Android6.0 Log 的工作机制

文档修订记录

时间	修改者	版本	备注
2016-12-7	徐玉松	V1.0	新建文档
2016-12-10	徐玉松	V1.1	添加客户端读写
2016-12-13	徐玉松	V2.0	添加服务端

目录

Android6.0log 新机制

Android6.0 后 Android 日志系统做了很大的改变，但是对于应用层改变是透明的，原因是由于日志系统只是针对底层做了相应改变。之前的系统通过读写设备文件的方式记录 Android 系统日志，而现在主要使用 socket 进程间通信读写日志。

Logcat 用法

Android 为我们提供了一个十分方便的命令行工具来读日志。Logcat 是一个命令行工具，用于转储系统消息日志，其中包括设备引发错误时的堆叠追踪以及从您的应用使用 Log 类编写的消息。在 Android6.0 中 logcat 又增加了几个命令来管理日志。接下下来详细介绍一下 logcat 的各个命令的用法。

命令行语法

```
[adb] logcat [<option>] ... [<filter-spec>] ...
```

您可以 adb 命令的形式运行 `logcat`，或在模拟器或所连接设备的 shell 提示符中直接运行。若要使用 adb 查看日志输出，请导航到 SDK `platform-tools/` 目录并执行：

```
$ adb logcat
```

启动 logcat

以下是通过 ADB shell 运行 logcat 的一般用法：

```
[adb] logcat [<option>] ... [<filter-spec>] ...
```

您可以从开发计算机或通过模拟器/设备实例中的远程 adb shell 使用 `logcat` 命令。在开发计算机中查看日志输出可使用

```
$ adb logcat
```

从远程 adb shell 查看日志输出可使用

```
# logcat
```

选项

下表介绍的是 logcat 的命令行选项。

-c	清除（刷新）整个日志并退出。
-d	将日志转储到屏幕并退出。
-f <filename>	将日志消息输出写入 <filename>。默认值为 stdout。
-g	打印指定日志缓冲区的大小并退出。
-n <count>	将已旋转日志的最大数量设置为 <count>。默认值为 4。 需要使用 -r 选项。
-r <kbytes>	每输出 <kbytes> 时旋转日志文件。默认值为 16。需要使用 -f 选项。
-s	将默认过滤器规则设为静默式。
-v <format>	设置日志消息的输出格式。默认值为 brief 格式有关支持的格式列表。
-t <count>	打印最近的<count> 条日志数据（Android6.0 新）
-t <time>	打印特定时间点的日志（Android6.0 新）
-g	打印出环形缓冲区的大小并推出（Android6.0 新）
-L	打印上一次重启的日志信息（Android6.0 新）
-b <buffer>	请求环形缓冲区，'main'，'system'，'radio'， 'events'，'crash '或'all'。 多个-b 参数 允许和结果交错。 默认值为 -b main –b system –b crash。（Android6.0 新）
-B	输出为二进制文件（Android6.0 新）
-S	输出统计日志的情况（Android6.0 新）
-G <size>	设置环形缓冲区的日志大小，可以添加后缀用 K 或者 M（Android6.0 新）
-p	打印设置的白名单和黑名单（Android6.0 新）

`-P <list>`	设置黑名单和白名单（Android6.0 新）

过滤日志输出

每个 Android 日志消息都有与其关联的*标记*和*优先级*。

- 日志消息的标记是一个简短的字符串，其表示消息所源自的系统组件（例如，“View”代表视图系统）。
- 优先级由以下某个字符值表示（按从最低到最高优先级的顺序排列）：
  - `V` — 详细（最低优先级）
  - `D` — 调试
  - `I` — 信息
  - `W` — 警告
  - `E` — 错误
  - `F` — 致命
  - `S` — 静默（最高优先级，不会打印任何内容）

通过运行 logcat 并观察每条消息的前两列，您可以获取系统中使用的标记列表及优先级，格式为 `<priority>/<tag>`。

下面是 logcat 输出的一个示例，其表明消息与优先级“I”和标记“ActivityManager”相关：

```
I/ActivityManager(  585): Starting activity: Intent
{ action=android.intent.action...}
```

若要将日志输出降低到可管理的水平，您可以使用*过滤器表达式*限制日志输出。过滤器表达式允许您向系统表明您感兴趣的标记-优先级组合—系统针对指定的标记阻止其他消息。

过滤器表达式遵循 `tag:priority ...` 这个格式，其中 `tag` 表示感兴趣的标记，`priority` 表示将该标记报告的最低优先级。将优先级等于或高于指定优先级的标记的消息写入日志。您可以在一个过滤器表达式中提供任意数量的 `tag:priority` 规则。一系列规则使用空格分隔。

下面是一个过滤器表达式的示例，该表达式将阻止除了带有标记“ActivityManager”、优先级等于或高于“信息”的日志消息以及带有标记“MyApp”、优先级等于或高于“调试”的日志消息外的所有其他日志消息。

```
adb logcat ActivityManager:I MyApp:D *:S
```

上述表达式中最后一个元素 `*:S` 将所有标记的优先级设为“静默”，从而确保系统仅显示带有“ActivityManager”和“MyApp”标记的日志消息。使用 `*:S` 可有效地确保日志输出受限于您已明确指定的过滤器 — 它允许过滤器充当日志输出的“白名单”。

以下过滤器表达式显示所有标记上优先级等于或高于“警告”的所有日志消息：

```
adb logcat *:W
```

如果您从开发计算机运行 logcat（相对于在远程 adb shell 运行它），您也可以通过导出环境变量 `ANDROID_LOG_TAGS` 的值设置默认过滤器表达式：

```
export ANDROID_LOG_TAGS="ActivityManager:I MyApp:D *:S"
```

请注意，如果您从远程 shell 或使用 `adb shell logcat` 运行 logcat，系统不会将 `ANDROID_LOG_TAGS` 过滤器导出到模拟器/设备实例。

控制日志输出格式

除标记和优先级外，日志消息还包含许多元数据字段。您可以修改消息的输出格式，以便它们可显示特定的元数据字段。为此，您可以使用 `-v` 选项，并指定下面列出的支持的输出格式之一。

- `brief` — 显示优先级/标记以及发出消息的进程的 PID（默认格式）。
- `process` — 仅显示 PID。
- `tag` — 仅显示优先级/标记。
- `raw` — 显示原始日志消息，不显示其他元数据字段。
- `time` — 显示日期、调用时间、优先级/标记以及发出消息的进程的 PID。
- `threadtime` — 显示日期、调用时间、优先级、标记以及发出消息的线程的 PID 和 TID。
- `long` — 显示所有元数据字段，并使用空白行分隔消息。

启动 logcat 时，您可以使用 `-v` 选项指定您需要的输出格式：

```
[adb] logcat [-v <format>]
```

下面的例子展示如何生成 `thread` 输出格式的消息：

```
adb logcat -v thread
```

请注意，使用 -v 选项，您只能指定一个输出格式。

查看日志缓冲区

Android 日志系统保留日志消息的多个循环缓冲区，而不是发送到默认循环缓冲区的所有日志消息。如需查看其他日志消息，您可以使用 -b 选项运行 logcat 命令，以请求查看循环缓冲区。您可以查看下列备用缓冲区的任意一个：

- radio — 查看包含无线装置/电话相关消息的缓冲区。
- events — 查看包含事件相关消息的缓冲区。
- main — 查看主要日志缓冲区（默认值）包括了 system 和 crash 系统默认
- kernel— 查看内核日志

以下是 -b 选项的用法：

```
[adb] logcat [-b <buffer>]
```

以下示例展示如何查看包含无线装置和电话消息的日志缓冲区。

```
adb logcat -b radio
```

查看 stdout 和 stderr

默认情况下，Android 系统将 stdout 和 stderr（System.out 和 System.err）输出发送到 /dev/null。在运行 Dalvik VM 的进程中，您可以让系统将输出的副本写入日志文件。在此情况下，系统使用日志标记 stdout 和 stderr（优先级都是 I）将消息写入日志。

要通过此方式路由输出，您需要停止运行的模拟器/设备实例，然后使用 shell 命令 setprop 以启用输出重定向。下面是具体做法：

```
$ adb shell stop
$ adb shell setprop log.redirect-stdio true
$ adb shell start
```

系统保留此设置，直至您终止模拟器/设备实例。若要在模拟器/设备实例上将此设置用作默认值，您可以在设备上向 `/data/local.prop` 添加一个条目。

通过代码记录日志

Log 类允许您在 logcat 工具中显示的代码中创建日志条目。常用的日志记录方法包括：

- Log.v(String, String)（详细）
- Log.d(String, String)（调试）
- Log.i(String, String)（信息）
- Log.w(String, String)（警告）
- Log.e(String, String)（错误）

例如，使用以下调用：

```
Log.i("MyActivity", "MyClass.getView() — get item number " + position);
```

logcat 输出类似于如下：

```
I/MyActivity( 1557): MyClass.getView() — get item number 1
```

Socket 客户端的读写

Logcat 是一个 C++可执行程序，其位于 Android 系统中/system/bin/logcat,他是整个日志系统读取的入口，logcat 入口是下面的 main 函数：
/system/core/logcat/logcat.cpp

```
ret = getopt(argc, argv, ":cdDLt:T:gG:sQf:r:n:v:b:BSpP:K");
00572: switch(ret) {
00573: case 's':
00574: // default to all silent
00575: android_log_addFilterRule(g_logformat, "*:s");
00576: break;
00578: case 'c':
00579: clearLog = 1;
00580: mode |= ANDROID_LOG_WRONLY;
00581: break;
00587: case 'd':
00588: mode |= ANDROID_LOG_RDONLY | ANDROID_LOG_NONBLOCK;
```

```
00589: break;
00591: case 't':
00594: case 'T':
00628: // MStar Android Patch Begin
00629: case 'K':
00630: android::g_kmsg = 1;
00631: break;
00632: // MStar Android Patch End
00633: ……..
00831:
00832: default:
00833: logcat_panic(true, "Unrecognized Option %c\n", optopt);
00834: break;
```

etopt 被用来解析命令行选项参数。

#include <unistd.h>

extern char *optarg; //选项的参数指针

extern int optind, //下一次调用 getopt 的时，从 optind 存储的位置处重新开始检查选项。

extern int opterr, //当 opterr=0 时，getopt 不向 stderr 输出错误信息。

extern int optopt; //当命令行选项字符不包括在 optstring 中或者选项缺少必要的参数时，该选项存储在 optopt 中，getopt 返回'？'。

int getopt(int argc, char * const argv[], const char *optstring);

调用一次，返回一个选项。 在命令行选项参数再也检查不到 optstring 中包含的选项时，返回－1，同时 optind 储存第一个不包含选项的命令行参数。

首先说一下什么是选项，什么是参数。

1.单个字符，表示选项，

2.单个字符后接一个冒号：表示该选项后必须跟一个参数。参数紧跟在选项后或者以空格隔开。该参数的指针赋给 optarg。

3 单个字符后跟两个冒号，表示该选项后必须跟一个参数。参数必须紧跟在选项后不能以空格隔开。该参数的指针赋给 optarg。（这个特性是 GNU 的扩张）。

获取得到命令行参数后对参数进行解析。不同的参数给予不同的操作模式。比如说 logcat –b main，b 即为选项,main 参数。

`getopt(argc, argv, ":cdDLt:T:gG:sQf:r:n:v:b:BSpP:K")`有冒号即代表其后必须接参数

读取日志

Android6.0 通过使用 socket 进行日志的读写操作，其中 socket 又包含了客户端和服务端，作为 logcat 是读取日志的通道，自然需要实现的 socket 通信的客户端。

重要结构体

要了解其中 log 日志怎么在各个程序传送到控制台的，必须了解整个日志读写全过程。首先我们的日志将会被封装成什么样的结构发送到客户端，同时客户端将会发送怎么样的结构去请求我们需要的日志的。

```
00049: struct log_device_t {
00050: const char* device;
00051: bool binary;
00052: struct logger *logger;
00053: struct logger_list *logger_list;
00054: bool printed;
00056: log_device_t* next;
00058: log_device_t(const char* d, bool b) {
00059: device = d;
00060: binary = b;
00061: next = NULL;
00062: printed = false;
00063: logger = NULL;
00064: logger_list = NULL;
00065: } };
```

不难发现，上面的结构体是一个组成链表的结构体，该链表是用来记录需要处理的日志设备，每个日志设备又对应不同的内存，其中的命名为下面的数组：

\system\core\liblog\log_read.c

```
00205: static const char *LOG_NAME[LOG_ID_MAX] = {
00206: [LOG_ID_MAIN] = "main",
00207: [LOG_ID_RADIO] = "radio",
00208: [LOG_ID_EVENTS] = "events",
00209: [LOG_ID_SYSTEM] = "system",
00210: [LOG_ID_CRASH] = "crash",
00211: [LOG_ID_KERNEL] = "kernel",
00212: };
```

以上的数据便是记录在一个 log_id_t 中，便是如下这个结构体中的 log_id_t 中，这个结构体是组成 logger_list 的一个节点，因为一条命令中可能指定多个缓冲区的数据：例如 logcat –b main kernel

\system\core\liblog\log_read.c

```
00255: struct logger {
00256: struct listnode node;
00257: struct logger_list *top;
00258: log_id_t id;
00259: };
```

如下便是 logger_list。

list_add_tail(&logger_list->node, &logger->node);

\system\core\liblog\log_read.c

```
00246: struct logger_list {
00247: struct listnode node;
00248: int mode;
00249: unsigned int tail;
00250: log_time start;
00251: pid_t pid;
00252: int sock;
00253: };
```

Logger_list 便记录了这个设备需要进行操作的 pid、socket、mode（操作的模式）。其中不同的操作数又对应不同的操作模式：比如

```
00587: case 'd':
00588: mode |= ANDROID_LOG_RDONLY | ANDROID_LOG_NONBLOCK;
```

上面定义的模式 便是使用只读模式和非阻塞模式读取日志。阻塞和非阻塞模式请自行百度了解，这里不展开。
如上只是了解了其中一个参数-b 中读取不同缓冲区中数据的例子，其他参数比较简单，不做展开。

socket 读取日志

那么拿到对应的参数和操作方法后，我们该如何去读取对应的日志呢，答案就是一个死循环。

\system\core\logcat\logcat.cpp

```
01071: while (1) {
01072: // MStar Android Patch Begin
01073: if (android::g_kmsg) {
01074:   int ret = 0;
01075:   char kernel_buffer[1024];
01076: if ((ret = klogctl(KLOG_SIZE_UNREAD, kernel_buffer, sizeof(kernel_buffer))) >
0)      {
01077:       if ((ret = klogctl(KLOG_READ, kernel_buffer, sizeof(kernel_buffer))) > 0)
          {
01078:            if (android::printKernelBuffer(kernel_buffer, ret) < 0) {
01079:          perror("write kernel log error");
01080:       }
01081:    }
01082:   }
01083: }
01084: // MStar Android Patch End
01086:    struct log_msg log_msg;
01087:    log_device_t* d;
```

```
01088:    int ret = android_logger_list_read(logger_list, &log_msg);
01094: if (ret < 0) {
01095:  if (ret == -EAGAIN) {
01096:      break;
01097: }
01098: // MStar Android Patch Begin
01099: if ((connectRetryCount < 10) && (ret == -ECONNREFUSED)) {
01100:     connectRetryCount++;
01101:     printf("read: Connection refused , retry 10 times(%d).", connectRetryCount);
01102:     usleep(100000);
01103:     continue;
01104: }
01105: // MStar Android Patch End
01107: if (ret == -EIO) {
01108:     logcat_panic(false, "read: unexpected EOF!\n");
01109: }
01116: for(d = devices; d; d = d->next) {
01117:     if (android_name_to_log_id(d->device) == log_msg.id()) {
01118:     break;
01119:  }
01120: }
01136: } ? end while 1 ?
```

该循环中主要是由 android_logger_list_read(logger_list, &log_msg);读取，并将结果写入到了 log_msg 中返回。这个读取过程是怎样的呢：

```
00739: /* Read from the selected logs */
00740: int android_logger_list_read(struct logger_list *logger_list,
00741: struct log_msg *log_msg)
00742: {
00743: int ret, e;
00744: struct logger *logger;
00745: struct sigaction ignore;
00746: struct sigaction old_sigaction;
00747: unsigned int old_alarm = 0;
00764: if (logger_list->sock < 0) {
00765:       char buffer[256], *cp, c;
00767:       int sock = socket_local_client("logdr",
00768:         ANDROID_SOCKET_NAMESPACE_RESERVED,SOCK_SEQPACKET);
00774:        return sock;
00775: }
00776:
00777: strcpy(buffer,
00778: (logger_list->mode & ANDROID_LOG_NONBLOCK) ? "dumpAndClose" : "stream");
00779: cp = buffer + strlen(buffer);
```

```
            . . . . . .
00816: if (logger_list->mode & ANDROID_LOG_NONBLOCK) {
00817: /* Deal with an unresponsive logd */
00818: //sigaction是一个函数，可以用来查询或设置信号处理方式。
00819:    sigaction(SIGALRM, &ignore, &old_sigaction);
00820:    old_alarm = alarm(30);
00821: }
00822: ret = write(sock, buffer, cp - buffer);
00832: if (ret <= 0) {
00833:     close(sock);
00834:     if ((ret == -1) && e) {
00835:         return -e;
00836: }
00837: if (ret == 0) {
00838: return -EIO;
00839: }
00840: return ret;
00841: }
00842:
00843: logger_list->sock = sock;
00844: } ? end if logger_list->sock<0 ?
00846: ret = 0;
00847: while(1) {
00848:     memset(log_msg, 0, sizeof(*log_msg));
00849:
00850:    if (logger_list->mode & ANDROID_LOG_NONBLOCK) {
00851:        /* particularily useful if tombstone is reporting for logd */
00852:        sigaction(SIGALRM, &ignore, &old_sigaction);
00853:        old_alarm = alarm(30);
00854: }
00855:     /* NOTE: SOCK_SEQPACKET guarantees we read exactly one full entry */
00856:    ret = recv(logger_list->sock, log_msg, LOGGER_ENTRY_MAX_LEN, 0);
00863:     alarm(old_alarm);
00874: logger_for_each(logger, logger_list) {
00875: if (log_msg->entry.lid == logger->id) {
00876:       return ret;
00877:     }
00878:   }
00879: } ? end while 1 ?
00880: /* NOTREACH */
00881: return ret;
00882: } ? end android_logger_list_read ?
```

上面便是，整个读取日志的过程，其主要是新建一个 socket 连接，通过 socket

发送请求参数，服务端返回日志的形式进行。
分为以下三步：
第一步：新建连接：

```
int sock =socket_local_client("logdr",ANDROID_SOCKET_NAMESPACE_RESERVED,SOCK_SEQPACKET);
```

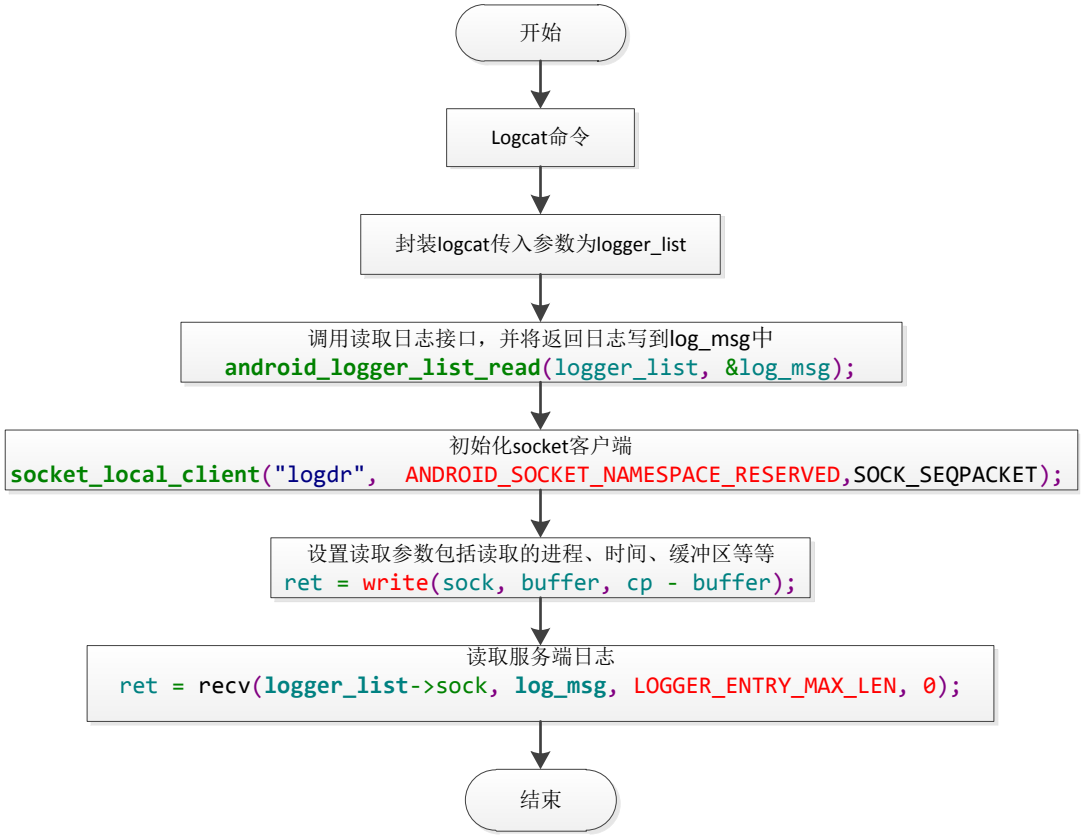
返回一个socket句柄。并且设置连接属性，SOCK_SEQPACKET使用数据包的形式进行通信，能确保其传输的可靠性。
第二步：设置参数：

```
strcpy(buffer, (logger_list->mode & ANDROID_LOG_NONBLOCK) ? "dumpAndClose" : "stream");
ret = write(sock, buffer, cp - buffer);
```

这两句话就是往服务器设置我们需要读取日志的方式。我们使用非阻塞，模式读取日志。除此之外我们还应根据需要设置其读取的进程、时间、缓冲区等等参数。
第三步：读取日志：

```
/* NOTE: SOCK_SEQPACKET guarantees we read exactly one full entry */
ret = recv(logger_list->sock, log_msg, LOGGER_ENTRY_MAX_LEN, 0);
```

从注释中可以看到，我们使用SOCK_SEQPACKET模式传输能确保我们能得到一个完整日志实体。通过recv函数可以将套接字中的数据读取出来。并存放到对用的buff中即log_msg。
如上便是整个日志的读取流程。如下流程图所示：



图一 客户端读取日志

日志写入过程

当然是用工 Log 类的读者都知道，我们在 Java 或者 Android 代码中简单的使用

Log.i(“myTAG”,”this is my log”)；类似这样的例子就可以往系统中写入日志，那么这些日志被写到哪里了，如何写入的呢。
首先我们看 framework 层中是如何定义这些接口的：
\frameworks\base\core\java\android\util\Log.java

```
public static int i(String tag, String msg) {
return println_native(LOG_ID_MAIN, INFO, tag, msg);
}
```

首先在 framework 中调用本地方法 println_native(LOG_ID_MAIN, INFO, tag, msg);，这个方法被注册在 Java 虚拟机中：
\frameworks\base\core\jni\android_util_Log.cpp

```
00114: static JNINativeMethod gMethods[] = {
00115: /* name, signature, funcPtr */
00116: { "isLoggable", "(Ljava/lang/String;I)Z", (void*) android_util_Log_isLoggable },
00117: { "println_native", "(IILjava/lang/String;Ljava/lang/String;)I", (void*)
00117: android_util_Log_println_native },
00118: };
```

在本地方法中注册 println_native 为 android_util_Log_println_native 方法，其中具体实现就调用了写日志的方法：
\frameworks\base\core\jni\android_util_Log.cpp

```
00082: static jint android_util_Log_println_native(JNIEnv* env, jobject
clazz,
00083: jint bufID, jint priority, jstring tagObj, jstring msgObj)
00101:              . . . . . .
00102: int res = __android_log_buf_write(bufID, (android_LogPriority)priority, tag, msg);
00103:
00104: if (tag != NULL)
00105: env->ReleaseStringUTFChars(tagObj, tag);
00106: env->ReleaseStringUTFChars(msgObj, msg);
00107:
00108: return res;
00109: } ? end android_util_Log_println_native ?
```

其中主要的方法便是__android_log_buf_write(bufID, (android_LogPriority)priority, tag, msg)他调用了底层写日志的方法。

图二 各个缓冲区对应类

其中 bufID 确定使用哪个缓冲区（main、system 等）中的数据，priority 代表日志的优先级（I、E 等）,tag 表示标签，msg 表示写入的日志。不同缓冲区的 Log 对应不同的类，比如 main 对应的类为 Log 而 system 对应的为 Slog，在顶层实现方面，Android6.0 没有改变，这里不过多介绍。我们详细看一下 __android_log_buf_write(bufID, (android_LogPriority)priority, tag, msg);的实现原理。其中先调用了 __write_to_log_initialize 初始化连接，然后调用 __write_to_log_daemon 往服务端写日志。

__write_to_log_initialize 函数：

```
00094: static int ___write_to_log_initialize()
00095: {
00098: #if FAKE_LOG_DEVICE
00100:    for (i = 0; i < LOG_ID_MAX; i++) {
00101:         char buf[sizeof("/dev/log_system")];
00102:         snprintf(buf, sizeof(buf), "/dev/log_%s", android_log_id_to_name(i));
00103:         log_fds[i] = fakeLogOpen(buf, O_WRONLY);
00104: }
00106: #else
00112: if (logd_fd < 0) {
00113: i = TEMP_FAILURE_RETRY(socket(PF_UNIX, SOCK_DGRAM | SOCK_CLOEXEC, 0));
00114: if (i < 0) {
00115:   ret = -errno;
00144: } ? end if i<0 ? else if (TEMP_FAILURE_RETRY(fcntl(i, F_SETFL, O_NONBLOCK)) <
0) {
00145:   ret = -errno;
00146:   close(i);
00147: } else {
00148:   struct sockaddr_un un;
00149:    memset(&un, 0, sizeof(struct sockaddr_un));
00155:    un.sun_family = AF_UNIX;
00156:    strcpy(un.sun_path, "/dev/socket/logdw");
```

```
00158: if (TEMP_FAILURE_RETRY(connect(i, (struct sockaddr *)&un,
00159: sizeof(struct sockaddr_un))) < 0) {
00160:     ret = -errno;
00161:     close(i);
00162:  } else {
00163:  logd_fd = i;
00164: }
00165: }
```

其中使用到很多进程通信相关的机制，如下做简单介绍：

一、创建 socket 流程

（1）创建 socket，类型为 AF_LOCAL 或 AF_UNIX(地址族)，PF_UNIX 或者 PF_LOCAL（协议族），均表示用于进程通信：

创建套接字需要使用 socket 系统调用，其原型如下：

int socket(int domain, int type, int protocol);

其中，domain 参数指定协议族，对于本地套接字来说，其值须被置为 AF_UNIX 枚举值；type 参数指定套接字类型，protocol 参数指定具体协议；type 参数可被设置为 SOCK_STREAM（流式套接字）或 SOCK_DGRAM（数据报式套接字），protocol 字段应被设置为 0；其返回值为生成的套接字描述符。

对于本地套接字来说，流式套接字（SOCK_STREAM）是一个有顺序的、可靠的双向字节流，相当于在本地进程之间建立起一条数据通道；数据报式套接字（SOCK_DGRAM）相当于单纯的发送消息，在进程通信过程中，理论上可能会有信息丢失、复制或者不按先后次序到达的情况，但由于其在本地通信，不通过外界网络，这些情况出现的概率很小。

二、命名 socket。

SOCK_STREAM 式本地套接字的通信双方均需要具有本地地址，其中服务器端的本地地址需要明确指定，指定方法是使用 struct sockaddr_un 类型的变量。

```
struct sockaddr_un {
    sa_family_t     sun_family;     /* AF_UNIX */
    char    sun_path[UNIX_PATH_MAX];        /* 路径名 */
};
```

这里面有一个很关键的东西，socket 进程通信命名方式有两种。一是普通的命名，socket 会根据此命名创建一个同名的 socket 文件，客户端连接的时候通过读取该 socket 文件连接到 socket 服务端。这种方式的弊端是服务端必须对 socket 文件的路径具备写权限，客户端必须知道 socket 文件路径，且必须对该路径有读权限。另外一种命名方式是抽象命名空间，这种方式不需要创建 socket 文件，只需要命名一个全局名字，即可让客户端根据此名字进行连接。后者的实现过程与前者的差别是，后者在对地址结构成员 sun_path 数组赋值的时候，必须把第一个字节置 0，即 sun_path[0] = 0，下面用代码说明：

第一种方式：

```
1.  //name the server socket
2.      server_addr.sun_family = AF_UNIX;
3.      strcpy(server_addr.sun_path," /dev/socket/logdw ");
4.      server_len = sizeof(struct sockaddr_un);
5.      client_len = server_len;
```

第二种方式：

```
1.  //name the socket
2.     server_addr.sun_family = AF_UNIX;
3.     strcpy(server_addr.sun_path, SERVER_NAME);
4.     server_addr.sun_path[0]=0;
5.     //server_len = sizeof(server_addr);
6.     server_len = strlen(SERVER_NAME)  + offsetof(struct sockaddr_un, sun_path
    );
```

1、获取文件的 flags，即 open 函数的第二个参数:

flags = fcntl(fd,F_GETFL,0);

2、设置文件的 flags:

fcntl(fd,F_SETFL,flags);

3、增加文件的某个 flags，比如文件是阻塞的，想设置成非阻塞:

flags = fcntl(fd,F_GETFL,0);

flags |= O_NONBLOCK;

fcntl(fd,F_SETFL,flags);

4、取消文件的某个 flags，比如文件是非阻塞的，想设置成为阻塞:

flags = fcntl(fd,F_GETFL,0);

flags &= ~O_NONBLOCK;

fcntl(fd,F_SETFL,flags);

有了上述基本知识，我们可以明显的看出我们 log 读取方式就是使用的本地进程间通信的方式，而且采取协议 SOCK_DGRAM，使用数据报的方式，而在建立连接的过程中，采用的第二种方式，通过文件的方式与服务端取得连接，细心的读者可能会发现在读取日志的时候，获取连接是使用这么一段代码：

```
\system\core\liblog\log_read.c
00112: case ANDROID_SOCKET_NAMESPACE_RESERVED:
00113: namelen = strlen(name) + strlen(ANDROID_RESERVED_SOCKET_PREFIX);
00114: /* unix_path_max appears to be missing on linux */
00115: if (namelen > sizeof(*p_addr)
00116: - offsetof(struct sockaddr_un, sun_path) - 1) {
00117: goto  error;
00118: }
```

使用的是第一种方式与服务器建立连接的，那究竟为什么在读取的时候不和写入的时候用同一种方式呢，由于日志写入的过程是在程序中进行的，然而日志的读取是通过 logcat 命令去获取的，当你用普通用户去运行命令时，如果对 /dev/socket/logdw 下的文件没有读写的权限，就不能获取日志了，因此在读取日志的过程中使用的第二种方式。相关参数初始化完成后，调用

__write_to_log_daemon 写日志。

__write_to_log_daemon 函数：

\system\core\liblog\logd_write.c

```
00172: static int ___write_to_log_daemon(log_id_t log_id, struct iovec *vec,
size_t nr)
00173: {
00174: ssize_t ret;
00175: #if FAKE_LOG_DEVICE
00176: int log_fd;
00177:
00178: if (/*(int)log_id >= 0 &&*/ (int)log_id < (int)LOG_ID_MAX) {
00179: log_fd = log_fds[(int)log_id];
00180: } else {
00181: return -EBADF;
00182: }
00183: do {
00184:     ret = fakeLogWritev(log_fd, vec, nr);
00185:   if (ret < 0) {
00186:     ret = -errno;
00187:    }
00188: } while (ret == -EINTR);
00189: #else
00190:    static const unsigned header_length = 2;
00191:    struct iovec newVec[nr + header_length];
00192:    android_log_header_t header;
00193:    android_pmsg_log_header_t pmsg_header;
00194:    struct timespec ts;
00195:    size_t i, payload_size;
00196:    static uid_t last_uid = AID_ROOT; /* logd *always* starts up as AID_ROOT */
00197:    static pid_t last_pid = (pid_t) -1;
00198:    static atomic_int_fast32_t dropped;
00266:    header.id = log_id;
00267:             . . . .
00300: /*
00301: * The write below could be lost, but will never block.
00302: *
00303: * To logd, we drop the pmsg_header
00304: *
00305: * ENOTCONN occurs if logd dies.
00306: * EAGAIN occurs if logd is overloaded.
00307: */
00308:    ret = TEMP_FAILURE_RETRY(writev(logd_fd, newVec + 1, i - 1));
00309:    if (ret < 0) {
00310:      ret = -errno;
```

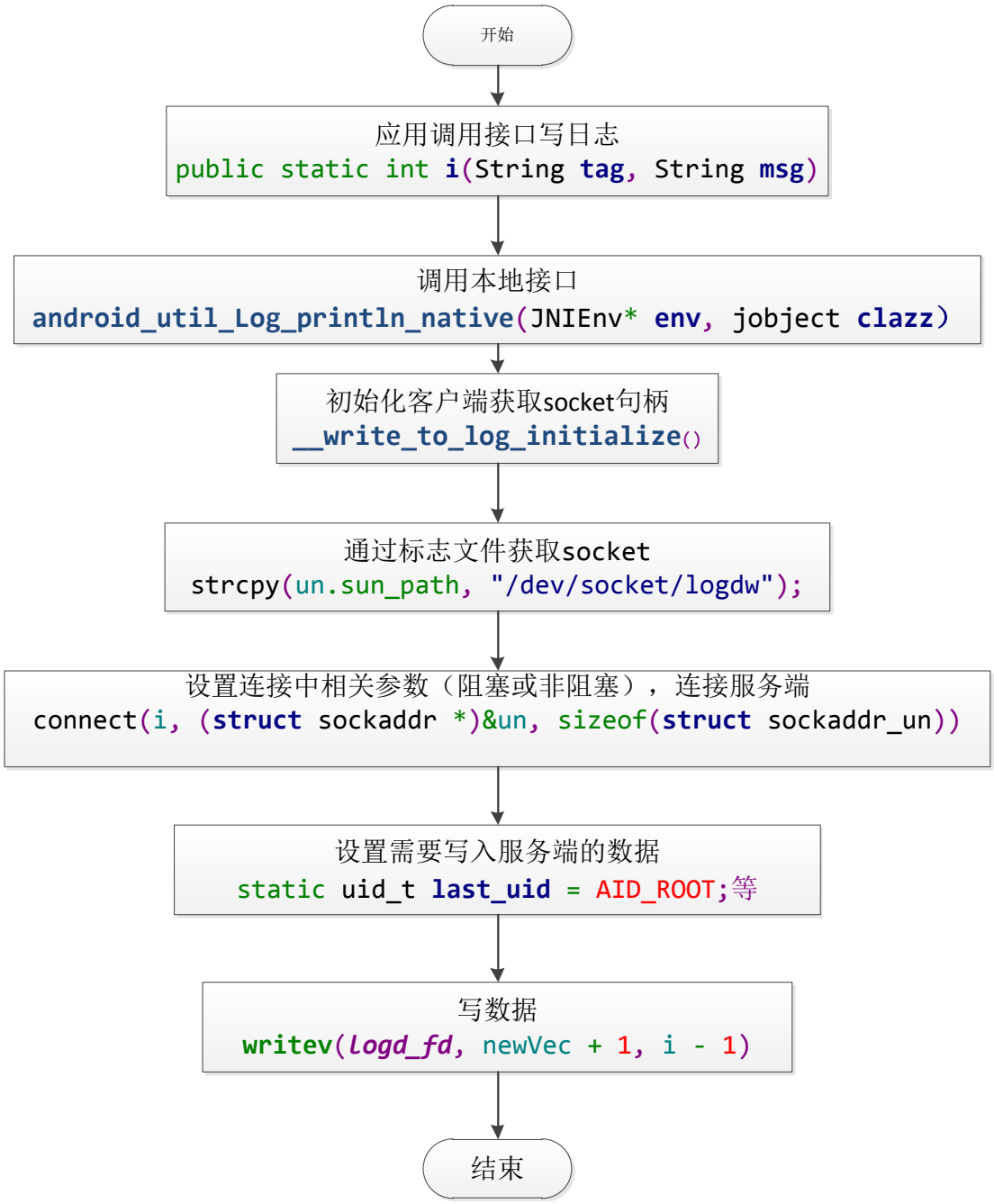
```
00311:       if (ret == -ENOTCONN) {
00315:          close(logd_fd);
00316:          logd_fd = -1;
00317:        ret = __write_to_log_initialize();
00326:        ret = TEMP_FAILURE_RETRY(writev(logd_fd, newVec + 1, i - 1));
00327:     if (ret < 0) {
00328:        ret = -errno;
00329:    }
00330:    } ? end if ret==-ENOTCONN ?
00331: } ? end if ret<0 ?
00332:
00333: if (ret > (ssize_t)sizeof(header)) {
00334:     ret -= sizeof(header);
00335:   } else if (ret == -EAGAIN) {
00336:      atomic_fetch_add_explicit(&dropped, 1, memory_order_relaxed);
00337: }
00338: #endif
00340: return ret;

00341: } ? end __write_to_log_daemon ?
```

当然刚开始，我们都只是出事化一些参数，这里不做分析，我们关心的只是一些重要的点，首先在初始化的时候我们回去到了一个 socket *logd_fd* 句柄，他是关于进程 logd 的一个句柄，这个进程便是 socket 进程，用于守护整个日志系统的通信，当然这个进程是由服务端启动和维护的，之后在讲解服务端的时候会具体讲解到。对应于读取函数 `recv(logger_list->sock, log_msg, LOGGER_ENTRY_MAX_LEN, 0);`在写入日志时 我们用到的是 `writev(logd_fd, newVec + 1, i - 1);`用于向服务端写入相应数据。其中需要注意的一点：函数返回

由于使用的是 SOCK_DGRAM 数据报的形式，所有的写入操作都是可能丢失的，但是这个函数不会阻塞，当 logd 进程如果被意外杀死，返回 ENOTCONN，表示连接错误，并且重新初始化连接，返回 EAGAIN 表示 socket 过载，至于为什么会过载，在服务端会详细讲解到，执行 `atomic_fetch_add_explicit(&dropped, 1, memory_order_relaxed);`在单线程中删除当前操作，这便是整个日志写入过程。

流程如图所示：



图三 客户端写日志

Socket 服务端的启动和读写

讲了半天只是讲了客户端，之前一直提到服务端，那么服务端是如何启动的？启动的时机是什么？

Socket 服务启动

Socket 服务是一个 logd 服务，是在开机启动的。

```
130|shell@synsepalum:/ $ ps | grep logd
logd      1598  1     14056  4620  sigsuspend 0000000000 S /system/bin/logd
shell@synsepalum:/ $
```

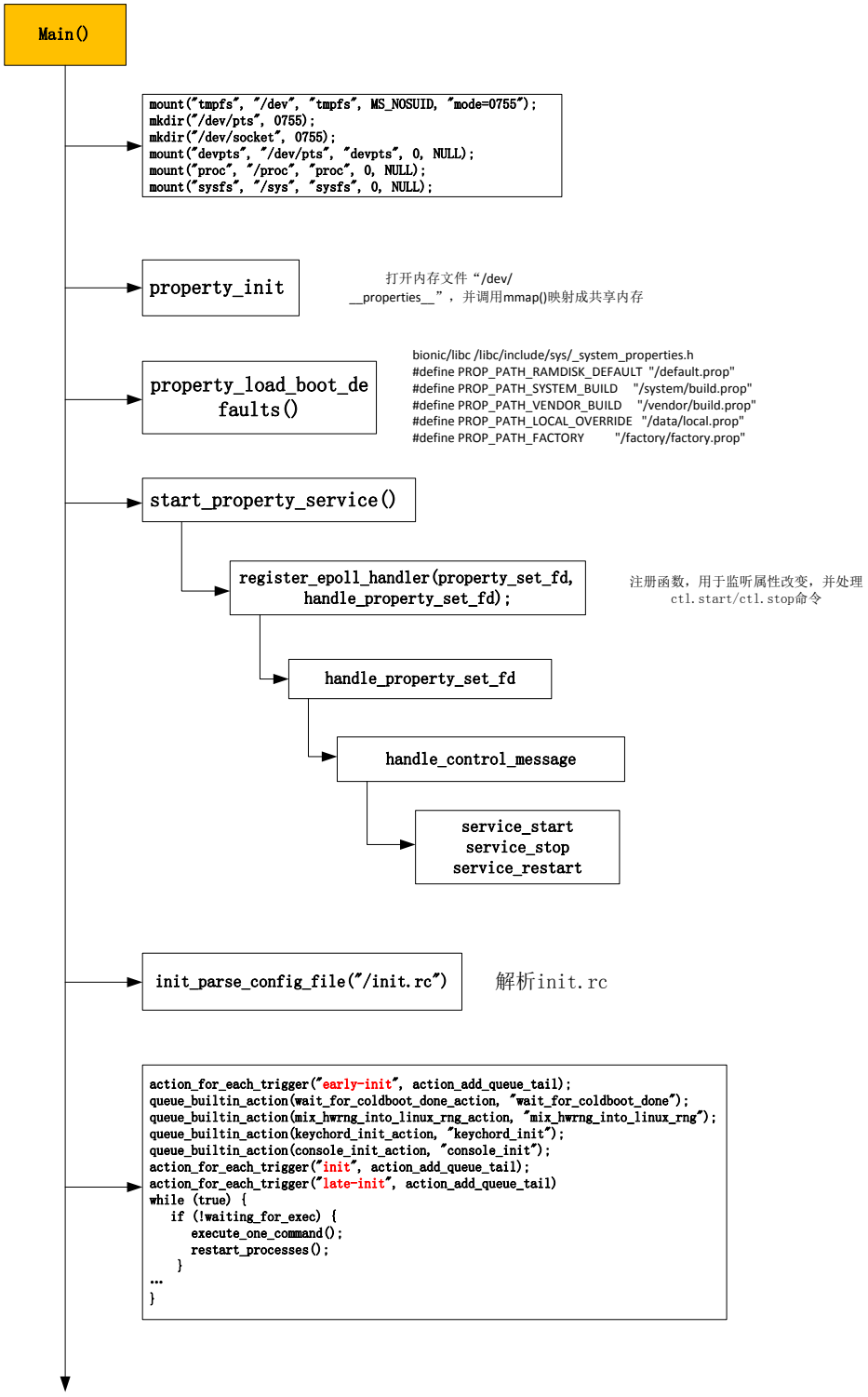
图四 logd进程

开机时系统会先启动 Linux 内核，在此过程中会先启动 init 进程。该进程中，主要任务：1、创建文件系统（只创建启动必须的，其他的在 init.rc 创建）

2、属性服务

3、init.rc 文件解析

4、启动服务



图五 开机init进程

那么我们所说的 socket 服务端进程在什么地方启动的呢，就在 init.rc 文件之中创建的:

Init.rc 文件

启动 logd 服务

on load_all_props_action

load_all_props

start logd

start logd-reinit

创建用于 logcat 连接服务端 socket 服务，并且在/dev/socket 下建立了 logd、logdr、logdw 三个文件。用于 logcat 用来连接。

service logd /system/bin/logd

class core

socket logd stream 0666 logd logd

socket logdr seqpacket 0666 logd logd

socket logdw dgram 0222 logd logd

group root system

那么我们说的如上所示的几种模式创建的 socket 服务

SOCK_STREAM： 提供面向连接的稳定数据传输，即 TCP 协议。

SOCK_DGRAM： 使用不连续不可靠的数据包连接。

SOCK_SEQPACKET： 提供连续可靠的数据包连接。

SOCK_RAW： 提供原始网络协议存取。

SOCK_RDM： 提供可靠的数据包连接。

SOCK_PACKET： 与网络驱动程序直接通信。

而 socket 这个设备文件是在 init 进程中创建的：

\system\core\init\init.cpp

```
01033: int main(int argc, char** argv) {
01049: // Get the basic filesystem setup we need put together in the initramdisk
01050: // on / and then we'll let the rc file figure out the rest.
01051: if (is_first_stage) {
01052:   mount("tmpfs", "/dev", "tmpfs", MS_NOSUID, "mode=0755");
01053:   mkdir("/dev/pts", 0755);
01054:   mkdir("/dev/socket", 0755);
01055:   mount("devpts", "/dev/pts", "devpts", 0, NULL);
01056:   mount("proc", "/proc", "proc", 0, NULL);
01057:   mount("sysfs", "/sys", "sysfs", 0, NULL);
01058: }
```

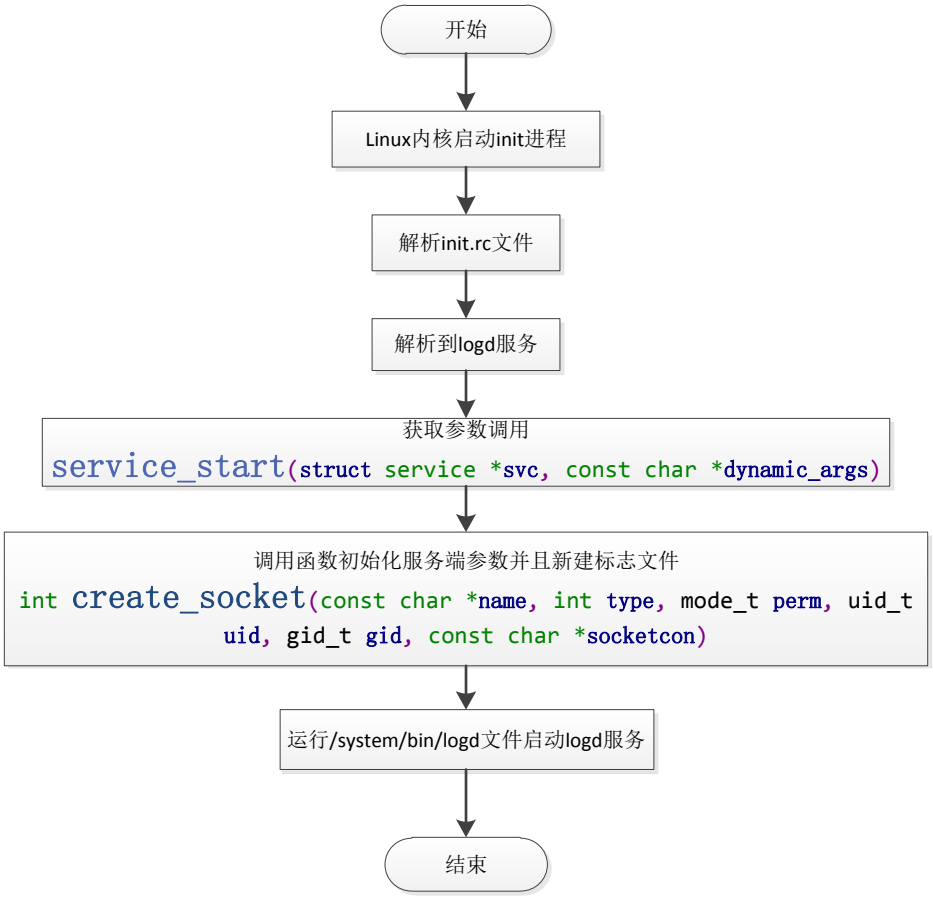
而在每次启动服务的时候，都会通过 init.cpp 中的 service_start 方法，其中就有创建 socket 服务的方法：create_socket();其中的定义为：

\system\core\init\util.cpp

```
00091: int create_socket(const char *name, int type, mode_t perm, uid_t uid,
00092: gid_t gid, const char *socketcon)
```

```
00093: {
00094: struct sockaddr_un addr;
00098: if (socketcon)
00099: setsockcreatecon(socketcon);
00100:
00101: fd = socket(PF_UNIX, type, 0);
00110: memset(&addr, 0 , sizeof(addr));
00111: addr.sun_family = AF_UNIX;
00112: snprintf(addr.sun_path, sizeof(addr.sun_path), ANDROID_SOCKET_DIR"/%s",
00113: name);
00119: }
00128: ret = bind(fd, (struct sockaddr *) &addr, sizeof (addr));
00129: if (ret) {
00131: goto  out_unlink;
00132: }
00137: chown(addr.sun_path, uid, gid);
00138: chmod(addr.sun_path, perm);
00139:
00140: INFO("Created socket '%s' with mode '%o', user '%d', group '%d'\n",
00141: addr.sun_path, perm, uid, gid);
00143: return fd;
00145: out_unlink:
00146: unlink(addr.sun_path);
00147: out_close:
00148: close(fd);
00149: return -1;
00150: } ? end create_socket ?
```

如上所示在init进程中创建了socket连接，`addr.sun_family = AF_UNIX`表示使用来进行进程间通信的。`snprintf(addr.sun_path, sizeof(addr.sun_path), ANDROID_SOCKET_DIR"/%s", name);` `ANDROID_SOCKET_DIR=”/dev/socket”` 根据在init.rc文件中定义的logd、logdr、logdw三个socket，通过bind后就会在/dev/socket下新建这三个文件，提供给logcat连接服务端。除了构建这些参数外，还启动了logd这个可执行文件。



图六 socket服务启动

这个文件是在 system/core/logd 下编译，其中的 Makefile 是：

```
LOCAL_PATH:= $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE:= logd

LOCAL_SRC_FILES := \
        main.cpp \
        LogCommand.cpp \
        CommandListener.cpp \
        LogListener.cpp \
        LogReader.cpp \
        FlushCommand.cpp \
        LogBuffer.cpp \
        LogBufferElement.cpp \
        LogTimes.cpp \
        LogStatistics.cpp \
        LogWhiteBlackList.cpp \
        libaudit.c \
        LogAudit.cpp \
```

```
    LogKlog.cpp \
    event.logtags

LOCAL_SHARED_LIBRARIES := \
    libsysutils \
    liblog \
    libcutils \
    libutils

# This is what we want to do:
#  event_logtags = $(shell \
#    sed -n \
#      "s/^\([0-9]*\)[ \t]*$1[ \t].*/-D`echo $1 | tr a-z A-Z`_LOG_TAG=\1/p" \
#      $(LOCAL_PATH)/$2/event.logtags)
#  event_flag := $(call event_logtags,auditd)
#  event_flag += $(call event_logtags,logd)
# so make sure we do not regret hard-coding it as follows:
event_flag := -DAUDITD_LOG_TAG=1003 -DLOGD_LOG_TAG=1004

LOCAL_CFLAGS := -Werror $(event_flag)

include $(BUILD_EXECUTABLE)

include $(CLEAR_VARS)

LOCAL_MODULE := logpersist.start
LOCAL_MODULE_TAGS := debug
LOCAL_MODULE_CLASS := EXECUTABLES
LOCAL_MODULE_PATH := $(bin_dir)
LOCAL_SRC_FILES := logpersist
ALL_TOOLS := logpersist.start logpersist.stop logpersist.cat
LOCAL_POST_INSTALL_CMD := $(hide) $(foreach t,$(filter-out $(LOCAL_MODULE),$(ALL_TOOLS)),ln -sf $(LOCAL_MODULE) $(TARGET_OUT)/bin/$(t);)
include $(BUILD_PREBUILT)

include $(call first-makefiles-under,$(LOCAL_PATH))
```

上面讲整个 logd 下的文件编译成为了一个名为 logd 的可执行文件，然后我们在 init.rc 中以服务的方式去启动它，如此整个 socket 服务端也就完全启动。

服务端监听器启动

那么这个服务端会处理什么呢：

1、首先是能够处理由 logcat 发送过来的控制台命令，是通过 logd 文件标识的

socket 来就收和处理的。比如获取缓冲区的大小，缓冲区的日志情况等等。
2、其次是能够监听来自客户的读日志的请求，并且处理返回相应的值。
3、最后是能够监听所有来自系统或者应用等程序中写日志的请求。

启动监听：在 logd 的 mian.cpp 中启动了三个监听器，用来监听上面的三个方面的请求。

\system\core\logd\main.cpp

```
00331: int main(int argc, char *argv[]) {
00332: int fdPmesg = -1;
00333: bool klogd = property_get_bool_svelte("logd.klogd");
00334: if (klogd) {
00335:   fdPmesg = open("/proc/kmsg", O_RDONLY | O_NDELAY);
00336: }
00337: fdDmesg = open("/dev/kmsg", O_WRONLY);
00339: // issue reinit command. KISS argument parsing.
00340: if ((argc > 1) && argv[1] && !strcmp(argv[1], "--reinit")) {
00341: int sock = TEMP_FAILURE_RETRY(
00342: socket_local_client("logd",
00343: ANDROID_SOCKET_NAMESPACE_RESERVED,
00344: SOCK_STREAM));
00345: if (sock < 0) {
00346: return -errno;
00347: }
00404: LastLogTimes *times = new LastLogTimes();
00406: // LogBuffer is the object which is responsible for holding all
00407: // log entries.
00408:
00409: logBuf = new LogBuffer(times);
00410:
00411: signal(SIGHUP, reinit_signal_handler);
00412:
00413: if (property_get_bool_svelte("logd.statistics")) {
00414: logBuf->enableStatistics();
00415: }
00417: // LogReader listens on /dev/socket/logdr. When a client
00418: // connects, log entries in the LogBuffer are written to the client.
00419:
00420: LogReader *reader = new LogReader(logBuf);
00421: if (reader->startListener()) {
00422: exit(1);
00423: }
00425: // LogListener listens on /dev/socket/logdw for client
00426: // initiated log messages. New log entries are added to LogBuffer
00427: // and LogReader is notified to send updates to connected clients.
```

```
00429: LogListener *swl = new LogListener(logBuf, reader);
00430: // Backlog and /proc/sys/net/unix/max_dgram_qlen set to large value
00431: if (swl->startListener(600)) {
00432: exit(1);
00433: }
00435: // Command listener listens on /dev/socket/logd for incoming logd
00436: // administrative commands.
00438: CommandListener *cl = new CommandListener(logBuf, reader, swl);
00439: if (cl->startListener()) {
00440: exit(1);
00441: }
00472: TEMP_FAILURE_RETRY(pause());
00473:
00474: exit(0);
00475: } ? end main ?
```

三个监听器分别为

（一）`LogReader *reader = new LogReader(logBuf);`

`reader->startListener()`：监听/dev/socket/logdr 上的 socket 客户端，每当有读请求到时就返回 log entries 给客户端。

（二） `LogListener *swl = new LogListener(logBuf, reader);`

`swl->startListener(600)`:监听/dev/socket/logdw 上的 socket 客户端请求，每当有日志写入时，就添加 log entries 到 LogBuffer 中，而且通知 logReader 读取日志。其中 600 表示负载，表示能同时处理 600 个来自客户端的写入请求，如果需要写入的日志较多，可以适当的增加这个值，这个值的最大值在 `/proc/sys/net/unix/max_dgram_qlen` 文件中修改。

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

`cl->startListener()`: 监听/dev/socket/logd来自客户端的命令，用于管理客户端的命令。其管理命令有如下这些：

\system\core\logd\CommandListener.cpp

```
00035: CommandListener::CommandListener(LogBuffer *buf, LogReader *
/*reader*/,
00036: LogListener * /*swl*/) :
00037: FrameworkListener(getLogSocket()),
00038: mBuf(*buf) {
00039: // registerCmd(new ShutdownCmd(buf, writer, swl));
00040: registerCmd(new ClearCmd(buf));
00041: registerCmd(new GetBufSizeCmd(buf));
00042: registerCmd(new SetBufSizeCmd(buf));
00043: registerCmd(new GetBufSizeUsedCmd(buf));
00044: registerCmd(new GetStatisticsCmd(buf));
00045: registerCmd(new SetPruneListCmd(buf));
```

```
00046: registerCmd(new GetPruneListCmd(buf));
00047: registerCmd(new ReinitCmd());
00048: }
```

当然在客户端也必须有对应的写入操作：客户端 只需每次调用 send_log_msg 就可以向服务端发送命令行指令，比如向服务端发送清除日志命令：

\system\core\liblog\log_read.c

```
00411: int android_logger_clear(struct logger *logger)
00412: {
00413: char buf[512];
00414:
00415: if (logger->top->mode & ANDROID_LOG_PSTORE) {
00416: if (uid_has_log_permission(get_best_effective_uid())) {
00417: return unlink("/sys/fs/pstore/pmsg-ramoops-0");
00418: }
            . . . .
00422: return check_log_success(buf,
00423: send_log_msg(logger, "clear %d", buf, sizeof(buf)));
00424: }
```

客户端发送指令：

\system\core\liblog\log_read.c

```
00283: static ssize_t send_log_msg(struct logger *logger,
00284: const char *msg, char *buf, size_t buf_size)
00285: {
00286: ssize_t ret;
00287: size_t len;
00290: int sock = socket_local_client("logd", ANDROID_SOCKET_NAMESPACE_RESERVED,
00291: SOCK_STREAM);
00296: if (msg) {
00297: snprintf(buf, buf_size, msg, logger ? logger->id : (unsigned) -1);
00298: }
00299:
00300: len = strlen(buf) + 1;
00301: ret = TEMP_FAILURE_RETRY(write(sock, buf, len));
00305:
00306: len = buf_size;
00307: cp = buf;
00308: while ((ret = TEMP_FAILURE_RETRY(read(sock, cp, len))) > 0) {
00309: struct pollfd p;
          . . . .
00339: done:
00340: if ((ret == -1) && errno) {
00341: errno_save = errno;
00342: }
```

```
00343: close(sock);
00344: if (errno_save) {
00345: errno = errno_save;
00346: }
00347: return ret;
00348: } ? end send_log_msg ?
```

具体的过程这里不多做解释，与读写日志的流程一致。当然细心的读者可能会发现，这个日志系统还有一个监听器，专门使用来监听内核日志的：

```
00443: // LogAudit listens on NETLINK_AUDIT socket for selinux
00444: // initiated log messages. New log entries are added to LogBuffer
00445: // and LogReader is notified to send updates to connected clients.
00446:
00447: bool auditd = property_get_bool("logd.auditd", true);
00448:
00449: LogAudit *al = NULL;
00450: if (auditd) {
00451: bool dmesg = property_get_bool("logd.auditd.dmesg", true);
00452: al = new LogAudit(logBuf, reader, dmesg ? fdDmesg : -1);
00453: }
00454:
00455: LogKlog *kl = NULL;
00456: if (klogd) {
00457: kl = new LogKlog(logBuf, reader, fdDmesg, fdPmesg, al != NULL);
00458: }
00459:
00460: readDmesg(al, kl);
00461:
00462: // failure is an option ... messages are in dmesg (required by standard)
00463:
00464: if (kl && kl->startListener()) {
00465: delete kl;
00466: }
00467:
00468: if (al && al->startListener()) {
00469: delete al;
00470: }
00471:
```

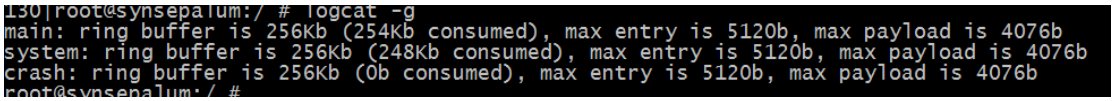
其中涉及到 seLinux 权限系统，需要有权限的用户才能读取，而且系统会有专门的文件(dev/dmsg)记录该日志，我们只需读取该文件就行，这里不多展开。我们只详细讲解普通日志的读写过程：

环形 buffer 初始化

首先我们发现有一个专门的 `LogBuffer *logBuf = new LogBuffer(times);`用来存储日志，其初始化过程为：
主要是给 LogBuffer 设置大小。

```
00095: void LogBuffer::init() {
00096: //256k set in
/packages/apps/Settings/src/com/android/settings/DevelopmentSettings.java
00097: static const char global_tuneable[] = "persist.logd.size"; // Settings App
00098: //
00099: static const char global_default[] = "ro.logd.size"; // BoardConfig.mk
00100:
00101: unsigned long default_size = property_get_size(global_tuneable);
00102: if (!default_size) {
00103: default_size = property_get_size(global_default);
00104: }
00105:
00106: log_id_for_each(i) {
00107: char key[PROP_NAME_MAX];
00108:
00109: snprintf(key, sizeof(key), "%s.%s",
00110: global_tuneable, android_log_id_to_name(i));
00111: unsigned long property_size = property_get_size(key);
00112:
00113: if (!property_size) {
00114: snprintf(key, sizeof(key), "%s.%s",
00115: global_default, android_log_id_to_name(i));
00116: property_size = property_get_size(key);
00117: }
00118:
00119: if (!property_size) {
00120: property_size = default_size;
00121: }
00122:
00123: if (!property_size) {
00124: property_size = LOG_BUFFER_SIZE;
00125: }
00126:
00127: if (setSize(i, property_size)) {
00128: setSize(i, LOG_BUFFER_MIN_SIZE);
00129: }
00130: }
00131: } ? end init ?
```

经过全局搜索发现persist.logd.size在/apps/Settings/src/com/android/settings/DevelopmentSettings.java中定义的其大小为256K而ro.logd.size并没有在makefile文件中定义，除此之外注意log_id_for_each(i)是为每个缓冲区定义大小，最后我们分析可以得出，对每种类型的buffer都设置了256K大小的空间，这点可一个通过logcat -g 验证。



图七 查看环形缓冲区

有了这个空间后，我们就可以往这个空间写数据，而且，还可以从这个空间往外读取数据，这个空间相当于一个环形空间，一旦被初始化后，就一直可以重复利用，有些时候会出现，写得太快，而读取太慢导致有些日志数据会被覆盖的可能，这是正常情况，读者可以通过过滤一些日志的方法来避免日志被覆盖的可能。

响应客户端读取日志

有了这个 buffer 后，如何使用呢，首先我们讲解如何在监听到客户端读操作的时候，如何将 buffer 中的日志写回客户端。从监听器上我们发现通过一个LogReader *reader = new LogReader(*logBuf*)这么一个类进行处理的。

1、提供一个接口供外部用来提示我们是否需要提醒客户端有新的日志写入，需要读取日志了。

\system\core\logd\LogReader.cpp

```
00033: // all of our listening sockets.
00034: void LogReader::notifyNewLog() {
00035: FlushCommand command(*this);
00036: runOnEachSocket(&command);
00037: }
```

2、提供客户端读取日志的接口，并能接收客户端请求参数，返回对应的日志。

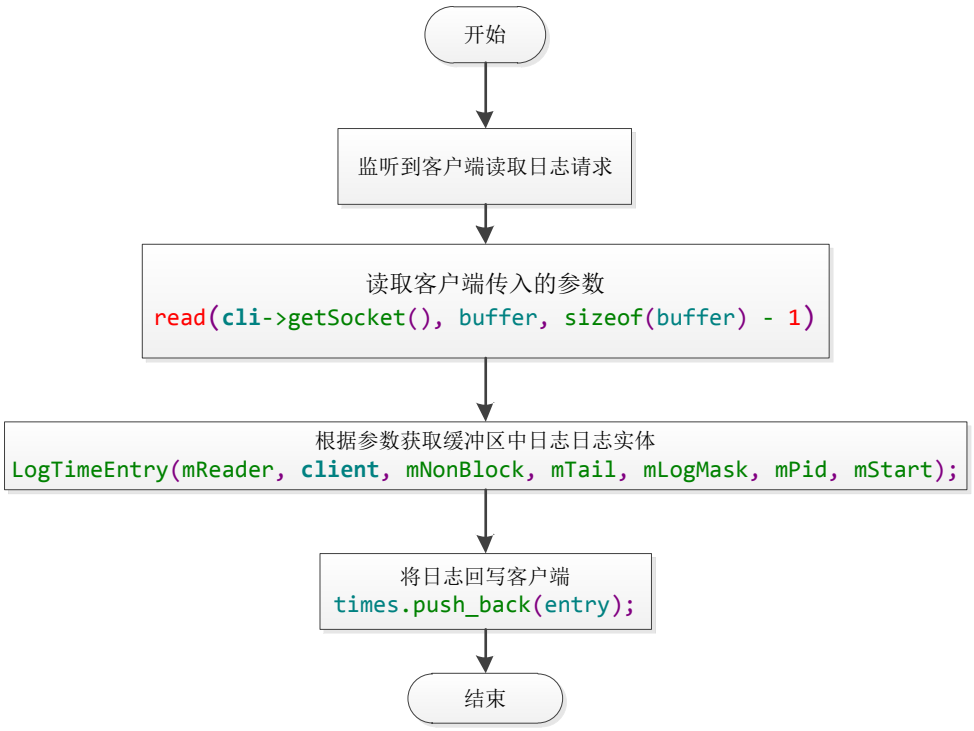
```
00039: bool LogReader::onDataAvailable(SocketClient *cli) {
00040: static bool name_set;
00041: if (!name_set) {
         //响应日志读操作
00042:   prctl(PR_SET_NAME, "logd.reader");
00043:   name_set = true;
00044: }
          //读取客户端传入参数
00048: int len = read(cli->getSocket(), buffer, sizeof(buffer) - 1);
00049: if (len <= 0) {
00050: doSocketDelete(cli);
00051: return false;
00052: }
00053: buffer[len] = '\0';
00054:     . . . . 省略一些参数设置
```

```
00151: logbuf().flushTo(cli, sequence, FlushCommand::hasReadLogs(cli),
00152: logFindStart.callback, &logFindStart);
00153:
00154: if (!logFindStart.found()) {
00155: if (nonBlock) {
00156: doSocketDelete(cli);
00157: return false;
00158: }
00159: sequence = LogBufferElement::getCurrentSequence();
00160: }
00161: } ? end if start!=log_time.EPOCH ?
00162:
00163: FlushCommand command(*this, nonBlock, tail, logMask, pid, sequence);
       客户
00164: command.runSocketCommand(cli);
00165: return true;
00166: } ? end onDataAvailable ?
```

先通过 `read(cli->getSocket(), buffer, sizeof(buffer) - 1)`获取到客户端传入的数据在 `command.runSocketCommand(cli);`向客户端写日志。具体写入方法是：

```
00048: void FlushCommand::runSocketCommand(SocketClient *client) {
00049: LogTimeEntry *entry = NULL;
00050: LastLogTimes &times = mReader.logbuf().mTimes;
              . . .
00074: entry = new LogTimeEntry(mReader, client, mNonBlock, mTail, mLogMask, mPid,
mStart);
00075: times.push_back(entry);
00076: }
00077:
00078: client->incRef();
00079:
00080: // release client and entry reference counts once done
00081: entry->startReader_Locked();
00082: LogTimeEntry::unlock();
00083: } ? end runSocketCommand ?
```

`times.push_back(entry);`会将整个读取到的日志回写到客户端中。客户端收到后输出对应位置。
流程图：



图八 服务端响应客户端读日志

响应客户端读取日志

当客户端需要读取日志时，向服务端发送请求，监听器通过 `LogListener *swl = new LogListener(logBuf, reader);`监听客户端请求，其具体实现方式为：

\system\core\logd\LogListener.cpp

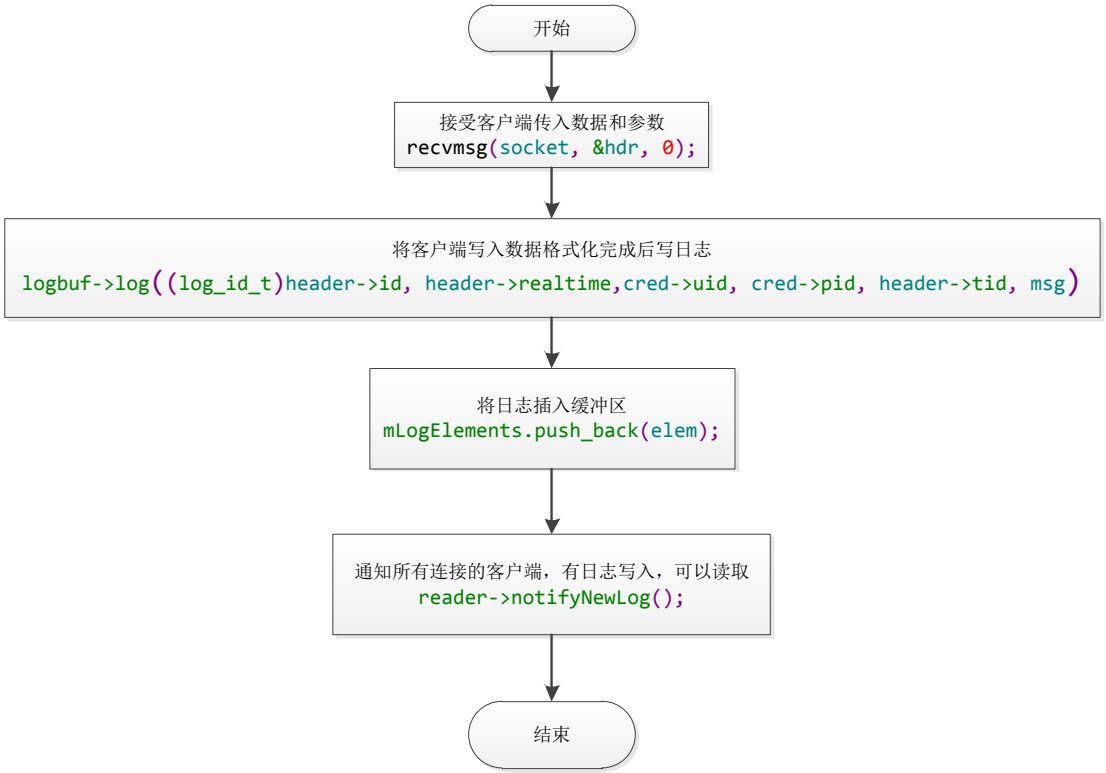
```
bool LogListener::onDataAvailable(SocketClient *cli) {
00038: static bool name_set;
00039: if (!name_set) {
      //响应写操作
00040: prctl(PR_SET_NAME, "logd.writer");
00041: name_set = true;
00042: }
00060: int socket = cli->getSocket();
       //接收数据
00062: ssize_t n = recvmsg(socket, &hdr, 0);
            . . . . 省略其中数据的处理
      //将日志写入缓冲区
00101: if (logbuf->log((log_id_t)header->id, header->realtime,
00102: cred->uid, cred->pid, header->tid, msg,
00103: ((size_t) n <= USHRT_MAX) ? (unsigned short) n : USHRT_MAX) >= 0) {
       //通知所有连接的客户端有新日志写入，可以读了
00104: reader->notifyNewLog();
00105: }
00106:
```

```
00107: return true;} ? end onDataAvailable ?
```

第一步：从 socket 中获取到客户端传入的数据
第二步：初始化好数据
第三步：将日志写入到对应的 buffer 中
第四步：通知所有客户端，有新日志写入可以读取。
写入buffer的`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)`函数实现：其主要用来写LogBuffer的。
\system\core\logd\LogBuffer.cpp

```
00139: int LogBuffer::log(log_id_t log_id, log_time realtime,
00140: uid_t uid, pid_t pid, pid_t tid,
00141: const char *msg, unsigned short len) {
            . . . . .省略参数设置
00197: if (!end_set || (end <= entry->mEnd)) {
00198: end = entry->mEnd;
00199: end_set = true;
00200: }
00201: }
00202: t++;
00203: }
00204:
00205: if (end_always
00206: || (end_set && (end >= (*last)->getSequence()))) {
00207: mLogElements.push_back(elem);
00208: } else {
00209: mLogElements.insert(last,elem);
00210: }
00211:
00212: LogTimeEntry::unlock();
00213: } ? end else ?
00214:
00215: stats.add(elem);
00216: maybePrune(log_id);
00217: pthread_mutex_unlock(&mLogElementsLock);
00218:
00219: return len;
00220: } ? end log ?
```

上面的代码省略了许多具体的实现，但是读者可以发现最终调用了`mLogElements.insert(last,elem)`方法，在buffer中插入了对应的日志实体。这个log方法主要是用来对logbuffer进行写操作，要了解更多对buffer的操作请，详细阅读\system\core\logd\LogBuffer.cpp文件。
流程图：



图九 服务端响应客户端写