2020 Android 复习资料汇总版

目录

2020 Android 复习资料汇总版 1
一、JAVA 知识点汇总 9
1.1 JVM 9
1.1.1 JVM 工作流程 9
1.1.2 运行时数据区（Runtime Data Area） 9
1.1.3 方法指令 11
1.1.4 类加载器 11
1.1.5 垃圾回收 gc 12
1.1.5.1 对象存活判断 12
1.1.5.2 垃圾收集算法 13
1.1.5.3 垃圾收集器 14
1.1.5.4 内存模型与回收策略 15
1.2 static 17
1.3 final 17
1.4 String、StringBuffer、StringBuilder 18
1.5 异常处理 18
1.6 内部类 19
1.6.1 匿名内部类 19
1.7 多态 20
1.8 抽象和接口 20
1.9 集合框架 20
1.9.1 HashMap 21
1.9.1.1 结构图 21
1.9.1.2 HashMap 的工作原理 22
1.9.1.3 HashMap 与 HashTable 对比 23
1.9.2 ConcurrentHashMap 26
1.9.2.1 Base 1.7 26
1.9.2.2 Base 1.8 27
1.9.3 ArrayList 30
1.9.4 LinkedList 31
1.9.5 CopyOnWriteArrayList 35
1.10 反射 37
1.11 单例 38
1.11.1 饿汉式 38
1.11.2 双重检查模式 39
1.11.3 静态内部类模式 40
1.12 线程 41
1.12.1 属性 41
1.12.2 状态 42

1.12.3 状态控制 43
1.13 volatile 44
1.14 synchronized 45
1.14.1 根据获取的锁分类 46
1.14.2 原理 46
1.15 Lock 47
1.15.1 锁的分类 48
1.15.1.1 悲观锁、乐观锁 48
1.15.1.2 自旋锁、适应性自旋锁 49
1.15.1.3 死锁 50
1.16 引用类型 50
1.17 动态代理 50
1.18 元注解 57
二、Android 知识点汇总 59
2.1 Activity 59
2.1.1 生命周期 59
2.1.2 启动模式 61
2.1.3 启动过程 61
2.2 Fragment 65
2.2.1 特点 65
2.2.2 生命周期 65
2.2.3 与 Activity 通信 67
2.3 Service 68
2.3.1 启动过程 68
2.3.2 绑定过程 70
2.3.3 生命周期 71
2.3.4 启用前台服务 73
2.4 BroadcastReceiver 73
2.4.1 注册过程 73
2.5 ContentProvider 74
2.5.1 基本使用 75
2.6 数据存储 77
2.7 View 77
2.7.1 MeasureSpec 79
2.7.2 MotionEvent 81
2.7.3 VelocityTracker 82
2.7.4 GestureDetector 82
2.7.5 Scroller 84
2.7.6 View 的滑动 85
2.7.7 View 的事件分发 86
2.7.8 在 Activity 中获取某个 View 的宽高 89
2.7.9 Draw 的基本流程 91
2.7.10 自定义 View 92
2.8 进程 92

2.8.1 进程生命周期 93
2.8.2 多进程 94
2.8.3 进程存活 95
2.8.3.1 OOM_ADJ 95
2.8.3.2 进程被杀情况 96
2.8.3.3 进程保活方案 97
2.9 Parcelable 接口 97
2.9.1 使用示例 98
2.9.2 方法说明 99
2.9.3 Parcelable 与 Serializable 对比 100
2.10 IPC 100
2.10.1 IPC 方式 100
2.10.2 Binder 101
2.10.3 AIDL 通信 107
2.10.4 Messenger 110
2.11 Window / WindowManager 110
2.11.1 Window 概念与分类 110
2.11.2 Window 的内部机制 111
2.11.3 Window 的创建过程 114
2.11.3.1 Activity 的 Window 创建过程 114
2.11.3.2 Dialog 的 Window 创建过程 117
2.11.3.3 Toast 的 Window 创建过程 118
2.12 Bitmap 121
2.12.1 配置信息与压缩方式 121
2.12.2 常用操作 123
2.12.2.1 裁剪、缩放、旋转、移动 123
2.12.2.2 Bitmap 与 Drawable 转换 124
2.12.2.3 保存与释放 124
2.12.2.4 图片压缩 125
2.12.3 BitmapFactory 126
2.12.3.1 Bitmap 创建流程 126
2.12.3.2 Option 类 126
2.12.3.3 基本使用 128
2.12.4 内存回收 129
2.13 屏幕适配 130
2.13.1 单位 130
2.13.2 头条适配方案 130
2.13.3 刘海屏适配 132
2.14 Context 133
2.15 SharedPreferences 135
2.15.1 获取方式 135
2.15.1.1 getPreferences 135
2.15.1.2 getDefaultSharedPreferences 136
2.15.1.3 getSharedPreferences 136

2.15.2 架构 137
2.15.3 apply / commit 139
2.15.4 注意 139
2.16 消息机制 139
2.16.1 Handler 机制 139
2.16.2 工作原理 140
2.16.2.1 ThreadLocal 141
2.16.2.2 MessageQueue 142
2.16.2.3 Looper 147
2.16.2.4 Handler 149
2.17 线程异步 149
2.17.1 AsyncTask 150
2.17.1.1 基本使用 150
2.17.1.2 工作原理 152
2.17.2 HandlerThread 155
2.17.3 IntentService 156
2.17.4 线程池 158
2.18 RecyclerView 优化 159
2.19 Webview 161
2.19.1 基本使用 161
2.19.1.1 WebView 162
2.19.1.2 WebSettings 162
2.19.1.3 WebViewClient 165
2.19.1.4 WebChromeClient 168
2.19.2 Webview 加载优化 171
2.19.3 内存泄漏 173
三、Android 扩展知识点 175
3.1 ART 175
3.1.1 ART 功能 175
3.1.1.1 预先 (AOT) 编译 175
3.1.1.2 垃圾回收优化 175
3.1.1.3 开发和调试方面的优化 176
3.1.2 ART GC 177
3.2 Apk 包体优化 178
3.2.1 Apk 组成结构 178
3.2.2 整体优化 179
3.2.3 资源优化 179
3.2.4 代码优化 180
3.2.5 .arsc 文件优化 180
3.2.6 lib 目录优化 180
3.3 Hook 181
3.3.1 基本流程 181
3.3.2 使用示例 181
3.4 Proguard 184

3.4.1 公共模板 184
3.4.2 常用的自定义混淆规则 190
3.4.3 aar 中增加独立的混淆配置 191
3.4.4 检查混淆和追踪异常 191
3.5 架构 192
3.5.1 MVC 192
3.5.2 MVP 194
3.5.3 MVVM 195
3.6 Jetpack 195
3.6.1 架构 195
3.6.2 使用示例 196
3.7 NDK 开发 201
3.7.1 JNI 基础 201
3.7.1.1 数据类型 201
3.7.1.2 String 字符串函数操作 202
3.7.1.3 常用 JNI 访问 Java 对象方法 203
3.7.2 NDK 开发 205
3.7.2.1 基础开发流程 205
3.7.2.2 System.loadLibrary() 207
3.7.3 CMake 构建 NDK 项目 212
3.7.4 常用的 Android NDK 原生 API 215
3.8 计算机网络基础 216
3.8.1 网络体系的分层结构 216
3.8.2 Http 相关 217
3.8.2.1 请求报文与响应报文 217
3.8.2.2 缓存机制 218
3.8.2.3 Https 220
3.8.2.4 Http 2.0 220
3.8.3 TCP/IP 221
3.8.3.1 三次握手 221
3.8.3.2 四次挥手 223
3.8.3.3 TCP 与 UDP 的区别 225
3.8.4 Socket 225
3.8.4.1 使用示例 225
3.9 类加载器 228
3.9.1 双亲委托模式 228
3.9.2 DexPathList 229
四、Android 开源库源码分析 230
4.1 LeakCanary 230
4.1.1 初始化注册 230
4.1.2 引用泄漏观察 233
4.1.3 Dump Heap 235
4.2 EventBus 237
4.2.1 自定义注解 237

4.2.2 注册订阅者 238
4.2.3 发送事件 241
五、设计模式汇总 244
5.1 设计模式分类 244
5.2 面向对象六大原则 245
5.3 工厂模式 246
5.4 单例模式 247
5.5 建造者模式 247
5.6 原型模式 249
5.7 适配器模式 250
5.8 观察者模式 253
5.9 代理模式 255
5.10 责任链模式 257
5.11 策略模式 258
5.12 备忘录模式 260
六、Gradle 知识点汇总 263
6.1 依赖项配置 263
七、常见面试算法题汇总 264
7.1 排序 264
7.1.1 比较排序 264
7.1.1.1 冒泡排序 264
7.1.1.2 归并排序 265
7.1.1.3 快速排序 266
7.1.2 线性排序 267
7.1.2.1 计数排序 267
7.1.2.2 桶排序 269
7.2 二叉树 270
7.2.1 顺序遍历 271
7.2.2 层次遍历 272
7.2.3 左右翻转 276
7.2.4 最大值 276
7.2.5 最大深度 277
7.2.6 最小深度 277
7.2.7 平衡二叉树 278
7.3 链表 278
7.3.1 删除节点 279
7.3.2 翻转链表 279
7.3.3 中间元素 280
7.3.4 判断是否为循环链表 281
7.3.5 合并两个已排序链表 281
7.3.6 链表排序 282
7.3.7 删除倒数第 N 个节点 284
7.3.8 两个链表是否相交 285
7.4 栈 / 队列 287

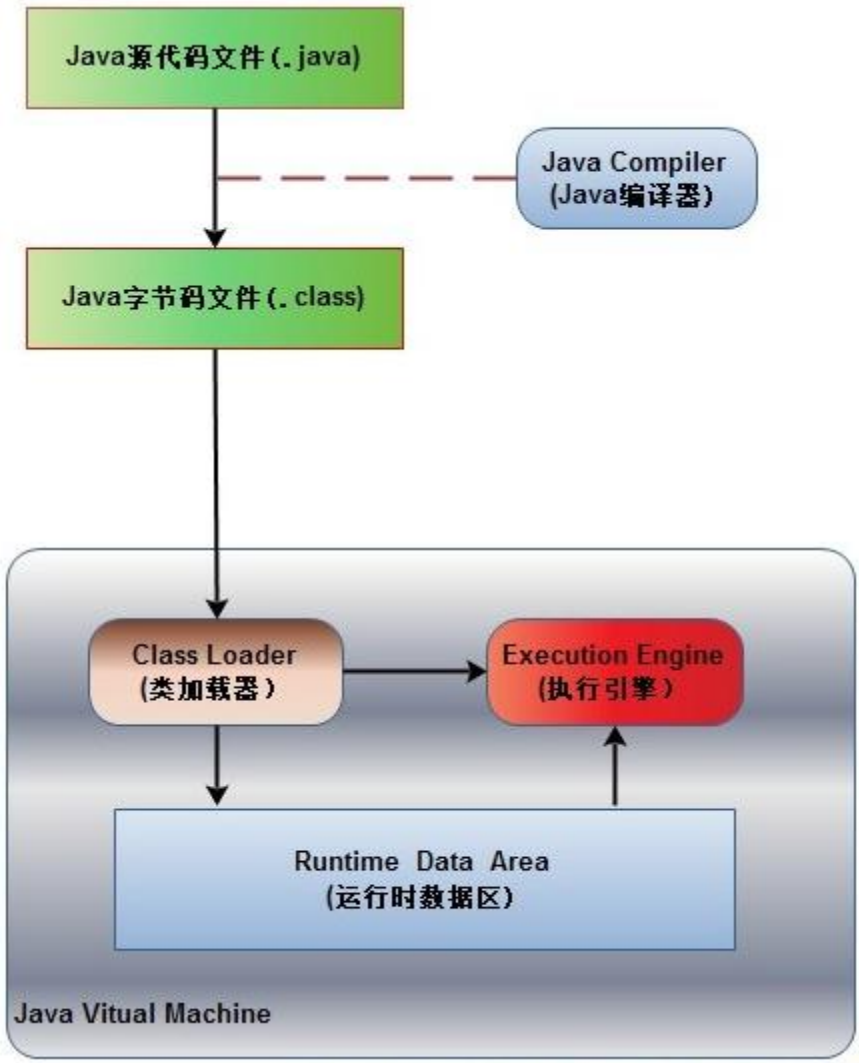
7.4.1 带最小值操作的栈 287
7.4.2 有效括号 288
7.4.3 用栈实现队列 289
7.4.4 逆波兰表达式求值 291
7.5 二分 292
7.5.1 二分搜索 292
7.5.2 X 的平方根 292
7.6 哈希表 293
7.6.1 两数之和 293
7.6.2 连续数组 294
7.6.3 最长无重复字符的子串 296
7.6.4 最多点在一条直线上 297
7.7 堆 / 优先队列 298
7.7.1 前 K 大的数 298
7.7.2 前 K 大的数 II 299
7.7.3 第 K 大的数 301
7.8 二叉搜索树 302
7.8.1 验证二叉搜索树 302
7.8.2 第 K 小的元素 303
7.9 数组 / 双指针 304
7.9.1 加一 304
7.9.2 删除元素 305
7.9.3 删除排序数组中的重复数字 306
7.9.4 我的日程安排表 I 306
7.9.5 合并排序数组 307
7.10 贪心 308
7.10.1 买卖股票的最佳时机 308
7.10.2 买卖股票的最佳时机 II 309
7.10.3 最大子数组 310
7.10.4 主元素 310
7.11 字符串处理 311
7.11.1 生成括号 311
7.11.2 Excel 表列标题 312
7.11.3 翻转游戏 313
7.11.4 翻转字符串中的单词 313
7.11.5 转换字符串到整数 314
7.11.6 最长公共前缀 316
7.11.7 回文数 317
7.12 动态规划 317
7.12.1 单词拆分 317
7.12.2 爬楼梯 319
7.12.3 打劫房屋 320
7.12.4 编辑距离 320
7.12.5 乘积最大子序列 321

7.13 矩阵 322
7.13.1 螺旋矩阵 322
7.13.2 判断数独是否合法 324
7.13.3 旋转图像 324
7.14 二进制 / 位运算 325
7.14.1 落单的数 325
7.14.2 格雷编码 326
7.15 其他 327
7.15.1 反转整数 327
7.15.2 LRU 缓存策略 327

一、JAVA 知识点汇总

1.1 JVM

1.1.1 JVM 工作流程



1.1.2 运行时数据区（Runtime Data Area）

区域	说明
程序计数器	每条线程都需要有一个程序计数器，计数器记录的是正在执行的指令地址，如果正在执行的是 Natvie 方法，这个计数器值为空（Undefined）
java 虚拟机栈	Java 方法执行的内存模型，每个方法执行的时候，都会创建一个栈帧用于保存局部变量表，操作数栈，动态链接，方法出口信息等。一个方法调用的过程就是一个栈帧从 VM 栈入栈到出栈的过程
本地方法栈	与 VM 栈发挥的作用非常相似，VM 栈执行 Java 方法（字节码）服务，Native 方法栈执行的是 Native 方法服务。
Java 堆	此内存区域唯一的目的就是存放对象实例，几乎所有的对象都在这分配内存
方法区	方法区是各个内存所共享的内存空间，方法区中主要存放被 JVM 加载的类信息、常量、静态变量、即时编译后的代码等数据

1.1.3 方法指令

指令	说明
invokeinterface	用以调用接口方法
invokevirtual	指令用于调用对象的实例方法
invokestatic	用以调用类/静态方法
invokespecial	用于调用一些需要特殊处理的实例方法，包括实例初始化方法、私有方法和父类方法

1.1.4 类加载器

类加载器	说明
BootstrapClassLoader	Bootstrap 类加载器负责加载 rt.jar 中的 JDK 类文件，它是所有类加载器的父加载器。Bootstrap 类加载器没有任何父类加载器，如果你调用 String.class.getClassLoader()，会返回 null，任何基于此的代码会抛出 NUllPointerException 异常。Bootstrap 加载器被称为初始类加载器
ExtClassLoader	而 Extension 将加载类的请求先委托给它的父加载器，也就是 Bootstrap，如果没有成功加载的话，再从 jre/lib/ext 目录下或者 java.ext.dirs 系统属性定义的目录下加载类。Extension 加载器由 sun.misc.Launcher$ExtClassLoader 实现
AppClassLoader	第三种默认的加载器就是 System 类加载器（又叫作 Application 类加载器）了。它负责从 classpath 环境变量中加载某些应用相关

类加载器	说明
	的类，classpath 环境变量通常由 -classpath 或 -cp 命令行选项来定义，或者是 JAR 中的 Manifest 的 classpath 属性。Application 类加载器是 Extension 类加载器的子加载器

工作原理	说明
委托机制	加载任务委托交给父类加载器，如果不行就向下传递委托任务，由其子类加载器加载，保证 java 核心库的安全性
可见性机制	子类加载器可以看到父类加载器加载的类，而反之则不行
单一性机制	父加载器加载过的类不能被子加载器加载第二次

1.1.5 垃圾回收 gc

1.1.5.1 对象存活判断

- **引用计数**

每个对象有一个引用计数属性，新增一个引用时计数加 1，引用释放时计数减 1，计数为 0 时可以回收。此方法简单，无法解决对象相互循环引用的问题。

- **可达性分析**

从 GC Roots 开始向下搜索，搜索所走过的路径称为引用链。当一个对象到 GC Roots 没有任何引用链相连时，则证明此对象是不可用的。不可达对象。

> 在 Java 语言中，GC Roots 包括：
>
> - 虚拟机栈中引用的对象。
> - 方法区中类静态属性实体引用的对象。

- 方法区中常量引用的对象。
- 本地方法栈中 JNI 引用的对象。

1.1.5.2 垃圾收集算法

- **标记 -清除算法**

“标记-清除”（Mark-Sweep）算法，如它的名字一样，算法分为“标记”和“清除”两个阶段：首先标记出所有需要回收的对象，在标记完成后统一回收掉所有被标记的对象。之所以说它是最基础的收集算法，是因为后续的收集算法都是基于这种思路并对其缺点进行改进而得到的。

它的主要缺点有两个：一个是效率问题，标记和清除过程的效率都不高；另外一个是空间问题，标记清除之后会产生大量不连续的内存碎片，空间碎片太多可能会导致，当程序在以后的运行过程中需要分配较大对象时无法找到足够的连续内存而不得不提前触发另一次垃圾收集动作。

- **复制算法**

“复制”（Copying）的收集算法，它将可用内存按容量划分为大小相等的两块，每次只使用其中的一块。当这一块的内存用完了，就将还存活着的对象复制到另外一块上面，然后再把已使用过的内存空间一次清理掉。

这样使得每次都是对其中的一块进行内存回收，内存分配时也就不用考虑内存碎片等复杂情况，只要移动堆顶指针，按顺序分配内存即可，实现简单，运行高效。只是这种算法的代价是将内存缩小为原来的一半，持续复制长生存期的对象则导致效率降低。

- **标记-整理算法**

复制收集算法在对象存活率较高时就要执行较多的复制操作，效率将会变低。更关键的是，如果不想浪费 50%的空间，就需要有额外的空间进行分配担保，以应对被使用的内存中所有对象都 100%存活的极端情况，所以在老年代一般不能直接选用这种算法。

根据老年代的特点，有人提出了另外一种“标记-整理”（Mark-Compact）算法，标记过程仍然与“标记-清除”算法一样，但后续步骤不是直接对可回收对象进行清理，而是让所有存活的对象都向一端移动，然后直接清理掉端边界以外的内存。

- **分代收集算法**

GC 分代的基本假设：绝大部分对象的生命周期都非常短暂，存活时间短。

“分代收集”（Generational Collection）算法，把 Java 堆分为新生代和老年代，这样就可以根据各个年代的特点采用最适当的收集算法。在新生代中，每次垃圾收集时都发现有大批对象死去，只有少量存活，那就选用复制算法，只需要付出少量存活对象的复制成本就可以完成收集。而老年代中因为对象存活率高、没有额外空间对它进行分配担保，就必须使用“标记-清理”或“标记-整理”算法来进行回收。

1.1.5.3 垃圾收集器

- **CMS 收集器**

> CMS（Concurrent Mark Sweep）收集器是一种以获取最短回收停顿时间为目标的收集器。目前很大一部分的 Java 应用都集中在互联网站或 B/S 系统的服务端上，这类应用尤其重视服务的响应速度，希望系统停顿时间最短，以给用户带来较好的体验。

从名字（包含“Mark Sweep”）上就可以看出 CMS 收集器是基于“标记-清除”算法实现的，它的运作过程相对于前面几种收集器来说要更复杂一些，整个过程分为 4 个步骤，包括：

- 初始标记（CMS initial mark）
- 并发标记（CMS concurrent mark）
- 重新标记（CMS remark）
- 并发清除（CMS concurrent sweep）

其中初始标记、重新标记这两个步骤仍然需要“Stop The World”。初始标记仅仅只是标记一下 GC Roots 能直接关联到的对象，速度很快，并发标记阶段就是进行 GC Roots Tracing 的过程，

而重新标记阶段则是为了修正并发标记期间，因用户程序继续运作而导致标记产生变动的那一部分对象的标记记录，这个阶段的停顿时间一般会比初始标记阶段稍长一些，但远比并发标记的时间短。

由于整个过程中耗时最长的并发标记和并发清除过程中，收集器线程都可以与用户线程一起工作，所以总体上来说，CMS 收集器的内存回收过程是与用户线程一起并发地执行。老年代收集器（新生代使用 ParNew）

- **G1 收集器**

与 CMS 收集器相比 G1 收集器有以下特点：

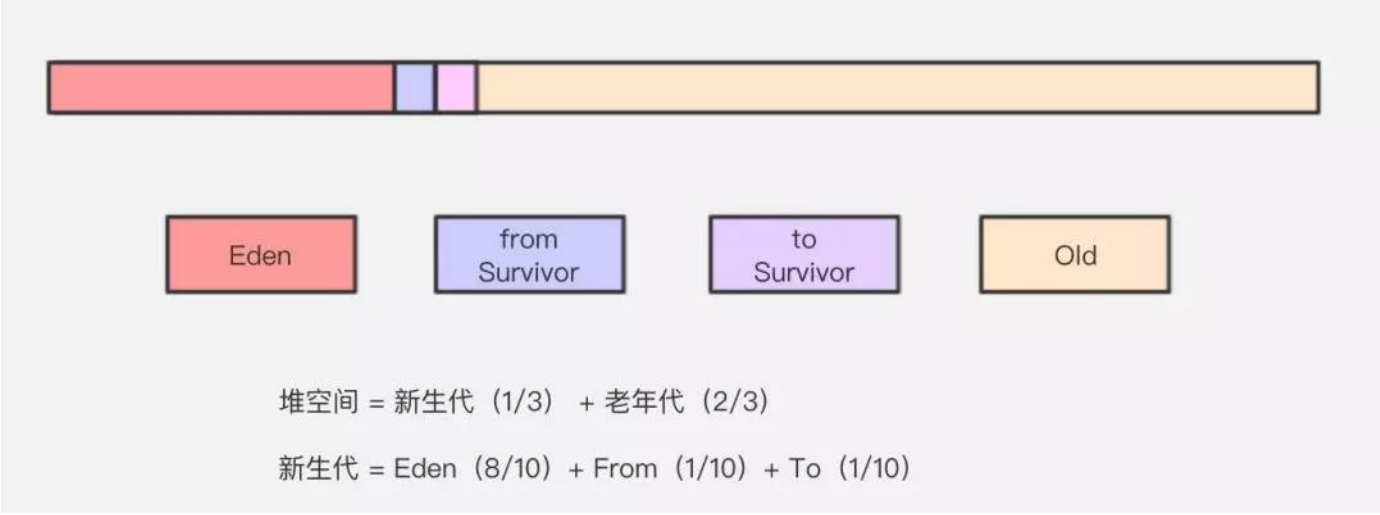
1、空间整合，G1 收集器采用标记整理算法，不会产生内存空间碎片。分配大对象时不会因为无法找到连续空间而提前触发下一次 GC。

2、可预测停顿，这是 G1 的另一大优势，降低停顿时间是 G1 和 CMS 的共同关注点，但 G1 除了追求低停顿外，还能建立可预测的停顿时间模型，能让使用者明确指定在一个长度为 N 毫秒的时间片段内，消耗在垃圾收集上的时间不得超过 N 毫秒，这几乎已经是实时 Java（RTSJ）的垃圾收集器的特征了。

使用 G1 收集器时，Java 堆的内存布局与其他收集器有很大差别，它将整个 Java 堆划分为多个大小相等的独立区域（Region），虽然还保留有新生代和老年代的概念，但新生代和老年代不再是物理隔阂了，它们都是一部分（可以不连续）Region 的集合。

G1 的新生代收集跟 ParNew 类似，当新生代占用达到一定比例的时候，开始出发收集。和 CMS 类似，G1 收集器收集老年代对象会有短暂停顿。

1.1.5.4 内存模型与回收策略



Java 堆（Java Heap）是 JVM 所管理的内存中最大的一块，堆又是垃圾收集器管理的主要区域，Java 堆主要分为 2 个区域-年轻代与老年代，其中年轻代又分 Eden 区和 Survivor 区，其中 Survivor 区又分 From 和 To 2 个区。

- **Eden 区**

大多数情况下，对象会在新生代 Eden 区中进行分配，当 Eden 区没有足够空间进行分配时，虚拟机会发起一次 Minor GC，Minor GC 相比 Major GC 更频繁，回收速度也更快。通过 Minor GC 之后，Eden 会被清空，Eden 区中绝大部分对象会被回收，而那些无需回收的存活对象，将会进到 Survivor 的 From 区（若 From 区不够，则直接进入 Old 区）。

- **Survivor 区**

Survivor 区相当于是 Eden 区和 Old 区的一个缓冲，类似于我们交通灯中的黄灯。Survivor 又分为 2 个区，一个是 From 区，一个是 To 区。每次执行 Minor GC，会将 Eden 区和 From 存活的对象放到 Survivor 的 To 区（如果 To 区不够，则直接进入 Old 区）。Survivor 的

存在意义就是减少被送到老年代的对象，进而减少 Major GC 的发生。Survivor 的预筛选保证，只有经历 16 次 Minor GC 还能在新生代中存活的对象，才会被送到老年代。

- **Old 区**

老年代占据着 2/3 的堆内存空间，只有在 Major GC 的时候才会进行清理，每次 GC 都会触发“Stop-The-World”。内存越大，STW 的时间也越长，所以内存也不仅仅是越大就越好。由于复制算法在对象存活率较高的老年代会进行很多次的复制操作，效率很低，所以老年代这里采用的是标记——整理算法。

1.2 static

- static 关键字修饰的方法或者变量不需要依赖于对象来进行访问，只要类被加载了，就可以通过类名去进行访问。
- 静态变量被所有的对象所共享，在内存中只有一个副本，它当且仅当在类初次加载时会被初始化。
- 能通过 this 访问静态成员变量吗?所有的静态方法和静态变量都可以通过对象访问（只要访问权限足够）。
- static 是不允许用来修饰局部变量

1.3 final

- 可以声明成员变量、方法、类以及本地变量
- final 成员变量必须在声明的时候初始化或者在构造器中初始化，否则就会报编译错误
- final 变量是只读的
- final 申明的方法不可以被子类的方法重写
- final 类通常功能是完整的，不能被继承

- final 变量可以安全的在多线程环境下进行共享，而不需要额外的同步开销
- final 关键字提高了性能，JVM 和 Java 应用都会缓存 final 变量，会对方法、变量及类进行优化
- 方法的内部类访问方法中的局部变量，但必须用 final 修饰才能访问

1.4 String、StringBuffer、StringBuilder

- String 是 final 类，不能被继承。对于已经存在的 Stirng 对象，修改它的值，就是重新创建一个对象
- StringBuffer 是一个类似于 String 的字符串缓冲区，使用 append() 方法修改 Stringbuffer 的值，使用 toString() 方法转换为字符串，是线程安全的
- StringBuilder 用来替代于 StringBuffer，StringBuilder 是非线程安全的，速度更快

1.5 异常处理

- Exception、Error 是 Throwable 类的子类
- Error 类对象由 Java 虚拟机生成并抛出，不可捕捉
- 不管有没有异常，finally 中的代码都会执行
- 当 try、catch 中有 return 时，finally 中的代码依然会继续执行

常见的 Error		
OutOfMemoryError	StackOverflowError	NoClassDeffoundError

常见的 Exception		

常见的 Exception		
常见的非检查性异常		
ArithmeticException	ArrayIndexOutOfBoundsException	ClassCastException
IllegalArgumentException	IndexOutOfBoundsException	NullPointerException
NumberFormatException	SecurityException	UnsupportedOperationException
常见的检查性异常		
IOException	CloneNotSupportedException	IllegalAccessException
NoSuchFieldException	NoSuchMethodException	FileNotFoundException

1.6 内部类

- 非静态内部类没法在外部类的静态方法中实例化。
- 非静态内部类的方法可以直接访问外部类的所有数据，包括私有的数据。
- 在静态内部类中调用外部类成员，成员也要求用 static 修饰。
- 创建静态内部类的对象可以直接通过外部类调用静态内部类的构造器；创建非静态的内部类的对象必须先创建外部类的对象，通过外部类的对象调用内部类的构造器。

1.6.1 匿名内部类

- 匿名内部类不能定义任何静态成员、方法

- 匿名内部类中的方法不能是抽象的
- 匿名内部类必须实现接口或抽象父类的所有抽象方法
- 匿名内部类不能定义构造器
- 匿名内部类访问的外部类成员变量或成员方法必须用 final 修饰

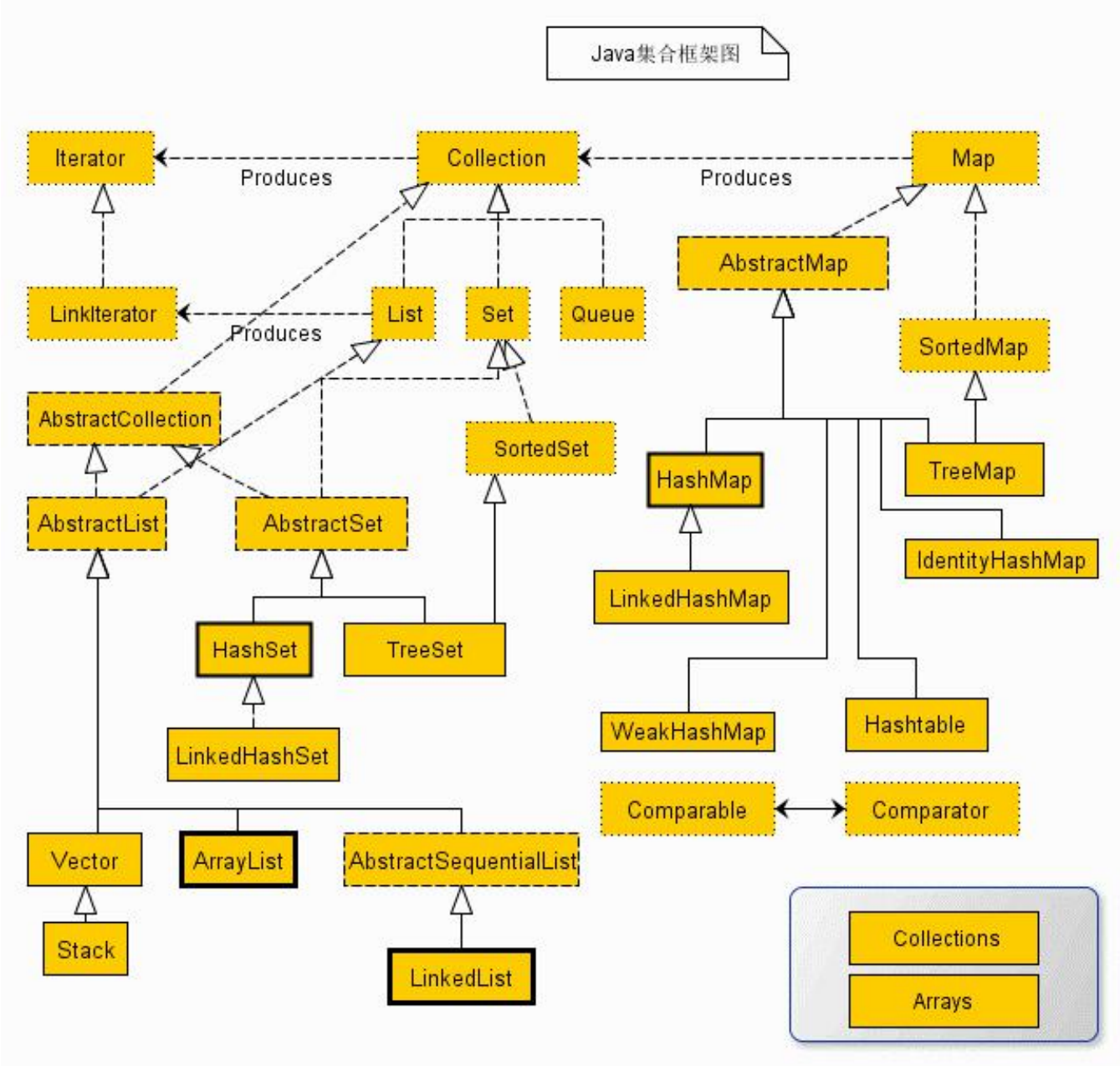
1.7 多态

- 父类的引用可以指向子类的对象
- 创建子类对象时，调用的方法为子类重写的方法或者继承的方法
- 如果我们在子类中编写一个独有的方法，此时就不能通过父类的引用创建的子类对象来调用该方法

1.8 抽象和接口

- 抽象类不能有对象（不能用 new 关键字来创建抽象类的对象）
- 抽象类中的抽象方法必须在子类中被重写
- 接口中的所有属性默认为：public static final ****；
- 接口中的所有方法默认为：public abstract ****；

1.9 集合框架



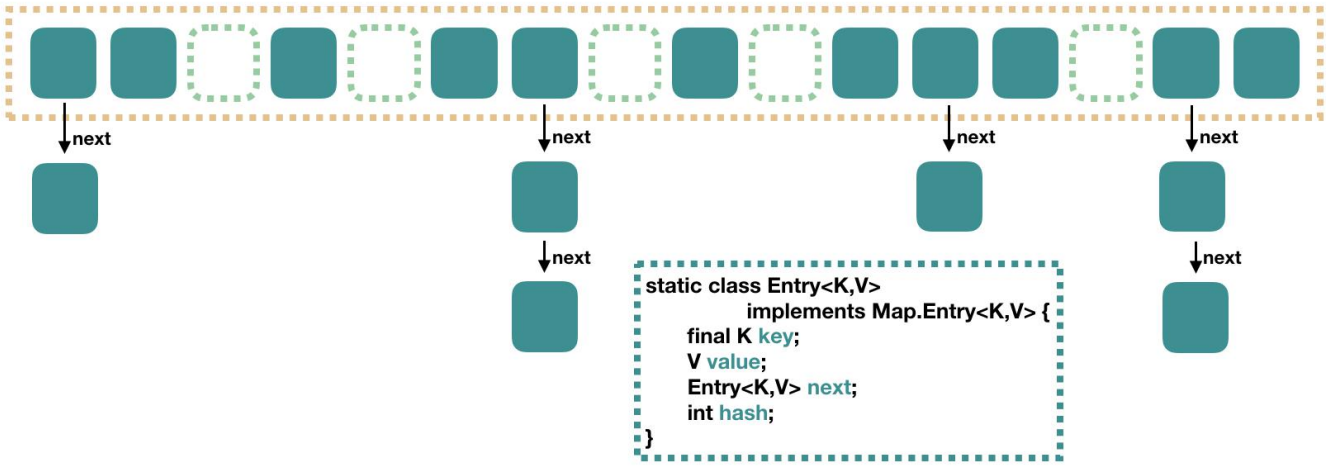
- List 接口存储一组不唯一，有序（插入顺序）的对象, Set 接口存储一组唯一，无序的对象。

1.9.1 HashMap

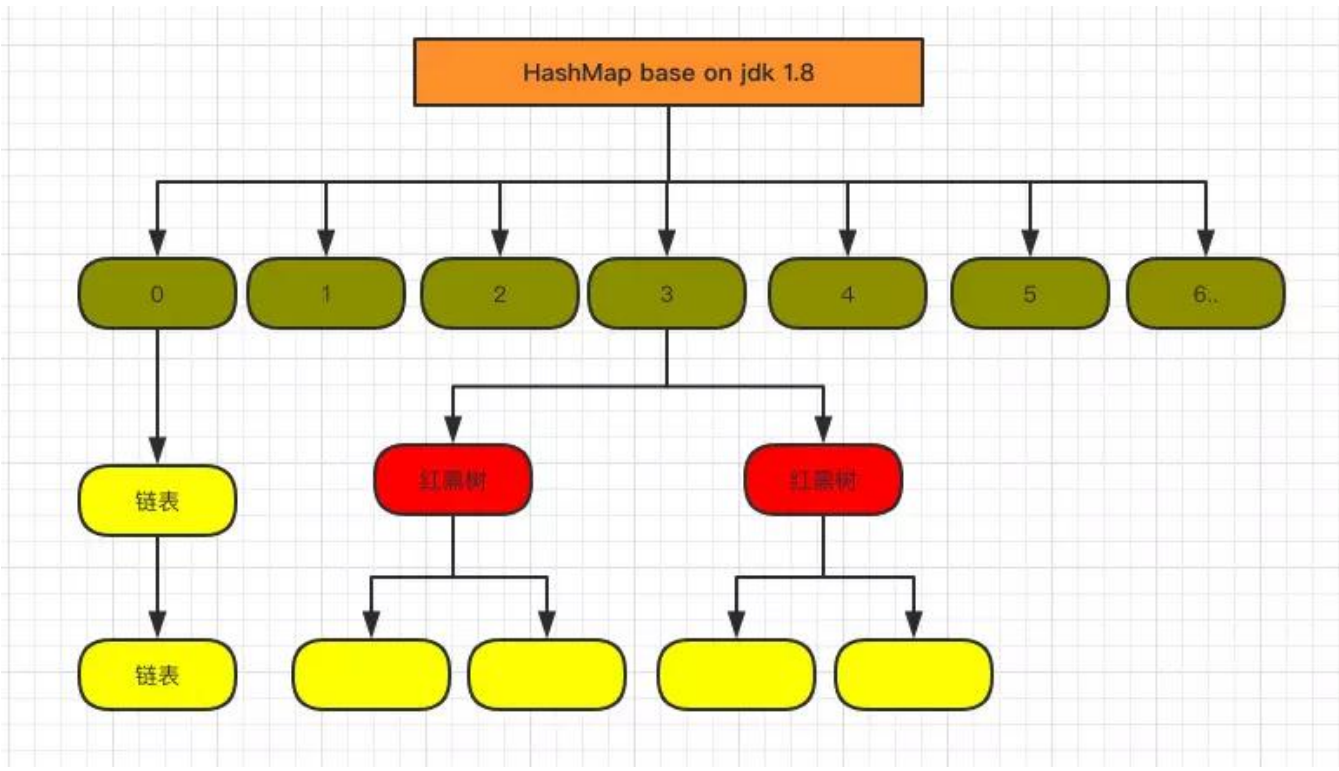
1.9.1.1 结构图

- **JDK 1.7 HashMap 结构图**

Java7 HashMap 结构



- **JDK 1.8 HashMap** 结构图



1.9.1.2 HashMap 的工作原理

HashMap 基于 hashing 原理，我们通过 put() 和 get() 方法储存和获取对象。当我们将键值对传递给 put() 方法时，它调用键对象的 hashCode() 方法来计算 hashcode，让后找到 bucket 位置来储存 Entry 对象。当两个对象的 hashcode 相同时，它们的 bucket 位置相同，‘碰撞’会发生。因为 HashMap 使用链表存储对象，这个 Entry 会存储在链表中，当获取对象时，通过键对象的 equals() 方法找到正确的键值对，然后返回值对象。

如果 HashMap 的大小超过了负载因子(load factor)定义的容量，怎么办？

默认的负载因子大小为 0.75，也就是说，当一个 map 填满了 75% 的 bucket 时候，和其它集合类(如 ArrayList 等)一样，将会创建原来 HashMap 大小的两倍的 bucket 数组，来重新调整 map 的大小，并将原来的对象放入新的 bucket 数组中。这个过程叫作 rehashing，因为它调用 hash 方法找到新的 bucket 位置。

为什么 String, Interger 这样的 wrapper 类适合作为键?

因为 String 是不可变的，也是 final 的，而且已经重写了 equals() 和 hashCode() 方法了。其他的 wrapper 类也有这个特点。不可变性是必要的，因为为了要计算 hashCode()，就要防止键值改变，如果键值在放入时和获取时返回不同的 hashcode 的话，那么就不能从 HashMap 中找到你想要的对象。不可变性还有其他的优点如线程安全。如果你可以仅仅通过将某个 field 声明成 final 就能保证 hashCode 是不变的，那么请这么做吧。因为获取对象的时候要用到 equals() 和 hashCode() 方法，那么键对象正确的重写这两个方法是非常重要的。如果两个不相等的对象返回不同的 hashcode 的话，那么碰撞的几率就会小些，这样就能提高 HashMap 的性能。

1.9.1.3 HashMap 与 HashTable 对比

HashMap 是非 synchronized 的，性能更好，HashMap 可以接受为 null 的 key-value，而 Hashtable 是线程安全的，比 HashMap 要慢，不接受 null 的 key-value。

HashMap.java

```
public class HashMap<K,V> extends AbstractMap<K,V>
    implements Map<K,V>, Cloneable, Serializable {
    ...

    public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }
    ...

    public V get(Object key) {
        Node<K,V> e;
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }
    ...
}
```

HashTable.java

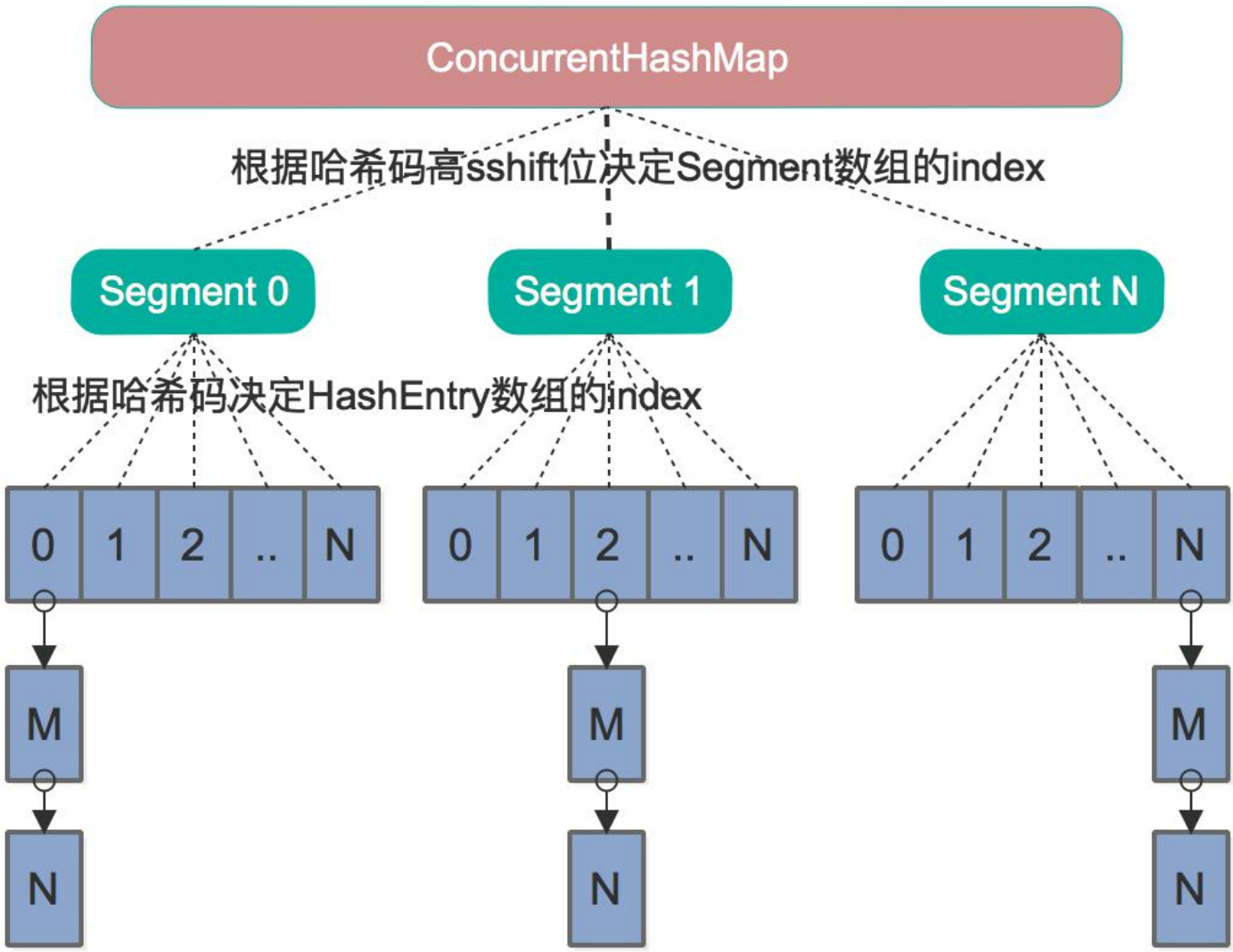
```
public class Hashtable<K,V>
    extends Dictionary<K,V>
    implements Map<K,V>, Cloneable, java.io.Serializable {
    ...
```

```
    public synchronized V put(K key, V value) {
        // Make sure the value is not null
        if (value == null) {
            throw new NullPointerException();
        }

        ...
        addEntry(hash, key, value, index);
        return null;
    }
    ...

    public synchronized V get(Object key) {
        HashtableEntry<?,?> tab[] = table;
        int hash = key.hashCode();
        int index = (hash & 0x7FFFFFFF) % tab.length;
        for (HashtableEntry<?,?> e = tab[index] ; e != null ; e = e.next) {
            if ((e.hash == hash) && e.key.equals(key)) {
                return (V)e.value;
            }
        }
        return null;
    }
    ...
}
```

1.9.2 ConcurrentHashMap

1.9.2.1 Base 1.7

ConcurrentHashMap 最外层不是一个大的数组，而是一个 Segment 的数组。每个 Segment 包含一个与 HashMap 数据结构差不多的链表数组。



在读写某个 Key 时，先取该 Key 的哈希值。并将哈希值的高 N 位对 Segment 个数取模从而得到该 Key 应该属于哪个 Segment，接着如同操作 HashMap 一样操作这个 Segment。

Segment 继承自 ReentrantLock，可以很方便的对每一个 Segmen 上锁。

对于读操作，获取 Key 所在的 Segment 时，需要保证可见性。具体实现上可以使用 volatile 关键字，也可使用锁。但使用锁开销太大，而使用 volatile 时每次写操作都会让所有 CPU 内缓存无效，也有一定开销。ConcurrentHashMap 使用如下方法保证可见性，取得最新的 Segment：

```
Segment<K,V> s = (Segment<K,V>)UNSAFE.getObjectVolatile(segments, u)
```

获取 Segment 中的 HashEntry 时也使用了类似方法：

```
HashEntry<K,V> e = (HashEntry<K,V>) UNSAFE.getObjectVolatile
  (tab, ((long)(((tab.length - 1) & h)) << TSHIFT) + TBASE)
```

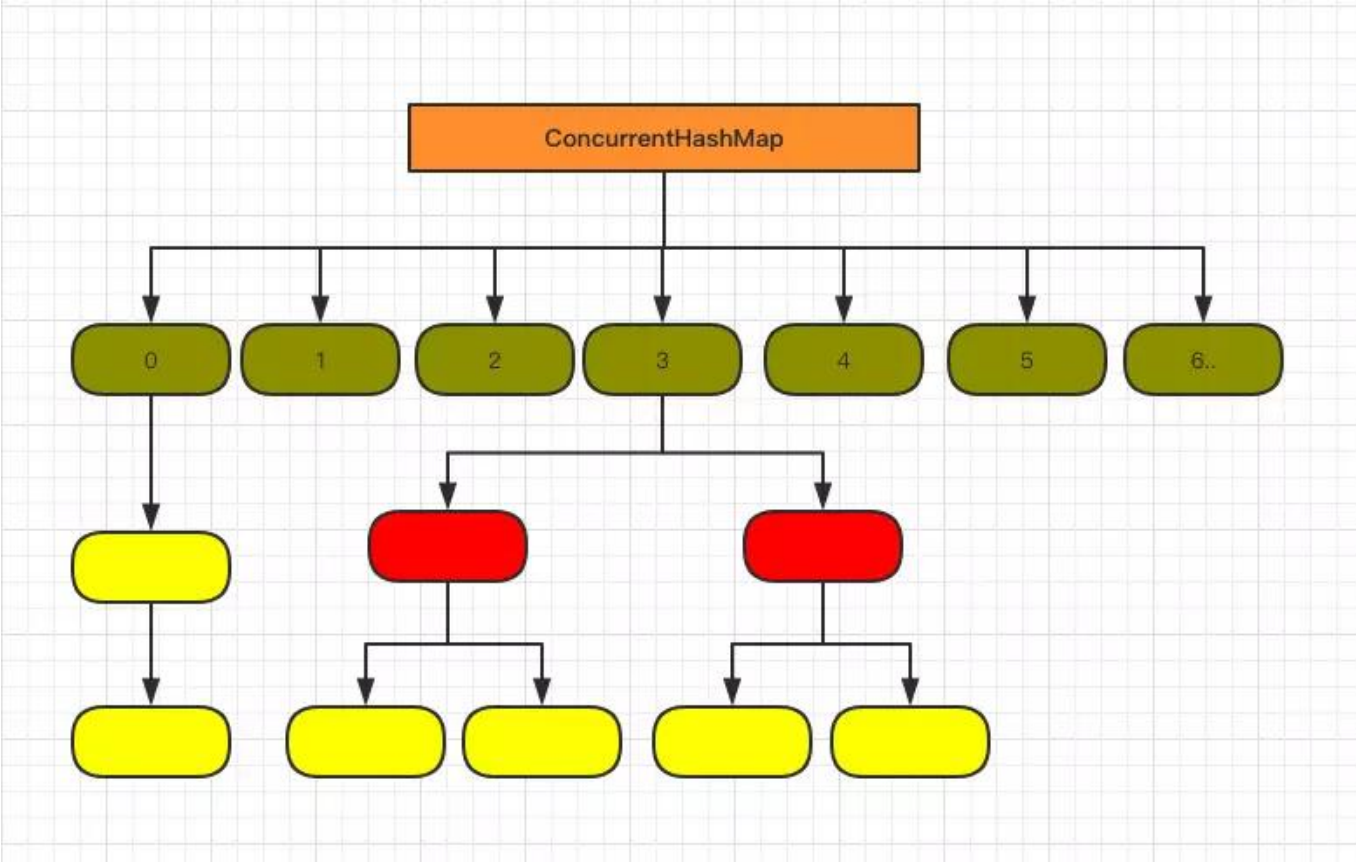
对于写操作，并不要求同时获取所有 Segment 的锁，因为那样相当于锁住了整个 Map。它会先获取该 Key-Value 对所在的 Segment 的锁，获取成功后就可以像操作一个普通的 HashMap 一样操作该 Segment，并保证该 Segment 的安全性。同时由于其它 Segment 的锁并未被获取，因此理论上可支持 concurrencyLevel（等于 Segment 的个数）个线程安全的并发读写。

获取锁时，并不直接使用 lock 来获取，因为该方法获取锁失败时会挂起。事实上，它使用了自旋锁，如果 tryLock 获取锁失败，说明锁被其它线程占用，此时通过循环再次以 tryLock 的方式申请锁。如果在循环过程中该 Key 所对应的链表头被修改，则重置 retry 次数。如果 retry 次数超过一定值，则使用 lock 方法申请锁。

这里使用自旋锁是因为自旋锁的效率比较高，但是它消耗 CPU 资源比较多，因此在自旋次数超过阈值时切换为互斥锁。

1.9.2.2 Base 1.8

1.7 已经解决了并发问题，并且能支持 N 个 Segment 这么多次数的并发，但依然存在 HashMap 在 1.7 版本中的问题：查询遍历链表效率太低。因此 1.8 做了一些数据结构上的调整。



其中抛弃了原有的 Segment 分段锁，而采用了 CAS + synchronized 来保证并发安全性。

ConcurrentHashMap.java

```
final V putVal(K key, V value, boolean onlyIfAbsent) {
    if (key == null || value == null) throw new NullPointerException();
    int hash = spread(key.hashCode());
    int binCount = 0;
    for (Node<K,V>[] tab = table;;) {
        Node<K,V> f; int n, i, fh;
        if (tab == null || (n = tab.length) == 0)
```

```
            tab = initTable();
        else if ((f = tabAt(tab, i = (n - 1) & hash)) == null) {
            if (casTabAt(tab, i, null,
                         new Node<K,V>(hash, key, value, null)))
                break;                   // no lock when adding to empty bin
        }
        else if ((fh = f.hash) == MOVED)
            tab = helpTransfer(tab, f);
        else {
            V oldVal = null;
            synchronized (f) {
                if (tabAt(tab, i) == f) {
                    if (fh >= 0) {
                        binCount = 1;
                        ...
                    }
                    else if (f instanceof TreeBin) {
                       ...
                    }
                    else if (f instanceof ReservationNode)
                        throw new IllegalStateException("Recursive update");
                }
            }
            ...
    }
```

```
    addCount(1L, binCount);

    return null;

}
```

1.9.3 ArrayList

ArrayList 本质上是一个动态数组，第一次添加元素时，数组大小将变化为

DEFAULT_CAPACITY 10，不断添加元素后，会进行扩容。删除元素时，会按照位置关系把数组元素整体（复制）移动一遍。

ArrayList.java

```
public class ArrayList<E> extends AbstractList<E>

        implements List<E>, RandomAccess, Cloneable, java.io.Serializable

    ...



    // 增加元素

    public boolean add(E e) {

        ensureCapacityInternal(size + 1);  // Increments modCount!!

        elementData[size++] = e;

        return true;

    }

    ...



    // 删除元素

    public E remove(int index) {

        if (index >= size)

            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));
```

```
        modCount++;
        E oldValue = (E) elementData[index];

        int numMoved = size - index - 1;
        if (numMoved > 0)
            System.arraycopy(elementData, index+1, elementData, index,
                             numMoved);
        elementData[--size] = null; // clear to let GC do its work

        return oldValue;
    }
    ...

    // 查找元素
    public E get(int index) {
        if (index >= size)
            throw new IndexOutOfBoundsException(outOfBoundsMsg(index));

        return (E) elementData[index];
    }
    ...
}
```

1.9.4 LinkedList

LinkedList 本质上是一个双向链表的存储结构。

LinkedList.java

```
public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable
{
    ....

    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;

        Node(Node<E> prev, E element, Node<E> next) {
            this.item = element;
            this.next = next;
            this.prev = prev;
        }
    }
    ...

    // 增加元素
    void linkLast(E e) {
        final Node<E> l = last;
```

```
        final Node<E> newNode = new Node<>(l, e, null);
        last = newNode;
        if (l == null)
            first = newNode;
        else
            l.next = newNode;
        size++;
        modCount++;
    }
    ...


    // 删除元素
    E unlink(Node<E> x) {
        final E element = x.item;
        final Node<E> next = x.next;
        final Node<E> prev = x.prev;

        if (prev == null) {
            first = next;
        } else {
            prev.next = next;
            x.prev = null;
        }

        if (next == null) {
```

```
            last = prev;
        } else {
            next.prev = prev;
            x.next = null;
        }

        x.item = null;
        size--;
        modCount++;
        return element;
    }
    ...

    // 查找元素
    Node<E> node(int index) {
        // assert isElementIndex(index);

        if (index < (size >> 1)) {
            Node<E> x = first;
            for (int i = 0; i < index; i++)
                x = x.next;
            return x;
        } else {
            Node<E> x = last;
            for (int i = size - 1; i > index; i--)
```

```
                x = x.prev;
            return x;
        }
    }
    ...
}
```

对于元素查询来说，ArrayList 优于 LinkedList，因为 LinkedList 要移动指针。对于新增和删除操作，LinedList 比较占优势，因为 ArrayList 要移动数据。

1.9.5 CopyOnWriteArrayList

CopyOnWriteArrayList 是线程安全容器(相对于 ArrayList)，增加删除等写操作通过加锁的形式保证数据一致性，通过复制新集合的方式解决遍历迭代的问题。

CopyOnWriteArrayList.java

```
public class CopyOnWriteArrayList<E>
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable {


    final transient Object lock = new Object();
    ...


    // 增加元素
    public boolean add(E e) {
        synchronized (lock) {
            Object[] elements = getArray();
            int len = elements.length;
```

```
            Object[] newElements = Arrays.copyOf(elements, len + 1);
            newElements[len] = e;
            setArray(newElements);
            return true;
        }
    }
    ...


    // 删除元素
    public E remove(int index) {
        synchronized (lock) {
            Object[] elements = getArray();
            int len = elements.length;
            E oldValue = get(elements, index);
            int numMoved = len - index - 1;
            if (numMoved == 0)
                setArray(Arrays.copyOf(elements, len - 1));
            else {
                Object[] newElements = new Object[len - 1];
                System.arraycopy(elements, 0, newElements, 0, index);
                System.arraycopy(elements, index + 1, newElements, index,
                                 numMoved);
                setArray(newElements);
            }
            return oldValue;
```

```
        }
    }
    ...


    // 查找元素
    private E get(Object[] a, int index) {
        return (E) a[index];
    }
}
```

1.10 反射

```
try {
    Class cls = Class.forName("com.jasonwu.Test");
    //获取构造方法
    Constructor[] publicConstructors = cls.getConstructors();
    //获取全部构造方法
    Constructor[] declaredConstructors = cls.getDeclaredConstructors();
    //获取公开方法
    Method[] methods = cls.getMethods();
    //获取全部方法
    Method[] declaredMethods = cls.getDeclaredMethods();
    //获取公开属性
    Field[] publicFields = cls.getFields();
    //获取全部属性
    Field[] declaredFields = cls.getDeclaredFields();
```

```
    Object clsObject = cls.newInstance();
    Method method = cls.getDeclaredMethod("getModule1Functionality");
    Object object = method.invoke(null);
} catch (ClassNotFoundException e) {
    e.printStackTrace();
} catch (IllegalAccessException e) {
    e.printStackTrace();
} catch (InstantiationException e) {
    e.printStackTrace();
} catch (NoSuchMethodException e) {
    e.printStackTrace();
} catch (InvocationTargetException e) {
    e.printStackTrace();
}
```

1.11 单例

1.11.1 饿汉式

```
public class CustomManager {
    private Context mContext;
    private static final Object mLock = new Object();
    private static CustomManager mInstance;

    public static CustomManager getInstance(Context context) {
        synchronized (mLock) {
```

```
            if (mInstance == null) {
                mInstance = new CustomManager(context);
            }

            return mInstance;
        }
    }

    private CustomManager(Context context) {
        this.mContext = context.getApplicationContext();
    }
}
```

1.11.2 双重检查模式

```
public class CustomManager {
    private Context mContext;
    private volatile static CustomManager mInstance;

    public static CustomManager getInstance(Context context) {
        // 避免非必要加锁
        if (mInstance == null) {
            synchronized (CustomManger.class) {
                if (mInstance == null) {
                    mInstacne = new CustomManager(context);
                }
```

```
            }
        }

        return mInstacne;
    }

    private CustomManager(Context context) {
        this.mContext = context.getApplicationContext();
    }
}
```

1.11.3 静态内部类模式

```
public class CustomManager{
    private CustomManager(){}

    private static class CustomManagerHolder {
        private static final CustomManager INSTANCE = new CustomManager();
    }

    public static CustomManager getInstance() {
        return CustomManagerHolder.INSTANCE;
    }
}
```

静态内部类的原理是：

当 SingleTon 第一次被加载时，并不需要去加载 SingleTonHoler，只有当 getInstance() 方法第一次被调用时，才会去初始化 INSTANCE，这种方法不仅能确保线程安全，也能保证单例的唯一性，同时也延迟了单例的实例化。getInstance 方法并没有多次去 new 对象，取的都是同一个 INSTANCE 对象。

虚拟机会保证一个类的`<clinit>()`方法在多线程环境中被正确地加锁、同步，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>()`方法，其他线程都需要阻塞等待，直到活动线程执行`<clinit>()`方法完毕

缺点在于无法传递参数，如 Context 等

1.12 线程

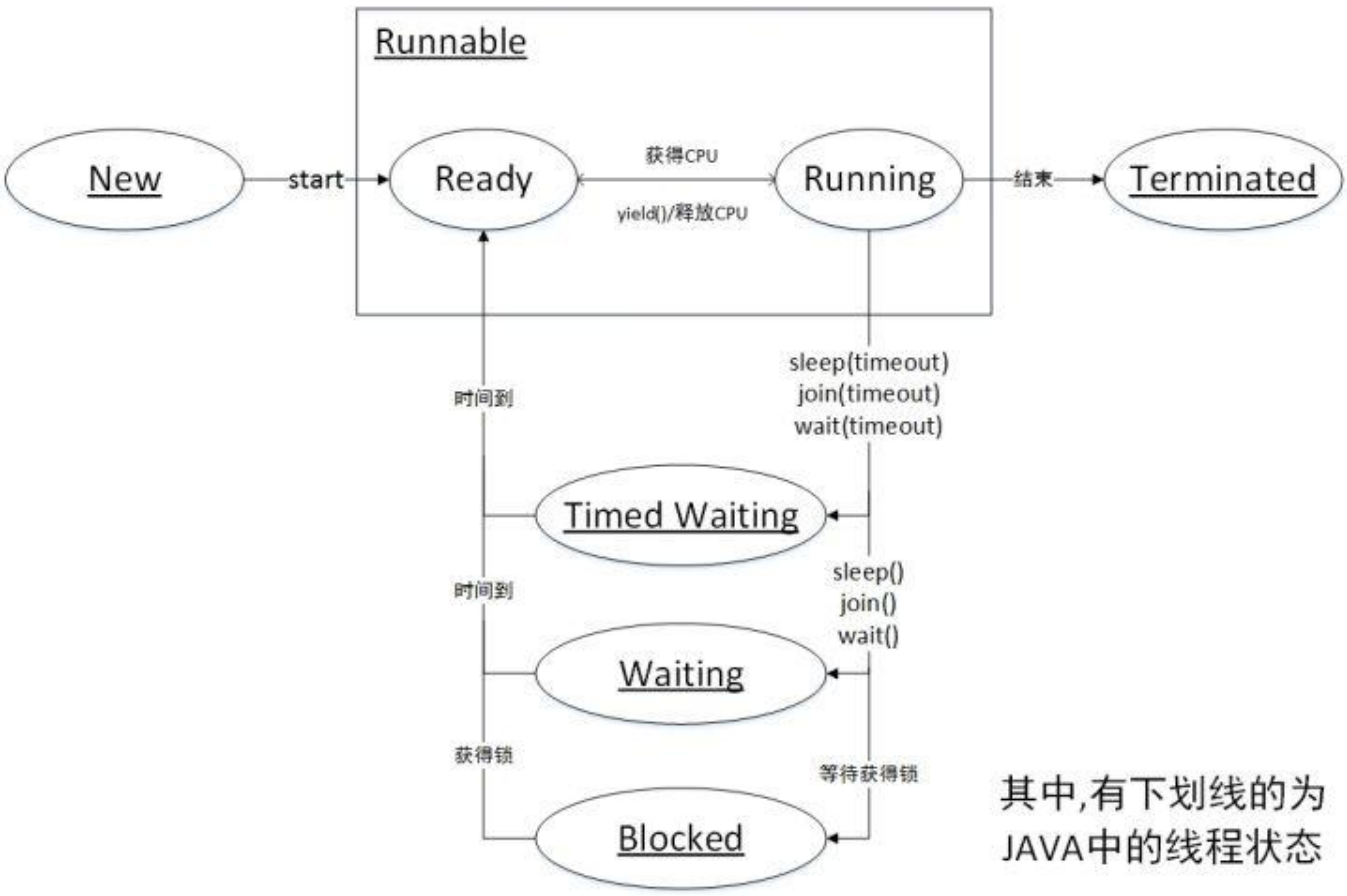
线程是进程中可独立执行的最小单位，也是 CPU 资源（时间片）分配的基本单位。同一个进程中的线程可以共享进程中的资源，如内存空间和文件句柄。

1.12.1 属性

属性	说明
id	线程 id 用于标识不同的线程。编号可能被后续创建的线程使用。编号是只读属性，不能修改
name	名字的默认值是 Thread-(id)
daemon	分为守护线程和用户线程，我们可以通过 setDaemon(true) 把线程设置为守护线程。守护线程通常用于执行不重要的任务，比如监控其他线程的运行情况，GC 线程就是一个守护线程。setDaemon() 要在线程启动前设置，否则 JVM 会抛出非法线程状态异常，可被继承。
priority	线程调度器会根据这个值来决定优先运行哪个线程（不保证），优先级的取值范

属性	说明
	围为 1~10，默认值是 5，可被继承。Thread 中定义了下面三个优先级常量： - 最低优先级：MIN_PRIORITY = 1 - 默认优先级：NORM_PRIORITY = 5 - 最高优先级：MAX_PRIORITY = 10

1.12.2 状态



状态	说明
New	新创建了一个线程对象，但还没有调用 start()方法。

状态	说明
Runnable	Ready 状态 线程对象创建后，其他线程(比如 main 线程）调用了该对象的 start() 方法。该状态的线程位于可运行线程池中，等待被线程调度选中 获取 cpu 的使用权。Running 绪状态的线程在获得 CPU 时间片后变为运行中状态（running）。
Blocked	线程因为某种原因放弃了 cpu 使用权（等待锁），暂时停止运行
Waiting	线程进入等待状态因为以下几个方法： - Object#wait() - Thread#join() - LockSupport#park()
Timed Waiting	有等待时间的等待状态。
Terminated	表示该线程已经执行完毕。

1.12.3 状态控制

- wait() / notify() / notifyAll()

`wait()`，`notify()`，`notifyAll()`是定义在 Object 类的实例方法，用于控制线程状态，三个方法都必须在 synchronized 同步关键字所限定的作用域中调用，否则会报错`java.lang.IllegalMonitorStateException`。

方法	说明

方法	说明
`wait()`	线程状态由 的使用权。Running 变为 Waiting, 并将当前线程放入等待队列中
`notify()`	notify() 方法是将等待队列中一个等待线程从等待队列移动到同步队列中
`notifyAll()`	则是将所有等待队列中的线程移动到同步队列中

被移动的线程状态由 Running 变为 Blocked，notifyAll 方法调用后，等待线程依旧不会从 wait() 返回,需要调用 notify() 或者 notifyAll() 的线程释放掉锁后，等待线程才有机会从 wait() 返回。

- join() / sleep() / yield()

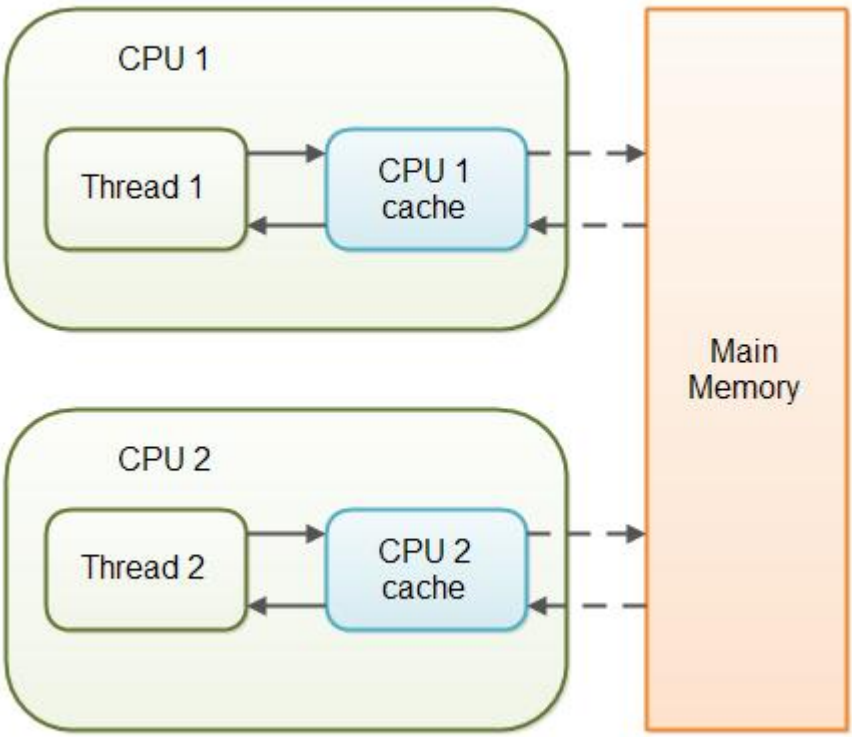
在很多情况，主线程创建并启动子线程，如果子线程中需要进行大量的耗时计算，主线程往往早于子线程结束。这时，如果主线程想等待子线程执行结束之后再结束，比如子线程处理一个数据，主线程要取得这个数据，就要用 `join()`方法。

`sleep(long)`方法在睡眠时不释放对象锁，而 `join()`方法在等待的过程中释放对象锁。

`yield()`方法会临时暂停当前正在执行的线程，来让有同样优先级的正在等待的线程有机会执行。如果没有正在等待的线程，或者所有正在等待的线程的优先级都比较低，那么该线程会继续运行。执行了 yield 方法的线程什么时候会继续运行由线程调度器来决定。

1.13 volatile

当把变量声明为 volatile 类型后，编译器与运行时都会注意到这个变量是共享的，因此不会将该变量上的操作与其他内存操作一起重排序。volatile 变量不会被缓存在寄存器或者对其他处理器不可见的地方，JVM 保证了每次读变量都从内存中读，跳过 CPU cache 这一步，因此在读取 volatile 类型的变量时总会返回最新写入的值。



当一个变量定义为 volatile 之后，将具备以下特性：

- 保证此变量对所有的线程的可见性，不能保证它具有原子性（可见性，是指线程之间的可见性，一个线程修改的状态对另一个线程是可见的）
- 禁止指令重排序优化
- volatile 的读性能消耗与普通变量几乎相同，但是写操作稍慢，因为它需要在本地代码中插入许多内存屏障指令来保证处理器不发生乱序执行

AtomicInteger 中主要实现了整型的原子操作，防止并发情况下出现异常结果，其内部主要依靠 JDK 中的 unsafe 类操作内存中的数据来实现的。volatile 修饰符保证了 value 在内存中其他线程可以看到其值得改变。CAS（Compare and Swap）操作保证了 AtomicInteger 可以安全的修改 value 的值。

1.14 synchronized

当它用来修饰一个方法或者一个代码块的时候，能够保证在同一时刻最多只有一个线程执行该段代码。

在 Java 中，每个对象都会有一个 monitor 对象，这个对象其实就是 Java 对象的锁，通常会被称为“内置锁”或“对象锁”。类的对象可以有多个，所以每个对象有其独立的对象锁，互不干扰。针对每个类也有一个锁，可以称为“类锁”，类锁实际上是通过对象锁实现的，即类的 Class 对象锁。每个类只有一个 Class 对象，所以每个类只有一个类锁。

Monitor 是线程私有的数据结构，每一个线程都有一个可用 monitor record 列表，同时还有一个全局的可用列表。每一个被锁住的对象都会和一个 monitor 关联，同时 monitor 中有一个 Owner 字段存放拥有该锁的线程的唯一标识，表示该锁被这个线程占用。Monitor 是依赖于底层的操作系统的 Mutex Lock（互斥锁）来实现的线程同步。

1.14.1 根据获取的锁分类

获取对象锁

- synchronized(this|object) {}
- 修饰非静态方法

获取类锁

- synchronized(类.class) {}
- 修饰静态方法

1.14.2 原理

同步代码块：

- monitorenter 和 monitorexit 指令实现的

同步方法

- 方法修饰符上的 ACC_SYNCHRONIZED 实现

1.15 Lock

```
public interface Lock {

    void lock();

    void lockInterruptibly() throws InterruptedException;

    boolean tryLock();

    boolean tryLock(long time, TimeUnit unit) throws InterruptedException;

    void unlock();

    Condition newCondition();

}
```

方法	说明
lock()	用来获取锁，如果锁被其他线程获取，处于等待状态。如果采用 Lock，必须主动去释放锁，并且在发生异常时，不会自动释放锁。因此一般来说，使用 Lock 必须在 try{}catch{} 块中进行，并且将释放锁的操作放在 finally 块中进行，以保证锁一定被被释放，防止死锁的发生。
lockInterruptibly()	通过这个方法去获取锁时，如果线程正在等待获取锁，则这个线程能够响应中断，即中断线程的等待状态。
tryLock()	tryLock 方法是有返回值的，它表示用来尝试获取锁，如果获取成功，则返回 true，如果获取失败（即锁已被其他线程获取），则返回 false，也就说这个方法无论如何都会立即返回。在拿不到锁时不会一直在那等待。
tryLock(long, TimeUnit)	与 tryLock 类似，只不过是有等待时间，在等待时间内获取到锁返

方法	说明
	回 true，超时返回 false。

1.15.1 锁的分类

1.15.1.1 悲观锁、乐观锁

悲观锁认为自己在使用数据的时候一定有别的线程来修改数据，因此在获取数据的时候会先加锁，确保数据不会被别的线程修改。Java 中，synchronized 关键字和 Lock 的实现类都是悲观锁。悲观锁适合写操作多的场景，先加锁可以保证写操作时数据正确。

而乐观锁认为自己在使用数据时不会有别的线程修改数据，所以不会添加锁，只是在更新数据的时候去判断之前有没有别的线程更新了这个数据。如果这个数据没有被更新，当前线程将自己修改的数据成功写入。如果数据已经被其他线程更新，则根据不同的实现方式执行不同的操作（例如报错或者自动重试）。乐观锁在 Java 中是通过使用无锁编程来实现，最常采用的是 CAS 算法，Java 原子类中的递增操作就通过 CAS 自旋实现。乐观锁适合读操作多的场景，不加锁的特点能够使其读操作的性能大幅提升。

1.15.1.2 自旋锁、适应性自旋锁

阻塞或唤醒一个 Java 线程需要操作系统切换 CPU 状态来完成，这种状态转换需要耗费处理器时间。如果同步代码块中的内容过于简单，状态转换消耗的时间有可能比用户代码执行的时间还要长。

在许多场景中，同步资源的锁定时间很短，为了这一小段时间去切换线程，线程挂起和恢复现场的花费可能会让系统得不偿失。如果物理机器有多个处理器，能够让两个或以上的线程同时并行执行，我们就可以让后面那个请求锁的线程不放弃 CPU 的执行时间，看看持有锁的线程是否很快就会释放锁。

而为了让当前线程“稍等一下”，我们需让当前线程进行自旋，如果在自旋完成后前面锁定同步资源的线程已经释放了锁，那么当前线程就可以不必阻塞而是直接获取同步资源，从而避免切换线程的开销。这就是自旋锁。

自旋锁本身是有缺点的，它不能代替阻塞。自旋等待虽然避免了线程切换的开销，但它要占用处理器时间。如果锁被占用的时间很短，自旋等待的效果就会非常好。反之，如果锁被占用的时间很长，那么自旋的线程只会白浪费处理器资源。所以，自旋等待的时间必须要有一定的限度，如果自旋超过了限定次数（默认是 10 次，可以使用 -XX:PreBlockSpin 来更改）没有成功获得锁，就应当挂起线程。

自旋锁的实现原理同样也是 CAS，AtomicInteger 中调用 unsafe 进行自增操作的源码中的 do-while 循环就是一个自旋操作，如果修改数值失败则通过循环来执行自旋，直至修改成功。

1.15.1.3 死锁

当前线程拥有其他线程需要的资源，当前线程等待其他线程已拥有的资源，都不放弃自己拥有的资源。

1.16 引用类型

强引用 > 软引用 > 弱引用

引用类型	说明
StrongReferenc（强引用）	当一个对象具有强引用，那么垃圾回收器是绝对不会的回收和销毁它的，**非静态内部类会在其整个生命周期中持有对它外部类的强引用**
WeakReference （弱引用）	在垃圾回收器运行的时候，如果对一个对象的所有引用都是弱引用的话，该对象会被回收
SoftReference（软引用）	如果一个对象只具有软引用，若内存空间足够，垃圾回收器就不会回收它；如果内存空间不足了，才会回收这些对象的内存
PhantomReference（虚引用）	一个只被虚引用持有的对象可能会在任何时候被 GC 回收。虚引用对对象的生存周期完全没有影响，也无法通过虚引用来获取对象实例，仅仅能在对象被回收时，得到一个系统通知（只能通过是否被加入到 ReferenceQueue 来判断是否被 GC，这也是唯一判断对象是否被 GC 的途径）。

1.17 动态代理

示例：

```
// 定义相关接口 public interface BaseInterface {

    void doSomething();

}
```

```
// 接口的相关实现类 public class BaseImpl implements BaseInterface {

    @Override

    public void doSomething() {

        System.out.println("doSomething");

    }

}

public static void main(String args[]) {

    BaseImpl base = new BaseImpl();

    // Proxy 动态代理实现

    BaseInterface proxyInstance = (BaseInterface)
Proxy.newProxyInstance(base.getClass().getClassLoader(), base.getClass().getInterfaces(),
new InvocationHandler() {

        @Override

        public Object invoke(Object proxy, Method method, Object[] args) throws Throwable
{

            if (method.getName().equals("doSomething")) {

                method.invoke(base, args);

                System.out.println("do more");

            }

            return null;

        }

    });



    proxyInstance.doSomething();

}
```

Proxy.java

```
public class Proxy implements java.io.Serializable {


    // 代理类的缓存
    private static final WeakCache<ClassLoader, Class<?>[], Class<?>>
        proxyClassCache = new WeakCache<>(new KeyFactory(), new ProxyClassFactory());
    ...


    // 生成代理对象方法入口
    public static Object newProxyInstance(ClassLoader loader,
                                          Class<?>[] interfaces,
                                          InvocationHandler h)
    throws IllegalArgumentException
    {
        Objects.requireNonNull(h);


        final Class<?>[] intfs = interfaces.clone();
        // 找到并生成相关的代理类
        Class<?> cl = getProxyClass0(loader, intfs);


        // 调用代理类的构造方法生成代理类实例
        try {
            final Constructor<?> cons = cl.getConstructor(constructorParams);
            final InvocationHandler ih = h;
            if (!Modifier.isPublic(cl.getModifiers())) {
                cons.setAccessible(true);
```

```
        }
        return cons.newInstance(new Object[]{h});
    }
    ...
}
...

// 定义和返回代理类的工厂类
private static final class ProxyClassFactory
    implements BiFunction<ClassLoader, Class<?>[], Class<?>>
{
    // 所有代理类的前缀
    private static final String proxyClassNamePrefix = "$Proxy";

    //  用于生成唯一代理类名称的下一个数字
    private static final AtomicLong nextUniqueNumber = new AtomicLong();

    @Override
    public Class<?> apply(ClassLoader loader, Class<?>[] interfaces) {

        Map<Class<?>, Boolean> interfaceSet = new IdentityHashMap<>(interfaces.length);
        ...

        String proxyPkg = null;     // 用于定义代理类的包名
        int accessFlags = Modifier.PUBLIC | Modifier.FINAL;
```

```
// 确保所有 non-public 的代理接口在相同的包里
for (Class<?> intf : interfaces) {
    int flags = intf.getModifiers();
    if (!Modifier.isPublic(flags)) {
        accessFlags = Modifier.FINAL;
        String name = intf.getName();
        int n = name.lastIndexOf('.');
        String pkg = ((n == -1) ? "" : name.substring(0, n + 1));
        if (proxyPkg == null) {
            proxyPkg = pkg;
        } else if (!pkg.equals(proxyPkg)) {
            throw new IllegalArgumentException(
                "non-public interfaces from different packages");
        }
    }
}

if (proxyPkg == null) {
    // 如果没有 non-public 的代理接口，使用默认的包名
    proxyPkg = "";
}

{
    List<Method> methods = getMethods(interfaces);
```

```
                Collections.sort(methods, ORDER_BY_SIGNATURE_AND_SUBTYPE);

                validateReturnTypes(methods);

                List<Class<?>[]> exceptions = deduplicateAndGetExceptions(methods);


                Method[] methodsArray = methods.toArray(new Method[methods.size()]);

                Class<?>[][] exceptionsArray = exceptions.toArray(new
Class<?>[exceptions.size()][]);


                // 生成代理类的名称

                long num = nextUniqueNumber.getAndIncrement();

                String proxyName = proxyPkg + proxyClassNamePrefix + num;


                // Android 特定修改：直接调用 native 方法生成代理类

                return generateProxy(proxyName, interfaces, loader, methodsArray,

                                     exceptionsArray);


                // JDK 使用的 ProxyGenerator.generateProxyClas 方法创建代理类

                byte[] proxyClassFile = ProxyGenerator.generateProxyClass(

                    proxyName, interfaces, accessFlags);

                try {

                    return defineClass0(loader, proxyName,

                                        proxyClassFile, 0, proxyClassFile.length);

                } ...

        }

    }
```

```
    ...


    // 最终调用 native 方法生成代理类
    @FastNative
    private static native Class<?> generateProxy(String name, Class<?>[] interfaces,
                                                 ClassLoader loader, Method[] methods,
                                                 Class<?>[][] exceptions);


}
```

ProxyGenerator.java

```
public static byte[] generateProxyClass(final String name,
                                        Class[] interfaces)
{
    ProxyGenerator gen = new ProxyGenerator(name, interfaces);
    final byte[] classFile = gen.generateClassFile();


    if (saveGeneratedFiles) {
        java.security.AccessController.doPrivileged(
        new java.security.PrivilegedAction<Void>() {
            public Void run() {
                try {
                    FileOutputStream file =
                        new FileOutputStream(dotToSlash(name) + ".class");
                    file.write(classFile);
                    file.close();
```

```
                return null;
            } catch (IOException e) {
                throw new InternalError(
                    "I/O exception saving generated file: " + e);
            }
        }
    });
}

return classFile;
}
```

1.18 元注解

@Retention：保留的范围，可选值有三种。

RetentionPolicy	说明
SOURCE	注解将被编译器丢弃（该类型的注解信息只会保留在源码里，源码经过编译后，注解信息会被丢弃，不会保留在编译好的 class 文件里），如 @Override
CLASS	注解在 class 文件中可用，但会被 VM 丢弃（该类型的注解信息会保留在源码里和 class 文件里，在执行的时候，不会加载到虚拟机中），请注意，当注解未定义 Retention 值时，默认值是 CLASS。
RUNTIME	注解信息将在运行期 (JVM) 也保留，因此可以通过反射机制读取注解的

RetentionPolicy	说明
	信息（源码、class 文件和执行的时候都有注解的信息），如 @Deprecated

@Target：可以用来修饰哪些程序元素，如 TYPE, METHOD, CONSTRUCTOR, FIELD, PARAMETER 等，未标注则表示可修饰所有

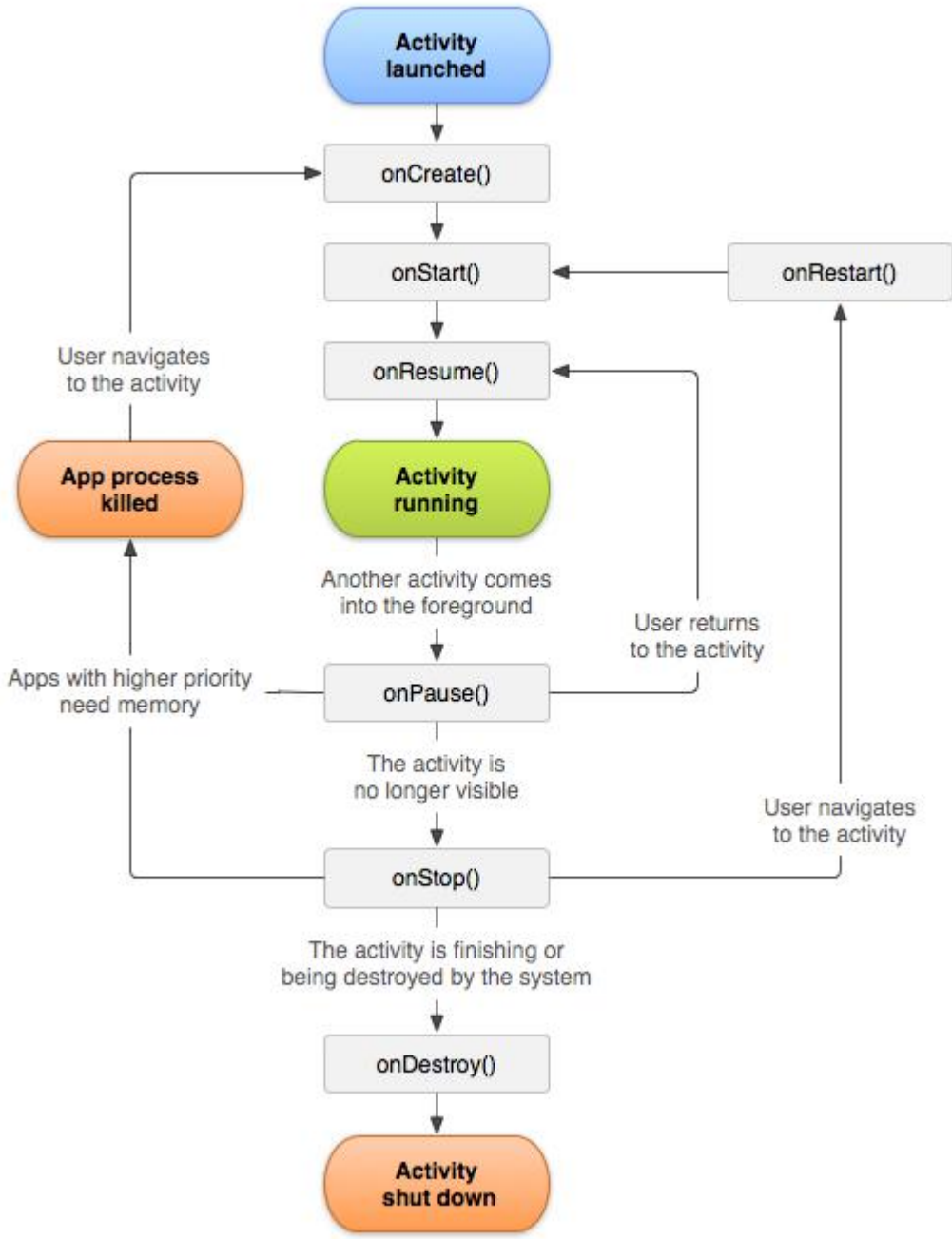
@Inherited：是否可以被继承，默认为 false

@Documented：是否会保存到 Javadoc 文档中

二、Android 知识点汇总

2.1 Activity

2.1.1 生命周期



- Activity A 启动另一个 Activity B，回调如下:

  Activity A 的 onPause() → Activity B 的 onCreate() → onStart() → onResume() → Activity A 的 onStop()；如果 B 是透明主题又或则是个 DialogActivity，则不会回调 A 的 onStop；

- 使用 onSaveInstanceState（）保存简单，轻量级的 UI 状态

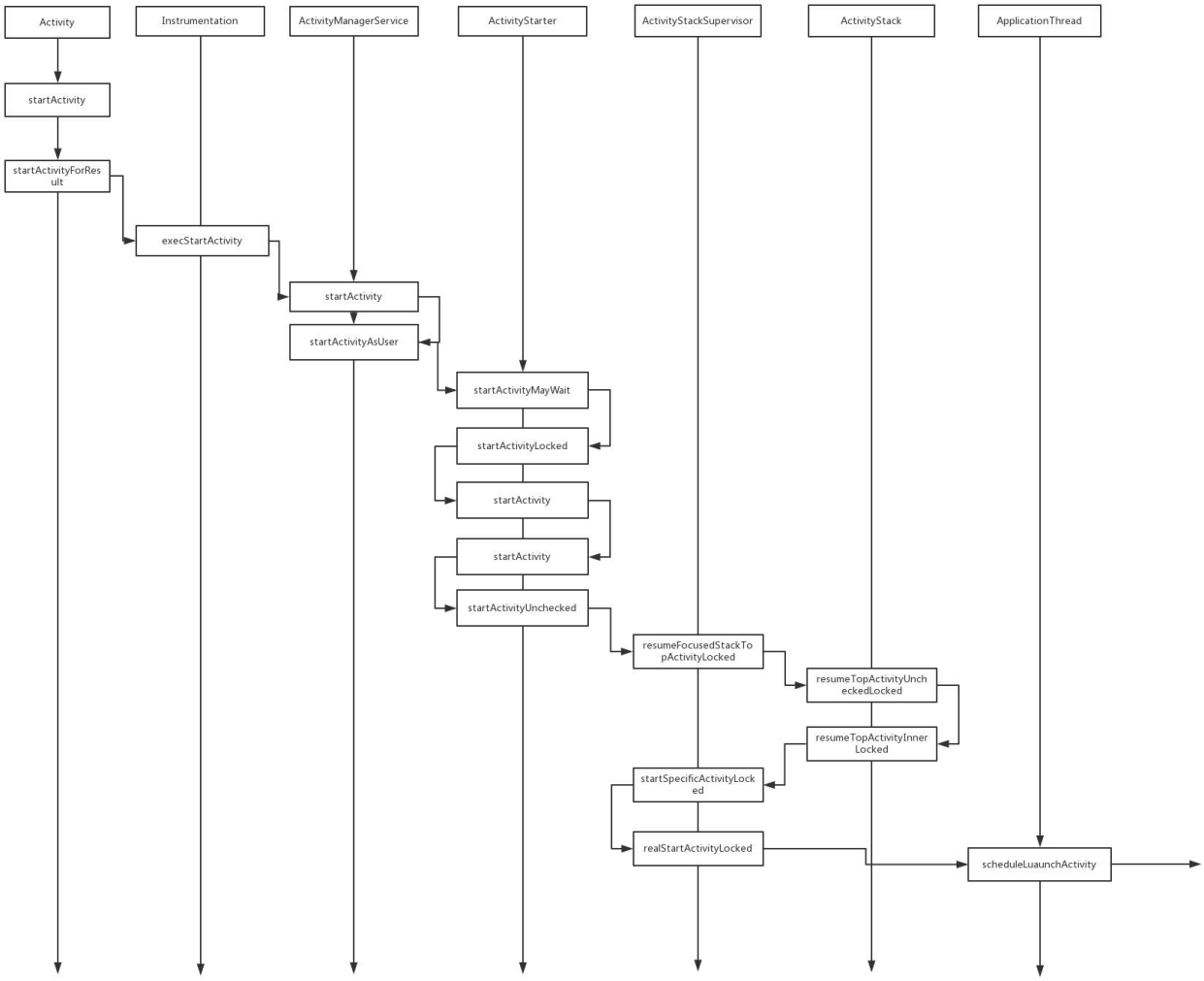
```
lateinit var textView: TextView
var gameState: String? = null

override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    gameState = savedInstanceState?.getString(GAME_STATE_KEY)
    setContentView(R.layout.activity_main)
    textView = findViewById(R.id.text_view)
}

override fun onRestoreInstanceState(savedInstanceState: Bundle?) {
    textView.text = savedInstanceState?.getString(TEXT_VIEW_KEY)
}

override fun onSaveInstanceState(outState: Bundle?) {
    outState?.run {
        putString(GAME_STATE_KEY, gameState)
        putString(TEXT_VIEW_KEY, textView.text.toString())
    }
    super.onSaveInstanceState(outState)
```

```
}
```

2.1.2 启动模式

LaunchMode	说明
standard	系统在启动它的任务中创建 activity 的新实例
singleTop	如果 activity 的实例已存在于当前任务的顶部，则系统通过调用其 onNewIntent()，否则会创建新实例
singleTask	系统创建新 task 并在 task 的根目录下实例化 activity。但如果 activity 的实例已存在于单独的任务中，则调用其 onNewIntent() 方法，其上面的实例会被移除栈。一次只能存在一个 activity 实例
singleInstance	相同 singleTask，activity 始终是其 task 的唯一成员；任何由此开始的 activity 都在一个单独的 task 中打开

2.1.3 启动过程



ActivityThread.java

```
private Activity performLaunchActivity(ActivityClientRecord r, Intent customIntent) {
    ...
    ActivityInfo aInfo = r.activityInfo;
    if (r.packageInfo == null) {
        //step 1: 创建 LoadedApk 对象
        r.packageInfo = getPackageInfo(aInfo.applicationInfo, r.compatInfo,
                Context.CONTEXT_INCLUDE_CODE);
    }
    ... //component 初始化过程
```

```
    java.lang.ClassLoader cl = r.packageInfo.getClassLoader();
    //step 2: 创建 Activity 对象
    Activity activity = mInstrumentation.newActivity(cl, component.getClassName(),
r.intent);
    ...

    //step 3: 创建 Application 对象
    Application app = r.packageInfo.makeApplication(false, mInstrumentation);

    if (activity != null) {
        //step 4: 创建 ContextImpl 对象
        Context appContext = createBaseContextForActivity(r, activity);
        CharSequence title = r.activityInfo.loadLabel(appContext.getPackageManager());
        Configuration config = new Configuration(mCompatConfiguration);
        //step5: 将 Application/ContextImpl 都 attach 到 Activity 对象
        activity.attach(appContext, this, getInstrumentation(), r.token,
                r.ident, app, r.intent, r.activityInfo, title, r.parent,
                r.embeddedID, r.lastNonConfigurationInstances, config,
                r.referrer, r.voiceInteractor);

        ...
        int theme = r.activityInfo.getThemeResource();
        if (theme != 0) {
            activity.setTheme(theme);
```

```
    }


    activity.mCalled = false;
    if (r.isPersistable()) {
        //step 6: 执行回调 onCreate
        mInstrumentation.callActivityOnCreate(activity, r.state, r.persistentState);
    } else {
        mInstrumentation.callActivityOnCreate(activity, r.state);
    }


    r.activity = activity;
    r.stopped = true;
    if (!r.activity.mFinished) {
        activity.performStart(); //执行回调 onStart
        r.stopped = false;
    }
    if (!r.activity.mFinished) {
        //执行回调 onRestoreInstanceState
        if (r.isPersistable()) {
            if (r.state != null || r.persistentState != null) {
                mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state,
                        r.persistentState);
            }
        } else if (r.state != null) {
            mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
```

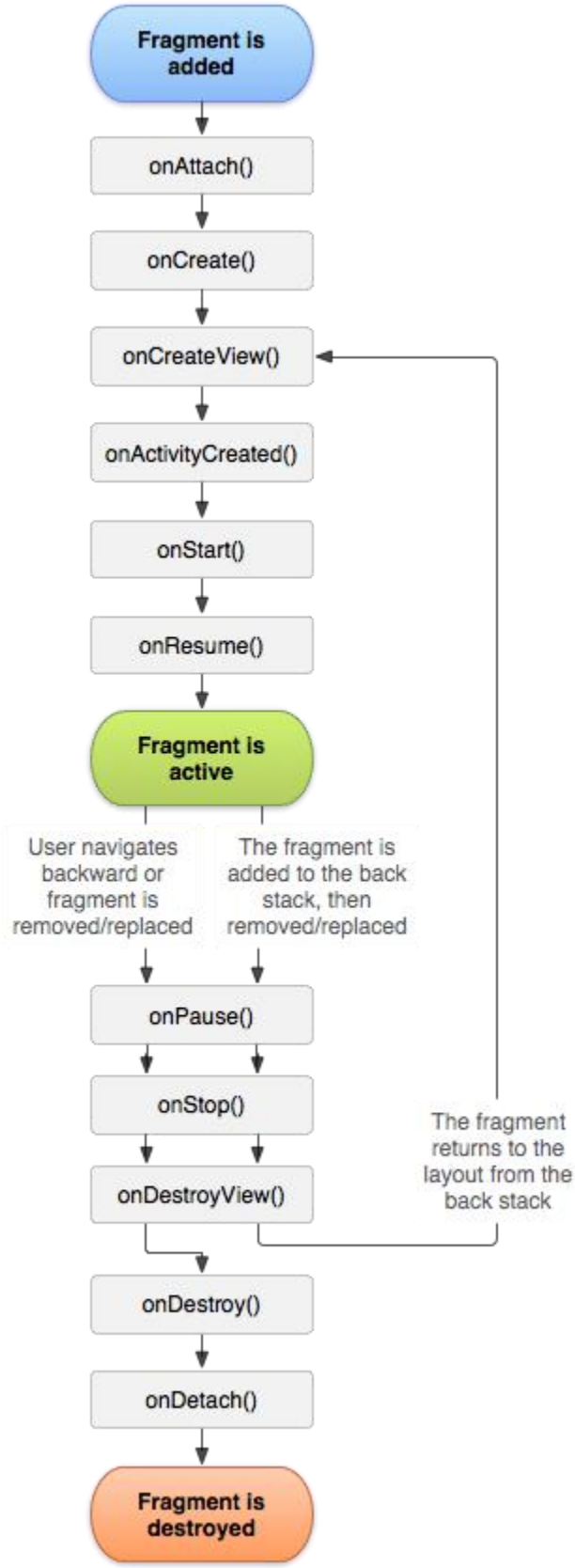
```
            }
        }
        ...
        r.paused = true;
        mActivities.put(r.token, r);
    }

    return activity;
}
```

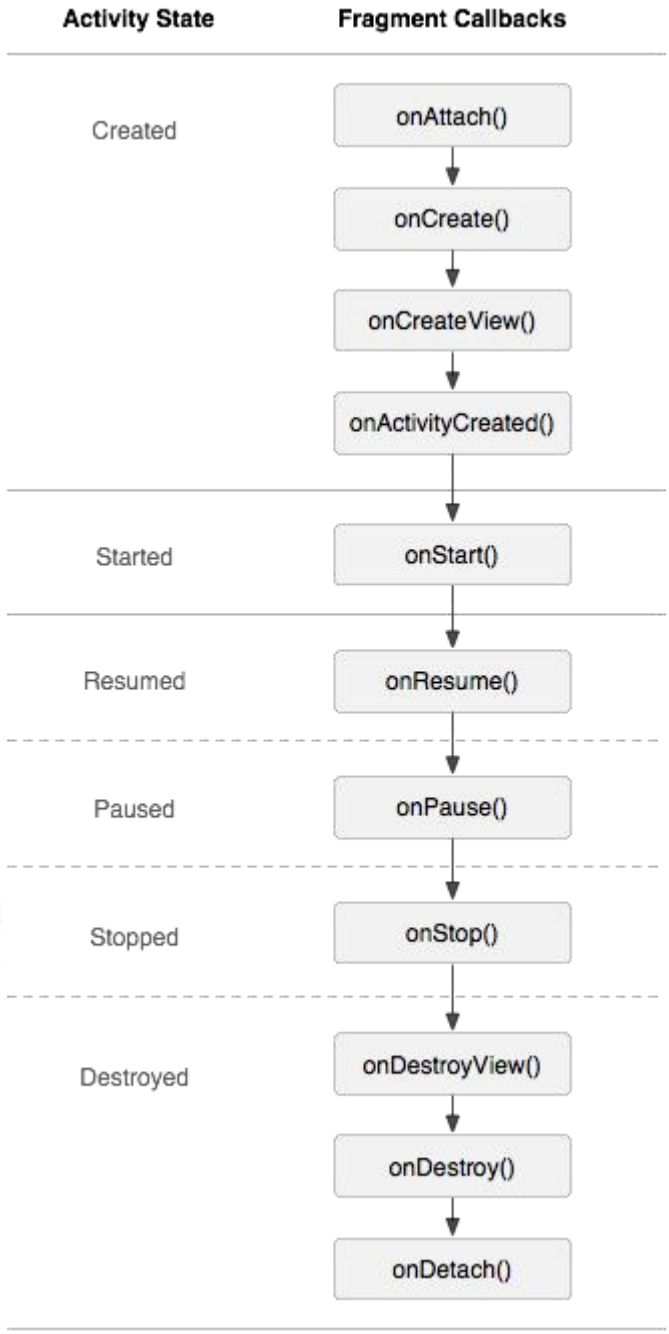
2.2 Fragment

2.2.1 特点

- Fragment 解决 Activity 间的切换不流畅，轻量切换
- 可以从 startActivityForResult 中接收到返回结果，但是 View 不能
- 只能在 Activity 保存其状态（用户离开 Activity）之前使用 commit() 提交事务。如果您试图在该时间点后提交，则会引发异常。 这是因为如需恢复 Activity，则提交后的状态可能会丢失。 对于丢失提交无关紧要的情况，请使用 commitAllowingStateLoss()。

2.2.2 生命周期


Fragment is added
onAttach()
onCreate()
onCreateView()
onActivityCreated()
onStart()
onResume()
Fragment is active
User navigates backward or fragment is removed/replaced
The fragment is added to the back stack, then removed/replaced
onPause()
onStop()
onDestroyView()
The fragment returns to the layout from the back stack
onDestroy()
onDetach()
Fragment is destroyed


Activity State
Fragment Callbacks
Created
onAttach()
onCreate()
onCreateView()
onActivityCreated()
Started
onStart()
Resumed
onResume()
Paused
onPause()
Stopped
onStop()
Destroyed
onDestroyView()
onDestroy()
onDetach()

2.2.3 与 Activity 通信

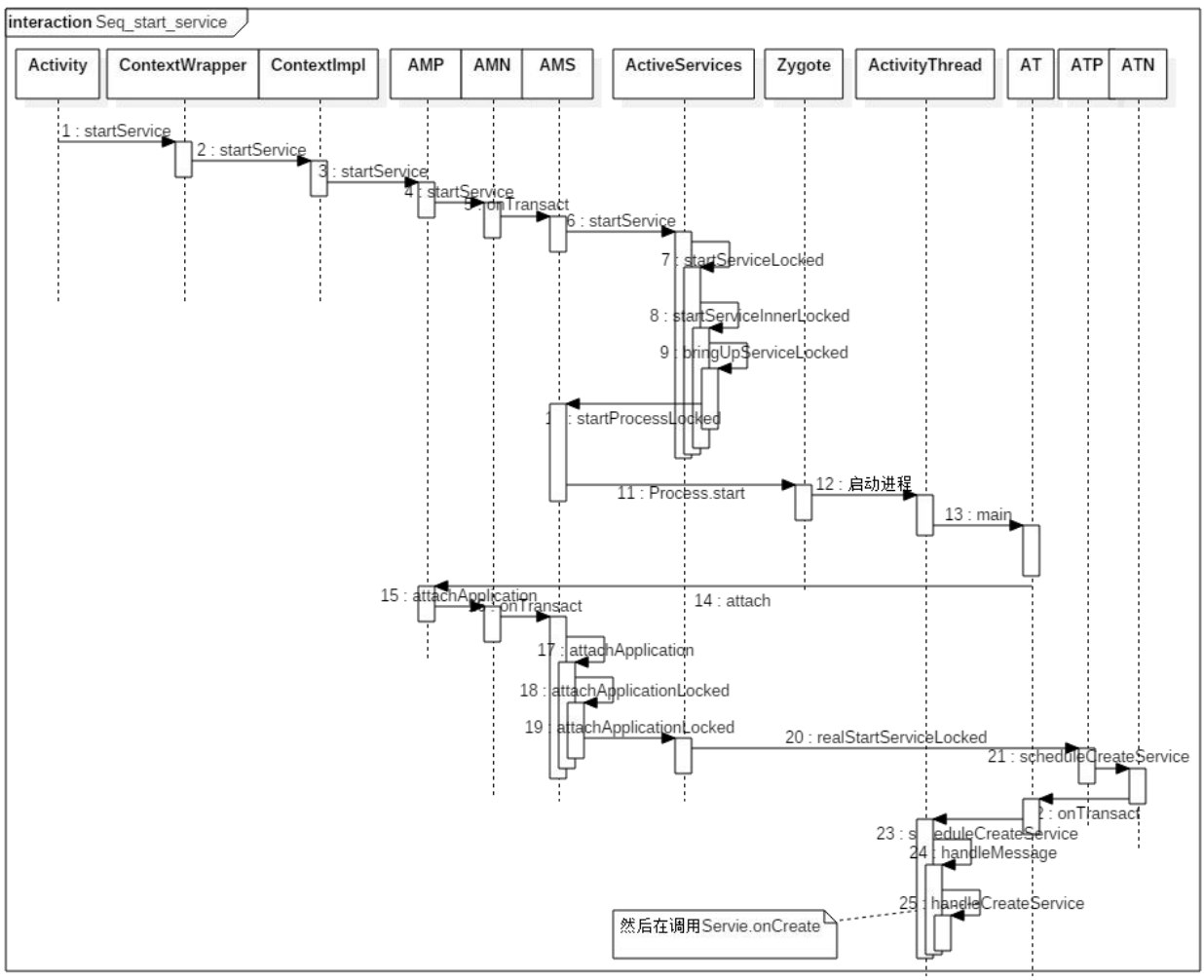
执行此操作的一个好方法是，在片段内定义一个回调接口，并要求宿主 Activity 实现它。

```
public static class FragmentA extends ListFragment {
    ...
    // Container Activity must implement this interface
    public interface OnArticleSelectedListener {
        public void onArticleSelected(Uri articleUri);
    }
    ...
}
public static class FragmentA extends ListFragment {
    OnArticleSelectedListener mListener;
    ...
    @Override
    public void onAttach(Activity activity) {
        super.onAttach(activity);
        try {
            mListener = (OnArticleSelectedListener) activity;
        } catch (ClassCastException e) {
            throw new ClassCastException(activity.toString());
        }
    }
    ...
}
```

2.3 Service

Service 分为两种工作状态，一种是启动状态，主要用于执行后台计算；另一种是绑定状态，主要用于其他组件和 Service 的交互。

2.3.1 启动过程



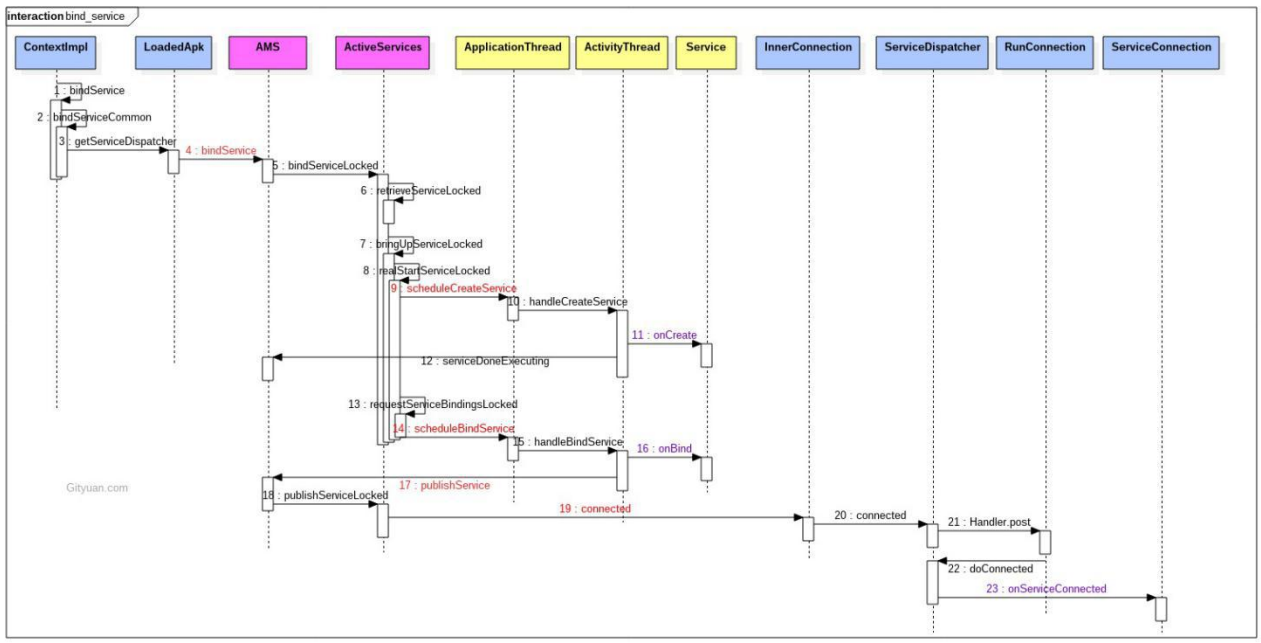
ActivityThread.java

```
@UnsupportedAppUsageprivate void handleCreateService(CreateServiceData data) {
    ...
```

```
LoadedApk packageInfo = getPackageInfoNoCheck(
        data.info.applicationInfo, data.compatInfo);
Service service = null;
try {
    java.lang.ClassLoader cl = packageInfo.getClassLoader();
    service = packageInfo.getAppFactory()
            .instantiateService(cl, data.info.name, data.intent);
}
...


try {
    if (localLOGV) Slog.v(TAG, "Creating service " + data.info.name);

    ContextImpl context = ContextImpl.createAppContext(this, packageInfo);
    context.setOuterContext(service);

    Application app = packageInfo.makeApplication(false, mInstrumentation);
    service.attach(context, this, data.info.name, data.token, app,
            ActivityManager.getService());
    service.onCreate();
    mServices.put(data.token, service);
    try {
        ActivityManager.getService().serviceDoneExecuting(
                data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
    } catch (RemoteException e) {
```

```
            throw e.rethrowFromSystemServer();
        }
    }
    ...
}
```

2.3.2 绑定过程

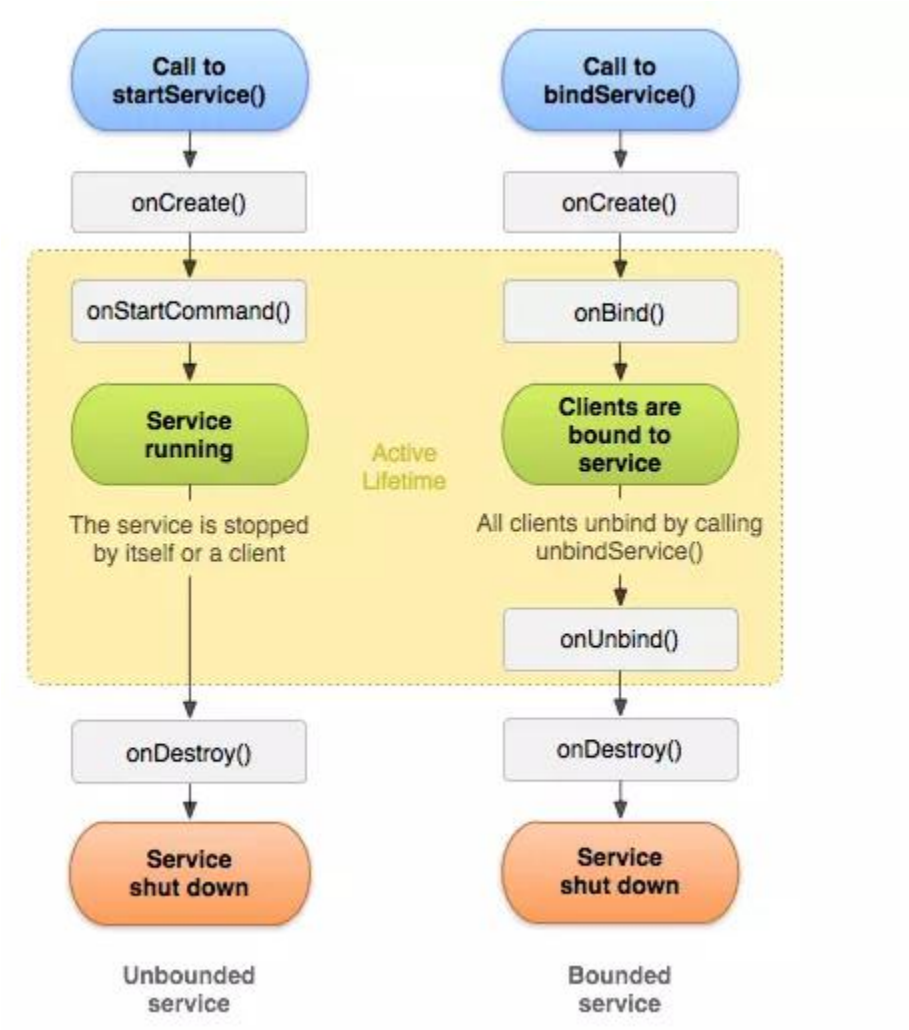


ActivityThread.java

```
private void handleBindService(BindServiceData data) {
    Service s = mServices.get(data.token);
    ...
    if (s != null) {
        try {
            data.intent.setExtrasClassLoader(s.getClassLoader());
            data.intent.prepareToEnterProcess();
            try {
```

```
                if (!data.rebind) {
                    IBinder binder = s.onBind(data.intent);
                    ActivityManager.getService().publishService(
                            data.token, data.intent, binder);
                } else {
                    s.onRebind(data.intent);
                    ActivityManager.getService().serviceDoneExecuting(
                            data.token, SERVICE_DONE_EXECUTING_ANON, 0, 0);
                }
            } catch (RemoteException ex) {
                throw ex.rethrowFromSystemServer();
            }
        }
        ...
    }
}
```

2.3.3 生命周期



| 值 | 说明 ||-----|-----||

START_NOT_STICKY | 如果系统在 onStartCommand() 返回后终止服务，则除非有挂起 Intent 要传递，否则系统不会重建服务。这是最安全的选项，可以避免在不必要时以及应用能够轻松重启所有未完成的作业时运行服务 || START_STICKY | 如果系统在 onStartCommand() 返回后终止服务，则会重建服务并调用 onStartCommand()，但不会重新传递最后一个 Intent。相反，除非有挂起 Intent 要启动服务（在这种情况下，将传递这些 Intent ），否则系统会通过空 Intent 调用 onStartCommand()。这适用于不执行命令、但无限期运行并等待作业的媒体播放器（或类似服务 || START_REDELIVER_INTENT | 如果系统在 onStartCommand() 返回后终止服务，则会重建服务，并通过传递给服务的最后一个 Intent

调用 onStartCommand()。任何挂起 Intent 均依次传递。这适用于主动执行应该立即恢复的作业（例如下载文件）的服务 |

2.3.4 启用前台服务

```
<uses-permission android:name="android.permission.FOREGROUND_SERVICE"/>

Notification notification = new Notification(icon, text, System.currentTimeMillis());Intent notificationIntent = new Intent(this, ExampleActivity.class);PendingIntent pendingIntent = PendingIntent.getActivity(this, 0, notificationIntent, 0);

notification.setLatestEventInfo(this, title, mmessage, pendingIntent);

startForeground(ONGOING_NOTIFICATION_ID, notification);
```

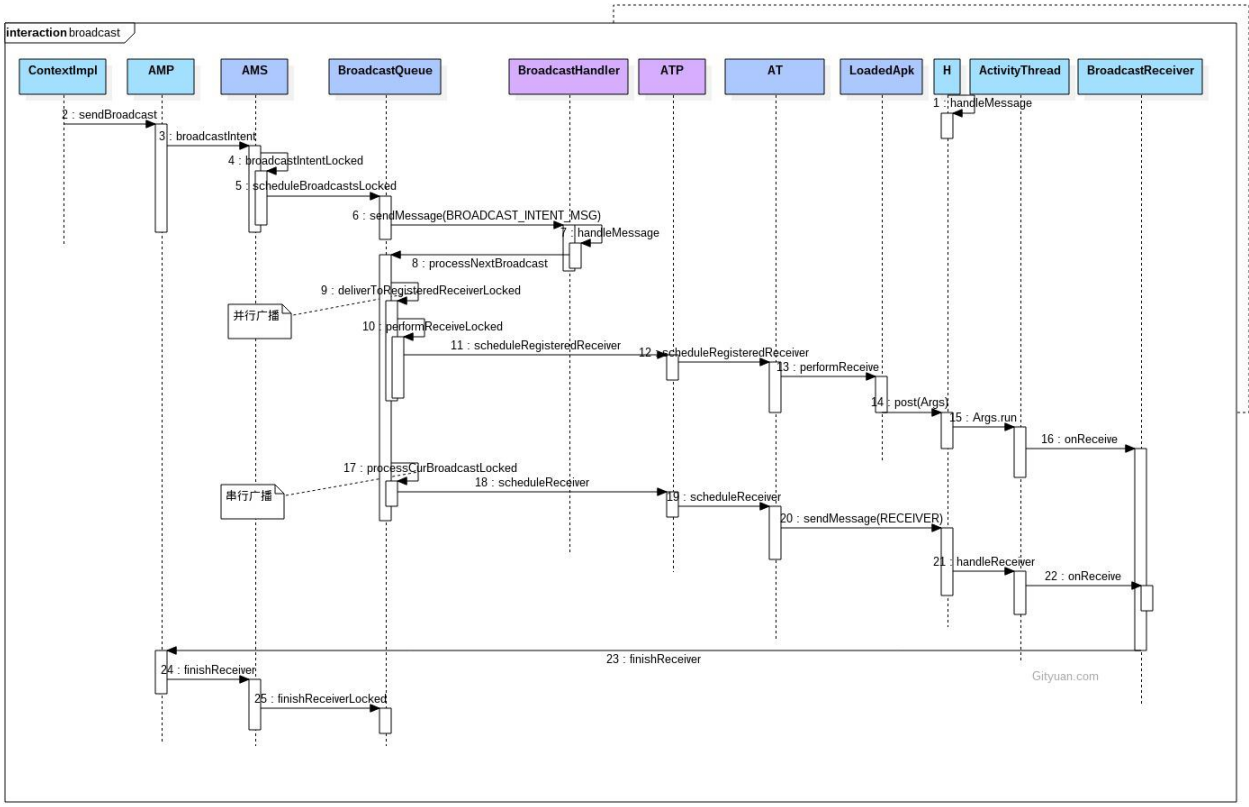
2.4 BroadcastReceiver

target 26 之后，无法在 AndroidManifest 显示声明大部分广播，除了一部分必要的广播，如：

- ACTION_BOOT_COMPLETED
- ACTION_TIME_SET
- ACTION_LOCALE_CHANGED

```
LocalBroadcastManager.getInstance(MainActivity.this).registerReceiver(receiver, filter);
```

2.4.1 注册过程



2.5 ContentProvider

ContentProvider 管理对结构化数据集的访问。它们封装数据，并提供用于定义数据安全性的机制。 内容提供程序是连接一个进程中的数据与另一个进程中运行的代码的标准界面。

ContentProvider 无法被用户感知，对于一个 ContentProvider 组件来说，它的内部需要实现增删该查这四种操作，它的内部维持着一份数据集合，这个数据集合既可以是数据库实现，也可以是其他任何类型，如 List 和 Map，内部的 insert、delete、update、query 方法需要处理好线程同步，因为这几个方法是在 Binder 线程池中被调用的。

ContentProvider 通过 Binder 向其他组件乃至其他应用提供数据。当 ContentProvider 所在的进程启动时，ContentProvider 会同时启动并发布到 AMS 中，需要注意的是，这个时候 ContentProvider 的 onCreate 要先于 Application 的 onCreate 而执行。

2.5.1 基本使用

```
// Queries the user dictionary and returns results

mCursor = getContentResolver().query(

    UserDictionary.Words.CONTENT_URI,   // The content URI of the words table

    mProjection,                        // The columns to return for each row

    mSelectionClause                    // Selection criteria

    mSelectionArgs,                     // Selection criteria

    mSortOrder);                        // The sort order for the returned rows
public class Installer extends ContentProvider {



    @Override

    public boolean onCreate() {

        return true;

    }



    @Nullable

    @Override

    public Cursor query(@NonNull Uri uri, @Nullable String[] projection, @Nullable String
selection, @Nullable String[] selectionArgs, @Nullable String sortOrder) {

        return null;

    }



    @Nullable

    @Override

    public String getType(@NonNull Uri uri) {
```

```
        return null;
    }

    @Nullable
    @Override
    public Uri insert(@NonNull Uri uri, @Nullable ContentValues values) {
        return null;
    }

    @Override
    public int delete(@NonNull Uri uri, @Nullable String selection, @Nullable String[]
selectionArgs) {
        return 0;
    }

    @Override
    public int update(@NonNull Uri uri, @Nullable ContentValues values, @Nullable String
selection, @Nullable String[] selectionArgs) {
        return 0;
    }
}
```

ContentProvider 和 sql 在实现上有什么区别?

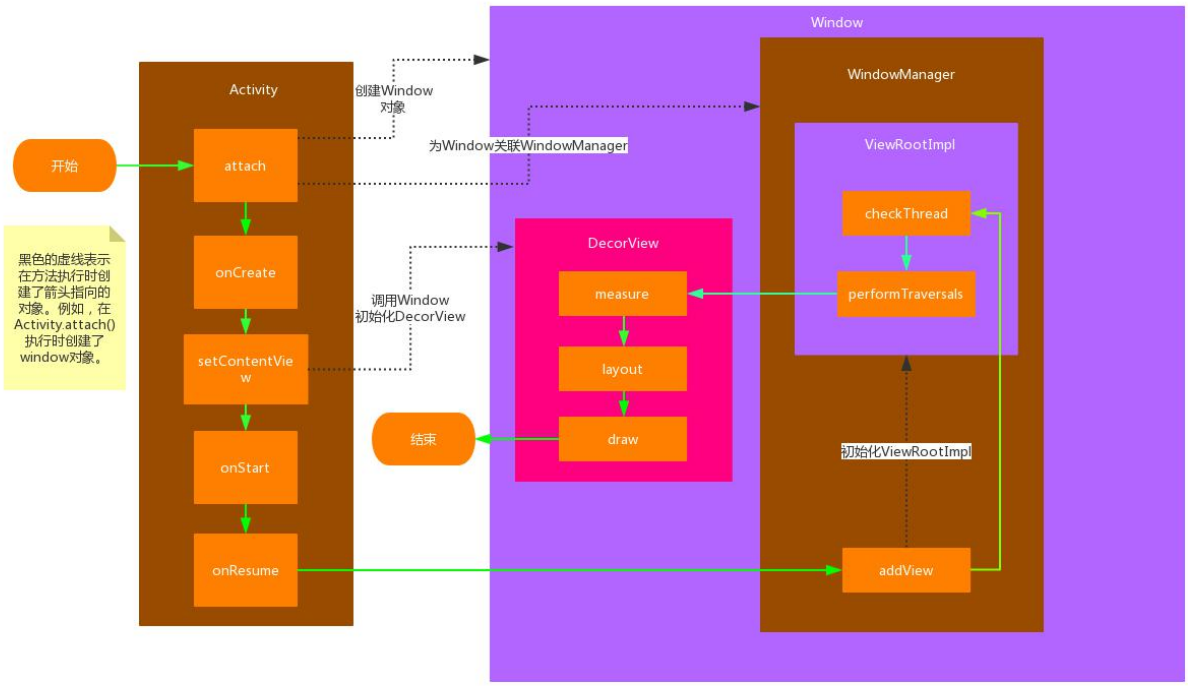
- ContentProvider 屏蔽了数据存储的细节，内部实现透明化，用户只需关心 uri 即可(是否匹配)

- ContentProvider 能实现不同 app 的数据共享，sql 只能是自己程序才能访问
- Contentprovider 还能增删本地的文件,xml 等信息

2.6 数据存储

存储方式	说明
SharedPreferences	在键值对中存储私有原始数据
内部存储	在设备内存中存储私有数据
外部存储	在共享的外部存储中存储公共数据
SQLite 数据库	在私有数据库中存储结构化数据

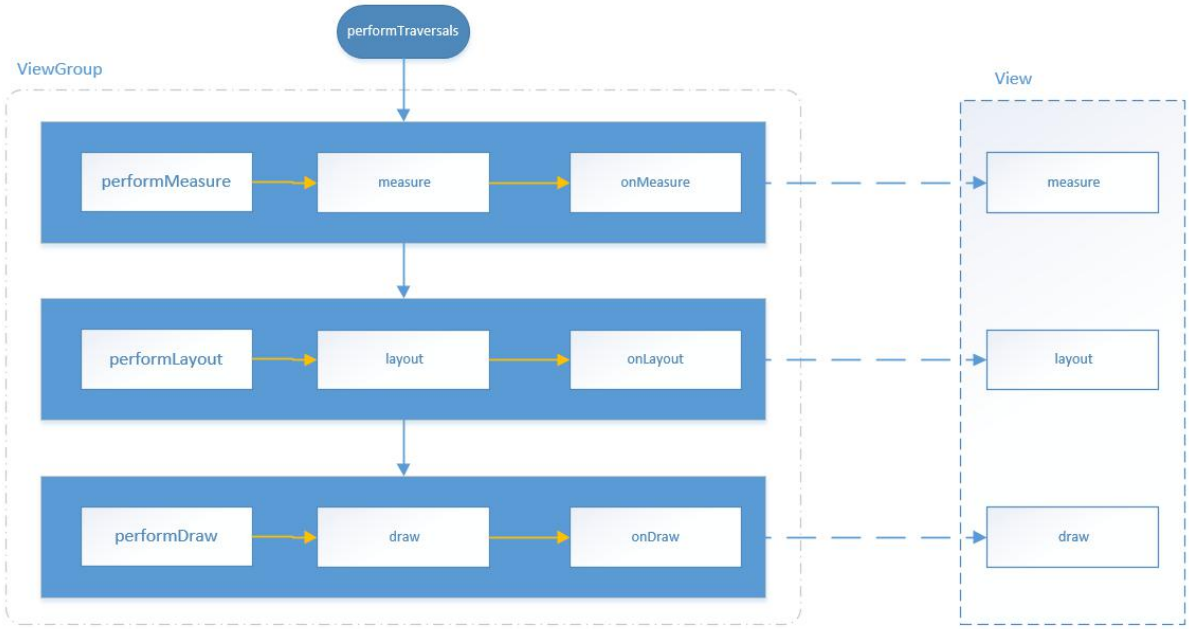
2.7 View



ViewRoot 对应于 ViewRootImpl 类，它是连接 WindowManager 和 DecorView 的纽带，View 的三大流程均是通过 ViewRoot 来完成的。在 ActivityThread 中，当 Activity 对象被创建完毕后，会将 DecorView 添加到 Window 中，同时会创建 ViewRootImpl 对象，并将 ViewRootImpl 对象和 DecorView 建立关联

View 的整个绘制流程可以分为以下三个阶段：

- measure: 判断是否需要重新计算 View 的大小，需要的话则计算
- layout: 判断是否需要重新计算 View 的位置，需要的话则计算
- draw: 判断是否需要重新绘制 View，需要的话则重绘制



performTraversals 工作流程

2.7.1 MeasureSpec

MeasureSpec 表示的是一个 32 位的整形值，它的高 2 位表示测量模式 SpecMode，低 30 位表示某种测量模式下的规格大小 SpecSize。MeasureSpec 是 View 类的一个静态内部类，用来说明应该如何测量这个 View

Mode	说明
UNSPECIFIED	不指定测量模式, 父视图没有限制子视图的大小，子视图可以是想要的任何尺寸，通常用于系统内部，应用开发中很少用到。
EXACTLY	精确测量模式，视图宽高指定为 match_parent 或具体数值时生效，表示父视图已经决定了子视图的精确大小，这种模式下 View 的测量值就是 SpecSize 的值

Mode	说明
AT_MOST	最大值测量模式，当视图的宽高指定为 wrap_content 时生效，此时子视图的尺寸可以是不超过父视图允许的最大尺寸的任何尺寸

对于 DecorView 而言，它的 MeasureSpec 由窗口尺寸和其自身的 LayoutParams 共同决定；对于普通的 View，它的 MeasureSpec 由父视图的 MeasureSpec 和其自身的 LayoutParams 共同决定

childLayoutParams/parentSpecMode	EXACTLY	AT_MOST
dp/px	EXACTLY(childSize)	EXACTLY(childSize)
match_parent	EXACTLY(childSize)	AT_MOST(parentSize)
wrap_content	AT_MOST(parentSize)	AT_MOST(parentSize)

直接继承 View 的控件需要重写 onMeasure 方法并设置 wrap_content 时的自身大小，因为 View 在布局中使用 wrap_content，那么它的 specMode 是 AT_MOST 模式，在这种模式下，它的宽/高等于父容器当前剩余的空间大小，就相当于使用 match_parent。这解决方式如下：

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
    super.onMeasure(widthMeasureSpec, heightMeasureSpec);
    int widthSpecMode = MeasureSpec.getMode(widthMeasureSpec);
    int widthSpecSize = MeasureSpec.getSize(widthMeasureSpec);
    int heightSpecMode = MeasureSpec.getMode(heightMeasureSpec);
    int heightSpecSize = MeasureSpec.getSize(heightMeasureSpec);
    // 在 wrap_content 的情况下指定内部宽/高(mWidth 和 mHeight`)
```

```
    if (widthSpecMode == MeasureSpec.AT_MOST && heightSpecMode == MeasureSpec.AT_MOST) {
        setMeasuredDimension(mWidth, mHeight);
    } else if (widthSpecMode == MeasureSpec.AT_MOST) {
        setMeasureDimension(mWidth, heightSpecSize);
    } else if (heightSpecMode == MeasureSpec.AT_MOST) {
        setMeasureDimension(widthSpecSize, mHeight);
    }
}
```

2.7.2 MotionEvent

事件	说明
ACTION_DOWN	手指刚接触到屏幕
ACTION_MOVE	手指在屏幕上移动
ACTION_UP	手机从屏幕上松开的一瞬间
ACTION_CANCEL	触摸事件取消

点击屏幕后松开，事件序列为 DOWN -> UP，点击屏幕滑动松开，事件序列为 DOWN -> MOVE -> ...> MOVE -> UP。

`getX/getY` 返回相对于当前 View 左上角的坐标，`getRawX/getRawY` 返回相对于屏幕左上角的坐标

TouchSlop 是系统所能识别出的被认为滑动的最小距离，不同设备值可能不相同，可通过 `ViewConfiguration.get(getContext()).getScaledTouchSlop()` 获取。

2.7.3 VelocityTracker

VelocityTracker 可用于追踪手指在滑动中的速度：

```
view.setOnTouchListener(new View.OnTouchListener() {

    @Override

    public boolean onTouch(View v, MotionEvent event) {

        VelocityTracker velocityTracker = VelocityTracker.obtain();

        velocityTracker.addMovement(event);

        velocityTracker.computeCurrentVelocity(1000);

        int xVelocity = (int) velocityTracker.getXVelocity();

        int yVelocity = (int) velocityTracker.getYVelocity();

        velocityTracker.clear();

        velocityTracker.recycle();

        return false;

    }

});
```

2.7.4 GestureDetector

GestureDetector 辅助检测用户的单击、滑动、长按、双击等行为：

```
final GestureDetector mGestureDetector = new GestureDetector(this, new
GestureDetector.OnGestureListener() {

    @Override

    public boolean onDown(MotionEvent e) { return false; }



    @Override
```

```
    public void onShowPress(MotionEvent e) { }

    @Override
    public boolean onSingleTapUp(MotionEvent e) { return false; }

    @Override
    public boolean onScroll(MotionEvent e1, MotionEvent e2, float distanceX, float distanceY)
{ return false; }

    @Override
    public void onLongPress(MotionEvent e) { }

    @Override
    public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX, float velocityY)
{ return false; }
});
mGestureDetector.setOnDoubleTapListener(new OnDoubleTapListener() {
    @Override
    public boolean onSingleTapConfirmed(MotionEvent e) { return false; }

    @Override
    public boolean onDoubleTap(MotionEvent e) { return false; }

    @Override
    public boolean onDoubleTapEvent(MotionEvent e) { return false; }
});// 解决长按屏幕后无法拖动的问题
```

```
mGestureDetector.setIsLongpressEnabled(false);
imageView.setOnTouchListener(new View.OnTouchListener() {
    @Override
    public boolean onTouch(View v, MotionEvent event) {
        return mGestureDetector.onTouchEvent(event);
    }
});
```

如果是监听滑动相关，建议在 onTouchEvent 中实现，如果要监听双击，那么就使用 GestureDectector。

2.7.5 Scroller

弹性滑动对象，用于实现 View 的弹性滑动，**Scroller** 本身无法让 View 弹性滑动，需要和 View 的 computeScroll 方法配合使用。startScroll 方法是无法让 View 滑动的，invalidate 会导致 View 重绘，重回后会在 draw 方法中又会去调用 computeScroll 方法，computeScroll 方法又会去向 Scroller 获取当前的 scrollX 和 scrollY，然后通过 scrollTo 方法实现滑动，接着又调用 postInvalidate 方法如此反复。

```
Scroller mScroller = new Scroller(mContext);
private void smoothScrollTo(int destX) {
    int scrollX = getScrollX();
    int delta = destX - scrollX;
    // 1000ms 内滑向 destX，效果就是慢慢滑动
    mScroller.startScroll(scrollX, 0 , delta, 0, 1000);
    invalidate();
```

```
}
@Overridepublic void computeScroll() {
    if (mScroller.computeScrollOffset()) {
        scrollTo(mScroller.getCurrX(), mScroller.getCurrY());
        postInvalidate();
    }
}
```

2.7.6 View 的滑动

- scrollTo/scrollBy

  适合对 View 内容的滑动。scrollBy 实际上也是调用了 scrollTo 方法：

```
public void scrollTo(int x, int y) {
    if (mScrollX != x || mScrollY != y) {
        int oldX = mScrollX;
        int oldY = mScrollY;
        mScrollX = x;
        mScrollY = y;
        invalidateParentCaches();
        onScrollChanged(mScrollX, mScrollY, oldX, oldY);
        if (!awakenScrollBars()) {
            postInvalidateOnAnimation();
        }
    }
}
public void scrollBy(int x, int y) {
```

```
    scrollTo(mScrollX + x, mScrollY + y);

}
```

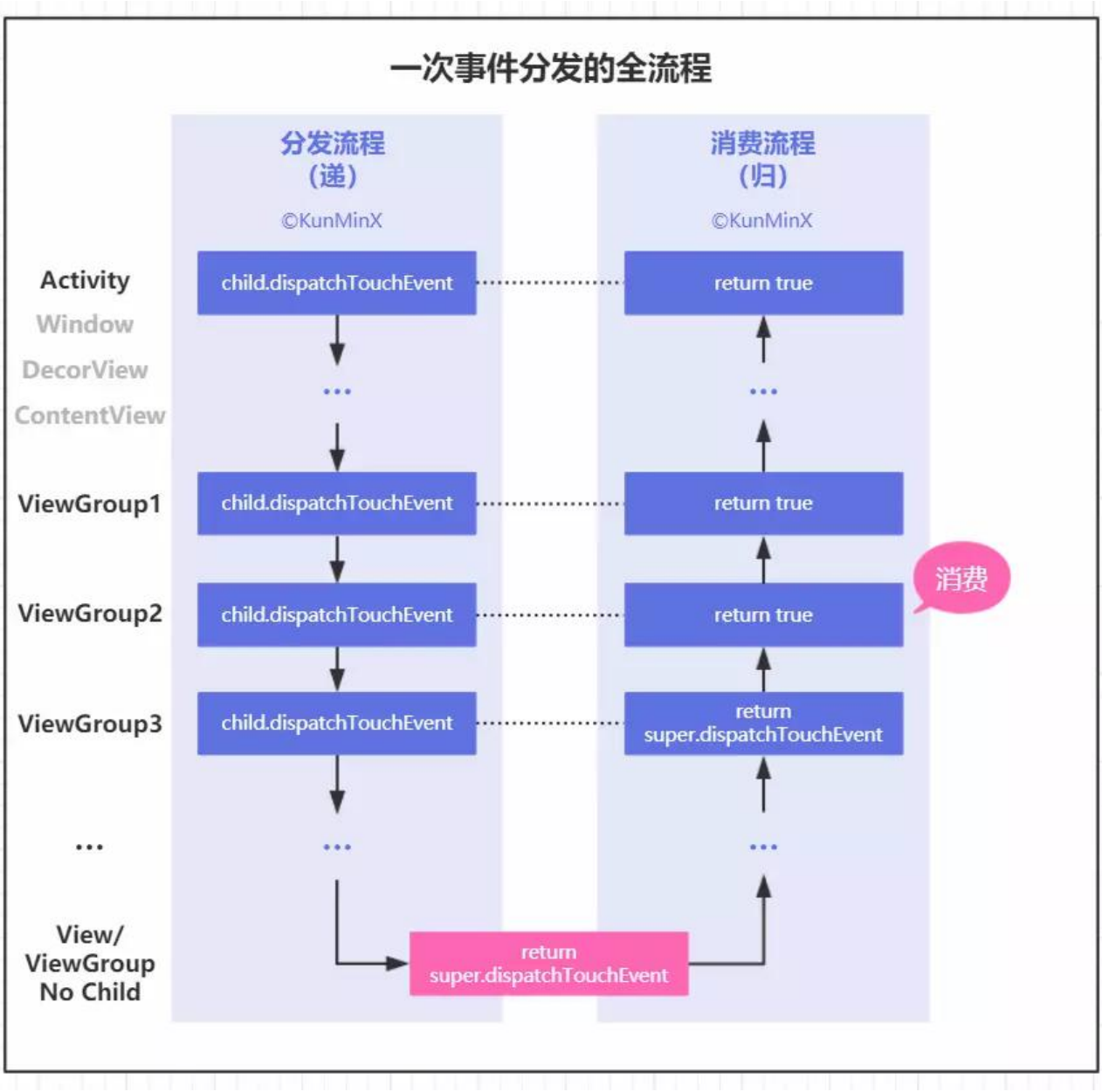
mScrollX 的值等于 View 的左边缘和 View 内容左边缘在水平方向的距离，mScrollY 的值等于 View 上边缘和 View 内容上边缘在竖直方向的距离。scrollTo 和 scrollBy 只能改变 View 内容的位置而不能改变 View 在布局中的位置。

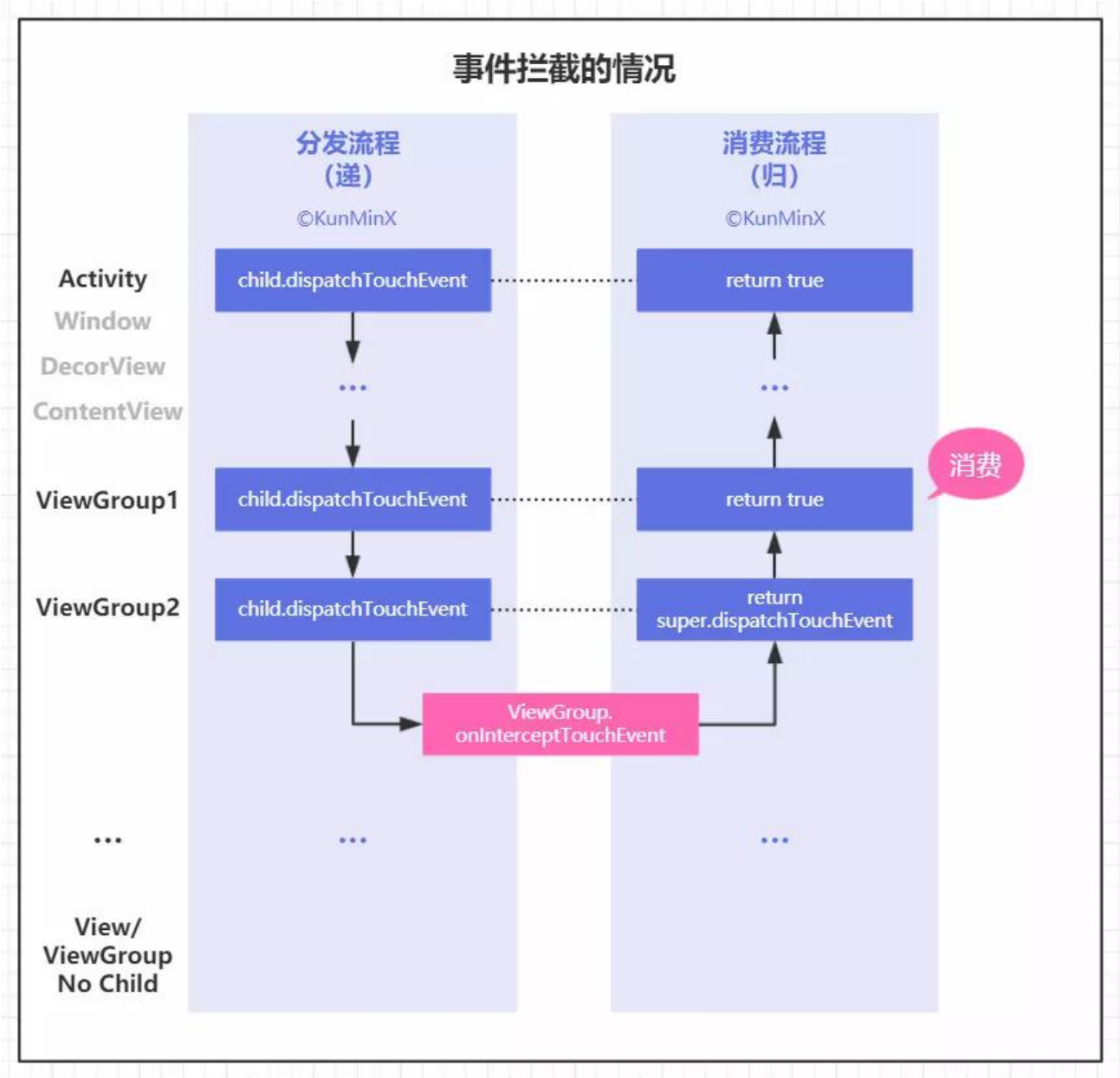
- 使用动画
  操作简单，主要适用于没有交互的 View 和实现复杂的动画效果。
- 改变布局参数 操作稍微复杂，适用于有交互的 View.

```
ViewGroup.MarginLayoutParams params = (ViewGroup.MarginLayoutParams)
view.getLayoutParams();

params.width += 100;

params.leftMargin += 100;

view.requestLayout();//或者 view.setLayoutParams(params);
```

2.7.7 View 的事件分发

点击事件达到顶级 View(一般是一个 ViewGroup)，会调用 ViewGroup 的 dispatchTouchEvent 方法，如果顶级 ViewGroup 拦截事件即 onInterceptTouchEvent 返回 true，则事件由 ViewGroup 处理，这时如果 ViewGroup 的 mOnTouchListener 被设置，则 onTouch 会被调用，否则 onTouchEvent 会被调用。也就是说如果都提供的话，onTouch 会屏蔽掉 onTouchEvent。在 onTouchEvent 中，如果设置了 mOnClickListenser，则 onClick 会被调用。如果顶级 ViewGroup 不拦截事件，则事件会传递给它所在的点击事件链上的子 View，这时子 View 的 dispatchTouchEvent 会被调用。如此循环。


一次事件分发的全流程
分发流程
（递）
©KunMinX
消费流程
（归）
©KunMinX
Activity
Window
DecorView
ContentView
ViewGroup1
ViewGroup2
ViewGroup3
...
View/
ViewGroup
No Child
child.dispatchTouchEvent
child.dispatchTouchEvent
child.dispatchTouchEvent
child.dispatchTouchEvent
return
super.dispatchTouchEvent
return true
return true
return true
return
super.dispatchTouchEvent
消费



- ViewGroup 默认不拦截任何事件。ViewGroup 的 onInterceptTouchEvent 方法默认返回 false。
- View 没有 onInterceptTouchEvent 方法，一旦有点击事件传递给它，onTouchEvent 方法就会被调用。
- View 在可点击状态下，onTouchEvent 默认会消耗事件。

- ACTION_DOWN 被拦截了，onInterceptTouchEvent 方法执行一次后，就会留下记号（mFirstTouchTarget == null）那么往后的 ACTION_MOVE 和 ACTION_UP 都会拦截。

、

2.7.8 在 Activity 中获取某个 View 的宽高

- Activity/View#onWindowFocusChanged

```
// 此时 View 已经初始化完毕

// 当 Activity 的窗口得到焦点和失去焦点时均会被调用一次

// 如果频繁地进行 onResume 和 onPause，那么 onWindowFocusChanged 也会被频繁地调用

public void onWindowFocusChanged(boolean hasFocus) {

    super.onWindowFocusChanged(hasFocus);

    if (hasFocus) {

        int width = view.getMeasureWidth();

        int height = view.getMeasuredHeight();

    }

}
```

- view.post(runnable)

```
// 通过 post 可以将一个 runnable 投递到消息队列的尾部，// 然后等待 Looper 调用次 runnable 的时候，View 也已经初

// 始化好了

protected void onStart() {

    super.onStart();

    view.post(new Runnable() {
```

```
        @Override
        public void run() {
            int width = view.getMeasuredWidth();
            int height = view.getMeasuredHeight();
        }
    });
}
```

- ViewTreeObserver

```
// 当 View 树的状态发生改变或者 View 树内部的 View 的可见// 性发生改变时，onGlobalLayout 方法将被回调 protected void onStart() {
    super.onStart();

    ViewTreeObserver observer = view.getViewTreeObserver();
    observer.addOnGlobalLayoutListener(new OnGlobalLayoutListener() {

        @SuppressWarnings("deprecation")
        @Override
        public void onGlobalLayout() {
            view.getViewTreeObserver().removeGlobalOnLayoutListener(this);
            int width = view.getMeasuredWidth();
            int height = view.getMeasuredHeight();
        }
    });
}
```

2.7.9 Draw 的基本流程

```
// 绘制基本上可以分为六个步骤 public void draw(Canvas canvas) {

    ...

    // 步骤一：绘制 View 的背景

    drawBackground(canvas);

    ...

    // 步骤二：如果需要的话，保持 canvas 的图层，为 fading 做准备

    saveCount = canvas.getSaveCount();

    ...

    canvas.saveLayer(left, top, right, top + length, null, flags);

    ...

    // 步骤三：绘制 View 的内容

    onDraw(canvas);

    ...

    // 步骤四：绘制 View 的子 View

    dispatchDraw(canvas);

    ...

    // 步骤五：如果需要的话，绘制 View 的 fading 边缘并恢复图层

    canvas.drawRect(left, top, right, top + length, p);

    ...

    canvas.restoreToCount(saveCount);

    ...

    // 步骤六：绘制 View 的装饰(例如滚动条等等)

    onDrawForeground(canvas)
```

```
}
```

2.7.10 自定义 View

- 继承 View 重写 onDraw 方法

主要用于实现一些不规则的效果，静态或者动态地显示一些不规则的图形，即重写 onDraw 方法。采用这种方式需要自己支持 wrap_content，并且 padding 也需要自己处理。

- 继承 ViewGroup 派生特殊的 Layout

主要用于实现自定义布局，采用这种方式需要合适地处理 ViewGroup 的测量、布局两个过程，并同时处理子元素的测量和布局过程。

- 继承特定的 View

用于扩张某种已有的 View 的功能

- 继承特定的 ViewGroup

用于扩张某种已有的 ViewGroup 的功能

2.8 进程

当某个应用组件启动且该应用没有运行其他任何组件时，Android 系统会使用单个执行线程为应用启动新的 Linux 进程。默认情况下，同一应用的所有组件在相同的进程和线程（称为“主”线程）中运行。

各类组件元素的清单文件条目<activity>、<service>、<receiver> 和 <provider>—均支持 android:process 属性，此属性可以指定该组件应在哪个进程运行。

2.8.1 进程生命周期

1、前台进程

- 托管用户正在交互的 Activity（已调用 Activity 的 onResume() 方法）
- 托管某个 Service，后者绑定到用户正在交互的 Activity
- 托管正在“前台”运行的 Service（服务已调用 startForeground()）
- 托管正执行一个生命周期回调的 Service（onCreate()、onStart() 或 onDestroy()）
- 托管正执行其 onReceive() 方法的 BroadcastReceiver

2、可见进程

- 托管不在前台、但仍对用户可见的 Activity（已调用其 onPause() 方法）。例如，如果 re 前台 Activity 启动了一个对话框，允许在其后显示上一 Activity，则有可能会发生这种情况。
- 托管绑定到可见（或前台）Activity 的 Service

3、服务进程

- 正在运行已使用 startService() 方法启动的服务且不属于上述两个更高类别进程的进程。

4、后台进程

- 包含目前对用户不可见的 Activity 的进程（已调用 Activity 的 onStop() 方法）。通常会有很多后台进程在运行，因此它们会保存在 LRU （最近最少使用）列表中，以确保包含用户最近查看的 Activity 的进程最后一个被终止。

5、空进程

- 不含任何活动应用组件的进程。保留这种进程的的唯一目的是用作缓存，以缩短下次在其中运行组件所需的启动时间。 为使总体系统资源在进程缓存和底层内核缓存之间保持平衡，系统往往会终止这些进程。\

2.8.2 多进程

如果注册的四大组件中的任意一个组件时用到了多进程，运行该组件时，都会创建一个新的 Application 对象。对于多进程重复创建 Application 这种情况，只需要在该类中对当前进程加以判断即可。

```
public class MyApplication extends Application {


    @Override
    public void onCreate() {
        Log.d("MyApplication", getProcessName(android.os.Process.myPid()));
        super.onCreate();
    }


    /**     * 根据进程 ID 获取进程名     * @param pid 进程 id     * @return 进程名     */
    public  String getProcessName(int pid){
        ActivityManager am = (ActivityManager)getSystemService(Context.ACTIVITY_SERVICE);
        List<ActivityManager.RunningAppProcessInfo> processInfoList =
am.getRunningAppProcesses();
```

```
        if (processInfoList == null) {
            return null;
        }
        for (ActivityManager.RunningAppProcessInfo processInfo : processInfoList) {
            if (processInfo.pid == pid) {
                return processInfo.processName;
            }
        }
        return null;
    }
}
```

一般来说，使用多进程会造成以下几个方面的问题：

- 静态成员和单例模式完全失效
- 线程同步机制完全失效
- SharedPreferences 的可靠性下降
- Application 会多次创建

2.8.3 进程存活

2.8.3.1 OOM_ADJ

ADJ 级别	取值	解释
UNKNOWN_ADJ	16	一般指将要会缓存进程，无法获取确定值
CACHED_APP_MAX_ADJ	15	不可见进程的 adj 最大值

ADJ 级别	取值	解释
CACHED_APP_MIN_ADJ	9	不可见进程的 adj 最小值
SERVICE_B_AD	8	B List 中的 Service（较老的、使用可能性更小）
PREVIOUS_APP_ADJ	7	上一个 App 的进程(往往通过按返回键)
HOME_APP_ADJ	6	Home 进程
SERVICE_ADJ	5	服务进程(Service process)
HEAVY_WEIGHT_APP_ADJ	4	后台的重量级进程，system/rootdir/init.rc 文件中设置
BACKUP_APP_ADJ	3	备份进程
PERCEPTIBLE_APP_ADJ	2	可感知进程，比如后台音乐播放
VISIBLE_APP_ADJ	1	可见进程(Visible process)
FOREGROUND_APP_ADJ	0	前台进程（Foreground process)
PERSISTENT_SERVICE_ADJ	-11	关联着系统或 persistent 进程
PERSISTENT_PROC_ADJ	-12	系统 persistent 进程，比如 telephony
SYSTEM_ADJ	-16	系统进程
NATIVE_ADJ	-17	native 进程（不被系统管理）

2.8.3.2 进程被杀情况

进程杀死场景	调用接口	可能影响范围
触发系统进程管理机制	Lowmemorykiller	从进程importance值由大到小依次杀死，释放内存
被第三方应用杀死（无Root）	killBackgroundProcess	只能杀死OOM_ADJ为4以上的进程
被第三方应用杀死（有Root）	force-stop或者kill	理论上可以杀所有进程，一般只杀非系统关键进程和非前台和可见进程
厂商杀进程功能	force-stop或者kill	理论上可以杀所有进程，包括Native进程
用户主动"强行停止"进程	force-stop	只能停用第三方和非system/phone进程应用（停用system进程应用会造成Android重启）

2.8.3.3 进程保活方案

- 开启一个像素的 Activity
- 使用前台服务
- 多进程相互唤醒
- JobSheduler 唤醒
- 粘性服务 & 与系统服务捆绑

2.9 Parcelable 接口

只要实现了 Parcelable 接口，一个类的对象就可以实现序列化并可以通过 Intent 和 Binder 传递。

2.9.1 使用示例

```
import android.os.Parcel;import android.os.Parcelable;

public class User implements Parcelable {


    private int userId;


    protected User(Parcel in) {
        userId = in.readInt();
    }


    public static final Creator<User> CREATOR = new Creator<User>() {
        @Override
        public User createFromParcel(Parcel in) {
            return new User(in);
        }


        @Override
        public User[] newArray(int size) {
            return new User[size];
        }
    };


    @Override
    public int describeContents() {
```

```
        return 0;
    }

    @Override
    public void writeToParcel(Parcel dest, int flags) {
        dest.writeInt(userId);
    }

    public int getUserId() {
        return userId;
    }
}
```

2.9.2 方法说明

Parcel 内部包装了可序列化的数据，可以在 Binder 中自由传输。序列化功能由 `writeToParcel` 方法完成，最终是通过 Parcel 中的一系列 write 方法完成。反序列化功能由 CREATOR 来完成，通过 Parcel 的一系列 read 方法来完成反序列化过程。

方法	功能
createFromParcel(Parcel in)	从序列化后的对象中创建原始对象
newArray(int size)	创建指定长度的原始对象数组
User(Parcel in)	从序列化后的对象中创建原始对象
writeToParcel(Parcel dest,	将当前对象写入序列化结构中，其中 flags 标识有两种值：0

方法	功能
int flags)	或者 1。为 1 时标识当前对象需要作为返回值返回，不能立即释放资源，几乎所有情况都为 0
describeContents	返回当前对象的内容描述。如果含有文件描述符，返回 1，否则返回 0，几乎所有情况都返回 0

2.9.3 Parcelable 与 Serializable 对比

- Serializable 使用 I/O 读写存储在硬盘上，而 Parcelable 是直接在内存中读写
- Serializable 会使用反射，序列化和反序列化过程需要大量 I/O 操作， Parcelable 自已实现封送和解封（marshalled &unmarshalled）操作不需要用反射，数据也存放在 Native 内存中，效率要快很多

2.10 IPC

IPC 即 Inter-Process Communication (进程间通信)。Android 基于 Linux，而 Linux 出于安全考虑，不同进程间不能之间操作对方的数据，这叫做“进程隔离”。

> 在 Linux 系统中，虚拟内存机制为每个进程分配了线性连续的内存空间，操作系统将这种虚拟内存空间映射到物理内存空间，每个进程有自己的虚拟内存空间，进而不能操作其他进程的内存空间，只有操作系统才有权限操作物理内存空间。 进程隔离保证了每个进程的内存安全。

2.10.1 IPC 方式

名称	优点	缺点	适用场景

名称	优点	缺点	适用场景
Bundle	简单易用	只能传输 Bundle 支持的数据类型	四大组件间的进程间通信
文件共享	简单易用	不适合高并发场景，并且无法做到进程间即时通信	无并发访问情形，交换简单的数据实时性不高的场景
AIDL	功能强大，支持一对多并发通信，支持实时通信	使用稍复杂，需要处理好线程同步	一对多通信且有 RPC 需求
Messenger	功能一般，支持一对多串行通信，支持实时通信	不能很处理高并发清醒，不支持 RPC，数据通过 Message 进行传输，因此只能传输 Bundle 支持的数据类型	低并发的一对多即时通信，无 RPC 需求，或者无需返回结果的 RPC 需求
ContentProvider	在数据源访问方面功能强大，支持一对多并发数据共享，可通过 Call 方法扩展其他操作	可以理解为受约束的 AIDL，主要提供数据源的 CRUD 操作	一对多的进程间数据共享
Socket	可以通过网络传输字节流，支持一对多并发实时通信	实现细节稍微有点烦琐，不支持直接的 RPC	网络数据交换

2.10.2 Binder

Binder 是 Android 中的一个类，实现了 IBinder 接口。从 IPC 角度来说，Binder 是 Android 中的一种扩进程通信方方式。从 Android 应用层来说，Binder 是客户端和服务器端进行通信的媒介，当 bindService 的时候，服务端会返回一个包含了服务端业务调用的 Binder 对象。

Binder 相较于传统 IPC 来说更适合于 Android 系统，具体原因的包括如下三点：

- Binder 本身是 C/S 架构的，这一点更符合 Android 系统的架构
- 性能上更有优势：管道，消息队列，Socket 的通讯都需要两次数据拷贝，而 Binder 只需要一次。要知道，对于系统底层的 IPC 形式，少一次数据拷贝，对整体性能的影响是非常之大的
- 安全性更好：传统 IPC 形式，无法得到对方的身份标识（UID/GID)，而在使用 Binder IPC 时，这些身份标示是跟随调用过程而自动传递的。Server 端很容易就可以知道 Client 端的身份，非常便于做安全检查

示例：

- **新建 AIDL 接口文件**

RemoteService.aidl

```
package com.example.mystudyapplication3;

interface IRemoteService {


    int getUserId();


}
```

系统会自动生成 IRemoteService.java:

```
/* * This file is auto-generated.  DO NOT MODIFY. */package
com.example.mystudyapplication3;// Declare any non-default types here with import
statements//import com.example.mystudyapplication3.IUserBean;

public interface IRemoteService extends android.os.IInterface {

    /**     * Local-side IPC implementation stub class.     */
```

```
    public static abstract class Stub extends android.os.Binder implements
com.example.mystudyapplication3.IRemoteService {

        private static final java.lang.String DESCRIPTOR =
"com.example.mystudyapplication3.IRemoteService";



        /**         * Construct the stub at attach it to the interface.         */

        public Stub() {

            this.attachInterface(this, DESCRIPTOR);

        }



        /**         * Cast an IBinder object into an
com.example.mystudyapplication3.IRemoteService interface,         * generating a proxy if
needed.         */

        public static com.example.mystudyapplication3.IRemoteService
asInterface(android.os.IBinder obj) {

            if ((obj == null)) {

                return null;

            }

            android.os.IInterface iin = obj.queryLocalInterface(DESCRIPTOR);

            if (((iin != null) && (iin instanceof
com.example.mystudyapplication3.IRemoteService))) {

                return ((com.example.mystudyapplication3.IRemoteService) iin);

            }

            return new com.example.mystudyapplication3.IRemoteService.Stub.Proxy(obj);

        }



        @Override

        public android.os.IBinder asBinder() {
```

```
            return this;
        }

        @Override
        public boolean onTransact(int code, android.os.Parcel data, android.os.Parcel reply,
int flags) throws android.os.RemoteException {
            java.lang.String descriptor = DESCRIPTOR;
            switch (code) {
                case INTERFACE_TRANSACTION: {
                    reply.writeString(descriptor);
                    return true;
                }
                case TRANSACTION_getUserId: {
                    data.enforceInterface(descriptor);
                    int _result = this.getUserId();
                    reply.writeNoException();
                    reply.writeInt(_result);
                    return true;
                }
                default: {
                    return super.onTransact(code, data, reply, flags);
                }
            }
        }
```

```
        private static class Proxy implements
com.example.mystudyapplication3.IRemoteService {

            private android.os.IBinder mRemote;



            Proxy(android.os.IBinder remote) {

                mRemote = remote;

            }



            @Override

            public android.os.IBinder asBinder() {

                return mRemote;

            }



            public java.lang.String getInterfaceDescriptor() {

                return DESCRIPTOR;

            }



            @Override

            public int getUserId() throws android.os.RemoteException {

                android.os.Parcel _data = android.os.Parcel.obtain();

                android.os.Parcel _reply = android.os.Parcel.obtain();

                int _result;

                try {

                    _data.writeInterfaceToken(DESCRIPTOR);

                    mRemote.transact(Stub.TRANSACTION_getUserId, _data, _reply, 0);
```

```
                    _reply.readException();
                    _result = _reply.readInt();
                } finally {
                    _reply.recycle();
                    _data.recycle();
                }
                return _result;
            }
        }

        static final int TRANSACTION_getUserId = (android.os.IBinder.FIRST_CALL_TRANSACTION
+ 0);
    }

    public int getUserId() throws android.os.RemoteException;
}
```

方法	含义
DESCRIPTOR	Binder 的唯一标识，一般用当前的 Binder 的类名表示
asInterface(IBinder obj)	将服务端的 Binder 对象成客户端所需的 AIDL 接口类型对象，这种转换过程是区分进程的，如果位于同一进程，返回的就是 Stub 对象本身，否则返回的是系统封装后的 Stub.proxy 对象。
asBinder	用于返回当前 Binder 对象
onTransact	运行在服务端中的 Binder 线程池中，远程请求会通过系统底层封装

方法	含义
	后交由此方法来处理

定向 tag	含义
in	数据只能由客户端流向服务端，服务端将会收到客户端对象的完整数据，客户端对象不会因为服务端对传参的修改而发生变动。
out	数据只能由服务端流向客户端，服务端将会收到客户端对象，该对象不为空，但是它里面的字段为空，但是在服务端对该对象作任何修改之后客户端的传参对象都会同步改动。
inout	服务端将会接收到客户端传来对象的完整信息，并且客户端将会同步服务端对该对象的任何变动。

2.10.3 AIDL 通信

Android Interface Definition Language

使用示例：

- **新建 AIDL 接口文件**

```
// RemoteService.aidlpackage com.example.mystudyapplication3;

interface IRemoteService {



    int getUserId();



}
```

- **创建远程服务**

```
public class RemoteService extends Service {

```

```
    private int mId = -1;


    private Binder binder = new IRemoteService.Stub() {


        @Override
        public int getUserId() throws RemoteException {
            return mId;
        }
    };


    @Nullable
    @Override
    public IBinder onBind(Intent intent) {
        mId = 1256;
        return binder;
    }
}
```

- 声明远程服务

```
<service
    android:name=".RemoteService"
    android:process=":aidl" />
```

- 绑定远程服务

```
public class MainActivity extends AppCompatActivity {


    public static final String TAG = "wzq";
```

```
    IRemoteService iRemoteService;
    private ServiceConnection mConnection = new ServiceConnection() {
        @Override
        public void onServiceConnected(ComponentName name, IBinder service) {
            iRemoteService = IRemoteService.Stub.asInterface(service);
            try {
                Log.d(TAG, String.valueOf(iRemoteService.getUserId()));
            } catch (RemoteException e) {
                e.printStackTrace();
            }
        }

        @Override
        public void onServiceDisconnected(ComponentName name) {
            iRemoteService = null;
        }
    };

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
        bindService(new Intent(MainActivity.this, RemoteService.class), mConnection,
Context.BIND_AUTO_CREATE);
```

```
    }
}
```

2.10.4 Messenger

Messenger 可以在不同进程中传递 Message 对象，在 Message 中放入我们需要传递的数据，就可以轻松地实现数据的进程间传递了。Messenger 是一种轻量级的 IPC 方案，底层实现是 AIDL。

2.11 Window / WindowManager

2.11.1 Window 概念与分类

Window 是一个抽象类，它的具体实现是 PhoneWindow。WindowManager 是外界访问 Window 的入口，Window 的具体实现位于 WindowManagerService 中，WindowManager 和 WindowManagerService 的交互是一个 IPC 过程。Android 中所有的视图都是通过 Window 来呈现，因此 Window 实际是 View 的直接管理者。

Window 类型	说明	层级
Application Window	对应着一个 Activity	1~99
Sub Window	不能单独存在，只能附属在父 Window 中，如 Dialog 等	1000~1999
System Window	需要权限声明，如 Toast 和 系统状态栏等	2000~2999

2.11.2 Window 的内部机制

Window 是一个抽象的概念，每一个 Window 对应着一个 View 和一个 ViewRootImpl。

Window 实际是不存在的，它是以 View 的形式存在。对 Window 的访问必须通过

WindowManager，WindowManager 的实现类是 WindowManagerImpl：

WindowManagerImpl.java

```
@Overridepublic void addView(@NonNull View view, @NonNull ViewGroup.LayoutParams params) {
    applyDefaultToken(params);
    mGlobal.addView(view, params, mContext.getDisplay(), mParentWindow);
}
@Overridepublic void updateViewLayout(@NonNull View view, @NonNull ViewGroup.LayoutParams
params) {
    applyDefaultToken(params);
    mGlobal.updateViewLayout(view, params);
}
@Overridepublic void removeView(View view) {
    mGlobal.removeView(view, false);
}
```

WindowManagerImpl 没有直接实现 Window 的三大操作，而是全部交给

WindowManagerGlobal 处理，WindowManagerGlobal 以工厂的形式向外提供自己的实例：

WindowManagerGlobal.java

```
// 添加 public void addView(View view, ViewGroup.LayoutParams params,
        Display display, Window parentWindow) {
    ...
```

```
// 子 Window 的话需要调整一些布局参数
final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams) params;
if (parentWindow != null) {
    parentWindow.adjustLayoutParamsForSubWindow(wparams);
} else {
    ...
}
ViewRootImpl root;
View panelParentView = null;
synchronized (mLock) {
    // 新建一个 ViewRootImpl，并通过其 setView 来更新界面完成 Window 的添加过程
    ...
    root = new ViewRootImpl(view.getContext(), display);
    view.setLayoutParams(wparams);
    mViews.add(view);
    mRoots.add(root);
    mParams.add(wparams);
    // do this last because it fires off messages to start doing things
    try {
        root.setView(view, wparams, panelParentView);
    } catch (RuntimeException e) {
        // BadTokenException or InvalidDisplayException, clean up.
        if (index >= 0) {
            removeViewLocked(index, true);
        }
```

```
            throw e;
        }
    }
}
// 删除@UnsupportedAppUsagepublic void removeView(View view, boolean immediate) {
    ...
    synchronized (mLock) {
        int index = findViewLocked(view, true);
        View curView = mRoots.get(index).getView();
        removeViewLocked(index, immediate);
        ...
    }
}
private void removeViewLocked(int index, boolean immediate) {
    ViewRootImpl root = mRoots.get(index);
    View view = root.getView();
    if (view != null) {
        InputMethodManager imm = InputMethodManager.getInstance();
        if (imm != null) {
            imm.windowDismissed(mViews.get(index).getWindowToken());
        }
    }
    boolean deferred = root.die(immediate);
    if (view != null) {
        view.assignParent(null);
```

```
        if (deferred) {
            mDyingViews.add(view);
        }
    }
}
// 更新 public void updateViewLayout(View view, ViewGroup.LayoutParams params) {
    ...
    final WindowManager.LayoutParams wparams = (WindowManager.LayoutParams)params;
    view.setLayoutParams(wparams);
    synchronized (mLock) {
        int index = findViewLocked(view, true);
        ViewRootImpl root = mRoots.get(index);
        mParams.remove(index);
        mParams.add(index, wparams);
        root.setLayoutParams(wparams, false);
    }
}
```

在 ViewRootImpl 中最终会通过 WindowSession 来完成 Window 的添加、更新、删除工作，mWindowSession 的类型是 IWindowSession，是一个 Binder 对象，真正地实现类是 Session，是一个 IPC 过程。

2.11.3 Window 的创建过程

2.11.3.1 Activity 的 Window 创建过程

在 Activity 的创建过程中，最终会由 ActivityThread 的 performLaunchActivity() 来完成整个启动过程，该方法内部会通过类加载器创建 Activity 的实例对象，并调用 attach 方法关联一系列上下文环境变量。在 Activity 的 attach 方法里，系统会创建所属的 Window 对象并设置回调接口，然后在 Activity 的 setContentView 方法中将视图附属在 Window 上：

Activity.java

```
final void attach(Context context, ActivityThread aThread,
        Instrumentation instr, IBinder token, int ident,
        Application application, Intent intent, ActivityInfo info,
        CharSequence title, Activity parent, String id,
        NonConfigurationInstances lastNonConfigurationInstances,
        Configuration config, String referrer, IVoiceInteractor voiceInteractor,
        Window window, ActivityConfigCallback activityConfigCallback) {
    attachBaseContext(context);

    mFragments.attachHost(null /*parent*/);

    mWindow = new PhoneWindow(this, window, activityConfigCallback);
    mWindow.setWindowControllerCallback(this);
    mWindow.setCallback(this);
    mWindow.setOnWindowDismissedCallback(this);
    mWindow.getLayoutInflater().setPrivateFactory(this);
    if (info.softInputMode != WindowManager.LayoutParams.SOFT_INPUT_STATE_UNSPECIFIED) {
        mWindow.setSoftInputMode(info.softInputMode);
    }
```

```
    if (info.uiOptions != 0) {
        mWindow.setUiOptions(info.uiOptions);
    }
    ...
}
...
public void setContentView(@LayoutRes int layoutResID) {
    getWindow().setContentView(layoutResID);
    initWindowDecorActionBar();
}
```

PhoneWindow.java

```
@Overridepublic void setContentView(int layoutResID) {
    if (mContentParent == null) { // 如果没有 DecorView，就创建
        installDecor();
    } else {
        mContentParent.removeAllViews();
    }
    mLayoutInflater.inflate(layoutResID, mContentParent);
    final Callback cb = getCallback();
    if (cb != null && !isDestroyed()) {
        // 回调 Activity 的 onContentChanged 方法通知 Activity 视图已经发生改变
        cb.onContentChanged();
    }
}
```

这个时候 DecorView 还没有被 WindowManager 正式添加。在 ActivityThread 的 handleResumeActivity 方法中，首先会调用 Activity 的 onResume 方法，接着调用 Activity 的 makeVisible()，完成 DecorView 的添加和显示过程：

Activity.java

```
void makeVisible() {
    if (!mWindowAdded) {
        ViewManager wm = getWindowManager();
        wm.addView(mDecor, getWindow().getAttributes());
        mWindowAdded = true;
    }
    mDecor.setVisibility(View.VISIBLE);
}
```

2.11.3.2 Dialog 的 Window 创建过程

Dialog 的 Window 的创建过程和 Activity 类似，创建同样是通过 PolicyManager 的 makeNewWindow 方法完成的，创建后的对象实际就是 PhoneWindow。当 Dialog 被关闭时，会通过 WindowManager 来移除 DecorView：mWindowManager.removeViewImmediate(mDecor)。

Dialog.java

```
Dialog(@NonNull Context context, @StyleRes int themeResId, boolean
createContextThemeWrapper) {
    ...
    mWindowManager = (WindowManager) context.getSystemService(Context.WINDOW_SERVICE);
```

```
    final Window w = new PhoneWindow(mContext);
    mWindow = w;
    w.setCallback(this);
    w.setOnWindowDismissedCallback(this);
    w.setOnWindowSwipeDismissedCallback(() -> {
        if (mCancelable) {
            cancel();
        }
    });
    w.setWindowManager(mWindowManager, null, null);
    w.setGravity(Gravity.CENTER);

    mListenersHandler = new ListenersHandler(this);
}
```

普通 Dialog 必须采用 Activity 的 Context，采用 Application 的 Context 就会报错，是因为应用 token 所导致，应用 token 一般只有 Activity 拥有。系统 Window 比较特殊，不需要 token。

2.11.3.3 Toast 的 Window 创建过程

Toast 属于系统 Window ，由于其具有定时取消功能，所以系统采用了 Handler。Toast 的内部有两类 IPC 过程，第一类是 Toast 访问 NotificationManagerService，第二类是 NotificationManagerService 回调 Toast 里的 TN 接口。

Toast 内部的视图由两种方式，一种是系统默认的样式，另一种是 setView 指定一个自定义 View，它们都对应 Toast 的一个内部成员 mNextView。

Toast.java

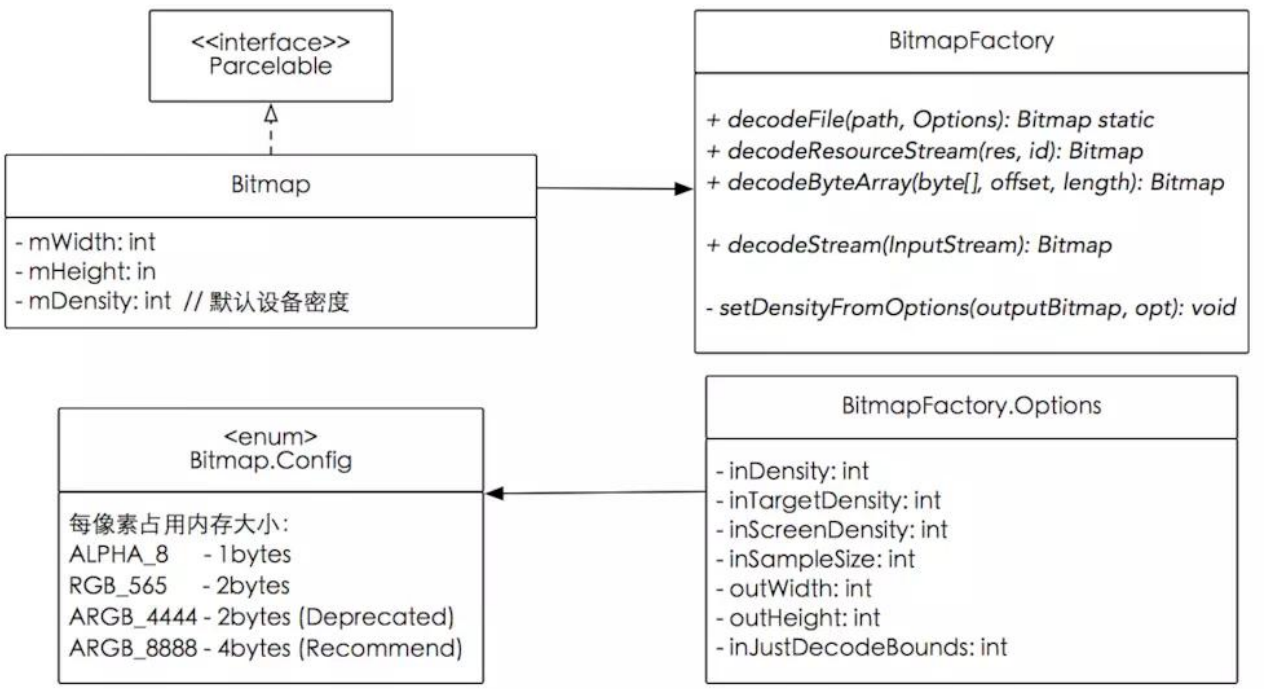
```
public void show() {
    if (mNextView == null) {
        throw new RuntimeException("setView must have been called");
    }

    INotificationManager service = getService();
    String pkg = mContext.getOpPackageName();
    TN tn = mTN;
    tn.mNextView = mNextView;

    try {
        service.enqueueToast(pkg, tn, mDuration);
    } catch (RemoteException e) {
        // Empty
    }
}
...
public void cancel() {
    mTN.cancel();
}
```

NotificationManagerService.java

```
private void showNextToastLocked() {
    ToastRecord record = mToastQueue.get(0);
    while (record != null) {
        if (DBG) Slog.d(TAG, "Show pkg=" + record.pkg + " callback=" + record.callback);
        try {
            record.callback.show();
            scheduleTimeoutLocked(record, false);
            return;
        } catch (RemoteException e) {
            Slog.w(TAG, "Object died trying to show notification " + record.callback
                    + " in package " + record.pkg);
            // remove it from the list and let the process die
            int index = mToastQueue.indexOf(record);
            if (index >= 0) {
                mToastQueue.remove(index);
            }
            keepProcessAliveLocked(record.pid);
            if (mToastQueue.size() > 0) {
                record = mToastQueue.get(0);
            } else {
                record = null;
            }
        }
    }
}
```

```
···private void scheduleTimeoutLocked(ToastRecord r, boolean immediate)
{
    Message m = Message.obtain(mHandler, MESSAGE_TIMEOUT, r);
    long delay = immediate ? 0 : (r.duration == Toast.LENGTH_LONG ? LONG_DELAY : SHORT_DELAY);
    mHandler.removeCallbacksAndMessages(r);
    mHandler.sendMessageDelayed(m, delay);
}
```

2.12 Bitmap



2.12.1 配置信息与压缩方式

Bitmap 中有两个内部枚举类：

- Config 是用来设置颜色配置信息

- CompressFormat 是用来设置压缩方式

Config	单位像素所占字节数	解析
Bitmap.Config.ALPHA_8	1	颜色信息只由透明度组成，占 8 位
Bitmap.Config.ARGB_4444	2	颜色信息由 rgba 四部分组成，每个部分都占 4 位，总共占 16 位
Bitmap.Config.ARGB_8888	4	颜色信息由 rgba 四部分组成，每个部分都占 8 位，总共占 32 位。是 Bitmap 默认的颜色配置信息，也是最占空间的一种配置
Bitmap.Config.RGB_565	2	颜色信息由 rgb 三部分组成，R 占 5 位，G 占 6 位，B 占 5 位，总共占 16 位
RGBA_F16	8	Android 8.0 新增（更丰富的色彩表现 HDR）
HARDWARE	Special	Android 8.0 新增（Bitmap 直接存储在 graphic memory）

通常我们优化 Bitmap 时，当需要做性能优化或者防止 OOM，我们通常会使用 Bitmap.Config.RGB_565 这个配置，因为 Bitmap.Config.ALPHA_8 只有透明度，显示一般图片没有意义，Bitmap.Config.ARGB_4444 显示图片不清楚，Bitmap.Config.ARGB_8888 占用内存最多。

CompressFormat	解析

CompressFormat	解析
Bitmap.CompressFormat.JPEG	表示以 JPEG 压缩算法进行图像压缩，压缩后的格式可以是 .jpg 或者 .jpeg，是一种有损压缩
Bitmap.CompressFormat.PNG	颜色信息由 rgba 四部分组成，每个部分都占 4 位，总共占 16 位
Bitmap.Config.ARGB_8888	颜色信息由 rgba 四部分组成，每个部分都占 8 位，总共占 32 位。是 Bitmap 默认的颜色配置信息，也是最占空间的一种配置
Bitmap.Config.RGB_565	颜色信息由 rgb 三部分组成，R 占 5 位，G 占 6 位，B 占 5 位，总共占 16 位

2.12.2 常用操作

2.12.2.1 裁剪、缩放、旋转、移动

```
Matrix matrix = new Matrix();  // 缩放

matrix.postScale(0.8f, 0.9f);  // 左旋，参数为正则向右旋

matrix.postRotate(-45);  // 平移，在上一次修改的基础上进行再次修改 set 每次操作都是最新的 会覆盖上次的操作

matrix.postTranslate(100, 80);// 裁剪并执行以上操作 Bitmap bitmap =
Bitmap.createBitmap(source, 0, 0, source.getWidth(), source.getHeight(), matrix, true);
```

> 虽然 Matrix 还可以调用 postSkew 方法进行倾斜操作，但是却不可以在此时创建 Bitmap 时使用。

2.12.2.2 Bitmap 与 Drawable 转换

```
// Drawable -> Bitmappublic static Bitmap drawableToBitmap(Drawable drawable) {

    Bitmap bitmap = Bitmap.createBitmap(drawable.getIntrinsicWidth(),
drawable.getIntrinsicHeight(), drawable.getOpacity() != PixelFormat.OPAQUE ?
Bitmap.Config.ARGB_8888 : Bitmap.Config.RGB_565);

    Canvas canvas = new Canvas(bitmap);

    drawable.setBounds(0, 0, drawable.getIntrinsicWidth(), drawable.getIntrinsicHeight());

    drawable.draw(canvas);

    return bitmap;

}

// Bitmap -> Drawablepublic static Drawable bitmapToDrawable(Resources resources, Bitmap bm)
{

    Drawable drawable = new BitmapDrawable(resources, bm);

    return drawable;

}
```

2.12.2.3 保存与释放

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.test);File file =
new File(getFilesDir(),"test.jpg");if(file.exists()){

    file.delete();

}try {

    FileOutputStream outputStream=new FileOutputStream(file);

    bitmap.compress(Bitmap.CompressFormat.JPEG,90,outputStream);

    outputStream.flush();

    outputStream.close();

} catch (FileNotFoundException e) {
```

```
    e.printStackTrace();
} catch (IOException e) {
    e.printStackTrace();
}//释放 bitmap 的资源，这是一个不可逆转的操作
bitmap.recycle();
```

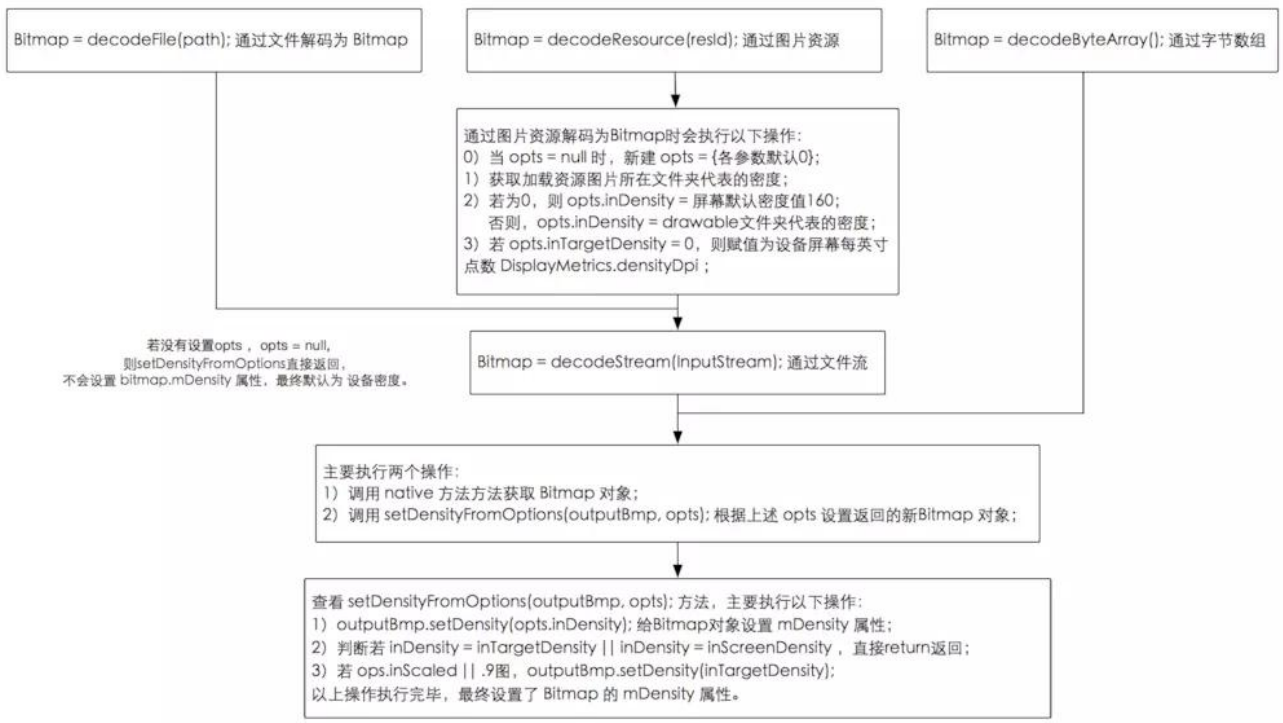
2.12.2.4 图片压缩

```
public static Bitmap compressImage(Bitmap image) {
    if (image == null) {
        return null;
    }
    ByteArrayOutputStream baos = null;
    try {
        baos = new ByteArrayOutputStream();
        image.compress(Bitmap.CompressFormat.JPEG, 100, baos);
        byte[] bytes = baos.toByteArray();
        ByteArrayInputStream isBm = new ByteArrayInputStream(bytes);
        Bitmap bitmap = BitmapFactory.decodeStream(isBm);
        return bitmap;
    } catch (OutOfMemoryError e) {
        e.printStackTrace();
    } finally {
        try {
            if (baos != null) {
                baos.close();
```

```
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    return null;
}
```

2.12.3 BitmapFactory

2.12.3.1 Bitmap 创建流程



2.12.3.2 Option 类

常用方法	说明

常用方法	说明
boolean inJustDecodeBounds	如果设置为 true，不获取图片，不分配内存，但会返回图片的高度宽度信息
int inSampleSize	图片缩放的倍数
int outWidth	获取图片的宽度值
int outHeight	获取图片的高度值
int inDensity	用于位图的像素压缩比
int inTargetDensity	用于目标位图的像素压缩比（要生成的位图）
byte[] inTempStorage	创建临时文件，将图片存储
boolean inScaled	设置为 true 时进行图片压缩，从 inDensity 到 inTargetDensity
boolean inDither	如果为 true,解码器尝试抖动解码
Bitmap.Config inPreferredConfig	设置解码器这个值是设置色彩模式，默认值是 ARGB_8888，在这个模式下，一个像素点占用 4bytes 空间，一般对透明度不做要求的话，一般采用 RGB_565 模式，这个模式下一个像素点占用 2bytes
String outMimeType	设置解码图像
boolean inPurgeable	当存储 Pixel 的内存空间在系统内存不足时是否可以被回收
boolean inInputShareable	inPurgeable 为 true 情况下才生效，是否可以共享一个

常用方法	说明
	InputStream
boolean inPreferQualityOverSpeed	为 true 则优先保证 Bitmap 质量其次是解码速度
boolean inMutable	配置 Bitmap 是否可以更改，比如：在 Bitmap 上隔几个像素加一条线段
int inScreenDensity	当前屏幕的像素密度

2.12.3.3 基本使用

```
try {
    FileInputStream fis = new FileInputStream(filePath);
    BitmapFactory.Options options = new BitmapFactory.Options();
    options.inJustDecodeBounds = true;
    // 设置 inJustDecodeBounds 为 true 后，再使用 decodeFile()等方法，并不会真正的分配空间，即解码出来的Bitmap为null,但是可计算出原始图片的宽度和高度,即options.outWidth和options.outHeight
    BitmapFactory.decodeFileDescriptor(fis.getFD(), null, options);
    float srcWidth = options.outWidth;
    float srcHeight = options.outHeight;
    int inSampleSize = 1;

    if (srcHeight > height || srcWidth > width) {
        if (srcWidth > srcHeight) {
            inSampleSize = Math.round(srcHeight / height);
        } else {
```

```
            inSampleSize = Math.round(srcWidth / width);
        }
    }

    options.inJustDecodeBounds = false;
    options.inSampleSize = inSampleSize;

    return BitmapFactory.decodeFileDescriptor(fis.getFD(), null, options);
} catch (Exception e) {
    e.printStackTrace();
}
```

2.12.4 内存回收

```
if(bitmap != null && !bitmap.isRecycled()){
    // 回收并且置为 null
    bitmap.recycle();
    bitmap = null;
}
```

Bitmap 类的构造方法都是私有的，所以开发者不能直接 new 出一个 Bitmap 对象，只能通过 BitmapFactory 类的各种静态方法来实例化一个 Bitmap。仔细查看 BitmapFactory 的源代码可以看到，生成 Bitmap 对象最终都是通过 JNI 调用方式实现的。所以，加载 Bitmap 到内存里以后，是包含两部分内存区域的。简单的说，一部分是 Java 部分的，一部分是 C 部分的。这个 Bitmap 对象是由 Java 部分分配的，不用的时候系统就会自动回收了，但是那个对应的 C 可用的内存区域，虚拟机是不能直接回收的，这个只能调用底层的功能释放。所

以需要调用 recycle() 方法来释放 C 部分的内存。从 Bitmap 类的源代码也可以看到，recycle() 方法里也的确是调用了 JNI 方法了的。

2.13 屏幕适配

2.13.1 单位

- dpi 每英寸像素数(dot per inch)
- dp

  密度无关像素 - 一种基于屏幕物理密度的抽象单元。 这些单位相对于 160 dpi 的屏幕，因此一个 dp 是 160 dpi 屏幕上的一个 px。 dp 与像素的比率将随着屏幕密度而变化，但不一定成正比。为不同设备的 UI 元素的实际大小提供了一致性。
- sp

  与比例无关的像素 - 这与 dp 单位类似，但它也可以通过用户的字体大小首选项进行缩放。建议在指定字体大小时使用此单位，以便根据屏幕密度和用户偏好调整它们。

```
dpi = px / inch


density = dpi / 160


dp = px / density
```

2.13.2 头条适配方案

```
private static void setCustomDensity(@NonNull Activity activity, @NonNull final Application
application) {
```

```
    final DisplayMetrics appDisplayMetrics =
application.getResources().getDisplayMetrics();

    if (sNoncompatDensity == 0) {

        sNoncompatDensity = appDisplayMetrics.density;

        sNoncompatScaledDensity = appDisplayMetrics.scaledDensity;

        // 监听字体切换

        application.registerComponentCallbacks(new ComponentCallbacks() {

            @Override

            public void onConfigurationChanged(Configuration newConfig) {

                if (newConfig != null && newConfig.fontScale > 0) {

                    sNoncompatScaledDensity =
application.getResources().getDisplayMetrics().scaledDensity;

                }

            }



            @Override

            public void onLowMemory() {



            }

        });

    }



    // 适配后的 dpi 将统一为 360dpi

    final float targetDensity = appDisplayMetrics.widthPixels / 360;

    final float targetScaledDensity = targetDensity * (sNoncompatScaledDensity /
sNoncompatDensity);
```

```
    final int targetDensityDpi = (int)(160 * targetDensity);


    appDisplayMetrics.density = targetDensity;
    appDisplayMetrics.scaledDensity = targetScaledDensity;
    appDisplayMetrics.densityDpi = targetDensityDpi;


    final DisplayMetrics activityDisplayMetrics =
activity.getResources().getDisplayMetrics();
    activityDisplayMetrics.density = targetDensity;
    activityDisplayMetrics.scaledDensity = targetScaledDensity;
    activityDisplayMetrics.densityDpi = targetDensityDpi
}
```

2.13.3 刘海屏适配

- Android P 刘海屏适配方案

Android P 支持最新的全面屏以及为摄像头和扬声器预留空间的凹口屏幕。通过全新的 DisplayCutout 类，可以确定非功能区域的位置和形状，这些区域不应显示内容。要确定这些凹口屏幕区域是否存在及其位置，使用 getDisplayCutout() 函数。

DisplayCutout 类方法	说明
getBoundingRects()	返回 Rects 的列表，每个 Rects 都是显示屏上非功能区域的边界矩形
getSafeInsetLeft ()	返回安全区域距离屏幕左边的距离，单位是 px

DisplayCutout 类方法	说明
getSafeInsetRight ()	返回安全区域距离屏幕右边的距离，单位是 px
getSafeInsetTop ()	返回安全区域距离屏幕顶部的距离，单位是 px
getSafeInsetBottom()	返回安全区域距离屏幕底部的距离，单位是 px

Android P 中 WindowManager.LayoutParams 新增了一个布局参数属性

layoutInDisplayCutoutMode：

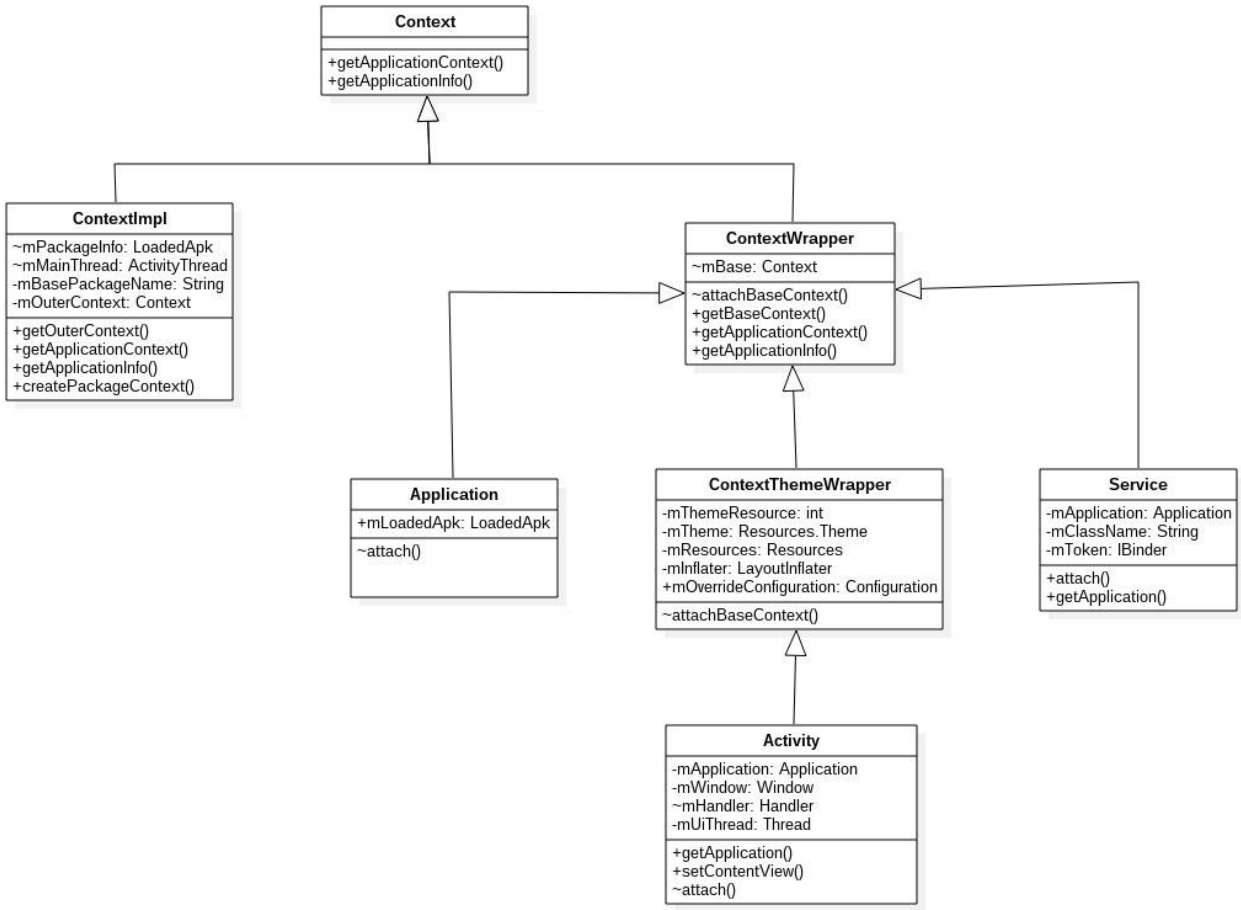
模式	模式说明
LAYOUT_IN_DISPLAY_CUTOUT_MODE_DEFAULT	只有当 DisplayCutout 完全包含在系统栏中时，才允许窗口延伸到 DisplayCutout 区域。 否则，窗口布局不与 DisplayCutout 区域重叠。
LAYOUT_IN_DISPLAY_CUTOUT_MODE_NEVER	该窗口决不允许与 DisplayCutout 区域重叠。
LAYOUT_IN_DISPLAY_CUTOUT_MODE_SHORT_EDGES	该窗口始终允许延伸到屏幕短边上的 DisplayCutout 区域。

- Android P 之前的刘海屏适配

不同厂商的刘海屏适配方案不尽相同，需分别查阅各自的开发者文档。

2.14 Context

Context 本身是一个抽象类，是对一系列系统服务接口的封装，包括：内部资源、包、类加载、I/O 操作、权限、主线程、IPC 和组件启动等操作的管理。ContextImpl, Activity, Service, Application 这些都是 Context 的直接或间接子类, 关系如下:



ContextWrapper 是代理 Context 的实现，简单地将其所有调用委托给另一个 Context（mBase）。

Application、Activity、Service 通过 `attach()` 调用父类 ContextWrapper 的 `attachBaseContext()`, 从而设置父类成员变量 mBase 为 ContextImpl 对象, ContextWrapper 的核心工作都是交给 mBase(ContextImpl) 来完成，这样可以子类化 Context 以修改行为而无需更改原始 Context。

2.15 SharedPreferences

SharedPreferences 采用 key-value（键值对）形式，主要用于轻量级的数据存储，尤其适合保存应用的配置参数，但不建议使用 SharedPreferences 来存储大规模的数据，可能会降低性能.

SharedPreferences 采用 xml 文件格式来保存数据，该文件所在目录位于 `/data/data/<package name>/shared_prefs`，如：

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
   <string name="blog">https://github.com/JasonWu1111/Android-Review</string>
</map>
```

从 Android N 开始，创建的 SP 文件模式，不允

许 `MODE_WORLD_READABLE` 和 `MODE_WORLD_WRITEABLE` 模块，否则会直接抛出异常

SecurityException。`MODE_MULTI_PROCESS` 这种多进程的方式也是 Google 不推荐的方式，后续同样会不再支持。

当设置 MODE_MULTI_PROCESS 模式，则每次 getSharedPreferences 过程，会检查 SP 文件上次修改时间和文件大小，一旦所有修改则会重新从磁盘加载文件。

2.15.1 获取方式

2.15.1.1 getPreferences

Activity.getPreferences(mode): 以当前 Activity 的类名作为 SP 的文件名. 即

xxxActivity.xml `Activity.java`

```
public SharedPreferences getPreferences(int mode) {
    return getSharedPreferences(getLocalClassName(), mode);
}
```

2.15.1.2 getDefaultSharedPreferences

PreferenceManager.getDefaultSharedPreferences(Context): 以包名加上 _preferences 作为文件名，以 MODE_PRIVATE 模式创建 SP 文件. 即 packgeName_preferences.xml.

```
public static SharedPreferences getDefaultSharedPreferences(Context context) {
    return context.getSharedPreferences(getDefaultSharedPreferencesName(context),
            getDefaultSharedPreferencesMode());
}
```

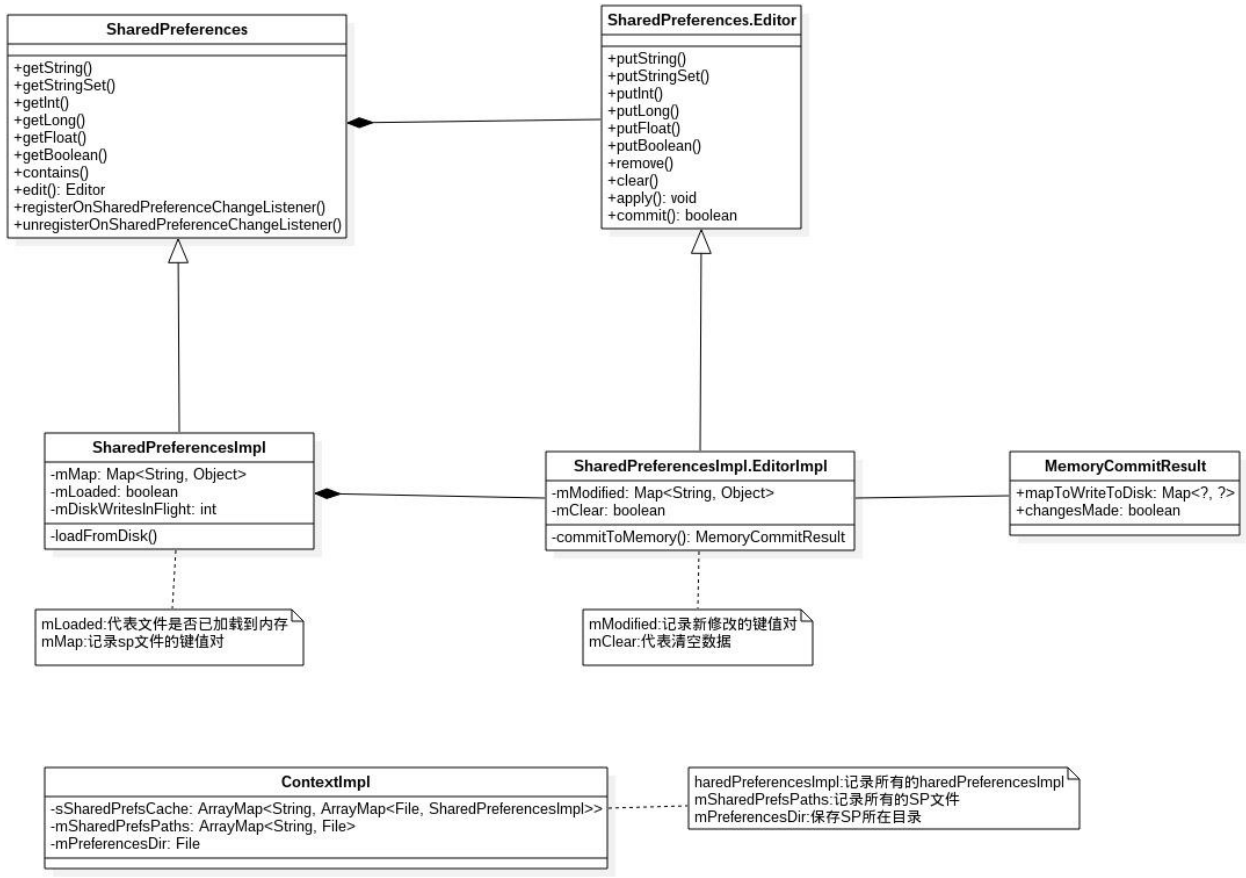
2.15.1.3 getSharedPreferences

直接调用 Context.getSharedPreferences(name, mode)，所有的方法最终都是调用到如下方法：

```
class ContextImpl extends Context {
    private ArrayMap<String, File> mSharedPrefsPaths;

    public SharedPreferences getSharedPreferences(String name, int mode) {
        File file;
        synchronized (ContextImpl.class) {
            if (mSharedPrefsPaths == null) {
                mSharedPrefsPaths = new ArrayMap<>();
            }
            //先从 mSharedPrefsPaths 查询是否存在相应文件
```

```
            file = mSharedPrefsPaths.get(name);
            if (file == null) {
                //如果文件不存在，则创建新的文件
                file = getSharedPreferencesPath(name);
                mSharedPrefsPaths.put(name, file);
            }
        }

        return getSharedPreferences(file, mode);
    }
}
```

2.15.2 架构



SharedPreferences 与 Editor 只是两个接口. SharedPreferencesImpl 和 EditorImpl 分别实现了对应接口。另外, ContextImpl 记录着 SharedPreferences 的重要数据。

`putxxx()` 操作把数据写入到 EditorImpl.mModified；

`apply()/commit()` 操作先调用 commitToMemory(), 将数据同步到 SharedPreferencesImpl 的 mMap, 并保存到 MemoryCommitResult 的 mapToWriteToDisk，再调用 enqueueDiskWrite(), 写入到磁盘文件; 先之前把原有数据保存到 .bak 为后缀的文件,用于在写磁盘的过程出现任何异常可恢复数据;

`getxxx()` 操作从 SharedPreferencesImpl.mMap 读取数据.

2.15.3 apply / commit

- apply 没有返回值, commit 有返回值能知道修改是否提交成功
- apply 是将修改提交到内存，再异步提交到磁盘文件，而 commit 是同步的提交到磁盘文件
- 多并发的提交 commit 时，需等待正在处理的 commit 数据更新到磁盘文件后才会继续往下执行，从而降低效率; 而 apply 只是原子更新到内存，后调用 apply 函数会直接覆盖前面内存数据，从一定程度上提高很多效率。

2.15.4 注意

- 强烈建议不要在 sp 里面存储特别大的 key/value，有助于减少卡顿 / anr
- 不要高频地使用 apply，尽可能地批量提交
- 不要使用 MODE_MULTI_PROCESS
- 高频写操作的 key 与高频读操作的 key 可以适当地拆分文件，由于减少同步锁竞争
- 不要连续多次 edit()，应该获取一次获取 edit()，然后多次执行 putxxx()，减少内存波动

2.16 消息机制

2.16.1 Handler 机制

Handler 有两个主要用途：（1）安排 Message 和 runnables 在将来的某个时刻执行; （2）将要在不同于自己的线程上执行的操作排入队列。(在多个线程并发更新 UI 的同时保证线程安全。)

Android 规定访问 UI 只能在主线程中进行，因为 Android 的 UI 控件不是线程安全的，多线程并发访问会导致 UI 控件处于不可预期的状态。为什么系统不对 UI 控件的访问加上锁机制？缺点有两个：加锁会让 UI 访问的逻辑变得复杂；其次锁机制会降低 UI 访问的效率。如果子线程访问 UI，那么程序就会抛出异常。ViewRootImpl 对 UI 操作做了验证，这个验证工作是由 ViewRootImpl 的 `checkThread` 方法完成：

ViewRootImpl.java

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

- Message：Handler 接收和处理的消息对象
- MessageQueue：Message 的队列，先进先出，每一个线程最多可以拥有一个
- Looper：消息泵，是 MessageQueue 的管理者，会不断从 MessageQueue 中取出消息，并将消息分给对应的 Handler 处理，每个线程只有一个 Looper。

Handler 创建的时候会采用当前线程的 Looper 来构造消息循环系统，需要注意的是，线程默认是没有 Looper 的，直接使用 Handler 会报错，如果需要使用 Handler 就必须为线程创建 Looper，因为默认的 UI 主线程，也就是 ActivityThread，ActivityThread 被创建的时候就会初始化 Looper，这也是在主线程中默认可以使用 Handler 的原因。

2.16.2 工作原理

2.16.2.1 ThreadLocal

ThreadLocal 是一个线程内部的数据存储类，通过它可以在指定的线程中存储数据，其他线程则无法获取。Looper、ActivityThread 以及 AMS 中都用到了 ThreadLocal。当不同线程访问同一个 ThreadLocal 的 get 方法，ThreadLocal 内部会从各自的线程中取出一个数组，然后再从数组中根据当前 ThreadLcoal 的索引去查找对应的 value 值。 ThreadLocal.java

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


···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();
}
```

2.16.2.2 MessageQueue

MessageQueue 主要包含两个操作：插入和读取。读取操作本身会伴随着删除操作，插入和读取对应的方法分别是 enqueueMessage 和 next。MessageQueue 内部实现并不是用的队列，实际上通过一个单链表的数据结构来维护消息列表。next 方法是一个无限循环的方法，如果消息队列中没有消息，那么 next 方法会一直阻塞。当有新消息到来时，next 方法会放回这条消息并将其从单链表中移除。

MessageQueue.java

```
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) {
            // New head, wake up the event queue if blocked.
            msg.next = p;
            mMessages = msg;
            needWake = mBlocked;
```

```
        } else {
            // Inserted within the middle of the queue.  Usually we don't have to wake
            // up the event queue unless there is a barrier at the head of the queue
            // and the message is the earliest asynchronous message in the queue.
            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; // invariant: p == prev.next
            prev.next = msg;
        }

        // We can assume mPtr != 0 because mQuitting is false.
        if (needWake) {
            nativeWake(mPtr);
        }
    }
```

```
    return true;
}
...Message next() {
    // Return here if the message loop has already quit and been disposed.
    // This can happen if the application tries to restart a looper after quit
    // which is not supported.
    ...
    for (;;) {
        ...
        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) {
                // Stalled by a barrier.  Find the next asynchronous message in the queue.
                do {
                    prevMsg = msg;
                    msg = msg.next;
                } while (msg != null && !msg.isAsynchronous());
            }
            if (msg != null) {
                if (now < msg.when) {
                    // Next message is not ready.  Set a timeout to wake up when it is ready.
```

```
                    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;
                    if (DEBUG) Log.v(TAG, "Returning message: " + msg);
                    msg.markInUse();
                    return msg;
                }
            } else {
                // No more messages.
                nextPollTimeoutMillis = -1;
            }
            ...
        }


        // Run the idle handlers.
        // We only ever reach this code block during the first iteration.
        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);
            }

            if (!keep) {
                synchronized (this) {
                    mIdleHandlers.remove(idler);
                }
            }
        }

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

        // While calling an idle handler, a new message could have been delivered
        // so go back and look again for a pending message without waiting.
        nextPollTimeoutMillis = 0;
    }
}
```

2.16.2.3 Looper

Looper 会不停地从 MessageQueue 中 查看是否有新消息，如果有新消息就会立刻处理，否则会一直阻塞。 Looper.java

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

可通过 Looper.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();
        Looper.loop();
    }
}.start();
```

除了 prepare 方法外，Looper 还提供了 prepareMainLooper 方法，主要是给 ActivityThread 创建 Looper 使用，本质也是通过 prepare 方法实现的。由于主线程的 Looper 比较特殊，所以 Looper 提供了一个 getMainLooper 方法来获取主线程的 Looper。

Looper 提供了 quit 和 quitSafely 来退出一个 Looper，二者的区别是：quit 会直接退出 Looper，而 quitSafly 只是设定一个退出标记，然后把消息队列中的已有消息处理完毕后才安

全地退出。Looper 退出后，通过 Handler 发送的消息会失败，这个时候 Handler 的 send 方法会返回 false。因此在不需要的时候应终止 Looper。

Looper.java

```
public static void loop() {
    final Looper me = myLooper();
    if (me == null) {
        throw new RuntimeException("No Looper; Looper.prepare() wasn't called on this thread.");
    }
    final MessageQueue queue = me.mQueue;
    ...
    for (;;) {
        Message msg = queue.next(); // might block
        if (msg == null) {
            // No message indicates that the message queue is quitting.
            return;
        }
        ...
        try {
            msg.target.dispatchMessage(msg);
            dispatchEnd = needEndTime ? SystemClock.uptimeMillis() : 0;
        } finally {
            if (traceTag != 0) {
                Trace.traceEnd(traceTag);
            }
```

```
        }
        ...
        msg.recycleUnchecked();
    }
}
```

loop 方法是一个死循环，唯一跳出循环的方式是 MessageQueue 的 next 方法返回了 null。当 Looper 的 quit 方法被调用时，Looper 就会调用 MessageQueue 的 quit 或者 qutiSafely 方法来通知消息队列退出，当消息队列被标记为退出状态时，它的 next 方法就会返回 null。loop 方法会调用 MessageQueue 的 next 方法来获取新消息，而 next 是一个阻塞操作，当没有消息时，next 会一直阻塞，导致 loop 方法一直阻塞。Looper 处理这条消息： msg.target.dispatchMessage(msg)，这里的 msg.target 是发送这条消息的 Handler 对象。

2.16.2.4 Handler

Handler 的工作主要包含消息的发送和接收的过程。消息的发送可以通过 post/send 的一系列方法实现，post 最终也是通过 send 来实现的。

2.17 线程异步

应用启动时，系统会为应用创建一个名为"主线程"的执行线程(UI 线程)。 此线程非常重要，因为它负责将事件分派给相应的用户界面小部件，其中包括绘图事件。 此外，它也是应用与 Android UI 工具包组件（来自 `android.widget` 和 `android.view` 软件包的组件）进行交互的线程。

系统不会为每个组件实例创建单独的线程。运行于同一进程的所有组件均在 UI 线程中实例化，并且对每个组件的系统调用均由该线程进行分派。 因此，响应系统回调的方法（例如，报告用户操作的 onKeyDown() 或生命周期回调方法）始终在进程的 UI 线程中运行。

Android 的单线程模式必须遵守两条规则:

- 不要阻塞 UI 线程
- 不要在 UI 线程之外访问 Android UI 工具包

为解决此问题，Android 提供了几种途径来从其他线程访问 UI 线程:

- `Activity.runOnUiThread(Runnable)`
- `View.post(Runnable)`
- `View.postDelayed(Runnable, long)`

2.17.1 AsyncTask

AsyncTask 封装了 Thread 和 Handler，并不适合特别耗时的后台任务，对于特别耗时的任务来说，建议使用线程池。

2.17.1.1 基本使用

方法	说明
onPreExecute()	异步任务执行前调用，用于做一些准备工作
doInBackground(Params...params)	用于执行异步任务，此方法中可以通过 publishProgress 方法来更新任务的进度， publishProgress 会调用 onProgressUpdate 方法
onProgressUpdate	在主线程中执行，后台任务的执行进度发生改变时调用

方法	说明
onPostExecute	在主线程中执行，在异步任务执行之后

```
import android.os.AsyncTask;

public class DownloadTask extends AsyncTask<String, Integer, Boolean> {


    @Override
    protected void onPreExecute() {
        super.onPreExecute();
    }


    @Override
    protected Boolean doInBackground(String... strings) {
        return null;
    }


    @Override
    protected void onProgressUpdate(Integer... values) {
        super.onProgressUpdate(values);
    }


    @Override
    protected void onPostExecute(Boolean aBoolean) {
        super.onPostExecute(aBoolean);
    }
}
```

- 异步任务的实例必须在 UI 线程中创建，即 AsyncTask 对象必须在 UI 线程中创建。
- execute(Params... params)方法必须在 UI 线程中调用。
- 不要手动调用 onPreExecute()，doInBackground()，onProgressUpdate()，onPostExecute() 这几个方法。
- 不能在 doInBackground() 中更改 UI 组件的信息。
- 一个任务实例只能执行一次，如果执行第二次将会抛出异常。
- execute() 方法会让同一个进程中的 AsyncTask 串行执行，如果需要并行，可以调用 executeOnExcutor 方法。

2.17.1.2 工作原理

AsyncTask.java

```
@MainThreadpublic final AsyncTask<Params, Progress, Result> execute(Params... params) {
    return executeOnExecutor(sDefaultExecutor, params);
}
@MainThreadpublic final AsyncTask<Params, Progress, Result> executeOnExecutor(Executor
exec,
        Params... params) {
    if (mStatus != Status.PENDING) {
        switch (mStatus) {
            case RUNNING:
                throw new IllegalStateException("Cannot execute task:"
                        + " the task is already running.");
            case FINISHED:
                throw new IllegalStateException("Cannot execute task:"
```

```
                    + " the task has already been executed "
                    + "(a task can be executed only once)");
        }
    }

    mStatus = Status.RUNNING;

    onPreExecute();

    mWorker.mParams = params;
    exec.execute(mFuture);

    return this;
}
```

sDefaultExecutor 是一个串行的线程池，一个进程中的所有的 AsyncTask 全部在该线程池中执行。AysncTask 中有两个线程池（SerialExecutor 和 THREAD_POOL_EXECUTOR）和一个 Handler（InternalHandler），其中线程池 SerialExecutor 用于任务的排队，THREAD_POOL_EXECUTOR 用于真正地执行任务，InternalHandler 用于将执行环境从线程池切换到主线程。

AsyncTask.java

```
private static Handler getMainHandler() {
    synchronized (AsyncTask.class) {
        if (sHandler == null) {
            sHandler = new InternalHandler(Looper.getMainLooper());
```

```
        }
        return sHandler;
    }
}
private static class InternalHandler extends Handler {
    public InternalHandler(Looper looper) {
        super(looper);
    }

    @SuppressWarnings({"unchecked", "RawUseOfParameterizedType"})
    @Override
    public void handleMessage(Message msg) {
        AsyncTaskResult<?> result = (AsyncTaskResult<?>) msg.obj;
        switch (msg.what) {
            case MESSAGE_POST_RESULT:
                // There is only one result
                result.mTask.finish(result.mData[0]);
                break;
            case MESSAGE_POST_PROGRESS:
                result.mTask.onProgressUpdate(result.mData);
                break;
        }
    }
}
```

```
private Result postResult(Result result) {
    @SuppressWarnings("unchecked")
    Message message = getHandler().obtainMessage(MESSAGE_POST_RESULT,
            new AsyncTaskResult<Result>(this, result));
    message.sendToTarget();
    return result;
}
```

2.17.2 HandlerThread

HandlerThread 集成了 Thread，却和普通的 Thread 有显著的不同。普通的 Thread 主要用于在 run 方法中执行一个耗时任务，而 HandlerThread 在内部创建了消息队列，外界需要通过 Handler 的消息方式通知 HanderThread 执行一个具体的任务。

HandlerThread.java

```
@Overridepublic void run() {
    mTid = Process.myTid();
    Looper.prepare();
    synchronized (this) {
        mLooper = Looper.myLooper();
        notifyAll();
    }
    Process.setThreadPriority(mPriority);
    onLooperPrepared();
    Looper.loop();
    mTid = -1;
```

```
}
```

2.17.3 IntentService

IntentService 可用于执行后台耗时的任务，当任务执行后会自动停止，由于其是 Service 的原因，它的优先级比单纯的线程要高，所以 IntentService 适合执行一些高优先级的后台任务。在实现上，IntentService 封装了 HandlerThread 和 Handler。

IntentService.java

```
@Overridepublic void onCreate() {
    // TODO: It would be nice to have an option to hold a partial wakelock
    // during processing, and to have a static startService(Context, Intent)
    // method that would launch the service & hand off a wakelock.

    super.onCreate();
    HandlerThread thread = new HandlerThread("IntentService[" + mName + "]");
    thread.start();

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

IntentService 第一次启动时，会在 onCreatea 方法中创建一个 HandlerThread，然后使用的 Looper 来构造一个 Handler 对象 mServiceHandler，这样通过 mServiceHandler 发送的消息最终都会在 HandlerThread 中执行。每次启动 IntentService，它的 onStartCommand 方

法就会调用一次，onStartCommand 中处理每个后台任务的 Intent，onStartCommand 调用了 onStart 方法：

IntentService.java

```
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);
    }
}

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

可以看出，IntentService 仅仅是通过 mServiceHandler 发送了一个消息，这个消息会在 HandlerThread 中被处理。mServiceHandler 收到消息后，会将 Intent 对象传递给

onHandlerIntent 方法中处理，执行结束后，通过 stopSelf(int startId) 来尝试停止服务。

（stopSelf() 会立即停止服务，而 stopSelf(int startId) 则会等待所有的消息都处理完毕后才终止服务）。

2.17.4 线程池

线程池的优点有以下：

- 重用线程池中的线程，避免因为线程的创建和销毁带来性能开销。
- 能有效控制线程池的最大并发数，避免大量的线程之间因互相抢占系统资源而导致的阻塞现象。
- 能够对线程进行管理，并提供定时执行以及定间隔循环执行等功能。

java 中，ThreadPoolExecutor 是线程池的真正实现：

ThreadPoolExecutor.java

```
/**     * Creates a new {@code ThreadPoolExecutor} with the given initial     * parameters.
*     * @param corePoolSize 核心线程数     * @param maximumPoolSize 最大线程数     * @param
keepAliveTime 非核心线程闲置的超时时长     * @param unit 用于指定 keepAliveTime 参数的时间单位
* @param 任务队列，通过线程池的 execute 方法提交的 Runnable 对象会存储在这个参数中     * @param
threadFactory 线程工厂，用于创建新线程     * @param handler 任务队列已满或者是无法成功执行任务时
调用     */public ThreadPoolExecutor(int corePoolSize,
                           int maximumPoolSize,
                           long keepAliveTime,
                           TimeUnit unit,
                           BlockingQueue<Runnable> workQueue,
                           ThreadFactory threadFactory,
                           RejectedExecutionHandler handler) {
    ...
}
```

类型	创建方法	说明

类型	创建方法	说明
FixedThreadPool	Executors.newFixedThreadPool(int nThreads)	一种线程数量固定的线程池，只有核心线程并且不会被回收，没有超时机制
CachedThreadPool	Executors.newCachedThreadPool()	一种线程数量不定的线程池，只有非核心线程，当线程都处于活动状态时，会创建新线程来处理新任务，否则会利用空闲的线程，超时时长为 60s
ScheduledThreadPool	Executors.newScheduledThreadPool(int corePoolSize)	核心线程数是固定的，非核心线程数没有限制，非核心线程闲置时立刻回收，主要用于执行定时任务和固定周期的重复任务
SingleThreadExecutor	Executors.newSingleThreadExecutor()	只有一个核心线程，确保所有任务在同一线程中按顺序执行

2.18 RecyclerView 优化

- 数据处理和视图加载分离：数据的处理逻辑尽可能放在异步处理，onBindViewHolder 方法中只处理数据填充到视图中。
- 数据优化：分页拉取远端数据，对拉取下来的远端数据进行缓存，提升二次加载速度；对于新增或者删除数据通过 DiffUtil 来进行局部刷新数据，而不是一味地全局刷新数据。

示例

```
public class AdapterDiffCallback extends DiffUtil.Callback {

```

```
    private List<String> mOldList;

    private List<String> mNewList;


    public AdapterDiffCallback(List<String> oldList, List<String> newList) {

        mOldList = oldList;

        mNewList = newList;

        DiffUtil.DiffResult

    }


    @Override

    public int getOldListSize() {

        return mOldList.size();

    }


    @Override

    public int getNewListSize() {

        return mNewList.size();

    }


    @Override

    public boolean areItemsTheSame(int oldItemPosition, int newItemPosition) {

        return
mOldList.get(oldItemPosition).getClass().equals(mNewList.get(newItemPosition).getClass())
;

    }
```

```
    @Override
    public boolean areContentsTheSame(int oldItemPosition, int newItemPosition) {
        return mOldList.get(oldItemPosition).equals(mNewList.get(newItemPosition));
    }
}
DiffUtil.DiffResult diffResult = DiffUtil.calculateDiff(new AdapterDiffCallback(oldList, newList));
diffResult.dispatchUpdatesTo(mAdapter);
```

- 布局优化：减少布局层级，简化 ItemView
- 升级 RecycleView 版本到 25.1.0 及以上使用 Prefetch 功能
- 通过重写 RecyclerView.onViewRecycled(holder) 来回收资源
- 如果 Item 高度是固定的话，可以使用 RecyclerView.setHasFixedSize(true); 来避免 requestLayout 浪费资源
- 对 ItemView 设置监听器，不要对每个 Item 都调用 addXxListener，应该大家公用一个 XxListener，根据 ID 来进行不同的操作，优化了对象的频繁创建带来的资源消耗
- 如果多个 RecycledView 的 Adapter 是一样的，比如嵌套的 RecyclerView 中存在一样的 Adapter，可以通过设置 RecyclerView.setRecycledViewPool(pool)，来共用一个 RecycledViewPool。

2.19 Webview

2.19.1 基本使用

2.19.1.1 WebView

```
// 获取当前页面的 URLpublic String getUrl();// 获取当前页面的原始 URL(重定向后可能当前 url 不同)// 就是 http headers 的 Referer 参数，loadUrl 时为 nullpublic String getOriginalUrl();// 获取当前页面的标题 public String getTitle();// 获取当前页面的 faviconpublic Bitmap getFavicon();// 获取当前页面的加载进度 public int getProgress();

// 通知 WebView 内核网络状态// 用于设置 JS 属性`window.navigator.isOnline`和产生 HTML5 事件`online/offline`public void setNetworkAvailable(boolean networkUp)

// 设置初始缩放比例 public void setInitialScale(int scaleInPercent);
```

2.19.1.2 WebSettings

```
WebSettings settings = web.getSettings();

// 存储(storage)// 启用 HTML5 DOM storage API，默认值 false

settings.setDomStorageEnabled(true); // 启用 Web SQL Database API，这个设置会影响同一进程内的所有 WebView，默认值 false// 此 API 已不推荐使用，参考：https://www.w3.org/TR/webdatabase/

settings.setDatabaseEnabled(true);  // 启用 Application Caches API，必需设置有效的缓存路径才能生效，默认值 false// 此 API 已废弃，参考：https://developer.mozilla.org/zh-CN/docs/Web/HTML/Using_the_application_cache

settings.setAppCacheEnabled(true);

settings.setAppCachePath(context.getCacheDir().getAbsolutePath());

// 定位(location)

settings.setGeolocationEnabled(true);

// 是否保存表单数据

settings.setSaveFormData(true);// 是否当 webview 调用 requestFocus 时为页面的某个元素设置焦点，默认值 true

settings.setNeedInitialFocus(true);

// 是否支持 viewport 属性，默认值 false// 页面通过`<meta name="viewport" ... />`自适应手机屏幕

settings.setUseWideViewPort(true);// 是否使用 overview mode 加载页面，默认值 false// 当页面宽度大于 WebView 宽度时，缩小使页面宽度等于 WebView 宽度

settings.setLoadWithOverviewMode(true);// 布局算法
```

```
settings.setLayoutAlgorithm(WebSettings.LayoutAlgorithm.NORMAL);

// 是否支持 Javascript，默认值 false

settings.setJavaScriptEnabled(true); // 是否支持多窗口，默认值 false

settings.setSupportMultipleWindows(false);// 是否可用 Javascript(window.open)打开窗口，默认值 false

settings.setJavaScriptCanOpenWindowsAutomatically(false);

// 资源访问

settings.setAllowContentAccess(true); // 是否可访问 Content Provider 的资源，默认值 true

settings.setAllowFileAccess(true);    // 是否可访问本地文件，默认值 true// 是否允许通过 file url 加载的 Javascript 读取本地文件，默认值 false

settings.setAllowFileAccessFromFileURLs(false);  // 是否允许通过 file url 加载的 Javascript 读取全部资源(包括文件,http,https)，默认值 false

settings.setAllowUniversalAccessFromFileURLs(false);

// 资源加载

settings.setLoadsImagesAutomatically(true); // 是否自动加载图片

settings.setBlockNetworkImage(false);       // 禁止加载网络图片

settings.setBlockNetworkLoads(false);       // 禁止加载所有网络资源

// 缩放(zoom)

settings.setSupportZoom(true);          // 是否支持缩放

settings.setBuiltInZoomControls(false); // 是否使用内置缩放机制

settings.setDisplayZoomControls(true);  // 是否显示内置缩放控件

// 默认文本编码，默认值 "UTF-8"

settings.setDefaultTextEncodingName("UTF-8");

settings.setDefaultFontSize(16);        // 默认文字尺寸，默认值 16，取值范围 1-72

settings.setDefaultFixedFontSize(16);   // 默认等宽字体尺寸，默认值 16

settings.setMinimumFontSize(8);         // 最小文字尺寸，默认值 8
```

```
settings.setMinimumLogicalFontSize(8);  // 最小文字逻辑尺寸，默认值 8
settings.setTextZoom(100);              // 文字缩放百分比，默认值 100
// 字体
settings.setStandardFontFamily("sans-serif");   // 标准字体，默认值 "sans-serif"
settings.setSerifFontFamily("serif");           // 衬线字体，默认值 "serif"
settings.setSansSerifFontFamily("sans-serif");  // 无衬线字体，默认值 "sans-serif"
settings.setFixedFontFamily("monospace");       // 等宽字体，默认值 "monospace"
settings.setCursiveFontFamily("cursive");       // 手写体(草书)，默认值 "cursive"
settings.setFantasyFontFamily("fantasy");       // 幻想体，默认值 "fantasy"

if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.KITKAT) {
    // 用户是否需要通过手势播放媒体(不会自动播放)，默认值 true
    settings.setMediaPlaybackRequiresUserGesture(true);
}if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.LOLLIPOP) {
    // 5.0 以上允许加载 http 和 https 混合的页面(5.0 以下默认允许，5.0+默认禁止)
    settings.setMixedContentMode(WebSettings.MIXED_CONTENT_ALWAYS_ALLOW);
}if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
    // 是否在离开屏幕时光栅化(会增加内存消耗)，默认值 false
    settings.setOffscreenPreRaster(false);
}
if (isNetworkConnected(context)) {
    // 根据 cache-control 决定是否从网络上取数据
    settings.setCacheMode(WebSettings.LOAD_DEFAULT);
} else {
    // 没网，离线加载，优先加载缓存(即使已经过期)
```

```
    settings.setCacheMode(WebSettings.LOAD_CACHE_ELSE_NETWORK);

}

// deprecated

settings.setRenderPriority(WebSettings.RenderPriority.HIGH);

settings.setDatabasePath(context.getDir("database", Context.MODE_PRIVATE).getPath());

settings.setGeolocationDatabasePath(context.getFilesDir().getPath());
```

2.19.1.3 WebViewClient

```
// 拦截页面加载，返回 true 表示宿主 app 拦截并处理了该 url，否则返回 false 由当前 WebView 处理// 此方法在 API24 被废弃，不处理 POST 请求 public boolean shouldOverrideUrlLoading(WebView view, String url) {

    return false;

}

// 拦截页面加载，返回 true 表示宿主 app 拦截并处理了该 url，否则返回 false 由当前 WebView 处理// 此方法添加于 API24，不处理 POST 请求，可拦截处理子 frame 的非 http 请求
@TargetApi(Build.VERSION_CODES.N)public boolean shouldOverrideUrlLoading(WebView view, WebResourceRequest request) {

    return shouldOverrideUrlLoading(view, request.getUrl().toString());

}

// 此方法废弃于 API21，调用于非 UI 线程// 拦截资源请求并返回响应数据，返回 null 时 WebView 将继续加载资源 public WebResourceResponse shouldInterceptRequest(WebView view, String url) {

    return null;

}

// 此方法添加于 API21，调用于非 UI 线程// 拦截资源请求并返回数据，返回 null 时 WebView 将继续加载资源@TargetApi(Build.VERSION_CODES.LOLLIPOP)public WebResourceResponse shouldInterceptRequest(WebView view, WebResourceRequest request) {

    return shouldInterceptRequest(view, request.getUrl().toString());

}

// 页面(url)开始加载 public void onPageStarted(WebView view, String url, Bitmap favicon) {
```

```
}

// 页面(url)完成加载 public void onPageFinished(WebView view, String url) {

}

// 将要加载资源(url)public void onLoadResource(WebView view, String url) {

}

// 这个回调添加于 API23，仅用于主框架的导航// 通知应用导航到之前页面时，其遗留的 WebView 内容将不再被绘制。// 这个回调可以用来决定哪些 WebView 可见内容能被安全地回收，以确保不显示陈旧的内容// 它最早被调用，以此保证 WebView.onDraw 不会绘制任何之前页面的内容，随后绘制背景色或需要加载的新内容。// 当 HTTP 响应 body 已经开始加载并体现在 DOM 上将在随后的绘制中可见时，这个方法会被调用。// 这个回调发生在文档加载的早期，因此它的资源(css,和图像)可能不可用。// 如果需要更细粒度的视图更新，查看 postVisualStateCallback(long, WebView.VisualStateCallback).// 请注意这上边的所有条件也支持 postVisualStateCallback(long ,WebView.VisualStateCallback)public void onPageCommitVisible(WebView view, String url) {

}

// 此方法废弃于 API23// 主框架加载资源时出错 public void onReceivedError(WebView view, int errorCode, String description, String failingUrl) {

}

// 此方法添加于 API23// 加载资源时出错，通常意味着连接不到服务器// 由于所有资源加载错误都会调用此方法，所以此方法应尽量逻辑简单@TargetApi(Build.VERSION_CODES.M)public void onReceivedError(WebView view, WebResourceRequest request, WebResourceError error) {

    if (request.isForMainFrame()) {

        onReceivedError(view, error.getErrorCode(), error.getDescription().toString(), request.getUrl().toString());

    }

}

// 此方法添加于 API23// 在加载资源(iframe,image,js,css,ajax...)时收到了 HTTP 错误(状态码>=400)public void onReceivedHttpError(WebView view, WebResourceRequest request, WebResourceResponse errorResponse) {

}
```

```
// 是否重新提交表单,默认不重发 public void onFormResubmission(WebView view, Message dontResend, Message resend) {

    dontResend.sendToTarget();

}

// 通知应用可以将当前的 url 存储在数据库中，意味着当前的访问 url 已经生效并被记录在内核当中。// 此方法在网页加载过程中只会被调用一次，网页前进后退并不会回调这个函数。public void doUpdateVisitedHistory(WebView view, String url, boolean isReload) {

}

// 加载资源时发生了一个 SSL 错误，应用必需响应(继续请求或取消请求)// 处理决策可能被缓存用于后续的请求，默认行为是取消请求 public void onReceivedSslError(WebView view, SslErrorHandler handler, SslError error) {

    handler.cancel();

}

// 此方法添加于 API21，在 UI 线程被调用// 处理 SSL 客户端证书请求，必要的话可显示一个 UI 来提供 KEY。// 有三种响应方式：proceed()/cancel()/ignore()，默认行为是取消请求// 如果调用 proceed()或 cancel()，Webview 将在内存中保存响应结果且对相同的"host:port"不会再次调用 onReceivedClientCertRequest// 多数情况下，可通过 KeyChain.choosePrivateKeyAlias 启动一个 Activity 供用户选择合适的私钥@TargetApi(Build.VERSION_CODES.LOLLIPOP)public void onReceivedClientCertRequest(WebView view, ClientCertRequest request) {

    request.cancel();

}

// 处理 HTTP 认证请求，默认行为是取消请求 public void onReceivedHttpAuthRequest(WebView view, HttpAuthHandler handler, String host, String realm) {

    handler.cancel();

}

// 通知应用有个已授权账号自动登陆了 public void onReceivedLoginRequest(WebView view, String realm, String account, String args) {

}// 给应用一个机会处理按键事件// 如果返回 true，WebView 不处理该事件，否则 WebView 会一直处理，默认返回 falsepublic boolean shouldOverrideKeyEvent(WebView view, KeyEvent event) {

    return false;

}
```

```
// 处理未被 WebView 消费的按键事件// WebView 总是消费按键事件，除非是系统按键或
shouldOverrideKeyEvent 返回 true// 此方法在按键事件分派时被异步调用 public void
onUnhandledKeyEvent(WebView view, KeyEvent event) {

    super.onUnhandledKeyEvent(view, event);

}

// 通知应用页面缩放系数变化 public void onScaleChanged(WebView view, float oldScale, float
newScale) {

}
```

2.19.1.4 WebChromeClient

```
// 获得所有访问历史项目的列表，用于链接着色。public void
getVisitedHistory(ValueCallback<String[]> callback) {

}

// <video /> 控件在未播放时，会展示为一张海报图，HTML 中可通过它的'poster'属性来指定。// 如果未
指定'poster'属性，则通过此方法提供一个默认的海报图。public Bitmap getDefaultVideoPoster() {

    return null;

}

// 当全屏的视频正在缓冲时，此方法返回一个占位视图(比如旋转的菊花)。public View
getVideoLoadingProgressView() {

    return null;

}

// 接收当前页面的加载进度 public void onProgressChanged(WebView view, int newProgress) {

}

// 接收文档标题 public void onReceivedTitle(WebView view, String title) {

}

// 接收图标(favicon)public void onReceivedIcon(WebView view, Bitmap icon) {

}
```

```
// Android 中处理 Touch Icon 的方案//
http://droidyue.com/blog/2015/01/18/deal-with-touch-icon-in-android/index.htmlpublic
void onReceivedTouchIconUrl(WebView view, String url, boolean precomposed) {

}

// 通知应用当前页进入了全屏模式，此时应用必须显示一个包含网页内容的自定义 Viewpublic void
onShowCustomView(View view, CustomViewCallback callback) {

}

// 通知应用当前页退出了全屏模式，此时应用必须隐藏之前显示的自定义 Viewpublic void
onHideCustomView() {

}



// 显示一个 alert 对话框 public boolean onJsAlert(WebView view, String url, String message,
JsResult result) {

    return false;

}

// 显示一个 confirm 对话框 public boolean onJsConfirm(WebView view, String url, String message,
JsResult result) {

    return false;

}

// 显示一个 prompt 对话框 public boolean onJsPrompt(WebView view, String url, String message,
String defaultValue, JsPromptResult result) {

    return false;

}

// 显示一个对话框让用户选择是否离开当前页面 public boolean onJsBeforeUnload(WebView view,
String url, String message, JsResult result) {

    return false;

}
```

```
// 指定源的网页内容在没有设置权限状态下尝试使用地理位置 API。// 从 API24 开始，此方法只为安全的源(https)调用，非安全的源会被自动拒绝 public void onGeolocationPermissionsShowPrompt(String origin, GeolocationPermissions.Callback callback) {

}

// 当前一个调用 onGeolocationPermissionsShowPrompt() 取消时，隐藏相关的 UI。public void onGeolocationPermissionsHidePrompt() {

}

// 通知应用打开新窗口 public boolean onCreateWindow(WebView view, boolean isDialog, boolean isUserGesture, Message resultMsg) {

    return false;

}

// 通知应用关闭窗口 public void onCloseWindow(WebView window) {

}

// 请求获取取焦点 public void onRequestFocus(WebView view) {

}

// 通知应用网页内容申请访问指定资源的权限(该权限未被授权或拒绝)@TargetApi(Build.VERSION_CODES.LOLLIPOP)public void onPermissionRequest(PermissionRequest request) {

    request.deny();

}

// 通知应用权限的申请被取消，隐藏相关的 UI。@TargetApi(Build.VERSION_CODES.LOLLIPOP)public void onPermissionRequestCanceled(PermissionRequest request) {

}

// 为'<input type="file" />'显示文件选择器，返回 false 使用默认处理 @TargetApi(Build.VERSION_CODES.LOLLIPOP)public boolean onShowFileChooser(WebView webView, ValueCallback<Uri[]> filePathCallback, FileChooserParams fileChooserParams) {

    return false;

}
```

```
// 接收 JavaScript 控制台消息 public boolean onConsoleMessage(ConsoleMessage consoleMessage)
{
    return false;
}
```

2.19.2 Webview 加载优化

- 使用本地资源替代

可以 将一些资源文件放在本地的 asset s 目录, 然后重 写 WebViewClient

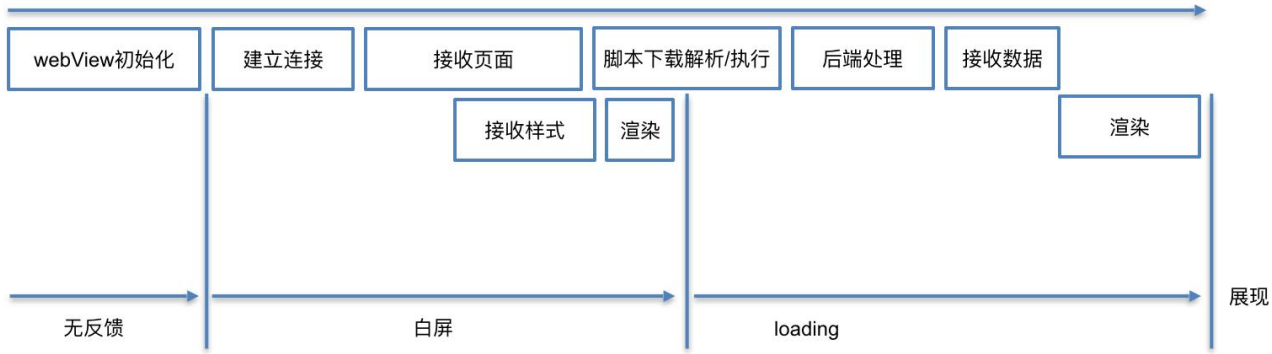
的 shouldInterceptRequest 方法，对访问地址进行拦截，当 url 地址命中本地配置的 url 时，使用本地资源替代，否则就使用网络上的资源。

```
mWebview.setWebViewClient(new WebViewClient() {
     // 设置不用系统浏览器打开,
    @Override
    public boolean shouldOverrideUrlLoading(WebView view, String url) {
        view.loadUrl(url);
        return true;
    }

    @Override

public WebResourceResponse shouldInterceptRequest(WebView view, String url) {      // 如果命中本地资源，使用本地资源替代
        if (mDataHelper.hasLocalResource(url)){
            WebResourceResponse response = mDataHelper.getReplacedWebResourceResponse(getApplicationContext(), url);
           if (response !=\u25a1 \u25a1 null) {
```

```
                return response;
            }
        }
        return super.shouldInterceptRequest(view, url);
    }

    @TargetApi(VERSION_CODES.LOLLIPOP)@Override
public WebResourceResponse shouldInterceptRequest(WebView view,WebResourceRequest reques
t) {
        String url = request.getUrl().toString();
        if (mDataHelper.hasLocalResource(url)) {
WebResourceResponse response =  mDataHelper.getReplacedWebResourceResponse(getApplicatio
nContext(), url);
            if (response !=null) {
                return response;
            }
        }
        return super.shouldInterceptRequest(view, request);
    }
});
```

- WebView 初始化慢，可以在初始化同时先请求数据，让后端和网络不要闲着。
- 后端处理慢，可以让服务器分 trunk 输出，在后端计算的同时前端也加载网络静态资源。
- 脚本执行慢，就让脚本在最后运行，不阻塞页面解析。
- 同时，合理的预加载、预缓存可以让加载速度的瓶颈更小。

- WebView 初始化慢，就随时初始化好一个 WebView 待用。
- DNS 和链接慢，想办法复用客户端使用的域名和链接。
- 脚本执行慢，可以把框架代码拆分出来，在请求页面之前就执行好。



2.19.3 内存泄漏

直接 new WebView 并传入 application context 代替在 XML 里面声明以防止 activity 引用被滥用，能解决 90+%的 WebView 内存泄漏。

```
vWeb =  new WebView(getContext().getApplicationContext());

container.addView(vWeb);
```

销毁 WebView

```
if (vWeb != null) {

    vWeb.setWebViewClient(null);

    vWeb.setWebChromeClient(null);

    vWeb.loadDataWithBaseURL(null, "", "text/html", "utf-8", null);

    vWeb.clearHistory();
```

```
    ((ViewGroup) vWeb.getParent()).removeView(vWeb);

    vWeb.destroy();

    vWeb = null;

}
```

三、Android 扩展知识点

3.1 ART

ART 代表 Android Runtime，其处理应用程序执行的方式完全不同于 Dalvik，Dalvik 是依靠一个 Just-In-Time (JIT) 编译器去解释字节码。开发者编译后的应用代码需要通过一个解释器在用户的设备上运行，这一机制并不高效，但让应用能更容易在不同硬件和架构上运 行。ART 则完全改变了这套做法，在应用安装时就预编译字节码到机器语言，这一机制叫 Ahead-Of-Time (AOT）编译。在移除解释代码这一过程后，应用程序执行将更有效率，启动更快。

3.1.1 ART 功能

3.1.1.1 预先 (AOT) 编译

ART 引入了预先编译机制，可提高应用的性能。ART 还具有比 Dalvik 更严格的安装时验证。在安装时，ART 使用设备自带的 dex2oat 工具来编译应用。该实用工具接受 DEX 文件作为输入，并为目标设备生成经过编译的应用可执行文件。该工具应能够顺利编译所有有效的 DEX 文件。

3.1.1.2 垃圾回收优化

垃圾回收 (GC) 可能有损于应用性能，从而导致显示不稳定、界面响应速度缓慢以及其他问题。ART 通过以下几种方式对垃圾回收做了优化：

- 只有一次（而非两次）GC 暂停
- 在 GC 保持暂停状态期间并行处理

- 在清理最近分配的短时对象这种特殊情况中，回收器的总 GC 时间更短
- 优化了垃圾回收的工效，能够更加及时地进行并行垃圾回收，这使得 GC_FOR_ALLOC 事件在典型用例中极为罕见
- 压缩 GC 以减少后台内存使用和碎片

3.1.1.3 开发和调试方面的优化

- 支持采样分析器

一直以来，开发者都使用 Traceview 工具（用于跟踪应用执行情况）作为分析器。虽然

Traceview 可提供有用的信息，但每次方法调用产生的开销会导致 Dalvik 分析结果出现偏差，而且使用该工具明显会影响运行时性能

ART 添加了对没有这些限制的专用采样分析器的支持，因而可更准确地了解应用执行情况，

而不会明显减慢速度。KitKat 版本为 Dalvik 的 Traceview 添加了采样支持。

- 支持更多调试功能

ART 支持许多新的调试选项，特别是与监控和垃圾回收相关的功能。例如，查看堆栈跟踪中保留了哪些锁，然后跳转到持有锁的线程；询问指定类的当前活动的实例数、请求查看实例，以及查看使对象保持有效状态的参考；过滤特定实例的事件（如断点）等。

- 优化了异常和崩溃报告中的诊断详细信息

当发生运行时异常时，ART 会为您提供尽可能多的上下文和详细信息。ART 会提

供 `java.lang.ClassCastException`、

`java.lang.ClassNotFoundException` 和 `java.lang.NullPointerException` 的更多异常详细信息

（较高版本的 Dalvik 会提

供 `java.lang.ArrayIndexOutOfBoundsException` 和 `java.lang.ArrayStoreException` 的更多异常详细信息，这些信息现在包括数组大小和越界偏移量；ART 也提供这类信息）。

3.1.2 ART GC

ART 有多个不同的 GC 方案，这些方案包括运行不同垃圾回收器。默认方案是 CMS（并发标记清除）方案，主要使用粘性 CMS 和部分 CMS。粘性 CMS 是 ART 的不移动分代垃圾回收器。它仅扫描堆中自上次 GC 后修改的部分，并且只能回收自上次 GC 后分配的对象。除 CMS 方案外，当应用将进程状态更改为察觉不到卡顿的进程状态（例如，后台或缓存）时，ART 将执行堆压缩。

除了新的垃圾回收器之外，ART 还引入了一种基于位图的新内存分配程序，称为 RosAlloc（插槽运行分配器）。此新分配器具有分片锁，当分配规模较小时可添加线程的本地缓冲区，因而性能优于 DlMalloc。

与 Dalvik 相比，ART CMS 垃圾回收计划在很多方面都有一定的改善：

- 与 Dalvik 相比，暂停次数从 2 次减少到 1 次。Dalvik 的第一次暂停主要是为了进行根标记，即在 ART 中进行并发标记，让线程标记自己的根，然后马上恢复运行。
- 与 Dalvik 类似，ART GC 在清除过程开始之前也会暂停 1 次。两者在这方面的主要差异在于：在此暂停期间，某些 Dalvik 环节在 ART 中并发进行。这些环节包括 java.lang.ref.Reference 处理、系统弱清除（例如，jni 弱全局等）、重新标记非线程根和卡片预清理。在 ART 暂停期间仍进行的阶段包括扫描脏卡片以及重新标记线程根，这些操作有助于缩短暂停时间。

- 相对于 Dalvik，ART GC 改进的最后一个方面是粘性 CMS 回收器增加了 GC 吞吐量。

  不同于普通的分代 GC，粘性 CMS 不移动。系统会将年轻对象保存在一个分配堆栈（基本上是 java.lang.Object 数组）中，而非为其设置一个专属区域。这样可以避免移动所需的对象以维持低暂停次数，但缺点是容易在堆栈中加入大量复杂对象图像而使堆栈变长。

ART GC 与 Dalvik 的另一个主要区别在于 ART GC 引入了移动垃圾回收器。使用移动 GC 的目的在于通过堆压缩来减少后台应用使用的内存。目前，触发堆压缩的事件是 ActivityManager 进程状态的改变。当应用转到后台运行时，它会通知 ART 已进入不再“感知”卡顿的进程状态。此时 ART 会进行一些操作（例如，压缩和监视器压缩），从而导致应用线程长时间暂停。目前正在使用的两个移动 GC 是同构空间压缩和半空间压缩。

- 半空间压缩将对象在两个紧密排列的碰撞指针空间之间进行移动。这种移动 GC 适用于小内存设备，因为它可以比同构空间压缩稍微多节省一点内存。额外节省出的空间主要来自紧密排列的对象，这样可以避免 RosAlloc/DlMalloc 分配器占用开销。由于 CMS 仍在前台使用，且不能从碰撞指针空间中进行收集，因此当应用在前台使用时，半空间还要再进行一次转换。这种情况并不理想，因为它可能引起较长时间的暂停。

- 同构空间压缩通过将对象从一个 RosAlloc 空间复制到另一个 RosAlloc 空间来实现。这有助于通过减少堆碎片来减少内存使用量。这是目前非低内存设备的默认压缩模式。相比半空间压缩，同构空间压缩的主要优势在于应用从后台切换到前台时无需进行堆转换。

3.2 Apk 包体优化

3.2.1 Apk 组成结构

文件/文件夹	作用/功能
res	包含所有没有被编译到 .arsc 里面的资源文件

文件/文件夹	作用/功能
lib	引用库的文件夹
assets	assets 文件夹相比于 res 文件夹，还有可能放字体文件、预置数据和 web 页面等,通过 AssetManager 访问
META_INF	存放的是签名信息，用来保证 apk 包的完整性和系统的安全。在生成一个 APK 的时候，会对所有的打包文件做一个校验计算，并把结果放在该目录下面
classes.dex	包含编译后的应用程序源码转化成的 dex 字节码。APK 里面，可能会存在多个 dex 文件
resources.arsc	一些资源和标识符被编译和写入这个文件
Androidmanifest.xml	编译时，应用程序的 AndroidManifest.xml 被转化成二进制格式

3.2.2 整体优化

- 分离应用的独立模块，以插件的形式加载
- 解压 APK，重新用 7zip 进行压缩
- 用 apksigner 签名工具 替代 java 提供的 jarsigner 签名工具

3.2.3 资源优化

- 可以只用一套资源图片，一般采用 xhdpi 下的资源图片
- 通过扫描文件的 MD5 值，找出名字不同，内容相同的图片并删除

- 通过 Lint 工具扫描工程资源，移除无用资源
- 通过 Gradle 参数配置 shrinkResources=true
- 对 png 图片压缩
- 图片资源考虑采用 WebP 格式
- 避免使用帧动画，可使用 Lottie 动画库
- 优先考虑能否用 shape 代码、.9 图、svg 矢量图、VectorDrawable 类来替换传统的图片

3.2.4 代码优化

- 启用混淆以移除无用代码
- 剔除 R 文件
- 用注解替代枚举

3.2.5 .arsc 文件优化

- 移除未使用的备用资源来优化 .arsc 文件

```
android {
    defaultConfig {
        ...
        resConfigs "zh", "zh_CN", "zh_HK", "en"
    }
}
```

3.2.6 lib 目录优化

- 只提供对主流架构的支持，比如 arm，对于 mips 和 x86 架构可以考虑不提供支持

```
android {
    defaultConfig {
        ...
        ndk {
            abiFilters  "armeabi-v7a"
        }
    }
}
```

3.3 Hook

3.3.1 基本流程

1、根据需求确定 要 hook 的对象

2、寻找要 hook 的对象的持有者，拿到要 hook 的对象

3、定义"要 hook 的对象"的代理类，并且创建该类的对象

4、使用上一步创建出来的对象，替换掉要 hook 的对象

3.3.2 使用示例

```
/*** hook 的核心代码* 这个方法的唯一目的：用自己的点击事件，替换掉 View 原来的点击事件** @param view hook 的范围仅限于这个 view*/@SuppressLint({"DiscouragedPrivateApi", "PrivateApi"})public static void hook(Context context, final View view) {//
    try {
        // 反射执行 View 类的 getListenerInfo()方法，拿到 v 的 mListenerInfo 对象，这个对象就是点击事件的持有者
```

```
        Method method = View.class.getDeclaredMethod("getListenerInfo");

        method.setAccessible(true);//由于 getListenerInfo()方法并不是 public 的，所以要加这个代码来保证访问权限

        Object mListenerInfo = method.invoke(view);//这里拿到的就是 mListenerInfo 对象，也就是点击事件的持有者



        // 要从这里面拿到当前的点击事件对象

        Class<?> listenerInfoClz = Class.forName("android.view.View$ListenerInfo");// 这是内部类的表示方法

        Field field = listenerInfoClz.getDeclaredField("mOnClickListener");

        final View.OnClickListener onClickListenerInstance = (View.OnClickListener) field.get(mListenerInfo);//取得真实的 mOnClickListener 对象



        // 2. 创建我们自己的点击事件代理类

        //   方式 1：自己创建代理类

        //   ProxyOnClickListener proxyOnClickListener = new ProxyOnClickListener(onClickListenerInstance);

        //   方式 2：由于 View.OnClickListener 是一个接口，所以可以直接用动态代理模式

        // Proxy.newProxyInstance 的 3 个参数依次分别是：

        // 本地的类加载器;

        // 代理类的对象所继承的接口（用 Class 数组表示，支持多个接口）

        // 代理类的实际逻辑，封装在 new 出来的 InvocationHandler 内

        Object proxyOnClickListener = Proxy.newProxyInstance(context.getClass().getClassLoader(), new Class[]{View.OnClickListener.class}, new InvocationHandler() {

            @Override

            public Object invoke(Object proxy, Method method, Object[] args) throws Throwable {
```

```
                Log.d("HookSetOnClickListener", "点击事件被 hook 到了");//加入自己的逻辑
                return method.invoke(onClickListenerInstance, args);//执行被代理的对象的逻辑
            }
        });
        // 3. 用我们自己的点击事件代理类，设置到"持有者"中
        field.set(mListenerInfo, proxyOnClickListener);
    } catch (Exception e) {
        e.printStackTrace();
    }
}
// 自定义代理类 static class ProxyOnClickListener implements View.OnClickListener {
    View.OnClickListener oriLis;

    public ProxyOnClickListener(View.OnClickListener oriLis) {
        this.oriLis = oriLis;
    }

    @Override
    public void onClick(View v) {
        Log.d("HookSetOnClickListener", "点击事件被 hook 到了");
        if (oriLis != null) {
            oriLis.onClick(v);
        }
    }
}
```

3.4 Proguard

Proguard 具有以下三个功能：

- 压缩（Shrink）：检测和删除没有使用的类，字段，方法和特性
- 优化（Optimize）：分析和优化 Java 字节码
- 混淆（Obfuscate）：使用简短的无意义的名称，对类，字段和方法进行重命名

3.4.1 公共模板

```
##############################################

#

# 对于一些基本指令的添加

#

##############################################

# 代码混淆压缩比，在 0~7 之间，默认为 5，一般不做修改

-optimizationpasses 5



# 混合时不使用大小写混合，混合后的类名为小写

-dontusemixedcaseclassnames



# 指定不去忽略非公共库的类

-dontskipnonpubliclibraryclasses



# 这句话能够使我们的项目混淆后产生映射文件

# 包含有类名->混淆后类名的映射关系
```

```
-verbose


# 指定不去忽略非公共库的类成员

-dontskipnonpubliclibraryclassmembers


# 不做预校验，preverify 是 proguard 的四个步骤之一，Android 不需要 preverify，去掉这一步能够加快混淆速度。

-dontpreverify


# 保留 Annotation 不混淆

-keepattributes *Annotation*,InnerClasses


# 避免混淆泛型

-keepattributes Signature


# 抛出异常时保留代码行号

-keepattributes SourceFile,LineNumberTable


# 指定混淆是采用的算法，后面的参数是一个过滤器

# 这个过滤器是谷歌推荐的算法，一般不做更改

-optimizations !code/simplification/cast,!field/*,!class/merging/*




##############################################

#
```

```
# Android 开发中一些需要保留的公共部分

#

############################################


# 保留我们使用的四大组件，自定义的 Application 等等这些类不被混淆

# 因为这些子类都有可能被外部调用

-keep public class * extends android.app.Activity

-keep public class * extends android.app.Appliction

-keep public class * extends android.app.Service

-keep public class * extends android.content.BroadcastReceiver

-keep public class * extends android.content.ContentProvider

-keep public class * extends android.app.backup.BackupAgentHelper

-keep public class * extends android.preference.Preference

-keep public class * extends android.view.View

-keep public class com.android.vending.licensing.ILicensingService




# 保留 support 下的所有类及其内部类

-keep class android.support.** { *; }


# 保留继承的

-keep public class * extends android.support.v4.**

-keep public class * extends android.support.v7.**

-keep public class * extends android.support.annotation.**
```

```
# 保留 R 下面的资源
-keep class **.R$* { *; }

# 保留本地 native 方法不被混淆
-keepclasseswithmembernames class * {
    native <methods>;
}

# 保留在 Activity 中的方法参数是 view 的方法，
# 这样以来我们在 layout 中写的 onClick 就不会被影响
-keepclassmembers class * extends android.app.Activity {
    public void *(android.view.View);
}

# 保留枚举类不被混淆
-keepclassmembers enum * {
    public static **[] values();
    public static ** valueOf(java.lang.String);
}

# 保留我们自定义控件（继承自 View）不被混淆
-keep public class * extends android.view.View {
    *** get*();
    void set*(***);
    public <init>(android.content.Context);
```

```
    public <init>(android.content.Context, android.util.AttributeSet);
    public <init>(android.content.Context, android.util.AttributeSet, int);
}

# 保留 Parcelable 序列化类不被混淆
-keep class * implements android.os.Parcelable {
    public static final android.os.Parcelable$Creator *;
}

# 保留 Serializable 序列化的类不被混淆
-keepnames class * implements java.io.Serializable
-keepclassmembers class * implements java.io.Serializable {
    static final long serialVersionUID;
    private static final java.io.ObjectStreamField[] serialPersistentFields;
    !static !transient <fields>;
    !private <fields>;
    !private <methods>;
    private void writeObject(java.io.ObjectOutputStream);
    private void readObject(java.io.ObjectInputStream);
    java.lang.Object writeReplace();
    java.lang.Object readResolve();
}

# 对于带有回调函数的 onXXEvent、**On*Listener 的，不能被混淆
-keepclassmembers class * {
```

```
    void *(**On*Event);
    void *(**On*Listener);
}

# webView 处理，项目中没有使用到 webView 忽略即可
-keepclassmembers class fqcn.of.javascript.interface.for.webview {
    public *;
}
-keepclassmembers class * extends android.webkit.webViewClient {
    public void *(android.webkit.WebView, java.lang.String, android.graphics.Bitmap);
    public boolean *(android.webkit.WebView, java.lang.String);
}
-keepclassmembers class * extends android.webkit.webViewClient {
    public void *(android.webkit.webView, java.lang.String);
}

# js
-keepattributes JavascriptInterface
-keep class android.webkit.JavascriptInterface { *; }
-keepclassmembers class * {
    @android.webkit.JavascriptInterface <methods>;
}

# @Keep
-keep,allowobfuscation @interface android.support.annotation.Keep
```

```
-keep @android.support.annotation.Keep class *

-keepclassmembers class * {

    @android.support.annotation.Keep *;

}
```

3.4.2 常用的自定义混淆规则

```
# 通配符*，匹配任意长度字符，但不含包名分隔符(.)

# 通配符**，匹配任意长度字符，并且包含包名分隔符(.)



# 不混淆某个类

-keep public class com.jasonwu.demo.Test { *; }



# 不混淆某个包所有的类

-keep class com.jasonwu.demo.test.** { *; }



# 不混淆某个类的子类

-keep public class * com.jasonwu.demo.Test { *; }



# 不混淆所有类名中包含了 ``model`` 的类及其成员

-keep public class **.*model*.** {*;}



# 不混淆某个接口的实现

-keep class * implements com.jasonwu.demo.TestInterface { *; }



# 不混淆某个类的构造方法
```

```
-keepclassmembers class com.jasonwu.demo.Test {

  public <init>();

}



# 不混淆某个类的特定的方法

-keepclassmembers class com.jasonwu.demo.Test {

  public void test(java.lang.String);

}
```

3.4.3 aar 中增加独立的混淆配置

build.gralde

```
android {

    ...

    defaultConfig {

        ...

        consumerProguardFile 'proguard-rules.pro'

    }

    ...

}
```

3.4.4 检查混淆和追踪异常

开启 Proguard 功能，则每次构建时 ProGuard 都会输出下列文件：

- dump.txt

  说明 APK 中所有类文件的内部结构。

- mapping.txt

  提供原始与混淆过的类、方法和字段名称之间的转换。

- seeds.txt

  列出未进行混淆的类和成员。

- usage.txt

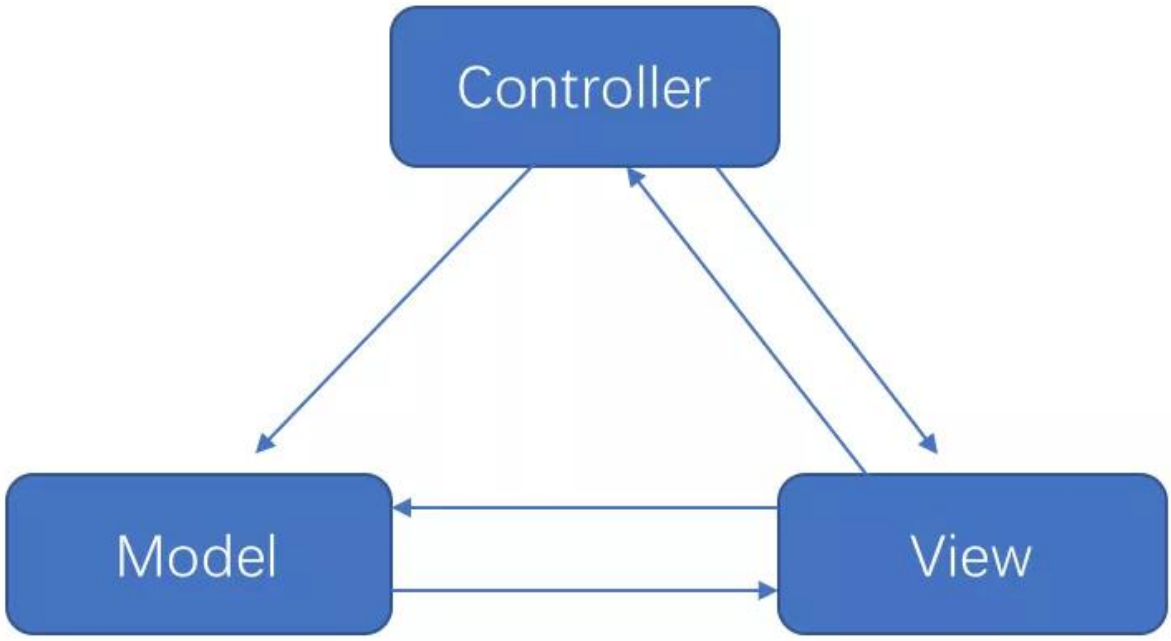
  列出从 APK 移除的代码。

这些文件保存在 /build/outputs/mapping/release/ 中。我们可以查看 seeds.txt 里面是否是我们需要保留的，以及 usage.txt 里查看是否有误删除的代码。 mapping.txt 文件很重要，由于我们的部分代码是经过重命名的，如果该部分出现 bug，对应的异常堆栈信息里的类或成员也是经过重命名的，难以定位问题。我们可以用 retrace 脚本（在 Windows 上为 retrace.bat；在 Mac/Linux 上为 retrace.sh）。它位于 /tools/proguard/ 目录中。该脚本利用 mapping.txt 文件和你的异常堆栈文件生成没有经过混淆的异常堆栈文件,这样就可以看清是哪里出问题了。使用 retrace 工具的语法如下：

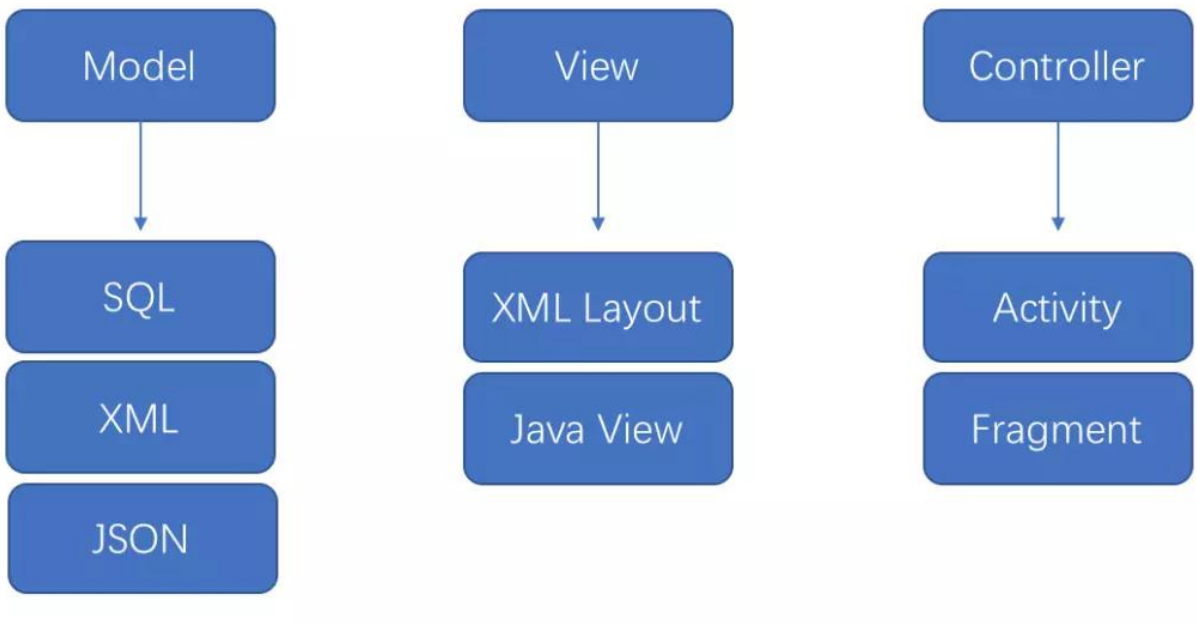
```
retrace.bat|retrace.sh [-verbose] mapping.txt [<stacktrace_file>]
```

3.5 架构

3.5.1 MVC



在 Android 中，三者的关系如下：

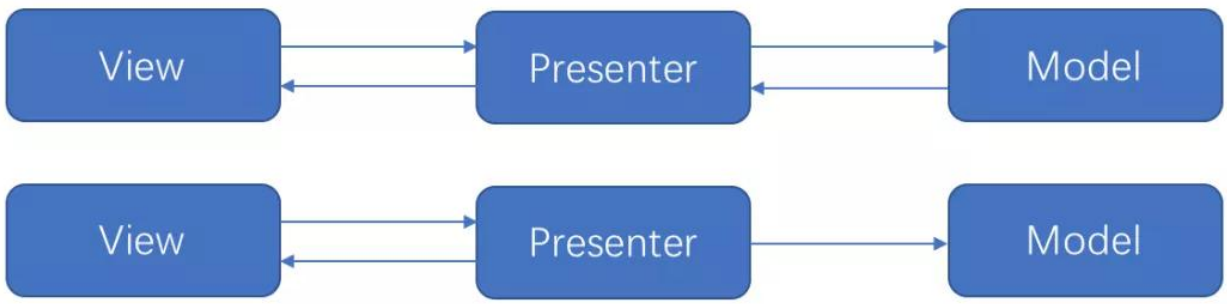


由于在 Android 中 xml 布局的功能性太弱，所以 Activity 承担了绝大部分的工作，所以在 Android 中 mvc 更像：

总结：

- 具有一定的分层，model 解耦，controller 和 view 并没有解耦
- controller 和 view 在 Android 中无法做到彻底分离，Controller 变得臃肿不堪
- 易于理解、开发速度快、可维护性高

3.5.2 MVP

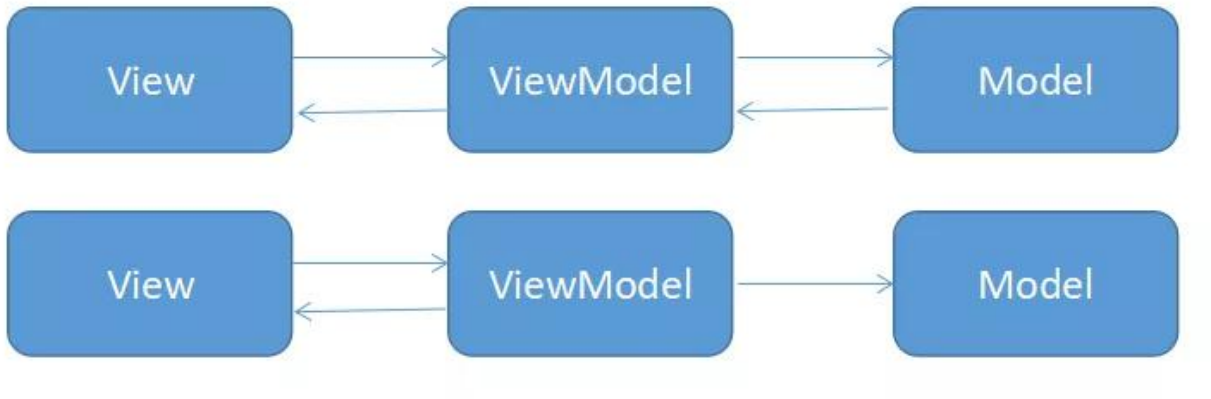


通过引入接口 BaseView，让相应的视图组件如 Activity，Fragment 去实现 BaseView，把业务逻辑放在 presenter 层中，弱化 Model 只有跟 view 相关的操作都由 View 层去完成。

总结：

- 彻底解决了 MVC 中 View 和 Controller 傻傻分不清楚的问题
- 但是随着业务逻辑的增加，一个页面可能会非常复杂，UI 的改变是非常多，会有非常多的 case，这样就会造成 View 的接口会很庞大
- 更容易单元测试

3.5.3 MVVM



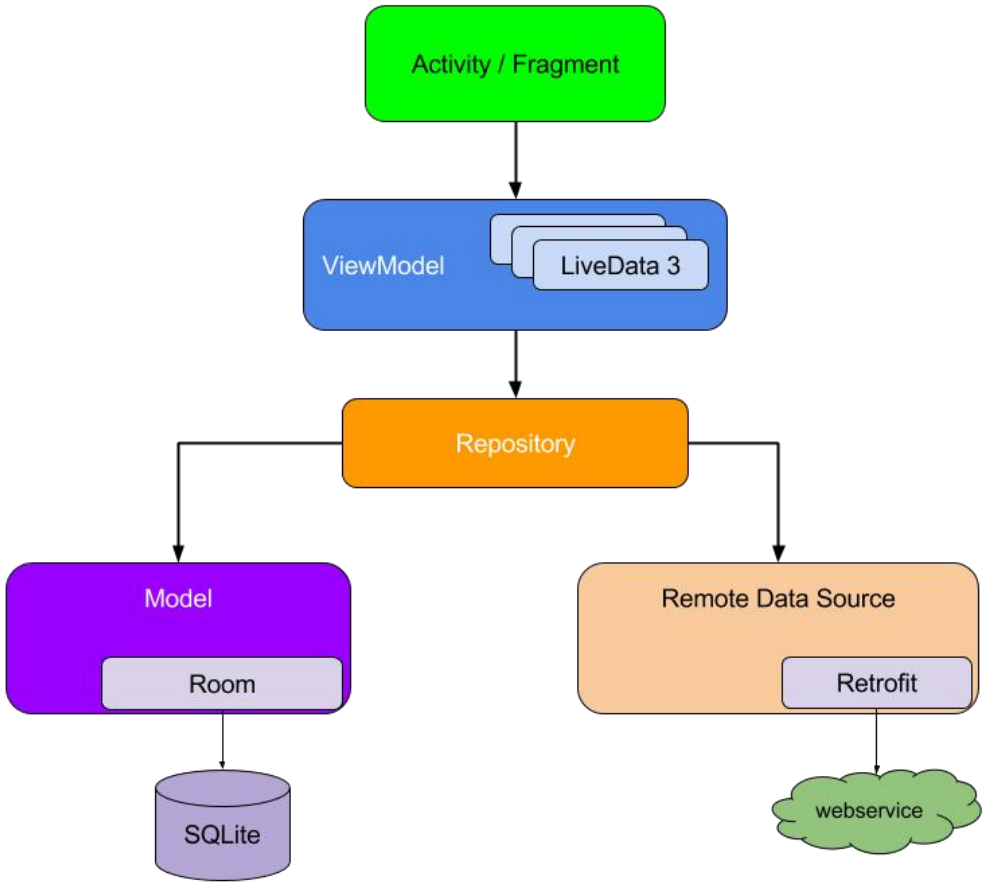
在 MVP 中 View 和 Presenter 要相互持有，方便调用对方，而在 MVP 中 View 和 ViewModel 通过 Binding 进行关联，他们之前的关联处理通过 DataBinding 完成。

总结：

- 很好的解决了 MVC 和 MVP 的问题
- 视图状态较多，ViewModel 的构建和维护的成本都会比较高
- 但是由于数据和视图的双向绑定，导致出现问题时不太好定位来源

3.6 Jetpack

3.6.1 架构



3.6.2 使用示例

build.gradle

```
android {
    ...
    dataBinding {
        enabled = true
    }
}
dependencies {
    ...
    implementation "androidx.fragment:fragment-ktx:$rootProject.fragmentVersion"
```

```
    implementation
"androidx.lifecycle:lifecycle-extensions:$rootProject.lifecycleVersion"

    implementation
"androidx.lifecycle:lifecycle-livedata-ktx:$rootProject.lifecycleVersion"

    implementation
"androidx.lifecycle:lifecycle-viewmodel-ktx:$rootProject.lifecycleVersion"

}
```

fragment_plant_detail.xml

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"

    xmlns:app="http://schemas.android.com/apk/res-auto"

    xmlns:tools="http://schemas.android.com/tools">



    <data>

        <variable

            name="viewModel"

            type="com.google.samples.apps.sunflower.viewmodels.PlantDetailViewModel" />

    </data>



    <androidx.constraintlayout.widget.ConstraintLayout

        android:layout_width="match_parent"

        android:layout_height="match_parent">



        <TextView

            ...

            android:text="@{viewModel.plant.name}"/>
```

```
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

PlantDetailFragment.kt

```
class PlantDetailFragment : Fragment() {


    private val args: PlantDetailFragmentArgs by navArgs()
    private lateinit var shareText: String


    private val plantDetailViewModel: PlantDetailViewModel by viewModels {
        InjectorUtils.providePlantDetailViewModelFactory(requireActivity(), args.plantId)
    }


    override fun onCreateView(
        inflater: LayoutInflater,
        container: ViewGroup?,
        savedInstanceState: Bundle?
    ): View? {
        val binding = DataBindingUtil.inflate<FragmentPlantDetailBinding>(
                inflater, R.layout.fragment_plant_detail, container, false).apply {
            viewModel = plantDetailViewModel
            lifecycleOwner = this@PlantDetailFragment
        }


        plantDetailViewModel.plant.observe(this) { plant ->
            // 更新相关 UI
```

```
        }


        return binding.root
    }
}
```

Plant.kt

```
data class Plant (
    val name: String
)
```

PlantDetailViewModel.kt

```
class PlantDetailViewModel(
    plantRepository: PlantRepository,
    private val plantId: String
) : ViewModel() {


    val plant: LiveData<Plant>


    override fun onCleared() {
        super.onCleared()
        viewModelScope.cancel()
    }


    init {
        plant = plantRepository.getPlant(plantId)
    }
```

```
}
```

PlantDetailViewModelFactory.kt

```
class PlantDetailViewModelFactory(
    private val plantRepository: PlantRepository,
    private val plantId: String
) : ViewModelProvider.NewInstanceFactory() {

    @Suppress("UNCHECKED_CAST")
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return PlantDetailViewModel(plantRepository, plantId) as T
    }
}
```

InjectorUtils.kt

```
object InjectorUtils {
    private fun getPlantRepository(context: Context): PlantRepository {
        ...
    }

    fun providePlantDetailViewModelFactory(
        context: Context,
        plantId: String
    ): PlantDetailViewModelFactory {
        return PlantDetailViewModelFactory(getPlantRepository(context), plantId)
    }
}
```

3.7 NDK 开发

> NDK 全称是 Native Development Kit，是一组可以让你在 Android 应用中编写实现 C/C++ 的工具，可以在项目用自己写源代码构建，也可以利用现有的预构建库。

使用 NDK 的使用目的有：

- 从设备获取更好的性能以用于计算密集型应用，例如游戏或物理模拟
- 重复使用自己或其他开发者的 C/C++ 库，便利于跨平台。
- NDK 集成了譬如 OpenSL、Vulkan 等 API 规范的特定实现，以实现在 java 层无法做到的功能如提升音频性能等
- 增加反编译难度

3.7.1 JNI 基础

3.7.1.1 数据类型

- 基本数据类型

Java 类型	Native 类型	符号属性	字长
boolean	jboolean	无符号	8 位
byte	jbyte	无符号	8 位
char	jchar	无符号	16 位
short	jshort	有符号	16 位
int	jnit	有符号	32 位

Java 类型	Native 类型	符号属性	字长
long	jlong	有符号	64 位
float	jfloat	有符号	32 位
double	jdouble	有符号	64 位

- 引用数据类型

Java 引用类型	Native 类型	Java 引用类型	Native 类型
All objects	jobject	char[]	jcharArray
java.lang.Class	jclass	short[]	jshortArray
java.lang.String	jstring	int[]	jintArray
Object[]	jobjectArray	long[]	jlongArray
boolean[]	jbooleanArray	float[]	jfloatArray
byte[]	jbyteArray	double[]	jdoubleArray
java.lang.Throwable	jthrowable		

3.7.1.2 String 字符串函数操作

JNI 函数	描述
GetStringChars / ReleaseStringChars	获得或释放一个指向 Unicode 编码的字符串的指

JNI 函数	描述
	针（指 C/C++ 字符串）
GetStringUTFChars / ReleaseStringUTFChars	获得或释放一个指向 UTF-8 编码的字符串的指针（指 C/C++ 字符串）
GetStringLength	返回 Unicode 编码的字符串的长度
getStringUTFLength	返回 UTF-8 编码的字符串的长度
NewString	将 Unicode 编码的 C/C++ 字符串转换为 Java 字符串
NewStringUTF	将 UTF-8 编码的 C/C++ 字符串转换为 Java 字符串
GetStringCritical / ReleaseStringCritical	获得或释放一个指向字符串内容的指针(指 Java 字符串)
GetStringRegion	获取或者设置 Unicode 编码的字符串的指定范围的内容
GetStringUTFRegion	获取或者设置 UTF-8 编码的字符串的指定范围的内容

3.7.1.3 常用 JNI 访问 Java 对象方法

MyJob.java

```
package com.example.myjniproject;

public class MyJob {
```

```
    public static String JOB_STRING = "my_job";

    private int jobId;


    public MyJob(int jobId) {

        this.jobId = jobId;

    }


    public int getJobId() {

        return jobId;

    }

}
```

native-lib.cpp

```
#include <jni.h>

extern "C"

JNIEXPORT jint JNICALLJava_com_example_myjniproject_MainActivity_getJobId(JNIEnv *env,
jobject thiz, jobject job) {


    // 根据实力获取 class 对象

    jclass jobClz = env->GetObjectClass(job);

    // 根据类名获取 class 对象

    jclass jobClz = env->FindClass("com/example/myjniproject/MyJob");


    // 获取属性 id

    jfieldID fieldId = env->GetFieldID(jobClz, "jobId", "I");
```

```
    // 获取静态属性 id
    jfieldID sFieldId = env->GetStaticFieldID(jobClz, "JOB_STRING", "Ljava/lang/String;");

    // 获取方法 id
    jmethodID methodId = env->GetMethodID(jobClz, "getJobId", "()I");
    // 获取构造方法 id
    jmethodID  initMethodId = env->GetMethodID(jobClz, "<init>", "(I)V");

    // 根据对象属性 id 获取该属性值
    jint id = env->GetIntField(job, fieldId);
    // 根据对象方法 id 调用该方法
    jint id = env->CallIntMethod(job, methodId);

    // 创建新的对象
    jobject newJob = env->NewObject(jobClz, initMethodId, 10);

    return id;
}
```

3.7.2 NDK 开发

3.7.2.1 基础开发流程

- 在 java 中声明 native 方法

```
public class MainActivity extends AppCompatActivity {
```

```
    // Used to load the 'native-lib' library on application startup.
    static {
        System.loadLibrary("native-lib");
    }

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        Log.d("MainActivity", stringFromJNI());
    }

    private native String stringFromJNI();
}
```

- 在 app/src/main 目录下新建 cpp 目录，新建相关 cpp 文件，实现相关方法（AS 可用快捷键快速生成）

native-lib.cpp

```
#include <jni.h>

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_myjniproject_MainActivity_stringFromJNI(
        JNIEnv *env,
        jobject /* this */) {
    std::string hello = "Hello from C++";
```

```
    return env->NewStringUTF(hello.c_str());
}
```

- 函数名的格式遵循遵循如下规则：Java_包名_类名_方法名。
- extern "C" 指定采用 C 语言的命名风格来编译，否则由于 C 与 C++ 风格不同，导致链接时无法找到具体的函数
- JNIEnv*：表示一个指向 JNI 环境的指针，可以通过他来访问 JNI 提供的接口方法
- jobject：表示 java 对象中的 this
- JNIEXPORT 和 JNICALL：JNI 所定义的宏，可以在 jni.h 头文件中查找到

- 通过 CMake 或者 ndk-build 构建动态库

3.7.2.2 System.loadLibrary()

java/lang/System.java:

```
@CallerSensitivepublic static void load(String filename) {
    Runtime.getRuntime().load0(Reflection.getCallerClass(), filename);
}
```

- 调用 Runtime 相关 native 方法

java/lang/Runtime.java:

```
private static native String nativeLoad(String filename, ClassLoader loader, Class<?>
caller);
```

- native 方法的实现如下：

dalvik/vm/native/java_lang_Runtime.cpp:

```
static void Dalvik_java_lang_Runtime_nativeLoad(const u4* args,
    JValue* pResult)
{
    ...
    bool success;

    assert(fileNameObj != NULL);
    // 将 Java 的 library path String 转换到 native 的 String
    fileName = dvmCreateCstrFromString(fileNameObj);

    success = dvmLoadNativeCode(fileName, classLoader, &reason);
    if (!success) {
        const char* msg = (reason != NULL) ? reason : "unknown failure";
        result = dvmCreateStringFromCstr(msg);
        dvmReleaseTrackedAlloc((Object*) result, NULL);
    }
    ...
}
```

- dvmLoadNativeCode 函数实现如下：

dalvik/vm/Native.cpp

```
bool dvmLoadNativeCode(const char* pathName, Object* classLoader,
        char** detail)
{
```

```
SharedLib* pEntry;
void* handle;
...
*detail = NULL;

// 如果已经加载过了，则直接返回 true
pEntry = findSharedLibEntry(pathName);
if (pEntry != NULL) {
    if (pEntry->classLoader != classLoader) {
        ...
        return false;
    }
    ...
    if (!checkOnLoadResult(pEntry))
        return false;
    return true;
}

Thread* self = dvmThreadSelf();
ThreadStatus oldStatus = dvmChangeStatus(self, THREAD_VMWAIT);
// 把.so mmap 到进程空间，并把 func 等相关信息填充到 soinfo 中
handle = dlopen(pathName, RTLD_LAZY);
dvmChangeStatus(self, oldStatus);
...
// 创建一个新的 entry
```

```
SharedLib* pNewEntry;

pNewEntry = (SharedLib*) calloc(1, sizeof(SharedLib));

pNewEntry->pathName = strdup(pathName);

pNewEntry->handle = handle;

pNewEntry->classLoader = classLoader;

dvmInitMutex(&pNewEntry->onLoadLock);

pthread_cond_init(&pNewEntry->onLoadCond, NULL);

pNewEntry->onLoadThreadId = self->threadId;


// 尝试添加到列表中

SharedLib* pActualEntry = addSharedLibEntry(pNewEntry);


if (pNewEntry != pActualEntry) {

    ...

    freeSharedLibEntry(pNewEntry);

    return checkOnLoadResult(pActualEntry);

} else {

    ...

    bool result = true;

    void* vonLoad;

    int version;

    // 调用该 so 库的 JNI_OnLoad 方法

    vonLoad = dlsym(handle, "JNI_OnLoad");

    if (vonLoad == NULL) {

        ...
```

```
    } else {
        // 调用 JNI_Onload 方法，重写类加载器。
        OnLoadFunc func = (OnLoadFunc)vonLoad;
        Object* prevOverride = self->classLoaderOverride;

        self->classLoaderOverride = classLoader;
        oldStatus = dvmChangeStatus(self, THREAD_NATIVE);
        ...
        version = (*func)(gDvmJni.jniVm, NULL);
        dvmChangeStatus(self, oldStatus);
        self->classLoaderOverride = prevOverride;

        if (version != JNI_VERSION_1_2 && version != JNI_VERSION_1_4 &&
            version != JNI_VERSION_1_6)
        {
            ...
            result = false;
        } else {
            ...
        }
    }

    if (result)
        pNewEntry->onLoadResult = kOnLoadOkay;
    else
```

```
            pNewEntry->onLoadResult = kOnLoadFailed;


        pNewEntry->onLoadThreadId = 0;


        // 释放锁资源
        dvmLockMutex(&pNewEntry->onLoadLock);
        pthread_cond_broadcast(&pNewEntry->onLoadCond);
        dvmUnlockMutex(&pNewEntry->onLoadLock);
        return result;
    }
}
```

3.7.3 CMake 构建 NDK 项目

> CMake 是一个开源的跨平台工具系列，旨在构建，测试和打包软件，从 Android Studio 2.2 开始，Android Sudio 默认地使用 CMake 与 Gradle 搭配使用来构建原生库。

启动方式只需要在 `app/build.gradle` 中添加相关：

```
android {
    ...
    defaultConfig {
        ...
        externalNativeBuild {
            cmake {
                cppFlags ""
            }
```

```
        }


        ndk {
            abiFilters 'arm64-v8a', 'armeabi-v7a'
        }
    }
    ...
    externalNativeBuild {
        cmake {
            path "CMakeLists.txt"
        }
    }
}
```

然后在对应目录新建一个 CMakeLists.txt 文件：

```
# 定义了所需 CMake 的最低版本
cmake_minimum_required(VERSION 3.4.1)


# add_library() 命令用来添加库
# native-lib 对应着生成的库的名字
# SHARED 代表为分享库
# src/main/cpp/native-lib.cpp 则是指明了源文件的路径。
add_library( # Sets the name of the library.
        native-lib
```

```
        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        src/main/cpp/native-lib.cpp)

# find_library 命令添加到 CMake 构建脚本中以定位 NDK 库，并将其路径存储为一个变量。
# 可以使用此变量在构建脚本的其他部分引用 NDK 库
find_library( # Sets the name of the path variable.
        log-lib

        # Specifies the name of the NDK library that
        # you want CMake to locate.
        log)

# 预构建的 NDK 库已经存在于 Android 平台上，因此，无需再构建或将其打包到 APK 中。
# 由于 NDK 库已经是 CMake 搜索路径的一部分，只需要向 CMake 提供希望使用的库的名称，并将其关联到自己的原生库中

# 要将预构建库关联到自己的原生库
target_link_libraries( # Specifies the target library.
        native-lib

        # Links the target library to the log library
        # included in the NDK.
```

```
        ${log-lib})

...
```

- CMake 命令详细信息文档

3.7.4 常用的 Android NDK 原生 API

支持 NDK 的 API 级别	关键原生 API	包括
3	Java 原生接口	#include <jni.h>
3	Android 日志记录 API	#include <android/log.h>
5	OpenGL ES 2.0	#include <GLES2/gl2.h> #include <GLES2/gl2ext.h>
8	Android 位图 API	#include <android/bitmap.h>
9	OpenSL ES	#include <SLES/OpenSLES.h> #include <SLES/OpenSLES_Platform.h> #include <SLES/OpenSLES_Android.h> #include <SLES/OpenSLES_AndroidConfiguration.h>

支持 NDK 的 API 级别	关键原生 API	包括
9	原生应用 API	#include <android/rect.h> #include <android/window.h> #include<android/native_activity.h> ...
18	OpenGL ES 3.0	#include <GLES3/gl3.h> #include <GLES3/gl3ext.h>
21	原生媒体 API	#include <media/NdkMediaCodec.h> #include <media/NdkMediaCrypto.h> ...
24	原生相机 API	#include <camera/NdkCameraCaptureSession.h> #include <camera/NdkCameraDevice.h> ...
...		

3.8 计算机网络基础

3.8.1 网络体系的分层结构

分层	说明
应用层（HTTP、FTP、DNS、SMTP 等）	定义了如何包装和解析数据，应用层是 http 协议的话，则会按照协议规定包装数据，如按照请求行、请求头、请求体包装，包装好数据后将数据传至运输层
运输层（TCP、UDP 等）	运输层有 TCP 和 UDP 两种，分别对应可靠和不可靠的运输。在这一层，一般都是和 Socket 打交道，Socket 是一组封装的编程调用接口，通过它，我们就能操作 TCP、UDP 进行连接的建立等。这一层指定了把数据送到对应的端口号
网络层（IP 等）	这一层 IP 协议，以及一些路由选择协议等等，所以这一层的指定了数据要传输到哪个 IP 地址。中间涉及到一些最优线路，路由选择算法等
数据链路层（ARP）	负责把 IP 地址解析为 MAC 地址，即硬件地址，这样就找到了对应的唯一的机器
物理层	提供二进制流传输服务，也就是真正开始通过传输介质（有线、无线）开始进行数据的传输

3.8.2 Http 相关

3.8.2.1 请求报文与响应报文

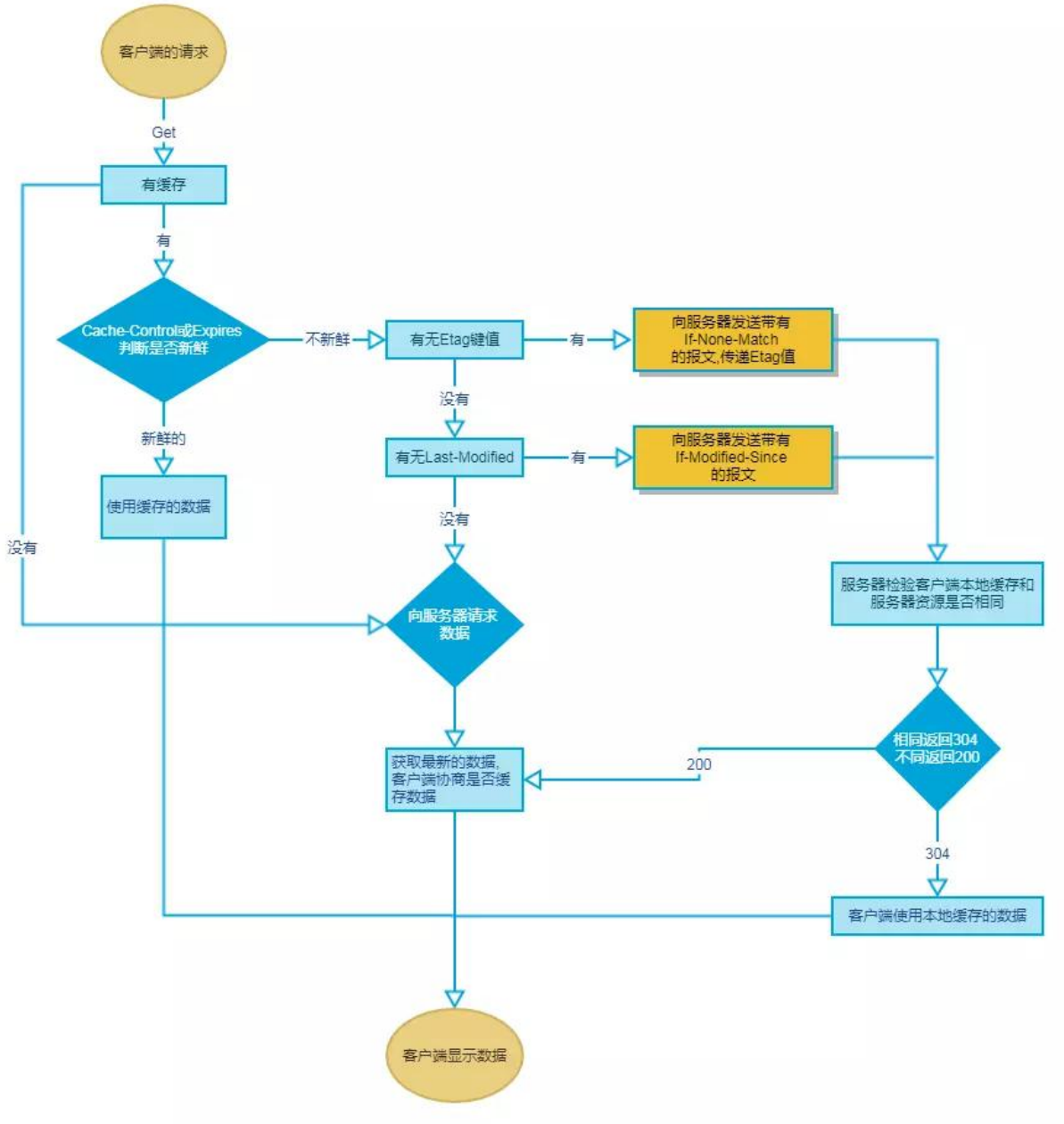
- 请求报文

名称	组成
请求行	请求方法如 post/get、请求路径 url、协议版本等
请求头	即 header，里面包含了很多字段
请求体	发送的数据

- 响应报文

名称	组成
状态行	状态码如 200、协议版本等
响应头	即返回的 header
响应体	响应的正文数据

3.8.2.2 缓存机制



- Cache-control 主要包含以下几个字段：

字段	说明
private	只有客户端可以缓存

字段	说明
public	客户端和代理服务器都可以缓存
max-age	缓存的过期时间
no-cache	需要使用对比缓存来验证缓存数据，如果服务端确认资源没有更新，则返回 304，取本地缓存即可，如果有更新，则返回最新的资源。做对比缓存与 Etag 有关。
no-store	这个字段打开，则不会进行缓存，也不会取缓存

- Etag：当客户端发送第一次请求时服务端会下发当前请求资源的标识码 Etag ，下次再请求时，客户端则会通过 header 里的 If-None-Match 将这个标识码 Etag 带上，服务端将客户端传来的 Etag 与最新的资源 Etag 做对比，如果一样，则表示资源没有更新，返回 304。

3.8.2.3 Https

Https 保证了我们数据传输的安全，Https = Http + Ssl，之所以能保证安全主要的原理就是利用了非对称加密算法，平常用的对称加密算法之所以不安全，是因为双方是用统一的密匙进行加密解密的，只要双方任意一方泄漏了密匙，那么其他人就可以利用密匙解密数据。

3.8.2.4 Http 2.0

Okhttp 支持配置使用 Http 2.0 协议，Http2.0 相对于 Http1.x 来说提升是巨大的，主要有以下几点：

- **二进制格式**：http1.x 是文本协议，而 http2.0 是二进制以帧为基本单位，是一个二进制协议，一帧中除了包含数据外同时还包含该帧的标识：Stream Identifier，即标识了该帧属于哪个 request,使得网络传输变得十分灵活。
- **多路复用**：多个请求共用一个 TCP 连接，多个请求可以同时在这个 TCP 连接上并发，一个是解决了建立多个 TCP 连接的消耗问题，一个也解决了效率的问题。
- **header 压缩**：主要是通过压缩 header 来减少请求的大小，减少流量消耗，提高效率。
- **支持服务端推送**

3.8.3 TCP/IP

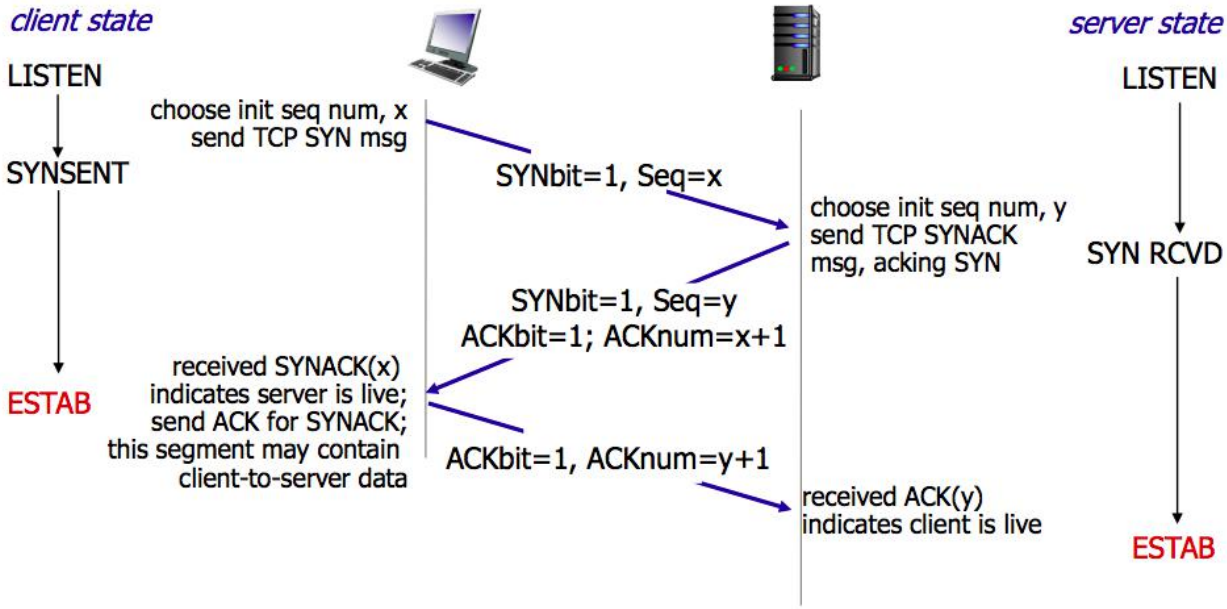
IP（Internet Protocol）协议提供了主机和主机间的通信，为了完成不同主机的通信，我们需要某种方式来唯一标识一台主机，这个标识，就是著名的 IP 地址。通过 IP 地址，IP 协议就能够帮我们把一个数据包发送给对方。

TCP 的全称是 Transmission Control Protocol，TCP 协议在 IP 协议提供的主机间通信功能的基础上，完成这两个主机上进程对进程的通信。

3.8.3.1 三次握手

所谓三次握手(Three-way Handshake)，是指建立一个 TCP 连接时，需要客户端和服务器总共发送 3 个包。

三次握手的目的是连接服务器指定端口，建立 TCP 连接，并同步连接双方的序列号和确认号，交换 TCP 窗口大小信息。在 socket 编程中，客户端执行 connect() 时。将触发三次握手。



- 第一次握手(SYN=1, seq=x):

客户端发送一个 TCP 的 SYN 标志位置 1 的包，指明客户端打算连接的服务器的端口，以及初始序号 X，保存在包头的序列号 (Sequence Number) 字段里。

发送完毕后，客户端进入 `SYN_SEND` 状态。

- 第二次握手(SYN=1, ACK=1, seq=y, ACKnum=x+1):

服务器发回确认包(ACK)应答。即 SYN 标志位和 ACK 标志位均为 1。服务器端选择自己 ISN 序列号，放到 Seq 域里，同时将确认序号(Acknowledgement Number)设置为客户的 ISN 加 1，即 X+1。 发送完毕后，服务器端进入 `SYN_RCVD` 状态。

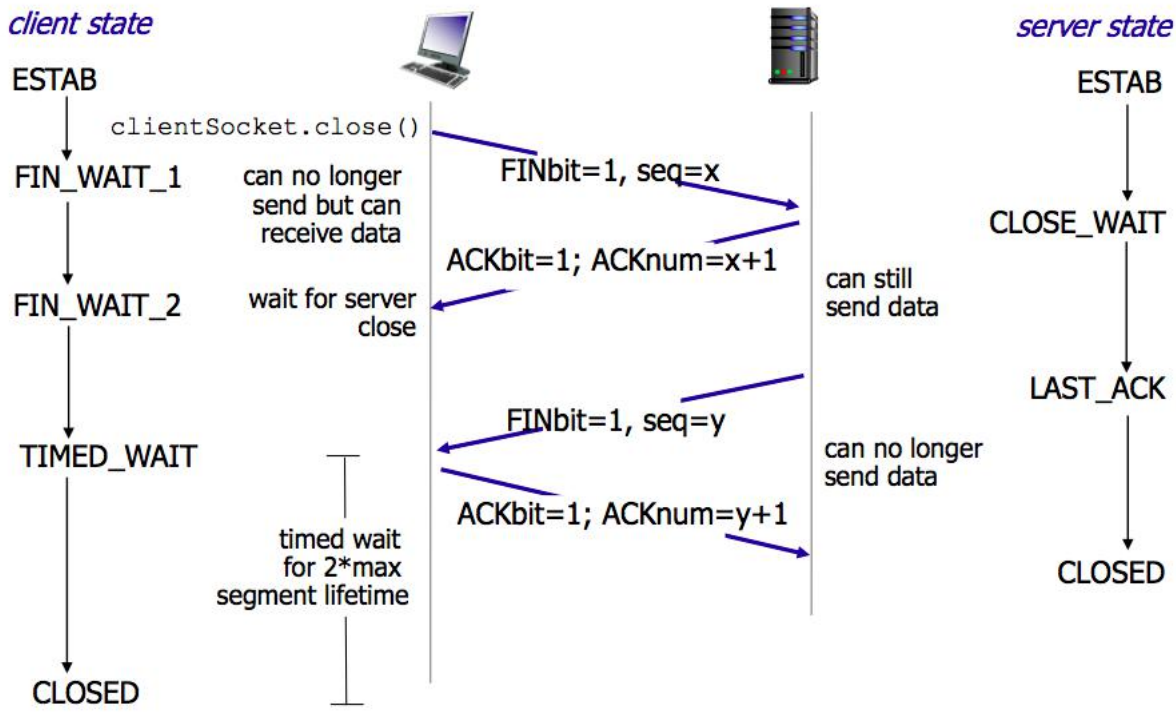
- 第三次握手(ACK=1，ACKnum=y+1)

客户端再次发送确认包(ACK)，SYN 标志位为 0，ACK 标志位为 1，并且把服务器发来 ACK 的序号字段 +1，放在确定字段中发送给对方，并且在数据段放写 ISN 的 +1

发送完毕后，客户端进入 ESTABLISHED 状态，当服务器端接收到这个包时，也进入 ESTABLISHED 状态，TCP 握手结束。

3.8.3.2 四次挥手

TCP 的连接的拆除需要发送四个包，因此称为四次挥手(Four-way handshake)，也叫做改进的三次握手。客户端或服务器均可主动发起挥手动作，在 socket 编程中，任何一方执行 close() 操作即可产生挥手操作。



- 第一次挥手(FIN=1，seq=x)

假设客户端想要关闭连接，客户端发送一个 FIN 标志位置为 1 的包，表示自己已经没有数据可以发送了，但是仍然可以接受数据。

发送完毕后，客户端进入 FIN_WAIT_1 状态。

- 第二次挥手(ACK=1，ACKnum=x+1)

服务器端确认客户端的 FIN 包，发送一个确认包，表明自己接受到了客户端关闭连接的请求，但还没有准备好关闭连接。

发送完毕后，服务器端进入 CLOSE_WAIT 状态，客户端接收到这个确认包之后，进入 FIN_WAIT_2 状态，等待服务器端关闭连接。

- 第三次挥手(FIN=1，seq=y)

服务器端准备好关闭连接时，向客户端发送结束连接请求，FIN 置为 1。

发送完毕后，服务器端进入 LAST_ACK 状态，等待来自客户端的最后一个 ACK。

- 第四次挥手(ACK=1，ACKnum=y+1)

客户端接收到来自服务器端的关闭请求，发送一个确认包，并进入 TIME_WAIT 状态，等待可能出现的要求重传的 ACK 包。

服务器端接收到这个确认包之后，关闭连接，进入 CLOSED 状态。

客户端等待了某个固定时间（两个最大段生命周期，2MSL，2 Maximum Segment Lifetime）之后，没有收到服务器端的 ACK ，认为服务器端已经正常关闭连接，于是自己也关闭连接，进入 CLOSED 状态。

3.8.3.3 TCP 与 UDP 的区别

区别点	TCP	UDP
连接性	面向连接	无连接
可靠性	可靠	不可靠
有序性	有序	无序
面向	字节流	报文（保留报文的边界）
有界性	有界	无界
流量控制	有（滑动窗口）	无
拥塞控制	有（慢开始、拥塞避免、快重传、快恢复）	无
传输速度	慢	快
量级	重量级	轻量级
双工性	全双工	一对一、一对多、多对一、多对多
头部	大（20-60 字节）	小（8 字节）
应用	文件传输、邮件传输、浏览器等	即时通讯、视频通话等

3.8.4 Socket

Socket 是一组操作 TCP/UDP 的 API，像 HttpURLConnection 和 Okhttp 这种涉及到比较底层的网络请求发送的，最终当然也都是通过 Socket 来进行网络请求连接发送，而像 Volley、Retrofit 则是更上层的封装。

3.8.4.1 使用示例

使用 socket 的步骤如下：

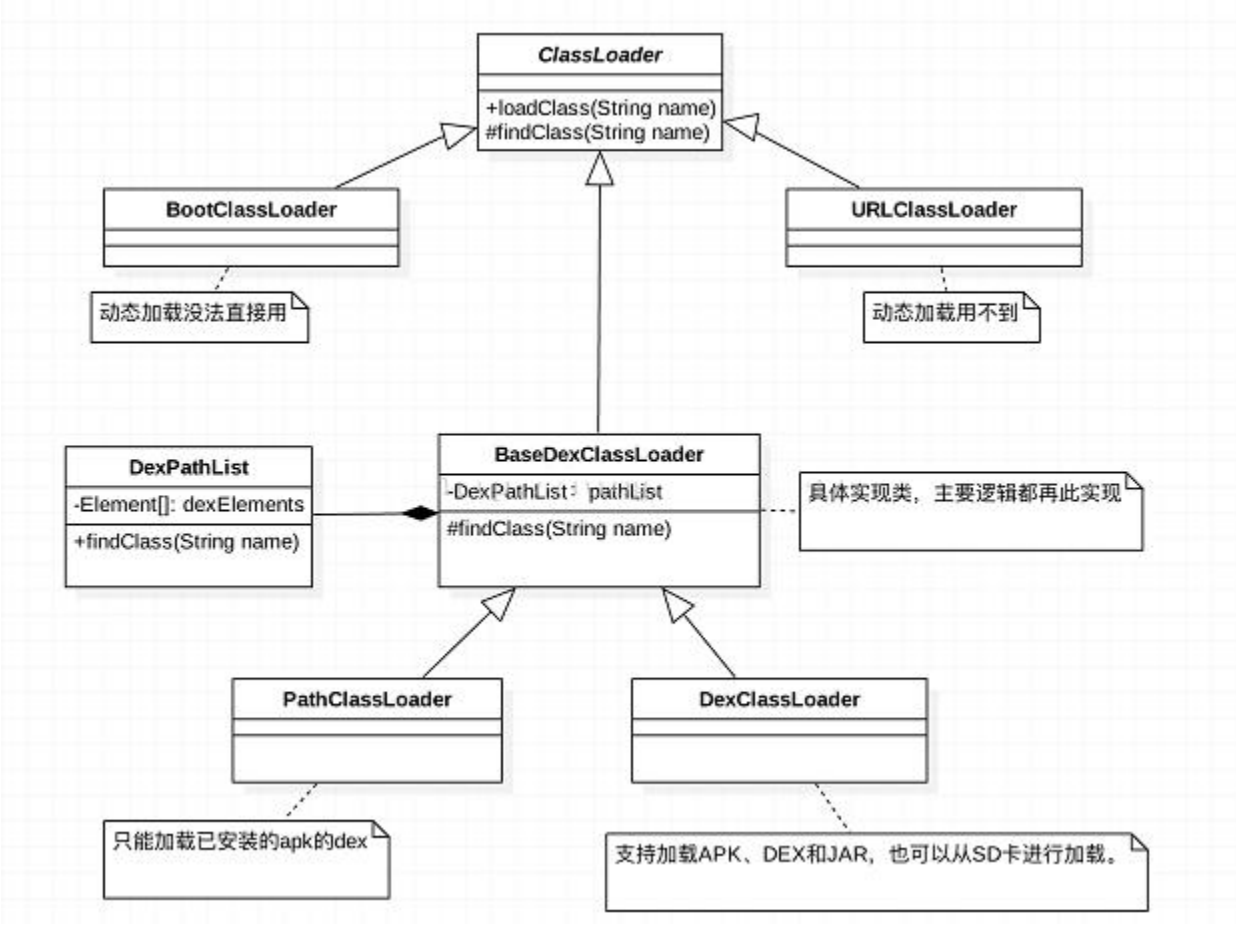
- 创建 ServerSocket 并监听客户连接；
- 使用 Socket 连接服务端；
- 通过 Socket.getInputStream()/getOutputStream() 获取输入输出流进行通信。

```
public class EchoClient {


    private final Socket mSocket;


    public EchoClient(String host, int port) throws IOException {
        // 创建 socket 并连接服务器
        mSocket = new Socket(host, port);
    }


    public void run() {
        // 和服务端进行通信
        Thread readerThread = new Thread(this::readResponse);
        readerThread.start();


        OutputStream out = mSocket.getOutputStream();
        byte[] buffer = new byte[1024];
        int n;
        while ((n = System.in.read(buffer)) > 0) {
            out.write(buffer, 0, n);
```

```
    }
}

private void readResponse() {
    try {
        InputStream in = mSocket.getInputStream();
        byte[] buffer = new byte[1024];
        int n;
        while ((n = in.read(buffer)) > 0) {
            System.out.write(buffer, 0, n);
        }
    } catch (IOException e) {
        e.printStackTrace();
    }
}


public static void main(String[] argv) {
    try {
        // 由于服务端运行在同一主机，这里我们使用 localhost
        EchoClient client = new EchoClient("localhost", 9877);
        client.run();
    } catch (IOException e) {
        e.printStackTrace();
    }
```

```
    }

}
```

3.9 类加载器



3.9.1 双亲委托模式

某个特定的类加载器在接到加载类的请求时，首先将加载任务委托给父类加载器，依次递归，如果父类加载器可以完成类加载任务，就成功返回；只有父类加载器无法完成此加载任务时，才自己去加载。

因为这样可以避免重复加载，当父亲已经加载了该类的时候，就没有必要子 ClassLoader 再加载一次。如果不使用这种委托模式，那我们就可以随时使用自定义的类来动态替代一些核心的类，存在非常大的安全隐患。

3.9.2 DexPathList

DexClassLoader 重载了 `findClass` 方法，在加载类时会调用其内部的 DexPathList 去加载。

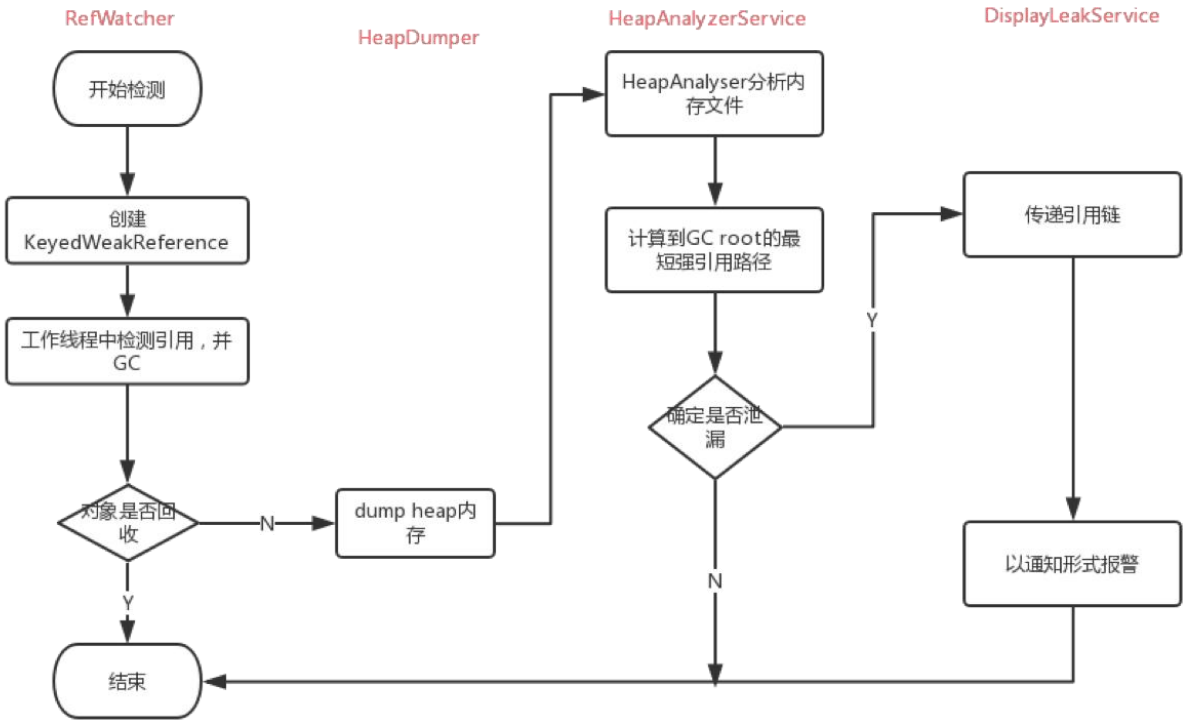
DexPathList 是在构造 DexClassLoader 时生成的，其内部包含了 DexFile。

DexPathList.java

```
...public Class findClass(String name) {
    for (Element element : dexElements) {
        DexFile dex = element.dexFile;
        if (dex != null) {
            Class clazz = dex.loadClassBinaryName(name, definingContext);
            if (clazz != null) {
                return clazz;
            }
        }
    }
    return null;
}
...
```

四、Android 开源库源码分析

4.1 LeakCanary



4.1.1 初始化注册

在清单文件中注册了一个 ContentProvider 用于在应用启动时初始化代码：

leakcanary-leaksentry/*/AndroidManifest.xml

```
...

    <application>

        <provider

            android:name="leakcanary.internal.LeakSentryInstaller"

            android:authorities="${applicationId}.leak-sentry-installer"
```

```
            android:exported="false"/>

    </application>

...
```

在 LeakSentryInstaller 生命周期 `onCreate()` 方法中完成初始化步骤：

LeakSentryInstaller.kt

```
internal class LeakSentryInstaller : ContentProvider() {



    override fun onCreate(): Boolean {

        CanaryLog.logger = DefaultCanaryLog()

        val application = context!!.applicationContext as Application

        InternalLeakSentry.install(application)

        return true

    }

...
```

然后分别注册 Activity/Fragment 的监听：

InternalLeakSentry.kt

```
...

    fun install(application: Application) {

        CanaryLog.d("Installing LeakSentry")

        checkMainThread()

        if (this::application.isInitialized) {

        return

        }

        InternalLeakSentry.application = application
```

```
        val configProvider = { LeakSentry.config }
        ActivityDestroyWatcher.install(
            application, refWatcher, configProvider
        )
        FragmentDestroyWatcher.install(
            application, refWatcher, configProvider
        )
        listener.onLeakSentryInstalled(application)
    }
...
```

ActivityDestroyWatcher.kt

```
...
    fun install(application: Application,refWatcher: RefWatcher,configProvider: () ->
Config
        ) {
            val activityDestroyWatcher = ActivityDestroyWatcher(refWatcher, configProvider)

application.registerActivityLifecycleCallbacks(activityDestroyWatcher.lifecycleCallbacks
)
        }
    }
...
```

AndroidOFragmentDestroyWatcher.kt

```
...
    override fun watchFragments(activity: Activity) {
```

```
        val fragmentManager = activity.fragmentManager
        fragmentManager.registerFragmentLifecycleCallbacks(fragmentLifecycleCallbacks,
true)
    }
...
```

AndroidXFragmentDestroyWatcher.kt

```
...
    override fun watchFragments(activity: Activity) {
        if (activity is FragmentActivity) {
        val supportFragmentManager = activity.supportFragmentManager

supportFragmentManager.registerFragmentLifecycleCallbacks(fragmentLifecycleCallbacks,
true)
        }
    }
...
```

4.1.2 引用泄漏观察

RefWatcher.kt

```
...
    @Synchronized fun watch(watchedInstance: Any, name: String) {
        if (!isEnabled()) {
            return
        }
        removeWeaklyReachableInstances()
        val key = UUID.randomUUID().toString()
```

```
        val watchUptimeMillis = clock.uptimeMillis()
        val reference = KeyedWeakReference(watchedInstance, key, name, watchUptimeMillis,
queue)
        CanaryLog.d(
            "Watching %s with key %s",
            ((if (watchedInstance is Class<*>) watchedInstance.toString() else "instance of
${watchedInstance.javaClass.name}") + if (name.isNotEmpty()) " named $name" else ""), key
        )

        watchedInstances[key] = reference
        checkRetainedExecutor.execute {
            moveToRetained(key)
        }
    }

    @Synchronized private fun moveToRetained(key: String) {
        removeWeaklyReachableInstances()
        val retainedRef = watchedInstances[key]
        if (retainedRef != null) {
            retainedRef.retainedUptimeMillis = clock.uptimeMillis()
            onInstanceRetained()
        }
    }
...
```

InternalLeakCanary.kt

```
...
```

```
    override fun onReferenceRetained() {
        if (this::heapDumpTrigger.isInitialized) {
            heapDumpTrigger.onReferenceRetained()
        }
    }
...
```

4.1.3 Dump Heap

发现泄漏之后，获取 Heamp Dump 相关文件：

AndroidHeapDumper.kt

```
...
    override fun dumpHeap(): File? {
        val heapDumpFile = leakDirectoryProvider.newHeapDumpFile() ?: return null
        ...
        return try {
            Debug.dumpHprofData(heapDumpFile.absolutePath)
            if (heapDumpFile.length() == 0L) {
                CanaryLog.d("Dumped heap file is 0 byte length")
                null
            } else {
                heapDumpFile
            }
        } catch (e: Exception) {
            CanaryLog.d(e, "Could not dump heap")
            // Abort heap dump
```

```
            null
        } finally {
            cancelToast(toast)
            notificationManager.cancel(R.id.leak_canary_notification_dumping_heap)
        }
    }
...
```

HeapDumpTrigger.kt

```
...
    private fun checkRetainedInstances(reason: String) {
        ...
        val heapDumpFile = heapDumper.dumpHeap()
        ...
        lastDisplayedRetainedInstanceCount = 0
        refWatcher.removeInstancesWatchedBeforeHeapDump(heapDumpUptimeMillis)

        HeapAnalyzerService.runAnalysis(application, heapDumpFile)
    }
...
```

启动一个 HeapAnalyzerService 来分析 heapDumpFile：

HeapAnalyzerService.kt

```
...
    override fun onHandleIntentInForeground(intent: Intent?) {
        ...
```

```
        val heapAnalyzer = HeapAnalyzer(this)

        val config = LeakCanary.config



        val heapAnalysis =

        heapAnalyzer.checkForLeaks(

            heapDumpFile, config.referenceMatchers, config.computeRetainedHeapSize,
config.objectInspectors,

            if (config.useExperimentalLeakFinders) config.objectInspectors else listOf(

                AndroidObjectInspectors.KEYED_WEAK_REFERENCE

            )

        )



        config.analysisResultListener(application, heapAnalysis)

    }

...
```

Heap Dump 之后，可以查看以下内容：

- 应用分配了哪些类型的对象，以及每种对象的数量。
- 每个对象使用多少内存。
- 代码中保存对每个对象的引用。
- 分配对象的调用堆栈。（调用堆栈当前仅在使用 Android 7.1 及以下时有效。）

4.2 EventBus

4.2.1 自定义注解

- 申明注解类

```
@Documented@Retention(RetentionPolicy.RUNTIME)@Target({ElementType.METHOD})public
@interface Subscribe {

    // 线程模式

    ThreadMode threadMode() default ThreadMode.POSTING;



    // 是否为粘性事件

    boolean sticky() default false;



    // 事件的优先级

    int priority() default 0;

}
```

- 注册订阅事件

```
@Subscribe(threadMode = ThreadMode.MAIN, priority = 1, sticky = true)public void
onEventMainThreadP1(IntTestEvent event) {

    handleEvent(1, event);

}
```

4.2.2 注册订阅者

```
EventBus.getDefault().register(object);
```

Eventbus.java

```
public void register(Object subscriber) {

    Class<?> subscriberClass = subscriber.getClass();

    List<SubscriberMethod> subscriberMethods =
subscriberMethodFinder.findSubscriberMethods(subscriberClass);

    synchronized (this) {

        for (SubscriberMethod subscriberMethod : subscriberMethods) {
```

```
            subscribe(subscriber, subscriberMethod);
        }
    }
}
```

- 通过反射查找订阅者类里的订阅事件，添加到 METHOD_CACHE 中

SubscriberMethodFinder.java

```
private static final Map<Class<?>, List<SubscriberMethod>> METHOD_CACHE = new
ConcurrentHashMap<>();
...
List<SubscriberMethod> findSubscriberMethods(Class<?> subscriberClass) {
    List<SubscriberMethod> subscriberMethods = METHOD_CACHE.get(subscriberClass);
    if (subscriberMethods != null) {
        return subscriberMethods;
    }

    if (ignoreGeneratedIndex) {
        subscriberMethods = findUsingReflection(subscriberClass);
    } else {
        subscriberMethods = findUsingInfo(subscriberClass);
    }
    if (subscriberMethods.isEmpty()) {
        throw new EventBusException("Subscriber " + subscriberClass
                + " and its super classes have no public methods with the @Subscribe
annotation");
    } else {
```

```
        METHOD_CACHE.put(subscriberClass, subscriberMethods);
        return subscriberMethods;
    }
}
...
private void findUsingReflectionInSingleClass(FindState findState) {
    Method[] methods;
    try {
        // This is faster than getMethods, especially when subscribers are fat classes like
Activities
        methods = findState.clazz.getDeclaredMethods();
    } catch (Throwable th) {
        // Workaround for java.lang.NoClassDefFoundError, see
https://github.com/greenrobot/EventBus/issues/149
        methods = findState.clazz.getMethods();
        findState.skipSuperClasses = true;
    }
    for (Method method : methods) {
        int modifiers = method.getModifiers();
        if ((modifiers & Modifier.PUBLIC) != 0 && (modifiers & MODIFIERS_IGNORE) == 0) {
            Class<?>[] parameterTypes = method.getParameterTypes();
            if (parameterTypes.length == 1) {
                Subscribe subscribeAnnotation = method.getAnnotation(Subscribe.class);
                if (subscribeAnnotation != null) {
                    Class<?> eventType = parameterTypes[0];
                    if (findState.checkAdd(method, eventType)) {
```

```
                    ThreadMode threadMode = subscribeAnnotation.threadMode();

                    findState.subscriberMethods.add(new SubscriberMethod(method,
eventType, threadMode,

                            subscribeAnnotation.priority(),
subscribeAnnotation.sticky()));

                }

            }

        } else if (strictMethodVerification &&
method.isAnnotationPresent(Subscribe.class)) {

            String methodName = method.getDeclaringClass().getName() + "." +
method.getName();

            throw new EventBusException("@Subscribe method " + methodName +

                    "must have exactly 1 parameter but has " + parameterTypes.length);

        }

    } else if (strictMethodVerification && method.isAnnotationPresent(Subscribe.class))
{

        String methodName = method.getDeclaringClass().getName() + "." +
method.getName();

        throw new EventBusException(methodName +

                " is a illegal @Subscribe method: must be public, non-static, and
non-abstract");

    }

}

}
```

4.2.3 发送事件

```
EventBus.getDefault().post(object);
```

- 根据事件类型获取到对应的订阅者信息

Subscription.java

```
final class Subscription {

    final Object subscriber;

    final SubscriberMethod subscriberMethod;

    ...

}
```

EventBus.java

```
private final Map<Class<?>, CopyOnWriteArrayList<Subscription>> subscriptionsByEventType;

...

private boolean postSingleEventForEventType(Object event, PostingThreadState postingState,
Class<?> eventClass) {

    CopyOnWriteArrayList<Subscription> subscriptions;

    synchronized (this) {

        subscriptions = subscriptionsByEventType.get(eventClass);

    }

    if (subscriptions != null && !subscriptions.isEmpty()) {

        for (Subscription subscription : subscriptions) {

            postingState.event = event;

            postingState.subscription = subscription;

            boolean aborted = false;

            try {

                postToSubscription(subscription, event, postingState.isMainThread);

                aborted = postingState.canceled;

            } finally {

                postingState.event = null;
```

```
                postingState.subscription = null;
                postingState.canceled = false;
            }
            if (aborted) {
                break;
            }
        }
        return true;
    }
    return false;
}
...
```

- 根据注册已获得的 Method 对象调用相关注册方法

EventBus.java

```
void invokeSubscriber(Subscription subscription, Object event) {
    try {
        subscription.subscriberMethod.method.invoke(subscription.subscriber, event);
    } catch (InvocationTargetException e) {
        handleSubscriberException(subscription, event, e.getCause());
    } catch (IllegalAccessException e) {
        throw new IllegalStateException("Unexpected exception", e);
    }
}
```

五、设计模式汇总

5.1 设计模式分类

模式 & 描述	包括
创建型模式 提供了一种在创建对象的同时隐藏创建逻辑的方式。	工厂模式（Factory Pattern） 抽象工厂模式（Abstract Factory Pattern） 单例模式（Singleton Pattern） 建造者模式（Builder Pattern） 原型模式（Prototype Pattern）
结构型模式 关注类和对象的组合。	适配器模式（Adapter Pattern） 桥接模式（Bridge Pattern） 过滤器模式（Filter、Criteria Pattern） 组合模式（Composite Pattern） 装饰器模式（Decorator Pattern） 外观模式（Facade Pattern） 享元模式（Flyweight Pattern） 代理模式（Proxy Pattern）
行为型模式 特别关注对象之间的通信。	责任链模式（Chain of Responsibility Pattern） 命令模式（Command Pattern）

模式 & 描述	包括
	解释器模式（Interpreter Pattern） 迭代器模式（Iterator Pattern） 中介者模式（Mediator Pattern） 备忘录模式（Memento Pattern） 观察者模式（Observer Pattern） 状态模式（State Pattern） 空对象模式（Null Object Pattern） 策略模式（Strategy Pattern） 模板模式（Template Pattern） 访问者模式（Visitor Pattern）

5.2 面向对象六大原则

原则	描述
单一职责原则	一个类只负责一个功能领域中的相应职责。
开闭原则	对象应该对于扩展是开放的，对于修改是封闭的。
里氏替换原则	所有引用基类的地方必须能透明地使用其子类的对象。
依赖倒置原则	高层模块不依赖低层模块，两者应该依赖其对象；抽象不应该依赖细节；细节应该依赖抽象。
接口隔离原则	类间的依赖关系应该建立在最小的接口上。
迪米特原则	也称最少知识原则，一个对象对其他对象有最少的了解。

5.3 工厂模式

适用于复杂对象的创建。

示例：

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.demo);
```

BitmapFactory.java

```
// 生成 Bitmap 对象的工厂类 BitmapFactorypublic class BitmapFactory {
    ...
    public static Bitmap decodeFile(String pathName) {
        ...
    }
    ...


    public static Bitmap decodeResource(Resources res, int id, Options opts) {
        validate(opts);
        Bitmap bm = null;
        InputStream is = null;


        try {
            final TypedValue value = new TypedValue();
            is = res.openRawResource(id, value);


            bm = decodeResourceStream(res, value, is, null, opts);
        }
        ...
```

```
        return bm;
    }
    ...
}
```

5.4 单例模式

确保某一个类只有一个实例，并自动实例化向整个系统提供这个实例，且可以避免产生多个对象消耗资源。

示例：

InputMethodManager.java

```
/*** Retrieve the global InputMethodManager instance, creating it if it* doesn't already exist.* @hide*/public static InputMethodManager getInstance() {
    synchronized (InputMethodManager.class) {
        if (sInstance == null) {
            try {
                sInstance = new InputMethodManager(Looper.getMainLooper());
            } catch (ServiceNotFoundException e) {
                throw new IllegalStateException(e);
            }
        }
        return sInstance;
    }
}
```

5.5 建造者模式

将一个复杂对象的构建与它的表示分离，使得同样的构建过程可以创建不同的表示，适用于初始化的对象比较复杂且参数较多的情况。

示例：

```
AlertDialog.Builder builder = new AlertDialog.Builder(this)

       .setTitle("Title")

       .setMessage("Message");AlertDialog dialog = builder.create();

dialog.show();
```

AlertDialog.java

```
public class AlertDialog extends Dialog implements DialogInterface {

    ...



    public static class Builder {

        private final AlertController.AlertParams P;

        ...



        public Builder(Context context) {

            this(context, resolveDialogTheme(context, ResourceId.ID_NULL));

        }

        ...



        public Builder setTitle(CharSequence title) {

            P.mTitle = title;

            return this;

        }

        ...
```

```
        public Builder setMessage(CharSequence message) {
            P.mMessage = message;
            return this;
        }
        ...

        public AlertDialog create() {
            // Context has already been wrapped with the appropriate theme.
            final AlertDialog dialog = new AlertDialog(P.mContext, 0, false);
            P.apply(dialog.mAlert);
            ...
            return dialog;
        }
        ...
    }
}
```

5.6 原型模式

用原型模式实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

示例：

```
ArrayList<T> newArrayList = (ArrayList<T>) arrayList.clone();
```

ArrayList.java

```
/** * Returns a shallow copy of this <tt>ArrayList</tt> instance.  (The * elements themselves are not copied.) * * @return a clone of this <tt>ArrayList</tt> instance */public Object clone()
{
```

```
    try {
        ArrayList<?> v = (ArrayList<?>) super.clone();
        v.elementData = Arrays.copyOf(elementData, size);
        v.modCount = 0;
        return v;
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
}
```

5.7 适配器模式

适配器模式把一个类的接口变成客户端所期待的另一种接口，从而使原因接口不匹配而无法一起工作的两个类能够在一起工作。

示例：

```
RecyclerView recyclerView = findViewById(R.id.recycler_view);
recyclerView.setAdapter(new MyAdapter());
private class MyAdapter extends RecyclerView.Adapter<RecyclerView.ViewHolder> {

    @NonNull
    @Override
    public RecyclerView.ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        ...
    }
```

```
    ...
}
```

RecyclerView.java

```
...private void setAdapterInternal(@Nullable Adapter adapter, boolean
compatibleWithPrevious,
        boolean removeAndRecycleViews) {
    if (mAdapter != null) {
        mAdapter.unregisterAdapterDataObserver(mObserver);
        mAdapter.onDetachedFromRecyclerView(this);
    }
    ...
    mAdapterHelper.reset();
    final Adapter oldAdapter = mAdapter;
    mAdapter = adapter;
    if (adapter != null) {
        adapter.registerAdapterDataObserver(mObserver);
        adapter.onAttachedToRecyclerView(this);
    }
    if (mLayout != null) {
        mLayout.onAdapterChanged(oldAdapter, mAdapter);
    }
    mRecycler.onAdapterChanged(oldAdapter, mAdapter, compatibleWithPrevious);
    mState.mStructureChanged = true;
}
```

```
···public final class Recycler {
    @Nullable
    ViewHolder tryGetViewHolderForPositionByDeadline(int position,
            boolean dryRun, long deadlineNs) {
        ...
        ViewHolder holder = null;
        ...
        if (holder == null) {
            ...
            holder = mAdapter.createViewHolder(RecyclerView.this, type);
            ...
        }
        ...
        return holder;
    }
}


···public abstract static class Adapter<VH extends ViewHolder> {
    ...
    @NonNull
    public abstract VH onCreateViewHolder(@NonNull ViewGroup parent, int viewType);


    @NonNull
    public final VH createViewHolder(@NonNull ViewGroup parent, int viewType) {
        try {
```

```
            TraceCompat.beginSection(TRACE_CREATE_VIEW_TAG);
            final VH holder = onCreateViewHolder(parent, viewType);
            ...
            holder.mItemViewType = viewType;
            return holder;
        } finally {
            TraceCompat.endSection();
        }
    }
    ...
}
```

5.8 观察者模式

定义对象间一种一对多的依赖关系，使得每当一个对象改变状态，则所有依赖于它的对象都会得到通知并被自动更新。

示例：

```
MyAdapter adapter = new MyAdapter();
recyclerView.setAdapter(adapter);
adapter.notifyDataSetChanged();
```

RecyclerView.java

```
···private final RecyclerViewDataObserver mObserver = new RecyclerViewDataObserver();

···private void setAdapterInternal(@Nullable Adapter adapter, boolean
compatibleWithPrevious,
        boolean removeAndRecycleViews) {
    ...
```

```
    mAdapter = adapter;
    if (adapter != null) {
        adapter.registerAdapterDataObserver(mObserver);
        adapter.onAttachedToRecyclerView(this);
    }
    ...
}

...public abstract static class Adapter<VH extends ViewHolder> {
    private final AdapterDataObservable mObservable = new AdapterDataObservable();
    ...
    public void registerAdapterDataObserver(@NonNull AdapterDataObserver observer) {
        mObservable.registerObserver(observer);
    }

    ...
    public final void notifyDataSetChanged() {
        mObservable.notifyChanged();
    }
}
static class AdapterDataObservable extends Observable<AdapterDataObserver> {
    ...
    public void notifyChanged() {
        for (int i = mObservers.size() - 1; i >= 0; i--) {
            mObservers.get(i).onChanged();
```

```
        }
    }
    ...
}
private class RecyclerViewDataObserver extends AdapterDataObserver {
    ...
    @Override
    public void onChanged() {
        assertNotInLayoutOrScroll(null);
        mState.mStructureChanged = true;

        processDataSetCompletelyChanged(true);
        if (!mAdapterHelper.hasPendingUpdates()) {
            requestLayout();
        }
    }
    ...
}
```

5.9 代理模式

为其他的对象提供一种代理以控制对这个对象的访问。适用于当无法或不想直接访问某个对象时通过一个代理对象来间接访问，为了保证客户端使用的透明性，委托对象与代理对象需要实现相同的接口。

示例：

Context.java

```
public abstract class Context {
```

```
    ...

    public abstract void startActivity(@RequiresPermission Intent intent);

    ...

}
```

ContextWrapper.java

```
public class ContextWrapper extends Context {

    Context mBase; // 代理类，实为 ContextImpl 对象

    ...



    protected void attachBaseContext(Context base) {

        if (mBase != null) {

            throw new IllegalStateException("Base context already set");

        }

        mBase = base;

    }

    ...



    @Override

    public void startActivity(Intent intent) {

        mBase.startActivity(intent); // 核心工作交由给代理类对象 mBase 实现

    }

    ...

}
```

ContextImpl.java

```
// Context 的真正实现类class ContextImpl extends Context {
```

```
    ...
    @Override
    public void startActivity(Intent intent) {
        warnIfCallingFromSystemProcess();
        startActivity(intent, null);
    }
    ...
}
```

5.10 责任链模式

使多个对象都有机会处理请求，从而避免了请求的发送者和接受者之间的耦合。将这些对象连成一条链，并沿着这条链传递该请求，直到有对象处理它为止。

ViewGroup.java

```
@UiThreadpublic abstract class ViewGroup extends View implements ViewParent, ViewManager {
    ...
    private boolean dispatchTransformedTouchEvent(MotionEvent event, boolean cancel,
            View child, int desiredPointerIdBits) {
        final boolean handled;
        ...
        final MotionEvent transformedEvent;
        if (newPointerIdBits == oldPointerIdBits) {
            if (child == null || child.hasIdentityMatrix()) {
                if (child == null) {
                    handled = super.dispatchTouchEvent(event);
                } else {
                    ...
```

```
                    // 获取子 view 处理的结果
                    handled = child.dispatchTouchEvent(event);
                }
                return handled;
            }
            transformedEvent = MotionEvent.obtain(event);
        } else {
            transformedEvent = event.split(newPointerIdBits);
        }

        // Perform any necessary transformations and dispatch.
        if (child == null) {
            handled = super.dispatchTouchEvent(transformedEvent);
        } else {
            ...
            // 获取子 view 处理的结果
            handled = child.dispatchTouchEvent(transformedEvent);
        }
        ...
        return handled;
    }
    ...
}
```

5.11 策略模式

策略模式定义了一系列的算法，并封装起来，提供针对同一类型问题的多种处理方式。

示例：

```
// 匀速

animation.setInterpolator(new LinearInterpolator());// 加速

animation.setInterpolator(new AccelerateInterpolator());

...
```

BaseInterpolator.java

```
/** * An abstract class which is extended by default interpolators. */abstract public class BaseInterpolator implements Interpolator {

    private @Config int mChangingConfiguration;

    /**     * @hide     */

    public @Config int getChangingConfiguration() {

        return mChangingConfiguration;

    }



    /**     * @hide     */

    void setChangingConfiguration(@Config int changingConfiguration) {

        mChangingConfiguration = changingConfiguration;

    }

}
```

LinearInterpolator.java

```
@HasNativeInterpolatorpublic class LinearInterpolator extends BaseInterpolator implements NativeInterpolatorFactory {

    ...

}
```

AccelerateInterpolator.java

```
@HasNativeInterpolatorpublic class AccelerateInterpolator extends BaseInterpolator implements NativeInterpolatorFactory {

    ...

}
```

5.12 备忘录模式

在不破坏封闭的前提下，在对象之外保存保存对象的当前状态，并且在之后可以恢复到此状态。

示例：

Activity.java

```
// 保存状态 protected void onSaveInstanceState(Bundle outState) {

    // 存储当前窗口的视图树的状态

    outState.putBundle(WINDOW_HIERARCHY_TAG, mWindow.saveHierarchyState());



    outState.putInt(LAST_AUTOFILL_ID, mLastAutofillId);

    // 存储 Fragment 的状态

    Parcelable p = mFragments.saveAllState();

    if (p != null) {

        outState.putParcelable(FRAGMENTS_TAG, p);

    }

    if (mAutoFillResetNeeded) {

        outState.putBoolean(AUTOFILL_RESET_NEEDED, true);

        getAutofillManager().onSaveInstanceState(outState);

    }

    // 调用 ActivityLifecycleCallbacks 的 onSaveInstanceState 进行存储状态

    getApplication().dispatchActivitySaveInstanceState(this, outState);
```

```
}


...// onCreate 方法中恢复状态 protected void onCreate(@Nullable Bundle savedInstanceState) {
    ...
    if (savedInstanceState != null) {
        mAutoFillResetNeeded = savedInstanceState.getBoolean(AUTOFILL_RESET_NEEDED,
false);
        mLastAutofillId = savedInstanceState.getInt(LAST_AUTOFILL_ID,
                View.LAST_APP_AUTOFILL_ID);

        if (mAutoFillResetNeeded) {
            getAutofillManager().onCreate(savedInstanceState);
        }

        Parcelable p = savedInstanceState.getParcelable(FRAGMENTS_TAG);
        mFragments.restoreAllState(p, mLastNonConfigurationInstances != null
                ? mLastNonConfigurationInstances.fragments : null);
    }
    mFragments.dispatchCreate();
    getApplication().dispatchActivityCreated(this, savedInstanceState);
    ...
    mRestoredFromBundle = savedInstanceState != null;
    mCalled = true;
}
```

ActivityThread.java

```
@Overridepublic void handleStartActivity(ActivityClientRecord r,
        PendingTransactionActions pendingActions) {
    final Activity activity = r.activity;
    ...
    // Start
    activity.performStart("handleStartActivity");
    r.setState(ON_START);
    ...
    // Restore instance state
    if (pendingActions.shouldRestoreInstanceState()) {
        if (r.isPersistable()) {
            if (r.state != null || r.persistentState != null) {
                mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state,
                        r.persistentState);
            }
        } else if (r.state != null) {
            mInstrumentation.callActivityOnRestoreInstanceState(activity, r.state);
        }
    }
    ...
}
```

六、Gradle 知识点汇总

6.1 依赖项配置

配置	说明
implementation	Gradle 会将依赖项添加到编译类路径，并将依赖项打包到编译输出。 不过，当模块配置 implementation 依赖项时，其他模块只有在运行时才能使用该依赖项。
api	Gradle 会将依赖项添加到编译类路径和编译输出。当一个模块包含 api 依赖项时，会让 Gradle 了解该模块要以传递方式将该依赖项导出到其他模块，以便这些模块在运行时和编译时都可以使用该依赖项。
compileOnly	Gradle 只会将依赖项添加到编译类路径（也就是说，不会将其添加到编译输出）。
runtimeOnly	Gradle 只会将依赖项添加到编译输出，以便在运行时使用。也就是说，不会将其添加到编译类路径。
annotationProcessor	要添加对作为注解处理器的库的依赖关系，必须使用 annotationProcessor 配置将其添加到注解处理器类路径。

七、常见面试算法题汇总

7.1 排序

7.1.1 比较排序

7.1.1.1 冒泡排序

重复地走访过要排序的数列，每次比较相邻两个元素，如果它们的顺序错误就把它们交换过来，越大的元素会经由交换慢慢“浮”到数列的尾端。

```
public void bubbleSort(int[] arr) {
    int temp = 0;
    boolean swap;
    for (int i = arr.length - 1; i > 0; i--) { // 每次需要排序的长度
        // 增加一个 swap 的标志，当前一轮没有进行交换时，说明数组已经有序
        swap = false;
        for (int j = 0; j < i; j++) { // 从第一个元素到第 i 个元素
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swap = true;
            }
        }
        if (!swap){
            break;
```

```
        }
    }
}
```

7.1.1.2 归并排序

分解待排序的数组成两个各具 n/2 个元素的子数组，递归调用归并排序两个子数组，合并两个已排序的子数组成一个已排序的数组。

```
public void mergeSort(int[] arr) {
    int[] temp = new int[arr.length];
    internalMergeSort(arr, temp, 0, arr.length - 1);
}
private void internalMergeSort(int[] arr, int[] temp, int left, int right) {
    // 当 left == right 时，不需要再划分
    if (left < right) {
        int mid = (left + right) / 2;
        internalMergeSort(arr, temp, left, mid);
        internalMergeSort(arr, temp, mid + 1, right);
        mergeSortedArray(arr, temp, left, mid, right);
    }
}
// 合并两个有序子序列 public void mergeSortedArray(int[] arr, int[] temp, int left, int mid,
int right) {
    int i = left;
    int j = mid + 1;
    int k = 0;
    while (i <= mid && j <= right) {
```

```
        temp[k++] = arr[i] < arr[j] ? arr[i++] : arr[j++];
    }
    while (i <= mid) {
        temp[k++] = arr[i++];
    }
    while (j <= right) {
        temp[k++] = arr[j++];
    }
    // 把 temp 数据复制回原数组
    for (i = 0; i < k; i++) {
        arr[left + i] = temp[i];
    }
}
```

7.1.1.3 快速排序

在待排序的数组选取一个元素作为基准，将待排序的元素进行分区，比基准元素大的元素放在一边，比其小的放另一边，递归调用快速排序对两边的元素排序。选取基准元素并分区的过程采用双指针左右交换。

```
public void quickSort(int[] arr){
    quickSort(arr, 0, arr.length-1);
}
private void quickSort(int[] arr, int low, int high){
    if (low >= high)
        return;
    int pivot = partition(arr, low, high);        //将数组分为两部分
    quickSort(arr, low, pivot - 1);                  //递归排序左子数组
```

```
    quickSort(arr, pivot + 1, high);                    //递归排序右子数组
}
private int partition(int[] arr, int low, int high){
    int pivot = arr[low];     //基准
    while (low < high){
        while (low < high && arr[high] >= pivot) {
            high--;
        }
        arr[low] = arr[high];             //交换比基准大的记录到左端
        while (low < high && arr[low] <= pivot) {
            low++;
        }
        arr[high] = arr[low];           //交换比基准小的记录到右端
    }
    //扫描完成，基准到位
    arr[low] = pivot;
    //返回的是基准的位置
    return low;
}
```

7.1.2 线性排序

7.1.2.1 计数排序

根据待排序的数组中最大和最小的元素，统计数组中每个值为 i 的元素出现的次数，存入数组 C 的第 i 项，对所有的计数累加，然后反向填充目标数组。

```java
public void countSort(int[] arr) {
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for(int i = 0; i < arr.length; i++){
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
    }

    int[] b = new int[arr.length]; // 存储数组
    int[] count = new int[max - min + 1]; // 计数数组

    for (int num = min; num <= max; num++) {
        // 初始化各元素值为 0，数组下标从 0 开始因此减 min
        count[num - min] = 0;
    }

    for (int i = 0; i < arr.length; i++) {
        int num = arr[i];
        count[num - min]++; // 每出现一个值，计数数组对应元素的值+1
        // 此时 count[i]表示数值等于 i 的元素的个数
    }

    for (int i = min + 1; i <= max; i++) {
        count[i - min] += count[i - min - 1];
        // 此时 count[i]表示数值<=i 的元素的个数
```

```
    }


    for (int i = 0; i < arr.length; i++) {
            int num = arr[i]; // 原数组第 i 位的值
            int index = count[num - min] - 1; //加总数组中对应元素的下标
            b[index] = num; // 将该值存入存储数组对应下标中
            count[num - min]--; // 加总数组中，该值的总和减少 1。
    }


    // 将存储数组的值替换给原数组
    for(int i=0; i < arr.length;i++){
        arr[i] = b[i];
    }
}
```

7.1.2.2 桶排序

找出待排序数组中的最大值 max、最小值 min，数组 ArrayList 作为桶，桶里放的元素用 ArrayList 存储。计算每个元素 arr[i] 放的桶，每个桶各自排序，遍历桶数组，把排序好的元素放进输出数组。

```
public static void bucketSort(int[] arr){
    int max = Integer.MIN_VALUE;
    int min = Integer.MAX_VALUE;
    for(int i = 0; i < arr.length; i++){
        max = Math.max(max, arr[i]);
        min = Math.min(min, arr[i]);
```

```
    }
    // 桶数
    int bucketNum = (max - min) / arr.length + 1;
    ArrayList<ArrayList<Integer>> bucketArr = new ArrayList<>(bucketNum);
    for(int i = 0; i < bucketNum; i++){
        bucketArr.add(new ArrayList<Integer>());
    }
    // 将每个元素放入桶
    for(int i = 0; i < arr.length; i++){
        int num = (arr[i] - min) / (arr.length);
        bucketArr.get(num).add(arr[i]);
    }
    // 对每个桶进行排序
    for(int i = 0; i < bucketArr.size(); i++){
        Collections.sort(bucketArr.get(i));
        for (int j = 0; j < bucketArr.get(i).size(); j++) {
            arr[j] = bucketArr.get(i).get(j);
        }
    }
}
```

7.2 二叉树

```
class TreeNode {
    public TreeNode left, right;
    public int val;
```

```
    public TreeNode(int val) {
        this.val = val;
    }
}
```

7.2.1 顺序遍历

先序遍历: 根->左->右

中序遍历: 左->根->右

后序遍历: 左->右->根

```
// 先序遍历 public void preTraverse(TreeNode root) {
    if (root != null) {
        System.out.println(root.val);
        preTraverse(root.left);
        preTraverse(root.right);
    }
}
// 中序遍历 public void inTraverse(TreeNode root) {
    if (root != null) {
        inTraverse(root.left);
        System.out.println(root.val);
        inTraverse(root.right);
    }
}
```

```
// 后序遍历 public void postTraverse(TreeNode root) {
    if (root != null) {
        postTraverse(root.left);
        postTraverse(root.right);
        System.out.println(root.val);
    }
}
```

7.2.2 层次遍历

```
// 层次遍历(DFS)public static List<List<Integer>> levelOrder(TreeNode root) {
    List<List<Integer>> res = new ArrayList<>();
    if (root == null) {
        return res;
    }

    dfs(root, res, 0);
    return res;
}
private void dfs(TreeNode root, List<List<Integer>> res, int level) {
    if (root == null) {
        return;
    }
    if (level == res.size()) {
        res.add(new ArrayList<>());
    }
```

```
    res.get(level).add(root.val);


    dfs(root.left, res, level + 1);
    dfs(root.right, res, level + 1);
}
// 层次遍历(BFS)public List<List<Integer>> levelOrder(TreeNode root) {
    List result = new ArrayList();


    if (root == null) {
        return result;
    }


    Queue<TreeNode> queue = new LinkedList<TreeNode>();
    queue.offer(root);


    while (!queue.isEmpty()) {
        ArrayList<Integer> level = new ArrayList<Integer>();
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            TreeNode head = queue.poll();
            level.add(head.val);
            if (head.left != null) {
                queue.offer(head.left);
            }
            if (head.right != null) {
```

```
                queue.offer(head.right);
            }
        }
        result.add(level);
    }

    return result;
}
// "Z"字遍历 public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();

    if (root == null){
        return result;
    }

    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    boolean isFromLeft = false;
    while(!queue.isEmpty()){
        int size = queue.size();
        isFromLeft = !isFromLeft;
        List<Integer> list = new ArrayList<>();
        for(int i = 0; i < size; i++){
            TreeNode node;
            if (isFromLeft){
```

```
            node = queue.pollFirst();
        }else{
            node = queue.pollLast();
        }
        list.add(node.val);

        if (isFromLeft){
            if (node.left != null){
                queue.offerLast(node.left);
            }
            if (node.right != null){
                queue.offerLast(node.right);
            }
        }else{
            if (node.right != null){
                queue.offerFirst(node.right);
            }
            if (node.left != null){
                queue.offerFirst(node.left);
            }
        }
    }
    result.add(list);
}
```

```
    return result;

}
```

7.2.3 左右翻转

```
public void invert(TreeNode root) {

    if (root == null) {

        return;

    }

    TreeNode temp = root.left;

    root.left = root.right;

    root.right = temp;



    invert(root.left);

    invert(root.right);

}
```

7.2.4 最大值

```
public int getMax(TreeNode root) {

    if (root == null) {

        return Integer.MIN_VALUE;

    } else {

        int left = getMax(root.left);

        int right = getMax(root.right);

        return Math.max(Math.max(left, rigth), root.val);

    }
```

```
}
```

7.2.5 最大深度

```
public int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int left = maxDepth(root.left);
    int right = maxDepth(root.right);
    return Math.max(left, right) + 1;
}
```

7.2.6 最小深度

```
public int minDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int left = minDepth(root.left);
    int right = minDepth(root.right);

    if (left == 0) {
        return right + 1;
    } else if (right == 0) {
```

```
        return left + 1;
    } else {
        return Math.min(left, right) + 1;
    }
}
```

7.2.7 平衡二叉树

平衡二叉树每一个节点的左右两个子树的高度差不超过 1

```
public boolean isBalanced(TreeNode root) {
    return maxDepth(root) != -1;
}
private int maxDepth(TreeNode root) {
    if (root == null) {
        return 0;
    }

    int left = maxDepth(root.left);
    int right = maxDepth(root.right);
    if (left == -1 || right == -1 || Math.abs(left - right) > 1) {
        return -1;
    }
    return Math.max(left, right) + 1;
}
```

7.3 链表

```
public class ListNode {
     int val;
     ListNode next;
     ListNode(int x) {
         val = x;
         next = null;
     }
}
```

7.3.1 删除节点

```
public void deleteNode(ListNode node) {
    if (node.next == null){
        node = null;
        return;
    }
    // 取缔下一节点
    node.val = node.next.val
    node.next = node.next.next
}
```

7.3.2 翻转链表

```
public ListNode reverse(ListNode head) {
    //prev 表示前继节点
    ListNode prev = null;
    while (head != null) {
```

```
        //temp 记录下一个节点，head 是当前节点
        ListNode temp = head.next;
        head.next = prev;
        prev = head;
        head = temp;
    }
    return prev;
}
```

7.3.3 中间元素

```
public ListNode findMiddle(ListNode head){
    if(head == null){
        return null;
    }

    ListNode slow = head;
    ListNode fast = head;

    // fast.next = null 表示 fast 是链表的尾节点
    while(fast != null && fast.next != null){
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow;
}
```

7.3.4 判断是否为循环链表

```
public Boolean hasCycle(ListNode head) {
    if (head == null || head.next == null) {
        return false;
    }

    ListNode slow = head;
    ListNode fast = head.next;

    while (fast != slow) {
        if(fast == null || fast.next == null) {
            return false;
        }
        fast = fast.next.next;
        slow = slow.next;
    }
    return true;
}
```

7.3.5 合并两个已排序链表

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode lastNode = dummy;
```

```
    while (l1 != null && l2 != null) {
        if (l1.val < l2.val) {
            lastNode.next = l1;
            l1 = l1.next;
        } else {
            lastNode.next = l2;
            l2 = l2.next;
        }
        lastNode = lastNode.next;
    }

    if (l1 != null) {
        lastNode.next = l1;
    } else {
        lastNode.next = l2;
    }

    return dummy.next;
}
```

7.3.6 链表排序

可利用归并、快排等算法实现

```
// 归并排序 public ListNode sortList(ListNode head) {
    if (head == null || head.next == null) {
        return head;
```

```
    }

    ListNode mid = findMiddle(head);

    ListNode right = sortList(mid.next);
    mid.next = null;
    ListNode left = sortList(head);

    return mergeTwoLists(left, right);
}
// 快速排序 public ListNode sortList(ListNode head) {
    quickSort(head, null);
    return head;
}
private void quickSort(ListNode start, ListNode end) {
    if (start == end) {
        return;
    }

    ListNode pt = partition(start, end);
    quickSort(start, pt);
    quickSort(pt.next, end);
}
private ListNode partition(ListNode start, ListNode end) {
    int pivotKey = start.val;
```

```
    ListNode p1 = start, p2 = start.next;
    while (p2 != end) {
        if (p2.val < pivotKey) {
            p1 = p1.next;
            swapValue(p1, p2);
        }
        p2 = p2.next;
    }

    swapValue(start, p1);
    return p1;
}
private void swapValue(ListNode node1, ListNode node2) {
    int tmp = node1.val;
    node1.val = node2.val;
    node2.val = tmp;
}
```

7.3.7 删除倒数第 N 个节点

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    if (n <= 0) {
        return null;
    }

    ListNode dummy = new ListNode(0);
```

```
    dummy.next = head;


    ListNode preDelete = dummy;
    for (int i = 0; i < n; i++) {
        if (head == null) {
            return null;
        }
        head = head.next;
    }
    // 此时 head 为正数第 N 个节点
    while (head != null) {
        head = head.next;
        preDelete = preDelete.next;
    }
    preDelete.next = preDelete.next.next;
    return dummy.next;
}
```

7.3.8 两个链表是否相交

```
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    if (headA == null || headB == null) {
        return null;
    }


    ListNode currA = headA;
```

```
ListNode currB = headB;
int lengthA = 0;
int lengthB = 0;

// 让长的先走到剩余长度和短的一样
while (currA != null) {
    currA = currA.next;
    lengthA++;
}
while (currB != null) {
    currB = currB.next;
    lengthB++;
}

currA = headA;
currB = headB;
while (lengthA > lengthB) {
    currA = currA.next;
    lengthA--;
}
while (lengthB > lengthA) {
    currB = currB.next;
    lengthB--;
}
```

```
    // 然后同时走到第一个相同的地方
    while (currA != currB) {
        currA = currA.next;
        currB = currB.next;
    }
    // 返回交叉开始的节点
    return currA;
}
```

7.4 栈 / 队列

7.4.1 带最小值操作的栈

实现一个栈，额外支持一个操作：min() 返回栈中元素的最小值

```
public class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack; // 维护一个辅助栈，传入当前栈的最小值

    public MinStack() {
        stack = new Stack<Integer>();
        minStack = new Stack<Integer>();
    }

    public void push(int number) {
        stack.push(number);
        if (minStack.isEmpty()) {
```

```
            minStack.push(number);
        } else {
            minStack.push(Math.min(number, minStack.peek()));
        }
    }

    public int pop() {
        minStack.pop();
        return stack.pop();
    }

    public int min() {
        return minStack.peek();
    }
}
```

7.4.2 有效括号

给定一个字符串所表示的括号序列，包含以下字符： '(', ')', '{', '}', '[' and ']'， 判定是否是有效的括号序列。括号必须依照 "()" 顺序表示， "()[]{}" 是有效的括号，但 "([)]" 则是无效的括号。

```
public boolean isValidParentheses(String s) {
    Stack<Character> stack = new Stack<Character>();
    for (Character c : s.toCharArray()) {
        if ("({[".contains(String.valueOf(c))) {
            stack.push(c);
```

```
        } else {
            if (!stack.isEmpty() && isValid(stack.peek(), c)) {
                stack.pop();
            } else {
                return false;
            }
        }
    }
    return stack.isEmpty();
}
private boolean isValid(char c1, char c2) {
    return (c1 == '(' && c2 == ')') || (c1 == '{' && c2 == '}')
        || (c1 == '[' && c2 == ']');
}
```

7.4.3 用栈实现队列

```
public class MyQueue {
    private Stack<Integer> outStack;
    private Stack<Integer> inStack;

    public MyQueue() {
       outStack = new Stack<Integer>();
       inStack = new Stack<Integer>();
    }
```

```
    private void in2OutStack(){
        while(!inStack.isEmpty()){
            outStack.push(inStack.pop());
        }
    }

    public void push(int element) {
        inStack.push(element);
    }

    public int pop() {
        if(outStack.isEmpty()){
            this.in2OutStack();
        }
        return outStack.pop();
    }

    public int top() {
        if(outStack.isEmpty()){
            this.in2OutStack();
        }
        return outStack.peek();
    }
}
```

7.4.4 逆波兰表达式求值

在反向波兰表示法中计算算术表达式的值, ["2", "1", "+", "3", "*"] -> (2 + 1) * 3 -> 9

```
public int evalRPN(String[] tokens) {
    Stack<Integer> s = new Stack<Integer>();
    String operators = "+-*/";
    for (String token : tokens) {
        if (!operators.contains(token)) {
            s.push(Integer.valueOf(token));
            continue;
        }

        int a = s.pop();
        int b = s.pop();
        if (token.equals("+")) {
            s.push(b + a);
        } else if(token.equals("-")) {
            s.push(b - a);
        } else if(token.equals("*")) {
            s.push(b * a);
        } else {
            s.push(b / a);
        }
    }
```

```
    return s.pop();

}
```

7.5 二分

7.5.1 二分搜索

```
public int binarySearch(int[] arr, int start, int end, int hkey){

    if (start > end) {

        return -1;

    }



    int mid = start + (end - start) / 2;    //防止溢位

    if (arr[mid] > hkey) {

        return binarySearch(arr, start, mid - 1, hkey);

    }

    if (arr[mid] < hkey) {

        return binarySearch(arr, mid + 1, end, hkey);

    }

    return mid;

}
```

7.5.2 X 的平方根

```
public int sqrt(int x) {

    if (x < 0)  {

        throw new IllegalArgumentException();
```

```
    } else if (x <= 1) {
        return x;
    }

    int start = 1, end = x;
    // 直接对答案可能存在的区间进行二分 => 二分答案
    while (start + 1 < end) {
        int mid = start + (end - start) / 2;
        if (mid == x / mid) {
            return mid;
        }  else if (mid < x / mid) {
            start = mid;
        } else {
            end = mid;
        }
    }
    if (end > x / end) {
        return start;
    }
    return end;
}
```

7.6 哈希表

7.6.1 两数之和

> 给一个整数数组，找到两个数使得他们的和等于一个给定的数 target。需要实现的函数 twoSum 需要返回这两个数的下标。

用一个 hashmap 来记录，key 记录 target-numbers[i]的值，value 记录 numbers[i]的 i 的值，如果碰到一个 numbers[j]在 hashmap 中存在，那么说明前面的某个 numbers[i]和 numbers[j]的和为 target，i 和 j 即为答案

```
public int[] twoSum(int[] numbers, int target) {


    HashMap<Integer,Integer> map = new HashMap<>();


    for (int i = 0; i < numbers.length; i++) {
        if (map.containsKey(numbers[i])) {
            return new int[]{map.get(numbers[i]), i};
        }
        map.put(target - numbers[i], i);
    }


    return new int[]{};
}
```

7.6.2 连续数组

> 给一个二进制数组，找到 0 和 1 数量相等的子数组的最大长度

使用一个数字 sum 维护到 i 为止 1 的数量与 0 的数量的差值。在 loop i 的同时维护 sum 并将其插入 hashmap 中。对于某一个 sum 值，若 hashmap 中已有这个值，则当前的 i 与 sum 上一次出现的位置之间的序列 0 的数量与 1 的数量相同。

```
public int findMaxLength(int[] nums) {
    Map<Integer, Integer> prefix = new HashMap<>();
    int sum = 0;
    int max = 0;
    prefix.put(0, -1); // 当第一个 0 1 数量相等的情况出现时，数组下标减去-1 得到正确的长度
    for (int i = 0; i < nums.length; i++) {
        int num = nums[i];
        if (num == 0) {
            sum--;
        } else {
            sum++;
        }

        if (prefix.containsKey(sum)) {
            max = Math.max(max, i - prefix.get(sum));
        } else {
            prefix.put(sum, i);
        }
    }

    return max;
```

```
}
```

7.6.3 最长无重复字符的子串

用 HashMap 记录每一个字母出现的位置。设定一个左边界，到当前枚举到的位置之间的字符串为不含重复字符的子串。若新碰到的字符的上一次的位置在左边界右边，则需要向右移动左边界

```
public int lengthOfLongestSubstring(String s) {
    if (s == null || s.length() == 0) {
        return 0;
    }
    HashMap<Character, Integer> map = new HashMap<>();
    int max = Integer.MIN_VALUE;
    int start = -1; // 计算无重复字符子串开始的位置
    int current = 0;
    for (int i = 0; i < s.length(); i++) {
        if (map.containsKey(s.charAt(i))) {
            int tmp = map.get(s.charAt(i));
            if (tmp >= start) { // 上一次的位置在左边界右边，则需要向右移动左边界
                start = tmp;
            }
        }

        map.put(s.charAt(i), i);
        max = Math.max(max, i - start);
    }
```

```
    return max;

}
```

7.6.4 最多点在一条直线上

给出二维平面上的 n 个点，求最多有多少点在同一条直线上

```
class Point {

    int x;

    int y;

    Point() {

        x = 0; y = 0;

    }

    Point(int a, int b) {

        x = a; y = b;

    }

}
```

通过 HashMap 记录下两个点之间的斜率相同出现的次数，注意考虑点重合的情况

```
public int maxPoints(Point[] points) {

    if (points == null) {

        return 0;

    }



    int max = 0;

    for (int i = 0; i < points.length; i++) {

        Map<Double, Integer> map = new HashMap<>();
```

```
        int maxPoints = 0;
        int overlap = 0;
        for (int j = i + 1; j < points.length; j++) {
            if (points[i].x == points[j].x && points[i].y == points[j].y) {
                overlap++; // 两个点重合的情况记录下来
                continue;
            }
            double rate = (double)(points[i].y - points[j].y) / (points[i].x - points[j].x);
            if (map.containsKey(rate)) {
                map.put(rate, map.get(rate) + 1);
            } else {
                map.put(rate, 2);
            }
            maxPoints = Math.max(maxPoints, map.get(rate));
        }
        if (maxPoints == 0) maxPoints = 1;
        max = Math.max(max, maxPoints + overlap);
    }
    return max;
}
```

7.7 堆 / 优先队列

7.7.1 前 K 大的数

```
// 维护一个 PriorityQueue，以返回前 K 的数 public int[] topk(int[] nums, int k) {
```

```
    int[] result = new int[k];
    if (nums == null || nums.length < k) {
        return result;
    }

    Queue<Integer> pq = new PriorityQueue<>();
    for (int num : nums) {
        pq.add(num);
        if (pq.size() > k) {
            pq.poll();
        }
    }

    for (int i = k - 1; i >= 0; i--) {
        result[i] = pq.poll();
    }

    return result;
}
```

7.7.2 前 K 大的数 II

实现一个数据结构，提供下面两个接口：1.add(number) 添加一个元素 2.topk() 返回前 K 大的数

```
public class Solution {
    private int maxSize;
```

```
private Queue<Integer> minheap;

public Solution(int k) {

    minheap = new PriorityQueue<>();

    maxSize = k;

}


public void add(int num) {

    if (minheap.size() < maxSize) {

        minheap.offer(num);

        return;

    }


    if (num > minheap.peek()) {

        minheap.poll();

        minheap.offer(num);

    }

}


public List<Integer> topk() {

    Iterator it = minheap.iterator();

    List<Integer> result = new ArrayList<Integer>();

    while (it.hasNext()) {

        result.add((Integer) it.next());

    }

    Collections.sort(result, Collections.reverseOrder());
```

```
        return result;
    }
}
```

7.7.3 第 K 大的数

```
public int kthLargestElement(int k, int[] nums) {
    if (nums == null || nums.length == 0 || k < 1 || k > nums.length){
        return -1;
    }
    return partition(nums, 0, nums.length - 1, nums.length - k);
}
private int partition(int[] nums, int start, int end, int k) {
    if (start >= end) {
        return nums[k];
    }

    int left = start, right = end;
    int pivot = nums[(start + end) / 2];

    while (left <= right) {
        while (left <= right && nums[left] < pivot) {
            left++;
        }
        while (left <= right && nums[right] > pivot) {
            right--;
```

```
        }
        if (left <= right) {
            swap(nums, left, right);
            left++;
            right--;
        }
    }

    if (k <= right) {
        return partition(nums, start, right, k);
    }
    if (k >= left) {
        return partition(nums, left, end, k);
    }
    return nums[k];
}
private void swap(int[] nums, int i, int j) {
    int tmp = nums[i];
    nums[i] = nums[j];
    nums[j] = tmp;
}
```

7.8 二叉搜索树

7.8.1 验证二叉搜索树

```
public boolean isValidBST(TreeNode root) {
    return isValidBST(root, Long.MIN_VALUE, Long.MAX_VALUE);
}
private boolean isValidBST(TreeNode root, long min, long max){
    if (root == null) {
        return true;
    }

    if (root.val <= min || root.val >= max){
        return false;
    }

    return isValidBST(root.left, min, root.val) && isValidBST(root.right, root.val, max);
}
```

7.8.2 第 K 小的元素

增加 getCount 方法来获取传入节点的子节点数（包括自己），从 root 节点开始判断 k 值和子节点数的大小决定递归路径是往左还是往右。

```
public int kthSmallest(TreeNode root, int k) {
    if (root == null) {
        return 0;
    }

    int leftCount = getCount(root.left);
    if (leftCount >= k) {
```

```
        return kthSmallest(root.left, k);
    } else if (leftCount + 1 == k) {
        return root.val;
    } else {
        return kthSmallest(root.right, k - leftCount - 1);
    }
}
private int getCount(TreeNode root) {
    if (root == null) {
        return 0;
    }

    return getCount(root.left) + getCount(root.right) + 1;
}
```

7.9 数组 / 双指针

7.9.1 加一

给定一个非负数，表示一个数字数组，在该数的基础上+1，返回一个新的数组。

该数字按照数位高低进行排列，最高位的数在列表的最前面。

```
public int[] plusOne(int[] digits) {
    int carries = 1;
    for(int i = digits.length - 1; i >= 0 && carries > 0; i--){
        int sum = digits[i] + carries;
        digits[i] = sum % 10;
        carries = sum / 10;
```

```
    }

    if(carries == 0) {

        return digits;

    }



    int[] rst = new int[digits.length + 1];

    rst[0] = 1;

    for(int i = 1; i < rst.length; i++){

        rst[i] = digits[i - 1];

    }

    return rst;

}
```

7.9.2 删除元素

给定一个数组和一个值，在原地删除与值相同的数字，返回新数组的长度。

```
public int removeElement(int[] A, int elem) {

    if (A == null || A.length == 0) {

        return 0;

    }



    int index = 0;

    for (int i = 0; i < A.length; i++) {

        if (A[i] != elem) {

            A[index++] = A[i];

        }

    }
```

```
    return index;
}
```

7.9.3 删除排序数组中的重复数字

在原数组中“删除”重复出现的数字，使得每个元素只出现一次，并且返回“新”数组的长度。

```
public int removeDuplicates(int[] A) {
    if (A == null || A.length == 0) {
        return 0;
    }

    int size = 0;
    for (int i = 0; i < A.length; i++) {
        if (A[i] != A[size]) {
            A[++size] = A[i];
        }
    }
    return size + 1;
}
```

7.9.4 我的日程安排表 I

实现 MyCalendar 类来存储活动。如果新添加的活动没有重复，则可以添加。类将有方法 book(int start，int end)。这代表左闭右开的间隔[start，end)有了预定，

范围内的实数 x，都满足 start <= x < end，返回 true。 否则，返回 false，并且事件不会添加到日历中。

TreeMap 是一个有序的 key-value 集合，它通过 红黑树 实现，继承于 AbstractMap，所以它是一个 Map，即一个 key-value 集合。TreeMap 可以查询小于等于某个值的最大的 key，也可查询大于等于某个值的最小的 key。 元素的顺序可以改变，并且对新的数组不会有影响。

```
class MyCalendar {

    TreeMap<Integer, Integer> calendar;



    MyCalendar() {

        calendar = new TreeMap();

    }



    public boolean book(int start, int end) {

        Integer previous = calendar.floorKey(start), next = calendar.ceilingKey(start);

        if ((previous == null || calendar.get(previous) <= start) && (next == null || end
<= next)) {

            calendar.put(start, end);

            return true;

        }

        return false;

    }

}
```

7.9.5 合并排序数组

合并两个排序的整数数组 A 和 B 变成一个新的数组。可以假设 A 具有足够的空间去添加 B 中的元素。

```
public void mergeSortedArray(int[] A, int m, int[] B, int n) {
    int i = m - 1, j = n - 1, index = m + n - 1;
    while (i >= 0 && j >= 0) {
        if (A[i] > B[j]) {
            A[index--] = A[i--];
        } else {
            A[index--] = B[j--];
        }
    }
    while (i >= 0) {
        A[index--] = A[i--];
    }
    while (j >= 0) {
        A[index--] = B[j--];
    }
}
```

7.10 贪心

7.10.1 买卖股票的最佳时机

假设有一个数组，它的第 i 个元素是一支给定的股票在第 i 天的价格。如果你最多只允许完成一次交易(例如，一次买卖股票)，设计一个算法来找出最大利润。

```
public int maxProfit(int[] prices) {
```

```
    if (prices == null || prices.length == 0) {
        return 0;
    }

    int min = Integer.MAX_VALUE;  //记录最低的价格
    int profit = 0;
    for (int price : prices) {
        min = Math.min(price, min);
        profit = Math.max(price - min, profit);
    }

    return profit;
}
```

7.10.2 买卖股票的最佳时机 II

> 给定一个数组 prices 表示一支股票每天的价格。可以完成任意次数的交易，不过不能同时参与多个交易，设计一个算法求出最大的利润。

贪心：只要相邻的两天股票的价格是上升的，我们就进行一次交易，获得一定利润。

```
public int maxProfit(int[] prices) {
    int profit = 0;
    for (int i = 0; i < prices.length - 1; i++) {
        int diff = prices[i + 1] - prices[i];
        if (diff > 0) {
            profit += diff;
        }
```

```
    }

    return profit;

}
```

7.10.3 最大子数组

给定一个整数数组，找到一个具有最大和的子数组，返回其最大和。

```
public int maxSubArray(int[] A) {

    if (A == null || A.length == 0){

        return 0;

    }

    //max 记录全局最大值，sum 记录区间和，如果当前 sum>0，那么可以继续和后面的数求和，否则就从 0 开始

    int max = Integer.MIN_VALUE, sum = 0;

    for (int i = 0; i < A.length; i++) {

        sum += A[i];

        max = Math.max(max, sum);

        sum = Math.max(sum, 0);

    }



    return max;

}
```

7.10.4 主元素

给定一个整型数组，找出主元素，它在数组中的出现次数严格大于数组元素个数的二分之一(可以假设数组非空，且数组中总是存在主元素)。

```
public int majorityNumber(List<Integer> nums) {
    int currentMajor = 0;
    int count = 0;

    for(Integer num : nums) {
        if(count == 0) {
            currentMajor = num;
        }

        if(num == currentMajor) {
            count++;
        } else {
            count--;
        }
    }
    return currentMajor;
}
```

7.11 字符串处理

7.11.1 生成括号

给定 n，表示有 n 对括号，请写一个函数以将其生成所有的括号组合，并返回组合结果。

```
public List<String> generateParenthesis(int n) {
    List<String> res = new ArrayList<>();
    helper(n, n, "", res);
```

```
    return res;
}
// DFSprivate void helper(int nL, int nR, String parenthesis, List<String> res) {
    // nL 和 nR 分别代表左右括号剩余的数量
    if (nL < 0 || nR < 0) {
        return;
    }

    if (nL == 0 && nR == 0) {
        res.add(parenthesis);
        return;
    }
    helper(nL - 1, nR, parenthesis + "(", res);
    if (nL >= nR) {
        return;
    }
    helper(nL, nR - 1, parenthesis + ")", res);
}
```

7.11.2 Excel 表列标题

给定一个正整数，返回相应的列标题，如 Excel 表中所示。如 1 -> A，2 -> B...26 -> Z，27 -> AA

```
public String convertToTitle (int n) {
    StringBuilder str = new StringBuilder();
```

```
    while (n > 0) {
        n--;
        str.append ( (char) ( (n % 26) + 'A'));
        n /= 26;
    }
    return str.reverse().toString();
}
```

7.11.3 翻转游戏

翻转游戏：给定一个只包含两种字符的字符串：+和-，你和你的小伙伴轮流翻转"++"变成"--"。当一个人无法采取行动时游戏结束，另一个人将是赢家。编写一个函数，计算字符串在一次有效移动后的所有可能状态。

```
public List<String> generatePossibleNextMoves (String s) {
    List list = new ArrayList();
    for (int i = -1; (i = s.indexOf ("++", i + 1)) >= 0;) {
        list.add (s.substring (0, i) + "--" + s.substring (i + 2));
    }
    return list;
}
```

7.11.4 翻转字符串中的单词

给定一个字符串，逐个翻转字符串中的每个单词。

```
public String reverseWords(String s) {
    if(s.length() == 0 || s == null){
        return " ";
    }
```

```
    //按照空格将 s 切分
    String[] array = s.split(" ");
    StringBuilder sb = new StringBuilder();
    //从后往前遍历 array，在 sb 中插入单词
    for(int i = array.length - 1; i >= 0; i--){
        if(!array[i].equals("")) {
            if (sb.length() > 0) {
                sb.append(" ");
            }

            sb.append(array[i]);
        }
    }
    return sb.toString();
}
```

7.11.5 转换字符串到整数

实现 atoi 这个函数，将一个字符串转换为整数。如果没有合法的整数，返回 0。

如果整数超出了 32 位整数的范围，返回 INT_MAX(2147483647)如果是正整数，

或者 INT_MIN(-2147483648)如果是负整数。

```
public int myAtoi(String str) {
    if(str == null) {
        return 0;
    }
    str = str.trim();
```

```
if (str.length() == 0) {
    return 0;
}

int sign = 1;
int index = 0;

if (str.charAt(index) == '+') {
    index++;
} else if (str.charAt(index) == '-') {
    sign = -1;
    index++;
}
long num = 0;
for (; index < str.length(); index++) {
    if (str.charAt(index) < '0' || str.charAt(index) > '9') {
        break;
    }
    num = num * 10 + (str.charAt(index) - '0');
    if (num > Integer.MAX_VALUE ) {
        break;
    }
}
if (num * sign >= Integer.MAX_VALUE) {
    return Integer.MAX_VALUE;
```

```
    }

    if (num * sign <= Integer.MIN_VALUE) {

        return Integer.MIN_VALUE;

    }

    return (int)num * sign;

}
```

7.11.6 最长公共前缀

```
public String longestCommonPrefix(String[] strs) {

    if (strs == null || strs.length == 0) {

        return "";

    }

    String prefix = strs[0];

    for(int i = 1; i < strs.length; i++) {

        int j = 0;

        while (j < strs[i].length() && j < prefix.length() && strs[i].charAt(j) ==
prefix.charAt(j)) {

            j++;

        }

        if( j == 0) {

            return "";

        }

        prefix = prefix.substring(0, j);

    }

    return prefix;

}
```

7.11.7 回文数

> 判断一个正整数是不是回文数。回文数的定义是，将这个数反转之后，得到的数仍然是同一个数。

```
public boolean palindromeNumber(int num) {

    // Write your code here

    if(num < 0){

        return false;

    }

    int div = 1;

    while(num / div >= 10){

        div *= 10;

    }

    while(num > 0){

        if(num / div != num % 10){

            return false;

        }

        num = (num % div) / 10;

        div /= 100;

    }

    return true;

}
```

7.12 动态规划

7.12.1 单词拆分

给定字符串 s 和单词字典 dict，确定 s 是否可以分成一个或多个以空格分隔的子串，并且这些子串都在字典中存在。

```
public boolean wordBreak(String s, Set<String> dict) {
    // write your code here
    int maxLength = getMaxLength(dict);

    // 长度为 n 的单词 有 n + 1 个切割点 比如: _l_i_n_t_
    boolean[] canBreak = new boolean[s.length() + 1];
    // 当 s 长度为 0 时
    canBreak[0] = true;

    for(int i = 1; i < canBreak.length; i++){
        for(int j = 1; j <= maxLength && j <= i; j++){
            //i - j 表示从 i 点开始往前 j 个点的位置
            String str = s.substring(i - j,i);
            //如果此 str 在词典中 并且 str 之前的 字符串可以拆分
            if(dict.contains(str) && canBreak[i - j]){
                canBreak[i] = true;
                break;
            }
        }
    }

    return canBreak[canBreak.length - 1];
}
```

```
private int getMaxLength(Set<String> dict){
    int max = 0;
    for(String s : dict){
        max = Math.max(max,s.length());
    }
    return max;
}
```

7.12.2 爬楼梯

> 假设你正在爬楼梯，需要 n 步你才能到达顶部。但每次你只能爬一步或者两步，你能有多少种不同的方法爬到楼顶部？

```
public int climbStairs(int n) {
    if (n == 0) return 0;
    int[] array = new int[n + 1];
    array[0] = 1;
    if (array.length > 1) {
        array[1] = 1;
    }

    for(int i = 2; i < array.length; i++) {
        array[i] = array[i - 1] + array[i - 2];
    }
    return array[n];
}
```

7.12.3 打劫房屋

假设你是一个专业的窃贼，准备沿着一条街打劫房屋。每个房子都存放着特定金额的钱。你面临的唯一约束条件是：相邻的房子装着相互联系的防盗系统，且 当相邻的两个房子同一天被打劫时，该系统会自动报警。给定一个非负整数列表，表示每个房子中存放的钱， 算一算，如果今晚去打劫，在不触动报警装置的情况下, 你最多可以得到多少钱 。

```
public long houseRobber(int[] A) {

    if (A.length == 0) return 0;

    long[] res = new long[A.length + 1];

    res[0] = 0;

    res[1] = A[0];

    for (int i = 2; i < res.length; i++) {

        res[i] = Math.max(res[i - 2] + A[i - 1], res[i - 1]);

    }

    return res[A.length];

}
```

7.12.4 编辑距离

给出两个单词 word1 和 word2，计算出将 word1 转换为 word2 的最少操作次数。你总共三种操作方法：插入一个字符、删除一个字符、替换一个字符。

```
public int minDistance(String word1, String word2) {

    // write your code here

    int n = word1.length();

    int m = word2.length();

    int[][] dp = new int[n + 1][m + 1];

    for (int i = 0; i < n + 1; i++){
```

```
        dp[i][0] = i;
    }
    for (int j = 0; j < m + 1; j++){
        dp[0][j] = j;
    }
    for (int i = 1; i< n + 1; i++){
        for (int j = 1; j < m + 1; j++){
            if (word1.charAt(i - 1) == word2.charAt(j - 1)){
                dp[i][j] = dp[i - 1][j - 1];
            } else {
                dp[i][j] = 1 + Math.min(dp[i - 1][j - 1], Math.min(dp[i][j - 1], dp[i - 1][j]));
            }
        }
    }
    return  dp[n][m];
}
```

7.12.5 乘积最大子序列

```
public int maxProduct(List<Integer> nums) {
    // 分别记录正数最大值和负数最小值
    int[] max = new int[nums.size()];
    int[] min = new int[nums.size()];

    min[0] = max[0] = nums.get(0);
    int result = nums.get(0);
```

```
    for (int i = 1; i < nums.size(); i++) {
        min[i] = max[i] = nums.get(i);
        if (nums.get(i) > 0) {
            max[i] = Math.max(max[i], max[i - 1] * nums.get(i));
            min[i] = Math.min(min[i], min[i - 1] * nums.get(i));
        } else if (nums.get(i) < 0) {
            max[i] = Math.max(max[i], min[i - 1] * nums.get(i));
            min[i] = Math.min(min[i], max[i - 1] * nums.get(i));
        }

        result = Math.max(result, max[i]);
    }

    return result;
}
```

7.13 矩阵

7.13.1 螺旋矩阵

给定一个包含 m x n 个要素的矩阵，（m 行, n 列），按照螺旋顺序，返回该矩阵中的所有要素。

```
public List<Integer> spiralOrder(int[][] matrix) {
    ArrayList<Integer> rst = new ArrayList<Integer>();
    if(matrix == null || matrix.length == 0) {
        return rst;
    }
```

```
int rows = matrix.length;
int cols = matrix[0].length;
int count = 0;
while(count * 2 < rows && count * 2 < cols){
    for (int i = count; i < cols - count; i++) {
        rst.add(matrix[count][i]);
    }

    for (int i = count + 1; i < rows - count; i++) {
        rst.add(matrix[i][cols - count - 1]);
    }

    if (rows - 2 * count == 1 || cols - 2 * count == 1) { // 如果只剩 1 行或 1 列
        break;
    }

    for (int i = cols - count - 2; i >= count; i--) {
        rst.add(matrix[rows - count - 1][i]);
    }

    for (int i = rows - count - 2; i >= count + 1; i--) {
        rst.add(matrix[i][count]);
    }
```

```
        count++;
    }
    return rst;
}
```

7.13.2 判断数独是否合法

> 请判定一个数独是否有效。该数独可能只填充了部分数字，其中缺少的数字用 . 表示。

维护一个 HashSet 用来记同一行、同一列、同一九宫格是否存在相同数字

```
public boolean isValidSudoku(char[][] board) {
    Set seen = new HashSet();
    for (int i=0; i<9; ++i) {
        for (int j=0; j<9; ++j) {
            char number = board[i][j];
            if (number != '.')
                if (!seen.add(number + " in row " + i) ||
                    !seen.add(number + " in column " + j) ||
                    !seen.add(number + " in block " + i / 3 + "-" + j / 3))
                    return false;
        }
    }
    return true;
}
```

7.13.3 旋转图像

给定一个 N×N 的二维矩阵表示图像，90 度顺时针旋转图像。

```
public void rotate(int[][] matrix) {
    if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
        return;
    }

    int length = matrix.length;

    for (int i = 0; i < length / 2; i++) {
        for (int j = 0; j < (length + 1) / 2; j++){
            int tmp = matrix[i][j];
            matrix[i][j] = matrix[length - j - 1][i];
            matrix[length -j - 1][i] = matrix[length - i - 1][length - j - 1];
            matrix[length - i - 1][length - j - 1] = matrix[j][length - i - 1];
            matrix[j][length - i - 1] = tmp;
        }
    }
}
```

7.14 二进制 / 位运算

7.14.1 落单的数

给出 2 * n + 1 个数字，除其中一个数字之外其他每个数字均出现两次，找到这个数字。

异或运算具有很好的性质，相同数字异或运算后为 0，并且具有交换律和结合律，故将所有数字异或运算后即可得到只出现一次的数字。

```
public int singleNumber(int[] A) {
    if(A == null || A.length == 0) {
        return -1;
    }
    int rst = 0;
    for (int i = 0; i < A.length; i++) {
        rst ^= A[i];
    }
    return rst;
}
```

7.14.2 格雷编码

> 格雷编码是一个二进制数字系统，在该系统中，两个连续的数值仅有一个二进制的差异。给定一个非负整数 n ，表示该代码中所有二进制的总数，请找出其格雷编码顺序。一个格雷编码顺序必须以 0 开始，并覆盖所有的 2n 个整数。
>
> 例子——输入：2；输出：[0, 1, 3, 2]；解释: 0 - 00，1 - 01，3 - 11，2 - 10

格雷码生成公式：G(i) = i ^ (i >> 2)

```
public ArrayList<Integer> grayCode(int n) {
    ArrayList<Integer> result = new ArrayList<Integer>();
    for (int i = 0; i < (1 << n); i++) {
        result.add(i ^ (i >> 1));
    }
```

```
    return result;

}
```

7.15 其他

7.15.1 反转整数

> 将一个整数中的数字进行颠倒，当颠倒后的整数溢出时，返回 0 (标记为 32 位整数)。

```
public int reverseInteger(int n) {

    int reversed_n = 0;



    while (n != 0) {

        int temp = reversed_n * 10 + n % 10;

        n = n / 10;

        if (temp / 10 != reversed_n) {

            reversed_n = 0;

            break;

        }

        reversed_n = temp;

    }

    return reversed_n;

}
```

7.15.2 LRU 缓存策略

为最近最少使用（LRU）缓存策略设计一个数据结构，它应该支持以下操作：获取数据（get）和写入数据（set）。获取数据 get(key)：如果缓存中存在 key，则获取其数据值（通常是正数），否则返回-1。 写入数据 set(key, value)：如果 key 还没有在缓存中，则写入其数据值。当缓存达到上限，它应该在写入新数据之前删除最近最少使用的数据用来腾出空闲位置。

```
public class LRUCache {

    private class Node{

        Node prev;

        Node next;

        int key;

        int value;


        public Node(int key, int value) {

            this.key = key;

            this.value = value;

            this.prev = null;

            this.next = null;

        }

    }


    private int capacity;

    private HashMap<Integer, Node> hs = new HashMap<Integer, Node>();

    private Node head = new Node(-1, -1);

    private Node tail = new Node(-1, -1);
```

```
public LRUCache(int capacity) {
    this.capacity = capacity;
    tail.prev = head;
    head.next = tail;
}

public int get(int key) {
    if( !hs.containsKey(key)) {                //key 找不到
        return -1;
    }

    // remove current
    Node current = hs.get(key);
    current.prev.next = current.next;
    current.next.prev = current.prev;

    // move current to tail
    move_to_tail(current);                     //每次 get，使用次数+1，最近使用，放于尾部

    return hs.get(key).value;
}

public void set(int key, int value) {                  //数据放入缓存
    // get 这个方法会把 key 挪到最末端，因此，不需要再调用 move_to_tail
    if (get(key) != -1) {
```

```
            hs.get(key).value = value;
            return;
        }

        if (hs.size() == capacity) {                //超出缓存上限
            hs.remove(head.next.key);               //删除头部数据
            head.next = head.next.next;
            head.next.prev = head;
        }

        Node insert = new Node(key, value);                 //新建节点
        hs.put(key, insert);
        move_to_tail(insert);                                       //放于尾部
    }

    private void move_to_tail(Node current) {    //移动数据至尾部
        current.prev = tail.prev;
        tail.prev = current;
        current.prev.next = current;
        current.next = tail;
    }
}
```