Android 资源管理器分析

shecenon@gmail.com

2012-09-04

目录

Android 在应用程序的开发中大量使用了资源，他也把资源的使用发挥了极致。资源管理的核心代码在 frameworks/base/libs/utils 中，主要有 Asset.cpp,AssetManager.cpp,AssetDir.cpp,ResourceTypes.cpp 以及基础 StringPool.cpp, String16.cpp, String8.cpp。这是资源的编解码库。aapt 是生成资源的工具，也使用了这个库。而 Android 应用程序也是使用这个库来解析资源。这从 utils 目录下的 Android.mk 就可以看出来，把编译了本地的库和目标机器的库。

1 资源管理器一瞥

AssetManager 资源管理器 aapt 是比较复杂的，不适合入门。bootanimation [1] 有个简单的，可作示例：

```
1  status_t BootAnimation::initTexture(Texture* texture,
2                  AssetManager& assets, const char* name) {
3      Asset* asset = assets.open(name, Asset::ACCESS_BUFFER);
4      if (!asset)
5          return NO_INIT;
6      SkBitmap bitmap;
7      SkImageDecoder::DecodeMemory(asset->getBuffer(false),
8              asset->getLength(), &bitmap, SkBitmap::kNo_Config,
9              SkImageDecoder::kDecodePixels_Mode);
10     asset->close();
11     delete asset;
12     . . .
13 }
```

这个 name 就是路径，如"images/android-logo-shine.png"，因为这个不是 Android 应用程序，这个资源就是"framework/framework-res.apk"，即 AssetManager.cpp 的 kSystemAssets 值。

[1] frameworks/base/cmds/bootanimation

从这段代码看，就是使用 AssetManager 打开一个文件，得到的是 Asset，从 Asset 调用 getBuffer 得到数据，调用 getLength 得到数据长度，调用 close 关闭资源，然后要删除 delete 这个对象（因为动态分配的）。

1.1 创建资源管理器

BootAnimation 有个疑问就是 AssetManager 对象的不知如何创建的？不过，android_util_AssetManager.cpp 的 android_content_AssetManager_init 给的很详细：

```
1  static void android_content_AssetManager_init(JNIEnv* env, jobject
     clazz)
2  {
3      AssetManager* am = new AssetManager();
4      if (am == NULL) {
5          jniThrowException(env, "java/lang/OutOfMemoryError", "");
6          return;
7      }
8
9      am->addDefaultAssets();
10
11     LOGV("Created AssetManager %p for Java object %p\n", am, clazz);
12     env->SetIntField(clazz, gAssetManagerOffsets.mObject, (jint)am);
13 }
```

AssetManager.java 的构造器都要调用 init 方法, ActivityThread.java, PackageParser.java 会创建 AssetManager 对象。

aapt 中 AssetManager 对象在什么地方创建？就是 AssetManager 类型数据成员 mIncludedAssets。

1.2 加入资源包

调用 addDefaultAssets 会把 framework/framework-res.apk 加入资源管理器，bootanimation 就调用了他。

addDefaultAssets 是对 addAssetPath 的包装，而应用程序自己的资源包就使用 addAssetPath，对 Android 应用来讲 PackageParser.java , ActivityThread.java 会使用到的。对于 PackageParser.java 来讲，他是要获取并解析 AndroidManifest.xml 中的内容。对于 ActivityThread.java 来讲，他把 AssetManager 加入到 Resources，应用会使用 Resources 获取资源的。对于 Resources 来讲，framework-res.apk 在 AssetManager 对象创建时就已经加入的资源管理器。

1.3 获取资源文件

AssetManager 提供了如下几个函数：[2]

1. open
   打开 apk 中未压缩的文件。AssetManager 的 open 函数会在 name 前加上"assets"。比如，bootanimation 中打开文件"images/android-logo-shine.png" 就是获取 framework-res.apk 中的压缩文件 assets/images/android-logo-shine.png
2. openNonAsset
   获取 apk 经压缩处理的文件。比如经过 android_util_AssetManager.cpp, AssetManager.java 的封装，成为 Android 层 AssetManager 的 openXmlResourceParser。打开 apk 包里 xml 文件，并以解析器的形式返回，如 PackageParser.java 的 parsePackage [3]：

```
1  XmlResourceParser parser = null;
2  AssetManager assmgr = null;
3  boolean assetError = true;
4  try {
5      assmgr = new AssetManager();
6      int cookie = assmgr.addAssetPath(mArchiveSourcePath);
7      if(cookie != 0) {
8          parser = assmgr.openXmlResourceParser(cookie, "
             AndroidManifest.xml");
9          assetError = false;
10     } else {
11         Log.w(TAG, "Failed adding asset path:"+
             mArchiveSourcePath);
12     }
13 }
```

3. getResources
   对 getResTable 的封装，获取 apk 包中的 resources.arsc 文件的资源接口。AssetManager 的 getResTable 是没有被外面调用的，因为申明为 private 的成员函数了 (AssetManager.h)。
4. openNonAssetDir
   aapt 工具使用。

1.4 获取资源的内容

这些工作在 ResourceTypes.cpp, Asset.cpp 中完成的，对不同类型的资源，使用不同的方法。

[2] 资源管理器中，Asset 代表的是一个文件。AssetDir 表示目录。
[3] JarFile 也可以获得 apk 包里的文件，见 collectCertificates

AssetManager 接口	处理资源接口	接口文件
open	Asset	Asset.c
openNonAsset	ResXMLTree Asset	ResourceTypes.cpp Asset.c
getResources	ResTable	ResourceTypes.cpp
openDir	AssetDir	AssetDir.cpp

Android 编译后，apk 中文件，有 AssetManager 的可以分为以下类别：

1. classes.dex
   由虚拟机解析。
2. resources.arsc
   由 ResTable 解析，ResTable_header、ResTable_package 、ResStringPool_header 等数据结构。
3. XML 文件
   包括 AndroidManifest.xml 以及 res/目录下 layout、xml、anim、drawable*、color 等目录里 xml。
   XML 文件在 apk 中以 ResXMLTree 形式组织，以方便在设备中解析 (ResXMLParser) 。涉及 ResXMLTree_header、ResXMLTree_node、ResXMLTree_attribute 等数据结构。
4. 原始文件，包括 asset 目录, res/目录下 drawable 等目录下的 png 文件、raw 目录。
5. META-INF 目录下文件。

上述 resources.arsc、XML 文件的数据是按数据块 (chunk) 划分的树形结构，每个数据块都以 struct ResChunk_header 开头，占 8 个字节：

type
: (uint16_t) 表示数据块类型，

headerSize
: (uint16_t) 表示数据块头的大小，他不是 struct ResChunk_header 本身大小，而是包含 struct ResChunk_header 的数据结构的大小；

size
: (uint32_t) 表示数据块的大小，包括 ResChunk_header 所在数据结构本身大小，其后附件的一些索引数组，数据以及其包含的其他数据块 (chunk) 的大小。所以，组织结构相当于一个树形结构，一个跟节点，中间节点可能只是数据，也可能包含其他的数据块。

表 1: 重要数据块的一些特征

数据结构	头部长度 (headerSize)	.arsc 起始数据 (type+headerSize)
ResStringPool_header	0x1c(28)	0100 1C00
ResTable_header	0x0c(12)	0200 0C00
ResTable_package	0x11c(284)	0002 1C01
ResTable_type [a]	0x34(52)	0102 3400
ResTable_typeSpec	0x10(16)	0202 1000
ResXMLTree_header	0x08(8)	0300 0800
ResXMLTree_node	0x10(16)	0[0-4]01 1000

[a] ResTable_config 含有的成员函数不占据内存的, 所以只有 32 字节。

数据结构名以 ResTable 开头的适用于 resources.arsc 文件，这个文件以 0200 0C00 (ResTable_header) 开始的。

数据结构名以 ResXMLTree 开头的适用于 xml 文件，这种 xml 文件以 0300 0800 (ResXMLTree_header) 开始的。

2 资源表数据结构

AssetManager::getResTable 创建的大概流程：

```
1  ResTable* rt = mResources;
2  Asset* ass = NULL;
3  ass = const_cast<AssetManager*>(this)->
4            openNonAssetInPathLocked("resources.arsc",
5            Asset::ACCESS_BUFFER,
6            ap);
7  mResources = rt = new ResTable();
8  rt->setParameters(mConfig); //in updateResourceParamsLocked();
9  rt->add(ass, (void*)(i+1), !shared);
```

ResTable 解析 resources.arsc 的成员函数是 add。

2.1 ResTable_header

从数据块角度，resources.arsc 是一个 ResTable_header，里面包了一个 ResStringPool_header 数据块和一个或多个 ResTable_package 数据块，其中 ResTable_package 的个数包含在 ResTable_header 数据成员 packageCount。见 ResourceTypes.h 中数据结构 ResTable_header 定义的说明。ResTable_package 的类型是 RES_TABLE_TYPE(即其包含的 ResChunk_header 的 type 值)。

2.2 ResStringPool_header

ResStringPool_header 数据块解析：

- stringStart 字符串池数据起始地址的相对于 ResStringPool_header 开始地址的偏移量
- stringCount 字符串池的字符串数量。

ResStringPool_header 结尾与字符串池数据存放地址之间，有个数组，存放每个字符串起始地址相对字符串池数据起始地址的偏移量。

字符串以 utf-16 编码存放 (对 ascii 字符讲就是占 2 个字节)，各个字符串连接在一起。每个字符串以其字符长度 (也是 uint16_t 数据存放) 开始，以 0x0000 结束，长度不包含结尾的 0x0000。但是也可以在 flags 设置 UTF8_FLAG (1<<8)，这样字符串就是以 UTF-8 编码。

2.3 ResTable_package

ResTable_package 旗下包含了 2 个 ResStringPool_header，

资源类型符号表
: 例如 attr, drawable, string, layout 等各种资源类型。

资源关键字符号表
: 比如，values/strings.xml 有下面一条：
<string name="hello">Hello World, Activity!</string>
其中，"hello" 就放在关键字符号表，而"Hello World, Activity!" 放在 ResTable_header 中的字符串池中。对于 drawable, layout 中的 xml 文件和图片，其文件名放在关键字符号表 (不包含扩展名)，文件的路径放在 ResTable_header 中的字符串池中, 例如，layout/-main.xml，关键字符号表包含字符串"main" ,ResTable_header 中的字符串池包含字符串"layout/main.xml"。

ResTable_package 中这两个字符串池后面跟一个或多个 ResTable_type 和 ResTable_typeSpec 数据结构。结构是以一个 ResTable_typeSpec + 掩码数组 + ResTable_type 组成一个单元，这样一个单元可以描述一个 R.java 的内部类，比如 R.drawable，R.string 等。R.java 里有几个内部类，就有几个这样的单元。R.java 中，attr 类总是存在的，不管其有无属性，其他的有属性则存在，无则不存在。不过，attr 没有属性时，仅有 ResTable_typeSpec，没有掩码数组 + ResTable_type。

2.3.1 ResTable_typeSpec

每个资源类型**??**(resource type) 必须有一个 ResTable_typeSpec. 其 id 从 1 开始，1，2，3，4，依次递增。entryCount 可以为 0，这样的 typeSpec 也会出现的，当为 0 时，后面没有数组，若不为 0，typeSpec 后面就跟着 entryCount 个 uint32_t 数据，一个 unit32_t 表示一个配置的掩码，表示有是否有多个配置以及是否是公开的。

typeSpec 表示的是 R.java 中 R 的内部类，entryCount 表示这些内部类的属性的个数，掩码就是表明这些属性在有几个不同配置 (或者说是值)。例如：下面的 R.java 对应的 typeSpec

```
1  public final class R {
2      public static final class attr {
3      }
4      public static final class drawable {
5          public static final int icon=0x7f020000;
6          public static final int img=0x7f020001;
7          public static final int star=0x7f020002;
8      }
9      public static final class layout {
10         public static final int main=0x7f030000;
11     }
12     public static final class string {
13         public static final int app_name=0x7f040001;
14         public static final int hello=0x7f040000;
15     }
16 }
```

内部类	id	entryCount
attr	01	0
drawable	02	03
layout	03	01
string	04	02

还可以注意到，id 值与内部类属性值的关联，比如 drawable 的 id 为 02，那 drawable 属性值的 4-5 位便是 02, 另外，id - 1 就是 ResTable_package 资源类型符号表的索引，通过他可以确定是 att, layout, string 等值. . entryCount 就是每个内部类的属性个数。

2.3.2 ResTable_type

每个 ResTable_typeSpec 后面跟的就是 ResTable_type，这两个数据结构都有 id 和 entryCount 数据成员，对这样一对的数据来讲，其 id 值相等的，entryCount 的值也是一样的。ResTable_type 后面跟着 4 * entryCount 个字节的数据，是后面 ResTable_entry 数组元素的偏移量，其值是：数组下标 * sizeof(ResTable_entry)。再后面就是 entryCount 个 ResTable_entry 数据，这些 ResTable_entry 对起始地址与 ResTable_type 的起始地址的偏移值就是 entriesStart。

$$entriesStart = sizeof(ResTable\_type) + sizeof(uint32\_t) * entryCount;$$

ResourceTable::flatten**??**

```
1  const size_t typeSize = sizeof(ResTable_type) + sizeof(uint32_t)*N;
```

```
2   .. .. ..
3
4   const size_t typeStart = data->getSize();
5
6   ResTable_type* tHeader = (ResTable_type*)
7       (((uint8_t*)data->editData(typeStart+typeSize)) + typeStart);
8   if (tHeader == NULL) {
9       .. .. ..
10      return NO_MEMORY;
11  }
12
13  memset(tHeader, 0, sizeof(*tHeader));
14  tHeader->header.type = htods(RES_TABLE_TYPE_TYPE);
15  tHeader->header.headerSize = htods(sizeof(*tHeader));
16  tHeader->id = ti+1;
17  tHeader->entryCount = htodl(N);
18  tHeader->entriesStart = htodl(typeSize);
19  tHeader->config = config;
```

ResTable_type 包含一个 ResTable_config 类型成员 config，对同一个资源类型来讲，可能有多个 ResTable_type 的数据块，每个代表不同的配置。

2.3.3 ResTable_entry

8 个字节长度，开始数据是 0x0800, 紧接着可能的取值是 0x0000 到 0x0003。key 虽是 ResStringPool_ref，其实就是一个 uint32_t 类型的数值，作为 ResTable_header 中的字符串池的索引值，从 0 开始计数的。从 R.java 来讲，ResTable_entry 表示的是其内部类的各个属性，对于 drawable，通过 key 我们可以定位这个图片的位置，如 res/drawable/icon.png。一个 ResTable_entry 跟一个 Res_value 或 ResTable_map。

Res_value 也是 8 个字节长度，开始数据也是 0x0800。如果 Res_value 开始的值为 0800 0000，ResTable_entry 开始的值为 0800 0003，那么 Res_value 数据成员 data 的值，和 ResTable_entry 数据成员 key 的值相等的。

2.3.4 ResTable_map_entry

ResTable_map_entry 是 ResTable_map 的派生类，ResTable_map_entry 对应的值是 ResTable_map, 就像 ResTable_entry 对应的值是 Res_value。

3 资源表主要函数

主要讲述的是 ResTable 类中的函数。其实，生成资源表是 appt 负责，但是与 ResTable 也有关系的。这里不要混淆 ResourceTable 和 ResTable 类了。

代码 1: Android XML 举例

```
1  <TextView
2      android:text="@string/weekpicker_title"
3      android:textAppearance="?android:attr/textAppearanceMedium"
4      android:layout_gravity="center_horizontal"
5      android:layout_width="wrap_content"
6      android:layout_height="wrap_content" />
7
8  <TextView android:id="@+id/headerText"
9      android:layout_width="match_parent"
10     android:layout_height="0dip"
11     android:layout_weight="1.0"
12     android:gravity="center"
13     android:textSize="18sp"/>
14
15 <View
16      android:background="@*android:drawable/code_lock_top"
17      android:layout_width="match_parent"
18      android:layout_height="2dip" />
```

3.1 stringToValue

ResTable::stringToValue 实际上只被 ResourceTable::stringToValue 调用，这是因为她只用于构建资源表的，而不是解析/使用资源表的。ResourceTable 只是 aapt 用于构造 ResTable 的类，与使用 apk 无直接关系。此函数与 Accessor?? 也有比较紧密的联系。在 ResourceTypes.h 中对其的注释是

```
// Convert a string to a resource value.  Handles standard "@res",
// "#color", "123", and "0x1bd" types; performs escaping of strings.
// The resulting value is placed in 'outValue'; if it is a string type,
// 'outString' receives the string.  If 'attrID' is supplied, the value is
// type checked against this attribute and it is used to perform enum
// evaluation.  If 'acccessor' is supplied, it will be used to attempt to
// resolve resources that do not exist in this ResTable.  If 'attrType' is
// supplied, the value will be type checked for this format if 'attrID'
// is not supplied or found.
```

ResTable::stringToValue 对属性值作检测，如属性值开头的'@', '#', '?', "true", "TRUE", "false", "FALSE"。ResTable::collectString 对转义字符串以及上面提及的特殊字符会做处理。

@	表示引用	TYPE_REFERENCE
#	表示颜色	TYPE_INT_COLOR_*
?	表示属性	TYPE_ATTRIBUTE

3.1.1 有关'@' 的说明

@null	空值，没有引用值。
@+	如果资源不存在则创建。一般是 android:id="@+id/myid"。
@*	引用的公共的资源。
@	一般的引用，可以是私有的，也可以是公共。

代码 2: stringToValue 对 @ 的处理代码

```
1  bool createIfNotFound = false;
2  const char16_t* resourceRefName;
3  int resourceNameLen;
4  if (len > 2 && s[1] == '+') {
5      createIfNotFound = true;
6      resourceRefName = s + 2;
7      resourceNameLen = len - 2;
8  } else if (len > 2 && s[1] == '*') {
9      enforcePrivate = false;
10     resourceRefName = s + 2;
11     resourceNameLen = len - 2;
12 } else {
13     createIfNotFound = false;
14     resourceRefName = s + 1;
15     resourceNameLen = len - 1;
16 }
```

通过变量 createIfNotFound ，我们可以查到函数调用

```
1  accessor->getCustomResourceWithCreation(package, type, name,
     createIfNotFound);
```

使用了此变量。所以，对于"@+"，创建的新的资源项，就在 getCustomResourceWithCreation 中实现的。

搜索"'@'"，注意带'，有几个地方：expandResourceRef, identifierForName, stringToValue, collectString 有对其处理。

我们顺着**??**往下看，

- expandResourceRef 把节点属性值分解成包 (package)、类型 (type)、资源名 (name)，见 **??**的分析，这个函数一定要返回成功，否则就退出了。
- identifierForName 获得资源名对应的 id，此函数最后会调用 Res_MAKEID 产生 id，这个函数只是从已有的资源里获取资源的信息来提供 id，并不会创建资源[4]，所以找不到资源时会返回 id 为 0。

[4]创建资源请见 ResourceTable::addEntry **??**

当 identifierForName 返回的 id 不为 0，accessor 参数为 null 时，stringToValue 就返回这个 id。当 accessor 不为 null，调用其 getRemappedPackage，但是现在这个函数只是直接返回参数值，所以，最后其返回的值跟 accessor 为 null 时一样的。**??**的第二个实现也是有这种情形。

- Accessor::getCustomResourceWithCreation 当 identifierForName 返回 id 为 0，而 accessor 不为 null(一般这个是成立的, 因为 Accessor 是个抽象类，其具体派生类是 ResourceTable)就调用这个函数对应的就是 ResourceTable::getCustomResourceWithCreation**??**.

3.2 identifierForName

这里的 name 指的就是节点的属性值。从函数结尾调用 Res_MAKEID 的参数来看，PackageGroup 相当于一个 Pakcage。

包名 (packageEnd) 以":" 作为标志，类型名 (typeEnd) 以"/" 作为标志。下面的代码就是解析节点的值中的包名、类型以及入口，

代码 3: identifierForName 解析节点的代码

```
1  // Figure out the package
     and type we are looking in
     ...
2
3  const char16_t* packageEnd =
     NULL;
4  const char16_t* typeEnd =
     NULL;
5  const char16_t* const
     nameEnd = name+nameLen;
6  const char16_t* p = name;
7  while (p < nameEnd) {
8      if (*p == ':')
         packageEnd = p;
9      else if (*p == '/')
         typeEnd = p;
10     p++;
11 }
12 if (*name == '@') name++;
13 if (name >= nameEnd) {
14     return 0;
15 }
16
17 if (packageEnd) {
18     package = name;
19     packageLen = packageEnd-
         name;
20     name = packageEnd+1;
21 } else if (!package) {
22     return 0;
23 }
24
25 if (typeEnd) {
26     type = name;
27     typeLen = typeEnd-name;
28     name = typeEnd+1;
29 } else if (!type) {
30     return 0;
31 }
32
33 if (name >= nameEnd) {
34     return 0;
35 }
36 nameLen = nameEnd-name;
```

ResTable::expandResourceRef 开头的部分代码也是处理这个工作。这里包名跳过了'@'，但是'+', '?', '*' 未跳过，需要 stringToValue **??** 处理。

3.2.1 资源项的 id 生成

生成资源 id 有两种方法 :Res_MAKEID 宏、函数 ResourceTable::makeResId 。

代码 4: ResourceTypes.h Res_相关的宏定义

```
1  /**
2   * Macros for building/splitting resource identifiers.
3   */
4  #define Res_VALIDID(resid) (resid != 0)
5  #define Res_CHECKID(resid) ((resid&0xFFFF0000) != 0)
6
7  #define Res_MAKEID(package, type, entry) \
8      (((package+1)<<24) | (((type+1)&0xFF)<<16) | (entry&0xFFFF))
9
10 #define Res_GETPACKAGE(id) ((id>>24)-1)
11 #define Res_GETTYPE(id) (((id>>16)&0xFF)-1)
12 #define Res_GETENTRY(id) (id&0xFFFF)
13
14 #define Res_INTERNALID(resid) ((resid&0xFFFF0000) != 0 && (resid&0
     xFF0000) == 0)
15 #define Res_MAKEINTERNAL(entry) (0x01000000 | (entry&0xFFFF))
16 #define Res_MAKEARRAY(entry) (0x02000000 | (entry&0xFFFF))
17
18 #define Res_MAXPACKAGE 255
```

代码 5: ResourceTable.h 的 ResourceTable::makeResId

```
1  static inline uint32_t makeResId(uint32_t packageId,
2                                   uint32_t typeId,
3                                   uint32_t nameId)
4  {
5      return nameId | (typeId<<16) | (packageId<<24);
6  }
```

ResTable::identifierForName 打印分析，只负责了 0x7f01xxxx(class attr), 0x7f08xxxx(class id)，0x7f0dxxxx (class style) 以及 Android 包中的内容。

3.3 getBasePackageId

```
1  uint32_t ResTable::getBasePackageId(size_t idx) const
2  {
3   if (mError != NO_ERROR) {
4   return 0;
5   }
6   LOG_FATAL_IF(idx >= mPackageGroups.size(),
7   "Requested package index %d past package count %d",
8   (int)idx, (int)mPackageGroups.size());
9   return mPackageGroups[idx]->id;
10 }
```

3.4 Accessor 类

ResTable::Accessor (ResourceTypes.h) 声明了纯虚成员函数 reportError，ResourceTable (ResourceTable.h) 继承 ResTable::Accessor。Accessor 主要是为 aapt 实现的，由于 asset manger 库和 aapt 的交换。Accessor 只传给 ResTable::stringToValue[??] 使用。

例如代码：

```
1 <TextView  android:id = "@idd/myTextView"      ... />
```

属性“android:id”的值应该是引用 myTextView 的字符串，即"@id/myTextView"，这里多了一个"d"，写成了"idd"。构建资源时就报

“Error :No resource found that matches the given name (at "id" with value '@idd/myTextView')."

此错误在函数 ResTable::StringToValue 报出，它是调用了 ResourceTable::reportError。类似的，getCustomResourceWithCreation 也是这个原理。

3.5 getTableStringBlock

android_content_AssetManager_loadResourceValue

3.6 getResource / resolveReference

lockBag

bag_entry struct bag_entry ssize_t stringBlock; ResTable_map map; ;

4 扩展标记语言文本

ResXMLTree 是 ResXMLParser 的派生类，ResXMLParser::next -》ResXMLParser::nextNode ResXMLParser::indexOfAttribute ，比较简单，遍历元素的所有属性即可。ResXMLParser::getAttributeStringValue

ResXMLTree 字面含义应该是个树形结构，其实是个流或者说数列，使用方法

```
1  ResXMLTree block;
2  status_t err = parseXMLResource(in, &block, false, true);
3
4  if (err != NO_ERROR) {
5          return err;
6      }
7  block.next();
8
9  block.getElementName(&len);     //获取标记的名字
10 size_t len;
11 ssize_t itemIdentIdx = block.indexOfAttribute(NULL, "name"); //获取属
     性名的属性索引name
12 if (itemIdentIdx >= 0) {
13     // 通过索引获取属性值
14     itemIdent = String16(block.getAttributeStringValue(itemIdentIdx,
         &len));
15 }
```

4.1 声明文件的解析

Android 的 AssetManager 对资源的解析中要用到资源 Resouces 类 PackageParser.java 对 AndroidManifest.xml 的解析需要 AndroidManifestActivity frameworks/base/core/res/res/values/attrs_manifest.xml

树形结构，树形解析 PackageParser.parseActivity 解析 activity 标记，并调用 PackageParser.parseIntent 解析 activity 标记下的 intent-filter

PreferenceInflater.onCreateCustomFromTag 调用 Intent.parseIntent 解析标记，而 AliasActivity.parseAlias 也调用 Intent.parseIntent 解析 intent 标记

PackageParser 里之所以不使用 Intent.parseIntent 解析 intent-filter 标签，是因为他希望解析结果以 ItentInfo 对象保存，如 ServiceIntentInfo、ActivityIntentInfo，而 Intent.parseIntent 解析的结果是 Intent 对象。但是有一点是相同的，即分层解析。以 PackageParser 类为例：

1. 解析第一层标记类的初始化函数 public Package parsePackage(File, String, DisplayMetrics, int) ，主要作了两件事：

   创建解析文件 **AndroidManifest.xml** 的解析器

   parser = assmgr.openXmlResourceParser(cookie, "AndroidManifest.xml");

   调用解析函数解析

   private Package parsePackage(Resources, XmlResourceParser, int, String[]) 这个函数解析的是 AndroidManifest.xml 中顶级标记 manifest 下一层的所有标签，manifest 标记带有 package 属性。比如，调用 parseApplication 对 application 标记解析; 调用 parsePermission 解析标记 permission; 对标记 instrumentation 使用方法 parseInstrumentation 分析。
2. 解析第二层标记以 parseApplication 为例，对 application 标记下的子标记 activity，service，provider 等分别调用 parseActivity，parseService，parseProvider 解析。
3. 解析第三层再以 parseActivity 为例，调用方法 parseIntent 和 parseMetaData 分别解析标记 intent-filter 与 meta-data。
4. 解析叶子标记所谓叶子标记是其下没有了子标记，节点标记其下有子标记，也就是标记容器。对于叶子标记的解析方法就是 Resources.obtainAttributes。对于节点标记，除了要获取属性 (Resources.obtainAttributes)，还有一个重要的功能，就是拉出下一个标记 (XmlPullParser.next)，如果拉出的标记深度 (XmlPullParser.getDepth) 大于容器标记深度，就解析，否则退出，这是一种递归算法，保证解析的标记都是同一个层次。

XmlPullParser 解析出来的是流，即按文件中出现的顺序解析节点，而不是节点的层次。采用层次处理解析的标记流，把不同层次的标记放在不同方法，同一个层次的都在一个方法，清晰而简洁。

```
1  platform$ find . -name "*.java" | xargs grep --color -n -C 0 "
     implements\s*XmlPullParser"
2  ./libcore/xml/src/main/java/org/apache/harmony/xml/ExpatPullParser.
     java:34:public class ExpatPullParser implements XmlPullParser {
3  ./libcore/xml/src/main/java/org/kxml2/io/KXmlParser.java:33:public
     class KXmlParser implements XmlPullParser {
```

5 资源包装工具

aapt

aapt p -M AndroidManifest.xml -S res -v

先目录，再文件。先文件，再 xml。xml 先节点，再属性，及属性值。函数调用顺序(main -> doPackage -> writeResourceSymbols -> writeSymbolClass -> writeLayoutClssses -> getAttributeComment)

对 AaptAssets.cpp 是处理目录、文件层次的工作，基本元素就是 AaptFile，代表的是一个文件，而 AaptDir 是代表一个目录，可以包括子目录和文件，这是 AaptFile 在文件系统的组织结构，而 AaptGroup 是 Android 的文件组织结构，他是把不同 AaptDir(即目录)下同名的文件组织在一起，这些文件代表了是不同配置下的同一个属性的不同取值。而 AaptAssets 是 AaptDir 的派生类，这个结构是从 apk 的角度来看的，或者说是表示资源目录的根，对于一个 apk 而言，其运行时不仅仅依赖于其中的 res 目录和 AndroidManifest.xml，也依赖于 frameworks-res.apk，甚至有覆盖 (overlay) 的 apk 包。AaptGroupEntry 也是表示文件，不过附加了 Android 中的各种配置信息。

Command.c 的 doPackage 是创建 apk 的函数，其中的 bundle(Bundle*) 主要是处理命令行参数的，是输入内容。之后就是解析，主要是创建 AaptAssets 对象，然后调用 buildResources，最后调用 writeResourceSymbols、writeAPK 输出文件。

doRemove Command.cpp Bundle 是保存 aapt 命令行信息。bundle->getFileSpecCount 命令行中包含的文件数，第一个文件名应该是压缩文件名 const char* fileName = bundle->getFileSpecEntry(i); 通过下标获取命令行中的文件名。

代码 6: doPackage 的核心代码

```
1  // Load the assets.
2  assets = new AaptAssets();
3  err = assets->slurpFromArgs(bundle);
4  .. .. ..
5
6  // If they asked for any files that need to be compiled, do so.
```

```
 7  if (bundle->getResourceSourceDirs().size() ||
 8      bundle->getAndroidManifestFile()) {
 9      err = buildResources(bundle, assets);
10      .. .. ..
11  }
```

表 2: aapt 主要源码一览

Main.cpp	程序入口，解析命令行参数。
Commands.cpp	执行 aapt 的各种命令，是处理框架。
Resource.cpp	apk 资源解析的处理模块， 全局函数及类 PackageInfo, ResourceDirIterator 实现
AaptAssets.cpp	AaptAssets,AaptDir,AaptGroup, AaptFile,AaptGroupEntry 类的实现
Package.cpp	实现函数 writeAPK。
ResourceTable.cpp	ResourceTable, ResourceTable::Package, ResourceTable::Type,ResourceTable::ConfigList, ResourceTable::Entry, ResourceTable::Item, ResourceFilter 等类的实现， compileXmlFile，compileResourceFile 等函数的实现。
Images.cpp	preProcessImage, postProcessImage 实现， 其他函数都是内部函数
XMLNode.cpp	XMLNode 类以及函数 parseXMLResource 实现
StringPool.cpp	StringPool 类的实现。
SourcePos.cpp	ErrorPos, SourcePos 类的实现。
ZipEntry.cpp	ZipEntry, ZipEntry::LocalFileHeader, ZipEntry::CentralDirEntry
ZipFile.cpp	ZipFile, ZipFile::EndOfCentralDir 类实现。

5.1 主要数据结构

5.1.1 AaptDir

AaptDir::slurpFullTree 先把当前目录下的文件和目录的路径全保存到 fileNames，然后遍历 fileNames，若是目录，并已存入 mDir，则从 mDir 取出 subdir(sp<AaptDir>)，否则创建 subdir，并对 subdir 调用 slurpFullTree，最后把 subdir 存入 mDir。假如是文件，创建一个 file(sp<AaptFile>)，然后调用 addLeafFile，addLeafFile 先以文件名从 mFiles 获取所对应的 AaptGroup，如失败则创建 AaptGroup，并和文件名一起加入到 mFiles，最后把 file(sp<AaptFile>) 加入到 AaptGroup。这种情形应该是不同名录下，同名的文件加入到一个 AaptGroup，比如 res 目录各个 drawable* 目录下同名的 png 文件，又或者各个 value 下 string.xml，等等。

```
1       String8 mLeaf;
2       String8 mPath;
3
```

```
4       DefaultKeyedVector<String8, sp<AaptGroup> > mFiles;
5       DefaultKeyedVector<String8, sp<AaptDir> > mDirs;
```

AaptAssets.cpp 函数 isHidden 根据文件名把一些特定的目录和文件排除掉。AaptAssets::slurpResourceTree 及 AaptDir::slurpFullTree 用到。AaptAssets::slurpResourceTree 只被 AaptAssets::slurpFromArgs 调用到。

Resource.cpp 中定义的函数 collect_files

5.1.2 AaptGroup

AaptGroup :A group of related files (the same file, with different vendor/locale variations). 他的一个主要数据结构是：

```
1       String8 mLeaf;
2       String8 mPath;
3
4       DefaultKeyedVector<AaptGroupEntry, sp<AaptFile> > mFiles;
```

从这里可见，AaptGroup 是一系列 AaptFile 的有序集合，这些 AaptFile 以 AaptGroupEntry 的次序排列。为此 AaptGroupEntry 提供了 compare 来比较两个对象的大小，这个函数其实按一定顺序比较各个数据成员（这个顺序其实就是数据成员的优先级）。

各个 AaptGroup 可以用字符串作为关键字，所以这些关键字也是这个 AaptGroup 的一个属性或者说附加信息，而且在一个 AaptDir 内是唯一的：

代码 7: AaptDir 的属性

```
1       DefaultKeyedVector<String8, sp<AaptGroup> > mFiles;
```

5.1.3 AaptAssets

AaptAssets 是 aapt 中核心的数据结构，Images.cpp、AaptAssets.cpp、Package.cpp、ResourceTable.cpp、Resource.cpp、XMLNode.cpp、Command.cpp 等文件都见到她的身影。而相对来讲，AaptDir 只是 AaptAssets 的基类，主要负责处理文件收集、归类的一般性工作，AaptAssets.cpp、Package.cpp、Resource.cpp 、AaptAssets.h 中有出现。

resDir 根据参数指定的目录名返回一个 AaptDir 智能指针, resDirs() 返回的是一个列表，元素是 AaptDir 的智能指针

5.1.4 AaptFile

AaptFile 说是文件，其实更恰当的是一个 buffer，他不关心 buffer 数据的格式、含义，只管申请空间、读、写数据。buffer 相关的数据成员，如 mData 即指向 buffer，mDataSize 标示 mData 里使用的空间，mBufferSize 是 mData 总的空间(从这里看出代码命名的混乱)。虽然对 buffer 里面内容不关心，但是在 buffer 整体标示，AaptFile 还是有相关的数据成员，如 mGroupEntry 是 AaptGroupEntry，mResourceType（String8），以及原来的文件名、路径和数据压缩算法等相关信息。

5.1.5 AaptGroupEntry

AaptGroupEntry 作为 AaptFile 的关键字，就像 String8 对于 AaptGroup，AaptGroupEntry 也是每个 AaptFile 的附加信息，在一个 AaptGroup 的各个 AaptFile 可以以 AaptGroupEntry 区别，这就是说一个文件的不同配置下的具体内容，如 values-zh-rCN, values, values-zh-rTW 目录中的 strings.xml 的标志:

代码 8: AaptDir 的属性

```
1  DefaultKeyedVector<AaptGroupEntry, sp<AaptFile> > mFiles;
```

AaptGroupEntry 类似于 ResTable_config, IMSI, 语言、屏幕大小、分辨率、尺寸规格、键盘等等，当然有版本号，不然版本升级时由于数据不兼容而产生严重的错误。为此 AaptGroupEntry 提供了 compare 来比较两个对象的大小，这个函数其实按一定顺序比较各个数据成员(这个顺序其实就是数据成员的优先级)。而这些数据成员都是 String8 类型的，String8 类也提供了 compare 方法，他也就是调用了 strcmp，所以到底还是字符串比较。

5.1.6 ResourceTypeSet

资源类型从 buildResources 中的代码看到：

```
1       sp<ResourceTypeSet> drawables;
2       sp<ResourceTypeSet> layouts;
3       sp<ResourceTypeSet> anims;
4       sp<ResourceTypeSet> xmls;
5       sp<ResourceTypeSet> raws;
6       sp<ResourceTypeSet> colors;
7       sp<ResourceTypeSet> menus;
```

比如 drawadble-hdpi/icon.png, drawable/icon.png, drawable-ldpi/icon.png 组成一个 AaptGroup，drawable-hdpi/background.png, drawable/background.png, drawable-ldpi/backgoround.png 组成有一个 AaptGroup, 这两个 AaptGroup 均属于 drawable 资源类型。所有 drawable 资源类

型的 AaptGroup 就组成了一个 ResourceTypeSet。比如，xml 文件也按这种方式组织。

5.1.7 ResourceDirIterator

遍历一个 ResourceTypeSet 中包含的所有文件。过程是遍历一个 AaptGroup 的 AaptFile，再遍历下一个 AaptGroup 中的 AaptFile，直至把 ResourceTypeSet 中的所有 AaptGroup 遍历完毕。核心是 next(), 还提供了一系列获得当前文件的各种属性的函数。

5.2 Resource.cpp 主要函数

Resource.cpp[7] 是解析、编译的主要源文件，包括入口，处理，输出，辅助等几类功能的函数。

5.2.1 buildResources

这是解析输入的核心函数。根据其代码里的注释，可以看到处理流程：

```
1   // First, look for a package file to parse.  This is required to
2   // be able to generate the resource information.
3   .. .. ..
4
5  // ---------------------------------------------------------------
6   // First, gather all resource information.
7   // ---------------------------------------------------------------
8   .. .. ..
9
10  // ---------------------------------------------------------------
11  // Assignment of resource IDs and initial generation of resource
      table.
12  // ---------------------------------------------------------------
13
14
15  // ---------------------------------------------------------------
16  // Finally, we can now we can compile XML files, which may
17  // reference resources.
18  // ---------------------------------------------------------------
19  .. .. ..
20
21 // ---------------------------------------------------------------
22  // Generate the final resource table.
23  // Re-flatten because we may have added new resource IDs
24  // ---------------------------------------------------------------
25  .. .. ..
```

大概分五步：

- package file 就是指 AndroidManifest.xml，使用 parsePackage

解析获取包名 (android:package 属性值)，并以此初始化 ResourceTable 类的对象 table，把 asset(AaptAssets) 加入到 table 中。

- 收集资源信息。
  - 集合文件。调用对 asset 调用 collect_files，结果放入 resources。asset 调用 setResources，保存 resources。
  - 遍历资源的覆盖，并收集文件。AaptAssets 成员 mOverlay (sp<AaptAssets>), 假如他有赋值，则想上面一样处理，代码稍有不同，结果是一样的。因为都是 AaptAssets。而从 AaptAssets 角度，这个 mOverlay 也有自己的 mOverlay，所以循环直至 mOverlay 为空。

```
1  sp<AaptAssets> current = assets->getOverlay();
2  while(current.get()) { //current 是栈对象，不为空，get 说明有指
     向对象
3      KeyedVector<String8, sp<ResourceTypeSet> > *resources
4          = new KeyedVector<String8, sp<ResourceTypeSet> >;
5      current->setResources(resources);
6      collect_files(current, resources);
7      current = current->getOverlay();
8  }
```

  - 对 drawable，layout，anim，xml，raw，color，menu 资源类型调用 applyFileOverlay，结构保存到 drawables, layouts, anims, xmls, raws, colors, menus 变量。
  - 对 drawables 先调用 preProcessImages，再调用 makeFileResources；对 layouts, anims, xmls, raws 调用 makeFileResources[??]。
  - 编译资源。对 AaptAssets 及其 overlay 的 values 类型 AaptFile 调用 compileResourceFile[??]。
  - 针对 colors，menus 调用 makeFileResources。
- 调用 table.assignResourceIds()[??]，分配资源 IDs，生成初始资源表。
- 编译 XML 文件，因为这里涉及了交叉引用的问题。
  - 对 layout，anim，xml 资源类型的变量 layouts, anims, xmls 使用 compileXmlFile;
  - 对 drawable 的各个目录使用 postProcessImages 处理;
  - 对 color,menu 资源类型使用 compileXmlFile;
  - 编译 AndroidMainfest.xml 文件。
- 生成最终的资源表。把类 R 的符号加入 table，并调用 table.flatten()，因为可能要分配新的 ID。

5.2.2 makeFileResources

makeFileResources 是以文件作为资源单位编译，使用 ResourceDirIterator [??] 枚举 ResourceTypeSet 文件，对文件作三个步骤：

- 对文件名作判断(文件名只能在 [a-z0-9_.] 中取值,所以大写字母是不行的),
- 调用 table->addEntry[??] 把文件加入到 ResourceTable 的一个 Entry 中,
- assets->addResource[??],把文件加入到 AaptAssets 中的相关数据,形成树形目录结构。

这个 Entry 的源位置是文件路径,行号为 0, Package ,type 就是 makeFileResources 最后参数,即目录名就是类型名,name 是文件名,value 是路径(根据平台,会把"\" 转换为"/")。注意到 makeFileResources 的最后参数,她不仅仅作为目录名给 ResourceDirIterator 枚举文件,而且在 table->addEntry 作为 Type。

Type has an ConfigList , ConfigDescription Entry has associated structure Item and SourcePos SourcePos has only two member: filename, line number For Id, which attribute name is “android:id” , the finename is “<generated>”, and line number is 0. The following statement in function ResourceTable::Entry::generateAttributes tell us this information: status_t err = table->addEntry(SourcePos(String8("<generated> 0), package, id16, key, value);

ResourceTable* table, table->addEntry . Const sp<AaptAssets>& assets assets->addResource 因为知道,估计 addEntry 是构建内存 XML 模型树,Assets 是加入到。Apk 包中的文件。从此入手,搜索"addEntry",compileResourceFile ( ),generateAttributes

5.2.3 isValidResourceType

```
1  bool isValidResourceType(const String8& type)
2  {
3      return type == "anim" || type == "drawable" || type == "layout"
4          || type == "values" || type == "xml" || type == "raw"
5          || type == "color" || type == "menu";
6  }
```

5.2.4 parsePackage

调用 parseXMLResource 处理 AndroidManifest.xml 文件,涉及节点"package" 以及"uses-sdk"。AaptAssets 根据"package" 的值调用 setPackage 。

buildResources 是其唯一被调用的地方。从其中代码可以,AaptGroup 的是同名文件的集合,对于"AndroidManifest.xml" 只需取其第一个即可。

```
1  status_t buildResources(Bundle* bundle, const sp<AaptAssets>& assets
     )
2  {
```

```
3       // First, look for a package file to parse.  This is required to
4       // be able to generate the resource information.
5       sp<AaptGroup> androidManifestFile =
6               assets->getFiles().valueFor(String8("AndroidManifest.xml
                  "));
7       if (androidManifestFile == NULL) {
8           fprintf(stderr, "ERROR: No AndroidManifest.xml file found.\n
                  ");
9           return UNKNOWN_ERROR;
10      }
11
12      status_t err = parsePackage(bundle, assets, androidManifestFile)
          ;
13      .. .. ..
14  }
15
16  static status_t parsePackage(Bundle* bundle, const sp<AaptAssets>&
      assets, const sp<AaptGroup>& grp)
17  {
18      .. .. ..
19      sp<AaptFile> file = grp->getFiles().valueAt(0);
20
21      ResXMLTree block;
22      status_t err = parseXMLResource(file, &block);
23      .. .. ..
24  }
```

5.2.5 collect_files

这个函数就是把 AaptDir 中的 AaptGroup 中按资源类型(ResourceType)组织为 ResourceTypeSet[??]。不同的 ResourceTypeSet 以链表 resources (KeyedVector<String8, sp<ResourceTypeSet> >*) 保存在一起。

他遍历 AaptDir 的各个 AaptGroup，然后取出 AaptFile 的链表 files，由于 AaptGroup 中保存的 AaptFile 的 mResourceType 是一样的，所以就对第一个 AaptFile 调用 getResourceType() 获取资源类型。获取资源类型资源，查看 resources 中有无此资源类型，无则创建资源类型 set，把 AaptGroup 加入到 set，再把 set 挂到列表 resources。若已有，则查看资源类型有无相同的 AaptGroup，这是根据 AaptGroup 的 mLeaf (即文件名)，没有 AaptGroup 就简单，把他加入到 set 中即可；如果有的话，把新的 AaptGroup 的 AaptFile 加入到已存在的 AaptGroup 中。

DefaultKeyedVector<AaptGroupEntry, sp<AaptFile>

5.2.6 getResourceFile

getResourceFile 就是为了获取 resources.arsc。(assets->getFiles().valueFor(String8("re
因为 AaptAssets 继承 AaptDir，getFiles 是后者实现的，也就是说在 AaptAssets 的根目录下找 resources.arsc 这个文件，没有的话就加入 resources.arsc 这个节点(assets->addFile(String8("resources.arsc"),

AaptGroupEntry(), String8(), NULL, String8());)当然也是在根目录下，addFile 也是类 AaptDir 的接口

5.2.7 postProcessImages

postProcessImage 处理 drawable 目录。

5.2.8 writeResourceSymbols

writeResourceSymbols 生成文件 R.java 及 Manifest.java，这两个文件中的头部声明"/* AUTO-GENERATED FILE. DO NOT MODIFY. ..." 就是由这个函数打印的，但是她仅仅是合成文件名和路径，打印头部声明，真正干活还要靠 writeSymbolClass ，即 writeResourceSymbols 调用了 writeSymbolClass，这两个函数均在 Resource.cpp 实现。

因为可以看到 AUTO-GENERATED FILE. DO NOT MODIFY. 头部内容 writeSymbolClass 是干活。

5.2.9 writeSymbolClass

5.3 ResourceTable.cpp 的主要函数

强调一下，ResourceTable 的对象是在 buildResources 函数开始创建的，别的地方没有创建他的。

5.3.1 compileXmlFile

compileXmlFile 是以文件为单位编译的，编译后还是一个文件，但不再是文本文件，而是二进制文件，APK 资源的目录结构和源代码中的一样，即编译前后文件的相对路径(从 res 开始)没改变。R.java 相对应的资源是 resources.arsc，这里当然有以文件为单位的资源索引，也有像 resources 标记下的各种资源。

```
1    status_t err = root->parseValues(assets, table);
2    err = root->flatten(target,
3              (options&XML_COMPILE_STRIP_COMMENTS) != 0,
4              (options&XML_COMPILE_STRIP_RAW_VALUES) != 0);
5
6    //设置文件的压缩算法
7    target->setCompressionMethod(ZipEntry::kCompressDeflated);
8
9    assignResourceIds
10
11   generateAttributes
```

```
12    setIndex     setEntryIndex
13    sp<Entry>::assignResourceIds
```

5.3.2 compileResourceFile

compileResourceFile 从函数体开头部分字符串的定义可知，其解析 <resources/> 标签下的元素，

- Top-level tag : resources
- Identifier declaration tags : declare-styleable, attr
- Data creation organizational tags : string, drawable, color, bool, integer, dimen, fraction, style, plurals, array, string-array, integer-array, public-padding, private-symbols, add-resource, skip, eat-comment
- Data creation tags : bag, item
- Attribute type constants : enum
- plural values : other, zero, one, two, few, many,
- useful attribute names and special values: name, translatable, false

最后两个不是 tag，而是属性名和值

代码 9: Android XML 举例 (续)

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <resources>
3    <dimen name="title_texture_width">120px</dimen>
4    <color name="delete_color_filter">#A5FF0000</color>
5    <integer name="config_allAppsBatchSize">0</integer>
6    <bool name="lockscreen_isPortrait">true</bool>
7
8    <style name="AnimationPreview">
9        <item name="android:windowEnterAnimation">@anim/fade_in_fast</
           item>
10       <item name="android:windowExitAnimation">@anim/fade_out_fast</
           item>
11   </style>
12
13   <string-array name="menu_task_operation">
14         <item>@string/switch</item>
15         <item>@string/uninstall</item>
16   </string-array>
17
18   <attr name="direction">
19       <enum name="vertical" value="0" />
20       <enum name="horizontal" value="1" />
21   </attr>
22
23   <skip />
24   <style name="Widget.KeyboardView" parent="android:Widget">
25       <item name="android:background">@android:drawable/
           keyboard_background</item>
```

```
26          <item name="android:keyBackground">@android:drawable/
              btn_keyboard_key</item>
27          <item name="android:verticalCorrection">-10dip</item>
28          <item name="android:shadowColor">#BB000000</item>
29      </style>
30
31      <declare-styleable name="Favorite">
32          <attr name="className" format="string" />
33          <attr name="title" format="reference" />
34      </declare-styleable>
35      <declare-styleable name="AndroidManifestApplication" parent="
          AndroidManifest">
36          <attr name="name" />
37          <attr name="theme" />
38          <attr name="label" />
39          <attr name="icon" />
40          .. .. ..
41      </declare-styleable>
42      <private-symbols package="com.android.internal" />
43      <public type="attr" name="screenDensity" id="0x010102cb" />
44      <public type="drawable" name="presence_video_away" id="0x010800ac"
          />
45      <public type="style" name="Theme.Wallpaper.NoTitleBar" id="0
          x0103005f" />
```

compileResourceFile 函数比较庞大，有 800 多行，但是结构还是简单的。原理跟 PackageParser.java 解析 AndroidManifest.xml 一样，即把标记流按层处理，这也是流 -拉模型的典型处理方法了。下面简单分析一下:

- 定义标签变量
  基本上变量名就是标签名加上 16，标签名有'-'，变量名换为'_'，比如 string16 表示标签"string"。

  ```
  1  const String16 skip16("skip");
  2  const String16 eat_comment16("eat-comment");
  3
  4  // Data creation tags.
  5  const String16 bag16("bag");
  6  const String16 item16("item");
  ```

- parseXMLResource[??]
  这个函数比较简单，现在不清楚具体做什么，但可以肯定的是她调用 XML 解析器引擎初始化 ResXMLTree 结构。
- 拉出 resources 标记
  跳过所有的(命名空间)声明后，判断是否为标记，然后检查标记名

  ```
  1  if (strcmp16(block.getElementName(&len), resources16.string())
       != 0) {
  2      ... ...
  3      return UNKNOWN_ERROR;
  4  }
  ```

  resources16 就是值为"resources" 的常量对象。
- 处理 resources 标记
  这个语句块的结构如下：

  ```
  1  while ((code=block.next()) != ResXMLTree::END_DOCUMENT && code
       != ResXMLTree::BAD_DOCUMENT) {
  ```

```
2       if (code == ResXMLTree::START_TAG) {
3           ... ...     // 处理语句块tag
4       }
5       else if (code == ResXMLTree::START_NAMESPACE || code ==
          ResXMLTree::END_NAMESPACE) {
6       }
7       else if (code == ResXMLTree::TEXT) {
8           if (isWhitespace(block.getText(&len))) {
9               continue;
10          }
11          ... ...
12          return UNKNOWN_ERROR;
13      }
14  }
```

处理 tag 语句块基本结构就是 if-else if-else 的控制语句，判断条件就是 tag 名，tag 处理一般形式如下：

```
1   if (strcmp16(block.getElementName(&len), skip16.string()) == 0) {
2           ... ... // 子标记解析语句块
3           while ((code=block.next()) != ResXMLTree::END_DOCUMENT
4                   && code != ResXMLTree::BAD_DOCUMENT) {
5               if (code == ResXMLTree::END_TAG) {
6                   if (strcmp16(block.getElementName(&len), skip16.
                      string()) == 0) {
7                       break;
8                   }
9               }
10              else if (code == ResXMLTree::START_TAG) {
11                  ... ... //子标记中的子标记处理语句
12              }
13          }
14          continue;
15
16      } else if () {
17          ......
18      }
```

针对每个标签要给下列的值赋值：curTag ,curType ,curFormat ,curIsBag ,curIsBagReplaceOnOverwrite ,curIsStyled ,curIsPseudolocalizable ,curIsFormatted localHasErrors

skip16、eat_comment16 最简单的形式，继续拉，直至标记结束，没有解析语句，因为这两标签是注释的。同样是 xml 的注释，放在这两个标签里的注释是不会出现在 R.java 或 Manifest.java

public16，public_padding16，private_symbols16，add_resource16，有子标记解析语句块;

attr16 调用了 compileAttribute 处理。

declare_styleable16 标记处理比较麻烦，因为底下又子标记，所以有子标记中的子标记处理语句，这些标记有 skip16，eat_comment16，attr16 。(addSymbol, appendComment) 以上这些标签处理完了直接 continue，执行下一次大循环。

item16，string16，drawable16，color16, bool16, integer16, dimen16, fraction16 以及 bag16，style16，plurals16，array16，string_array16，integer_array16 根据标记名仅仅作了少量处理，比如设置了一些变量名，然后放在后面统一处理。他们都有一个共通的特点，就是有"name" 属性。但自 bag16 后的标记，需要是个容器或者说袋子，里面还有 item16 标记要处理，与 declare_styleable16 ，处理大概流程就是上面的代码片断，完了调用 parseAndAddBag 。前面的其他标记处理就比较简单，调用 parseAndAddEntry 即可。

因为对这些标签处理都是比较类似的，在他们的 if/else if 里面只是修改了 curTag ,curType ,curFormat ,curIsBag 值，然后统一处理，不想上面的，直接 continue，跳到大的 while 循环。对数组 (array, string16-array)、复数、style 都是 bag，是个容器，要确定其父节点：

```
1  String16 ident;
2  ssize_t identIdx = block.indexOfAttribute(NULL, "name");
3  if (identIdx >= 0) {
4      ident = String16(block.getAttributeStringValue(identIdx, &len));
5  } else {
6  .. .. ..
7  // Figure out the parent of this bag...
8  String16 parentIdent;
9  ssize_t parentIdentIdx = block.indexOfAttribute(NULL, "parent");
10 if (parentIdentIdx >= 0) {
11     parentIdent = String16(block.getAttributeStringValue(
         parentIdentIdx, &len));
12 } else {
13     ssize_t sep = ident.findLast('.');
14     if (sep >= 0) {
15         parentIdent.setTo(ident, sep);
16     }
17 }
```

bag 的父节点先是查找属性 parent 的值，若无则分析属性 name 的值，其值"." 之前的部分就是其父节点。

```
1  <style name="Theme.Dialog.AppError">
2  <style name="Theme.SearchBar" parent="Theme.Panel">
```

第一种，父节点就是 Theme.Dialog 的 style。第二种，父节点是 Theme.Panel 的 style。

5.3.3 compileAttribute

compileAttribute(R.java declare-style 的属性的产生)

5.3.4 ResourceTable:addIncludedResources

回过头看，buildResources 开头的一条语句

```
1  err = table.addIncludedResources(bundle, assets);
```

ResourceTable::addIncludedResources 函数的最后面调用 ResourceTable::getType 建立了 attr 类型，即 attr 类，这就是 R.java 的开头总是能看到内部类 attr，即使是空的。

5.3.5 Package 类和 ResourceTable::getPackage

```
1               mHaveAppPackage = true;
2               p = new Package(package, 127);
3           } else {
4               p = new Package(package, mNextPackageId);
5           }
```

127 即 0x7F, 这是 R.java 常见的 0x7fxxxxxx 的来源吧？Entry. Package 表示一个 apk 包，或者说是 R 类。

ResourceTable::getPackage 跟 mNextPackageId 关系不大，mNextPackageId 为 2“*** NEW PACKAGE: "com.android.speechrecorder" id=127” ResourceTable::mNextPackageId 与分配的 Package ID 关系密切，ResourceTable::ResourceTable 构造函数把此值初始化为 1。Package ID 的值一般只有两个

1 → android
127 → com.android.launcher

android 、com.android.launcher 是 AndroidManifest.xml 的"manifest" 标记的 package 属性值，只不过一个是 frameworks/base/core/res 下的，表示 Android 系统的，一个是普通应用程序的。

仅在 ResourceTable.cpp 模块内部使用，由 getPackage 创建 Package 对象。搜索"new\s*Package" 即可定位创建对象的地方。

5.3.6 Type 类和 ResourceTable::getType

ResourceTable::Package::getType 包装，先在 mType 查找，找不到创建一个 Type 对象。获取 Type，getType 查询 Package 的 mTypes 里有没有这个 Type，查询没有就建立一个。

Type 表示一个 R 的内部类。Type 不直接包含 Entry，而是包含了 ConfigList

```
1  SortedVector<ConfigDescription> mUniqueConfigs;
2  DefaultKeyedVector<String16, sp<ConfigList> > mConfigs;
3  Vector<sp<ConfigList> > mOrderedConfigs;
```

ConfigList 才拥有 Entry：

```
1  DefaultKeyedVector<ConfigDescription, sp<Entry> > mEntries;
```

也就是说同一个 Type 的 Entry 按配置归类，即同一个文件名在有不同配置，他们都放在同一个 ConfigList，这就是相当于 AaptGroup。ConfigDescription 相当于 AaptGroupEntry, Entry 相当于 AaptFile。ConfigDescription 是 ResTable_config 的派生类，但没有增加数据成员，

ResTable_config 对配置的描述的用 union，每个 union 的域是基本数据类型，如 uint8_t, uint16_t, uint32_t, 面向机器；而 AaptGroupEntry 中对配置的描述都是 String8，即字符串，面向程序员。

5.3.7 Entry 类和 ResourceTable::addEntry

他实际上就是 ResourceTable::getEntry 包装，后者调用 getType ?? 获得对应的 Type，再调用 Type::getEntry 获取 Entry。先查询，若无就 new 一个 Entry。Entry 表示 R 的内部类的一个属性。

5.3.8 ResourceTable::getResId

这个函数名对应了三个函数，其获取资源 id 的过程类似于 stringToValue 对引用的解析 ?? 的三部曲：expandResourceRef, identifierForName, 获取资源 id。最原始的

```
1  uint32_t ResourceTable::getResId(const String16& ref,
2                                   const String16* defType,
3                                   const String16* defPackage,
4                                   const char** outErrorMsg,
5                                   bool onlyPublic) const
```

只知道资源项的引用(即完整的资源名)，不知到具体类型和包，需要调用 ResTable::expandResourceRef 解析出来，这里注意 defPackage 在调用时赋为 NULL，那么，传给 expandResourceRef 就是 mAssetsPackage，后者是 ResourceTable 初始化时赋值的 ??。然后他调用上一级的：

```
1  uint32_t ResourceTable::getResId(const String16& package,
2                                   const String16& type,
3                                   const String16& name,
4                                   bool onlyPublic) const
```

这里知道了包名(package)、类型(type)、资源名(name), 他先资源名去查所包含的资源池里有无此资源项

```
1      uint32_t rid = mAssets->getIncludedResources()
2          .identifierForName(name.string(), name.size(),
3                             type.string(), type.size(),
4                             package.string(), package.size(),
5                             &specFlags);
```

此后的处理流程也是和 stringToValue 对引用的解析 ?? 类似的。

如果这个步骤获取不到，即所包含的资源池没有此资源，在当前解析的资源包里找，其原理同 ResourceTable::getCustomResource，虽然 getResId 自己写了代码，但是一致的。

```
1  inline uint32_t ResourceTable::getResId(const sp<Package>& p,
2                                          const sp<Type>& t,
3                                          uint32_t nameId)
```

```
4 {
5     return makeResId(p->getAssignedId(), t->getIndex(), nameId);
6 }
```

makeResId 就是把三个数值移位，位或运算。

5.3.9 ResourceTable::getCustomResourceWithCreation

这个函数会创建新的资源项 (Entry), 并返回 id

- getCustomResource
- addEntry
- getResId

Custom 就是自定义的意思，针对“@+”这种。创建资源主要是 addEntry。

5.3.10 ResourceTable::assignResourceIds

此函数只被 buildResources[??] 调用。ResourceTable::Entry::assignResourceIds ResourceTable::Package::getAssignedId ResourceTable::Package::mIncludedId 只在对象初始化时涉及到

5.3.11 ResourceTable::flatten

5.4 XMLNode.cpp 的主要函数

5.4.1 parseXMLResource

parseXMLResource 初始化 XML 解析。因为 parseXMLResource 调用方法 XMLNode::parse，后者正是调用 XML_Parser 解析器引擎。XMLNode::parse 通过读取 XML 文件把标记保存到内存中，解析器的用户数据是一个 ParseState 类型结构体(她是 XMLNode 类的私有结构体)变量，解析出来的数据保存在成员 stack（这是一个 Vector 对象，元素是 sp<XMLNode>)，结构体成员 root（sp<XMLNode> 类型)是指向 stack 中的第一个节点。TODO：:parseXMLResource 转化到 XMLNode 树形结构，再到 AaptFile，最后到 ResXMLTree 结构。从 XML 的树形 -》Expat 的流 -》XMLNode 的树 -》ResXMLTree

5.5 主要输出函数和数据结构

5.5.1 writePublicDefinitions

-P 选项(不是-p[ackage])指定的文件，<public type="string" name="group_-applications" id="0x7f0c000f" />

-G out/target/common/obj/APPS/Launcher2_intermediates/proguard_options

5.6 Package.cpp 主要函数

5.6.1 writeAPK

Package.cpp 定义的，生成.apk，APK 的文件名由 getOutputAPKFile 获取 TODO：:可以比较 writeAPK 与 writeSymbolClass 的异同啊！！！

创建一个 ZipFile 文件 zip，先由 processAssets 对 AaptAssets 处理，把结果放入 zip 中，然后，processJarFiles 再对 zip 进行处理，最后，删除不必要的文件。

5.7 AaptAssets.cpp

5.7.1 AaptDir::makeDir

```
1  sp<AaptDir> AaptDir::makeDir(const String8& path)
2  {
3      String8 name;
4      String8 remain = path;
5
6      sp<AaptDir> subdir = this;
7      while (name = remain.walkPath(&remain), remain != "") {
8          subdir = subdir->makeDir(name);
9      }
10
11     ssize_t i = subdir->mDirs.indexOfKey(name);
12     if (i >= 0) {
13         return subdir->mDirs.valueAt(i);
14     }
15     sp<AaptDir> dir = new AaptDir(name, subdir->mPath.appendPathCopy
         (name));
16     subdir->mDirs.add(name, dir);
17     return dir;
18 }
```

subdir 初始值是 this。对于 path 为"res"，如果未对其 makeDir，while、if 两个判断都不成立，创建一个 AaptDir，并加入到 subdir; 以后对其直接 makeDir，在 if 中返回。对于 path 为"res/layout/mypng"，初次对其直接

makeDir，在 while 处，第一次 name 为"res"，递归调用返回"res" 所在的 AaptDir，这时 subdir 换成了"res" 所在的；第二次，name 为"layout"，while，if 为假，新建一个 AaptDir，并返回，subdir 为"layout" 所在的目录,while,if 不成立，新建 AaptDir；两次循环后，name 为"mypng"，while 条件不成立，循环结束；最先的 makeDir 才走到 if 判断，这时 name 为 mypng，if 不成立成立。第二次及其后面，对"res/layout" 直接 makeDir，while 循环的 makeDir 调用中 while 不成立，if 条件成立。

对于 while 要说明一下，发起调用的 makeDir 的执行中，他要等待 while 循环执行完毕，才能往下执行。而由 while 里发起的 makeDir 执行中，不管是发起的、还是发起的 while 中的 subdir 等，其 while 循环都不会被执行，因为其 path 参数不含有"/"。发起的调用在 while 循环执行结束后 (只要其进入了 while)，subdir 是倒数次级，或是主目录，如 path 为"res" 或"res/drawable",subdir 就指向"res"，paht 为"res/-drawable/mypng", subdir 就是"drawable"。这个函数执行的结果就是形成跟目录一样的树形结构。并且返回最末端(最深)的目录所在的目录，如果没有就是他自己。

对于不存在的目录 foo/bar/a，通过 while 循环中的递归，迭代最后，while 及 if 判断都不成立, 执行最后的新建 AaptDir 对象，最后效果同 mkdir -p foo/bar/a。如果 foo/bar/a 已存在，那么最后递归到 if 返回，前面的递归也到 if 止。所以，while 循环可以保证目录创建。其最大递归深度与目录的路径深度关联。

while 与 if 的先后顺序可否对换？

5.7.2 AaptAssets::addResource

```
1  void AaptAssets::addResource(const String8& leafName, const String8&
       path,
2                  const sp<AaptFile>& file , const String8& resType)
3  {
4      sp<AaptDir> res = AaptDir::makeDir(kResString);
5      String8 dirname = file->getGroupEntry().toDirName(resType);
6      sp<AaptDir> subdir = res->makeDir(dirname);
7      sp<AaptGroup> grr = new AaptGroup(leafName, path);
8      grr->addFile(file);
9
10     subdir->addFile(leafName, grr);
11 }
```

第一行，由于 AaptAssets 是 AaptDir 的派生类，所以可以调用父类的 makeDir，kResString 是全局的，指向"res" 目录。所以，res 就是代表目录"res"。

两个疑问 :dirname ，包含“res”？每个 AaptGroup 只有一个文件？从

addResource 看是这样子的。后续有无合并的动作？

5.8 注释

R.java 里的注释函数 compileAttribute、compileResourceFile, mayOrMust, 以及数组 gFormatFlags 均会给节点增加一点注释。而 declare-styleable 声明属性时，生成的 attr、styleable 内部类及相关的域，会有大量的注释，他们是有 writeLayoutClasses 搞的。

appendComment

5.9 属性

只有在编译时检测到的属性才能在 XML layout 文件中设置。标签的值列表参见：frameworks/base/core/res/res/values/public.xml frameworks/base/api/*.xml

XMLNode.cpp 定义了几个常见的命名空间（NameSpace）XLIFF_XMLNS RESOURCE_ROOT_NAMESPACE 上面的标记会有很多的属性，这些属性怎么来的呢？这些属性都是由 declare-styleable 声明的，Android 系统级的属性，参见 SDK 文档 docs/reference/android/package-summary.html docs/reference/android/R.id.html 或者编译后的文档 out/target/common/docs/offline-sdk/reference/android/package-summary.html out/target/common/docs/offline-sdk/reference/android/R.id.html 应用中，我们一般把声明属性的 declare-styleable 标记放在文件 res/values/attrs.xml，而使用这些属性的 XML 文件不能与 attrs.xml 同目录，即不能放在 res/values，可以放在 res/xml, res/color, res/anim 等等，或者 res/drawable 也行？

parseStyledString 递归函数

后者调用调用 compileAttribute (R.java declare-style 的属性的产生) 解析文件中的各个标签。

5.10 多语言

string16 中可能与多语言相关的函数 translatable16, getAttributeStringValue, getResourceSourceDirs。

5.11 覆盖

```
1  static bool applyFileOverlay(Bundle *bundle,
2                               const sp<AaptAssets>& assets,
3                               sp<ResourceTypeSet> *baseSet,
4                               const char *resType)
```

```
5
6       // look for same flavor.  For a given file (strings.xml, for
          example)
7       // there may be a locale specific or other flavors – we want to
          match
8       // the same flavor.
```

5.12 文件压缩

ZipFile* zip = openReadWrite(zipFileName, false); 这个是打开apk压缩文件，apk采用的压缩格式其实就是Zip算法。ZipEntry就是压缩文件的单位文件的数据结构。

```
1  ZipEntry* entry = zip->getEntryByName(fileName);
2
3  zip->remove(entry);                                          //删除
4  zip->add(fileName, bundle->getCompressionMethod(), NULL);    //增加
5  zip->addGzip      //增加 .gz 文件
6  zip->flush();
7  delete zip;
```

参考文献

[1] frameworks/base/libs/utils/AssetManager.cpp

[2] frameworks/base/core/jni/android_util_AssetManager.cpp,

[3] frameworks/base/core/jni/android_util_XmlBlock.cpp,

[4] frameworks/base/core/jni/android_util_StringBlock.cpp

[5] frameworks/base/core/java/android/content/res/AssetManager.java

[6] frameworks/base/core/java/android/content/pm/PackageParser.java

[7] frameworks/base/core/java/android/content/pm/PackageManager.java

[8] frameworks/base/native/android/asset_manager.cpp

[9]