7月18日

今天的主要任务是学习Systrace中的Vsync、Binder和锁竞争，在此基础上会学习一些Android开发的基础知识，用以辅助验证Systrace学习中学到的知识点。

学习内容：

安卓开发中启动活动的最佳写法

学习收获：

多个活动之间进行传递的时候需要使用Intent作为数据传递的媒介，如下

```
Intent intent = new Intent(FirstActivity.this, SecondActivity.class);
intent.putExtra("param1", "data1");
intent.putExtra("param2", "data2");
startActivity(intent);
```

这样并没有问题，但是在复杂的开发中往往不能便捷的知道putExtra中的参数代表的具体信息，在应用性能追踪时会给我们造成阻碍，故可以使用以下添加以下方法：

```
public class SecondActivity extends BaseActivty{
    public static void actionStart(Context context,String data1,String data2){
        Intent intent = new Intent(context,SecondActivity.class);
        intent.putExtra("param1",data1);
        intent.putExtra("param2",data2);
        context.startActivity(intent);
    }
}
```

在活动First中，使用actionStart启动活动second就可以很清晰的知道需要填入哪些参数。可以节省时间，不需要几个活动代码直接来回查看。

学习内容：

安卓开发中的常用控件（TextView，Button）

学习收获：

控件一般都需要几种常用的属性，用id确定唯一标识符，layout_width，layout_height指定控件的长宽，修改对齐方式可以使用gravity，字体大小和颜色使用textSize，textColor进行修改，Button相比于TextView多了一个textAllCaps，因为Button会将text自动大写，可以通过这个进行关闭。

学习内容：

Systrace中的Vsync模块

学习收获：

学习内容主要分以下几个模块

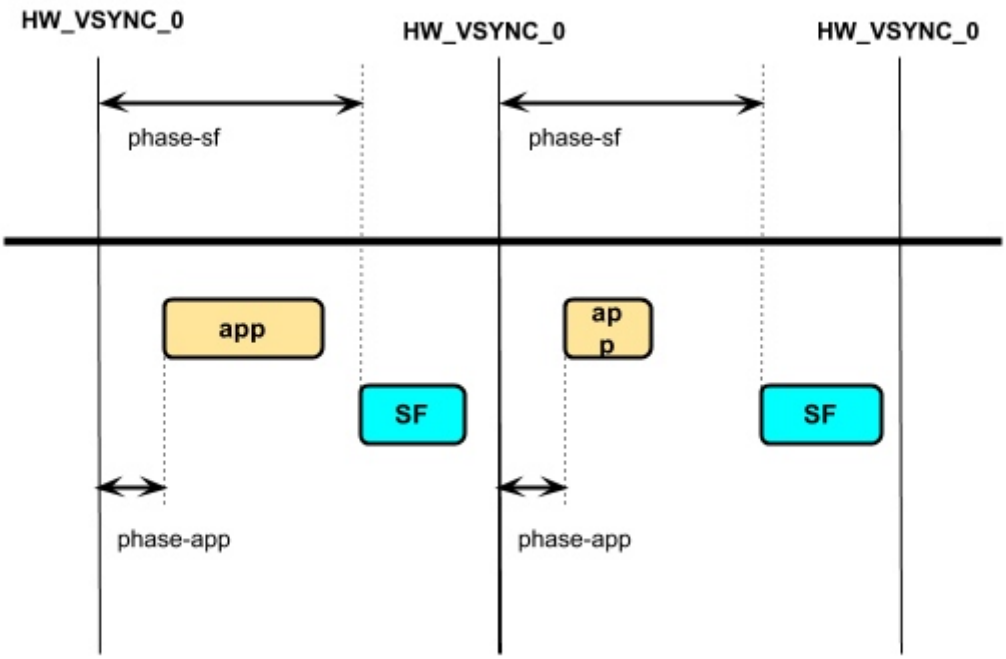
Android和Systrace的图形数据流向：了解到系统的图像数据流传输过程，了解了Vsync分别去向APP和SF，并且知道了在时间序列上它们的实现方式：

1. **App在收到Vsync信号后，在主线程进行measure、layout、draw操作**，这里对应的是Systrace中的**doFrame**操作。
2. **CPU将数据上传给GPU**，这里ARM设备内存一般都是CPU和GPU一起共享的，这里对应Systrace中渲染线程的flush drawing commands操作。
3. **通知GPU进行渲染**，一般来说，真机的GPU渲染不会阻塞，等待GPU渲染结束，就会通知CPU渲染结束，CPU就会返回继续执行其他任务，在这里一般会使用**Fence**机制进行CPU与GPU的同步操作，所以这个过程对应着Systrace中的Fence过程
4. **swapBuffers，并通知SurfaceFlinger图层合成。**这里对应这渲染线程的**eglSwapBuffersWithDamageKHR**操作。
5. surfaceFlinger收到了Vsync-SF信号开始去取Buffer进行合成。

在这其中还学习了有关Fence机制，大致上就是一个资源锁，类似与操作系统中的PV原语，主要是保证Buffer不会被同时读取和写入。

了解了图像数据流动过程对应的Systrace表现形式，能根据Systrace呈现的信息了解当前处于数据流动中的哪一步。

学习了Vsync Offset机制，知道了Vsync-app与Vsync-SF之间存在的间隔，分析了Offset的具体作用和优缺点：



Vsync Offset的作用：在上述过程中，我们发现每次渲染完都要等Vsync-SF的信号到来才能进行合成操作，如果SF和App同步，那么每次都在渲染上一帧，响应的操作不能更快的显现出来，如果我们设置好对应的Offset，使得App刚渲染完，SF的信号就到达，那么就能马上进行合成，这样给用户的感觉会更流畅跟手。

Offset并没有一个最优的数值，都是根据使用场景和当前性能而变化，很多情况下不好设置

1. 如果配置时间过短，很可能在APP还没有渲染完，SF就收到信号了，那么时长就变成了Offset+Vsync
2. 如果配置时间过长，那么就没有效果了

最后对HW_Vsync进行了一些简单的了解。

7月19日

今天主要学习了安卓开发的一些基本控件和Systrace中Binder、锁竞争和CPU Info模块。

学习内容

主要包括以下几个控件，EditText，ImageView和ProgressBar，并且学习了这几个控件和Button的点击事件之间的联动。

学习收获

1、EditText

允许用户输入输出，还是很常用的

输入id、layout_width、layout_height就可以了

hint代表输入框中的提示文字

输入内容过多的时候如果使用wrap_content就会很难看，一般使用maxLines来解决这个问题。这样editText最多就是两行，如果内容过多就会向上滚动，而不会继续拉伸。

2、ImageView

ImageView是用于在界面上展示图片的一个控件，它可以让我们的程序界面变得更加丰富多彩。学习这个控件需要提前准备好一些图片。图片通常是放在以drawable开头的目录下的，并且要带上具体的分辨率。现在最主流的手机屏幕分辨率大多是xxhdpi的，所以我们在res目录下再新建一个drawable-xxhdpi目录，然后将事先准备好的的图片复制到该目录当中。

同时我们也可以将该控件和Button的点击事件联系起来，如下

```
//制作点击更改图片按钮
设置全局变量ImageView imageView

//onCreate函数中添加啊
Button changeImage = findViewById(R.id.change_image_button);
imageView = findViewById(R.id.imageView);
changeImage.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        imageView.setImageResource(R.drawable.klee);
    }
});
```

3、ProgressBar

这个玩意是个进度条，表示我们的程序正在加载一些数据。它的用法也非常简单，修改activity_main.xml中的代码就行。

并且在实际使用中，往往都是当操作完成了，进度条就会自动消失，所以我们需要知道让控件消失的方法。

即更改Android控件的可见属性。

所有的Android控件都具有这个属性，可以通过android:visibility进行指定，可选值有3种：visible、invisible和gone。

visible表示控件是可见的，这个值是默认值，不指定android:visibility时，控件都是可见的。

invisible表示控件不可见，但是它仍然占据着原来的位置和大小，可以理解成控件变成透明状态了。

gone则表示控件不仅不可见，而且不再占用任何屏幕空间。

我们可以通过代码来设置控件的可见性，使用的是setVisibility()方法，允许传入View.VISIBLE、View.INVISIBLE和View.GONE这3种值。

```
//这个是制作一个按钮控制进度条的出现和消失
Button progressBarButton = findViewById(R.id.progressBar_button);
progressBar = (ProgressBar) findViewById(R.id.progressBar);
progressBarButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (progressBar.getVisibility() == View.GONE)
            progressBar.setVisibility(View.VISIBLE);
        else
            progressBar.setVisibility(View.GONE);
    }
});
```

同时我们很多时候都有更换进度条样式的情况，默认的进度条样式都是圆形的，可以在xml中更改为水平的

```
<ProgressBar
    android:id="@+id/progressBar"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    style="@style/Widget.AppCompat.ProgressBar.Horizontal"
    android:max="100"/>
```

```
//这个是制作一个按钮控制进度条的出现和消失
Button progressBarButton = findViewById(R.id.progressBar_button);
progressBar = findViewById(R.id.progressBar);//在xml中可以更改进度条样式
progressBarButton.setOnClickListener(new View.OnClickListener() {
    @Override
    public void onClick(View view) {
        if (progressBar.getVisibility() == View.GONE)
            progressBar.setVisibility(View.VISIBLE);
        else
            progressBar.setVisibility(View.GONE);

        int progress = progressBar.getProgress();//获得进度
        progress += 10;
        progressBar.setProgress(progress);//点一次加10
    }
```

在学习过程中我发现针对于控件的操作很多时候都是差不多的，往往分为以下几个步骤

1. 先给活动对应的xml添加控件，配置好相应的属性
2. 在活动类中使用**findViewById**方法找到具体的控件
3. 接着使用View对象的一些API实现功能，如Button的**setOnClickListener**方法

总的来说，对于Android如果操作控件，使用控件传递数据这一类的操作是已经掌握了。

学习内容

Systrace中的Binder和锁竞争

学习收获

本次主要了解了Binder在Systrace中的具体表现，在具体的应用进程中，往往能够看到Binder transaction，这就代表着进行了Binder操作，同时，Binder操作往往都会伴随着锁的出现，很多卡顿分析和响应问题也是和锁息息相关。

关于锁，大概分为下面几个分析过程

1. 需要解读当前锁的信息，一般搜索monitor得到因为锁而无法进行的操作信息
2. 根据信息找到引起锁的进程，查看它当前调用的函数
3. 根据调用的函数，查阅源码，分析锁的出现

把握好上述几点，还需要注意将当前因为锁而引发的阻塞队列带来的潜在卡顿可能。

学习内容

Systrace中的CPU相关信息

学习收获

Systrace 中 CPU Info 一般在最上面，里面经常会用到的信息包括：

1. CPU 频率变化情况
2. 任务执行情况
3. 大小核的调度情况
4. CPU Boost 调度情况

我们在查看CPU学习的收获重点关注以下几个问题

1. 某个场景的任务执行比较慢，我们就可以查看是不是这个任务被调度到了小核？
2. 某个场景的任务执行比较慢，当前执行这个任务的 CPU 频率是不是不够？
3. 我的任务比较特殊，比如指纹解锁，能不能把我这个任务放到大核去跑？
4. 我这个场景对 CPU 要求很高，我能不能要求在我这个场景运行的时候，限制 CPU 最低频率？

针对一些对特殊应用，厂商往往会使用绑核操作，将一些任务分给特定的CPU组，达到理想的运行效果和能耗。

在一些特定场景下，如果让调度器负责拉频率和迁核，会造成一定的延迟，比如一开始跑在小核，后面一级一级往上拉，非常耗时。基于这种情况，一般选择的都是暴力拉核，直接将所有性能拉高。

目前在以下几个场景中可能会选择锁频

1. 应用启动
2. 应用安装
3. 转屏
4. 窗口动画
5. List Fling
6. Game

7月20日-7月21日

20日主要进行的是算法练习

21日开始进行Systrace卡顿分析实战

具体如下

学习内容：

将6月份参加过的几次pro考试没做出的题目进行回顾重做，题目为protect island、lucy 外卖与Survival Train

学习收获：

通过Survival Train这道算法题，我学会了使用自定义的Comparator对treeset进行自定义排序，并且按照题目要求做成了字典排序，同时使用了HashMap进行存储，并且使用了自己设计的hash函数，因为共有26个字母，我取2的5次方作为进制，同时还需要主要会出现数据溢出的情况。

在lucy外卖中，我学习到了removeIf函数的使用方法，在实际过程中，我们对treeset进行迭代的时候是不能进行remove操作的，这样会导致迭代器定位错误，故不能用传统的方法对特定的元素进行删除，最终我选择了两种方法，其一是使用HashMap记录数据，HashMap与treeset的元素出自同一处，通过HahMap提供具体元素进行定位，第二种方法是使用removeIf函数，通过lambda表达式传递判断函数体，或者使用Predicate接口进行判定。

在protect island中着重学习了在有关图的问题上，该如何拆解问题，比如在安装墙体的过程中墙体有四种方向，我们可以通过对墙体做逆置，对地图做转置来实现这四种方向。并且学会了根据具体的题目分析当前该选择深度搜索还是广度搜索。

学习内容：

将Android系统部分机制复习了一遍，正式开始分析实战

学习收获：

从三个方面定义卡顿

1. **从现象上来说，在App连续的动画播放或者手指滑动列表时，如果连续两帧或以上，应用的画面没有发生变化，那么我们认为这里发生了掉帧。**
2. **从SurfaceFlinger的角度来说，在app连续的动画或者手指滑动列表时，如果有一个Vsync到来的时候，App没有可以用来合成的Buffer，那么这个Vsync周期SurfaceFlinger就不会走合成的逻辑，那么这一帧就会显示App上一帧的画面，我们认为这里发生了卡顿。**
3. **从app的角度来看，如果渲染线程在一个Vsync周期内没有queueBuffer到SurfaceFlinger中App对应的Buffer Queue中，那么我们认为这里发生了卡顿。**

卡顿分为三类

1. **流畅度（滑动掉帧、窗口动画不连贯、重进桌面卡顿）**
2. **响应速度（启动白屏过长、点击电源键亮屏慢、滑动不跟手）**
3. **稳定性（界面操作没有反应然后闪退、点击图标没有响应）**

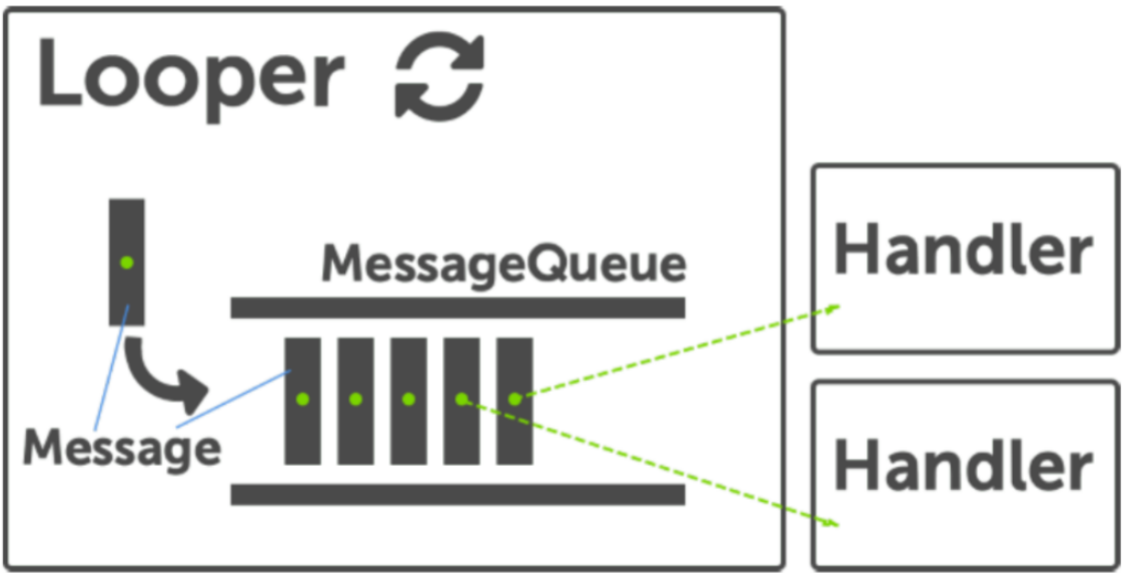
同时重温了一边系统机制，大致分为以下几个部分

1. App 主线程运行原理
2. Message、Handler、MessageQueue、Looper 机制
3. 屏幕刷新机制和 Vsync

查看了相应源码，弄清楚了Message、Handler、MessageQueue、Looper 这几个对象具体创建过程

学习了Message机制，主要分为四个核心：：**Handler**、**Looper**、**MessageQueue**、**Message**

[



1. **Handler** : Handler 主要是用来处理 Message，应用可以在任何线程创建 Handler，只要在创建的时候指定对应的 Looper 即可。
2. **Looper :** Looper 可以看成是一个循环器，其 loop 方法开启后，不断地从 MessageQueue 中获取 Message
3. **MessageQueue**：MessageQueue 、如上图所示，就是一个 Message 管理器，队列中是 Message，列
4. **Message**：Message 是传递消息的对象

接下来是屏幕刷新机制与Vsync

这部分内容比较简单，大致有几个要点：屏幕刷新率，FPS，Vsync

屏幕刷新率是硬件层面的，代表这个屏幕一秒能刷新多少次，FPS是软件层面的，代表一秒钟软件能生成多少张图片，Vsync就是垂直同期(Vertical Synchronization)的简称，大概作用就是将屏幕刷新率与FPS联系起来，避免出现掉帧撕裂的现象。

7月22日

学习内容

复习了Choreographer、Triple Buffer和input流程

学习收获

复习了Choreographer机制，该机制主要是实现Vsync的周期到达，保证帧率的稳定，给予用户流畅的使用体验。在功能上主要是接收和处理APP各种更新信息和回调，在系统中扮演者承上启下的角色。

Triple Buffer机制主要是用来缓和掉帧的，在Android系统的渲染绘图过程中，主要分为三个模块，分别是CPU、GPU、SurfaceFlinger，其中CPU和GPU分别对应着App主线程和渲染线程。使用Triple Buffer机制可以使得当Vsync-SF信号到来时GPU没有及时生成最新的Buffer，SurfaceFlinger可以通过合成之缓冲区中多余的Buffer达到避免掉帧的效果。同时，之所以使用3Buffer而不是其他是因为在渲染过程在主要有三个模块，使用Triple Buffer可以将性能利用最大化。

通过复习input流程，可以很清晰的了解一个事件在Android系统中的处理流程，流程大致如下

InputReader -> InputDispatcher -> OutboundQueue -> WaitQueue -> PendingInputEventQueue -> deliverInputEvent(在App的主线程中)

通过学习这个流程，在实际的工作中，可以根据该流程逐层排查，最终定位问题。

学习内容：

模拟Systrace实战，跟随实际教程，进行掉帧卡顿分析

学习收获：

学习了在实际操作中，如何进行问题排查

1. 首先看APP的主线程和渲染线程，观测超过Vsync周期的部分（不一定掉帧）
2. 分析SurfaceFlinger进程的主线程和Binder线程
3. 观察在Vsync-SF信号到来时，SurfaceFlinger是否进行合成操作，并且查看App对应的Buffer是否可用。

以下总结了两种卡顿情况

1、在某个Vsync周期中，SurfaceFlinger没有合成任务，但Vsync-APP有信号，这是观察APP对应的Buffer中是否有可用Buffer，若无就是掉帧

2、SurfaceFlinger 进行了合成，而且 App 在这一个 Vsync 周期(vsync-app)进行了正常的工作，**但是对应的 App 的 BufferQueue 里面没有可用的 Buffer，那么这一帧也是卡了。**这里合成的是其他APP的Buffer

发生了上述两种情况，往往伴随着App渲染线程的超时，针对这种情况，要么是该软件本身耗时，要么就是APP跑到小核上了。

最后在分析完所有的卡断点后，需要通过控制变量去验证，一旦解决了刚刚分析的卡顿原因，还是否会出现卡顿。