Android 端埋点框架 README

1. 依附埋点参数到视图树

`ITrackModel` 接口定义了一个数据填充能力，你可以创建它的实现类来定义一个数据节点，并在 fillTrackParams() 方法中声明参数。例如：MyGoodsViewHolder 实现了 ITrackMode 接口，在 fillTrackParams() 方法中声明参数（goods_id / goods_name）。

随后，通过 View 的扩展函数`View.trackModel()`将其依附到视图节点上。扩展函数 View.trackModel() 内部基于 View#setTag() 实现。

`MyGoodsViewHolder.kt`

```
class MyGoodsViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView),
ITrackModel {

    private var mItem: GoodsItem? = null

    init {
        // Java: EasyTrackUtilsKt.setTrackModel(itemView, this);
        itemView.trackModel = this
    }

     override fun fillTrackParams(params: TrackParams) {
        mItem?.let {
            params.setIfNull("goods_id", it.id)
            params.setIfNull("goods_name", it.goods_name)
        }
    }
}
```

`EasyTrackUtils.kt`

```
/**
 * Attach track model on the view.
 */
var View.trackModel: ITrackModel?
    get() = this.getTag(R.id.tag_id_track_model) as? ITrackModel
    set(value) {
        this.setTag(R.id.tag_id_track_model, value)
    }
```

`ITrackModel.kt`

```
/**
 * 定义数据填充能力
 */
interface ITrackModel : Serializable {
    /**
     * 数据填充
     */
    fun fillTrackParams(params: TrackParams)
}
```

2. 触发事件埋点

在需要埋点的地方，直接通过定义在 View 上的扩展函数 `trackEvent(事件名)` 触发埋点事件，它会以该扩展函数的接收者对象为起点，逐级向上层视图节点收集参数。另外，它还有多个定义在 Activity、Fragment、ViewHolder 上的扩展函数，但最终都会调用到 View.trackEvent。

```
class MyGoodsViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
     fun bind(item: GoodsItem) {
        ...
        trackEvent(GOODS_EXPOSE)
    }
}
```

`EasyTrackUtils.kt`

```
@JvmOverloads
fun Activity?.trackEvent(eventName: String, params: TrackParams? = null) =
    findRootView(this)?.doTrackEvent(eventName, params)

@JvmOverloads
fun Fragment?.trackEvent(eventName: String, params: TrackParams? = null) =
    this?.requireView()?.doTrackEvent(eventName, params)

@JvmOverloads
fun RecyclerView.ViewHolder?.trackEvent(eventName: String, params: TrackParams? =
null) {
    this?.itemView?.let {
        if (null == it.parent) {
            it.post { it.doTrackEvent(eventName, params) }
        } else {
            it.doTrackEvent(eventName, params)
        }
    }
}

@JvmOverloads
fun View?.trackEvent(eventName: String, params: TrackParams? = null):
TrackParams? =
    this?.doTrackEvent(eventName, params)
```

查看 logcat 日志，可以看到以下日志，显示埋点并没有生效。这是因为没有为 EasyTrack 配置埋点数据上报和统计分析的能力。

`logcat 日志`

```
EasyTrackLib: Try track event goods_expose, but the providers is Empty.
```

3. 实现 ITrackProvider 接口

EasyTrack 的职责在于收集分散的埋点数据，本身没有提供埋点数据上报和统计分析的能力。因此，你需要实现 ITrackProvider 接口进行依赖注入。例如，这里模拟实现友盟数据埋点提供器，在 onInit() 方法中进行初始化，在 onEvent() 方法中调用友盟 SDK 事件上报方法。

`MockUmengProvider.kt`

```
/**
 * 模拟友盟数据上报
 */
class MockUmengProvider : ITrackProvider() {

    companion object {
        const val TAG = "Umeng"
    }

    /**
     * 是否启用
     */
    override var enabled = true

    /**
     * 名称
     */
    override var name = TAG

    /**
     * 初始化
     */
    override fun onInit() {
        Log.d(TAG, "Init Umeng provider.")
    }

    /**
     * 执行事件上报
     */
    override fun onEvent(eventName: String, params: TrackParams) {
        Log.d(TAG, params.toString())
    }
}
```

4. 配置 EasyTrack

在应用初始化时，进行 EasyTrack 的初始化配置。我们可以将相关的初始化代码单独封装起来，例如：

`StatisticsUtils.kt`

```
// 模拟友盟数据统计提供器
val umengProvider by lazy {
    MockUmengProvider()
}

// 模拟神策数据统计提供器
```

```
val sensorProvider by lazy {
    MockSensorProvider()
}

/**
 * 初始化 EasyTrack，在 Application 初始化时调用
 */
fun init(context: Context) {
    configStatistics(context)
    registerProviders(context)
}

/**
 * 配置
 */
private fun configStatistics(context: Context) {
    // 调试开关
    EasyTrack.debug = BuildConfig.DEBUG
    // 页面间参数映射
    EasyTrack.referrerKeyMap = mapOf(
        CUR_PAGE to FROM_PAGE,
        CUR_TAB to FROM_TAB
    )
}

/**
 * 注册提供器
 */
private fun registerProviders(context: Context) {
    EasyTrack.registerProvider(umengProvider)
    EasyTrack.registerProvider(sensorProvider)
}
```

`EventConstants.java`

```
public static final String FROM_PAGE = "from_page";
public static final String CUR_PAGE = "cur_page";
public static final String FROM_TAB = "from_tab";
public static final String CUR_TAB = "cur_tab";
```

配置	类型	描述
debug	Boolean	调试开关
referrerKeyMap	Map<String,String>	全局页面间参数映射
registerProvider()	ITrackProvider	底层数据埋点能力

以上步骤是 EasyTrack 的必选步骤，完成后重新执行 trackEvent() 后可以看到以下日志：

`logcat 日志`

```
/EasyTrackLib:
onEvent：goods_expose
goods_id= 10000
goods_name = 商品名
Try track event goods_expose with provider Umeng.
Try track event goods_expose with provider Sensor.
--------------------------------------------------------
```

5. 页面间参数映射

上一节中有一个`referrerKeyMap`配置项，**定义了全局的页面间参数映射。** 举个例子，在分析不同入口的转化率时，不仅仅需要上报当前页面的数据，还需要上报来源页面的信息。这样我们才能分析用户经过怎样的路径来到当前页面，并最终触发了某个行为。

需要注意的是，来源页面的参数往往不能直接添加到当前页面的埋点参数中，这里一般会有一定的转换规则 / 映射关系。例如：**来源页面的 cur_page 参数，在当前页面应该映射为 from_page 参数。** 在这个例子里，我们配置的映射关系是：

- 来源页面的 cur_page 映射为当前页面的 from_page；
- 来源页面的 cur_tab 映射为当前页面的 from_tab。

因此，假设来源页面传递给当前页面的参数是 A，则当前页面在触发事件时的收集参数是 B：

```
A (来源页面)：
{
    "cur_page" : "list"
    ...
}

B (当前页面)：
{
    "cur_page" : "detail",
    "from_page" : "list",
    ...
}
```

`BaseTrackActivity` 实现了页面间参数映射，你可以创建 BaseActivity 类并继承于 BaseTrackActivity，或者将其内部的逻辑迁移到你的 BaseActivity 中。这一步是可选的，如果你不使用页面间参数映射的特性，你那大可不必使用 BaseTrackActivity。

操作	描述
定义映射关系	1、EasyTrack.referrerKeyMap 配置项 2、重写 BaseTrackActivity #referrerKeyMap() 方法
传递页面间参数	Intent.referrerSnapshot(TrackParams) 扩展函数

`MyGoodsDetailActivity.java`

```
public class MyGoodsDetailActivity extends MyBaseActivity {

    private static final String EXTRA_GOODS = "extra_goods";

    public static void start(Context context, GoodsItem item, TrackParams params)
{
```

```
        Intent intent = new Intent(context, GoodsDetailActivity.class);
        intent.putExtra(EXTRA_GOODS, item);
        EasyTrackUtilsKt.setReferrerSnapshot(intent, params);
        context.startActivity(intent);
    }

    @Nullable
    @Override
    protected String getCurPage() {
        return GOODS_DETAIL_NAME;
    }

    @Nullable
    @Override
    public Map<String, String> referrerKeyMap() {
        Map<String, String> map = new HashMap<>();
        map.put(STORE_ID, STORE_ID);
        map.put(STORE_NAME, STORE_NAME);
        return map;
    }
}
```

需要注意的是，BaseTrackActivity 不会将来源页面的全部参数都添加到当前页面的参数中，**只有在全局 referrerKeyMap 配置项或 referrerKeyMap() 方法中定义了映射关系的参数，才会添加到当前页面。**例如：MyGoodsDetailActivity 继承于 BaseActivity，并重写 referrerKeyMap() 定义了感兴趣的参数（STORE_ID、STORE_NAME）。最终触发埋点时的日志如下：

`logcat 日志`

```
/EasyTrackLib:
onEvent：goods_detail_expose
goods_id= 10000
goods_name = 商品名
store_id = 10000
store_name = 商店名
from_page = Recommend
cur_page = goods_detail
Try track event goods_expose with provider Umeng.
Try track event goods_expose with provider Sensor.
------------------------------------------------------
```

在一般的埋点模型中，每个 Activity (页面) 都有对应一个唯一的 page_id，因此你可以重写 fillTrackParams() 方法追加这些固定的参数。例如：MyBaseActivity 定义了 getCurPage() 方法，子类可以通过重写 getCurPage() 来设置 page_id。

`MyBaseActivity.java`

```
abstract class MyBaseActivity : BaseTrackActivity() {

    @CallSuper
    override fun fillTrackParams(params: TrackParams) {
        super.fillTrackParams(params)
        // 填充页面统一参数
        getCurPage()?.also {
            params.setIfNull(CUR_PAGE, it)
        }
    }

    protected open fun getCurPage(): String? = null
}
```

6. TrackParams 参数容器

TrackParams 是 EasyTrack 收集参数的中间容器，最终会分发给 ITrackProvider 使用。

方法	描述
set(key: String, value: Any?)	设置参数，无论无何都覆盖
setIfNull(key: String, value: Any?)	设置参数，如果已经存在该参数则丢弃
get(key: String): String?	获取参数值，参数不存在则返回 null
get(key: String, default: String?)	获取参数值，参数不存在则返回默认值 default

7. 使用 Kotlin 委托依附参数

如果你觉得每次定义 ITrackModel 数据节点后都需要调用 View.trackModel，你可以使用我定义的 Kotlin 委托 “跳过” 这个步骤，例如：

`MyFragment.kt`

```
private val trackNode by track()
```

`EasyTrackUtils.kt`

```
fun <F : Fragment> F.track(): TrackNodeProperty<F> = FragmentTrackNodeProperty()

fun RecyclerView.ViewHolder.track(): TrackNodeProperty<RecyclerView.ViewHolder>
=
    LazyTrackNodeProperty() viewFactory@{
        return@viewFactory itemView
    }

fun View.track(): TrackNodeProperty<View> = LazyTrackNodeProperty()
viewFactory@{
    return@viewFactory it
}
```

如果你还不了解委托属性，可以看下我之前写过的一篇文章，这里不解释其原理了：[Android | ViewBinding 与 Kotlin 委托双剑合璧]