



~(^ _ ^) 三月学长的根据地 (^ _ ^)~



Android Binder 分析——多线程支持

By Mingming

© 2015 1月 28 更新日期:2017 2月 7

前面普通服务篇那里说到 ActivityManager (AM) 里锁的问题, 其实不光 AM, WindowManager (WM)、PackageMananger (PM) 中基本上很多对外的业务函数里面都是加锁的, 所以这些 SS 里面有会有带 Locked 结尾的函数 (这些函数都是在锁里执行)。这里就提出一个疑问为什么要加锁。这篇就来解答这个问题, 顺带扯出 binder 的多线程支持的问题。

文章目录

1. SS 多线程的例子
2. 进程对象和线程对象
3. 多线程接口实现.上
4. android 中的线程
5. 多线程接口实现.下
6. 自动创建线程
7. java 层的 SS
8. 总结

照例先把相关源码位置啰嗦一下 (4.4) :

```
1  # java 层 SS 相关代码
2  frameworks/base/services/java/com/android/server/SystemServer.java
3
4  # java 层 zygote 相关接口
5  frameworks/base/core/java/com/android/internal/os/ZygoteInit.java
6  frameworks/base/core/java/com/android/internal/os/RuntimeInit.java
7  libcore/dalvik/src/main/java/dalvik/system/Zygote.java
8
9  # jni runtime 接口
10 frameworks/base/core/jni/AndroidRuntime.cpp
11
12 # native app process 程序
13 frameworks/base/cmds/app_process/app_main.cpp
14
15 # native binder 库
```

```
16 frameworks/native/libs/binder/ProcessState.cpp
17 frameworks/native/libs/binder/IPCThreadState.cpp
18 frameworks/native/libs/binder/Static.cpp
19
20 # native SS 主程序
21 frameworks/native/services/surfaceflinger/main_surfaceflinger.cp
22 frameworks/native/services/sensorsservice/main_sensorsservice.cpp
23
24 # libutils 库 (线程支持)
25 system/core/include/utils/AndroidThreads.h
26 system/core/include/utils/Thread.h
27 system/core/libutils/Threads.cpp
28
29 # kernel binder 驱动
30 kernel/drivers/staging/android/binder.h
31 kernel/drivers/staging/android/binder.c
32
```

SS 多线程的例子

前面说了 binder 是 C/S 模型。既然是 C/S 模型，很容易让人想到会服务器同时可能会被多个客户端连接、请求。传统的服务器，都会开一个线程池来应对这种情况，避免请求数一多，某些客户端长时间没响应的情况。

这个在 binder 中是一样的，只不过 binder 是本地的而已（这里说的本地是相对网络上的服务器来说的）。你想想你写 android 代码的时候，经常就 get AM，然后 AM xx 的调用了。所以说 binder 中的 Bn 端经常会同一时候有多个 Bp 请求的，特别是 SS（谁让它们都是公用的）。既然这样的话，我们在写服务的时候是不是也要像传统的服务器一样考虑使用线程池来提高服务的并发响应能力咧。先别急，android 早就帮我们整好一个现成的框架了。

我们先来看看一些实际的例子。我们以 sensorsservice 为例（先说 native 的 SS，java 层的要绕好久，后面再说）。sensorsservice 的 main 函数：

```
1 // main_sensorsservice.cpp =====
2
3 // 这个 main 函数就一句话 ... ..
4 int main(int argc, char** argv) {
5     SensorService::publishAndJoinThreadPool();
6     return 0;
}
```

```

7   }
8

```

SensorService 是继承自 BinderService 的类，我们去看看 BinderService:

```

1  // BinderService.h =====
2
3  // 这是一个模板类
4  template<typename SERVICE>
5  class BinderService
6  {
7  public:
8      static status_t publish(bool allowIsolated = false) {
9          // 获取 SM
10         sp<IServiceManager> sm(defaultServiceManager());
11         // new Service 对象, 然后 add 到 SM 中
12         return sm->addService(
13             String16(SERVICE::getServiceName()),
14             new SERVICE(), allowIsolated);
15     }
16
17     static void publishAndJoinThreadPool(bool allowIsolated = fa
18         publish(allowIsolated);
19         joinThreadPool();
20     }
21
22     static void instantiate() { publish(); }
23
24     static status_t shutdown() { return NO_ERROR; }
25
26 private:
27     static void joinThreadPool() {
28         // 初始化 ProcessState 对象
29         sp<ProcessState> ps(ProcessState::self());
30         // 启动线程池
31         ps->startThreadPool();
32         ps->giveThreadPoolName();
33         // 当前线程加入线程池
34         IPCThreadState::self()->joinThreadPool();
35     }
36 };
37

```

这个 BinderService.h 是在 native binder 的 include 头文件中，看样子 android 直接提供了一个模板，SS 直接继承，然后在 main 函数调用一句话就行了。模

板就是先 new 一个 Service 对象，然后 add 到 SM 中，然后构造一个 ProcessState 对象（这个对象一个进程只有一个），调用 ProcessState->startThreadPool 开线程池（新开了一个线程），最后调用 IPCThreadState->joinThreadPool 将当前进程加入到线程池中（开始阻塞等待了）。

进程对象和线程对象

new Service 对象不说了，不同的服务逻辑不一样的。先来看 ProcessState。这个东西前面通信篇介绍过的，一个进程就一个对象，怎么保证的呢，来看它的典型调用方法：

```

1  // Static.cpp =====
2
3  // 原来整了一个静态变量噻
4  Mutex gProcessMutex;
5  sp<ProcessState> gProcess;
6
7  // ProcessState.h =====
8
9  class ProcessState : public virtual RefBase
10 {
11 public:
12     // 这个也是一个静态函数
13     static sp<ProcessState>    self();
14
15     ... ..
16
17 };
18
19 // ProcessState.cpp =====
20
21 sp<ProcessState> ProcessState::self()
22 {
23     Mutex::Autolock _l(gProcessMutex);
24     if (gProcess != NULL) {
25         return gProcess;
26     }
27     gProcess = new ProcessState;
28     return gProcess;
29 }
30

```

就是搞了一个静态变量，然后一个进程只会创建一次，之后就直接返回这个静态变量就行了，借此来保证一个进程就只有一个 ProcessState 对象，怪不得叫 ProcessState。然后 self() 是静态的，而且无参数，所以进程任何地方任何时候都能够使用。

既然说到了 ProcessState 的 self()，我们顺带也来看下 IPCThreadState 的 self()。看名字这个应该就是一个线程有一个这样对象的了。这个是怎么实现的咧：

```

1  // IPCThreadState.h =====
2
3  class IPCThreadState

```

```
4  {
5  public:
6      static IPThreadState*      self();
7
8      ... ..
9
10 };
11
12 // IPThreadState.cpp =====
13
14 static pthread_mutex_t gTLMutex = PTHREAD_MUTEX_INITIALIZER;
15 static bool gHaveTLS = false;
16 static pthread_key_t gTLS = 0;
17 static bool gShutdown = false;
18 static bool gDisableBackgroundScheduling = false;
19
20 IPThreadState* IPThreadState::self()
21 {
22     // 首先得确保 pthread_key 创建了
23     if (gHaveTLS) {
24 restart:
25         const pthread_key_t k = gTLS;
26         // 如果之前设置了线程私有变量, 取线程私有变量返回
27         IPThreadState* st = (IPThreadState*)pthread_getspecifici
28         if (st) return st;
29         // 否则 new 一个新对象
30         return new IPThreadState;
31     }
32
33     // 如果线程已经退出了, 就直接返回 NULL 了
34     if (gShutdown) return NULL;
35
36     // pthread_key 没创建的话, 先创建 pthread_key
37     pthread_mutex_lock(&gTLMutex);
38     if (!gHaveTLS) {
39         if (pthread_key_create(&gTLS, threadDestructor) != 0) {
40             pthread_mutex_unlock(&gTLMutex);
41             return NULL;
42         }
43         gHaveTLS = true;
44     }
45     pthread_mutex_unlock(&gTLMutex);
46     // 回去取线程私有变量
47     goto restart;
48 }
49
```

IPCThreadState 的 self() 稍微复杂点，因为一个进程有多个线程，要保证一个线程一个对象，这个时候就有利用 linux pthread 的线程私有变量，这个东西可以给每一个 pthread 线程（android 使用 linux 的 pthread 作为多线程的支持）创建一个线程独立的私有变量，这个变量是线程私有的，不用考虑多线程的互斥、锁问题。利用这个特性就很容易显示一个线程一个对象了，把 IPCThreadState 对象保存为 pthread 的线程私有变量就行了，用得时候就去取线程私有变量，每个线程都是自己的。有了这些知识，上面的代码就不难理解了，就是开始看有没有设置线程私有变量（取之前得先创建 pthread_key，这方面的用法可以去看《UNIX环境高级编程》这本书），有的话直接去线程私有变量就行了，没有的话就 new 一个 IPCThreadState 出来。但是是在哪设置线程私有变量的咧，我们来看看 IPCThreadState 的构造函数：

```

1  IPCThreadState::IPCThreadState()
2      : mProcess(ProcessState::self()),
3        mMyThreadId(androidGetTid()),
4        mStrictModePolicy(0),
5        mLastTransactionBinderFlags(0)
6  {
7      // 就是这里把自己的对象设置为线程私有变量啦
8      pthread_setspecific(gTLS, this);
9      clearCaller();
10     mIn.setDataCapacity(256);
11     mOut.setDataCapacity(256);
12 }
13
```

第一句就是设置。然后我们回去看下 `pthread_key_create` 第二参数 threadDestructor，这是一个函数指针，是说线程退出的话，就调用这个函数，我们看看里面做了什么：

```

1  void IPCThreadState::threadDestructor(void *st)
2  {
3      IPCThreadState* const self = static_cast<IPCThreadState*
4      if (self) {
5          self->flushCommands();
6      #if defined(HAVE_ANDROID_OS)
7          if (self->mProcess->mDriverFD > 0) {
8              // 向 binder 发送线程退出的命令
9              ioctl(self->mProcess->mDriverFD, BINDER_THREAD_EXIT,
```

```

10         }
11     #endif
12         delete self;
13     }
14 }
15

```

这个退出函数调用 `ioctl` 对 binder 驱动发送了一条 `BINDER_THREAD_EXIT` 退出的消息。这个消息后面再说，不过到最后你会发现这个函数其实暂时没用的。

现在我们知道了在进程任何地方调用 `ProcessState::self()` 就能取到本进程的进程对象，并且可以调用一些 binder 进程相关的接口，在进程任何地方调用 `IPCThreadState::self()` 就能取得当前线程的对象，并且可以调用一些 binder 线程相关的接口。

多线程接口实现.上

其实前面调用的接口就2个：`ProcessState` 的 `startThreadPool` 和 `IPCThreadState` 的 `joinThreadPool`。我们先来看 `startThreadPool`：

```

1  void ProcessState::startThreadPool()
2  {
3      AutoMutex _l(mLock);
4      // 看样子这个判断，startThreadPool 的调用只会有效一次
5      if (!mThreadPoolStarted) {
6          mThreadPoolStarted = true;
7          spawnPooledThread(true);
8      }
9  }
10

```

然后看下 `spawnPooledThread`：

```

1  // 上面传递过来的 isMain 是 true
2  void ProcessState::spawnPooledThread(bool isMain)
3  {
4      if (mThreadPoolStarted) {
5          String8 name = makeBinderThreadName();
6          ALOGV("Spawning new pooled thread, name=%s\n", name.toString());
7          sp<Thread> t = new PoolThread(isMain);

```



```

8         t->run(name.string());
9     }
10 }
11

```

这个函数很简单，主要是 new 了一个 PoolThread 对象，然后调用了它的 run 方法。这里就先要暂停一下，插说下 android 线程的东西。

android 中的线程

android 中的线程到底是啥东西。这里先说下答案：就是 pthread，native 层的从代码可以看得出，java 层的虽然是虚拟机实现的，但是应该还是用 pthread 实现的（我猜的，没去看代码）。这里就看下 native 的就行了。

首先之前在 ProcessState 中 new 的 PoolThread：

```

1  class PoolThread : public Thread
2  {
3  public:
4      PoolThread(bool isMain)
5          : mIsMain(isMain)
6      {
7      }
8
9  protected:
10     virtual bool threadLoop()
11     {
12         IPCThreadState::self()->joinThreadPool(mIsMain);
13         return false;
14     }
15
16     const bool mIsMain;
17 };
18

```

PoolThread 很简单，它继承自 libutils 里面的 Thread：

```

1  class Thread : virtual public RefBase
2  {
3  public:
4      // Create a Thread object, but doesn't create or start the a

```

```

5      // thread. See the run() method.
6          Thread(bool canCallJava = true);
7      virtual ~Thread();
8
9      // Start the thread in threadLoop() which needs to be implem
10     virtual status_t run(    const char* name = 0,
11                             int32_t priority = PRIORITY_DEFA
12                             size_t stack = 0);
13
14     ... ..
15
16     private:
17         // Derived class must implement threadLoop(). The thread sta
18         // here. There are two ways of using the Thread object:
19         // 1) loop: if threadLoop() returns true, it will be called
20         //         requestExit() wasn't called.
21         // 2) once: if threadLoop() returns false, the thread will e
22         virtual bool threadLoop() = 0;
23
24     ... ..
25
26 };
27

```

这个就算是基类了（父类都是我讨厌的那个引用计数的玩意了）。这类就封装了 android 线程的实现。有2个重要的函数，看注释。一个是 run，看注释说 new 了 Thread 对象后，线程并没有执行，要调用 run 才会跑的。第二是 threadLoop，子类要重载这个函数，就是线程真正的执行函数（回调），如果返回 false，调用一次线程就会结束，如果返回 true，这个函数会循环调用执行。我们看到 PoolThread 重载了 threadLoop 这个函数，在里面调用了 IPCThread 的 joinThreadPool，然后返回 false。

我们一个一个看，首先看下 Thread 的构造函数：

```

1  Thread::Thread(bool canCallJava)
2      :    mCanCallJava(canCallJava),
3          mThread(thread_id_t(-1)),
4          mLock("Thread::mLock"),
5          mStatus(NO_ERROR),
6          mExitPending(false), mRunning(false)
7  #ifdef HAVE_ANDROID_OS
8      , mTid(-1)
9  #endif

```

```
10 {
11 }
12
13 Thread::~Thread()
14 {
15 }
16
```

构造函数挺简单，我们这里关心的是 `mCanCallJava` 这个 `bool` 值，默认是 `true`，这个值表示在 `native` 的线程能不能调用虚拟机 `java` 的环境（例如利用个反射，调用一些 `java` 层的函数等等）。然后我们看 `run`：

```
1  status_t Thread::run(const char* name, int32_t priority, size_t
2  {
3      Mutex::Autolock _l(mLock);
4
5      // 正在运行的，不允许再执行
6      if (mRunning) {
7          // thread already started
8          return INVALID_OPERATION;
9      }
10
11     // reset status and exitPending to their default value, so w
12     // try again after an error happened (either below, or in re
13     mStatus = NO_ERROR;
14     mExitPending = false;
15     mThread = thread_id_t(-1);
16
17     // hold a strong reference on ourself
18     mHoldSelf = this;
19
20     mRunning = true;
21
22     // 这里，前面说了默认是 true，就是走上面那个分支
23     bool res;
24     if (mCanCallJava) {
25         res = createThreadEtc(_threadLoop,
26                             this, name, priority, stack, &mThread);
27     } else {
28         res = androidCreateRawThreadEtc(_threadLoop,
29                                         this, name, priority, stack, &mThread);
30     }
31
32     if (res == false) {
33         mStatus = UNKNOWN_ERROR;    // something happened!
34         mRunning = false;
```

```

35         mThread = thread_id_t(-1);
36         mHoldSelf.clear(); // "this" may have gone away after t
37
38         return UNKNOWN_ERROR;
39     }
40
41     // Do not refer to mStatus here: The thread is already runni
42     // already have exited with a valid mStatus result). The NO_
43     // here merely indicates successfully starting the thread an
44     // imply successful termination/execution.
45     return NO_ERROR;
46
47     // Exiting scope of mLock is a memory barrier and allows new
48 }
49

```

这个函数其实最关键就是 mCanCallJava 这里的这个2个分支，这才是创建线程的地方。mCanCallJava 前面说了默认是 true，就是说一般是走前面那个分支的，但是这里我们先说下面那个分支（至于原因后面你就知道了）：

```

1  // entryFunction 是线程执行函数指针
2  int androidCreateRawThreadEtc(android_thread_func_t entryFunction
3                                void *userData,
4                                const char* threadName,
5                                int32_t threadPriority,
6                                size_t threadStackSize,
7                                android_thread_id_t *threadId)
8  {
9      // 呵呵，是 pthread 的调用了吧
10     pthread_attr_t attr;
11     pthread_attr_init(&attr);
12     pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
13
14     #ifdef HAVE_ANDROID_OS /* valgrind is rejecting RT-priority cre
15     if (threadPriority != PRIORITY_DEFAULT || threadName != NULL
16         // Now that the pthread_t has a method to find the assoc
17         // android_thread_id_t (pid) from pthread_t, it would be
18         // this trampoline in some cases as the parent could set
19         // for the child. However, there would be a race condit
20         // child becomes ready immediately, and it doesn't work
21         // prctl(PR_SET_NAME) only works for self; prctl(PR_SET_
22         // proposed but not yet accepted.
23         thread_data_t* t = new thread_data_t;
24         t->priority = threadPriority;
25         t->threadName = threadName ? strdup(threadName) : NULL;
26         t->entryFunction = entryFunction;

```

```

27         t->userData = userData;
28         entryFunction = (android_thread_func_t)&thread_data_t::t
29         userData = t;
30     }
31 #endif
32
33     if (threadStackSize) {
34         pthread_attr_setstacksize(&attr, threadStackSize);
35     }
36
37     errno = 0;
38     pthread_t thread;
39     int result = pthread_create(&thread, &attr,
40                               (android_pthread_entry)entryFunction, userDa
41     pthread_attr_destroy(&attr);
42     if (result != 0) {
43         ALOGE("androidCreateRawThreadEtc failed (entry=%p, res=%
44              "(android threadPriority=%d)",
45              entryFunction, result, errno, threadPriority);
46         return 0;
47     }
48
49     // Note that *threadID is directly available to the parent c
50     // assigned after the child starts. Use memory barrier / lc
51     // or other threads also need access.
52     if (threadId != NULL) {
53         *threadId = (android_thread_id_t)thread; // XXX: this is
54     }
55     return 1;
56 }
57

```

前面也好说了对 pthread 的接口不太清楚的去看《UNIX高级环境编程》。这里简单说下，传了线程执行函数、线程名字、优先级、线程堆栈、用户自定义数据进来，然后那个 threadId 是个输出参数，返回的是线程的 id 号（tid，类似 pid）。函数返回后，`pthread_create` 就会创建线程，并且执行 entryFunction。这里传过来的 entryFunction 是 `_threadLoop`：

```

1  int Thread::_threadLoop(void* user)
2  {
3      Thread* const self = static_cast<Thread*>(user);
4
5      sp<Thread> strong(self->mHoldSelf);
6      wp<Thread> weak(strong);

```

```
7         self->mHoldSelf.clear();
8
9         #ifdef HAVE_ANDROID_OS
10            // this is very useful for debugging with gdb
11            self->mTid = gettid();
12        #endif
13
14        bool first = true;
15
16        do {
17            // 还是调用 threadLoop 的
18            bool result;
19            if (first) {
20                first = false;
21                self->mStatus = self->readyToRun();
22                result = (self->mStatus == NO_ERROR);
23
24                if (result && !self->exitPending()) {
25                    // Binder threads (and maybe others) rely on thr
26                    // running at least once after a successful ::re
27                    // (unless, of course, the thread has already be
28                    // at that point).
29                    // This is because threads are essentially used
30                    // (new ThreadSubclass())->run();
31                    // The caller therefore does not retain a strong
32                    // the thread and the thread would simply disapp
33                    // successful ::readyToRun() call instead of ent
34                    // threadLoop at least once.
35                    result = self->threadLoop();
36                }
37            } else {
38                result = self->threadLoop();
39            }
40
41            // 知道前面的注释为什么说返回 false 只会执行一次了
42            // establish a scope for mLock
43            {
44                Mutex::Autolock _l(self->mLock);
45                if (result == false || self->mExitPending) {
46                    self->mExitPending = true;
47                    self->mRunning = false;
48                    // clear thread ID so that requestExitAndWait() does
49                    // called by a new thread using the same thread ID a
50                    self->mThread = thread_id_t(-1);
51                    // note that interested observers blocked in request
52                    // awoken by broadcast, but blocked on mLock until b
53                    self->mThreadExitedCondition.broadcast();
54                    break;
55                }
56            }
57        }
```

```

56         }
57
58         // Release our strong reference, to let a chance to the
59         // to die a peaceful death.
60         strong.clear();
61         // And immediately, re-acquire a strong reference for th
62         strong = weak.promote();
63     } while(strong != 0);
64
65     return 0;
66 }
67

```

线程函数，一旦执行完，线程就会退出。这里的 `_threadLoop` 里面是一个循环，取决于 `threadLoop` 这个的返回值（怪不得叫 `loop`）。所以 `Thread` 的子类只要重载 `threadLoop` 然后根据自己的需要返回合适值就不用管线程的创建、运行之类的了。

然后我们回去看看 `mCanCallJava` 那个分支上的那个函数：

```

1  // AndroidThreads.h =====
2
3  // inline 又挂马甲
4  // Create thread with lots of parameters
5  inline bool createThreadEtc(thread_func_t entryFunction,
6                             void *userData,
7                             const char* threadName = "android:ur
8                             int32_t threadPriority = PRIORITY_DE
9                             size_t threadStackSize = 0,
10                             thread_id_t *threadId = 0)
11  {
12      return androidCreateThreadEtc(entryFunction, userData, threa
13      threadPriority, threadStackSize, threadId) ? true : fals
14  }
15
16  // Threads.cpp =====
17
18  // 这个函数指针一开始指向的就是 mCanCallJava == false 那个分支的那个函数
19  static android_create_thread_fn gCreateThreadFn = androidCreateF
20
21  int androidCreateThreadEtc(android_thread_func_t entryFunction,
22                             void *userData,
23                             const char* threadName,
24                             int32_t threadPriority,
25                             size_t threadStackSize,

```

```

26         android_thread_id_t *threadId)
27     {
28         // 是调用上面那个函数指针说指向的函数
29         return gCreateThreadFn(entryFunction, userData, threadName,
30             threadPriority, threadStackSize, threadId);
31     }
32
33     // 有接口设置的哦
34     void androidSetCreateThreadFunc(android_create_thread_fn func)
35     {
36         gCreateThreadFn = func;
37     }
38

```

这个分支，按照代码写的最开始和 `mCanCallJava == false` 是一样的。但是注意，Threads 有个接口可以设置这个函数指针的。我搜了一下代码，发现有一个地方设置了这个函数指针：

```

1  /*
2   * Register android native functions with the VM.
3   */
4  /*static*/ int AndroidRuntime::startReg(JNIEnv* env)
5  {
6      /*
7       * This hook causes all future threads created in this process
8       * attached to the JavaVM. (This needs to go away in favor of
9       * Attach calls.)
10     */
11     androidSetCreateThreadFunc((android_create_thread_fn) javaCr
12
13     ALOGV("--- registering native functions ---\n");
14
15     /*
16      * Every "register" function calls one or more things that require
17      * a local reference (e.g. FindClass). Because we haven't yet
18      * started the VM yet, they're all getting stored in the baselink
19      * and never released. Use Push/Pop to manage the storage.
20     */
21     env->PushLocalFrame(200);
22
23     if (register_jni_procs(gRegJNI, NELEM(gRegJNI), env) < 0) {
24         env->PopLocalFrame(NULL);
25         return -1;
26     }
27     env->PopLocalFrame(NULL);
28

```



```

29     //createJavaThread("fubar", quickTest, (void*) "hello");
30
31     return 0;
32 }
33

```

这个是 AndroidRuntime.cpp 里面注册 jni 函数的时候设置的。这个前面说 SystemService 好像有说过启 Zygote 会调用这个东西。反正就是初始化 java 虚拟机的时候设置了，看注释是为能让在 natvie 的线程中调用 java 中的东西。设置的函数是 javaCreateThreadEtc:

```

1  /*
2   * This is invoked from androidCreateThreadEtc() via the callbac
3   * set with androidSetCreateThreadFunc().
4   *
5   * We need to create the new thread in such a way that it gets h
6   * into the VM before it really starts executing.
7   */
8  /*static*/ int AndroidRuntime::javaCreateThreadEtc(
9      android_thread_func_t entryFuncnt
10     void* userData,
11     const char* threadName,
12     int32_t threadPriority,
13     size_t threadStackSize,
14     android_thread_id_t* threadId)
15  {
16     void** args = (void**) malloc(3 * sizeof(void*)); // javaT
17     int result;
18
19     assert(threadName != NULL);
20
21     // 真正的线程执行函数
22     args[0] = (void*) entryFunction;
23     // 用户自定义数据
24     args[1] = userData;
25     // 线程名字
26     args[2] = (void*) strdup(threadName); // javaThreadShell m
27
28     // 知道前面为什么先说这个函数了吧，最后还是调用同一个
29     result = androidCreateRawThreadEtc(AndroidRuntime::javaThrea
30         threadName, threadPriority, threadStackSize, threadId);
31     return result;
32 }
33

```

知道前面为什么先说 androidCreateRawThreadEtc 了不，其中最终还是调用同一个，所以之后都是 pthread 的调用。但是执行函数不一样，这里的是 javaThreadShell:

```

1  /*
2   * When starting a native thread that will be visible from the v
3   * bounce through this to get the right attach/detach action.
4   * Note that this function calls free(args)
5   */
6  /*static*/ int AndroidRuntime::javaThreadShell(void* args) {
7      void* start = ((void**)args)[0];
8      void* userData = ((void**)args)[1];
9      char* name = (char*) ((void**)args)[2];          // we own th
10     free(args);
11     JNIEnv* env;
12     int result;
13
14     // 就是多了一个 java attach
15     /* hook us into the VM */
16     if (javaAttachThread(name, &env) != JNI_OK)
17         return -1;
18
19     /* start the thread running */
20     result = (*(android_thread_func_t)start)(userData);
21
22     // 和 unattach 吧
23     /* unhook us */
24     javaDetachThread();
25     free(name);
26
27     return result;
28 }
29

```

这里就多了一个 attach 到 JVM 中。这里不多分析这个，因为这篇不是主要说 JVM 的，反正这么整一下之后，native 的线程对 JVM 可见，native 的线程也可以调用 java 的东西。

多线程接口实现.下

简单的说了下 android 的线程后，回来继续到 ProcessState 中。前面说了

spawnPooledThread 那里 new 了一个 PoolThread 出来，然后跑 run，之后就会执行 PoolThread 的 threadLoop：

```

1  virtual bool threadLoop()
2  {
3      IPCThreadState::self()->joinThreadPool(mIsMain);
4      return false;
5  }
6

```

调用的正好是我们下面要说的 IPCThreadState 的 joinThreadPool。这里 mIsMain，ProcessState startThreadPool 转过来的是 true。然后 threadLoop 返回的是 false。就是说这个函数只会执行一次，但是 binder 的 Bn 端不是应该循环等待 Bp 的请求么，往下看你就知道只要一次就够了。

前面那个模板 BinderService 最后也调用了 joinThreadPool 了，这个函数是然当前线程加入到线程池，是当前线程。所以那个模板 BinderService 是主线程加入，这里是 new PoolThread 的线程（pthread 创建的）。我来具体看下：

```

1  void IPCThreadState::joinThreadPool(bool isMain)
2  {
3      LOG_THREADPOOL("**** THREAD %p (PID %d) IS JOINING THE THREA
4
5      mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPE
6
7      // This thread may have been spawned by a thread that was in
8      // scheduling group, so first we will make sure it is in the
9      // one to avoid performing an initial transaction in the bac
10     set_sched_policy(mMyThreadId, SP_FOREGROUND);
11
12     status_t result;
13     do {
14         processPendingDerefs();
15         // now get the next command to be processed, waiting if
16         result = getAndExecuteCommand();
17
18         if (result < NO_ERROR && result != TIMED_OUT && result !
19             ALOGE("getAndExecuteCommand(fd=%d) returned unexpect
20                 mProcess->mDriverFD, result);
21             abort();
22     }
23

```

```

24         // Let this thread exit the thread pool if it is no long
25         // needed and it is not the main process thread.
26         if(result == TIMED_OUT && !isMain) {
27             break;
28         }
29     } while (result != -ECONNREFUSED && result != -EBADF);
30
31     LOG_THREADPOOL("***** THREAD %p (PID %d) IS LEAVING THE THREA
32         (void*)pthread_self(), getpid(), (void*)result);
33
34     mOut.writeInt32(BC_EXIT_LOOPER);
35     talkWithDriver(false);
36 }
37

```

这个函数其实之前通信模型篇里有说过。里面有个 do while 的循环（前面知道为什么那个 threadLoop 返回 false 了吧，一次就够了，一直在循环咧）。那个 isMain 的区别就是，如果 isMain 是 false，getAndExecuteCommand 如果返回是 `TIMED_OUT` 的话就会退出这个线程。然后 getAndExecuteCommand 返回 `TIMED_OUT` 的条件是 kernel binder 驱动给你返回 `BR_FINISHED`，但是目前的 binder 驱动（通信协议版本 version7）根本就没返回 `BR_FINISHED` 这个值，所以说目前这个 isMain 可以忽略不计，可以认为都是 true。

getAndExecuteCommand 这些我就不说了，通信模型篇里说得比较清楚了。但是这里你会有个疑问，按照 BinderService 的写法，目前也就只有2个线程再等待 Bp 端的请求而已。如果同一个时候的 Bp 多于2个的话，不是要等待么。别急，binder 在 kernel 会自动帮你处理这种情况的。接下来我们要去 kernel 里面看看 binder 驱动的处理。

自动创建线程

通信模型篇，我们知道 binder 驱动中结构体 `binder_proc` 代表一个进程，结构体 `binder_thread` 代表一个线程。我们先看看 `binder_proc` 中几个变量：

```

1  struct binder_proc {
2

```

```

3    ... ..
4
5    // 保存这个进程中所有线程的红黑树的根节点
6    struct rb_root threads;
7
8    // 运行开启的最大线程数
9    int max_threads;
10   // 请求开启的线程数
11   int requested_threads;
12   // 应请求运行的线程数
13   int requested_threads_started;
14   // 准备好的线程（空闲的线程）
15   int ready_threads;
16
17   ... ..
18
19   };
20

```

这个结构在 `binder_open` 的时候会创建（`binder open` 的操作在 `ProcessState` 的构造函数那，所以一个进程只会 `open` 一次，所以一个进程 `binder_proc` 只会创建一次）：

```

1  static int binder_open(struct inode *nodp, struct file *filp)
2  {
3      struct binder_proc *proc;
4
5      binder_debug(BINDER_DEBUG_OPEN_CLOSE, "binder_open: %d:%d\n",
6                  current->group_leader->pid, current->pid);
7
8      proc = kzalloc(sizeof(*proc), GFP_KERNEL);
9      if (proc == NULL)
10         return -ENOMEM;
11     get_task_struct(current);
12     proc->tsk = current;
13     INIT_LIST_HEAD(&proc->todo);
14     init_waitqueue_head(&proc->wait);
15     proc->default_priority = task_nice(current);
16     mutex_lock(&binder_lock);
17     binder_stats_created(BINDER_STAT_PROC);
18     hlist_add_head(&proc->proc_node, &binder_procs);
19     proc->pid = current->group_leader->pid;
20     INIT_LIST_HEAD(&proc->delivered_death);
21     filp->private_data = proc;
22     mutex_unlock(&binder_lock);
23

```

```

24     if (binder_debugfs_dir_entry_proc) {
25         char strbuf[11];
26         snprintf(strbuf, sizeof(strbuf), "%u", proc->pid);
27         proc->debugfs_entry = debugfs_create_file(strbuf, S_IRUGO
28             binder_debugfs_dir_entry_proc, proc, &binder_proc_fc
29     }
30
31     return 0;
32 }
33

```

没有显示的初始化那几个变量，不过默认都是 0。然后来看 `binder_thread` 的创建：

```

1  static long binder_ioctl(struct file *filp, unsigned int cmd, un
2  {
3      int ret;
4      struct binder_proc *proc = filp->private_data;
5      struct binder_thread *thread;
6      unsigned int size = _IOC_SIZE(cmd);
7      void __user *ubuf = (void __user *)arg;
8
9      /*printk(KERN_INFO "binder_ioctl: %d:%d %x %lx\n", proc->pid
10
11      ret = wait_event_interruptible(binder_user_error_wait, binde
12      if (ret)
13          return ret;
14
15      mutex_lock(&binder_lock);
16      thread = binder_get_thread(proc);
17      if (thread == NULL) {
18          ret = -ENOMEM;
19          goto err;
20      }
21
22      ... ..
23
24  }
25

```

`binder_thread` 的创建在 `binder_ioctl` 的 `binder_get_thread` 这个函数中。只要调用 binder 的 ioctl 接口就会触发这个：

```

1  // 咋顺带把这个结构体也贴一下

```

```

2  struct binder_thread {
3      // 这个线程所在的进程对象
4      struct binder_proc *proc;
5      struct rb_node rb_node;
6      int pid;
7      // 线程当前运行状态
8      int looper;
9      struct binder_transaction *transaction_stack;
10     struct list_head todo;
11     uint32_t return_error; /* Write failed, return error code in
12     uint32_t return_error2; /* Write failed, return error code i
13         /* buffer. Used when sending a reply to a dead process t
14         /* we are also waiting on */
15     wait_queue_head_t wait;
16     struct binder_stats stats;
17 };
18
19 static struct binder_thread *binder_get_thread(struct binder_proc
20 {
21     struct binder_thread *thread = NULL;
22     struct rb_node *parent = NULL;
23     struct rb_node **p = &proc->threads.rb_node;
24
25     // proc 中的 thread 按 pid 保存在 proc 的红黑树中
26     while (*p) {
27         parent = *p;
28         thread = rb_entry(parent, struct binder_thread, rb_node)
29
30         if (current->pid < thread->pid)
31             p = &(*p)->rb_left;
32         else if (current->pid > thread->pid)
33             p = &(*p)->rb_right;
34         else
35             break;
36     }
37     // 如果之前没创建过, 就 new 一个新的出来
38     if (*p == NULL) {
39         thread = kzalloc(sizeof(*thread), GFP_KERNEL);
40         if (thread == NULL)
41             return NULL;
42         binder_stats_created(BINDER_STAT_THREAD);
43         // 设置线程所在的 proc 和 pid
44         // 这里的 pid 相当于是 tid
45         thread->proc = proc;
46         thread->pid = current->pid;
47         init_waitqueue_head(&thread->wait);
48         INIT_LIST_HEAD(&thread->todo);
49         // 保存到 proc 中
50         rb_link_node(&thread->rb_node, parent, p);

```

```

51         rb_insert_color(&thread->rb_node, &proc->threads);
52         // 注意初始化状态
53         thread->looper |= BINDER_LOOPER_STATE_NEED_RETURN;
54         thread->return_error = BR_OK;
55         thread->return_error2 = BR_OK;
56     }
57     return thread;
58 }
59

```

这里要说下 `current` 这个全局变量。这个东西代表当前运行的线程的一些结构（其实结构应该是 `task_struct`）。具体的可以去看 linux 内核相关的书。反正用这个东西可以取得到当前运行的线程的一些信息就对了（好像是通过取堆栈最顶的东西）。

然后我们看下 `binder_thread_read` 这里：

```

1  enum {
2      BINDER_LOOPER_STATE_REGISTERED = 0x01,
3      BINDER_LOOPER_STATE_ENTERED   = 0x02,
4      BINDER_LOOPER_STATE_EXITED    = 0x04,
5      BINDER_LOOPER_STATE_INVALID    = 0x08,
6      BINDER_LOOPER_STATE_WAITING    = 0x10,
7      BINDER_LOOPER_STATE_NEED_RETURN = 0x20
8  };
9
10 // Bn 会阻塞等待在这等待 Bp 的请求的到来
11 static int binder_thread_read(struct binder_proc *proc,
12                             struct binder_thread *thread,
13                             void __user *buffer, int size,
14                             signed long *consumed, int non_block)
15 {
16     void __user *ptr = buffer + *consumed;
17     void __user *end = buffer + size;
18
19     int ret = 0;
20     int wait_for_proc_work;
21
22     if (*consumed == 0) {
23         if (put_user(BR_NOOP, (uint32_t __user *)ptr))
24             return -EFAULT;
25         ptr += sizeof(uint32_t);
26     }
27
28     retry:

```



```

29     // transaction_stack == NULL 代表是第一次的 read (Bn 的阻塞 read
30     // Bn 的阻塞等待的 read todo list 也是空的
31     // 所以 Bn 的阻塞 read 这里的 wait_for_proc_work 是 true
32     wait_for_proc_work = thread->transaction_stack == NULL &&
33         list_empty(&thread->todo);
34
35     if (thread->return_error != BR_OK && ptr < end) {
36         if (thread->return_error2 != BR_OK) {
37             if (put_user(thread->return_error2, (uint32_t __user *)p
38                 return -EFAULT;
39             ptr += sizeof(uint32_t);
40             if (ptr == end)
41                 goto done;
42             thread->return_error2 = BR_OK;
43         }
44         if (put_user(thread->return_error, (uint32_t __user *)p
45             return -EFAULT;
46         ptr += sizeof(uint32_t);
47         thread->return_error = BR_OK;
48         goto done;
49     }
50
51     // 前面说了这个 looper 是当前线程的状态,
52     // 注意这里设置为 WAITING 了, 表示正在等待
53     thread->looper |= BINDER_LOOPER_STATE_WAITING;
54     // Bn read 这里是 true, 表示本进程空闲的进程数加1
55     if (wait_for_proc_work)
56         proc->ready_threads++;
57     mutex_unlock(&binder_lock);
58     if (wait_for_proc_work) {
59         // 这里检测 thread 是不是有下面这2个标志, 这2个标志后面会说到。
60         // 还有注意前面设置那个 WAITTING 的是用 | 设置的, 然后这里检测
61         // 然后看看这几个标志定义的值, 会发现这里微妙的用法
62         if (!(thread->looper & (BINDER_LOOPER_STATE_REGISTERED
63             BINDER_LOOPER_STATE_ENTERED))) {
64             binder_user_error("binder: %d:%d ERROR: Thread wait
65                 "for process work before calling BC_REGISTER_"
66                 "LOOPER or BC_ENTER_LOOPER (state %x)\n",
67                 proc->pid, thread->pid, thread->looper);
68             wait_event_interruptible(binder_user_error_wait,
69                 binder_stop_on_user_error < 2);
70         }
71         binder_set_nice(proc->default_priority);
72         if (non_block) {
73             if (!binder_has_proc_work(proc, thread))
74                 ret = -EAGAIN;
75         } else
76             // 这里就阻塞在这里, 等 thread 的 todo list 不为空 (Bp
77         ret = wait_event_interruptible_exclusive(proc->wait

```

```

78     } else {
79         if (non_block) {
80             if (!binder_has_thread_work(thread))
81                 ret = -EAGAIN;
82         } else
83             ret = wait_event_interruptible(thread->wait, binder
84     }
85     mutex_lock(&binder_lock);
86     // 如果这个等待的线程被唤醒了（有 Bp 请求来了），
87     // 把这个进程空闲的线程数减1，
88     // 因为这个线程后面马上就要到用户空间去执行相关业务的函数了。
89     if (wait_for_proc_work)
90         proc->ready_threads--;
91     // 把线程的 WAITTING 标志去掉
92     thread->looper &= ~BINDER_LOOPER_STATE_WAITING;
93
94     // wait 出错的话，返回错误值
95     if (ret)
96         return ret;
97
98     ...
99
100    done:
101
102        *consumed = ptr - buffer;
103        // 最后这里 requested_threads 表示发出请求要启动的线程数，
104        // ready_threads 表示空闲的线程数。
105        // 如果这2个加起来 == 0 就表示当前进程（服务进程）没有空闲的线程来处
106        // 并且还没请求去启动线程，所以需要启动一个新的线程来等待 Bp 的请求。
107        // requested_threads_started 表示本进程应请求启动的线程数，
108        // 这个不能超过 max_threads 设置的上限。
109        if (proc->requested_threads + proc->ready_threads == 0 &&
110            proc->requested_threads_started < proc->max_threads &&
111            (thread->looper & (BINDER_LOOPER_STATE_REGISTERED |
112                BINDER_LOOPER_STATE_ENTERED))) /* the user-space code f
113            /*spawn a new thread if we leave this out */) {
114            // 这里发 BR_SPAWN_LOOPER 到用户去创建新线程去了
115            // 然后把请求启动的线程数加1
116            proc->requested_threads++;
117            binder_debug(BINDER_DEBUG_THREADS,
118                "binder: %d:%d BR_SPAWN_LOOPER\n",
119                proc->pid, thread->pid);
120            if (put_user(BR_SPAWN_LOOPER, (uint32_t __user *)buffer
121                return -EFAULT;
122        }
123        return 0;
124    }
125

```

这里 binder 驱动其实用了个很简单的办法来管理线程。就是假设 Bn 端有一个线程 wait 在 read 那了，就相当于多了一个空闲线程（能够处理 Bp 的请求）。上面分析过了，BinderService 一开始就整个 2 个线程 wait read 那了。然后如果来一个 Bp 请求，`binder_transaction` 那里找到目标进程，然后把请求放到目标进程的 todo list 中：

```

1  static void binder_transaction(struct binder_proc *proc,
2                                struct binder_thread *thread,
3                                struct binder_transaction_data *tr, int reply
4  {
5      struct binder_transaction *t;
6      struct binder_work *tcomplete;
7      size_t *offp, *off_end;
8      struct binder_proc *target_proc;
9      struct binder_thread *target_thread = NULL;
10     struct binder_node *target_node = NULL;
11     struct list_head *target_list;
12     wait_queue_head_t *target_wait;
13     struct binder_transaction *in_reply_to = NULL;
14     struct binder_transaction_log_entry *e;
15     uint32_t return_error;
16
17     ... ..
18
19     // 第一次 Bp 发请求给 Bn target_thread 是 null, 走的是下面那个
20     if (target_thread) {
21         e->to_thread = target_thread->pid;
22         target_list = &target_thread->todo;
23         target_wait = &target_thread->wait;
24     } else {
25         // todo list 是 target_proc 的
26         target_list = &target_proc->todo;
27         target_wait = &target_proc->wait;
28     }
29
30     ... ..
31
32     t->work.type = BINDER_WORK_TRANSACTION;
33     // 把请求打包成工作 (work), 加入到 todo list 中
34     list_add_tail(&t->work.entry, target_list);
35     tcomplete->type = BINDER_WORK_TRANSACTION_COMPLETE;
36     list_add_tail(&tcomplete->entry, &thread->todo);
37     // 唤醒等待队列
38     if (target_wait)
39         wake_up_interruptible(target_wait);

```

```

40         return;
41
42     ... ..
43
44 }
45

```

`binder_transtion` 就是消耗 Bn 工作线程的地方了。通信模型篇里分析过了，Bp 第一次发请求给 Bn，是没 `target_thread` 的，所以请求就加入到 `target_proc` 的 todo list 中了。然后唤醒 Bn 在 read 那休眠的线程。

这里说下 wait queue 的小知识，`binder_thread_read` 使用的 `wait_event_interruptible_exclusive` 第二参数是一个检测条件，这里是检测 proc 的 todo list 是否为空，这个会是不停的检测的（应该不怎么耗 cpu 吧），一旦条件为 true 就唤醒继续执行。所以 `binder_transation` 应该没那个唤醒的操作也可以，不过也还是保险一点好。

然后你也发现上面 wait proc 用的是

`wait_event_interruptible_exclusive`，下面那个 wait thread 用的是 `wait_event_interruptible`。这2个有啥区别咧。我查到的是说 `wait_event_interruptible_exclusive` 在检测唤醒条件的时候是一个互斥过程，是不是说如果有多个线程 wait 的时候只检测一个线程的条件，因为之前的例子，Bn 那已经有2个线程 wait proc 那了。下面那个不带互斥，下面那个由于是 wait 本线程的，所以只会有一个线程 wait。还是说 wait queue 一次只会唤醒一个线程而已。我加的打印发现是唤醒的是最后那个等待的线程。由于比较难写 kernel 的小例子，我这里就不验证这个等待队列的用法了。反正由于种种原因这里一次就只能唤醒一个等待线程。

然后说下 `transaction_stack` 这个东西，看上面知道第一次

`transaction_stack` 是 NULL，也就是 Bn 等待 Bp 来请求的时候是 NULL，然后之后 Bp 通过 `binder_transaction` 把 work 加到 Bn 的 todo list，`transaction_stack` 就保存了 Bp 的 thread，然后 Bn `binder_thread_read` 之后，Bn proc 的 `transaction_stack` Bn 的 thread。然后 `binder_transaction` 一开始判断 `reply == true` 从

`transaction_stack` 去取 target。这么搞是为了保证，Bp 发送请求到 Bn 的线程处理后，Bn 能返回到正确的 Bp 线程，就是保证返回值能送到发送请求的那个线程处理。这个通信原理有具体分析代码，这里再点一下。这里结合后面说一个进程跑多个服务，binder 的多线程机制并不会导致混乱。

那这样就差把 binder 驱动里面做的事说完了。每当一个 Bn 的线程在 `binder_thread_read` 等待，`proc->ready_threads` 就会 +1，如果 todo list 里接到请求，最后那个等待的线程被唤醒，`proc->ready_threads` -1，然后唤醒的线程去执行 IPC 请求业务函数，在最后判断是否还有空闲的线程（已经在等待的（`ready_threads`）+发送启动请求的（`requested_threads`）是否为0）。如果没有 Bn 没空闲的线程，并且已经启动的线程（`requested_threads_started`）没超过限制（`max_threas`）就发一个 `BR_SPAWN_LOOPER` 给用户空间去创建线程。

我们来看看用户空间怎么处理 `BR_SPAWN_LOOPER` 的：

```

1  status_t IPCThreadState::executeCommand(int32_t cmd)
2  {
3      BBinder* obj;
4      RefBase::weakref_type* refs;
5      status_t result = NO_ERROR;
6
7      switch (cmd) {
8          ...
9      case BR_SPAWN_LOOPER:
10         // 调用的是 ProcessState 的 spawnPooledThread, 参数是 false
11         mProcess->spawnPooledThread(false);
12         break;
13
14     default:
15         printf("*** BAD COMMAND %d received from Binder driver\n", cmd);
16         result = UNKNOWN_ERROR;
17         break;
18     }
19
20     if (result != NO_ERROR) {
21         mLastError = result;
22     }
23
24     return result;
25 }
```

结果就是 kernel 帮我们自动调用 ProcessState 的 PooledThread 而已。这里的参数是 false 了，就是说 isMain 是 false 了。这个 isMain 前面说没啥用的，不过还是有点小区别。回去看 joinThreadPool (spwanPooledThread 最后还是调用了 joinThreadPool) 那：

```
1 mOut.writeInt32(isMain ? BC_ENTER_LOOPER : BC_REGISTER_LOOPER);
2
```

isMain 的对 binder 发的是 `BC_ENTER_LOOPER`，false 的发的是 `BC_REGISTER_LOOPER`。我去看下这2个有啥区别不：

```
1 int binder_thread_write(struct binder_proc *proc, struct binder_
2 void __user *buffer, int size, signed long *consumed
3 {
4     uint32_t cmd;
5     void __user *ptr = buffer + *consumed;
6     void __user *end = buffer + size;
7
8     while (ptr < end && thread->return_error == BR_OK) {
9         if (get_user(cmd, (uint32_t __user *)ptr))
10            return -EFAULT;
11        ptr += sizeof(uint32_t);
12        if (_IOC_NR(cmd) < ARRAY_SIZE(binder_stats.bc)) {
13            binder_stats.bc[_IOC_NR(cmd)]++;
14            proc->stats.bc[_IOC_NR(cmd)]++;
15            thread->stats.bc[_IOC_NR(cmd)]++;
16        }
17        switch (cmd) {
18            ...
19            case BC_REGISTER_LOOPER:
20                // 不允许已经 ENTER_LOOPER 的线程再 REGISTER_LOOPER
21                binder_debug(BINDER_DEBUG_THREADS,
22                    "binder: %d:%d BC_REGISTER_LOOPER\n",
23                    proc->pid, thread->pid);
24                if (thread->looper & BINDER_LOOPER_STATE_ENTERED) {
25                    thread->looper |= BINDER_LOOPER_STATE_INVALID;
26                    binder_user_error("binder: %d:%d ERROR:"
27                        " BC_REGISTER_LOOPER called "
28                        "after BC_ENTER_LOOPER\n",
29                        proc->pid, thread->pid);
30                }
31                // 没发起请求的也不允许 REGISTER_LOOPER
```

```

31         } else if (proc->requested_threads == 0) {
32             thread->looper |= BINDER_LOOPER_STATE_INVALID;
33             binder_user_error("binder: %d:%d ERROR:"
34                             " BC_REGISTER_LOOPER called "
35                             "without request\n",
36                             proc->pid, thread->pid);
37         } else {
38             // 请求线程数 -1
39             proc->requested_threads--;
40             // 已经启动的线程数 +1
41             proc->requested_threads_started++;
42         }
43         // 设置下线程状态
44         thread->looper |= BINDER_LOOPER_STATE_REGISTERED;
45         break;
46     case BC_ENTER_LOOPER:
47         // 不允许 REGISTER_LOOPER 的线程再 ENTER_LOOPER
48         binder_debug(BINDER_DEBUG_THREADS,
49                     "binder: %d:%d BC_ENTER_LOOPER\n",
50                     proc->pid, thread->pid);
51         if (thread->looper & BINDER_LOOPER_STATE_REGISTERED)
52             thread->looper |= BINDER_LOOPER_STATE_INVALID;
53         binder_user_error("binder: %d:%d ERROR:"
54                             " BC_ENTER_LOOPER called after "
55                             "BC_REGISTER_LOOPER\n",
56                             proc->pid, thread->pid);
57     }
58     // 设置下线程状态
59     thread->looper |= BINDER_LOOPER_STATE_ENTERED;
60     break;
61     case BC_EXIT_LOOPER:
62         // 这个退出只是把状态设置了下
63         binder_debug(BINDER_DEBUG_THREADS,
64                     "binder: %d:%d BC_EXIT_LOOPER\n",
65                     proc->pid, thread->pid);
66         thread->looper |= BINDER_LOOPER_STATE_EXITED;
67         break;
68     ... ..
69     ... ..
70     ... ..
71     default:
72         printk(KERN_ERR "binder: %d:%d unknown command %d\n"
73               proc->pid, thread->pid, cmd);
74         return -EINVAL;
75     }
76     *consumed = ptr - buffer;
77 }
78 return 0;
79 }

```


isMain 的除去是否会超时自动退出外（前面说了，这个机制目前是没用的），就是上面那些判断了，isMain 是 true 是手动创建的，所以没什么限制。false 则是 binder 驱动根据当前可用线程数的情况自动请求创建的，所以确定 binder 驱动确实请求了才允许创建（requested_threads 不为 0）。还有

`BC_ENTER_LOOPER` 的是不算在 `requested_threads_started` 里面的，所以手动启动的线程理论上可以无限个（不过我看到 SS 除了一开始手动调用一次 `startThreadPool`，然后把当前线程 `joinThreadPool` 之外，就再也没手动启动过线程，都是交由驱动管理）。

然后说下既然有创建线程，那什么时候退出呢。前面说了超时暂时没用，`joinThreadPool` 那里 io 错误也会导致退出，这些异常的不算。正常目前好像没那个地方会退出，binder 启动的线程。是的，没错，目前 binder 的线程一旦启动就不会退出的，直到达到上限为止，只要 binder 一发现当前服务进程没空闲线程了就会 spawn 一个出来。但是由于服务线程一旦执行完 IPC 调用就会变成空闲的，所以只要同一个时候没很多 Bp 请求过来，不会创建太多线程的。而且就算创建了不少线程，这些线程只是休眠而已，不乎占用 cpu 资源，所以没啥太大关系，可能后续的 binder 版本会考虑一段时间这个线程没有执行请求就把这个线程退出吧。

然后 binder 自动创建的线程数可以设置上限的。说起这个 binder 有条命令：

`BINDER_SET_MAX_THREADS`：

```
1  // binder_thread_write:
2
3  // 就是把进程的 max_threads 设置了一下
4  case BINDER_SET_MAX_THREADS:
5      if (copy_from_user(&proc->max_threads, ubuf, sizeof(proc
6          ret = -EINVAL;
7          goto err;
8      }
9      break;
10
```

说到这个，我们来看看默认设置最大线程数是多少吧：


```

1  static int open_driver()
2  {
3      int fd = open("/dev/binder", O_RDWR);
4      if (fd >= 0) {
5          fcntl(fd, F_SETFD, FD_CLOEXEC);
6          int vers;
7          status_t result = ioctl(fd, BINDER_VERSION, &vers);
8          if (result == -1) {
9              ALOGE("Binder ioctl to obtain version failed: %s", s
10                 close(fd);
11                 fd = -1;
12             }
13             if (result != 0 || vers != BINDER_CURRENT_PROTOCOL_VERSION) {
14                 ALOGE("Binder driver protocol does not match user space");
15                 close(fd);
16                 fd = -1;
17             }
18             // 最大线程数为 15
19             size_t maxThreads = 15;
20             result = ioctl(fd, BINDER_SET_MAX_THREADS, &maxThreads);
21             if (result == -1) {
22                 ALOGE("Binder ioctl to set max threads failed: %s",
23                     );
24             } else {
25                 ALOGW("Opening '/dev/binder' failed: %s\n", strerror(errno));
26             }
27             return fd;
28         }
29     }

```

ProcessState 的构造函数会调用 `open_driver()`，binder 默认允许最大的线程数为 15 个。一般来说继承 BindService 的默认就是用这个数量了，但是也有特殊的。SurfaceFlinger (SF) 没 BindService，手动写的：

```

1  // main_surfaceflinger.cpp =====
2
3  int main(int argc, char** argv) {
4      // 最大线程数为 4
5      // When SF is launched in its own process, limit the number
6      // binder threads to 4.
7      ProcessState::self()->setThreadPoolMaxThreadCount(4);
8
9      // 还有一个主线程 (isMain)
10     // start the thread pool
11     sp<ProcessState> ps(ProcessState::self());

```

```

12     ps->startThreadPool();
13
14     // instantiate surfaceflinger
15     sp<SurfaceFlinger> flinger = new SurfaceFlinger();
16
17     #if defined(HAVE_PTHREADS)
18         setpriority(PRIO_PROCESS, 0, PRIORITY_URGENT_DISPLAY);
19     #endif
20     set_sched_policy(0, SP_FOREGROUND);
21
22     // initialize before clients can connect
23     flinger->init();
24
25     // publish surface flinger
26     sp<IServiceManager> sm(defaultServiceManager());
27     sm->addService(String16(SurfaceFlinger::getServiceName()), f
28
29     // run in this thread
30     flinger->run();
31
32     return 0;
33 }
34
35 // ProcessState.cpp =====
36
37 status_t ProcessState::setThreadPoolMaxThreadCount(size_t maxThr
38     status_t result = NO_ERROR;
39     if (ioctl(mDriverFD, BINDER_SET_MAX_THREADS, &maxThreads) ==
40         result = -errno;
41         ALOGE("Binder ioctl to set max threads failed: %s", stre
42     }
43     return result;
44 }
45

```

SF 就只设置了4个可以自动创建的线程，还有一个主线程（应该还有一个吧，当前那个线程跑哪去了，以后分析 SF 再说吧）。这里注意一点，如果要手动写的话，主要设置要放到 IPCThreadState->joinThreadPool 的前面，因为调用 joinThreadPool 当前线程就加入到 binder 的主线程中去了，阻塞循环就开始了，除非线程退出了，否则后面的代码执行不到的。

前面讨论了线程过多的情况，如果 Bn 中的线程不够的话，会怎么样咧。就是当前服务进程中没空闲的线程了（都在执行业务函数），答案是 Bp 端等待。前面不是说 proc 有 todo list 么（thread 也有的），既然是 list 就可以保存多个工作

请求 (work)，然后一次处理一个，当没空闲线程马上处理的时候，把请求加到 todo list 后面，等线程执行完当前的业务函数，回到

`binder_thread_read` 那，准备等待的时候，发现 todo list 中不空，就马上取下一个 work 然后执行，就不会进入休眠，直到 todo list 为空。所以如果服务进程空闲线程不足，然后这个时候 Bp 请求又多的话，会导致 IPC 调用非常慢（要排队等，哥又想起排队抢火车票了）。所以要根据自身业务的情况去设置 binder 自动创建线程的上限，不要设得太小。顺带贴下线程取工作请求：

```

1  // binder_thread_read:
2
3      while (1) {
4          uint32_t cmd;
5          struct binder_transaction_data tr;
6          struct binder_work *w;
7          struct binder_transaction *t = NULL;
8
9          // 去 todo list 中取工作请求
10         if (!list_empty(&thread->todo))
11             w = list_first_entry(&thread->todo, struct binder_wc
12         else if (!list_empty(&proc->todo) && wait_for_proc_work)
13             w = list_first_entry(&proc->todo, struct binder_work
14         else {
15             if (ptr - buffer == 4 && !(thread->looper & BINDER_L
16                 goto retry;
17             break;
18         }
19
20         if (end - ptr < sizeof(tr) + 4)
21             break;
22
23         ... ..
24
25     }
26

```

所以那些 SS 里面很多函数都加了锁，因为有多线程的支持，可能同一个时间在执行不同客户端的业务函数。有可能 Bp1 的执行的函数是读一个变量，而 Bp2 执行的函数正好要写这个变量，这个时候需要互斥操作，所以很多操作都带锁了。但是注意不要什么操作都带锁，因为互斥锁会影响执行效率的，对于可以重入的函数不需要加锁（例如函数里面全是局部变量）。

还有要注意 SS 协同完成一些任务的时候的问题，例如一个 Proc A IPC 调用 Proc B 的某个函数，这个函数又跑去调用 AM 里面的某个函数，然后 AM 这个函数正好又去 IPC 调用 Proc B 中的某个函数。如果这些 IPC 函数都加了锁的话，就会死锁。关于这个例子可以去看 Binder 对象传递的普通服务篇。

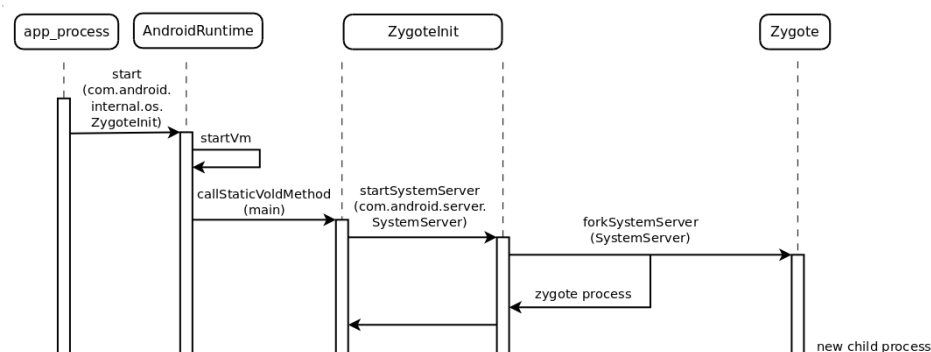
java 层的 SS

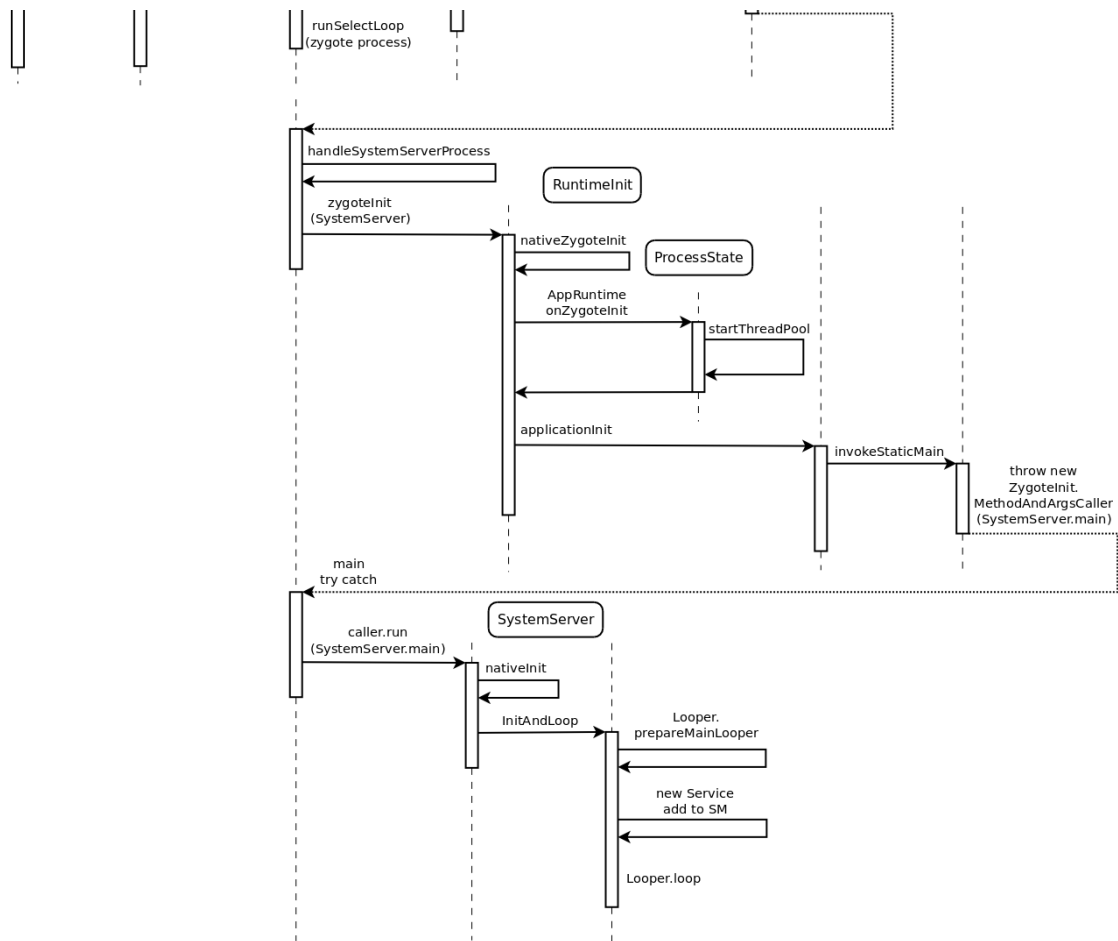
前面差不多 binder 多线程的支持说完了，也看了一些 native SS 的例子。最后来看下 java 层的，也是我们应用开发中接触最多的那一票系统服务（AM、WM、PM 等）。前面说这个有点麻烦，主要是它还得启动 java 虚拟机。我们来一点点看吧，顺带把 SS 启动过程也走一下。

首先说了 init.rc 里有启动 zygote（不是 SS 么，怎么变成 zygote 了，别急，往下看）：

```
service zygote /system/bin/app_process -Xzygote /system/bin
    class main
    socket zygote stream 660 root system
    onrestart write /sys/android_power/request_state wake
    onrestart write /sys/power/state on
    onrestart restart media
    onrestart restart netd
```

注意那个 `--start-system-server` 和 `--zygote` 的参数。其实是启动 `app_process` 这个 native 程序。我们来先上一张序列图，还挺麻烦的说（因为是通过启动一个程序（zygote），然后再 fork 出另外一个（SS））：





我来看看下 app_process 的 main 函数：

```

1  // app_main.cpp =====
2
3  int main(int argc, char* const argv[])
4  {
5
6  ... ..
7
8      AppRuntime runtime;
9      const char* argv0 = argv[0];
10
11     // Process command line arguments
12     // ignore argv[0]
13     argc--;
14     argv++;
15
16     // Everything up to '--' or first non '-' arg goes to the vm
17
18     int i = runtime.addVmArguments(argc, argv);
19

```

```
20 // init.rc 传过来的参数 --zygote 和 --start-system-server
21 // Parse runtime arguments. Stop at first unrecognized opti
22 bool zygote = false;
23 bool startSystemServer = false;
24 bool application = false;
25 const char* parentDir = NULL;
26 const char* niceName = NULL;
27 const char* className = NULL;
28 while (i < argc) {
29     const char* arg = argv[i++];
30     if (!parentDir) {
31         parentDir = arg;
32     } else if (strcmp(arg, "--zygote") == 0) {
33         zygote = true;
34         niceName = "zygote";
35     } else if (strcmp(arg, "--start-system-server") == 0) {
36         startSystemServer = true;
37     } else if (strcmp(arg, "--application") == 0) {
38         application = true;
39     } else if (strncmp(arg, "--nice-name=", 12) == 0) {
40         niceName = arg + 12;
41     } else {
42         className = arg;
43         break;
44     }
45 }
46
47 if (niceName && *niceName) {
48     setArgv0(argv0, niceName);
49     set_process_name(niceName);
50 }
51
52 runtime.mParentDir = parentDir;
53
54 if (zygote) {
55     // 那就是走这里的
56     runtime.start("com.android.internal.os.ZygoteInit",
57                 startSystemServer ? "start-system-server" : "");
58 } else if (className) {
59     // Remainder of args get passed to startup class main()
60     runtime.mClassName = className;
61     runtime.mArgC = argc - i;
62     runtime.mArgV = argv + i;
63     runtime.start("com.android.internal.os.RuntimeInit",
64                 application ? "application" : "tool");
65 } else {
66     fprintf(stderr, "Error: no class name or --zygote suppli
67     app_usage();
68     LOG_ALWAYS_FATAL("app_process: no class name or --zygote
```

```

69         return 10;
70     }
71 }
72

```

init.rc 传过来的参数来看，跑去调用 AppRuntime 的 start 去了。AppRuntime 在 `app_main.cpp` 中：

```

1  class AppRuntime : public AndroidRuntime
2  {
3  public:
4      AppRuntime()
5          : mParentDir(NULL)
6            , mClassName(NULL)
7            , mClass(NULL)
8            , mArgC(0)
9            , mArgV(NULL)
10     {
11     }
12
13     ... ..
14
15     // 注意这个函数
16     virtual void onZygoteInit()
17     {
18         // Re-enable tracing now that we're no longer in Zygote.
19         atrace_set_tracing_enabled(true);
20
21         // 这里初始化了 ProcessState，并调用了 startThreadPool
22         sp<ProcessState> proc = ProcessState::self();
23         ALOGV("App process: starting thread pool.\n");
24         proc->startThreadPool();
25     }
26
27     ... ..
28
29     const char* mParentDir;
30     const char* mClassName;
31     jclass mClass;
32     int mArgC;
33     const char* const* mArgV;
34 };
35

```

AppRuntime 继承 AndroidRuntime，我们得去看父类里面的 start：

```

1  // 前面传过来的 className 是: com.android.internal.os.ZygoteInit
2  // options 是: start-system-server
3  /*
4   * Start the Android runtime. This involves starting the virtu
5   * and calling the "static void main(String[] args)" method in
6   * named by "className".
7   *
8   * Passes the main function two arguments, the class name and t
9   * options string.
10  */
11 void AndroidRuntime::start(const char* className, const char* c
12 {
13     ALOGD("\n>>>>> AndroidRuntime START %s <<<<<\n",
14           className != NULL ? className : "(unknown)");
15
16     /*
17      * 'startSystemServer == true' means runtime is obsolete an
18      * init.rc anymore, so we print out the boot start event he
19      */
20     if (strcmp(options, "start-system-server") == 0) {
21         /* track our progress through the boot sequence */
22         const int LOG_BOOT_PROGRESS_START = 3000;
23         LOG_EVENT_LONG(LOG_BOOT_PROGRESS_START,
24                       ns2ms(systemTime(SYSTEM_TIME_MONOTONIC)))
25     }
26
27     const char* rootDir = getenv("ANDROID_ROOT");
28     if (rootDir == NULL) {
29         rootDir = "/system";
30         if (!hasDir("/system")) {
31             LOG_FATAL("No root directory specified, and /androi
32             return;
33         }
34         setenv("ANDROID_ROOT", rootDir, 1);
35     }
36
37     //const char* kernelHack = getenv("LD_ASSUME_KERNEL");
38     //ALOGD("Found LD_ASSUME_KERNEL='%s'\n", kernelHack);
39
40     // 启动 VM, 这里不管这个
41     /* start the virtual machine */
42     JniInvocation jni_invocation;
43     jni_invocation.Init(NULL);
44     JNIEnv* env;
45     if (startVm(&mJavaVM, &env) != 0) {
46         return;
47     }
48     // 之前的子类 AppRuntime 有重载, 不过这里启动 zygote 这个函数直接让

```



```
49     onVmCreated(env);
50
51     // 这个 startReg 还记得不,前面说 android 线程的时候
52     // 改变 libutils Threads 里面的线程函数指针就是在这个函数里面
53     /*
54      * Register android functions.
55      */
56     if (startReg(env) < 0) {
57         ALOGE("Unable to register all android natives\n");
58         return;
59     }
60
61     /*
62      * We want to call main() with a String array with argument
63      * At present we have two arguments, the class name and an
64      * Create an array to hold them.
65      */
66     jclass stringClass;
67     jobjectArray strArray;
68     jstring classNameStr;
69     jstring optionsStr;
70
71     // 下面这一大串就是取 className 的 main 函数,然后执行而已
72     // className 是 com.android.internal.os.ZygoteInit
73     stringClass = env->FindClass("java/lang/String");
74     assert(stringClass != NULL);
75     strArray = env->NewObjectArray(2, stringClass, NULL);
76     assert(strArray != NULL);
77     classNameStr = env->NewStringUTF(className);
78     assert(classNameStr != NULL);
79     env->SetObjectArrayElement(strArray, 0, classNameStr);
80     optionsStr = env->NewStringUTF(options);
81     // 把 start-system-service 参数打包成 java 函数的参数
82     env->SetObjectArrayElement(strArray, 1, optionsStr);
83
84     /*
85      * Start VM. This thread becomes the main thread of the VM
86      * not return until the VM exits.
87      */
88     char* slashClassName = toSlashClassName(className);
89     jclass startClass = env->FindClass(slashClassName);
90     if (startClass == NULL) {
91         ALOGE("JavaVM unable to locate class '%s'\n", slashClass
92             /* keep going */
93     } else {
94         // 取 className 的 main 函数
95         jmethodID startMeth = env->GetStaticMethodID(startClass
96             "([Ljava/lang/String;)V");
97         if (startMeth == NULL) {
```

```

 98         ALOGE("JavaVM unable to find main() in '%s'\n", cla
 99         /* keep going */
100     } else {
101         // 通过反射调用 main 函数,
102         // 注意把前面的 start-system-service 当作参数传过去了
103         env->CallStaticVoidMethod(startClass, startMeth, st
104
105     #if 0
106         if (env->ExceptionCheck())
107             threadExitUncaughtException(env);
108     #endif
109     }
110 }
111 free(slashClassName);
112
113 ALOGD("Shutting down VM\n");
114 if (mJavaVM->DetachCurrentThread() != JNI_OK)
115     ALOGW("Warning: unable to detach main thread\n");
116 if (mJavaVM->DestroyJavaVM() != 0)
117     ALOGW("Warning: VM did not shut down cleanly\n");
118 }
119

```

这里忽略了 JVM 的启动过程（和 binder 关系不大），然后后面主要就是启动 java 层里面的东西了。这里是去执行了 com.android.internal.os.ZygoteInit 的 main 函数（下面欢迎进入 java 世界）：

```

 1  public static void main(String argv[]) {
 2      try {
 3          // Start profiling the zygote initialization.
 4          SamplingProfilerIntegration.start();
 5
 6          // 打开 zygote 通信用的 socket
 7          registerZygoteSocket();
 8          // Finish profiling the zygote initialization.
 9          boolean isFirstBoot = false;
10          //if first time booting and zygote restart need preload
11          if(Process.myPid() > 300 || SystemProperties.getBoolean(
12              EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
13                  SystemClock.uptimeMillis());
14              preload();
15              isFirstBoot = true;
16              EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
17                  SystemClock.uptimeMillis());
18          }
19          // Finish profiling the zygote initialization.

```

```
20         SamplingProfilerIntegration.writeZygoteSnapshot();
21
22         // Do an initial gc to clean up after startup
23         gc();
24
25         // Disable tracing so that forked processes do not inher
26         // Zygote.
27         Trace.setTracingEnabled(false);
28
29         // If requested, start system server directly from Zygote
30         if (argv.length != 2) {
31             throw new RuntimeException(argv[0] + USAGE_STRING);
32         }
33
34         // 那个参数传了半天终于有用了, 这个函数就是去启动 SS 的进程
35         if (argv[1].equals("start-system-server")) {
36             startSystemServer();
37         } else if (!argv[1].equals("")) {
38             throw new RuntimeException(argv[0] + USAGE_STRING);
39         }
40
41         Log.i(TAG, "Accepting command socket connections");
42         if(!isFirstBootling){
43             EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_START,
44                 SystemClock.uptimeMillis());
45             preload();
46             EventLog.writeEvent(LOG_BOOT_PROGRESS_PRELOAD_END,
47                 SystemClock.uptimeMillis());
48             gc();
49         }
50         // 这个就是 zygote 循环阻塞等待请求的函数了
51         runSelectLoop();
52
53         // 上面那个函数退出就说明 zygote 退出了, 关闭之前打开的 socket
54         closeServerSocket();
55     } catch (MethodAndArgsCaller caller) {
56         caller.run();
57     } catch (RuntimeException ex) {
58         Log.e(TAG, "Zygote died with exception", ex);
59         closeServerSocket();
60         throw ex;
61     }
62 }
63
```

我们这里主要看 startSystemServer 这个函数, 其它的是 zygote 的, zygote 的话, 我有一篇工作小笔记 (换系统字体那个) 有说到, 这里不多说。

```

1  /*
2   * Prepare the arguments and fork for the system server process.
3   */
4  private static boolean startSystemServer()
5      throws MethodAndArgsCaller, RuntimeException {
6      long capabilities = posixCapabilitiesAsBits(
7          OsConstants.CAP_KILL,
8          OsConstants.CAP_NET_ADMIN,
9          OsConstants.CAP_NET_BIND_SERVICE,
10         OsConstants.CAP_NET_BROADCAST,
11         OsConstants.CAP_NET_RAW,
12         OsConstants.CAP_SYS_MODULE,
13         OsConstants.CAP_SYS_NICE,
14         OsConstants.CAP_SYS_RESOURCE,
15         OsConstants.CAP_SYS_TIME,
16         OsConstants.CAP_SYS_TTY_CONFIG
17     );
18     // 下面一堆参数，注意最下面那个类名（我们 SS 终于露面了）
19     /* Hardcoded command line to start the system server */
20     String args[] = {
21         "--setuid=1000",
22         "--setgid=1000",
23         "--setgroups=1001,1002,1003,1004,1005,1006,1007,1008,1009",
24         "--capabilities=" + capabilities + "," + capabilities,
25         "--runtime-init",
26         "--nice-name=system_server",
27         "com.android.server.SystemServer",
28     };
29     ZygoteConnection.Arguments parsedArgs = null;
30
31     int pid;
32
33     try {
34         // 将上面那堆字符串参数解析成一个专门用来 zygote 通信的数据结构
35         // 不过这里启动 SS 根本没和 zygote 通信，这里纯粹借用这个数据而已
36         parsedArgs = new ZygoteConnection.Arguments(args);
37         ZygoteConnection.applyDebuggerSystemProperty(parsedArgs);
38         ZygoteConnection.applyInvokeWithSystemProperty(parsedArgs);
39
40         // fork SS 进程
41         /* Request to fork the system server process */
42         pid = Zygote.forkSystemServer(
43             parsedArgs.uid, parsedArgs.gid,
44             parsedArgs.gids,
45             parsedArgs.debugFlags,
46             null,
47             parsedArgs.permittedCapabilities,
48             parsedArgs.effectiveCapabilities);

```

```

49     } catch (IllegalArgumentException ex) {
50         throw new RuntimeException(ex);
51     }
52
53     // pid == 0 是 fork 出来的 SS 进程, 要去启动 SS 服务了
54     /* For child process */
55     if (pid == 0) {
56         handleSystemServerProcess(parsedArgs);
57     }
58
59     return true;
60 }
61

```

我们去看下 Zygote.forkSystemServer 这个函数：

```

1  /*
2   * Special method to start the system server process. In additic
3   * common actions performed in forkAndSpecialize, the pid of the
4   * process is recorded such that the death of the child process
5   * zygote to exit.
6   *
7   * @param uid the UNIX uid that the new process should setuid()
8   * fork()ing and and before spawning any threads.
9   * @param gid the UNIX gid that the new process should setgid()
10  * fork()ing and and before spawning any threads.
11  * @param gids null-ok; a list of UNIX gids that the new process
12  * setgroups() to after fork and before spawning any threads.
13  * @param debugFlags bit flags that enable debugging features.
14  * @param rlimits null-ok an array of rlimit tuples, with the se
15  * dimension having a length of 3 and representing
16  * (resource, rlim_cur, rlim_max). These are set via the posix
17  * setrlimit(2) call.
18  * @param permittedCapabilities argument for setcap()
19  * @param effectiveCapabilities argument for setcap()
20  *
21  * @return 0 if this is the child, pid of the child
22  * if this is the parent, or -1 on error.
23  */
24  public static int forkSystemServer(int uid, int gid, int[] gids,
25      int[][] rlimits, long permittedCapabilities, long effect
26      preFork();
27      int pid = nativeForkSystemServer(
28          uid, gid, gids, debugFlags, rlimits, permittedCapabi
29      postFork();
30      return pid;
31  }

```

看注释这个函数专门为 SS 写的，不去深究这个函数了，最后应该 jni 调用 linux 的 fork 函数吧。SS 是 zygote 第一个 fork 出来的子进程。然后回到 startSystemService 那里，后面有 pid == 0 的，这个就表示 SS 的进程，然后进到 handleSystemServerProcess：

```
1  // 注意前面打包的那个参数数据结构
2  /*
3   * Finish remaining work for the newly forked system server proc
4   */
5  private static void handleSystemServerProcess(
6      ZygoteConnection.Arguments parsedArgs)
7      throws ZygoteInit.MethodAndArgsCaller {
8
9      // 由于 fork 继承了父进程的 socket ,
10     // SS 不需要这个，所以关掉
11     closeServerSocket();
12
13     // set umask to 0077 so new files and directories will default
14     Libcore.os.umask(S_IRWXG | S_IRWXO);
15
16     // 这个 niceName 是上面那个 system_server
17     // adb shell ps 能看得到 SS 的名字是 system_server
18     if (parsedArgs.niceName != null) {
19         Process.setArgV0(parsedArgs.niceName);
20     }
21
22     // 上面没设置 invokeWith 所以走的下面那个分支
23     if (parsedArgs.invokeWith != null) {
24         WrapperInit.execApplication(parsedArgs.invokeWith,
25                                     parsedArgs.niceName, parsedArgs.targetSdkVersion,
26                                     null, parsedArgs.remainingArgs);
27     } else {
28         // 这个remainingArgs是最后那个 com.android.server.SystemSe
29         /*
30          * Pass the remaining arguments to SystemServer.
31          */
32         RuntimeInit.zygoteInit(parsedArgs.targetSdkVersion, pars
33     }
34
35     /* should never reach here */
36 }
37
```

上面看那个 nineName 设置为 `system_server`，这个在 `adb shell ps` 能看得到的，还能看得出 SS 确实是 `zygote` 的子进程（看 SS 的父进程号）：

```
system 104 1 57768 3332 ffffffff 401977f4 S /system/bin/surfaceflinger
root 105 1 680544 42584 ffffffff 40159790 S zygote
drm 106 1 11652 4656 ffffffff 40143644 S /system/bin/drmserver
media 107 1 37024 8620 ffffffff 40144644 S /system/bin/mediaserver
install 108 1 996 216 c09b9a24 401a9380 S /system/bin/installd
keystore 109 1 3348 980 c08d26cc 4016e644 S /system/bin/keystore
shell 117 1 936 464 c072b31c 401f7380 S /system/bin/sh
root 118 1 4600 204 ffffffff 00019108 R /sbin/adbd
root 381 2 0 0 c060549c 00000000 S flush-179:0
root 428 100 920 320 c050ac24 401f91c8 S sh
root 429 428 1696 952 c05f236c 401a9790 S logcat
system 430 105 771428 45200 ffffffff 4015a7f4 S system_server
u0_a12 482 105 729600 59188 ffffffff 4015a7f4 S com.android.systemui
u0_a6 559 105 694436 23268 ffffffff 4015a7f4 S android.process.media
```

这里终于跑到 SS 进程里面去了，然后我接下去看 `RuntimeInit.zygoteInit`：

```
1  /*
2   * The main function called when started through the zygote proc
3   * could be unified with main(), if the native code in nativeFin
4   * were rationalized with Zygote startup.<p>
5   *
6   * Current recognized args:
7   * <ul>
8   *   <li> <code> [--] <start class name> <args>
9   * </ul>
10  *
11  * @param targetSdkVersion target SDK version
12  * @param argv arg strings
13  */
14  public static final void zygoteInit(int targetSdkVersion, String
15      throws ZygoteInit.MethodAndArgsCaller {
16      if (DEBUG) Slog.d(TAG, "RuntimeInit: Starting application fr
17
18      // 这个不管
19      redirectLogStreams();
20
21      // 这个 commonInit 这里也没什么要特别注意的
22      commonInit();
23      // 下面那个要注意下
24      nativeZygoteInit();
25
26      // 启动 SS 的 java 类的在这个函数里面
27      applicationInit(targetSdkVersion, argv);
28  }
29
```

这个 nativeZygoteInit 在 AndroidRuntime 里面：

```

1  static AndroidRuntime* gCurRuntime = NULL;
2
3  static void com_android_internal_os_RuntimeInit_nativeZygoteInit
4  {
5      gCurRuntime->onZygoteInit();
6  }
7
8  AndroidRuntime::AndroidRuntime() :
9      mExitWithoutCleanup(false)
10 {
11     SkGraphics::Init();
12     // this sets our preference for 16bit images during decode
13     // in case the src is opaque and 24bit
14     SkImageDecoder::SetDeviceConfig(SkBitmap::kRGB_565_Config);
15     // This cache is shared between browser native images, and j
16     // bitmaps. This globalpool is for images that do not either
17     // heap, or are not backed by ashmem. See BitmapFactory.cpp
18     // java call site.
19     SkImageRef_GlobalPool::SetRAMBudget(512 * 1024);
20     // There is also a global font cache, but its budget is spec
21     // see SkFontHost_android.cpp
22
23     // Pre-allocate enough space to hold a fair number of option
24     mOptions.setCapacity(20);
25
26     // gCurRuntime 每个进程一个
27     assert(gCurRuntime == NULL);          // one per process
28     gCurRuntime = this;
29 }
30

```

还记得最开始 AppRuntime 这个之类重载的 onZygoteInit 么，没错就是在这里调用了。这里初始化了 ProcessState（打开 binder 驱动，设置 binder 自动线程最大数目为 15），然后手动启动了一个主线程。

然后我们继续看 applicationInit：

```

1  private static void applicationInit(int targetSdkVersion, String
2      throws ZygoteInit.MethodAndArgsCaller {
3      // If the application calls System.exit(), terminate the proc
4      // immediately without running any shutdown hooks. It is nc
5      // shutdown an Android application gracefully. Among other

```



```

6      // Android runtime shutdown hooks close the Binder driver, w
7      // leftover running threads to crash before the process actu
8      nativeSetExitWithoutCleanup(true);
9
10     // 设置一下虚拟机的一些参数
11     // We want to be fairly aggressive about heap utilization, t
12     // holding on to a lot of memory that isn't needed.
13     VMRuntime.getRuntime().setTargetHeapUtilization(0.75f);
14     VMRuntime.getRuntime().setTargetSdkVersion(targetSdkVersion)
15
16     final Arguments args;
17     try {
18         args = new Arguments(argv);
19     } catch (IllegalArgumentException ex) {
20         Slog.e(TAG, ex.getMessage());
21         // let the process exit
22         return;
23     }
24
25     // 终于快到了
26     // Remaining arguments are passed to the start class's stati
27     invokeStaticMain(args.startClass, args.startArgs);
28 }
29

```

继续看 invokeStaticMain (名字都叫这个份上了)：

```

1  /*
2   * Invokes a static "main(argv[]) method on class "className".
3   * Converts various failing exceptions into RuntimeExceptions, w
4   * the assumption that they will then cause the VM instance to e
5   *
6   * @param className Fully-qualified class name
7   * @param argv Argument vector for main()
8   */
9  private static void invokeStaticMain(String className, String[]
10      throws ZygoteInit.MethodAndArgsCaller {
11      Class<?> cl;
12
13      // com.android.server.SystemServer 这个类名终于有用了
14      try {
15          cl = Class.forName(className);
16      } catch (ClassNotFoundException ex) {
17          throw new RuntimeException(
18              "Missing class when invoking static main " + cla
19              ex);
20      }

```

```

21
22     // 终于去取 main 函数了
23     Method m;
24     try {
25         m = cl.getMethod("main", new Class[] { String[].class });
26     } catch (NoSuchMethodException ex) {
27         throw new RuntimeException(
28             "Missing static main on " + className, ex);
29     } catch (SecurityException ex) {
30         throw new RuntimeException(
31             "Problem getting static main on " + className, e
32     }
33
34     int modifiers = m.getModifiers();
35     if (! (Modifier.isStatic(modifiers) && Modifier.isPublic(mod
36         throw new RuntimeException(
37             "Main method is not public and static on " + cla
38     }
39
40     // 这里搞了个花样，好像是通过抛出异常来清理调用堆栈
41     /*
42     * This throw gets caught in ZygoteInit.main(), which respon
43     * by invoking the exception's run() method. This arrangemen
44     * clears up all the stack frames that were required in sett
45     * up the process.
46     */
47     throw new ZygoteInit.MethodAndArgsCaller(m, argv);
48 }
49

```

这个函数前面都没什么，关键是最后一句，抛出一个异常。看注释说这个异常在 ZygoteInit.main 中有捕获，我们回去看一下：

```

1     public static void main(String argv[]) {
2         try {
3             ... ...
4
5         } catch (MethodAndArgsCaller caller) {
6             // 在这里运行抛过来的函数
7             caller.run();
8         } catch (RuntimeException ex) {
9             Log.e(TAG, "Zygote died with exception", ex);
10            closeServerSocket();
11            throw ex;
12        }
13    }

```

14

还真有捕获，然后在 catch 直接运行传递过来的 main 函数。看注释说这么做的原因是为了清除函数调用堆栈（确实从调用到 SS 的 main 函数，前面函数堆栈有好几层了）。这样能让 SS 感觉更像直接启动的吧（忽悠谁咧）。

然后我们终于能够去看 SS 的业务了，SS 的 main 函数：

```
1  public static void main(String[] args) {
2
3      /*
4       * In case the runtime switched since last boot (such as whe
5       * the old runtime was removed in an OTA), set the system
6       * property so that it is in sync. We can't do this in
7       * libnativehelper's JNIInvocation::Init code where we alrea
8       * had to fallback to a different runtime because it is
9       * running as root and we need to be the system user to set
10      * the property. http://b/11463182
11      */
12      SystemProperties.set("persist.sys.dalvik.vm.lib",
13                          VMRuntime.getRuntime().vmLibrary());
14
15      if (System.currentTimeMillis() < EARLIEST_SUPPORTED_TIME) {
16          // If a device's clock is before 1970 (before 0), a lot
17          // APIs crash dealing with negative numbers, notably
18          // java.io.File#setLastModified, so instead we fake it a
19          // hope that time from cell towers or NTP fixes it
20          // shortly.
21          Slog.w(TAG, "System clock is before 1970; setting to 197
22          SystemClock.setCurrentTimeMillis(EARLIEST_SUPPORTED_TIME
23      }
24
25      if (SamplingProfilerIntegration.isEnabled()) {
26          SamplingProfilerIntegration.start();
27          timer = new Timer();
28          timer.schedule(new TimerTask() {
29              @Override
30              public void run() {
31                  SamplingProfilerIntegration.writeSnapshot("syste
32              }
33          }, SNAPSHOT_INTERVAL, SNAPSHOT_INTERVAL);
34      }
35
36      // Mmmmmm... more memory!
37      dalvik.system.VMRuntime.getRuntime().clearGrowthLimit();
```

```

38         // The system server has to run all of the time, so it needs
39         // as efficient as possible with its memory usage.
40         VMRuntime.getRuntime().setTargetHeapUtilization(0.8f);
41
42         Environment.setUserRequired(true);
43
44         System.loadLibrary("android_servers");
45
46         Slog.i(TAG, "Entered the Android system server!");
47
48         // 启动 native 服务
49         // Initialize native services.
50         nativeInit();
51
52         // 启动 java 层服务
53         // This used to be its own separate thread, but now it is
54         // just the loop we run on the main thread.
55         ServerThread thr = new ServerThread();
56         thr.initAndLoop();
57     }
58
59

```

这个 main 其实不长（后面的在后面那个 ServerThread 里面），先是 nativeInit 启动 native 的服务：

```

1  // com_android_server_SystemServer.cpp =====
2
3  static void android_server_SystemServer_nativeInit(JNIEnv* env,
4      char propBuf[PROPERTY_VALUE_MAX];
5      property_get("system_init.startsensordservice", propBuf, "1")
6      // 就启动了一个 SensorService
7      if (strcmp(propBuf, "1") == 0) {
8          // Start the sensor service
9          SensorService::instantiate();
10     }
11 }
12

```

然后主要工作在 ServerThread 中。这个是 SystemServer 的内部类，看名字就很形象，服务线程啊：

```

1  // 这个函数巨长，有 1000 多行，我只贴一些有代表性的
2  public void initAndLoop() {

```

```
3      EventLog.writeEvent(EventLogTags.BOOT_PROGRESS_SYSTEM_F
4          SystemClock.uptimeMillis());
5
6      // 创建主线程 Looper
7      Looper.prepareMainLooper();
8
9      android.os.Process.setThreadPriority(
10         android.os.Process.THREAD_PRIORITY_FOREGROUND);
11
12      BinderInternal.disableBackgroundScheduling(true);
13      android.os.Process.setCanSelfBackground(false);
14
15      // Check whether we failed to shut down last time we tr
16      {
17         final String shutdownAction = SystemProperties.get(
18             ShutdownThread.SHUTDOWN_ACTION_PROPERTY, ""
19         if (shutdownAction != null && shutdownAction.length
20             boolean reboot = (shutdownAction.charAt(0) == '
21
22         final String reason;
23         if (shutdownAction.length() > 1) {
24             reason = shutdownAction.substring(1, shutdc
25         } else {
26             reason = null;
27         }
28
29         ShutdownThread.rebootOrShutdown(reboot, reason)
30     }
31 }
32
33 String factoryTestStr = SystemProperties.get("ro.factor
34 int factoryTest = "".equals(factoryTestStr) ? SystemSer
35     : Integer.parseInt(factoryTestStr);
36 final boolean headless = "1".equals(SystemProperties.ge
37
38 // 看到这一票熟悉的 Manager 了没
39 // 这里不是全部的, 下面还有, 不贴了
40 Installer installer = null;
41 AccountManagerService accountManager = null;
42 ContentService contentService = null;
43 LightsService lights = null;
44 PowerManagerService power = null;
45 DisplayManagerService display = null;
46 BatteryService battery = null;
47 VibratorService vibrator = null;
48 AlarmManagerService alarm = null;
49 MountService mountService = null;
50 NetworkManagementService networkManagement = null;
51 NetworkStatsService networkStats = null;
```

```

52     NetworkPolicyManagerService networkPolicy = null;
53     ConnectivityService connectivity = null;
54     EthernetService eth = null;
55     WifiP2pService wifiP2p = null;
56     WifiService wifi = null;
57     NsdService serviceDiscovery= null;
58     IPackageManager pm = null;
59     Context context = null;
60     WindowManagerService wm = null;
61     BluetoothManagerService bluetooth = null;
62     DockObserver dock = null;
63     UsbService usb = null;
64     SerialService serial = null;
65     TwilightService twilight = null;
66     UiModeManagerService uiMode = null;
67     RecognitionManagerService recognition = null;
68     NetworkTimeUpdateService networkTimeUpdater = null;
69     CommonTimeManagementService commonTimeMgmtService = nul
70     InputManagerService inputManager = null;
71     TelephonyRegistry telephonyRegistry = null;
72     ConsumerIrService consumerIr = null;
73
74     ... ..
75
76     try {
77         // new Service 对象, 然后 add 到 SM 中, 非常规范的操作
78         // 这里也只贴前面几个, 下面都差不多, 无非有几个玩非主流
79         // 后面分析到那些再说。
80         Slog.i(TAG, "Display Manager");
81         display = new DisplayManagerService(context, wmHand
82         ServiceManager.addService(Context.DISPLAY_SERVICE,
83
84         Slog.i(TAG, "Telephony Registry");
85         telephonyRegistry = new TelephonyRegistry(context);
86         ServiceManager.addService("telephony.registry", tel
87
88         Slog.i(TAG, "Scheduling Policy");
89         ServiceManager.addService("scheduling_policy", new
90
91         AttributeCache.init(context);
92
93         if (!display.waitForDefaultDisplay()) {
94             reportWtf("Timeout waiting for default display
95                 new Throwable());
96         }
97
98     ... ..
99
100    } catch (RuntimeException e) {

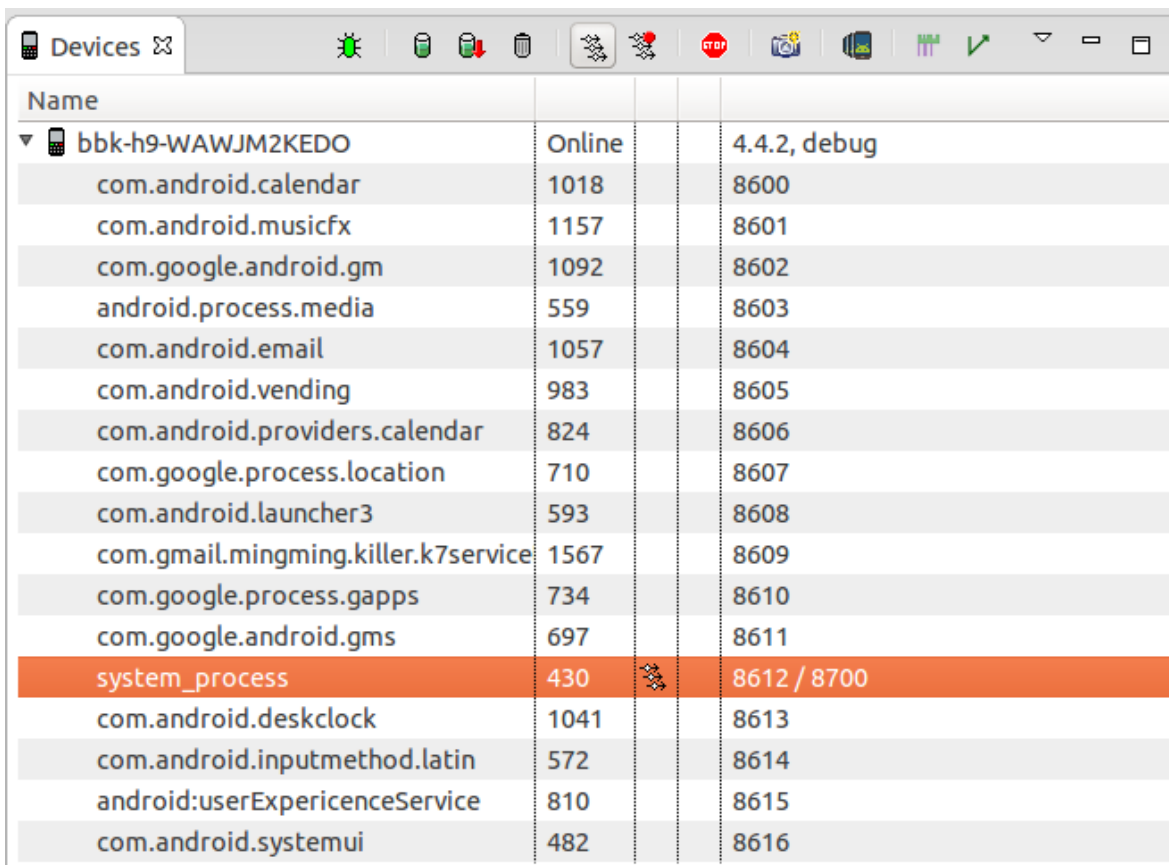
```

```

101         Slog.e("System", "*****");
102         Slog.e("System", "***** Failure starting cor
103     }
104
105     ... ..
106
107     // 主线程 Looper 开始循环
108     Looper.loop();
109     Slog.d(TAG, "System ServerThread is exiting!");
110 }
111

```

看 SS main 最后那里注释说，这些 SS 应该在单独的线程里面跑，但是现在暂时都在主线程中了。看 ServerThread 中代码，是都是在主线程中。SS 是一个典型的一个进程里面多个服务的例子（服务还很多，好像有十几个吧 -_-||）。DDMS 里面的截图很明了了：



Name	Online	4.4.2, debug
com.android.calendar	1018	8600
com.android.musicfx	1157	8601
com.google.android.gm	1092	8602
android.process.media	559	8603
com.android.email	1057	8604
com.android.vending	983	8605
com.android.providers.calendar	824	8606
com.google.process.location	710	8607
com.android.launcher3	593	8608
com.gmail.mingming.killer.k7service	1567	8609
com.google.process.gapps	734	8610
com.google.android.gms	697	8611
system_process	430	8612 / 8700
com.android.deskclock	1041	8613
com.android.inputmethod.latin	572	8614
android:userExperienceService	810	8615
com.android.systemui	482	8616

（pid 430 是上面 ps 看到的 SS 的）

Threads					
Heap Allocation Tracker Network Statistics File I					
ID	Tid	Status	utime	stime	Name
1	430	Native	284	163	main
*2	432	VmWait	82	2	GC
*3	434	VmWait	0	0	Signal Catcher
*4	435	Runnable	76	273	JDWP
*5	436	VmWait	69	115	Compiler
*6	437	Wait	1	0	ReferenceQueueDaemon
*7	439	Wait	4	4	FinalizerDaemon
*8	440	Wait	0	0	FinalizerWatchdogDaemon
9	441	Native	28	73	Binder_1
10	442	Native	39	72	Binder_2
11	443	Native	316	934	SensorService
12	444	Native	34	83	WindowManager
13	445	Native	160	440	ActivityManager
14	446	Native	188	395	android.bg
39	475	Native	0	10	AudioService
40	476	Native	0	4	UEventObserver
41	477	Native	0	0	backup
42	495	Native	0	0	Thread-51
43	496	Wait	2	3	AsyncTask #1
44	498	Native	0	0	Thread-53
45	497	Native	34	75	Binder_3
46	549	Native	0	0	CaptivePortalTracker
47	557	Native	0	0	UsbService host thread
48	558	TimedWait	4	7	watchdog
49	585	Native	1	0	NetworkTimeUpdateService
50	588	Native	25	67	Binder_4
51	589	Native	24	82	Binder_5
52	590	Native	0	0	UsbConnectedBlock-tip-thread
53	622	Native	39	64	Binder_6
54	623	Native	25	63	Binder_7
55	624	Native	22	73	Binder_8
56	924	Wait	2	4	AsyncTask #2
57	805	Native	0	0	SoundPool
58	806	Native	0	2	SoundPoolThread
59	930	Wait	1	6	AsyncTask #3
60	840	Wait	0	0	pool-1-thread-1

61	931	Wait	2	5	AsyncTask #4
62	932	Wait	2	5	AsyncTask #5
63	1269	Native	5	11	Binder_9
64	1344	Native	1	10	Binder_A
65	1382	Native	1	11	Binder_B
66	1421	Native	1	11	Binder_C

之前还觉得 binder 默认给自动线程设置 15 个是不是有点多，现在看起来这个值是不是就是对着 SS 来设的。结合前面的讨论，15 个加上前面那个主动开的线程应该能应付大多数情况。看截图一般高峰期就 11、12 个 Bp 吧

(Binder_C、Binder_D，前面说了 kernel 自动调节的线程目前一旦传了，不会退出的，所以这个最大数差不多能说明最大负载的情况吧)。然后看到这些线程的名字都是 Binder_X、这个名字是在 ProcessState 里取的：

```

1  String8 ProcessState::makeBinderThreadName() {
2      int32_t s = android_atomic_add(1, &mThreadPoolSeq);
3      String8 name;
4      name.appendFormat("Binder_%X", s);
5      return name;
6  }
7

```

这不是 isMain 的线程连名字都是大众脸（所以看到 Binder_X 的线程，就知道是 kernel binder 驱动自动调节创建出来的）。

然后这里再讨论一个问题，就是 SS 一个进程同时在跑多个不同的服务，那不同的线程如果能够把不同服务的 Bp 请求发送到对应的服务处理呢。刚开始我也觉得有点不好理解，但是从上一篇 binder 对象传递之后我就明白了，binder 根本不需要区分这个请求是服务进程中哪个服务的。还记得 binder 对象传递篇中，服务向 SM 注册的时候，传递过去的对象是 Bn，然后把本地对象的指针传递过去了，然后在 kernel 的 `binder_node` 中有保存这个本地指针的。所以只要能找到正确的 node（通过的 Bp 的 handle 值取 ref，再通过 ref 取 node，忘记了的回去看 binder 对象传递篇）就能取得服务的本地对象指针，然后在服务进程的线程中执行这个本地对象的方法就能执行正确的服务函数了。所以这些对于多线程，一个进程多个服务来说是透明，它们不用管这个 binder 对象是本进程

内哪个服务的，只管执行它的 transation 方法就行。

总结

android 真的为 IPC 做了很多事情，前面说的效率、框架封装不说，这里直接帮你把并发多线程支持给你解决了。但是还有 framework 还是有些服务不是用 binder 的（vold、zygote），唉，人多了，另起炉灶就再所难免了。

 [android](#)

 [Android Framework](#)



上一篇:

« [Android Binder 分析——普通服务 Binder 对象的传递](#)

下一篇:

» [Android Binder 分析——懒人的工具（AIDL）](#)

分类

[Android Development](#) ³⁵

[Android Framework](#) ⁴⁷

[Basics Knowledge](#) ¹¹

[Linux](#) ²⁵

[MiniGUI](#) ¹²

[Other](#) ⁸

[Server](#) ¹

Window¹⁰

标签


Linux¹ android⁸² basics¹¹ install⁹ linux²⁸ minigui¹³ opengl³
other⁵ server¹ shell⁵ window¹¹

哥的后勤处 o(^▽^)o

-  QQ 空间
-  GitHub
-  战斗力
-  Makrdown

最近冒泡的小伙伴 ~(•'~'•)~

友情链接 o(^▽^)o

-  桃园小七的博客

Powered by hexo and Theme by Lightmoon © 2017 Mingming