1. Log的种类

1.kernel log

2.user space log

1. kernel log的使用和抓取

printk的使用方法：

printk(KERN\_ALERT"This is the log printed by printk in linux kernel space.");

抓取：

root@android:/ # cat /proc/kmsg；

klogctl (KLOG\_READ\_ALL, kbuf, sizeof (kbuf) - 1);

1. 用户空间log系统：

Log的使用：

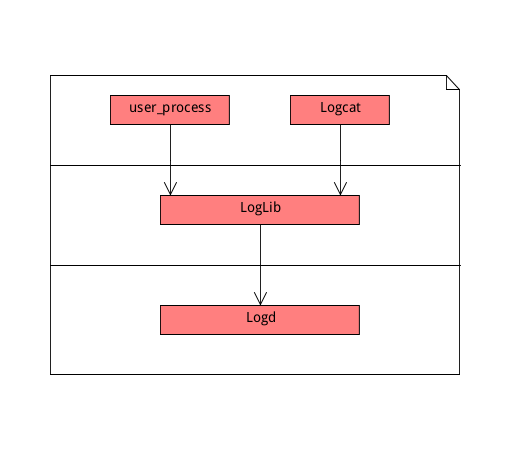
1.native：

包含#include <cutils/log.h>头文件：

ALOGI("SurfaceFlinger is starting");

2.java：

架构：



Log种类：

typedef enum log\_id {

LOG\_ID\_MIN = 0,

#ifndef LINT\_RLOG

LOG\_ID\_MAIN = 0,

#endif

LOG\_ID\_RADIO = 1,

#ifndef LINT\_RLOG

LOG\_ID\_EVENTS = 2,

LOG\_ID\_SYSTEM = 3,

LOG\_ID\_CRASH = 4,

LOG\_ID\_SECURITY = 5,

LOG\_ID\_KERNEL = 6, /\* place last, third-parties can not use it \*/

#endif

LOG\_ID\_MAX

} log\_id\_t;

ALOG：

android\_printLog---->\_\_android\_log\_print

使用LOG\_ID\_MAIN作为bufID。

所有log的buffer不能超过1024的长度。

LIBLOG\_ABI\_PUBLIC int \_\_android\_log\_print(int prio, const char \*tag,

const char \*fmt, ...)

{

va\_list ap;

char buf[LOG\_BUF\_SIZE];

va\_start(ap, fmt);

vsnprintf(buf, LOG\_BUF\_SIZE, fmt, ap);

va\_end(ap);

return \_\_android\_log\_write(prio, tag, buf);

}

LIBLOG\_ABI\_PUBLIC int \_\_android\_log\_write(int prio, const char \*tag,

const char \*msg)

{

return \_\_android\_log\_buf\_write(LOG\_ID\_MAIN, prio, tag, msg);

}

SLOGV：

使用LOG\_ID\_SYSTEM作为bufferID。

\_\_android\_log\_buf\_print(LOG\_ID\_SYSTEM, ANDROID\_LOG\_VERBOSE, LOG\_TAG, \_\_VA\_ARGS\_\_))

RLOGV：LOG\_ID\_RADIO

\_\_android\_log\_buf\_write函数的解析：

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 = "";

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);

}

构造3个元素的iovec的数组分别保存优先级、tag、message.

最后调用write\_to\_log函数指针。

write\_to\_log函数指针的初始化：

第一次调用的时候调用\_\_write\_to\_log\_init函数，在该函数中调用\_\_write\_to\_log\_initialize进行初始化动作和调用\_\_write\_to\_log\_daemon函数开始写入log，同时将write\_to\_log函数指针指向\_\_write\_to\_log\_daemon。

static int (\*write\_to\_log)(log\_id\_t, struct iovec \*vec, size\_t nr) = \_\_write\_to\_log\_init;

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();

if (ret < 0) {

\_\_android\_log\_unlock();

if (!list\_empty(&\_\_android\_log\_persist\_write)) {

\_\_write\_to\_log\_daemon(log\_id, vec, nr);

}

return ret;

}

write\_to\_log = \_\_write\_to\_log\_daemon;

}

\_\_android\_log\_unlock();

return write\_to\_log(log\_id, vec, nr);

}

\_\_write\_to\_log\_daemon函数的解析：

struct android\_log\_transport\_write {

struct listnode node;

const char \*name;

unsigned logMask; /\* cache of available success \*/

union android\_log\_context context; /\* Initialized by static allocation \*/

int (\*available)(log\_id\_t logId);

int (\*open)();

void (\*close)();

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

};

LIBLOG\_HIDDEN struct android\_log\_transport\_write logdLoggerWrite = {

.node = { &logdLoggerWrite.node, &logdLoggerWrite.node },

.context.sock = -1,

.name = "logd",

.available = logdAvailable,

.open = logdOpen,

.close = logdClose,

.write = logdWrite,

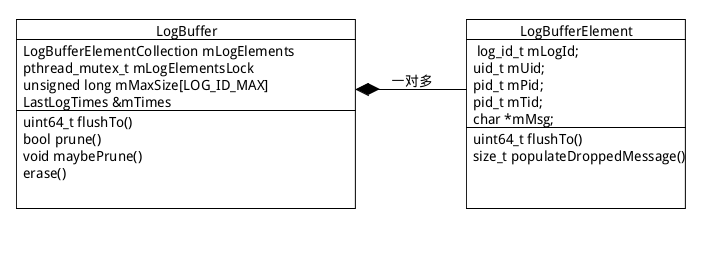
};

通过logdWrite函数使用/dev/socket/logdw作为域套接口进行通信，将log的信息发送到logd进程中。

**Logd的解析：**

1.logd进程通过init进程在开机的时候启动。

1.创建 LogBuffer对象：



为每种的类型的LOG\_ID设置buffer的size。

2.创建LogReader对象：

在/dev/socket目录下创建logdr socket并且启动一个name 为logd.reader的线程在该socket进行监听。当有client端连接该socket返回到onDataAvailable函数。

在该函数中通过FlushCommand为每个client创建一个name 为logd.reader.per线程进行通信，比如一个logcat的连接就创建一个FlushCommand。

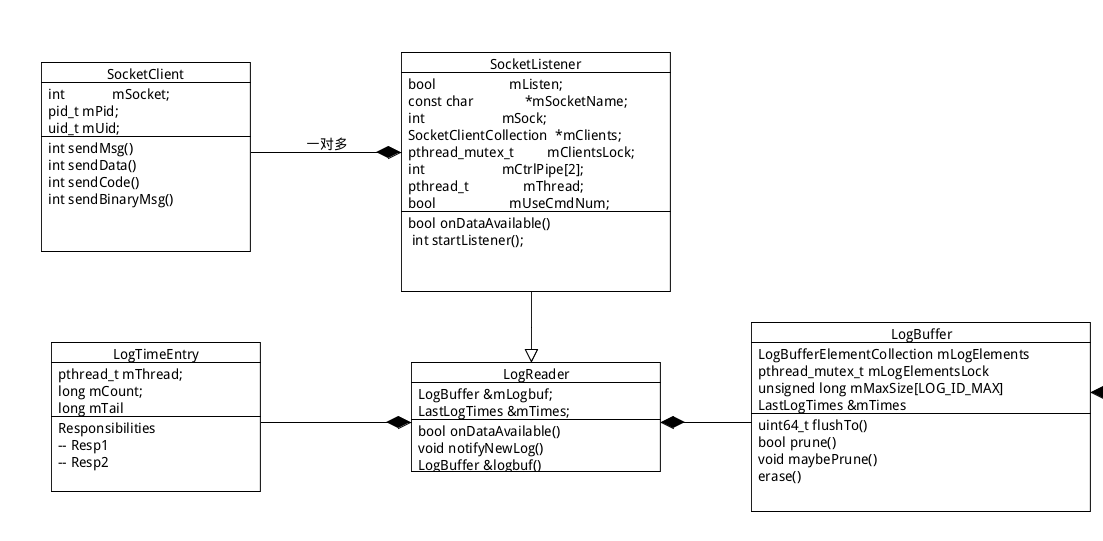
在logd.reader.per线程中通过不断的将LogBuffer中的log信息发送给client，也就是logcat。

LogReader \*reader = new LogReader(logBuf);

if (reader->startListener()) {

exit(1);

}



1. 创建LogListener对象：

// LogListener listens on /dev/socket/logdw for client

// initiated log messages. New log entries are added to LogBuffer

// and LogReader is notified to send updates to connected clients.

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

// Backlog and /proc/sys/net/unix/max\_dgram\_qlen set to large value

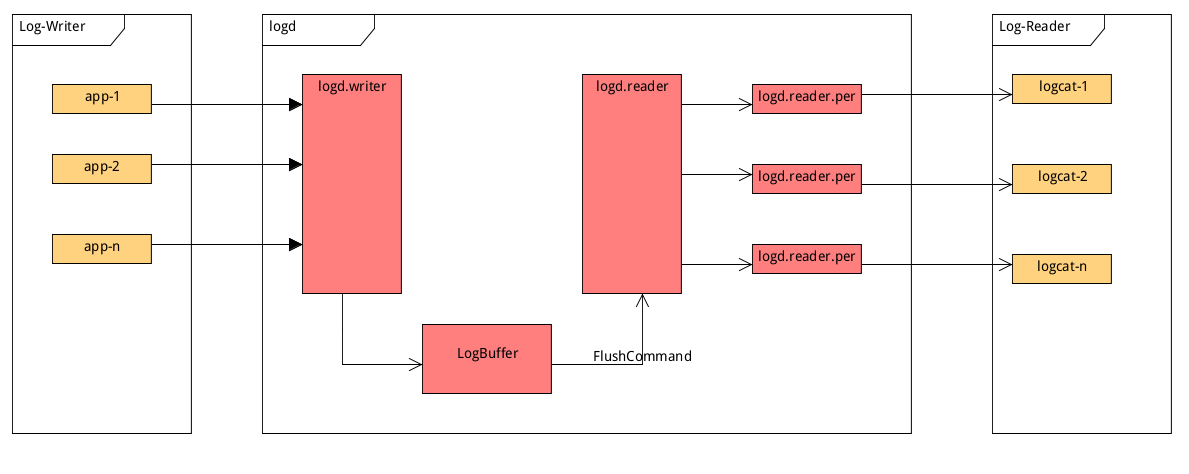
if (swl->startListener(600)) {

exit(1);

}

LogListener和LogReader一样继承于SocketListener，最终创建一个name 为logd.writer的监听线程对创建的/dev/socket/logdw进行监听。当app端由log需要打印出来的时候，将log信息保存到LogBuffer中，同时调用LogReader::notifyNewLog函数通知LogReader中所有的client有读事件。

总结：



Log系统的优化以及log工具的制作：

优化：

<0001-logd-fix-Adjust-socket-parameter-to-fine-tune-LOGD-l.patch>

<0001-logd-feature-Simplify-logd-for-better-performance-an.patch>

工具：

logd 595 1 22976 5044 sigsuspend 0000000000 S /system/bin/logd

root 838 1 5844 1368 do\_wait 0000000000 S /system/bin/htcpreloader

root 913 838 30932 3096 inet\_csk\_a 0000000000 S /system/bin/htcserviced

root 12961 913 6044 1656 \_\_skb\_recv 0000000000 S /system/bin/logcat

root 12970 913 6064 1632 \_\_skb\_recv 0000000000 S /system/bin/logcat

root 12974 913 6044 1652 \_\_skb\_recv 0000000000 S /system/bin/logcat

system 5572 816 1672048 85080 SyS\_epoll\_ 0000000000 S com.htc.android.ssdtest

Logd、htcpreloader、htcserviced在init进程中启动，当htcserviced检查到user打开log开关（通过com.htc.android.ssdtest app写入到system属性中）启动logcat抓取从logd获取的log。

1.log开关的控制

2.输出log类型

3.输出log的格式

root@archermind:/work/log/PERFUME\_UHL\_O80\_SENSE90GP\_MR\_R/log pme 按后台键，回到锁屏界面# adb shell ps -t 913

USER PID PPID VSIZE RSS WCHAN PC NAME

root 913 838 30932 3192 inet\_csk\_a 7f97e73250 S /system/bin/htcserviced

root 3975 913 30932 3192 poll\_sched 7f97e733d0 S uevent

root 3977 913 30932 3192 inet\_csk\_a 7f97e73250 S ghost

root 3978 913 30932 3192 inet\_csk\_a 7f97e73250 S dumpstate

root 3979 913 30932 3192 inet\_csk\_a 7f97e73250 S logctl

root 3980 913 30932 3192 poll\_sched 7f97e733d0 S htc\_if

root 3981 913 30932 3192 poll\_sched 7f97e733d0 S dumpstate:key

root 3982 913 30932 3192 poll\_sched 7f97e733d0 S ghost:monitor

root 3983 913 30932 3192 futex\_wait 7f97e24330 S logctl:monitor

root 3984 913 30932 3192 futex\_wait 7f97e24330 S logctl:killer

root 8822 913 30932 3192 inet\_csk\_a 7f97e73250 S proctrl

root 8842 913 30932 3192 futex\_wait 7f97e24330 S proctrl

root 9016 913 30932 3192 inet\_csk\_a 7f97e73250 S lognet

root 9017 913 30932 3192 futex\_wait 7f97e24330 S lognet:monitor

root 12962 913 30932 3192 pipe\_wait 7f97e73df0 S logdevice:route

root 12964 913 30932 3192 futex\_wait 7f97e24330 S logkmsg:logger

root 12965 913 30932 3192 hrtimer\_na 7f97e73ce8 S logkmsg:timegen

root 12971 913 30932 3192 pipe\_wait 7f97e73df0 S logevents:route

root 12975 913 30932 3192 pipe\_wait 7f97e73df0 S logradio:router

root 12977 913 30932 3192 futex\_wait 7f97e24330 S logmeminfo:logg

root 12978 913 30932 3192 do\_wait 7f97e743c0 S system:am broad