|  |
| --- |
| 1、Handler用于解决线程间的通信： 在后台线程进行耗时处理后（不能在子线程进行更新UI），得到的结果要发送到UI线程进行更新，就需要使用Handler。  不但是开发者使用Handler用来与UI线程的通信，在android系统中，子线程与UI线程的交互一样是使用Handler机制的。  还可以延伸，为什么只能UI线程更新UI。  如果多个线程同时对同一个UI控件进行更新，容易发生不可控的错误！  那么怎么解决这种线程安全问题？  最简单的，加锁，不是加一个，是每层都要加锁（用户代码→GUI顶层→GUI底层…），这样也以为着 耗时，UI更新效率变低；如果每层共用同一把锁，那就是单线程了。  Android没有采用「线程锁」，而是采用「单线程消息队列机制」，实现一个「伪锁」。  2、（1）在UI线程创建一个Handler对象，需要重写handleMessage(Message msg)方法；  （2）在子线程中，通过这个handler对象发送message：sendMessage(msg);  （3）需要传递的数据给Message：  可以post(Runnable),其实就是使用下面这个方法；  sendMessage(Message)；  通过obtainMessage()获取Message； //一个消息池，复用对象。（享元模式）  ......  3、UI线程出初始化时创建了Looper和Message，  而，子线程中需要自己来初始化：  Looper.prepare();  //....  Looper.loop();  4、使用Handler导致内存泄露的解决方法  方法一：通过程序逻辑来进行保护。  1.退出Activity的时候停掉你的后台线程。  2.如果你的Handler是被delay的Message持有了引用，那么使用相应的Handler的removeCallbacks()方法，把消息对象从消息队列移除就行了。  方法二：将Handler声明为静态类 + 弱引用。  弱引用就是在JVM进行GC时，发现这个弱引用对象，就会把它回收。  实际上，方法二，子线程还是可能会在运行，只是任务结束后，获取Activity时，发现被回收了。  所以，最好的办法还是既然你退出了，就应该将子线程的任务结束。  方法二只能算是个兜底策略。  5、Android在子线程更新UI的最常见的五种方式  （1）runOnUiThread(new Runnable());  （2）handler.post(new Runnable());  （3）handler.sendMessage();  （4）view.post(new Runnable());  （5）view.postDelayed(new Runnable(), long);  6、注意的API  //避免内存泄露的方法：  //移除标记为0x1的消息  new Handler().removeMessages(0x1);  //移除回调的消息  new Handler().removeCallbacks(Runnable);  //移除回调和所有message  new Handler().removeCallbacksAndMessages(null);  7、Handler机制解析  Handler 负责消息创建、发送消息和处理消息  Message 是消息的实体。  MessageQueue 消息队列。  Looper 负责消息队列的循环，包括两件事：第一创建和控制 MessageQueue；第二轮询MessageQueue读取Message信息派发给Handler  （1）Looper对象不需要开发人员去初始化，在每个线程里面他是存在的。通过Looper.prepare()将线程与Looper绑定，也和MessageQueue消息队列一对一绑定了。  （2）而在Handler初始化时，和自己所在的线程的MessageQueue绑定。  所以，Handler对象可以跨线程，它在子线程中将Message推入MessageQueue中。  （3）Looper发现MessageQueue有Message，于是获取该Message相应的Handler，并将Messager给Handler处理。  其实，在handler对象所在的线程，运行在loop()方法中，这是一个无限的死循环，从MessageQueue里面读取Message，如果消息暂时不被读取会被阻塞。  //如果消息队列为空，就是执行被阻塞，主线程会释放CPU资源进入休眠状态；当有下一个消息达到是时候，会唤醒主线程开始工作。 |

# 1.handler.post()与sendMessage()

我们都知道Handler中的post方法，并且也是经常使用它

handler.post(new Runnable(){

@Override

public void run() {

//do something

}});

用它可以更新一个组件的内容，我们也知道Hanlder中也有一handler.sendMessage(

Message msg)方法，这两个方法有什么区别呢？先看一下Message类中定义一个私有的变量：Runnable callback;

再来看一下handler.post(Runnable callback)方法的源码:

public final boolean post(Runnable r) {

return sendMessageDelayed(getPostMessage(r), 0);

}

再看一下sendMessageDelayed的源码:

public final boolean sendMessageDelayed(Message msg, long delayMillis)

{

if (delayMillis < 0) {

delayMillis = 0;

}

return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);

}

这里面有个关键就是方法getPostMessage(r)这个方法，他将Runnable转成一个Message，他内部到底干了什么呢？看一下他的源码:

private final Message getPostMessage(Runnable r) {

Message m = Message.obtain();

m.callback = r;

return m;

}

这里面就是将Runnable转化成一个Message，其他看他的代码很简单，就是先获取一个空消息Message.obtain(),然后将Message中的callback的值设置成Runnable，这时候就了解到了Message中的callback的作用了！

同时也了解一下View.post(Runnable r)方法的作用：看一下实例代码:

final Button btn = (Button)findViewById(R.id.btn);

btn.post(new Runnable(){

@Override

public void run() {

btn.setText("不是好人");

}

});

}

上面的代码就是更新btn中的内容，同样下面的代码也可以达到这种效果:

Handler handler = new Handler();

final Button btn = (Button)findViewById(R.id.btn);

handler.post(new Runnable(){

@Override

public void run() {

btn.setText("不是好人");

}

});

}

不同是这个是用handler.post方法，一个是用View.post方法，现在来看一下View.post方法的源代码:

public boolean post(Runnable action) {

Handler handler;

AttachInfo attachInfo = mAttachInfo;

if (attachInfo != null) {

handler = attachInfo.mHandler;

} else {

// Assume that post will succeed later

ViewRootImpl.getRunQueue().post(action);

return true;

}

return handler.post(action);

}

方法中主要的功能代码就是attachInfo.mHandler,获取当前线程的hanlder，和我们在一个线程中定义一个Handler的效果是一样的。

# 什么是handler？

handler是Android给我们提供用来更新UI的一套机制，也是一套消息处理机制，我们可以发消息，也可以通过它处理消息。

那为什么要用handler呢?我能不能不用？

当我们需要在子线程处理耗时的操作（例如访问网络，数据库的操作），而当耗时的操作完成后，需要更新UI，这就需要使用Handler来处理，因为子线程不能做更新UI的操作。Handler能帮我们很容易的把任务（在子线程处理）切换回它所在的线程。简单理解，Handler就是解决线程和线程之间的通信的。

Android为什么要设计只能用handler机制更新UI呢？

答：最根本的目的就是为了解决多线程并发的问题！

打个比方，如果在一个activity中有多个线程，并且没有加锁，就会出现界面错乱的问题。但是如果对这些更新UI的操作都加锁处理，又会导致性能下降。

      处于对性能的问题考虑，Android给我们提供这一套更新UI的机制我们只需要遵循这种机制就行了。不用再去关系多线程的问题，所有的更新UI的操作，都是在主线程的消息队列中去轮训的。

# Android在子线程更新UI的最常见的五种方式

1. runOnUiThread(new Runnable());
2. handler.post(new Runnable());
3. handler.sendMessage();
4. view.post(new Runnable());
5. view.postDelayed(new Runnable(), long);

# 消息的流转的架构：

•Handler 负责发送消息和处理消息

发送消息，将消息加入到消息队列；

处理消息，根据消息体，进行相应handler处理。

•Message 是消息的实体。

•MessageQueue 消息队列。

•Looper 负责消息队列的循环，包括两件事：第一创建和控制 MessageQueue；第二轮询MessageQueue读取Message信息派发给Handler

## 4.1 先说Looper，MessageQueue，Handler3者关联的思路：

主线程-->prepareMainLooper()（内部调用prepare() ，去实例化Looper，Looper实例化同时创建了messagequeue，11对应关系）-->主线程中的handler获取当前线程的Looper-->3者关联。

|  |
| --- |
| 插播主线程ActivityThread：  public static void main(String[] args) {  ......  Looper.prepareMainLooper();  ......  //区别:子线程是创建handler;  //主线程是通过getHandler()获取内部类实例  if(sMainThreadHandler==null){  sMainThreadHandler=thread.getHandler();  }  ......  }  private class H extends Handler{ .  .....  } |

子线程-->直接通过Looper.prepare()去实例化Looper，Looper实例化同时创建了messagequeue(11对应关系) -->实例化Handler同时获取当前子线程的Looper-->3者关联。

每一个线程都有自己的一个Looper，而每个Looper都有一个MessageQueue。

## 4.2 Looper.prepare();

Handler对象可以跨线程，它可以在子线程中将Message推入MessageQueue中。

　在Handler初始化时，会将自己所在的线程的Looper的MessageQueue绑定与自己绑定。

所以，之后发送的消息，也就是推入到绑定的MessageQueue。

|  |
| --- |
| public Handler() {  this(null, false);  }  public Handler(Callback callback, boolean async) {  ..............  //获取Looper对象  mLooper = Looper.myLooper();  if (mLooper == null) {  throw new RuntimeException(  "Can't create handler inside thread that has not called Looper.prepare()");  }  //绑定MessageQueue对象  mQueue = mLooper.mQueue;  mCallback = callback;  mAsynchronous = async;  } |

## 4.3 Looper.loop()

再次，Looper发现MessageQueue有Message，于是获取该Message相应的Handler，并将Messager给Handler处理。

Looper又是如何发现MessageQueue里面的Message，并且分配给指定的Handler？答案是通过Looper.loop()方法。

|  |
| --- |
| public static void loop() {  final Looper me = myLooper();  if (me == null) {  throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");  }  //获取到Looper对象的MessageQueued对象。  final MessageQueue queue = me.mQueue;  .............  //开始无限循环  for (;;) {  //从MessageQueue里面读取Message，如果消息暂时不被读取会被阻塞。  Message msg = queue.next();  if (msg == null) {  // No message indicates that the message queue is quitting.  //消息为空退出。  return;  }  ............  //此处给Message的target（也就是对应的Handler）指派消息。  msg.target.dispatchMessage(msg);  ............  //消息被回收  msg.recycle();  }  } |

最后，Handler处理该Messager

　　接上面的源码可知，最后Message又被它的发送者Handler进行处理，调用的方法是dispatchMessage(msg),该方法源码实现如下：

|  |
| --- |
| public void dispatchMessage(Message msg) {  if (msg.callback != null) {  handleCallback(msg);  } else {  if (mCallback != null) {  if (mCallback.handleMessage(msg)) {  return;  }  }  //该方法会被重写，从而实现自定义的UI改动  handleMessage(msg);  }  } |

## 4.4 小结

Handler通过sendMessage()方法发送Message到MessageQueue队列

当前Thread中Looper通过调用loop()，不断取出达到触发条件的Message，通过对应target（Handler）的dispatchMessage()方法,将Message交给Handler的handleMessage()方法来处理。

一个线程对应一个Looper，一个Looper对应一个MessageQueue，一个MessageQueue可以对用多个Message。但是一个Message只能让一个handler来处理（就是Message中target所指定的handler）。

# obtainMessage()

Message对象本身存在于一个消息池中。如果消息池中有消息，建议不要使用new的方式产生对象应该复用该对象。

Message message = myHandler.obtainMessage();

message.arg1 = 0;

myHandler.sendMessage(message);

# handler作用:

## 6.1 传递消息Message

//2种创建消息方法

//1.通过handler实例获取

Handler handler = new Handler();

Message message=handler.obtainMessage();

//2.通过Message获取Message message=Message.obtain();

//源码中第一种获取方式其实也是内部调用了第二种：

public final Message obtainMessage(){

return Message.obtain(this);

}

不建议直接new Message，Message内部保存了一个缓存的消息池，我们可以用obtain从缓存池获得一个消息，Message使用完后系统会调用recycle回收，如果自己new很多Message，每次使用完后系统放入缓存池，会占用很多内存的。

//传递的数据

Bundle bundle = new Bundle();

bundle.putString("msg", "传递我这个消息");

//发送数据

Message message = Message.obtain();

message.setData(bundle);

//message.obj=bundle //传值也行

message.what = 0x11;

handler.sendMessage(message);

//数据的接收

final Handler handler = new Handler() {

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

if (msg.what == 0x11) {

Bundle bundle = msg.getData();

String date = bundle.getString("msg");

}

}

};

## 6.2 常用api

//消息

Message message = Message.obtain();

//发送消息

new Handler().sendMessage(message);

//延时1s发送消息

new Handler().sendMessageDelayed(message, 1000);

//发送带标记的消息(内部创建了message,并设置msg.what = 0x1)

new Handler().sendEmptyMessage(0x1);

//延时1s发送带标记的消息

new Handler().sendEmptyMessageDelayed(0x1, 1000);

//延时1秒发送消息（第二个参数为：相对系统开机时间的绝对时间，而

// SystemClock.uptimeMillis()是当前开机时间）

new Handler().sendMessageAtTime(message, SystemClock.uptimeMillis() + 1000);

//避免内存泄露的方法：

//移除标记为0x1的消息

new Handler().removeMessages(0x1);

//移除回调的消息

new Handler().removeCallbacks(Runnable);

//移除回调和所有message

new Handler().removeCallbacksAndMessages(null);

# 7、什么说应用所有的操作都是在loop()中来管理？

首先，我们的每一个应用都存在于自己的虚拟机中，也就是说每一个应用都有自己的一个main函数，这个main函数就是ActivityThread.java的main()函数。

当我们在launcher界面启动一个应用的时候，这时候，系统就会用zygote给我们分配一个虚拟机，然后，这个应用就会运行在这个虚拟机上面。

应用运行到虚拟机之后，首先它要执行的就是启动ActivityThread，在ActivityThread中，它又会启动它的main()函数。

在main()函数中，它最重要的两行代码：

public static void main(String[] args) {

...

Looper.prepareMainLooper();

...

Looper.loop();

}

所以在程序运行的时候，主线程所有的代码都运行在这个Looper里面。

也就是说应用所有生命周期的函数（包括Activity、Service所有生命周期）都运行在这个Looper里面，而且，它们都是以消息的方式连接的。

都是通过发送不同的Message来通知进行不同的UI操作。

# 8、那为什么不会造成应用卡死？

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。而对于主线程肯定不能运行一段时间后就自动结束了，那么如何保证一直存活呢？简单的做法就是可执行代码能一直执行下去，死循环便能保证不会被退出，例如：binder线程也是采用死循环方法，通过循环方式不同与Binder驱动进行读写操作，当然并非简单的死循环，无消息时会休眠，但是死循环又如何处理其他事物呢？通过创建新的线程。真正卡死主线程操作的是在回调方法onCreate、onStart、onResume等操作时间过长，会导致掉帧甚至ANR，Looper.loop()本身不会导致应用卡死。

主线程的死循环一直运行会不会特别消耗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。