# android log机制

Last edited by **caoquanli** 1 month ago

# Android 8.1 Log机制分析

Table of Contents

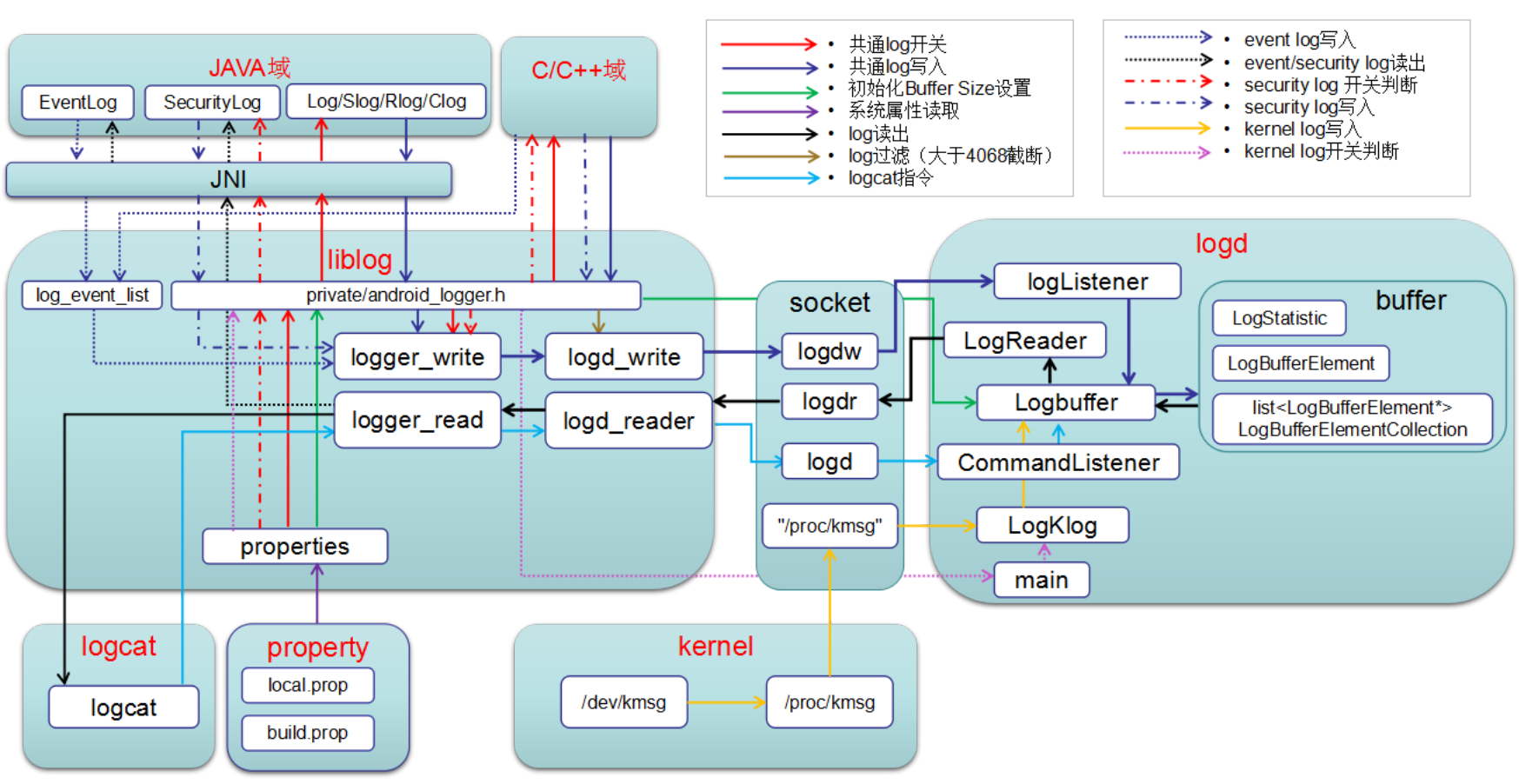


## 1. 概述

- Log是Android系统提供的用来存储输出系统运行日志的工具。不同版本系统的log机制有所调整，本文主要基于Android8.1源代码结合网上资源进行简单分析整理。

### 1.1. 整体框图

- 全文基于下面的log机制框图进行展开，主要包括java/c/c++/kernel写入、liblog接口、logcat读出、logd管理等几个模块之间的交互:




### 1.2. log分类

- Android 8.1按不同作用范围，将log分为了7个类型（定义在Android-8.1/system/core/liblog/include/log/log.h），用ID号予以区分，如下表：

| 类型 | LOG_ID | 备注 |
| --- | --- | --- |

| | | |
|---|---|---|
| MAIN | 0 | 应用相关log，Android应用开发时调试大量使用，标记程序运行流程 |
| RADIO | 1 | 通信相关log，通信模块涉及内容较多，单独分配一块缓存区进行记录管理 |
| EVENTS | 2 | 各种事件log，例如一个Activity生命周期从创建到消亡的各个阶段记录 |
| SYSTEM | 3 | 系统组件log，系统组件调用时的log记录，如ServiceManager的管理记录 |
| CRASH | 4 | 系统崩溃log，代码上看仅有RuntimeInit.java、libdebuggerd、async_safe三处调用 |
| SECURITY | 5 | 安全机制log，需要系统权限才能打印的log，与Android安全机制相关 |
| KERNE | 6 | linux内核log，开机启动、驱动层活动记录，如USB插拔 |

- 打印一条log时，可以标记一个优先级level（定义在Android-8.1/system/core/liblog/include/android/log.h），优先级可以用于后面log开关的判定；ID范围$0_5$（MAIN-SECURITY）的log可标记优先级如下表，其中常用优先级$_2$6：

| Android应用层log优先级 | level | 备注 |
|---|---|---|
| UNKNOWN | 0 | 未知级别？未见应用 |
| DEFAULT | 1 | 用于log开关设置，设置所有级别log输出 |
| VERBOSE | 2 | 开发调试过程中一些详细信息，不应该编译进产品中，只在开发阶段使用 |
| DEBUG | 3 | 用于调试的信息，编译进产品，但可以在运行时关闭 |
| INFO | 4 | 例如一些运行时的状态信息，这些状态信息在出现问题的时候能提供帮助 |
| WARN | 5 | 警告系统出现了异常，即将出现错误 |
| ERROR | 6 | 系统已经出现了错误 |
| FATAL | 7 | 致命错误出现，将导致crash |
| SILENT | 8 | 用于log开关设置，限制所有级别log输出 |

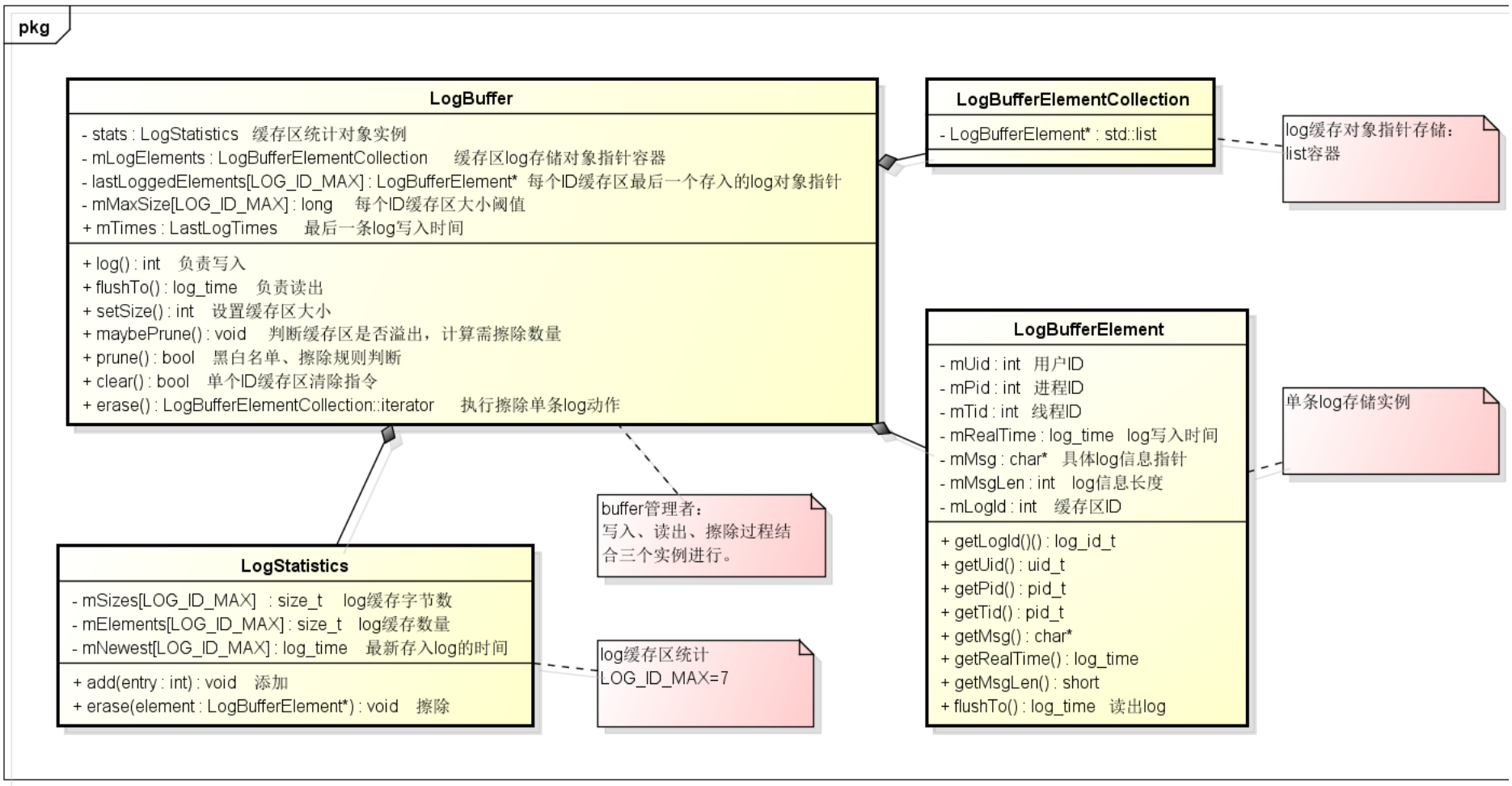
- kernel（ID = 6）的log优先级与Android应用层的log划分不同(在kernel-4.9/include/linux/kern_levels.h中定义)，如下表：

| 内核log 优先级 | level | 备注 |
|---|---|---|
| KERN_EMERG | 0 | 紧急事件消息，系统崩溃之前提示，表示系统不可用 |
| KERN_ALERT | 1 | 报告消息，表示必须采取措施 |
| KERN_CRIT | 2 | 临界条件，通常涉及严重的硬件或软件操作失败 |
| KERN_ERR | 3 | 错误条件，驱动程序常用KERN_ERR来报告硬件错误 |
| KERN_WARNING | 4 | 警告条件，对可能出现问题的情况进行警告 |
| KERN_NOTICE | 5 | 正常但又重要的条件，用于提醒 |
| KERN_INFO | 6 | 提示信息，如驱动程序启动时，打印硬件信息 |
| KERN_DEBUG | 7 | 调试级别的信息 |

- 当系统属性开启kernel log打印至应用层的权限后，kernel log会在打印至logd进程中进行优先级的转换。

### 1.3. 缓存区管理

- Android 8.1中，使用logd进程中的Logbuffer对象统一管理各个ID的log写入和读出，每个ID的缓存区并不是直接分配的内存空间，而是通过三个对象结合log_buffer_size这个阈值进行管理。简单描述为：LogStatistics对象统计每个ID的log缓存大小，LogBufferElement对象记录单体log的具体信息，list容器LogBufferElementCollection填装每一条log缓存对象指针。Logbuffer与这三个对象的类图如下：




- logd进程在初始化过程中会创建一个LogBuffer类对象，而LogBuffer类对象初始化过程中通过读取property中的属性设置为每一个LOG_ID设置log_buffer_size[LOG_ID]，默认是**256KB**。这个值可以看作为每个LOG_ID在buffer区的存储上限，当LogStatistics对象统计的该LOG_ID缓存SIZE超过上限，就会按一定规则进行擦除动作。
- 缓存区大小可通过读取系统属性设置与logcat指令进行调，如输入指令"setproppersist.logd.size.radio 1024k"或"logcat -b radio -G 10m"可设置Radio缓存区大小阈值。通过系统属性设置是需要重启生效，通过logcat指令设置立即生效，但重启后会失效。Android 8.1对log缓存区大小设置限制为**64KB~256MB**(定义在Android-8.1/system/core/liblog/include/private/android_logger.h)，在这个范围之外的缓存大小设置将被判定为无效设置而不能执行。
- 每一条log信息在缓存区都以一个LogBufferElement实例对象存在，单条log写入的时候信息长度不得超过**4068**字节（定义在Android-8.1/system/core/liblog/include/log/log_read.h），否则将在logd_write.cpp写入的时候就会被裁减掉。

### 1.4. log开关

- Android8.1原生的log开关包括一个普通的log写入限制开关、Security log开关、内核log写入开关，通过读取property中的属性值来限制相关的log写入与读出，这些开关在liblog中的接口如下：

| 原生log开关接口 | 属性Key |
| --- | --- |
| __android_log_is_loggable(int prio, const char* tag, int default_prio) | "persist.log.tag" |
| __android_log_security() | "persist.logd.security" "ro.device_owner" |
| __android_logger_property_get_bool(key, ...) | "logd.kernel"等 |

- 其中__android_log_is_loggable()开关中用来判断标签为tag、优先级为prio的log是否可写入，在java层的接口为isLoggable(Tag tag, int level)。有两个设置方法：

```
(1)、命令行设置，如设置tag为“AudioService.VOL”的log优先级为DEBUG以上写入有效，示例：setprop log.tag.AudioService.VOL  D
(2)、属性文件中添加，示例：在local.prop中添加log.tag.AudioService.VOL=D
```

- __android_log_security()用于判断安全机制的log是否可写入，对于不可写入的log将在logger_write中限制写入。
- __android_logger_property_get_bool是一个通用的属性判断接口，在logd进程初始化过程中用于Kernel的log是否可以写入到logd管理的缓存中的判断，默认开启。

### 1.5. log写入接口

- Android8.1中，ID为Main、Radio、System、Crash的log写入liblog的接口相同，ID为Events、Security的log写入liblog各自独占接口，JAVA层接口分散在不同目录下，最终调JNI接口整理如下：

| log类型 | JAVA→JNI接口 |
| --- | --- |
| MAIN | Log.println_native(Log.LOG_ID_MAIN, priority, tag, msg) |
| RADIO | Log.println_native(Log.LOG_ID_RADIO, priority, tag, msg) |
| EVENTS | writeEvent(int tag, Object… list)，输入参数格式与对应模块目录下后缀*.logtags文件中定义的保持一致 |
| SYSTEM | Log.println_native(Log.LOG_ID_SYSTEM, priority, tag, msg) |
| CRASH | Log.println_native(Log.LOG_ID_CRASH, priority, tag, msg) |
| SECURITY | writeEvent(int tag, Object… payloads) |

- C/C++层的log写入接口在android-8.1/system/core/include/log路径下定义，直接链接到liblog库中，使用时包含这个路径的对应头文件即可调用。
- ID为Kernal的log在系统属性设置开启权限后，可以直接经socket从内核写入logd进程管理的缓存区；此外在android-8.1/system/core/include/cutils/klog.h文件中定义了从本地层向内核层kmsg写入log的接口klog_write()，应用于init启动进程，需要相关操作权限，具体用法没有研究。

### 1.6. logd进程的通信方式

- 早期的Android系统采用位于内核层Logger驱动程序实现log信息的存储与读取，自Android5.0之后，系统应用层与内核层的log信息实现分开存储管理，在本地层通过创建一个logd进程主要负责android本地层与java层的log信息的存储与读出，而logd进程同时还可以读取内核printk、selinux的log。

- logd进程由系统init进程fork出来，即开机启动，在其启动时同时创建了3个Unix域socket，分别为/dev/socket/logd，/dev/socket/logdr，/dev/socket/logdw，其参数配置在/system/core/logd/logd.rc文件中，如下:

```
service logd /system/bin/logd
    socket logd stream 0666 logd logd
    socket logdr seqpacket 0666 logd logd
    socket logdw dgram+passcred 0222 logd logd
    file /proc/kmsg r
    file /dev/kmsg w
    user logd
    group logd system package_info readproc
    writepid /dev/cpuset/system-background/tasks
```

- logd进程启动后会创建对应上述三个socket的服务器端，分别用于接收logcat控制命令操作、logcat进程读log操作、log写入操作。需要注意的是，logdw的数据通信类型为dgram，说明log信息采用数据报文式写入到logd进程，因此在log信息写入过于频繁时存在丢失的情况。

- 同时可以看出logd进程对“/proc/kmsg”内核log文件具有读权限，当系统属性开启内核log向logd写入权限设置后，logd进程将以“/proc/kmsg”为描述符，从socket节点监听内核log，并写入buffer缓存区。

### 1.7. 其他

- 从基本Java层的log写入、读出来说，Android8.1的log机制可简单分为几段：

```
(1)、Java层通过JNI层向liblog本地库写入log信息，本地应用层直接向liblog写入log信息；
(2)、liblog本地库通过logdw向logd进程传递log信息；
(3)、logd进程接收log信息存入缓存区；
(4)、用户通过logcat应用发出读log请求；
(5)、logcat进程调用liblog库通过logdr向logd进程发送读log请求，并等待接收log信息；
(6)、logd进程解析log信息读取请求，从缓存区读出log信息通过logdr传回logcat进程；
(7)、logcat最后再对读出的log信息做相关处理，打印至终端或者文件。
```

- 后面内容将具体介绍这些流程。

- 参考资料：

  - 1、log机制总览
  - 2、log缓存区大小设置（最后一段）
  - 3、Android应用层与内核log优先级

## 2. Java层应用打印log信息

### 2.1. Java进程调用Log类打印log

- 在java层的可以调用Log、Slog、Rlog、EventLog、SecurityLog这几个类打印不同类型的log信息，以Log类打印ID为LOG_ID_MAIN的log为例，java应用程序打印log时，根据级别不同，调用log.v()~log.e()几个方法。

- 方法在/frameworks/base/core/java/android/util/Log.java中实现，log.v()代码执行如下：

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

### 2.2. Log类连接jni层函数

- println_native()方法的定义也在Log.java文件中，代码如下：

```
public static native int println_native(int bufID,
        int priority, String tag, String msg);
```

- native关键字表明该方法指向了JNI层，并传递了缓存区ID、优先级、tag标签和log信息。

### 2.3. jni层连接本地函数库

- 根据jni层与java层的交互机制，println_native()方法在jni层对应的函数名称为android_util_Log_println_native()，函数实现在/frameworks/base/core/jni/android_util_log.cpp文件中。调用该函数时首先对log信息和标签做了非空排除，并判断缓存ID是否正确，然后再调用了本地库函数_android_log_buf_write()继续传递log信息，如下：

```
int res = __android_log_buf_write(bufID,(android_LogPriority)priority, tag, msg);
```

- 在本地C/C++域调用log.h头文件打印log信息时，调用的是liblog本地库函数android_log_print()和android_log_buf_print()连接到本地库，然后再调用_android_log_buf_write()和_android_log_buf_write()，之后实现与Java层log信息共用本地库向logd进程传递log 。

### 2.4. 本地函数库向logdw节点发送log

- _android_log_buf_write()位于/system/core/liblog/logger_write.c文件中，函数中将log信息封装到了iovec类型结构体中，并调用了write_to_log()函数指针进行信息传递，部分代码如下：

```
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];
...
 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);
}
```

- write_to_log()函数指针，初始定义时指向了_write_to_log_init()函数，定义代码在logger_write.c文件的最上方，如下：

```
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;
```

- __write_to_log_init()函数中首先判断是否第一次调用write_to_log()，是的话就先调用_write_to_log_initialize()，之后再次调用write_to_log()则将其指向_write_to_log_daemon()，代码如下：

```
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);
}
```

- 上述三个函数同在logger_write.c文件中。先看一下_write_to_log_initialize()，这个函数首先定义了一个android_log_transport_write类的结构体指针transport以及一个链表n，随后调用了_android_log_config_write()和write_transport_for_each_safe()这两个函数，这两个函数分别定义在同目录下的config_write.c和config_write.h文件中，作用应该是链表的相关操作，然后将android_log_transport_write类的结构体指针指向了同目录下logd_write.c文件中android_log_transport_write类的结构体对象logdLoggerWrite，这个函数实现不是太懂，读者可以自己对照源码理解。_write_to_log_initialize()的代码实现如下：

```
static int __write_to_log_initialize() {
  struct android_log_transport_write* transport;
  struct listnode* n;
  int i = 0, ret = 0;

  __android_log_config_write();
  write_transport_for_each_safe(transport, n, &__android_log_transport_write) {
    __android_log_cache_available(transport);
    if (!transport->logMask) {
      list_remove(&transport->node);
      continue;
    }
    if (!transport->open || ((*transport->open)() < 0)) {
      if (transport->close) {
        (*transport->close)();
      }
      list_remove(&transport->node);
      continue;
    }
    ++ret;
  }
```

- __write_to_log_initialize()的代码中有这么一句“transport→open”，logd_write.c文件中定义的logdLoggerWrite成员中则有“.open = logdOpen”，因此理解此处调用了logd_write.c文件中logdOpen()函数。android_log_transport_write是函数指针类结构体，定义的logdLoggerWrite实际是指向了logd_write.c文件中一系列函数，代码如下：

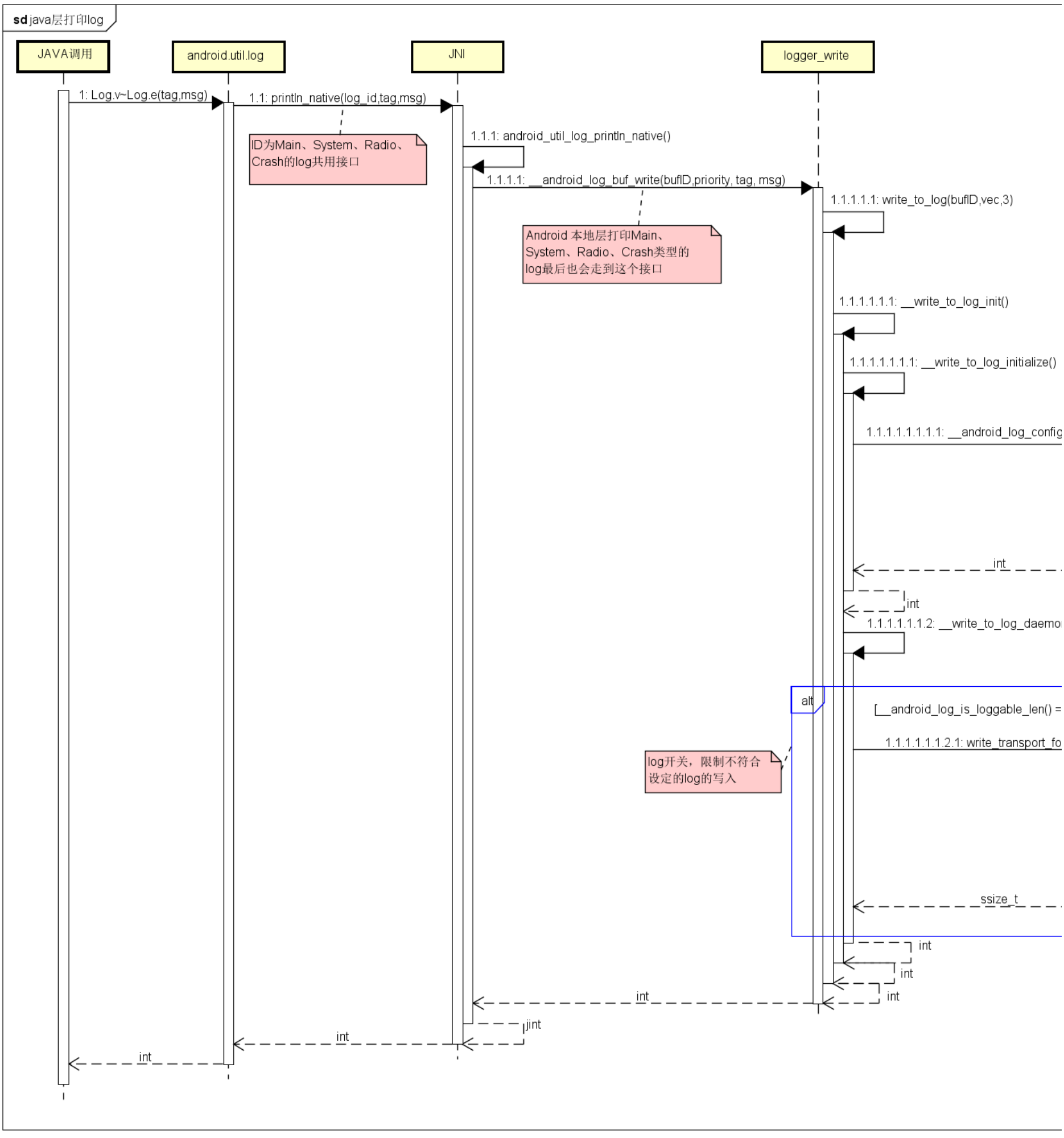
```
LIBLOG_HIDDEN struct android_log_transport_write logdLoggerWrite = {
  .node = { &logdLoggerWrite.node, &logdLoggerWrite.node },
  .context.sock = -EBADF,
  .name = "logd",
  .available = logdAvailable,
  .open = logdOpen,
  .close = logdClose,
  .write = logdWrite,
};
```

- logdOpen()函数中调用了socket()创建了logdw的客户端，并通过connect()连接到logdw服务器端。
- 再回头看下__write_to_log_daemon()，函数在一开始也定义了一个android_log_transport_write类的结构体指针node（链表节点？），并在函数的最后调用了config_write.h文件中的write_transport_for_each()，然后在该函数中通过“node→write”连接到了logd_write.c文件中的logdWrite()函数。
- logdWrite()函数是通过调用/system/core/liblog/uio.c文件中的writev()函数进行写log信息，而writev()函数中又进一步调用了的标准c库函数write()向dev/socket/logdw对应的socket缓冲区写入log。需要注意的是，在logdWrite()函数中通过LOGGER_ENTRY_MAX_PAYLOAD这一常量对log信息的长度进行了判断处理，超出进行裁剪，而LOGGER_ENTRY_MAX_PAYLOAD定义在/system/core/include/log/log_read.h，值为4068。裁剪代码段如下：

```
  for (payloadSize = 0, i = headerLength; i < nr + headerLength; i++) {
    newVec[i].iov_base = vec[i - headerLength].iov_base;
    payloadSize += newVec[i].iov_len = vec[i - headerLength].iov_len;

    if (payloadSize > LOGGER_ENTRY_MAX_PAYLOAD) {
      newVec[i].iov_len -= payloadSize - LOGGER_ENTRY_MAX_PAYLOAD;
      if (newVec[i].iov_len) {
        ++i;
      }
      break;
    }
  }
```

### 2.5. java打印log时序图




## 3. logd进程接收log信息并存入缓存区

### 3.1. logd进程创建缓存区与logdw服务器端

- 在前面的概述内容里说过，logd进程是开机启动的，而在其开机启动初始化函数main()(位于/system/core/logd/main.cpp)中创建了LogBuffer、LogReader、LogListener和CommandListener四个关键对象，代码段如下：

```
 logBuf = new LogBuffer(times);

signal(SIGHUP, reinit_signal_handler);

if (__android_logger_property_get_bool(
        "logd.statistics", BOOL_DEFAULT_TRUE | BOOL_DEFAULT_FLAG_PERSIST |
                           BOOL_DEFAULT_FLAG_ENG |
                           BOOL_DEFAULT_FLAG_SVELTE)) {
    logBuf->enableStatistics();
}

// LogReader listens on /dev/socket/logdr. When a client
// connects, log entries in the LogBuffer are written to the client.

LogReader* reader = new LogReader(logBuf);
if (reader->startListener()) {
    exit(1);
}

// 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);
}
```

```
// Command listener listens on /dev/socket/logd for incoming logd
// administrative commands.

CommandListener* cl = new CommandListener(logBuf, reader, swl);
if (cl->startListener()) {
    exit(1);
}
```

- 在同文件夹下LogBuffer.h头文件中可以看到LogBuffer类的相关定义，首先是创建了一存储log信息对象指针的容器mLogElements、一个log缓存区状态统计实例stats，然后设置了对应ID的缓冲区阈值，并定义了一些对log信息进行操作的关键函数，后面会用到。LogBuffer类定义部分代码段如下：

```
 typedef std::list<LogBufferElement*> LogBufferElementCollection;

class LogBuffer : public LogBufferInterface {
    LogBufferElementCollection mLogElements;
    pthread_rwlock_t mLogElementsLock;
    LogStatistics stats;
   ...
    unsigned long mMaxSize[LOG_ID_MAX];

    LogBufferElement* lastLoggedElements[LOG_ID_MAX];
    LogBufferElement* droppedElements[LOG_ID_MAX];
    void log(LogBufferElement* elem);
   ...
    int log(log_id_t log_id, log_time realtime, uid_t uid, pid_t pid, pid_t tid,
            const char* msg, unsigned short len) override;
    // lastTid is an optional context to help detect if the last previous
    // valid message was from the same source so we can differentiate chatty
    // filter types (identical or expired)
    log_time flushTo(SocketClient* writer, const log_time& start,
                     pid_t* lastTid,  // &lastTid[LOG_ID_MAX] or nullptr
                     bool privileged, bool security,
                     int (*filter)(const LogBufferElement* element,
                                   void* arg) = nullptr,
                     void* arg = nullptr);

    bool clear(log_id_t id, uid_t uid = AID_ROOT);
    unsigned long getSize(log_id_t id);
    int setSize(log_id_t id, unsigned long size);
    unsigned long getSizeUsed(log_id_t id);
```

- 回到前面说的四个对象，LogBuffer创建完后，对于logd进程接收log信息还需LogListener对象，该对象用来监听套接字dev/socket/logdw，对象相关类与函数的定义在/system/core/logd/LogListener.cpp和同目录的LogListener.h文件中。LogListener继承于 SocketListener类，在其构造函数中调用了getLogSocket()进行logdw服务器端的创建，getLogSocket()函数位于LogListener.cpp中，相关代码如下：

```
int LogListener::getLogSocket() {
    static const char socketName[] = "logdw";
    int sock = android_get_control_socket(socketName);

    if (sock < 0) {  // logd started up in init.sh
        sock = socket_local_server(
            socketName, ANDROID_SOCKET_NAMESPACE_RESERVED, SOCK_DGRAM);

        int on = 1;
        if (setsockopt(sock, SOL_SOCKET, SO_PASSCRED, &on, sizeof(on))) {
            return -1;
       }
    }
    return sock;
}
```

- getLogSocket()函数通过连接对应logdw的套接字判断服务器端是否创建，没有则通过调用socket_local_server()来创建服务器端。socket_local_server()函数位于/system/core/libcutils/socket_local_server_unix.c文件中，代码如下：

```
int socket_local_server(const char *name, int namespace, int type)
{
    int err;
    int s;

    s = socket(AF_LOCAL, type, 0);
    if (s < 0) return -1;

    err = socket_local_server_bind(s, name, namespace);

    if (err < 0) {
        close(s);
        return -1;
    }

    if ((type & SOCK_TYPE_MASK) == SOCK_STREAM) {
        int ret;

        ret = listen(s, LISTEN_BACKLOG);

        if (ret < 0) {
            close(s);
            return -1;
        }
    }

    return s;
}
```

- socket_local_server()函数开始调用了socket()创建套接字，然后调用同文件中的socket_local_server_bind()函数对logdw进行绑定，socket_local_server_bind()函数中实际也是通过调用socket的基本函数bind()进行绑定操作的。

### 3.2. 监听logdw线程开启

- 在logd进程main.cpp的main()函数中可以看到，在LogListener对象创建完成后(服务器端创建完成后)，调用了"swl→startListener(600)"进行监听线程的开启工作，这里的600不太懂，与socket通信属性相关。

- startListener()函数由LogListener对象从SocketListener类继承而来，其实现在/system/core/libsysutils/src/SocketListener.cpp文件中，函数中调用了同文件中的threadStart()函数，而threadStart()又直接指向了同文件中的runListener()函数，runListener()函数调用了socket的基本函数accept4()开启监听。

### 3.3. 读出logdw信息写入缓存区

- 当监听线程监听到有log信息写入套接字时，会调用LogListener::onDataAvailable()函数进行处理，onDataAvailable()函数的实现在/system/core/logd/LogListener.cpp中，函数中先调用了socket的基本函数recvmsg()读出log信息保存在结构体hdr中，做了一系列解析处理后又调用了同目录下LogBuffer.cpp文件中LogBuffer类的成员log()函数，将log信息传递进去做进一步处理。部分代码段如下:

```
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) };

    alignas(4) char control[CMSG_SPACE(sizeof(struct ucred))];
    struct msghdr hdr = {
        NULL, 0, &iov, 1, control, sizeof(control), 0,
    };

    int socket = cli->getSocket();

    // To clear the entire buffer is secure/safe, but this contributes to 1.68%
    // overhead under logging load. We are safe because we check counts.
    // memset(buffer, 0, sizeof(buffer));
    ssize_t n = recvmsg(socket, &hdr, 0);
    if (n <= (ssize_t)(sizeof(android_log_header_t))) {
        return false;
    }
...
    if (logbuf != nullptr) {
        int res = 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);
        if (res > 0 && reader != nullptr) {
            reader->notifyNewLog();
        }
    }

    return true;
}
```

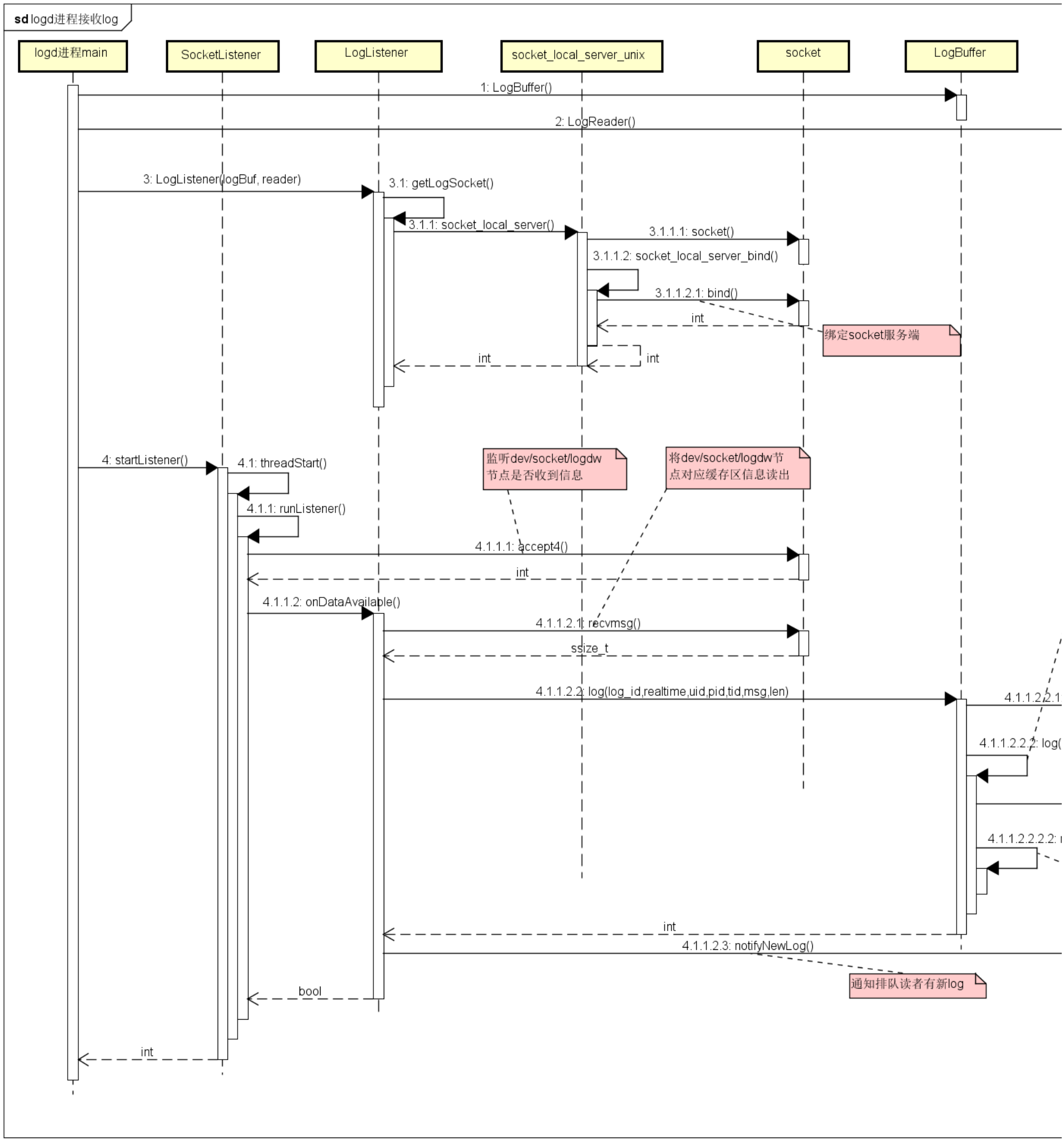
- log()函数一开始创建了一个LogBufferElement对象，并在其构造函数（/system/core/logd/LogBufferElement.cpp文件中）保存了这条log的基本信息。在利用一个状态机除去重复的log之后，log函数里又调用了同文件下的重构函数log(LogBufferElement* elem)将LogBufferElement对象指针传了进去。log(elem)函数主要做的是将LogBufferElement对象指针插入到容器mLogElements中。 之后，在log()函数中调用“stats.add(elem)”来更新缓存区状态信息，然后再调用maybePrune()函数进行log缓存区的溢出判断处理。

- 在log()函数操作结束后，回到onDataAvailable()函数中通过调用/system/core/logd/LogReader.cpp文件中的notifyNewLog()函数向排队等待的读者队列广播接收到新log的通知。

### 3.4. 判读缓存区是否超出

- maybePrune()函数里对刚刚写入log对应ID缓冲区log信息数量和已经占用缓存大小stats.sizes(id)与缓冲区大小阈值log_buffer_size(id)进行了综合比较，若实际占用缓存超出缓存区阈值，按一定规则删除4~256条log信息。实际删除工作由prune()函数完成，prune()函数中涉及到了log黑白名单的相关操作，这部分内容比较多，没有仔细看，有兴趣的可以去对照源代码分析。

### 3.5. logd进程接收log时序图




- 参考资料：
  - 1、logd接收log
  - 2、删除过多的log

## 4. logcat读取log信息

### 4.1. 常见logcat指令

- 使用adb工具连接android设备后，可以通过“adb logcat [选项] [过滤项]”指令调用logcat应用读取或设置设备log信息，也可以在adb shell进入设备的命令行界面后直接“logcat [选项] [过滤项]”进行logcat的指令操作。
- 常见的logcat指令有：

```
输出main缓存区的log信息：“logcat -b main”;
指定输出格式：“logcat -v <格式>”;
清空默认log缓存区：“logcat -c”;
保存log信息到指定文件：“logcat -f <保存路径>”;
查看log缓存区大小：“logcat -g”;
显示某一标签的log信息：“logcat -s < TAG名称> ”.
```

- 对log读出指令与log缓存区设置指令在logd进程中对应不同的监听线程，本小节以读log信息到文件为例，简单分析logcat读log的过程。

### 4.2. logcat指令解析

- logcat应用启动的初始化main()函数在/system/core/logcat/logcat_main.cpp文件中，函数通过调用同目录下logcat.cpp文件中的android_logcat_run_command()传入用户指令。
- android_logcat_run_command()函数创建了一个Context指针对象，封装了用户指令信息，并传入了同文件中的_logcat()函数中，代码如下：

```
    int android_logcat_run_command(android_logcat_context ctx,
                                   int output, int error,
                                   int argc, char* const* argv,
```

```
                                char* const* envp) {
    android_logcat_context_internal* context = ctx;

    context->output_fd = output;
    context->error_fd = error;
    context->argc = argc;
    context->argv = argv;
    context->envp = envp;
    context->stop = false;
    context->thread_stopped = false;
    return __logcat(context);
}
```

- __logcat()函数的前半部分为对用户指令的解析，后半部分为对本地库的函数调用以及对读出log信息的处理。部分指令解析代码如下：

```
     ret = getopt_long_r(argc, argv, ":cdDhLt:T:gG:sQf:r:n:v:b:BSpP:m:e:",
                         long_options, &option_index, &optctx);
    if (ret < 0) break;

    switch (ret) {
   ...
        case 'g':
            if (!optctx.optarg) {
                getLogSize = true;
                break;
            }
        // FALLTHRU

        case 'G': {
            char* cp;
            if (strtoll(optctx.optarg, &cp, 0) > 0) {
                setLogSize = strtoll(optctx.optarg, &cp, 0);
            } else {
                setLogSize = 0;
            }

            switch (*cp) {
                case 'g':
                case 'G':
                    setLogSize *= 1024;
                // FALLTHRU
                case 'm':
                case 'M':
                    setLogSize *= 1024;
                // FALLTHRU
                case 'k':
                case 'K':
                    setLogSize *= 1024;
                // FALLTHRU
                case '\0':
                    break;

                default:
                    setLogSize = 0;
            }

            if (!setLogSize) {
                logcat_panic(context, HELP_FALSE,
                             "ERROR: -G <num><multiplier>\n");
                goto exit;
            }
        } break;
    ...
        case 'f':
            if ((tail_time == log_time::EPOCH) && !tail_lines) {
                tail_time = lastLogTime(optctx.optarg);
            }
            // redirect output to a file
            context->outputFileName = optctx.optarg;
            break;
```

### 4.3. 向本地库写入读log请求

- 对于读log写入文件指令，函数里解析完“-f”指令之后，调用同文件下的setupOutputAndSchedulingPolicy()函数获取文件描述符，然后进入while循环开始循环调用本地库函数android_logger_list_read()发送读log请求，读出并写入文件。在不需要写文件时，输出描述符指向标准输出流stdout，也就是输出到终端显示。while循环部分代码段如下：

```
    while (!context->stop &&
           (!context->maxCount || (context->printCount < context->maxCount))) {
        struct log_msg log_msg;
        int ret = android_logger_list_read(logger_list, &log_msg);
       ...
        if (dev != d) {
            dev = d;
            maybePrintStart(context, dev, printDividers);
            if (context->stop) break;
        }
        if (context->printBinary) {
            printBinary(context, &log_msg);
        } else {
            processBuffer(context, dev, &log_msg);
        }
    }
```

- android_logger_list_read()位于/system/core/liblog/logger_read.c文件中，该函数定义了一个android_log_transport_context类结构体指针对象transp，用于后续转换读出的log信息到函数传进来的log_msg地址。函数的起始处调用了同文件中的init_transport_context()函数，传入指令参数结构体指针进行读log请求处理。

- init_transport_context()函数中定义了一个android_log_transport_read类结构体指针transport，对应在/system/core/liblog/logd_reader.c文件中定义了一个android_log_transport_read类结构体对象logdLoggerRead。由于android_log_transport_read是一类函数指针的集合，创建logdLoggerRead时则定义将这些函数指针指向了logd_reader.c文件中的各个函数，代码如下：

```
    LIBLOG_HIDDEN struct android_log_transport_read logdLoggerRead = {
      .node = { &logdLoggerRead.node, &logdLoggerRead.node },
      .name = "logd",
      .available = logdAvailable,
      .version = logdVersion,
      .read = logdRead,
      .poll = logdPoll,
      .close = logdClose,
      .clear = logdClear,
```

```
    .getSize = logdGetSize,
    .setSize = logdSetSize,
    .getReadableSize = logdGetReadableSize,
    .getPrune = logdGetPrune,
    .setPrune = logdSetPrune,
    .getStats = logdGetStats,
  };
```

- init_transport_context()函数中的transport与logd_reader.c文件中的logdLoggerRead是通过调用同目录下congfig_read.c文件中的_android_log_config_read()函数与congfig_read.h文件中read_transport_for_each()函数中进行连接的，又涉及到了链表的操作了，不是太懂。
- 在init_transport_context()函数中可以看到"transport→read"，可以理解为调用了logd_reader.c文件中的logdRead()函数。

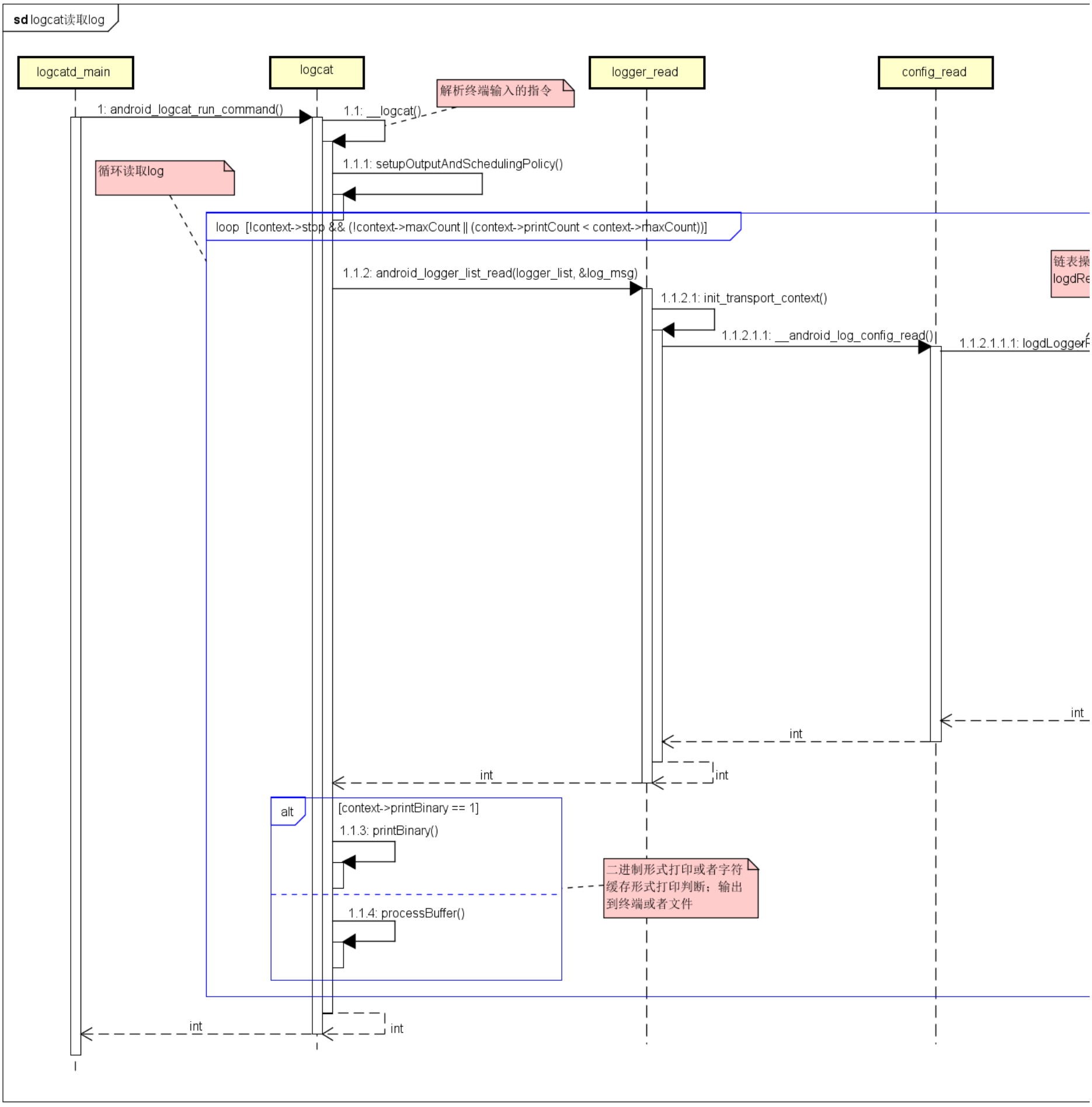
### 4.4. 本地函数库向logdr节点发送读log请求

- logdRead()函数起始处调用了同文件中的logdOpen()函数，logdOpen()函数中首先是调用/system/core/libcutils/socket_local_client_unix.c文件中的socket_local_client()函数创建连接logdr的客户端，连接成功后在logdOpen()函数又调用了标准c库函数write()向logdr对应的socket缓冲区写入读log指令。

### 4.5. 等待log信息返回

- 回到logdRead()函数中，在发完读log指令后，调用了socket的基本函数recv()来接收log信息并存入log_msg结构体指针对象log_msg中，后续回调操作应该都是对指向log信息的指针进行操作。
- 在_logcat()函数的循环中接收到log信息后，调用了printBinary()函数或processBuffer()函数进行写文件或者输出终端操作。

### 4.6. logcat读取log时序图




- 参考资料：
  - 1、logcat命令
  - 2、Android6.0 logcat读log（与8.1差异较大，简单参考）

## 5. logd进程处理log读请求

### 5.1. logd进程创建logdr服务器端

- 由第三节可知，logd进程启动初始化后也创建了一个LogReader类的对象，LogReader类也是继承于 SocketListener类，在其构造函数中由getLogSocket()函数通过连接对应logdr的套接字判断服务器端是否创建，没有则通过调用socket_local_server()来创建服务器端。过程与LogListener对象创建过程类似。

### 5.2. 监听logdr线程开启

- LogReader类的对象创建完成后，调用“reader→startListener()”进行监听线程的开启工作，线程开启过程与LogListener的监听线程开启过程基本一致，区别在于监听到logdr信息后响应的操作不同。

### 5.3. 解析logcat读log信息参数

- 在监听到logdr的信息写入后，线程中调用了LogReader::onDataAvailable()函数来解析出来读log指令，该函数在/system/core/logd/LogReader.cpp文件中实现。函数中首先通过标准c库的read()函数读出logdr处写入的读log指令信息，然后对读出的指令进行了解析。

- 在读log前，onDataAvailable()函数中先调用了一次/system/core/logd/LogBuffer.cpp文件中的LogBuffer::flushTo()函数进行优化，按照网上的资料介绍是通过设置其第6个参数作为过滤器使用遍历log队列，确保有符合条件的log项，如果没有，直接关闭socket并返回。这段代码没有仔细追究，大家可以对照这段代码进行分析：

```
    if (nonBlock && (sequence != log_time::EPOCH) && timeout) {
        class LogFindStart {  // A lambda by another name
           private:
            const pid_t mPid;
            const unsigned mLogMask;
            bool mStartTimeSet;
            log_time mStart;
            log_time& mSequence;
            log_time mLast;
            bool mIsMonotonic;

           public:
            LogFindStart(pid_t pid, unsigned logMask, log_time& sequence,
                         bool isMonotonic)
                : mPid(pid),
                  mLogMask(logMask),
                  mStartTimeSet(false),
                  mStart(sequence),
                  mSequence(sequence),
                  mLast(sequence),
                  mIsMonotonic(isMonotonic) {
            }

            static int callback(const LogBufferElement* element, void* obj) {
                LogFindStart* me = reinterpret_cast<LogFindStart*>(obj);
                if ((!me->mPid || (me->mPid == element->getPid())) &&
                    (me->mLogMask & (1 << element->getLogId()))) {
                    log_time real = element->getRealTime();
                    if (me->mStart == real) {
                        me->mSequence = real;
                        me->mStartTimeSet = true;
                        return -1;
                    } else if (!me->mIsMonotonic || android::isMonotonic(real)) {
                        if (me->mStart < real) {
                            me->mSequence = me->mLast;
                            me->mStartTimeSet = true;
                            return -1;
                        }
                        me->mLast = real;
                    } else {
                        me->mLast = real;
                    }
                }
                return false;
            }

            bool found() {
                return mStartTimeSet;
            }

        } logFindStart(pid, logMask, sequence,
                       logbuf().isMonotonic() && android::isMonotonic(start));

        logbuf().flushTo(cli, sequence, nullptr, FlushCommand::hasReadLogs(cli),
                         FlushCommand::hasSecurityLogs(cli),
                         logFindStart.callback, &logFindStart);

        if (!logFindStart.found()) {
            doSocketDelete(cli);
            return false;
        }
    }
```

### 5.4. 创建读log线程

- 读log优化操作之后，onDataAvailable()函数里创建了一个FlushCommand类实例对象command，FlushCommand类继承于SocketClientCommand，属于SocketListener框架的一部分，创建该类对象主要为响应notifyNewLog()函数的通知（也就是log写入时的广播通知），其构造函数在/system/core/logd/FlushCommand.cpp文件中。

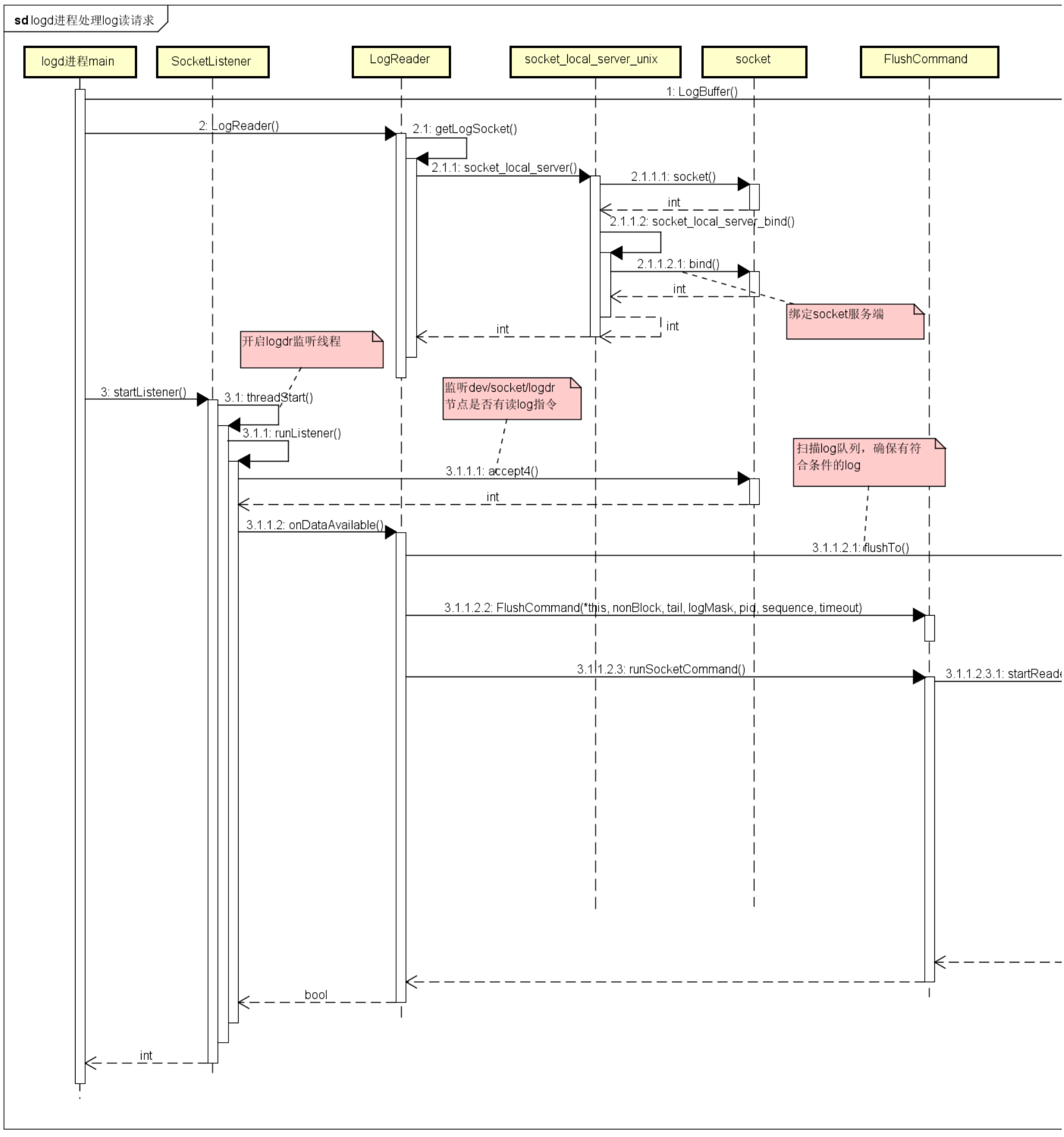
- 之后在onDataAvailable()函数里调用FlushCommand.cpp文件中的FlushCommand类的成员函数runSocketCommand()开始创建读取log线程工作。runSocketCommand()函数最后调用了/system/core/logd/LogTimes.cpp文件中的startReader_Locked()函数，startReader_Locked()函数又调用了同文件下的threadStart()函数开启线程。

### 5.5. 向logdr写入log信息

- 线程里又一次调用了LogBuffer::flushTo()函数开始读log操作。LogBuffer::flushTo()函数在存储log信息对象指针的容器中找到相应的log信息的指针，然后调用了/system/core/logd/LogBufferElement.cpp文件中的LogBufferElement::flushTo()函数开始传送实际的log信息。

- LogBufferElement::flushTo()函数将log信息封装到了iovec结构体对象，然后通过/system/core/libsysutils/src/SocketClient.cpp文件中的sendDataLockedv()函数传递log信息，sendDataLockedv()函数中通过同文件中的sendDataLockedv()函数最后调用writev()函数将log信息写入到logdr对应的socket缓存区。

### 5.6. logd进程处理读log请求时序图




- 参考资料：
  - 读取logd中的log数据（重要参考，建议查看）

## 6. logcat控制指令处理过程

- logd进程专门创建了一个CommandListener对象用于监听处理logcat发送的其他指令，如设置缓存区大小、获取缓存区大小、删除log缓存等。logcat的指令发送与logd进程监听接收指令过程和上两节logcat读log请求的发送与logd进程处理读log请求流程相似，主要区别在于过程中处理函数的不同。这里以设置log缓存区大小**“logcat -G < size >”**为例，简单说一下logcat控制指令的处理过程。

### 6.1. logcat通过本地函数库向logd节点发送控制指令

- 在上面4.2节粘贴的代码段中，可以看到__logcat()函数中对“G”的解析，这个字符就是设置log缓存区大小的关键字。解析完之后，在该函数的后半段会调用/system/core/liblog/logger_read.c文件中的android_logger_set_log_size()函数。
- android_logger_set_log_size()函数直接指向了同文件下的宏定义的LOGGER_FUNCTION()函数，LOGGER_FUNCTION()函数创建了android_log_transport_context类结构体指针transp，并通过/system/core/liblog/logger.h文件下宏定义transport_context_for_each()函数进行相关链表操作，然后调用“(transp→transport→func)(logger_internal, transp, ##args)”指向了/system/core/liblog/logd_reader.c文件下logdSetSize()函数。
- logdSetSize()函数调用同文件中的send_log_msg()发送设置缓存区大小信息，send_log_msg()函数里通过调用socket_local_client()函数创建对应/dev/socket/logd的客户端，之后直接调用write()函数向socket缓存区写入设置信息。

### 6.2. logd进程解析logcat指令并执行操作

- CommandListener类继承于FrameworkListener 类，当在logd进程初始化过程中创建其对象后，对应地开启监听/dev/socket/logd的线程。
- 在监听线程监听到logd节点写入信息后，响应的是/system/core/libsysutils/src/FrameworkListener.cpp文件中FrameworkListener::onDataAvailable()函数，该函数先调用read()函数读出logcat的指令信息，然后通过同文件下的dispatchCommand()调用“c→runCommand()”指向了/system/core/logd/CommandListener.cpp文件中的CommandListener::SetBufSizeCmd::runCommand()函数。
- runCommand()函数则调用了/system/core/logd/LogBuffer.cpp文件中的setSize()函数对log缓存区的大小按传递进来的参数进行设置。

## 7. *细节补充: security log的应用

### 7.1. security log的打印场景

#### 7.1.1. 应用分析

- security log是Android log系统中比较特殊的一种，与event log一样拥有专用的打印接口、固定的tag、信息解读格式。security log在打印和读取过程中均设置了多处权限检查，用户无法通过logcat工具直接读取security log，也无法看到security log缓存区的缓存状况。
- 虽然目前在Android 8.1中security log的打印只有设备密钥登录、adb传输文件、app进程启动三个场合，但在Andriod 9中扩展了更多打印security log的场合，因此还是有必要了解一下这一Android安全机制的相关log打印流程。

#### 7.1.2. JAVA层：

- 打印接口：SecurityLog.writeEvent()
- 应用位置DevicePolicyManagerService.java

```
(1)、reportFailedPasswordAttempt()密码登录失败
   SecurityLog.writeEvent(SecurityLog.TAG_KEYGUARD_DISMISS_AUTH_ATTEMPT, /*result*/ 0,
                  /*method strength*/ 1);
(2)、reportSuccessfulPasswordAttempt()密码登录成功
   SecurityLog.writeEvent(SecurityLog.TAG_KEYGUARD_DISMISS_AUTH_ATTEMPT, /*result*/ 1,
                  /*method strength*/ 1);
(3)、reportSuccessfulFingerprintAttempt()指纹识别成功
   SecurityLog.writeEvent(SecurityLog.TAG_KEYGUARD_DISMISS_AUTH_ATTEMPT, /*result*/ 1,
                  /*method strength*/ 0);
(4)、reportFailedFingerprintAttempt()指纹识别失败
   SecurityLog.writeEvent(SecurityLog.TAG_KEYGUARD_DISMISS_AUTH_ATTEMPT, /*result*/ 0,
                  /*method strength*/ 0);
(5)、reportKeyguardSecured()密钥保护启动
   SecurityLog.writeEvent(SecurityLog.TAG_KEYGUARD_SECURED);
(6)、reportKeyguardSecured()密钥保护解除
   SecurityLog.writeEvent(SecurityLog.TAG_KEYGUARD_SECURED);
```

- 应用位置ActivityManagerService → PackageManagerService.java → ProcessLoggingHandler.java

```
(1)、handleMessage()所有APP启动的时候都会打印app进程信息：
   SecurityLog.writeEvent(SecurityLog.TAG_APP_PROCESS_START, processName,
                        startTimestamp, uid, pid, seinfo, apkHash);
流程 startProcessLocked() -> logAppProcessStartIfNeeded() -> sendMessage() ->handleMessage() ->writeEvent()
```

#### 7.1.3. C/C++层：

- 打印接口：__android_log_security_bswrite()
- 应用位置file_sync_service.cpp （system/core/adb/目录下）

```
(1)、handle_send_file() adb发送文件
  __android_log_security_bswrite(SEC_TAG_ADB_SEND_FILE, path)
(2)、do_recv() adb接收文件
  __android_log_security_bswrite(SEC_TAG_ADB_RECV_FILE, path)
```

- 应用位置shell_service.cpp （system/core/adb/目录下）

```
(1)、Subprocess::ForkAndExec() shell命令行启动进程?
  __android_log_security_bswrite(SEC_TAG_ADB_SHELL_CMD, command_.c_str());
```

### 7.2. security log权限调查

- 目前logcat工具不具备system权限，因此不能查看security log，也不能设置security buffer。而system_server进程具有写读取security log权限，读写接口均在SecurityLog.java文件中，主要调用者则是运行在system_server进程之上的DevicePolicyManagerService。

#### 7.2.1. 写入权限

- security log从java层通过jni文件android_app_admin_SecurityLog.cpp调用_android_log_security_bwrite()接入liblog打印；C/C++层的security log则是直接通过android_logger.h调用_android_log_security_bswrite()接入liblog打印。之后则是按普通log的打印流程，从liblog向logd进程写入。
- 权限拦截1：在logger_write.c/__write_to_log_daemon()方法中对于LOG_ID_SECURITY的log调用check_log_uid_permissions判UID或GID是否为AID_SYSTEM、AID_ROOT和AID_LOG之一。
- 属性拦截2：在logger_write.c/__write_to_log_daemon()方法中对于LOG_ID_SECURITY的log调用_android_log_security()判断"persist.logd.security"和"ro.device_owner"属性是否都为true（默认为false），如果不是则拦截掉。
- 权限拦截3：LogListener.cpp/onDataAvailable()方法中对于LOG_ID_SECURITY的log调用__android_log_security()和clientHasLogCredentials()判断属性是否为true并且UID或GID否满足AID_SYSTEM、AID_ROOT、AID_LOG三者之一。

#### 7.2.2. 读取权限

- 权限拦截1：在logger_read.c/android_logger_list_read()方法中调用的init_transport_context()方法里对LOG_ID_SECURITY的操作，要求UID为AID_SYSTEM。
- 权限拦截2：上面init_transport_context()方法调用到的logd_reader.c/logdAvailable()方法里再次对LOG_ID_SECURITY的操作，要求UID为AID_SYSTEM。
- 权限拦截3：在LogBuffer.cpp/flushTo()方法中对LOG_ID_SECURITY的操作调用了hasSecurityLogs()要求客户端的UID或GID为AID_SYSTEM。
- logcat工具使用时为shell权限或root（eng版）权限，无权限读取security log。

#### 7.2.3. 缓存区查看及设置权限

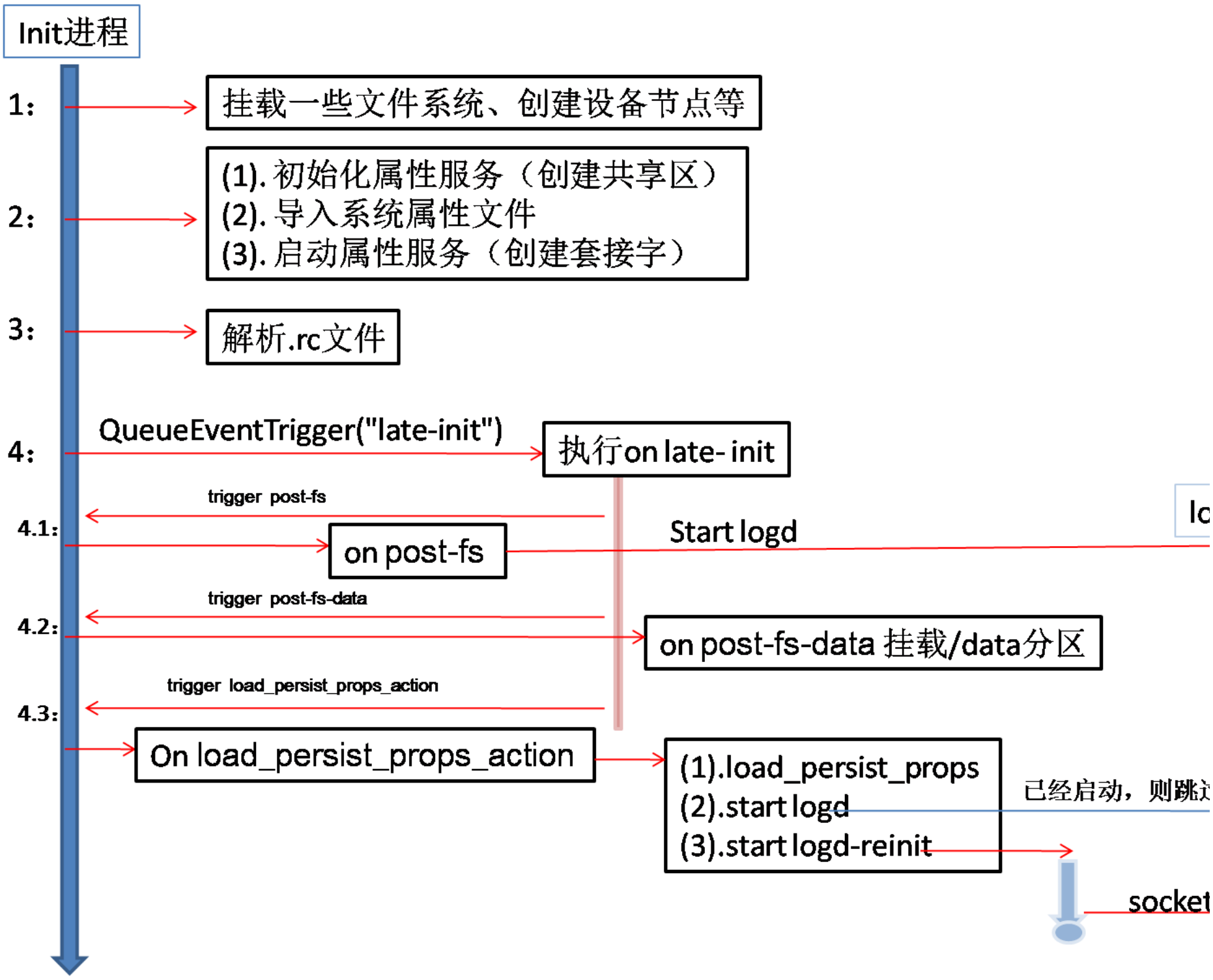
- 权限拦截1：查看设置log缓存区均调用到logger_read.c/LOGGER_FUNCTION()，该方法中调用的init_transport_context()方法里对LOG_ID_SECURITY的操作，要求UID为AID_SYSTEM。
- 权限拦截2：init_transport_context()方法调用到的logd_reader.c/logdAvailable()方法里再次对LOG_ID_SECURITY的操作，要求UID为AID_SYSTEM。
- logcat工具使用时为shell权限或root（eng版）权限，无权限设置和查看security log缓存区。

## 8. *细节补充：init进程启动logd流程

### 8.1. init进程启动logd与logd-reinit流程

#### 8.1.1. 补充背景

- 在Leepi项目中对应kernel log开关要求重启生效，因此考虑只在logd进程启动时判断kernel log开关是否开启，来决定是否启动logd.klog线程（读kernel log的线程）。
- Android原生的logd进程启动时通过"logd.kernel"属性来判断是否创建并启动logd.klog线程，默认开启。
- 由于需要重启生效，log开关的属性值可保存，我们把属性设置为"persist"类型。但随之而来的问题是"persist"类型属性文件保存在/data/property 目录，data分区的挂载却在logd进程启动之后，因此导致logd启动的时候不能立刻获取到"persist"类型属性值。
- 调查init进程启动属性服务和logd进程的简单流程如下：

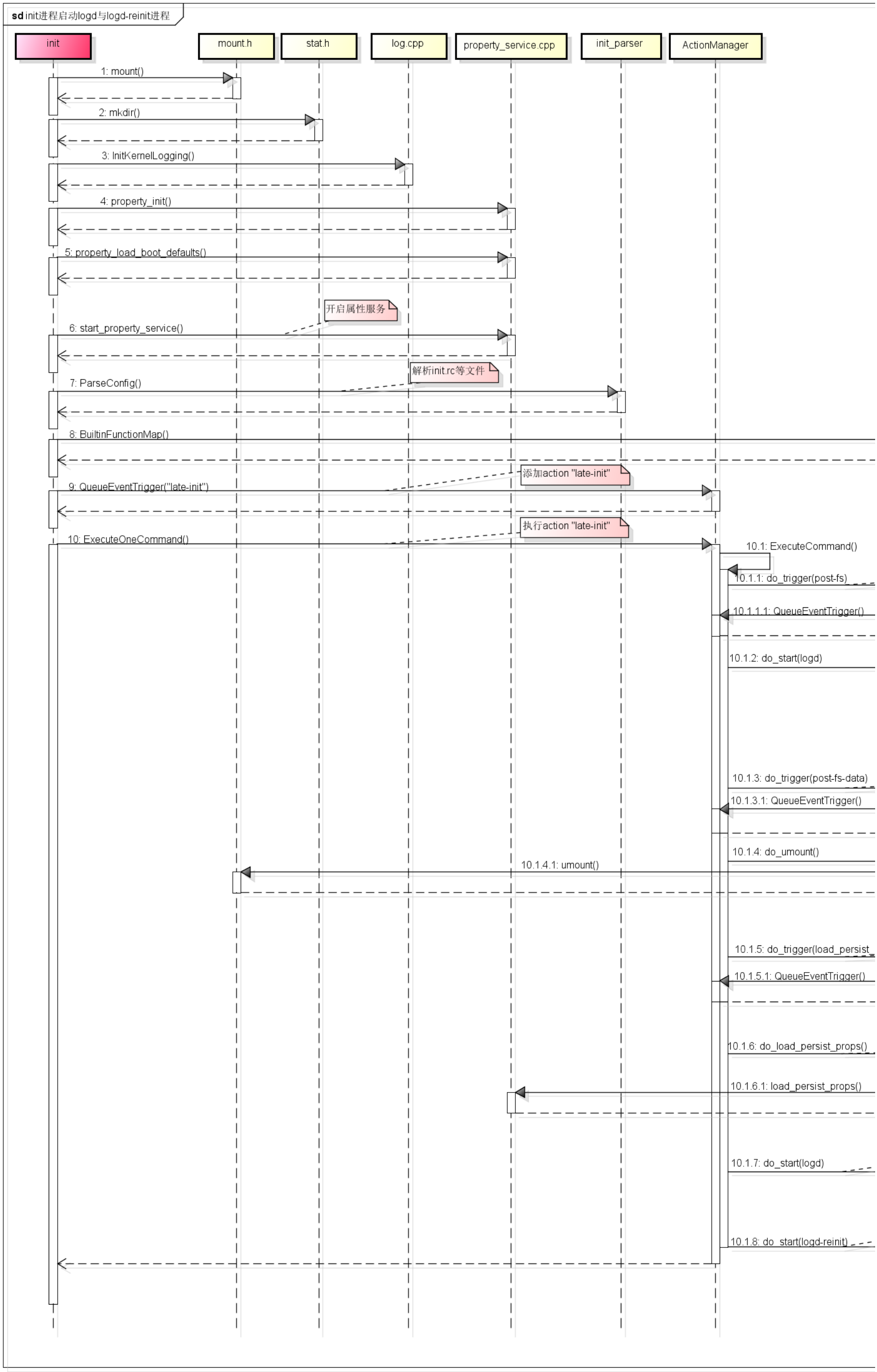



### 8.1.2. logd与logd-reinit进程启动时序

- logd-reinit进程用于发送指令重新初始化logd进程中的logBuf（log缓存），启动时机在data分区挂载之后，"persist"类型属性文件已经被属性服务加载。因此可以用于判断是否到了可以有效获取"persist"类型属性的时间点。
- init进程启动属性服务、logd进程、logd-reinit进程等主要步骤如下：

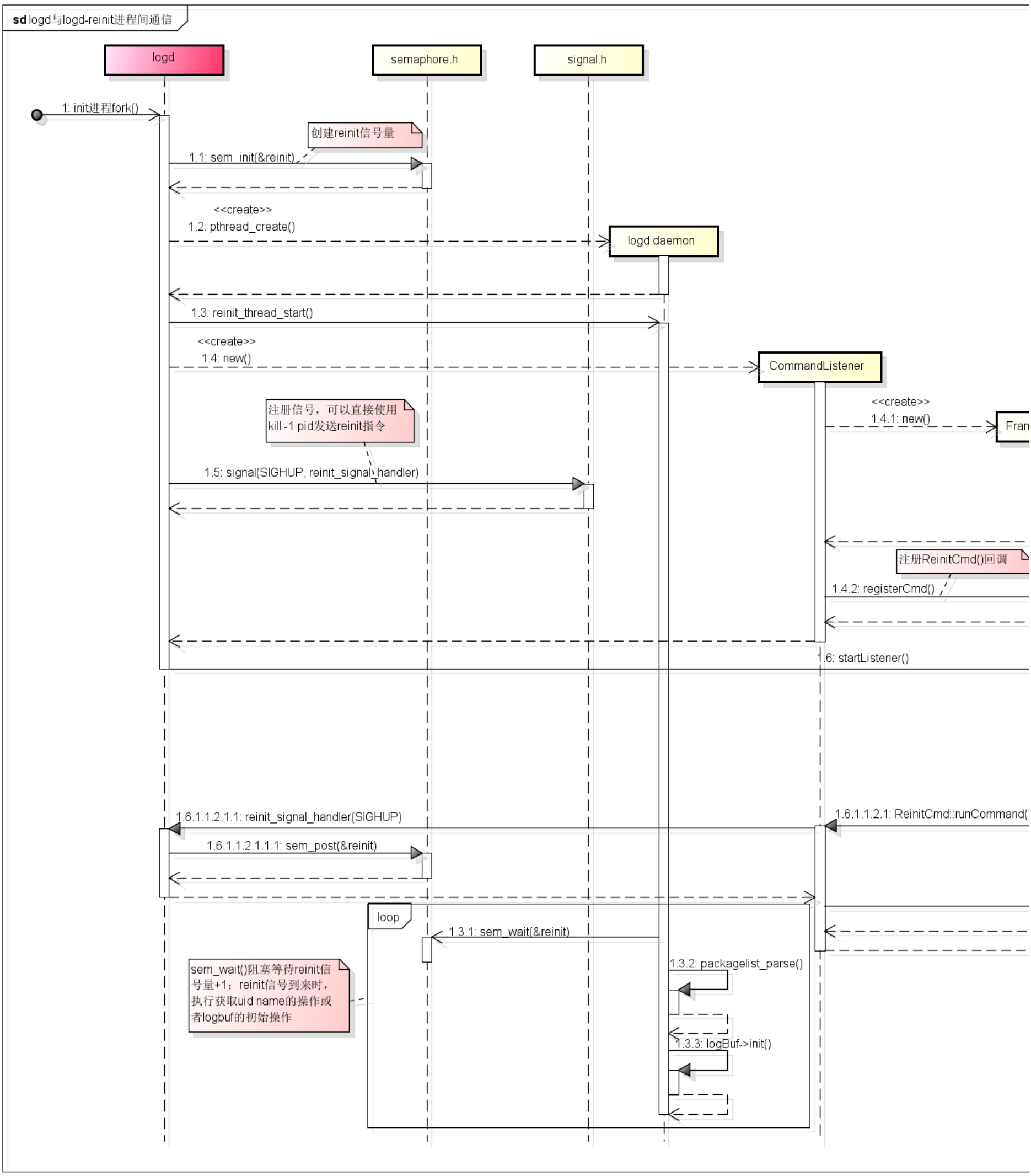
```
|1 挂载文件系统、创建设备节点
|2 初始化属性服务、导入默认属性文件、启动属性服务
|3 解析.rc文件
|4 添加nit.rc文件中late-init动作至动作列表
|5 ExecuteOneCommand()执行late-init动作
   |5.1 触发post-fs动作，在该动作中启动logd进程
   |5.2 触发post-fs-data动作，在该动作中挂载/data分区
   |5.3 触发load_persist_props_action动作，依次执行:
      |5.3.1 load_persist_props 导入persist属性文件
      |5.3.2 start logd启动logd进程(5.1已经启动,跳过)
      |5.3.3 启动logd-reinit进程
         | 5.3.3.1 logd-reinit进程与logd进程间通信
```

- 代码执行时序如下图：




**8.2. logd进程与logd-reinit进程间通信**

- logd进程启动后创建了一个"reinit"信号量,默认为数值为0；并创建了一个logd.daemon线程循环使用sem_wait阻塞等待该信号量+1。
- logd进程中的logd.control线程初始化时注册了包括了ReinitCmd在内各种指令，并通过"logd"socket监听外部控制指令。
- 当logd-reinit进程启动后，通过"logd"socket发送"reinit"指令，然后检测指令是否发送成功，最后结束进程；logd-reinit进程只在init进程时启动一次。
- logd.control线程接收到指令后调用reinit_signal_handler()函数通过sem_post给"reinit"信号量+1; logd.daemon线程中调用的sem_wait检测到信号+1后-1，并向内核写入"logd.daemon: reinit"日志，然后初始化logBuf。
- 需要注意的是在logd进程启动的时候注册了signal(SIGHUP, reinit_signal_handler)信号回调操作;因此可以直接通过kill -1 pid向logd进程的logd.daemon线程发送重新初始化指令。
- logd进程与logd-reinit进程间通信时序图下图：




- 参考资料：
  - Android — Init进程对属性系统的处理流程分析
  - Android 8.0 : 系统启动流程之init

注：Android8.1的log机制涉及内容较多，本文没有完全展开。本文仅供参考作用，具体的细节需要结合源码去看。文中可能有一些错误，看到误地方可以直接修改，有疑问可以一起探讨。