## new MediaPlayer()

frameworks\base\media\java\android\media\MediaPlayer.java

static {

System.loadLibrary("media\_jni"); //1.加载so

native\_init();//2.调用native\_init()方法

}

/\*\*

\* Default constructor. Consider using one of the create() methods for

\* synchronously instantiating a MediaPlayer from a Uri or resource.

\* <p>When done with the MediaPlayer, you should call {@link #release()},

\* to free the resources. If not released, too many MediaPlayer instances may

\* result in an exception.</p>

\*/

public MediaPlayer() {

Looper looper;

if ((looper = Looper.myLooper()) != null) {

mEventHandler = new EventHandler(this, looper);

} else if ((looper = Looper.getMainLooper()) != null) {

mEventHandler = new EventHandler(this, looper);

} else {

mEventHandler = null;

}

mTimeProvider = new TimeProvider(this);

mOpenSubtitleSources = new Vector<InputStream>();

IBinder b = ServiceManager.getService(Context.APP\_OPS\_SERVICE);

mAppOps = IAppOpsService.Stub.asInterface(b);

if (SystemProperties.get("ro.target.product").equals("tablet")&&SystemProperties.get("ro.board.platform").equals("rk3399")) {

IBinder binder = ServiceManager.getService("device");

mDeviceManager = IDeviceManager.Stub.asInterface(binder);

}

/\* Native setup requires a weak reference to our object.

\* It's easier to create it here than in C++.

\*/

native\_setup(new WeakReference<MediaPlayer>(this)); //3.调用native\_setup()方法传入当前对象的弱引用

}

### native\_init

frameworks\base\media\jni\android\_media\_MediaPlayer.cpp

static void

android\_media\_MediaPlayer\_native\_init(JNIEnv \*env)

{

jclass clazz;

clazz = env->FindClass("android/media/MediaPlayer");

if (clazz == NULL) {

return;

}

fields.context = env->GetFieldID(clazz, "mNativeContext", "J");

if (fields.context == NULL) {

return;

}

fields.post\_event = env->GetStaticMethodID(clazz, "postEventFromNative",

"(Ljava/lang/Object;IIILjava/lang/Object;)V");

if (fields.post\_event == NULL) {

return;

}

fields.surface\_texture = env->GetFieldID(clazz, "mNativeSurfaceTexture", "J");

if (fields.surface\_texture == NULL) {

return;

}

env->DeleteLocalRef(clazz);

clazz = env->FindClass("android/net/ProxyInfo");

if (clazz == NULL) {

return;

}

fields.proxyConfigGetHost =

env->GetMethodID(clazz, "getHost", "()Ljava/lang/String;");

fields.proxyConfigGetPort =

env->GetMethodID(clazz, "getPort", "()I");

fields.proxyConfigGetExclusionList =

env->GetMethodID(clazz, "getExclusionListAsString", "()Ljava/lang/String;");

env->DeleteLocalRef(clazz);

gPlaybackParamsFields.init(env);

gSyncParamsFields.init(env);

}

// ----------------------------------------------------------------------------

struct fields\_t {

jfieldID context;

jfieldID surface\_texture;

jmethodID post\_event;

jmethodID proxyConfigGetHost;

jmethodID proxyConfigGetPort;

jmethodID proxyConfigGetExclusionList;

};

static fields\_t fields;

初始化fields,成员变量赋值

android/media/MediaPlayer类中long类型的mNativeContext-> context

android/media/MediaPlayer类中postEventFromNative函数->post\_event

android/media/MediaPlayer类中long类型的long类型的->surface\_texture

### native\_setup

frameworks\base\media\jni\android\_media\_MediaPlayer.cpp

static void

android\_media\_MediaPlayer\_native\_setup(JNIEnv \*env, jobject thiz, jobject weak\_this)

{

ALOGV("native\_setup");

sp<MediaPlayer> mp = new MediaPlayer();

if (mp == NULL) {

jniThrowException(env, "java/lang/RuntimeException", "Out of memory");

return;

}

// create new listener and give it to MediaPlayer

sp<JNIMediaPlayerListener> listener = new JNIMediaPlayerListener(env, thiz, weak\_this);

mp->setListener(listener);

// Stow our new C++ MediaPlayer in an opaque field in the Java object.

setMediaPlayer(env, thiz, mp);

}

static sp<MediaPlayer> setMediaPlayer(JNIEnv\* env, jobject thiz, const sp<MediaPlayer>& player)

{

Mutex::Autolock l(sLock);

sp<MediaPlayer> old = (MediaPlayer\*)env->GetLongField(thiz, fields.context);

if (player.get()) {

player->incStrong((void\*)setMediaPlayer);

}

if (old != 0) {

old->decStrong((void\*)setMediaPlayer);

}

env->SetLongField(thiz, fields.context, (jlong)player.get());//设置当前这个MediaPlay的属性的值为MediaPlayer指针

return old;

}

logger\_reader,当前正在读取日志的进程

struct logger\_reader {

struct logger\_log \*log; //指向要读取的日志设备的对应的logger\_log

struct list\_head list; //所有读取同一日志类型的进程

size\_t r\_off; //要读取的下一条日志在日志缓冲区中的位置

};

初始化device\_initcall(logger\_init);即module\_init

#define DEFINE\_LOGGER\_DEVICE(VAR, NAME, SIZE) \

static unsigned char \_buf\_ ## VAR[SIZE]; \//\_buf\_log\_xx[64\*1024]

static struct logger\_log VAR = { \

.buffer = \_buf\_ ## VAR, \//初始化日志缓冲区长度

.misc = { \

.minor = MISC\_DYNAMIC\_MINOR, \//次设备号动态分配

.name = NAME, \//设备名

.fops = &logger\_fops, \ //日志设备对应的文件操作函数

.parent = NULL, \

}, \

.wq = \_\_WAIT\_QUEUE\_HEAD\_INITIALIZER(VAR .wq), \//初始化等待队列头

.readers = LIST\_HEAD\_INIT(VAR .readers), \

//#define LIST\_HEAD\_INIT(name) { &(name), &(name) }

//static inline void INIT\_LIST\_HEAD(struct list\_head \*list)

//{

// list->next = list;

// list->next = list;

//}

.mutex = \_\_MUTEX\_INITIALIZER(VAR .mutex), \//初始化互斥量

.w\_off = 0, \ //下一条要写入日志在缓冲区位置，初始化0

.head = 0, \//下一个新的进程要读取日志的位置，初始化0

.size = SIZE, \//缓冲区的长度

};

//初始化三个logger\_log类型的变量log\_main、log\_events、log\_radio，对应三个日志设备文件，system和main类型的日志保存在同一个日志设备文件log\_main中

DEFINE\_LOGGER\_DEVICE(log\_main, LOGGER\_LOG\_MAIN, 64\*1024)

DEFINE\_LOGGER\_DEVICE(log\_events, LOGGER\_LOG\_EVENTS, 256\*1024)

DEFINE\_LOGGER\_DEVICE(log\_radio, LOGGER\_LOG\_RADIO, 64\*1024)

static int \_\_init init\_log(struct logger\_log \*log)

{

int ret;

ret = misc\_register(&log->misc); //这里注册一个杂项设备，主设备号10, 使用misc\_register函数注册一个混杂设备，misc\_deregister移除一个杂项设备。注册成功后，linux内核会自动为该设备创建设备节点，在/dev/下会产生相应的节点

Android系统使用uevent机制来管理系统的设备文件，当Logger日志驱动程序注册一个日志设备时，内核会发出一个uevent事件，这个uevent最终由init进程中的handle\_device\_event函数处理

system/core/init/devices.c 最后日志文件设备会在dev/log/下被创建

//include\linux\miscdevice.h

//drivers/char/misc.c看一下misc中的代码

static int \_\_init misc\_init(void)

{

int err;

#ifdef CONFIG\_PROC\_FS

//这里创建proc文件系统，

proc\_create("misc", 0, NULL, &misc\_proc\_fops);

#endif

//在/sys/class/目录下创建misc

misc\_class = class\_create(THIS\_MODULE, "misc");

err = PTR\_ERR(misc\_class);

if (IS\_ERR(misc\_class))

goto fail\_remove;

err = -EIO;

//注册字符设备

if (register\_chrdev(MISC\_MAJOR,"misc",&misc\_fops))

goto fail\_printk;

return 0;

fail\_printk:

printk("unable to get major %d for misc devices\n", MISC\_MAJOR);

class\_destroy(misc\_class);

fail\_remove:

remove\_proc\_entry("misc", NULL);

return err;

}

//看一下misc\_register函数

int misc\_register(struct miscdevice \* misc)

{

struct miscdevice \*c;

dev\_t dev;

int err = 0;

INIT\_LIST\_HEAD(&misc->list);//初始化misc\_list链表

mutex\_lock(&misc\_mtx);

list\_for\_each\_entry(c, &misc\_list, list) {

//遍历misc\_list链表，看这个次设备号以前有没有被用过，如果次设备号已被占有则退出

if (c->minor == misc->minor) {

mutex\_unlock(&misc\_mtx);

return -EBUSY;

}

}

if (misc->minor == MISC\_DYNAMIC\_MINOR) {

//如果是动态分配次设备号

#define DYNAMIC\_MINORS 64 /\* like dynamic majors \*/

static unsigned char misc\_minors[DYNAMIC\_MINORS / 8];

int i = DYNAMIC\_MINORS;循环次数0-63

while (--i >= 0)

if ( (misc\_minors[i>>3] & (1 << (i&7))) == 0)右移相当于除2^n,右移三除以8的商，结果0-7，每个结果出现8次，i&7相当于0111取最后三位，可能的情况8种，0-7，也是出现8次，1左移0-7位相当于2^0-2^7

当前面 i>>3从7递减到0的过程中，右边，从128->递减到1，对应的取值二进制是

2^7 = 1000 0000 2^6 =1000000 ,也就是取0到8的每一位，==0 那就是没使用过，那么找到这一位，分配次设备号

break;

if (i<0) {

mutex\_unlock(&misc\_mtx);

return -EBUSY;

}

misc->minor = i;

}

if (misc->minor < DYNAMIC\_MINORS)//如果分配的次设备号，小于64，对应的位上设置1

misc\_minors[misc->minor >> 3] |= 1 << (misc->minor & 7);

dev = MKDEV(MISC\_MAJOR, misc->minor);//主设备号次设备号，合成设备号

//创建设备节点

misc->this\_device = device\_create(misc\_class, misc->parent, dev, NULL,

"%s", misc->name);

if (IS\_ERR(misc->this\_device)) {

err = PTR\_ERR(misc->this\_device);

goto out;

}

/\*

\* Add it to the front, so that later devices can "override"

\* earlier defaults

\*/

list\_add(&misc->list, &misc\_list);//miscdevice添加到misc\_list中

out:

mutex\_unlock(&misc\_mtx);

return err;

}

if (unlikely(ret)) {

printk(KERN\_ERR "logger: failed to register misc "

"device for log '%s'!\n", log->misc.name);

return ret;

}

printk(KERN\_INFO "logger: created %luK log '%s'\n",

(unsigned long) log->size >> 10, log->misc.name);

return 0;

}

static int \_\_init logger\_init(void)

{

int ret;

ret = init\_log(&log\_main);

if (unlikely(ret))

goto out;

ret = init\_log(&log\_events);

if (unlikely(ret))

goto out;

ret = init\_log(&log\_radio);

if (unlikely(ret))

goto out;

out:

return ret;

}

logger\_open日志设备打开函数

static int logger\_open(struct inode \*inode, struct file \*file)

{

struct logger\_log \*log;

int ret;

ret = nonseekable\_open(inode, file); //设置日志设备不可随机访问

if (ret)

return ret;

log = get\_log\_from\_minor(MINOR(inode->i\_rdev)); //根据次设备号找到对应的日志设备对应的logger\_log

if (!log)

return -ENODEV;

if (file->f\_mode & FMODE\_READ) {//检查文件是否可以读取

struct logger\_reader \*reader; //初始化一个logger\_reader

reader = kmalloc(sizeof(struct logger\_reader), GFP\_KERNEL);

if (!reader)

return -ENOMEM;

reader->log = log; //reader的log成员指向要读取的日志设备对应logger\_log

INIT\_LIST\_HEAD(&reader->list); //初始化reader里面的list

mutex\_lock(&log->mutex);

reader->r\_off = log->head; //要读取的下一条日志在日志缓冲区中的位置，从log->head位置开始读取

list\_add\_tail(&reader->list, &log->readers); //将reader的list\_head加入到log的list\_head中去

//头插法：

static inline void list\_add(struct list\_head \*new, struct list\_head \*head)

{

\_\_list\_add(new, head, head->next);

}

尾插法：

static inline void list\_add\_tail(struct list\_head \*new, struct list\_head \*head)

{

\_\_list\_add(new, head->prev, head);

}

真正的实现插入：

static inline void \_\_list\_add(struct list\_head \*new,

struct list\_head \*prev,

struct list\_head \*next)

{

next->prev = new;

new->next = next;

new->prev = prev;

prev->next = new;

}

\_\_list\_add(new, prev, next)：表示在prev和next之间添加一个新的节点new

mutex\_unlock(&log->mutex);

file->private\_data = reader; //将reader放入file中

} else

file->private\_data = log; //如果文件不可读取，则将对应的logger\_log结构体放到file的private\_data

return 0;

}

logger\_read日志设备读取函数

static ssize\_t logger\_read(struct file \*file, char \_\_user \*buf,

size\_t count, loff\_t \*pos)

{

struct logger\_reader \*reader = file->private\_data; //把open里面存放的logger\_reader取出来

struct logger\_log \*log = reader->log; //reader里面对应的日志设备类型logger\_log也取出来

ssize\_t ret;

DEFINE\_WAIT(wait); //定义了wait的 wait\_queue\_t类型的等待队列

//#define DEFINE\_WAIT\_FUNC(name, function) \

wait\_queue\_t name = { \

.private = current, \

.func = function, \

.task\_list = LIST\_HEAD\_INIT((name).task\_list), \

}

#define DEFINE\_WAIT(name) DEFINE\_WAIT\_FUNC(name, autoremove\_wake\_function)

start:

while (1) {

prepare\_to\_wait(&log->wq, &wait, TASK\_INTERRUPTIBLE); //将log里面的等待队列加入到里面

mutex\_lock(&log->mutex);

ret = (log->w\_off == reader->r\_off); //当前进程读取的地方和日志设备文件的写入的地方是否相等

mutex\_unlock(&log->mutex);

if (!ret) //如果不相等，有日志可读，那么跳出循环

break;

if (file->f\_flags & O\_NONBLOCK) {//相等，无日志可读，并且是无阻塞模式打开设备，直接跳出循环

ret = -EAGAIN;

break;

}

if (signal\_pending(current)) {//有信号处理，跳出循环，结束当前系统调用，以便回去处理信号

ret = -EINTR;

break;

}

schedule();//如果无日志可读，又是阻塞模式打开的，并且没有信号需要处理，那么请求内核进行一次进程调度，进入休眠状态，等待写入数据的时候，也就是有数据可读的时候，将它唤醒

}

finish\_wait(&log->wq, &wait); //从等待队列移除

if (ret)

return ret;

mutex\_lock(&log->mutex); //此处可能会获取互斥量失败，失败的时候会进入休眠，直到获取到互斥量，这个时候可能读取的位置已经发生了改变，所以下面重新对读取日志的位置进行判断

/\* is there still something to read or did we race? \*/

if (unlikely(log->w\_off == reader->r\_off)) {//如果日志设备的读取位置和进程读取的位置已经相等，则返回到循环

mutex\_unlock(&log->mutex);

goto start;

}

/\* get the size of the next entry \*/

ret = get\_entry\_len(log, reader->r\_off); //这边返回日志的长度

static \_\_u32 get\_entry\_len(struct logger\_log \*log, size\_t off)

{

\_\_u16 val;

switch (log->size - off) {//log的长度减去需要读取的地址，如果等于1，那么说明这条日志logger\_entry中的len成员被分成了俩部分，一个在这个日志缓冲区的末尾1个字节，另一个字节在buffer的最开始，所以从buffer加上off的偏移，读取一个字节，然后从buffer的地址开始的地方读取一个字节，拼起来赋值给va

否则从缓冲区读取位置，读取俩个字节就是这条日志的长度

case 1:

memcpy(&val, log->buffer + off, 1);

memcpy(((char \*) &val) + 1, log->buffer, 1);

break;

default:

memcpy(&val, log->buffer + off, 2);

}

return sizeof(struct logger\_entry) + val;

}

if (count < ret) {//如果读取的大小小于日志的长度

ret = -EINVAL;

goto out;

}

/\* get exactly one entry from the log \*/

ret = do\_read\_log\_to\_user(log, reader, buf, ret); //

static ssize\_t do\_read\_log\_to\_user(struct logger\_log \*log,

struct logger\_reader \*reader,

char \_\_user \*buf,

size\_t count)

{

size\_t len;

/\*

\* We read from the log in two disjoint operations. First, we read from

\* the current read head offset up to 'count' bytes or to the end of

\* the log, whichever comes first.

\*/

len = min(count, log->size - reader->r\_off);//读取长度，日志长度减去读取位置取个最小值

if (copy\_to\_user(buf, log->buffer + reader->r\_off, len))//从缓冲区读取位置，先读len个字节

return -EFAULT;

/\*

\* Second, we read any remaining bytes, starting back at the head of

\* the log.

\*/

if (count != len)//如果count==len那么这条日志的长度小于日志缓冲区剩余的长度，已经读取完，如果count!=len说明这条日志在日志缓冲区分成了2段，一部分在底部，一部分在开通，底部已经读取到用户层，剩下来，从buffer开始读取剩余没读取的字节数

if (copy\_to\_user(buf + len, log->buffer, count - len))

return -EFAULT;

reader->r\_off = logger\_offset(reader->r\_off + count);//将reader读取的位置更新

#define logger\_offset(n) ((n) & (log->size - 1))

这个定义log->size是2^n次方，2^n-1表示n位1111与上一个数，取传入数的n位

return count;

}

out:

mutex\_unlock(&log->mutex);

return ret;

}

logger\_aio\_write日志设备读取函数，file\_operations结构体的aio\_read、aio\_write和read、write都能处理同步和异步I/O操作。只是aio\_read和aio\_write函数可以获取更多的异步I/O的数据，并可以更好地控制异步I/O

ssize\_t logger\_aio\_write(struct kiocb \*iocb, const struct iovec \*iov,

unsigned long nr\_segs, loff\_t ppos)

{

struct logger\_log \*log = file\_get\_log(iocb->ki\_filp); //取出logger\_log

size\_t orig = log->w\_off; //当前log设备文件写入的位置

struct logger\_entry header; //初始化一条日志,下面是赋值操作

struct timespec now;

ssize\_t ret = 0;

now = current\_kernel\_time();

header.pid = current->tgid;

header.tid = current->pid;

header.sec = now.tv\_sec;

header.nsec = now.tv\_nsec;

header.len = min\_t(size\_t, iocb->ki\_left, LOGGER\_ENTRY\_MAX\_PAYLOAD); //日志长度和最大长度取一个最小值

/\* null writes succeed, return zero \*/

if (unlikely(!header.len))

return 0;

mutex\_lock(&log->mutex);

/\*

\* Fix up any readers, pulling them forward to the first readable

\* entry after (what will be) the new write offset. We do this now

\* because if we partially fail, we can end up with clobbered log

\* entries that encroach on readable buffer.

\*/

fix\_up\_readers(log, sizeof(struct logger\_entry) + header.len); //

static void fix\_up\_readers(struct logger\_log \*log, size\_t len)

{

size\_t old = log->w\_off;//写入位置

size\_t new = logger\_offset(old + len);//写入这条后，更新写入位置

struct logger\_reader \*reader;

if (clock\_interval(old, new, log->head))//这里判断读取的地方是否是新写入会覆盖的区域

log->head = get\_next\_entry(log, log->head, len);//如果是，那么更新log读取的位置

list\_for\_each\_entry(reader, &log->readers, list)

if (clock\_interval(old, new, reader->r\_off))

reader->r\_off = get\_next\_entry(log, reader->r\_off, len);//更新所有正在读取日志的进程的位置

}

do\_write\_log(log, &header, sizeof(struct logger\_entry)); //写入日志头部

static void do\_write\_log(struct logger\_log \*log, const void \*buf, size\_t count)

{

size\_t len;

len = min(count, log->size - log->w\_off);//取这个写入的长度，和缓冲区剩余长度比较，较小的那个

memcpy(log->buffer + log->w\_off, buf, len);//拷贝len

if (count != len)//如果count!=len说明写入内容被分成了2部分，把剩下那部分，写到缓冲区的头上

memcpy(log->buffer, buf + len, count - len);

log->w\_off = logger\_offset(log->w\_off + count);//更新log的写入位置

}

while (nr\_segs-- > 0) {//根据参数的个数

size\_t len;

ssize\_t nr;

/\* figure out how much of this vector we can keep \*/

len = min\_t(size\_t, iov->iov\_len, header.len - ret); //写入参数内容和剩余需要写入长度比较取最小

/\* write out this segment's payload \*/

nr = do\_write\_log\_from\_user(log, iov->iov\_base, len); //

static ssize\_t do\_write\_log\_from\_user(struct logger\_log \*log,

const void \_\_user \*buf, size\_t count)

{

size\_t len;

len = min(count, log->size - log->w\_off);

if (len && copy\_from\_user(log->buffer + log->w\_off, buf, len))

return -EFAULT;

if (count != len)

if (copy\_from\_user(log->buffer, buf + len, count - len))

return -EFAULT;

log->w\_off = logger\_offset(log->w\_off + count);

return count;

}

if (unlikely(nr < 0)) {

log->w\_off = orig;

mutex\_unlock(&log->mutex);

return nr;

}

iov++;

ret += nr;

}

mutex\_unlock(&log->mutex);

/\* wake up any blocked readers \*/

wake\_up\_interruptible(&log->wq); //唤醒前面读的在等待的进程

return ret;

}

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

main \_\_android\_log\_assert \_\_android\_log\_vprint \_\_android\_log\_print 这三个最后调用\_\_android\_log\_write

events \_\_android\_log\_btwrite \_\_android\_log\_bwrite

\_\_android\_log\_buf\_print 最后调到\_\_android\_log\_buf\_write

write\_to\_log \_\_weite\_to\_log\_init \_\_write\_to\_log\_kernel或者\_\_write\_to\_log\_null

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刚开始初始化指向\_\_write\_to\_log\_init

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

{

#ifdef HAVE\_PTHREADS

pthread\_mutex\_lock(&log\_init\_lock);

#endif

if (write\_to\_log == \_\_write\_to\_log\_init) {

log\_fds[LOG\_ID\_MAIN] = log\_open("/dev/"LOGGER\_LOG\_MAIN, O\_WRONLY);

log\_fds[LOG\_ID\_RADIO] = log\_open("/dev/"LOGGER\_LOG\_RADIO, O\_WRONLY);

log\_fds[LOG\_ID\_EVENTS] = log\_open("/dev/"LOGGER\_LOG\_EVENTS, O\_WRONLY);

log\_fds[LOG\_ID\_SYSTEM] = log\_open("/dev/"LOGGER\_LOG\_SYSTEM, O\_WRONLY);

#define LOGGER\_LOG\_MAIN "log/main"

#define LOGGER\_LOG\_RADIO "log/radio"

#define LOGGER\_LOG\_EVENTS "log/events"

#define LOGGER\_LOG\_SYSTEM "log/system"

write\_to\_log = \_\_write\_to\_log\_kernel; //将write\_to\_log指向\_\_write\_to\_log\_kernel

if (log\_fds[LOG\_ID\_MAIN] < 0 || log\_fds[LOG\_ID\_RADIO] < 0 ||

log\_fds[LOG\_ID\_EVENTS] < 0) {

log\_close(log\_fds[LOG\_ID\_MAIN]);

log\_close(log\_fds[LOG\_ID\_RADIO]);

log\_close(log\_fds[LOG\_ID\_EVENTS]);

log\_fds[LOG\_ID\_MAIN] = -1;

log\_fds[LOG\_ID\_RADIO] = -1;

log\_fds[LOG\_ID\_EVENTS] = -1;

write\_to\_log = \_\_write\_to\_log\_null; //main、radio、events有一个没打开，就将write\_to\_log指向\_\_write\_to\_log\_null

}

if (log\_fds[LOG\_ID\_SYSTEM] < 0) {

log\_fds[LOG\_ID\_SYSTEM] = log\_fds[LOG\_ID\_MAIN]; //system这里设置为main描述符，因为syste,和main的日志都写入日志设备文件/dev/log/main中

}

}

#ifdef HAVE\_PTHREADS

pthread\_mutex\_unlock(&log\_init\_lock);

#endif

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

}

static int \_\_write\_to\_log\_kernel(log\_id\_t log\_id, struct iovec \*vec, size\_t nr)

{

ssize\_t ret;

int log\_fd;

if (/\*(int)log\_id >= 0 &&\*/ (int)log\_id < (int)LOG\_ID\_MAX) {

log\_fd = log\_fds[(int)log\_id];

} else {

return EBADF;

}

do {

ret = log\_writev(log\_fd, vec, nr);

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

return ret;

}

/system/core/logcat/logcat.cpp

日志记录队列，每一种类型的日志记录都对应一个日志记录队列

struct queued\_entry\_t {

union {

unsigned char buf[LOGGER\_ENTRY\_MAX\_LEN + 1] \_\_attribute\_\_((aligned(4))); //分配空间时采用4 字节对齐方式

struct logger\_entry entry \_\_attribute\_\_((aligned(4)));

};

queued\_entry\_t\* next;

queued\_entry\_t() {

next = NULL;

}

};

log\_device\_t描述一个日志设备

struct log\_device\_t {

char\* device; //日志设备文件名称

bool binary; //日志内容是否是二进制

int fd; //对应的设备文件描述符

bool printed; //是否处于输出状态

char label; //标号’m’、‘s’、‘r’、’e’

queued\_entry\_t\* queue; //日志记录队列

log\_device\_t\* next; //连接下一个设备文件

log\_device\_t(char\* d, bool b, char l) {

device = d;

binary = b;

label = l;

queue = NULL;

next = NULL;

printed = false;

}

void enqueue(queued\_entry\_t\* entry) {//将一条日志记录保存到队列中

if (this->queue == NULL) {

this->queue = entry;

} else {

queued\_entry\_t\*\* e = &this->queue;

while (\*e && cmp(entry, \*e) >= 0) {

e = &((\*e)->next);

}

entry->next = \*e;

\*e = entry;

}

}

};

int main(int argc, char \*\*argv)

{

int err;

int hasSetLogFormat = 0;

int clearLog = 0;

int getLogSize = 0;

int mode = O\_RDONLY;

const char \*forceFilters = NULL;

log\_device\_t\* devices = NULL;

log\_device\_t\* dev;

bool needBinary = false;

g\_logformat = android\_log\_format\_new();

if (argc == 2 && 0 == strcmp(argv[1], "--test")) {

logprint\_run\_tests();

exit(0);

}

if (argc == 2 && 0 == strcmp(argv[1], "--help")) {

android::show\_help(argv[0]);

exit(0);

}

for (;;) {

int ret;

ret = getopt(argc, argv, "cdt:gsQf:r::n:v:b:B");

if (ret < 0) {

break;

}

switch(ret) {

case 's':

// default to all silent

android\_log\_addFilterRule(g\_logformat, "\*:s");

break;

case 'c':

clearLog = 1;

mode = O\_WRONLY;

break;

case 'd':

g\_nonblock = true;

break;

case 't':

g\_nonblock = true;

g\_tail\_lines = atoi(optarg);

break;

case 'g':

getLogSize = 1;

break;

case 'b': {

char\* buf = (char\*) malloc(strlen(LOG\_FILE\_DIR) + strlen(optarg) + 1);

strcpy(buf, LOG\_FILE\_DIR);

strcat(buf, optarg);

bool binary = strcmp(optarg, "events") == 0;

if (binary) {

needBinary = true;

}

if (devices) {

dev = devices;

while (dev->next) {

dev = dev->next;

}

dev->next = new log\_device\_t(buf, binary, optarg[0]);

} else {

devices = new log\_device\_t(buf, binary, optarg[0]);

}

android::g\_devCount++;

}

break;

case 'B':

android::g\_printBinary = 1;

break;

case 'f':

// redirect output to a file

android::g\_outputFileName = optarg;

break;

case 'r':

if (optarg == NULL) {

android::g\_logRotateSizeKBytes

= DEFAULT\_LOG\_ROTATE\_SIZE\_KBYTES;

} else {

long logRotateSize;

char \*lastDigit;

if (!isdigit(optarg[0])) {

fprintf(stderr,"Invalid parameter to -r\n");

android::show\_help(argv[0]);

exit(-1);

}

android::g\_logRotateSizeKBytes = atoi(optarg);

}

break;

case 'n':

if (!isdigit(optarg[0])) {

fprintf(stderr,"Invalid parameter to -r\n");

android::show\_help(argv[0]);

exit(-1);

}

android::g\_maxRotatedLogs = atoi(optarg);

break;

case 'v':

err = setLogFormat (optarg);

if (err < 0) {

fprintf(stderr,"Invalid parameter to -v\n");

android::show\_help(argv[0]);

exit(-1);

}

hasSetLogFormat = 1;

break;

case 'Q':

/\* this is a \*hidden\* option used to start a version of logcat \*/

/\* in an emulated device only. it basically looks for androidboot.logcat= \*/

/\* on the kernel command line. If something is found, it extracts a log filter \*/

/\* and uses it to run the program. If nothing is found, the program should \*/

/\* quit immediately \*/

#define KERNEL\_OPTION "androidboot.logcat="

#define CONSOLE\_OPTION "androidboot.console="

{

int fd;

char\* logcat;

char\* console;

int force\_exit = 1;

static char cmdline[1024];

fd = open("/proc/cmdline", O\_RDONLY);

if (fd >= 0) {

int n = read(fd, cmdline, sizeof(cmdline)-1 );

if (n < 0) n = 0;

cmdline[n] = 0;

close(fd);

} else {

cmdline[0] = 0;

}

logcat = strstr( cmdline, KERNEL\_OPTION );

console = strstr( cmdline, CONSOLE\_OPTION );

if (logcat != NULL) {

char\* p = logcat + sizeof(KERNEL\_OPTION)-1;;

char\* q = strpbrk( p, " \t\n\r" );;

if (q != NULL)

\*q = 0;

forceFilters = p;

force\_exit = 0;

}

/\* if nothing found or invalid filters, exit quietly \*/

if (force\_exit)

exit(0);

/\* redirect our output to the emulator console \*/

if (console) {

char\* p = console + sizeof(CONSOLE\_OPTION)-1;

char\* q = strpbrk( p, " \t\n\r" );

char devname[64];

int len;

if (q != NULL) {

len = q - p;

} else

len = strlen(p);

len = snprintf( devname, sizeof(devname), "/dev/%.\*s", len, p );

fprintf(stderr, "logcat using %s (%d)\n", devname, len);

if (len < (int)sizeof(devname)) {

fd = open( devname, O\_WRONLY );

if (fd >= 0) {

dup2(fd, 1);

dup2(fd, 2);

close(fd);

}

}

}

}

break;

default:

fprintf(stderr,"Unrecognized Option\n");

android::show\_help(argv[0]);

exit(-1);

break;

}

}

if (!devices) {

devices = new log\_device\_t(strdup("/dev/"LOGGER\_LOG\_MAIN), false, 'm');

android::g\_devCount = 1;

int accessmode =

(mode & O\_RDONLY) ? R\_OK : 0

| (mode & O\_WRONLY) ? W\_OK : 0;

// only add this if it's available

if (0 == access("/dev/"LOGGER\_LOG\_SYSTEM, accessmode)) {

devices->next = new log\_device\_t(strdup("/dev/"LOGGER\_LOG\_SYSTEM), false, 's');

android::g\_devCount++;

}

}

if (android::g\_logRotateSizeKBytes != 0

&& android::g\_outputFileName == NULL

) {

fprintf(stderr,"-r requires -f as well\n");

android::show\_help(argv[0]);

exit(-1);

}

android::setupOutput();

if (hasSetLogFormat == 0) {

const char\* logFormat = getenv("ANDROID\_PRINTF\_LOG");

if (logFormat != NULL) {

err = setLogFormat(logFormat);

if (err < 0) {

fprintf(stderr, "invalid format in ANDROID\_PRINTF\_LOG '%s'\n",

logFormat);

}

}

}

if (forceFilters) {

err = android\_log\_addFilterString(g\_logformat, forceFilters);

if (err < 0) {

fprintf (stderr, "Invalid filter expression in -logcat option\n");

exit(0);

}

} else if (argc == optind) {

// Add from environment variable

char \*env\_tags\_orig = getenv("ANDROID\_LOG\_TAGS");

if (env\_tags\_orig != NULL) {

err = android\_log\_addFilterString(g\_logformat, env\_tags\_orig);

if (err < 0) {

fprintf(stderr, "Invalid filter expression in"

" ANDROID\_LOG\_TAGS\n");

android::show\_help(argv[0]);

exit(-1);

}

}

} else {

// Add from commandline

for (int i = optind ; i < argc ; i++) {

err = android\_log\_addFilterString(g\_logformat, argv[i]);

if (err < 0) {

fprintf (stderr, "Invalid filter expression '%s'\n", argv[i]);

android::show\_help(argv[0]);

exit(-1);

}

}

}

dev = devices;

while (dev) {

dev->fd = open(dev->device, mode);

if (dev->fd < 0) {

fprintf(stderr, "Unable to open log device '%s': %s\n",

dev->device, strerror(errno));

exit(EXIT\_FAILURE);

}

if (clearLog) {

int ret;

ret = android::clearLog(dev->fd);

if (ret) {

perror("ioctl");

exit(EXIT\_FAILURE);

}

}

if (getLogSize) {

int size, readable;

size = android::getLogSize(dev->fd);

if (size < 0) {

perror("ioctl");

exit(EXIT\_FAILURE);

}

readable = android::getLogReadableSize(dev->fd);

if (readable < 0) {

perror("ioctl");

exit(EXIT\_FAILURE);

}

printf("%s: ring buffer is %dKb (%dKb consumed), "

"max entry is %db, max payload is %db\n", dev->device,

size / 1024, readable / 1024,

(int) LOGGER\_ENTRY\_MAX\_LEN, (int) LOGGER\_ENTRY\_MAX\_PAYLOAD);

}

dev = dev->next;

}

if (getLogSize) {

return 0;

}

if (clearLog) {

return 0;

}

//LOG\_EVENT\_INT(10, 12345);

//LOG\_EVENT\_LONG(11, 0x1122334455667788LL);

//LOG\_EVENT\_STRING(0, "whassup, doc?");

if (needBinary)

android::g\_eventTagMap = android\_openEventTagMap(EVENT\_TAG\_MAP\_FILE);

android::readLogLines(devices);

return 0;

}

## setDataSource

\system\core\init\init.cpp

### main

int main(int argc, char\*\* argv) {

…

init\_parse\_config\_file("/init.rc");

….

}

### init\_parse\_config\_file

int init\_parse\_config\_file(const char\* path) {

INFO("Parsing %s...\n", path);

Timer t;

std::string data;

if (!read\_file(path, &data)) {//根据传入的路径/init.rc,把文件中的内容读出来，并让data这个string类型的变量指向它

return -1;

}

data.push\_back('\n'); // TODO: fix parse\_config.

parse\_config(path, data);

dump\_parser\_state();

NOTICE("(Parsing %s took %.2fs.)\n", path, t.duration());

//system\core\init\util.h中定义的类

class Timer {

public:

Timer() : t0(gettime\_ns()) {初始化的时候获取当前纳秒初始化一个t0变量

}

double duration() {调用该函数的时候，当前纳秒数减去初始化的时候的纳秒数，除以9个0。等于秒数

return static\_cast<double>(gettime\_ns() - t0) / 1000000000.0;

}

private:

uint64\_t t0;

};

return 0;

}

#### read\_file

bool read\_file(const char\* path, std::string\* content) {

content->clear();//清空字符串

int fd = TEMP\_FAILURE\_RETRY(open(path, O\_RDONLY|O\_NOFOLLOW|O\_CLOEXEC)); //只读打开文件，所指的文件为一符号连接,则会令打开文件失败，O\_CLOEXEC模式打开的文件描述符在执行exec调用新程序中关闭,且为原子操作

if (fd == -1) {

return false;

}

// For security reasons, disallow world-writable

// or group-writable files.

struct stat sb;

if (fstat(fd, &sb) == -1) {//从文件描述符取得文件状态

ERROR("fstat failed for '%s': %s\n", path, strerror(errno));

return false;

}

//S\_IWGRP 用户组用户拥有写权限

//S\_IWOTH 其他用户拥有写权限

//表明文件不能拥有这些权限，如果有，则与运算不为0，那么跳过不安全的文件 return false

if ((sb.st\_mode & (S\_IWGRP | S\_IWOTH)) != 0) {

ERROR("skipping insecure file '%s'\n", path);

return false;

}

bool okay = android::base::ReadFdToString(fd, content); //system\core\base\file.cpp

//bool ReadFdToString(int fd, std::string\* content) {

content->clear();

char buf[BUFSIZ];

ssize\_t n;

while ((n = TEMP\_FAILURE\_RETRY(read(fd, &buf[0], sizeof(buf)))) > 0) {

content->append(buf, n);

}

return (n == 0) ? true : false;

}

close(fd);

return okay;

}

#### parse\_config

static void parse\_config(const char \*fn, const std::string& data)

{

struct listnode import\_list; //创建一个双向链表system\core\include\cutils\list.h中

// ·struct listnode

{

struct listnode \*next;

struct listnode \*prev;

};

struct listnode \*node; //创建一个节点

char \*args[INIT\_PARSER\_MAXARGS]; //#define INIT\_PARSER\_MAXARGS 64

int nargs = 0;

\*\*\*\*\*\*\*\*parse\_state init start\*\*\*\*\*\*\*\*

此部分总结:parse\_state的初始化

parse\_state state; //感觉是描述解析状态信息的一个类\*\*\*\*\*\*\*\*

//struct parse\_state

{

char \*ptr;

char \*text;

int line;

int nexttoken;

void \*context;

void (\*parse\_line)(struct parse\_state \*state, int nargs, char \*\*args);

const char \*filename;

void \*priv;

};

state.filename = fn; //解析文件的名字赋值

state.line = 0; //行数置0

state.ptr = strdup(data.c\_str()); // TODO: fix this code! //将文件的所有内容拷贝到state里面的ptr成员

state.nexttoken = 0; //置0

state.parse\_line = parse\_line\_no\_op; //state里面的成员函数parse\_line指向parse\_line\_no\_op,是一个空函数

list\_init(&import\_list); //初始化链表

state.priv = &import\_list; //state的priv指向这个链表

\*\*\*\*\*\*\*\*parse\_state init end\*\*\*\*\*\*\*\*

for (;;) {//for循环开始解析

switch (next\_token(&state)) {

case T\_EOF:

state.parse\_line(&state, 0, 0);

goto parser\_done;

case T\_NEWLINE:

state.line++;//返回是新的一行，那么行数加1，然后继续循环，刚开始跳过注释

if (nargs) {//import返回新的一行的时候，nargs不为0

int kw = lookup\_keyword(args[0]); //返回K\_import

#define kw\_is(kw, type) (keyword\_info[kw].flags & (type))

定义一个keyword\_info数组，数组大小等于枚举的大小，初始化第一个元素

其他的由keywords.h导入，由于#define KEYWORD(symbol, flags, nargs, func) \

[ K\_##symbol ] = { #symbol, func, nargs + 1, flags, },每个元素都由头文件初始化了

keyword\_info[KEYWORD\_COUNT] = {

[ K\_UNKNOWN ] = { "unknown", 0, 0, 0 },

#include "keywords.h"

};

\system\core\init

#define KEYWORD(symbol, flags, nargs, func) K\_##symbol,

enum {

K\_UNKNOWN,

KEYWORD(import, SECTION, 1, 0)

if (kw\_is(kw, SECTION)) {//import是SECTION

state.parse\_line(&state, 0, 0); //空实现

parse\_new\_section(&state, kw, nargs, args); state、K\_import..传入

} else {

state.parse\_line(&state, nargs, args);

}

nargs = 0;

}

break;

case T\_TEXT: //import走这里 #define INIT\_PARSER\_MAXARGS 64

if (nargs < INIT\_PARSER\_MAXARGS) {

args[nargs++] = state.text; //数组0，import字符串存这里

}

break;

}

}

parser\_done:

list\_for\_each(node, &import\_list) {

struct import \*import = node\_to\_item(node, struct import, list);

int ret;

ret = init\_parse\_config\_file(import->filename);

if (ret)

ERROR("could not import file '%s' from '%s'\n",

import->filename, fn);

}

}

#### next\_token

int next\_token(struct parse\_state \*state)

{

char \*x = state->ptr;

char \*s;

if (state->nexttoken) {//第一次进入这里0，跳过

int t = state->nexttoken;

state->nexttoken = 0;

return t;

}

for (;;) {

switch (\*x) {//取第一个字符

case 0:

state->ptr = x;

return T\_EOF;

case '\n':

x++;

state->ptr = x;

return T\_NEWLINE;

case ' ':

case '\t':

case '\r':

x++;

continue;

case '#': //表明此行是注释，会跳过

while (\*x && (\*x != '\n')) x++;//字符存在并且不等于换行，则一直加，加到行位

if (\*x == '\n') {//等于换行，表明一行结束

state->ptr = x+1; //+1.返回新的一行

return T\_NEWLINE;

} else {//不是换行符，表明文件已经到末尾，则返回T\_EOF

state->ptr = x;

return T\_EOF;

}

default: //import走这里

goto text;

}

}

textdone:

state->ptr = x;

\*s = 0;

return T\_TEXT;

text:

state->text = s = x;

textresume:

for (;;) {

switch (\*x) {

case 0: //import会一直加，最后走这里0表示结束符，没换行，只是import结束

goto textdone;

case ' ':

case '\t':

case '\r':

x++;

goto textdone;

case '\n':

state->nexttoken = T\_NEWLINE;

x++;

goto textdone;

case '"':

x++;

for (;;) {

switch (\*x) {

case 0:

/\* unterminated quoted thing \*/

state->ptr = x;

return T\_EOF;

case '"':

x++;

goto textresume;

default:

\*s++ = \*x++;

}

}

break;

case '\\':

x++;

switch (\*x) {

case 0:

goto textdone;

case 'n':

\*s++ = '\n';

break;

case 'r':

\*s++ = '\r';

break;

case 't':

\*s++ = '\t';

break;

case '\\':

\*s++ = '\\';

break;

case '\r':

/\* \ <cr> <lf> -> line continuation \*/

if (x[1] != '\n') {

x++;

continue;

}

case '\n':

/\* \ <lf> -> line continuation \*/

state->line++;

x++;

/\* eat any extra whitespace \*/

while((\*x == ' ') || (\*x == '\t')) x++;

continue;

default:

/\* unknown escape -- just copy \*/

\*s++ = \*x++;

}

continue;

default:

\*s++ = \*x++;

}

}

return T\_EOF;

}

## 严苛模式分析

检测程序中违例情况的开发者工具，比如检测主线程中本地磁盘和网络读写等耗时的操作，检测到违例的情况会做出相应的反应，如日志打印，弹出对话框亦或者崩溃等。

主要检测两大问题:

1.线程策略，TreadPolicy

2.VM策略，VmPolicy

常见用法举例:

private boolean DEV\_MODE = true;

public void onCreate() {

if (DEV\_MODE) {//开发模式下启用

StrictMode.setThreadPolicy(new StrictMode.ThreadPolicy.Builder()

.detectCustomSlowCalls() //API等级11，使用StrictMode.noteSlowCode 检测自定义的耗时调用 StrictMode.noteSlowCall("slowCall cost=" + cost);

.detectDiskReads()//检测磁盘读取操作

.detectDiskWrites()//检测磁盘写入操作

.detectNetwork() //检测网络操作

.penaltyDialog() //触发违规时，显示对违规信息对话框

.penaltyLog() //在Logcat中打印违规异常信息 adb logcat | grep StrictMode

.penaltyFlashScreen() //API等级11 闪屏

.penaltyDeathOnNetwork()//当触发网络违规时，Crash掉当前应用程序

.build());

StrictMode.setVmPolicy(new StrictMode.VmPolicy.Builder()

.detectActivityLeaks()//检测Activity泄露,举例 Activity中创建线程new Thread().start,线程未结束时，旋转屏幕。匿名内部类隐式持有外部对象，旋转屏幕，Activity会重新创建

.detectLeakedSqlLiteObjects()//检测Sqlite对象泄漏

.detectLeakedClosableObjects() //API等级11 检测未关闭的Closable对象泄露,举例 file读写未关闭

.setClassInstanceLimit(CastielClass.class, 2)//设置某个类的实例，在内存里的上限

.detectLeakedRegistrationObjects()//BroadcastReceiver 或者 ServiceConnection 注册类对象是否被正确释放

.penaltyLog()

.penaltyDeath()//当触发违规条件时，直接Crash掉当前应用程序

.build());

}

super.onCreate();

}

原理解析，通过StrictMode的setThreadPolicy方法，检测磁盘读取来分析

涉及的源码文件路径:

/frameworks/base/core/java/android/os/StrictMode.java

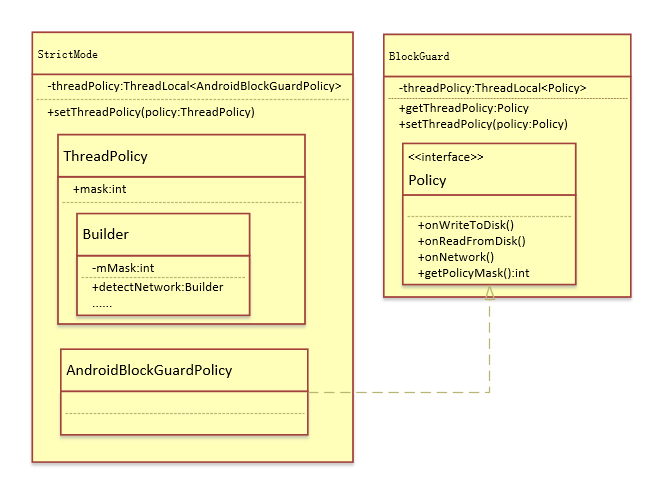
/libcore/dalvik/src/main/java/dalvik/system/BlockGuard.java

/libcore/luni/src/main/java/java/io/FileInputStream.java

/libcore/luni/src/main/java/libcore/io/Streams.java

/libcore/luni/src/main/java/libcore/io/IoBridge.java

/libcore/luni/src/main/java/libcore/io/Libcore.java



首先根据UML图可知，BlockGuard有个内部接口类Policy，和一个线程变量，变量存放的是Policy的一个空实现

看到StrictMode内setThreadPolicy方法

121public final class StrictMode {

....

859 /\*\*

860 \* Sets the policy for what actions on the current thread should

861 \* be detected, as well as the penalty if such actions occur.

862 \*

863 \* <p>Internally this sets a thread-local variable which is

864 \* propagated across cross-process IPC calls, meaning you can

865 \* catch violations when a system service or another process

866 \* accesses the disk or network on your behalf.

867 \*

868 \* @param policy the policy to put into place

869 \*/

870 public static void setThreadPolicy(final ThreadPolicy policy) {

871 setThreadPolicyMask(policy.mask);// new StrictMode.ThreadPolicy.Builder().build返回的是一个ThreadPolicy对象，这里取出对象中的mask，在创建对象时，设置检测种类，其实是或上这个mask,比如检测磁盘的读取操作detectDiskReads，就是将mask|DETECT\_DISK\_READ

872 }

873

874 private static void setThreadPolicyMask(final int policyMask) {

875 // In addition to the Java-level thread-local in Dalvik's

876 // BlockGuard, we also need to keep a native thread-local in

877 // Binder in order to propagate the value across Binder calls,

878 // even across native-only processes. The two are kept in

879 // sync via the callback to onStrictModePolicyChange, below.

880 setBlockGuardPolicy(policyMask);

881

882 // And set the Android native version...

883 Binder.setThreadStrictModePolicy(policyMask);

884 }

885

886 // Sets the policy in Dalvik/libcore (BlockGuard)

887 private static void setBlockGuardPolicy(final int policyMask) {

888 if (policyMask == 0) {//mask等于0，就往BlockGuard里面设置一个Policy的空实现,然后返回

889 BlockGuard.setThreadPolicy(BlockGuard.LAX\_POLICY);

890 return;

891 }

892 final BlockGuard.Policy policy = BlockGuard.getThreadPolicy();//取出BlockGuard里面的Policy类型的线程变量

893 final AndroidBlockGuardPolicy androidPolicy;

894 if (policy instanceof AndroidBlockGuardPolicy) {

//如果BlockGuard中的线程变量是AndroidBlockGuardPolicy类型的,转化成AndroidBlockGuardPolicy ,

895 androidPolicy = (AndroidBlockGuardPolicy) policy;

896 } else {

//如果BlockGuard中的变量类型不是AndroidBlockGuardPolicy,那么从StrictMode中的AndroidBlockGuardPolicy类型的线程变量threadAndroidPolicy取出

//然后设置给BlockGuard

897 androidPolicy = threadAndroidPolicy.get();

898 BlockGuard.setThreadPolicy(androidPolicy);

899 }

//将StrictMode中的AndroidBlockGuardPolicy类型的线程变量threadAndroidPolicy 里面的mask设置成传进来的值

900 androidPolicy.setPolicyMask(policyMask);

901 }

....

以上过程就是将BlockGuard中的Policy本地变量，设置成StrictMode中的AndroidBlockGuardPolicy具体实现，并且设置了StrictMode的一个AndroidBlockGuardPolicy属性里mask值

当调用FileInputStream的read(),读取一个字节时

172 @Override public int read() throws IOException {

173 return Streams.readSingleByte(this);

174 }

175

176 @Override public int read(byte[] buffer, int byteOffset, int byteCount) throws IOException {

177 return IoBridge.read(fd, buffer, byteOffset, byteCount);

178 }

Streams的readSingleByte(this);最终还是回到了FileInputStream中的read(byte[] buffer, int byteOffset, int byteCount)函数

34 /\*\*

35 \* Implements InputStream.read(int) in terms of InputStream.read(byte[], int, int).

36 \* InputStream assumes that you implement InputStream.read(int) and provides default

37 \* implementations of the others, but often the opposite is more efficient.

38 \*/

39 public static int readSingleByte(InputStream in) throws IOException {

40 byte[] buffer = new byte[1];

41 int result = in.read(buffer, 0, 1);

42 return (result != -1) ? buffer[0] & 0xff : -1;

43 }

接下来调用到IoBridge.read函数

458 /\*\*

459 \* java.io thinks that a read at EOF is an error and should return -1, contrary to traditional

460 \* Unix practice where you'd read until you got 0 bytes (and any future read would return -1).

461 \*/

462 public static int read(FileDescriptor fd, byte[] bytes, int byteOffset, int byteCount) throws IOException {

463 Arrays.checkOffsetAndCount(bytes.length, byteOffset, byteCount);

464 if (byteCount == 0) {

465 return 0;

466 }

467 try {

468 int readCount = Libcore.os.read(fd, bytes, byteOffset, byteCount);

469 if (readCount == 0) {

470 return -1;

471 }

472 return readCount;

473 } catch (ErrnoException errnoException) {

474 if (errnoException.errno == EAGAIN) {

475 // We return 0 rather than throw if we try to read from an empty non-blocking pipe.

476 return 0;

477 }

478 throw errnoException.rethrowAsIOException();

479 }

480 }

Libcore.os实际是BlockGuardOs对象

19public final class Libcore {

20 private Libcore() { }

21

22 public static Os os = new BlockGuardOs(new Posix());

23}

24

BlockGuardOs

223 @Override public int read(FileDescriptor fd, ByteBuffer buffer) throws ErrnoException, InterruptedIOException {

224 BlockGuard.getThreadPolicy().onReadFromDisk();//从BlockGuard中获取到之前设置的AndroidBlockGuardPolicy,调用到他的onReadFromDisk具体实现

225 return os.read(fd, buffer);//os是Posix里面最终调用了read，native方法，最终调用了read系统调用

226 }

AndroidBlockGuardPolicy

1254 // Part of BlockGuard.Policy interface:

1255 public void onReadFromDisk() {

1256 if ((mPolicyMask & DETECT\_DISK\_READ) == 0) {//这个标志位是否设置

1257 return;

1258 }

1259 if (tooManyViolationsThisLoop()) {//

1260 return;

1261 }//抛出异常

1262 BlockGuard.BlockGuardPolicyException e = new StrictModeDiskReadViolation(mPolicyMask);

1263 e.fillInStackTrace();

1264 startHandlingViolationException(e);

1265 }

## DatabaseHelper

486 DatabaseHelper(Context context) {

487 super(context, LauncherFiles.LAUNCHER\_DB, null, DATABASE\_VERSION); // "launcher.db"

488 mContext = context;

489 mAppWidgetHost = new AppWidgetHost(context, Launcher.APPWIDGET\_HOST\_ID);

490

491 // In the case where neither onCreate nor onUpgrade gets called, we read the maxId from

492 // the DB here

493 if (mMaxItemId == -1) {

494 mMaxItemId = initializeMaxItemId(getWritableDatabase());//"SELECT MAX(\_id) FROM favorites"

495 }

496 if (mMaxScreenId == -1) {

497 mMaxScreenId = initializeMaxScreenId(getWritableDatabase());//"SELECT MAX(\_id) FROM workspaceScreens”

498 }

499 }