Apk编译过程的简单回顾

一个Apk解压后有些什么?

```
chenlong@chenlong-PC:~/apkbuild/originApk
⇒ ll
total 6536
-rw-r--r--   1 chenlong  staff     2436 Jun 22 21:18 AndroidManifest.xml
drwxr-xr-x   5 chenlong  staff      170 Jun 22 21:20 META-INF
-rw-r--r--   1 chenlong  staff  3117688 Jun 22 21:18 classes.dex
drwxr-xr-x  29 chenlong  staff      986 Jun 22 21:20 res
-rw-r--r--   1 chenlong  staff   217864 Jun 22 21:18 resources.arsc
```

- AndroidManifest.xml:二进制的清单文件
- classes.dex:编译后的代码文件
- res:编译后的资源文件
- resource.arsc:资源索引表

gradle 与 buildTools

> gradle是构建工具，不是编译工具，gradle驱动了buildTools编译
> 我们的代码。

不包含混淆的情况下：

- aapt：
  编译资源 res/ -> res/+resources.arsc+R.java
  生成apk文件
- javac：编译代码 *.java* -> .class
- dx：[*.class] -> classes.dex
- zipalign：优化apk文件，进行4字节对齐
- jarsigner：签名

编译流程

Step 1:生成R.java 文件

aapt package \ #打包资源文件
-f \ #强制覆盖已有文件
-m \ #使R文件在-J参数指定的位置生成
-S res \ #资源目录
-J gen \ #R.java的位置
-I $ANDROID_HOME/platforms/android-23/android.jar \ #base-package

-M AndroidManifest.xml #清单文件的路径

Step 2:编译代码

javac -classpath \ #添加依赖包，多个jar包用:分割
$ANDROID_HOME/platforms/android-23/android.jar \ #sdk-23
-source 1.7 -target 1.7 \ #指明源码版本和字节码版本
-d ./build \ #编译后的class文件的路径
./java/com/haizhi/oa/buildtest/*.java \ #源码1，这是我们写的Activity
./gen/com/haizhi/oa/buildtest/R.java #源码2，R.java

Step 3:编译为dex文件

dx --dex --output=classes.dex . #指定输出为classes.dex 输入为当前目录

Step 4: 打包所有的资源文件

aapt package \
-f \
-S res \
-I $ANDROID_HOME/platforms/android-23/android.jar \
-M AndroidManifest.xml \
-F test.apk.u #生成apk文件

Step 5:classes.dex文件加入apk中

aapt add -f test.apk.u classes.dex

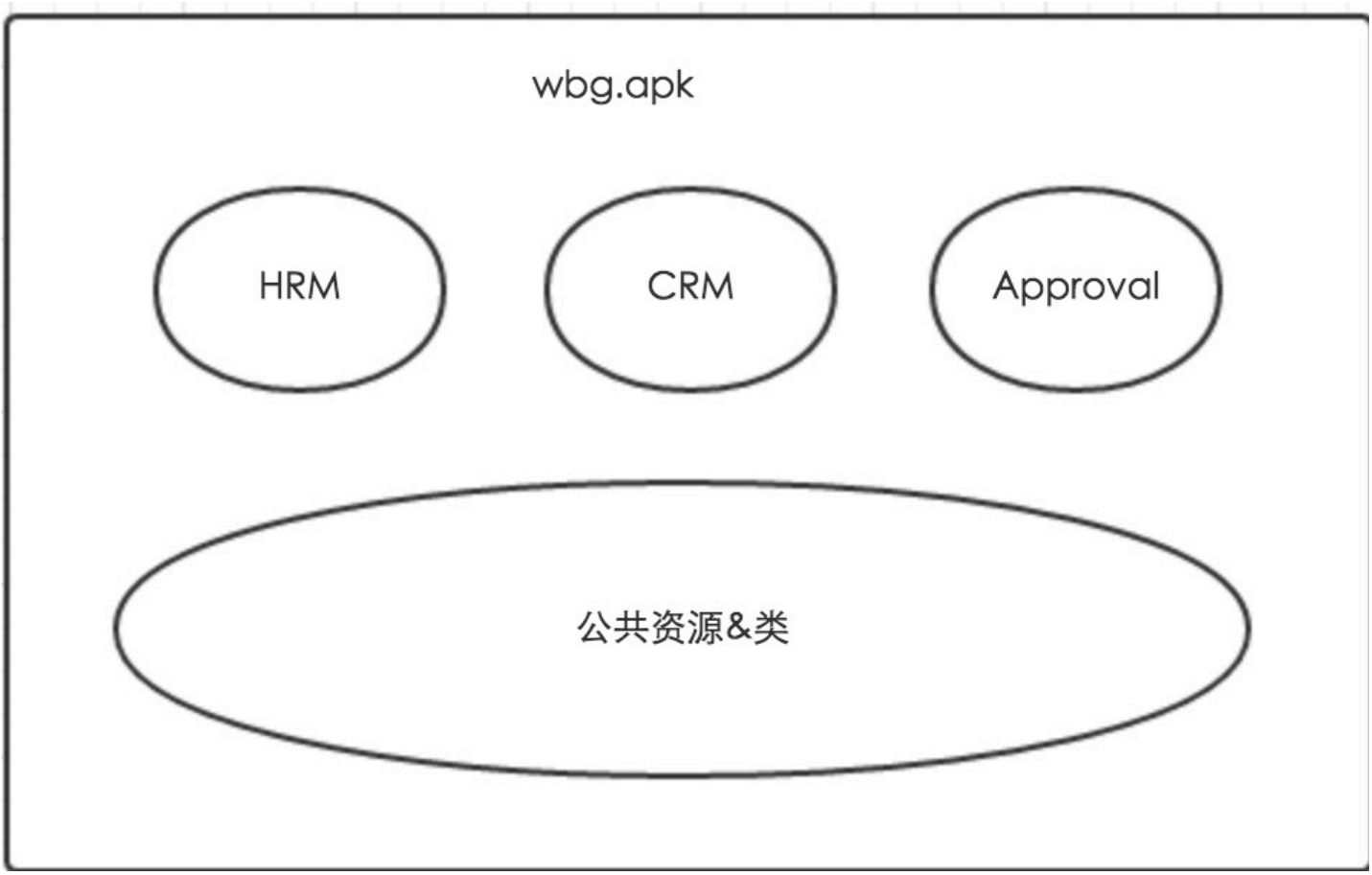
Step 6: 签名，对齐

签名:
jarsigner -storepass **密*码** -keystore ../chenlong.keystore test.apk.u chenlong

对齐:
zipalign 4 test.apk.u test.apk

我们需要解决什么问题?

理想的宿主和插件的关系



那么问题来了

1. 如何让插件在编译时/运行时引用到公共的资源?
2. 如何让插件之间资源id不会冲突?
3. 如何让插件在编译时/运行时引用到公共的类?

因为运行时的处理，我也还没有完全搞清楚，所以我们今天先来看看编译时

阅读Aapt的代码

工程结构

这次阅读的是aosp项目中android-6.0.1_r66的代码。

aapt位于/platform/frameworks/base/tools/aapt

```
chenlong@ubuntu-server:~/aosp/aosp/frameworks/base/tools/aapt|aapt⚡
⇒ ls -a
.               CacheUpdater.h      OutputSet.h          StringPool.cpp
..              Command.cpp         Package.cpp          StringPool.h
AaptAssets.cpp  ConfigDescription.h pseudolocalize.cpp   Symbol.h
AaptAssets.h    CrunchCache.cpp     pseudolocalize.h     tests
AaptConfig.cpp  CrunchCache.h       Resource.cpp         WorkQueue.cpp
AaptConfig.h    DirectoryWalker.h   ResourceFilter.cpp   WorkQueue.h
AaptUtil.cpp    FileFinder.cpp      ResourceFilter.h     XMLNode.cpp
AaptUtil.h      FileFinder.h        ResourceIdCache.cpp  XMLNode.h
AaptXml.cpp     Images.cpp          ResourceIdCache.h    ZipEntry.cpp
AaptXml.h       Images.h            ResourceTable.cpp    ZipEntry.h
Android.mk      IndentPrinter.h     ResourceTable.h      ZipFile.cpp
ApkBuilder.cpp  Main.cpp            SdkConstants.h       ZipFile.h
ApkBuilder.h    Main.h              SourcePos.cpp
Bundle.h        NOTICE              SourcePos.h
```

我们同时还需要/platform/frameworks/base/include/androidfw的一些头文件

```
chenlong@ubuntu-server:~/aosp/aosp/frameworks/base/include/androidfw|aapt⚡
⇒ ll
总用量 160
-rw-rw-r-- 1 chenlong chenlong  4375 Sep 22 12:38 AssetDir.h
-rw-rw-r-- 1 chenlong chenlong 10106 Sep 22 12:38 Asset.h
-rw-rw-r-- 1 chenlong chenlong 14115 Sep 22 12:38 AssetManager.h
-rw-rw-r-- 1 chenlong chenlong  6803 Sep 22 12:38 AttributeFinder.h
-rw-rw-r-- 1 chenlong chenlong  4398 Sep 22 12:38 BackupHelpers.h
-rw-rw-r-- 1 chenlong chenlong  2591 Sep 22 12:38 ByteBucketArray.h
-rw-rw-r-- 1 chenlong chenlong  5747 Sep 22 12:38 CursorWindow.h
-rw-rw-r-- 1 chenlong chenlong  1420 Sep 22 12:38 misc.h
-rw-rw-r-- 1 chenlong chenlong  3256 Sep 22 12:38 ObbFile.h
-rw-rw-r-- 1 chenlong chenlong 67266 Sep 22 12:38 ResourceTypes.h
-rw-rw-r-- 1 chenlong chenlong  2913 Sep 22 12:38 StreamingZipInflater.h
-rw-rw-r-- 1 chenlong chenlong  2179 Sep 22 12:38 TypeWrappers.h
-rw-rw-r-- 1 chenlong chenlong  5248 Sep 22 12:38 ZipFileRO.h
-rw-rw-r-- 1 chenlong chenlong  2821 Sep 22 12:38 ZipUtils.h
```

Aapt编译的几个阶段

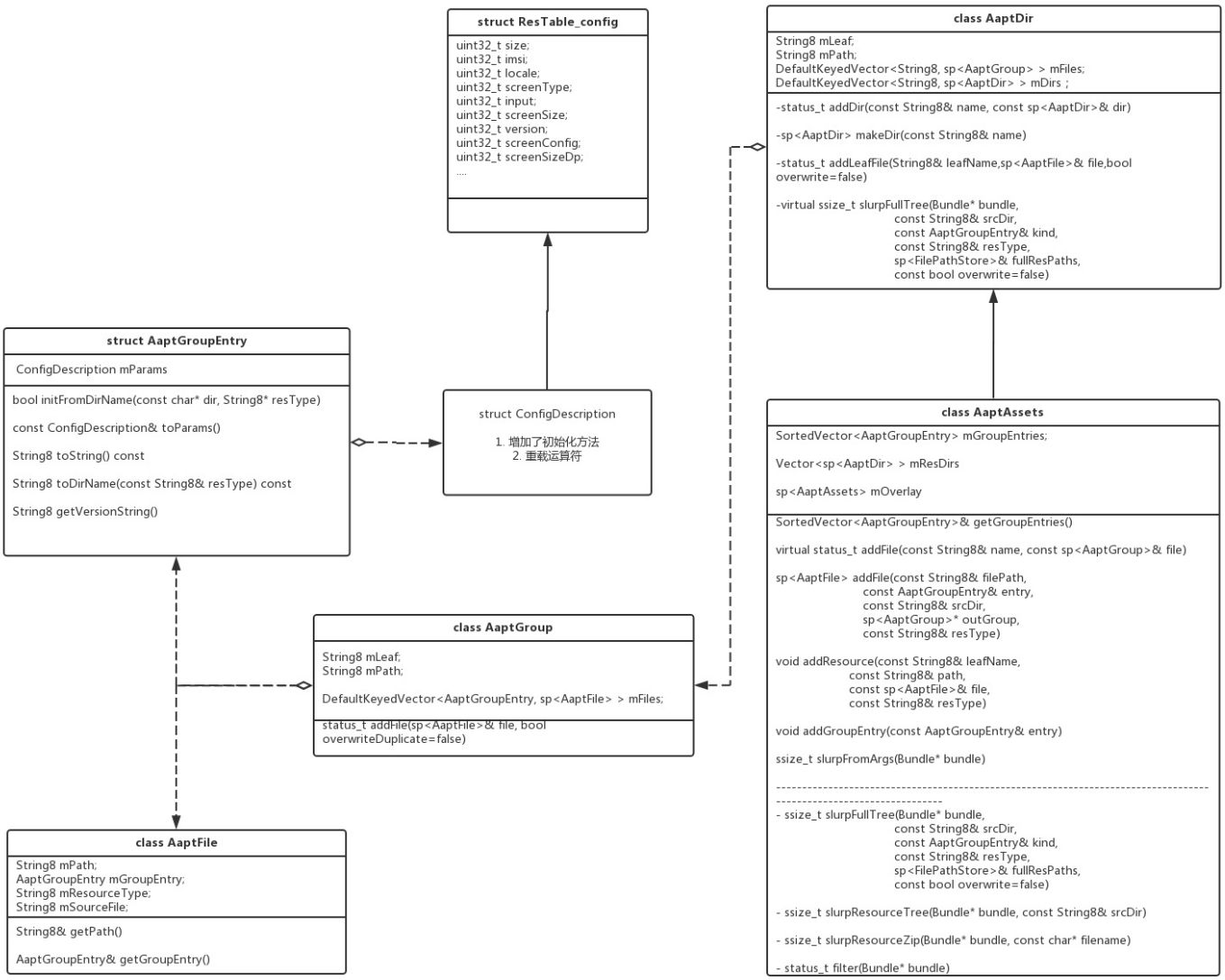
1. 收集资源
2. 编译资源
3. 生成资源索引表
4. 生成R.java和apk文件

其中，编译资源又可以细分为以下几个步骤

1. 编译values
2. 分配资源id
3. 编译xml文件(系统资源是如何引用的？)
4. 编译AndroidManifest.xml

---

1. 收集资源



数据结构

- AaptAssets 描述了一个res/文件夹。
- AaptDir 描述了一个资源文件夹，这里的资源文件夹不仅指的是res/下的layout/，values/文件夹，同时也包括layout/下的子文件夹，assets/目录下的子文件等。
- AaptGroup 描述了一个资源集合。
- AaptFile 描述了一个资源文件。
- AaptGroupEntry 描述了资源的配置信息。

收集资源的流程

- 收集AndroidManifest.xml文件

- 收集assets资源
- 收集res/下的资源
- 收集直接指定的raw资源 aapt 所有参数解析结束后，剩下的路径名全部被解析成raw类型资源
- 校验和过滤

需要关注的点

1. 通常我们写的layout-xhdpi,layout-xxhdpi在编译时时如何处理的

```
/**
* 根据文件夹的名字获取资源的类型以及资源的配置信息
* 注意一点，虽然我们通常在res目录下会创建同类型资源的多个
* 文件夹，例如，res/layout,res/layout-xhdpi,res/layout-xxhdpi
* 但是，在收集资源的时候，这些文件夹都被认为是layout类型，通
* 过 ‘-’分割后，后面的字符串会被处理成资源的配置信息，写入
* AaptGroupEntry中。同时，通过'-'连接多个描述信息需要严格按照
* 文档中定义的顺序，此处初始化AaptGroupEntry时会对顺序做校验
*/
```

---

2. 编译资源

数据结构

- ResourceTable resources.arsc
- Package 需要编译的资源包 AaptAssets
- Type 资源类型 AaptDir
- ConfigList 单个资源 AaptGroup
- Entry 具体的资源项 AaptFile

---

1). 编译values

需要关注的点:
1) 标签的作用不仅仅是指定一个资源的id，更重要的是让这个资源
可以被其他package引用

2）Bag资源。对于有层级的资源，通过bag描述。

```
/**
 * bag类型
 * <bag>,<attr>,<style>,<plugrals>,<array>,
 * <string-array>,<integer-array>
 *
 * <array>,<string-array>,<integer-array>
 * 标签生成的都是array类型的资源。
 * <array>类型的资源下的bag 的key,类似下述结构
 *       ^index_{idex}
 * 对于bag类型的资源,只允许出现<item>子标签
 */
```

Bag资源的特点

1. 有parent

```
<style name="AppTheme.BaseTheme">
    ...
</style>
```

或者

```
<style name="AppTheme" parent="@*com.haizhi.oa:style/BaseTheme">
    ...
</style>
```

1. 有子节点

```
<array name="test_arrays">
    <item>测试1</item>
    <item>测试2</item>
    <item>测试3</item>
</array>
```

---

2). 分配资源ID

需要关注的点:

1.资源ID的格式

0xPPTTEEEE
PP: packageID
TT: typeID
EEEE: entryID

2.第一个type必须是attr

```
LOG_ALWAYS_FATAL_IF(ti == 0 && attr != t,
                    "First type is not attr!");
```

解析引用ID

引用类型的资源，如何解析成ID。e.g. bag资源的parent如果是一个引用类型

step 1: 解析引用字符串

```
while (p < end) {
    if (*p == ':') packageEnd = p;
    else if (*p == '/') {
        typeEnd = p;
        break;
    }
    p++;
}
p = refStr;
if (*p == '@') p++;

if (outPublicOnly != NULL) {
    *outPublicOnly = true;
}
if (*p == '*') {
    p++;
    if (outPublicOnly != NULL) {
        *outPublicOnly = false;
    }
}
```

step 2: 获取引用资源的id

```
// First look for this in the included resources...
uint32_t specFlags = 0;
uint32_t rid = mAssets->getIncludedResources()
    .identifierForName(name.string(), name.size(),
                       type.string(), type.size(),
                       package.string(), package.size(),
                       &specFlags);
if (rid != 0) {
    if (onlyPublic) {
        if ((specFlags & ResTable_typeSpec::SPEC_PUBLIC) == 0) {
            return 0;
        }
    }
```

3. 生成资源索引表

数据结构：

ResTable_header

ResStringPool_header

ResTable_package

ResTable_lib_header

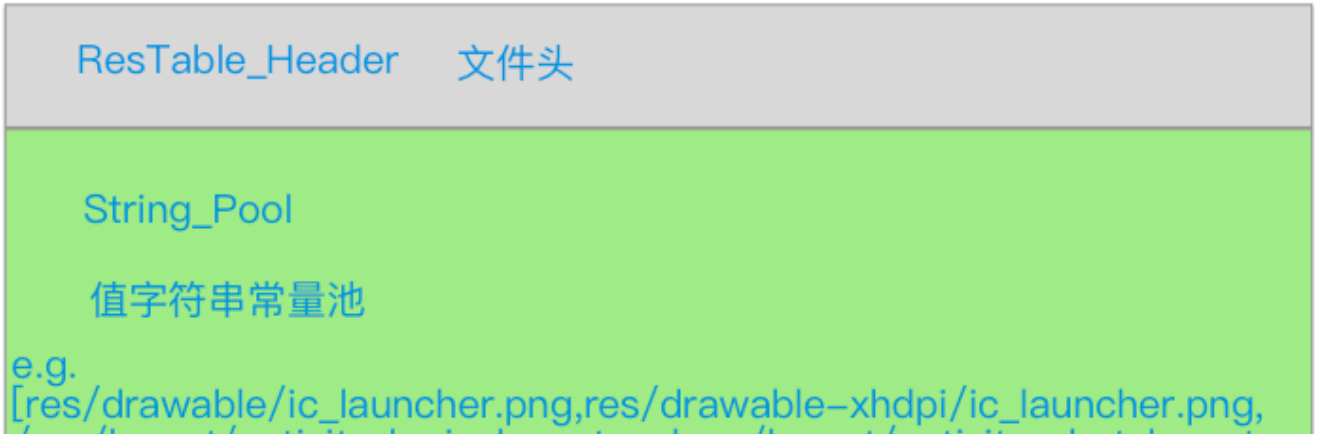
ResTable_lib_entry

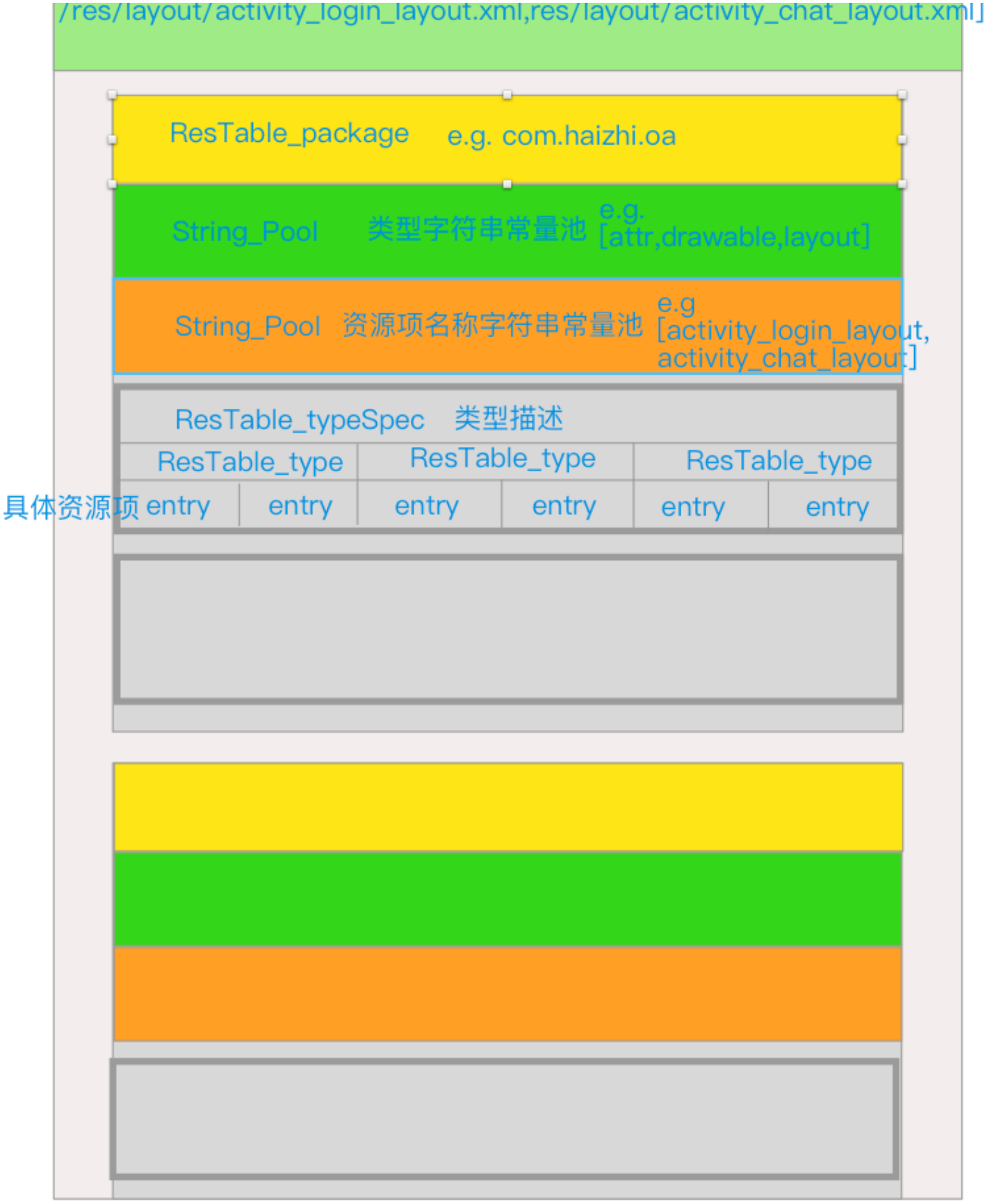
ResTable_typeSpec

ResTable_type

ResTable_entry

arsc文件结构





遗漏了 ResTable_lib_header ResTable_lib_entry 两个chunk

需要关注的点

1. 依赖的资源是如何处理的。

```
// The libraries this table references.
Vector<sp<Package> > libraryPackages;
const ResTable& table = mAssets->getIncludedResources();
const size_t basePackageCount = table.getBasePackageCount();
for (size_t i = 0; i < basePackageCount; i++) {
    size_t packageId = table.getBasePackageId(i);
    String16 packageName(table.getBasePackageName(i));
    if (packageId > 0x01
        && packageId != 0x7f
        && packageName != String16("android")) {
        libraryPackages.add(sp<Package>(new Package(packageName, packageId)));
    }
}
```

1. ResTable_lib_entry和DynamicRefTable的关系

```
/**
 * 在前一步中,已经对每一个package 创建好了 package chunk,
 * 此时只需要按照顺序将每一个段写入resource.arsc中
 *  ----------------------
 * |    ResTable_header   |            --- RES_TABLE_TYPE
 *  ----------------------
 * |----------------------|
 * |  ResStringPool_header |            --- RES_STRING_POOL_TYPE
 * |     stringPool       |
 * |----------------------|
 * |----------------------|
 * |   ResTable_package   |            --- RES_TABLE_PACKAGE_TYPE
 * |     typeStringPool   |
 * |     keyStringPool    |
 * |   ResTable_lib_header |            --- RES_TABLE_LIBRARY_TYPE
 * |   ResTable_lib_entry  |
 * |   ResTable_typeSpec  |            --- RES_TABLE_TYPE_SPEC_TYPE
 * |   ResTable_type      |            --- RES_TABLE_TYPE_TYPE
 * |      entry           |
 * |----------------------|
 *
 * DynamicRefTable 是一个运行时的数据结构,实际上处理了resources.arsc中的
 * ResTable_lib_header和ResTable_lib_entry两个数据结构,ResTable_lib_header
 * 包含了一个公用的ResChunk_header用来方便读取头部信息,另外一个字段count是当前package引用的
 * package的数量,在ResTable_lib_header后面是`count`个ResTable_lib_entry,每一个
 * ResTable_lib_entry包含两个字段,packageId和packageName,这样就建立了编译时的
 * packageName和packageId的映射关系,在运行时,由AssetManager解析一个resource.arsc文件,
 * 由于引用的资源的加载顺序不确定,但是packName是不变的,所以,通过DynamicRefTable,使用
 * packageName建立build-time的packageId和run-time的packageId的映射关系。
 * 解析和建立映射关系的过程,还需要去研究一下AssetManager的代码。
 *
 */
```

1. 运行时DynamicRefTable的映射流程
   Step 1: ResTable::parsePackage //解析一个带ResTable_lib_entry的package
   Step 2: dynamicRefTable.load //装载ResTable_lib_entry
   Step 3: dynamicRefTable.addMapping //建立映射

```
    /************************************
     *       解析ResTable_lib_entry段
     ************************************/
} else if (ctype == RES_TABLE_LIBRARY_TYPE) {
    if (group->dynamicRefTable.entries().size() == 0) {
        //Step 1 : 装载
        status_t err = group->dynamicRefTable.load((const ResTable_lib_header*) chunk);
        if (err != NO_ERROR) {
            return (mError=err);
        }

        // Fill in the reference table with the entries we already know about.
        //Step 2 : 映射
        size_t N = mPackageGroups.size();
        for (size_t i = 0; i < N; i++) {
            group->dynamicRefTable.addMapping(mPackageGroups[i]->name, mPackageGroups[i]->id);
        }
    } else {
        ALOGW("Found multiple library tables, ignoring...");
    }
```

4. 生成R.java和apk文件

可以解决我们的问题了吗?

解决编译时的资源问题

1. 公共资源放入宿主，先编译宿主，生成oa.ap_文件
2. 插件中需要引用宿主资源的地方使用@*packageName:resourceType/resourceName的形式依赖
3. 手动指定packageID

解决class依赖的问题

1. 公共代码放入宿主，先编译宿主，导出一部分的类,打成jar包
2. 编译插件时，通过provided的形式依赖这些jar包

改造我们的工程

1. 导出uicomp包的class

```
task exportRJar(type: Jar) {
    //指定生成的jar名
    baseName 'rClasses'
    from buildDir.absolutePath + '/intermediates/classes/debug/'
    include '**/R.class'
    include '**/R$*.class'
    //打包到jar后的目录结构
    //into("com/xxx/xxx")
    destinationDir = file(buildDir.absolutePath + '/outputs/')
}
```

2. 编译宿主

3. 修改插件对公共资源的依赖方式

```
<!-- Base application theme. -->
<style name="squarecamera__CameraFullScreenTheme" parent="@*com.haizhi.oa:style/AppTheme">
    <item name="android:windowNoTitle">true</item>
    <item name="android:windowActionBar">false</item>
    <item name="android:windowFullscreen">true</item>
    <item name="android:windowContentOverlay">@null</item>
</style>
```

4. 修改插件对公共代码的依赖方式

```
provided files('libs/android-support-annotations.jar')

provided files(project(':haizhisdk:uicomp').getBuildDir().absolutePath
        + '/outputs/rClasses.jar')
provided files(project(':haizhisdk:uicomp').getBuildDir().absolutePath
        + '/intermediates/exploded-aar/com.android.support/support-v4/23.4.0/jars/classes.jar')
provided files(project(':haizhisdk:uicomp').getBuildDir().absolutePath
        + '/intermediates/exploded-aar/com.android.support/support-v4/23.4.0/jars/libs/internal_impl-23.4.0.jar')
provided files(project(':haizhisdk:uicomp').getBuildDir().absolutePath
        + '/intermediates/exploded-aar/com.android.support/appcompat-v7/23.4.0/jars/classes.jar')
```

5. 添加aapt的编译依赖

```
aaptOptions.additionalParameters '-I', '/Users/chenlong/git/wbg/oa/build/outputs/apk/oa-debug.apk'
                                ,'--PLUG-resoure-id', '0x7d'
```

下一步的计划

- 搞清楚运行时的几个注入点
- 改造一个业务模块作为插件
- 测试，评估风险，插件加载失败的回退机制
- 搭建插件平台
    - 运行时监控
    - 编译时脚本

- 插件发布工具