# 一．Handler

## 1.Handler的原理是什么？

一、Handler封装了消息的发送（主要包括消息发送给谁，以及如何去发送。默认情况下消息都是发送给Handler自己）。

二、Looper：轮询，消息封装的一个载体，

1、Looper内部包含一个消息队列也就是MessageQueue，所有的Handler发送的消息都走向这个消息队列；

2、Looper.looper方法，就是一个死循环，不断地从MessageQueue取消息，如有消息就处理，没有就阻塞；

三、MessageQueue，就是一个消息队列，可以添加消息，并处理消息。

四、Handler也很简单，内部会跟Looper进行关联，也就是说在Handler的内部可以找到Looper，找到了Looper也就找到了MessageQueue，在Handler中发送消息，其实就是向MessageQueue队列中发送消息。

总结：Handler负责发送消息，Looper负责接收Handler发送的消息，并直接把消息回传给Handler自己，MessageQueue就是一个存储消息的容器。

## 2.为什么系统不建议在子线程访问UI？（为什么不能在子线程更新UI？）

答：Android的UI访问是没有加锁的，多个线程可以同时访问更新操作同一个UI控件。也就是说访问UI的时候，android系统当中的控件都不是线程安全的，这将导致在多线程模式下，当多个线程共同访问更新操作同一个UI控件时容易发生不可控的错误，而这是致命的。所以Android中规定只能在UI线程中访问UI，这相当于从另一个角度给Android的UI访问加上锁，一个伪锁。

## 3.一个Thread可以有几个Looper？几个Handler？

答：只能有一个Looper，不然调用Looper.prepare()会抛出运行时异常，提示“Only one Looper may be created per thread”

可以创建无数个Handler，但是他们使用的消息队列都是同一个，也就是同一个Looper

## 4.Handler实现子线程与子线程、主线程之间通信

### 一、子线程向主线程传值：

因为主线程自带Looper机制，所有我们不用创建Looper了，首先在主线程里创建一个Handler，然后开启一个子线程，在子线程里直接使用Handler发送消息即可

Handler mHandler = new Handler(){

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

switch (msg.what) {

case 0:

//do something,refresh UI;

break;

default:

break;

}

}

};

然后开启一个子线程，在子线程里直接使用Handler发送消息即可

new Thread() {

public void run() {

Message message = new Message();

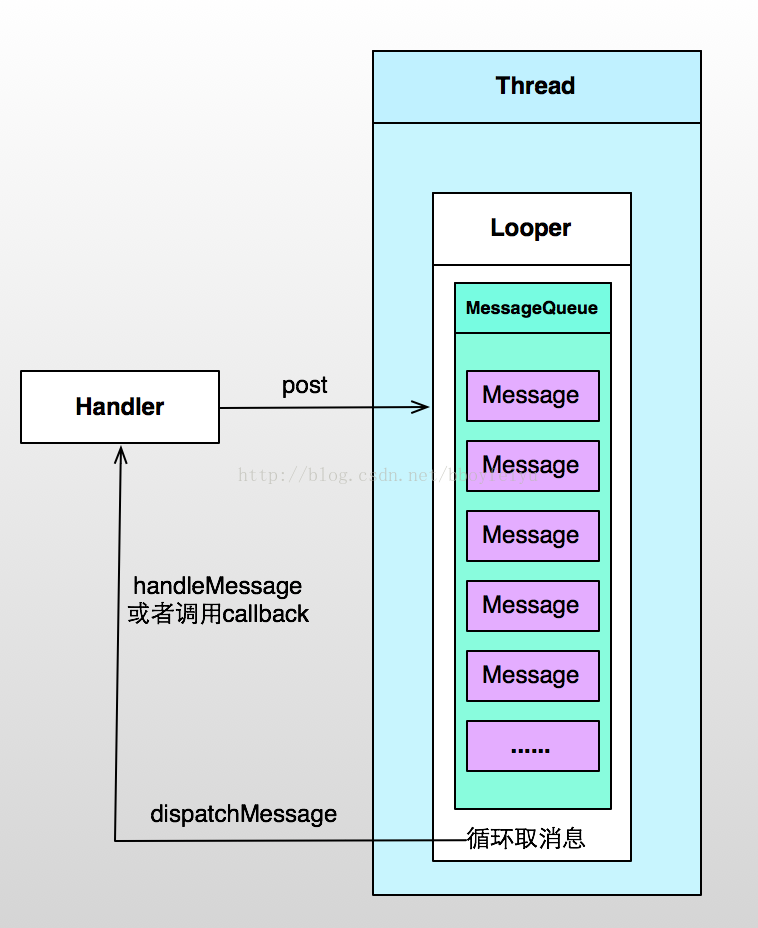
message.obj = "子线程发送的消息Hi~Hi";

mHandler .sendMessage(message);

};

}.start();

### 二、主线程向子线程里发送消息（HandlerThread）：



主线程向子线程发送消息的话，我们需要在子线程里初始化Looper，并在主线程里创建的Handler引用子线程的Looper（Handler中引用的是哪个线程的Looper，就在哪个线程里处理消息），下面看代码：

public class ThreadHandlerActivity extends Activity{

//创建子线程

class MyThread extends Thread{

private Looper looper;//取出该子线程的Looper

public void run() {

Looper.prepare();//创建该子线程的Looper

looper = Looper.myLooper();//取出该子线程的Looper

Looper.loop();//只要调用了该方法才能不断循环取出消息

}

}

private Handler mHandler;//将mHandler指定轮询的Looper

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

setContentView(R.layout.main);

thread = new MyThread();

thread.start();//千万别忘记开启这个线程

//下面是主线程发送消息

mHandler = new Handler(thread.looper){

public void handleMessage(android.os.Message msg) {

Log.d("当前子线程是----->",Thread.currentThread()+"");

};

};

mHandler.sendEmptyMessage(1);

}

}

其实这样就可以达到主线程向子线程发送消息了，然而当我们运行后发现程序会Crash掉，报了一个控制针，这是因为在Handler初始化的时候，thread.looper还没有初始化，所以会报控制针，这时我们可以让主线程等待一下子线程，也可以来一个while循环来判断thread.looper是否初始化完成。不过Android本身还提供了一个方法，那就是HandlerThread：

protected void onCreate(Bundle savedInstanceState) {

super.onCreate(savedInstanceState);

tv = new TextView(this);

tv.setText("Handler实验");

setContentView(tv);

//实例化一个特殊的线程HandlerThread，必须给其指定一个名字

HandlerThread thread = new HandlerThread("handler thread");

thread.start();//千万不要忘记开启这个线程

//将mHandler与thread相关联

mHandler = new Handler(thread.getLooper()){

public void handleMessage(android.os.Message msg) {

Log.d("当前子线程是----->", Thread.currentThread()+"");

};

};

mHandler.sendEmptyMessage(1);//发送消息

}

这时HandlerMessage所在的线程就是HandlerThread 的子线程。

然而HandlerThread 所创建处理的子线程里是不能重写Run（）方法的，你写了以后，会发现，HandlerMessage不执行了，这时因为HandlerMessage本身实现了Run（）方法，我们看一下内部实现：

@Override

public void run() {

mTid = Process.myTid();

Looper.prepare();

synchronized (this) {

mLooper = Looper.myLooper();

notifyAll();

}

Process.setThreadPriority(mPriority);

onLooperPrepared();

Looper.loop();

mTid = -1;

}

在源代码的第4行，进行了实例化自己的Looper，如果继续追踪源代码翻看其getLooper方法你会发现，如果一个Handler在与HandlerThread进行绑定时，发现Looper为空，Handler则会一直等待直到Looper被创建出来为止，然后才继续执行后续的代码。所以我们重写了HandlerThread的run方法，肯定就不会去创建Looper对象，那么绑定的Handler就会永远处于等待状态，自然而然就不会执行到HandlerMessage信息了。这也是为什么我们要使用HandlerThread这个特殊的线程，因为使用这个，我们不必关心多线程会混乱，Looper会为空等一系列问题，只要去关心我们要实现的逻辑就行了。

### 三、子线程和子线程之间通信：

其实子线程向子线程之间通信，其实就是在一个子线程中创建一个Handler，它的回调自然就在此子线程中，然后在另一个子线程中调用此handler来发送消息就可以了，不过记得写上Looper哦，下面看代码：

new Thread(new Runnable() {

@Override

public void run() {

String msg;

Looper.prepare();

childHandler = new Handler() {

@Override

public void handleMessage(Message msg) {

super.handleMessage(msg);

System.out.println("这个消息是从-->>" + msg.obj+ "过来的，在" + "btn的子线程当中" + "中执行的");

}

};

Looper.loop();//开始轮循

}

}).start();

其中 Looper.prepare()和Looper.loop()一定不要忘了写。

然后我们创建第二个子线程

new Thread(new Runnable() {

@Override

public void run() {

Looper loop = Looper.myLooper();

Message msg = childHandler.obtainMessage();

msg.obj = "btn2当中子线程";

childHandler.sendMessage(msg);

}

}).start();

Handler线程通信：

[**https://blog.csdn.net/androidsj/article/details/79816866**](https://blog.csdn.net/androidsj/article/details/79816866)

你真的懂Handler.postDelayed()的原理吗?

https://blog.csdn.net/qingtiantianqing/article/details/72783952

# 二 HandelThread

## HandlerThread是什么？

开启子线程进行耗时操作，多次创建和销毁子线程是很耗费资源的，谷歌考虑了这点为我们专门开发出了HandlerThread机制，与线程池注重并发不同，HandlerThread是一个串行队列，HandlerThread背后只有一个线程，它的特点如下：

本质：Handler + Thread + Looper，是一个Thread内部有Looper。

  a.HandlerThread本质上是一个线程类，它继承了Thread。

  b.HandlerThread有自己内部的Looper对象，可以进行Looper循环。

  c.通过获取HandlerThread的Looper对象传递给Handler对象，可以在handlerMessage方法中执行异步任务。

  d.优点是不会有堵塞，减少对性能的消耗，缺点是不能进行多任务的处理，需要等待进行处理，处理效率较低。

  e.与线程池注重并发不同，HandlerThread是一个串行队列，HandlerThread背后只有一个线程。

## 2. HandlerThread的使用

HandlerThread的使用就是为了主线程通知子线程，再由子线程通知主线程，这里面建立一套完整的通信体系，并不像传统的开启子线程，使用HandlerThread的好处就是主线程可以发送信息告诉子线程要干什么，而且可以一直发送信息，无论在主线程的何处，都可以发送合适的信息告诉子线程，它要干什么，子线程做完了信息中交代的事情，然后再通知主线程更新等操作。说白了HandlerThread提供了主线程向子线程的通信。

具体使用见4.2，源码分析见以下网址：

<https://blog.csdn.net/pgg_cold/article/details/79403793>

## 3.Handler和HandlerThread的异同

共同点：

都是用于多个线程间通信，通知其他线程某些任务已经处理完成。

区别：

Handler主要是子线程通知Android的UI线程（主线程）更新界面。

HandlerThread主要是Android的UI线程（主线程）通知子线程中的handler去做异步任务。

# 三 线程/线程池

## 1. 创建线程的三种方式及其对比

### 1.1 继承Thread类创建线程类

（1）定义Thread类的子类，并重写该类的run方法，该run方法的方法体就代表了线程要完成的任务。因此把run()方法称为执行体。

（2）创建Thread子类的实例，即创建了线程对象。

（3）调用线程对象的start()方法来启动该线程。

### 1.2 通过Runnable接口创建线程类

（1）定义runnable接口的实现类，并重写该接口的run()方法，该run()方法的方法体同样是该线程的线程执行体。

（2）创建 Runnable实现类的实例，并依此实例作为Thread的target来创建Thread对象，该Thread对象才是真正的线程对象。

（3）调用线程对象的start()方法来启动该线程。

### 1.3通过Callable和Future创建线程

（1）创建Callable接口的实现类，并实现call()方法，该call()方法将作为线程执行体，并且有返回值。

（2）创建Callable实现类的实例，使用FutureTask类来包装Callable对象，该FutureTask对象封装了该Callable对象的call()方法的返回值。

（3）使用FutureTask对象作为Thread对象的target创建并启动新线程。

（4）调用FutureTask对象的get()方法来获得子线程执行结束后的返回值

### 1.4创建线程的三种方式的对比

采用实现Runnable、Callable接口的方式创见多线程时，优势是：

线程类只是实现了Runnable接口或Callable接口，还可以继承其他类。

在这种方式下，多个线程可以共享同一个target对象，所以非常适合多个相同线程来处理同一份资源的情况，从而可以将CPU、代码和数据分开，形成清晰的模型，较好地体现了面向对象的思想。

劣势是：

编程稍微复杂，如果要访问当前线程，则必须使用Thread.currentThread()方法

使用继承Thread类的方式创建多线程时优势是：

编写简单，如果需要访问当前线程，则无需使用Thread.currentThread()方法，直接使用this即可获得当前线程。

劣势是：

线程类已经继承了Thread类，所以不能再继承其他父类。

## 2. 谈谈对wait和notify关键字的理解？

使用wait()、notify()和notifyAll()时需要首先对调用对象加锁

调用wait()方法后，线程状态会从RUNNING变为WAITING，并将当线程加入到lock对象的等待队列中

调用notify()或者notifyAll()方法后，等待在lock对象的等待队列的线程不会马上从wait()方法返回，必须要等到调用notify()或者notifyAll()方法的线程将lock锁释放，等待线程才有机会从等待队列返回。这里只是有机会，因为锁释放后，等待线程会出现竞争，只有竞争到该锁的线程才会从wait()方法返回，其他的线程只能继续等待

notify()方法将等待队列中的一个线程移到lock对象的同步队列，notifyAll()方法则是将等待队列中所有线程移到lock对象的同步队列，被移动的线程的状态由WAITING变为BLOCKED

wait()方法上等待锁，可以通过wait(long timeout)设置等待的超时时间

## 3.Synchronized关键字

类锁：在代码中的方法上加了static和synchronized的锁，或者synchronized(xxx.class）的代码段

2.对象锁：在代码中的方法上加了synchronized的锁，或者synchronized(this）的代码段

3.私有锁：在类内部声明一个私有属性如private Object lock，在需要加锁的代码段synchronized(lock）

对象锁：假设我有一个类ClassA，其中有一个方法synchronized methodA()，那么当这个方法被调用的时候你获得就是对象锁。

举例：ClassA a = new ClassA(); ClassA b = new ClassA(); 那么如果你在a这对象上调用了methodA，不会影响b这个对象，也就是说对于b这个对象，他也可以调用methodA，因为这是两对象，所以说对象锁是针对对象的

类锁，其实没有所谓的类锁，因为类锁实际上就是这个类的对象的对象锁

举例:我有一个类ClassA，其中有一个方法synchronized static methodA()，注意这个方法是静态的了，那就是说这个类的所有的对象都公用一个这个方法了，那如果你在这个类的某个对象上调用了这个方法，那么其他的对象如果想要用这个方法就得等着锁被释放，所以感觉就好像这个类被锁住了一样。

## 4.线程池

### 4.1 什么是线程池？

线程池是一种多线程处理形式，处理过程中将任务提交到线程池，任务的执行交由线程池来管理。

如果每个请求都创建一个线程去处理，那么服务器的资源很快就会被耗尽，使用线程池可以减少创建和销毁线程的次数，每个工作线程都可以被重复利用，可执行多个任务。

### 4.2 为什么要使用线程池？

创建线程和销毁线程的花销是比较大的，这些时间有可能比处理业务的时间还要长。这样频繁的创建线程和销毁线程，再加上业务工作线程，消耗系统资源的时间，可能导致系统资源不足。（我们可以把创建和销毁的线程的过程去掉）

### 4.3 线程池有什么作用？

线程池作用就是限制系统中执行线程的数量。

1、提高效率 创建好一定数量的线程放在池中，等需要使用的时候就从池中拿一个，这要比需要的时候创建一个线程对象要快的多。

2、方便管理 可以编写线程池管理代码对池中的线程同一进行管理，比如说启动时有该程序创建100个线程，每当有请求的时候，就分配一个线程去工作，如果刚好并发有101个请求，那多出的这一个请求可以排队等候，避免因无休止的创建线程导致系统崩溃。

### 4.4 说说几种常见的线程池及使用场景

1、newSingleThreadExecutor

创建一个单线程化的线程池，它只会用唯一的工作线程来执行任务，保证所有任务按照指定顺序(FIFO, LIFO, 优先级)执行。

2、newFixedThreadPool

创建一个定长线程池，可控制线程最大并发数，超出的线程会在队列中等待。

3、newCachedThreadPool

创建一个可缓存线程池，如果线程池长度超过处理需要，可灵活回收空闲线程，若无可回收，则新建线程。

4、newScheduledThreadPool

5、newWorkStealingPool：一个拥有多个任务队列的线程池，可以减少连接数，创建当前可用cpu数量的线程来并行执行。