Questions

1. Looper 死循环为什么不会导致应用卡死，会消耗大量资源吗?
2. 主线程的消息循环机制是什么（死循环如何处理其它事务）？
3. ActivityThread 的动力是什么？（ActivityThread执行Looper的线程是什么）
4. Handler 是如何能够线程切换，发送Message的？（线程间通讯）
5. 子线程有哪些更新UI的方法。
6. 子线程中Toast，showDialog，的方法。（和子线程不能更新UI有关吗）
7. 如何处理Handler 使用不当导致的内存泄露?

1. Looper 死循环为什么不会导致应用卡死?

> 线程默认没有Looper的，如果需要使用Handler就必须为线程创建Looper。我们经常提到的主线程，也叫UI线程，它就是ActivityThread，ActivityThread被创建时就会初始化Looper，这也是在主线程中默认可以使用Handler的原因。

首先我们看一段代码

```
new Thread(new Runnable() {
    @Override
                public void run() {
        Log.e("qdx", "step 0 ");
        Looper.prepare();
        Toast.makeText(MainActivity.this, "run on Thread",
Toast.LENGTH_SHORT).show();
        Log.e("qdx", "step 1 ");
        Looper.loop();
        Log.e("qdx", "step 2 ");
    }
}
).start();
```

我们知道`Looper.loop();`里面维护了一个死循环方法，所以按照理论，上述代码执行的应该是 step 0 –>step 1 也就是说循环在`Looper.prepare();`与`Looper.loop();`之间。

```
E/qdx: step 0
E/qdx: step 1
```

> 在子线程中，如果手动为其创建了Looper，那么在所有的事情完成以后应该调用quit方法来终止消息循环，否则这个子线程就会一直处于等待（阻塞）状态，而如果退出Looper以后，这个线程就会立刻（执行所有方法并）终止，因此建议不需要的时候终止Looper。

执行结果也正如我们所说，这时候如果了解了`ActivityThread`，并且在main方法中我们会看到主线程也是通过Looper方式来维持一个消息循环。

```
public static void main(String[] args) {
            Looper.prepareMainLooper();
    //创建Looper和MessageQueue对象，用于处理主线程的消息
    ActivityThread thread = new ActivityThread();
    thread.attach(false);
    //建立Binder通道 (创建新线程)
    if (sMainThreadHandler == null) {
```

```
        sMainThreadHandler = thread.getHandler();
    }
    Trace.traceEnd(Trace.TRACE_TAG_ACTIVITY_MANAGER);
    Looper.loop();
    //如果能执行下面方法，说明应用崩溃或者是退出了...
    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

那么回到我们的问题上，这个死循环会不会导致应用卡死，即使不会的话，它会慢慢的消耗越来越多的资源吗?

摘自：Gityuan

> 对于线程即是一段可执行的代码，当可执行代码执行完成后，线程生命周期便该终止了，线程退出。而对于主线程，我们是绝不希望会被运行一段时间，自己就退出，那么如何保证能一直存活呢？简单做法就是可执行代码是能一直执行下去的，死循环便能保证不会被退出，例如，binder线程也是采用死循环的方法，通过循环方式不同与Binder驱动进行读写操作，当然并非简单地死循环，无消息时会休眠。但这里可能又引发了另一个问题，既然是死循环又如何去处理其他事务呢？通过创建新线程的方式。真正会卡死主线程的操作是在回调方法onCreate/onStart/onResume等操作时间过长，会导致掉帧，甚至发生ANR，looper.loop本身不会导致应用卡死。

> 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，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。

2. 主线程的消息循环机制是什么?

> 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");
    }
```

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：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。

> thread.attach(false)方法函数中便会创建一个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。
>
> 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。
>
> 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

---

system_server进程

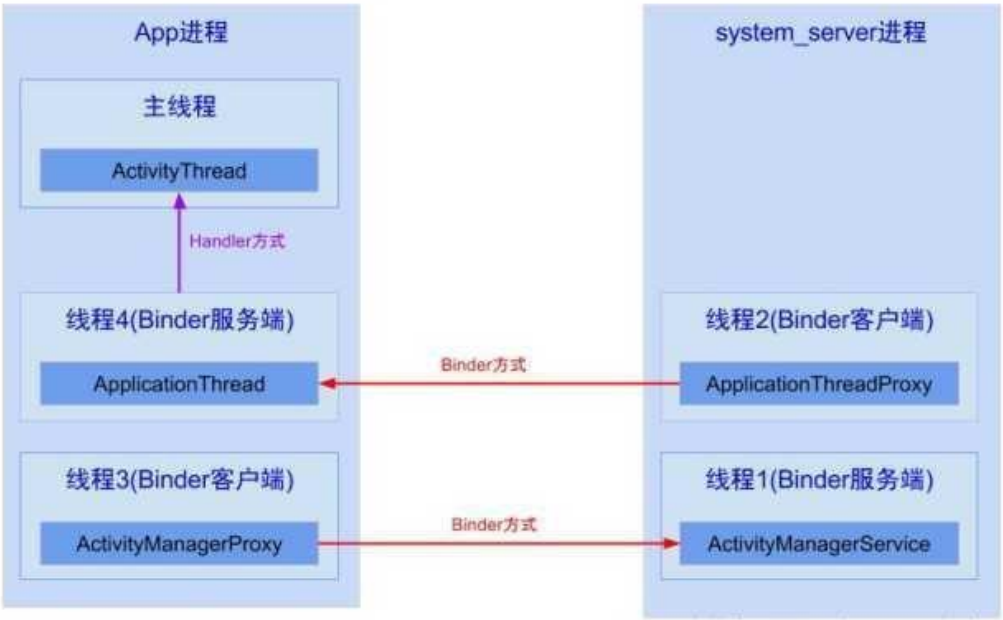
> system_server进程是系统进程，java framework框架的核心载体，里面运行了大量的系统服务，比如这里提供`ApplicationThreadProxy`（简称ATP），`ActivityManagerService`（简称AMS），这个两个服务都运行在system_server进程的不同线程中，由于ATP和AMS都是基于IBinder接口，都是binder线程，binder线程的创建与销毁都是由binder驱动来决定的。

App进程

> 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

Binder

> 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。

补充:

> 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。

最后通过《Android开发艺术探索》的一段话总结：

> ActivityThread通过ApplicationThread和AMS进行进程间通讯，AMS以进程间通信的方式完成ActivityThread的请求后会回调ApplicationThread中的Binder方法，然后ApplicationThread会向H发送消息，H收到消息后会将ApplicationThread中的逻辑切换到ActivityThread中去执行，即切换到主线程中去执行，这个过程就是。主线程的消息循环模型

> 另外，ActivityThread实际上并非线程，不像HandlerThread类，ActivityThread并没有真正继承Thread类

那么问题又来了，既然ActivityThread不是一个线程，那么ActivityThread中Looper绑定的是哪个Thread，也可以说它的动力是什么?

3. ActivityThread 的动力是什么?

进程

每个app运行时前首先创建一个进程，该进程是由Zygote fork出来的，用于承载App上运行的各种Activity/Service等组件。进程对于上层应用来说是完全透明的，这也是google有意为之，让App程序都是运行在Android Runtime。大多数情况一个App就运行在一个进程中，除非在AndroidManifest.xml中配置Android:process属性，或通过native代码fork进程。

线程

线程对应用来说非常常见，比如每次new Thread().start都会创建一个新的线程。该线程与App所在进程之间资源共享，从Linux角度来说进程与线程除了是否共享资源外，并没有本质的区别，都是一个task_struct结构体，在CPU看来进程或线程无非就是一段可执行的代码，CPU采用CFS调度算法，保证每个task都尽可能公平的享有CPU时间片。

> 其实承载ActivityThread的主线程就是由Zygote fork而创建的进程。

4. Handler 是如何能够线程切换

其实看完上面我们大致也清楚，线程间是共享资源的。所以Handler处理不同线程问题就只要注意异步情况即可。

这里再引申出Handler的一些小知识点。

Handler创建的时候会采用当前线程的Looper来构造消息循环系统，Looper在哪个线程创建，就跟哪个线程绑定，并且Handler是在他关联的Looper对应的线程中处理消息的。（敲黑板）

那么Handler内部如何获取到当前线程的Looper呢—–ThreadLocal。ThreadLocal可以在不同的线程中互不干扰的存储并提供数据，通过ThreadLocal可以轻松获取每个线程的Looper。当然需要注意的是①线程是默认没有Looper的，如果需要使用Handler，就必须为线程创建Looper。我们经常提到的主线程，也叫UI线程，它就是ActivityThread，②ActivityThread被创建时就会初始化Looper，这也是在主线

程中默认可以使用Handler的原因。

系统为什么不允许在子线程中访问UI？（摘自《Android开发艺术探索》）

这是因为Android的UI控件不是线程安全的，如果在多线程中并发访问可能会导致UI控件处于不可预期的状态，那么为什么系统不对UI控件的访问加上锁机制呢？缺点有两个：

1. 首先加上锁机制会让UI访问的逻辑变得复杂
2. 锁机制会降低UI访问的效率，因为锁机制会阻塞某些线程的执行。

所以最简单且高效的方法就是采用单线程模型来处理UI操作。

5. 子线程有哪些更新UI的方法。

1. 主线程中定义Handler，子线程通过mHandler发送消息，主线程Handler的`handleMessage`更新UI。
2. 用Activity对象的runOnUiThread方法。
3. 创建Handler，传入`getMainLooper`。
4. View.post(Runnable r) 。

runOnUiThread

第一种咱们就不分析了，我们来看看第二种比较常用的写法。

先重新温习一下上面说的

> Looper在哪个线程创建，就跟哪个线程绑定，并且Handler是在他关联的Looper对应的线程中处理消息的。（敲黑板）

```
new Thread(new Runnable() {
    @Override
                public void run() {
        runOnUiThread(new Runnable() {
            @Override
                                public void run() {
                //DO UI method
            }
        }
        );
    }
}
).start();

//Activity
final Handler mHandler = new Handler();
public final void runOnUiThread(Runnable action) {
    if (Thread.currentThread() != mUiThread) {
        mHandler.post(action);
        //子线程（非UI线程）
    } else {
        action.run();
    }
}
```

进入Activity类里面，可以看到如果是在子线程中，通过`mHandler`发送的更新UI消息。

而这个Handler是在Activity中创建的，也就是说在主线程中创建，所以便和我们在主线程中使用Handler更新UI没有差别。

因为这个Looper，就是ActivityThread中创建的Looper（`Looper.prepareMainLooper()`）。

创建Handler，传入`getMainLooper`

那么同理，我们在子线程中，是否也可以创建一个Handler，并获取`MainLooper`，从而在子线程中更新UI呢?

首先我们看到，在`Looper`类中有静态对象`sMainLooper`，并且这个`sMainLooper`就是在ActivityThread中创建的`MainLooper`。

```
private static Looper sMainLooper;
// guarded by Looper.class
public static void prepareMainLooper() {
    prepare(false);
    synchronized (Looper.class) {
        if (sMainLooper != null) {
            throw new IllegalStateException("The main Looper has already been
prepared.");
        }
        sMainLooper = myLooper();
    }
}
```

所以不用多说，我们就可以通过这个`sMainLooper`来进行更新UI操作。

```
new Thread(new Runnable() {
    @Override
                public void run() {
        Log.e("qdx", "step 1 "+Thread.currentThread().getName());
        Handler handler=new Handler(getMainLooper());
        handler.post(new Runnable() {
            @Override
                                public void run() {
                //Do Ui method
                Log.e("qdx", "step 2 "+Thread.currentThread().getName());
            }
        }
        );
    }
}
).start();
```

```
E/qdx: step 1 Thread-113
E/qdx: step 2 main
```

View.post(Runnable r)

老样子，我们点入源码

```
//View
/**
     * <p>Causes the Runnable to be added to the message queue.
     * The runnable will be run on the user interface thread.</p>
```

```
     *
     * @param action The Runnable that will be executed.
     *
     * @return Returns true if the Runnable was successfully placed in to the
     *         message queue.  Returns false on failure, usually because the
     *         looper processing the message queue is exiting.
     *
     */
public Boolean post(Runnable action) {
    final AttachInfo attachInfo = mAttachInfo;
    if (attachInfo != null) {
        return attachInfo.mHandler.post(action);
        //一般情况走这里
    }
    // Postpone the runnable until we know on which thread it needs to run.
    // Assume that the runnable will be successfully placed after attach.
    getRunQueue().post(action);
    return true;
}
/**
         * A Handler supplied by a view's {@link android.view.ViewRootImpl}.
This
         * handler can be used to pump events in the UI events queue.
         */
final Handler mHandler;
```

居然也是Handler从中作祟，根据Handler的注释，也可以清楚该Handler可以处理UI事件，也就是说它的Looper也是主线程的`sMainLooper`。这就是说我们常用的更新UI都是通过Handler实现的。

另外更新UI 也可以通过`AsyncTask`来实现，难道这个`AsyncTask`的线程切换也是通过 Handler 吗?

没错，也是通过Handler.....

`Handler`实在是牛

6.子线程中Toast，showDialog，的方法。

可能有些人看到这个问题，就会想： 子线程本来就不可以更新UI的啊

而且上面也说了更新UI的方法

兄台且慢，且听我把话写完

```
new Thread(new Runnable() {
    @Override
                public void run() {
        Toast.makeText(MainActivity.this, "run on thread",
Toast.LENGTH_SHORT).show();
        //崩溃无疑
    }
}
).start();
```

```
E/AndroidRuntime: FATAL EXCEPTION: Thread-96
                  Process: qdx.aidlclient, PID: 6653
                  java.lang.RuntimeException: Can't create handler inside thread that has not called Looper.prepare()
                      at android.os.Handler.<init>(Handler.java:200)
                      at android.os.Handler.<init>(Handler.java:114)
                      at android.widget.Toast$TN.<init>(Toast.java:327)
                      at android.widget.Toast.<init>(Toast.java:92)
                      at android.widget.Toast.makeText(Toast.java:241)
                      at qdx.aidlclient.MainActivity$3.run(MainActivity.java:89)
                      at java.lang.Thread.run(Thread.java:841)
```

看到这个崩溃日志，是否有些疑惑，因为一般如果子线程不能更新UI控件是会报如下错误的（子线程不能更新UI）

```
E/AndroidRuntime: FATAL EXCEPTION: Thread-99
                  Process: qdx.aidlclient, PID: 11129
                  android.view.ViewRootImpl$CalledFromWrongThreadException: Only the original thread that created a view hierarchy can touch its views.
                      at android.view.ViewRootImpl.checkThread(ViewRootImpl.java:6118)
                      at android.view.ViewRootImpl.invalidateChildInParent(ViewRootImpl.java:881)
                      at android.view.ViewGroup.invalidateChild(ViewGroup.java:4320)
                      at android.view.View.invalidate(View.java:10935)
                      at android.view.View.invalidate(View.java:10890)
                      at android.widget.TextView.checkForRelayout(TextView.java:6587)
                      at android.widget.TextView.setText(TextView.java:3813)
                      at android.widget.TextView.setText(TextView.java:3671)
                      at android.widget.TextView.setText(TextView.java:3646)
                      at qdx.aidlclient.MainActivity$3.run(MainActivity.java:89)
                      at java.lang.Thread.run(Thread.java:841)
```

所以子线程不能更新Toast的原因就和Handler有关了，据我们了解，每一个Handler都要有对应的Looper对象，那么。
满足你。

```
new Thread(new Runnable() {
    @Override
                public void run() {
        Looper.prepare();
        Toast.makeText(MainActivity.this, "run on thread",
Toast.LENGTH_SHORT).show();
        Looper.loop();
    }
}
).start();
```

这样便能在子线程中Toast，不是说子线程...?

老样子，我们追根到底看一下Toast内部执行方式。

```
//Toast


    /**
     * Show the view for the specified duration.
     */
    public void show() {
``

        INotificationManager service = getService();//从SMgr中获取名为notification
的服务
        String pkg = mContext.getOpPackageName();
        TN tn = mTN;
        tn.mNextView = mNextView;

        try {
            service.enqueueToast(pkg, tn, mDuration);//enqueue? 难不成和Handler的队
列有关?
        } catch (RemoteException e) {
            // Empty
        }
```

```
    }
```

在`show`方法中，我们看到Toast的show方法和普通UI 控件不太一样，并且也是通过Binder进程间通讯方法执行Toast绘制。这其中的过程就不在多讨论了，有兴趣的可以在`NotificationManagerService`类中分析。

现在把目光放在`TN`这个类上（难道越重要的类命名就越简洁，如`H`类），通过`TN`类，可以了解到它是Binder的本地类。在Toast的show方法中，将这个`TN`对象传给`NotificationManagerService`就是为了通讯！并且我们也在TN中发现了它的show方法。

```
private static class TN extends ITransientNotification.Stub {
    //Binder服务端的具体实现类
    /**
         * schedule handleShow into the right thread
         */
    @Override
            public void show(IBinder windowToken) {
        mHandler.obtainMessage(0, windowToken).sendToTarget();
    }
    final Handler mHandler = new Handler() {
        @Override
                    public void handleMessage(Message msg) {
            IBinder token = (IBinder) msg.obj;
            handleShow(token);
        }
    }
    ;
}
```

看完上面代码，就知道子线程中Toast报错的原因，因为在`TN`中使用Handler，所以需要创建`Looper`对象。

那么既然用Handler来发送消息，就可以在`handleMessage`中找到更新Toast的方法。

在`handleMessage`看到由`handleShow`处理。

```
//Toast的TN类
public void handleShow(IBinder windowToken) {
``
                    mWM =
(WindowManager)context.getSystemService(Context.WINDOW_SERVICE);
    mParams.x = mX;
    mParams.y = mY;
    mParams.verticalMargin = mVerticalMargin;
    mParams.horizontalMargin = mHorizontalMargin;
    mParams.packageName = packageName;
    mParams.hideTimeoutMilliseconds = mDuration ==
                        Toast.LENGTH_lONG ? LONG_DURATION_TIMEOUT :
SHORT_DURATION_TIMEOUT;
    mParams.token = windowToken;
    if (mView.getParent() != null) {
        mWM.removeView(mView);
    }
    mWM.addView(mView, mParams);
    //使用WindowManager的addView方法
    trySendAccessibilityEvent();
```

```
}
}
```

看到这里就可以总结一下：

`Toast`本质是通过`window`显示和绘制的（操作的是`window`），而主线程不能更新UI 是因为`ViewRootImpl`的`checkThread`方法在Activity维护的View树的行为。

`Toast`中`TN`类使用`Handler`是为了用队列和时间控制排队显示`Toast`，所以为了防止在创建`TN`时抛出异常，需要在子线程中使用`Looper.prepare();`和`Looper.loop();`（但是不建议这么做，因为它会使线程无法执行结束，导致内存泄露）

Dialog亦是如此。同时我们又多了一个知识点要去研究：Android 中Window是什么，它内部有什么机制?

7. 如何处理Handler 使用不当导致的内存泄露?

首先上文在子线程中为了节目效果，使用如下方式创建Looper

```
Looper.prepare();
``
                Looper.loop();
```

实际上这是非常危险的一种做法

> 在子线程中，如果手动为其创建Looper，那么在所有的事情完成以后应该调用quit方法来终止消息循环，否则这个子线程就会一直处于等待的状态，而如果退出Looper以后，这个线程就会立刻终止，因此建议不需要的时候终止Looper。(【 Looper.myLooper().quit(); 】)

那么，如果在Handler的`handleMessage`方法中（或者是run方法）处理消息，如果这个是一个延时消息，会一直保存在主线程的消息队列里，并且会影响系统对Activity的回收，造成内存泄露。

总结一下，解决Handler内存泄露主要2点

1. 有延时消息，要在Activity销毁的时候移除`Messages`
2. 匿名内部类 导致的泄露改为匿名静态内部类，并且对上下文或者Activity使用弱引用。

总结

想不到Handler居然可以腾出这么多浪花，与此同时感谢前辈的摸索。

另外Handler还有许多不为人知的秘密，等待大家探索，下面我再简单的介绍两分钟

- HandlerThread
- IdleHandler

HandlerThread

> HandlerThread继承Thread，它是一种可以使用Handler的Thread，它的实现也很简单，在run方法中也是通过`Looper.prepare()`来创建消息队列，并通过`Looper.loop()`来开启消息循环（与我们手动创建方法基本一致），这样在实际的使用中就允许在HandlerThread中创建Handler了。
>
> 由于HandlerThread的run方法是一个无限循环，因此当不需要使用的时候通过quit或者quitSafely方法来终止线程的执行。

HandlerThread的本质也是线程，所以切记关联的Handler中处理消息的`handleMessage`为子线程。

IdleHandler

```
/**
     * Callback interface for discovering when a thread is going to block
     * waiting for more messages.
     */
public static interface IdleHandler {
    /**
         * Called when the message queue has run out of messages and will now
         * wait for more.  Return true to keep your idle handler active, false
         * to have it removed.  This may be called if there are still messages
         * pending in the queue, but they are all scheduled to be dispatched
         * after the current time.
         */
    Boolean queueIdle();
}
```

根据注释可以了解到，这个接口方法是在消息队列全部处理完成后或者是在阻塞的过程中等待更多的消息的时候调用的，返回值false表示只回调一次，true表示可以接收多次回调。

具体使用如下代码

```
Looper.myQueue().addIdleHandler(new MessageQueue.IdleHandler() {
    @Override
                public Boolean queueIdle() {
        return false;
    }
}
);
```

另外提供一个小技巧：在HandlerThread中获取Looper的MessageQueue方法之反射。

因为

1. Looper.myQueue()如果在主线程调用就会使用主线程looper
2. 使用handlerThread.getLooper().getQueue()最低版本需要23

```
//HandlerThread中获取MessageQueue

            Field field = Looper.class.getDeclaredField("mQueue");
            field.setAccessible(true);
            MessageQueue queue = (MessageQueue)
field.get(handlerThread.getLooper());
```

那么Android的消息循环机制是通过Handler，是否可以通过`IdleHandler`来判断Activity的加载和绘制情况(measure,layout,draw等)呢？并且`IdleHandler`是否也隐藏着不为人知的特殊功能？