简介

Tracker 是一个用户操作路径跟踪库。根据预先创建好的一系列路径，当用户的操作和设定一样时，通过回调的方式告知订阅者。

通过 Tracker ，可以实现 指定路径无痕埋点。

观察者-订阅: 观察的意思是指不对原 view 进行任何多余的操作。如 viewClick(R.id.button) 如果你没有对 R.id.button setOnClick 事件，那么你监听是无论如何也不会触发。

使用方式：

1、监听页面的跳转，如 A->B

```
Track.from(MainActivity.class).to(SecondActivity.class).subscribe
(new OnSubscribe<Intent>() {
    @Override
    public void call(Intent intent) {
        Log.d(TAG, "A->B disposable intent=" + intent + " t=" +
Thread.currentThread());
    }
});
```

意思是指 A 跳转（startActivity...）到 B 页面时，会调用 OnSubscribe 回调，在回调中做具体埋点逻辑，参数 intent 是跳转的 intent，可以通过它获取某些数据。

注意：

1、所有的回调, call 方法都是执行在子线程的，不要在此操作 UI。

当然还可以继续 to

```
Track.from(MainActivity.class).to(SecondActivity.class).to(ThirdA
ctivity.class)...
```

表示 A->B->C 的情况。

注意：

1、而且必须是 A->B 后 B->C，如果 A->B->D，D 页面中跳 C，这时是不会触发的，因为第二个 to(C)，其实就指定了它的 from 必须为 B

2、如果：A->B->D ，在 D 中，再返回到 B，再由 B->C ，那么也会触发事件，这里称 B->D （B 跳 D），再 D<-B（D 返回到 B）的操作称为中间操作，无论你的中间操作是什么，点击，还是跳转，都是无关的，因为最终还是做了 B->C 的操作，所以无论你中间做了什么，也还是认为符合你需要的，所以会触发回调。

再来看页面中某 view 被点击的情况

```
Track.from(MainActivity.class).viewClick(R.id.tv_text).subscribe(
new OnSubscribe<View>() {
    @Override
    public void call(View view) {
        Log.d(TAG, "A.viewClick(R.id.tv_text) v:" + view + " t=" +
Thread.currentThread());
    }
});
```

表示的是 MainActivity 中一个 R.id.tv_text 这个 view 被点击时触发。回调参数 view 为被点击的 view 。回调在子线程，可以获取属性，但不能操作 UI.

注: 因为同一 id 可以在多个页面存在, 所以这里是指必须为 MainActivity 页面中的 tv_text Id view 被点击。
View 系列包含下列方法

Track 方法	Aop 对应方法
viewClick(viewid)	view.setOnClickListener
viewVisibility(viewId)	View.setVisiblity
viewLongClick(viewId)	View.setOnLongClickListener

现在可以通过组合成这样的代码:

```
Track.from(MainActivity.class).to(SecondActivity.class).viewClick(R.id.butto
n).to(ThirdActivity.class).subscribe(new OnSubscribe<Intent>() {
    @Override
    public void call(Intent view) {
        Log.d(TAG,"A->B.c->C view:"+view+" t="+Thread.currentThread());
    }
});
```

表示 A->B ,然后 B 页面的 button 被点击，然后跳转到 C 页面。当然，可能是 button 的点击事件中写的跳转，或者是先点击 button ，然后其它操作导致跳转，无所谓，这两种都会触发。但一定是点击事件在跳转前，如果 B 页面的其它操作跳转，但是还没有点击过 button ，此时也不会触发。规则是一定要和定义时事件的顺序一样。

from 表示事件的起源，一般为 activity 。基于此 activity 后面发生的一些操作。
当然，常用 activity 生命周期也可以监听，实现方式是通过 application.registerActivityLifecycleCallbacks.

Track 方法	对应 activity 生命周期
activityOnCreated	onCreate
activityOnStarted	onStart
activityOnResumed	onResume
activityOnPaused	onPause
activityOnStoped	onStop
activityOnDestroyed	onDestroy
activityOnSaveInstanceState	onSaveInstanceState

当然，fragment 系列生命周期也是可以监听的

Track 方法	对应 fragment 生命周期
fragmentOnCreated(xxx.class)	onCreate
fragmentOnStart(xxx.class)	onStart
fragmentOnResumed(xxx.class)	onResume
fragmentOnPaused(xxx.class)	onPause
fragmentOnStoped(xxx.class)	onStop
fragmentOnDestroyed(xxx.class)	onDestroy

`fragmentOnHiddenChanged(...)`	`OnHiddenChanged`
`fragmentSetUserVisibleHintd(..)`	`SetUserVisibleHint`

你可能发现，fragment 的生命周期参数中需要填写具体 fragment.class ，而 activity 生命周期不需要。因为 fragment 不能做为 from ，它必须依附在某 FragmentActivity 上。

```
Track.from(TabFragmentActivity.class).fragmentOnCreate(MainFragment.class)...
```

所以这条代表 TabFragmentActivity 中有一个 fragment（MainFragment）执行了 onCreate 生命周期时触发。

注：fragment 必须是 android.support.v4.app.Fragment，因为 android.app.Fragment 属于 framework 层，无法用 Aop 切入

Dialog 支持方法如下：支持所有 dialog 的子类

Track 方法	对应 Aop 方法
dialogShow(xxxDialog.class)	dialog.show()
dialogDismiss(xxxDialog.class)	dialog.dismiss()
dialogButtonClick(int buttonId)	DialogInterface.Onclickener.onClick

其中 dialogDismiss 事件，只有当你主动调用 dialog.dismiss() 时才会触发。虽然库可以通过主动调用 dialog.setOnDismissListener 来监听，但是不符合观察者模式，所以放弃，不会替用户层做任何多余操作。dialogButtonClick 是 dialog 的三个按钮点击事件，指 POSITIVE,NEGATIVE,NEUTRAL。

PopupWindow 支持方法如下：支持所有 popupWindow 的子类

Track 方法	对应 Aop 方法
popupWindowShow(xxxWindiw.class)	popup.show()
dialogWindowDismiss(xxxWindiw.class)	popup.dismiss()

其中 dialogWindowDismiss 事件，只有当你主动调用 popup.dismiss() 时才会触发。

到目前为止，已经对一些常用控件，常见事件进行了监听，但有一些是我们在代码的逻辑操作，如接口回调，如数据的空与有，需要进行不同的操作，这时该怎么办呢？就算把所有的 view 事件添加进来，也肯定是有不全的，所以此时就需要一种万能事件。onMethodCall。

```
public Track<Object[]> onMethodCall(Class<?> returnClass, String methodName, Class... args) {
```

这个操作符是什么意思呢，就是当某一个方法被调用时，进行回调。首先在方法上添加 @OnMethodCall 注解，（无论是你自定的方法，还是某接口回调的方法，只要你能在源码中添加就可以监听），

方法参数说明 ：返回值 class,方法名 string，参数 class,如有多个，填多个，无则不填。

还有几个重载方法，请看源码。监听的返回值 `Track<Object[]>`就是方法被调时传进来的参数，如:

```
@OnMethodCall
private View setVisibili(TextView view) {
    if (view.getVisibility() == View.VISIBLE) {
        view.setVisibility(View.GONE);
```

```
    } else {
        view.setVisibility(View.VISIBLE);
    }
    return view;
}
```

监听:

```
Track.from(MainActivity.class)
        .onMethodCall(View.class,"setVisibili", TextView.class)
        .subscribe(new OnSubscribe<Object[]>() {
    @Override
    public void call(Object[] objects) {
        Log.d(TAG, "MainActivity onMethodCall.setVisibili args:" +
Arrays.toString(objects));

    }
});
```

注意：这里的

```
Track.from(MainActivity.class).onMethodCall(View.class,"setVisibi
li", TextView.class) 不是说 MainActivity 中的 setVisibili 方法被调用，而
是 setVisibili 这个方法可以在任何地方被调用，那怕是 SecondActivity 中被调用。
是一种和上下文无关的。如果需要精准对某类的某方法被调用时用，用:
Track.fromObject(MainActivity.class).onMethodCall(View.class,"set
Visibili", TextView.class) .fromObject 表示 MainActivity 中，有一个方
法 setVisibili 被调用，等于这种情况  MainActivity.this.setVisibili 或
mMainActivity.setVisibili,两种区别需要注意.
```

再介绍 路径熄灭 lightOff，使路径回到初始状态