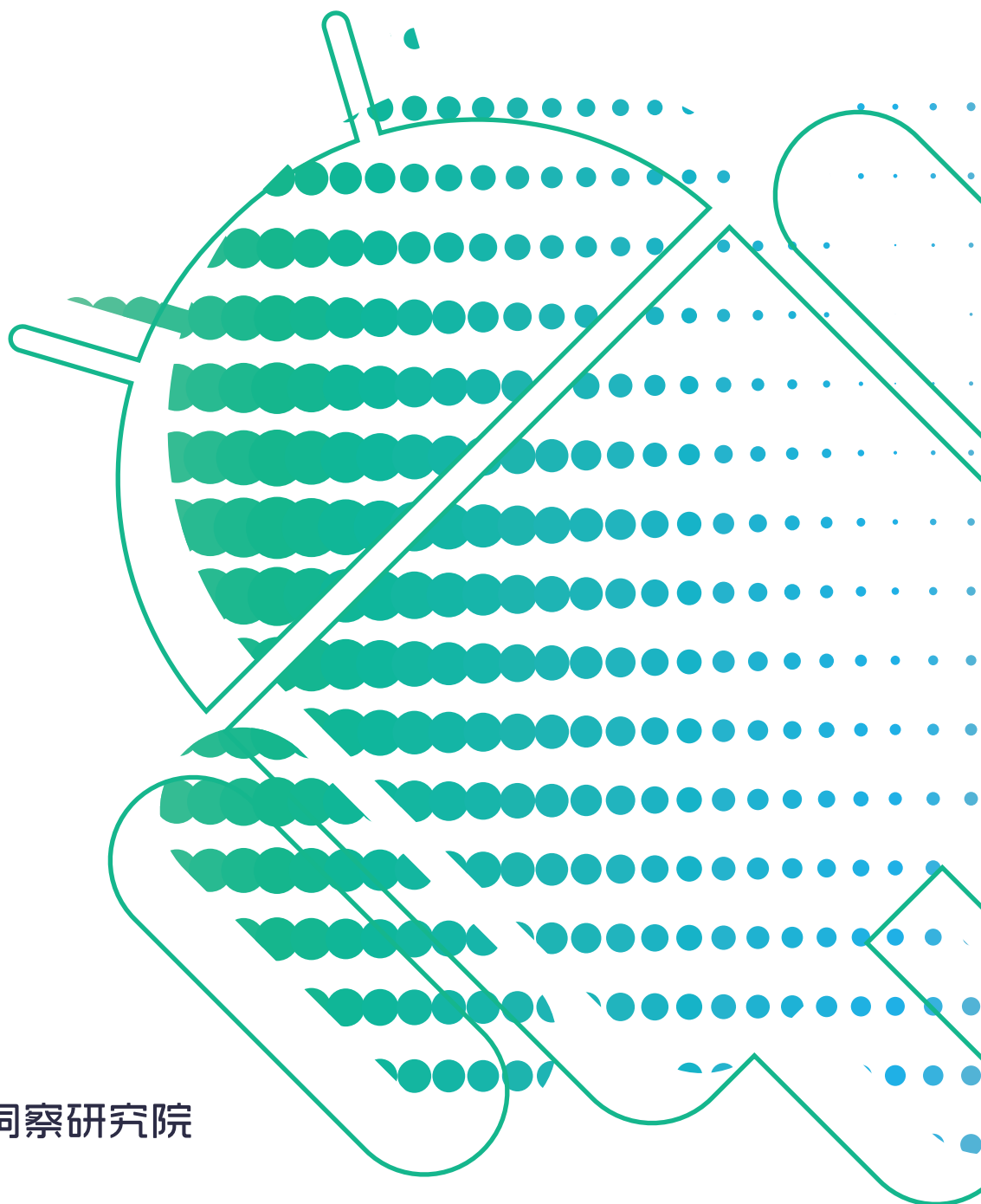


Android 全埋点技术白皮书

ANDROID AUTOTRACK

TECHNOLOGY WHITEPAPER



目录

| | | |
|----|--|----|
| 01 | 全埋点概述 | 02 |
| 02 | \$AppViewScreen 全埋点 | |
| | Application.ActivityLifecycleCallbacks | 05 |
| | 原理概述 | 06 |
| | 实现步骤 | 06 |
| | 缺点 | 06 |
| | 知识点 | 06 |
| 03 | \$AppStart 、\$AppEnd 全埋点 | |
| | 原理概述 | 08 |
| | 实现步骤 | 09 |
| | 知识点 | 09 |
| 04 | \$AppClick 全埋点 | |
| | 一. 代理 View.OnClickListener | 11 |
| | 二. 代理 Window.Callback | 15 |
| | 三. 代理 View.AccessibilityDelegate | 17 |
| | 四. AspectJ | 20 |
| | 五. ASM | 24 |
| | 六. Javassist | 32 |
| | 七. AST | 37 |

01

全埋点概述

全埋点概述

全埋点，也叫无埋点、无码埋点、自动埋点。全埋点是指预先收集用户的所有行为数据，然后再根据实际分析需求从中提取行为数据。

全埋点采集的事件主要包括下面四种：

- **\$AppStart 事件**：指 App 启动，包括冷启动和热启动。
- **\$AppEnd 事件**：指 App 退出，包括正常退出、进入后台、App 崩溃、App 被强杀。
- **\$AppViewScreen 事件**：指 App 页面浏览，对于 Android 来说，就是指切换 Activity。
- **\$AppClick 事件**：指 App 控件被点击。

在采集的这四种事件当中，最重要并且采集难度最大的是 \$AppClick 事件。所以，全埋点基本上也都是围绕着如何采集 \$AppClick 事件。

全埋点的整体解决思路，就是要找到那个被点击的 View 的点击处理逻辑（也叫原处理逻辑），然后利用一定的技术原理，对原处理逻辑进行“拦截”，或者在原处理逻辑的前面或者后面“插入”相应的埋点代码，从而达到自动埋点的效果。

至于如何做到自动“拦截”View 的原点击处理逻辑，一般都是参考 Android 系统 View 点击事件处理机制来进行的。至于如何做到自动“插入”埋点代码，基本上都是参考编译器对 Java 代码的处理流程来进行的，即：

JavaCode --> .java --> .class --> .dex

选择在不同的处理阶段“插入”代码，其所用的技术或者原理也不尽相同，所以全埋点的解决方案也是多种多样的。

面对这么多的全埋点方案，我们究竟该如何选择呢？

在选择全埋点的方案时，需要从效率、兼容性、扩展性等方面综合考虑。

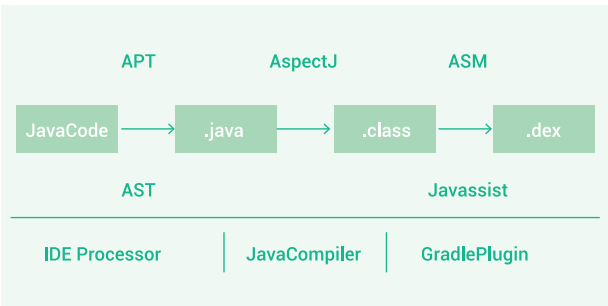
• 效率

全埋点的基本原理，其实就是利用某些技术对某些方法（View 被点击时的处理逻辑）进行代理（或者叫 Hook），或者“插入”代码。按照“在什么时候去代理或者插入代码”这个条件来区分的话，Android 全埋点技术可以大致分为下面两种方式：

静态代理

所谓静态，就是指通过 Gradle Plugin 在编译期间“插入”或者修改代码（.class 文件）。比如 AspectJ、ASM、javassist、AST 等方案均是这种方式，我们后面介绍的第四种方案到第七种方案均属于静态代理。

这几种方式处理的时机可以参考下图：



动态代理

所谓动态，就是指在代码运行的时候去进行代理。比如我们比较常见的代理 View.OnClickListener、Window.Callback、View.AccessibilityDelegate 等方案均是这种方式。我们后面介绍的第一种方案到第三种方案均属于动态代理。

不同的方案，其处理、运行效率也各不相同，同时对 App

的侵入程度以及对 App 的整体性能的影响也各不相同。整理上来说，静态代理明显优于动态代理，这是因为静态代理的“动作”是在 App 的编译阶段处理的，不会对 App 的正常业务（App 运行时）逻辑有太大的影响。

• 兼容性

随着 Android 生态系统的快速发展，不管是 Android 系统本身，还是与 Android App 开发相关的组件和技术，都在飞速发展快速迭代，从而也给研发全埋点方案带来一定的难度。比如不同的 Android App 有不同的开发语言（Java、Kotlin）、也有不同的 Java 版本（Java7、Java8）、也有不同的开发 IDE（eclipse、Android Studio）、更有不同的开发方式（原生开发、H5、混合开发）、不同的第三方开发框架（React Native、APICloud、Weex）、不同的 Gradle 版本、以及 Lambda、D8、Instant Run、DataBinding、Fragment 等，都会给全埋点带来很多兼容性方面的问题。

• 扩展性

随着业务的发展和数据分析需求的不断提高，即使对于全埋点，也提出了更高的采集要求。一方面要求能全部自动埋点（采集的范围），同时又要求能有更精细化的采集控制（采集自定义）。比如，如何给某个控件添加自定义属性？如果不想采集某个控件的点击事件如何处理？如果不想采集某种控件类型（ImageView）的点击事件如何处理？如果某个页面（Activity）上所有控件的点击事件都不想采集如何处理？.....

任何一种全埋点的方案，都有其优点和缺点。没有普适的完美方案。针对不同的应用场景，选择最合适的数据采集方案即可。

02

\$AppViewScreen 全埋点

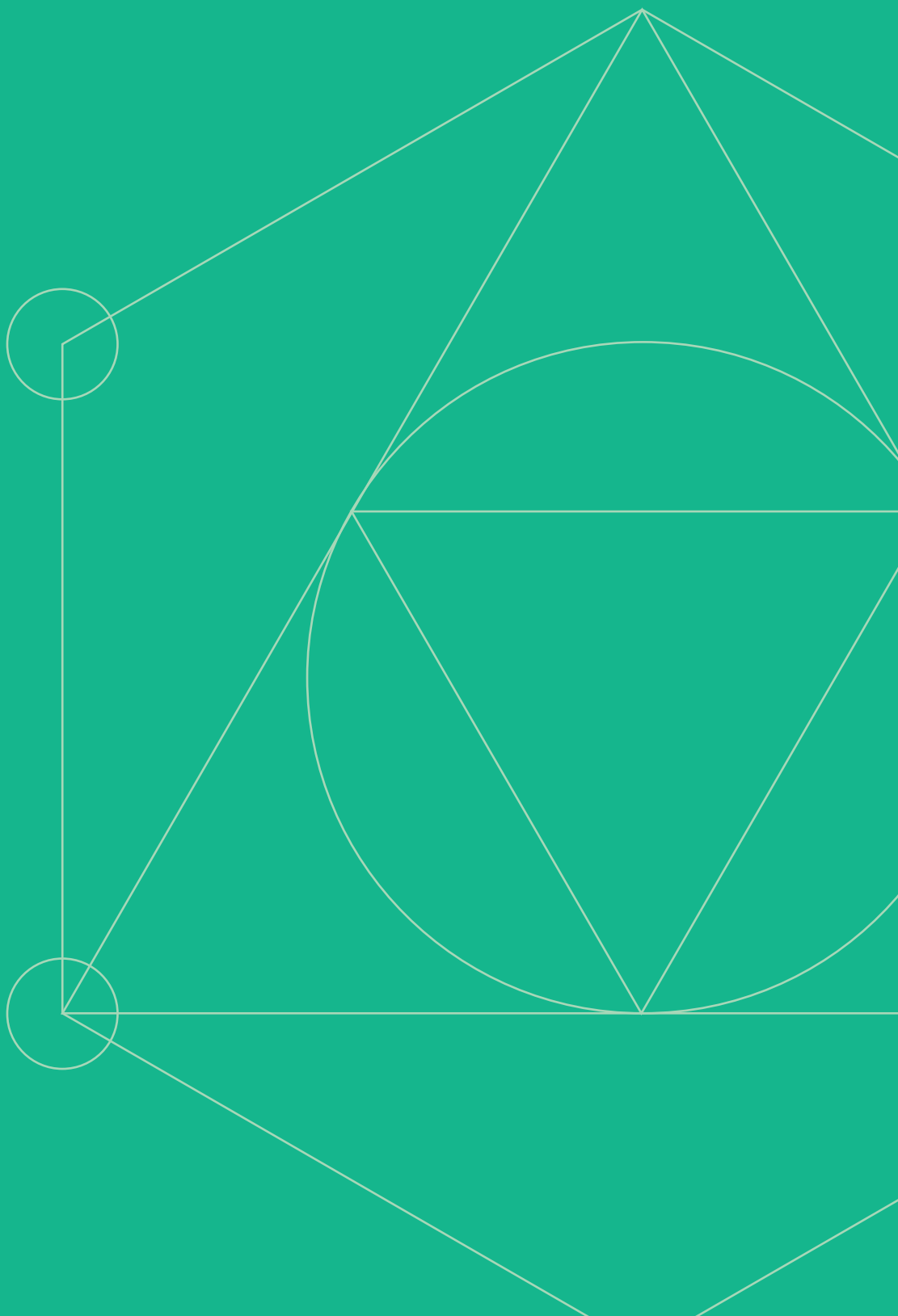
Application.ActivityLifecycleCallbacks

原理概述

实现步骤

缺点

知识点



\$AppViewScreen 全埋点

\$AppViewScreen 事件，即页面浏览事件。在 Android 系统中，页面浏览，其实就是指切换不同的 Activity。那对于一个 Activity，哪个生命周期执行了代表该页面显示出来了呢？通过对 Activity 的生命周期了解可知，其实就是 onResume 生命周期。

Application.ActivityLifecycleCallbacks

ActivityLifecycleCallbacks 是 Application 的一个内部接口，从 API 14 开始提供的。Application 通过此接口提供了一套回调方法，用于让开发者可以对 Activity 的所有生命周期事件进行集中处理（或者叫监控）。可以通过 application.registerActivityLifecycleCallback 注册 ActivityLifecycleCallbacks。

Application.ActivityLifecycleCallbacks 接口定义如下：

```
public interface ActivityLifecycleCallbacks {  
    void onActivityCreated(Activity activity, Bundle savedInstanceState);  
  
    void onActivityStarted(Activity activity);  
  
    void onActivityResumed(Activity activity);  
  
    void onActivityPaused(Activity activity);  
  
    void onActivityStopped(Activity activity);  
  
    void onActivitySaveInstanceState(Activity activity, Bundle outState);  
  
    void onActivityDestroyed(Activity activity);  
}
```

以 onResume(Activity activity) 为例，如果注册了 ActivityLifecycleCallbacks，Android 系统会先回调 ActivityLifecycleCallbacks 的 onActivityResumed 方法，然后再执行 Activity 本身的 onResume 函数（请注意这个调用顺序）。通过 registerActivityLifecycleCallback 方法名中的“register”字样可以知道，一个 Application 是可以 register 多个 ActivityLifecycleCallbacks 的，通过 registerActivityLifecycleCallback 的内部实现也可以证实这一点。

```
public void registerActivityLifecycleCallbacks  
(ActivityLifecycleCallbacks callback) {  
    synchronized (mActivityLifecycleCallbacks)  
    {  
        mActivityLifecycleCallbacks.add(callback);  
    }  
}
```

原理概述

在应用程序自定义的 Application 对象的 onCreate() 方法中初始化埋点 SDK，并传入当前的 Application 对象。SDK 拿到 Application 对象之后，通过 registerActivityLifecycleCallback 方法注册 Application.ActivityLifecycleCallbacks。这样 SDK 就能对 App 中所有的 Activity 的生命周期事件进行集中处理（监控）了。在注册的 Application.ActivityLifecycleCallbacks 的 onActivityResumed 回调方法中，我们可以拿到当前正在显示的 Activity 对象，然后调用 SDK 的相关接口触发页面浏览事件即（\$AppViewScreen）。

实现步骤

完整的项目源码后续会 release 给大家。

缺点

注册 Application.ActivityLifecycleCallbacks 要求 API 14+。

知识点

Application.ActivityLifecycleCallbacks

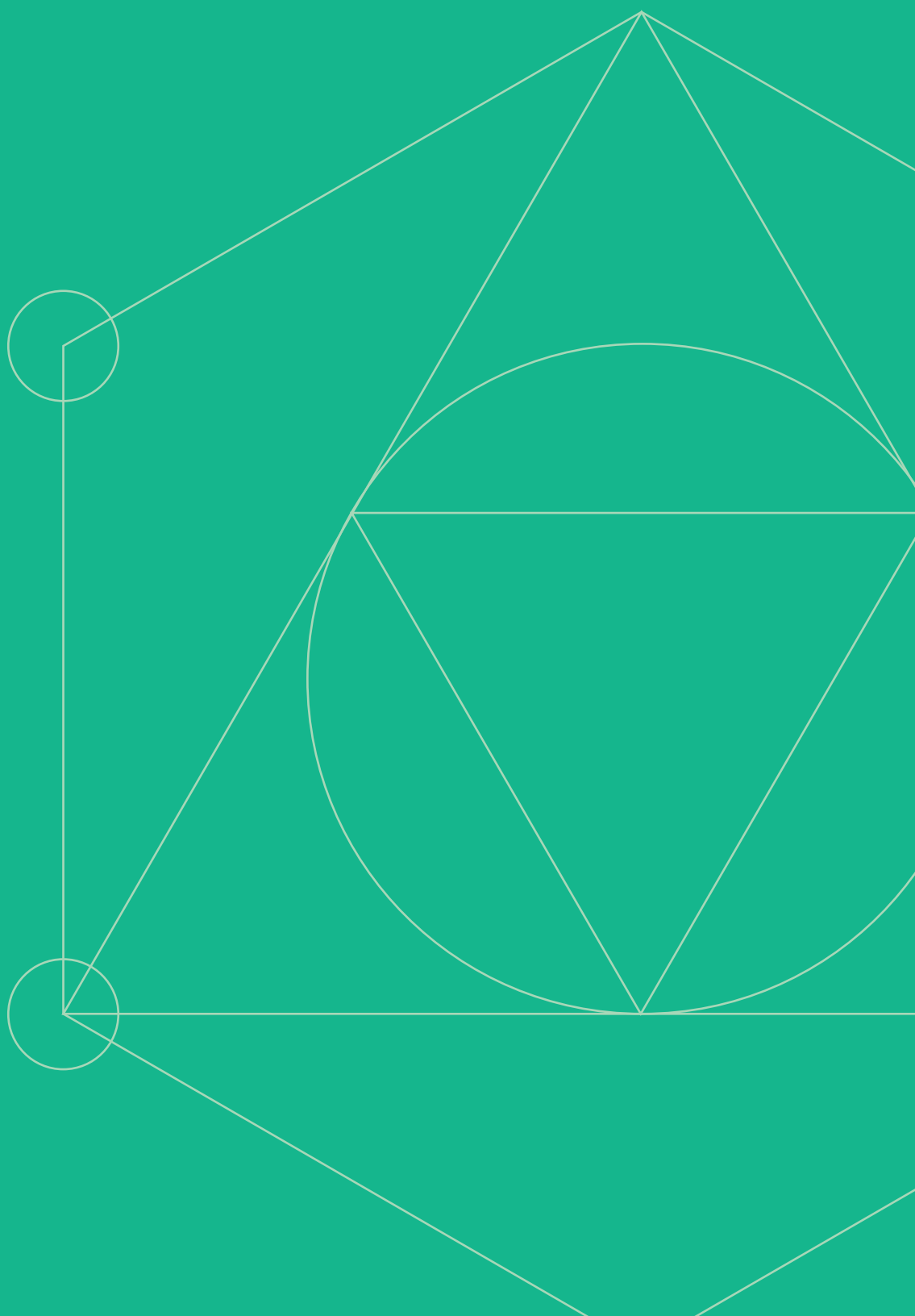
03

\$AppStart、\$AppEnd全埋点

原理概述

实现步骤

知识点



\$AppStart、\$AppEnd全埋点

对于 \$AppStart 和 \$AppEnd 而言，归根结底就是判断当前 App 是处于前台还是处于后台。而 Android 系统本身没有给 App 提供相关的接口来判断这些状态，所以我们只能借助其它方式来间接判断。

目前，业界也有很多种方法来判断一个 App 是处于前台还是后台，以 Github 上的一个开源项目为例：<https://github.com/wenmingvs/AndroidProcess>
这个开源项目目前提供了 6 种方案：

| 方案 | 含义 | 需要权限 | 特点 |
|-----|----------------------------|------|------------|
| 方案一 | RunningTask | 否 | 5.0 此方法被废弃 |
| 方案二 | RunningProcess | 否 | 无 |
| 方案三 | ActivityLifecycleCallbacks | 否 | 简单、代码量少 |
| 方案四 | UsageStatsManager | 是 | 需要用户手动授予权限 |
| 方案五 | 无障碍服务 | 否 | 需要用户手动授予权限 |
| 方案六 | 读取 /proc 目录下的信息 | 是 | 效率 |

关于这 6 种方案详细源码，可以参考开源项目。

以上这 6 种方案，各有优缺点，但同时，都无法解决下面两个问题：

- App 有多个进程如何判断？
- App 崩溃或者被强杀了如何判断？

原理概述

对于多进程间的数据共享问题，我们采用 ContentProvider 机制来解决。一方面 ContentProvider 是基于 Binder 机制封装的系统组件，目的就是解决跨进程的数据共享问题。另一方面，Android 系统提供了针对 ContentProvider 的数据回调监听，即 ContentObserver，这样就更加能满足跨进程间的数据通信。

一般情况下，针对跨进程数据共享采用的是 ContentProvider + SQLite 方案，但是基于我们的实际情况，使用 SQLite 数据库存储一些简单数据、标记位明显太过重量级了。通常在 Android 系统中，针对一些比较简单数据存储，一般是采用 SharedPreferences 进行快速读写。所以在这里我们采用的跨进程数据共享实现方式是基于 ContentProvider + SharedPreferences 的方案。

对于 App 崩溃或者应用进程被强杀的场景，我们引入了 Session 的概念。即：对于一个 App，当一个页面退出了，如果 30s 之内没有新的页面打开，我们就认为 App 处于后台了；当一个页面显示了，如果与上一个页面退出时间的间隔超过 30s，我们就认为 App 重新处于前台了。

我们首先注册一个 ActivityLifecycleCallbacks 回调，来监听应用程序内所有 Activity 的生命周期。处理业务时涉及到到标记位的保存以及跨进程间的数据通信，我们采用 ContentProvider + SharedPreferences 的方式实现进程间数据共享，同时注册 ContentObserver 来监听跨进程间的数据通信。

下面分两种情况进行处理：

在页面退出的时候 (onPause)，我们启动一个倒计时 30s 定时器，如果 30s 之内没有新的页面显示，则触发 \$AppEnd 事件；如果有新的页面进来，我们存储一个标记位来标记新页面进来。这里需要注意的是，由于 Activity 之间可能是跨进程的，所以标记位需要实现进程间的共享，即通过 ContentProvider + SharedPreferences 进行存储。然后通过 ContentObserver 监听到新页面进来的标记位改变，然后取消定时器。如果 30s 之内没有新的页面进来（比如用户按 Home 键 / 返回键退出 App、App 崩溃、App 被强杀），我们会下次启动的时候补发一个 \$AppEnd 事件。

在下次页面启动的时候 (onStart)，我们判断一下与上个页面的退出时间间隔是否超过了 30s，如果没有超过 30s，则无需补发 \$AppEnd 事件，直接触发 \$AppScreen 事件。接下来判断是否已触发 \$AppEnd 事件的标记位，如果标记位为 true，则触发 \$AppStart 事件，反之不触发；如果超过了 30s，我们判断一下是否已经触发了 \$AppEnd 事件，如果没有则先触发 \$AppEnd 事件，然后再触发 \$AppStart 和 \$AppScreen 事件。

实现步骤

完整的项目源码后续会 release 给大家。

知识点

ContentProvider

04

\$AppClick全埋点

一、代理 View.OnClickListener

二、代理 Window.Callback

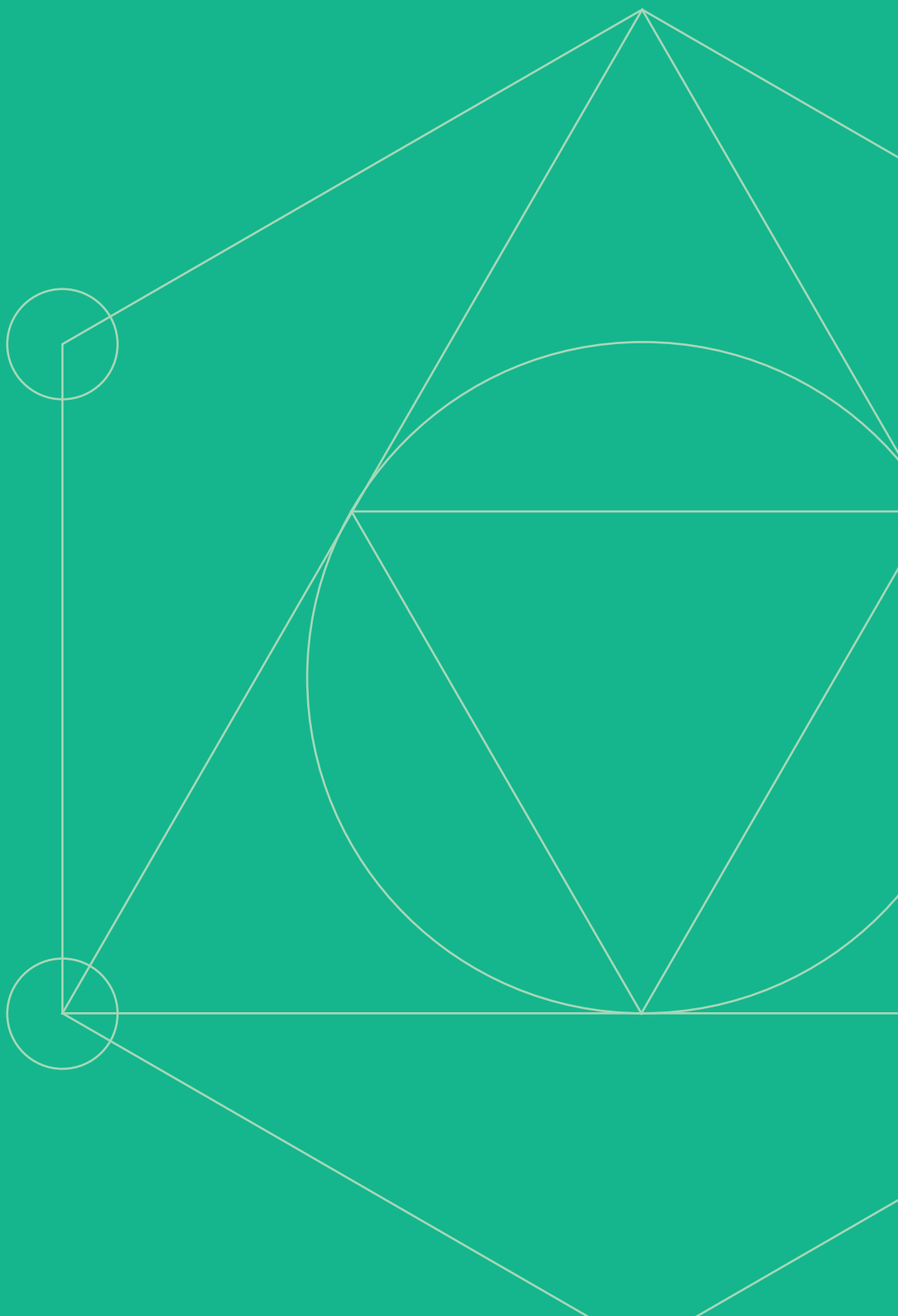
三、代理 View.AccessibilityDelegate

四、AspectJ

五、ASM

六、Javassist

七、AST



一、代理 View.OnClickListener

android.R.id.content

android.R.id.content 对应的 View 是一个 FrameLayout，只有一个子元素，就是我们平时开发的时候，在 onCreate 方法中通过 setContentView 设置的 View。也即，当我们在 layout 文件中设置一个布局文件时，实际上该布局会被一个 FrameLayout 容器所包含，这个 FrameLayout 容器的 android:id 属性值就是 android.R.id.content。

需要引起我们注意的是，在不同的 SDK 版本下，android.R.id.content 所指的显示区域也是有所不同的。具体的差异如下：

- 在 SDK 14+ (Native ActionBar)，该显示区域指的是 ActionBar 下面的那部分
- 在 Support Library Revision lower than 19，使用 AppCompat，则显示区域包含 ActionBar
- 在 Support Library Revision 19 (or greater)，使用 AppCompat，则显示区域不包含 ActionBar，即与第一种情况相同

所以如果不使用 Support Library 或使用 Support Library 的最新版本，则 android.R.id.content 所指的区域都是 ActionBar 以下的内容。

原理概述

在应用程序自定义的 Application 对象的 onCreate() 方法中初始化埋点 SDK，并传入当前的 Application 对象。SDK 拿到 Application 对象之后，就可以通过 registerActivityLifecycleCallback 方法注册 ActivityLifecycleCallbacks。这样 SDK 就能对 App 中所有 Activity 的生命周期事件进行集中处理（监控）了。在注册的 Application.ActivityLifecycleCallbacks 的 onActivityResumed(Activity activity) 回调方法中，我们可以拿到当前正在显示的 Activity，通过 activity.findViewById(android.R.id.content) 方法就可以拿到整个内容区域对应的 View（是一个 FrameLayout）了，本书有可能会用 rootView 和 ViewTree 来混称这个 View。然后 SDK 再逐层遍历这个 rootView，并判断当前 View 是否设置了 onClickListener，如果已设置 onClickListener 并且又不是我们自定义的 WrapperOnClickListener 类型，则通过自定义的 WrapperOnClickListener 代理当前 View 设置的 View.OnClickListener，然后并重新设置 View 的 onClickListener 为 WrapperOnClickListener。WrapperOnClickListener 实现了 View.OnClickListener 接口，在 WrapperOnClickListener 的 onClick 里会先调用 View 的原有 OnClickListener 处理逻辑，然后再调用埋点代码，实现了“插入”埋点代码，从而达到自动埋点的效果。

实现步骤

完整的项目源码后续会 release 给大家。

引入 DecorView

通过测试发现，目前基于代理 View 的 OnClickListener 的方案无法采集 MenuItem 控件的点击事件。

这又是为什么呢？

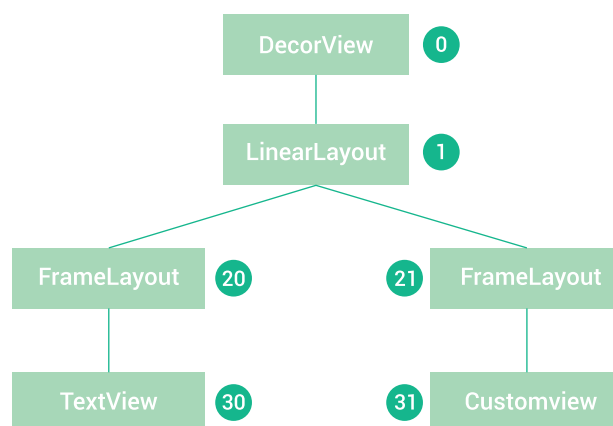
其实，这是因为我们通过 `android.R.id.content` 获取到的 RootView 是不包含 Activity 标题栏的，也就是不包括 MenuItem 的父容器。所以当我们去遍历 RootView 时是无法遍历到 MenuItem 的，因此无法代理其 OnClickListener，从而导致无法采集 MenuItem 的点击事件。

下面我们使用 DecorView 来解决这个问题。

那什么是 DecorView 呢？

DecorView 是整个 Window 界面的最顶层的 View（右图中编号为 0 的 View）。DecorView 只有一个子元素为 LinearLayout（右图编号 1），代表整个 Window 界面，包含通知栏、标题栏、内容显示栏三块区域。这个 LinearLayout 里有两个 FrameLayout 子元素。第一个 FrameLayout（右图编号 20）为标题栏显示界面。第二个 FrameLayout（右图编号 21）为内容栏显示界面，就是上面所说的 `android.R.id.content`。

所以，我们只需要将之前方案中 `activity.findViewById` (`android.R.id.content`) 换成 `activity.getWindow().getDecorView()`，就可以遍历到 MenuItem 了，也就可以自动采集到 MenuItem 点击事件了。



引入 ViewTreeObserver.OnGlobalLayoutListener

通过继续测试可以发现，当前的方案还有一个问题，即：目前该方案是无法采集 onResume() 生命周期之后动态创建的 View 的点击事件的。比如我们点击一个按钮，在其 OnClickListener 里动态创建一个 Button，然后通过 addView 添加到页面上：

```
ViewGroup rootView = findViewById(R.id.rootView);

AppCompatActivity button = new AppCompatActivity(this);

button.setText(" 动态创建的 Button");

button.setOnClickListener(new View.OnClickListener() {

    @Override

    public void onClick(View view) {

    }

});

rootView.addView(button);
```

此时，点击这个动态创建的 Button，是没有点击事件的。

这是因为我们是在 Activity 的 onResume 生命周期之前去遍历整个 rootView 并代理其 View.OnClickListener 的。如果是在 onResume 之后动态创建的 View，当时肯定是无法遍历到的，后来我们又没有再次去遍历一次，所以它的 mOnClickListener 就没有被代理过，所以点击时，我

们是无法采集其点击事件的。

下面我们通过 ViewTreeObserver.OnGlobalLayoutListener 来解决这个问题。

那什么是 ViewTreeObserver.OnGlobalLayoutListener 呢？

OnGlobalLayoutListener 是 ViewTreeObserver 的一个内部接口。当一个视图树的布局发生改变时，如果我们给当前的 View 设置了 ViewTreeObserver.OnGlobalLayoutListener 监听器，就可以被 ViewTreeObserver.OnGlobalLayoutListener 监听到（实际上是触发 onGlobalLayout 回调）。所以，基于这个原理，我们可以给 rootView 也添加一个 ViewTreeObserver.OnGlobalLayoutListener 监听器，当收到 onGlobalLayout 回调时（即视图树的布局发生变化，比如新的 View 被创建），我们再重新去遍历一次 rootView，然后找到那些没有被代理过 mOnClickListener 的 View 并进行代理。

关于 ViewTreeObserver.OnGlobalLayoutListener，建议在页面退出的时候 remove 掉，即在 onStop 的时候调用 removeOnGlobalLayoutListener 方法。

缺点

- 由于使用反射，效率比较低，对 App 的整体性能有一定的影响，也可能会引入兼容性方面的风险
- Application.ActivityLifecycleCallbacks 要求 API 14+
- View.hasOnClickListeners() 要求 API 15+
- removeOnGlobalLayoutListener 要求 API 16+
- 无法直接支持采集游离于 Activity 之上的 View 的点击，比如 Dialog、PopupWindow

知识点

- android.R.id.content
- Application.ActivityLifecycleCallbacks
- DecorView
- ViewTreeObserver.OnGlobalLayoutListener
- 反射
- 代理

参考资料

- [1] <https://github.com/fengcunhan/AutoTrace>
- [2] <http://lingnanlu.github.io/2015/12/24/androidridcontent>
- [3] <https://stackoverflow.com/questions/24712227/android-r-id-content-as-container-for-fragment>

二、代理 Window.Callback

Window.Callback

Window.Callback 是 Window 的一个内部接口。该接口包含了一系列的类似于 dispatchXXX 和 onXXX 的接口。当 Window 接收到外界的状态改变通知时就会回调其中的相应方法。比如，当用户点击某个控件时，就会回调 Window.Callback 中的 dispatchTouchEvent(MotionEvent event) 方法。

Window.Callback 定义如下：

```
/**
 * API from a Window back to its caller. This allows the client to
 * intercept key dispatching, panels and menus, etc.
 */
public interface Callback {
    .....

    /**
     * Called to process touch screen events. At the very least your
     * implementation must call
     * {@link android.view.Window#superDispatchTouchEvent} to do the
     * standard touch screen processing.
     *
     * @param event The touch screen event.
     *
     * @return boolean Return true if this event was consumed.
     */
    public boolean dispatchTouchEvent(MotionEvent event);

    .....
}
```

关于 Window.callback 更详细的信息，可以参考如下链接：

<http://www.android-doc.com/reference/android/view/Window.Callback.html>

原理概述

在应用程序自定义的 Application 的 onCreate() 方法中初始化埋点 SDK，并传入当前的 Application 对象。SDK 在拿到这个 Application 对象之后，通过 application.register-ActivityLifecycleCallbacks 注册 Application.ActivityLifecycleCallbacks。这样 SDK 就能对 App 中所有的 Activity 的生命周期事件进行集中处理（监控）了。在 ActivityLifecycleCallbacks 的 onActivityCreated(Activity activity, Bundle bundle) 回调方法中，我们就可以拿到当前正在显示的 Activity 对象，通过 activity.getWindow() 方法可以拿到这个 Activity 对应的 Window 对象，再通过 window.getCallback() 方法就可以拿到 Window.Callback 对象，最后通过自定义的 WrapperWindowCallback 代理这个 Window.Callback 对象。然后在 WrapperWindowCallback 的 dispatchTouchEvent(MotionEvent event) 方法中通过 MotionEvent 参数找到那个被点击的 View，插入埋点代码，最后再调用原有 Window.Callback 的 dispatchTouchEvent(MotionEvent event) 方法，即可达到自动埋点的效果。

实现步骤

完整的项目源码后续会 release 给大家。

缺点

- 由于每次点击触发时，都需要遍历一次 RootView，所以效率比较低，对 App 整体性能影响比较大
- view.hasOnClickListeners() 要求 API 15+
- Application.ActivityLifecycleCallbacks 要求 API 14+
- 无法采集 Dialog、PopupWindow 的点击事件

知识点

- Window.callback
- Application.ActivityLifecycleCallbacks
- ViewTreeObserver.OnGlobalLayoutListener
- 代理

参考资料

[1] <https://github.com/hellozhixue/BehaviorCollect>

三、代理 View.AccessibilityDelegate

Accessibility

Accessibility，即辅助功能。许多 Android 用户有不同的能力（限制），这要求他们可能会以不同的方式使用他们的 Android 设备。这些限制包括视力、肢体、年龄等，这些限制阻碍了他们看到或充分使用触摸屏，而用户的听力丧失，让他们可能无法感知声音信息和警报。

Android 提供了辅助功能的特性和服务，帮助这些用户更容易的使用他们的设备，这些功能包括语音合成、触觉反馈、手势导航、轨迹球和方向键导航。Android 应用程序开发人员可以利用这些服务，使他们的应用程序更贴近用户。该辅助服务工作在后台，由系统调用，用户界面的一些状态（比如 Button 被点击了）的改变可以通过回调 AccessibilityService 方法来通知。

比如下面的这个 Button：

```
<android.support.v7.widget.AppCompatButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text=" 神策数据 "
    android:contentDescription="SensorsData"/>
```

由于添加了 android:contentDescription 属性，当用户移动焦点到这个按钮或将鼠标悬停在它上面时，提供口头反馈的辅助功能服务就会发出“SensorsData”的声音。

View.AccessibilityDelegate

我们先看一下 View.java 的 performClick() 源码:

```
/**
 * Call this view's OnClickListener, if it is defined.
 * Performs all normal
 * actions associated with clicking: reporting
 * accessibility event, playing
 * a sound, etc.
 *
 * @return True there was an assigned OnClick-
 * Listener that was called, false
 * otherwise is returned.
 */
public boolean performClick() {
    final boolean result;

    final ListenerInfo li = mListenerInfo;
    if (li != null && li.mOnClickListener != null) {
        playSoundEffect(SoundEffectConstants.CLICK);
        li.mOnClickListener.onClick(this);
        result = true;
    } else {
        result = false;
    }

    sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED);
    notifyEnterOrExitForAutoFillIfNeeded(true);
    return result;
}

.....
```

```
public void sendAccessibilityEvent(int eventType){
    if (mAccessibilityDelegate != null) {
        mAccessibilityDelegate.sendAccessibilityEvent(this, eventType);
    } else {
        sendAccessibilityEventInternal(eventType);
    }
}

.....

public void setAccessibilityDelegate(@Nullable
AccessibilityDelegate delegate){
    mAccessibilityDelegate = delegate;
}
```

从上面的源码可以很容易的看出来, 当一个 View 被点击的时候, 系统会先调用当前 View 已设置的 mOnClickListener 的 onClick(view) 方法, 然后再调用 sendAccessibilityEvent(AccessibilityEvent.TYPE_VIEW_CLICKED) 的内部方法。在 sendAccessibilityEvent(int eventType) 方法的内部实现里, 其实是调用 mAccessibilityDelegate 的 sendAccessibilityEvent 方法, 并传入当前 View 对象和 AccessibilityEvent.TYPE_VIEW_CLICKED。因此, 我们只需要代理 View 的 mAccessibilityDelegate, 当一个 View 被点击时, 在原有 mOnClickListener 执行之后, 我们就能收到“消息”。代理 mAccessibilityDelegate 之后, 我们就能拿到当前被点击的 View 对象, 从而可以加入自动埋点的逻辑, 达到自动埋点的效果。

原理概述

在应用程序自定义的 Application 的 onCreate() 方法中初始化埋点 SDK，并传入当前的 Application 对象。SDK 就可以拿到这个 Application 对象，然后我们就可以通过 application.registerActivityLifecycleCallback 这个方法来注册 Application.ActivityLifecycleCallbacks。这样 SDK 就可以对 App 中所有的 Activity 的生命周期事件进行集中处理（监控）了。在 ActivityLifecycleCallbacks 的 onActivityResumed(Activity activity, Bundle bundle) 方法中，我们可以拿到当前正在显示的 Activity 对象，然后再通过 activity.getWindow().getDecorView() 方法或者 activity.findViewById(android.R.id.content) 方法拿到当前 Activity 的 rootView，通过 rootView.getViewTreeObserver() 方法可以拿到 rootView 的 ViewTreeObserver 对象，然后再通过 addOnGlobalLayoutListener() 方法给 rootView 注册 ViewTreeObserver.OnGlobalLayoutListener 监听器，这样我们就可以在收到当前 Activity 的视图状态发生改变时去主动遍历一次 rootView，并用我们自定义的 SensorsDataAccessibilityDelegate 代理当前 View 的 mAccessibilityDelegate 属性。在我们自定义的 SensorsDataAccessibilityDelegate 文件中的 public void sendAccessibilityEvent(View host, int eventType) 方法中，我们先调用原有的 mAccessibilityDelegate 的 sendAccessibilityEvent 方法，然后再插入埋点代码，其中 host 即是当前被点击的 View 对象，从而可以做到自动埋点的效果。

实现步骤

完整的项目源码后续会 release 给大家。

缺点

- Application.ActivityLifecycleCallbacks 要求 API 14+
- view.hasOnClickListeners() 要求 API 15+
- 无法采集 Dialog、PopupWindow 的点击事件
- 每次点击都需要遍历一次 rootView，效率比较低

知识点

- Application.ActivityLifecycleCallbacks
- Android 系统事件处理机制
- View Elevation

参考资料

[1]<https://github.com/foolchen/AndroidTracker>

四、AspectJ

AOP

AOP 是 Aspect Oriented Programming 的缩写，即“面向切面编程”。使用 AOP，可以在编译期间对代码进行动态管理，以达到统一维护的目的。AOP 是 OOP 编程的一种延续，也是 Spring 框架中的一个重要模块。利用 AOP 可以对业务逻辑的各个模块进行隔离，从而使得业务逻辑各个部分之间的耦合度降低，提高程序的可重用性，同时提高开发的效率。利用 AOP，我们可以在无侵入的在宿主中插入一些代码逻辑，从而可以实现一些特殊的功能，比如日志埋点、性能监控、动态权限控制、代码调试等。

AOP 术语

Advice：增强

也叫“通知”。增强是织入到目标类连接点上的一段程序代码。在 Spring 中，增强除了用于描述一段程序代码外，还拥有另一个和连接点相关的信息，这便是执行点的方位。结合执行点方位信息和切点信息，我们就可以找到特定的连接点。

JoinPoint：连接点

程序执行的某个特定位置，如类开始初始化前、类初始化后、类中某个方法调用前、调用后、方法抛出异常后。一个类或一段程序代码拥有一些具有边界性质的特定点，这些点中的特定点就称为“连接点”。Spring 仅支持方法的连接点，即仅能在方法调用前、方法调用后、方法抛出异常时以及方法调用前后这些程序执行点织入增强。连接点由两个信息确定：第一是用方法表示的程序执行点；第二是用相对点表示的方位。

PointCut：切点

也叫“切入点”。每个程序类都拥有多个连接点，如一个拥有两个方法的类，这两个方法都是连接点，即连接点是程序类中客观存在的事物。AOP 通过“切点”定位特定的连接点。连接点相当于数据库中的记录，而切点相当于查询条件。切点和连接点不是一对一的关系，一个切点可以匹配多个连接点。在 Spring 中，切点通过 Pointcut 接口进行描述，它使用类和方法作为连接点的查询条件，Spring AOP 的规则解析引擎负责切点所设定的查询条件，找到对应的连接点。其实确切地说，不能称之为查询连接点，因为连接点是方法执行前、执行后等包括方位信息的具体程序执行点，而切点只定位到某个方法上，所以如果希望定位到具体连接点上，还需要提供方位信息。

Aspect：切面

切面由切点和增强组成，它既包括了横切逻辑的定义，也包括了连接点的定义，Spring AOP 就是负责实施切面的框架，它将切面所定义的横切逻辑织入到切面所指定的连接点中。

Weaving：织入

织入是将增强添加对目标类具体连接点上的过程。AOP 像一台织布机，将目标类、增强通过 AOP 这台织布机天衣无缝地编织到一起。

根据不同的实现技术，AOP 有三种织入的方式：

- 1) 编译期织入，这要求使用特殊的 Java 编译器
- 2) 类装载期织入，这要求使用特殊的类装载器
- 3) 动态代理织入，在运行期为目标类添加增强生成子类的方式

Spring 采用动态代理织入，而我们本周要讲的 AspectJ 是采用编译期织入和类装载期织入。

Target：目标对象

增强逻辑的织入目标类。如果没有 AOP，目标业务类需要自己实现所有逻辑，而在 AOP 的帮助下，目标业务类只实现那些非横切逻辑的程序逻辑，而性能监视和事务管理等这些横切逻辑则可以使用 AOP 动态织入到特定的连接点上。

以上概念，如果之前没有接触过，确实挺晦涩的。

我们下面有一段“白话”总结一下：

第一步：

我们通过定义一个表达式（PointCut）来告诉程序，我们需要对哪些地方增加额外的操作。通过这个表达式（PointCut），我们得到那些需要通知的方法（JoinPoint）。

第二步：

我们还需要告诉程序，这些方法（JoinPoint）需要做怎样的增强（Advice）：

- 1) 什么时候进行额外的操作（执行前 / 执行后 / 返回之前）？
- 2) 额外操作具体要做什么？

我们把以上两个步骤定义到一个地方（Aspect）。

上面两个步骤涉及到的被修改的对象，我们称之为目标对象（Target）。

完成上面的所有操作的动作，我们总称为织入（Weaving）。

AspectJ

AOP 是一个概念，一个规范，本身并没有设定具体语言的实现。AspectJ 实际上是对 AOP 编程思想的一个实现，它能够和 Java 配合起来使用。

AspectJ 的核心就是它的编译器（ajc），它就做了一件事，将 AspectJ 的代码在编译期插入到目标程序当中，运行时跟在其他地方没什么两样。因此想要使用它最关键的的就是使用它的编译器去编译代码。ajc 会构建目标程序与 AspectJ 代码的联系，在编译期将 AspectJ 代码插入被切出的 PointCut 中，达到 AOP 的目的。

关于 AspectJ 更详细的介绍，可以参考其官网：<http://www.eclipse.org/aspectj/>

原理概述

对于 Android 系统中的 View，它的点击处理逻辑，都是通过设置相应的 OnClickListener，然后重写相应的方法实现的。比如对于 Button、ImageView 等控件，它设置的 listener 均是 android.view.View.OnClickListener，然后重写 onClick(android.view.View) 方法。我们只要在其 onClick(android.view.View) 方法中插入埋点代码，即可做到自动埋点。

我们可以把 AspectJ 的处理脚本放到自定义的插件里，之后编写相应的切面类，然后我们再定义合适的 PointCut 用来匹配我们的织入目标方法，比如 android.view.View.OnClickListener.onClick(android.view.View)，最后在编译期间插入埋点代码，从而就可达到自动埋点的效果。

实现步骤

完整的项目源码后续会 release 给大家。

缺点

- 无法织入第三方的库
- 由于定义的点切依赖编程语言，该方案无法兼容 Lambda 语法
- 会有一些兼容性方面的问题，比如：D8、Gradle 4.x 等

知识点

- AOP
- AspectJ
- Gradle
- Gradle Plugin
- 注解

参考资料

- [1] <https://www.cnblogs.com/yangyquin/p/5462488.html>
- [2] http://www.360doc.com/content/11/0414/16/3639038_109610987.shtml
- [3] <https://www.jianshu.com/p/873eae38cc1>
- [4] <https://www.jianshu.com/p/aa1112dbebc7>
- [5] <https://deemons.cn/2017/10/16/> 自定义 %20Gradle%20 插件 /
- [6] <https://blog.csdn.net/Deemons/article/details/78473874>
- [7] <https://github.com/uPhyca/gradle-android-aspectj-plugin>
- [8] <https://github.com/JakeWharton/hugo>
- [9] https://github.com/HujiangTechnology/gradle_plugin_android_aspectjx
- [10] <https://www.eclipse.org/aspectj/>

五、ASM

Android App 的打包流程，可以参考右图：

通过右图可知，我们只要在图中红圈处拦截，就可以拿到所有的 .class 文件，然后遍历 .class 文件中的所有方法，再根据条件找到目标方法，最后进行修改并保存，就可以插入埋点代码了。

Google 从 Android Gradle 1.5.0 开始，提供了 Transform API，允许第三方的插件（Plugin）在 Android App 打包成 .dex 文件之前的编译过程中操作 .class 文件。我们只要实现一套 Transform，去遍历所有 .class 文件的所有方法，然后进行修改，最后再对原文件进行替换，即可达到插入代码的目的。

Gradle Transform

Gradle Transform 是 Android 官方提供给开发者在项目构建阶段，即由 .class 到 .dex 转换期间修改 .class 文件的一套 API。目前比较经典的应用是字节码插桩、代码注入技术。

概括来说，Transform 就是把输入的 .class 文件转变成目标字节码文件。

我们先了解一下 Transform 的两个概念：

• TransformInput

TransformInput 是指这些输入文件的抽象。它包括两部分：

1) DirectoryInput 集合

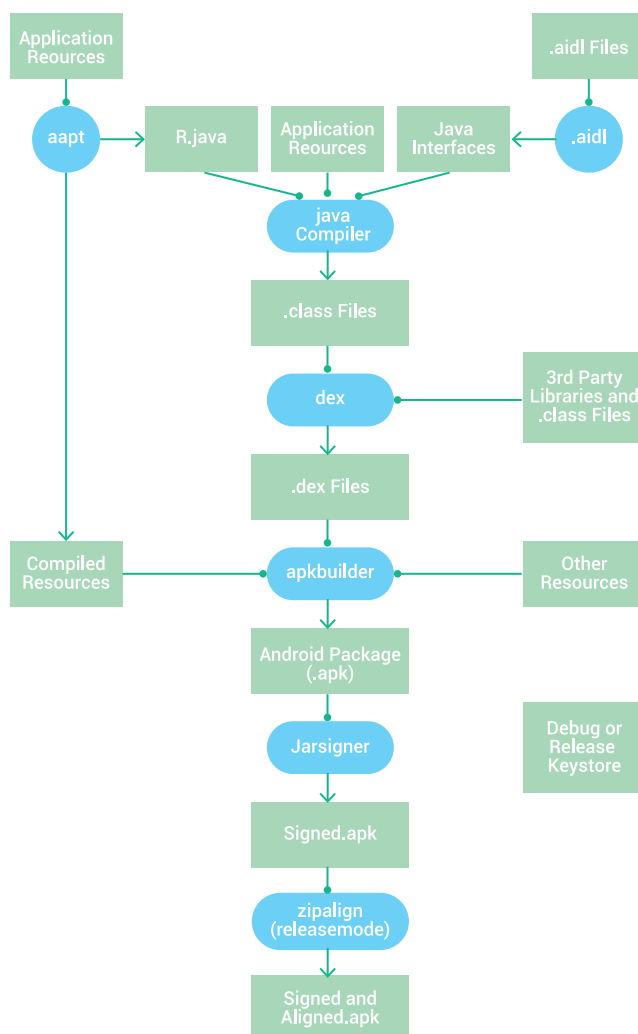
是指以源码方式参与项目编译的所有目录结构及其目录下的源码文件。

2) JarInput 集合

是指以 jar 包方式参与项目编译的所有本地 jar 包和远程 jar 包。

• TransformOutputProvider

是指 Transform 的输出，通过它可以获取输出路径。



我们下面了解一下 Transform.java 的定义。

Transform.java 是一个抽象类，它的定义如下：

```
public abstract class Transform {  
    @NonNull  
    public abstract String getName();  
  
    @NonNull  
    public abstract Set<ContentType> getInputTypes();  
  
    @NonNull  
    public Set<ContentType> getOutputTypes() {  
        return getInputTypes();  
    }  
  
    @NonNull  
    public abstract Set<Scope> getScopes();  
  
    public abstract boolean isIncremental();  
  
    public void transform(@NonNull TransformInvocation transformInvocation)  
        throws TransformException, InterruptedException, IOException {  
        transform(transformInvocation.getContext(), transformInvocation.getInputs(),  
            transformInvocation.getReferencedInputs(),  
            transformInvocation.getOutputProvider(),  
            transformInvocation.isIncremental());  
    }  
    .....  
}
```

它定义了几个抽象方法如下：



getName



getInputTypes



getScopes



isIncremental

getName

代表该 Transform 的 Task 名称

getInputTypes

指定 Transform 要处理的数据类型。目前支持两种类型：

CLASSES

表示要处理编译后的字节码，可能是 jar 包也可能是目录

RESOURCES

表示处理标准的 java 资源

getScopes

指定 Transform 的作用域。常见的作用域有：

PROJECT

只处理当前项目

SUB_PROJECTS

只处理子项目

PROJECT_LOCAL_DEPS

只处理当前项目的本地依赖，例如 jar, aar

SUB_PROJECTS_LOCAL_DEPS

只处理子项目的本地依赖，例如 jar, aar

EXTERNAL_LIBRARIES

只处理外部的依赖库

PROVIDED_ONLY

只处理本地或远程以 provided 形式引入的依赖库

TESTED_CODE

测试代码

isIncremental

是否是增量构建

Gradle Transform 实例

我们下面实现一个 Gradle Transform 的实例，该实例其实没有什么特定功能，仅仅是把所有的输入文件原封不动的拷贝到输出目录。

通过 Transform 提供的 API 可以遍历所有文件，包括目录和 jar 包。但是要实现 Transform 的遍历 .class 文件的操作，需要通过 Gradle 插件来实现。

完整的项目源码后续会 release 给大家。

ASM

ASM 是一个功能比较齐全的 Java 字节码操作与分析框架。它能被用来动态生成类或者增强既有类的功能。ASM 可以直接产生二进制 class 文件，也可以在类被加载入 Java 虚拟机之前动态改变类的行为。Java class 被存储在严格格式定义的 .class 文件里，这些类文件拥有足够的元数据来解析类中的所有元素，包括类名称、方法、属性以及 Java 字节码（指令）。ASM 从类文件中读入这些信息后，能够改变类行为、分析类的信息，甚至能够根据具体的要求生成新的类。

我们下面简单的介绍一个 ASM 框架中几个核心的类：

ClassReader

该类用来解析编译过的 class 字节码文件。

ClassWriter

该类用来重新构建编译后的类，比如说修改类名、属性以及方法，甚至可以生成新的类的字节码文件。

ClassVisitor

主要负责“拜访”类成员信息。其中包括标记在类上的注解、类的构造方法、类的字段、类的方法、静态代码块。

AdviceAdapter

实现了 MethodVisitor 接口，主要负责“拜访”方法的信息，用来进行具体的方法字节码操作。

ClassVisitor 定义了一系列的 API，它按照一定的标准次序来遍历类中的成员。

在 ClassVisitor 中，我们可以根据实际的需求进行条件判断，只要满足我们特定条件的类，我们才会去修改它的方法。比如，我们要自动采集 Button 的点击事件，那么只有实现了 View\$OnClickListener 接口的类，我们才会去遍历它的方法并找到 onClick(view) 方法，然后进行修改操作。

我们下面重点介绍 ClassVisitor 中的 visit 方法和 visitMethod 方法。

visit 方法

该方法是当扫描类时第一个拜访的方法。
方法定义如下：

```
void visit(int version, int access, String name, String signature, String superName, String[] interfaces);
```

各参数解释如下：

• version

表示 JDK 的版本，比如 51，代表 JDK 版本 1.7。

各个 JDK 版本对应的数值如下：

| JDK 版本 | int 数值 |
|----------|--------|
| J2SE 8 | 52 |
| J2SE 7 | 51 |
| J2SE 6.0 | 50 |
| J2SE 5.0 | 49 |
| JDK 1.4 | 48 |
| JDK 1.3 | 47 |
| JDK 1.2 | 46 |
| JDK 1.1 | 45 |

• access

类的修饰符。修饰符在 ASM 中是以“ACC_”开头的常量。可以作用到类级别上的修饰符有：

| 修饰符 | 含义 |
|----------------|----------------------|
| ACC_PUBLIC | public |
| ACC_PRIVATE | private |
| ACC_PROTECTED | protected |
| ACC_FINAL | final |
| ACC_SUPER | extends |
| ACC_INTERFACE | 接口 |
| ACC_ABSTRACT | 抽象类 |
| ACC_ANNOTATION | 注解类型 |
| ACC_ENUM | 枚举类型 |
| ACC_DEPRECATED | 标记了 @Deprecated 注解的类 |
| ACC_SYNTHETIC | javac 生成 |

• name

类的名称。通常会使用完整的包名 + 类名来表示类，比如：a.b.c.MyClass，但是在字节码中是以路径的形式表示，即：a/b/c/MyClass。值得注意的是，虽然是路径表示法但是不需要写明类的“.class”扩展名。

• signature

表示泛型信息，如果类并未定义任何泛型该参数为空。

• superName

表示所继承的父类。由于 Java 的类是单根结构，即所有类都继承自 java.lang.Object。因此可以简单的理解为任何类都会具有一个父类。虽然在编写 Java 程序时我们没有去写 extends 关键字去明确继承的父类，但是 JDK 在编译时总会为我们加上“extends Object”。

• interfaces

表示类实现的接口，在 Java 中，类是可以实现多个不同的接口，因此该参数是一个数组。

visitMethod 方法

该方法是当扫描器扫描到类的方法时进行调用。

方法定义如下：

```
MethodVisitor visitMethod(int access, String name, String desc, String signature, String[] exceptions);
```

各参数解释如下：

• access

表示方法的修饰符。

可以作用到方法级别上的修饰符有：

| 修饰符 | 含义 |
|------------------|-------------|
| ACC_PUBLIC | public |
| ACC_PRIVATE | private |
| ACC_PROTECTED | protected |
| ACC_STATIC | static |
| ACC_FINAL | final |
| ACC_SYNCHRONIZED | 同步的 |
| ACC_VARARGS | 不定参数个数的方法 |
| ACC_NATIVE | native 类型方法 |
| ACC_ABSTRACT | 抽象的方法 |

| | |
|----------------|----------------------|
| ACC_DEPRECATED | 标记了 @Deprecated 注解的类 |
| ACC_SYNTHETIC | javac 生成 |

• name

表示方法名。

• desc

表示方法签名，方法签名的格式如下：“(参数列表) 返回值类型”。在 ASM 中不同的类型对应不同的代码：

| 代码 | 类型 |
|-----------|---------|
| “I” | int |
| “B” | byte |
| “C” | char |
| “D” | double |
| “F” | float |
| “J” | long |
| “S” | short |
| “Z” | boolean |
| “V” | void |
| “[...,” | 数组 |
| “[[...,” | 二维数组 |
| “[[[...,” | 三维数组 |

下面举几个方法参数列表对应的方法签名示例：

| 参数列表 | 方法参数 |
|---------------------------------------|--|
| String[] | [Ljava/lang/String; |
| String[][] | [[Ljava/lang/String; |
| int, String, String[] | ILjava/lang/String; [Ljava/lang/String; |
| int, boolean, long, String[], double | IZJ[Ljava/lang/String;D |
| Class<?>, String, Object... paramType | Ljava/lang/Class;Ljava/lang/String; [Ljava/lang/Object; |
| int[] | [I |

- **signature**

表示泛型相关的信息。

- **exceptions**

表示将会抛出的异常，如果方法不会抛出异常，该参数为空。

原理概述

我们自定义一个 Gradle Plugin，可以注册一个 Transform 对象，然后在 transform 方法里，分别遍历目录和 jar 包，然后我们就可以遍历所有的 .class 文件。然后再利用 ASM 的相关 API，加载相应的 .class 文件、解析 .class 文件，就可以找到满足一定特定条件的 .class 文件和相关方法，最后去修改相应的方法以动态插入埋点字节码，从而达到自动埋点的效果。

实现步骤

完整的项目源码后续会 release 给大家。

缺点

- 暂时没有什么发现缺点

知识点

- 字节码语法
- Gradle Plugin
- Transform API
- ASM

参考资料

- [1] <https://www.jianshu.com/p/9039a3e46dbc>
- [2] <http://tools.android.com/tech-docs/new-build-system/transform-api>
- [3] <https://blog.csdn.net/tscyds/article/details/78082861>
- [4] https://blog.csdn.net/Neacy_Zz/article/details/78546237
- [5] <http://asm.ow2.org/>
- [6] <https://blog.csdn.net/byeweyyang/article/details/80127789>

六、Javassist

Javassist

Java 字节码以二进制的形式存储在 .class 文件中，每一个 .class 文件包含一个 Java 类或接口。Javassist 就是一个用来处理 Java 字节码的类库。它可以在一个已经编译好的类中添加新的方法，或者是修改已有的方法，并且不需要对字节码方面有深入的了解。

Javassist 可以绕过编译，直接操作字节码，从而实现代码注入。所以使用 Javassist 的时机就是在构建工具 Gradle 将源文件编译成 .class 文件之后，在将 .class 打包成 .dex 文件之前。

Javassist 基础

• 读写字节码

在 Javassist 中，.class 文件是用类 Javassist.CtClass 表示。一个 CtClass 对象可以处理一个 .class 文件。

```
ClassPool pool = ClassPool.getDefault();
CtClass aClass = pool.get("com.sensorsdata.analytics.android.sdk.SensorsDataAutoTrackHelper")
aClass.setSuperclass("java.lang.Object")
aClass.writeFile()
```

在上面这个示例中，先获取一个 ClassPool 对象。ClassPool 是 CtClass 对象的容器。它按需读取类文件来创建 CtClass 对象，并且保存 CtClass 对象以便以后会被使用到。

为了修改类的定义，首先需要使用 ClassPool.get() 方法从 ClassPool 中获得一个 CtClass 对象。使用 getDefault() 方法获取的 ClassPool 对象使用的是默认系统的类搜索路径。

ClassPool 是一个存储 CtClass 的 Hash 表，类的名称作为 Hash 表的 key。ClassPool 的 get() 函数会从 Hash 表查找 key 对应的 CtClass 对象。如果没有找到，get() 函数会创建并返回一个新的 CtClass 对象，这个对象会保存在 Hash 表中。

从 ClassPool 中获取的 CtClass 对象是可以被修改的。在上面的例子中，com.sensorsdata.analytics.android.sdk.SensorsDataAutoTrackHelper 的父类被设置为 java.lang.Object。调用 writeFile() 后，这项修改会被写入原始类文件中。

writeFile() 会将 CtClass 对象转换成类文件并写到本地磁盘。同时,也可以使用 toBytecode() 函数来获取修改过的字节码:

```
byte[] b = aClass.toBytecode();
```

也可以使用 toClass() 函数直接将 CtClass 转换成 Class 对象:

```
Class clazz = aClass.toClass();
```

toClass() 请求当前线程的 ClassLoader 加载 CtClass 所代表的类文件,它返回此类文件的 java.lang.Class 对象。

• 冻结类

如果一个 CtClass 对象通过 writeFile()、toClass()、toBytecode() 等方法被转换成一个类文件,此 CtClass 对象就会被冻结起来,不允许再被修改,这是因为一个类只能被 JVM 加载一次。

其实,一个冻结的 CtClass 对象也可以被解冻,比如:

```
CtClass aClass = ...;

.....

aClass.writeFile();

aClass.defrost();

// 因为类已经被解冻,所以这里是可以被修改成功的

aClass.setSuperClass(...);
```

此处调用 defrost() 方法之后,这个 CtClass 对象就又可以被修改了。

• 类搜索路径

通过 ClassPool.getDefault() 获取的 ClassPool 使用 JVM 的类搜索路径。如果程序运行在 JBoss 或者 Tomcat 等 Web 服务器上,ClassPool 可能无法找到用户的类,因为 Web 服务器使用多个类加载器作为系统类加载器。在这种情况下,ClassPool 必须添加额外的类搜索路径。

```
ClassPool pool = ClassPool.getDefault();

pool.insertClassPath(new ClassClassPath(this.getClass()));
```

上面的代码示例,将 this 指向的类添加到 ClassPool 的类加载路径中。你可以使用任意 Class 对象来代替 this.getClass(),

从而将 Class 对象添加到类加载路径中。同时，也可以注册一个目录作为搜索路径。比如：

```
ClassPool pool = ClassPool.getDefault();  
pool.insertClassPath( "/usr/local/Library/" );
```

上面的例子是将 “/usr/local/Library/” 目录添加到类搜索路径中。

• ClassPool

ClassPool 是 CtClass 对象的容器。因为编译器在编译引用 CtClass 代表的 Java 类的源代码时，可能会引用 CtClass 对象，所以一旦一个 CtClass 被创建，它就会被保存在 CtClass 中。

• 避免内存溢出

如果 CtClass 对象的数量变得非常多，ClassPool 有可能会造成巨大的内存消耗。为了避免这个问题，我们可以从 ClassPool 中显式删除不必要的 CtClass 对象。如果对 CtClass 对象调用 detach() 方法，那么该 CtClass 对象将会被从 ClassPool 中删除。比如：

```
CtClass aClass = ...;  
aClass.writeFile();  
aClass.detach();
```

在调用 detach() 方法之后，就不能再调用这个 CtClass 对象的任何有关方法了。如果调用 ClassPool 的 get() 方法，ClassPool 会再次读取这个类文件，并创建一个新的 CtClass 对象。

• 在方法体中插入代码

CtMethod 和 CtConstructor 均提供了 insertBefore()、insertAfter() 及 addCatch() 等方法。它们可以把用 Java 编写的代码片段插入到现有的方法体中。Javassist 包括一个用于处理源代码的小型编译器，它接收用 Java 编写的源代码，然后将其编译成 Java 字节码，并内联到方法体中。

也可以按行号来插入代码段（如果行号表包含在类文件中）。向 CtMethod 和 CtConstructor 中的 insertAt() 方法提供源代码和原始类定义中的源文件的行号，就可以将编译后的代码插入到指定行号位置。

insertBefore()、insertAfter()、addCatch() 和 insertAt() 等方法都能接收一个表示语句或语句块的 String 对象。一个语句是一个单一的控制结构，比如 if 和 while 或者以分号结尾的表达式。语句块是一组用大括号 {} 包围的语句。

语句和语句块可以引用字段和方法。但不允许访问在方法中声明的局部变量，尽管在块中声明一个新的局部变量是允许的。

传递给方法 insertBefore()、insertAfter()、addCatch() 和 insertAt() 的 String 对象是由 Javassist 的编译器编译的。由于编译器支持语言扩展，所以以 \$ 开头的几个标识符都有特殊的含义：

\$0, \$1, \$2, ...

传递给目标方法的参数使用 \$1, \$2, ... 来访问, 而不是原始的参数名称。\$1 表示第一个参数, \$2 表示第二个参数, 以此类推。这些变量的类型与参数类型相同。\$0 等价于 this 指针。如果方法是静态的, 则 \$0 不可用。

\$args

变量 \$args 表示所有参数的数组。该变量的类型是 Object 类型的数组。如果参数类型是原始类型 (如 int、boolean 等), 则该参数值将被转换为包装器对象 (如 java.lang.Integer) 以存储在 \$args 中。因此, 如果第一个参数的类型不是原始类型, 那么 \$args[0] 等于 \$1。注意 \$args[0] 不等于 \$0, 因为 \$0 表示 this。

\$\$

变量 \$\$ 是所有参数列表的缩写, 用逗号分隔。

\$_

CtMethod 中的 insertAfter() 是在方法的末尾插入编译的代码。传递给 insertAfter() 的语句中, 不但可以使用特殊符号如 \$0, \$1。也可以使用 \$_ 来表示方法的结果值。

该变量的类型是方法的返回结果类型 (返回类型)。如果返回结果类型为 void, 那么 \$_ 的类型为 Object, \$_ 的值为 null。

虽然由 insertAfter() 插入的编译代码通常在方法返回之前执行, 但是当方法抛出异常时, 它也可以执行。要在抛出异常时执行它, insertAfter() 的第二个参数 asFinally 必须为 true。

如果抛出异常, 由 insertAfter() 插入的编译代码将作为 finally 子句执行。\$_ 的值 0 或 null。在编译代码的执行终止后, 最初抛出的异常被重新抛出给调用者。注意, \$_ 的值不会被抛给调用者, 它将被丢弃。

• addCatch

addCatch() 插入方法体抛出异常时执行的代码, 控制权会返回给调用者。在插入的源代码中, 异常用 \$e 表示。

```
CtMethod m = ...;
CtClass etype = ClassPool.getDefault().get("java.io.IOException");
m.addCatch("{ System.out.println($e); throw $e; }", etype);
```

转换成对应的 java 代码如下:

```
try {
    // the original method body
} catch (java.io.IOException e) {
    System.out.println(e);
    throw e;
}
```

请注意, 插入的代码片段必须以 throw 或 return 语句结束。

• 注解 (Annotations)

CtClass、CtMethod、CtField 和 CtConstructor 均提供了 getAnnotations() 方法，用于读取注解。它返回一个注解类型的对象数组。

我们目前只介绍当前全埋点方案会用到的关于 Javassist 的相关基础知识，关于 Javassist 更详细的用法，可以参考：

<https://github.com/jboss-javassist/javassist/wiki/Tutorial-1>

原理概述

在自定义的 plugin 里，注册一个自定义的 Transform，然后可以分别对目录和 jar 包进行遍历。在遍历的过程中，利用 Javassist 的 API 来对满足特定条件的方法进行修改，插入相关埋点代码。原理与 ASM 类似，只是把操作 .class 文件的库由 ASM 换成 Javassist。

实现步骤

完整的项目源码后续会 release 给大家。

缺点

- 暂时没有什么发现缺点

知识点

- 汇编相关知识

参考资料

- [1] <https://www.jianshu.com/p/43424242846b>
- [2] <https://blog.csdn.net/Deemons/article/details/78473874>
- [3] <https://blog.csdn.net/yulong0809/article/details/77752098>
- [4] <https://juejin.im/post/58fea36bda2f60005dd1b7c5>
- [5] <https://www.jianshu.com/p/417589a561da>
- [6] <http://www.javassist.org>
- [7] <https://github.com/jboss-javassist/javassist>
- [8] <https://github.com/jboss-javassist/javassist/wiki/Tutorial-1>
- [9] <https://github.com/jboss-javassist/javassist/wiki/Tutorial-2>
- [10] <https://github.com/jboss-javassist/javassist/wiki/Tutorial-3>

七、AST

APT

APT 是 Annotation Processing Tool 的缩写，即注解处理器，是一种处理注解的工具。确切的说它是 javac 的一个工具，它用来在编译时扫描和处理注解。注解处理器以 Java 代码（或者编译过的字节码）作为输入，生成 .java 文件作为输出。简单来说就是在编译期，通过注解生成 .java 文件。权限控制、代码调试等。

Element

自定义注解处理器，需要继承 AbstractProcessor 类，而 AbstractProcessor 最终要的就是 process 方法。process 方法处理的核心是 Element 对象。

Element 的源码源码如下：

```
package javax.lang.model.element;

import java.lang.annotation.Annotation;
import java.util.List;
import java.util.Set;
import javax.lang.model.AnnotatedConstruct;
import javax.lang.model.type.TypeMirror;

public interface Element extends AnnotatedConstruct {
    TypeMirror asType();
    ElementKind getKind();
    Set<Modifier> getModifiers();
    Name getSimpleName();
    Element getEnclosingElement();
    List<? extends Element> getEnclosedElements();
    boolean equals(Object var1);
    int hashCode();
    List<? extends AnnotationMirror> getAnnotationMirrors();
}
```

```
<A extends Annotation> A getAnnotation(Class<A> var1);  
<R, P> R accept(ElementVisitor<R, P> var1, P var2);  
}
```

从定义可以看出，Element 是一个接口，它定义了外部可以调用的方法。

asType

返回此元素定义的类型

getKind

返回此元素的种类：包、类、接口、方法、字段等

getModifiers

返回此元素的修饰符

getSimpleName

返回此元素的简单名称，如类名

getEnclosedElements

返回封装此元素的最里层元素

getAnnotation

返回此元素针对指定类型的注解。注解可以是继承的，也可以是直接存在于此元素上的

Element 有 5 个直接子类，它们分别代表一种特定类型的元素。五个子类各有各的用处并且有各种独有的方法，在使用的时候可以强制将 Element 对象转换成其中的任何一种，但是必须满足转换的条件，否则会抛出异常。

TypeElement

一个类或接口程序元素

VariableElement

一个字段、enum 常量、方法或构造方法参数、局部变量或异常参数

ExecutableElement

某个类或接口的方法、构造方法或初始化程序（静态或实例），包括注解类型元素

PackageElement

一个包程序元素

TypeParameterElement

一般类、接口、方法或构造方法元素的泛型参数

其中 TypeElement 和 VariableElement 是最核心的两个 Element，也是我们下文会使用到的。

ATP 实例

我们通过 ATP 来实现一个功能，功能类似于 ButterKnife 中的 @BindView 注解。即：通过对 View 变量的注解，实现 View 的绑定（无需调用 findViewById）。

完整的项目源码后续会 release 给大家。

AST

AST，是 Abstract Syntax Tree 的缩写，即“抽象语法树”，是编辑器对代码的第一步加工之后的结果，是一个树形式表示的源代码。源代码的每个元素映射到一个节点或子树。

Java 的编译过程可以分成三个阶段：

第一阶段：

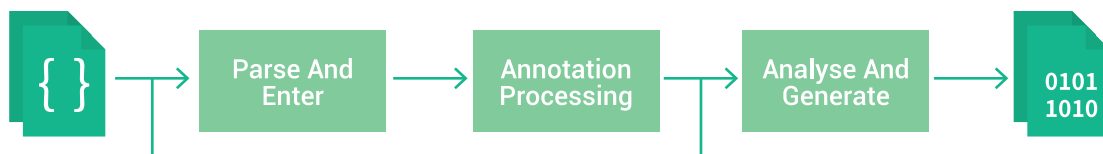
所有的源文件会被解析成语法树；

第二阶段：

调用注解处理器，即 ATP。如果注解处理器产生了新的源文件，新的源文件也要参与编译；

第三阶段：

语法树会被分析并转化成类文件。



原理概述

编辑器对代码处理的流程大概是：

```
JavaTXT-> 词语法分析 -> 生成 AST -> 语义分析 -> 编译字节码
```

通过操作 AST，可以达到修改源代码的功能。

可以在注解处理器的 process 函数里，通过 `roundEnvironment.getRootElements()` 方法可以拿到所有的 Element 对象，通过 `trees.getTree(element)` 方法可以拿到对应的抽象语法树（AST），然后我们自定义一个 `TreeTranslator`，在 `visitMethodDef` 里即可对方法进行判断。如果是目标处理方法，则通过 AST 的相关 API 插入埋点代码。

实现步骤

完整的项目源码后续会 release 给大家

知识点

- APT
- AST

缺点

- `com.sun.tools.javac.tree` 相关 API 语法晦涩，理解难度大，要求对编译原理有一定的基础
- APT 无法扫描其他 module，导致 AST 无法处理其他 module
- 不支持 Lambda 语法
- 带有返回值的方法，很难把埋点代码插入到方法之后

参考资料

- [1] <https://www.jianshu.com/p/5514cf705666>
- [2] <https://www.jianshu.com/p/7af58e8e3e18>
- [3] http://www.eclipse.org/articles/Article-JavaCodeManipulation_AST/
- [4] <http://developer.51cto.com/art/201305/392858.htm>
- [5] <https://juejin.im/entry/5ae06228518825671278334d>
- [6] <http://www.massapi.com/source/bitbucket/20/86/2086842069/-Processor/src/TimeAnnotationProcessor.java.html>

关于神策数据

神策数据是专业的大数据分析平台服务提供商，致力于帮助客户实现数据驱动。公司围绕用户级大数据分析和需求，推出神策分析、神策客景、神策自动化运营、神策智能推荐等产品。神策分析是深度用户行为分析平台，支持私有化部署、基础数据采集与建模，并作为 PaaS 平台支持二次开发，同时还支持用户精准分群，用户标签体系的构建；神策客景是基于行为数据的客户全生命周期分析平台，创造性将用户行为数据融入客户生命周期的管理与分析，实现客群健康度分析，流失预警等重要价值，并应用到企业服务、工具软件等多个领域；神策自动化运营是基于分群标签的全流程运营闭环分析系统，通过用户精准分群、灵活创建并管理营销活动计划，一键执行计划触达用户，并实时分析活动效果，真正形成自动化、精细化的运营闭环；神策智能推荐是基于用户行为分析的全流程智能推荐系统，帮助企业实现对用户“千人千面”的个性化内容推荐，改善用户体验，持续提升核心业务指标。

此外，还提供大数据相关咨询和完整解决方案。神策数据积累了中国银联、中国电信、百度视频、百联、万达、小米、中邮消费金融、广发证券、中原银行、百信银行、聚美优品、中商惠民、纷享销客、Keep、36 氪、中青旅、平安寿险、链家、四川航空等 500 余家付费企业用户的服务和客户成功经验，为客户全面提供指标梳理、数据模型搭建等专业的咨询、实施和技术支持服务。希望更深入了解神策数据或有数据驱动相关问题，请拨打 4006509827 电话咨询，会有专业的工作人员为您解答。

关于神策数据用户行为洞察研究院

Sensors Data User Behavior Insights Research

神策数据用户行为洞察研究院，由神策数据核心业务与技术骨干组成、并与各行业领先企业的业务与技术专家精诚合作，聚焦于大数据与用户行为分析的技术与实践前沿研究；旨在提供更具行业深度的洞察、领先的行业最佳实践、创新的技术解决方案等，为广大企业客户、大数据产业链从业者及神策数据自身未来的发展提供指导。未来研究院，将会进一步联合行业优秀创新典范、各类型合作伙伴、学术界与行业专家一起，汇聚大数据与用户行为分析领域的最佳创新实践和行业深度洞察，每年定期发布“数据驱动系列”研究。

声明

本白皮书版权归神策数据所有，未经神策数据书面许可，任何其他个人或组织均不得以任何形式将本白皮书的全部或部分内容转载、复制、编辑或发布使用于其他任何场合。白皮书内的信息“按现状”提供，不附有任何种类的（无论是明示的还是默示的）保证，包括不附有关于适销性、适用于某种特定用途的任何保证以及非侵权的任何保证或条件。本白皮书目的仅为提供通用指南，并不视为针对企业提供的专业建议。



联系我们

邮箱：contact@sensorsdata.cn

电话：400 650 9827

网址：www.sensorsdata.cn

扫码了解更多