事实上，会在进入死循环之前便创建了新binder线程，在代码ActivityThread.main()中：

public static void main(String[] args) {

....

//创建Looper和MessageQueue对象，用于处理主线程的消息

Looper.prepareMainLooper();

//创建ActivityThread对象

ActivityThread thread = new ActivityThread();

//建立Binder通道 (创建新线程)

thread.attach(false);

Looper.loop(); //消息循环运行

throw new RuntimeException("Main thread loop unexpectedly exited");

}

作者：Gityuan  
链接：https://www.zhihu.com/question/34652589/answer/90344494  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

**thread.attach(false)；便会创建一个Binder线程（具体是指ApplicationThread，Binder的服务端，用于接收系统服务AMS发送来的事件），该Binder线程通过Handler将Message发送给主线程**，具体过程可查看 [startService流程分析](https://link.zhihu.com/?target=http%3A//gityuan.com/2016/03/06/start-service/)，这里不展开说，简单说Binder用于进程间通信，采用C/S架构。关于binder感兴趣的朋友，可查看我回答的另一个知乎问题：  
[为什么Android要采用Binder作为IPC机制？ - Gityuan的回答](https://www.zhihu.com/question/39440766/answer/89210950)

另外，**ActivityThread实际上并非线程**，不像HandlerThread类，ActivityThread并没有真正继承Thread类，只是往往运行在主线程，该人以线程的感觉，其实承载ActivityThread的主线程就是由Zygote fork而创建的进程。

**主线程的死循环一直运行是不是特别消耗CPU资源呢？** 其实不然，这里就涉及到**Linux pipe/epoll机制**，简单说就是在主线程的MessageQueue没有消息时，便阻塞在loop的queue.next()中的nativePollOnce()方法里，详情见[Android消息机制1-Handler(Java层)](https://link.zhihu.com/?target=http%3A//www.yuanhh.com/2015/12/26/handler-message-framework/%23next)，此时主线程会释放CPU资源进入休眠状态，直到下个消息到达或者有事务发生，通过往pipe管道写端写入数据来唤醒主线程工作。这里采用的epoll机制，是一种IO多路复用机制，可以同时监控多个描述符，当某个描述符就绪(读或写就绪)，则立刻通知相应程序进行读或写操作，本质同步I/O，即读写是阻塞的。 **所以说，主线程大多数时候都是处于休眠状态，并不会消耗大量CPU资源。**

**(3) Activity的生命周期是怎么实现在死循环体外能够执行起来的？**

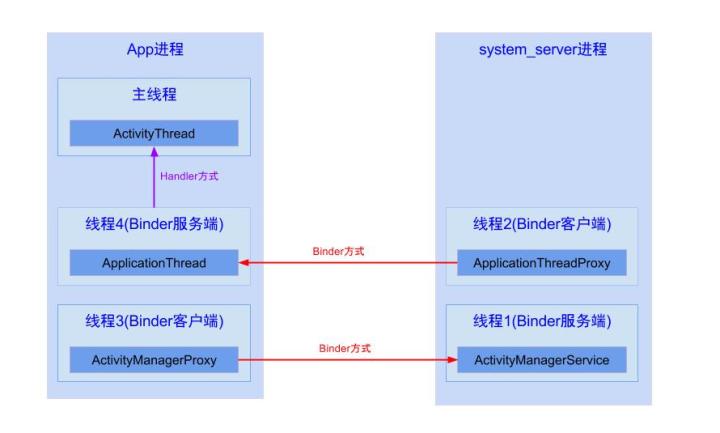
ActivityThread的内部类H继承于Handler，通过handler消息机制，简单说Handler机制用于同一个进程的线程间通信。

**Activity的生命周期都是依靠主线程的Looper.loop，当收到不同Message时则采用相应措施：**  
在H.handleMessage(msg)方法中，根据接收到不同的msg，执行相应的生命周期。

比如收到msg=H.LAUNCH\_ACTIVITY，则调用ActivityThread.handleLaunchActivity()方法，最终会通过反射机制，创建Activity实例，然后再执行Activity.onCreate()等方法；  
再比如收到msg=H.PAUSE\_ACTIVITY，则调用ActivityThread.handlePauseActivity()方法，最终会执行Activity.onPause()等方法。 上述过程，我只挑核心逻辑讲，真正该过程远比这复杂。

**主线程的消息又是哪来的呢？**当然是App进程中的其他线程通过Handler发送给主线程，请看接下来的内容：

--------------------------------------------------------------------------------------------------------------------------------------  
**最后，从进程与线程间通信的角度，通过一张图加深大家对App运行过程的理解：**

&lt;img src="https://pic1.zhimg.com/50/7fb8728164975ac86a2b0b886de2b872\_hd.jpg" data-rawwidth="890" data-rawheight="535" class="origin\_image zh-lightbox-thumb" width="890" data-original="https://pic1.zhimg.com/7fb8728164975ac86a2b0b886de2b872\_r.jpg"&gt;  
**system\_server进程是系统进程**，java framework框架的核心载体，里面运行了大量的系统服务，比如这里提供ApplicationThreadProxy（简称ATP），ActivityManagerService（简称AMS），这个两个服务都运行在system\_server进程的不同线程中，由于ATP和AMS都是基于IBinder接口，都是binder线程，binder线程的创建与销毁都是由binder驱动来决定的。

**App进程则是我们常说的应用程序**，主线程主要负责Activity/Service等组件的生命周期以及UI相关操作都运行在这个线程； 另外，每个App进程中至少会有两个binder线程 ApplicationThread(简称AT)和ActivityManagerProxy（简称AMP），除了图中画的线程，其中还有很多线程，比如signal catcher线程等，这里就不一一列举。

Binder用于不同进程之间通信，由一个进程的Binder客户端向另一个进程的服务端发送事务，比如图中线程2向线程4发送事务；而handler用于同一个进程中不同线程的通信，比如图中线程4向主线程发送消息。

**结合图说说Activity生命周期，比如暂停Activity，流程如下：**

1. 线程1的AMS中调用线程2的ATP；（由于同一个进程的线程间资源共享，可以相互直接调用，但需要注意多线程并发问题）
2. 线程2通过binder传输到App进程的线程4；
3. 线程4通过handler消息机制，将暂停Activity的消息发送给主线程；
4. 主线程在looper.loop()中循环遍历消息，当收到暂停Activity的消息时，便将消息分发给ActivityThread.H.handleMessage()方法，再经过方法的调用，最后便会调用到Activity.onPause()，当onPause()处理完后，继续循环loop下去。

作者：于翔  
链接：https://www.zhihu.com/question/34652589/answer/157834250  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

简单一句话是：Android应用程序的主线程在进入消息循环过程前，会在内部创建一个Linux管道（Pipe），这个管道的作用是使得Android应用程序主线程在消息队列为空时可以进入空闲等待状态，并且使得当应用程序的消息队列有消息需要处理时唤醒应用程序的主线程。

---  
这一题是需要从消息循环、消息发送和消息处理三个部分理解Android应用程序的消息处理机制了，这里我对一些要点作一个总结：

A. Android应用程序的消息处理机制由消息循环、消息发送和消息处理三个部分组成的。

B. Android应用程序的主线程在进入消息循环过程前，会在内部创建一个Linux管道（Pipe），这个管道的作用是使得Android应用程序主线程在消息队列为空时可以进入空闲等待状态，并且使得当应用程序的消息队列有消息需要处理时唤醒应用程序的主线程。

C. Android应用程序的主线程进入空闲等待状态的方式实际上就是在管道的读端等待管道中有新的内容可读，具体来说就是是通过Linux系统的Epoll机制中的epoll\_wait函数进行的。

D. 当往Android应用程序的消息队列中加入新的消息时，会同时往管道中的写端写入内容，通过这种方式就可以唤醒正在等待消息到来的应用程序主线程。

E. 当应用程序主线程在进入空闲等待前，会认为当前线程处理空闲状态，于是就会调用那些已经注册了的IdleHandler接口，使得应用程序有机会在空闲的时候处理一些事情。

了解下linux的epoll你就知道为什么不会被卡住了，先说结论：阻塞是有的，但是不会卡住  
主要原因有2个

1，epoll模型  
当没有消息的时候会epoll.wait，等待句柄写的时候再唤醒，这个时候其实是阻塞的。

2，所有的ui操作都通过handler来发消息操作。  
比如屏幕刷新16ms一个消息，你的各种点击事件，所以就会有句柄写操作，唤醒上文的wait操作，所以不会被卡死了。

作者：陈昱全  
链接：https://www.zhihu.com/question/34652589/answer/59571257  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

作者：Rocko  
链接：https://www.zhihu.com/question/34652589/answer/59614987  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

Handler 机制应该基本都知道了，题主点进去源码看下 ActivityThread 就能基本搞明白了，为什么主线程不会因为 Looper.loop() 里的死循环卡死？

首先 ActivityThread 并不是一个 Thread，就只是一个 final 类而已。我们常说的主线程就是从这个类的 main 方法开始，main 方法很简短，一眼就能看全，我们看到里面有 Looper 了，那么接下来就**找找 ActivityThread 对应的 Handler** 啊，就是内部类 H，其继承 Handler，贴出 handleMessage 的小部分：

public void handleMessage(Message msg) {

if (DEBUG\_MESSAGES) Slog.v(TAG, ">>> handling: " + codeToString(msg.what));

switch (msg.what) {

case LAUNCH\_ACTIVITY: {

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityStart");

final ActivityClientRecord r = (ActivityClientRecord) msg.obj;

r.packageInfo = getPackageInfoNoCheck(

r.activityInfo.applicationInfo, r.compatInfo);

handleLaunchActivity(r, null);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

} break;

case RELAUNCH\_ACTIVITY: {

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityRestart");

ActivityClientRecord r = (ActivityClientRecord)msg.obj;

handleRelaunchActivity(r);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

} break;

case PAUSE\_ACTIVITY:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityPause");

handlePauseActivity((IBinder)msg.obj, false, (msg.arg1&1) != 0, msg.arg2,

(msg.arg1&2) != 0);

maybeSnapshot();

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case PAUSE\_ACTIVITY\_FINISHING:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityPause");

handlePauseActivity((IBinder)msg.obj, true, (msg.arg1&1) != 0, msg.arg2,

(msg.arg1&1) != 0);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case STOP\_ACTIVITY\_SHOW:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityStop");

handleStopActivity((IBinder)msg.obj, true, msg.arg2);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case STOP\_ACTIVITY\_HIDE:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityStop");

handleStopActivity((IBinder)msg.obj, false, msg.arg2);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case SHOW\_WINDOW:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityShowWindow");

handleWindowVisibility((IBinder)msg.obj, true);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case HIDE\_WINDOW:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityHideWindow");

handleWindowVisibility((IBinder)msg.obj, false);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case RESUME\_ACTIVITY:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityResume");

handleResumeActivity((IBinder) msg.obj, true, msg.arg1 != 0, true);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

case SEND\_RESULT:

Trace.traceBegin(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER, "activityDeliverResult");

handleSendResult((ResultData)msg.obj);

Trace.traceEnd(Trace.TRACE\_TAG\_ACTIVITY\_MANAGER);

break;

...........

}

看完这 Handler 里处理消息的内容应该明白了吧， Activity 的生命周期都有对应的 case 条件了，ActivityThread 有个 getHandler 方法，得到这个 handler 就可以发送消息，然后 loop 里就分发消息，然后就发给 handler, 然后就执行到 H（Handler ）里的对应代码。所以这些代码就不会卡死～，有消息过来就能执行。举个例子，在 ActivityThread 里的内部类 ApplicationThread 中就有很多 sendMessage 的方法：

......

public final void schedulePauseActivity(IBinder token, boolean finished,

boolean userLeaving, int configChanges, boolean dontReport) {

sendMessage(

finished ? H.PAUSE\_ACTIVITY\_FINISHING : H.PAUSE\_ACTIVITY,

token,

(userLeaving ? 1 : 0) | (dontReport ? 2 : 0),

configChanges);

}

public final void scheduleStopActivity(IBinder token, boolean showWindow,

int configChanges) {

sendMessage(

showWindow ? H.STOP\_ACTIVITY\_SHOW : H.STOP\_ACTIVITY\_HIDE,

token, 0, configChanges);

}

public final void scheduleWindowVisibility(IBinder token, boolean showWindow) {

sendMessage(

showWindow ? H.SHOW\_WINDOW : H.HIDE\_WINDOW,

token);

}

public final void scheduleSleeping(IBinder token, boolean sleeping) {

sendMessage(H.SLEEPING, token, sleeping ? 1 : 0);

}

public final void scheduleResumeActivity(IBinder token, int processState,

boolean isForward, Bundle resumeArgs) {

updateProcessState(processState, false);

sendMessage(H.RESUME\_ACTIVITY, token, isForward ? 1 : 0);

}

......

去看下handler机制就明白了，网上一大把。

1.handler机制是使用pipe来实现的

2.主线程没有消息处理时阻塞在管道的读端

3.binder线程会往主线程消息队列里添加消息，然后往管道写端写一个字节，这样就能唤醒主线程从管道读端返回，也就是说queue.next()会调用返回

4.dispatchMessage()中调用onCreate, onResume

作者：知乎用户  
链接：https://www.zhihu.com/question/34652589/answer/59578531  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

ActivityThread是应用程序的入口，这里你可以看到写Java程序时司空见惯的main方法，而main方法正是整个Java程序的入口。

ActivityThread的main方法主要就是做消息循环，一旦退出消息循环，那么你的程序也就可以退出了。

从消息队列中取消息可能会阻塞，取到消息会做出相应的处理。如果某个消息处理时间过长，就可能会影响UI线程的刷新速率，造成卡顿的现象。

[发布于 2015-08-17](/question/34652589/answer/59558499)

作者：hi大头鬼hi  
链接：https://www.zhihu.com/question/34652589/answer/59558499  
来源：知乎  
著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。