> title: Launcher启动分析以及优化建议-系统优化
>
> author: 欧杰

1.Launcher耗时大方向优化

1.1 IO优化

MyApplication和LauncherActivity Oncreate()中IO操作判断哪些是必须的，非必须的，使用子线程或者延后加载。默认启动VPA界面，可以先将布局加载出来之后，然后通知退出开机动画，使用IdleHandle加载资源显示具体的资源内容。

1.2 懒加载或者初始化时机

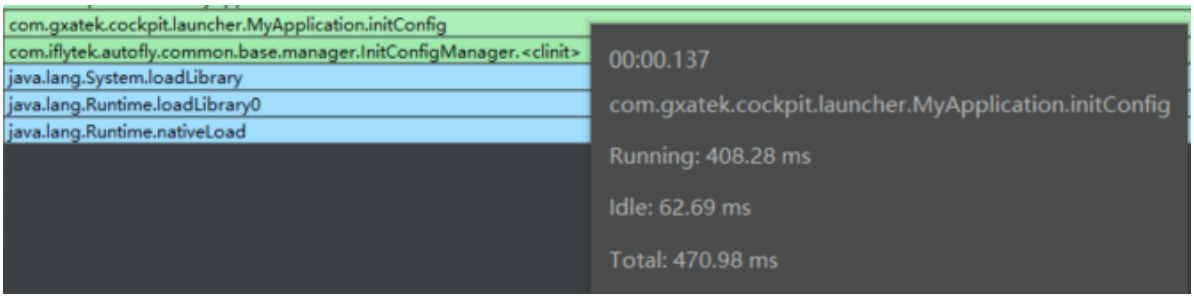
Launcher有三个Fragment：MapFragment、VpaFragment和CarFragment。默认启动的VpaFragment，其他两个Fragment不显示，相关资源不要在启动界面加载。

1.3 找退出开机动画的时机

由于当前项目开机动画由QNX来控制，Android原生开机动画被QNX图层覆盖，因此Launcher需要

2.MyApplication耗时分析

2.1 initConfig();



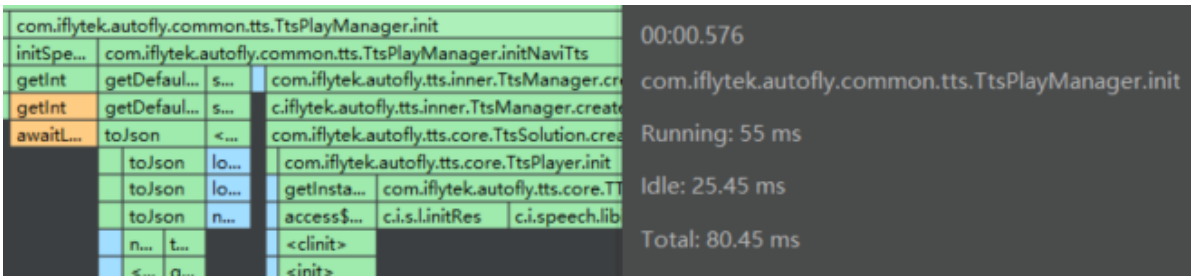
初始化配置耗费470ms, 其中加载so库耗时231ms, 初始化资源花费212ms.

此部分主要涉及IO操作：

建议：

- so库如果属于VPA不相关的库，延迟加载该库。
- initRes()方法中有大量地图相关的配置文件和资源加载，尽量延后到VPA启动之后或者在加载MapFragment的时候在初始化这部分资源。

2.2 TtsPlayManager.init(this);



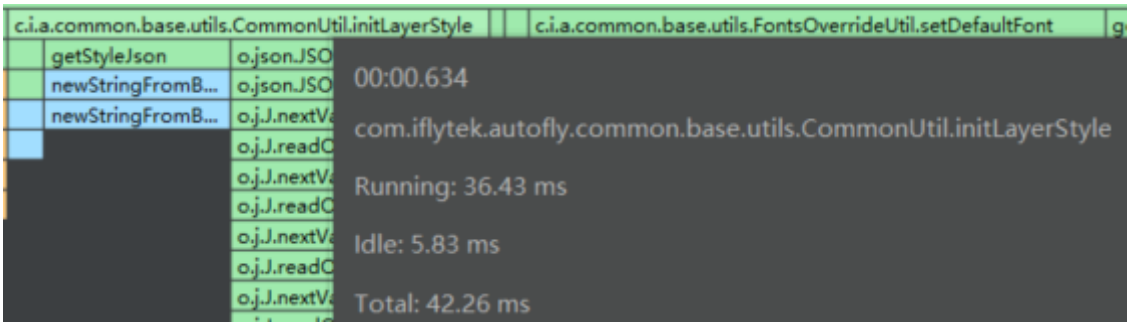
TtsPlayManager的初始化方法主要实现了初始speaker和初始化naviTts.

其中初始speaker消耗8.6ms, 初始化navi Tts消耗70.2ms.

建议：

- navi Tts初始化推迟加载

2.3 CommonUtil.initLayerStyle();

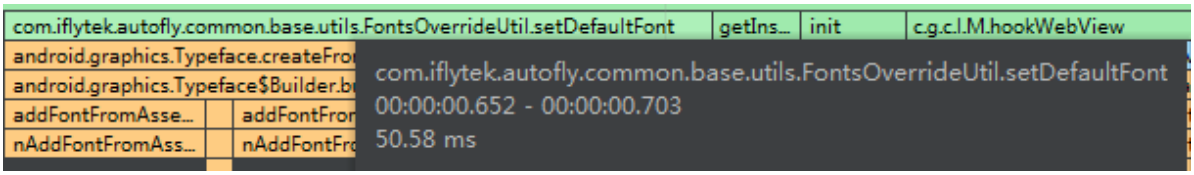


存解析json配置文件，消耗42ms.

建议：

- 如果和地图相关的初始化是否可以和地图初始化放一起。

2.4 FontsOverrideUtil.setDefaultFont();



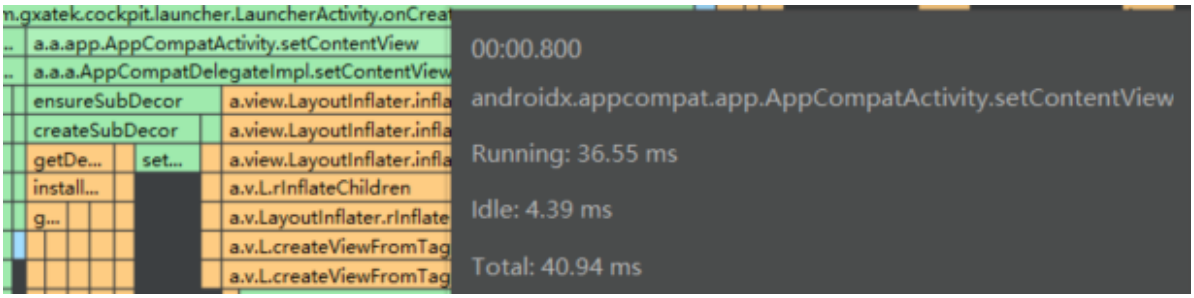
加载三类字体，每个耗时17ms左右

建议：

- 此处只是列出耗时，如果却有必要，可不优化。

3. LauncherActivity耗时分析

3.1 加载布局文件



主界面加载布局文件消耗41ms。此处看加载布局文件耗时不算多，确实需要优化可以考虑使用异步加载布局。

建议：

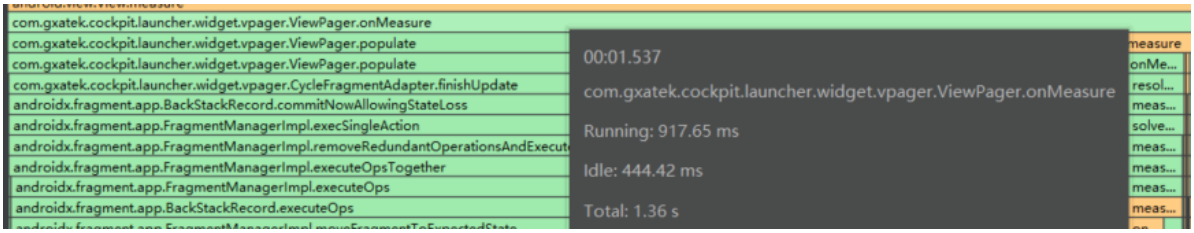
- 异步加载布局

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);

    new AsyncLayoutInflater(this).inflate(R.layout.activity_main, null, new
AsyncLayoutInflater.OnInflateFinishedListener() {
        @Override
        public void onInflateFinished(@NonNull View view, int resid, @Nullable
ViewGroup parent) {
            setContentView(view);
        }
    });
    .....
}
```

- **注意此处或者当VPA fragment加载完成之后考虑一下是否可以通知开机动画退出？**内容的展示后面加载出来在更新。

3.2 自定义viewpage耗时

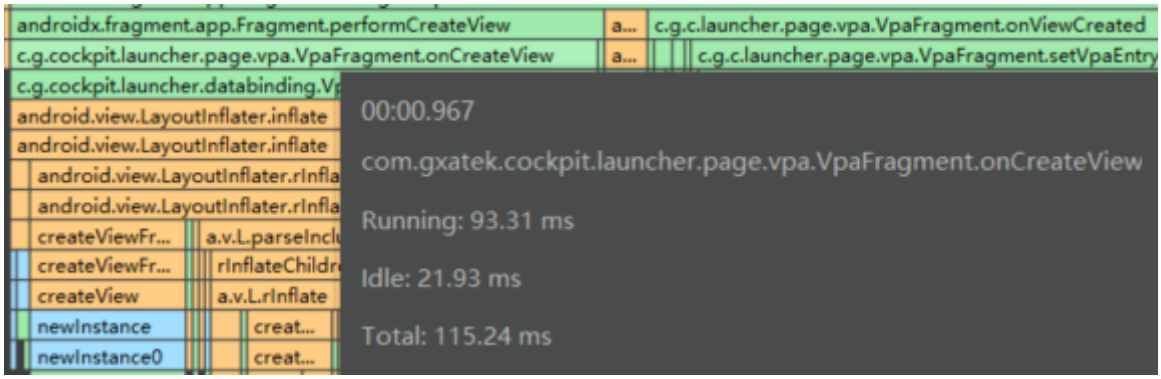


从图中可以看到自定义ViewPage在测量阶段耗时1.36s。(后面细化分析每个fragment启动耗时)

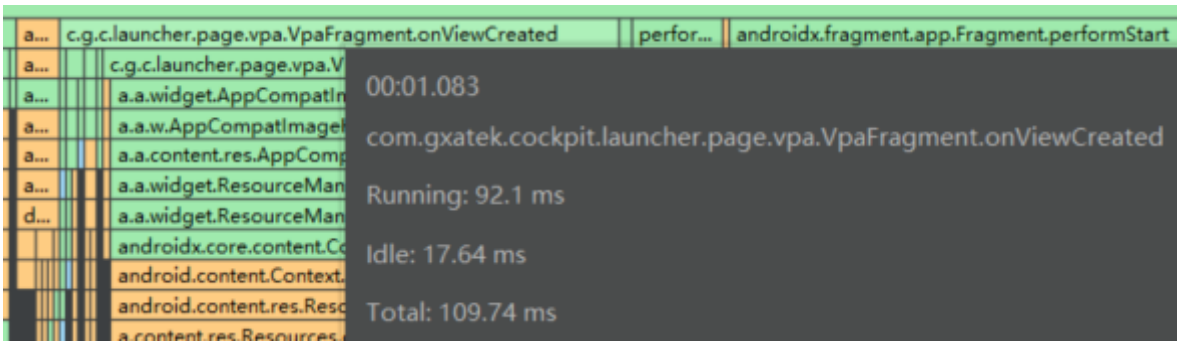
从下面细化调用栈看，ViewPage中有加载三个fragment：MapFragment、VpaFragment和CarFragment。

3.2.1 VPA Fragment加载耗时

VPA fragment由两部分内容构成，如下：



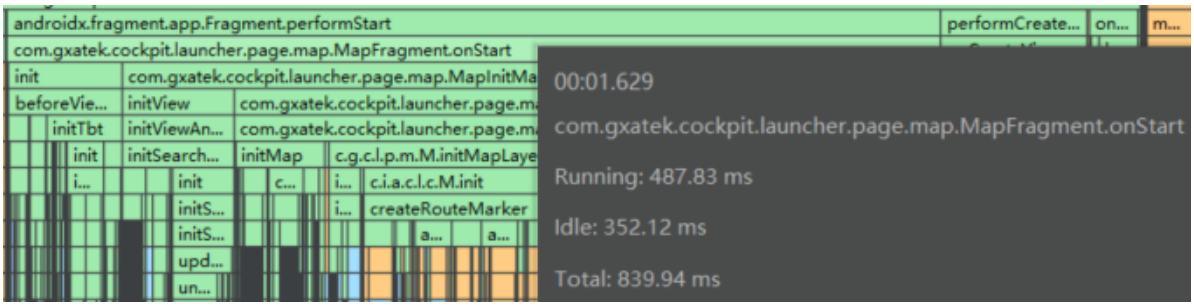
第一部分加载布局耗时115ms。



第二部分加载布局耗时110ms。

看了代码以上两部分耗时主要是加载布局文件耗时，可以暂时不优化。

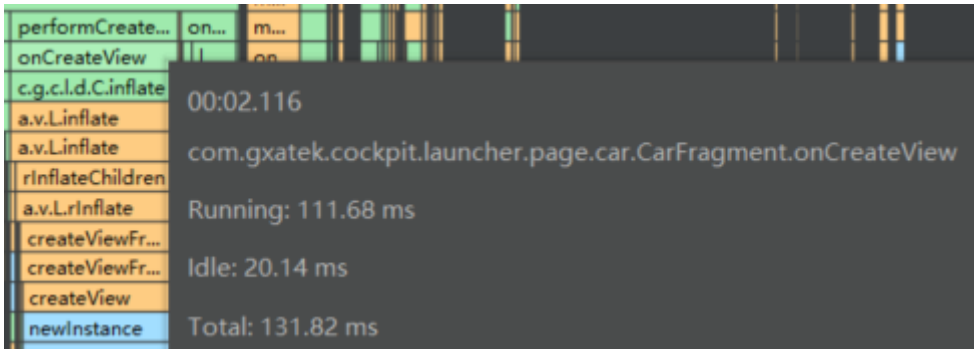
3.2.2 MapFragment加载耗时



从上图中可以看到map fragment加载耗时840ms, 其中初始化消耗103ms, 加载显示地图资源(showMap())消耗736ms。

在showMap()方法中主要完成两部分工作：初始化view(101ms)和初始化地图资源(635ms)。

3.2.3 CarFragment加载耗时



从图中可以看出CarFragment主要完成的是布局加载，可以暂时不优化。

3.2.4 fragment加载总结

结合当前交付场景：Launcher启动之后只需要展示VPA Fragment，因此需要FO实现viewpage懒加载机制。

我看当前代码是自定义viewgroup来实现的viewpage，是否可以直接使用viewpage2自带懒加载效果，或者结合viewpage+fragment，利用setUserVisibleHint(boolean isvisible)和isViewCreated()来实现懒加载效果。

从上面分析看，如果只加载VPA Fragment,不加载MapFragment和Car Fragment，应该至少可以优化 **1s** 以上的启动耗时。

4. 通知QNX退出开机动画

```
// com.gxatek.cockpit.launcher.LauncherActivity
@Override
protected void onStart() {
    super.onStart();
    Logger.d(TAG, "onStart, CarPowerManager.connect()");
    mPower = CarPowerManager.getInstance(getApplicationContext(),
carPowerEventListener);
    mPower.connect();
    if (curIndex == 0) {
        ((MapFragment) currentFragment).showMap();
    }
}

CarPowerEventListener carPowerEventListener = new CarPowerEventListener() {
    ......

    @Override
    public void onCarPowerServiceConnected() {
        Logger.d(TAG, "CarPowerManager onCarPowerServiceConnected,
CarPowerManager.exitAnimationReq");
        mPower.exitAnimationReq();
    }

    ....
};
```

从上面代码可以看出，Launcher通知开机动画退出的时机是：当所有布局和资源加载完成(onCreate())之后，再去绑定CarPower服务，绑定成功之后通知qnx退出开机动画。

正常情况下绑定服务也会耗时200-300ms。

因此在此环节的优化建议：

- 在onCreate()中加载完布局之后就绑定CarPower服务
- 当VPA Fragment布局加载完成之后就通知QNX退出开机动画。

5.其他优化建议

5.1 Arouter插件优化

我看项目代码中有用到Arouter插件，可以在项目中集成Arouter Gradle 插件来实现自动注册功能。

6. Using the custom gradle plugin to autoload the routing table

```
apply plugin: 'com.alibaba.arouter'

buildscript {
    repositories {
        mavenCentral()
    }

    dependencies {
        // Replace with the latest version
        classpath "com.alibaba:arouter-register:?"
    }
}
```

Optional, use the registration plugin provided by the ARouter to automatically load the routing table(power by AutoRegister). By default, the ARouter will scanned the dex files . Performing an auto-registration via the gradle plugin can shorten the initialization time , it should be noted that the plugin must be used with api above 1.3.0!

5.2 EventBus Apt优化

*EventBus 3.0*之前的版本是没有索引的，检索订阅方法是通过反射获取的。我们都知道反射的效率令人堪忧，如果频繁地调用的话，肯定会对程序的性能造成影响。而*greenrobot*也意识到这个问题，所以在*EventBus 3.0*版本新增一个索引的功能，它主要是通过在编译期处理，生成订阅者和订阅方法的对应关系并缓存起来，从而在程序运行时能快速索引。

因为生成索引是在编译期的，所以需要添加一些配置。首先在project下build.gradle添加：

```
buildscript {
    dependencies {
        classpath 'com.neenbedankt.gradle.plugins:android-apt:1.8'
    }
}
```

然后在app或lib下的build.gradle添加apt插件和 `EventBus apt` 工具：

```
apply plugin: 'com.neenbedankt.android-apt'

dependencies {
    apt 'org.greenrobot:eventbus-annotation-processor:3.0.1'
}
```

最后，也是在app或lib下的build.gradle apt参数：

```
apt {
    arguments {
        eventBusIndex "com.leo.eventbus.sample.SampleBusIndex" // 生成索引类，包名和类名可自定义
        verbose "true" // 是否打印编译调试日志
    }
}
```

5.3 SharedPreference优化

如果启动时加载的sharedpreference过大，可以拆分sharedpreference，将启动需要的配置分成一个sharedpreference，不相关的配置分成另一个sharedpreference。降低启动依赖配置文件的加载和解析时间。

参考文档：

1.《你知道android的MessageQueue.IdleHandler吗？》

2.《alibaba/ARouter：A framework for assisting in the renovation of Android componentization (帮助 Android App 进行组件化改造的路由框架) (github.com)》

3.《EventBus源码详解（二）：进阶使用》