**[Android 从StackTraceElement反观Log库](http://blog.csdn.net/lmj623565791/article/details/52506545)**

## 一、概述

大家编写项目的时候，肯定会或多或少的使用Log，尤其是发现bug的时候，会连续在多个类中打印Log信息，当问题解决了，然后又像狗一样一行一行的去删除刚才随便添加的Log，有时候还要几个轮回才能删除干净。

当然了，我们有很多方案可以不去删除：

* 我们可以通过gradle去配置debug、release常量去区分
* 可以对Log进行一层封装，通过debug开关常量来控制

当然了，更多时候我们是不得不删除的，比如修bug着急的时候，一些Log.e("TAG","马丹，到底是不是null,obj = "+=obj)，各种词汇符号应该都会有。

所以，我们的需求是这样的：

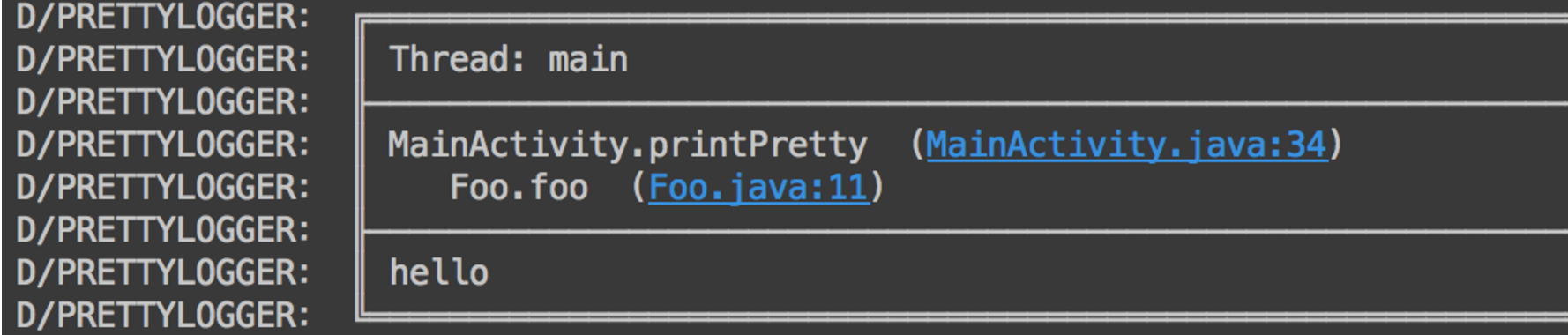
1. 可以对Log封装，通过debug开关来控制正常日志信息的输出
2. 在修bug时，用于定位的杂乱log日志，我们希望可以在bug解除后，很快的定位到，然后删除灭迹。

ok，我们今天要谈的就是Log的封装，当然封装不仅仅是是上述的好处，我们还可以让使用更加便捷，打出来的Log信息展示的更加优雅。

比如：

* <https://github.com/orhanobut/logger>

这个库，就对Log的信息的展示做了非常多的处理，展示给大家是一个非常nice的效果：



当然今天的博文不是去介绍该库，或者是源码解析，不过解析的文章我最后收到了投稿，可以关注我的公众号，近期应该会推送。

今天文章的目标是：掌握这类库的核心原理，以后只要遇到该类库，大家都能说出其本质，以及可以自己去封装一个适合自己的日志库。

## 二、可行性

对于好用，我觉得如下用法就可以：

L.e("heiheihei");

对于好定位，当然是可以通过日志信息点击，定位到具体行，所以今天demo代码的效果是这样的：



当然了，你可以根据自己喜好，去添加各种信息，以及装饰。

那么，现在最大的一个问题就是

* 我怎么输出具体的日志调用行呢？

这个秘密就在：

Thread.currentThread().getStackTrace();

我们可以通过当前的线程，拿到当前调用的栈帧集合（称呼不一定准备）。

* 这个栈帧集合是什么玩意呢？

你可以理解为当我们调用方法的时候，每进入一个方法，会将该方法的相关信息（例如：类名，方法名，方法调用行数等）存储下来，压入到一个栈中，当方法返回的时候再将其出栈。

下面看个具体的例子：

@Override

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.activity\_main);

a();

}

void a() {

b();

}

void b() {

StringBuffer err = new StringBuffer();

StackTraceElement[] stack = Thread.currentThread().getStackTrace();

for (int i = 0; i < stack.length; i++) {

err.append("\tat ");

err.append(stack[i].toString());

err.append("\n");

}

Log.e("TAG", err.toString());

}

我在onCreate中，调用了a方法，然后a中调用的b方法。在b方法中打印出当前线程中的栈帧集合信息。

at dalvik.system.VMStack.getThreadStackTrace(Native Method)

at java.lang.Thread.getStackTrace(Thread.java:579)

at com.zxy.recovery.test.MainActivity.b(MainActivity.java:26)

at com.zxy.recovery.test.MainActivity.a(MainActivity.java:21)

at com.zxy.recovery.test.MainActivity.onCreate(MainActivity.java:17)

at android.app.Activity.performCreate(Activity.java:5231)

...

可以看到我们整个方法的调用过程，底部的最先开始调用，顺序为onCreate->a->b->Thread.getStackTrace->VMStack.getThreadStackTrace.

最后两个是因为我们的stacks是在VMStack.getThreadStackTrace方法中获取，然后返回的，所以包含了这两个的内部调用信息。

这里我们直接调用的StackTraceElement的toString方法，它内部有:

* getClassName
* getMethodName
* getFileName
* getLineNumber

看名字就知道什么意思了，我们可以根据这些信息拼接要打印的信息。

所以，不管怎么说，我们现在已经确定了，可以通过该种方式得到我们的调用某个方法的行数，而且是支持点击跳转到指定位置的。

到这里相当于，方案的可行性就通过了，剩下就是码代码了。

## 三、实现

先写个大致的代码：

public class L{

private static boolean sDebug = true;

private static String sTag = "zhy";

public static void init(boolean debug, String tag){

L.sDebug = debug;

L.sTag = tag;

}

public static void e(String msg, Object... params){

e(null, msg, params);

}

public static void e(String tag, String msg, Object[] params){

if (!sDebug) return;

tag = getFinalTag(tag);

//TODO 通过stackElement打印具体log执行的行数

Log.e(tag, content);

}

private static String getFinalTag(String tag){

if (!TextUtils.isEmpty(tag)){

return tag;

}

return sTag;

}

}

因为我平时基本上只用Log.e，所以我就不对其他方法进行处理了，你可以根据你的喜好来决定。

ok，那么现在只有一个地方没有处理，就是打印log执行的类以及代码行。

我在onCreate的17行调用了：

L.e("Hello World");

然后在e()方法中，打印了所有的栈帧信息：

E/zhy: at dalvik.system.VMStack.getThreadStackTrace(Native Method)

at java.lang.Thread.getStackTrace(Thread.java:579)

at com.zxy.recovery.test.L.e(L.java:32)

at com.zxy.recovery.test.L.e(L.java:25)

at com.zxy.recovery.test.MainActivity.onCreate(MainActivity.java:19)

at android.app.Activity.performCreate(Activity.java:5231)

//...

E/zhy: Hello World

我们要输出的就是上述的MainActivity.onCreate(MainActivity.java:19)

* 那么我们如何定位呢？

观察上面的信息，因为我们的入口是L类的方法，所以，我们直接遍历，L类相关的下一个**非L类的栈帧信息**就是具体调用的方法。

于是我们这么写：

private StackTraceElement getTargetStackTraceElement() {

// find the target invoked method

StackTraceElement targetStackTrace = null;

boolean shouldTrace = false;

StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();

for (StackTraceElement stackTraceElement : stackTrace) {

boolean isLogMethod = stackTraceElement.getClassName().equals(L.class.getName());

if (shouldTrace && !isLogMethod) {

targetStackTrace = stackTraceElement;

break;

}

shouldTrace = isLogMethod;

}

return targetStackTrace;

}

拿到确定的方法调用相关的栈帧之后，就是输出啦~~

添加到e()方法中：

public static void e(String tag, String msg, Object... params) {

if (!sDebug) return;

String finalTag = getFinalTag(tag);

StackTraceElement targetStackTraceElement = getTargetStackTraceElement();

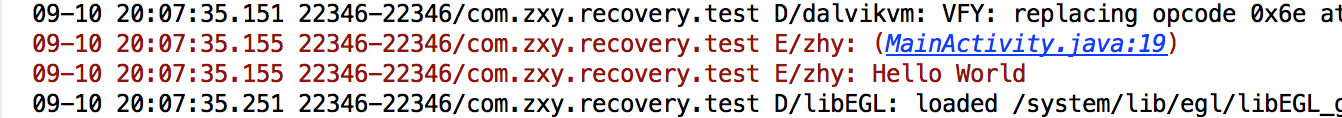
Log.e(finalTag, "(" + targetStackTraceElement.getFileName() + ":"

+ targetStackTraceElement.getLineNumber() + ")");

Log.e(finalTag, String.format(msg, params));

}

现在再看下输出结果：



现在就可以迅速的定位到日志输出行，再也不要全局搜索去查找了~

到这里，对于我个人的需求已经满足了，如果你有特殊需要，比如也想像logger那样搞个框，那就自己绘制吧，也可以参考它的源码。

对了，还有json，有时候希望可以看json字符串更加的直观，像looger那样：

你可以参考它的做法，其实就是将json字符串，通过JsonArray和JsonObject进行了一个类似format这样的操作。

private static String getPrettyJson(String jsonStr) {

try {

jsonStr = jsonStr.trim();

if (jsonStr.startsWith("{")) {

JSONObject jsonObject = new JSONObject(jsonStr);

return jsonObject.toString(JSON\_INDENT);

}

if (jsonStr.startsWith("[")) {

JSONArray jsonArray = new JSONArray(jsonStr);

return jsonArray.toString(JSON\_INDENT);

}

} catch (JSONException e) {

e.printStackTrace();

}

return "Invalid Json, Please Check: " + jsonStr;

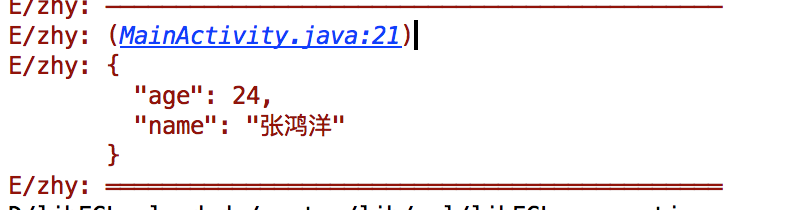
}

重点就是文本的处理了，其他的和普通log一致。

你可以独立一个L.json()方法。

L.json("{\"name\":\"张鸿洋\",\"age\":24}");

效果如下：



好了，我自己在每次输出前后加了个横线，根据自己的喜欢定制吧。

## 四、其他用法

StackElementStack在其他一些SDK里面也会用到，比如处理app的crash，有时候会重新处理下信息。

还有就是一些统计PV相关的SDK，会强制要求在某些方法中执行某个方法，例如，必须在Activity.onResume中执行，PVSdk.onResume，如果你之前遇到过某个SDK给你抛了类似的异常，那么它的原理就是这么实现的。

大致的代码如下，可能会有漏洞，随手写的：

public class PVSdk {

public static void onResume() {

StackTraceElement[] stackTrace = Thread.currentThread().getStackTrace();

boolean result = false;

for (StackTraceElement stackTraceElement : stackTrace) {

String methodName = stackTraceElement.getMethodName();

String className = stackTraceElement.getClassName();

try {

boolean assignableFromClass = Class.forName(className).isAssignableFrom(Activity.class);

if (assignableFromClass && "onResume".equals(methodName)) {

result = true;

break;

}

} catch (ClassNotFoundException e) {

// ignored

}

}

if (!result)

throw new RuntimeException("PVSdk.onResume must in Activity.onResume");

//do other things

}

}

大多时候上述代码实在debug时候开启的，发版状态可能会关闭检查，具体看自己的需求了。

包括自己再写一些库的时候，强绑定生命周期也能这么去简单的check.

## 五、总结

那么到此文章就结束了，虽然文章比较容易，不过我觉得也能解决一类问题，希望看了这个文章以后，对于任何的日志库脑子里对其实现的原理都非常清晰，看到其本质，很多时候就觉得这个东西很简单了。

最后，文章中的代码，和源码略有不同，因为源码可能会是封装后的，文章中代码是为了便于描述，都是越直观越好。

源码点击下载：

* <https://github.com/hongyangAndroid/basetools>

have a nice day ~~~