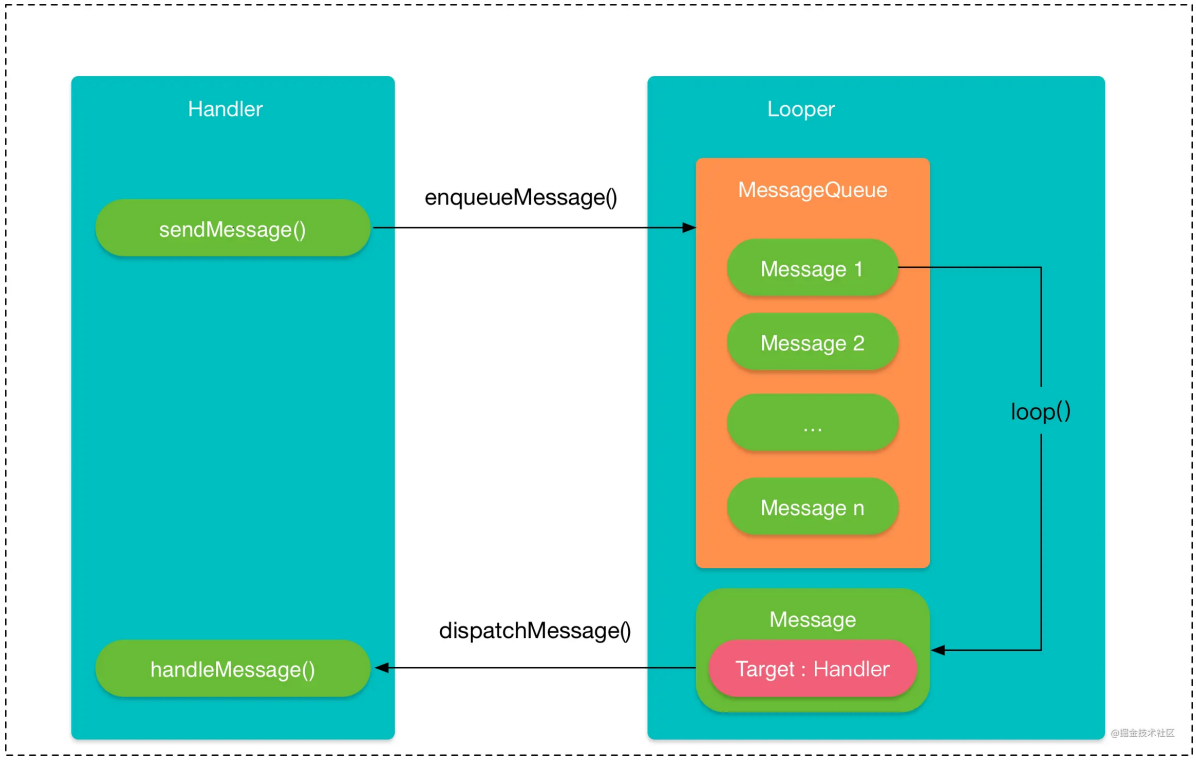
Handler构成

`Handler`：消息机制的上层接口，将任务切换到其指定的线程中执行，如用Handler在子线程更新UI。

Handler 发送的消息由 MessageQueue 存储管理，并由 Loopler 负责回调消息到 handleMessage()。

线程的转换由 Looper 完成，handleMessage() 执行的线程由 Looper.loop() 调用者所在线程决定。



> 每个Handler都会跟一个线程绑定，与该线程的Looper、MessageQueue关联，实现消息的管理及线程间通信。
>
> Handler运行依赖MessageQueue、Looper，及Looper内部的ThreadLocal。

Android是**事件驱动**的程序，界面刷新、交互等都是事件，被作为`Message`发送到了`MessageQueue`中。由`Looper`进行分发，由`Handler`处理。

主线程做耗时操作不是阻塞了主线程，而是阻塞了`Looper`的`loop`方法。导致`loop`方法无法处理其他事件，出现了ANR事件

Handler

Handler 对外暴露，内部包含一个Looper，负责Message的发送和处理

1. `Handler.enqueueMessage()`：将消息加入MessageQueue队列中

```
private boolean enqueueMessage(@NonNull MessageQueue queue, @NonNull Message
msg,long uptimeMillis) {
        msg.target = this;  // 每个发出去的Message都持有把它发出去的Handler的引用
        msg.workSourceUid = ThreadLocalWorkSource.getUid();
        if (mAsynchronous) {
            msg.setAsynchronous(true);
        }
        return queue.enqueueMessage(msg, uptimeMillis); // 进入
MessageQueue.enqueueMessage
}
```

2. `Handler()构造方法`：获取mLooper、mQueue对象

```
public Handler(... ){
    mLooper = Looper.myLooper();
    if (mLooper == null) {
        throw new RuntimeException(
           "Can't create handler inside thread " + Thread.currentThread()
            + " that has not called Looper.prepare()");
    }
    mQueue = mLooper.mQueue;
}
```

3. `Handle.send()系列方法`：发送消息，各种sendMessage方法最终都会进入enqueueMessage中

```
public final boolean sendMessage(@NonNull Message msg) {
        return sendMessageDelayed(msg, 0);
}
public final boolean sendMessageDelayed(@NonNull Message msg, long delayMillis)
{
        if (delayMillis < 0) {
            delayMillis = 0;
        }
        return sendMessageAtTime(msg, SystemClock.uptimeMillis() + delayMillis);
}
public boolean sendMessageAtTime(@NonNull Message msg, long uptimeMillis) {
        MessageQueue queue = mQueue;
        if (queue == null) {
            RuntimeException e = new RuntimeException(
                    this + " sendMessageAtTime() called with no mQueue");
            Log.w("Looper", e.getMessage(), e);
            return false;
        }
        return enqueueMessage(queue, msg, uptimeMillis);
}
```

4. `Handler.dispatchMessage()`：处理消息

```
public void dispatchMessage(@NonNull Message msg) {
        if (msg.callback != null) { // 1.首先分发给Message.callback
            handleCallback(msg);
        } else {
            if (mCallback != null) {    // 2.若无，则分发给Handler.callback
                if (mCallback.handleMessage(msg)) {
                    return;
                }
            }
            handleMessage(msg); // 3.可重写的handleMessage方法
        }
}
```

MessageQueue

消息队列是一个根据消息【执行时间先后】连接起来的、FIFO的单向链表

`Message`：消息体，内部包含一个目标处理器target，即最终处理该消息的Handler

1. `MessageQueue.enqueue(Message msg, long when)`方法：将msg加入消息队列中
    1. 获取队头消息mMessage
    2. 队列为空|当前消息立即执行|当前消息执行时间早于队头Message -> 将当前消息放到队头
    3. 若以上均否定，则将其插入队列中间位置，通过遍历整个队列，当队列中某消息执行时间晚于当前消息，将当前消息插入到该消息前

Looper

在消息队列中轮询消息，若有则取出，无则等待

消息循环的核心，内部包含一个MessageQueue，记录所有待处理消息；通过Looper.loop()轮询MessageQueue取出Message，分发消息给target handler，线程切换在这完成

1. `Looper.myLooper()`：获得本线程的Looper对象

```
public static @Nullable Looper myLooper() {
        return sThreadLocal.get();
}
// sThreadLocal.get() will return null unless you've called prepare().
@UnsupportedAppUsage
static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();
// sThreadLocal是一个ThreadLocal类，并且它的泛型是Looper对象
```

2. `Looper.prepare()`：创建Looper对象，必须调用且只能调用一次。**在构造Handler之前，必须调用`Looper`的`prepare`方法创建`Looper`，主线程不可调用该方法**

```
public static void prepare() {
        prepare(true);
}
private static void prepare(boolean quitAllowed) {
        if (sThreadLocal.get() != null) {
            throw new RuntimeException("Only one Looper may be created per
thread");
        }
        sThreadLocal.set(new Looper(quitAllowed));
}
public void set(T value) {
```

```
        Thread t = Thread.currentThread();
        ThreadLocalMap map = getMap(t);
        if (map != null)
            map.set(this, value);
        else
            createMap(t, value);
}
// key -> currentThread，value -> Looper。线程和Looper是一一对应的，Looper和线程绑定就是以键值对形式存进了一个map中
```

ThreadLocal提供了线程的局部变量，每个线程都可通过set/get操作该变量，但不会与其他线程的局部变量冲突，**实现线程的数据隔离。**

3. `Looper()`构造方法：获取Looper持有的MessageQueue及本线程的引用

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

4. `Looper().loop()`：开启轮询，死循环方式，不断尝试从MessageQueue中获取消息

```
public static void loop() {
    final Looper me = myLooper();   // 拿到当前线程的Looper
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
    }
    final MessageQueue queue = me.mQueue;   // 拿到Looper的消息队列
    // ...
    for (;;) {  // 1.这里开启死循环
        Message msg = queue.next(); // 阻塞式
        if (msg == null) {  // No message indicates that the message queue is quitting.
            return;
        }
        try {
            msg.target.dispatchMessage(msg);    // 调用Message.target.dispatchMessage方法
            // ...
        } catch (Exception exception) {
            // ...
            throw exception;
        } finally {
            // ...
        }
        // ...
        msg.recycleUnchecked();
    }
}
```

主线程Handler流程

ActivityThread.main()

```
public static void main(String[] args) {
    // ...
    Looper.prepareMainLooper(); // 1.获取主线程的Looper对象
    ActivityThread thread = new ActivityThread();
   // ...
    if (sMainThreadHandler == null) {
        sMainThreadHandler = thread.getHandler();   // 2.获得主线程Handler对象
    }
    // ...
    // End of event ActivityThreadMain.
    Looper.loop();  // 3.开始轮询，正常状态下不会执行最后一句抛出异常

    throw new RuntimeException("Main thread loop unexpectedly exited");
}
```

1. `prepareMainLooper()`：获取主线程Looper，不允许退出，与主线程放入map做绑定
2. `APP`启动的时候，main方法中自动创建`Looper`，并且和主线程绑定，平常用的`Handler`中的`Looper`就是主线程中创建的`Looper`。
3. 完整流程：
    1. `mainThread`中`ActivityThread`首先创建了一个运行在主线程的`Looper`，并且把它和主线程进行了绑定。
    2. `Looper`又创建了一个`MessageQueue`，然后调用`Looper.loop`方法不断地在主线程中尝试取出`Message`
    3. `Looper`如果取到了`Message`，那么就在主线程中调用发送这个`Message`的`Handler`的`handleMessage`方法。
    4. 我们在主线程或者子线程中通过`Looper.getMainLooper`为参数创建了一个`Handler`。
    5. 在子线程中发送了`Message`，主线程中的`Looper`不断循环，终于收到了`Message`，在主线程中调用了这个`Handler`的`handleMessage`方法。

Handler使用

使用步骤

1. 执行线程，Looper.prepare()创建Looper实例
2. 执行线程，创建Handler实例
3. 执行线程，Looper.loop()开启轮询
4. 调度线程，使用Handler发送消息

```
class LooperThread extends Thread {
    public Handler mHandler;
    public void run() {
        Looper.prepare();   // 创建Looper实例
        mHandler = new Handler() {  // 创建Handler实例，重写handleMessage方法
            public void handleMessage(Message msg) {
                // process incoming messages here
            }
        };
        Looper.loop();  // 启动Looper轮询
    }
}
```

使用场景

1. 子线程访问UI出现异常，使用Handler切换更新UI任务到主线程执行
2. 主线程不执行耗时操作（ANR），在子线程执行耗时操作，使用Handler切换回主线程更新UI

> 只允许在UI线程(主线程)执行UI更新相关操作，使用**单线程模型处理UI操作**的原因：
>
> 1. 多线程操作需要加锁，使UI访问复杂
> 2. 多线程操作加锁降低UI访问效率

基本用法

实例化Handler -> 重写handleMessage()方法 -> 调用send()/post()系列方法发送消息

```
Handler handler = new Handler() {
    @Override
    public void handleMessage(final Message msg) {
        // 接收并处理消息
    }
};
// 发送消息
handler.sendMessage(msg);
handler.post(runnable);
```

线程切换

1. 子线程用`Handler`发送消息，消息进入与主线程相关联的`MessageQueue`，由`MainLooper`轮询取出消息，使用`Message.what`属性调用该消息的`宿主Handler.dispatchMessage()`，即最后处理消息在主线程中使用子线程创建的`Handler`的处理消息方法。只有发送消息是在子线程，其它都是在主线程，**Handler与哪个线程的Looper相关联，消息处理逻辑就在与之相关的线程中执行**，相应的消息的走向也就在相关联的MessageQueue中。
2. 子线程异步网络请求+主线程更新UI操作的组合：主线程中创建子线程执行网络请求，得到响应后通过`子线程.Handler`发送`Message(UI更新操作)`到主线程的`MessageQueue`中，`主线程.Looper`轮询取得该消息，通过`msg.what`找到`子线程.Handler`，在主线程中调用`子线程.Handler.dispatchMessage()`,执行UI更新操作。