7月25日

学习内容:

本日任务主要是刷最新的一道算法题，并且对之前完成的算法进行了一定程度的优化，阅读了一些算法优化书籍。

学习收获:

在最新的算法题中，主要是实现一个会议预定系统，这个系统的关键就在于每一个团队在预定会议的时候都需要考虑团队中全部成员的空闲时间， 并且还得距离当前日期最近，这就不能简单的使用星期排序，查找空闲时间的方法，需要有一个变量来记录当日期，并且将一个团队中的所有人空闲时间做遍历比较，这种方法的时间复杂度会非常高，很有可能超时，这时我参考了一些相关问题的解决方法，采用了时间线这一做法，将每个team都维护一个时间线，时间线是一个包含整个工作时间的数组，每次需要预定的时候就会根据时间线选择可用的时间。

在具体的优化过程中，我在Main函数运行时会在函数前后添加系统时间，用于记录函数耗时，在每次运行完后根据系统耗时优化对应函数的方法和数据结构。

7月26日

学习内容:

学习了安卓开发中的 AlterDialog ， ProgressDialog 和 Android 的线性布局

学习收获:

AlterDialog 该插件是再当前界面弹出一个对话框，这个对话框是置于所有界面元素之上，能屏蔽掉其他控件的交互能力，一般是用于误删或者退出，下面是我在thirdActivity中“直接退出按钮”做的一个警告框。

在代码中需要分别定义两个按钮的具体点击事件

```
dialog.setPositiveButton("确认", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialogInterface, int i) {
        //再点击确认就会彻底退出
        ActivityCollector.finishAll();
    }
});
dialog.setNegativeButton("取消", new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialogInterface, int i) {
        //点击取消啥也不发生
    }
});
```

最后一定要记得加上 show() 函数激活警告框

ProgressDialog 这个和上一个很像，但是包含一个进度条，通常用于比较耗时的操作，希望用户耐心等待。

代码和上面差不多。需要多设置一下Message

```
progressDialog.setMessage("给我等嗷...");
progressDialog.setCancelable(true);
```

在Android中，布局是可以嵌套布局的，很多时候可以进行多层嵌套用于优化布局。

线性布局是最简单的一种布局，需要注意的是有垂直和水平两种，在使用线性布局的时候常常会使用`match_parent`这一属性，需要记住的是在垂直中高度不能使用该值，同样的在水平中宽度不能该值。

学习内容：

主要是跟随文档，将卡顿分析中遇到了一些问题进行了总结归纳。

学习收获：

有些时候主线程和渲染线程在这一帧超时，但是往往掉帧都会发生在后面几帧，并且主线程需要等渲染线程执行完syncFrameState 之后才能 unlockMainThread ，这样下一帧的主线程才能正常工作。

同时，在很多情况下，应用的一些操作并不会引发明显卡顿，在这种时候就需要一些其他的手段进行分析，比如使用以下指令

```
adb shell dumpsys gfxinfo
```

以下是具体的操作过程

1. 先确定好要测试的包名
2. 执行两三次下述命令，清除历史数据

```
adb shell dumpsys gfxinfo xxx(包名) framestats reset
```

3. 开始复现需要测试的操作
4. 操作结束后，执行

   ```
   adb shell dumpsys gfxinfo xxx(包名) framestats
   ```

   这个时候会出现一堆数据，我们只需要关注其中的一部分。
5. 重点关注的数据

   01、**Janky frames**：超过Vsync周期的Frame，但是不一定卡顿

   02、**95th percentile**： 95%的值

   03、**HISTOGRAM**：原始数值

但是这个方法也有很大的局限性，就是它读取的帧数有限，并不能作为最优解。

7月27日-7月28日

学习内容：

主要学习了响应速度的实战分析，总结了一些分析中需要注意的问题

学习收获：

因为响应速度和流畅度不同，它没有一个具体的量化指标，如并没有一个统一的标准规定判断一个响应场景具体的起点和终点。在实际生活中，主要是分为三个角色来进行响应评估：

- **系统开发者**：往往从 input 中断开始看，部分以应用第一帧为结束点，部分以应用加载完成作为结束点，主要是以优化应用的整体性能为主，涉及到的方面就比较广，包括 input 事件传递、SystemServer、SurfaceFlinger、Kernel、Launcher 等。
- **App开发者**：一般从 Application 的 onCreate 或者 attachContext 开始看，大部分以页面完全加载或者用户可操作作为借宿点，因为是自己的应用，结束点在代码里面可以主动加，主要还是以优化应用自身的启动速度为主，市面上将启动优化的，大部分是讲这部分。
- **测试同学** :则更多从用户的真实体验角度来看，以桌面点击应用图标且应用图标变色为第一帧，内容完全加载为结束点。测试过程一般使用 **高速相机 + 自动化**，通过**机械手**和**图形识别技术**。

接下来我会详细介绍一下响应问题的分析思路

1、需要分清起点和终点

2、需要能根据 Systrace 的数据大致判断是属于哪种类型的问题导致响应慢。

3、具体判断到底是系统导致的响应慢，还是应用自身导致的响应慢。

以下总结了一些常见的系统问题和它们在 Systrace 中的具体表现。

1. CPU频率不足：主线程处于 Running 状态，但是执行耗时变长。
2. CPU大小核调度：关键任务跑到了小核，表现是主线程处于 Running 状态，但是执行耗时变长。
3. SystemServer 繁忙：响应 App 主线程 Binder 调用处理耗时，在 APP 端的表现是主线程处于 Sleep 状态，在等待 Binder 调用返回。
4. SurfaceFlinger 繁忙：主要是影响应用的渲染线程的 dequeueBuffer、queueBuffer，主要表现为应用渲染线程的 dequeueBuffer、queueBuffer 处于 Binder 等待状态。
5. 系统低内存：

   01、系统低内存的时候，有些应用会频繁的被杀和启动，而应用启动时一个重操作，会占用CPU资源，导致前台App启动变慢，主要表现就是应用主线程 Runnable 状态变多，Running 状态变少，整体函数执行耗时增加。

   02、低内存的时候，很容易触发各个进程的 GC ，用于内存回收的 HeapTaskDeamon、kswapd0 出现非常频繁。主要表现为应用主线程Runnable 状态变多，Running 状态变少，整体函数执行耗时增加。

   03、低内存会导致磁盘 IO 变多，如果频繁进行磁盘 IO ，由于磁盘 IO 很慢，那么主线程会有很多进程处于等 IO 的状态，也就是我们经常看到的 Uninterruptible Sleep 和 Uninterruptible Sleep - IO 状态变多，Running 状态变少，整体函数执行耗时增加。
6. 系统触发温控频率被限制：由于温度过高，CPU 最高频率被限制，主要表现和降频一样
7. 整机 CPU 繁忙：可能有多个高负载进程同时在运行，或者有单个进程负载过高跑满了 CPU，具体表象是 CPU 区域任务非常满，所有的核心上都有任务在执行，APP 的主线程和渲染线程多处于 Runnable 状态，或者频繁在 Runnable 和 Running 之间切换。

在这之中很多都是之前 Systrace 基础学习中学习过的，没接触过的为 HeapTaskDeamon、kswapd0 这两个方法

通过查阅相关资料学习到

HeapTaskDeamon：这个方法是一个关于堆的守护线程，负责裁剪空闲的堆交还给内存

kswapd0：kswapd是内存回收进程，会创建node，每个node节点各自创建一个kswapd线程

以下我总结的关于应用启动响应速度的具体实践要点：

1、和分析思路一样，需要分清起点和终点

2、将整个过程进行分段，通过对比分析查看是那一块的时间明显增加。

3、分析上一步找到的耗时点，在应用层主要查看应用的主线程、渲染线程和 Binder 通信是否出现了异常。

4、在系统层分析问题，可以根据上面提到的一些常见问题进行对应分析，主要看 Systrace 中的4个部分

01、**Kernel**：

- **查看关键任务是否跑在了小核**： 一般是0-3，如果启动是很好的关键任务跑到了小核，执行速度也会变慢。
- **查看频率是否跑满**：表现是核心频率没有达到最大值
- **查看 CPU 使用率**：表现是在 CPU 区域的各个核心上，任务和任务之间没有空隙。
- **查看是否低内存：**应用进程状态有大量的 Uninterruptible Sleep | WakeKill - Block I/O

  HeapTaskDeamon 任务执行频繁 和 kswapd0任务执行频繁。

02、**SystemServer 进程区域**：

- **input事件的读取和分发是否有异常**：表示是 input 传递耗时，较为少见。
- **Binder 执行是否耗时**：表现是 SystemServer 对应的 Binder 执行代码逻辑耗时。
- **是否有应用频繁启动或被杀**：在 Systrace 中查看 startProcess，或者查看 EventLog

03、**SurfaceFlinger进程区域：**

- **dequeueBuffer 和 queueBuffer 是否耗时**：表现是 SurfaceFlinger 的对应的 Binder 执行 dequeueBuffer 和 queueBuffer 耗时，app dequeue 在 surfaceFlinger 中能看到。
- **主线程执行是否耗时：**表现为 SurfaceFlinger 主线程耗时，可能是在执行其他任务。

04、**Launcher 进程区域（冷热启动场景）**：

- **Launcher 进程处理点击事件是否耗时**：表现在处理input事件耗时
- **Launcher 自身 pause 是否耗时**：表现在执行 onPause 耗时
- **Launcher 应用启动动画是否耗时或者卡顿**：表现在动画耗时或这卡顿

冷启动：指 app 被后台杀死后，在这个状态打开 app，这种启动方式叫做冷启动。

热启动：指 app 没有被后台杀死，仍然在后台运行，通常我们再次去打开这个 app，这种启动方式叫热启动。

5、最后就是总结找到的问题，进行同类问题类比，验证分析结果是否具有普适性，最终将其生成为报告。

7月29日

学习内容：

算法考试，响应速度知识延伸，TraceView的使用说明

学习收获:

算法考试中考了BFS，看似简单，但是最后运行超时了，原因是没有堆题目中的矩阵进行压缩，导致遍历太多，拖慢了速度，感觉还是需要加强数据结构的学习，对一些算法优化方面的方法了解还不够，需要加强学习。

学会了TraceView的使用，使用了 Android Studio 自带的 CPU profiler 对自己写的安卓 APP 进行方法追踪，因为APP 是自己编写的，在方法追踪中可以清晰的理解系统和应用层的方法调用关系，方便理解 Android App 具体运行中的运行逻辑，有利于之后对于 APP 的性能优化。