转载请注明链接：https://blog.csdn.net/feather_wch/article/details/79263855

详解Handler消息机制，包括Handler、MessageQueue、Looper和LocalThread。

本文是我一点点归纳总结的干货，但是难免有疏忽和遗漏，希望不吝赐教。

Handler消息机制详解(29题)

版本：2018/9/7-1(2359)

- Handler消息机制详解(29题)
  - 问题汇总
  - 消息机制概述(8)
  - ThreadLocal(4)
  - MessageQueue(3)
  - Looper(8)
    - 主线程的消息循环
  - Handler(6)
    - 内存泄漏

问题汇总

> 以问答形式将本文的所有内容进行汇总。考考你哟。

1. Handler是什么?
2. 消息机制是什么?
3. MeesageQueue是什么?
4. Looper是什么?
5. ThreadLocal是什么?
6. 为什么不能在子线程中访问UI?
7. Handler的运行机制概述(Handler的创建、Looper的创建、MessageQueue的创建、以及消息的接收和处理)
8. 主线程向子线程发送消息的方法?
9. ThreadLocal的作用?
10. ThreadLocal的应用场景
11. ThreadLocal的使用
12. ThreadLocal的set()源码分析
13. ThreadLocal的get()源码分析
14. MessageQueue的主要操作

15. MessageQueue的插入和读取源码分析
16. MessageQueue的next源码详解
17. Looper的构造中做了什么?
18. 子线程中如何创建Looper?
19. 主线程ActivityThread中的Looper是如何创建和获取的?
20. Looper如何退出?
21. quit和quitSafely的区别
22. Looper的loop()源码中的4个工作?
23. Android如何保证一个线程最多只能有一个Looper?
24. Looper的构造器为什么是private的?
25. Handler消息机制中，一个looper是如何区分多个Handler的?
26. 主线程ActivityThread的消息循环
27. ActivityThread中Handler H是做什么的?
28. Handler的post/send()逻辑流程
29. Handler的postDelayed逻辑流程
30. Handler的消息处理的流程?
31. Handler的特殊构造方法中，为什么有Callback接口?
32. Handler的内存泄漏如何避免?

消息机制概述(8)

1、Handler是什么?

> 1. Android消息机制的上层接口(从开发角度)
> 2. 能轻松将任务切换到Handler所在的线程中去执行
> 3. 主要目的在于解决在子线程中无法访问UI的矛盾

2、消息机制?

> 1. `Android`的消息机制 主要就是指 `Handler`的运行机制
> 2. `Handler` 的运行需要底层 `MessageQueue` 和 `Looper` 的支撑

3、MeesageQueue是什么?

> 1. 消息队列
> 2. 内部存储结构并不是真正的队列，而是单链表的数据结构来存储消息列表
> 3. 只能存储消息，而不能处理

4、Looper是什么?

> 1. 消息循环
> 2. `Looper` 以无限循环的形式去查找是否有新消息，有就处理消息，没有就一直等待着。

5、ThreadLocal是什么?

> 1. Looper 中一种特殊的概念
> 2. ThreadLocal 并不是线程，作用是可以在每个线程中互不干扰的 存储数据 和 提供数据 。
> 3. Handler 创建时会采用当前线程的 Looper 来构造消息循环系统， Handler 内部就是通过 ThreadLocal 来获取当前线程的 Looper 的
> 4. 线程默认是没有 Looper 的，如果需要使用 Handler 就必须为线程创建 Looper
> 5. UI线程 就是 ActivityThread ， 被创建时会初始化 Looper ， 因此 UI线程 中默认是可以使用 Handler 的

6、为什么不能在子线程中访问UI?

> ViewRootImpl会对UI操作进行验证，禁止在子线程中访问UI:

```
void checkThread(){
  if(mThread != Thread.currentThread()){
    throw new CalledFromWrongThreadException("Only th original thread that created a view hiera
  }
}
```

7、Handler的运行机制概述(Handler的创建、Looper的创建、MessageQueue的创建、以及消息的接收和处理)

> 1. Handler创建时会采用当前线程的Looper
> 2. 如果当前线程没有 Looper 就会报错，要么创建 Looper ， 要么在有 Looper 的线程中创建 Handler
> 3. Handler 的 post 方法会将一个 Runnable 投递到 Handler 内部的 Looper 中处理（本质也是通过 send 方法完成）
> 4. Handler 的 send 方法被调用时，会调用 MessageQueue 的 enqueueMessage 方法将消息放入消息队列, 然后 Looper 发现有新消息到来时，就会处理这个消息，最终消息中的 Runnable 或者 Handler 的 handleMessage 就会被调用
> 5. 因为 Looper 是运行在创建 Handler 所在的线程中的，所以通过 Handler 执行的业务逻辑就会被切换到 Looper 所在的线程中执行。

8、主线程向子线程发送消息的方法?

> 1. 通过在主线程调用子线程中Handler的post方法，完成消息的投递。
> 2. 通过 HandlerThread 实现该需求。

ThreadLocal(4)

1、ThreadLocal的作用

1. ThreadLocal 是线程内部的数据存储类，可以在指定线程中存储数据，之后只有在指定线程中才开业读取到存储的数据
2. 应用场景1：某些数据是以线程为作用域，并且不同线程具有不同的数据副本的时候。 ThreadLocal 可以轻松实现 Looper 在线程中的存取。
3. 应用场景2：在复杂逻辑下的对象传递，通过 ThreadLocal 可以让对象成为线程内的全局对象，线程内部通过 get 就可以获取。

2、ThreadLocal的使用

```
mBooleanThreadLocal.set(true);
Log.d("ThreadLocal", "[Thread#main]" + mBooleanThreadLocal.get());

new Thread("Thread#1"){
    @Override
    public void run(){
        mBooleanThreadLocal.set(false);
        Log.d("ThreadLocal", "[Thread#1]" + mBooleanThreadLocal.get());
    }
}.start();

new Thread("Thread#2"){
    @Override
    public void run(){
        Log.d("ThreadLocal", "[Thread#2]" + mBooleanThreadLocal.get());
    }
}.start();
```

1. 最终 main 中输出 true； Thread#1 中输出 false； Thread#2 中输出 null
2. ThreadLocal 内部会从各自线程中取出数组，再根据当前 ThreadLocal 的索引去查找出对应的 value 值。

3、ThreadLocal的set()源码分析

```
//ThreadLocal.java
public void set(T value) {
    //1. 获取当前线程
    Thread t = Thread.currentThread();
    //2. 获取当前线程对应的ThreadLocalMap
    ThreadLocalMap map = getMap(t);
    if (map != null)
        //3. map存在就进行存储
        map.set(this, value);
    else
        //4. 不存在就创建map并且存储
        createMap(t, value);
}
//ThreadLocal.java内部类: ThreadLocalMap
private void set(ThreadLocal<?> key, Object value) {
    //1. table为Entry数组
    Entry[] tab = table;
    int len = tab.length;
    int i = key.threadLocalHashCode & (len-1);
    //2. 根据当前ThreadLocal获取到Hash key，并以此从table中查询出Entry
    for (Entry e = tab[i]; e != null; e = tab[i = nextIndex(i, len)]) {
        ThreadLocal<?> k = e.get();
        //3. 如果Entry的ThreadLocal与当前的ThreadLocal相同，则用新值覆盖e的value
        if (k == key) {
            e.value = value;
            return;
        }
        //4. Entry没有ThreadLocal则把当前ThreadLocal置入，并存储value
        if (k == null) {
            replaceStaleEntry(key, value, i);
            return;
        }
    }
    //5. 没有查询到Entry，则新建Entry并且存储value
    tab[i] = new Entry(key, value);
    int sz = ++size;
    if (!cleanSomeSlots(i, sz) && sz >= threshold)
        rehash();
}
//ThreadLocal内部类ThreadLocalMap的静态内部类
static class Entry extends WeakReference<ThreadLocal<?>> {
    /** The value associated with this ThreadLocal. */
    Object value;
    Entry(ThreadLocal<?> k, Object v) {
        super(k);
        value = v;
    }
}
```

4、ThreadLocal的get()源码分析

```
public T get() {
    //1. 获取当前线程对应的ThreadLocalMap
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        //2. 取出map中的对应该ThreadLocal的Entry
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            //3. 获取到entry后返回其中的value
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    //4. 没有ThreadLocalMap或者没有获取到ThreadLocal对应的Entry，返回规定数值
    return setInitialValue();
}
private T setInitialValue() {
    //1. value = null
    T value = initialValue();//返回null
    Thread t = Thread.currentThread();
    //2. 若不存在则新ThreadLocalMap，在里面以threadlocal为key,value为值,存入entry
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
    return value;
}
```

> 1. 当前线程对应了一个 `ThreadLocalMap`
> 2. 当前线程的 `ThreadLocal` 对应一个Map中的 `Entry` (存在table中)
> 3. `Entry` 中 `key` 会获取其对应的ThreadLocal，`value` 就是存储的数值

MessageQueue(3)

1、MessageQueue的主要操作

> 1. `enqueueMessage`：往消息队列中插入一条消息
> 2. `next`： 取出一条消息，并且从消息队列中移除
> 3. 本质采用 单链表 的数据结构来维护消息队列，而不是采用队列

2、MessageQueue的插入和读取源码分析

```
//MessageQueue.java：插入数据
boolean enqueueMessage(Message msg, long when) {
    //1. 主要就是单链表的插入操作
    synchronized (this) {
        ......
    }
    return true;
}
/**========================================
 * 功能：读取并且删除数据
 * 内部是无限循环，如果消息队列中没有消息就会一直阻塞。
 * 一旦有新消息到来，next方法就会返回该消息并且将其从单链表中移除
 *========================================*/
Message next() {
    for (;;) {
        ......
    }
}
```

3、MessageQueue的next源码详解

```
Message next() {
    int nextPollTimeoutMillis = 0;
    for (;;) {
        /**=========================================================================
         * 1、精确阻塞指定时间。第一次进入时因为nextPollTimeoutMillis=0，因此不会阻塞。
         *   1-如果nextPollTimeoutMillis=-1，一直阻塞不会超时。
         *   2-如果nextPollTimeoutMillis=0，不会阻塞，立即返回。
         *   3-如果nextPollTimeoutMillis>0，最长阻塞nextPollTimeoutMillis毫秒(超时)，如果期间
         *=========================================================================*/
        nativePollOnce(ptr, nextPollTimeoutMillis);

        synchronized (this) {
            // 当前时间
            final long now = SystemClock.uptimeMillis();
            Message msg = mMessages;
            /**=========================================================================
             * 2、当前Msg为消息屏障
             *   1-说明有重要的异步消息需要优先处理
             *   2-遍历查找到异步消息并且返回。
             *   3-如果没查询到异步消息，会continue，且阻塞在nativePollOnce直到有新消息
             *=========================================================================*/
            if (msg != null && msg.target == null) {
                // 遍历寻找到异步消息，或者末尾都没找到异步消息。
                do {
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
            /**=================================================================
             *  3、获取到消息
             *    1-消息时间已到，返回该消息。
             *    2-消息时间没到，表明有个延时消息，会修正nextPollTimeoutMillis。
             *    3-后面continue，精确阻塞在nativePollOnce方法
             *=================================================================*/
            if (msg != null) {
                // 延迟消息的时间还没到，因此重新计算nativePollOnce需要阻塞的时间
                if (now < msg.when) {
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALU
                } else {
                    // 返回获取到的消息(可以为一般消息、时间到的延迟消息、异步消息)
                    return msg;
                }
            } else {
                /**============================
                 * 4、没有找到消息或者异步消息
                 *============================*/
                nextPollTimeoutMillis = -1;
            }

            /**=======================================
             * 5、没有获取到消息，进行下一次循环。
             *   (1)此时可能处于的情况：
             *      1-没有获取到消息-nextPollTimeoutMillis = -1
             *      2-没有获取到异步消息(接收到同步屏障却没找到异步消息)-nextPollTimeoutMillis
             *      3-延时消息的时间没到-nextPollTimeoutMillis = msg.when-now
```

```
                *    (2)根据nextPollTimeoutMillis的数值，最终都会阻塞在nativePollOnce(-1)，
                *       直到enqueueMessage将消息添加到队列中。
                *=============================================*/
               if (pendingIdleHandlerCount <= 0) {
                   // 用于enqueueMessage进行精准唤醒
                   mBlocked = true;
                   continue;
               }
           }
       }
   }
```

1. 如果是一般消息，会去获取消息，没有获取到就会阻塞(native方法)，直到enqueueMessage插入新消息。获取到直接返回Msg。
2. 如果是同步屏障，会去循环查找异步消息，没有获取到会进行阻塞。获取到直接返回Msg。
3. 如果是延时消息，会计算时间间隔，并进行精准定时阻塞(native方法)。直到时间到达或者被enqueueMessage插入消息而唤醒。时间到后就返回Msg。

Looper(8)

1、Looper的构造

```
private Looper(boolean quitAllowed) {
    //1. 会创建消息队列: MessageQueue
    mQueue = new MessageQueue(quitAllowed);
    //2. 当前线程
    mThread = Thread.currentThread();
}
```

2、为线程创建Looper

```
//1. 在没有Looper的线程创建Handler会直接异常
new Thread("Thread#2"){
    @Override
    public void run(){
        Handler handler = new Handler();
    }
}.start();
```

> 异常:
> java.lang.RuntimeException: Can't create handler inside thread that has not called Looper.prepare()

```
//2. 用prepare为当前线程创建一个Looper
new Thread("Thread#2"){
    @Override
    public void run(){
        Looper.prepare();
        Handler handler = new Handler();
        //3. 开启消息循环
        Looper.loop();
    }
}.start();
```

3、主线程ActivityThread中的Looper

> 1. 主线程中使用 `prepareMainLooper()` 创建 `Looper`
> 2. `getMainLooper` 能够在任何地方获取到主线程的 `Looper`

4、Looper的退出

> 1. `Looper` 的退出有两个方法： `quit` 和 `quitSafely`
> 2. `quit` 会直接退出 `Looper`
> 3. `quitSafely` 只会设置退出标记，在已有消息全部处理完毕后才安全退出
> 4. `Looper` 退出后， `Handler` 的发行的消息会失败，此时 `send` 返回 `false`
> 5. 子线程 中如果手动创建了 `Looper` ， 应该在所有事情完成后调用 `quit` 方法来终止消息循环

5、Looper的loop()源码分析

```
//Looper.java
public static void loop() {
    //1. 获取Looper
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
    }
    //2. 获取消息队列
    final MessageQueue queue = me.mQueue;
    ......
    for (; ; ) {
        //3. 获取消息，如果没有消息则会一直阻塞
        Message msg = queue.next();
        /**================================
         * 4. 如果消息获得为null，则退出循环
         * -Looper退出后，next就会返回null
         *================================*/
        if (msg == null) {
            return;
        }
        ......
        /**============================================================
         * 5. 处理消息
         *  -msg.target:是发送消息的Handler
         *  -最终在该Looper中执行了Handler的dispatchMessage()
         *  -成功将代码逻辑切换到指定的Looper(线程)中执行
         *============================================================*/
        msg.target.dispatchMessage(msg);
        ......
    }
}
```

6、Android如何保证一个线程最多只能有一个Looper?

> 1-Looper的构造方法是private，不能直接构造。需要通过Looper.prepare()进行创建，

```
private Looper(boolean quitAllowed) {
    mQueue = new MessageQueue(quitAllowed);
    mThread = Thread.currentThread();
}
```

> 2-如果在已有Looper的线程中调用 `Looper.prepare()` 会抛出RuntimeException异常

```
public class Looper {

    static final HashMap<Long, Looper> looperRegistry = new HashMap<Long, Looper>();

    private static void prepare() {
        synchronized(Looper.class) {
            long currentThreadId = Thread.currentThread().getId();
            // 根据线程ID查询Looper
            Looper l = looperRegistry.get(currentThreadId);
            if (l != null)
                throw new RuntimeException("Only one Looper may be created per thread");
            looperRegistry.put(currentThreadId, new Looper(true));
        }
    }
    ...
}
```

7、Handler消息机制中，一个looper是如何区分多个Handler的?

> 1. Looper.loop()会阻塞于MessageQueue.next()
> 2. 取出msg后，msg.target成员变量就是该msg对应的Handler
> 3. 调用msg.target的disptachMessage()进行消息分发。这样多个Handler是很容易区分的。

主线程的消息循环

8、主线程ActivityThread的消息循环

```
//ActivityThread.java
public static void main(String[] args) {
    //1. 创建主线程的Looper和MessageQueue
    Looper.prepareMainLooper();
    ......
    //2. 开启消息循环
    Looper.loop();
}
/**=========================================
 * ActivityThread中需要Handler与消息队列进行交互
 * -内部定义一系列消息类型：主要有四大组件等
 * //ActivityThread.java
 *=========================================*/
private class H extends Handler {
    public static final int LAUNCH_ACTIVITY         = 100;
    public static final int PAUSE_ACTIVITY          = 101;
    public static final int PAUSE_ACTIVITY_FINISHING= 102;
    ......
}
```

> 1. ActivityThread 通过 ApplicationThread 和 AMS 进行 IPC通信
> 2. AMS 完成请求的工作后会回调 ApplicationThread 中的 Binder 方法
> 3. ApplicationThread 会向 Handler H 发送消息

> 4. H 接收到消息后会将 ApplicationThread 的逻辑切换到 ActivityThread 中去执行

Handler(6)

1、Handler使用实例post/sendMessage

> post

```
handler.post(new Runnable() {
        @Override
        public void run() {

        }
});
```

> sendMessage

```
// 自定义msg的what
static final int INT_WHAT_MSG = 1;
// 0、两种创建Msg的方法
Message message = new Message();
message = Message.obtain();
// 1、自定义MSG的类型，通过what进行区分
message.what = INT_WHAT_MSG;
// 2、发送Msg
handler.sendMessage(message);
// 3、自定义Handler处理Msg
class MsgHandler extends android.os.Handler{
        @Override
        public void handleMessage(Message msg) {
            switch (msg.what){
                case INT_WHAT_MSG:
                    // 识别出了Msg，进行逻辑处理
                    break;
                default:
                    break;
            }
        }
}
```

> post内部还是通过sendMessage实现的。

2、Handler的post/send()源码分析

```
//Handler.java: post最终是通过send系列方法实现的
//Handler.java
public final boolean sendMessage(Message msg)
{
    return sendMessageDelayed(msg, 0);
}
//Handler.java
public final boolean sendMessageDelayed(Message msg, long delayMillis)
{
    if (delayMillis < 0) {
        delayMillis = 0;
    }
    return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}
//Handler.java
public boolean sendMessageAtTime(Message msg, long uptimeMillis) {
    MessageQueue queue = mQueue;
    if (queue == null) {
        RuntimeException e = new RuntimeException(this + " sendMessageAtTime() called with no m
        Log.w("Looper", e.getMessage(), e);
        return false;
    }
    return enqueueMessage(queue, msg, uptimeMillis);
}
//Handler.java
private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
    msg.target = this;
    if (mAsynchronous) {
        msg.setAsynchronous(true);
    }
    //1. 最终是向消息队列插入一条消息
    return queue.enqueueMessage(msg, uptimeMillis);
}
```

1. `sendMessage()` 会将消息插入到 消息队列中
2. `MessageQueue` 的 `next` 方法就会返回这条消息交给 `Looper`
3. 最终 `Looper` 会把消息交给 `Handler` 的 `dispatchMessage`

3、Handler的postDelayed源码分析

```
//Handler.java---层层传递，和一般的post调用的同一个底层方法.
public final boolean postDelayed(Runnable r, long delayMillis)
{
    return sendMessageDelayed(getPostMessage(r), delayMillis);
}
//xxxxxx
//Handler.java
private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
    ...
    return queue.enqueueMessage(msg, uptimeMillis);
}
//MessageQueue.java
boolean enqueueMessage(Message msg, long when) {
    //会直接加进队列
}
```

1. postDelayed和post调用的底层sendMessage系列方法，只不过前者有延迟，后者延迟参数=0。
2. 最终会直接将Msg加入到队列中。
3. MessageQueue.next()在取出Msg时，如果发现消息A有延迟且时间没到，会阻塞消息队列。
4. 如果此时有非延迟的新消息B，会将其加入消息队列, 且处于消息A的前面，并且唤醒阻塞的消息队列。
5. 唤醒后会拿出队列头部的消息B，进行处理。然后会继续因为消息A而阻塞。
6. 如果达到了消息A延迟的时间，会取出消息A进行处理。

4、Handler的消息处理源码

```
//Handler.java
public void dispatchMessage(Message msg) {
    //1. Msg的callback存在时处理消息——Handler的post所传递的Runnable
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        /**==============================================
         *2. mCallback不为null时调用handleMessage
         * -Handler handle = new Handler(callback)
         * -好处在于不需要派生Handler子类并且也不需要重写其handleMessage
         *==============================================*/
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        //3. 如果其他条件都不符合，最后会调用Handler的handleMessage进行处理
        handleMessage(msg);
    }
}
//Handler.java-调用Handler的post所传递的Runnable的run()方法
private static void handleCallback(Message message) {
    message.callback.run();
}
//Handler.java-Callback接口用于不需要派生Handler就能完成功能
public interface Callback {
    public boolean handleMessage(Message msg);
}
```

5、Handler的特殊构造方法

> 1. `Handler handle = new Handler(callback);` -不需要派生Handler
> 2. 通过特定 `Looper` 构造 `Handler`

```
public Handler(Looper looper) {
    this(looper, null, false);
}
```

> 3. 默认构造函数

```
public Handler(Callback callback, boolean async) {
    ......
    mLooper = Looper.myLooper();
    //1. 在没有Looper的线程中创建Handler
    if (mLooper == null) {
        throw new RuntimeException(
                "Can't create handler inside thread that has not called Looper.prepare()");
    }
    mQueue = mLooper.mQueue;
    mCallback = callback;
    mAsynchronous = async;
}
```

内存泄漏

6、Handler的内存泄漏如何避免?

> 1. 采用静态内部类： `static handler = xxx`
> 2. Activity结束时，调用 `handler.removeCallback()` 、然后handler设置为null
> 3. 如果使用到Context等引用，要使用 弱引用