frameworks\base\packages\Shell\src\com\android\shell

framework/native/cmds/bugreport/bugreport.cpp

framework/native/cmds/dumpstate/dumpstate.cpp

framework/native/cmds/dumpstate/utils.c

<http://gityuan.com/2016/06/10/bugreport/>

<http://blog.csdn.net/feitian_666/article/details/52880213>

# TODO

# Cmd

// TODO: block until a result is returned to MyResultReceiver.

IBinder::shellCommand(service, STDIN\_FILENO, STDOUT\_FILENO, STDERR\_FILENO, args,

new MyResultReceiver());

down vote

Another way. By using "adb shell cmd"

1) Expand

adb shell cmd statusbar expand-notifications

2) Collapse

adb shell cmd statusbar collapse

3) You can expand quick settings also

adb shell cmd statusbar expand-settings

Please check help for more:

adb shell cmd statusbar help

https://github.com/fAndreuzzi/TUI-ConsoleLauncher/wiki/Root-commands

# atrace

frameworks/native/cmds/atrace/atrace.cpp

## TODO

打印到trace：

应用层：

ATRACE\_CALL();

定义ATRACE\_TAG：Hwcomposer使用了ATRACE\_TAG\_GRAPHICS，表示它和Graphics相关。

ATRACE\_INIT：用于统计某个变量使用的情况。下文将见到代码中”VSYNC”的统计结果。

ATRACE\_CALL：用于统计函数的调用情况。

内核层：

trace\_printk("1\n");

小机

cd /sys/kernel/debug/tracing

echo 1 > tracing\_on   //使能trace

cat trace\_pipe //实时抓取trace

cat trace // 缓存抓取trace

cd /sys/kernel/debug/tracing/events/sched

echo 1 > sched\_switch/enable // 使能抓取任务切换到trace

echo 1 > sched\_wakeup/enable // 使能抓取任务唤醒到trace

查看可以抓取的trace类型：进对应的event下enable即可。

cat /sys/kernel/debug/tracing/available\_events

---------------------

作者：简单的过客

来源：CSDN

原文：https://blog.csdn.net/zjli321/article/details/51833877

版权声明：本文为博主原创文章，转载请附上博文链接！

## Systrace UI性能分析

在应用程序开发过程中，UI（用户界面）的流畅度是体验的核心，特别是在动画、跳转或者列表的滑动过程中，出现卡顿和无响应是非常影响用户体验的，要解决这些问题，首先要找到问题的原因，前面介绍的TraceView是分析性能的一款利器，下面再介绍一个分析应用程序UI性能的工具：Systrace。

Systrace是Android 4.1及以上版本提供的性能数据采样和分析工具。它可以帮助开发者收集Android关键子系统（如surfaceflinger、WindowManagerService等Framework部分关键模块、服务，View系统等）的运行信息，从而帮助开发者更直观地分析系统瓶颈，改进性能。Systrace的功能包括跟踪系统的I/O操作、内核工作队列、CPU负载等，在UI显示性能分析上提供很好的数据，特别是在动画播放不流畅、渲染卡等问题上。Systrace工具可以跟踪、收集、检查定时信息，可以很直观地查看CPU周期消耗的具体时间，显示每个线程和进程的跟踪信息，使用不同颜色来突出问题的严重性，并提供如何解决这些问题的建议。

[图片]注意　由于Systrace是以系统的角度返回一些信息，并不能定位到具体耗时的方法，要进一步获取CPU满负荷运行的原因，就需要使用前面介绍过的工具Traceview。

### 埋点

Cpp层非常简单，依赖对应的libcutils

bool SurfaceFlinger::handleMessageInvalidate() {

ATRACE\_CALL();

return handlePageFlip();

}

Java层比较麻烦

Systrace不会追踪应用的所有工作，所以在有需求的情况下，需要添加要追踪的代码部分。在Android 4.3及以上版本的代码中，可以通过Trace类来实现这个功能。它能够让你在任何时候跟踪应用的一举一动。在获取Trace的过程中，即Trace.beginSection（）与Trace.endSection（）之间的代码工作会一直被追踪。

在代码中加入Trace跟踪需要注意以下两点：

·在Trace被嵌套在另一个Trace中时，endSection（）方法只会结束离它最近的一个beginSection（String），即在一个Trace的过程中是无法中断其他Trace的。所以要保证endSection（）与beginSection（String）调用次数匹配。

·Trace的begin与end必须在同一线程中执行。

下面这部分代码为使用Trace的例子，在整个方法中含有两个Trace块，可以根据需求定义更多的块，但都要成对出现，如果有开始块但没有结束块，会严重影响应用的性能。

**private void** handleTest() {  
 Trace.*beginSection*(**"handleTest"**);  
 Log.*d*(**"keytest"**,**"handleTest"**);  
 Trace.*endSection*();  
}

### Systrace发起命令

Systrace的使用不复杂。但跟踪的设备必须是Android 4.1（API16）或更高版本。在4.3版本和4.3以前版本的使用上有些区别，后面会讲到。

https://wenku.baidu.com/content/6a2246463a3567ec102de2bd960590c69ec3d8d2?m=1f7b13269e62442f93caaa768a7b08f0&type=pic&src=dfbc68b946951c40d1263f8f2ddadb8b.jpg注意　4.3以前系统版本的设备需要打开Settings>Developer options>Monitoring>Enable traces。

1. 在DDMS上使用

在Eclipse和Android Studio中都可以在DDMS直接使用Systrace，其他IDE也能支持，且流程都相同，下面以Android Studio为例说明其使用流程。

1）打开Android Device Monitor，连接手机并准备需要抓取的界面。

2）单击Systrace按钮进入抓取前的设置，选择需要跟踪的内容（见图2-18）：

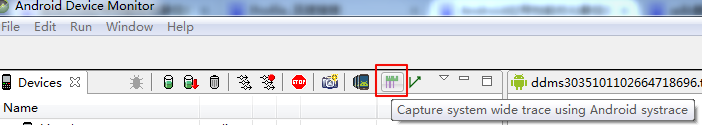


图2-18　DDMS上打开Systrace

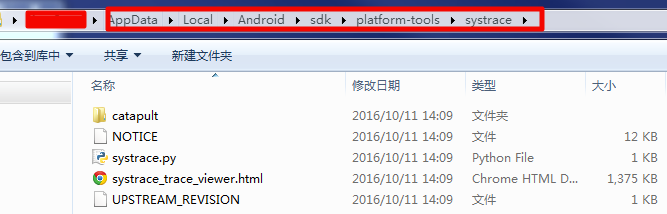
3）手机上开始操作需要跟踪的过程（如滑动列表）。

4）到了设定好的时间后，生成Trace文件。

5）使用Chrome打开文件即可分析。

**（2）使用命令行**

使用命令行方式更灵活，速度更快，并且配置好后再使用能快速得到结果，在Android 4.3及更高版本的设备上使用Systrace时，可以省略设置跟踪类别标签来获取默认值，或者可以手动列入指定标签。路径和命令如下：



[代码] python systrace.py --time=10 -o mynewtrace.html sched gfx view wm

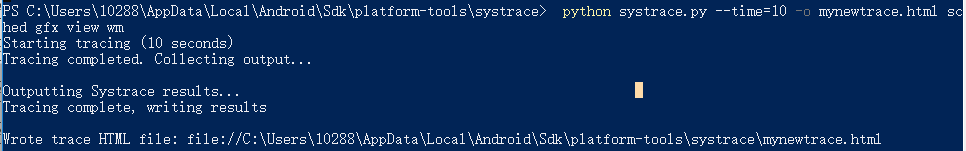
或者

python C:\Users\10288\AppData\Local\Android\Sdk\platform-tools\systrace\systrace.py --time=10 -o trace\_home.html sched gfx view wm

pause

注意：初次运行报错ImportError: No module named win32con

C:\Python27\Scripts> .\pip.exe install pypiwin32



其中参数设置对应的功能如表2-2所示。



表2-2　System参数命令

其中categories中的标签比较多，可以从[官方的文档上查询](http://developer.android.com/intl/zh-cn/tools/help/systrace.html)：

### 分析Systrace报告

通过前面方法获取到的trace.html文件，需要使用Chrome打开，有一些常用的快捷键，定义如表2-3所示。



目前Systrace产生的trace文件只能使用Chrome打开，使用Chrome打开文件后如图2-19所示。

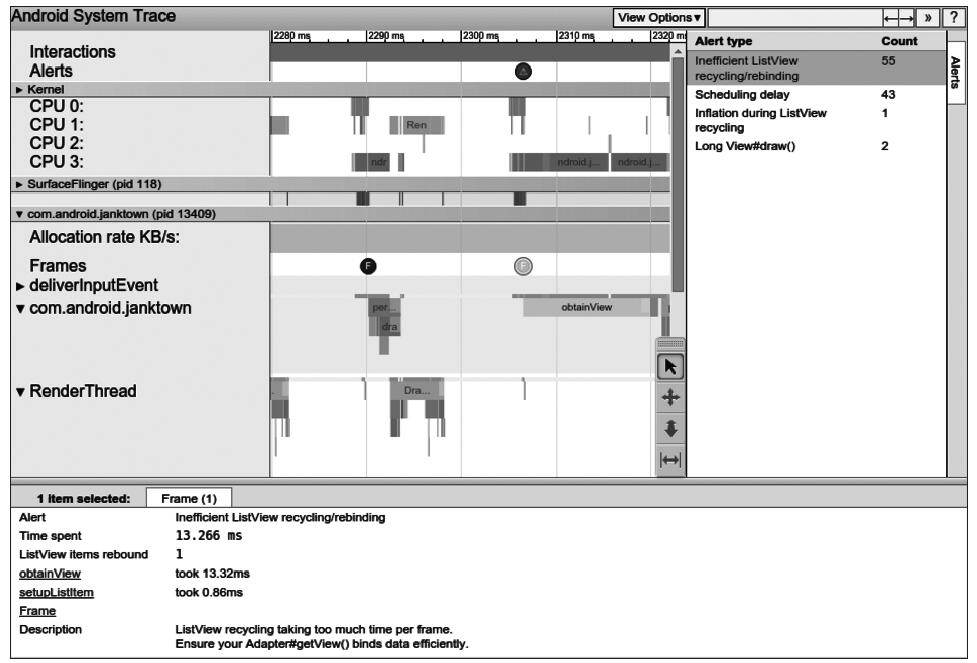


图2-19　Systrace Viewer

从图2-19中可以看到完整的数据，其中和UI绘制关系最密切的是Alerts和Frame两个数据，接下来重点介绍Alerts和Frame。

（1）Alerts

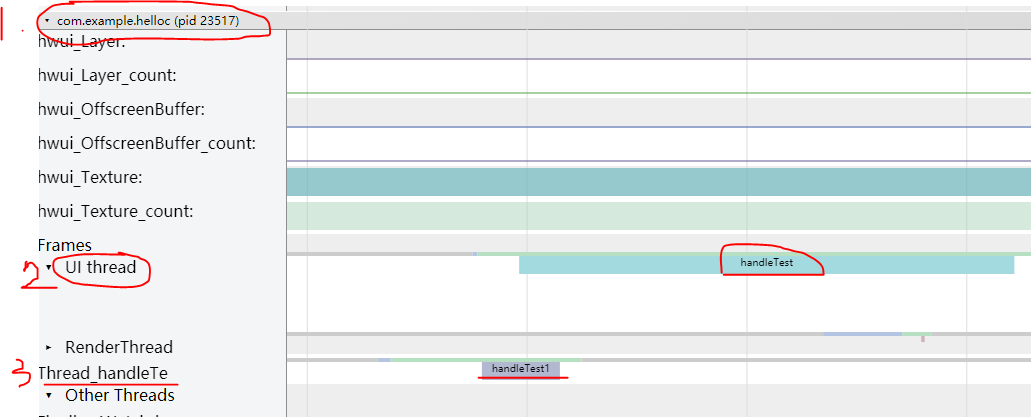
从图2-19可以看到，Alerts一栏标记了性能有问题的点，单击该点可以查看详细信息，在右边侧边栏还有一个Alerts框，单击可以查看每个类型的Alerts的数量，单击某一个Alert可以看到问题的详细描述。

（2）Frame

每个应用都有一行专门显示frame，每一帧就显示为一个绿色的圆圈。当显示为黄色或者红色时，它的渲染时间超过了16.6ms（即达不到60fps的水准）。使用W键放大，看看这一帧的渲染过程中系统到底做了什么，同时它会将任何它认为性能有问题的东西都高亮警告，并提示要怎么优化。如图2-19所示，在Frame栏有一个F帧（第二帧）黄色告警，从下面的问题详细描述可以看出，警告的主要原因是ListView的回收和重新绑定花费太多时间。在Systrace中也会提供一些对应链接，提供更多解释。

如果想知道UI线程怎么会花费这么多时间的话，就需要使用2.2.2节讲到的TraceView，来分析具体是哪些函数在消耗时间。

注意，先找到对应的进程1，然后找线程2和3，最后



对应的代码片段是

@Override  
**protected void** onCreate(Bundle savedInstanceState) {  
 **super**.onCreate(savedInstanceState);  
 setContentView(R.layout.***activity\_main***);  
  
 *// Example of a call to a native method* TextView tv = findViewById(R.id.***sample\_text***);  
 tv.setText(stringFromJNI());  
 **mH**.sendEmptyMessage(0);  
 Thread thread = **new** Thread(**"Thread\_handleTest1"**){  
 @Override  
 **public void** run() {  
 **while** (**true**){  
 handleTest1();  
 **try** {*sleep*(133); }**catch** (Exception ex){ }  
 }  
 }  
 };  
 thread.start();  
}  
  
Handler **mH** = **new** Handler(){  
 @Override  
 **public void** handleMessage(Message msg) {  
 handleTest0();  
 **mH**.sendEmptyMessageDelayed(0, 1000);  
 }  
};  
**int cnt** = 0;  
**private void** handleTest0() {  
 Trace.*beginSection*(**"handleTest"**);  
 ((TextView)findViewById(R.id.***textView***)).setText(**"cnt="**+ **cnt**++ );  
 Log.*d*(**"keytest"**,**"handleTest"**);  
 Trace.*endSection*();  
}  
  
**int cnt1** = 0;  
**private void** handleTest1() {  
 Trace.*beginSection*(**"handleTest1"**);  
 Log.*d*(**"keytest"**,**"handleTest1 = "**+**cnt1**++);  
 Trace.*endSection*();  
}

## Trace底层原理

system/core/libcutils/trace-dev.c

使用方法

#include<utils\Trace.h>

bool SurfaceFlinger::handleMessageInvalidate() {

ATRACE\_CALL();

return handlePageFlip();

}

**ATRACE\_CALL()**为开始标记，这就开始trace该函数了，这个宏定义在 \system\core\include\utils\Trace.h 中

### ATRACE\_CALL()展开

#define ATRACE\_NAME(name) android::ScopedTrace \_\_\_tracer(ATRACE\_TAG, name)  
*// ATRACE\_CALL is an ATRACE\_NAME that uses the current function name.*#define ATRACE\_CALL() ATRACE\_NAME(\_\_FUNCTION\_\_)  
  
**namespace** android {  
  
**class** ScopedTrace {  
**public**:  
**inline** ScopedTrace(uint64\_t tag, **const char**\* name)  
 : mTag(tag) {  
 atrace\_begin(mTag,name);  
}  
  
**inline** ~ScopedTrace() {  
 atrace\_end(mTag);  
}  
  
**private**:  
 uint64\_t mTag;  
};  
  
};

最终展开android::ScopedTrace \_\_\_tracer(ATRACE\_TAG, “handleMessageInvalidate”)

### TAG类型

定义在 \system\core\include\cutils\trace.h 中：

看下TAG类型：

#define ATRACE\_TAG\_NEVER 0 *// This tag is never enabled.*#define ATRACE\_TAG\_ALWAYS (1<<0) *// This tag is always enabled.*#define ATRACE\_TAG\_GRAPHICS (1<<1)  
#define ATRACE\_TAG\_INPUT (1<<2)  
#define ATRACE\_TAG\_VIEW (1<<3)  
#define ATRACE\_TAG\_WEBVIEW (1<<4)  
#define ATRACE\_TAG\_WINDOW\_MANAGER (1<<5)  
#define ATRACE\_TAG\_ACTIVITY\_MANAGER (1<<6)  
#define ATRACE\_TAG\_SYNC\_MANAGER (1<<7)  
#define ATRACE\_TAG\_AUDIO (1<<8)  
#define ATRACE\_TAG\_VIDEO (1<<9)  
#define ATRACE\_TAG\_CAMERA (1<<10)  
#define ATRACE\_TAG\_HAL (1<<11)  
#define ATRACE\_TAG\_APP (1<<12)  
#define ATRACE\_TAG\_RESOURCES (1<<13)  
#define ATRACE\_TAG\_DALVIK (1<<14)  
#define ATRACE\_TAG\_RS (1<<15)  
#define ATRACE\_TAG\_BIONIC (1<<16)  
#define ATRACE\_TAG\_POWER (1<<17)  
#define ATRACE\_TAG\_PACKAGE\_MANAGER (1<<18)  
#define ATRACE\_TAG\_SYSTEM\_SERVER (1<<19)  
#define ATRACE\_TAG\_DATABASE (1<<20)  
#define ATRACE\_TAG\_NETWORK (1<<21)  
#define ATRACE\_TAG\_LAST ATRACE\_TAG\_NETWORK

frameworks\base\core\java\android\os\Trace.java

也有java版的定义

**public final class** Trace {  
 */\*  
 \* Writes trace events to the kernel trace buffer. These trace events can be  
 \* collected using the "atrace" program for offline analysis.  
 \*/* **private static final** String ***TAG*** = **"Trace"**;  
  
 *// These tags must be kept in sync with system/core/include/cutils/trace.h.  
 // They should also be added to frameworks/native/cmds/atrace/atrace.cpp.  
 /\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_NEVER*** = 0;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_ALWAYS*** = 1L << 0;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_GRAPHICS*** = 1L << 1;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_INPUT*** = 1L << 2;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_VIEW*** = 1L << 3;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_WEBVIEW*** = 1L << 4;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_WINDOW\_MANAGER*** = 1L << 5;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_ACTIVITY\_MANAGER*** = 1L << 6;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_SYNC\_MANAGER*** = 1L << 7;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_AUDIO*** = 1L << 8;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_VIDEO*** = 1L << 9;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_CAMERA*** = 1L << 10;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_HAL*** = 1L << 11;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_APP*** = 1L << 12;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_RESOURCES*** = 1L << 13;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_DALVIK*** = 1L << 14;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_RS*** = 1L << 15;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_BIONIC*** = 1L << 16;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_POWER*** = 1L << 17;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_PACKAGE\_MANAGER*** = 1L << 18;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_SYSTEM\_SERVER*** = 1L << 19;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_DATABASE*** = 1L << 20;  
 */\*\** ***@hide*** *\*/* **public static final long *TRACE\_TAG\_NETWORK*** = 1L << 21;

其中上面在SurfaceFlinger中定义成了 **ATRACE\_TAG\_GRAPHICS**

### atrace\_begin

system/core/include/cutils/trace.h

首先会去check 当前来的tag 是否 enable，会进行判断去初始化：

*/\*\*  
 \* Trace the beginning of a context. name is used to identify the context.  
 \* This is often used to time function execution.  
 \*/*#define ATRACE\_BEGIN(name) atrace\_begin(ATRACE\_TAG, name)  
**static inline void** atrace\_begin(uint64\_t tag, **const char**\* name)  
{  
 **if** (CC\_UNLIKELY(atrace\_is\_tag\_enabled(tag))) {  
 **void** atrace\_begin\_body(**const char**\*);  
 atrace\_begin\_body(name);  
 }  
}  
  
*/\*\*  
 \* Trace the end of a context.  
 \* This should match up (and occur after) a corresponding ATRACE\_BEGIN.  
 \*/*#define ATRACE\_END() atrace\_end(ATRACE\_TAG)  
**static inline void** atrace\_end(uint64\_t tag)  
{  
 **if** (CC\_UNLIKELY(atrace\_is\_tag\_enabled(tag))) {  
 **char** c = **'E'**;  
 write(atrace\_marker\_fd, &c, 1);  
 }  
}

#### atrace\_is\_tag\_enabled

#define ATRACE\_ENABLED() atrace\_is\_tag\_enabled(ATRACE\_TAG)  
**static inline** uint64\_t atrace\_is\_tag\_enabled(uint64\_t tag)  
{  
 **return** atrace\_get\_enabled\_tags() & tag;  
}

#define ATRACE\_GET\_ENABLED\_TAGS() atrace\_get\_enabled\_tags()  
**static inline** uint64\_t atrace\_get\_enabled\_tags()  
{  
 atrace\_init();  
 **return** atrace\_enabled\_tags;  
}

#define ATRACE\_INIT() atrace\_init()  
**static inline void** atrace\_init()  
{  
 **if** (CC\_UNLIKELY(!atomic\_load\_explicit(&atrace\_is\_ready, memory\_order\_acquire))) {  
 atrace\_setup();  
 }  
}

### atrace\_setup()

system/core/libcutils/trace-dev.c

**void** atrace\_setup()  
{  
 pthread\_once(&atrace\_once\_control, atrace\_init\_once);  
}

通常当初始化应用程序时，可以比较容易地将其放在main函数中。但当你写一个库函数时，就不能在main里面初始化了，你可以用静态初始化，但使用一次初始（pthread\_once\_t）会比较容易些。<https://www.jianshu.com/p/a69745fc0a44>

定义一个pthread\_once\_t变量atrace\_once\_control就ok了

#### atrace\_init\_once

**static void** atrace\_init\_once()  
{  
 atrace\_marker\_fd = open(**"/sys/kernel/debug/tracing/trace\_marker"**, O\_WRONLY | O\_CLOEXEC);  
 **if** (atrace\_marker\_fd == -1) {  
 ALOGE(**"Error opening trace file: %s (%d)"**, strerror(errno), errno);  
 atrace\_enabled\_tags = 0;  
 **goto** done;  
 }  
  
 atrace\_enabled\_tags = atrace\_get\_property();  
  
done:  
 atomic\_store\_explicit(&atrace\_is\_ready, **true**, memory\_order\_release);  
}

#### atrace\_get\_property

一般默认为

rm500:/ $ getprop debug.atrace.tags.enableflags

0

执行监控的时候

python systrace.py --time=10 -o mynewtrace.html gfx view wm sched

gj500:/ $ getprop debug.atrace.tags.enableflags

0x2a=0010 1010

#define ATRACE\_TAG\_GRAPHICS (1<<1)

#define ATRACE\_TAG\_VIEW (1<<3)

#define ATRACE\_TAG\_WINDOW\_MANAGER (1<<5)

#define ATRACE\_TAG\_SYNC\_MANAGER (1<<7)

*// Read the sysprop and return the value tags should be set to***static** uint64\_t atrace\_get\_property()  
{  
 **char** value[PROPERTY\_VALUE\_MAX];  
 **char** \*endptr;  
 uint64\_t tags;  
  
 property\_get(**"debug.atrace.tags.enableflags"**, value, **"0"**);  
 errno = 0;  
 tags = strtoull(value, &endptr, 0);  
 **if** (value[0] == **'\0'** || \*endptr != **'\0'**) {  
 ALOGE(**"Error parsing trace property: Not a number: %s"**, value);  
 **return** 0;  
 } **else if** (errno == ERANGE || tags == ULLONG\_MAX) {  
 ALOGE(**"Error parsing trace property: Number too large: %s"**, value);  
 **return** 0;  
 }  
  
 *// Only set the "app" tag if this process was selected for app-level debug  
 // tracing.* **if** (atrace\_is\_app\_tracing\_enabled()) {  
 tags |= ATRACE\_TAG\_APP;  
 } **else** {  
 tags &= ~ATRACE\_TAG\_APP;  
 }  
  
 **return** (tags | ATRACE\_TAG\_ALWAYS) & ATRACE\_TAG\_VALID\_MASK;  
}

/sys/kernel/debug/tracing/trace\_marker 是重点，为trace在kernel里面创建的文件节点

打开之后保存全局的文件描述符：atrace\_marker\_fd

现在回头看atrace\_begin 和 atrace\_end 就是向这个节点write 对应要打开的trace的开关数据

kernel中的trace驱动自然会有file operation处理

### java层->native

g

**private void** handleTest() {  
 Trace.*beginSection*(**"handleTest"**);  
 Log.*d*(**"keytest"**,**"handleTest"**);  
 Trace.*endSection*();  
}

#### java

**public static void** beginSection(String sectionName) {  
 **if** (isTagEnabled(TRACE\_TAG\_APP)) {  
 **if** (sectionName.length() > MAX\_SECTION\_NAME\_LEN) {  
 **throw new** IllegalArgumentException(**"sectionName is too long"**);  
 }  
 nativeTraceBegin(TRACE\_TAG\_APP, sectionName);  
 }  
}

**public static void** endSection() {  
 **if** (isTagEnabled(TRACE\_TAG\_APP)) {  
 nativeTraceEnd(TRACE\_TAG\_APP);  
 }  
}

*nativeTraceBegin*(***TRACE\_TAG\_APP***, sectionName);

#### jni

frameworks/base/core/jni/android\_os\_Trace.cpp

**static void** android\_os\_Trace\_nativeTraceBegin(JNIEnv\* env, jclass clazz,  
 jlong tag, jstring nameStr) {  
 ALOGV(**"%s: %"** PRId64 **" %s"**, \_\_FUNCTION\_\_, tag, utf8Chars.string());  
 atrace\_begin(tag, utf8Chars.string());  
}

### 参考

<https://blog.csdn.net/jscese/article/details/48373319>

## ATRACE\_INT

system/core/include/cutils/trace.h

*/\*\*  
 \* Traces an integer counter value. name is used to identify the counter.  
 \* This can be used to track how a value changes over time.  
 \*/*#define ATRACE\_INT(name, value) atrace\_int(ATRACE\_TAG, name, value)  
**static inline void** atrace\_int(uint64\_t tag, **const char**\* name, int32\_t value)  
{  
 **if** (CC\_UNLIKELY(atrace\_is\_tag\_enabled(tag))) {  
 **void** atrace\_int\_body(**const char**\*, int32\_t);  
 atrace\_int\_body(name, value);  
 }  
}

## cmd剖析

### 参数定义

内部k\_categories定义了具体的参数值，忽然明白了python systrace.py --time=10 -o mynewtrace.html sched gfx view wm类别参数的含义，如果记不住怎么办呢利用help工具吧atrace -d错误参数触发一下

255|marlin:/ # atrace -d

atrace: invalid option -- d

**static const** TracingCategory k\_categories[] = {  
 { **"gfx"**, **"Graphics"**, ATRACE\_TAG\_GRAPHICS, {  
 { OPT, **"events/mdss/enable"** },  
 { OPT, **"events/sde/enable"** },  
 } },  
 { **"input"**, **"Input"**, ATRACE\_TAG\_INPUT, { } },  
 { **"view"**, **"View System"**, ATRACE\_TAG\_VIEW, { } },  
 { **"webview"**, **"WebView"**, ATRACE\_TAG\_WEBVIEW, { } },  
 { **"wm"**, **"Window Manager"**, ATRACE\_TAG\_WINDOW\_MANAGER, { } },  
 { **"am"**, **"Activity Manager"**, ATRACE\_TAG\_ACTIVITY\_MANAGER, { } },  
 { **"sm"**, **"Sync Manager"**, ATRACE\_TAG\_SYNC\_MANAGER, { } },  
 { **"audio"**, **"Audio"**, ATRACE\_TAG\_AUDIO, { } },  
 { **"video"**, **"Video"**, ATRACE\_TAG\_VIDEO, { } },  
 { **"camera"**, **"Camera"**, ATRACE\_TAG\_CAMERA, { } },  
 { **"hal"**, **"Hardware Modules"**, ATRACE\_TAG\_HAL, { } },  
 { **"res"**, **"Resource Loading"**, ATRACE\_TAG\_RESOURCES, { } },  
 { **"dalvik"**, **"Dalvik VM"**, ATRACE\_TAG\_DALVIK, { } },  
 { **"rs"**, **"RenderScript"**, ATRACE\_TAG\_RS, { } },  
 { **"bionic"**, **"Bionic C Library"**, ATRACE\_TAG\_BIONIC, { } },  
 { **"power"**, **"Power Management"**, ATRACE\_TAG\_POWER, { } },  
 { **"pm"**, **"Package Manager"**, ATRACE\_TAG\_PACKAGE\_MANAGER, { } },  
 { **"ss"**, **"System Server"**, ATRACE\_TAG\_SYSTEM\_SERVER, { } },  
 { **"database"**, **"Database"**, ATRACE\_TAG\_DATABASE, { } },  
 { **"network"**, **"Network"**, ATRACE\_TAG\_NETWORK, { } },  
 { **"adb"**, **"ADB"**, ATRACE\_TAG\_ADB, { } },  
 { **"vibrator"**, **"Vibrator"**, ATRACE\_TAG\_VIBRATOR, { } },  
 { **"aidl"**, **"AIDL calls"**, ATRACE\_TAG\_AIDL, { } },  
 { k\_coreServiceCategory, **"Core services"**, 0, { } },  
 { k\_pdxServiceCategory, **"PDX services"**, 0, { } },  
 { **"sched"**, **"CPU Scheduling"**, 0, {  
 { REQ, **"events/sched/sched\_switch/enable"** },  
 { REQ, **"events/sched/sched\_wakeup/enable"** },  
 { OPT, **"events/sched/sched\_waking/enable"** },  
 { OPT, **"events/sched/sched\_blocked\_reason/enable"** },  
 { OPT, **"events/sched/sched\_cpu\_hotplug/enable"** },  
 { OPT, **"events/sched/sched\_pi\_setprio/enable"** },  
 { OPT, **"events/cgroup/enable"** },  
 } },  
 { **"irq"**, **"IRQ Events"**, 0, {  
 { REQ, **"events/irq/enable"** },  
 { OPT, **"events/ipi/enable"** },  
 } },  
 { **"irqoff"**, **"IRQ-disabled code section tracing"**, 0, {  
 { REQ, **"events/preemptirq/irq\_enable/enable"** },  
 { REQ, **"events/preemptirq/irq\_disable/enable"** },  
 } },  
 { **"preemptoff"**, **"Preempt-disabled code section tracing"**, 0, {  
 { REQ, **"events/preemptirq/preempt\_enable/enable"** },  
 { REQ, **"events/preemptirq/preempt\_disable/enable"** },  
 } },  
 { **"i2c"**, **"I2C Events"**, 0, {  
 { REQ, **"events/i2c/enable"** },  
 { REQ, **"events/i2c/i2c\_read/enable"** },  
 { REQ, **"events/i2c/i2c\_write/enable"** },  
 { REQ, **"events/i2c/i2c\_result/enable"** },  
 { REQ, **"events/i2c/i2c\_reply/enable"** },  
 { OPT, **"events/i2c/smbus\_read/enable"** },  
 { OPT, **"events/i2c/smbus\_write/enable"** },  
 { OPT, **"events/i2c/smbus\_result/enable"** },  
 { OPT, **"events/i2c/smbus\_reply/enable"** },  
 } },  
 { **"freq"**, **"CPU Frequency"**, 0, {  
 { REQ, **"events/power/cpu\_frequency/enable"** },  
 { OPT, **"events/power/clock\_set\_rate/enable"** },  
 { OPT, **"events/power/clock\_disable/enable"** },  
 { OPT, **"events/power/clock\_enable/enable"** },  
 { OPT, **"events/clk/clk\_set\_rate/enable"** },  
 { OPT, **"events/clk/clk\_disable/enable"** },  
 { OPT, **"events/clk/clk\_enable/enable"** },  
 { OPT, **"events/power/cpu\_frequency\_limits/enable"** },  
 } },  
 { **"membus"**, **"Memory Bus Utilization"**, 0, {  
 { REQ, **"events/memory\_bus/enable"** },  
 } },  
 { **"idle"**, **"CPU Idle"**, 0, {  
 { REQ, **"events/power/cpu\_idle/enable"** },  
 } },  
 { **"disk"**, **"Disk I/O"**, 0, {  
 { OPT, **"events/f2fs/f2fs\_sync\_file\_enter/enable"** },  
 { OPT, **"events/f2fs/f2fs\_sync\_file\_exit/enable"** },  
 { OPT, **"events/f2fs/f2fs\_write\_begin/enable"** },  
 { OPT, **"events/f2fs/f2fs\_write\_end/enable"** },  
 { OPT, **"events/ext4/ext4\_da\_write\_begin/enable"** },  
 { OPT, **"events/ext4/ext4\_da\_write\_end/enable"** },  
 { OPT, **"events/ext4/ext4\_sync\_file\_enter/enable"** },  
 { OPT, **"events/ext4/ext4\_sync\_file\_exit/enable"** },  
 { REQ, **"events/block/block\_rq\_issue/enable"** },  
 { REQ, **"events/block/block\_rq\_complete/enable"** },  
 } },  
 { **"mmc"**, **"eMMC commands"**, 0, {  
 { REQ, **"events/mmc/enable"** },  
 } },  
 { **"load"**, **"CPU Load"**, 0, {  
 { REQ, **"events/cpufreq\_interactive/enable"** },  
 } },  
 { **"sync"**, **"Synchronization"**, 0, {  
 *// before linux kernel 4.9* { OPT, **"events/sync/enable"** },  
 *// starting in linux kernel 4.9* { OPT, **"events/fence/enable"** },  
 } },  
 { **"workq"**, **"Kernel Workqueues"**, 0, {  
 { REQ, **"events/workqueue/enable"** },  
 } },  
 { **"memreclaim"**, **"Kernel Memory Reclaim"**, 0, {  
 { REQ, **"events/vmscan/mm\_vmscan\_direct\_reclaim\_begin/enable"** },  
 { REQ, **"events/vmscan/mm\_vmscan\_direct\_reclaim\_end/enable"** },  
 { REQ, **"events/vmscan/mm\_vmscan\_kswapd\_wake/enable"** },  
 { REQ, **"events/vmscan/mm\_vmscan\_kswapd\_sleep/enable"** },  
 { OPT, **"events/lowmemorykiller/enable"** },  
 } },  
 { **"regulators"**, **"Voltage and Current Regulators"**, 0, {  
 { REQ, **"events/regulator/enable"** },  
 } },  
 { **"binder\_driver"**, **"Binder Kernel driver"**, 0, {  
 { REQ, **"events/binder/binder\_transaction/enable"** },  
 { REQ, **"events/binder/binder\_transaction\_received/enable"** },  
 { OPT, **"events/binder/binder\_set\_priority/enable"** },  
 } },  
 { **"binder\_lock"**, **"Binder global lock trace"**, 0, {  
 { OPT, **"events/binder/binder\_lock/enable"** },  
 { OPT, **"events/binder/binder\_locked/enable"** },  
 { OPT, **"events/binder/binder\_unlock/enable"** },  
 } },  
 { **"pagecache"**, **"Page cache"**, 0, {  
 { REQ, **"events/filemap/enable"** },  
 } },  
};

## QA

### 如何抓取应用层

### 可否不依赖py和chrome

不能：py有很复杂的转换，保证chrome可以识别到

Chrome本身就提供了丰富的交互控制

py方法

python systrace.py --time=10 -o mynewtrace.html sched gfx view wm

adb shell atrace -t 10 app > trace.html

adb shell atrace -t=10 -o mynewtrace.html sched gfx view wm

adb root && adb remount && adb shell cat /d/tracing/trace >> trance.txt

android-7.1.1\_r1/external/chromium-trace/catapult/tracing/tracing\_build/trace2html.py

trace2html.py trance.txt –output trance.html

s\native

\native\cmds\atrace\atrace.rc

# Tracing disabled by default

write /sys/kernel/debug/tracing/tracing\_on 0

下载AS，并在其tools目录下找到monitor.bat，打开monitor.bat

使用Android Studio抓取systrace，转成html

<https://jingyan.baidu.com/article/fdffd1f86dc77df3e88ca149.html>

# bugreport

## 概述

通过adb命令可获取bugrepport信息，并输出到文件当前路径的bugreport.txt文件：

adb bugreport > bugreport.txt

对于Android系统调试分析，bugreport信息量非常之大，几乎涵盖整个系统各个层面内容，对于分析BUG是一大利器，本文先从从源码角度来分析一下Bugreport的实现原理。

## 原理分析

Android系统源码中framework/native/cmds/bugreport目录通过Android.mk定义了bugreport项目，在系统编译完成后会生成bugreport可执行文件，位于系统/system/bin/bugreport。当执行adb bugreport时，便会调用这个可执行文件，进入bugreport.cpp中的main()方法。

### Mk

LOCAL\_SRC\_FILES:= bugreport.cpp

LOCAL\_MODULE:= bugreport

LOCAL\_CFLAGS := -Wall

LOCAL\_SHARED\_LIBRARIES := libcutils

### bugreport.main

bugreport.cpp

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **int** main() {
2. //启动dumpstate服务
3. property\_set("ctl.start", "dumpstate");
4. //需要多次尝试，直到dumpstate服务启动完成，才能建立socket通信
5. **int** s;
6. **for** (**int** i = 0; i < 20; i++) {
7. s = socket\_local\_client("dumpstate", ANDROID\_SOCKET\_NAMESPACE\_RESERVED,
8. SOCK\_STREAM);
9. **if** (s >= 0)
10. **break**;
11. //休眠1s后再次尝试连接
12. sleep(1);
13. }
14. **if** (s == -1) {
15. printf("Failed to connect to dumpstate service: %s\n", strerror(errno));
16. **return** 1;
17. }
18. //当3分钟没有任何数据可读，则超时停止读取并退出。
19. //dumpstate服务中不存在大于1分钟的timetout，因而不可预见的超时的情况下留有很大的回旋余地。
20. struct timeval tv;
21. tv.tv\_sec = 3 \* 60;
22. tv.tv\_usec = 0;
23. **if** (setsockopt(s, SOL\_SOCKET, SO\_RCVTIMEO, &tv, sizeof(tv)) == -1) {
24. printf("WARNING: Cannot set socket timeout: %s\n", strerror(errno));
25. }
26. **while** (1) {
27. **char** buffer[65536];
28. ssize\_t bytes\_read = TEMP\_FAILURE\_RETRY(read(s, buffer, sizeof(buffer)));
29. **if** (bytes\_read == 0) {
30. **break**;
31. } **else** **if** (bytes\_read == -1) {
32. // EAGAIN意味着timeout，Bugreport读异常终止
33. **if** (errno == EAGAIN) {
34. errno = ETIMEDOUT;
35. }
36. **break**;
37. }
38. ssize\_t bytes\_to\_send = bytes\_read;
39. ssize\_t bytes\_written;
40. //不断循环得将读取数据输出到stdout
41. **do** {
42. bytes\_written = TEMP\_FAILURE\_RETRY(write(STDOUT\_FILENO,
43. buffer + bytes\_read - bytes\_to\_send, bytes\_to\_send));
44. **if** (bytes\_written == -1) {
45. **return** 1; //将数据无法写入stdout
46. }
47. bytes\_to\_send -= bytes\_written;
48. } **while** (bytes\_written != 0 && bytes\_to\_send > 0);
49. }
50. close(s);
51. **return** 0;
52. }

property\_set(“ctl.start”, “dumpstate”)会触发init进程,来fork进程/system/bin/dumpstate, 作为dumpstate服务的进程. Bugreport再通过socket建立于dumpstate的通信，这个过程会尝试20次socket连接建立直到成功连接。 在socket通道中如果持续3分钟没有任何数据可读，则超时停止读取并退出。由于dumpstate服务中不存在大于1分钟的timetout，因而不可预见的超时的情况下留有很大的回旋余地。

当从socket读取到数据后，写入到标准时输出或者重定向到文件。可见bugreport数据的来源都是dumpstate服务，那么接下来去看看dumpstate服务的工作。

## Dumpstate

Sgf

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **int** main(**int** argc, **char** \*argv[]) {
2. struct sigaction sigact;
3. **int** do\_add\_date = 0;
4. **int** do\_vibrate = 1;
5. **char**\* use\_outfile = 0;
6. **int** use\_socket = 0;
7. **int** do\_fb = 0;
8. **int** do\_broadcast = 0;
9. **if** (getuid() != 0) {
10. //兼容性考虑，旧版本支持直接调用dumpstate命令，新版本通过调用/system/bin/bugreport来替代。
11. //当检测到直接调用，则强制执行bugreport命令。
12. **return** execl("/system/bin/bugreport", "/system/bin/bugreport", NULL);
13. }
14. ALOGI("begin\n");
15. //清空句柄SIGPIPE
16. memset(&sigact, 0, sizeof(sigact));
17. sigact.sa\_handler = sigpipe\_handler;
18. sigaction(SIGPIPE, &sigact, NULL);
19. //提高当前进程的优先级，防止被OOM Killer杀死
20. setpriority(PRIO\_PROCESS, 0, -20);
21. FILE \*oom\_adj = fopen("/proc/self/oom\_adj", "we");
22. **if** (oom\_adj) {
23. fputs("-17", oom\_adj);
24. fclose(oom\_adj);
25. }
26. //参数解析
27. **int** c;
28. **while** ((c = getopt(argc, argv, "dho:svqzpB")) != -1) {
29. **switch** (c) {
30. **case** 'd': do\_add\_date = 1;       **break**;
31. **case** 'o': use\_outfile = optarg;  **break**;
32. **case** 's': use\_socket = 1;        **break**;
33. **case** 'v': **break**;  // compatibility no-op
34. **case** 'q': do\_vibrate = 0;        **break**;
35. **case** 'p': do\_fb = 1;             **break**;
36. **case** 'B': do\_broadcast = 1;      **break**;
37. **case** '?': printf("\n");
38. **case** 'h':
39. usage();
40. exit(1);
41. }
42. }
43. //建立socket
44. **if** (use\_socket) {
45. redirect\_to\_socket(stdout, "dumpstate");
46. }
47. //打开vibrator
48. FILE \*vibrator = 0;
49. **if** (do\_vibrate) {
50. vibrator = fopen("/sys/class/timed\_output/vibrator/enable", "we");
51. **if** (vibrator) {
52. vibrate(vibrator, 150);
53. }
54. }
55. //读取/proc/cmdline
56. FILE \*cmdline = fopen("/proc/cmdline", "re");
57. **if** (cmdline != NULL) {
58. fgets(cmdline\_buf, sizeof(cmdline\_buf), cmdline);
59. fclose(cmdline);
60. }
61. //收集虚拟机和native进程的stack traces(需要root权限)
62. dump\_traces\_path = dump\_traces();
63. //获取tombstone文件描述符
64. get\_tombstone\_fds(tombstone\_data);
65. //确保capabilities
66. **if** (prctl(PR\_SET\_KEEPCAPS, 1) < 0) {
67. ALOGE("prctl(PR\_SET\_KEEPCAPS) failed: %s\n", strerror(errno));
68. **return** -1;
69. }
70. //切换到非root用户和组，在切换之前都是处于root权限
71. gid\_t groups[] = { AID\_LOG, AID\_SDCARD\_R, AID\_SDCARD\_RW,
72. AID\_MOUNT, AID\_INET, AID\_NET\_BW\_STATS };
73. **if** (setgroups(sizeof(groups)/sizeof(groups[0]), groups) != 0) {
74. ALOGE("Unable to setgroups, aborting: %s\n", strerror(errno));
75. **return** -1;
76. }
77. **if** (setgid(AID\_SHELL) != 0) {
78. ALOGE("Unable to setgid, aborting: %s\n", strerror(errno));
79. **return** -1;
80. }
81. **if** (setuid(AID\_SHELL) != 0) {
82. ALOGE("Unable to setuid, aborting: %s\n", strerror(errno));
83. **return** -1;
84. }
85. struct \_\_user\_cap\_header\_struct capheader;
86. struct \_\_user\_cap\_data\_struct capdata[2];
87. memset(&capheader, 0, sizeof(capheader));
88. memset(&capdata, 0, sizeof(capdata));
89. capheader.version = \_LINUX\_CAPABILITY\_VERSION\_3;
90. capheader.pid = 0;
91. capdata[CAP\_TO\_INDEX(CAP\_SYSLOG)].permitted = CAP\_TO\_MASK(CAP\_SYSLOG);
92. capdata[CAP\_TO\_INDEX(CAP\_SYSLOG)].effective = CAP\_TO\_MASK(CAP\_SYSLOG);
93. capdata[0].inheritable = 0;
94. capdata[1].inheritable = 0;
95. **if** (capset(&capheader, &capdata[0]) < 0) {
96. ALOGE("capset failed: %s\n", strerror(errno));
97. **return** -1;
98. }
99. //如果需要，则重定向输出
100. **char** path[PATH\_MAX], tmp\_path[PATH\_MAX];
101. pid\_t gzip\_pid = -1;
102. **if** (!use\_socket && use\_outfile) {
103. strlcpy(path, use\_outfile, sizeof(path));
104. **if** (do\_add\_date) {
105. **char** date[80];
106. time\_t now = time(NULL);
107. strftime(date, sizeof(date), "-%Y-%m-%d-%H-%M-%S", localtime(&now));
108. strlcat(path, date, sizeof(path));
109. }
110. **if** (do\_fb) {
111. strlcpy(screenshot\_path, path, sizeof(screenshot\_path));
112. strlcat(screenshot\_path, ".png", sizeof(screenshot\_path));
113. }
114. strlcat(path, ".txt", sizeof(path));
115. strlcpy(tmp\_path, path, sizeof(tmp\_path));
116. strlcat(tmp\_path, ".tmp", sizeof(tmp\_path));
117. redirect\_to\_file(stdout, tmp\_path);
118. }
119. //这里是真正干活的地方 【见小节 2.3】
120. dumpstate();
121. //通过震动提醒已完成所有dump操作
122. **if** (vibrator) {
123. **for** (**int** i = 0; i < 3; i++) {
124. vibrate(vibrator, 75);
125. usleep((75 + 50) \* 1000);
126. }
127. fclose(vibrator);
128. }
129. //等待gzip的完成，等进程退出时则会被杀
130. **if** (gzip\_pid > 0) {
131. fclose(stdout);
132. waitpid(gzip\_pid, NULL, 0);
133. }
134. //重命名.tmp文件到最终位置
135. **if** (use\_outfile && rename(tmp\_path, path)) {
136. fprintf(stderr, "rename(%s, %s): %s\n", tmp\_path, path, strerror(errno));
137. }
138. //通过发送广播告知ActivityManager已完成bugreport操作
139. **if** (do\_broadcast && use\_outfile && do\_fb) {
140. run\_command(NULL, 5, "/system/bin/am", "broadcast", "--user", "0",
141. "-a", "android.intent.action.BUGREPORT\_FINISHED",
142. "--es", "android.intent.extra.BUGREPORT", path,
143. "--es", "android.intent.extra.SCREENSHOT", screenshot\_path,
144. "--receiver-permission", "android.permission.DUMP", NULL);
145. }
146. ALOGI("done\n");
147. **return** 0;
148. }

整个过程的工作流程：

1. 提高执行dumpsate所在进程的优先级，防止被OOM Killer杀死；
2. 参数解析，可通过命令adb shell dumpstate -h查看dumpstate命令所支持的参数；
3. 打开vibrator，用于在执行bugreport时，手机会先震动一下用于提醒开始抓取系统信息；
4. 通过dump\_traces()来完成收集虚拟机和native进程的stack traces；
5. 通过get\_tombstone\_fds来获取tombstone文件描述符；
6. 开始执行切换到非root用户和组，在这之前的执行都处于root权限；
7. **执行dumpstate()，这里是真正干活的地方**；
8. 再次通过震动以提醒dump操作执行完成；
9. 发送广播，告知ActivityManager已完成bugreport操作。

接下来就重点说说dumpstate()功能：

#### dumpstate()

afds

**[java]** [view plain](http://blog.csdn.net/qq1028850792/article/details/79346153) [copy](http://blog.csdn.net/qq1028850792/article/details/79346153)

1. **static** **void** dumpstate() {
2. ...
3. property\_get("ro.build.display.id", build, "(unknown)");
4. property\_get("ro.build.fingerprint", fingerprint, "(unknown)");
5. property\_get("ro.build.type", build\_type, "(unknown)");
6. property\_get("ro.baseband", radio, "(unknown)");
7. property\_get("ro.bootloader", bootloader, "(unknown)");
8. property\_get("gsm.operator.alpha", network, "(unknown)");
9. strftime(date, sizeof(date), "%Y-%m-%d %H:%M:%S", localtime(&now));
10. //开头信息
11. printf("========================================================\n");
12. printf("== dumpstate: %s\n", date);
13. printf("========================================================\n");
14. printf("\n");
15. printf("Build: %s\n", build);
16. printf("Build fingerprint: '%s'\n", fingerprint);
17. printf("Bootloader: %s\n", bootloader);
18. printf("Radio: %s\n", radio);
19. printf("Network: %s\n", network);
20. printf("Kernel: "); dump\_file(NULL, "/proc/version");
21. printf("Command line: %s\n", strtok(cmdline\_buf, "\n"));
22. printf("\n");
23. //记录系统运行时长和休眠时长
24. run\_command("UPTIME", 10, "uptime", NULL);
26. //输出mmcblk0设备信息
27. dump\_files("UPTIME MMC PERF", mmcblk0, skip\_not\_stat, dump\_stat\_from\_fd);
29. dump\_file("MEMORY INFO", "/proc/meminfo");
30. run\_command("CPU INFO", 10, "top", "-n", "1", "-d", "1", "-m", "30", "-t", NULL);
31. run\_command("PROCRANK", 20, "procrank", NULL);
32. dump\_file("VIRTUAL MEMORY STATS", "/proc/vmstat");
33. dump\_file("VMALLOC INFO", "/proc/vmallocinfo");
34. dump\_file("SLAB INFO", "/proc/slabinfo");
35. dump\_file("ZONEINFO", "/proc/zoneinfo");
36. dump\_file("PAGETYPEINFO", "/proc/pagetypeinfo");
37. dump\_file("BUDDYINFO", "/proc/buddyinfo");
38. dump\_file("FRAGMENTATION INFO", "/d/extfrag/unusable\_index");
39. dump\_file("KERNEL WAKELOCKS", "/proc/wakelocks");
40. dump\_file("KERNEL WAKE SOURCES", "/d/wakeup\_sources");
41. dump\_file("KERNEL CPUFREQ", "/sys/devices/system/cpu/cpu0/cpufreq/stats/time\_in\_state");
42. dump\_file("KERNEL SYNC", "/d/sync");
43. run\_command("PROCESSES", 10, "ps", "-P", NULL);
44. run\_command("PROCESSES AND THREADS", 10, "ps", "-t", "-p", "-P", NULL);
45. run\_command("PROCESSES (SELINUX LABELS)", 10, "ps", "-Z", NULL);
46. run\_command("LIBRANK", 10, "librank", NULL);
48. //输出kernel log
49. do\_dmesg();
51. //所有已打开文件
52. run\_command("LIST OF OPEN FILES", 10, SU\_PATH, "root", "lsof", NULL);
53. //遍历所有进程的show map
54. for\_each\_pid(do\_showmap, "SMAPS OF ALL PROCESSES");
55. //显示所有线程的blocked位置
56. for\_each\_tid(show\_wchan, "BLOCKED PROCESS WAIT-CHANNELS");
58. //SYSTEM LOG
59. timeout = logcat\_timeout("main") + logcat\_timeout("system") + logcat\_timeout("crash");
60. **if** (timeout < 20000) {
61. timeout = 20000;
62. }
63. run\_command("SYSTEM LOG", timeout / 1000, "logcat", "-v", "threadtime", "-d", "\*:v", NULL);
65. //EVENT LOG
66. timeout = logcat\_timeout("events");
67. **if** (timeout < 20000) {
68. timeout = 20000;
69. }
70. run\_command("EVENT LOG", timeout / 1000, "logcat", "-b", "events", "-v", "threadtime", "-d", "\*:v", NULL);
72. //RADIO LOG
73. timeout = logcat\_timeout("radio");
74. **if** (timeout < 20000) {
75. timeout = 20000;
76. }
77. run\_command("RADIO LOG", timeout / 1000, "logcat", "-b", "radio", "-v", "threadtime", "-d", "\*:v", NULL);
79. //Log统计信息
80. run\_command("LOG STATISTICS", 10, "logcat", "-b", "all", "-S", NULL);
82. //输出当前虚拟机和native进程的vm traces
83. **if** (dump\_traces\_path != NULL) {
84. dump\_file("VM TRACES JUST NOW", dump\_traces\_path);
85. }
87. //输出上次发生ANR时vm traces，即路径/data/anr/traces.txt
88. struct stat st;
89. **char** anr\_traces\_path[PATH\_MAX];
90. property\_get("dalvik.vm.stack-trace-file", anr\_traces\_path, "");
91. **if** (!anr\_traces\_path[0]) {
92. printf("\*\*\* NO VM TRACES FILE DEFINED (dalvik.vm.stack-trace-file)\n\n");
93. } **else** {
94. **int** fd = TEMP\_FAILURE\_RETRY(open(anr\_traces\_path,
95. O\_RDONLY | O\_CLOEXEC | O\_NOFOLLOW | O\_NONBLOCK));
96. **if** (fd < 0) {
97. printf("\*\*\* NO ANR VM TRACES FILE (%s): %s\n\n", anr\_traces\_path, strerror(errno));
98. } **else** {
99. dump\_file\_from\_fd("VM TRACES AT LAST ANR", anr\_traces\_path, fd);
100. }
101. }
103. //输出慢操作的vm traces，例如/data/anr/slow1.txt
104. **if** (anr\_traces\_path[0] != 0) {
105. **int** tail = strlen(anr\_traces\_path)-1;
106. **while** (tail > 0 && anr\_traces\_path[tail] != '/') {
107. tail--;
108. }
109. **int** i = 0;
110. **while** (1) {
111. //例如trace文件为/data/anr/slow1.txt
112. sprintf(anr\_traces\_path+tail+1, "slow%02d.txt", i);
113. **if** (stat(anr\_traces\_path, &st)) {
114. **break**;
115. }
116. dump\_file("VM TRACES WHEN SLOW", anr\_traces\_path);
117. i++;
118. }
119. }
121. //输出tombstone信息，NUM\_TOMBSTONES=10，例如/data/tombstones/tombstone\_1
122. **int** dumped = 0;
123. **for** (size\_t i = 0; i < NUM\_TOMBSTONES; i++) {
124. **if** (tombstone\_data[i].fd != -1) {
125. dumped = 1;
126. dump\_file\_from\_fd("TOMBSTONE", tombstone\_data[i].name, tombstone\_data[i].fd);
127. tombstone\_data[i].fd = -1;
128. }
129. }
130. **if** (!dumped) {
131. printf("\*\*\* NO TOMBSTONES to dump in %s\n\n", TOMBSTONE\_DIR);
132. }
134. dump\_file("NETWORK DEV INFO", "/proc/net/dev");
135. dump\_file("QTAGUID NETWORK INTERFACES INFO", "/proc/net/xt\_qtaguid/iface\_stat\_all");
136. dump\_file("QTAGUID NETWORK INTERFACES INFO (xt)", "/proc/net/xt\_qtaguid/iface\_stat\_fmt");
137. dump\_file("QTAGUID CTRL INFO", "/proc/net/xt\_qtaguid/ctrl");
138. dump\_file("QTAGUID STATS INFO", "/proc/net/xt\_qtaguid/stats");
140. //输出上次的kernel log
141. **if** (!stat(PSTORE\_LAST\_KMSG, &st)) {
142. //文件为/sys/fs/pstore/console-ramoops
143. dump\_file("LAST KMSG", PSTORE\_LAST\_KMSG);
144. } **else** {
145. //文件为/proc/last\_kmsg
146. dump\_file("LAST KMSG", "/proc/last\_kmsg");
147. }
149. //输出上次 logcat，内核必须设置CONFIG\_PSTORE\_PMSG
150. run\_command("LAST LOGCAT", 10, "logcat", "-L", "-v", "threadtime",
151. "-b", "all", "-d", "\*:v", NULL);
153. //wifi驱动/固件 以及ip相关信息
154. run\_command("NETWORK INTERFACES", 10, "ip", "link", NULL);
155. run\_command("IPv4 ADDRESSES", 10, "ip", "-4", "addr", "show", NULL);
156. run\_command("IPv6 ADDRESSES", 10, "ip", "-6", "addr", "show", NULL);
157. run\_command("IP RULES", 10, "ip", "rule", "show", NULL);
158. run\_command("IP RULES v6", 10, "ip", "-6", "rule", "show", NULL);
159. dump\_route\_tables();
160. run\_command("ARP CACHE", 10, "ip", "-4", "neigh", "show", NULL);
161. run\_command("IPv6 ND CACHE", 10, "ip", "-6", "neigh", "show", NULL);
162. run\_command("IPTABLES", 10, SU\_PATH, "root", "iptables", "-L", "-nvx", NULL);
163. run\_command("IP6TABLES", 10, SU\_PATH, "root", "ip6tables", "-L", "-nvx", NULL);
164. run\_command("IPTABLE NAT", 10, SU\_PATH, "root", "iptables", "-t", "nat", "-L", "-nvx", NULL);
165. run\_command("IPTABLE RAW", 10, SU\_PATH, "root", "iptables", "-t", "raw", "-L", "-nvx", NULL);
166. run\_command("IP6TABLE RAW", 10, SU\_PATH, "root", "ip6tables", "-t", "raw", "-L", "-nvx", NULL);
167. run\_command("WIFI NETWORKS", 20, SU\_PATH, "root", "wpa\_cli", "IFNAME=wlan0", "list\_networks", NULL);
169. //中断向量表
170. dump\_file("INTERRUPTS (1)", "/proc/interrupts");
171. run\_command("NETWORK DIAGNOSTICS", 10, "dumpsys", "connectivity", "--diag", NULL);
172. //中断向量表(二次输出)
173. dump\_file("INTERRUPTS (2)", "/proc/interrupts");
175. //获取properties属性值
176. print\_properties();
177. run\_command("VOLD DUMP", 10, "vdc", "dump", NULL);
178. run\_command("SECURE CONTAINERS", 10, "vdc", "asec", "list", NULL);
179. //可用空间
180. run\_command("FILESYSTEMS & FREE SPACE", 10, "df", NULL);
181. run\_command("LAST RADIO LOG", 10, "parse\_radio\_log", "/proc/last\_radio\_log", NULL);
183. //背光信息
184. printf("------ BACKLIGHTS ------\n");
185. printf("LCD brightness="); dump\_file(NULL, "/sys/class/leds/lcd-backlight/brightness");
186. printf("Button brightness="); dump\_file(NULL, "/sys/class/leds/button-backlight/brightness");
187. printf("Keyboard brightness="); dump\_file(NULL, "/sys/class/leds/keyboard-backlight/brightness");
188. printf("ALS mode="); dump\_file(NULL, "/sys/class/leds/lcd-backlight/als");
189. printf("LCD driver registers:\n"); dump\_file(NULL, "/sys/class/leds/lcd-backlight/registers");
190. printf("\n");
192. //Binder相关
193. dump\_file("BINDER FAILED TRANSACTION LOG", "/sys/kernel/debug/binder/failed\_transaction\_log");
194. dump\_file("BINDER TRANSACTION LOG", "/sys/kernel/debug/binder/transaction\_log");
195. dump\_file("BINDER TRANSACTIONS", "/sys/kernel/debug/binder/transactions");
196. dump\_file("BINDER STATS", "/sys/kernel/debug/binder/stats");
197. dump\_file("BINDER STATE", "/sys/kernel/debug/binder/state");
199. printf("========================================================\n");
200. printf("== Board\n");
201. printf("========================================================\n");
202. dumpstate\_board(); printf("\n");
204. //输出framework各种服务的dumpsys信息
205. printf("========================================================\n");
206. printf("== Android Framework Services\n");
207. printf("========================================================\n");
208. run\_command("DUMPSYS", 60, "dumpsys", NULL); //很耗时则timeout=60s
210. printf("========================================================\n");
211. printf("== Checkins\n");
212. printf("========================================================\n");
213. run\_command("CHECKIN BATTERYSTATS", 30, "dumpsys", "batterystats", "-c", NULL);
214. run\_command("CHECKIN MEMINFO", 30, "dumpsys", "meminfo", "--checkin", NULL);
215. run\_command("CHECKIN NETSTATS", 30, "dumpsys", "netstats", "--checkin", NULL);
216. run\_command("CHECKIN PROCSTATS", 30, "dumpsys", "procstats", "-c", NULL);
217. run\_command("CHECKIN USAGESTATS", 30, "dumpsys", "usagestats", "-c", NULL);
218. run\_command("CHECKIN PACKAGE", 30, "dumpsys", "package", "--checkin", NULL);
220. //输出当前 运行中activity/service/provider信息
221. printf("========================================================\n");
222. printf("== Running Application Activities\n");
223. printf("========================================================\n");
224. run\_command("APP ACTIVITIES", 30, "dumpsys", "activity", "all", NULL);
225. printf("========================================================\n");
226. printf("== Running Application Services\n");
227. printf("========================================================\n");
228. run\_command("APP SERVICES", 30, "dumpsys", "activity", "service", "all", NULL);
229. printf("========================================================\n");
230. printf("== Running Application Providers\n");
231. printf("========================================================\n");
232. run\_command("APP SERVICES", 30, "dumpsys", "activity", "provider", "all", NULL);
233. printf("========================================================\n");
234. printf("== dumpstate: done\n");
235. printf("========================================================\n");
236. }

该方法涉及run\_command其他几个方法见下方

### run\_command()

功能是fork子进程并等待它执行完成，或者超时退出。当命令title不为空时，每次输出结果，都分别以下面作为开头和结尾:

------ <title> (<command>) ------

[<command>: <执行时长> elapsed]

### dump\_file()

。。。。。。。。。。。。。。。。。。。。。。。。。

## 总结

bugreport通过socket与dumpstate服务建立通信，在dumpstate.cpp中的dumpstate()方法完成核心功能，该功能依次输出内容项， 主要分为5大类：

current log： kernel,system, event, radio;

last log： kernel, system, radio;

vm traces： just now, last ANR, tombstones

dumpsys： all, checkin, app

system info：cpu, memory, io等

从bugreport内容的输出顺序的角度，再详细列举其内容：

系统build以及运行时长等相关信息；

内存/CPU/进程等信息；

kernel log；

lsof、map及Wait-Channels；

system log；

event log；

radio log;

vm traces：

VM TRACES JUST NOW (/data/anr/traces.txt.bugreport) (抓bugreport时主动触发)

VM TRACES AT LAST ANR (/data/anr/traces.txt) (存在则输出)

TOMBSTONE (/data/tombstones/tombstone\_xx) (存在这输出)

network相关信息；

last kernel log;

last system log;

ip相关信息；

中断向量表

property以及fs等信息

last radio log;

Binder相关信息；

dumpsys all：

dumpsys checkin相关:

dumpsys batterystats电池统计；

dumpsys meminfo内存

dumpsys netstats网络统计；

dumpsys procstats进程统计；

dumpsys usagestats使用情况；

dumpsys package.

dumpsys app相关

dumpsys activity;

dumpsys activity service all;

dumpsys activity provider all.

Tips： bugreport几乎涵盖整个系统信息，内容非常长，每一个子项都以------ xxx ------开头。 例如APP ACTIVITIES的开头便是 ------ APP ACTIVITIES (dumpsys activity all) ------，其中括号内的便是输出该信息指令，即dumpsys activity all，还有可能是内容所在节点，各个子项目类似的规律，看完前面的源码分析过程，相信你肯定能明白。下面一篇文章再进一步从bugreport内容的角度来说明其寓意。

# bugreport实战篇

文章Bugreport源码篇(一)从源码角度阐释了Bugreport，那么本文则是从实战角度来说说，如何看懂bugreport，对于通过bugreport获取到内容往往非常长，这里简单划分为19部分，先整体上简单说说每部分log的title信息。

## 系统build相关信息

系统build信息：

* dumpstate: date
* Build: getprop ro.build.display.id
* Build fingerprint: getprop ro.build.fingerprint
* Bootloader: getprop ro.bootloader
* Radio: getprop ro.baseband
* Network: getprop gsm.operator.alpha
* Kernel: dump\_file /proc/version
* Command line: cat /proc/cmdline

系统运行时间：

------ UPTIME (`uptime`) ------

up time: 00:01:53, idle time: 00:02:44, sleep time: 00:00:00

[uptime: 0.015s elapsed]

* up time：系统运行时长
* idle time：系统空闲时长
* sleep time：系统休眠时长
* elapsed: uptime指令执行时长

## 内存/CPU/进程等信息

下面列举bugreport结果相关项的title，其中括号中内容便是相应的命令

------ UPTIME MMC PERF (/sys/block/mmcblk0/) ------

------ MEMORY INFO (/proc/meminfo) ------

------ CPU INFO (top -n 1 -d 1 -m 30 -t) ------

------ PROCRANK (procrank) ------

------ VIRTUAL MEMORY STATS (/proc/vmstat) ------

------ VMALLOC INFO (/proc/vmallocinfo) ------

------ SLAB INFO (/proc/slabinfo) ------

------ ZONEINFO (/proc/zoneinfo) ------

------ PAGETYPEINFO (/proc/pagetypeinfo) ------

------ BUDDYINFO (/proc/buddyinfo) ------

------ FRAGMENTATION INFO (/d/extfrag/unusable\_index) ------

------ KERNEL WAKELOCKS (/proc/wakelocks) ------

------ KERNEL WAKE SOURCES (/d/wakeup\_sources: 1970-01-01 08:00:00) ------

------ KERNEL CPUFREQ (/sys/devices/system/cpu/cpu0/cpufreq/stats/time\_in\_state) ------

------ KERNEL SYNC (/d/sync) ------

------ PROCESSES (ps -P) ------

------ PROCESSES AND THREADS (ps -t -p -P) ------

------ PROCESSES (SELINUX LABELS) (ps -Z) ------

------ LIBRANK (librank) ------

这里的涉及的信息较多，这里就先介绍下/sys/block/mmcblk0/:

命令：dump\_files(“UPTIME MMC PERF”, mmcblk0, skip\_not\_stat, dump\_stat\_from\_fd); 输出/sys/block/mmcblk0/stat节点以及遍历/sys/block/mmcblk0/mmcblk0p/stat

stat: 51897 32911 5984151 171690 198408 171047 29988513 2001070 0 334260 2172230

stat: read: 17845KB/s write: 1233KB/s

mmcblk0p24/stat: 16 28 346 50 6 2 64 20 0 70 70

mmcblk0p24/stat: read: 3543KB/s write: 1638KB/s

...

另外read = 512*data3/data4, write = 512*data7/data8。

| **Name** | **单位** | **含义** |
| --- | --- | --- |
| read I/Os | requests | 已处理的I/O读操作请求个数 |
| read merges | requests | I/O读操作的合并请求个数 |
| read sectors | sectors(512B) | 读取的扇区数 |
| read ticks | ms | 读请求的总等待时间 |
| write I/Os | requests | 已处理的I/O写操作请求个数 |
| write merges | requests | I/O写操作的合并请求个数 |
| write sectors | sectors(512B) | 写入的扇区数 |
| write ticks | ms | 写请求的总等待时间 |
| in\_flight | requests | 正在处理中的I/O请求数 |
| io\_ticks | ms | 该块设备处于活跃态的总时长 |
| time\_in\_queue | ms | 所有请求的总等待时长 |

## kernel log

内核log信息，通过命令dmesg可获取

------ KERNEL LOG (dmesg) ------

## 4. lsof、map及Wait-Channels

### 4.1 lsof

------ LIST OF OPEN FILES (/system/xbin/su root lsof) ------

COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME

zygote 286 root mem ??? b3:17 16384 2827 /system/vendor/lib/libcneconn.so

...

所有进程打开的文件，例如pid=286的zygote进程打开libcneconn.so文件。

### show map

------ SHOW MAP 1 (/init) (/system/xbin/su root showmap 1) ------

------ SHOW MAP 2 () (/system/xbin/su root showmap 2) ------

...

------ SHOW MAP 3649 () (/system/xbin/su root showmap 3649) ------

所有进程的show map

### 4.3 Wait-Channels

------ BLOCKED PROCESS WAIT-CHANNELS ------

1 /init SyS*\_epoll\_*wait

...

所有线程的Wait-Channels，记录着所有线程最后停留所在位置，例如上面是指pid=1的进程init，最后blocked在SyS\_epoll\_wait方法内。

## system log

------ SYSTEM LOG (logcat -v threadtime -d \*:v) ------

## 6. event log

------ EVENT LOG (logcat -b events -v threadtime -d \*:v) -----

## 7. radio log

------ RADIO LOG (logcat -b radio -v threadtime -d \*:v) ------

另外log统计信息，记录main/radio/events/system/crash/kernel各个log使用情况 —— LOG STATISTICS (logcat -b all -S) ——

## 8. vm traces

------ VM TRACES JUST NOW (/data/anr/traces.txt.bugreport: 2016-06-09 18:57:29) ------

------ VM TRACES AT LAST ANR (/data/anr/traces.txt: 2016-06-09 18:47:39) ------

#### 8.1 just now的栈信息

关键词为”VM TRACES JUST NOW”，数据来源”/data/anr/traces.txt.bugreport”

#### 8.2 last ANR的栈信息

关键词为”VM TRACES AT LAST ANR”，数据来源”/data/anr/traces.txt”。

若存在该ANR则输出相应traces，否则输出：

*\*\*\** NO ANR VM TRACES FILE (/data/anr/traces.txt): No such file or directory

#### tombstones信息

tombstones则是由debuggerd生成的，后续再单独用一篇文章来说明debuggerd的工作原理

------ TOMBSTONE (/data/tombstones/tombstone\_01: 2016-06-18 11:16:18) ------

如果tombstones文件不存在则输出：

\*\*\* NO TOMBSTONES to dump in /data/tombstones

## network

------ NETWORK DEV INFO (/proc/net/dev) ------

------ QTAGUID NETWORK INTERFACES INFO (/proc/net/xt\_qtaguid/iface\_stat\_all) ------

------ QTAGUID NETWORK INTERFACES INFO (xt) (/proc/net/xt\_qtaguid/iface\_stat\_fmt) ------

------ QTAGUID CTRL INFO (/proc/net/xt\_qtaguid/ctrl) ------

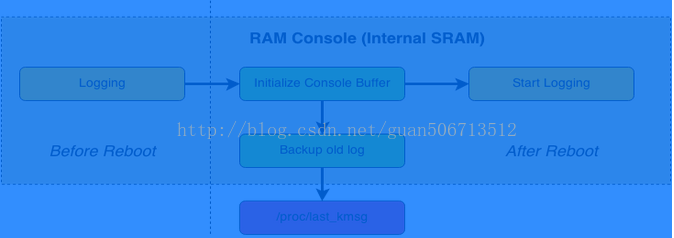
------ QTAGUID STATS INFO (/proc/net/xt\_qtaguid/stats) ------

## last kernel log

当 Linux Kernel 出现 BUG 的时候，后走入 panic flow，这个时候由于 Kernel 出现了严重的问题，Native 程序全部都无法运行，adbd 也无法响应 adb 连接请求，这个时候想透过读取 Kernel Log Buffer 来看 Kernel Log 是不可能的。那怎么办？

由于 Console Log 是不需要下任何命令就可以查看的，所以可以抓取 UART Log 分析，但是抓 UART Log 就要飞线，就会比较麻烦，有没有一种简单的方法可以让我们知道系统大概发生了什么事情呢？那就是 RAM Console Log。

1. 下图所示是 RAM Console 的工作原理。



其大致的工作流程如下：

1. 当系统开机初始化好 RAM Console Buffer 的时候就会开始进行 Logging。
2. 一旦系统发生严重异常就会发生重启。
3. 重启后系统启动又会开始初始化 RAM Console Buffer，在初始化 RAM Console Buffer 之前 Buffer 里面还保存了重启前的 Log 信息，在初始化的时候可以先将重启前的 Log 备份到另外一个 Buffer。
4. 备份的 Buffer 可以通过 cat /proc/last\_kmsg 来访问其中的内容。

通过上述的机制我们就可以在重启后通过 /proc/last\_kmsg 来查看重启前发生了什么事情。

### 应用场景

RAM Console 主要可以帮助我们理清两件事情：

1. 系统是否发生了重启。因为 /proc/last\_kmsg 只有重启之后才会存在，所以只要 /proc/last\_kmsg 存在我们就知道系统重启过了。（不是所有的重启都是异常重启）
2. 如果系统重启了，我们可以根据 /proc/last\_kmsg 的内容查看重启前系统都做了什么事情。

------ LAST KMSG (/proc/last\_kmsg) ------

## last system log

------ LAST LOGCAT (logcat -L -v threadtime -b all -d \*:v) ------

## 2. ip相关

------ NETWORK INTERFACES (ip link) ------

------ IPv4 ADDRESSES (ip -4 addr show) ------

------ IPv6 ADDRESSES (ip -6 addr show) ------

------ IP RULES (ip rule show) ------

------ IP RULES v6 (ip -6 rule show) ------

------ RT\_TABLES (/data/misc/net/rt\_tables: 2016-06-08 22:21:12) ------

------ ROUTE TABLE IPv4 (ip -4 route show table 255) ------

------ ROUTE TABLE IPv6 (ip -6 route show table 255) ------

...

------ ARP CACHE (ip -4 neigh show) ------

------ IPv6 ND CACHE (ip -6 neigh show) ------

------ IPTABLES (/system/xbin/su root iptables -L -nvx) ------

------ IP6TABLES (/system/xbin/su root ip6tables -L -nvx) ------

------ IPTABLE NAT (/system/xbin/su root iptables -t nat -L -nvx) ------

------ IPTABLE RAW (/system/xbin/su root iptables -t raw -L -nvx) ------

------ IP6TABLE RAW (/system/xbin/su root ip6tables -t raw -L -nvx) ------

------ WIFI NETWORKS (/system/xbin/su root wpa\_cli IFNAME=wlan0 list\_networks) ------

## 13. 中断向量表

中断向量表，记录着中断号对应的中断模块。

------ INTERRUPTS (1) (/proc/interrupts) ------

------ NETWORK DIAGNOSTICS (dumpsys connectivity --diag) ------

------ INTERRUPTS (2) (/proc/interrupts) ------

## 14. property信息

------ SYSTEM PROPERTIES ------

------ VOLD DUMP (vdc dump) ------

------ SECURE CONTAINERS (vdc asec list) ------

------ FILESYSTEMS & FREE SPACE (df) ------

## 15. last radio log

------ LAST RADIO LOG (parse\_radio\_log /proc/last\_radio\_log) ------

当然，背光信息 —— BACKLIGHTS ——

* LCD brightness= dump\_file /sys/class/leds/lcd-backlight/brightness
* Button brightness= dump\_file /sys/class/leds/button-backlight/brightness
* Keyboard brightness= dump\_file /sys/class/leds/keyboard-backlight/brightness
* ALS mode= dump\_file /sys/class/leds/lcd-backlight/als
* LCD driver registers: dump\_file /sys/class/leds/lcd-backlight/registers

## 16. Binder相关

------ BINDER FAILED TRANSACTION LOG (/sys/kernel/debug/binder/failed\_transaction\_log) ------

------ BINDER TRANSACTION LOG (/sys/kernel/debug/binder/transaction\_log) ------

------ BINDER TRANSACTIONS (/sys/kernel/debug/binder/transactions) ------

------ BINDER STATS (/sys/kernel/debug/binder/stats) ------

------ BINDER STATE (/sys/kernel/debug/binder/state) ------

紧接着之后还有：

------ DUMP VENDOR RIL LOGS (/system/xbin/su root vril-dump) ------

## 17. dumpsys相关：

------ DUMPSYS (dumpsys) ------

通过dumpsys -l可查看系统所有服务，不带参数的dumpsys命令会输出系统中所有的服务：每个服务开头信息：

DUMP OF SERVICE SurfaceFlinger:

DUMP OF SERVICE alarm:

DUMP OF SERVICE cpuinfo:

DUMP OF SERVICE dropbox:

...

## 18. dumpsys checkin相关

------ CHECKIN BATTERYSTATS (dumpsys batterystats -c) ------

------ CHECKIN MEMINFO (dumpsys meminfo --checkin) ------

------ CHECKIN NETSTATS (dumpsys netstats --checkin) ------

------ CHECKIN PROCSTATS (dumpsys procstats -c) ------

------ CHECKIN USAGESTATS (dumpsys usagestats -c) ------

------ CHECKIN PACKAGE (dumpsys package --checkin) ------

dumpsys电池统计，内存、网络统计、进程、使用情况、包这些统计类信息。

## 19. dumpsys app相关

------ APP ACTIVITIES (dumpsys activity all) ------

------ APP SERVICES (dumpsys activity service all) ------

------ APP SERVICES (dumpsys activity provider all) ------

dumpsys应用activity信息以及service和provider信息。

## 20. Tips

当需要搜索bugreport中内容时，可通过搜索------ 查找所有的子项目。 比如查看dumpsys信息，则可通过搜索------ DUMPSYS即可跳转到相应内容的开头。

bugreport具体内容涉及面很广，后续文章会介绍bugreport涉及的DropBoxManagerService，debuggerd等相关问题。 这里先说说一款开源的bugreport分析工具

# ChkBugReport

ChkBugReport是一款由[sonymobile](http://developer.sonymobile.com/2012/01/25/new-bugreport-analysis-tool-released-as-open-source/)开源的工具，用于分析Bugreport并提供可视化阅读的html格式文件。

### 2.1 用法

(1)通过命令生成bugreport文件

adb shell bugreport > bugreport.txt

(2)执行chkbugreport，命令中.jar 和.txt都必须填写相应文件所在的完全路径。

java –jar chkbugreport.jar bugreport.txt

当然也可以把.jar添加到path，则直接使用chkbugreport bugreport.txt

(3) 通过浏览器打开/bugreport\_out/index.html，可视化信息便呈现眼前。

### 2.2 chkbugreport分析

* [chkbugreport源码位于github](https://github.com/sonyxperiadev/ChkBugReport)；
* [chkbugreport jar](https://github.com/sonyxperiadev/ChkBugReport)直接下载，现有jar比较老，很久没有维护，建议自行下载源码，重新打包，功能更全。
* chkbugreport实例：
  + [进程死锁](http://sonyxperiadev.github.io/ChkBugReport/examples/bugreport_deadlock_out/index.html)
  + [AIDL死锁](http://sonyxperiadev.github.com/ChkBugReport/examples/bugreport_aidldeadlock_out/index.html)
  + [混合死锁](http://sonyxperiadev.github.io/ChkBugReport/examples/bugreport_hybriddeadlock_out/index.html)

对于死锁问题，要留意state=MONITOR的线程，以及main线程的情况。

**Tips**：Chkbugreport虽然是一款比较方便的可视化分析工具，但分析问题仍不够全面，加上各大厂商会有自身的定制，导致信息遗漏。因此往往很多时候分析问题，只能以Chkbugreport辅助分析，还是需要自己去分析bugreport信息，因此还是很有必要掌握bugreport每一项信息的含义

## REF

<http://gityuan.com/2016/06/11/bugreport-2/>

# Bootanimation

## 概述

### 目录结构

frameworks/base/cmds/bootanimation

目录中包含如下文件

文件名 作用

Android.mk mk文件，编译模块使用

AudioPlayer.cpp、AudioPlayer.h 音频播放

BootAnimation.cpp、BootAnimation.h 开机动画的源文件和头文件

bootanimation\_main.cpp 开机动画的入口

### 功能原理

Android系统的开机动画是由应用程序bootanimation来实现的，它位于/system/bin目录下，它的具体实现可以参考Android系统的开机画面显示过程分析一文。为什么要选择Android系统的开机动画来分析Android应用程序与SurfaceFlinger服务的连接过程呢？首先，负责实现开机动画的应用程序bootanimation也是一个Android应用程序，只不过它是使用C++语言来开发的；其次，应用程序bootanimation是与UI相关的，即它与使用Java语言来开发的标准Android应用程序一样，都需要使用SurfaceFlinger服务来创建和渲染自己的Surface，即开机动画；第三，由于应用程序bootanimation不涉及用户输入，即不需要与用户进行交互（触摸屏、键盘等），因此它能够以最简洁的方式来体现Android应用程序与SurfaceFlinger服务的关系。

## 调用入口

调用入口

应用程序bootanimation来实现的位于/system/bin目录下，bootanimation在init.rc中的定义如下：

1. service bootanim /system/bin/bootanimation
2. class core
3. user graphics
4. group graphics audio
5. disabled
6. oneshot

可见，由于设置为"disable"，该应用在init启动过程中是不会启动的，需要其他地方显示的调用才能启动。那是什么时候启动的呢？当SurfaceFlinger服务启动时，会修改系统属性值ctl.start，通知init进程启动bootanimation。

在早期的Android版本中，SurfaceFlinger服务是由SystemServer启动的。但在Android5.1中，该服务是init进程启动过程中就启动了。在init.rc中能看到对该服务的描述：

1. service surfaceflinger /system/bin/surfaceflinger
2. class core
3. user system
4. group graphics drmrpc
5. onrestart restart zygote

SurfaceFlinger服务源码路径为：frameworks\native\services\surfaceflinger

服务的入口在main\_surfaceflinger.cpp中，具体为：

1. int main(int, char\*\*) {
2. // When SF is launched in its own process, limit the number of
3. // binder threads to 4.
4. ProcessState::self()-**>**setThreadPoolMaxThreadCount(4);
6. // start the thread pool
7. sp**<ProcessState>** ps(ProcessState::self());
8. ps-**>**startThreadPool();
10. // instantiate surfaceflinger
11. sp**<SurfaceFlinger>** flinger = new SurfaceFlinger();
13. #if defined(HAVE\_PTHREADS)
14. setpriority(PRIO\_PROCESS, 0, PRIORITY\_URGENT\_DISPLAY);
15. #endif
16. set\_sched\_policy(0, SP\_FOREGROUND);
18. // initialize before clients can connect
19. flinger-**>**init();
21. // publish surface flinger
22. sp**<IServiceManager>** sm(defaultServiceManager());
23. sm-**>**addService(String16(SurfaceFlinger::getServiceName()), flinger, false);
25. // run in this thread
26. flinger-**>**run();
28. return 0;
29. }

主要工作是：新建一个SurfaceFlinger对象，然后调用其中的init()方法，最后调用其中的run()方法。

下面主要看一下SurfaceFlinger::init()方法，具体代码为：

1. void SurfaceFlinger::init() {
2. ALOGI(  "SurfaceFlinger's main thread ready to run. "
3. "Initializing graphics H/W...");
5. ......
7. // start boot animation
8. startBootAnim();
9. }

SurfaceFlinger调用了startBootAnim()启动了开机动画。该函数代码如下：

1. void SurfaceFlinger::startBootAnim() {
2. // start boot animation
3. property\_set("service.bootanim.exit", "0");
4. property\_set("ctl.start", "bootanim");
5. }

剩下部分属于属性系统，不做分析了

开机动画的显示过程

http://blog.csdn.net/u010753159/article/details/51325500

## 源码分析

### 入口类bootanimation\_main.cpp

bootanimation\_main.cpp

文件中定义main函数，也就是C语言中的执行文件的入口函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | <code>int main(int argc, char\*\* argv)  {  //宏定义判断是否设置进程的优先级  #if defined(HAVE\_PTHREADS)      setpriority(PRIO\_PROCESS, 0, ANDROID\_PRIORITY\_DISPLAY);  #endif        char value[PROPERTY\_VALUE\_MAX];      //这个配置项决定是否存在开机动画      property\_get("debug.sf.nobootanimation", value, "0");      int noBootAnimation = atoi(value);      ALOGI\_IF(noBootAnimation,  "boot animation disabled");      if (!noBootAnimation) {          //创建ProcessSate对象          // 这个过程会打开/dev/binder设备，形成和内核binder机制的交互的通道; 映射fd到内存          sp<processstate> proc(ProcessState::self());          //创建线程并加入到线程池          ProcessState::self()->startThreadPool();            // 创建开机动画对象          sp<bootanimation> boot = new BootAnimation();          //把主线程加入到线程池          IPCThreadState::self()->joinThreadPool();        }      return 0;  } |

### BootAnimation.cpp

Sdf

1. namespace android {
3. class AudioPlayer;
4. class Surface;
5. class SurfaceComposerClient;
6. class SurfaceControl;
8. *// ---------------------------------------------------------------------------*
10. class BootAnimation : public Thread, public IBinder::DeathRecipient
11. {
12. public:
13. BootAnimation();
14. virtual ~BootAnimation();
16. sp<surfacecomposerclient> session() const;
18. private:
19. virtual bool threadLoop();
20. virtual status\_t readyToRun();
21. virtual void onFirstRef();
22. virtual void binderDied(const wp<ibinder>& who);
24. *//Texture类定义*
25. struct Texture {
26. GLint w; *//宽度*
27. GLint h; *//高度*
28. GLuint name; *//名称*
29. };
30. *//动画内容结构体*
31. struct Animation {
32. *//动画帧*
33. struct Frame {
34. String8 name;
35. FileMap\* map;
36. mutable GLuint tid;
37. bool operator < (const Frame& rhs) const {
38. return name < rhs.name;
39. }
40. };
41. *//动画部分，因为动画可能是由几个部分组成*
42. struct Part {
43. int count;
44. int pause;
45. String8 path;
46. SortedVector frames;
47. bool playUntilComplete;
48. float backgroundColor[3];
49. FileMap\* audioFile;
50. };
51. int fps;
52. int width;
53. int height;
54. Vector<part> parts;
55. };
57. status\_t initTexture(Texture\* texture, AssetManager& asset, const char\* name);
58. status\_t initTexture(const Animation::Frame& frame);
59. bool android();
60. bool readFile(const char\* name, String8& outString);
61. bool movie();
63. void checkExit();
65. sp<surfacecomposerclient> mSession;
66. sp mAudioPlayer;
67. AssetManager mAssets;
68. Texture mAndroid[2];
69. int mWidth;
70. int mHeight;
71. EGLDisplay mDisplay;
72. EGLDisplay mContext;
73. EGLDisplay mSurface;
74. sp<surfacecontrol> mFlingerSurfaceControl;
75. sp<surface> mFlingerSurface;
76. ZipFileRO \*mZip;
77. };
79. *// ---------------------------------------------------------------------------*
81. }; *// namespace android*

#### 构造函数

1. BootAnimation::BootAnimation() : Thread(false), mClockEnabled(true), mTimeIsAccurate(false),
2. mTimeCheckThread(NULL) {
3. *////用于界面显示的mSession，与SurfaceFlinger交互的客户端*
4. mSession = new SurfaceComposerClient();
6. *// If the system has already booted, the animation is not being used for a boot.*
7. mSystemBoot = !property\_get\_bool(BOOT\_COMPLETED\_PROP\_NAME, 0);

mSession是BootAnimation类的成员变量，它是一个类型为SurfaceComposerClient的强指针，即sp<SurfaceComposerClient>

然后执行

void BootAnimation::onFirstRef() {

    status\_t err = mSession->linkToComposerDeath(this);

    ALOGE\_IF(err, "linkToComposerDeath failed (%s) ", strerror(-err));

    if (err == NO\_ERROR) {

        run("BootAnimation", PRIORITY\_DISPLAY);

    }

}

由于BootAnimation继承Thread类，首先会调用readyToRun函数

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76 | <code>status\_t BootAnimation::readyToRun() {      mAssets.addDefaultAssets();        sp<ibinder> dtoken(SurfaceComposerClient::getBuiltInDisplay(              ISurfaceComposer::eDisplayIdMain));      DisplayInfo dinfo;      status\_t status = SurfaceComposerClient::getDisplayInfo(dtoken, &dinfo);      if (status)          return -1;        // create the native surface      sp<surfacecontrol> control = session()->createSurface(String8("BootAnimation"),              dinfo.w, dinfo.h, PIXEL\_FORMAT\_RGB\_565);        SurfaceComposerClient::openGlobalTransaction();      control->setLayer(0x40000000);      SurfaceComposerClient::closeGlobalTransaction();        sp<surface> s = control->getSurface();        // initialize opengl and egl      const EGLint attribs[] = {              EGL\_RED\_SIZE,   8,              EGL\_GREEN\_SIZE, 8,              EGL\_BLUE\_SIZE,  8,              EGL\_DEPTH\_SIZE, 0,              EGL\_NONE      };      EGLint w, h, dummy;      EGLint numConfigs;      EGLConfig config;      EGLSurface surface;      EGLContext context;        EGLDisplay display = eglGetDisplay(EGL\_DEFAULT\_DISPLAY);        eglInitialize(display, 0, 0);      eglChooseConfig(display, attribs, &config, 1, &numConfigs);      surface = eglCreateWindowSurface(display, config, s.get(), NULL);      context = eglCreateContext(display, config, NULL, NULL);      eglQuerySurface(display, surface, EGL\_WIDTH, &w);      eglQuerySurface(display, surface, EGL\_HEIGHT, &h);        if (eglMakeCurrent(display, surface, surface, context) == EGL\_FALSE)          return NO\_INIT;        mDisplay = display;      mContext = context;      mSurface = surface;      mWidth = w;      mHeight = h;      mFlingerSurfaceControl = control;      mFlingerSurface = s;        // If the device has encryption turned on or is in process      // of being encrypted we show the encrypted boot animation.      char decrypt[PROPERTY\_VALUE\_MAX];      property\_get("vold.decrypt", decrypt, "");        bool encryptedAnimation = atoi(decrypt) != 0 || !strcmp("trigger\_restart\_min\_framework", decrypt);        ZipFileRO\* zipFile = NULL;      if ((encryptedAnimation &&              (access(SYSTEM\_ENCRYPTED\_BOOTANIMATION\_FILE, R\_OK) == 0) &&              ((zipFile = ZipFileRO::open(SYSTEM\_ENCRYPTED\_BOOTANIMATION\_FILE)) != NULL)) ||                ((access(OEM\_BOOTANIMATION\_FILE, R\_OK) == 0) &&              ((zipFile = ZipFileRO::open(OEM\_BOOTANIMATION\_FILE)) != NULL)) ||                ((access(SYSTEM\_BOOTANIMATION\_FILE, R\_OK) == 0) &&              ((zipFile = ZipFileRO::open(SYSTEM\_BOOTANIMATION\_FILE)) != NULL))) {          mZip = zipFile;      }        return NO\_ERROR;  }</surface></surfacecontrol></ibinder></code> |

上面主要做两个操作：

初始化显示界面用于播放开机动画，egl的一些内容；  
2. 根据手机是否[加密](https://www.2cto.com/article/jiami/)选择不同的开机动画文件，然后拿到入口zipFile  
其中用到了一个文件路径在BootAnimation.cpp的开头有定义

[?](https://www.2cto.com/kf/201608/536186.html)

|  |  |
| --- | --- |
| 1  2  3  4  5 | <code>#define OEM\_BOOTANIMATION\_FILE "/oem/media/bootanimation.zip"   //这个应该是OEM厂商自己定制    #define SYSTEM\_BOOTANIMATION\_FILE "/system/media/bootanimation.zip"   //正常情况下的Android原始开机动画    #define SYSTEM\_ENCRYPTED\_BOOTANIMATION\_FILE "/system/media/bootanimation |

readyToRun方法执行完之后，接着看threaLoop函数

### threaLoop

dsf

bool BootAnimation::threadLoop()

{

bool r;

// We have no bootanimation file, so we use the stock android logo

// animation.

if (mZipFileName.isEmpty()) {

r = android();

} else {//存在开机动画文件，则执行对应的开机动画文件解析出来的内容

r = movie();

}

eglMakeCurrent(mDisplay, EGL\_NO\_SURFACE, EGL\_NO\_SURFACE, EGL\_NO\_CONTEXT);

eglDestroyContext(mDisplay, mContext);

eglDestroySurface(mDisplay, mSurface);

mFlingerSurface.clear();

mFlingerSurfaceControl.clear();

eglTerminate(mDisplay);

IPCThreadState::self()->stopProcess();

return r;

#### android

fg

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41 | <code>bool BootAnimation::android()  {      //初始化两个纹理用于显示logo      initTexture(&mAndroid[0], mAssets, "images/android-logo-mask.png");      initTexture(&mAndroid[1], mAssets, "images/android-logo-shine.png");        // clear screen 清屏      glShadeModel(GL\_FLAT);      glDisable(GL\_DITHER);      glDisable(GL\_SCISSOR\_TEST);      glClearColor(0,0,0,1);      glClear(GL\_COLOR\_BUFFER\_BIT);      eglSwapBuffers(mDisplay, mSurface);        glEnable(GL\_TEXTURE\_2D);      glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);        const GLint xc = (mWidth  - mAndroid[0].w) / 2;      const GLint yc = (mHeight - mAndroid[0].h) / 2;      const Rect updateRect(xc, yc, xc + mAndroid[0].w, yc + mAndroid[0].h);        glScissor(updateRect.left, mHeight - updateRect.bottom, updateRect.width(),              updateRect.height());        // Blend state      glBlendFunc(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA);      glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);        const nsecs\_t startTime = systemTime();      //不停的显示知道exitPending()返回true      do {          ......          EGLBoolean res = eglSwapBuffers(mDisplay, mSurface);          ......            checkExit();      } while (!exitPending());      glDeleteTextures(1, &mAndroid[0].name);      glDeleteTextures(1, &mAndroid[1].name);      return false;  }</code> |

主要工作，初始化显示的logo纹理，不断刷新界面直到exitPending()返回true，exitPenging()是Thread类中定义的函数，在checkExit()函数中通过requestExit()来执行退出

[?](https://www.2cto.com/kf/201608/536186.html)

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | <code>void BootAnimation::checkExit() {      // Allow surface flinger to gracefully request shutdown      char value[PROPERTY\_VALUE\_MAX];      //EXIT\_PROP\_NAME "service.bootanim.exit" 这个配置项在SurfaceFlinger的bootFinished函数中设置为1，然后这里才能退出开机动画，这个过程设计到开机启动到launcher的整个过程，这里不赘述      property\_get(EXIT\_PROP\_NAME, value, "0");      int exitnow = atoi(value);      if (exitnow) {          requestExit();          if (mAudioPlayer != NULL) {              mAudioPlayer->requestExit();          }      }  }</code> |

#### movie()

ie()这个函数有点长，我们截断一点一点的看

##### parseAnimationDesc

g if (!readFile("desc.txt", desString)) {

return false;

}

char const\* s = desString.string();

//读取desc.txt文件

// Create and initialize an AudioPlayer if we have an audio\_conf.txt file

String8 audioConf;

//判断是否需要创建AudioPlayer,这部分我们暂时不关注

if (readFile("audio\_conf.txt", audioConf)) {

mAudioPlayer = new AudioPlayer;

if (!mAudioPlayer->init(audioConf.string())) {

ALOGE("mAudioPlayer.init failed");

mAudioPlayer = NULL;

}

}

解析desc.txt文件，就是上面拿到的那个char const \*s

for (;;) {

//一行一行的截取

const char\* endl = strstr(s, "\n");

if (!endl) break;

String8 line(s, endl - s);

const char\* l = line.string();

//几个需要捕获的参数,帧率 宽高，次数

int fps, width, height, count, pause;

char path[ANIM\_ENTRY\_NAME\_MAX];

char color[7] = "000000"; // default to black if unspecified

char pathType;

//读取帧率和宽高

if (sscanf(l, "%d %d %d", &width, &height, &fps) == 3) {

// ALOGD("> w=%d, h=%d, fps=%d", width, height, fps);

animation.width = width;

animation.height = height;

animation.fps = fps;

}

//或者读取part内容

else if (sscanf(l, " %c %d %d %s #%6s", &pathType, &count, &pause, path, color) >= 4) {

// ALOGD("> type=%c, count=%d, pause=%d, path=%s, color=%s", pathType, count, pause, path, color);

Animation::Part part;

part.playUntilComplete = pathType == 'c';

part.count = count;

part.pause = pause;

part.path = path;

part.audioFile = NULL;

if (!parseColor(color, part.backgroundColor)) {

ALOGE("> invalid color '#%s'", color);

part.backgroundColor[0] = 0.0f;

part.backgroundColor[1] = 0.0f;

part.backgroundColor[2] = 0.0f;

}

animation.parts.add(part);

}

s = ++endl;

}

读取所有的数据

// read all the data structures

const size\_t pcount = animation.parts.size();

void \*cookie = NULL;

if (!mZip->startIteration(&cookie)) {

return false;

}

ZipEntryRO entry;

char name[ANIM\_ENTRY\_NAME\_MAX];

while ((entry = mZip->nextEntry(cookie)) != NULL) {

const int foundEntryName = mZip->getEntryFileName(entry, name, ANIM\_ENTRY\_NAME\_MAX);

if (foundEntryName > ANIM\_ENTRY\_NAME\_MAX || foundEntryName == -1) {

ALOGE("Error fetching entry file name");

continue;

}

const String8 entryName(name);

const String8 path(entryName.getPathDir());

const String8 leaf(entryName.getPathLeaf());

if (leaf.size() > 0) {

for (size\_t j=0 ; j<pcount files="" if="" int="" mzip-="" only="" path="=" png="" stored="" supports="">getEntryInfo(entry, &method, NULL, NULL, NULL, NULL, NULL)) {

if (method == ZipFileRO::kCompressStored) {

FileMap\* map = mZip->createEntryFileMap(entry);

if (map) {

Animation::Part& part(animation.parts.editItemAt(j));

if (leaf == "audio.wav") {

// a part may have at most one audio file

part.audioFile = map;

} else {

Animation::Frame frame;

frame.name = leaf;

frame.map = map;

part.frames.add(frame);

}

}

}

}

}

}

}

}

mZip->endIteration(cookie);

##### 显示动画

Rgs

// clear screen

glShadeModel(GL\_FLAT);

glDisable(GL\_DITHER);

glDisable(GL\_SCISSOR\_TEST);

glDisable(GL\_BLEND);

glClearColor(0,0,0,1);

glClear(GL\_COLOR\_BUFFER\_BIT);

eglSwapBuffers(mDisplay, mSurface);

glBindTexture(GL\_TEXTURE\_2D, 0);

glEnable(GL\_TEXTURE\_2D);

glTexEnvx(GL\_TEXTURE\_ENV, GL\_TEXTURE\_ENV\_MODE, GL\_REPLACE);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_S, GL\_REPEAT);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_WRAP\_T, GL\_REPEAT);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

const int xc = (mWidth - animation.width) / 2;

const int yc = ((mHeight - animation.height) / 2);

nsecs\_t lastFrame = systemTime();

nsecs\_t frameDuration = s2ns(1) / animation.fps;

Region clearReg(Rect(mWidth, mHeight));

clearReg.subtractSelf(Rect(xc, yc, xc+animation.width, yc+animation.height));

for (size\_t i=0 ; i<pcount 0="" const="" size\_t="" fcount="part.frames.size();" for="" int="" r="=" part.count="" exit="" any="" non="" playuntil="" complete="" parts="" immediately="" play="" audio="" file="" the="" first="" time="" we="" animate="" part="" if="" maudioplayer="" maudioplayer-="" data-cke-pa-only="">playFile(part.audioFile);

}

glClearColor(

part.backgroundColor[0],

part.backgroundColor[1],

part.backgroundColor[2],

1.0f);

for (size\_t j=0 ; j<fcount const="" nsecs\_t="" lastframe="systemTime();" if="" r=""> 0) {

glBindTexture(GL\_TEXTURE\_2D, frame.tid);

} else {

if (part.count != 1) {

glGenTextures(1, &frame.tid);

glBindTexture(GL\_TEXTURE\_2D, frame.tid);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MIN\_FILTER, GL\_LINEAR);

glTexParameterx(GL\_TEXTURE\_2D, GL\_TEXTURE\_MAG\_FILTER, GL\_LINEAR);

}

initTexture(frame);

}

if (!clearReg.isEmpty()) {

Region::const\_iterator head(clearReg.begin());

Region::const\_iterator tail(clearReg.end());

glEnable(GL\_SCISSOR\_TEST);

while (head != tail) {

const Rect& r(\*head++);

glScissor(r.left, mHeight - r.bottom,

r.width(), r.height());

glClear(GL\_COLOR\_BUFFER\_BIT);

}

glDisable(GL\_SCISSOR\_TEST);

}

glDrawTexiOES(xc, yc, 0, animation.width, animation.height);

eglSwapBuffers(mDisplay, mSurface);

nsecs\_t now = systemTime();

nsecs\_t delay = frameDuration - (now - lastFrame);

//ALOGD("%lld, %lld", ns2ms(now - lastFrame), ns2ms(delay));

lastFrame = now;

if (delay > 0) {

struct timespec spec;

spec.tv\_sec = (now + delay) / 1000000000;

spec.tv\_nsec = (now + delay) % 1000000000;

int err;

do {

err = clock\_nanosleep(CLOCK\_MONOTONIC, TIMER\_ABSTIME, &spec, NULL);

} while (err<0 && errno == EINTR);

}

checkExit();

}

usleep(part.pause \* ns2us(frameDuration));

// For infinite parts, we've now played them at least once, so perhaps exit

if(exitPending() && !part.count)

break;

}

// free the textures for this part

if (part.count != 1) {

for (size\_t j=0 ; j<fcount const="" pre="">

## 修改Android系统的开机动画

Android系统中，与开机动画相关的文件都放在一个叫bootanimation.zip的压缩包文件内，bootanimation.zip随system.img一起发布。需要放到system/media目录中.如果我们要查找系统自带的bootanimation.zip文件，可以将system目录还原，在其下的media目录下存在一个bootanimation.zip文件，替换该文件，我们就能完成开机动画的修改。bootanimation.zip文件解压看看里面的具体包含的内容：首先包含一个desc.txt文件，还会包含若干类似part0，part1的目录，其中desc.txt文件是必须的，part0、part1等目录至少要存在一个。part0、part1目录中存放的是图像文件，诸如001.png、002.png类似的命名有规律的图像文件。Android系统中读取这些静态图像，并按一定的显示规律，和频率产生动画效果，desc.txt文件就是用来描述加载规律和频率信息的文件。

为了使Android拥有Root权限，需要在代码中执行su命令。在执行su命令的过程中会创建一个新的拥有root权限的进程，通过该进程进行的任何操作都是在root权限下进行的

//执行su命令，并创建一个新进程（Process对象）

Process process = Runtime.getRuntime().exec("su");

//获取新进程的OutputStream对象，可以通过该对象发出要执行的命令

OutputStream os = process.getOutputStream();

//获取新进程的InputStream对象，可以通过该对象获取命令执行后返回的数据

InputStream is = process.getInputStream();

......

最后还要实现设备的重启，在Android系统中，要实现重启有两种方式：

1. 执行reboot命令.执行reboot命令需要root权限。也就是说，只要拥有了root权限，任何应用程序都可以重启Android设备

2. 调用PowerManager.reboot命令。该种重启方式只有System用户能使用，自由system用户才允许设置android.permission.REBOOT权限

PowerManager pm = (PowerManager)getSystemService(Context.POWER\_SERVICE);

pm.reboot("change boot animation");

最后执行重启设备命令需要在清单配置文件中配置android.permission.REBOOT权限

# Service

frameworks/native/cmds/service

这个工具的作用是让你能够在 shell 中调用 android system services 的一部分接口

Usage: service [-h|-?]

service list

service check SERVICE

service call SERVICE CODE [i32 INT | s16 STR] ...

Options:

i32: Write the integer INT into the send parcel.

s16: Write the UTF-16 string STR into the send parcel.

## Check

这个 check 命名就是调用 SM 的 checkService 接口。至于这个接口是干啥的，[看这里](http://light3moon.com/2015/01/28/Android%20Binder%20%E5%88%86%E6%9E%90%E2%80%94%E2%80%94%E7%B3%BB%E7%BB%9F%E6%9C%8D%E5%8A%A1%20Binder%20%E5%AF%B9%E8%B1%A1%E7%9A%84%E4%BC%A0%E9%80%92)。不过不知道是不是权限原因，这里从 SM 返回好像都是 NULL，然后老是打印 Service xx not found 。我没去深究，因为这个命令好像也没啥用

**if** (strcmp(argv[optind], **"check"**) == 0) {  
 optind++;  
 **if** (optind < argc) {  
 sp<IBinder> service = sm->checkService(String16(argv[optind]));  
 aout << **"Service "** << argv[optind] <<  
 (service == NULL ? **": not found"** : **": found"**) << endl;  
 } **else** {  
 aerr << **"service: No service specified for check"** << endl;  
 wantsUsage = **true**;  
 result = 10;  
 }

## list

列出当前系统有所有的 system service :

Vector<String16> services = sm->listServices();  
aout << **"Found "** << services.size() << **" services:"** << endl;  
**for** (**unsigned** i = 0; i < services.size(); i++) {  
 String16 name = services[i];  
 sp<IBinder> service = sm->checkService(name);  
 aout << i   
 << **"\t"** << good\_old\_string(name)   
 << **": ["** << good\_old\_string(get\_interface\_name(service)) << **"]"** << endl;  
}

### get\_interface\_name

get\_interface\_name 就是上面后面 [] 打印的东西，这个是每个 service 的标示，之前 binder 几篇的文章有说到每次 binder 通信，首先会检测这个标示的。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | // get the name of the generic interface we hold a reference to  static String16 get\_interface\_name(sp<IBinder> service)  {  if (service != NULL) {  Parcel data, reply;  status\_t err = service->transact(IBinder::INTERFACE\_TRANSACTION, data, &reply);  if (err == NO\_ERROR) {  return reply.readString16();  }  }  return String16();  } |

然后在 Binder.cpp 中，就是 binder Bn 的父类中默认处理中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | status\_t BBinder::onTransact(  uint32\_t code, const Parcel& data, Parcel\* reply, uint32\_t flags)  {  switch (code) {  case INTERFACE\_TRANSACTION:  reply->writeString16(getInterfaceDescriptor());  return NO\_ERROR;  ... ...  default:  return UNKNOWN\_TRANSACTION;  }  } |

这就获取每个 system service 的标示。

## service call

用指定 service 的接口

其中service call SERVICE CODE [i32 INT | s16 STR]...表明了该命令调用的格式：

SERVICE : 执行的service名称

CODE : 执行的方法id值

［i32 INT | s16 STR］: 参数类型以及紧跟参数值，i32代表int类型，s16代表String类型。

### 具体使用举例

自己通过list 然后去类里面去查找id吧

<https://www.jianshu.com/p/737427375091>

# Expand status bar

service call statusbar 1

# Collapse status bar

service call statusbar 2

风扇开关

XXX:/ $ service call kjilink 1019 i32 1

Result: Parcel(NULL)

XXX:/ $ service call kjilink 1019 i32 0

Result: Parcel(NULL)

### 源码

Parcel data, reply;  
  
*// the interface name is first*data.writeInterfaceToken(ifName);

**if** (strcmp(argv[optind], **"i32"**) == 0) {  
 optind++;  
 **if** (optind >= argc) {  
 aerr << **"service: no integer supplied for 'i32'"** << endl;  
 wantsUsage = **true**;  
 result = 10;  
 **break**;  
 }  
 data.writeInt32(atoi(argv[optind++]));  
}

service->transact(code, data, &reply);

### service call window 1 i32 4939

这个 TRANSACTION\_CODE 怎么确定咧。这个就要去翻源码啦。例如上面 wm 的那个接口，我在 IWindwManager.java（wm 是用 aidl 的，要去 out 下面去翻 aidl 自动生成的 java 文件） 中：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | static final int TRANSACTION\_startViewServer = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 0);  static final int TRANSACTION\_stopViewServer = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 1);  static final int TRANSACTION\_isViewServerRunning = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 2);  // 然后去翻下 IBinder 发现是从 1 开始的  enum {  FIRST\_CALL\_TRANSACTION = 0x00000001,  LAST\_CALL\_TRANSACTION = 0x00ffffff,  ... ...  }; |

所以要调用 wm 的 startViewServer 接口，code 就要填 1 啦。然后我们再去看下 startViewServer 的接口：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | /\*  \* Starts the view server on the specified port.  \*  \* @param port The port to listener to.  \*  \* @return True if the server was successfully started, false otherwise.  \*  \* @see com.android.server.wm.ViewServer  \* @see com.android.server.wm.ViewServer#VIEW\_SERVER\_DEFAULT\_PORT  \*/  @Override  public boolean startViewServer(int port) |

接口参数就一个 int 的 port（socket 的端口吧），所以后面那个参数就要填 i32 4939 （话说网上怎么知道端口要用 4939 的）。

然后我们来看下 IWindowManager.aidl 中一段有意思的注释：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | /\*\*  \* ===== NOTICE =====  \* The first three methods must remain the first three methods. Scripts  \* and tools rely on their transaction number to work properly.  \*/  // This is used for debugging  boolean startViewServer(int port); // Transaction #1  boolean stopViewServer(); // Transaction #2  boolean isViewServerRunning(); // Transaction #3 |

大致意思是说：请把这2个接口放在最前面声明，因为 aidl 是按顺序生成 code 号的，这样就能保证这3个接口的 code 固定是 1、2、3。这个明显是 android 故意留给这个 shell 命令的小工具的。所以你要自己要加一些 service 的接口，尽量从 LAST\_CALL\_TRANSACTION 从后往前面加，避免影响一些系统工具的使用。

然后如果是没有参数的话，后面那个参数可以不填，例如查询 wm view server 是否开启的接口：

service call window 3

然后就是返回值了。binder 的返回值都是通过 parcel 打包的，前面 binder 分析过了，每个服务的 Bp 都会自己解析返回值的 parcel 包，然后里面的返回值拆出来，转化成接口具体的返回值返回给调用者。但是这里就直接打印 parcel 了。前面 binder 也分析过，parcel 就是简单的把数据塞到内存里面而已（当然有一些类型有一点点格式）。这里也只是把 parcel 内存中的东西打印出来而已。

我们来看上面那个查询 wm view server 是否开启的结果：

root@H9:/ # service call window 3

Result: Parcel(00000000 00000001 '........')

返回值，第一个 int32 是错误值，0 表示没有错误。然后从第二开始才是真正的返回值。这个 startViewServer 的返回值看代码就是一个 int（那个接口是 boolean，在 Bp 那是根据 0、1 来判断的）。所以那个 00000001 就代表 view server 已经启动咯。如果是 00000000 就是没启动。

好，我们再来个稍微复杂点的例子。wm 中有一个这个接口：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | @Override  public float getAnimationScale(int which) {  switch (which) {  case 0: return mWindowAnimationScale;  case 1: return mTransitionAnimationScale;  case 2: return mAnimatorDurationScale;  }  return 0;  } |

而且这个正好在 Settings 中的开发这选项有可以设置（可以拿来测试用）。开发者选项—>绘图：

窗口动画缩放 --> mWindowAnimationScale

过渡动画缩放 --> mTransitionAnimationScale

动画程序时长缩放 --> mAnimatorDurationScale

我们拿第一个玩一下，因为第一个窗口动画缩放，只要设置成 10x 窗口的缩放动画就会变得很慢。去翻下 IWindowManager.java 4.4 的是：

|  |  |
| --- | --- |
| 1  2 | static final int TRANSACTION\_getAnimationScale = (android.os.IBinder.FIRST\_CALL\_TRANSACTION + 48); |

所以要获取 mWindowAnimationScale 的值的话就应该这么敲：

service call window 49 i32 0

得到的结果如下：

root@H9:/ # service call window 49 i32 0

Result: Parcel(00000000 3f800000 '.......?')

嘿嘿，返回的是 float，顺带学习下 float 在内存中的表示：[看这里啦](http://light3moon.com/2015/01/19/%5b%E8%BD%AC%5d%20float%20%E7%B1%BB%E5%9E%8B%E5%9C%A8%E5%86%85%E5%AD%98%E4%B8%AD%E7%9A%84%E8%A1%A8%E7%A4%BA)。

0x3f800000 就是 1.0 的 float。然后去 Settings 的调试选项，把这个调成 动画缩放 10x，再敲一下，结果是：

root@H9:/ # service call window 49 i32 0

Result: Parcel(00000000 41200000 '...... A')

0x41200000 是 10.0 的 float。

## 限制

从上面来看，这个小工具其实挺有用的。可以很方便的自己留个小后门或是加一些调试小接口。这个东西，需要知道接口的 TRANSACTION\_CODE 和参数才能正常调用，很符合留后门给自己用的风范 ^\_^。

还有从上面的分析来看，这个工具只能调用一些参数是 void、int、String 或是 intent 相关的接口。还有返回值如果是比较复杂的 class（结构体）要分析也比较困难，简单的 int、float、String 还好看懂。不过虽然有局限性，还是一个不错的小工具了

## 参考

Service clall的原理

<http://light3moon.com/2015/01/30/Android%20%E4%B8%80%E4%BA%9B%E6%9C%89%E6%84%8F%E6%80%9D%E7%9A%84%E5%91%BD%E4%BB%A4%E5%B0%8F%E5%B7%A5%E5%85%B7%20%E2%80%94%E2%80%94%20service/>