

TrackerSDK简介

原创 | © 2021-01-14 11:06 | ✍ 2022-11-14 18:06 编辑过 | 👁 69次浏览 | 💬 2条评论

为什么需要本地日志回捞

某些线上崩溃问题，bugly日志提供的信息很有限，仅靠它们并不能快速定位和解决问题，还有一些问题并不导致崩溃，却能导致非预期的结果，当出现这些问题时，如果对应的客户端本地有比较详细的日志记录，我们通过某种机制将它拿到，就可以更好地解决问题。

以往，58同城app使用TEG提供的wlog库实现本地日志采集及上传功能，但wlog库记录的日志信息不全，尤其是崩溃前的日志会丢失，而这部分日志恰恰是定位和解决问题的关键，所以需要更换一个更可靠的方式完成这个任务。

TrackerSDK有什么功能？

1. 可靠的日志本地持久化、上传功能及自动清理策略。
2. 公共行为搜集记录（包括组件生命周期、用户交互事件、网络访问）
3. 灵活的可选配置，包括复用app层OkHttpClient、可配置的异步实现及UI实现等。

现状描述

app里的log可以大致分为两种，一种是利用Android SDK提供的Log，在控制台输出以便调试等用途，另一种是需要做本地持久化和上传的。在本地版APP中，主要通过TLog实现前者，但TLog存在于TownLib中，所以PowerLib是用过WubaLogLib中的LOGGER实现日志输出的，本地版目前没有对日志进行持久化；而同城app主要通过LOGGER实现对应功能，LOGGER存在于WubaLogLib

WubaLogLib和WLog

据了解，WLog库最早是由同城移交给TEG的，之后的WLog库由TEG开发迭代，这两个库里都有名为WLog的类，但在WubaLogLib中，将**WLog**的**本地持久化及本地日志上传功能**注释掉了，这样，它目前只具有本地控制台输入日志的功能，并且针对这一功能，又封装了LOGGER这个对外入口。

可以想到，只是注释掉这部分功能的代码，接口没有变动，当需要本地持久化及上传时，不需要改动App代码，只需要在WubaLogLib中改就可以了，那么我们新设计的日志库，也必须能做到这点。

在WubaLogLib中，本地持久化及上传服务端是通过Collector类实现的，它内部持有的WLog类属于TEG的WLog库。目前，本地版并没有用到这个类。

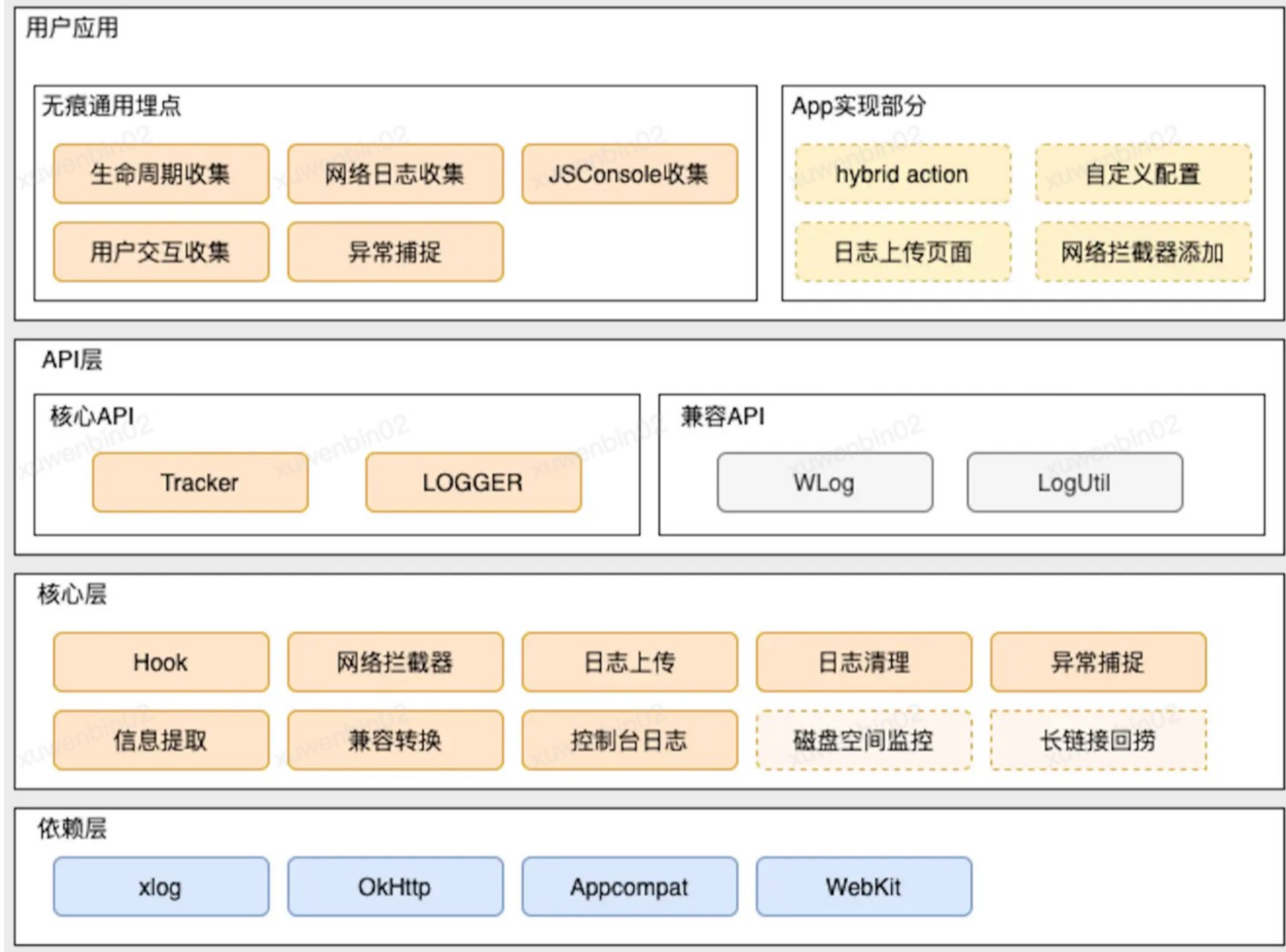
开源日志库选择

之前调研了腾讯的XLog和美团Logan，二者都基于linux的mmap函数实现，都可以保证日志完整性。在Demo工程中，对于两个库我都实现了相应的Module封装，目前接入到程序中的是XLog。

	XLog	Logan
体积	951kb	178kb
日志压缩算法	LZ77	流式gzip压缩（16kb）
加密方式	ECDH、TEA	RSA、AES
server端	提供了解压解密的python脚本	提供了基于Spring、mysql的Server端程序及日志分析平台

关于加密解密细节可以参见最后一部分

整体结构设计



核心API

核心对外API是Tracker和LOGGER，前者负责配置和持久化日志记录，后者负责控制台日志输出，由于LOGGER也是WubaLogLib中的控制台日志输出类，出于兼容性考虑，保留了原包名

兼容包

要整体替换掉WubaLogLib这个库，为了保证APP中代码的最少改动，新的库必须最大化地保持WubaLogLib中对外入口类的特征不变（包名、类名、方法名），并且要保持原有结构的扩展性。兼容包包含了WubaLogLib的对外API。

详细流程设计



李运哲

作者

+关注

技术工程平台群-Android技术部

文章	访问	获赞	评论
5	274	6	3

🔥 神奇豆 0	🏆 总排名 16068
---------	-------------

相关文章

🔄 换一换

- 

【创新大赛火热报名ing】基于AIGC技术创新大赛开始...
2023-03-25 20:55:26
- 

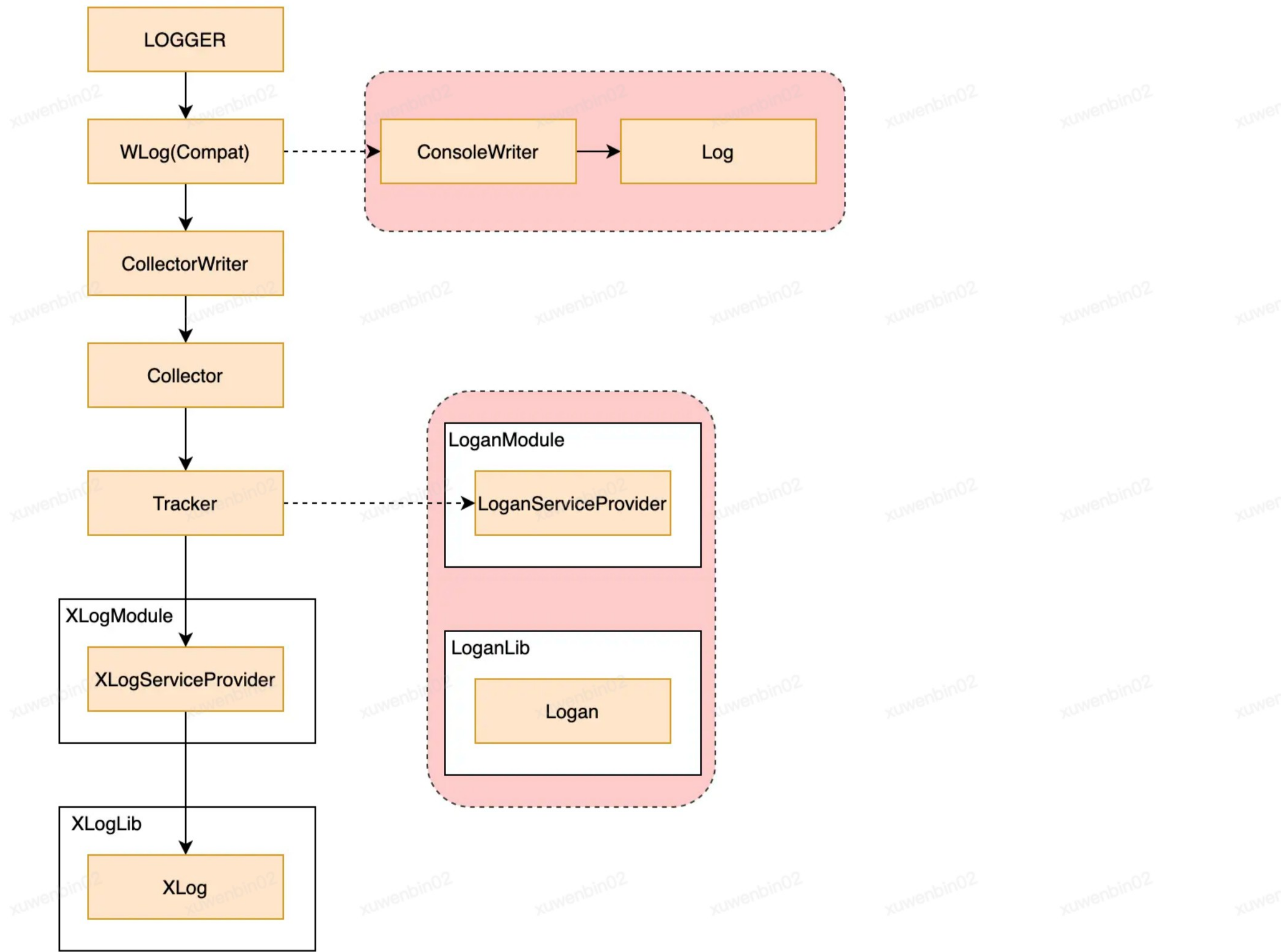
58集团战略项目管理白皮书倾心发布
2023-03-22 15:16
- 

WMB事务消息应用场景及技术解析
2023-01-11 18:39:42
- 

【AI创未来创新大赛】不是终点，而是起点！
2023-04-21 14:41:42
- 

WTrace 支持全量采集啦!!!
2023-04-12 14:32:9
- 

服务覆盖率平台：可针对未覆盖代码进行分析
2023-03-6 15:2:15



Tracker类作为统一的对外API，app中建议直接通过此类访问tracker库，而之前对WubaLogLib有依赖的app，也无需改动代码，Collector在签名不变的前提下，通过Tracker类进行日志持久化。如图，LOGGER依赖WLog，WLog中可以通过配置Writer实现切换日志的本地写入或控制台输出等行为，红色虚线框里的是可选部分，对于LOGGER及WLog这两个对外入口，如果把WLog中的Writer设置为ConsoleWriter，那么日志只会在控制台输出，如果设置为CollectorWriter，则会通过Tracker类进行本地持久化（默认只持久化Error级别日志），持久化的具体方式取决于Collector中设置的LogServiceProvider，目前我们接入的XLog，由XLog负责这一事宜。

Activity生命周期信息收集

ActivityLifecycleHook类实现ActivityLifecycleCallbacks接口，对activity生命周期监听，它主要实现以下功能：

- 1. 注册UncaughtExceptionHandler实现异常捕捉记录，并以代理模式保证了其他业务注册的UncaughtExceptionHandler能够正常运行。
- 2. 在Activity创建时对supportFragmentManager注册fragment生命周期hook。
- 3. 监听Activity生命周期，并在对应方法中输入日志，这种输出是可配置的，在无特殊配置的前提下（默认）会对所有生命周期方法记录日志，可以通过全局配置来设置需要监听记录某些生命周期或对特定类进行更详细的记录。

在activity的onCreate中会输出Bundle信息。

Fragment生命周期收集

ActivityLifecycleHook在activity初始化时判定如果是FragmentActivity，就会为他注册fragment生命周期监听，跟Activity一样，这个也支持通过配置更改是否要对特定方法进行监听。

Activity及Fragment生命周期监听的配置

生命周期方法记录日志的流程如下图：

1. 通用配置

分别通过一个int值来配置对具体方法的监听，int值的各个位取值是1则监听对应的方法，定义大致如下：

```
public static final int FLAG_FRAGMENT_ATTACHED = 1;
public static final int FLAG_FRAGMENT_CREATED = 1 << 1;
public static final int FLAG_FRAGMENT_VIEW_CREATED = 1 << 2;
public static final int FLAG_FRAGMENT_STARTED = 1 << 3;
public static final int FLAG_FRAGMENT_RESUMED = 1 << 4;
public static final int FLAG_FRAGMENT_PAUSED = 1 << 5;
public static final int FLAG_FRAGMENT_STOP = 1 << 6;
public static final int FLAG_FRAGMENT_VIEW_DESTROY = 1 << 7;
public static final int FLAG_FRAGMENT_DESTROY = 1 << 8;
public static final int FLAG_FRAGMENT_DETACH = 1 << 9;
```

2. 特殊配置

可以通过类全名来设置对特定Activity或Fragment的所有生命周期进行监听，特殊配置的优先级高于通用配置。

View交互信息收集

通过ActivityLifecycleHook在activity创建时获取rootView并且递归设置交互代理，交互代理可选方案有两种：OnClickListener 和 AccessibilityDelegate，OnClickListener只能监听点击事件，而且它在View中的实例包裹在ListenerInfo中，相关方法都是package权限，只能通过反射获取和设置，随着androidSDK后续版本更新，这种方式具有较高风险。而 AccessibilityDelegate 能监听点击、长按等多种交互事件，而且从API LEVEL 26之后，View提供了getter来获取旧的AccessibilityDelegate实例，我们可以根据版本判断，在这个版本之前，用反射获取，这样是万无一失的。代码如下：


```
        if(Build.VERSION.SDK_INT > Build.VERSION_CODES.P){
            return view.getAccessibilityDelegate();
        }else{
            try {
                return (View.AccessibilityDelegate) mAccessibilityDelegateField.get(view);
            }catch (Exception e){
                Collector.e(e);
                return null;
            }
        }
    }
}
```

View交互监听需要记录的事件类型也一样可以通过配置动态设置。目前程序中只监听click和longClick。

View日志正文格式

以下面一条日志正文为例（不包括tag部分）：

```
[com.wuba.wbtown.components.bottomnavigations.FixedBottomNavigationTab{id=-1, textContent= 信息 } >> onClicked
```

View日志分为两部分，以 >> 分隔，前面是View基本信息，后面是事件信息，对于点击事件，它只有事件名。我们可以看到，这是一个自定义View，通过查看源码可以知道它继承的是FrameLayout，日志记录了它的id和文本内容，可以很容易知道用户具体是点击了哪个view。对于不同类型的View，我们输出的信息也不同，最终目的是通过日志能够看到事件关联的是哪个View来准确定位问题。那么，这个问题其实就是如何能把任意view转换成相对更有可读性的字符串信息，对此，我们提供了转换器接口 Stringifier 和一个可动态配置的转换器容器 StringifierStore。

转换器接口接受任意类型并将其转换为字符串：

```
public interface Stringifier<T> {
    String stringify(T t);
}
```

转换器容器 StringifierStore 内部维护了一个map，将类型和字符串转换器进行映射，当交互事件被触发时，会以目标View实例的具体类型去容器中寻找转换器，如果找不到，则向上寻找其父类，我们提供了以下几种默认的转换器：

转换器	转换目标类	输出字符串内容
TextViewStringifier	TextView	类全名、id、屏幕坐标、文字内容
ViewGroupStringifier	ViewGroup	类全名、id、文字内容（递归寻找其子View中TextView的文字内容）
ViewStringifier	View	类全名、id、屏幕坐标
ObjectStringifier	Object	输出其toString方法（View体系接触不到这个，字符串转换器在设计时并不只用于View体系）

就像上文提到的，这个转换器容器是可配置的，对于某些View如果要以特定格式输出你关注的信息，可以自定义转换器放入容器中，容器提供了 registerStringifier(Class keyClass, Stringifier stringifier) 方法，也可以通过这个方法来替换默认的转换器。

JS控制台日志抓取

上文描述了在HookView交互事件时，采用遍历viewTree的方式为每个可点击可见View设置交互事件代理，在这个过程中，我们同时进行了判断，如果是WebView，就为它添加WebChromeClient代理：

```
private void replaceOnClickListenerWithDelegateRecursively(View view) {
    if (view != null && view.getVisibility() == View.VISIBLE) {
        if (view instanceof ViewGroup) {
            replaceOnClickListenerWithDelegateByReflectIfNeeded(view);
            ViewGroup viewGroup = (ViewGroup) view;
            for (int i = 0; i < viewGroup.getChildCount(); i++) {
                replaceOnClickListenerWithDelegateRecursively(viewGroup.getChildAt(i));
            }
        } else if (view instanceof WebView){
            WebViewHookHelper.hookWebView((WebView) view);
        }else {
            replaceOnClickListenerWithDelegateByReflectIfNeeded(view);
        }
    }
}
```

具体做法跟前面一样，先取出原有的WebChromeClient，以静态代理模式先执行我们的逻辑，再将函数调用转发给原WebChromeClient。
取出原WebChromeClient的方式是通过androidx的webkit库中的WebViewCompat方法，可以保证兼容性：

```
public static void hookWebView(WebView webView){
    if(WebViewFeature.isFeatureSupported(WebViewFeature.GET_WEB_CHROME_CLIENT)) {
        WebChromeClient webChromeClient = WebViewCompat.getWebChromeClient(webView);
        if(webChromeClient instanceof NestedWebChromeClient){
            return;
        }
        webView.setWebChromeClient(new NestedWebChromeClient(webChromeClient));
    }
}
```

在我们的代理WebChromeClient中，通过拦截onConsoleMessage方法，将js的控制台日志持久化到我们的本地日志里。考虑到正式版本的前端页面基本都会关闭用于开发和调试的日志，剩下的应该都是报错信息，不会有太多噪音，这方面可以先试运行看效果，如果有问题再调整为其他方案，比如只记录error级别的信息等。

异常捕捉

WubaTrackerLib库提供了ExceptionCollectWatchDog类捕捉异常，它实现了ActivityLifecycleCallbacks，需要在Application初始化时注册，它内部持有UncaughtExceptionHandler的实现，在发生异常时，调用Collector记录异常信息，而实现ActivityLifecycleCallbacks的意义是保证UncaughtExceptionHandler不被替换，在Activity初始化时，判断如果Thread的UncaughtExceptionHandler不是我们设置的，就把原来的取出来，放到我们的UncaughtExceptionHandler中，发生异常时，我们先执行我们的逻辑，再把方法调用转发给原有UncaughtExceptionHandler。

含有保活策略的代理模式嵌套问题

上文提到的异常捕捉、生命周期及用户交互事件监控等都是通过hook方式用代理模式实现，代理模式保证了执行我们业务逻辑的同时也不丢失早于我们设置的处理逻辑，但如果有两方都使用这种方式，而且其中一方有保活策略的话，就会形成循环引用，被代理对象是彼此，或者更复杂一些的情况可能是环状引用，这样在执行的时候就会stackoverflow，解决这个问题，我们主要看具体情况：

对于UncaughtExceptionHandler，我们观察它的处理函数：


```
@Override
public void uncaughtException(Thread t, Throwable e) {
    if(lastThrowable != e) {
        lastThrowable = e;
        Tracker.e(LogTags.TAG_EXCEPTION, e);
        Tracker.flush();
        if (innerHandler != null) {
            innerHandler.uncaughtException(t, e);
        }
    }
}
```

这个函数中，由于发生异常时throwable是唯一的，只要在处理时记录它的引用，下次再进入这个函数判断引用相等就一定是环状调用，直接终止即可。

而参数不具有唯一性的函数，我们就不能这样处理了，只能通过分析调用栈，如果调用栈轨迹重复进入，则判定是环状调用，终止即可。当然，这种方法性能不及前者，所以优先使用前者。

日志上传

通用上传模块

理论上通用上传模块不应该依赖具体网络技术，只实现输入时间范围，输出文件列表的功能，实际考虑目前绝大多数app都是用okhttp作为网络访问库，Tracker直接依赖okhttp也不会导致存在两个网络访问库的情况，但不能忽视的是OkHttpClient的复用，由于OkHttpClient会持有线程池及网络连接资源，一般情况下都是应该进程内唯一的，对此，Tracker的网络模块提供了设置OkHttpClient的方法，当Tracker需要访问网络时，只有未设置的情况才会自己创建新的。

日志action

日志action依赖我们的hybridSDK，这部分不属于TrackerSDK的范围，需要在app内实现，前端同学会把要记录的信息通过js收集拼装分组，调用我们的action进行本地持久化。action名称：behaviorRecord，参数content是前端自己拼装的json结构，格式不是我们的关注点，只要他们能解析即可。

通用日志标签

以一条日志为例：

[D][2020-11-27 +8.0 18:00:28.594][23857, 2*][wuba][, , 0]

这是xlog默认的日志标签部分，其中从左至右分别为：[日志级别]、[时间]、[进程id,线程id]、[自定义标签]、[文件名,函数名,行数]

其中最后一部分[文件名,函数名,行数]实测并没有什么作用，考虑去除，针对通用日志，自定义标签我们定义了以下分类：

标签名	内容
L	生命周期
I	交互事件
N	网络请求
E	异常捕捉
H	js控制台输出
T	Tracker库内部输出

专用日志标签由具体负责同学自定义，尽量简短并要留下文档。

回捞方案

基于长链接

在bugly中记录用户IMEI，根据用户IMEI，通过长链接向用户设备发送消息，并在服务端数据库记录一个待上传标识，用户收到消息后， 查询对应接口，如果标识未被清除，则上传，上传成功后调用接口更改标识。

另外，在每次启动app的时候也要请求这个接口以防基于长链接的即时消息丢失导致无法上传。

主动上传

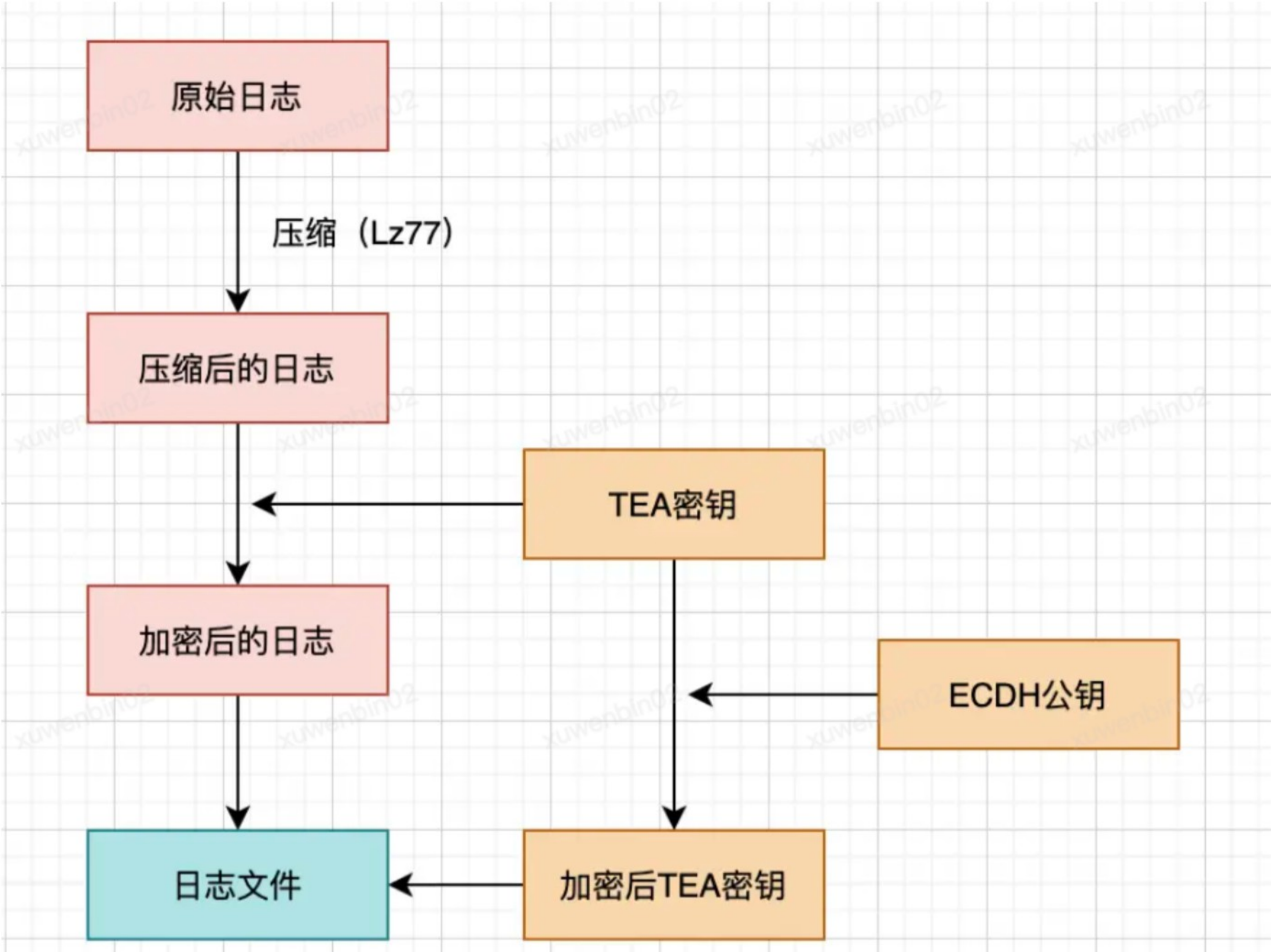
用户通过某页面隐藏入口进入上传页面上传本地日志，好处是实现简单，坏处是只能通过主动联系发生问题的用户来回捞日志

附录 加密解密流程

日志文件的加密方式，XLog和Logan都是采用客户端随机生成对称加密密钥用来加密压缩后的日志正文，然后用配置好的非对称加密密钥来加密生成的对称加密密钥，将加密后的密钥和正文一起写入文件，解密同理，但在加密算法选择上二者有所不同。XLog使用ECDH作为非对称加密算法，TEA作为对称加密算法，而Logan使用RSA作为非对称加密算法，AES作为对称加密算法。

XLog

加密流程




图中所示为XLog压缩加密流程，ECDH的一对密钥是用腾讯提供的python脚本生成，脚本逻辑很简单，只是简单调用了python的加密库 `pyelliptic`，使用椭圆曲线 `secp256k1` 生成公私钥来加密和解密对称加密密钥，对称加密使用的是腾讯惯用的16轮迭代TEA加密算法+改良CBC模式加密数据块。

代码实现上，加密逻辑在 `log_crypt.cc` 中，如果需要更改加密算法，可以修改 `CryptSyncLog` 函数和 `CryptAsyncLog` 函数，它们分别用来加密同步模式和异步模式的日志，修改后需要同时修改python解密脚本

[点赞](#) | [收藏](#) | [邮件分享](#) | [微信分享](#) | [复制链接](#) | [转发](#)

文彤

评论一个吧...



☐ 匿名 [?](#)

评论

全部评论 (2)



凌日新 : 

2021-01-14 16:45:29



赞 (0)

回复



崔乾博 : 

2021-01-14 11:14:58



赞 (0)

回复