# TODO

logcat命令执行后的动作

覆盖了一些uid的日志，systemui的日志被覆盖掉了

<https://blog.csdn.net/c_z_w/article/details/82187616>

ALOGV 在release版本不打印？？

http://gityuan.com/2018/02/03/log-fc/

# wtf

可怕的失败：报告一个永远不可能发生的情况。大概是用来打印正常情况下永远不会发生bug？的意思是what a terrible failure，抛异常设计技巧

**try** {  
 mPackageManagerService.systemReady();  
} **catch** (Throwable e) {  
 reportWtf(**"making Package Manager Service ready"**, e);  
}

# 概述

无论是Android系统开发，还是应用开发，都离不开log，Androd上层采用logcat输出log。

## 文件目录

frameworks/base/core/java/android/util/

- Log.java

- Slog.java

- EventLog.java

frameworks/base/core/jni/android\_util\_Log.cpp

/system/core/logcat/logcat.cpp

/system/core/liblog/logd\_write.c

/system/core/liblog/uio.c

/system/core/logd/

- main.cpp

- LogBuffer.cpp

- LogStatistics.cpp

/system/core/libsysutils/src/SocketListener.cpp

## logcat命令说明

可通过adb命令直接输出指定的log:

logcat -b events // 输出指定buffer的log

logcat -s "ActivityManager"

logcat -L //上次重启时的log

logcat -f [filename] //将log保存到指定文件

logcat -g //缓冲区大小

logcat -S //统计log信息

-b 默认是指-b main -b system -b crash, 当然一可以指定其他参数, 或者直接指定all.

## log函数api使用

### 2.1 Java层

默认定义了5个Buffer缓冲区,如下:

| **ID** | **名称** | **使用方式** |
| --- | --- | --- |
| LOG\_ID\_MAIN | main | Log.i |
| LOG\_ID\_RADIO | radio | Rlog.i |
| LOG\_ID\_EVENTS | events | EventLog.writeEvent |
| LOG\_ID\_SYSTEM | system | Slog.i |
| LOG\_ID\_CRASH | crash | - |

log级别:

| **级别** | **对应值** | **使用场景** |
| --- | --- | --- |
| VERBOSE | 2 | 冗长信息 |
| DEBUG | 3 | 调试信息 |
| INFO | 4 | 普通信息 |
| WARN | 5 | 警告信息 |
| ERROR | 6 | 错误信息 |
| ASSERT | 7 | 普通但重要的信息 |

### 2.2 Kernel层

Linux Kernel最常使用的是printk，用法如下：

//第一个参数是级别， 第二个是具体log内容

printk(KERN\_INFO x);

日志级别的定义位于kernel/include/linux/printk.h文件，如下：

| **级别** | **对应值** | **使用场景** |
| --- | --- | --- |
| KERN\_EMERG | <0> | 系统不可用状态 |
| KERN\_ALERT | <1> | 警报信息，必须立即采取信息 |
| KERN\_CRIT | <2> | 严重错误信息 |
| KERN\_ERR | <3> | 错误信息 |
| KERN\_WARNING | <4> | 警告信息 |
| KERN\_NOTICE | <5> | 普通但重要的信息 |
| KERN\_INFO | <6> | 普通信息 |
| KERN\_DEBUG | <7> | 调试信息 |

日志输出到文件/proc/kmsg，可通过cat /proc/kmsg来获取内核log信息。

cat /proc/sys/kernel/printk

### 2.4 buffer大小

LogBuffer.cpp

可知 radio = 4M, 其他都为2M

# 原理分析

## Log.i

的

[-> android/util/Log.java]

public static int i(String tag, String msg) {

// [见小节3.2]

return println\_native(LOG\_ID\_MAIN, INFO, tag, msg);

}

Log.java中的方法都是输出到main buffer, 其中println\_native是Native方法， 通过JNI调用如下方法

frameworks/base/core/jni/android\_util\_Log.cpp

## println\_native

**static** jint android\_util\_Log\_println\_native(JNIEnv\* env, jobject clazz,  
 jint bufID, jint priority, jstring **tagObj**, jstring **msgObj**)  
{  
 **const char**\* tag = NULL;  
 **const char**\* msg = NULL;  
  
 **if** (msgObj == NULL) {  
 jniThrowNullPointerException(env, **"println needs a message"**);  
 **return** -1;  
 }  
  
 **if** (bufID < 0 || bufID >= LOG\_ID\_MAX) {  
 jniThrowNullPointerException(env, **"bad bufID"**);  
 **return** -1;  
 }  
  
 **if** (tagObj != NULL)  
 tag = env->GetStringUTFChars(tagObj, NULL);  
 msg = env->GetStringUTFChars(msgObj, NULL);  
  
 **int** res = \_\_android\_log\_buf\_write(bufID, (android\_LogPriority)priority, tag, msg);  
  
 **if** (tag != NULL)  
 env->ReleaseStringUTFChars(tagObj, tag);  
 env->ReleaseStringUTFChars(msgObj, msg);  
  
 **return** res;  
}

## \_\_android\_log\_buf\_write

* 对于满足特殊条件的tag，则会输出到LOG\_ID\_RADIO缓冲区；
* vec数组依次记录着log的级别，tag, msg

LIBLOG\_ABI\_PUBLIC **int** \_\_android\_log\_buf\_write(**int** bufID, **int** prio,  
 **const char** \*tag, **const char** \*msg)  
{  
 **struct** iovec vec[3];  
 **char** tmp\_tag[32];  
  
 **if** (!tag)  
 tag = **""**;  
  
 */\* XXX: This needs to go! \*/* **if** ((bufID != LOG\_ID\_RADIO) &&  
 (!strcmp(tag, **"HTC\_RIL"**) ||  
 !strncmp(tag, **"RIL"**, 3) || */\* Any log tag with "RIL" as the prefix \*/* !strncmp(tag, **"IMS"**, 3) || */\* Any log tag with "IMS" as the prefix \*/* !strcmp(tag, **"AT"**) ||  
 !strcmp(tag, **"GSM"**) ||  
 !strcmp(tag, **"STK"**) ||  
 !strcmp(tag, **"CDMA"**) ||  
 !strcmp(tag, **"PHONE"**) ||  
 !strcmp(tag, **"SMS"**))) {  
 bufID = LOG\_ID\_RADIO;  
 */\* Inform third party apps/ril/radio.. to use Rlog or RLOG \*/* snprintf(tmp\_tag, **sizeof**(tmp\_tag), **"use-Rlog/RLOG-%s"**, tag);  
 tag = tmp\_tag;  
 }  
  
 vec[0].iov\_base = (**unsigned char** \*)&prio;  
 vec[0].iov\_len = 1;  
 vec[1].iov\_base = (**void** \*)tag;  
 vec[1].iov\_len = strlen(tag) + 1;  
 vec[2].iov\_base = (**void** \*)msg;  
 vec[2].iov\_len = strlen(msg) + 1;  
  
 **return** write\_to\_log(bufID, vec, 3);  
}

其中write\_to\_log函数指针指向\_\_write\_to\_log\_init

#define LOG\_BUF\_SIZE 1024  
  
**static int** \_\_write\_to\_log\_init(log\_id\_t, **struct** iovec \*vec, size\_t nr);  
**static int** (\*write\_to\_log)(log\_id\_t, **struct** iovec \*vec, size\_t nr) = \_\_write\_to\_log\_init;

## 3.4 write\_to\_log

[-> logd\_write.c]

**static int** \_\_write\_to\_log\_init(log\_id\_t log\_id, **struct** iovec \*vec, size\_t nr)  
{  
 \_\_android\_log\_lock();  
  
 **if** (write\_to\_log == \_\_write\_to\_log\_init) {  
 **int** ret;  
  
 ret = \_\_write\_to\_log\_initialize();//执行log初始化【见小节3.4.1】  
 **if** (ret < 0) {  
 \_\_android\_log\_unlock();  
 **if** (!list\_empty(&\_\_android\_log\_persist\_write)) {  
 \_\_write\_to\_log\_daemon(log\_id, vec, nr); //【见小节3.5】  
 }  
 **return** ret;  
 }  
  
 write\_to\_log = \_\_write\_to\_log\_daemon;  
 }  
  
 \_\_android\_log\_unlock();  
  
 **return** write\_to\_log(log\_id, vec, nr);  
}

### \_\_write\_to\_log\_initialize

### \_\_write\_to\_log\_daemon

该方法的主要功能, 准备log相关的信息:

* pid
* uid
* tid
* realtime
* msg

**static int** \_\_write\_to\_log\_daemon(log\_id\_t log\_id, **struct** iovec \*vec, size\_t nr)  
{  
 **struct** android\_log\_transport\_write \*node;  
 **int** ret;  
 **struct** timespec ts;  
 size\_t len, i;  
  
 **for** (len = i = 0; i < nr; ++i) {  
 len += vec[i].iov\_len;  
 }  
  
 */\* simulate clock\_gettime(CLOCK\_REALTIME, &ts); \*/* {  
 **struct** timeval tv;  
  **gettimeofday(&tv, NULL);**  
 ts.tv\_sec = tv.tv\_sec;  
 ts.tv\_nsec = tv.tv\_usec \* 1000;  
 }  
  
 ret = 0;  
 i = 1 << log\_id;  
 write\_transport\_for\_each(node, &\_\_android\_log\_transport\_write) {  
 **if** (node->logMask & i) {  
 ssize\_t retval;  
 retval = (\*node->write)(log\_id, &ts, vec, nr);  
 **if** (ret >= 0) {  
 ret = retval;  
 }  
 }  
 }  
  
 write\_transport\_for\_each(node, &\_\_android\_log\_persist\_write) {  
 **if** (node->logMask & i) {  
 (**void**)(\*node->write)(log\_id, &ts, vec, nr);  
 }  
 }  
  
 **return** ret;  
}

C:\k\android-7.1.1\_r1\system\core\liblog\logd\_writer.c

**static int** logdWrite(log\_id\_t logId, **struct** timespec \*ts,  
 **struct** iovec \*vec, size\_t nr)

## writev

[-> uio.c]

LIBLOG\_ABI\_PUBLIC **int** writev(**int** fd, **const struct** iovec \*vecs, **int** count)  
{  
 **int** total = 0;  
  
 **for** ( ; count > 0; count--, vecs++ ) {  
 **const char**\* buf = vecs->iov\_base;  
 **int** len = vecs->iov\_len;  
  
 **while** (len > 0) {  
 **int** ret = write( fd, buf, len );  
 **if** (ret < 0) {  
 **if** (total == 0)  
 total = -1;  
 **goto** Exit;  
 }  
 **if** (ret == 0)  
 **goto** Exit;  
  
 total += ret;  
 buf += ret;  
 len -= ret;  
 }  
 }  
Exit:  
 **return** total;  
}

## 3.7 小节

一句话总结就是Log.i()最终是通过调用write()向logd守护进程的socket(“/dev/socket/logdw”)端写入需要打印的日志信息。 接下来再来看logd的工作过程。

# logd守护进程

logd是由init进程所启动的守护进程

system/core/logd/logd.rc

创建3个socket通道,用于进程间通信

service logd /system/bin/logd  
  **socket logd stream 0666 logd logd  
 socket logdr seqpacket 0666 logd logd  
 socket logdw dgram 0222 logd logd**  
 group root system readproc  
 writepid /dev/cpuset/system-background/tasks  
  
service logd-reinit /system/bin/logd --reinit  
 oneshot  
 disabled  
 writepid /dev/cpuset/system-background/tasks

### 4.1 main()

[-> /system/core/logd/main.cpp]

int main(int argc, char \*argv[]) {

int fdPmesg = -1;

bool klogd = true;

if (klogd) {

//以只读方式 打开内核log的/proc/kmsg

fdPmesg = open("/proc/kmsg", O\_RDONLY | O\_NDELAY);

}

//以读写方式打开/dev/kmsg

fdDmesg = open("/dev/kmsg", O\_WRONLY);

//处理reinit命令

if ((argc > 1) && argv[1] && !strcmp(argv[1], "--reinit")) {

int sock = TEMP\_FAILURE\_RETRY(socket\_local\_client("logd",

ANDROID\_SOCKET\_NAMESPACE\_RESERVED,SOCK\_STREAM));

...

static const char reinit[] = "reinit";

//写入"reinit"

ssize\_t ret = TEMP\_FAILURE\_RETRY(write(sock, reinit, sizeof(reinit)));

...

struct pollfd p;

memset(&p, 0, sizeof(p));

p.fd = sock;

p.events = POLLIN;

ret = TEMP\_FAILURE\_RETRY(poll(&p, 1, 100)); //进入poll轮询

...

static const char success[] = "success";

char buffer[sizeof(success) - 1];

memset(buffer, 0, sizeof(buffer));

//读取数据,保存到buffer

ret = TEMP\_FAILURE\_RETRY(read(sock, buffer, sizeof(buffer)));

...

//比较读取的数据是否为"success"

return strncmp(buffer, success, sizeof(success) - 1) != 0;

}

sem\_init(&reinit, 0, 0);

sem\_init(&uidName, 0, 0);

sem\_init(&sem\_name, 0, 1);

pthread\_attr\_t attr;

if (!pthread\_attr\_init(&attr)) {

...

if (!pthread\_attr\_setdetachstate(&attr, PTHREAD\_CREATE\_DETACHED)) {

pthread\_t thread;

reinit\_running = true;

//创建线程"logd.daemon", 该线程入口函数reinit\_thread\_start()

if (pthread\_create(&thread, &attr, reinit\_thread\_start, NULL)) {

reinit\_running = false;

}

}

pthread\_attr\_destroy(&attr);

}

...

LastLogTimes \*times = new LastLogTimes();

//创建LogBuffer对象

logBuf = new LogBuffer(times);

signal(SIGHUP, reinit\_signal\_handler);

if (property\_get\_bool\_svelte("logd.statistics")) {

logBuf->enableStatistics();

}

//监听/dev/socket/logdr, 当client连接上则将buffer信息写入client.

LogReader \*reader = new LogReader(logBuf);

if (reader->startListener()) {

exit(1);

}

//监听/dev/socket/logdw, 新日志添加到LogBuffer, 并且LogReader发送更新给已连接的client

LogListener \*swl = new LogListener(logBuf, reader);

if (swl->startListener(300)) {

exit(1);

}

//监听/dev/socket/logd, 处理logd管理命令

CommandListener \*cl = new CommandListener(logBuf, reader, swl);

if (cl->startListener()) {

exit(1);

}

bool auditd = property\_get\_bool("logd.auditd", true);

LogAudit \*al = NULL;

if (auditd) {

bool dmesg = property\_get\_bool("logd.auditd.dmesg", true);

al = new LogAudit(logBuf, reader, dmesg ? fdDmesg : -1);

}

LogKlog \*kl = NULL;

if (klogd) {

kl = new LogKlog(logBuf, reader, fdDmesg, fdPmesg, al != NULL);

}

readDmesg(al, kl);

if (kl && kl->startListener()) {

delete kl;

}

if (al && al->startListener()) {

delete al;

}

TEMP\_FAILURE\_RETRY(pause());

exit(0);

}

该方法功能:

1. LogReader: 监听/dev/socket/logdr, 当client连接上则将buffer信息写入client. 所对应线程名”logd.reader”
2. LogListener: 监听/dev/socket/logdw, 新日志添加到LogBuffer, 并且LogReader发送更新给已连接的client. 所对应线程名”logd.writer”
3. CommandListener: 监听/dev/socket/logd, 处理logd管理命令. 所对应线程名”logd.control”
4. LogAudit: 所对应线程名”logd.auditd”
5. LogKlog: 所对应线程名”logd.klogd”
6. 入口reinit\_thread\_start: 所对应线程名”logd.daemon”
7. LogTimeEntry::threadStart: 所对应线程名”ogd.reader.per”

另外, ANDROID\_SOCKET\_NAMESPACE\_RESERVED代表位于/dev/socket名字空间。 通过adb命令, 可以看到logd进程有9个子线程。

logd 381 1 21880 9132 sigsuspend 7f8301fdac S /system/bin/logd

system 382 381 21880 9132 futex\_wait 7f82fcf9c4 S logd.daemon

logd 383 381 21880 9132 poll\_sched 7f8301fd1c S logd.reader

logd 384 381 21880 9132 poll\_sched 7f8301fd1c S logd.writer

logd 385 381 21880 9132 poll\_sched 7f8301fd1c S logd.control

logd 392 381 21880 9132 poll\_sched 7f8301fd1c S logd.klogd

logd 393 381 21880 9132 poll\_sched 7f8301fd1c S logd.auditd

logd 3716 381 21880 9132 futex\_wait 7f82fcf9c4 S logd.reader.per

logd 4329 381 21880 9132 futex\_wait 7f82fcf9c4 S logd.reader.per

logd 5224 381 21880 9132 futex\_wait 7f82fcf9c4 S logd.reader.per

接下来, 继续回到前面log输出过程, 接下来进入logd的LogListener处理过程, 如下:

### 4.2 LogListener

[-> LogListener.cpp]

int main(int argc, char \*argv[]) {

...

logBuf = new LogBuffer(times); //[见小节4.2.1]

LogListener \*swl = new LogListener(logBuf, reader); //[见小节4.2.2]

if (swl->startListener(300)) { //[见小节4.3]

exit(1);

}

...

}

#### LogBuffer.init

[-> LogBuffer.cpp]

void LogBuffer::init() {

**static const char** global\_tuneable[] = **"persist.logd.size"**; *// Settings App***static const char** global\_default[] = **"ro.logd.size"**; *// BoardConfig.mk*

/获取buffer默认大小

**unsigned long** default\_size = property\_get\_size(global\_tuneable);  
**if** (!default\_size) {  
 default\_size = property\_get\_size(global\_default);  
 **if** (!default\_size) {  
 default\_size = property\_get\_bool(**"ro.config.low\_ram"**,  
 BOOL\_DEFAULT\_FALSE)  
 ? LOG\_BUFFER\_MIN\_SIZE *// 64K* : LOG\_BUFFER\_SIZE; *// 256K* }  
}

log\_id\_for\_each(i) {  
 mLastSet[i] = **false**;  
 mLast[i] = mLogElements.begin();  
  
 **char** key[PROP\_NAME\_MAX];  
//比如获取的是persist.logd.size.system所对应的值  
 snprintf(key, **sizeof**(key), **"%s.%s"**,  
 global\_tuneable, android\_log\_id\_to\_name(i));  
 **unsigned long** property\_size = property\_get\_size(key);  
  
 **if** (!property\_size) {  
 snprintf(key, **sizeof**(key), **"%s.%s"**,  
 global\_default, android\_log\_id\_to\_name(i));  
 property\_size = property\_get\_size(key);  
 }  
  
 **if** (!property\_size) {  
 property\_size = default\_size;  
 }  
  
 **if** (!property\_size) {  
 property\_size = LOG\_BUFFER\_SIZE; //此值为256k  
 }  
  
 **if** (setSize(i, property\_size)) {  
 setSize(i, LOG\_BUFFER\_MIN\_SIZE); //此值为64K

}  
}

buffer大小设置的优先级顺序为:

1. persist.logd.size.xxx; 比如persist.logd.size.system;
2. persist.logd.size;
3. ro.logd.size;
4. LOG\_BUFFER\_SIZE, 即256k;
5. LOG\_BUFFER\_MIN\_SIZE, 即64k。

#### .2.2 LogListener

[-> LogListener.cpp]

LogListener::LogListener(LogBuffer \*buf, LogReader \*reader) :

SocketListener(getLogSocket(), false),

logbuf(buf),

reader(reader) {

}

此处getLogSocket()过程创建logdw的服务端,并监听客户端消息.

#### 4.2.3 SocketListener

SocketListener::SocketListener(int socketFd, bool listen) {

init(NULL, socketFd, listen, false);

}

void SocketListener::init(const char \*socketName, int socketFd, bool listen, bool useCmdNum) {

mListen = listen; // mListen=false

mSocketName = socketName;

mSock = socketFd;

mUseCmdNum = useCmdNum;

pthread\_mutex\_init(&mClientsLock, NULL);

mClients = new SocketClientCollection();

}

SocketListener对象创建完成，则开始执行SocketListener来监听socket请求。

### 4.3 startListener

[-> SocketListener.cpp]

int SocketListener::startListener() {

return startListener(4);

}

int SocketListener::startListener(int backlog) {

if (!mSocketName && mSock == -1) {

...

} else if (mSocketName) {

if ((mSock = android\_get\_control\_socket(mSocketName)) < 0) {

...

}

fcntl(mSock, F\_SETFD, FD\_CLOEXEC);

}

if (mListen && listen(mSock, backlog) < 0) {

return -1;

} else if (!mListen)

mClients->push\_back(new SocketClient(mSock, false, mUseCmdNum));

if (pipe(mCtrlPipe)) {

return -1;

}

//创建线程[见小节4.4]

if (pthread\_create(&mThread, NULL, SocketListener::threadStart, this)) {

return -1;

}

return 0;

}

通过调用pthread\_create创建完线程，在新创建的线程中执行threadStart()过程。

### 4.4 threadStart

[-> SocketListener.cpp]

void \*SocketListener::threadStart(void \*obj) {

SocketListener \*me = reinterpret\_cast<SocketListener \*>(obj);

me->runListener(); //[见小节4.5]

pthread\_exit(NULL);

return NULL;

}

### 4.5 runListener

[-> SocketListener.cpp]

void SocketListener::runListener() {

SocketClientCollection pendingList;

while(1) {

...

while (!pendingList.empty()) {

it = pendingList.begin(); //找到第一个即将要处理的客户端

SocketClient\* c = \*it;

pendingList.erase(it);

if (!onDataAvailable(c)) { //处理该消息[见小节4.6]

release(c, false);

}

c->decRef();

}

}

}

### 4.6 onDataAvailable

[-> LogListener.cpp]

bool LogListener::onDataAvailable(SocketClient \*cli) {

static bool name\_set;

if (!name\_set) {

prctl(PR\_SET\_NAME, "logd.writer");

name\_set = true;

}

char buffer[sizeof\_log\_id\_t + sizeof(uint16\_t) + sizeof(log\_time)

+ LOGGER\_ENTRY\_MAX\_PAYLOAD];

struct iovec iov = { buffer, sizeof(buffer) };

memset(buffer, 0, sizeof(buffer));

char control[CMSG\_SPACE(sizeof(struct ucred))];

struct msghdr hdr = {

NULL,

0,

&iov, //记录buffer地址指针

1,

control,

sizeof(control),

0,

};

int socket = cli->getSocket();

//通过socket接收消息,保存到hdr,其n代表消息长度

ssize\_t n = recvmsg(socket, &hdr, 0);

...

struct ucred \*cred = NULL;

struct cmsghdr \*cmsg = CMSG\_FIRSTHDR(&hdr);

//获取ucred信息

while (cmsg != NULL) {

if (cmsg->cmsg\_level == SOL\_SOCKET

&& cmsg->cmsg\_type == SCM\_CREDENTIALS) {

cred = (struct ucred \*)CMSG\_DATA(cmsg);

break;

}

cmsg = CMSG\_NXTHDR(&hdr, cmsg);

}

if (cred == NULL) {

return false;

}

...

//获取android\_log\_header\_t结构体指针

android\_log\_header\_t \*header = reinterpret\_cast<android\_log\_header\_t \*>(buffer);

if (header->id >= LOG\_ID\_MAX || header->id == LOG\_ID\_KERNEL) {

return false;

}

char \*msg = ((char \*)buffer) + sizeof(android\_log\_header\_t);

n -= sizeof(android\_log\_header\_t);

//[见小节4.7]

if (logbuf->log((log\_id\_t)header->id, header->realtime,

cred->uid, cred->pid, header->tid, msg,

((size\_t) n <= USHRT\_MAX) ? (unsigned short) n : USHRT\_MAX) >= 0) {

//[见小节4.8]

reader->notifyNewLog();

}

return true;

}

LogBuffer.log()的参数说明:

* android\_log\_header\_t提供 log\_id, realtime, tid
* ucred提供 uid, pid.
* msghdr提供 msg

### 4.7 LogBuffer.log

[-> LogBuffer.cpp]

int LogBuffer::log(log\_id\_t log\_id, log\_time realtime,

uid\_t uid, pid\_t pid, pid\_t tid,

const char \*msg, unsigned short len) {

...

//创建一条log信息

LogBufferElement \*elem = new LogBufferElement(log\_id, realtime,

uid, pid, tid, msg, len);

int prio = ANDROID\_LOG\_INFO;

const char \*tag = NULL;

if (log\_id == LOG\_ID\_EVENTS) {

tag = android::tagToName(elem->getTag());

} else {

prio = \*msg;

tag = msg + 1;

}

if (!\_\_android\_log\_is\_loggable(prio, tag, ANDROID\_LOG\_VERBOSE)) {

pthread\_mutex\_lock(&mLogElementsLock);

stats.add(elem); //对于不运行输出log的状态下, 只统计log信息, 记录log本身

stats.subtract(elem);

pthread\_mutex\_unlock(&mLogElementsLock);

delete elem;

return -EACCES;

}

pthread\_mutex\_lock(&mLogElementsLock);

LogBufferElementCollection::iterator it = mLogElements.end();

LogBufferElementCollection::iterator last = it;

//根据时间排序,找到应该插入的点

while (last != mLogElements.begin()) {

--it;

if ((\*it)->getRealTime() <= realtime) {

break;

}

last = it;

}

//将log信息插入合适的位置

if (last == mLogElements.end()) {

mLogElements.push\_back(elem);

} else {

uint64\_t end = 1;

bool end\_set = false;

bool end\_always = false;

LogTimeEntry::lock();

LastLogTimes::iterator t = mTimes.begin();

while(t != mTimes.end()) {

LogTimeEntry \*entry = (\*t);

if (entry->owned\_Locked()) {

if (!entry->mNonBlock) {

end\_always = true;

break;

}

if (!end\_set || (end <= entry->mEnd)) {

end = entry->mEnd;

end\_set = true;

}

}

t++;

}

if (end\_always|| (end\_set && (end >= (\*last)->getSequence()))) {

mLogElements.push\_back(elem);

} else {

mLogElements.insert(last,elem);

}

LogTimeEntry::unlock();

}

stats.add(elem); //[4.7.1]

maybePrune(log\_id); //[4.7.2]

pthread\_mutex\_unlock(&mLogElementsLock);

return len;

}

#### 4.7.1 stats.add

[-> LogStatistics.cpp]

void LogStatistics::add(LogBufferElement \*e) {

log\_id\_t log\_id = e->getLogId();

unsigned short size = e->getMsgLen();

mSizes[log\_id] += size; //对应的buffer所使用大小增加

++mElements[log\_id]; //对应的buffer中log记录加1;

mSizesTotal[log\_id] += size;

++mElementsTotal[log\_id];

if (log\_id == LOG\_ID\_KERNEL) {

return;

}

//以uid为单位, 添加到uidTable表格

uidTable[log\_id].add(e->getUid(), e);

if (!enable) {

return;

}

pidTable.add(e->getPid(), e);

tidTable.add(e->getTid(), e);

uint32\_t tag = e->getTag();

if (tag) {

tagTable.add(tag, e);

}

}

将log信息分别记录到uidTable, pidTable, tidTable, tagTable.

#### 4.7.2 maybePrune

[-> LogBuffer.cpp]

void LogBuffer::maybePrune(log\_id\_t id) {

size\_t sizes = stats.sizes(id); //log占用内存大小

unsigned long maxSize = log\_buffer\_size(id); //最大上限,比如2M

if (sizes > maxSize) {

size\_t sizeOver = sizes - ((maxSize \* 9) / 10); //超出90%的部分大小

size\_t elements = stats.realElements(id); // 真实的log行数

size\_t minElements = elements / 100; // 真实的log行数的1%行

//minPrune = 4, 保证1%的log行数>=4

if (minElements < minPrune) {

minElements = minPrune;

}

unsigned long pruneRows = elements \* sizeOver / sizes;

if (pruneRows < minElements) { //保证>= 1%的log行数

pruneRows = minElements;

}

//maxPrune = 256, 保证pruneRows<=256;

if (pruneRows > maxPrune) {

pruneRows = maxPrune;

}

prune(id, pruneRows); //[见小节4.7.3]

}

}

假设某个buffer的大小为2M:

pruneRows = elements \* sizeOver / sizes

= elements \* (1 - 0.9\*(maxSize/sizes))

= elements \* (1 - 1.8/sizes);

其中elements代表的是当前buffer的log总行数;sizes代表对的是当前buffer的log总大小。

这就意味着某个buffer中的log行数越多，或者log占用内存越大，则需要裁剪的日志行数越多。每次裁剪日志行数等于总行数的10%，并且需要大于等于4行，且不超过256行。

#### 4.7.3 prune

[-> LogBuffer.cpp]

bool LogBuffer::prune(log\_id\_t id, unsigned long pruneRows, uid\_t caller\_uid) {

LogTimeEntry \*oldest = NULL;

bool busy = false;

bool clearAll = pruneRows == ULONG\_MAX;

LogTimeEntry::lock();

LastLogTimes::iterator times = mTimes.begin();

while(times != mTimes.end()) {

LogTimeEntry \*entry = (\*times);

if (entry->owned\_Locked() && entry->isWatching(id)

&& (!oldest ||

(oldest->mStart > entry->mStart) ||

((oldest->mStart == entry->mStart) &&

(entry->mTimeout.tv\_sec || entry->mTimeout.tv\_nsec)))) {

oldest = entry;

}

times++;

}

LogBufferElementCollection::iterator it;

if (caller\_uid != AID\_ROOT) {

it = mLastSet[id] ? mLast[id] : mLogElements.begin();

while (it != mLogElements.end()) {

LogBufferElement \*element = \*it;

if ((element->getLogId() != id) || (element->getUid() != caller\_uid)) {

++it;

continue;

}

if (!mLastSet[id] || ((\*mLast[id])->getLogId() != id)) {

mLast[id] = it;

mLastSet[id] = true;

}

if (oldest && (oldest->mStart <= element->getSequence())) {

busy = true;

if (oldest->mTimeout.tv\_sec || oldest->mTimeout.tv\_nsec) {

oldest->triggerReader\_Locked();

} else {

oldest->triggerSkip\_Locked(id, pruneRows);

}

break;

}

it = erase(it);

pruneRows--;

}

LogTimeEntry::unlock();

return busy;

}

//修剪 log最多的内容: 黑名单, uid, system uid的pid

bool hasBlacklist = (id != LOG\_ID\_SECURITY) && mPrune.naughty();

while (!clearAll && (pruneRows > 0)) {

uid\_t worst = (uid\_t) -1;

size\_t worst\_sizes = 0;

size\_t second\_worst\_sizes = 0;

pid\_t worstPid = 0;

if (worstUidEnabledForLogid(id) && mPrune.worstUidEnabled()) {

{

std::unique\_ptr<const UidEntry \*[]> sorted = stats.sort(

AID\_ROOT, (pid\_t)0, 2, id);

if (sorted.get() && sorted[0] && sorted[1]) {

worst\_sizes = sorted[0]->getSizes();

//buffer总大小的1/8为阈值

size\_t threshold = log\_buffer\_size(id) / 8;

if ((worst\_sizes > threshold)

&& (worst\_sizes > (10 \* sorted[0]->getDropped()))) {

worst = sorted[0]->getKey();

second\_worst\_sizes = sorted[1]->getSizes();

if (second\_worst\_sizes < threshold) {

second\_worst\_sizes = threshold;

}

}

}

}

if ((worst == AID\_SYSTEM) && mPrune.worstPidOfSystemEnabled()) {

// 对于system\_server进程,根据pid来决策

std::unique\_ptr<const PidEntry \*[]> sorted = stats.sort(

worst, (pid\_t)0, 2, id, worst);

if (sorted.get() && sorted[0] && sorted[1]) {

worstPid = sorted[0]->getKey(); //最糟糕的pid

second\_worst\_sizes = worst\_sizes

- sorted[0]->getSizes()

+ sorted[1]->getSizes();

}

}

}

if ((worst == (uid\_t) -1) && !hasBlacklist) {

break;

}

bool kick = false;

bool leading = true;

it = mLastSet[id] ? mLast[id] : mLogElements.begin();

bool gc = pruneRows <= 1;

if (!gc && (worst != (uid\_t) -1)) {

{

LogBufferIteratorMap::iterator found = mLastWorstUid[id].find(worst);

if ((found != mLastWorstUid[id].end())

&& (found->second != mLogElements.end())) {

leading = false;

it = found->second;

}

}

if (worstPid) {

// begin scope for pid worst found iterator

LogBufferPidIteratorMap::iterator found

= mLastWorstPidOfSystem[id].find(worstPid);

if ((found != mLastWorstPidOfSystem[id].end())

&& (found->second != mLogElements.end())) {

leading = false;

it = found->second;

}

}

}

static const timespec too\_old = {

EXPIRE\_HOUR\_THRESHOLD \* 60 \* 60, 0

};

LogBufferElementCollection::iterator lastt;

lastt = mLogElements.end();

--lastt;

LogBufferElementLast last;

while (it != mLogElements.end()) {

LogBufferElement \*element = \*it;

if (oldest && (oldest->mStart <= element->getSequence())) {

busy = true;

if (oldest->mTimeout.tv\_sec || oldest->mTimeout.tv\_nsec) {

oldest->triggerReader\_Locked();

}

break;

}

if (element->getLogId() != id) {

++it;

continue;

}

if (leading && (!mLastSet[id] || ((\*mLast[id])->getLogId() != id))) {

mLast[id] = it;

mLastSet[id] = true;

}

unsigned short dropped = element->getDropped();

// remove any leading drops

if (leading && dropped) {

it = erase(it);

continue;

}

if (dropped && last.coalesce(element, dropped)) {

it = erase(it, true);

continue;

}

if (hasBlacklist && mPrune.naughty(element)) {

last.clear(element);

it = erase(it);

if (dropped) {

continue;

}

pruneRows--;

if (pruneRows == 0) {

break;

}

if (element->getUid() == worst) {

kick = true;

if (worst\_sizes < second\_worst\_sizes) {

break;

}

worst\_sizes -= element->getMsgLen();

}

continue;

}

if ((element->getRealTime() < ((\*lastt)->getRealTime() - too\_old))

|| (element->getRealTime() > (\*lastt)->getRealTime())) {

break;

}

if (dropped) {

last.add(element);

if (worstPid

&& ((!gc && (element->getPid() == worstPid))

|| (mLastWorstPidOfSystem[id].find(element->getPid())

== mLastWorstPidOfSystem[id].end()))) {

mLastWorstPidOfSystem[id][element->getUid()] = it;

}

if ((!gc && !worstPid && (element->getUid() == worst))

|| (mLastWorstUid[id].find(element->getUid())

== mLastWorstUid[id].end())) {

mLastWorstUid[id][element->getUid()] = it;

}

++it;

continue;

}

if ((element->getUid() != worst)

|| (worstPid && (element->getPid() != worstPid))) {

leading = false;

last.clear(element);

++it;

continue;

}

pruneRows--;

if (pruneRows == 0) {

break;

}

kick = true;

unsigned short len = element->getMsgLen();

// do not create any leading drops

if (leading) {

it = erase(it);

} else {

stats.drop(element);

element->setDropped(1);

if (last.coalesce(element, 1)) {

it = erase(it, true);

} else {

last.add(element);

if (worstPid && (!gc

|| (mLastWorstPidOfSystem[id].find(worstPid)

== mLastWorstPidOfSystem[id].end()))) {

mLastWorstPidOfSystem[id][worstPid] = it;

}

if ((!gc && !worstPid) || (mLastWorstUid[id].find(worst)

== mLastWorstUid[id].end())) {

mLastWorstUid[id][worst] = it;

}

++it;

}

}

if (worst\_sizes < second\_worst\_sizes) {

break;

}

worst\_sizes -= len;

}

last.clear();

if (!kick || !mPrune.worstUidEnabled()) {

break; // the following loop will ask bad clients to skip/drop

}

}

bool whitelist = false;

bool hasWhitelist = (id != LOG\_ID\_SECURITY) && mPrune.nice() && !clearAll;

it = mLastSet[id] ? mLast[id] : mLogElements.begin();

while((pruneRows > 0) && (it != mLogElements.end())) {

LogBufferElement \*element = \*it;

if (element->getLogId() != id) {

it++;

continue;

}

if (!mLastSet[id] || ((\*mLast[id])->getLogId() != id)) {

mLast[id] = it;

mLastSet[id] = true;

}

if (oldest && (oldest->mStart <= element->getSequence())) {

busy = true;

if (whitelist) {

break;

}

if (stats.sizes(id) > (2 \* log\_buffer\_size(id))) {

// kick a misbehaving log reader client off the island

oldest->release\_Locked();

} else if (oldest->mTimeout.tv\_sec || oldest->mTimeout.tv\_nsec) {

oldest->triggerReader\_Locked();

} else {

oldest->triggerSkip\_Locked(id, pruneRows);

}

break;

}

if (hasWhitelist && !element->getDropped() && mPrune.nice(element)) {

// WhiteListed

whitelist = true;

it++;

continue;

}

it = erase(it);

pruneRows--;

}

// Do not save the whitelist if we are reader range limited

if (whitelist && (pruneRows > 0)) {

it = mLastSet[id] ? mLast[id] : mLogElements.begin();

while((it != mLogElements.end()) && (pruneRows > 0)) {

LogBufferElement \*element = \*it;

if (element->getLogId() != id) {

++it;

continue;

}

if (!mLastSet[id] || ((\*mLast[id])->getLogId() != id)) {

mLast[id] = it;

mLastSet[id] = true;

}

if (oldest && (oldest->mStart <= element->getSequence())) {

busy = true;

if (stats.sizes(id) > (2 \* log\_buffer\_size(id))) {

// kick a misbehaving log reader client off the island

oldest->release\_Locked();

} else if (oldest->mTimeout.tv\_sec || oldest->mTimeout.tv\_nsec) {

oldest->triggerReader\_Locked();

} else {

oldest->triggerSkip\_Locked(id, pruneRows);

}

break;

}

it = erase(it);

pruneRows--;

}

}

LogTimeEntry::unlock();

return (pruneRows > 0) && busy;

}

在PruneList::init()过程会完成黑白名单.

日志裁剪功能说明：

1. 裁剪黑名单以及log打印最多的那个uid, 以及system uid中打印日志最多的pid进程中的日志;
2. 白名单的不删除

system/core/logd/LogWhiteBlackList.cpp

### 4.8 小节

每一行log记录为LogBufferElement，logd的执行调用链如下

SocketListener::runListener()

LogListener.onDataAvailable

LogBuffer::log

LogBuffer::maybePrune

LogReader::notifyNewLog

SocketListener::runOnEachSocket

FlushCommand::runSocketCommand

## 五. 总结

当日志输出过于频繁或者日志占用内存过大时，会有日志裁剪的动作：每次裁剪日志行数等于总行数的10%，并且需要大于等于4行，且不超过256行，优先裁剪黑名单以及log打印最多的那个uid, 以及system uid中打印日志最多的pid进程中的日志，也可以设置不裁剪的白名单。

buffer大小设置的优先级顺序为:

1. persist.logd.size.xxx; 比如persist.logd.size.system;
2. persist.logd.size;
3. ro.logd.size;
4. LOG\_BUFFER\_SIZE, 即256k;
5. LOG\_BUFFER\_MIN\_SIZE, 即64k。

参数说明：

| **属性名** | **类型** | **默认值** | **含义** |
| --- | --- | --- | --- |
| logd.auditd | bool | true | 启动selinux审核守护进程 |
| logd.auditd.dmesg | bool | true | selinux审核信息发送到dmesg log |
| logd.klogd | bool | depends | 启动klogd守护进程 |
| logd.statistics | bool | depends | 使能logcat -S statistics |
| persist.logd.logpersistd | string |  | 启用logpersist守护进程 |
| persist.logd.size | number | 256K | 所有日志缓存区大小的默认大小 |
| persist.logd.size.main | number | 256K | main日志缓存区大小 |
| persist.logd.size.system | number | 256K | system日志缓存区大小 |
| persist.logd.size.radio | number | 256K | radio日志缓存区大小 |
| persist.logd.size.event | number | 256K | event日志缓存区大小 |
| persist.logd.size.crash | number | 256K | crash日志缓存区大小 |

例如: setprop persist.logd.size.system 2m来调整日志缓存区大小。

# REF

<http://gityuan.com/2018/01/27/android-log/>

# Logcat

在Android中不同的log写到不同的设备中，共有/dev/log/system, /dev/log/main, /dev/log/radion, /dev/log/events四中类型。其中默认Log.v等写入/dev/log/main中。Slog写入/dev/log/system中。

<https://blog.csdn.net/hudashi/article/details/7073155>

## EventLog概述

在调试分析Android的过程中，经常会查看EventLog，它非常简洁明了地展现当前Activity各种状态，当然不至于此，比如还有window、surfaceFlinger、battery等其他的信息。

  如果在终端中输入：logcat -b events，就能够输出大量类似下面的日志信息

**[javascript]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. 07-02 01:38:27.718  3623  3636 I am\_home\_stack\_moved: [0,0,1,1,startingNewTask]
2. 07-02 01:38:27.719  3623  3636 I wm\_task\_moved: [6,1,0]
3. 07-02 01:38:27.722  3623  3636 I am\_create\_task: [0,7]
4. 07-02 01:38:27.722  3623  3636 I am\_create\_activity: [0,224793551,7,com.android.settings/.Settings,android.intent.action
5. .MAIN,NULL,NULL,807403520]
6. 07-02 01:38:27.723  3623  3636 I wm\_task\_created: [7,1]
7. 07-02 01:38:27.723  3623  3636 I wm\_task\_moved: [7,1,1]

通过字面意思，就能得到不少信息量，比如am\_create\_activity，创建activity，但是后面括号中内容的具体含义，其实有很高的价值。既然Event日志辣么辣么重要

在Android源码中，是通过如下语句打印出event日志的：（这里我以ActivityStackSupervisor.java文件说明，其它文件方法一样）

**[java]** [view plain](https://blog.csdn.net/qq1028850792/article/details/80287021) [copy](https://blog.csdn.net/qq1028850792/article/details/80287021)

1. **import** android.util.EventLog;
2. **public** **final** **class** ActivityStackSupervisor **implements** DisplayListener {
3. **void** moveHomeStack(**boolean** toFront, String reason, ActivityStack lastFocusedStack) {
4. EventLog.writeEvent(EventLogTags.AM\_HOME\_STACK\_MOVED,
5. mCurrentUser, toFront ? 1 : 0, stacks.get(topNdx).getStackId(),
6. mFocusedStack == **null** ? -1 : mFocusedStack.getStackId(), reason);
7. }
8. }

EventLog.writeEvent

android\_util\_EventLog\_writeEvent\_Array //android\_util\_EventLog.cpp

android\_bWriteLog // system/core/include/log/Log.h

## 源码分析

### Event TAG

<https://blog.csdn.net/yaowei514473839/article/details/53513435>

日志

EventLog.*writeEvent*(EventLogTags.***LOCK\_SCREEN\_TYPE***, unlockMethod);

# system

Slog.*d*

adb shell logcat -b main -b system -b radio -b events -v time > logcat0630.log

adb bugreport > bugreport0622.txt

adb pull /data/anr

# EventLog

Sd

## 概述

在调试分析Android的过程中，比较常用的地查看EventLog，非常简洁明了地展现当前Activity各种状态，当然不至于此，比如还有window的信息。那么本文就列举以下am相关的tags含义。

本文涉及的源码类有EventLog.java, EventLogTags.java，另外tags格式的定义位于文件/system/etc/event-log-tags。

如果在终端输入：

logcat -b events

那么会输出大量类似这样的信息：

06-01 13:44:55.518 7361 8289 I am\_create\_service: [0,111484394,.StatService,10094,7769]

06-01 13:44:55.540 7361 8343 I am\_proc\_bound: [0,3976,com.android.providers.calendar]

06-01 13:44:55.599 7361 8033 I am\_create\_service: [0,61349752,.UpdateService,10034,1351]

06-01 13:44:55.625 7361 7774 I am\_destroy\_service: [0,61349752,1351]

...

通过字面意思，就能得到不少信息量，比如am\_create\_service，创建service，但是后面括号中内容的具体含义，其实有很高的价值。 接下来通过一张表格来展示含义。

## 二. EventLog

### 2.1 ActivityManager

| **Num** | **TagName** | **格式** | **功能** |
| --- | --- | --- | --- |
| 30001 | am\_finish\_activity | User,Token,TaskID,ComponentName,Reason |  |
| 30002 | am\_task\_to\_front | User,Task |  |
| 30003 | am\_new\_intent | User,Token,TaskID,ComponentName,Action,MIMEType,URI,Flags |  |
| 30004 | am\_create\_task | User ,Task ID |  |
| 30005 | am\_create\_activity | User ,Token ,TaskID ,ComponentName,Action,MIMEType,URI,Flags |  |
| 30006 | am\_restart\_activity | User ,Token ,TaskID,ComponentName |  |
| 30007 | am\_resume\_activity | User ,Token ,TaskID,ComponentName |  |
| 30008 | am\_anr | User ,pid ,Package Name,Flags ,reason | ANR |
| 30009 | am\_activity\_launch\_time | User ,Token ,ComponentName,time |  |
| 30010 | am\_proc\_bound | User ,PID ,ProcessName |  |
| 30011 | am\_proc\_died | User ,PID ,ProcessName |  |
| 30012 | am\_failed\_to\_pause | User ,Token ,Wanting to pause,Currently pausing |  |
| 30013 | am\_pause\_activity | User ,Token ,ComponentName |  |
| 30014 | am\_proc\_start | User ,PID ,UID ,ProcessName,Type,Component |  |
| 30015 | am\_proc\_bad | User ,UID ,ProcessName |  |
| 30016 | am\_proc\_good | User ,UID ,ProcessName |  |
| 30017 | am\_low\_memory | NumProcesses | Lru |
| 30018 | am\_destroy\_activity | User ,Token ,TaskID,ComponentName,Reason |  |
| 30019 | am\_relaunch\_resume\_activity | User ,Token ,TaskID,ComponentName |  |
| 30020 | am\_relaunch\_activity | User ,Token ,TaskID,ComponentName |  |
| 30021 | am\_on\_paused\_called | User ,ComponentName |  |
| 30022 | am\_on\_resume\_called | User ,ComponentName |  |
| 30023 | am\_kill | User ,PID ,ProcessName,OomAdj ,Reason | 杀进程 |
| 30024 | am\_broadcast\_discard\_filter | User ,Broadcast ,Action,ReceiverNumber,BroadcastFilter |  |
| 30025 | am\_broadcast\_discard\_app | User ,Broadcast ,Action,ReceiverNumber,App |  |
| 30030 | am\_create\_service | User ,ServiceRecord ,Name,UID ,PID |  |
| 30031 | am\_destroy\_service | User ,ServiceRecord ,PID |  |
| 30032 | am\_process\_crashed\_too\_much | User ,Name,PID |  |
| 30033 | am\_drop\_process | PID |  |
| 30034 | am\_service\_crashed\_too\_much | User ,Crash Count,ComponentName,PID |  |
| 30035 | am\_schedule\_service\_restart | User ,ComponentName,Time |  |
| 30036 | am\_provider\_lost\_process | User ,Package Name,UID ,Name |  |
| 30037 | am\_process\_start\_timeout | User ,PID ,UID ,ProcessName | timeout |
| 30039 | am\_crash | User ,PID ,ProcessName,Flags ,Exception,Message,File,Line | Crash |
| 30040 | am\_wtf | User ,PID ,ProcessName,Flags ,Tag,Message | Wtf |
| 30041 | am\_switch\_user | id |  |
| 30042 | am\_activity\_fully\_drawn\_time | User ,Token ,ComponentName,time |  |
| 30043 | am\_focused\_activity | User ,ComponentName |  |
| 30044 | am\_home\_stack\_moved | User ,To Front ,Top Stack Id ,Focused Stack Id ,Reason |  |
| 30045 | am\_pre\_boot | User ,Package |  |
| 30046 | am\_meminfo | Cached,Free,Zram,Kernel,Native | 内存 |
| 30047 | am\_pss | Pid, UID, ProcessName, Pss, Uss | 进程 |

下面列举**tag可能使用的部分场景**：

* am\_low\_memory：位于AMS.killAllBackgroundProcesses或者AMS.appDiedLocked，记录当前Lru进程队列长度。
* am\_pss：位于AMS.recordPssSampleLocked(
* am\_meminfo：位于AMS.dumpApplicationMemoryUsage
* am\_proc\_start:位于AMS.startProcessLocked，启动进程
* am\_proc\_bound:位于AMS.attachApplicationLocked
* am\_kill: 位于ProcessRecord.kill，杀掉进程
* am\_anr: 位于AMS.appNotResponding
* am\_crash:位于AMS.handleApplicationCrashInner
* am\_wtf:位于AMS.handleApplicationWtf
* am\_activity\_launch\_time：位于ActivityRecord.reportLaunchTimeLocked()，后面两个参数分别是thisTime和 totalTime.
* am\_activity\_fully\_drawn\_time:位于ActivityRecord.reportFullyDrawnLocked, 后面两个参数分别是thisTime和 totalTime
* am\_broadcast\_discard\_filter:位于BroadcastQueue.logBroadcastReceiverDiscardLocked
* am\_broadcast\_discard\_app:位于BroadcastQueue.logBroadcastReceiverDiscardLocked

Activity生命周期相关的方法:

* am\_on\_resume\_called: 位于AT.performResumeActivity
* am\_on\_paused\_called: 位于AT.performPauseActivity, performDestroyActivity
* am\_resume\_activity: 位于AS.resumeTopActivityInnerLocked
* am\_pause\_activity: 位于AS.startPausingLocked
* am\_finish\_activity: 位于AS.finishActivityLocked, removeHistoryRecordsForAppLocked
* am\_destroy\_activity: 位于AS.destroyActivityLocked
* am\_focused\_activity: 位于AMS.setFocusedActivityLocked, clearFocusedActivity
* am\_restart\_activity: 位于ASS.realStartActivityLocked
* am\_create\_activity: 位于ASS.startActivityUncheckedLocked
* am\_new\_intent: 位于ASS.startActivityUncheckedLocked
* am\_task\_to\_front: 位于AS.moveTaskToFrontLocked

Window相关

* wm\_task\_moved: 位于TaskStack.positionTask()
  + TaskId, toTop ? 1 : 0, position；
  + 206，1，3，是指吧TaskId=206的移动到栈顶(即该栈的长度为4)
* am\_home\_stack\_moved: 位于ASS.moveHomeStack
  + CurrentUser, toFront ? 1:0 , homStackId, FocusedStackId
  + 0,1,0,0, 是指userId=0, home栈顶的StackId=0, 当前focusedStackId=0,

### 2.2 Power

| **Num** | **TagName** | **格式** | **功能** |
| --- | --- | --- | --- |
| 2722 | battery\_level | level, voltage, temperature |  |
| 2723 | battery\_status | status,health,present,plugged,technology |  |
| 2730 | battery\_discharge | duration, minLevel,maxLevel |  |
| 2724 | power\_sleep\_requested | wakeLocksCleared | 唤醒锁数量 |
| 2725 | power\_screen\_broadcast\_send | wakelockCount |  |
| 2726 | power\_screen\_broadcast\_done | on, broadcastDuration, wakelockCount |  |
| 2727 | power\_screen\_broadcast\_stop | which,wakelockCount | 系统还没进入ready状态 |
| 2728 | power\_screen\_state | offOrOn, becauseOfUser, totalTouchDownTime, touchCycles |  |
| 2729 | power\_partial\_wake\_state | releasedorAcquired, tag |  |

部分含义：

* battery\_level: [19,3660,352] //剩余电量19%, 电池电压3.66v, 电池温度35.2℃
* power\_screen\_state: [0,3,0,0] // 灭屏状态(0), 屏幕超时(3). 当然还有其他设备管理策略(1),其他理由都为用户行为(2)
* power\_screen\_state: [1,0,0,0] // 亮屏状态(1)

下面列举**tag可能使用的部分场景**：

* power\_sleep\_requested: 位于PMS.goToSleepNoUpdateLocked
* power\_screen\_state:位于Notifer.handleEarlyInteractiveChange, handleLateInteractiveChange

## 三. EventLog完整语义分析

在源码EventLogTags.java中,有大量类似的定义,那么括号中数字是什么含义呢? (以进程启动为例)

30014 am\_proc\_start (User|1|5),(PID|1|5),(UID|1|5),(Process Name|3),(Type|3),(Component|3)

am\_proc\_start之后紧跟着的几个括号，其中括号里的内容**格式**如下：

(<name>|data type[|data unit])

(<名字>|数据类型[|数据单位])

那么(User|1|5) ==> 名字为User, 数据类型为1，数据单位为5，下面再来看看数据类型和数据单位：

### 3.1 数据类型

* 1: int
* 2: long
* 3: string
* 4: list

数据类型中int和string用得最多.

### 3.2 数据单位

* 1: Number of objects(对象个数)
* 2: Number of bytes(字节数)
* 3: Number of milliseconds(毫秒)
* 4: Number of allocations(分配个数)
* 5: Id
* 6: Percent(百分比)

### 3.3 实例解析

有了前面的准备知识，再来完整的看看如下语句：

am\_proc\_start (User|1|5),(PID|1|5),(UID|1|5),(Process Name|3),(Type|3),(Component|3) am\_proc\_start: [0,9227,10002,com.android.browser,content provider,com.android.browser/.provider.BrowserProvider2]

含义如下：

进程启动: UserId=0, pid=9227, uid=10002, ProcessName=com.android.browser, 数据类型=ContentProvider, 组件=com.android.browser/.provider.BrowserProvider2

## 餐卡破

http://gityuan.com/2016/05/15/event-log/