

另外我还分享一些整理的Android开发相关的学习资料（100G左右），资料包括开发工具、入门基础知识、进阶、项目实战的源码及视频，还有电子书。在我公众号“优派编程”后台回复“57966”获取！



介绍

《面试题思考与解答》系列期刊是将每月的知识点进行总结汇总。

要声明的一点是：面试题的目的不是为了让大家背题，而是从不同维度帮助大家复习，取长补短。

希望大家都能找到满意的工作。

以下为2022年3月刊内容。

Activity、PhoneWindow、DecorView、ViewRootImpl 之间的关系？

PhoneWindow：是Activity和View交互的中间层，帮助Activity管理View。

DecorView：是所有View的最顶层View，是所有View的parent。

ViewRootImpl：用于处理View相关的事件，比如绘制，事件分发，也是DecorView的parent。

四者的创建时机？

Activity创建于performLaunchActivity方法中，在startActivity时候触发。

PhoneWindow，同样创建于performLaunchActivity方法中，再具体点就是Activity的attach方法。

DecorView，创建于setContentView->PhoneWindow.installDecor。

`ViewRootImpl`，创建于`handleResumeActivity`方法中，最后通过`addView`被创建。

View的第一次绘制发生在什么时候？

第一次绘制就是发生在`handleResumeActivity`方法中，通过`addView`方法，创建了`ViewRootImpl`，并调用了其`setView`方法。

最后调用到`requestLayout`方法开始了布局、测量、绘制的流程。

线程更新UI导致崩溃的原因？

在触发绘制方法`requestLayout`中，有个`checkThread`方法：

```
void checkThread() {
    if (mThread != Thread.currentThread()) {
        throw new CalledFromWrongThreadException(
            "Only the original thread that created a view hierarchy can touch its views.");
    }
}
```

其中对`mThread`和当前线程进行了比较。而`mThread`是在`ViewRootImpl`实例化的时候赋值的。

所以崩溃的原因就是 view被绘制到界面时候的线程（也就是`ViewRootImpl`被创建时候的线程）和进行UI更新时候的线程不是同一个线程。

Activity、Dialog、PopupWindow、Toast 与Window的关系

这是扩展的一题，简单的从创建方式的角度来说一说：

- Activity。在Activity创建过程中所创建的`PhoneWindow`，是层级最小的Window，叫做应用Window，层级范围1-99。（层级范围大的Window可以覆盖层级小的Window）
- Dialog。Dialog的显示过程和Activity基本相同，也是创建了`PhoneWindow`，初始化`DecorView`，并将Dialog的视图添加到`DecorView`中，最终通过`addView`显示出来。

但是有一点不同的是，Dialog的Window并不是应用窗口，而是子窗口，层级范围1000-1999，子Window的显示必须依附于应用窗口，也会覆盖应用级Window。这也就是为什么Dialog传入的上下文必须为Activity的Context了。

- `PopupWindow`。`PopupWindow`的显示就有所不同了，它没有创建`PhoneWindow`，而是直接创建了一个View（`PopupDecorView`），然后通过`WindowManager`的`addView`方法显示出来了。

没有创建`PhoneWindow`，是不是就跟Window没关系了呢？

并不是，其实只要是调用了`WindowManager`的`addView`方法，那就是创建了Window，跟你有没有创建`PhoneWindow`无关。View就是Window的表现形式，只不过`PhoneWindow`的存在让Window形象更立体了一

些。

所以**PopupWindow**也是通过Window展示出来的，而它的Window层级属于子Window，必须依附与应用窗口。

- Toast。Toast和**PopupWindow**比较像，没有新建**PhoneWindow**，直接通过**addView**方法显示View即可。不同的是它属于系统级Window,层级范围2000-2999，所以无须依附于Activity。

四个比较下来，可以发现，只要想显示View，就会涉及到**WindowManager**的**addView**方法，也就用到了Window这个概念，然后会根据不同的分层依次显示覆盖到界面上。

不同的是，Activity和Dialog涉及到了布局比较复杂，还会有布局主题等元素，所以用到了**PhoneWindow**进行一个解耦，帮助他们管理View。而**PopupWindow**和Toast结构比较简单，所以直接新建一个类似**DecorView**的View，通过**addView**显示到界面。

为什么限制在应用间共享文件

打个比方，应用A有一个文件，绝对路径为file:///storage/emulated/0/Download/photo.jpg

现在应用A想通过其他应用来完成一些需求，比如拍照，就把他的这个文件路径发给了照相应用B，然后应用B照完相就把照片存储到了这个绝对路径。

看起来似乎没有什么问题，但是如果这个应用B是个“坏应用”呢？

- 泄漏了文件路径，也就是应用隐私。

如果这个应用A是“坏应用”呢？

- 自己可以不用申请存储权限，利用应用B就达到了存储文件的这一危险权限。

可以看到，这个之前落伍的方案，从自身到对方，都是不太好的选择。

所以Google就想了一个办法，把对文件的访问限制在应用内部。

- 如果要分享文件路径，不要分享file:// URI这种文件的绝对路径，而是分享content:// URI，这种相对路径，也就是这种格式：content://com.jimu.test.fileprovider/external/photo.jpg
- 然后其他应用可以通过这个绝对路径来向文件所属应用 索要 文件数据，所以文件所属的应用本身必须拥有文件的访问权限。

也就是应用A分享相对路径给应用B，应用B拿着这个相对路径找到应用A，应用A读取文件内容返给应用B。

介绍下FileProvider

涉及到应用间通信的问题，还记得IPC的几种方式吗？

- 文件
- AIDL
- ContentProvider
- Socket
- 等等。

从易用性，安全性，完整度等各个方面考虑，Google选择了ContentProvider为这次限制应用分享文件的 解决方案。于是，FileProvider诞生了。

具体做法就是：

```
<!-- 配置FileProvider-->

<provider
    android:name="androidx.core.content.FileProvider"
    android:authorities="${applicationId}.provider"
    android:exported="false"
    android:grantUriPermissions="true">
    <meta-data
        android:name="android.support.FILE_PROVIDER_PATHS"
        android:resource="@xml/provider_paths"/>
</provider>

<?xml version="1.0" encoding="utf-8"?>
<paths xmlns:android="http://schemas.android.com/apk/res/android">
    <external-path name="external" path="."/>
</paths>
```

```
//修改文件URL获取方式
```

```
Uri photoURI = FileProvider.getUriForFile(context, context.getApplicationContext().getPackageName() + ".provider", c
```

这样配置之后，就能生成content:// URI，并且也能通过这个URI来传输文件内容给外部应用。

FileProvider这些配置属性也就是ContentProvider的通用配置：

- android:name，是ContentProvider的类路径。
- android:authorities，是唯一标示，一般为包名+.provider。
- android:exported，表示该组件是否能被其他应用使用。
- android:grantUriPermissions，表示是否允许授权文件的临时访问权限。

其中要注意的是android:exported正常应该是true，因为要给外部应用使用。

但是FileProvider这里设置为false，并且必须为false。

这主要为了保护应用隐私，如果设置为`true`，那么任何一个应用都可以来访问当前应用的FileProvider了，对于应用文件来说肯定是不可取的，所以Android7.0以上会通过其他方式让外部应用安全的访问到这个文件，而不是普通的ContentProvider访问方式，后面会说到。

当然，也正是因为这个属性为`true`，所以在Android7.0以下，Android默认是将它当成一个普通的ContentProvider，外部无法通过content:// URI来访问文件。所以一般要判断下系统版本再确定传入的Uri到底是File格式还是content格式。

Service与子线程

关于Service，我的第一反应是运行在后台的服务。

关于后台，我的第一反应又是子线程。

那么Service和子线程到底是什么关系呢？

Service有两个比较重要的元素：

- 1、长时间运行。Service可以在Activity被销毁，程序被关闭之后都可以继续运行。
- 2、不提供界面的应用组件。这其实解释了后台的意义，Service的后台指的是不和界面交互，不依赖UI元素。

而且比较关键的点是，Service也是运行在主线程之中。

所以运行在后台的Service和运行在后台的线程区别还是挺大的。

首先，所运行的线程不同。Service还是运行在主线程，而子线程肯定是开辟了新的线程。

其次，后台的概念不同。Service的后台指的是不与界面交互，子线程的后台指的是异步运行。

最后，Service作为四大组件之一，控制它更方便，只要有上下文就可以对其进行控制。

当然，虽然两者概念不同，但是还是有很多合作之处。

Service作为后台运行的组件，其实很多时候也会被用来做耗时操作，那运行在主线程的Service肯定不能直接进行耗时操作，这就需要子线程了。

开启一个后台Service，然后在Service里面进行子线程操作，这样的结合给项目带来的可能性就更大了。

Google也是考虑到这一点，设计出了IntentService这种已经结合好的组件供我们使用。

后台和前台Service

这就涉及到Service的分类了。

如果从是否无感知来分类，Service可以分为前台和后台。前台Service会通过通知的方式让用户感知到，后台有这么一个玩意在运行。

比如音乐类APP，在后台播放音乐的同时，可以发现始终有一个通知显示在前台，让用户知道，后台有一个这么音乐相关的服务。

在Android8.0，Google要求如果程序在后台，那么就不能创建后台服务，已经开启的后台服务会在一定时间后被停止。

所以，建议使用前台Service，它拥有更高的优先级，不易被销毁。使用方法如下：

```
startForegroundService(intent);

public void onCreate() {
    super.onCreate();
    Notification notification = new Notification.Builder(this)
        .setChannelId(CHANNEL_ID)
        .setContentTitle("主服务")//标题
        .setContentText("运行中...")//内容
        .setSmallIcon(R.mipmap.ic_launcher)
        .build();
    startForeground(1,notification);
}

<!--android 9.0上使用前台服务，需要添加权限-->
<uses-permission android:name="android.permission.FOREGROUND_SERVICE" />
```

那后台任务该怎么办呢？官方建议使用 `JobScheduler`。

说说JobScheduler

任务调度`JobScheduler`，Android5.0被推出。（可能有的朋友感觉比较陌生，其实他也是通过Service实现的，这个待会再说）

它能做的工作就是可以在你所规定的要求下进行自动任务执行。比如规定时间、网络为WIFI情况、设备空闲、充电时等各种情况下后台自动运行。

所以Google让它来替代后台Service的一部分功能，使用：

首先，创建一个`JobService`：

```
public class MyJobService extends JobService {

    @Override
    public boolean onStartJob(JobParameters params) {
```

```

        return false;
    }

    @Override
    public boolean onStopJob(JobParameters params) {
        return false;
    }
}

```

然后，注册这个服务（因为 `JobService` 也是 `Service`）：

```

<service android:name=".MyJobService"
    android:permission="android.permission.BIND_JOB_SERVICE" />

```

最后，创建一个 `JobInfo` 并执行：

```

JobScheduler scheduler = (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);
ComponentName jobService = new ComponentName(this, MyJobService.class);

JobInfo jobInfo = new JobInfo.Builder(ID, jobService)
    .setMinimumLatency(5000) // 任务最少延迟时间
    .setOverrideDeadline(60000) // 任务deadline，当到期没达到指定条件也会开始执行
    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED) // 网络条件，默认值NETWORK_TYPE_NONE
    .setRequiresCharging(true) // 是否充电
    .setRequiresDeviceIdle(false) // 设备是否空闲
    .setPersisted(true) // 设备重启后是否继续执行
    .setBackoffCriteria(3000, JobInfo.BACKOFF_POLICY_LINEAR) // 设置退避/重试策略
    .build();
scheduler.schedule(jobInfo);

```

简单说下原理：

`JobSchedulerService` 是在 `SystemService` 中启动的服务，然后会遍历没有完成的任务，通过 `Binder` 找到对应的 `JobService`，执行 `onStartJob` 方法，完成任务。具体可以看看参考链接的分析。

所以也就知道了，在 5.0 之后，如果有需要后台任务执行，特别是需要满足一定条件触发的任务，比如网络电量等等情况，就可以使用 `JobScheduler`。

有的人可能要问了，5.0 之前怎么办呢？

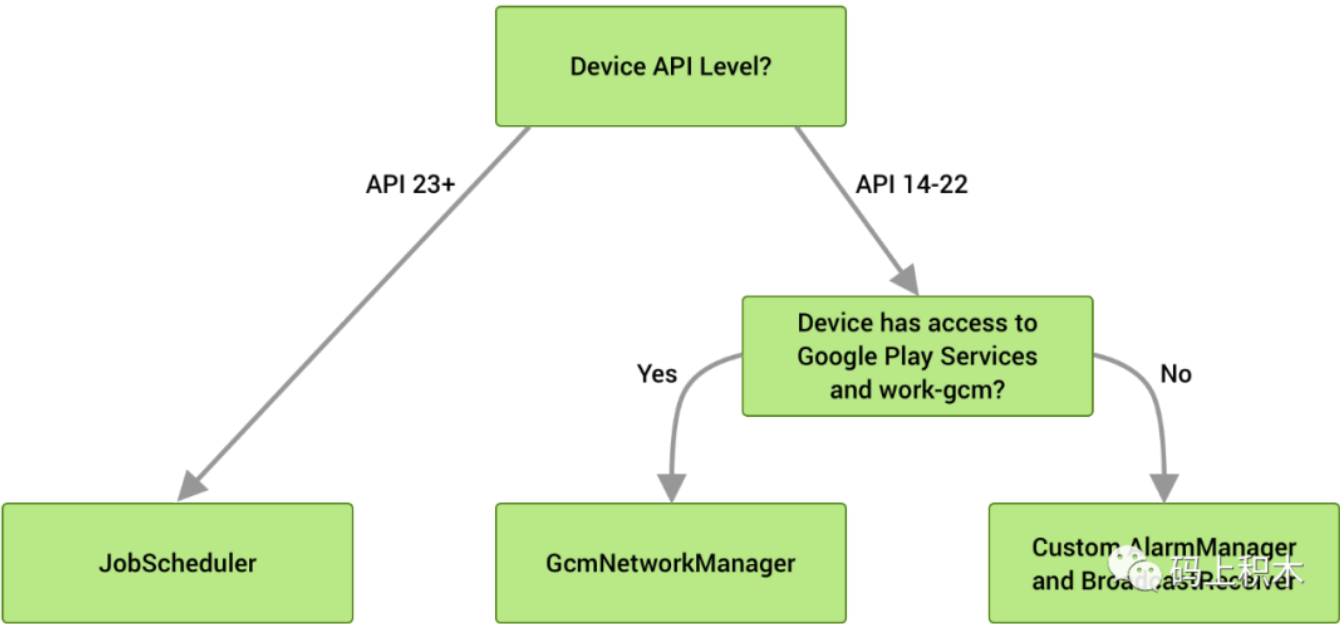
可以使用 `GcmNetworkManager` 或者 `BroadcastReceiver` 等处理部分情况下的任务需求。

Google 也是考虑到了这一点，所以将 5.0 之后的 `JobScheduler` 和 5.0 之前的 `GcmNetworkManager`、`GcmNetworkManager`、`AlarmManager` 等和任务相关的 API 相结合，设计出了 `WorkManager`。

说说 WorkManager

WorkManager 是一个 API，可供您轻松调度那些即使在退出应用或重启设备后仍应运行的可延期异步任务。

作为Jetpack的一员，并不算很新的内容，它的本质就是结合已有的任务调度相关的API，然后根据版本需求等来执行这些任务，官网有一张图：



所以WorkManager到底能做什么呢？

- 1、对于一些任务约束能很好的执行，比如网络、设备空闲状态、足够存储空间等条件下需要执行的任务。
- 2、可以重复、一次性、稳定的执行任务。包括在设备重启之后都能继续任务。
- 3、可以定义不同工作任务的衔接关系。比如设定一个任务接着一个任务。

总之，它是后台执行任务的一大利器。

onStart可见的解释？onStart和onResume两种状态的设计。

首先，科普官方定义的两个状态。

- 1、onStart到onStop中间的状态叫做“已开始”状态。
- 2、onResume到onPause中间的状态叫做“已恢复”状态。


然后我们做个小实验，定义ActivityA 和 ActivityB，ActivityB为Dialog主题，ActivityA中点击可以跳转到B：

```
image.setOnClickListener {
    startActivity(Intent(this, ActivityB::class.java))
}

<activity android:name=".activity.ActivityB"
    android:theme="@style/Theme.AppCompat.Light.Dialog"
    android:launchMode="standard">
</activity>
```


进入ActivityA后，点击按钮，跳转到B，这时候A的生命周期走到了`onPause`，也就是回到了已开始状态。

```
E/jimu: onStart  
E/jimu: onResume  
E/jimu: onPause
```

 码上积木

这个时候，界面是这个样子：

StudyNote3

ActivityA处在已开始状态，对用户可见。

这里的可见是不是就很好理解了，确实对我们可见了，只不过 不在前台，不能交互。

所以延伸到普通的Activity，这个可见，并不是表示用户能用肉眼看到了，而是想表达：Activity已经显示出来了，但是还不在前台，所以只是可见，但不可交互。

这个可见状态是从onStart开始，onStop结束，我们可以分为两个阶段：

- 1、onStart到onResume。这个阶段，Activity被创建，布局已加载，但是界面还没绘制，可以说界面都不存在。
- 2、onPause到onStop。这个阶段，就是我们刚才所做的实验，Activity有界面，只是被新的界面所遮挡，也就是不在前台。

所以综合两个阶段，我们把这种Activity被创建或已经显示出来，但是不在前台，介于两者之间的状态叫做 可见 状态。

到此，我们知道了可见的意思，其实也就知道了另外一个问题，也就是为什么要设计出onStart和onResume这两种状态。

- 1、onStart和onStop，是从Activity是否可见的角度设计的。
- 2、onResume和onPause，是从Activity是否位于前台的角度设计的。

所以Activity的生命周期又可以解释为：

被创建 (onCreate) ——> 可见 (onStart) ——> 位于前台 (onResume) ——> 可见但不在前台 (onPause)

onStart可见的解释？可见进程

从另外的角度看，这个可见 可以指的是 可见进程。这就涉及到进程的分类。

为了确定在内存不足时应该终止哪些进程，Android 会根据每个进程中运行的组件以及这些组件的状态，将它们放入“重要性层次结构”。这些进程类型包括（按重要性排序）：前台进程,可见进程,服务流程,缓存进程。

这些进程是什么意思呢？

- 前台进程是用户目前执行操作所需的进程。比如 正在用户的互动屏幕上运行一个 Activity。（其 `onResume()` 方法已被调用）
- 可见进程是正在进行用户当前知晓的任务。比如 正在运行的 Activity 在屏幕上对用户可见，但不在前台。（其 `onPause()` 方法已被调用）
- 服务流程包含一个已使用 `startService()` 方法启动的 Service。
- 缓存进程是目前不需要的进程。比如 当前不可见的一个或多个 Activity 实例。（`onStop()` 方法已被调用并返回）

所以Activity的生命周期又可以通过进程分为：

可见进程（`onStart`）——> 前台进程（`onResume`）——> 可见进程（`onPause`）——> 缓存进程（`onStop`）

这些进程有什么用呢？

我们都知道，在Android系统中有很多很多运行中的APP，也就代表了不同的进程。

当内存不够时（达到了某个阈值），系统首先会通过`onTrimMemory()`回调方法告诉应用，让应用自己来处理低内存情况下的减少内存操作。这之后，如果内存还是很紧张，那么就会开始对一些进程的杀除，以释放内存。这里就需要判断进程的优先级了，从低优先级开始按顺序终止进程。

所以，进程的分类作用就在这了。优先级的高低其实就代表了 终止进程的顺序，也代表了对用户的影响程度。

当然实际代码中，进程优先级是有数字表示的，也就是ADJ，而上面说的进程类型都有相应的进程优先级数字范围。比如：

```
public final class ProcessList {
    //可见进程
    static final int VISIBLE_APP_ADJ = 100;

    // 前台进程
    static final int FOREGROUND_APP_ADJ = 0;

    // 服务进程
    static final int SERVICE_ADJ = 500;

    // 缓存进程
    static final int CACHED_APP_MIN_ADJ = 900;

    //...
}
```

再回到我们的问题上来：

其中，可见进程这里也出现了可见的概念，给出的解释是：用户知晓。

当我们点击一个页面，我们知道这个页面将要显示出来，也知道之前的页面在这个页面后面。所以这些页面和进程都是我们所知晓的，只是不在前台。

所以onStart表示的可见，也可以理解为可见进程，意思是这个Activity所在的进程任务已经被创建并显示，我们知晓它，只是没在前台。

介绍下okhttp中的设计模式

外观模式。通过OkHttpClient这个外观去实现内部各种功能。

建造者模式。构建不同的Request对象。

工厂模式。通过OkHttpClient生产出产品RealCall。

享元模式。通过线程池、连接池共享对象。

责任链模式。将不同功能的拦截器形成一个链。

具体讲解可以看之前的文章：

https://mp.weixin.qq.com/s/eHLXxjvMgII6c_FVRwwdjg

介绍下okhttp的拦截器

addInterceptor(Interceptor)，这是由开发者设置的，会按照开发者的要求，在所有的拦截器处理之前进行最早的拦截处理，比如一些公共参数，Header都可以在这里添加。

RetryAndFollowUpInterceptor，这里会对连接做一些初始化工作，以及请求失败的重试工作，重定向的后续请求工作。

BridgeInterceptor，这里会为用户构建一个能够进行网络访问的请求，同时后续工作将网络请求回来的响应Response转化为用户可用的Response，比如添加文件类型，content-length计算添加，gzip解包。

CacheInterceptor，这里主要是处理cache相关处理，会根据OkHttpClient对象的配置以及缓存策略对请求值进行缓存，而且如果本地有了可用的Cache，就可以在没有网络交互的情况下就返回缓存结果。

ConnectInterceptor，这里主要就是负责建立连接了，会建立TCP连接或者TLS连接，以及负责编码解码的HttpCodec。

networkInterceptors，这里也是开发者自己设置的，所以本质上和第一个拦截器差不多，但是由于位置不同，用处也不同。这个位置添加的拦截器可以看到请求和响应的数据了，所以可以做一些网络调试。

CallServerInterceptor，这里就是进行网络数据的请求和响应了，也就是实际的网络I/O操作，通过socket读写数据。

okhttp的连接池工作流程，说说ConnectInterceptor。

连接拦截器，之前说了是关于TCP连接的。

```
object ConnectInterceptor : Interceptor {
    @Throws(IOException::class)
    override fun intercept(chain: Interceptor.Chain): Response {
        val realChain = chain as RealInterceptorChain
        val exchange = realChain.call.initExchange(chain)
        val connectedChain = realChain.copy(exchange = exchange)
        return connectedChain.proceed(realChain.request)
    }
}
```

代码看着倒是挺少的，但其实这里面很复杂很复杂，不着急，我们慢慢说。这段代码就执行了一个方法就是 **initExchange** 方法：

```
internal fun initExchange(chain: RealInterceptorChain): Exchange {
    val codec = exchangeFinder.find(client, chain)
    val result = Exchange(this, eventListener, exchangeFinder, codec)
    return result
}

fun find(
    client: OkHttpClient,
    chain: RealInterceptorChain
): ExchangeCodec {
    try {
        val resultConnection = findHealthyConnection(
            connectTimeout = chain.connectTimeoutMillis,
            readTimeout = chain.readTimeoutMillis,
            writeTimeout = chain.writeTimeoutMillis,
            pingIntervalMillis = client.pingIntervalMillis,
            connectionRetryEnabled = client.retryOnConnectionFailure,
            doExtensiveHealthChecks = chain.request.method != "GET"
        )
        return resultConnection.newCodec(client, chain)
    }
}
```

好像有一点眉目了，找到一个 **ExchangeCodec** 类，并封装成一个 **Exchange** 类。

ExchangeCodec：是一个连接所用的编码解码器，用于编码HTTP请求和解码HTTP响应。

Exchange：封装这个编码解码器的一个工具类，用于管理 **ExchangeCodec**，处理实际的 I/O。

明白了，这个连接拦截器（**ConnectInterceptor**）就是找到一个可用连接呗，也就是TCP连接，这个连接就是用于HTTP请求和响应的。你可以把它可以理解为一个管道，有了这个管道，才能把数据丢进去，也才可以从管道里面取数据。

而这个`ExchangeCodec`，编码解码器就是用来读取和输送到这个管道的一个工具，相当于把你的数据封装成这个连接（管道）需要的格式。我咋知道的？我贴一段`ExchangeCodec`代码你就明白了：

```
//Http1ExchangeCodec.java
fun writeRequest(headers: Headers, requestLine: String) {
    check(state == STATE_IDLE) { "state: $state" }
    sink.writeUtf8(requestLine).writeUtf8("\r\n")
    for (i in 0 until headers.size) {
        sink.writeUtf8(headers.name(i))
            .writeUtf8(": ")
            .writeUtf8(headers.value(i))
            .writeUtf8("\r\n")
    }
    sink.writeUtf8("\r\n")
    state = STATE_OPEN_REQUEST_BODY
}
```

这里贴的是`Http1ExchangeCodec`的write代码，也就是Http1的编码解码器。

很明显，就是将Header信息一行一行写到sink中，然后再由sink交给输出流，具体就不分析了。只要知道这个编码解码器就是用来处理连接中进行输送的数据即可。

然后就是这个拦截器的关键了，连接到底是怎么获取的呢？继续看看：

```
private fun findConnection(): RealConnection {

    // 1、复用当前连接
    val callConnection = call.connection
    if (callConnection != null) {
        //检查这个连接是否可用和可复用
        if (callConnection.noNewExchanges || !sameHostAndPort(callConnection.route().address.url)) {
            toClose = call.releaseConnectionNoEvents()
        }
        return callConnection
    }

    //2、从连接池中获取可用连接
    if (connectionPool.callAcquirePooledConnection(address, call, null, false)) {
        val result = call.connection!!
        eventListener.connectionAcquired(call, result)
        return result
    }

    //3、从连接池中获取可用连接（通过一组路由routes）
    if (connectionPool.callAcquirePooledConnection(address, call, routes, false)) {
        val result = call.connection!!
        return result
    }
    route = localRouteSelection.next()

    // 4、创建新连接
    val newConnection = RealConnection(connectionPool, route)
    newConnection.connect

    // 5、再获取一次连接，防止在新建连接过程中有其他竞争连接被创建了
    if (connectionPool.callAcquirePooledConnection(address, call, routes, true)) {
        return result
    }

    //6、还是要使用创建的新连接，放入连接池，并返回
    connectionPool.put(newConnection)
    return newConnection
}
```


获取连接的过程很复杂，为了方便看懂，我简化了代码，分成了6步。

1、检查当前连接是否可用。

怎么判断可用的？主要做了两个判断 1) 判断是否不再接受新的连接 2) 判断和当前请求有相同的主机名和端口号。

这倒是很好理解，要这个连接是连接的同一个地方才能复用是吧，同一个地方怎么判断？就是判断主机名和端口号。

还有个问题就是为什么有当前连接？？明明还没开始连接也没有获取连接啊，怎么连接就被赋值了？

还记得重试和重定向拦截器吗？对了，就是当请求失败需要重试的时候或者重定向的时候，这时候连接还在呢，是可以直接进行复用的。

2和3、从连接池中获取可用连接。

第2步和第3步都是从连接池获取连接，有什么不一样吗？

```
connectionPool.callAcquirePooledConnection(address, call, null, false)
connectionPool.callAcquirePooledConnection(address, call, routes, false)
```

好像多了一个routes字段？

这里涉及到HTTP/2的一个技术，叫做 **HTTP/2 CONNECTION COALESCING**（连接合并），什么意思呢？

假设有两个域名，可以解析为相同的IP地址，并且是可以使用相同的TLS证书（比如通配符证书），那么客户端可以重用相同的TCP连接从这两个域名中获取资源。

再看回我们的连接池，这个routes就是当前域名（主机名）可以被解析的ip地址集合，这两个方法的区别也就是一个传了路由地址，一个没有传。

继续看**callAcquirePooledConnection**代码：

```
internal fun isEligible(address: Address, routes: List<Route>?): Boolean {
    if (address.url.host == this.route().address.url.host) {
        return true
    }

    //HTTP/2 CONNECTION COALESCING
    if (http2Connection == null) return false
    if (routes == null || !routeMatchesAny(routes)) return false
    if (address.hostnameVerifier != OkHostnameVerifier) return false
    return true
}
```

1) 判断主机名、端口号等，如果请求完全相同就直接返回这个连接。2) 如果主机名不同，还可以判断是不是HTTP/2请求，如果是就继续判断路由地址，证书，如果都能匹配上，那么这个连接也是可用的。

4、创建新连接。

如果没有从连接池中获取到新连接，那么就创建一个新连接，这里就不多说了，其实就是调用到`socket.connect`进行TCP连接。

5、再从连接池获取一次连接，防止在新建连接过程中有其他竞争连接被创建了。

创建了新连接，为什么还要去连接池获取一次连接呢？因为在这个过程中，有可能有其他的请求和你一起创建了新连接，所以我们需要再去取一次连接，如果有可以用的，就直接用它，防止资源浪费。

其实这里又涉及到HTTP2的一个知识点：多路复用。

简单的说，就是不需要当前连接的上一个请求结束之后再去进行下一次请求，只要有连接就可以直接用。

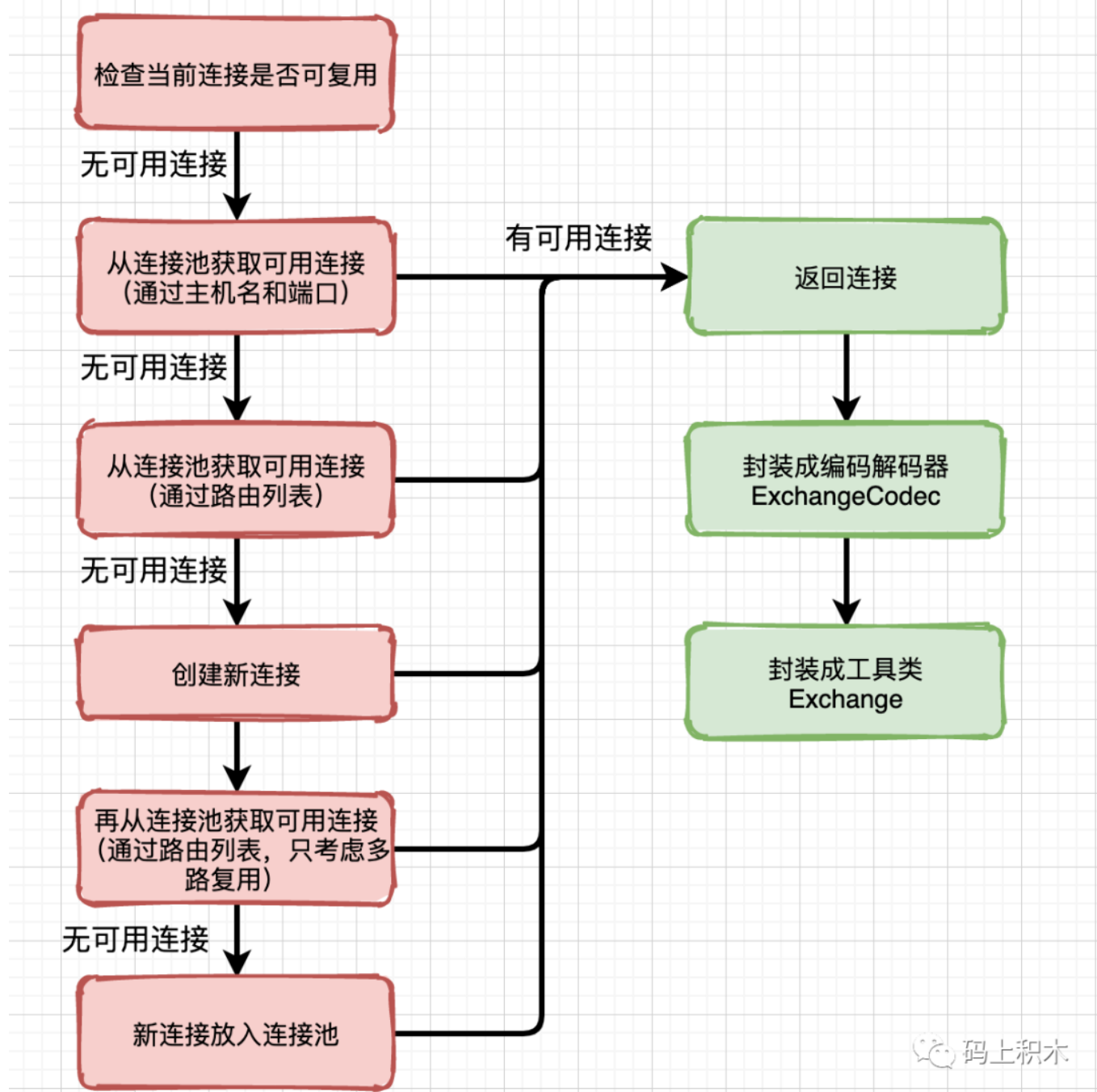
HTTP/2引入二进制数据帧和流的概念，其中帧对数据进行顺序标识，这样在收到数据之后，就可以按照序列对数据进行合并，而不会出现合并后数据错乱的情况。同样是因为有了序列，服务器就可以并行的传输数据，这就是流所做的事情。

所以在HTTP/2中可以保证在同一个域名只建立一路连接，并且可以并发进行请求。

6、新连接放入连接池，并返回。

最后一步好理解吧，走到这里说明就要用这个新连接了，那么就把它存到连接池，返回这个连接。

这个拦截器确实麻烦，大家好好梳理下吧，我也再来个图：



饿汉单例为什么是线程安全的？

保证一个实例很简单，只要每次返回同一个实例就可以，关键是如何保证实例化过程的线程安全？

这里先回顾下类的初始化。

在类实例化之前，JVM会执行类加载。

而类加载的最后一步就是进行类的初始化，在这个阶段，会执行类构造器`<clinit>`方法，其主要工作就是初始化类中静态的变量，代码块。

而`<clinit>()`方法是阻塞的，在多线程环境下，如果多个线程同时去初始化一个类，那么只会有一个线程去执行这个类的`<clinit>()`，其他线程都会被阻塞。换句话说，`<clinit>`方法被赋予了线程安全的能力。

再结合我们要实现的单例，就很容易想到可以通过静态变量的形式创建这个单例，这个过程是线程安全的，所以我们得出了第一种单例实现方法：

```
private static Singleton singleton = new Singleton();

public static Singleton getSingleton() {
    return singleton;
}
```

很简单，就是通过静态变量实现唯一单例，并且是线程安全的。

看似比较完美的一个方法，也是有缺点的，就是有可能我还没有调用`getSingleton`方法的时候，就进行了类的加载，比如用到了反射或者类中其他的静态变量静态方法。所以这个方法的缺点就是有可能会造成资源浪费，在我没用到这个单例的时候就对单例进行了实例化。

在同一个类加载器下，一个类型只会被初始化一次，一共有六种能够触发类初始化的时机：

- 1、虚拟机启动时，初始化包含 main 方法的主类。
- 2、new等指令创建对象实例时。
- 3、访问静态方法或者静态字段的指令时。
- 4、子类的初始化过程如果发现其父类还没有进行过初始化。
- 5、使用反射API 进行反射调用时。
- 6、第一次调用`java.lang.invoke.MethodHandle`实例时。

这种我不管你用不用，只要我这个类初始化了，我就要实例化这个单例，被类比为 饿汉方法。（是真饿了，先实例化出来放着吧，要吃的时候就可以直接吃了）

缺点就是 有可能造成资源浪费。（到最后，饭也没吃上，饭就浪费了）

但其实这种模式一般也够用了，因为一般情况下用到这个实例的时候才会去用这个类，很少存在需要使用这个类但是不使用其单例的时候。

当然，话不能说绝了，也是有更好的办法来解决这种可能的资源浪费。

kotlin 单例为什么这么简单？

```
object Singleton
```

没了？嗯，没了。

这里涉及到一个kotlin中才有的关键字：`object`（对象）。

关于`object`主要有三种用法：

1、对象表达式。

主要用于创建一个继承自某个（或某些）类型的匿名类的对象。

```
window.addMouseListener(object : MouseAdapter() {  
    override fun mouseClicked(e: MouseEvent) { /*.....*/ }  
  
    override fun mouseEntered(e: MouseEvent) { /*.....*/ }  
})
```

2、对象声明。

主要用于单例。也就是我们今天用到的用法。

```
object Singleton
```

我们可以通过Android Studio 的 Show Kotlin Bytecode 功能，看到反编译后的java代码：

```
public final class Singleton {  
    public static final Singleton INSTANCE;  
  
    private Singleton() {  
    }  
  
    static {  
        Singleton var0 = new Singleton();  
        INSTANCE = var0;  
    }  
}
```

很显然，跟我们上一节写的饿汉差不多，都是在类的初始化阶段就会实例化出来单例，只不过一个是通过静态代码块，一个是通过静态变量。

3、伴生对象。

类内部的对象声明可以用 `companion` 关键字标记，有点像静态变量，但是并不是真的静态变量。

```
class MyClass {  
    companion object Factory {  
        fun create(): MyClass = MyClass()  
    }  
}  
  
//使用  
MyClass.create()
```

反编译成Java代码：

```
public final class MyClass {
    public static final MyClass.Factory Factory = new MyClass.Factory((DefaultConstructorMarker)null);
    public static final class Factory {
        @NotNull
        public final MyClass create() {
            return new MyClass();
        }

        private Factory() {
        }

        // $FF: synthetic method
        public Factory(DefaultConstructorMarker $constructor_marker) {
            this();
        }
    }
}
```

其原理还是一个静态内部类，最终调用的还是这个静态内部类的方法，只不过省略了静态内部类的名称。

要想实现真正的静态成员需要 `@JvmField` 修饰变量。

静态内部类单例的实现原理

有什么办法可以不浪费这个实例呢？也就是达到 按需加载 单例？

这就要涉及到另外一个知识点了，静态内部类的加载时机。

刚才说到类的加载时候，初始化过程只会加载静态变量和代码块，所以是不会加载静态内部类的。

静态内部类是延时加载的，意思就是说只有在明确用到内部类时才加载。只使用外部类时不加载。

根据这个信息，我们就可以优化刚才的 饿汉模式，改成静态内部类模式(java和kotlin版本)：

```
private static class SingletonHolder {
    private static Singleton INSTANCE = new Singleton();
}

public static Singleton getSingleton() {
    return SingletonHolder.INSTANCE;
}
companion object {
    val instance = SingletonHolder.holder
}

private object SingletonHolder {
    val holder = SingletonDemo()
}
```

同样是通过类的初始化 `<clinit>()` 方法保证线程安全，并且在此之上，将单例的实例化过程向后移，移到静态内部类。所以就变成了当调用 `getSingleton` 方法的时候才会去初始化这个静态内部类，也就是才会实例化静态单例。

如此一整，这种方法就完美了...吗？好像也有缺点啊，比如我调用`getSingleton`方法创建实例的时候想传入参数怎么办呢？

可以，但是需要一开始就设置好参数值，无法通过调用`getSingleton`方法来动态设置参数。比如这样写：

```
private static class SingletonHolder {
    private static String test="123";
    private static Singleton INSTANCE = new Singleton(test);
}

public static Singleton getSingleton() {
    SingletonHolder.test="12345";
    return SingletonHolder.INSTANCE;
}
```

最终实例化进去的test只会是123，而不是12345。因为只要你开始用到`SingletonHolder`内部类，单例`INSTANCE`就会最开始完成了实例化，即使你赋值了test，也是单例实例化之后的事了。

这个就是 静态内部类方法的缺点了。如果不用动态传参数，那么这个方法已经足够了。

双重校验单例方式的原理

加锁怎么加，也是个问题。

首先肯定的是，我们加的锁肯定是类锁，因为要针对这个类进行加锁，保证同一时间只有一个线程进行单例的实例化操作。

那么类锁就有两种加法了，修饰静态方法和修饰类对象：

```
//方法1，修饰静态方法
public synchronized static Singleton getSingleton() {
    if (singleton == null) {
        singleton = new Singleton();
    }
    return singleton;
}

//方法2，代码块修饰类对象
public static Singleton getSingleton() {
    if (singleton == null) {
        synchronized (Singleton.class) {
            if (singleton == null) {
                singleton = new Singleton();
            }
        }
    }
    return singleton;
}
```

方法2这种方式就是我们常说的双重校验的模式。

比较下两种方式其实区别也就是在这个双重校验，首先判断单例是否为空，如果为空再进入加锁阶段，正常走单例的实例化代码。

那么，为什么要这么做呢？

第一个判断，是为了性能。当这个`singleton`已经实例化之后，我们再取值其实是不需要再进入加锁阶段的，所以第一个判断就是为了减少加锁。把加锁只控制在第一次实例化这个过程中，后续就可以直接获取单例即可。

第二个判断，是防止重复创建对象。当两个线程同时走到`synchronized`这里，线程A获得锁，进入创建对象。创建完对象后释放锁，然后线程B获得锁，如果这时候没有判断单例是否为空，那么就会再次创建对象，重复了这个操作。

到这里，看似问题都解决了。

等等，`new Singleton()`这个实例化过程真的没问题吗？

在JVM中，有一种操作叫做指令重排：

JVM为了优化指令，提高程序运行效率，在不影响单线程程序执行结果的前提下，会将指令进行重新排序，但是这种重新排序不会对单线程程序产生影响。

简单的说，就是在不影响最终结果的情况下，一些指令顺序可能会被打乱。

再看看在对象实例化中的指令主要有这三步操作：

- 1、分配对象内存空间。
- 2、初始化对象。
- 3、`instance`指向刚分配的内存地址。

如果我们将第二步和第三步重排一下，结果也是不影响的：

- 1、分配对象内存空间。
- 2、`instance`指向刚分配的内存地址。
- 3、初始化对象。

这种情况下，就有问题了：

当线程A进入实例化阶段，也就是`new Singleton()`，刚完成第二步分配好内存地址。这时候线程B调用了`getSingleton()`方法，走到第一个判空，发现不为空，返回单例，结果用的时候就有问题了，对象都没有初始化完成。

这就是指令重排有可能导致的问题。

所以，我们需要禁止指令重排，`volatile` 登场。

`volatile` 主要有两个特性：

- 1、可见性。也就是写操作会对其他线程可见。
- 2、禁止指令重排。

所以再加上`volatile` 对变量进行修饰，这个双重校验的单例模式也就完整了。

```
private volatile static Singleton singleton;
```

Handler被设计出来的原因？有什么用？

一种东西被设计出来肯定就有它存在的意义，而Handler的意义就是切换线程。

作为Android消息机制的主要成员，它管理着所有与界面有关的消息事件，常见的使用场景有：

- 1、跨进程之后的界面消息处理。

比如Activity的启动，就是AMS在进行进程间通信的时候，通过Binder线程 将消息发送给`ApplicationThread`的消息处理器Handler，然后再将消息分发给主线程中去执行。

- 2、网络交互后切换到主线程进行UI更新。

当子线程网络操作之后，需要切换到主线程进行UI更新。

总之一句话，Handler的存在就是为了解决在子线程中无法访问UI的问题。

为什么建议子线程不访问（更新）UI？

因为Android中的UI控件不是线程安全的，如果多线程访问UI控件那还不乱套了。

那为什么不加锁呢？

- 会降低UI访问的效率。本身UI控件就是离用户比较近的一个组件，加锁之后自然会发生阻塞，那么UI访问的效率会降低，最终反应到用户端就是这个手机有点卡。
- 太复杂了。本身UI访问时一个比较简单的操作逻辑，直接创建UI，修改UI即可。如果加锁之后就让这个UI访问的逻辑变得很复杂，没必要。

所以，Android设计出了 单线程模型 来处理UI操作，再搭配上Handler，是一个比较合适的解决方案。

子线程访问UI的 崩溃原因 和 解决办法？

崩溃发生在ViewRootImpl类的checkThread方法中：

```
void checkThread() {  
    if (mThread != Thread.currentThread()) {  
        throw new CalledFromWrongThreadException(  
            "Only the original thread that created a view hierarchy can touch its views.");  
    }  
}
```

其实就是判断了当前线程 是否是 ViewRootImpl创建时候的线程，如果不是，就会崩溃。

而ViewRootImpl创建的时机就是界面被绘制的时候，也就是onResume之后，所以如果在子线程进行UI更新，就会发现当前线程（子线程）和View创建的线程（主线程）不是同一个线程，发生崩溃。

解决办法有三种：

- 1、在新建视图的线程进行这个视图的UI更新，主线程创建View，主线程更新View。
- 2、在ViewRootImpl创建之前进行子线程的UI更新，比如onCreate方法中进行子线程更新UI。
- 3、子线程切换到主线程进行UI更新，比如Handler、view.post方法。

MessageQueue是干嘛呢？用的什么数据结构来存储数据？

看名字应该是个队列结构，队列的特点是什么？先进先出，一般在队尾增加数据，在队首进行取数据或者删除数据。

那Handler中的消息似乎也满足这样的特点，先发的消息肯定就会先被处理。但是，Handler中还有比较特殊的情况，比如延时消息。

延时消息的存在就让这个队列有些特殊性了，并不能完全保证先进先出，而是需要根据时间来判断，所以Android中采用了链表的形式来实现这个队列，也方便了数据的插入。

来一起看看消息的发送过程，无论是哪种方法发送消息，都会走到sendMessageDelayed方法。

```
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;
    return enqueueMessage(queue, msg, uptimeMillis);
}

```

`sendMessageDelayed`方法主要计算了消息需要被处理的时间，如果`delayMillis`为0，那么消息的处理时间就是当前时间。

然后就是关键方法`enqueueMessage`。

```

boolean enqueueMessage(Message msg, long when) {
    synchronized (this) {
        msg.markInUse();
        msg.when = when;
        Message p = mMessages;
        boolean needWake;
        if (p == null || when == 0 || when < p.when) {
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
        } else {
            needWake = mBlocked && p.target == null && msg.isAsynchronous();
            Message prev;
            for (;;) {
                prev = p;
                p = p.next;
                if (p == null || when < p.when) {
                    break;
                }
                if (needWake && p.isAsynchronous()) {
                    needWake = false;
                }
            }
            msg.next = p;
            prev.next = msg;
        }

        if (needWake) {
            nativeWake(mPtr);
        }
    }
    return true;
}

```

不懂得地方先不看，只看我们想看的：

- 首先设置了`Message`的`when`字段，也就是代表了这个消息的处理时间。
- 然后判断当前队列是不是为空，是不是即时消息，是不是执行时间`when`大于表头的消息时间，满足任意一个，就把当前消息`msg`插入到表头。
- 否则，就需要遍历这个队列，也就是链表，找出`when`小于某个节点的`when`，找到后插入。

好了，其他内容暂且不看，总之，插入消息就是通过消息的执行时间，也就是`when`字段，来找到合适的位置插入链表。

具体方法就是通过死循环，使用快慢指针`p`和`prev`，每次向后移动一格，直到找到某个节点`p`的`when`大于我们要插入消息的`when`字段，则插入到`p`和`prev`之间。或者遍历到链表结束，插入到链表结尾。

所以，`MessageQueue`就是一个用于存储消息、用链表实现的特殊队列结构。

延迟消息是怎么实现的？

总结上述内容，延迟消息的实现主要跟消息的统一存储方法有关，也就是上文说过的`enqueueMessage`方法。

无论是即时消息还是延迟消息，都是计算出具体的时间，然后作为消息的`when`字段进程赋值。

然后在`MessageQueue`中找到合适的位置（安排`when`小到大排列），并将消息插入到`MessageQueue`中。

这样，`MessageQueue`就是一个按照消息时间排列的一个链表结构。

MessageQueue的消息怎么被取出来的？

刚才说过了消息的存储，接下来看看消息的取出，也就是`queue.next`方法。

```
Message next() {
    for (;;) {
        if (nextPollTimeoutMillis != 0) {
            Binder.flushPendingCommands();
        }

        nativePollOnce(ptr, nextPollTimeoutMillis);

        synchronized (this) {
            // Try to retrieve the next message. Return if found.
            final long now = SystemClock.uptimeMillis();
            Message prevMsg = null;
            Message msg = mMessages;
            if (msg != null && msg.target == null) {
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
            if (msg != null) {
                if (now < msg.when) {
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALUE);
                } else {
                    // Got a message.
                    mBlocked = false;
                    if (prevMsg != null) {
                        prevMsg.next = msg.next;
                    } else {
                        mMessages = msg.next;
                    }
                    msg.next = null;
                    msg.markInUse();
                    return msg;
                }
            } else {
                if (nextPollTimeoutMillis != 0) {
                    Binder.flushPendingCommands();
                }
            }
        }
    }
}
```

```

        // No more messages.
        nextPollTimeoutMillis = -1;
    }
}
}
}
}

```

奇怪，为什么取消息也是用的死循环呢？

其实死循环就是为了保证一定要返回一条消息，如果没有可用消息，那么就阻塞在这里，一直到有新消息的到来。

其中，`nativePollOnce`方法就是阻塞方法，`nextPollTimeoutMillis`参数就是阻塞的时间。

那什么时候会阻塞呢？两种情况：

1、有消息，但是当前时间小于消息执行时间，也就是代码中的这一句：

```

if (now < msg.when) {
    nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALUE);
}

```

这时候阻塞时间就是消息时间减去当前时间，然后进入下一次循环，阻塞。

2、没有消息的时候，也就是上述代码的最后一句：

```

if (msg != null) {}
else {
    // No more messages.
    nextPollTimeoutMillis = -1;
}

```

-1就代表一直阻塞。

MessageQueue没有消息时候会怎样？阻塞之后怎么唤醒呢？说说pipe/epoll机制？

接着上文的逻辑，当消息不可用或者没有消息的时候就会阻塞在`next`方法，而阻塞的办法是通过pipe/epoll机制。

epoll机制是一种IO多路复用的机制，具体逻辑就是一个进程可以监视多个描述符，当某个描述符就绪（一般是读就绪或者写就绪），能够通知程序进行相应的读写操作，这个读写操作是阻塞的。在Android中，会创建一个Linux管道（Pipe）来处理阻塞和唤醒。

- 当消息队列为空，管道的读端等待管道中有新内容可读，就会通过epoll机制进入阻塞状态。
- 当有消息要处理，就会通过管道的写端写入内容，唤醒主线程。

那什么时候会怎么唤醒消息队列线程呢？

还记得刚才插入消息的`enqueueMessage`方法中有个`needWake`字段吗，很明显，这个就是表示是否唤醒的字段。

其中还有个字段是`mBlocked`，看字面意思是阻塞的意思，去代码里面找找：

```
Message next() {
    for (;;) {
        synchronized (this) {
            if (msg != null) {
                if (now < msg.when) {
                    nextPollTimeoutMillis = (int) Math.min(msg.when - now, Integer.MAX_VALUE);
                } else {
                    // Got a message.
                    mBlocked = false;
                    return msg;
                }
            }
            if (pendingIdleHandlerCount <= 0) {
                // No idle handlers to run. Loop and wait some more.
                mBlocked = true;
                continue;
            }
        }
    }
}
```

在获取消息的方法`next`中，有两个地方对`mBlocked`赋值：

- 当获取到消息的时候，`mBlocked`赋值为`false`，表示不阻塞。
- 当没有消息要处理，也没有`idleHandler`要处理的时候，`mBlocked`赋值为`true`，表示阻塞。

好了，确实这个字段就表示是否阻塞的意思，再去看看`enqueueMessage`方法中，唤醒机制：

```
boolean enqueueMessage(Message msg, long when) {
    synchronized (this) {
        boolean needWake;
        if (p == null || when == 0 || when < p.when) {
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
        } else {
            needWake = mBlocked && p.target == null && msg.isAsynchronous();
            Message prev;
            for (;;) {
                prev = p;
                p = p.next;
                if (p == null || when < p.when) {
                    break;
                }
                if (needWake && p.isAsynchronous()) {
                    needWake = false;
                }
            }
            msg.next = p;
            prev.next = msg;
        }

        if (needWake) {
            nativeWake(mPtr);
        }
    }
}
```



```
return true;
}
```

- 当链表为空或者时间小于表头消息时间，那么就插入表头，并且设置是否唤醒为mBlocked。

再结合上述的例子，也就是当有新消息要插入表头了，这时候如果之前是阻塞状态（`mBlocked=true`），那么就要唤醒线程了。

- 否则，就需要取链表中找到某个节点并插入消息，在这之前需要赋值`needWake = mBlocked && p.target == null && msg.isAsynchronous()`

也就是在插入消息之前，需要判断是否阻塞，并且表头是不是屏障消息，并且当前消息是不是异步消息。也就是如果现在是同步屏障模式下，那么要插入的消息又刚好是异步消息，那就不用管插入消息问题了，直接唤醒线程，因为异步消息需要先执行。

- 最后一点，是在循环里，如果发现之前就存在异步消息，那就还是设置是否唤醒为false。

意思就是，如果之前有异步消息了，那肯定之前就唤醒过了，这时候就不需要再次唤醒了。

最后根据needWake的值，决定是否调用`nativeWake`方法唤醒`next()`方法。

同步屏障和异步消息是怎么实现的？

其实在Handler机制中，有三种消息类型：

同步消息。也就是普通的消息。

异步消息。通过`setAsynchronous(true)`设置的消息。

同步屏障消息。通过`postSyncBarrier`方法添加的消息，特点是`target`为空，也就是没有对应的handler。

这三者之间的关系如何呢？

- 正常情况下，同步消息和异步消息都是正常被处理，也就是根据时间`when`来取消息，处理消息。
- 当遇到同步屏障消息的时候，就开始从消息队列里面去找异步消息，找到了再根据时间决定阻塞还是返回消息。

也就是说同步屏障消息不会被返回，他只是一个标志，一个工具，遇到它就代表要去先行处理异步消息了。

所以同步屏障和异步消息的存在的意义就在于有些消息需要“加急处理”。

同步屏障和异步消息有具体的使用场景吗？

使用场景就很多了，比如绘制方法`scheduleTraversals`。

```
void scheduleTraversals() {
    if (!mTraversalScheduled) {
        mTraversalScheduled = true;
        // 同步屏障，阻塞所有的同步消息
        mTraversalBarrier = mHandler.getLooper().getQueue().postSyncBarrier();
        // 通过 Choreographer 发送绘制任务
        mChoreographer.postCallback(
            Choreographer.CALLBACK_TRAVERSAL, mTraversalRunnable, null);
    }
}

Message msg = mHandler.obtainMessage(MSG_DO_SCHEDULE_CALLBACK, action);
msg.arg1 = callbackType;
msg.setAsynchronous(true);
mHandler.sendMessageAtTime(msg, dueTime);
```

在该方法中加入了同步屏障，后续加入一个异步消息`MSG_DO_SCHEDULE_CALLBACK`，最后会执行到`FrameDisplayEventReceiver`，用于申请VSYNC信号。

更多Choreographer相关内容可以看看这篇文章：

<https://www.jianshu.com/p/86d00bbdaf60>

Message消息被分发之后会怎么处理？消息怎么复用的？

再看看loop方法，在消息被分发之后，也就是执行了`dispatchMessage`方法之后，还偷偷做了一个操作——`recycleUnchecked`。

```
public static void loop() {
    for (;;) {
        Message msg = queue.next(); // might block

        try {
            msg.target.dispatchMessage(msg);
        }

        msg.recycleUnchecked();
    }
}
```

```
//Message.java
private static Message sPool;
private static final int MAX_POOL_SIZE = 50;
```

```
void recycleUnchecked() {
    flags = FLAG_IN_USE;
    what = 0;
    arg1 = 0;
    arg2 = 0;
    obj = null;
    replyTo = null;
    sendingUid = UID_NONE;
    workSourceUid = UID_NONE;
    when = 0;
    target = null;
    callback = null;
    data = null;
```

```
synchronized (sPoolSync) {
```

```

        if (sPoolSize < MAX_POOL_SIZE) {
            next = sPool;
            sPool = this;
            sPoolSize++;
        }
    }
}

```

在`recycleUnchecked`方法中，释放了所有资源，然后将当前的空消息插入到sPool表头。

这里的sPool就是一个消息对象池，它也是一个链表结构的消息，最大长度为50。

那么Message又是怎么复用的呢？在Message的实例化方法`obtain`中：

```

public static Message obtain() {
    synchronized (sPoolSync) {
        if (sPool != null) {
            Message m = sPool;
            sPool = m.next;
            m.next = null;
            m.flags = 0; // clear in-use flag
            sPoolSize--;
            return m;
        }
    }
    return new Message();
}

```

直接复用消息池sPool中的第一条消息，然后sPool指向下一个节点，消息池数量减一。

Looper是干嘛呢？怎么获取当前线程的Looper？为什么不直接用Map存储线程和对象呢？

在Handler发送消息之后，消息就被存储到`MessageQueue`中，而Looper就是一个管理消息队列的角色。Looper会从`MessageQueue`中不断的查找消息，也就是`loop`方法，并将消息交回给Handler进行处理。

而Looper的获取就是通过ThreadLocal机制:

```

static final ThreadLocal<Looper> sThreadLocal = new ThreadLocal<Looper>();

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 static @Nullable Looper myLooper() {
    return sThreadLocal.get();
}

```

通过`prepare`方法创建Looper并且加入到`sThreadLocal`中，通过`myLooper`方法从`sThreadLocal`中获取Looper。

ThreadLocal运行机制？这种机制设计的好处？

下面就具体说说`ThreadLocal`运行机制。

```
//ThreadLocal.java
public T get() {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null) {
        ThreadLocalMap.Entry e = map.getEntry(this);
        if (e != null) {
            @SuppressWarnings("unchecked")
            T result = (T)e.value;
            return result;
        }
    }
    return setInitialValue();
}

public void set(T value) {
    Thread t = Thread.currentThread();
    ThreadLocalMap map = getMap(t);
    if (map != null)
        map.set(this, value);
    else
        createMap(t, value);
}
```

从`ThreadLocal`类中的`get`和`set`方法可以大致看出来，有一个`ThreadLocalMap`变量，这个变量存储着键值对形式的数据。

- `key`为`this`，也就是当前`ThreadLocal`变量。
- `value`为`T`，也就是要存储的值。

然后继续看看`ThreadLocalMap`哪来的，也就是`getMap`方法：

```
//ThreadLocal.java
ThreadLocalMap getMap(Thread t) {
    return t.threadLocals;
}

//Thread.java
ThreadLocal.ThreadLocalMap threadLocals = null;
```

原来这个`ThreadLocalMap`变量是存储在线程类`Thread`中的。

所以`ThreadLocal`的基本机制就搞清楚了：

在每个线程中都有一个`threadLocals`变量，这个变量存储着`ThreadLocal`和对应的需要保存的对象。

这样带来的好处就是，在不同的线程，访问同一个`ThreadLocal`对象，但是能获取到的值却不一样。

挺神奇的是不是，其实就是其内部获取到的Map不同，Map和Thread绑定，所以虽然访问的是同一个ThreadLocal对象，但是访问的Map却不是同一个，所以取得值也不一样。

这样做有什么好处呢？为什么不直接用Map存储线程和对象呢？

打个比方：

- ThreadLocal就是老师。
- Thread就是同学。
- Looper（需要的值）就是铅笔。

现在老师买了一批铅笔，然后想把这些铅笔发给同学们，怎么发呢？两种办法：

1、老师把每个铅笔上写好每个同学的名字，放到一个大盒子里面去（map），用的时候就让同学们自己来找。

这种做法就是Map里面存储的是同学和铅笔，然后用的时候通过同学来从这个Map里找铅笔。

这种做法就有点像使用一个Map，存储所有的线程和对象，不好的地方就在于会很混乱，每个线程之间有了联系，也容易造成内存泄漏。

2、老师把每个铅笔直接发给每个同学，放到同学的口袋里（map），用的时候每个同学从口袋里面拿出铅笔就可以了。

这种做法就是Map里面存储的是老师和铅笔，然后用的时候老师说一声，同学只需要从口袋里拿出来就行了。

很明显这种做法更科学，这也就是ThreadLocal的做法，因为铅笔本身就是同学自己在用，所以一开始就把铅笔交给同学自己保管是最好的，每个同学之间进行隔离。

还有哪些地方运用到了ThreadLocal机制？

比如：Choreographer。

```
public final class Choreographer {  
  
    // Thread local storage for the choreographer.  
    private static final ThreadLocal<Choreographer> sThreadInstance =  
        new ThreadLocal<Choreographer>() {  
        @Override  
        protected Choreographer initialValue() {  
            Looper looper = Looper.myLooper();  
            if (looper == null) {  
                throw new IllegalStateException("The current thread must have a looper!");  
            }  
            Choreographer choreographer = new Choreographer(looper, VSYNC_SOURCE_APP);  
            if (looper == Looper.getMainLooper()) {  
                mMainInstance = choreographer;  
            }  
            return choreographer;  
        }  
    };  
  
    private static volatile Choreographer mMainInstance;  
}
```

`Choreographer`主要是主线程用的，用于配合 VSYNC中断信号。

所以这里使用`ThreadLocal`更多的意义在于完成线程单例的功能。

可以多次创建Looper吗？

Looper的创建是通过`Looper.prepare`方法实现的，而在`prepare`方法中就判断了，当前线程是否存在Looper对象，如果有，就会直接抛出异常：

```
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));
}

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

所以同一个线程，只能创建一个Looper，多次创建会报错。

Looper中的quitAllowed字段是啥？有什么用？

按照字面意思就是是否允许退出，我们看看他都在哪些地方用到了：

```
void quit(boolean safe) {
    if (!mQuitAllowed) {
        throw new IllegalStateException("Main thread not allowed to quit.");
    }

    synchronized (this) {
        if (mQuitting) {
            return;
        }
        mQuitting = true;

        if (safe) {
            removeAllFutureMessagesLocked();
        } else {
            removeAllMessagesLocked();
        }
    }
}
```

哦，就是这个`quit`方法用到了，如果这个字段为`false`，代表不允许退出，就会报错。

但是这个`quit`方法又是干嘛的呢？从来没用过呢。还有这个`safe`又是啥呢？

其实看名字就差不多能了解了，`quit`方法就是退出消息队列，终止消息循环。

- 首先设置了 `mQuitting` 字段为 `true`。
- 然后判断是否安全退出，如果安全退出，就执行 `removeAllFutureMessagesLocked` 方法，它内部的逻辑是清空所有的延迟消息，之前没处理的非延迟消息还是需要取处理，然后设置非延迟消息的下一个节点为空（`p.next=null`）。
- 如果不是安全退出，就执行 `removeAllMessagesLocked` 方法，直接清空所有的消息，然后设置消息队列指向空（`mMessages = null`）。

然后看看当调用 `quit` 方法之后，消息的发送和处理：

```
//消息发送
boolean enqueueMessage(Message msg, long when) {
    synchronized (this) {
        if (mQuitting) {
            IllegalStateException e = new IllegalStateException(
                msg.target + " sending message to a Handler on a dead thread");
            Log.w(TAG, e.getMessage(), e);
            msg.recycle();
            return false;
        }
    }
}
```

当调用了 `quit` 方法之后，`mQuitting` 为 `true`，消息就发不出去了，会报错。

再看看消息的处理，`loop` 和 `next` 方法：

```
Message next() {
    for (;;) {
        synchronized (this) {
            if (mQuitting) {
                dispose();
                return null;
            }
        }
    }
}

public static void loop() {
    for (;;) {
        Message msg = queue.next();
        if (msg == null) {
            // No message indicates that the message queue is quitting.
            return;
        }
    }
}
```

很明显，当 `mQuitting` 为 `true` 的时候，`next` 方法返回 `null`，那么 `loop` 方法中就会退出死循环。

那么这个 `quit` 方法一般是什么时候使用呢？

- 主线程中，一般情况下肯定不能退出，因为退出后主线程就停止了。所以当APP需要退出的时候，就会调用 `quit` 方法，涉及到的消息是 `EXIT_APPLICATION`，大家可以搜索下。
- 子线程中，如果消息都处理完了，就需要调用 `quit` 方法停止消息循环。

Looper.loop方法是死循环，为什么不会卡死（ANR）？

关于这个问题，强烈建议看看Gityuan的回答：<https://www.zhihu.com/question/34652589>

我大致总结下：

- 1、主线程本身就是需要一直运行的，因为要处理各个View，界面变化。所以需要这个死循环来保证主线程一直执行下去，不会被退出。
- 2、真正会卡死的操作是在某个消息处理的时候操作时间过长，导致掉帧、ANR，而不是loop方法本身。
- 3、在主线程以外，会有其他的线程来处理接受其他进程的事件，比如Binder线程（`ApplicationThread`），会接受AMS发送来的事件。
- 4、在收到跨进程消息后，会交给主线程的Handler再进行消息分发。所以Activity的生命周期都是依靠主线程的`Looper.loop`，当收到不同Message时则采用相应措施，比如收到`msg=H.LAUNCH_ACTIVITY`，则调用`ActivityThread.handleLaunchActivity()`方法，最终执行到`onCreate`方法。
- 5、当没有消息的时候，会阻塞在`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。所以死循环也不会特别消耗CPU资源。

Message是怎么找到它所属的Handler然后进行分发的？

在`loop`方法中，找到要处理的Message，然后调用了这么一句代码处理消息：

```
msg.target.dispatchMessage(msg);
```

所以是将消息交给了`msg.target`来处理，那么这个`target`是啥呢？

找找它的来头：

```
//Handler
private boolean enqueueMessage(MessageQueue queue, Message msg, long uptimeMillis) {
    msg.target = this;
    return queue.enqueueMessage(msg, uptimeMillis);
}
```

在使用Handler发送消息的时候，会设置`msg.target = this`，所以`target`就是当初把消息加到消息队列的那个Handler。

Handler 的 post(Runnable) 与 sendMessage 有什么区别

Handler中主要的发送消息可以分为两种：

1、`post(Runnable)`

2、`sendMessage`

```
public final boolean post(@NonNull Runnable r) {
    return sendMessageDelayed(getPostMessage(r), 0);
}
private static Message getPostMessage(Runnable r) {
    Message m = Message.obtain();
    m.callback = r;
    return m;
}
```

通过`post`的源码可知，其实`post`和`sendMessage`的区别就在于：

`post`方法给Message设置了一个`callback`。

那么这个`callback`有什么用呢？我们再转到消息处理的方法`dispatchMessage`中看看：

```
public void dispatchMessage(@NonNull Message msg) {
    if (msg.callback != null) {
        handleCallback(msg);
    } else {
        if (mCallback != null) {
            if (mCallback.handleMessage(msg)) {
                return;
            }
        }
        handleMessage(msg);
    }
}

private static void handleCallback(Message message) {
    message.callback.run();
}
```

这段代码可以分为三部分看：

- 1、如果`msg.callback`不为空，也就是通过`post`方法发送消息的时候，会把消息交给这个`msg.callback`进行处理，然后就没有后续了。
- 2、如果`msg.callback`为空，也就是通过`sendMessage`发送消息的时候，会判断Handler当前的`mCallback`是否为空，如果不为空就交给`Handler.Callback.handleMessage`处理。
- 3、如果`mCallback.handleMessage`返回`true`，则无后续了。
- 4、如果`mCallback.handleMessage`返回`false`，则调用handler类重写的`handleMessage`方法。

所以`post(Runnable)`与`sendMessage`的区别就在于后续消息的处理方式，是交给`msg.callback`还是`Handler.Callback`或者`Handler.handleMessage`。

Handler.Callback.handleMessage 和 Handler.handleMessage 有什么不一样？为什么这么设计？

接着上面的代码说，这两个处理方法的区别在于`Handler.Callback.handleMessage`方法是否返回`true`：

如果为`true`，则不再执行`Handler.handleMessage`。

如果为`false`，则两个方法都要执行。

那么什么时候有`Callback`，什么时候没有呢？这涉及到两种Handler的创建方式：

```
val handler1= object : Handler(){
    override fun handleMessage(msg: Message) {
        super.handleMessage(msg)
    }
}

val handler2 = Handler(object : Handler.Callback {
    override fun handleMessage(msg: Message): Boolean {
        return true
    }
})
```

常用的方法就是第1种，派生一个Handler的子类并重写`handleMessage`方法。而第2种就是系统给我们提供了一种不需要派生子类的使用方法，只需要传入一个`Callback`即可。

Handler、Looper、MessageQueue、线程是一一对应关系吗？

一个线程只会有一个Looper对象，所以线程和Looper是一一对应的。

`MessageQueue`对象是在`new Looper`的时候创建的，所以Looper和`MessageQueue`是一一对应的。

Handler的作用只是将消息加到`MessageQueue`中，并后续取出消息后，根据消息的`target`字段分发给当初的那个handler，所以Handler对于Looper是可以多对一的，也就是多个Handler对象都可以用同一个线程、同一个Looper、同一个`MessageQueue`。

总结：`Looper`、`MessageQueue`、线程是一一对应关系，而它们与Handler是可以一对多的。

ActivityThread中做了哪些关于Handler的工作？（为什么主线程不需要单独创建Looper）

主要做了两件事：

1、在`main`方法中，创建了主线程的`Looper`和`MessageQueue`，并且调用`loop`方法开启了主线程的消息循环。

```

public static void main(String[] args) {

    Looper.prepareMainLooper();

    if (sMainThreadHandler == null) {
        sMainThreadHandler = thread.getHandler();
    }

    Looper.loop();

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

```

2、创建了一个Handler来进行四大组件的启动停止等事件处理。

```

final H mH = new H();

class H extends Handler {
    public static final int BIND_APPLICATION      = 110;
    public static final int EXIT_APPLICATION      = 111;
    public static final int RECEIVER             = 113;
    public static final int CREATE_SERVICE       = 114;
    public static final int STOP_SERVICE         = 116;
    public static final int BIND_SERVICE        = 121;
}

```

IdleHandler是啥？有什么使用场景？

之前说过，当MessageQueue没有消息的时候，就会阻塞在next方法中，其实在阻塞之前，MessageQueue还会做一件事，就是检查是否存在IdleHandler，如果有，就会去执行它的queueIdle方法。

```

private IdleHandler[] mPendingIdleHandlers;

Message next() {
    int pendingIdleHandlerCount = -1;
    for (;;) {
        synchronized (this) {
            //当消息执行完毕，就设置pendingIdleHandlerCount
            if (pendingIdleHandlerCount < 0
                && (mMessages == null || now < mMessages.when)) {
                pendingIdleHandlerCount = mIdleHandlers.size();
            }

            //初始化mPendingIdleHandlers
            if (mPendingIdleHandlers == null) {
                mPendingIdleHandlers = new IdleHandler[Math.max(pendingIdleHandlerCount, 4)];
            }
            //mIdleHandlers转为数组
            mPendingIdleHandlers = mIdleHandlers.toArray(mPendingIdleHandlers);
        }

        // 遍历数组，处理每个IdleHandler
        for (int i = 0; i < pendingIdleHandlerCount; i++) {
            final IdleHandler idler = mPendingIdleHandlers[i];
            mPendingIdleHandlers[i] = null; // release the reference to the handler

            boolean keep = false;
            try {
                keep = idler.queueIdle();
            } catch (Throwable t) {
                Log.wtf(TAG, "IdleHandler threw exception", t);
            }

            //如果queueIdle方法返回false，则处理完就删除这个IdleHandler
            if (!keep) {
                synchronized (this) {
                    mIdleHandlers.remove(idler);
                }
            }
        }
    }
}

```

```

    }
}

// Reset the idle handler count to 0 so we do not run them again.
pendingIdleHandlerCount = 0;
}
}

```

当没有消息处理的时候，就会去处理这个 `mIdleHandlers` 集合里面的每个 `IdleHandler` 对象，并调用其 `queueIdle` 方法。最后根据 `queueIdle` 返回值判断是否用完删除当前的 `IdleHandler`。

然后看看 `IdleHandler` 是怎么加进去的：

```

Looper.myQueue().addIdleHandler(new IdleHandler() {
@Override
public boolean queueIdle() {
//做事情
return false;
}
});

public void addIdleHandler(@NonNull IdleHandler handler) {
if (handler == null) {
throw new NullPointerException("Can't add a null IdleHandler");
}
synchronized (this) {
mIdleHandlers.add(handler);
}
}
}

```

ok，综上所述，`IdleHandler` 就是当消息队列里面没有当前要处理的消息了，需要堵塞之前，可以做一些空闲任务的处理。

常见的使用场景有：启动优化。

我们一般会把一些事件（比如界面view的绘制、赋值）放到 `onCreate` 方法或者 `onResume` 方法中。但是这两个方法其实都是在界面绘制之前调用的，也就是说一定程度上这两个方法的耗时会影响到启动时间。

所以我们可以把一些操作放到 `IdleHandler` 中，也就是界面绘制完成之后才去调用，这样就能减少启动时间了。

但是，这里需要注意下可能会有坑。

如果使用不当，`IdleHandler` 会一直不执行，比如在View的 `onDraw` 方法里面无限制的直接或者间接调用View的 `invalidate` 方法。

其原因就在于 `onDraw` 方法中执行 `invalidate`，会添加一个同步屏障消息，在等到异步消息之前，会阻塞在 `next` 方法，而等到 `FrameDisplayEventReceiver` 异步任务之后又会执行 `onDraw` 方法，从而无限循环。

具体可以看看这篇文章：https://mp.weixin.qq.com/s/dh_71i8J5ShpgxgWN5SPEw

HandlerThread是啥？有什么使用场景？

直接看源码：

```
public class HandlerThread extends Thread {
    @Override
    public void run() {
        Looper.prepare();
        synchronized (this) {
            mLooper = Looper.myLooper();
            notifyAll();
        }
        Process.setThreadPriority(mPriority);
        onLooperPrepared();
        Looper.loop();
    }
}
```

哦，原来如此。`HandlerThread`就是一个封装了Looper的Thread类。

就是为了让我们在子线程里面更方便的使用Handler。

这里的加锁就是为了保证线程安全，获取当前线程的Looper对象，获取成功之后再通过`notifyAll`方法唤醒其他线程，那哪里调用了`wait`方法呢？

```
public Looper getLooper() {
    if (!isAlive()) {
        return null;
    }

    // If the thread has been started, wait until the looper has been created.
    synchronized (this) {
        while (isAlive() && mLooper == null) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
    }
    return mLooper;
}
```

就是`getLooper`方法，所以wait的意思就是等待Looper创建好，那边创建好之后再通知这边正确返回Looper。

IntentService是啥？有什么使用场景？

老规矩，直接看源码：

```
public abstract class IntentService extends Service {

    private final class ServiceHandler extends Handler {
        public ServiceHandler(Looper looper) {
            super(looper);
        }
    }

    @Override
    public void handleMessage(Message msg) {
        onHandleIntent((Intent)msg.obj);
    }
}
```

```

        stopSelf(msg.arg1);
    }
}

@Override
public void onCreate() {
    super.onCreate();
    HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");
    thread.start();

    mServiceLooper = thread.getLooper();
    mServiceHandler = new ServiceHandler(mServiceLooper);
}

@Override
public void onStart(@Nullable Intent intent, int startId) {
    Message msg = mServiceHandler.obtainMessage();
    msg.arg1 = startId;
    msg.obj = intent;
    mServiceHandler.sendMessage(msg);
}

```

理一下这个源码：

- 首先，这是一个Service。
- 并且内部维护了一个HandlerThread，也就是有完整的Looper在运行。
- 还维护了一个子线程的ServiceHandler。
- 启动Service后，会通过Handler执行onHandleIntent方法。
- 完成任务后，会自动执行stopSelf停止当前Service。

所以，这就是一个可以在子线程进行耗时任务，并且在任务执行后自动停止的Service。

BlockCanary使用过吗？说说原理

BlockCanary是一个用来检测应用卡顿耗时的三方库。

上文说过，View的绘制也是通过Handler来执行的，所以如果能知道每次Handler处理消息的时间，就能知道每次绘制的耗时了？那Handler消息的处理时间怎么获取呢？

再去loop方法中找找细节：

```

public static void loop() {
    for (;;) {
        // This must be in a local variable, in case a UI event sets the logger
        Printer logging = me.mLogging;
        if (logging != null) {
            logging.println(">>>> Dispatching to " + msg.target + " " +
                msg.callback + ": " + msg.what);
        }

        msg.target.dispatchMessage(msg);

        if (logging != null) {

```

```
logging.println("<<<< Finished to " + msg.target + " " + msg.callback);
    }
}
}
```

可以发现，`loop`方法内有一个Printer类，在`dispatchMessage`处理消息的前后分别打印了两次日志。

那我们把这个日志类Printer替换成我们自己的Printer，然后统计两次打印日志的时间不就相当于处理消息的时间了？

```
Looper.getMainLooper().setMessageLogging(mainLooperPrinter);

public void setMessageLogging(@Nullable Printer printer) {
    mLogging = printer;
}
```

这就是BlockCanary的原理。

具体介绍可以看看作者的说明：

<http://blog.zhayifan.cn/2016/01/16/BlockCanaryTransparentPerformanceMonitor/>

说说Hanlder内存泄露问题。

这也是常常被问的一个问题，Handler内存泄露的原因是什么？

"内部类持有了外部类的引用，也就是Hanlder持有了Activity的引用，从而导致无法被回收呗。"

其实这样回答是错误的，或者说没回答到点子上。

我们必须找到那个最终的引用者，不会被回收的引用者，其实就是主线程，这条完整引用链应该是这样：

主线程 —> threadlocal —> Looper —> MessageQueue —> Message —> Handler —> Activity

具体分析可以看看我之前写的这篇文章：

<https://juejin.cn/post/6909362503898595342>

利用Handler机制设计一个不崩溃的App？

主线程崩溃，其实都是发生在消息的处理内，包括生命周期、界面绘制。

所以如果我们能控制这个过程，并且在发生崩溃后重新开启消息循环，那么主线程就能继续运行。

```
Handler(Looper.getMainLooper()).post {
    while (true) {
        //主线程异常拦截
        try {
            Looper.loop()
        } catch (e: Throwable) {
        }
    }
}
```



```
}  
}  
}
```

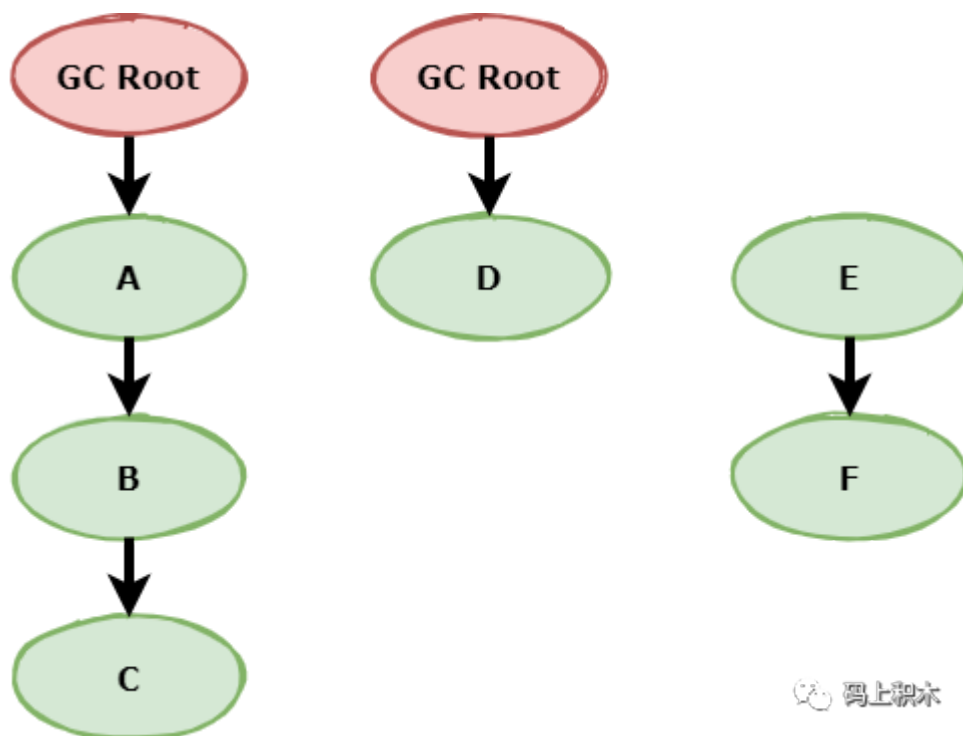
还有一些特殊情况处理，比如onCreate内发生崩溃，具体可以看看文章
《能否让APP永不崩溃》<https://juejin.cn/post/6904283635856179214>

JVM中如何决定对象是否可以回收

JVM中通过可达性分析算法来决定对象是否可以回收。

具体做法就是把内存中所有对象之间的引用关系看做一条关系链，比如A持有B的引用，B持有C的引用。而在JVM中有一组对象作为GC Root，也就是根节点，然后从这些节点开始往下搜索，查看引用链，最后判断对象的引用链是否可达来决定对象是否可以被回收。

为了方便大家理解，我画了一张图来说明：



很明显，ABCD四个引用都是GCRoot可达的，通俗点讲，就是跟GCRoot直接或间接有关系，有线连着的。而EF虽然直接连着线，但是他们和GCRoot是没关系的，也就是GCRoot不可达的对象组。

所以当GC发生的时候，EF就会被回收。

GC发生的内存区域

在说GC发生的内存区域之前，我们先聊聊JVM中的内存分配。

在JVM中，主要有内存分成了五个数据区域：

- 程序计数器：线程私有，主要用作记录当前线程执行的位置。

- 虚拟机栈：线程私有，描述Java方法执行的内存模型。
- 本地方法栈：线程私有，描述本地（ native ）方法执行的内存模型。
- 堆：存放对象实例。
- 方法区：存放类信息、常量、静态变量等。

通过上面的介绍，我们了解到前三个都是线程私有，所以会随着线程的死亡而消失。

而后面两块内存区域，也就是堆和方法区是所有线程共有的，如果不处理可能内存就会一直增长，直到超出可用内存。所以需要借助GC机制对这些区域内的无用内存进行回收，特别是堆区的内存，因为堆区就是存储对象实例的。

GC发生的时机

那具体什么时候会被回收呢？主要有两种情况：

- 1、在堆内存中分配时，如果因为可用剩余空间不足导致对象内存分配失败，这时系统会触发一次 GC。
- 2、在应用层，开发者可以调用System.gc()来请求一次 GC。

GCRoot的类型

刚才说过了可达性分析算法，所以大家应该知道GCRoot的重要性了。

GCRoot，说白了就是JVM认证的可以作为老大的人选，只有这些对象是可以作为引用链的头头，掌管并保护着有用的引用。

在Java中，有以下几种对象可以被作为GCRoot，这些对象是不会被GC的：

- Java 虚拟机栈（局部变量表）中的引用的对象。

这里又涉及到一个问题了，什么是局部变量表。

刚才说过虚拟机栈是用于支持方法调用或者执行的数据结构，具体是怎么操作的呢？

当某个方法被执行，就会在虚拟机栈中创建一个栈帧，也就是一个方法就对应着一个栈帧，栈帧会管理方法调用和执行所有的数据结构。

而栈帧中又分为几块存储空间，进行存储方法对应的不同的数据结构，比如局部变量表就是用于存储方法参数和方法内创建的局部变量。

所以这第一个GC Root 指得就是方法的参数或者方法中创建的参数。

```
public class GCTest {  
    public static void test1(){  
        //局部变量作为GCRoot  
        GCRoot root=new GCRoot();  
        System.gc();  
    }  
}
```

顺便说下栈帧中其他几个内存结构：

- 局部变量表：存储方法参数和方法内创建的局部变量。
- 操作数栈：后入先出栈。当方法执行过程中，就会通过操作数栈来进行参数传递，又或者进行加数。
- 动态连接：支持方法调用过程中的动态连接。
- 返回地址：在方法退出之后，都需要返回到方法被调用的位置，程序才能继续执行，方法返回时可能需要在栈帧中保存一些信息，用来帮助恢复它的上层方法的执行状态，而这个返回地址区域就是用于存储返回地址信息的。一般方法正常退出时，是可以将调用者的PC计数器值作为返回地址。
- 方法区中静态引用指向的对象。

这个很好理解，指得就是静态变量。

```
public class GCTest {  
    private static GCRoot root2;  
    public static void main(String[] args) {  
        //静态变量作为GCRoot  
        root2=new GCRoot();  
        System.gc();  
    }  
}
```

- 仍处于存活状态中的线程对象。

活着的线程，比如主线程，上一篇文章就说过Handler内存泄露的原因就是被主线程所引用，所以无法被回收。

```
Thread root3=new Thread(new Runnable() {  
    @Override  
    public void run() {  
        try {  
            Thread.sleep(2000);  
        } catch (InterruptedException e) {  
            e.printStackTrace();  
        }  
    }  
});  
  
public void test2(){  
    //活着的线程作为GCRoot  
    root3.start();  
    System.gc();  
}
```

- Native 方法中 JNI 引用的对象。

在JNI中有如下三种引用类型可供使用：

- 1、局部引用。
- 2、全局引用。
- 3、弱全局引用。

其中局部引用和全局引用都可以作为GC Root，不会被GC回收。

编译打包的过程中有哪些task会执行

```
//aidl 转换aidl文件为java文件
> Task :app:compileDebugAidl

//生成BuildConfig文件
> Task :app:generateDebugBuildConfig

//获取gradle中配置的资源文件
> Task :app:generateDebugResValues

// merge资源文件
> Task :app:mergeDebugResources

// merge assets文件
> Task :app:mergeDebugAssets
> Task :app:compressDebugAssets

// merge所有的manifest文件
> Task :app:processDebugManifest

//AAPT 生成R文件
> Task :app:processDebugResources

//编译kotlin文件
> Task :app:compileDebugKotlin

//javac 编译java文件
> Task :app:compileDebugJavaWithJavac

//转换class文件为dex文件
> Task :app:dexBuilderDebug

//打包成apk并签名
> Task :app:packageDebug
```

简单介绍v1、v2、v3、v4签名

之前大家比较熟知的签名工具是JDK提供的jarsigner，而apksigner是Google专门为Android提供的签名和签证工具。

其区别就在于jarsigner只能进行v1签名，而apksigner可以进行v2、v3、v4签名。

v1签名

v1签名方式主要是利用META-INF文件夹中的三个文件。

首先，将apk中除了META-INFO文件夹中的所有文件进行进行摘要写到 META-INFO/MANIFEST.MF；然后计算MANIFEST.MF文件的摘要写到CERT.SF；最后计算CERT.SF的摘要，使用私钥计算签名，将签名和开发者证书写到CERT.RSA。

所以META-INFO文件夹中这三个文件就能保证apk不会被修改。

但是缺点也很明显，META-INFO文件夹不会被签名，所以美团针对这种签名方式设计了一种多渠道打包方案：

利用pythone在META-INFO文件夹中创建一个文件，其名称就是渠道名，然后用java去读取文件名获取渠道。

v2签名

Android7.0之后，推出了v2签名，为了解决v1签名速度慢以及签名不完整的问题。

apk本质上是一个压缩包，而压缩包文件格式一般分为三块：

文件数据区，中央目录结果，中央目录结束节。

而v2要做的就是，在文件中插入一个APK签名分块，位于中央目录部分之前，如下图：



这样处理之后，文件就完成无法修改了。

v3签名

Android 9 推出了v3签名方案，和v2签名方式基本相同，不同的是在v3签名分块中添加了有关受支持的sdk版本和新旧签名信息，可以用作签名替换升级。

v4签名

Android 11 推出了v4签名方案。

v4 签名基于根据 APK 的所有字节计算得出的 Merkle 哈希树。它完全遵循 fs-verity 哈希树的结构，将签名存储在单独的.apk.idsig 文件中。

《Android开发艺术探索》

<https://juejin.cn/post/6896751245722615815>

<https://juejin.cn/post/6891911483379482637>

<https://mp.weixin.qq.com/s/kQmH2GnwW8FK-yNmWcheTA>

<https://segmentfault.com/a/1190000021357383>

<https://blog.csdn.net/lmj623565791/article/details/72859156>

<https://developer.android.google.cn/guide/components/services#Lifecycle>

http://gityuan.com/2017/03/10/job_scheduler_service/

<https://kaiwu.lagou.com/course/courseInfo.htm?courseId=67#/detail/pc?id=1856>

<https://www.zhihu.com/question/34652589>

<https://segmentfault.com/a/1190000003063859>

<https://juejin.cn/post/6844904150140977165>

<https://juejin.cn/post/6893791473121280013>

<https://www.jianshu.com/p/bfb13eb3a425>

<https://segmentfault.com/a/1190000020386580>

<https://www.jianshu.com/p/02db8b55aae9>

<https://kaiwu.lagou.com/course/courseInfo.htm?courseId=67#/detail/pc>

<https://www.runoob.com/design-pattern/design-pattern-tutorial.html>

<https://www.jianshu.com/p/ae2fe5481994>

<https://juejin.cn/post/6895369745445748749>

最后推荐一下我做的网站，方包博客: www.fang1688.cn，包含详尽的知识体系、好用的工具，还有本公众号文章合集，欢迎体验和收藏！

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器

福昕PDF编辑器