理解 Activity.runOnUiThread

在开发 Android 应用的时候我们总是要记住应用主线程。

主线程非常繁忙，因为它要处理绘制UI，响应用户的交互，默认情况下执行我们写下的大部分代码。

好的开发者知道他/她需要将重负荷的任务移除到工作线程避免主线程阻塞，同时获得更流畅的用户体验，避免ANR的发生。

但是，当需要更新UI的时候我们需要“返回”到主线程，因为只有它才可以更新应用 UI。

最常用的方式是调用 Activity 的 `runOnUiThread()` 方法：

```
runOnUiThread(new Runnable() {
     void run() {
         // Do stuff…
     }
});
```

这样就可以神奇的将 Runnable 任务放到主线程中执行。

魔法是很棒。。。但是它存在与我们的应用源码之外。在本文中，我将尝试阐述 `runOnUiThread()` 中发生的一切，并且（希望）能够破解魔法。

破解魔法

我们一起来看看 Activity 源码中的相关部分：

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

看起来非常简单，首先我们检查当前运行的线程是否是主线线程。

如果是主线程--很棒！只需要调用 Runnable 的 `run ()` 方法。

但是如果不是主线程呢?
在这种情况下，我们会调用 `mHandler.post()` 并将我们的 Runnable 传递过去。所以究竟发生了什么事情?

在回答这个问题之前我们真的需要讨论一下一个称为 Looper 的东西。

一切都从 Looper 开始

当我们创建一个新的 Java 线程时，我们重写它的 `run()` 方法。一个简单的线程实现看起来应该是这样的：

```
public class MyThread extends Thread {

    @Override
    public void run() {
        // Do stuff...
    }
}
```

好好的看一下 `run()` 方法，当线程执行完该方法中所有的语句后，线程就完成了。结束了。没用了。

如我我们想重复使用一个线程（一个很好的理由就是避免新线程创建以及减少内存消耗）我们必须让它保持存活状态并且等待接收新的指令。一个常用的方式就是在线程的 `run()` 方法里创建一个循环：

```
public class MyThread extends Thread {

    private boolean running;

    @Override
    public void run() {
        while (running) {
            // Do stuff...
        }
    }
}
```

只要 while 循环还在执行（即 `run()` 方法还没有执行完毕）--这个线程就保持存活状态。

这就是 Looper 所做的事情：

Looper。。。就是 LOOPING，并保持它的线程处于存活状态

关于 Looper 以下几点值得注意：

- 线程默认没有 Looper
- 你可创建一个 Looper 并将它绑定到一个线程
- 每一个线程只能绑定一个 Looper

所以，我们将线程中的 while 循环用 Looper 实现来替换：

```
public class MyThread extends Thread {

    @Override
    public void run() {
        Looper.prepare();
        Looper.loop();
    }
}
```

真的很简单：

调用 `Lopper.prepare()` 是检查当前线程是否还没有绑定 Lopper（记住，每一个线程只能绑定一个 Looper），如果没有就创建一个 Looper 并和当前线程绑定。

调用 `Looper.loop()` 触发我们的 Looper 开始循环。

所以，现在 Looper 开始循环并保持线程处于存活状态，但是如果不能传递指令、任务或者其他事情让线程执行实际的任务，那么保持线程存活没有任何意义。。。

幸好，Looper 不仅仅是循环。
当我们创建 Looper 的时候，会一并创建一个工作队列。这个队列称为消息队列因为它持有消息（**Message**）对象。

消息是什么?

这些消息对象实际上就是一系列**指令**。
他们可以持有数据比如字符串、整数等，也可以只有任务比如 Runnerables。

所以，当一个消息进入线程的 Looper消息队列，并且轮到它（毕竟它是一个队列）的时候--消息指令就会在**队列所在的线程**执行。这意味着。。。。：

如果我们希望一个 Runnable 在指定的线程运行，我们只需要将它放到一个消息里，并将这个消息放到对应线程的 Looper 消息队列就可以了!

很棒！我们怎么实现呢?
很简单。我们使用 **Handler**。

Handler

Handler 干了所有的活。

它负责向 Looper 的队列添加消息，当轮到消息执行时，它负责在 Looper 所在的线程中执行同一条消息。

当一个 Handler 被创建的时候，会被指向一个指定的 Looper（即，指向一个指定的线程）

创建 Handler 有两种方法：

- 1、在构造函数中指定 Looper：
  `Handler handler = new Handler(Looper looper);`

  现在 handler指向了我们提供的Looper（实际上是 Looper 的消息队列）
- 2、使用空的构造函数：
  `Handler handler = new Handler();`

  当我们使用空构造函数的时候，Handler 会自动指向和当前线程绑定的 Looper。真方便!
  Handler 提供了很方便的方法用于创建消息并自动将它们添加到 Looper 消息队列。

例如，`post()` 方法就创建一条消息并将它添加到 Looper 队列的尾部。

如果我们希望消息持有一个任务（一个 Runnable），我们简单的将 Runnable 对象传递给 `post()` 方法就可以：

```
handler.post(new Runnable() {
  @Override
  public void run() {
      // Do stuff...
  }
});
```

看起来很熟悉?

再来看看 Activity 的源码

现在我们再仔细的看一看`runOnUiThread():`

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

首先，mHandler 是使用空构造函数创建。

记住：这段代码是在主线程中执行，这意味着 `mHandler` 指向主线程的 Looper。

是的，应用主线程是唯一一个默认绑定了 Looper 线程。

所以。。。当这一行代码执行的时候：

```
mHandler.post(action);
```

Handler 会创建一条持有我们传入的 Runnable 的消息，这条消息随后被添加到主线程的消息队列，然后等待 Handler 在它的Looper线程（**主线程**）中执行。