Android 性能监控框架 Matrix

1.Matrix介绍

Matrix 是腾讯微信终端团队开发的一套应用性能监控系统（APM），GitHub 地址：Tencent - Matrix。

Matrix-android 当前监控范围包括：应用安装包大小、帧率变化、启动耗时、卡顿、慢方法、SQLite 操作优化、文件读写、内存泄漏等。整个库主要由 5 个组件构成：

1. APK Checker。针对 APK 安装包的分析检测工具，根据一系列设定好的规则，检测 APK 是否存在特定的问题，并输出较为详细的检测结果报告，用于分析排查问题以及版本追踪
2. Resource Canary。基于 WeakReference 的特性和 Square Haha 库开发的 Activity 泄漏和 Bitmap 重复创建检测工具
3. Trace Canary。监控界面流畅性、启动耗时、页面切换耗时、慢函数及卡顿等问题
4. IO Canary。检测文件 IO 问题，包括文件 IO 监控和 Closeable Leak 监控
5. SQLite Lint。按官方最佳实践自动化检测 SQLite 语句的使用质量

使用

Matrix 的使用方式很简单，在 Application 中初始化后启动即可：

```
Matrix.Builder builder = new Matrix.Builder(this);
// 添加需要的插件
builder.plugin(new TracePlugin(...));
builder.plugin(new ResourcePlugin(...));
builder.plugin(new IOCanaryPlugin(...));
builder.plugin(new SQLiteLintPlugin(...));
// 初始化
Matrix matrix = Matrix.init(builder.build());
// 启动
matrix.startAllPlugins();
```

Matrix 类相当于整个库的统一对外接口，资源监控、IO 监控、卡顿监控等功能实现是由其它具体的 Plugin 完成的。

也可以不在 Application 中启动全部插件，而是在某个场景中启动特定的插件，比如：

```
public class TestTraceMainActivity extends Activity {

    @Override
    protected void onCreate(@Nullable Bundle savedInstanceState) {
        // 启动插件
        Plugin plugin = Matrix.with().getPluginByClass(TracePlugin.class);
        if (!plugin.isPluginStarted()) {
            plugin.start();
        }
    }

    @Override
    protected void onDestroy() {
```

```
        super.onDestroy();
        // 停止插件
        Plugin plugin = Matrix.with().getPluginByClass(TracePlugin.class);
        if (plugin.isPluginStarted()) {
            plugin.stop();
        }
    }
}
```

每个具体的 Plugin 都会实现 IPlugin 接口：

```
public interface IPlugin {

    Application getApplication();

    void init(Application application, PluginListener pluginListener);

    void start();

    void stop();

    void destroy();

    String getTag();

    // 在应用可见/不可见时回调
    void onForeground(boolean isForeground);
}
```

可以通过 PluginListener 监听 Plugin 的生命周期变化，或在 Plugin 上报问题时回调：

```
public interface PluginListener {

    void onInit(Plugin plugin);

    void onStart(Plugin plugin);

    void onStop(Plugin plugin);

    void onDestroy(Plugin plugin);

    void onReportIssue(Issue issue);
}
```

上报的问题使用实体类 Issue 包装，Issue 包含 tag、type 等通用字段，详细信息可以通过 JSON 对象 content 获取：

```
public class Issue {
    private int        type;
    private String     tag;
    private String     key;
    private JSONObject content;
    private Plugin     plugin;
}
```

源码简析

Matrix

Matrix 是一个单例类，在构造函数执行时，Matrix 内部的所有 Plugin 都会被初始化：

```
public class Matrix {

    private final HashSet<Plugin> plugins;

    private Matrix(Application app, PluginListener listener, HashSet<Plugin>
plugins) {
        this.plugins = plugins;
        AppActiveMatrixDelegate.INSTANCE.init(application); // 下面会分析
        // 初始化所有 Plugin，并回调 pluginListener
        for (Plugin plugin : plugins) {
            plugin.init(application, pluginListener);
            pluginListener.onInit(plugin);
        }
    }
}
```

Plugin

Plugin 是一个抽象类，每次执行 init / start / stop / destroy 等方法时都会更新状态，并回调 PluginListener：

```
public abstract class Plugin implements IPlugin,
IssuePublisher.OnIssueDetectListener, IAppForeground {

    private int status = PLUGIN_CREATE;

    @Override
    public void init(Application app, PluginListener listener) {
        status = PLUGIN_INITED;
        AppActiveMatrixDelegate.INSTANCE.addListener(this); // 下面会分析
    }

    @Override
    public void start() {
        status = PLUGIN_STARTED;
        pluginListener.onStart(this);
    }

    ...
}
```

如果某个具体 Plugin 上报了一个问题，父类 Plugin 还会对该 Issue 填充 tag、type、process、time 等通用字段，并回调 PluginListener 的 onReportIssue 方法：

```
@Override
public void onDetectIssue(Issue issue) {
    issue.setPlugin(this);
    JSONObject content = issue.getContent();
    // 拼接 tag、type、process、time 等通用字段
    content.put(Issue.ISSUE_REPORT_TAG, issue.getTag());
    content.put(Issue.ISSUE_REPORT_TYPE, issue.getType());
    content.put(Issue.ISSUE_REPORT_PROCESS,
MatrixUtil.getProcessName(application));
    content.put(Issue.ISSUE_REPORT_TIME, System.currentTimeMillis());
    // 回调
    pluginListener.onReportIssue(issue);
}
```

AppActiveMatrixDelegate

Matrix 和 Plugin 都监听了 AppActiveMatrixDelegate，它的主要作用是在应用可见/不可见时通知观察者：

```
public enum AppActiveMatrixDelegate {

    INSTANCE; // 单例

    // 观察者列表
    private final Set<IAppForeground> listeners = new HashSet();

    // 应用可见时通知观察者
    private void onDispatchForeground(String visibleScene) {
        handler.post(() -> {
            isAppForeground = true;
            synchronized (listeners) {
                for (IAppForeground listener : listeners) {
                    listener.onForeground(true);
                }
            }
        });
    }

    // 应用不可见时通知观察者，逻辑和上面的一样
    private void onDispatchBackground(String visibleScene) {
        ...
    }
}
```

判断应用是否可见的逻辑是通过 ActivityLifecycleCallbacks 接口实现的：

```
private final class Controller implements
Application.ActivityLifecycleCallbacks, ComponentCallbacks2 {
```

```
    @Override
    public void onActivityStarted(Activity activity) {
        // 应用可见
        updateScene(activity);
        onDispatchForeground(getVisibleScene());
    }

    @Override
    public void onActivityStopped(Activity activity) {
        // 没有可见的 Activity 了，相当于进入了后台
        if (getTopActivityName() == null) {
            onDispatchBackground(getVisibleScene());
        }
    }

    ...

    @Override
    public void onTrimMemory(int level) {
        // 应用 UI 不可见
        if (level == TRIM_MEMORY_UI_HIDDEN && isAppForeground) { // fallback
            onDispatchBackground(visibleScene);
        }
    }
}
```

总结

Matrix-android 主要包含 5 个组件：APK Checker、Resource Canary、Trace Canary、IO Canary、SQLite Lint。其中 APK Checker 独立运行；其它 4 个模块需要在 Application 中，通过统一对外接口 Matrix 配置完成后执行。每一个模块相当于一个 Plugin，在执行初始化、启动、停止、销毁、报告问题等操作时，都会回调 PluginListener，并更新状态。

每一个 Issue 都有 tag、type、process、time 等 4 个通用字段。

可以监听 AppActiveMatrixDelegate，在应用可见/不可见时，回调 onForeground 方法，以执行相应的操作。应用可见指的是存在可见的 Activity，应用不可见指的是没有可见的 Activity，或者内存不足了，应用的 UI 不可见

2.内存泄漏监控及原理介绍

ResourceCanary 介绍

Matrix 的内存泄漏监控是由 ResourceCanary 实现的，准确的说，ResourceCanary 只能实现 Activity 的内存泄漏检测，但在出现 Activity 内存泄漏时，可以选择 dump 一个堆转储文件，通过该文件，可以分析应用是否存在重复的 Bitmap。

使用

ResourceCanary 是基于 WeakReference 特性和 Square Haha 库开发的 Activity 泄漏和 Bitmap 重复创建检测工具，使用之前，需要进行如下配置：

```
Matrix.Builder builder = new Matrix.Builder(this);

// 用于在用户点击生成的问题通知时，通过这个 Intent 跳转到指定的 Activity
Intent intent = new Intent();
intent.setClassName(this.getPackageName(),
"com.tencent.mm.ui.matrix.ManualDumpActivity");

ResourceConfig resourceConfig = new ResourceConfig.Builder()
        .dynamicConfig(new DynamicConfigImplDemo()) // 用于动态获取一些自定义的选项，
不同 Plugin 有不同的选项
        .setAutoDumpHprofMode(ResourceConfig.DumpMode.AUTO_DUMP) // 自动生成
Hprof 文件
//         .setDetectDebuger(true) //matrix test code
        .setNotificationContentIntent(intent) // 问题通知
        .build();

builder.plugin(new ResourcePlugin(resourceConfig));

// 这个类可用于修复一些内存泄漏问题
ResourcePlugin.activityLeakFixer(this);
```

如果想要在具体的 Activity 中检测内存泄漏，那么获取 Plugin 并执行 start 方法（一般在 onCreate 方法中执行）即可：

```
Plugin plugin = Matrix.with().getPluginByClass(ResourcePlugin.class);
if (!plugin.isPluginStarted()) {
    plugin.start();
}
```

捕获到问题后，会上报信息如下：

```
{
    "tag": "memory",
    "type": 0,
    "process": "sample.tencent.matrix",
    "time": 1590396618440,
    "activity": "sample.tencent.matrix.resource.TestLeakActivity",
}
```

如果 DumpMode 为 AUTO_DUMP，还会生成一个压缩文件，里面包含一个堆转储文件和一个 result.info 文件，可以根据 result.info 文件发现具体是哪一个 Activity 泄漏了：

```
{
    "tag":"memory",
    "process":"com.tencent.mm",

 "resultZipPath":"/storage/emulated/0/Android/data/com.tencent.mm/cache/matrix_r
esource/dump_result_17400_20170713183615.zip",
    "activity":"com.tencent.mm.plugin.setting.ui.setting.SettingsUI",
}
```

配置

ResourcePlugin 执行之前，需要通过 ResourceConfig 配置，配置选项有：

```
public static final class Builder {
    private DumpMode mDefaultDumpHprofMode = DEFAULT_DUMP_HPROF_MODE;
    private IDynamicConfig dynamicConfig;
    private Intent mContentIntent;
    private boolean mDetectDebugger = false;
}
```

其中， ContentIntent 用于发送通知。

DumpMode 用于控制检测到问题后的行为，可选值有：

1. NO_DUMP，是一个轻量级的模式，会回调 Plugin 的 onDetectIssue 方法，但只报告出现内存泄漏问题的 Activity 的名称
2. SILENCE_DUMP，和 NO_DUMP 类似，但会回调 IActivityLeakCallback
3. MANUAL_DUMP，用于生成一个通知，点击后跳转到对应的 Activity，Activity 由 ContentIntent 指定
4. AUTO_DUMP，用于生成堆转储文件

IDynamicConfig 是一个接口，可用于动态获取一些自定义的选项值：

```
public interface IDynamicConfig {
    String get(String key, String defStr);
    int get(String key, int defInt);
    long get(String key, long defLong);
    boolean get(String key, boolean defBool);
    float get(String key, float defFloat);
}
```

和 Resource Canary 相关的选项有：

```
enum ExptEnum {
    //resource
    clicfg_matrix_resource_detect_interval_millis, // 后台线程轮询间隔
    clicfg_matrix_resource_detect_interval_millis_bg, // 应用不可见时的轮询间隔
    clicfg_matrix_resource_max_detect_times, // 重复检测多次后才认为出现了内存泄漏，避免误判
    clicfg_matrix_resource_dump_hprof_enable, // 没见代码有用到
}
```

实现该接口对应的方法，即可通过 ResourceConfig 获取上述选项的值：

```
public final class ResourceConfig {

    // 后台线程轮询间隔默认为 1min
    private static final long DEFAULT_DETECT_INTERVAL_MILLIS =
TimeUnit.MINUTES.toMillis(1);
    // 应用不可见时，后台线程轮询间隔默认为 1min
```

```
    private static final long DEFAULT_DETECT_INTERVAL_MILLIS_BG =
TimeUnit.MINUTES.toMillis(20);
    // 默认重复检测 10 次后，如果依然能获取到 Activity ，才认为出现了内存泄漏
    private static final int DEFAULT_MAX_REDETECT_TIMES = 10;

    public long getScanIntervalMillis() { ... }
    public long getBgScanIntervalMillis() { ... }
    public int getMaxRedetectTimes() { ... }
}
```

可以看到，默认情况下，Resource Canary 在应用可见（onForeground）时每隔 1 分钟检测一次，在应用不可见时每隔 20 分钟检测一次。对于同一个 Activity，在重复检测 10 次后，如果依然能通过弱引用获取，那么就认为出现了内存泄漏。

原理介绍

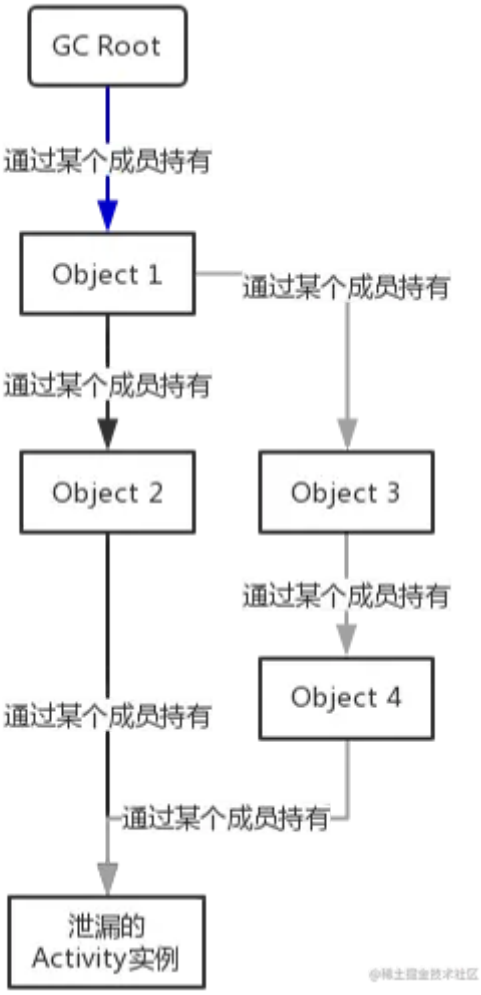
这部分内容摘抄自官方文档。

监测阶段

在监测阶段，对于 4.0 之前的版本，由于没有 ActivityLifecycleCallbacks，而使用反射有性能问题，使用 BaseActivity 又存在侵入性的问题，因此，ResourceCanary 放弃了对 Android 4.0 之前的版本的支持，直接使用 ActivityLifecycleCallbacks 和弱引用来检测 Activity 的内存泄漏。

分析阶段

在分析阶段，由于对 Activity 的强引用链很可能不止一条，因此问题的关键在于找到最短的引用链。比如有如下引用关系：



那么，将 GC Root 和 Object 1 的引用关系解除即可。对于多条 GC Root 引用链的情况，多次检测即可，这样至少保证了每次执行 ResourceCanary 模块的耗时稳定在一个可预计的范围内，不至于在极端情况下耽误其他流程。

LeakCanary 已实现了上述算法，但 Matrix 改进了其中的一些问题：

1. 增加一个一定能被回收的“哨兵”对象，用来确认系统确实进行了GC
2. 直接通过 WeakReference.get() 来判断对象是否已被回收，避免因延迟导致误判
3. 若发现某个 Activity 无法被回收，再重复判断 3 次（0.6.5 版本的代码默认是 10 次），以防在判断时该 Activity 被局部变量持有导致误判
4. 对已判断为泄漏的 Activity，记录其类名，避免重复提示该 Activity 已泄漏

从 Hprof 文件获取所有冗余的 Bitmap 对象

对于这个问题，Android Moniter 已经有完整的实现，原理简单粗暴：把所有未被回收的 Bitmap 的数据 buffer 取出来，然后先对比所有长度为 1 的 buffer，找出相同的，记录所属的 Bitmap 对象；再对比所有长度为 2 的、长度为 3 的 buffer……直到把所有 buffer 都比对完，这样就记录了所有冗余的 Bitmap 对象，接着再套用 LeakCanary 获取引用链的逻辑把这些 Bitmap 对象到 GC Root 的最短强引用链找出来即可。

性能开销

在监测阶段，Resource Canary 的周期性轮询是在后台线程执行的，默认轮询间隔为 1min，以微信通讯录、朋友圈界面的帧率作为参考，接入后应用的平均帧率下降了 10 帧左右，开销并不明显。但 Dump Hprof 的开销较大，整个 App 会卡死约 5～15s。

分析部分放到了服务器环境中执行。实际使用时分析一个 200M 左右的 Hprof 平均需要 15s 左右的时间。此部分主要消耗在引用链分析上，因为需要广度优先遍历完 Hprof 中记录的全部对象。

3.内存泄漏监控源码分析

修复内存泄漏

在开始监测 Activity 内存泄漏之前，Resource Canary 首先会尝试修复可能的内存泄漏问题，它是通过监听 ActivityLifeCycleCallbacks 实现的，在 Activity 回调 onDestroy 时，它会尝试解除 Activity 和 InputMethodManager、View 之间的引用关系：

```
public static void activityLeakFixer(Application application) {
    application.registerActivityLifecycleCallbacks(new
ActivityLifeCycleCallbacksAdapter() {
        @Override
        public void onActivityDestroyed(Activity activity) {
            ActivityLeakFixer.fixInputMethodManagerLeak(activity);
            ActivityLeakFixer.unbindDrawables(activity);
        }
    });
}
```

对于 InputMethodManager，它可能引用了 Activity 中的某几个 View，因此，将它和这几个 View 解除引用关系即可：

```
public static void fixInputMethodManagerLeak(Context destContext) {
    final InputMethodManager imm = (InputMethodManager)
destContext.getSystemService(Context.INPUT_METHOD_SERVICE);
    final String[] viewFieldNames = new String[]{"mCurRootView", "mServedView",
"mNextServedView"};
    for (String viewFieldName : viewFieldNames) {
        final Field paramField = imm.getClass().getDeclaredField(viewFieldName);
        ...
        // 如果 IMM 引用的 View 引用了该 Activity，则切断引用关系
        if (view.getContext() == destContext) {
            paramField.set(imm, null);
        }
    }
}
```

对于 View，它可能通过监听器或 Drawable 的形式关联 Activity，因此，我们需要把每一个可能的引用关系解除掉：

```
public static void unbindDrawables(Activity ui) {
    final View viewRoot = ui.getWindow().peekDecorView().getRootView();
    unbindDrawablesAndRecycle(viewRoot);
}

private static void unbindDrawablesAndRecycle(View view) {
    // 解除通用的 View 引用关系
    recycleView(view);
```

```
    // 不同类型的 View 可能有不同的引用关系，一一处理即可
    if (view instanceof ImageView) {
        recycleImageView((ImageView) view);
    }

    if (view instanceof TextView) {
        recycleTextView((TextView) view);
    }

    ...
}

// 将 Listener、Drawable 等可能存在的引用关系切断
private static void recycleView(View view) {
    view.setOnClickListener(null);
    view.setOnFocusChangeListener(null);
    view.getBackground().setCallback(null);
    view.setBackgroundDrawable(null);
    ...
}
```

监测内存泄漏

具体的监测工作，ResourcePlugin 交给了 ActivityRefWatcher 来完成。

ActivityRefWatcher 主要的三个方法：start、stop、destroy 分别用于启动监听线程、停止监听线程、结束监听。以 start 为例：

```
public class ActivityRefWatcher extends FilePublisher implements Watcher,
IAppForeground {

    @Override
    public void start() {
        stopDetect();
        final Application app = mResourcePlugin.getApplication();
        if (app != null) {
            // 监听 Activity 的 onDestroy 回调，记录 Activity 信息
            app.registerActivityLifecycleCallbacks(mRemovedActivityMonitor);
            // 监听 onForeground 回调，以便根据应用可见状态修改轮询间隔时长
            AppActiveMatrixDelegate.INSTANCE.addListener(this);
            // 启动监听线程
            scheduleDetectProcedure();
        }
    }
}
```

记录 Activity 信息

其中 mRemovedActivityMonitor 用于在 Activity 回调 onDestroy 时记录 Activity 信息，主要包括 Activity 的类名和一个根据 UUID 生成的 key：

```
// 用于记录 Activity 信息
private final ConcurrentLinkedQueue<DestroyedActivityInfo>
mDestroyedActivityInfos;

private final Application.ActivityLifecycleCallbacks mRemovedActivityMonitor =
new ActivityLifeCycleCallbacksAdapter() {

    @Override
    public void onActivityDestroyed(Activity activity) {
        pushDestroyedActivityInfo(activity);
    }
};

// 在 Activity 销毁时，记录 Activity 信息
private void pushDestroyedActivityInfo(Activity activity) {
    final String activityName = activity.getClass().getName();
    final UUID uuid = UUID.randomUUID();
    final String key = keyBuilder.toString(); // 根据 uuid 生成
    final DestroyedActivityInfo destroyedActivityInfo = new
DestroyedActivityInfo(key, activity, activityName);
    mDestroyedActivityInfos.add(destroyedActivityInfo);
}
```

DestroyedActivityInfo 包含信息如下：

```
public class DestroyedActivityInfo {
    public final String mKey; // 根据 uuid 生成
    public final String mActivityName; // 类名
    public final WeakReference<Activity> mActivityRef; // 弱引用
    public int mDetectedCount = 0; // 重复检测次数，默认检测 10 次后，依然能通过弱引用获
取，才认为发生了内存泄漏
}
```

启动监听线程

线程启动后，应用可见时，默认每隔 1min（通过 IDynamicConfig 指定） 将轮询任务发送到默认的后台线程（MatrixHandlerThread）执行：

```
// 自定义的线程切换机制，用于将指定的任务延时发送到主线程/后台线程执行
private final RetryableTaskExecutor             mDetectExecutor;

private ActivityRefWatcher(...) {
    HandlerThread handlerThread = MatrixHandlerThread.getDefaultHandlerThread();
    mDetectExecutor = new RetryableTaskExecutor(config.getScanIntervalMillis(),
handlerThread);
}

private void scheduleDetectProcedure() {
    // 将任务发送到 MatrixHandlerThread 执行
    mDetectExecutor.executeInBackground(mScanDestroyedActivitiesTask);
}
```

下面看轮询任务 mScanDestroyedActivitiesTask，它是一个内部类，代码很长，我们一点一点分析。

设置哨兵检测 GC 是否执行

首先，在上一篇文章关于原理的部分介绍过，ResourceCanary 会设置了一个哨兵元素，检测是否真的执行了 GC，如果没有，它不会往下执行：

```
private final RetryableTask mScanDestroyedActivitiesTask = new RetryableTask() {

    @Override
    public Status execute() {
        // 创建指向一个临时对象的弱引用
        final WeakReference<Object> sentinelRef = new WeakReference<>(new
Object());
        // 尝试触发 GC
        triggerGc();
        // 检测弱引用指向的对象是否存活来判断虚拟机是否真的执行了GC
        if (sentinelRef.get() != null) {
            // System ignored our gc request, we will retry later.
            return Status.RETRY;
        }
        ...
        return Status.RETRY; // 返回 retry，这个任务会一直执行
    }
};

private void triggerGc() {
    Runtime.getRuntime().gc();
    Runtime.getRuntime().runFinalization();
}
```

过滤已上报的 Activity

接着，遍历所有 DestroyedActivityInfo，并标记该 Activity，避免重复上报：

```
final Iterator<DestroyedActivityInfo> infoIt =
mDestroyedActivityInfos.iterator();

while (infoIt.hasNext()) {
    if (!mResourcePlugin.getConfig().getDetectDebugger()
            && isPublished(destroyedActivityInfo.mActivityName) // 如果已标记，则跳
过
            && mDumpHprofMode != ResourceConfig.DumpMode.SILENCE_DUMP) {
        infoIt.remove();
        continue;
    }

    if (mDumpHprofMode == ResourceConfig.DumpMode.SILENCE_DUMP) {
        if (mResourcePlugin != null &&
!isPublished(destroyedActivityInfo.mActivityName)) { // 如果已标记，则跳过
            ...
        }
```

```
        if (null != activityLeakCallback) { // 但还会回调 ActivityLeakCallback
            activityLeakCallback.onLeak(destroyedActivityInfo.mActivityName,
destroyedActivityInfo.mKey);
        }
    } else if (mDumpHprofMode == ResourceConfig.DumpMode.AUTO_DUMP) {
        ...
        markPublished(destroyedActivityInfo.mActivityName); // 标记
    } else if (mDumpHprofMode == ResourceConfig.DumpMode.MANUAL_DUMP) {
        ...
        markPublished(destroyedActivityInfo.mActivityName); // 标记
    } else { // NO_DUMP
        ...
        markPublished(destroyedActivityInfo.mActivityName); // 标记
    }
}
```

多次检测，避免误判

同时，在重复检测大于等于 mMaxRedetectTimes 次时（由 IDynamicConfig 指定，默认为 10），如果还能获取到该 Activity 的引用，才会认为出现了内存泄漏问题：

```
while (infoIt.hasNext()) {
    ...

    // 获取不到，Activity 已回收
    if (destroyedActivityInfo.mActivityRef.get() == null) {
        continue;
    }

    // Activity 未回收，可能出现了内存泄漏，但为了避免误判，需要重复检测多次，如果都能获取到
Activity，才认为出现了内存泄漏
    // 只有在 debug 模式下，才会上报问题，否则只会打印一个 log
    ++destroyedActivityInfo.mDetectedCount;
    if (destroyedActivityInfo.mDetectedCount < mMaxRedetectTimes
            || !mResourcePlugin.getConfig().getDetectDebugger()) {
        MatrixLog.i(TAG, "activity with key [%s] should be recycled but actually
still \n"
                + "exists in %s times, wait for next detection to confirm.",
            destroyedActivityInfo.mKey, destroyedActivityInfo.mDetectedCount);
        continue;
    }
}
```

需要注意的是，只有在 debug 模式下，才会上报问题，否则只会打印一个 log。

上报问题

对于 silence_dump 和 no_dump 模式，它只会记录 Activity 名，并回调 onDetectIssue：

```
final JSONObject resultJson = new JSONObject();
resultJson.put(SharePluginInfo.ISSUE_ACTIVITY_NAME,
destroyedActivityInfo.mActivityName);
mResourcePlugin.onDetectIssue(new Issue(resultJson));
```

对于 manual_dump 模式，它会使用 ResourceConfig 指定的 Intent 生成一个通知：

```
...
Notification notification = buildNotification(context, builder);
notificationManager.notify(NOTIFICATION_ID, notification);
```

对于 auto_dump，它会自动生成一个 hprof 文件并对该文件进行分析：

```
final File hprofFile = mHeapDumper.dumpHeap();
final HeapDump heapDump = new HeapDump(hprofFile, destroyedActivityInfo.mKey,
destroyedActivityInfo.mActivityName);
mHeapDumpHandler.process(heapDump);
```

生成 hprof 文件

dumpHeap 方法做了两件事：生成一个文件，写入 Hprof 数据到文件中：

```
public File dumpHeap() {
    final File hprofFile = mDumpStorageManager.newHprofFile();
    Debug.dumpHprofData(hprofFile.getAbsolutePath());
}
```

之后 HeapDumpHandler 就会处理该文件：

```
protected AndroidHeapDumper.HeapDumpHandler createHeapDumpHandler(...) {
    return new AndroidHeapDumper.HeapDumpHandler() {

        @Override
        public void process(HeapDump result) {
            CanaryWorkerService.shrinkHprofAndReport(context, result);
        }
    };
}
```

处理流程如下：

```
private void doShrinkHprofAndReport(HeapDump heapDump) {
    // 裁剪 hprof 文件
    new HprofBufferShrinker().shrink(hprofFile, shrinkedHProfFile);
    // 压缩裁剪后的 hprof 文件
    zos = new ZipOutputStream(new BufferedOutputStream(new
FileOutputStream(zipResFile)));
```

```
    copyFileToStream(shrinkedHProfFile, zos);
    // 删除旧文件
    shrinkedHProfFile.delete();
    hprofFile.delete();
    // 上报结果
    CanaryResultService.reportHprofResult(this, zipResFile.getAbsolutePath(),
heapDump.getActivityName());
}

private void doReportHprofResult(String resultPath, String activityName) {
    final JSONObject resultJson = new JSONObject();
    resultJson.put(SharePluginInfo.ISSUE_RESULT_PATH, resultPath);
    resultJson.put(SharePluginInfo.ISSUE_ACTIVITY_NAME, activityName);
    Plugin plugin =  Matrix.with().getPluginByClass(ResourcePlugin.class);
    plugin.onDetectIssue(new Issue(resultJson));
}
```

可以看到，由于原始 hprof 文件很大，因此 Matrix 先对它做了一个裁剪优化，接着再压缩裁剪后的文件，并删除旧文件，最后回调 onDetectIssue，上报文件位置、Activity 名称等信息。

分析结果

示例

检测到内存泄漏问题后，ActivityRefWatcher 会打印日志如下：

```
activity with key
[MATRIX_RESCANARY_REFKEY_sample.tencent.matrix.resource.TestLeakActivity_...]
was suspected to be a leaked instance. mode[AUTO_DUMP]
```

如果模式为 AUTO_DUMP，且设置了 mDetectDebugger 为 true，那么，还会生成一个 hprof 文件：

```
hprof: heap dump
"/storage/emulated/0/Android/data/sample.tencent.matrix/cache/matrix_resource/du
mp_*.hprof" starting...
```

裁剪压缩后在 /sdcard/data/[package name]/matrix_resource 文件夹下会生成一个 zip 文件，比如：

```
/storage/emulated/0/Android/data/sample.tencent.matrix/cache/matrix_resource/dum
p_result_*.zip
```

zip 文件里包括一个 dump*shinked.hprof 文件和一个 result.info 文件，其中 result.info 包含设备信息和关键 Activity 的信息，比如：

```
# Resource Canary Result Infomation. THIS FILE IS IMPORTANT FOR THE ANALYZER !!
sdkVersion=23
manufacturer=vivo
hprofEntry=dump_323ff84d95424d35b0f62ef6a3f95838_shrink.hprof
leakedActivityKey=MATRIX_RESCANARY_REFKEY_sample.tencent.matrix.resource.TestLeakActivity_8c5f3e9db8b54a199da6cb2abf68bd12
```

拿到这个 zip 文件，输入路径参数，执行 matrix-resource-canary-analyzer 中的 CLIMain 程序，即可得到一个 result.json 文件：

```
{
    "activityLeakResult": {
        "failure": "null",
        "referenceChain": ["static
sample.tencent.matrix.resource.TestLeakActivity testLeaks", ...,
"sample.tencent.matrix.resource.TestLeakActivity instance"],
        "leakFound": true,
        "className": "sample.tencent.matrix.resource.TestLeakActivity",
        "analysisDurationMs": 185,
        "excludedLeak": false
    },
    "duplicatedBitmapResult": {
        "duplicatedBitmapEntries": [],
        "mFailure": "null",
        "targetFound": false,
        "analyzeDurationMs": 387
    }
}
```

注意，CLIMain 在分析重复 Bitmap 时，需要反射 Bitmap 中的 "mBuffer" 字段，而这个字段在 API 26 已经被移除了，因此，对于 API 大于等于 26 的设备，CLIMain 只能分析 Activity 内存泄漏，无法分析重复 Bitmap。

分析过程

下面简单分析一下 CLIMain 的执行过程，它是基于 Square Haha 开发的，执行过程分为 5 步：

1. 根据 result.info 文件拿到 hprof 文件、sdkVersion 等信息
2. 分析 Activity 泄漏
3. 分析重复 Bitmap
4. 生成 result.json 文件并写入结果
5. 输出重复的 Bitmap 图像到本地

```
public final class CLIMain {
    public static void main(String[] args) {
        doAnalyze();
    }

    private static void doAnalyze() throws IOException {
        // 从 result.info 文件中拿到 hprof 文件、sdkVersion 等信息，接着开始分析
        analyzeAndStoreResult(tempHprofFile, sdkVersion, manufacturer,
leakedActivityKey, extraInfo);
```

```
    }

    private static void analyzeAndStoreResult(...) {
        // 分析 Activity 内存泄漏
        ActivityLeakResult activityLeakResult
                = new ActivityLeakAnalyzer(leakedActivityKey,
).analyze(heapSnapshot);

        // 分析重复 Bitmap
        DuplicatedBitmapResult duplicatedBmpResult
                = new DuplicatedBitmapAnalyzer(mMinBmpLeakSize,
excludedBmps).analyze(heapSnapshot);

        // 生成 result.json 文件并写入结果
        final File resultJsonFile = new File(outputDir, resultJsonName);
        resultJsonPW.println(resultJson.toString());

        // 输出重复的 Bitmap 图像
        for (int i = 0; i < duplicatedBmpEntryCount; ++i) {
            final BufferedImage img = BitmapDecoder.getBitmap(...);
            ImageIO.write(img, "png", os);
        }
    }
}
```

Activity 内存泄漏检测的关键是找到最短引用路径，原理是：

1. 根据 result.info 中的 leakedActivityKey 字段获取 Activity 结点
2. 使用一个集合，存储与该 Activity 存在强引用的所有结点
3. 从这些结点出发，使用宽度优先搜索算法，找到最近的一个 GC Root，GC Root 可能是静态变量、栈帧中的本地变量、JNI 变量等

重复 Bitmap 检测的原理在上一篇文章有介绍，这里跳过。

总结

Resource Canary 的实现原理

1. 注册 ActivityLifeCycleCallbacks，监听 onActivityDestroyed 方法，通过弱引用判断是否出现了内存泄漏，使用后台线程（MatrixHandlerThread）周期性地检测
2. 通过一个“哨兵”对象来确认系统是否进行了 GC
3. 若发现某个 Activity 无法被回收，再重复判断 3 次（0.6.5 版本的代码默认是 10 次），且要求从该 Activity 被记录起有 2 个以上的 Activity 被创建才认为是泄漏（没发现对应的代码），以防在判断时该 Activity 被局部变量持有导致误判
4. 不会重复报告同一个 Activity

Resource Canary 的限制

1. 只能在 Android 4.0 以上的设备运行，因为 ActivityLifeCycleCallbacks 是在 API 14 才加入进来的
2. 无法分析 Android 8.0 及以上的设备的重复 Bitmap 情况，因为 Bitmap 的 mBuffer 字段在 API 26 被移除了

可配置的选项

1. DumpMode。有 no_dump（报告 Activity 类名）、silence_dump（报告 Activity 类名，回调 ActivityLeakCallback）、auto_dump（生成堆转储文件）、manual_dump（发送一个通知） 四种
2. debug 模式，只有在 debug 模式下，DumpMode 才会起作用，否则会持续打印日志
3. ContentIntent，在 DumpMode 模式为 manual_dump 时，会生成一个通知，ContentIntent 可指定跳转的目标 Activity
4. 应用可见/不可见时监测线程的轮询间隔，默认分别是 1min、20min
5. MaxRedetectTimes，只有重复检测大于等于 MaxRedetectTimes 次之后，如果依然能获取到 Activity，才认为出现了内存泄漏

修复内存泄漏

在监测的同时，Resource Canary 使用 ActivityLeakFixer 尝试修复内存泄漏问题，实现原理是切断 InputMethodManager、View 和 Activity 的引用

hprof 文件处理

1. 在 debug 状态下，且 DumpMode 为 audo_dump 时，Matrix 才会在监测到内存泄漏问题后，自动生成一个 hprof 文件
2. 由于原文件很大，因此 Matrix 会对该文件进行裁剪优化，并将裁剪后的 hprof 文件和一个 result.info 文件压缩到一个 zip 包中，result.info 包括 hprof 文件名、sdkVersion、设备厂商、出现内存泄漏的 Activity 类名等信息
3. 拿到这个 zip 文件，输入路径参数，执行 matrix-resource-canary-analyzer 中的 CLIMain 程序，即可得到一个 result.json 文件，从这个文件能获取 Activity 的关键引用路径、重复 Bitmap 等信息

CLIMain 的解析步骤

1. 根据 result.info 文件拿到 hprof 文件、Activity 类名等关键信息
2. 分析 Activity 泄漏
3. 分析重复 Bitmap
4. 生成 result.json 文件并写入结果
5. 输出重复的 Bitmap 图像到本地

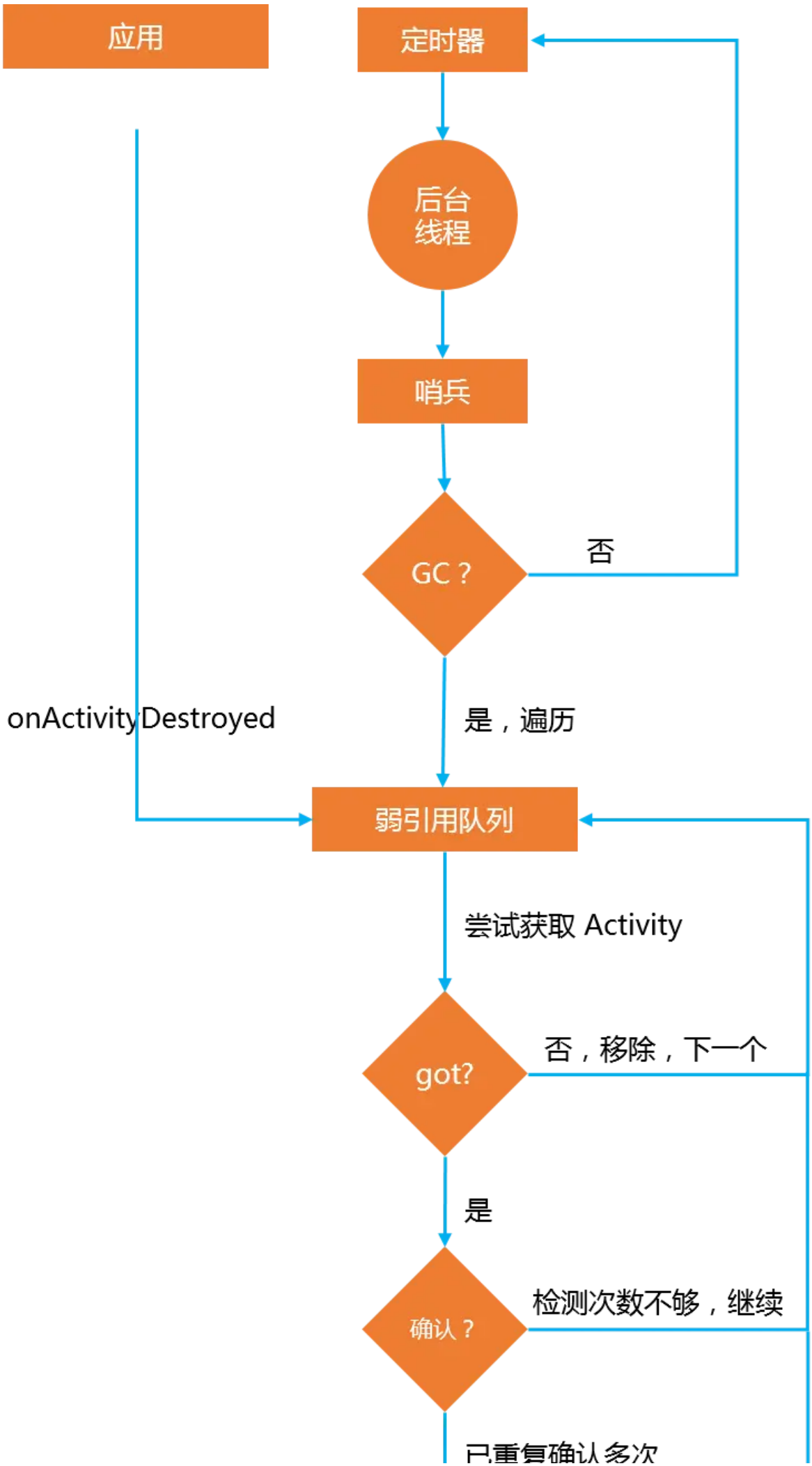
最短路径查找

Activity 内存泄漏检测的关键是找到最短引用路径，原理是：

1. 根据 result.info 中的 leakedActivityKey 字段获取 Activity 结点
2. 使用一个集合，存储与该 Activity 存在强引用的所有结点
3. 从这些结点出发，使用宽度优先搜索算法，找到最近的一个 GC Root，GC Root 可能是静态变量、栈帧中的本地变量、JNI 变量等

重复 Bitmap 的分析原理

把所有未被回收的 Bitmap 的数据 buffer 取出来，然后先对比所有长度为 1 的 buffer，找出相同的，记录所属的 Bitmap 对象；再对比所有长度为 2 的、长度为 3 的 buffer……直到把所有 buffer 都比对完，这样就记录了所有冗余的 Bitmap 对象。


应用
定时器
后台
线程
哨兵
GC ?
否
onActivityDestroyed
是，遍历
弱引用队列
尝试获取 Activity
got?
否，移除，下一个
是
确认 ?
检测次数不够，继续

4.Hprof 文件分析

Hprof 文件格式

Hprof 文件使用的基本数据类型为：u1、u2、u4、u8，分别表示 1 byte、2 byte、4 byte、8 byte 的内容，由文件头和文件内容两部分组成。

其中，文件头包含以下信息：

长度	含义
[u1]*	以 null 结尾的一串字节，用于表示格式名称及版本，比如 JAVA PROFILE 1.0.1（由 18 个 u1 字节组成）
u4	size of identifiers，即字符串、对象、堆栈等信息的 id 的长度（很多 record 的具体信息需要通过 id 来查找）
u8	时间戳，时间戳，1970/1/1 以来的毫秒数

文件内容由一系列 records 组成，每一个 record 包含如下信息：

长度	含义
u1	TAG，表示 record 类型
u4	TIME，时间戳，相对文件头中的时间戳的毫秒数
u4	LENGTH，即 BODY 的字节长度
u4	BODY，具体内容

查看 hprof.cc 可知，Hprof 文件定义的 TAG 有：

```
enum HprofTag {
  HPROF_TAG_STRING = 0x01,            // 字符串
  HPROF_TAG_LOAD_CLASS = 0x02,        // 类
  HPROF_TAG_UNLOAD_CLASS = 0x03,
  HPROF_TAG_STACK_FRAME = 0x04,        // 栈帧
  HPROF_TAG_STACK_TRACE = 0x05,        // 堆栈
  HPROF_TAG_ALLOC_SITES = 0x06,
  HPROF_TAG_HEAP_SUMMARY = 0x07,
  HPROF_TAG_START_THREAD = 0x0A,
  HPROF_TAG_END_THREAD = 0x0B,
  HPROF_TAG_HEAP_DUMP = 0x0C,            // 堆
```

```
  HPROF_TAG_HEAP_DUMP_SEGMENT = 0x1C,
  HPROF_TAG_HEAP_DUMP_END = 0x2C,
  HPROF_TAG_CPU_SAMPLES = 0x0D,
  HPROF_TAG_CONTROL_SETTINGS = 0x0E,
};
```

需要重点关注的主要是三类信息：

1. 字符串信息：保存着所有的字符串，在解析时可通过索引 id 引用
2. 类的结构信息：包括类内部的变量布局，父类的信息等等
3. 堆信息：内存占用与对象引用的详细信息

如果是堆信息，即 TAG 为 HEAP_DUMP 或 HEAP_DUMP_SEGMENT 时，那么其 BODY 由一系列子 record 组成，这些子 record 同样使用 TAG 来区分：

```
enum HprofHeapTag {
  // Traditional.
  HPROF_ROOT_UNKNOWN = 0xFF,
  HPROF_ROOT_JNI_GLOBAL = 0x01,         // native 变量
  HPROF_ROOT_JNI_LOCAL = 0x02,
  HPROF_ROOT_JAVA_FRAME = 0x03,
  HPROF_ROOT_NATIVE_STACK = 0x04,
  HPROF_ROOT_STICKY_CLASS = 0x05,
  HPROF_ROOT_THREAD_BLOCK = 0x06,
  HPROF_ROOT_MONITOR_USED = 0x07,
  HPROF_ROOT_THREAD_OBJECT = 0x08,
  HPROF_CLASS_DUMP = 0x20,              // 类
  HPROF_INSTANCE_DUMP = 0x21,           // 实例对象
  HPROF_OBJECT_ARRAY_DUMP = 0x22,       // 对象数组
  HPROF_PRIMITIVE_ARRAY_DUMP = 0x23,    // 基础类型数组

  // Android.
  HPROF_HEAP_DUMP_INFO = 0xfe,
  HPROF_ROOT_INTERNED_STRING = 0x89,
  HPROF_ROOT_FINALIZING = 0x8a,  // Obsolete.
  HPROF_ROOT_DEBUGGER = 0x8b,
  HPROF_ROOT_REFERENCE_CLEANUP = 0x8c,  // Obsolete.
  HPROF_ROOT_VM_INTERNAL = 0x8d,
  HPROF_ROOT_JNI_MONITOR = 0x8e,
  HPROF_UNREACHABLE = 0x90,  // Obsolete.
  HPROF_PRIMITIVE_ARRAY_NODATA_DUMP = 0xc3,  // Obsolete.
};
```

每一个 TAG 及其对应的内容可参考 HPROF Agent，比如，String record 的格式如下：

STRING IN UTF8	0x01	ID	ID for this string
		[u1]*	UTF8 characters for string (NOT NULL terminated)

因此，在读取 Hprof 文件时，如果 TAG 为 0x01，那么，当前 record 就是字符串，第一部分信息是字符串 ID，第二部分就是字符串的内容。

Hprof 文件裁剪

Matrix 的 Hprof 文件裁剪功能的目标是将 Bitmap 和 String 之外的所有对象的基础类型数组的值移除，因为 Hprof 文件的分析功能只需要用到字符串数组和 Bitmap 的 buffer 数组。另一方面，如果存在不同的 Bitmap 对象其 buffer 数组值相同的情况，则可以将它们指向同一个 buffer，以进一步减小文件尺寸。裁剪后的 Hprof 文件通常比源文件小 1/10 以上。

代码结构和 ASM 很像，主要由 HprofReader、HprofVisitor、HprofWriter 组成，分别对应 ASM 中的 ClassReader、ClassVisitor、ClassWriter。

HprofReader 用于读取 Hprof 文件中的数据，每读取到一种类型（使用 TAG 区分）的数据，就交给一系列 HprofVisitor 处理，最后由 HprofWriter 输出裁剪后的文件（HprofWriter 继承自 HprofVisitor）。

裁剪流程如下：

```
// 裁剪
public void shrink(File hprofIn, File hprofOut) throws IOException {
    // 读取文件
    final HprofReader reader = new HprofReader(new BufferedInputStream(is));
    // 第一遍读取
    reader.accept(new HprofInfoCollectVisitor());
    // 第二遍读取
    is.getChannel().position(0);
    reader.accept(new HprofKeptBufferCollectVisitor());
    // 第三遍读取，输出裁剪后的 Hprof 文件
    is.getChannel().position(0);
    reader.accept(new HprofBufferShrinkVisitor(new HprofWriter(os)));
}
```

可以看到，Matrix 为了完成裁剪功能，需要对输入的 hprof 文件重复读取三次，每次都由一个对应的 Visitor 处理。

读取 Hprof 文件

HprofReader 的源码很简单，先读取文件头，再读取 record，根据 TAG 区分 record 的类型，接着按照 HPROF Agent 给出的格式依次读取各种信息即可，读取完成后交给 HprofVisitor 处理。

读取文件头：

```
// 读取文件头
private void acceptHeader(HprofVisitor hv) throws IOException {
    final String text = IOUtil.readNullTerminatedString(mStreamIn); // 连续读取数据，直到读取到 null
    mIdSize = IOUtil.readBEInt(mStreamIn); // int 是 4 字节
    final long timestamp = IOUtil.readBELong(mStreamIn); // long 是 8 字节
    hv.visitHeader(text, idSize, timestamp); // 通知 Visitor
}
```

读取 record（以字符串为例）：

```
// 读取文件内容
private void acceptRecord(HprofVisitor hv) throws IOException {
    while (true) {
```

```
        final int tag = mStreamIn.read(); // TAG 区分类型
        final int timestamp = IOUtil.readBEInt(mStreamIn); // 时间戳
        final long length = IOUtil.readBEInt(mStreamIn) & 0x00000000FFFFFFFFL;
// Body 字节长
        switch (tag) {
            case HprofConstants.RECORD_TAG_STRING: // 字符串
                acceptStringRecord(timestamp, length, hv);
                break;
            ... // 其它类型
        }
    }
}

// 读取 String record
private void acceptStringRecord(int timestamp, long length, HprofVisitor hv)
throws IOException {
    final ID id = IOUtil.readID(mStreamIn, mIdSize); // IdSize 在读取文件头时确定
    final String text = IOUtil.readString(mStreamIn, length - mIdSize); // Body
字节长减去 IdSize 剩下的就是字符串内容
    hv.visitStringRecord(id, text, timestamp, length);
}
```

记录 Bitmap 和 String 类信息

为了完成上述裁剪目标，首先需要找到 Bitmap 及 String 类，及其内部的 mBuffer、value 字段，这也是裁剪流程中的第一个 Visitor 的作用：记录 Bitmap 和 String 类信息。

包括字符串 ID：

```
// 找到 Bitmap、String 类及其内部字段的字符串 ID
public void visitStringRecord(ID id, String text, int timestamp, long length) {
    if (mBitmapClassNameStringId == null &&
"android.graphics.Bitmap".equals(text)) {
        mBitmapClassNameStringId = id;
    } else if (mMBufferFieldNameStringId == null && "mBuffer".equals(text)) {
        mMBufferFieldNameStringId = id;
    } else if (mMRecycledFieldNameStringId == null && "mRecycled".equals(text))
{
        mMRecycledFieldNameStringId = id;
    } else if (mStringClassNameStringId == null &&
"java.lang.String".equals(text)) {
        mStringClassNameStringId = id;
    } else if (mValueFieldNameStringId == null && "value".equals(text)) {
        mValueFieldNameStringId = id;
    }
}
```

Class ID：

```
// 找到 Bitmap 和 String 的 Class ID
public void visitLoadClassRecord(int serialNumber, ID classObjectId, int
stackTraceSerial, ID classNameStringId, int timestamp, long length) {
    if (mBmpClassId == null && mBitmapClassNameStringId != null &&
mBitmapClassNameStringId.equals(classNameStringId)) {
        mBmpClassId = classObjectId;
    } else if (mStringClassId == null && mStringClassNameStringId != null &&
mStringClassNameStringId.equals(classNameStringId)) {
        mStringClassId = classObjectId;
    }
}
```

以及它们拥有的字段：

```
// 记录 Bitmap 和 String 类的字段信息
public void visitHeapDumpClass(ID id, int stackSerialNumber, ID superClassId, ID
classLoaderId, int instanceSize, Field[] staticFields, Field[] instanceFields) {
    if (mBmpClassInstanceFields == null && mBmpClassId != null &&
mBmpClassId.equals(id)) {
        mBmpClassInstanceFields = instanceFields;
    } else if (mStringClassInstanceFields == null && mStringClassId != null &&
mStringClassId.equals(id)) {
        mStringClassInstanceFields = instanceFields;
    }
}
```

第二个 Visitor 用于记录所有 String 对象的 value ID：

```
// 如果是 String 对象，则添加其内部字段 "value" 的 ID
public void visitHeapDumpInstance(ID id, int stackId, ID typeId, byte[]
instanceData) {
    if (mStringClassId != null && mStringClassId.equals(typeId)) {
        if (mValueFieldNameStringId.equals(fieldNameStringId)) {
            strValueId = (ID) IOUtil.readValue(bais, fieldType, mIdSize);
        }
        mStringValueIds.add(strValueId);
    }
}
```

以及 Bitmap 对象的 Buffer ID 与其对应的数组本身：

```
// 如果是 Bitmap 对象，则添加其内部字段 "mBuffer" 的 ID
public void visitHeapDumpInstance(ID id, int stackId, ID typeId, byte[]
instanceData) {
    if (mBmpClassId != null && mBmpClassId.equals(typeId)) {
        if (mMBufferFieldNameStringId.equals(fieldNameStringId)) {
            bufferId = (ID) IOUtil.readValue(bais, fieldType, mIdSize);
        }
        mBmpBufferIds.add(bufferId);
    }
}
```

```
// 保存 Bitmap 对象的 mBuffer ID 及数组的映射关系
public void visitHeapDumpPrimitiveArray(int tag, ID id, int stackId, int
numElements, int typeId, byte[] elements) {
    mBufferIdToElementDataMap.put(id, elements);
}
```

接着分析所有 Bitmap 对象的 buffer 数组，如果其 MD5 相等，说明是同一张图片，就将这些重复的 buffer ID 映射起来，以便之后将它们指向同一个 buffer 数组，删除其它重复的数组：

```
final String buffMd5 = DigestUtil.getMD5String(elementData);
final ID mergedBufferId = duplicateBufferFilterMap.get(buffMd5); // 根据该 MD5 值
对应的 buffer id
if (mergedBufferId == null) { // 如果 buffer id 为空，说明是一张新的图片
    duplicateBufferFilterMap.put(buffMd5, bufferId);
} else { // 否则是相同的图片，将当前的 Bitmap buffer 指向之前保存的 buffer id，以便之后删
除重复的图片数据
    mBmpBufferIdToDeduplicatedIdMap.put(mergedBufferId, mergedBufferId);
    mBmpBufferIdToDeduplicatedIdMap.put(bufferId, mergedBufferId);
}
```

裁剪 Hprof 文件数据

将上述数据收集完成之后，就可以输出裁剪后的文件了，裁剪后的 Hprof 文件的写入功能由 HprofWriter 完成，代码很简单，HprofReader 读取到数据之后就由 HprofWriter 原封不动地输出到新的文件即可，唯二需要注意的就是 Bitmap 和基础类型数组。

先看 Bitmap，在输出 Bitmap 对象时，需要将相同的 Bitmap 数组指向同一个 buffer ID，以便接下来剔除重复的 buffer 数据：

```
// 将相同的 Bitmap 数组指向同一个 buffer ID
public void visitHeapDumpInstance(ID id, int stackId, ID typeId, byte[]
instanceData) {
    if (typeId.equals(mBmpClassId)) {
        ID bufferId = (ID) IOUtil.readValue(bais, fieldType, mIdSize);
        // 找到共同的 buffer id
        final ID deduplicatedId = mBmpBufferIdToDeduplicatedIdMap.get(bufferId);
        if (deduplicatedId != null && !bufferId.equals(deduplicatedId) &&
!bufferId.equals(mNullBufferId)) {
            modifyIdInBuffer(instanceData, bufferIdPos, deduplicatedId);
        }
        // 修改完毕后再写入到新文件中
        super.visitHeapDumpInstance(id, stackId, typeId, instanceData);
    }

    // 修改成对应的 buffer id
    private void modifyIdInBuffer(byte[] buf, int off, ID newId) {
        final ByteBuffer bBuf = ByteBuffer.wrap(buf);
        bBuf.position(off);
        bBuf.put(newId.getBytes());
    }
}
```

对于基础类型数组，如果不是 Bitmap 中的 mBuffer 字段或者 String 中的 value 字段，则不写入到新文件中：

```
public void visitHeapDumpPrimitiveArray(int tag, ID id, int stackId, int
numElements, int typeId, byte[] elements) {
    final ID deduplicatedID = mBmpBufferIdToDeduplicatedIdMap.get(id);
    // 如果既不是 Bitmap 中的 mBuffer 字段，也不是 String 中的 value 字段，则舍弃该数据
    // 如果当前 id 不等于 deduplicatedID，说明这是另一张重复的图片，它的图像数据不需要重复
输出
    if (!id.equals(deduplicatedID) && !mStringValueIds.contains(id)) {
        return; // 直接返回，不写入新文件中
    }
    super.visitHeapDumpPrimitiveArray(tag, id, stackId, numElements, typeId,
elements);
}
```

总结

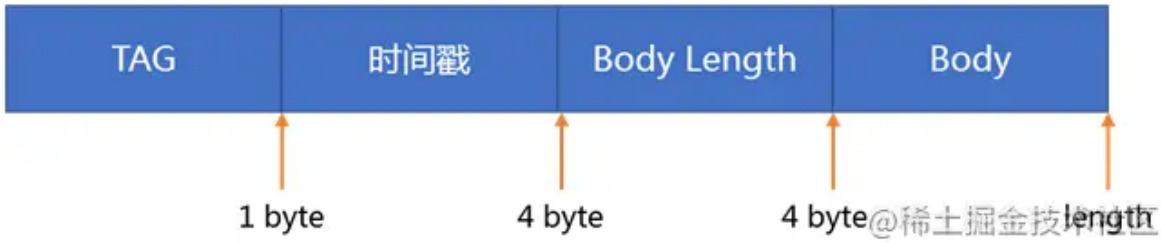
Hprof 文件格式

Hprof 文件由文件头和文件内容两部分组成，文件内容由一系列 records 组成，record 的类型则通过 TAG 来区分。

Hprof 文件格式示意图：

文件头：

record：



其中文件内容需要关注的主要是三类信息：

1. 字符串信息：保存着所有的字符串，在解析时可通过索引 id 引用
2. 类的结构信息：包括类内部的变量布局，父类的信息等等
3. 堆信息：内存占用与对象引用的详细信息

更详细的格式可参考文档 HPROF Agent。

Hprof 文件裁剪

Matrix 的 Hprof 文件裁剪功能的目标是将 Bitmap 和 String 之外的所有对象的基础类型数组的值移除，因为 Hprof 文件的分析功能只需要用到字符串数组和 Bitmap 的 buffer 数组。另一方面，如果存在不同的 Bitmap 对象其 buffer 数组值相同的情况，则可以将它们指向同一个 buffer，以进一步减小文件尺寸。裁剪后的 Hprof 文件通常比源文件小 1/10 以上。

Hprof 文件裁剪功能的代码结构和 ASM 很像，主要由 HprofReader、HprofVisitor、HprofWriter 组成，HprofReader 用于读取 Hprof 文件中的数据，每读取到一种类型（使用 TAG 区分）的数据（即 record），就交给一系列 HprofVisitor 处理，最后由 HprofWriter 输出裁剪后的文件（HprofWriter 继承自 HprofVisitor）。

裁剪流程如下：

1. 读取 Hprof 文件
2. 记录 Bitmap 和 String 类信息
3. 移除 Bitmap buffer 和 String value 之外的基础类型数组
4. 将同一张图片的 Bitmap buffer 指向同一个 buffer id，移除重复的 Bitmap buffer
5. 其它数据原封不动地输出到新文件中

需要注意的是，Bitmap 的 mBuffer 字段在 API 26 被移除了，因此 Matrix 无法分析 API 26 以上的设备的重复 Bitmap。

5.卡顿监控

使用

Matrix 中负责卡顿监控的组件是 TraceCanary，它是基于 ASM 插桩实现的，用于监控界面流畅性、启动耗时、页面切换耗时、慢函数及卡顿等问题。和 ResourceCanary 类似，使用前需要配置如下，主要包括帧率、耗时方法、ANR、启动等选项：

```
TraceConfig traceConfig = new TraceConfig.Builder()
        .dynamicConfig(dynamicConfig)
        .enableFPS(fpsEnable) // 帧率
        .enableEvilMethodTrace(traceEnable) // 耗时方法
        .enableAnrTrace(traceEnable) // ANR
        .enableStartup(traceEnable) // 启动
        .splashActivities("sample.tencent.matrix.SplashActivity;") // 可指定多个启动页，使用分号 ";" 分割
        .isDebug(true)
        .isDevEnv(false)
        .build();
TracePlugin tracePlugin = new TracePlugin(traceConfig);
```

接着在 Application / Activity 中启动即可：

```
tracePlugin.start();
```

除了以上配置之外，Trace Canary 还有如下自定义的配置选项：

```
enum ExptEnum {
    // trace
    clicfg_matrix_trace_care_scene_set, // 闪屏页
    clicfg_matrix_trace_fps_time_slice, // 如果同一 Activity 掉帧数 * 16.66ms > time_slice，就上报，默认为 10s
    clicfg_matrix_trace_evil_method_threshold, // 慢方法的耗时阈值，默认为 700ms

    clicfg_matrix_fps_dropped_normal, // 正常掉帧数，默认 [3, 9)
    clicfg_matrix_fps_dropped_middle, // 中等掉帧数，默认 [9, 24)
    clicfg_matrix_fps_dropped_high, // 高掉帧数，默认 [24, 42)
    clicfg_matrix_fps_dropped_frozen, // 卡顿掉帧数，默认 [42, ~)

    clicfg_matrix_trace_app_start_up_threshold, // 冷启动时间阈值，默认 10s
    clicfg_matrix_trace_warm_app_start_up_threshold, // 暖启动时间阈值，默认 4s
}
```

报告

相比内存泄漏，卡顿监控报告的信息复杂了很多。

ANR

出现 ANR 时，Matrix 上报信息如下：

```
{
    "tag": "Trace_EvilMethod",
    "type": 0,
    "process": "sample.tencent.matrix",
    "time": 1590397340910,
    "machine": "HIGH", // 设备等级
```

```
    "cpu_app": 0.001405802921652454, // 应用占用的 CPU 时间比例，appTime/cpuTime *
100
    "mem": 3030949888,   // 设备总运行内存
    "mem_free": 1695964, // 设备可用的运行内存，不绝对，可能有部分已经被系统内核使用
    "detail": "ANR",
    "cost": 5006,  // 方法执行总耗时
    "stackKey": "30|", // 关键方法
    "scene": "sample.tencent.matrix.trace.TestTraceMainActivity",
    "stack": "0,1048574,1,5006\n1,15,1,5004\n2,30,1,5004\n", // 方法执行关键路径
    "threadStack": " \nat android.os.SystemClock:sleep(120)\nat
sample.tencent.matrix.trace.TestTraceMainActivity:testInnerSleep(234)\nat
sample.tencent.matrix.trace.TestTraceMainActivity:testANR(135)\nat
java.lang.reflect.Method:invoke(-2)\nat
android.view.View$DeclaredOnClickListener:onClick(4461)\nat
android.view.View:performClick(5212)\nat
android.view.View$PerformClick:run(21214)\nat
android.os.Handler:handleCallback(739)\nat
android.os.Handler:dispatchMessage(95)\nat android.os.Looper:loop(148)\nat
android.app.ActivityThread:main(5619)\n", // 线程堆栈
    "processPriority": 20, // 进程优先级
    "processNice": 0,      // 进程的 nice 值
    "isProcessForeground": true,  // 应用是否可见
    "memory": {
        "dalvik_heap": 17898, // 虚拟机中已分配的 Java 堆内存，kb
        "native_heap": 6796,  // 已分配的本地内存，kb
        "vm_size": 858132,    // 虚拟内存大小，指进程总共可访问的地址空间，kb
    }
}
```

其中，设备分级如下：

1. BEST，内存大于等于 4 G
2. HIGH，内存大于等于 3G，或内存大于等于 2G 且 CPU 核心数大于等于 4 个
3. MIDDLE，内存大于等于 2G 且 CPU 核心数大于等于 2 个，或内存大于等于 1G 且 CPU 核心数大于等于 4 个
4. LOW，内存大于等于 1G
5. BAD，内存小于 1G

启动

正常启动情况下：

```
{
    "tag": "Trace_StartUp",
    "type": 0,
    "process": "sample.tencent.matrix",
    "time": 1590405971796,
    "machine": "HIGH",
    "cpu_app": 2.979125443261738E-4,
    "mem": 3030949888,
    "mem_free": 1666132,
    "application_create": 35, // 应用启动耗时
    "application_create_scene": 100, // 启动场景
    "first_activity_create": 318, // 第一个 activity 启动耗时
```

```
    "startup_duration": 2381, // 启动总耗时
    "is_warm_start_up": false, // 是否是暖启动
}
```

其中，application_create、first_activity_create、startup_duration 分别对应 applicationCost、firstScreenCost、coldCost：

```
firstMethod.i       LAUNCH_ACTIVITY   onWindowFocusChange   LAUNCH_ACTIVITY
 onWindowFocusChange
^                        ^                  ^                    ^
       ^
|                        |                  |                    |
       |
|---------app---------|---|---firstActivity---|---------...---------|---
careActivity---|
|<--applicationCost-->|
|<--------------firstScreenCost-------------->|
|<-----------------------------------coldCost-------------------------------
------>|
.                        |<-----warmCost---->|
```

启动场景分为 4 种：

1. 100，Activity 拉起的
2. 114，Service 拉起的
3. 113，Receiver 拉起的
4. -100，未知，比如 ContentProvider

如果是启动过慢的情况：

```
{
    "tag": "Trace_EvilMethod",
    "type": 0,
    "process": "sample.tencent.matrix",
    "time": 1590407016547,
    "machine": "HIGH",
    "cpu_app": 3.616498950411638E-4,
    "mem": 3030949888,
    "mem_free": 1604416,
    "detail": "STARTUP",
    "cost": 2388,
    "stack":
"0,2,1,43\n1,121,1,0\n1,1,8,0\n2,99,1,0\n0,1048574,1,0\n0,1048574,1,176\n1,15,1,
144\n0,1048574,1,41\n",
    "stackKey": "2|",
    "subType": 1 // 1 代表冷启动，2 代表暖启动
}
```

慢方法

```
{
```

```
    "tag": "Trace_EvilMethod",
    "type": 0,
    "process": "sample.tencent.matrix",
    "time": 1590407411286,
    "machine": "HIGH",
    "cpu_app": 8.439117339531338E-4,
    "mem": 3030949888,
    "mem_free": 1656536,
    "detail": "NORMAL",
    "cost": 804, // 方法执行总耗时
    "usage": "0.37%",  // 在方法执行总时长中，当前线程占用的 CPU 时间比例
    "scene": "sample.tencent.matrix.trace.TestTraceMainActivity",
    "stack": "0,1048574,1,804\n1,14,1,803\n2,29,1,798\n",
    "stackKey": "29|"
}
```

帧率

在出现掉帧的情况时，Matrix 上报信息如下：

```
{
    "tag": "Trace_FPS",
    "type": 0,
    "process": "sample.tencent.matrix",
    "time": 1590408900258,
    "machine": "HIGH",
    "cpu_app": 0.0030701181354057853,
    "mem": 3030949888,
    "mem_free": 1642296,
    "scene": "sample.tencent.matrix.trace.TestFpsActivity",
    "dropLevel": { // 不同级别的掉帧问题出现的次数
        "DROPPED_FROZEN": 0,
        "DROPPED_HIGH": 0,
        "DROPPED_MIDDLE": 3,
        "DROPPED_NORMAL": 14,
        "DROPPED_BEST": 451
    },
    "dropSum": { // 不同级别的掉帧问题对应的总掉帧数
        "DROPPED_FROZEN": 0,
        "DROPPED_HIGH": 0,
        "DROPPED_MIDDLE": 41,
        "DROPPED_NORMAL": 57,
        "DROPPED_BEST": 57
    },
    "fps": 46.38715362548828, // 帧率
    "dropTaskFrameSum": 0 // 意义不明，正常情况下值总是为 0
}
```

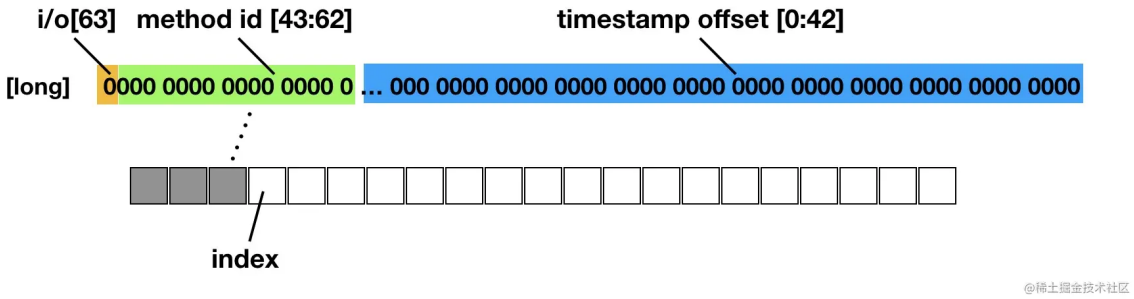
原理介绍

（注：这部分内容主要整理自 Matrix 的官方文档）

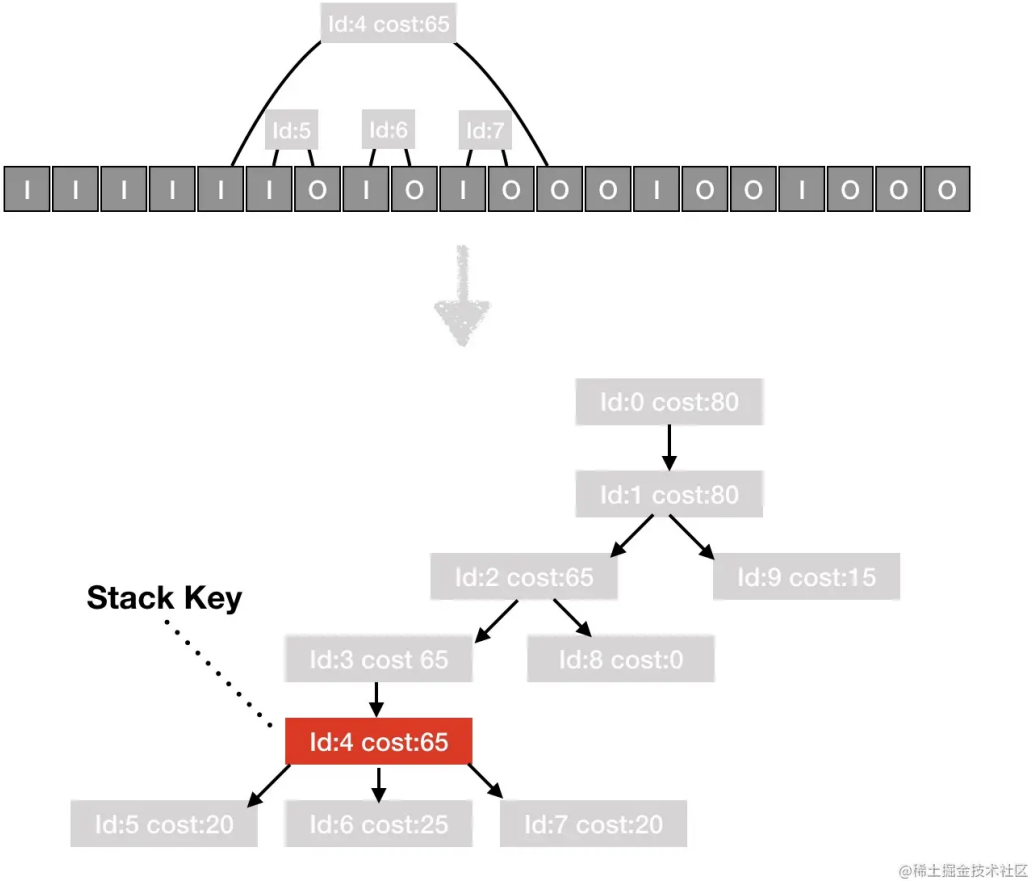
开头说到，Matrix 的卡顿监控是基于 ASM 插桩实现的，其原理是通过代理编译期间的任务 transformClassesWithDexTask，将全局 class 文件作为输入，利用 ASM 工具对所有 class 文件进行扫描及插桩，插桩的意思是在每一个方法的开头处插入 AppMethodBeat.i 方法，在方法的结尾处插入 AppMethodBeat.o 方法，并记录时间戳，这样就能知道该方法的执行耗时。

插桩过程有几个关键点：

1. 选择在编译任务执行时插桩，是因为 proguard 操作是在该任务之前就完成的，意味着插桩时的 class 文件已经被混淆过的。而选择 proguard 之后去插桩，是因为如果提前插桩会造成部分方法不符合内联规则，没法在 proguard 时进行优化，最终导致程序方法数无法减少，从而引发方法数过大问题
2. 为了减少插桩量及性能损耗，通过遍历 class 方法指令集，判断扫描的函数是否只含有 PUT/READ FIELD 等简单的指令，来过滤一些默认或匿名构造函数，以及 get/set 等简单不耗时函数。
3. 针对界面启动耗时，因为要统计从 Activity#onCreate 到 Activity#onWindowFocusChange 间的耗时，所以在插桩过程中需要收集应用内所有 Activity 的实现类，并覆盖 onWindowFocusChange 函数进行打点。
4. 为了方便及高效记录函数执行过程，Matrix 为每个插桩的函数分配一个独立 ID，在插桩过程中，记录插桩的函数签名及分配的 ID，在插桩完成后输出一份 mapping，作为数据上报后的解析支持。为了优化内存使用，method id 及时间戳是通过一个 long 数组记录的，格式如下：



1. 堆栈聚类问题： 如果将收集的原始数据进行上报，数据量很大而且后台很难聚类有问题的堆栈，所以在上报之前需要对采集的数据进行简单的整合及裁剪，并分析出一个能代表卡顿堆栈的 key，方便后台聚合。具体的方法是通过遍历采集的 buffer ，相邻 i 与 o 为一次完整函数执行，计算出一个调用树及每个函数执行耗时，并对每一级中的一些相同执行函数做聚合，最后通过一个简单策略，分析出主要耗时的那一级函数，作为代表卡顿堆栈的key。



帧率监控的方法是向 Choreographer 注册监听，在每一帧 doframe 回调时判断距离上一帧的时间差是否超出阈值（卡顿），如果超出阈值，则获取数组 index 前的所有数据（即两帧之间的所有函数执行信息）进行分析上报。

ANR 监控则更简单，在每一帧 doFrame 到来时，重置一个定时器，并往 buffer 数组里插入一个结点，如果 5s 内没有 cancel，则认为发生了 ANR，从之前插入的结点开始，到最后一个结点，收集中间执行过的方法数据，可以认为导致 ANR 的关键方法就在这里面，计算时间戳即可得到关键方法。

另外，考虑到每个方法执行前后都获取系统时间（System.nanoTime）会对性能影响比较大，而实际上，单个函数执行耗时小于 5ms 的情况，对卡顿来说不是主要原因，可以忽略不计，如果是多次调用的情况，则在它的父级方法中可以反映出来，所以为了减少对性能的影响，Matrix 创建了一条专门用于更新时间的线程，每 5ms 去更新一个时间变量，而每个方法执行前后只读取该变量来减少性能损耗。

6.卡顿监控源码解析

监控主线程

TraceCanary 模块只能在 API 16 以上的设备运行，内部分为 ANR、帧率、慢方法、启动四个监测模块，核心接口是 LooperObserver。

LooperObserver 是一个抽象类，顾名思义，它是 Looper 的观察者，在 Looper 分发消息、刷新 UI 时回调，这几个回调方法也是 ANR、慢方法等模块的判断依据：

```
public abstract class LooperObserver  {

    // 分发消息前
    @CallSuper
    public void dispatchBegin(long beginMs, long cpuBeginMs, long token) {
```

```
    }

    // UI 刷新
    public void doFrame(String focusedActivityName, long start, long end, long
frameCostMs,
                        long inputCostNs, long animationCostNs, long
traversalCostNs) {
    }

    // 分发消息后
    @CallSuper
    public void dispatchEnd(long beginMs, long cpuBeginMs, long endMs, long
cpuEndMs, long token, boolean isBelongFrame) {
    }
}
```

Looper 监控

Looper 的监控是由类 LooperMonitor 实现的，原理很简单，为主线程 Looper 设置一个 Printer 即可，但值得一提的是，LooperMonitor 不会直接设置 Printer，而是先获取旧对象，并创建代理对象，避免影响到其它用户设置的 Printer：

```
private synchronized void resetPrinter() {
    Printer originPrinter = ReflectUtils.get(looper.getClass(), "mLogging",
looper);;
    looper.setMessageLogging(printer = new LooperPrinter(originPrinter));
}

class LooperPrinter implements Printer {
    @Override
    public void println(String x) {
        if (null != origin) {
            origin.println(x); // 保证原对象正常执行
        }
        dispatch(x.charAt(0) == '>', x); // 分发，通过第一个字符判断是开始分发，还是结
束分发
    }
}
```

UI 刷新监控

UI 刷新监控是基于 Choreographer 实现的，TracePlugin 初始化时，UIThreadMoniter 就会通过反射的方式往 Choreographer 添加回调：

```
public class UIThreadMonitor implements BeatLifecycle, Runnable {

    // Choreographer 中一个内部类的方法，用于添加回调
    private static final String ADD_CALLBACK = "addCallbackLocked";

    // 回调类型，分别为输入事件、动画、View 绘制三种
```

```
    public static final int CALLBACK_INPUT = 0;
    public static final int CALLBACK_ANIMATION = 1;
    public static final int CALLBACK_TRAVERSAL = 2;

    public void init(TraceConfig config) {
        choreographer = Choreographer.getInstance();
        // 回调队列
        callbackQueues = reflectObject(choreographer, "mCallbackQueues");
        // 反射，找到在 Choreographer 上添加回调的方法
        addInputQueue =
reflectChoreographerMethod(callbackQueues[CALLBACK_INPUT], ADD_CALLBACK,
long.class, Object.class, Object.class);
        addAnimationQueue =
reflectChoreographerMethod(callbackQueues[CALLBACK_ANIMATION], ADD_CALLBACK,
long.class, Object.class, Object.class);
        addTraversalQueue =
reflectChoreographerMethod(callbackQueues[CALLBACK_TRAVERSAL], ADD_CALLBACK,
long.class, Object.class, Object.class);
    }
}
```

之所以通过反射的方式实现，而不是通过 postCallback，是为了把我们的 callback 放到头部，这样才能计算系统提交的输入事件、动画、View 绘制等事件的耗时。

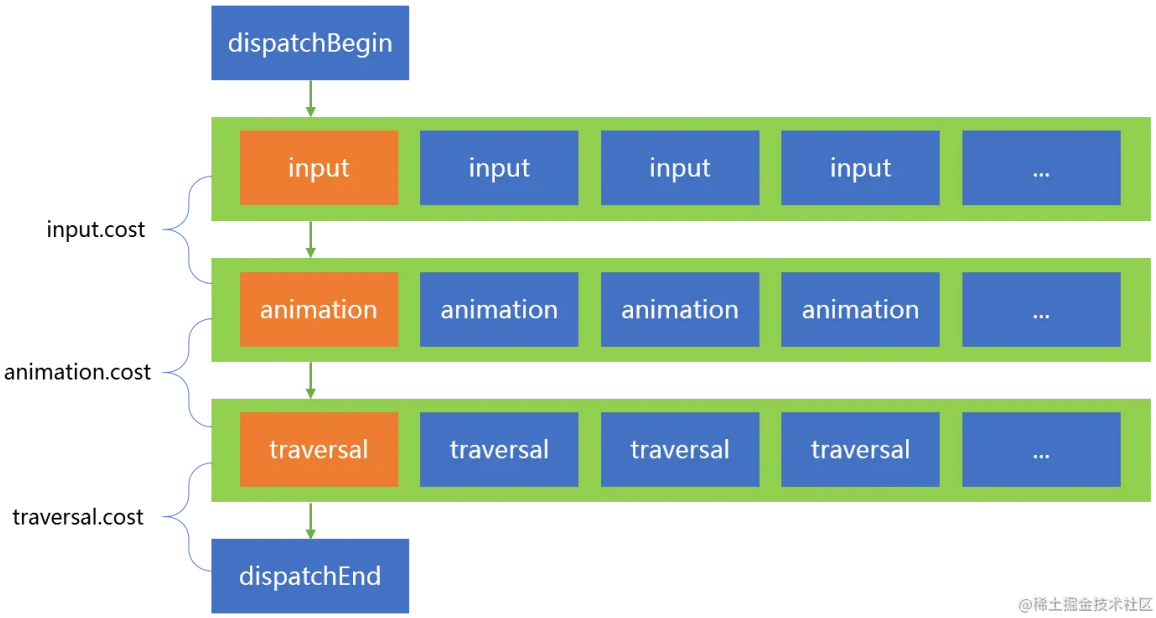
这样，等 Choreographer 监听到 vsync 信号时，UIThreadMonitor 和系统添加的回调都会被执行（比如在绘制 View 的时候，系统会往 Choreographer 添加一个 traversal callback）：

```
public final class Choreographer {

    private final class FrameDisplayEventReceiver extends DisplayEventReceiver
implements Runnable {
        @Override
        public void run() {
            doFrame(mTimestampNanos, mFrame);
        }
    }

    void doFrame(long frameTimeNanos, int frame) {
        doCallbacks(Choreographer.CALLBACK_INPUT, frameTimeNanos);
        doCallbacks(Choreographer.CALLBACK_ANIMATION, frameTimeNanos);
        doCallbacks(Choreographer.CALLBACK_TRAVERSAL, frameTimeNanos);
        ...
    }
}
```

因为 UIThreadMonitor 添加的回调在队列头部，可用于记录开始时间，而其它系统方法，比如 View 的 postOnAnimation 添加的回调在后面，因此所有同类型回调执行完毕后，就可以计算对应的事件（输入事件、动画、View 绘制等）的耗时。



ANR 监控

ANR 监控原理：在 Looper 分发消息时，往后台线程插入一个延时（5s 后执行）任务，Looper 消息分发完毕后就删除，如果过了 5s，该任务未被删除，就认为出现了 ANR。

```
public class AnrTracer extends Tracer {

    // onAlive 时初始化，onDead 时退出
    private Handler anrHandler;
    private volatile AnrHandleTask anrTask;

    public void dispatchBegin(long beginMs, long cpuBeginMs, long token) {
        // 插入方法结点，如果出现了 ANR，就从该结点开始收集方法执行记录
        anrTask = new
AnrHandleTask(AppMethodBeat.getInstance().maskIndex("AnrTracer#dispatchBegin"),
token);
        // 5 秒后执行
        // token 和 beginMs 相等，因此后一个减式用于减去回调该方法过程中所消耗的时间
        anrHandler.postDelayed(anrTask, Constants.DEFAULT_ANR -
(SystemClock.uptimeMillis() - token));
    }

    @Override
    public void dispatchEnd(long beginMs, long cpuBeginMs, long endMs, long
cpuEndMs, long token, boolean isBelongFrame) {
        if (null != anrTask) {
            anrTask.getBeginRecord().release();
            anrHandler.removeCallbacks(anrTask);
        }
    }
}
```

如果 5s 后该任务未被删除，那么 AnrTracer 就会开始收集进程、线程、内存、堆栈等信息，并上报。

启动监控

应用的启动监控以第一个执行的方法为起点：

```
public class AppMethodBeat implements BeatLifecycle {

    private static volatile int status = STATUS_DEFAULT;

    // 该方法会被插入到每一个方法的开头执行
    public static void i(int methodId) {

        if (status == STATUS_DEFAULT) { // 如果是默认状态，则说明是第一个方法
            realExecute();
            status = STATUS_READY;
        }
    }

    private static void realExecute() {
        // 记录时间戳
        ActivityThreadHacker.hackSysHandlerCallback();
        // 开始监控主线程 Looper
        LooperMonitor.register(looperMonitorListener);
    }
}
```

记录了第一个方法开始执行时的时间戳后，Matrix 还会通过反射的方式，接管 ActivityThread 的 Handler 的 Callback：

```
public class ActivityThreadHacker {

    public static void hackSysHandlerCallback() {
        // 记录时间戳，作为应用启用的开始时间
        sApplicationCreateBeginTime = SystemClock.uptimeMillis();
        // 反射 ActivityThread，接管 Handler
        Class<?> forName = Class.forName("android.app.ActivityThread");
        ...
    }
}
```

这样就能知道第一个 Activity 或 Service 或 Receiver 启动的具体时间了，这个时间戳可以作为 Application 启动的结束时间：

```
private final static class HackCallback implements Handler.Callback {
    private static final int LAUNCH_ACTIVITY = 100;
    private static final int CREATE_SERVICE = 114;
    private static final int RECEIVER = 113;
    private static boolean isCreated = false;

    @Override
    public boolean handleMessage(Message msg) {
        boolean isLaunchActivity = isLaunchActivity(msg);
        // 如果是第一个启动的 Activity 或 Service 或 Receiver，则以该时间戳作为
Application 启动的结束时间
```

```
        if (!isCreated) {
            if (isLaunchActivity || msg.what == CREATE_SERVICE || msg.what ==
RECEIVER) { // todo for provider
                ActivityThreadHacker.sApplicationCreateEndTime =
SystemClock.uptimeMillis();
                ActivityThreadHacker.sApplicationCreateScene = msg.what;
                isCreated = true;
            }
        }
    }
}
```

最后以主 Activity（闪屏页之后的第一个 Activity）的 onWindowFocusChange 方法作为终点，记录时间戳——Activity 的启动耗时可以通过 onWindowFocusChange 方法回调时的时间戳减去其启动时的时间戳。收集到上述信息之后即可统计启动耗时：

```
firstMethod.i       LAUNCH_ACTIVITY   onWindowFocusChange   LAUNCH_ACTIVITY
 onWindowFocusChange
^                        ^                  ^                    ^
       ^
|                        |                  |                    |
       |
|---------app---------|---|---firstActivity---|---------...---------|---
careActivity---|
|<--applicationCost-->|
|<--------------firstScreenCost-------------->|
|<-----------------------------------------coldCost-----------------------------
------>|
.                        |<-----warmCost---->|
```

如果冷启动/暖启动耗时超过某个阈值（可通过 IDynamicConfig 设置，默认分别为 10s、4s），那么就会从 AppMethodBeat 收集启动过程中的方法执行记录并上报，否则只会简单地上报耗时信息。

慢方法监控

慢方法监测的原理是在 Looper 分发消息时，计算分发耗时（endMs - beginMs），如果大于阈值（可通过 IDynamicConfig 设置，默认为 700ms），就收集信息并上报。

```
ublic class EvilMethodTracer extends Tracer {

    @Override
    public void dispatchBegin(long beginMs, long cpuBeginMs, long token) {
        super.dispatchBegin(beginMs, cpuBeginMs, token);
        // 插入方法结点，如果出现了方法执行过慢的问题，就从该结点开始收集方法执行记录
        indexRecord =
AppMethodBeat.getInstance().maskIndex("EvilMethodTracer#dispatchBegin");
    }

    @Override
    public void dispatchEnd(long beginMs, long cpuBeginMs, long endMs, long
cpuEndMs, long token, boolean isBelongFrame) {
        long dispatchCost = endMs - beginMs;
```

```
        // 耗时大于慢方法阈值
        if (dispatchCost >= evilThresholdMs) {
            long[] data = AppMethodBeat.getInstance().copyData(indexRecord);
            MatrixHandlerThread.getDefaultHandler().post(new AnalyseTask(...);
        }
    }

    private class AnalyseTask implements Runnable {
        void analyse() {
            // 收集进程与 CPU 信息
            int[] processStat = Utils.getProcessPriority(Process.myPid());
            String usage = Utils.calculateCpuUsage(cpuCost, cost);
            // 从插入结点开始收集并整理方法执行记录
            TraceDataUtils.structuredDataToStack(data, stack, true, endMs);
            TraceDataUtils.trimStack(stack, Constants.TARGET_EVIL_METHOD_STACK,
                    new TraceDataUtils.IStructuredDataFilter() { ... }
            // 上报问题
            TracePlugin plugin =
Matrix.with().getPluginByClass(TracePlugin.class);
            plugin.onDetectIssue(issue);
        }
    }
}
```

帧率监控

帧率监测的原理是监听 Choreographer，在所有回调都执行完毕后计算当前总共花费的时间，从而计算掉帧数及掉帧程度，当同一个 Activity/Fragment 掉帧程度超过阈值时，就上报问题。关键源码如下：

```
private class FPSCollector extends IDoFrameListener {
    @Override
    public void doFrameAsync(String visibleScene, long taskCost, long
frameCostMs,
                             int droppedFrames, boolean isContainsFrame) {
        // 使用 Map 保存同一 Activity/Fragment 的掉帧信息
        FrameCollectItem item = map.get(visibleScene);
        if (null == item) {
            item = new FrameCollectItem(visibleScene);
            map.put(visibleScene, item);
        }

        // 累计
        item.collect(droppedFrames, isContainsFrame);

        // 如果掉帧程度超过一定阈值，就上报问题，并重新计算
        // 总掉帧时间 sumFrameCost = 掉帧数 * 16.66ms
        // 掉帧上报阈值 timeSliceMs 可通过 IDynamicConfig 设置，默认为 10s
        if (item.sumFrameCost >= timeSliceMs) { // report
            map.remove(visibleScene);
            item.report();
        }
    }
}
```

但这里存在一个问题，那就是 Matrix 计算 UI 刷新耗时时，每次都会在掉帧数的基础上加 1：

```
private class FrameCollectItem {

    void collect(int droppedFrames, boolean isContainsFrame) {
        // 即使掉帧数为 0，这个值也会不断增加
        sumFrameCost += (droppedFrames + 1) * frameIntervalCost /
Constants.TIME_MILLIS_TO_NANO;
    }
}
```

而且，doFrame 方法不是只在 UI 刷新时回调，而是每次 Looper 分发消息完毕后都会回调，而 Lopper 分发消息的频率可能远远大于帧率，这就导致即使实际上没有出现掉帧的情况，但由于 Looper 不断分发消息的缘故，sumFrameCost 的值也会不断累加，很快就突破了上报的阈值，进而频繁地上报：

```
private void dispatchEnd() {
    ...
    synchronized (observers) {
        for (LooperObserver observer : observers) {
            if (observer.isDispatchBegin()) {
                observer.doFrame(...);
            }
        }
    }
}
```

解决方法是在 PluginListener 中手动过滤，或者修改源码。

总结

TraceCanary 分为慢方法、启动、ANR、帧率四个模块，每个模块的功能都是通过监听接口 LooperObserver 实现的，LooperObserver 用于对主线程的 Looper 和 Choreographer 进行监控。

Looper 的监控是通过 Printer 实现的，每次事件分发都会回调 LooperObserver 的 dispatchBegin、dispatchEnd 方法，计算这两个方法的耗时可以检测慢方法和 ANR 等问题。

Choreographer 的监控则是通过添加 input、animation、traversal 等各个类型的回调到 Choreographer 头部实现的，vsync 信号触发后，Choreographer 中各个类型的回调会被执行，两种类型的回调的开始时间的间隔就相当于第一种类型的事件的耗时（即 input.cost = animation.begin - input.begiin），最后一种事件（traversal）执行完毕后，Looper 的 diaptchEnd 方法也会被执行，因此 traversal.cost = Looper.dispatchEnd -traversal.begin。

各个模块的实现原理如下：

1. ANR：在 Looper 开始分发消息时，往后台线程插入一个延时（5s 后执行）任务，Looper 消息分发完毕后就删除，如果过了 5s，该任务未被删除，就认为出现了 ANR，收集信息，报告问题
2. 慢方法：在 Looper 分发消息时，计算分发耗时（endMs - beginMs），如果大于阈值（可通过 IDynamicConfig 设置，默认为 700ms），就收集信息并上报

3. 启动：以第一个执行的方法为起点记录时间戳，接着记录第一个 Activity 或 Service 或 Receiver 启动时的时间戳，作为 Application 启动的结束时间。最后以主 Activity（闪屏页之后的第一个 Activity）的 onWindowFocusChange 方法作为终点，记录时间戳。Activity 的启动耗时可以通过 onWindowFocusChange 方法回调时的时间戳减去其启动时的时间戳。收集到上述信息之后即可统计启动耗时。
4. 掉帧：监听 Choreographer，doFrame 回调时统计 UI 刷新耗时，计算掉帧数及掉帧程度，当同一个 Activity/Fragment 掉帧程度超过阈值时，就上报。但 Matrix 的计算方法存在问题，可能出现频繁上报的情况，需要自行手动过滤。

7.插桩

之前说到，Matrix 的卡顿监控关键在于插桩，下面来看一下它是怎么实现的。

Gradle 插件配置

Matrix 的 Gradle 插件的实现类为 MatrixPlugin，主要做了三件事：

1. 添加 Extension，用于提供给用户自定义配置选项

```
class MatrixPlugin implements Plugin<Project> {

    @Override
    void apply(Project project) {
        project.extensions.create("matrix", MatrixExtension)
        project.matrix.extensions.create("trace", MatrixTraceExtension)
        project.matrix.extensions.create("removeUnusedResources",
MatrixDelUnusedResConfiguration)
    }
}
```

其中 trace 可选配置如下：

```
public class MatrixTraceExtension {
    boolean enable; // 是否启用插桩功能
    String baseMethodMapFile;  // 自定义的方法映射文件，下面会说到
    String blackListFile; // 该文件指定的方法不会被插桩
    String customDexTransformName;
}
```

removeUnusedResources 可选配置如下：

```
class MatrixDelUnusedResConfiguration {
    boolean enable // 是否启用
    String variant // 指定某一个构建变体启用插桩功能，如果为空，则所有的构建变体都启用
    boolean needSign // 是否需要签名
    boolean shrinkArsc // 是否裁剪 arsc 文件
    String apksignerPath // 签名文件的路径
    Set<String> unusedResources // 指定要删除的不使用的资源
    Set<String> ignoreResources // 指定不需要删除的资源
}
```

1. 读取配置，如果启用插桩，则执行 MatrixTraceTransform，统计方法并插桩

```
// 在编译期执行插桩任务（project.afterEvaluate 代表 build.gradle 文件执行完毕），这是因为 proguard 操作是在该任务之前就完成的
project.afterEvaluate {
    android.applicationVariants.all { variant ->

        if (configuration.trace.enable) { // 是否启用，可在 gradle 文件中配置
            MatrixTraceTransform.inject(project, configuration.trace, variant.getVariantData().getScope())
        }

        ... // RemoveUnusedResourcesTask
    }
}
```

1. 读取配置，如果启用 removeUnusedResources 功能，则执行 RemoveUnusedResourcesTask，删除不需要的资源

方法统计及插桩

配置 Transform

MatrixTraceTransform 的 inject 方法主要用于读取配置，代理 transformClassesWithDexTask：

```
public class MatrixTraceTransform extends Transform {

    public static void inject(Project project, MatrixTraceExtension extension, VariantScope variantScope) {
        ... // 根据参数生成 Configuration 变量 config

        String[] hardTask = getTransformTaskName(extension.getCustomDexTransformName(), variant.getName());
        for (Task task : project.getTasks())
            for (String str : hardTask)
                if (task.getName().equalsIgnoreCase(str) && task instanceof TransformTask) {
                    Field field = TransformTask.class.getDeclaredField("transform");
                    field.set(task, new MatrixTraceTransform(config, task.getTransform()));
                    break;
                }
    }

    // 这两个 Transform 用于把 Class 文件编译成 Dex 文件
    // 因此，需要在这两个 Transform 执行之前完成插桩等工作
    private static String[] getTransformTaskName(String customDexTransformName, String buildTypeSuffix) {
        return new String[] {
                "transformClassesWithDexBuilderFor" + buildTypeSuffix,
                "transformClassesWithDexFor" + buildTypeSuffix,
            };;
    }
```

```
}
```

MatrixTraceTransform 的主要配置如下：

1. 处理范围为整个项目（包括当前项目、子项目、依赖库等）
2. 处理类型为 Class 文件

```
public class MatrixTraceTransform extends Transform {
    @Override
    public Set<QualifiedContent.ContentType> getInputTypes() { return
TransformManager.CONTENT_CLASS; }

    @Override
    public Set<QualifiedContent.Scope> getScopes() { return
TransformManager.SCOPE_FULL_PROJECT; }
}
```

执行方法统计及插桩任务

transform 主要分三步执行：

1. 根据配置文件分析方法统计规则，比如混淆后的类名和原始类名之间的映射关系、不需要插桩的方法黑名单等

```
private void doTransform(TransformInvocation transformInvocation) throws
ExecutionException, InterruptedException {
    // 用于分析和方法统计相关的文件，如 mapping.txt、blackMethodList.txt 等
    // 并将映射规则保存到 mappingCollector、collectedMethodMap 中
    futures.add(executor.submit(new ParseMappingTask(mappingCollector,
collectedMethodMap, methodId)));
}
```

1. 统计方法及其 ID，并写入到文件中

```
private void doTransform(TransformInvocation transformInvocation) {
    MethodCollector methodCollector = new MethodCollector(executor,
mappingCollector, methodId, config, collectedMethodMap);
    methodCollector.collect(dirInputOutMap.keySet(), jarInputOutMap.keySet());
}
```

1. 插桩

```
private void doTransform(TransformInvocation transformInvocation) {
    MethodTracer methodTracer = new MethodTracer(executor, mappingCollector,
config,
            methodCollector.getCollectedMethodMap(),
methodCollector.getCollectedClassExtendMap());
    methodTracer.trace(dirInputOutMap, jarInputOutMap);
}
```

分析方法统计规则

ParseMappingTask 主要用于分析方法统计相关的文件，如 mapping.txt（ProGuard 生成的）、blackMethodList.txt 等，并将映射规则保存到 HashMap 中。

mapping.txt 是 ProGuard 生成的，用于映射混淆前后的类名/方法名，内容如下：

```
MTT.ThirdAppInfoNew -> MTT.ThirdAppInfoNew: // oldClassName -> newClassName
    java.lang.String sAppName -> sAppName // oldMethodName -> newMethodName
    java.lang.String sTime -> sTime
    ...
```

blackMethodList.txt 则用于避免对特定的方法插桩，内容如下：

```
[package]
-keeppackage com/huluxia/logger/
-keepmethod com/example/Application attachBaseContext
(Landroid/content/Context;)V
...
```

如果有需要，还可以指定 baseMethodMapFile，将自定义的方法及其对应的方法 id 写入到一个文件中，内容格式如下：

```
// 方法 id、访问标志、类名、方法名、描述
1,1,eu.chainfire.libsuperuser.Application$1 run ()V
2,9,eu.chainfire.libsuperuser.Application toast
(Landroid.content.Context;Ljava.lang.String;)V
```

上述选项可在 gradle 文件配置，示例如下：

```
matrix {
    trace {
        enable = true
        baseMethodMapFile = "{projectDir.absolutePath}/baseMethodMapFile.txt"
        blackListFile = "{projectDir.absolutePath}/blackMethodList.txt"
    }
}
```

方法统计

顾名思义，MethodCollector 用于收集方法，它首先会把方法封装为 TraceMethod，并分配方法 id，再保存到 HashMap，最后写入到文件中。

为此，首先需要获取所有 class 文件：

```
public void collect(Set<File> srcFolderList, Set<File> dependencyJarList) throws
ExecutionException, InterruptedException {
    for (File srcFile : srcFolderList) {
        ...
        for (File classFile : classFileList) {
            futures.add(executor.submit(new CollectSrcTask(classFile)));
        }
    }

    for (File jarFile : dependencyJarList) {
        futures.add(executor.submit(new CollectJarTask(jarFile)));
    }
}
```

接着，借助 ASM 访问每一个 Class 文件：

```
class CollectSrcTask implements Runnable {
    @Override
    public void run() {
        InputStream is = new FileInputStream(classFile);
        ClassReader classReader = new ClassReader(is);
        ClassWriter classWriter = new ClassWriter(ClassWriter.COMPUTE_MAXS);
        ClassVisitor visitor = new TraceClassAdapter(Opcodes.ASM5, classWriter);
        classReader.accept(visitor, 0);
    }
}
```

及 Class 文件中的方法：

```
private class TraceClassAdapter extends ClassVisitor {

    @Override
    public MethodVisitor visitMethod(int access, String name, String desc,
                                     String signature, String[] exceptions) {
        if (isABSClass) { // 抽象类或接口不需要统计
            return super.visitMethod(access, name, desc, signature, exceptions);
        } else {
            return new CollectMethodNode(className, access, name, desc,
signature, exceptions);
        }
    }
}
```

最后，记录方法数据，并保存到 HashMap 中：

```
private class CollectMethodNode extends MethodNode {
    @Override
    public void visitEnd() {
        super.visitEnd();
        // 将方法数据封装为 TraceMethod
        TraceMethod traceMethod = TraceMethod.create(0, access, className, name,
desc);

        // 是否需要插桩，blackMethodList.txt 中指定的方法不会被插桩
        boolean isNeedTrace = isNeedTrace(configuration, traceMethod.className,
mappingCollector);
        // 过滤空方法、get & set 方法等简单方法
        if ((isEmptyMethod() || isGetSetMethod() || isSingleMethod()) &&
isNeedTrace) {
            return;
        }

        // 保存到 HashMap 中
        if (isNeedTrace &&
!collectedMethodMap.containsKey(traceMethod.getMethodName())) {
            traceMethod.id = methodId.incrementAndGet();
            collectedMethodMap.put(traceMethod.getMethodName(), traceMethod);
            incrementCount.incrementAndGet();
        } else if (!isNeedTrace &&
!collectedIgnoreMethodMap.containsKey(traceMethod.className)) {
            ... // 记录不需要插桩的方法
        }
    }
}
```

统计完毕后，将上述方法及其 ID 写入到一个文件中——因为之后上报问题只会上报 method id，因此需要根据该文件来解析具体的方法名及其耗时。

虽然上面的代码很长，但作用实际很简单：访问所有 Class 文件中的方法，记录方法 ID，并写入到文件中。

需要注意的细节有：

1. 统计的方法包括应用自身的、JAR 依赖包中的，以及额外添加的 ID 固定的 dispatchMessage 方法
2. 抽象类或接口类不需要统计
3. 空方法、get & set 方法等简单方法不需要统计
4. blackMethodList.txt 中指定的方法不需要统计

插桩

和方法统计一样，插桩也是基于 ASM 实现的，首先同样要找到所有 Class 文件，再针对文件中的每一个方法进行处理。

处理流程主要包含四步：

1. 进入方法时执行 AppMethodBeat.i，传入方法 ID，记录时间戳

```
public final static String MATRIX_TRACE_CLASS =
"com/tencent/matrix/trace/core/AppMethodBeat";
```

```
private class TraceMethodAdapter extends AdviceAdapter {

    @Override
    protected void onMethodEnter() {
        TraceMethod traceMethod = collectedMethodMap.get(methodName);
        if (traceMethod != null) { // 省略空方法、set & get 等简单方法
            mv.visitLdcInsn(traceMethod.id);
            mv.visitMethodInsn(INVOKESTATIC,
TraceBuildConstants.MATRIX_TRACE_CLASS, "i", "(I)V", false);
        }
    }
}
```

1. 退出方法时执行 AppMethodBeat.o，传入方法 ID，记录时间戳

```
private class TraceMethodAdapter extends AdviceAdapter {

    @Override
    protected void onMethodExit(int opcode) {
        TraceMethod traceMethod = collectedMethodMap.get(methodName);
        if (traceMethod != null) {
            ... // 跟踪 onWindowFocusChanged 方法，计算启动耗时
            mv.visitLdcInsn(traceMethod.id);
            mv.visitMethodInsn(INVOKESTATIC,
TraceBuildConstants.MATRIX_TRACE_CLASS, "o", "(I)V", false);
        }
    }
}
```

1. 如果是 Activity，并且没有 onWindowFocusChanged 方法，则插入该方法

```
private class TraceClassAdapter extends ClassVisitor {

    @Override
    public void visitEnd() {
        // 如果是 Activity，并且不存在 onWindowFocusChanged 方法，则插入该方法，用于统计
Activity 启动时间
        if (!hasWindowFocusMethod && isActivityOrSubClass && isNeedTrace) {
            insertWindowFocusChangeMethod(cv, className);
        }
        super.visitEnd();
    }
}
```

1. 跟踪 onWindowFocusChanged 方法，退出时执行 AppMethodBeat.at，计算启动耗时

```
public final static String MATRIX_TRACE_CLASS =
"com/tencent/matrix/trace/core/AppMethodBeat";

private void traceWindowFocusChangeMethod(MethodVisitor mv, String classname) {
    mv.visitMethodInsn(Opcodes.INVOKESTATIC,
TraceBuildConstants.MATRIX_TRACE_CLASS, "at", "(Landroid/app/Activity;Z)V",
false);
```

```
}

public class AppMethodBeat implements BeatLifecycle {

    public static void at(Activity activity, boolean isFocus) {
        for (IAppMethodBeatListener listener : listeners) {
            listener.onActivityFocused(activityName);
        }
    }
}
```

StartupTracer 就是 IAppMethodBeatListener 的实现类。

总结

Matrix 的 Gradle 插件的实现类为 MatrixPlugin，主要做了三件事：

1. 添加 Extension，用于提供给用户自定义配置选项
2. 读取 extension 配置，如果启用 trace 功能，则执行 MatrixTraceTransform，统计方法并插桩
3. 读取 extension 配置，如果启用 removeUnusedResources 功能，则执行 RemoveUnusedResourcesTask，删除不需要的资源

需要注意的是，插桩任务是在编译期执行的，这是为了避免对混淆操作产生影响。因为 proguard 操作是在该任务之前就完成的，意味着插桩时的 class 文件已经被混淆过的。而选择 proguard 之后去插桩，是因为如果提前插桩会造成部分方法不符合内联规则，没法在 proguard 时进行优化，最终导致程序方法数无法减少，从而引发方法数过大问题

transform 主要分三步执行：

1. 根据配置文件（mapping.txt、blackMethodList.txt、baseMethodMapFile）分析方法统计规则，比如混淆后的类名和原始类名之间的映射关系、不需要插桩的方法黑名单等
2. 借助 ASM 访问所有 Class 文件的方法，记录其 ID，并写入到文件中（methodMapping.txt）
3. 插桩

插桩处理流程主要包含四步：

1. 进入方法时执行 AppMethodBeat.i，传入方法 ID，记录时间戳
2. 退出方法时执行 AppMethodBeat.o，传入方法 ID，记录时间戳
3. 如果是 Activity，并且没有 onWindowFocusChanged 方法，则插入该方法
4. 跟踪 onWindowFocusChanged 方法，退出时执行 AppMethodBeat.at，计算启动耗时

值得注意的细节有：

1. 统计的方法包括应用自身的、JAR 依赖包中的，以及额外添加的 ID 固定的 dispatchMessage 方法
2. 抽象类或接口类不需要统计
3. 空方法、get & set 方法等简单方法不需要统计
4. blackMethodList.txt 中指定的方法不需要统计

8.资源优化

除了插桩之外，Matrix 还会根据用户配置选择是否执行资源优化的功能，以删除不必要的资源文件。

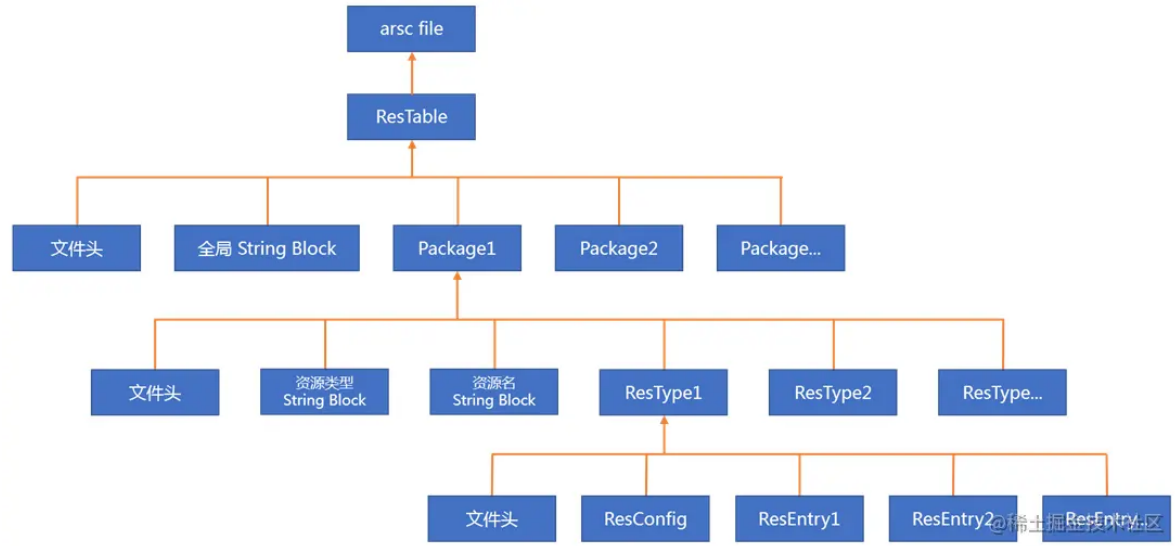
arsc 文件格式

Matrix 资源优化的其中一个功能是裁剪 resources.arsc，分析该功能之前，先简单了解一下 arsc 的文件格式。

首先介绍一下 arsc 文件中的几个概念：

1. Chunk，指一个数据块，下面介绍的 Table、Package、String Block、Type 的都是 Chunk，都有文件头、类型、size、对齐填充等信息
2. Resource Table，一个 arsc 文件对应一个 Resource Table
3. Package，用于描述一个包，一个 Table 对应多个 Package，而 packageID 即是资源 resID 的最高八位，一般来说系统 android 的是 1(0x01)，普通的例如 com.tencent.mm 会是 127(0x7f)，剩下的是从 2 开始起步，也可以在 aapt 中指定。
4. String Block，一个 Table 有一个全局的字符串资源池，一个 Package 有一个存储资源类型的字符串资源池，一个储存资源名的字符串资源池
5. Resource Type，资源类型，比如 attr、drawable、layout、id、color、anim 等，一个 Package 对应多个 Type
6. Config，用于描述资源的维度，例如横竖屏，屏幕密度，语言等，一个 Type 对应一个 Config
7. Entry，一个 Type 对应多个 Entry，例如 drawable-mdpi 中有 icon1.png、icon2.png 两个 drawable，那在 mdpi 这个 Type 中就存在两个 entry

文件结构如下图所示：



删除未使用的资源

下面开始分析 Matrix 是怎么执行资源优化的。

Matrix 首先会获取 apk 文件、R 文件、签名配置文件等文件信息：

```
String unsignedApkPath = output.outputFile.getAbsolutePath();
removeUnusedResources(unsignedApkPath,
        project.getBuildDir().getAbsolutePath() +
"/intermediates/symbols/${variant.name}/R.txt",
        variant.variantData.variantConfiguration.signingConfig);
```

接着，确定需要删除的资源信息，包括资源名及其 ID：

```
// 根据配置获取需要删除的资源
Set<String> ignoreRes =
project.extensions.matrix.removeUnusedResources.ignoreResources;
```

```
Set<String> unusedResources =
project.extensions.matrix.removeUnusedResources.unusedResources;
Iterator<String> iterator = unusedResources.iterator();
String res = null;
while (iterator.hasNext()) {
    res = iterator.next();
    if (ignoreResource(res)) { // 指定忽略的资源不需要删除
        iterator.remove();
    }
}

// 读取 R 文件，保存资源名及对应的 ID 到 resourceMap 中
Map<String, Integer> resourceMap = new HashMap();
Map<String, Pair<String, Integer>[]> styleableMap = new HashMap();
File resTxtFile = new File(rTxtFile);
readResourceTxtFile(resTxtFile, resourceMap, styleableMap); // 读取 R 文件

// 将 unusedResources 中的资源放到 removeResources 中
Map<String, Integer> removeResources = new HashMap<>();
for (String resName : unusedResources) {
    if (!ignoreResource(resName)) {
        // 如果资源会被删除，那么将它从 resourceMap 中移除
        removeResources.put(resName, resourceMap.remove(resName));
    }
}
```

之后就可以删除指定的资源了，删除方法是创建一个新的 apk 文件，并且忽略不需要的资源：

```
for (ZipEntry zipEntry : zipInputFile.entries()) {
    if (zipEntry.name.startsWith("res/")) {
        String resourceName = entryToResouceName(zipEntry.name);
        if (removeResources.containsKey(resourceName)) { // 需要删除的资源不会写入到
新文件中
            continue;
        } else { // 将正常的资源信息写入到新的 apk 文件中
            addZipEntry(zipOutputStream, zipEntry, zipInputFile);
        }
    }
}
```

如果启用了 shrinkArsc 功能，那么，还需要修改 arsc 文件，移除掉已删除的资源信息：

```
if (shrinkArsc && zipEntry.name.equalsIgnoreCase("resources.arsc") &&
unusedResources.size() > 0) {
    File srcArscFile = new File(inputFile.getParentFile().getAbsolutePath() +
"/resources.arsc");
    File destArscFile = new File(inputFile.getParentFile().getAbsolutePath() +
"/resources_shrinked.arsc");

    // 从 arsc 文件中读取资源信息
    ArscReader reader = new ArscReader(srcArscFile.getAbsolutePath());
    ResTable resTable = reader.readResourceTable();

    // 遍历需要删除的资源列表，将对应的资源信息从 arsc 文件中移除
```

```
    for (String resName : removeResources.keySet()) {
        ArscUtil.removeResource(resTable, removeResources.get(resName),
resName);
    }

    // 将裁剪后的 ResTable 写入到新的 arsc 文件中
    ArscWriter writer = new ArscWriter(destArscFile.getAbsolutePath());
    writer.writeResTable(resTable);

    // 将裁剪后的 arsc 文件写入到新的 apk 中
    addZipEntry(zipOutputStream, zipEntry, destArscFile);
}
```

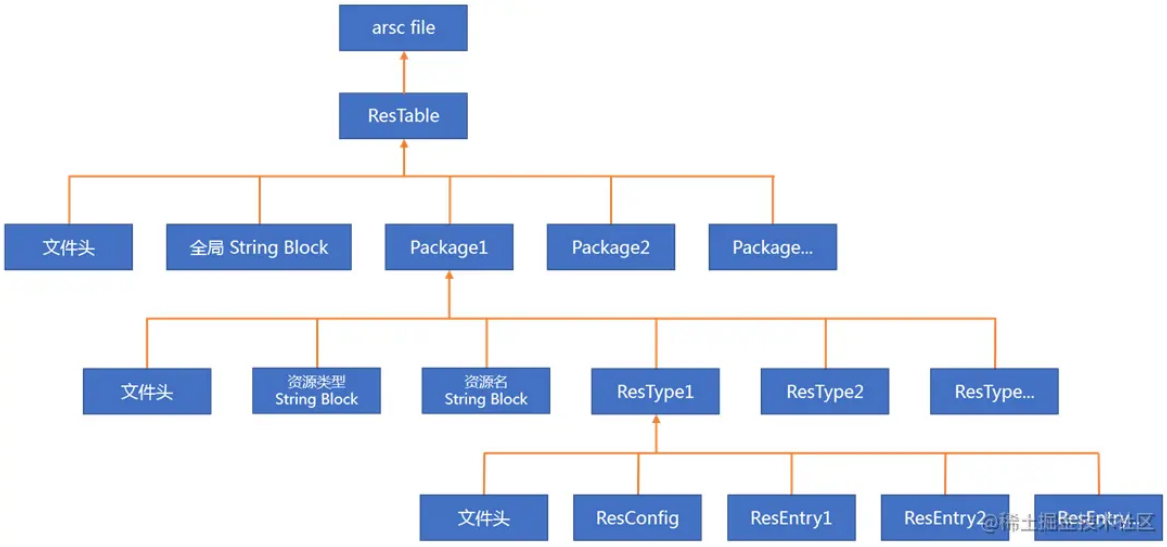
移除的方法是将其 Entry 置为 null：

```
public static void removeResource(ResTable resTable, int resourceId, String
resourceName) throws IOException {
    ResPackage resPackage = findResPackage(resTable, getPackageId(resourceId));
// 找到该 resId 对应的 package
    if (resPackage != null) {
        List<ResType> resTypeList = findResType(resPackage, resourceId);
        for (ResType resType : resTypeList) { // 遍历 package 中的 ResType，找到对
应类型
            int entryId = getResourceEntryId(resourceId); // 再找到对应的 entry
            resType.getEntryTable().set(entryId, null); // 设置为 null
            resType.getEntryOffsets().set(entryId,
ArscConstants.NO_ENTRY_INDEX);
            resType.refresh();
        }
        resPackage.refresh();
        resTable.refresh();
    }
}
```

以上，移除不必要的资源后的新的 apk 文件就写入完毕了。

总结

arsc 文件结构：



RemoveUnusedResourcesTask 执行步骤如下：

1. 获取 apk 文件、R 文件、签名配置文件等文件信息
2. 根据用户提供的 unusedResource 文件及 R 文件确定需要删除的资源信息，包括资源名及其 ID
3. 删除指定的资源，删除方法是在写入新的 apk 文件时，忽略该资源
4. 如果启用了 shrinkArsc 功能，那么，修改 arsc 文件，移除掉已删除的资源信息，移除方法是将其 Entry 置为 null
5. 其它数据原封不动地写入到新的 apk 文件中

9.I/O 监控及原理解析

使用

Matrix 中用于 I/O 监控的模块是 IOCanary，它是一个在开发、测试或者灰度阶段辅助发现 I/O 问题的工具，目前主要包括文件 I/O 监控和 Closeable Leak 监控两部分。

具体的问题类型有 4 种：

1. 在主线程执行了 IO 操作
2. 缓冲区太小
3. 重复读同一文件
4. 资源泄漏

IOCanary 采用 hook(ELF hook) 的方案收集 IO 信息，代码无侵入，从而使得开发者可以无感知接入。配置并启动 IOCanaryPlugin 即可：

```
IOCanaryPlugin ioCanaryPlugin = new IOCanaryPlugin(new IOConfig.Builder()
        .dynamicConfig(dynamicConfig)
        .build());
builder.plugin(ioCanaryPlugin);
```

与 IO 相关的配置选项有：

```
enum ExptEnum {
    // 监测在主线程执行 IO 操作的问题
```

```
    clicfg_matrix_io_file_io_main_thread_enable,
    clicfg_matrix_io_main_thread_enable_threshold,  // 读写耗时
    // 监测缓冲区过小的问题
    clicfg_matrix_io_small_buffer_enable,
    clicfg_matrix_io_small_buffer_threshold, // 最小 buffer size
    clicfg_matrix_io_small_buffer_operator_times, // 读写次数
    // 监测重复读同一文件的问题
    clicfg_matrix_io_repeated_read_enable,
    clicfg_matrix_io_repeated_read_threshold, // 重复读次数
    // 监测内存泄漏问题
    clicfg_matrix_io_closeable_leak_enable,
}
```

出现资源泄漏（比如未关闭读写流）时，报告信息示例如下：

```
{
    "tag": "io",
    "type": 4,
    "process": "sample.tencent.matrix",
    "time": 1590410170122,
    "stack":
"sample.tencent.matrix.io.TestIOActivity.leakSth(TestIOActivity.java:190)\nsampl
e.tencent.matrix.io.TestIOActivity.onClick(TestIOActivity.java:103)\njava.lang.r
eflect.Method.invoke(Native
Method)\nandroid.view.View$DeclaredOnClickListener.onClick(View.java:4461)\nandr
oid.view.View.performClick(View.java:5212)\nandroid.view.View$PerformClick.run(V
iew.java:21214)\nandroid.app.ActivityThread.main(ActivityThread.java:5619)\njava
.lang.reflect.Method.invoke(Native
Method)\ncom.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.j
ava:853)\ncom.android.internal.os.ZygoteInit.main(ZygoteInit.java:737)\n",
}
```

写入太多、缓冲区太小的报告示例如下：

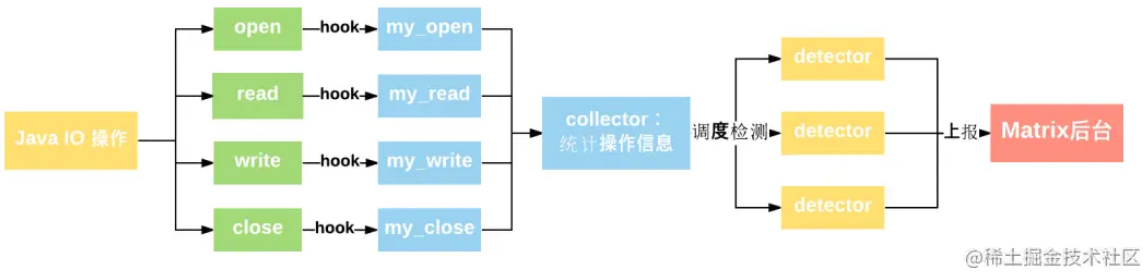
```
{
    "tag": "io",
    "type": 2, // 问题类型
    "process": "sample.tencent.matrix",
    "time": 1590409786187,
    "path": "/sdcard/a_long.txt", // 文件路径
    "size": 40960000, // 文件大小
    "op": 80000, // 读写次数
    "buffer": 512, // 缓冲区大小
    "cost": 1453, // 耗时
    "opType": 2, // 1 读 2 写
    "opSize": 40960000, // 读写总内存
    "thread": "main",
```

```
    "stack":
"sample.tencent.matrix.io.TestIOActivity.writeLongSth(TestIOActivity.java:129)\n
sample.tencent.matrix.io.TestIOActivity.onClick(TestIOActivity.java:99)\njava.la
ng.reflect.Method.invoke(Native
Method)\nandroid.view.View$DeclaredOnClickListener.onClick(View.java:4461)\nandr
oid.view.View.performClick(View.java:5212)\nandroid.view.View$PerformClick.run(V
iew.java:21214)\nandroid.app.ActivityThread.main(ActivityThread.java:5619)\njava
.lang.reflect.Method.invoke(Native
Method)\ncom.android.internal.os.ZygoteInit$MethodAndArgsCaller.run(ZygoteInit.j
ava:853)\ncom.android.internal.os.ZygoteInit.main(ZygoteInit.java:737)\n",
    "repeat": 0  // 重复读次数
}
```

需要注意的是，字段 repeat 在主线程 IO 事件中有不同的含义："1" 表示单次读写耗时过长；"2" 表示连续读写耗时过长（大于配置指定值）；"3" 表示前面两个问题都存在。

原理介绍

IOCanary 将收集应用的所有文件 I/O 信息并进行相关统计，再依据一定的算法规则进行检测，发现问题后再上报到 Matrix 后台进行分析展示。流程图如下：



IOCanary 基于 xHook 收集 IO 信息，主要 hook 了 os posix 的四个关键的文件操作接口：

```
int open(const char *pathname, int flags, mode_t mode); // 成功时返回值就是 fd
ssize_t read(int fd, void *buf, size_t size);
ssize_t write(int fd, const void *buf, size_t size);
int close(int fd);
```

以 open 为例，追根溯源，可以发现 open 函数最终是 libjavacore.so 执行的，因此 hook libjavacore.so 即可，找到 hook 目标 so 的目的是把 hook 的影响范围尽可能地降到最小。不同的 Android 版本可能会有些不同，目前兼容到 Android P。

另外，不同于其它 IO 事件，对于资源泄漏监控，Android 本身就支持了该功能，这是基于工具类 dalvik.system.CloseGuard 来实现的，因此在 Java 层通过反射 hook 相关 API 即可实现资源泄漏监控。

hook 介绍

想要了解 hook 技术，首先需要了解动态链接，了解动态链接之前，又需要从静态链接说起。

静态链接可以让开发者们相对独立地开发自己的程序模块，最后再链接到一起，但静态链接也存在浪费内存和磁盘更新、更新困难等问题。比如 program1 和 program2 都依赖 Lib.o 模块，那么，最终链接到可执行文件中的 Lib.o 模块将会有两份，极大地浪费了内存空间。同时，一旦程序中有任何模块更新，整个程序就要重新链接、发布给用户。

因此，要解决空间浪费和更新困难这两个问题，最简单的办法就是把程序的模块相互分割开来，形成独立的文件，而不再将它们静态地链接在一起。也就是说，要在程序运行时进行链接，这就是动态链接的基本思想。

虽然动态链接带来了很多优化，但也带来了一个新的问题：共享对象在装载时，如何确定它在进程虚拟地址空间中的位置？

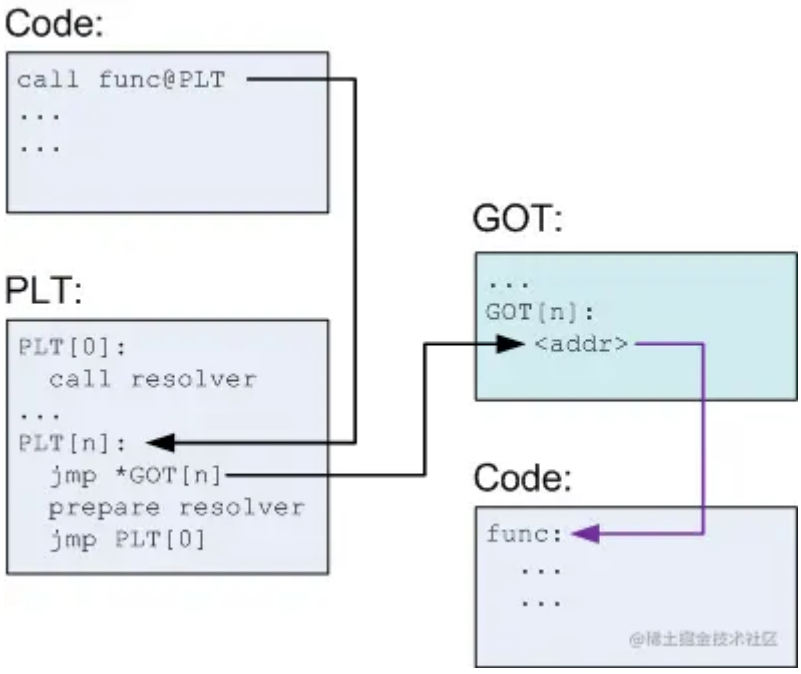
解决思路是把指令中那些需要修改的部分分离出来，和数据部分放在一起。

对于模块内部的数据访问、函数调用，因为它们之间的相对位置是固定的，因此这些指令不需要重定位。

对于模块外部的数据访问、函数调用，基本思想就是把地址相关的部分放到数据段里面，建立一个指向这些变量的指针数组，这个数据也被称为全局偏移表（Global Offset Table，GOT）。链接器在装载模块的时候会查找每个变量所在的地址，然后填充 GOT 中的各个项，以确保每个指针指向的地址正确。

但 GOT 也带来了新的问题——性能损失，动态链接比静态链接慢的主要原因就是动态链接对于全局和静态的数据访问都要进行复杂的 GOT 定位，然后间接寻址。

对于这个问题，在一个程序运行过程中，可能很多函数直到程序执行完毕都不会被用到，比如一些错误处理函数等，如果一开始就把所有函数都链接好实际上是一种浪费，所以 ELF 采用了延迟绑定的方法，基本思想是当函数第一次被用到时才由动态链接器来进行绑定（符号查找、重定位等）。延迟绑定对应的就是 PLT（Procedure Linkage Table） 段。也就是说，ELF 在 GOT 之上又增加了一层间接跳转。



因此，所谓 hook 技术，实际上就是修改 PLT/GOT 表中的内容。

源码解析

IOCanary 的源码结构是很清晰的，流程大致如下：

1. hook 目标 so 文件的 open、read、write、close 函数
2. 在执行文件 IO 时记录 IO 耗时、操作次数、缓冲区大小等信息，使用结构体 IOInfo 保存
3. 在 IO 执行完毕，调用 close 方法时，将 IOInfo 插入到一个队列

4. 后台线程循环从队列获取 IOInfo，并交给 Detector 检查
5. 如果 Detector 认为有问题，则上报

hook

IOCanary 的 hook 目标 so 文件包括 libopenjdkjvm.so、libjavacore.so、libopenjdk.so，每个 so 文件的 open 和 close 函数都会被 hook，如果是 libjavacore.so，read 和 write 函数也会被 hook。源码如下所示，

```
const static char* TARGET_MODULES[] = {
    "libopenjdkjvm.so",
    "libjavacore.so",
    "libopenjdk.so"
};
const static size_t TARGET_MODULE_COUNT = sizeof(TARGET_MODULES) /
sizeof(char*);

JNIEXPORT jboolean JNICALL
Java_com_tencent_matrix_iocanary_core_IOCanaryJniBridge_doHook(JNIEnv *env,
jclass type) {

    for (int i = 0; i < TARGET_MODULE_COUNT; ++i) {
        const char* so_name = TARGET_MODULES[i];

        void* soinfo = xhook_elf_open(so_name);

        // 将目标函数替换为自己的实现
        xhook_hook_symbol(soinfo, "open", (void*)ProxyOpen,
(void**)&original_open);
        xhook_hook_symbol(soinfo, "open64", (void*)ProxyOpen64,
(void**)&original_open64);

        bool is_libjavacore = (strstr(so_name, "libjavacore.so") != nullptr);
        if (is_libjavacore) {
            xhook_hook_symbol(soinfo, "read", (void*)ProxyRead,
(void**)&original_read);
            xhook_hook_symbol(soinfo, "__read_chk", (void*)ProxyReadChk,
(void**)&original_read_chk);
            xhook_hook_symbol(soinfo, "write", (void*)ProxyWrite,
(void**)&original_write);
            xhook_hook_symbol(soinfo, "__write_chk", (void*)ProxyWriteChk,
(void**)&original_write_chk);
        }

        xhook_hook_symbol(soinfo, "close", (void*)ProxyClose,
(void**)&original_close);

        xhook_elf_close(soinfo);
    }
}
```

统计 IO 操作

为了分析是否出现主线程 IO、缓冲区过小、重复读同一文件等问题，首先需要对每一次的 IO 操作进行统计，记录 IO 耗时、操作次数、缓冲区大小等信息。

这些信息最终都会由 Collector 保存，为此，在执行 open 操作时，需要创建一个 IOInfo，并保存到 map 里面，key 为文件句柄：

```
int ProxyOpen(const char *pathname, int flags, mode_t mode) {
    int ret = original_open(pathname, flags, mode);
    if (ret != -1) {
        DoProxyOpenLogic(pathname, flags, mode, ret);
    }
    return ret;
}

static void DoProxyOpenLogic(const char *pathname, int flags, mode_t mode, int
ret) {
    ... // 通过 Java 层的 IOCanaryJniBridge 获取 JavaContext
    iocanary::IOCanary::Get().OnOpen(pathname, flags, mode, ret, java_context);
}

void IOCanary::OnOpen(...) {
    collector_.OnOpen(pathname, flags, mode, open_ret, java_context);
}

void IOInfoCollector::OnOpen(...) {
    std::shared_ptr<IOInfo> info = std::make_shared<IOInfo>(pathname,
java_context);
    info_map_.insert(std::make_pair(open_ret, info));
}
```

接着，在执行 read/write 操作时，更新 IOInfo 的信息：

```
void IOInfoCollector::OnWrite(...) {
    CountRWInfo(fd, FileOpType::kWrite, size, write_cost);
}

void IOInfoCollector::CountRWInfo(int fd, const FileOpType &fileOpType, long
op_size, long rw_cost) {
    info_map_[fd]->op_cnt_ ++;
    info_map_[fd]->op_size_ += op_size;
    info_map_[fd]->rw_cost_us_ += rw_cost;
    ...
}
```

最后，在执行 close 操作时，将 IOInfo 插入到队列中：

```
void IOCanary::OnClose(int fd, int close_ret) {
    std::shared_ptr<IOInfo> info = collector_.OnClose(fd, close_ret);
    OfferFileIOInfo(info);
}

void IOCanary::OfferFileIOInfo(std::shared_ptr<IOInfo> file_io_info) {
    std::unique_lock<std::mutex> lock(queue_mutex_);
    queue_.push_back(file_io_info); // 将数据保存到队列中
    queue_cv_.notify_one(); // 唤醒后台线程，队列有新的数据了
    lock.unlock();
}
```

检测 IO 事件

后台线程被唤醒后，首先会从队列中获取一个 IOInfo：

```
int IOCanary::TakeFileIOInfo(std::shared_ptr<IOInfo> &file_io_info) {
    std::unique_lock<std::mutex> lock(queue_mutex_);

    while (queue_.empty()) {
        queue_cv_.wait(lock);
    }

    file_io_info = queue_.front();
    queue_.pop_front();
    return 0;
}
```

接着，将 IOInfo 传给所有已注册的 Detector，Detector 返回 Issue 后再回调上层 Java 接口，上报问题：

```
void IOCanary::Detect() {
    std::vector<Issue> published_issues;
    std::shared_ptr<IOInfo> file_io_info;
    while (true) {
        published_issues.clear();

        int ret = TakeFileIOInfo(file_io_info);
        for (auto detector : detectors_) {
            detector->Detect(env_, *file_io_info, published_issues); // 检查该 IO
事件是否存在问题
        }

        if (issued_callback_ && !published_issues.empty()) { // 如果存在问题
            issued_callback_(published_issues); // 回调上层 Java 接口并上报
        }
    }
}
```

以 small_buffer_detector 为例，如果 IOInfo 的 buffer_size_ 字段大于选项给定的值就上报问题：

```
void FileIOSmallBufferDetector::Detect(...) {
    if (file_io_info.op_cnt_ > env.kSmallBufferOpTimesThreshold // 连续读写次数
            && (file_io_info.op_size_ / file_io_info.op_cnt_) <
env.GetSmallBufferThreshold() // buffer size
            && file_io_info.max_continual_rw_cost_time_μs_ >=
env.kPossibleNegativeThreshold) /* 连续读写耗时 */ {
        PublishIssue(Issue(kType, file_io_info), issues);
    }
}
```

资源泄漏监控

Android framework 已实现了资源泄漏监控的功能，它是基于工具类 dalvik.system.CloseGuard 来实现的。以 FileInputStream 为例，在 GC 准备回收 FileInputStream 时，会调用 guard.warnIfOpen 来检测是否关闭了 IO 流：

```
public class FileInputStream extends InputStream {

    private final CloseGuard guard = CloseGuard.get();

    public FileInputStream(File file) {
        ...
        guard.open("close");
    }

    public void close() {
        guard.close();
    }

    protected void finalize() throws IOException {
        if (guard != null) {
            guard.warnIfOpen();
        }
    }
}
```

CloseGuard 的部分源码如下：

```
final class CloseGuard {
    public void warnIfOpen() {
        REPORTER.report(message, allocationSite);
    }
}
```

可以看到，执行 warnIfOpen 时如果未关闭 IO 流，就调用 REPORTER 的 report 方法。

因此，利用反射把 REPORTER 换成自己的就行了：

```
public final class CloseGuardHooker {
```

```
    private boolean tryHook() {
        Class<?> closeGuardCls = Class.forName("dalvik.system.CloseGuard");
        Class<?> closeGuardReporterCls =
Class.forName("dalvik.system.CloseGuard$Reporter");
        Method methodGetReporter =
closeGuardCls.getDeclaredMethod("getReporter");
        Method methodSetReporter =
closeGuardCls.getDeclaredMethod("setReporter", closeGuardReporterCls);
        Method methodSetEnabled = closeGuardCls.getDeclaredMethod("setEnabled",
boolean.class);

        sOriginalReporter = methodGetReporter.invoke(null);

        methodSetEnabled.invoke(null, true);

        ClassLoader classLoader = closeGuardReporterCls.getClassLoader();
        methodSetReporter.invoke(null, Proxy.newProxyInstance(classLoader,
            new Class<?>[]{closeGuardReporterCls},
            new IOCloseLeakDetector(issueListener, sOriginalReporter)));
    }
}
```

framework 很多代码都用了 CloseGuard ，因此，诸如文件资源没 close、Cursor 没有 close 等问题都能通过它来检测。

总结

IOCanary 是一个在开发、测试或者灰度阶段辅助发现 I/O 问题的工具，目前主要包括文件 I/O 监控和 Closeable Leak 监控两部分。具体的问题类型有 4 种：

1. 在主线程执行了 IO 操作
2. 缓冲区太小
3. 重复读同一文件
4. 资源泄漏

基于 xHook，IOCanary 将收集应用的所有文件 I/O 信息并进行相关统计，再依据一定的算法规则进行检测，发现问题后再上报到 Matrix 后台进行分析展示。

流程如下：

1. hook 目标 so 文件的 open、read、write、close 函数
2. 在执行文件 IO 时记录 IO 耗时、操作次数、缓冲区大小等信息，使用结构体 IOInfo 保存
3. 在 IO 执行完毕，调用 close 方法时，将 IOInfo 插入到一个队列
4. 后台线程循环从队列获取 IOInfo，并交给 Detector 检查
5. 如果 Detector 认为有问题，则上报

不同于其它 IO 事件，对于资源泄漏监控，Android 本身就支持了该功能，这是基于工具类 dalvik.system.CloseGuard 来实现的，因此在 Java 层通过反射 hook CloseGuard 即可实现资源泄漏监控。因为 Android 框架层很多代码都用了 CloseGuard ，因此，诸如文件资源没 close、Cursor 没有 close 等问题都能通过它来检测。