Android开发之ProcessState和IPCThreadState类分析

标签： Android ProcessState IPCThreadState Binder Thread

2013-08-04 18:25 3000人阅读 评论(1) 收藏 举报

分类：

Android 源码分析（26）

在Android中ProcessState是客户端和服务端公共的部分，作为Binder通信的基础，ProcessState是一个singleton类，每个
进程只有一个对象，这个对象负责打开Binder驱动，建立线程池，让其进程里面的所有线程都能通过Binder通信。
与之相关的是IPCThreadState，每个线程都有一个IPCThreadState实例登记在Linux线程的上下文附属数据中，主要负责
Binder的读取，写入和请求处理框架。IPCThreadState在构造的时候获取进程的ProcessState并记录在自己的成员变量
mProcess中，通过mProcess可以获得Binder的句柄。

[html] view plain copy print ?

```
frameworks/base/include/binder/ProcessState.h
class ProcessState : public virtual RefBase
{
public:
    static sp<ProcessState> self();   // 单例模式，获取实例

    void setContextObject(const sp<IBinder>& object);
    sp<IBinder> getContextObject(const sp<IBinder>& caller);

    void setContextObject(const sp<IBinder>& object, const String16& name);
    sp<IBinder> getContextObject(const String16& name, const sp<IBinder>& caller);

    void startThreadePool();

    typdef bool (*context_check_func)
(const String16& name, const sp<IBinder>& caller, void* userData);

    bool isContextManager(void) const;
    bool becomeContextManager(context_check_func checkFunc, void* userData);

    sp<IBinder> getStrongProxyForHandle(int32_t handle);
    wp<IBinder> getWeakProxyForHandle(int32_t handle);
```

加载插

```
    void espungeHandle(int32_t handle, IBinder* binder);

    void spawnPooledThread(boot isMain);
private:
    friend class IPCThreadState;
        ProcessState();
        ~ProcessState;
        ProcessState(const ProcessState& o);
    ProcessState& operator=(const ProcessState& o);

    struct hdndle_entry {
        IBinder* binder;
        RefBase::weakref_type* refs;
    };

    handle_entry* lookupHandleLocked(int32_t handle);

    int mDriverFD;       // 打开的binder驱动文件描述符
    void* mVMStart;

    Vector<handle_entry> mHandleToObject;

    bool mManagerContexts;
    context_check_func mBinderContextCheckFunc;
    void* mBinderContextUserData;
    KeyedVector<String16, sp<IBinder> > mContexts;  // 映射，服务名字 和 IBinder对应
    bool mThreadPoolStarted;    // 线程池是否已经创建
volatile int32_t mThreadPoolSeq;    // 这个进程中启动线程个数
};
```

1）获得ProcessState的实例

[html] view plain copy print ?

```
sp<ProcessState> proc(ProcessState::self());
调用函数:
sp<ProcessState> ProcessState::self()
{
    if (gProcess != NULL) return gProcess;
    AutoMutext _l(gProcessMutex);
    if(gProcess == NULL) gProcess = new ProcessState;
    return gProcess;
```

```
}
进入构造函数：
ProcessState::ProcessState() : mDriverFD(open_driver())
            , mVMStart(MAP_FAILED),
            , mManagerContexts(false)
            , mBinderContextCheckFunc(NULL)
            , mBinderContextUserData(NULL)
            , mThradPoolStarted(false)
            , mThreadPoolSeq(1)
{

}
```

加载插

这个构造函数里面调用open_driver()打开了/dev/binder设备驱动文件，返回文件描述符。这样我们就能通过这个mDriverFd

来和binder驱动交互了。

2）创建线程ProcessState::self()->startThreadPool();

```
[html] view plain copy print ?
void ProcessState::startThreadPool()
{
    AutoMutex _l(mLock);
    if(!mThreadPoolStarted) {
        mThreadPoolStarted = true;
        spawnPooledThread(true);
    }
}
void ProcessState::spawnPoolThread(bool isMain)
{
    if (mThreadPoolStarted) {
        int32_t s = android_atomic_add(1, &mThreadPoolSeq);
        sp<Thread> t = new PoolThread(isMain);
        t->run(buf);
    }
}
```

加载插

```
[html] view plain copy print ?
其实这里就是创建一个线程PoolThread，而PoolThread是一个继承于Thread的类。所以调用t->run()之后相当于调用
PoolThread类的threadLoop()函数，我们来看看PoolThread类的threadLoop线程函数。
virtual bool threadLoop()
```

```
{
    IPCThreadState::self()->joinThreadPool(mIsMain);
    // 这里线程函数调用了一次IPCThreadState::self()->joinThreadPool()后就退出了
    return false;
}
```

3）IPCThreadState::self()->joinThreadPool();

我们知道：进程调用spawnPoolThread()创建了一个线程，执行joinThreadPool(),而主线程也是调用这个函数。唯一区别
是参数，主线程调用的joinThreadPool(true)，创建的线程调用的是jointThreadPool(false)。
下面我们来分析下这个函数，首先我们来看看IPCThreadState这个类

[html] view plain copy print ?

```
frameworks/base/include/IPCThreadState.h
class IPCThreadState
{
public:
    static IPCThreadState* self();
    sp<ProcessState> process();
    ......
    void joinThradPool(bool isMain = true);
    status_t transact(int32_t handle, uint32_t code, const Parcel& data, Parcel* reply, uint32_t
    void incStrongHandle(int32_t handle);
    void decStrongHandle(int32_t handle);
    void incWeakHandle(int32_t handle);
    void decWeakHandle(int32_t handle);
private:
    IPCThraedState();
    ~IPCThreadState();
    status_t sendReplay(const Parcel& reply, uint32_t flags);
    status_t waitForResponse(Parcel& reply,  status_t *acquireResult = NULL);
    status_t talkWithDriver(bool doReceice = true);
    status_t writeTransactionData();
    status_t executeCommand();
private:
    sp<ProcessState> mProcess;
    Vector<BBinder> mPendingStrongDerefs;
    Vector<RefBase::weakref_type*> mPendingWeakDerefs;
    Parcel mIn;
    Parcel mOut;
```

```
}
上面是IPCThreadState类常用的几个函数。
IPCThreadState* IPCThreadState::self()
{
    if(gHaveTLS) {  // 第一次进来肯定为false
restart:
        const pthread_key_t k = gTLS;
        IPCThreadState* st = (IPCThreadState*)pthread_getspecific(k);
        if(st) return st;
        return new IPCThreadState;  // new 一个IPCThreadState对象
    }

    if(gShutdown) return NULL;

    pthread_mutex_lock(&gTLSMutex);
    if(!gHaveTLS) {
        // 第一个参数为指向一个键值的指针，第二个参数指明一个destructor函数，当线程结束时调用
        if(phread_key_create(&gTLS, threadDestructor) != 0) {
            pthread_mutex_unlock(&gTLSMutex);
            return NULL;
        }
        gHaveTLS = true;
    }
    pthread_mutex_unlock(&gTLSMutex);
    goto restart;
}
```

下面来说明下线程中特有的线程存储：Thread Specific Data.

在多线程中，所有线程共享程序中变量，如果每一个线程都希望单独拥有它，就需要线程存储了。即一个变量表面看起来是

全局变量，所有线程都可以使用它，它的值在每一个线程都是单独存储的。

用法：

1）创建一个类型为pthread_key_t 类型变量

2）pthread_key_create()创建改变量，第二个参数表上一个清理函数，用来在线程释放该线程存储的时候调用。

3）当线程中需要存储特殊值的时候，可以用pthread_setspecific()，第一个参数为pthread_key_t 变量，第二个参数为void* 变量，可以存储任何类型的值。

4）当需要取出存储值的时候，调用pthread_getspecific()，返回void*类型变量值。

好了我们现在知道pthread_key_t是干什么用的了？既然代码中有pthread_getspecific()获取IPCThreadState*对象的函数

那么肯定有设置这个变量值的地方？我们找到IPCThreadState的构造函数：

[html] view plain copy print ?

```
IPCThreadState:IPCThreadState()
    : mProcess(ProcessState::self()),
      mMyThreadId(androidGetTid()),
      mStrictModePolicy(0),
      mLastTransactionBinderFlags(0)
{
    pthread_setspecific(gTLS, this);    // 设置为当前this 指针
    clearCaller();
    mIn.setDataCapacity(256);   // 这里mIn 和 mOut分别表示Binder输入输出的变量，我们后面分析
    mOut.setDataCapacity(256);
}
```

加载捱

[html] view plain copy print ?

```
最后进入IPCThreadState::joinThreadPool(bool isMain)
void IPCThreadState::joinThreadPool(bool isMain)    // 默认为true
{
    mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
    do {
        int32_t cmd;

        if(mIn.dataPosition() >= mIn.dataSize()){
            size_t numPending = mPendingWeakDerefs.size();
            if(numPending > 0) {
                for(size_t i = 0; i < numPending; i++) {
                    RefBase::weakref_type* refs = mPendingWeakDerefs[i];
                    refs->decWeak(mProcess.get);
                }
                mPendingWeakDerefs.clear();
            }
            numPending = mPendingStrongDerefs.size();
            if(numPending > 0) {
                for(sizt_t i = 0; i < numPending; i++) {
                    BBinder* obj = mPendingStrongDerefs[i];
                    obj->decStrong(mProcess.get);
                }
                mPendingStrongDerefs.clear();
            }
```

加载捱

```
        }
        // 读取下一个command进行处理
        result = talkWithDriver();// 来等待Client的请求
        if(result >= NO_ERROR) {
            size_t IN = mIn.dataAvail();
            if(IN < sizeof(int32_t)) continue;
            cmd = mIn.readInt32();
        }
        result = executeCommand(cmd);

        if(result == TIMED_OUT && !isMain)
            break;
    } while(result != -ECONNREFUSED && result != -EBADF);

    mOut.writeInt32(BC_EXIT_LOOPER);
    talkWithDriver(false);
}
```

这里的talkWithDriver()里面之际调用的是ioctl(mProcess->mDriverFD, BINDER_WRITE_READ, &bwr)从/dev/binder读取

Client端发过来的请求，然后调用executeCommand处理

[html] view plain copy print ?

```
status_t IPCThreadState::executeCommand(int32_t cmd)
{
    BBinder* obj;
    RefBase::weakref_type* refs;
    status_t result = NO_ERROR;

    switch(cmd) {
        case BR_TRANSACTION:
        binder_transaction_data tr;
        result = mIn.read(&tr, sizeof(tr));
        ....
        Parcel reply;
        if(tr.target.ptr) {
            sp<BBinder> b((BBinder*)tr.cookie);
            const status_t error  = b->transact(tr.code, buffer, &reply, tr.flags);
        }
        ....
        break;

```

加载捱

```
    }
}
```

最后又调用到BBinder 的transact()函数里面去了。

```
status_t BBinder::transact(uint32_t code, const Parcel& data, Parcel* reply, uint32_t flags )
{
    data.setDataPosition(0);
    switch(code)
    {
        case PING_TRANSACTION:
            reply->writeInt32(pingBinder());
            break;
        default:
            err = onTransact(code, data, reply, flags);
            break;
    }
    return err;
}
```

到这里IPCThreadState类的流程就大概清楚了，线程调用joinThreadPool()从/dev/binder读取客户端的请求，然后调用

BBinder::transact()处理。那么这个BBinder是怎么来的呢？

上面代码中：sp<BBinder> b((BBinder*)tr.cookie)说明这个BBinder指针是从Binder驱动中获取到，肯定是客户端

发送过来的，那么它的实际类型又是什么呢？而BBinder调用的onTransact()函数只是一个虚函数，肯定由它的子类来实

现，那我们服务端又怎么找到这个BBinder的实际类型呢？

这些内容我们下一节通过MediaPlayer这个具体示例分析。

顶
4