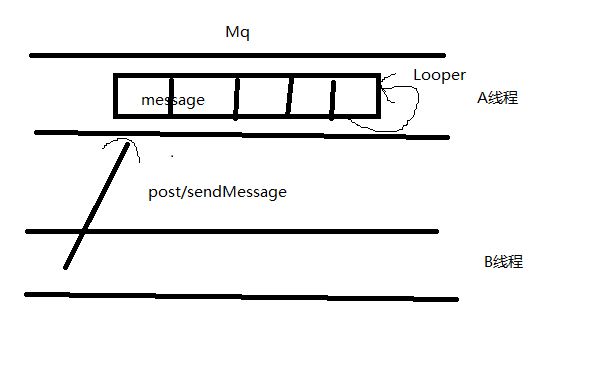
|  |  |  |
| --- | --- | --- |
| Handler原理 | threadlocal原理 | Binder原理 |
| 开机流程 | 应用启动流程 | 关机流程 |
| 事件传播机制 | view绘制流程 | SurfaceFlinger原理 |
| Synchronization，lock，volatile wait/notify | PMS | WMS |
| Service | BroadcastReceiver | ContentProvider |
| LeakCanary | WatchDog | HashMap/ArrayMap/SparseArray |
| 共享内存 | 性能优化 |  |
|  |  |  |
|  |  |  |

Handler原理

主要为了实现线程间通信。一共4个角色，Handler，Message，MessageQueue，Looper。

A线程new Handler之前需先执行Looper.prepare方法，再执行Looper.loop开启循环，我们平常在主线程中没执行是因为应用启动过程中已经先执行过了。B线程往A线程通过handler.post或者sendMessage后，都会把此message按时间顺序加入到mq中，Looper会一直循环查找此消息队列中是否有message处理，有的话就执行dispatchMessage，最终回调其handleMessage或者执行其run方法。

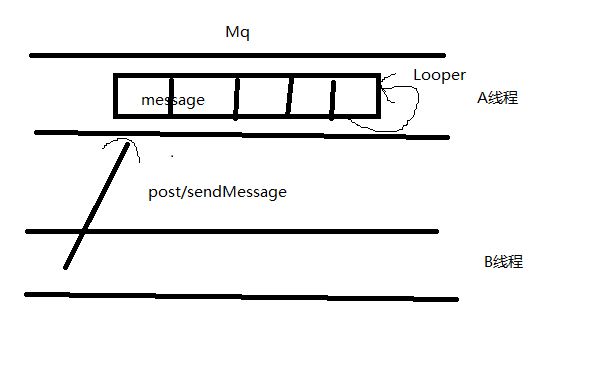


问：Looper在主线程死循环，为什么不会ANR？

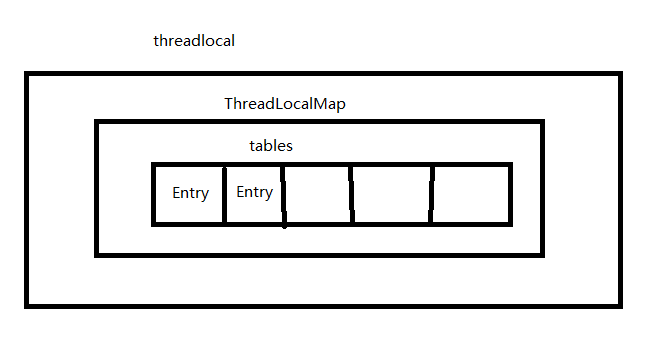
当没有消息时，线程会阻塞在queue.next，这里用到了Linux的epoll多路复用机制（可以同时监控多个文件描述符，当某个描述符就绪后，立即通知之相应程序进行读或者写操作），此时主线程会阻塞在这里，这个并不是ANR

问：同步屏障的机制。

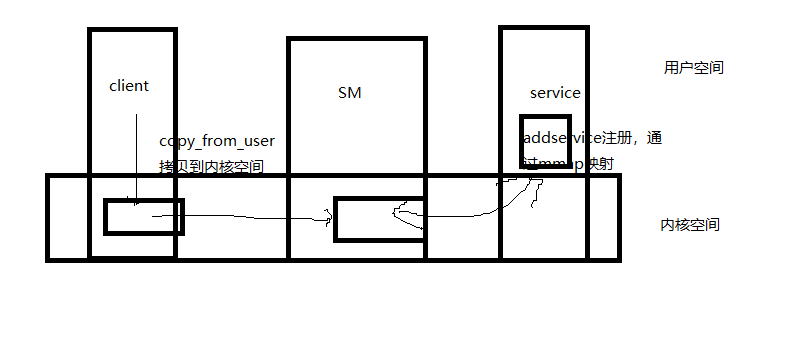
消息一共有3种，同步消息，异步消息，屏障消息。我们平常发的消息都是同步消息，同步屏障是为了实现异步消息优先处理。比如view绘制流程中就会先通过mq.postSyncBarrier发送屏障消息，其target为null，mq在获取到target为null时说明收到屏障消息，然后遍历其后面直到找到异步消息再处理，异步消息可以通过message.setAsynchronous(true)来设置其为异步消息。之后通过removeSyncBarrier来移除同步屏障。

ThreadLocal原理

Threadlocal主要是实现线程间数据隔离，即threadlocal填充的变量属于当前线程，其他线程是访问不到的。其get/set方法实际是是通过操作内部类ThreadLocalMap来完成存取的，可以把ThreadLocal理解成对ThreadLocalMao的封装。ThreadLocalMap本质可以理解成Hashmap，其持有了静态内部类Entry的数组tables,而Entry的key为TheadLocal的弱引用，而value是强引用，所以Threadlocal使用不当可能会内存泄露。最保险的是用完后记得remove



Binder原理

Binder主要是解决进程间通信的。Android是基于Linux系统，Linux有进程隔离，即两个进程间是不能直接通信的，我们平常的应用进程都在用户空间，用户空间和内核空间是可以通信的，Linux正常情况进程间通信需要拷贝两次，即从a进程到内核空间，再从内核空间到b进程，Android为了提高效率使用binder只需拷贝一次，binder是基于内存映射（mmap）来实现的，内存映射可以理解成把用户空间的一块内存区域映射到内核空间，映射建立后，用户对这块内存空间的修改可以直接反映到内核空间。有4个角色，client，service，ServiceManager，binder驱动。Service先通过binder驱动往SM中注册服务，key为具体的服务名，client根据服务名称通过binder驱动获取服务的代理对象，执行代理对象的方法时，会通过binder驱动调用到服务端真实的方法。开机过程中AMS，WMS等都是通过这种方式注册的。一般应用没有权限注册系统服务，Android提供了aidl这种匿名binder方式

开机流程

点电源开关后会启动主板预置Rom，加载bootloader引导程序，引导程序负责加载Linux kernel内核。Linux层会先启动swapper进程（0号进程），是所有进程的祖先，fork出1号进程（init进程，用户进程，是所有用户进程的祖先）和2号进程（kthread进程，内核进程，是所有linux内核进程的祖先）。Init进程解析init.rc配置文件并fork出zygote，ServiceManager,SurfaceFlinger等进程。会先加载app\_main.cpp的main方法，此方法会调用AndroidRuntime的start方法，这里会先启动jvm虚拟机，再反射调用zygoteInit.java的main方法，这个是运行的第一个java方法，此时运行正在zygote进程。此方法会先fork出systemserver进程，然后执行runselectLoop循环等待有新的fork请求过来。Fork机制是一次调用，两次返回，对于新fork出的systemserver进程，会反射调用SystemServer的main方法，这里会执行其run方法，启动引导服务（ams，pms，atms等），核心服务，其他服务（wms等），最后在ams.systemReady中启动launcher和systemui

应用启动流程

点击应用图标后Launcher会执行startActivity -> startActivityForResult ->Instrumentation.execStartActivity->ATMS.startActivity，这时到systemserver进程，startActivityAsUser->ActivityStarter.execute->startActivityUnchecked->startActivityInner,之后在TaskFragment.resumeTopActivity中会判断目标进程是否启动，如果没启动，会调用ATMS.startProcessAsync来启动，这里会通过socket通信通知zygote进程fork出指定进程，并回调ActivityThread.main方法。Main会先Looper.prepareMainLooper，所以在主线程中我们不需要在Looper.prepare。之后执行at.attach，这里会调用到ams的attachApplication方法，并传了ApplicationThread，此为binder对象，方便ams去回调ActivityThread。Ams会先记录此applicationthread，然后执行其bindApplication方法，然后执行atms.attachApplication方法。应用进程binderApplication会先new Instrumentation，应用的生命周期都是通过instrument来执行的，这里执行其callApplicationOnCreate方法，即回调application.onCreate。之后atms.attachApplication会在realStartActivityLocked方法中构建事务回调LaunchActivityItem，如果需要可见，指定最终生命周期状态为ResumeActivityItem，否则为PauseActivityItem，执行事务后会分别执行at的handleLaunchActivity和handleResumeActivity，其中handleLaunchActivity会先执行activity的attach方法，然后通过instrument执行callactivityonCreate，即activity.onCreate方法。我们一般会在oncreate中setContentView，这里会先new个DecorView，此时viewrootimpl还未创建。之后在handleResumeActivity中会先执行activity的onResume，此时view还没可见，之后通过wm.addview完成view具体绘制，之后才通过makeVisible把view可见

SurfaceFlinger

主要负责surface的管理和合成，把合成后的fence交给屏幕驱动以显示。SF在开机时通过sf.rc加载，会先加载main\_sf.main，其先创建sf并执行其init方法，这里面会创建compositionengine合成引擎和RenderEnginer，HWComposer并关联起来，之后添加到sf的系统服务中，最后while循环一直等待消息。在view绘制流程中，viewRootImpl.setView -> session.addToDisplayAsUser -> wms.addWindow，wms会创建WindowState，并执行其attach，这里应用第一次创建窗口时会创建SurfaceSession，对应于JNI层的SufaceComposerSession，其本质是与sf创建连接时返回sf创建的Client对象。之后在应用进程requestlayout开始绘制流程，scheduler/do/performTraversals,这里会先session.relayoutwindow，这里客户端传的是空的sc,再开始具体的绘制流程。Relayoutwindow会调用到wms.createSurfaceControl创建surfacecontrol，其实也是通过JNI层上步创建的client执行其createSurface，这里会根据不同的surface通过sf创建不同的layer，比如BufferqueueLayer，sf返回代理对象Handle，其本质是个binder对象，也就是说sc中包含了sf创建的Handle的代理地址，其关联了创建的layer。之后应用进程会基于此sc来创建BLASTBufferqueue，BBq构造函数中会创建BufferQueueCore和BBQBufferQueueProducer，BufferQueueConsume并关联起来。然后执行bbq的createSurface，其实返回的是BBqSurface对象。在view.draw前会先通过surface.lockCanvas先锁定一块内存返回一个有效的canvas，其内部会先通过生产者dequeueBuffer从bufferqueue队列中获取一块buffer，然后把buffer转化成skBitmap，SkBitmap可以被canvas直接绘制。view的draw就通过此canvas来，本质都是通过jni曾的SkiaCanvas绘制到SkBitmap这种。之后surface.unlockCanvasAndPost来释放并post此canvas，这里会queueBuffer把绘制后的buffer提交到缓冲队列。之后会调用到消费者的acquireBuffer取出graphBuffer并通过事务传递到sf进程。最终在Handler.handleMessage中会先调用sf的commit完成提交，如果返回true就走后面的composite合成流程。

Commit流程会把新创建的layer加入到以z轴刨析的layersSortedByz集合中，然后执行每个layer的latchBuffer来确定是否需要重新计算显示区域，再去计算边界和脏区域，如果需要合成就返回true。

合成流程会先构建合成需要的参数refreshArgs，把SF里保存的layer等信息传进去，再调用合成引擎的present方法。此方法会遍历所有的显示设备output，调用其prepare和present方法，其中prepare方法会从顶层到底层遍历所有的Layer，把不需要参与合成的layer放到mReleasedLayers中。present方法会调用prepareFrame通过HWC服务去确认是用客户端合成还是硬件合成，如果由硬件合成，合成流程直接结束，否则会在finishFrame里处理客户端合成，即GPU合成，这里会调用composeSurfaces去真正处理GPU合成，然后会postFramebuffer告诉HWC做最后的合成并显示，这里会调用hwc的presentAndGetReleaseFences先展示已经要显示的帧（FrameFences.presentFence），然后把当前的帧赋值到此字段，等待下一个vsync信号后再展示。合成完后会在postComposition通过binder调用到客户端的onTransactionCompleted方法，这里会调用BBQ的releaseBuffer方法，其会调用消费者的releaseBuffer，释放完后把slot放到空闲队列，如果有线程阻塞在dequeuebuffer的，此时会被唤醒。