[SCA] Application生命周期分发方案

版本	修订日期	修订者	修订内容
1.0.1	2021.09.28	lanjing.zeng@shopee.com	结合启动框架，配合使用 方案补充说明
1.0.0	2021.09.17	lanjing.zeng@shopee.com	初版

1 背景

目前组内大部分项目都是基于组件化编程，部分业务子Module需要在Application的生命周期中执行一些初始化、反初始化的操作。如果在这些业务子Module中自己单独定义一个Application类，在APK编译时会因为无法合并Manifest.xml文件而导致编译失败。因此目前普遍采用的方法是：

壳工程【app】Module以Implementation、api的形式依赖各个业务子Module，同时在【app】Module中定义一个顶层的Application类，在这个类中手动对业务子Module进行初始化。

这会带来两个问题：

1.【app】Module与业务子Module耦合；

2.当需要去掉某个业务子Module时，需要同时去掉【app】Module中相应的初始化、反初始化代码；这就要求维护者需要清楚的知道哪行初始化代码是跟哪个业务子Module对应的，随着项目迭代业务子Module逐渐增多，会变得不好维护；

```
public interface IApplicationLifecycle {
        int priority();

    void onCreate();

    void onConfigurationChanged(@NonNull Configuration newConfig);

    void onTerminate();

    void onTrimMemory(int level);
}
```

2 生命周期分发方案

关于这个问题，目前有如下几个实现方式：

2.1 借助Arouter实现

1. 在【Base】Module中定义一个IApplicationLifecycle的接口，需要监听Application生命周期的业务子Module同时实现这个接口与Arouter的IProvider接口；
2. 定义一个生命周期的管理类ApplicationLifecycleManager，并在其中以register的方式维护一个List<IApplicationLifecycle> list；
3. 业务子Module重写IProvider接口的init()，调用ApplicationLifecycleManager的api实现注册；
4. 在【Base】Module的Application的onCreate()中，获取业务子Module的IProvider，此时底层会自动调用init()方法完成注册；
5. 在【Base】Module的Application各个生命周期方法中，执行ApplicationLifecycleManager实现的每个接口方法；

优点：目前大部分项目都依赖了Arouter，不用在引入其他依赖库；

缺点：依赖Arouter，且需要在Arouter初始化完成后生效；

目前项目的做法：借助Arouter，但是没有定义IApplicationLifecycle接口，相当于只能在Application的onCreate()中执行，不能监听其他生命周期。

2.2 注解处理器+类扫描的形式实现

1. 在【Base】Module中定义一个IApplicationLifecycle的接口，需要监听Application生命周期的业务子Module实现这个接口，同时为该类添加一个注解；

2. 通过注解处理器为添加注解的类在同一个包中生成相应的代理类(在同一个包是为了后面好扫描)；
3. 定义一个生命周期的管理类ApplicationLifecycleManager，并在其中以register的方式维护一个List<IApplicationLifecycle> list；
4. ApplicationLifecycleManager也实现IApplicationLifecycle的接口，每个方法的实现就是以遍历List item的形式通知业务子Mudule执行相应的接口方法；
5. 在【Base】Module的Application的onCreate()中，执行扫描，找到生成的相应代理类，实例化后在ApplicationLifecycleManager中进行注册；
6. 在【Base】Module的Application各个生命周期方法中，执行ApplicationLifecycleManager实现的每个接口方法；

优点：使用时简单；

缺点：需要在应用启动时扫描整个dex文件；

2.3 插桩形式实现

该方法与2类似，只是注册时找到类的方式不一致。

1. 在【Base】Module中定义一个IApplicationLifecycle的接口，需要监听Application生命周期的业务子Module实现这个接口；
2. 定义一个生命周期的管理类ApplicationLifecycleManager，并在其中以register的方式维护一个List<IApplicationLifecycle> list；
3. ApplicationLifecycleManager也实现IApplicationLifecycle的接口，每个方法的实现就是以遍历List item的形式通知业务子Mudule执行相应的接口方法；
4. 通过ASM插桩技术，从class文件中找到实现了IApplicationLifecycle的类，实例化后在ApplicationLifecycleManager中进行注册；
5. 在【Base】Module的Application各个生命周期方法中，执行ApplicationLifecycleManager实现的每个接口方法；

优点：使用时比较方便；

缺点：实现相对会复杂，且因为要插桩，会修改字节码；

2.4 借助清单文件实现

1. 在【Base】Module中定义一个IApplicationLifecycle的接口，需要监听Application生命周期的业务子Module实现这个接口；
2. 定义一个生命周期的管理类ApplicationLifecycleManager，并在其中以register的方式维护一个List<IApplicationLifecycle> list；
3. ApplicationLifecycleManager也实现IApplicationLifecycle的接口，每个方法的实现就是以遍历List item的形式通知业务子Mudule执行相应的接口方法；
4. 在【Base】Module的Application的onCreate()中，从清单文件中读取配置信息，配置信息中包含有类的全限定名，通过反射实例化后在ApplicationLifecycleManager中进行注册；
5. 在【Base】Module的Application各个生命周期方法中，执行ApplicationLifecycleManager实现的每个接口方法；

优点：实现简单；

缺点：通过类名反射实例化，如果修改了类包名或类名没有同步修改清单文件会导致实例化失败；

3 生命周期分发的具体实现

3.1 方案选择及优化

选择使用2.4小节的方案来实现生命周期的分发，但是这一方案存在一个比较明显的问题：那就是如果修改了类包名或类名没有同步修改清单文件会导致实例化失败。

针对这一问题，主要通过以下优化方式来解决：

1. ~~自动配置Manifest.xml，但发现实际开发时不可行，详情如下~~

   > 尝试通过以下方式来自动配置Manifest.xml清单文件：
   >
   > 1.ASM在编译时扫描实现了IApplicationLifecycle的所有实现类；
   >
   > 2.通过gradle插件将这些实现类的信息自动添加至Menifest.xml中；
   >
   > 实际这种方式不可行，原因如下：
   >
   > Android构建apk其实是执行一系列Task的过程，Task的顺序大致为：先执行处理Menifest的Task，然后执行java文件-->class文件-->dex文件的Task。ASM类扫描操作是在Java文件编译成class文件后才执行的，如果当类扫描完成后再去处理Menifest文件，此时修改的Menifest文件并不会在最终的apk中生效。
2. 这一点配合第3点一起使用，IApplicationLifecycle的实现类添加特定注解，用于标记这个类是生效的；(因为可能存在定义了实现类但实际却不需要使用，又不想删除相关代码的情况)
3. 预处理错误：由于以上第1点自动配置无法实现，因此增加一个编译期的预处理，来预先发现错误，具体的：通过ASM扫描出IApplicationLifecycle的所有添加了注解的实现类，同时解析出现在Manifest.xml中配置的类信息，若二者不匹配则抛出异常终止编译流程；

4. ~~（如果做了第三点，这一点可以去掉了）增加运行时异常处理，通过反射实例化时若找不到对应的类直接在代码中抛出异常，达到快速发现错误的目的；~~

3.2 实现细节

3.2.1 定义关键接口及注解

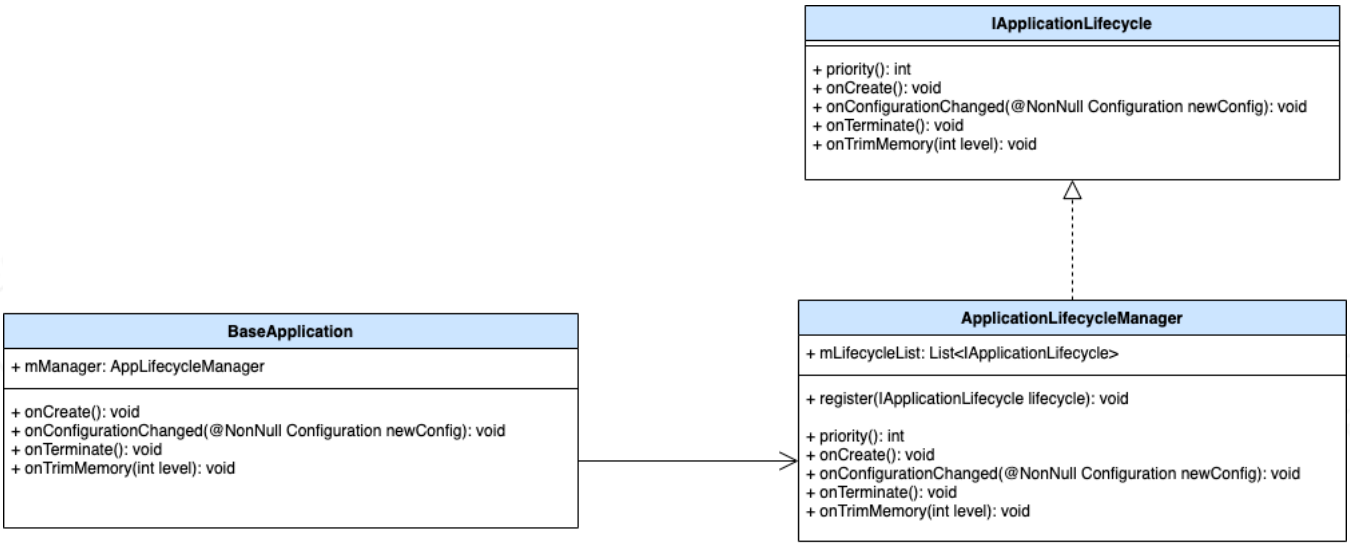
接口：

```
public interface IApplicationLifecycle {
    default int priority() {
        return 0;
    }
    void onCreate();

    default void onConfigurationChanged(@NonNull Configuration newConfig) {
    }

    default void onTerminate() {
    }

    default void onTrimMemory(int level) {
    }

    default void onLowMemory() {
    }
}
```

注解：

```
@Target({ElementType.TYPE})
@Retention(RetentionPolicy.CLASS)
public @interface ApplicationLifecycle {

}
```

3.2.2 生命周期实现管理类



生命周期的管理类ApplicationLifecycleManager：

- 该管理类以register的方式维护一个List<IApplicationLifecycle> list，注册时先读取Manifst.xml中meta节点中IApplicationLifecycle实现类的配置信息，然后通过反射实例化后再进行注册。
- 实现IApplicationLifecycle接口，在相应的接口方法中对生命周期进行分发；
- 在BaeApplication各个生命周期方法中，调用管理类的相应接口方法；

3.2.3 清单文件注册

在各个业务子Module中的Manifest.xml中以meta-data标签的形式注册IApplicationLifecycle接口的实现类信息。例如：

```
</application>
        <meta-data android:name="com.shopee.sc.lifecycledispatch.sample.SampleClassB" android:value="com.
shopee.sc.LifecycleDispatch" />
    <meta-data android:name="com.shopee.sc.lifecycledispatch.sample.SampleClassA" android:value="com.shopee.sc.
LifecycleDispatch" />
</application>
```

APK构建是会自动将各个子Module中的清单文件进行合并成一分，并打包在APK中。运行时可动态从清单文件中读取这些配置信息。

3.2.4 编译时预处理异常

1.在系统构建APK过程中，处理完Manifest后通过gradle插件，解析Manifest.xml中的meta-data节点，读取出文件中注册的AppicationLifyCycleClass信息；

2.定义Transfrom，通过ASM对字节码进行扫描，扫描出同时实现了IApplicationLifecycle接口且标注了@ApplicationLifecycle注解的实现类；

3.将读取的注册信息和扫描的信息进行比对，若不一致则终止构建流程；

4 结合启动框架

Application生命周期分发主要是用于解耦【app】Module对业务子Module的依赖，分发后初始化任务将分布在各个子Module中，启动框架的使用可能会受到限制：Module之间无法获取彼此的任务，也就无法创建任务之间的依赖关系，因此启动框架需要对这种情况做一下兼容。

4.1 分析

现状：

任务链：启动框架以Project来构造任务链，Project内部通过TaskFactory作为创建此任务链中任务的工厂类；

任务链内部：通过建造者模式，先创建一个Project.Builder对象，通过这个Builder对象添加任务、设置任务依赖关系，任务链内部支持Task、TaskName参数；

任务链外部：Project本身也是Task接口，可通过dependOn(Task task)、behind(Task task)设置任务链头尾的依赖关系；

普通任务：直接通过Task构造函数创建任务，通过dependOn(Task task)、behind(Task task)设置任务的依赖关系；

设置锚点：只能通过AnchorManager.getInstance().addAnchor(Task task) Task形式设置锚点；

开启任务：只能通过AnchorManager.getInstance().startTask(Task task) Task形式开启任务；

问题及解决方式：

将Applciation的生命周期分发后，初始化任务可能分布在各个子Module中，而各个子Module不存在相互依赖的关系，可能会带来如下问题：

- 不能将Module之间的任务链接成一个任务链；

----->目前启动框架通过建造者模式创建一个任务链即Project，底层考虑新增一个Project.newBuilder()方法，用于通过使用ProjectName获取Project后，调用newBuilder()方法返回一个新的Project.Builder对象，这样可以再次从这个任务链中添加新的任务；

- Module之间不能通过Task添加依赖关系；

----->这个问题考虑通过启动框架增加支持TaskName来添加任务来解决；

- 不能通过Task设置锚点、开启任务:

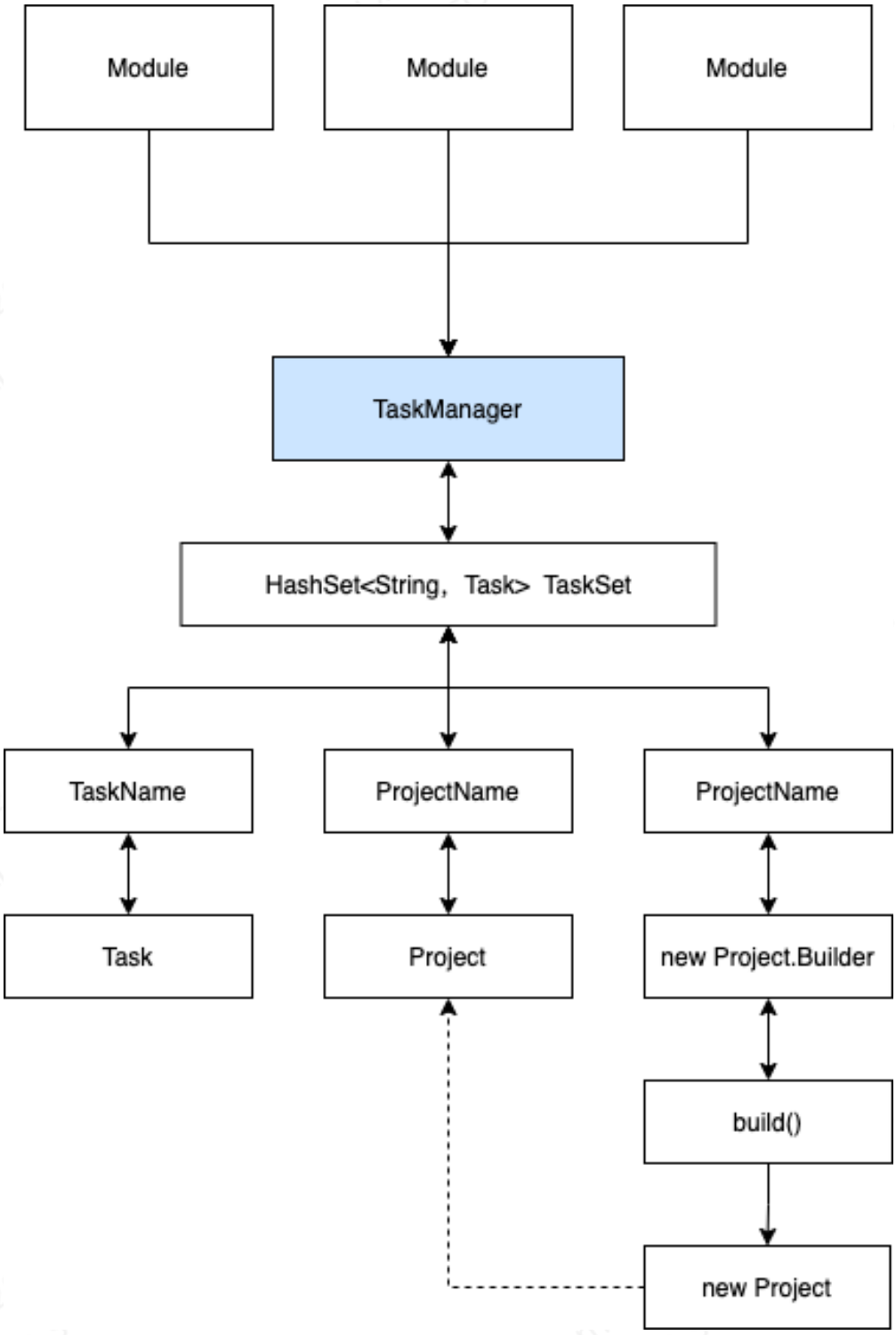
----->这个问题考虑通过启动框架增加支持TaskName来设置锚点、开启任务来解决；

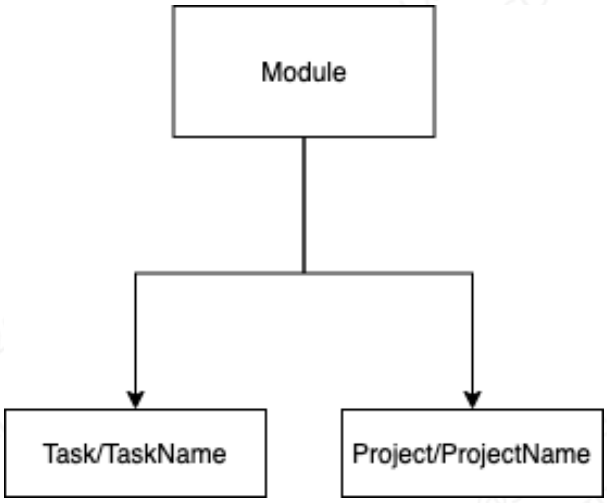
4.2 主要变更点

启动框架主要变更点如下：

- 创建一个TaskManager类，该类维护TaskName与Task，ProjectName与Project的对应关系；
- 修改Task的构造方法，构造Task时将这个Task添加进TaskManager中。Project继承自Task，即一个Project其实也是一个Task，ProjectName就是TaskName，构造Project时会执行Task的构造方法，这个Project也将被添加进TaskManager中；
- TaskManager支持通过TaskName获取Task；
- TasManager支持通过ProjectName获取一个Project，和通过TaskName获取Task一样，只是获取Task后判断如果是Project对象将其强转为Project即可；
- TasManager支持通过ProjectName从旧的Project获取一个新的ProjectBuilder对象(上面提到的Project.newBuilder()方法)，同时通过这个ProjectBuilder构造出来的新Project将覆盖旧的Project对象；
- 创建依赖关系、添加锚点、开启任务执行等API增加支持TaskName参数，从TaskManager中通过TaskName取出与之对应的Task，其他逻辑可直接复用使用Task参数方法的逻辑；
- 其他需要同步修改的地方，例如需要清空TaskManager中的任务；

4.3 关键部分调用关系图


Module
Module
Module
TaskManager
HashSet<String, Task> TaskSet
TaskName
ProjectName
ProjectName
Task
Project
new Project.Builder
build()
new Project



任务之间创建依赖关系：

以Project（Project也是一个Task）、Task的形式创建任务之间的依赖关系，依赖关系可同时以Task、TaskName的形式指定；

Task主要用于创建Module内部任务的依赖关系，TaskName主要用于Module之间创建任务的依赖关系；

这样可以将各个Module(包括BaseModule)的Task都可以建立起依赖关系了。

关于启动任务：

如果没有依赖的起始任务，建议构造一个空Task，以这个空的Task作为所有任务的起点，这样启动这个空的Task后，其他的所有任务都也将启动起来，例如：

UITHREAD_TASK_A --> PROJECT_9_start(1576044344839) --> TASK_90 --> TASK_91 --> PROJECT_9_end(1576044344839)

UITHREAD_TASK_A --> PROJECT_4_start(1576044344837) --> TASK_40 --> TASK_41 --> TASK_42 --> TASK_43 --> PROJECT_4_end(1576044344837)

UITHREAD_TASK_A --> PROJECT_3_start(1576044344836) --> TASK_30 --> TASK_31 --> TASK_32 --> TASK_33 --> PROJECT_3_end(1576044344836)

UITHREAD_TASK_A --> UITHREAD_TASK_B