Android逆向笔记 —— AndroidManifest.xml 文件格式解析

转载

u012551350 于 2019-05-28 08:00:00 发布 1504 收藏 7

温馨提示

请拖动到文章末尾，长按识别「抽奖」小程序。

做过 Android 开发的同学对

```
AndroidManifest.xml
```

文件肯定很熟悉，我们也叫它 **清单文件** ，之所以称之为清单文件，因为它的确是应用的 “清单”。

它包含了应用的包名，版本号，权限信息，所有的四大组件等信息。

在逆向的过程中，通过 apk 的清单文件，我们可以了解应用的一些基本信息，程序的入口 Activity，注册的服务，广播，内容提供者等等。

如果你尝试查看过 apk 中的

```
AndroidManifest.xml
```

文件，你会发现你看到的是一堆乱码，已经不是我们开发过程中编写的清单文件了。

因为在打包过程中，清单文件被编译成了二进制数据存储在安装包中。

这就需要我们了解

```
AndroidManifest.xml
```

的二进制文件结构，才可以读取到我们需要的信息。当然，已经有一些不错的开源工具可以读取编译后的清单文件，像

```
AXmlPrinter
```

，

```
apktool
```

等等。

当然，正是由于这些工具都是开源的，一些开发者会利用其中的漏洞对清单文件进行特定的处理，使得无法通过这些工具反编译清单文件。

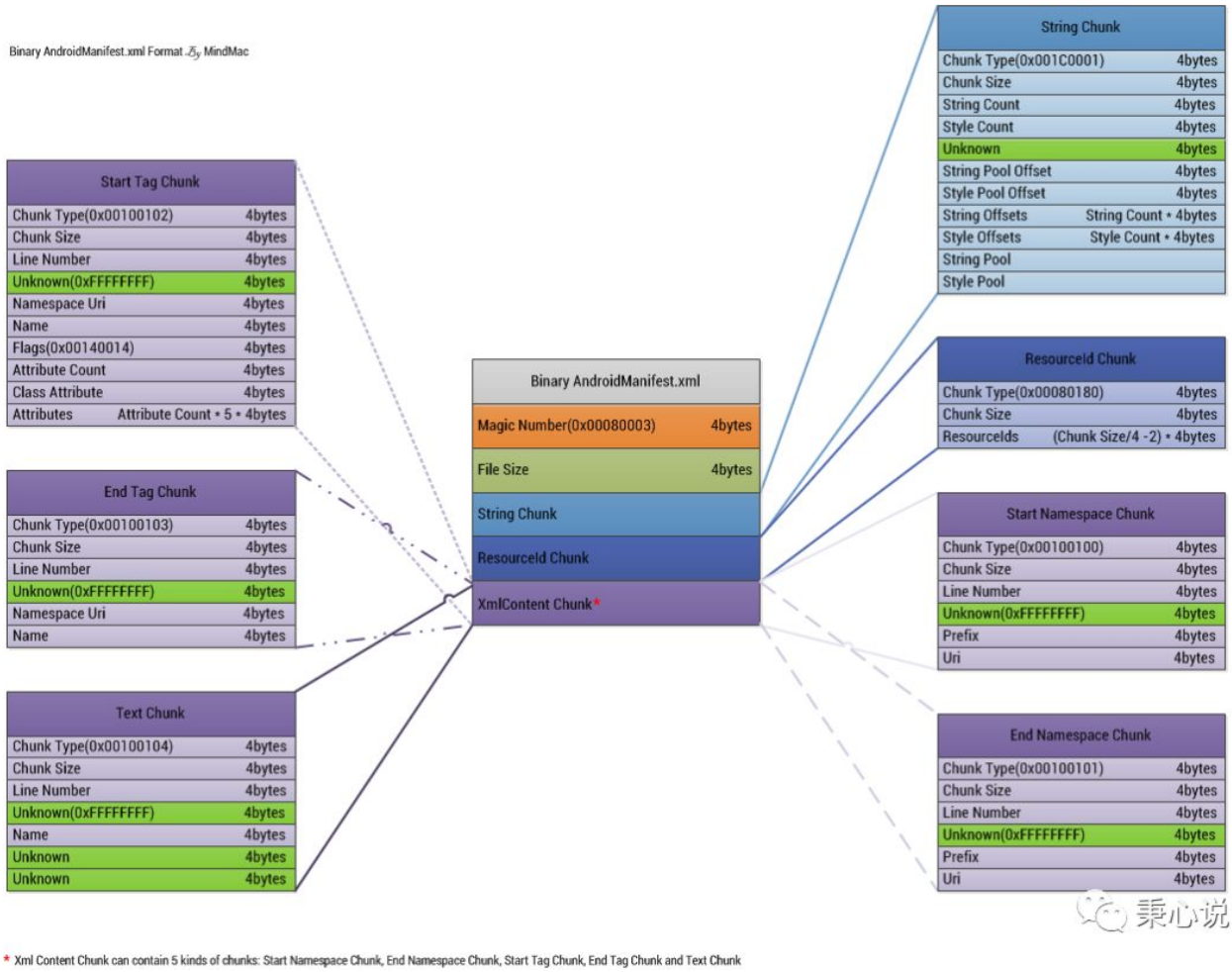
如果我们了解其二进制文件结构的话，就可以对症下药了。

和之前解析 Class 文件结构一样，仍然手写代码进行解析，这样才能真正的了解其文件结构。通过前辈们的资料和

```
010 editor
```

的使用，其实已经大大降低了解析的难度。

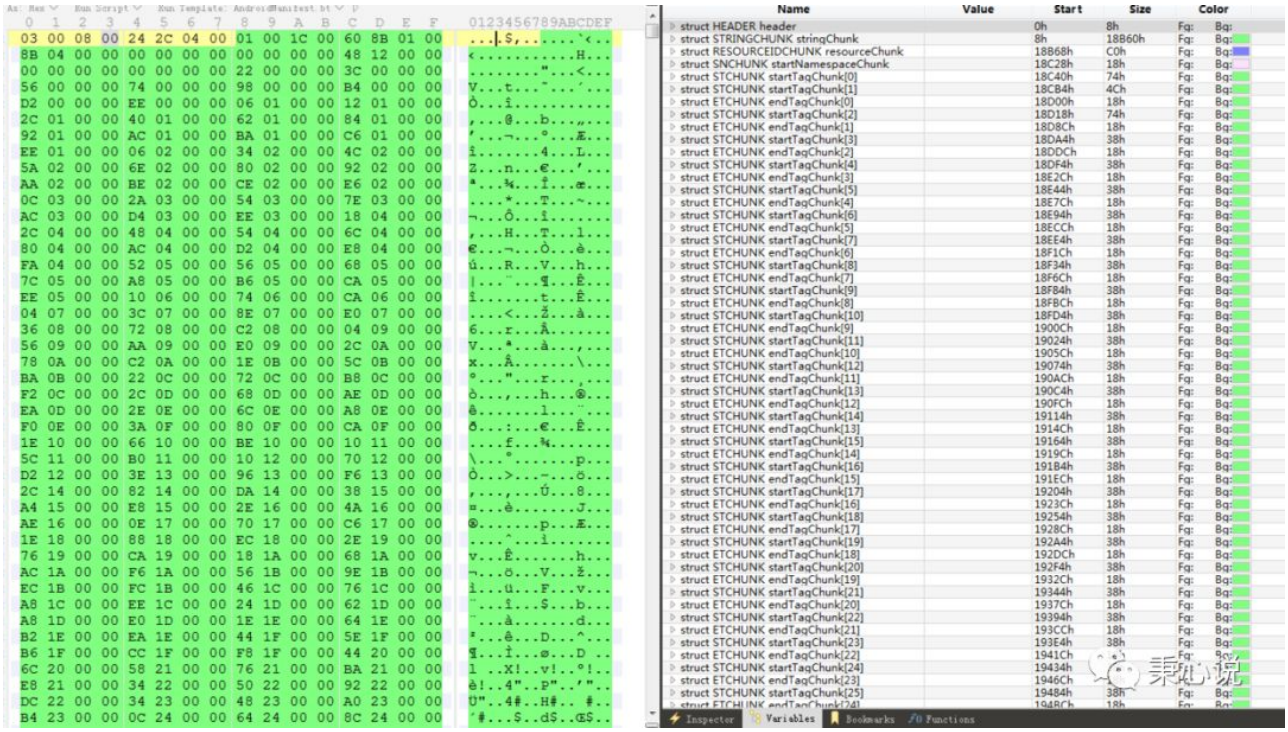
首先上一张看雪大神 MindMac 的神图（原图链接）：



xml_binary.png

这张图真的很经典，不妨可以打印出来对照着进行分析。

这篇文章以 QQ 的清单文件为例进行分析，下载 QQ 的安装包解压即可拿到清单文件。解析文件格式的惯例，首先用 010 editor 打开，基本结构如下图所示：



xml_all.png

运行的 Template 是

```
AndroidManifest.bt
```

。结合上面的结构图，对

```
AndroidManifest.xml
```

的总体结构应该有了大概的了解。总体上按顺序分为四大部分：

二进制

```
AndroidManifest.xml
```

大致上就是按照这几部分顺序排列组成的，下面就逐一部分详细解析。在这之前还需要知道的一点是，清单文件是小端表示的，ARM 平台下大多数都是小端表示的。

Header

xml_header.png

头部由

`Magic Number`

和

`File Size`

组成，各自都是 4 字节。

对应的解析代码：

-
-
-
-
-
-
-
-
-
-
-
-
-

```
private void parseHeader() {
    try {
        Xml.nameSpaceMap.clear();
        String magicNumber = reader.readHexString(4);
        log("magic number: %s", magicNumber);

        int fileSize = reader.readInt();
        log("file size: %d", fileSize);
```

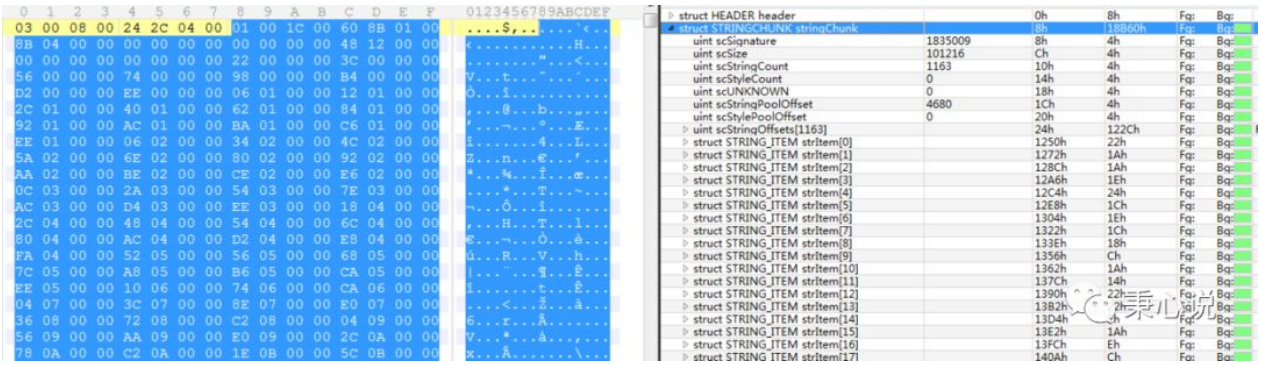
```
    } catch (IOException e) {
        e.printStackTrace();
        log("parse header error!");
    }
}
```

解析结果：

```
magic number: 0x00080003
file size: 273444
```

String Chunk

先来看一下 010 editor 中这一块的内容：

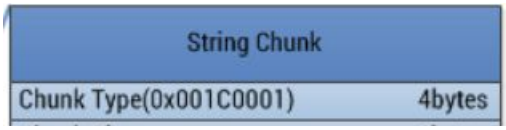


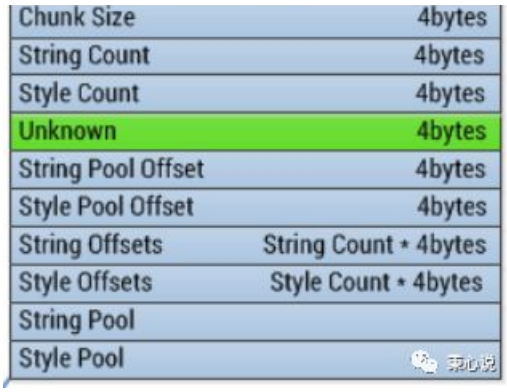
xml_string_chunk.png

对应看雪神图的

StringChunk

模块：





kanxue_string_chunk.png

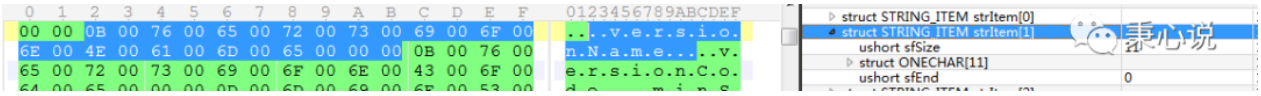
`String Chunk`

主要存储了清单文件中的所有字符串信息。结构还是很清晰的。结合上图逐条解释一下：

字符串池中的字符串存储也有特定的格式，以

`versionName`

为例：



xml_versioname.png

前两个字节表示字符串的字符数，注意一个字符是两个字节。如上图所示，字符数为

`11`

，则后面

`22`

个字节表示字符串内容，最后以

`0000`

结尾。如此循环。

样式池在解析过程中一般都为空，样式数量也为 `0`。

了解了

`String Chunk`

的结构之后，解析就很简单了。直接上代码：

```
private void parseStringChunk() {
        try {
            String chunkType = reader.readHexString(4);
            log("chunk type: %s", chunkType);

            int chunkSize = reader.readInt();
            log("chunk size: %d", chunkSize);

            int stringCount = reader.readInt();
            log("string count: %d", stringCount);

            int styleCount = reader.readInt();
            log("style count: %d", styleCount);

            reader.skip(4);  // unknown

            int stringPoolOffset = reader.readInt();
            log("string pool offset: %d", stringPoolOffset);

            int stylePoolOffset = reader.readInt();
            log("style pool offset: %d", stylePoolOffset);

            // 每个 string 的偏移量
            List<Integer> stringPoolOffsets = new ArrayList<>(stringCount);
            for (int i = 0; i < stringCount; i++) {
                stringPoolOffsets.add(reader.readInt());
            }

            // 每个 style 的偏移量
            List<Integer> stylePoolOffsets = new ArrayList<>(styleCount);
            for (int i = 0; i < styleCount; i++) {
                stylePoolOffsets.add(reader.readInt());
            }

            log("string pool:");
            for (int i = 1; i <= stringCount; i++) { // 没有读最后一个字符串
                String string;
                if (i == stringCount) {
                    int lastStringLength = reader.readShort() * 2;
                    string = new String(moveBlank(reader.readOrigin(lastStringLength)));
                    reader.skip(2);
                } else {
                    reader.skip(2); // 字符长度
                    // 根据偏移量读取字符串
                    byte[] content = reader.readOrigin(stringPoolOffsets.get(i) - stringPoolOffsets
                    reader.skip(2); // 跳过结尾的 0000
                    string = new String(moveBlank(content));

                }
```

```
                }
                log("   %s", string);
                stringChunkList.add(string);
            }


            log("style pool:");
            for (int i = 1; i < styleCount; i++) {
                reader.skip(2);
                byte[] content = reader.readOrigin(stylePoolOffsets.get(i) - stylePoolOffsets.get(i
                reader.skip(2);
                String string = new String(content);
                log("   %s", string);
            }

        } catch (IOException e) {
            e.printStackTrace();
            log("parse StringChunk error!");
        }
    }
```

解析结果如下：

```
chunk type: 0x001C0001chunk size: 101216string count: 1163style count: 0string pool offset: 4680style pool offset: 0string pool:    installLo
```

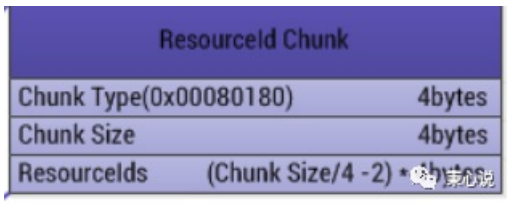
ResourceId Chunk

资源 Id 块，存储了清单文件中用到的系统属性的资源 Id 值。还是先看一下 010 edtior 中的对应块：

xml_resourceid_chunk.png

对应列表雷神图中

对应到看雪神图中：



kanxue_resourceid_chunk.png

解析代码：

```
private void parseResourceIdChunk() {
        try {
            String chunkType = reader.readHexString(4);
            log("chunk type: %s", chunkType);

            int chunkSize = reader.readInt();
            log("chunk size: %d", chunkSize);

            int resourcesIdChunkCount = (chunkSize - 8) / 4;
            for (int i = 0; i < resourcesIdChunkCount; i++) {
                String resourcesId = reader.readHexString(4);
                log("resource id[%d]: %s", i, resourcesId);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

解析结果：

```
chunk type: 0x00080180chunk size: 192resource id[0]: 0x010102B7resource id[1]: 0x0101021Cresource id[2]: 0x0101021Bresource id[3]: 0x01
```

XmlContent Chunk

这一块代码中存储了清单文件的详细信息。其中包含了五种 Chunk 类型，从下面的解析代码中就可以看出来：

```
private void parseXmlContentChunk() {
        try {
            while (reader.avaliable() > 0) {
                int chunkType = reader.readInt();
                switch (chunkType) {
                    case Xml.START_NAMESPACE_CHUNK_TYPE:
                        parseStartNamespaceChunk();
                        break;
                    case Xml.START_TAG_CHUNK_TYPE:
                        parseStartTagChunk();
                        break;
                    case Xml.END_TAG_CHUNK_TYPE:
                        parseEndTagChunk();
                        break;
                    case Xml.END_NAMESPACE_CHUNK_TYPE:
                        parseEndNamespaceChunk();
                        break;
                    case Xml.TEXT_CHUNK_TYPE:
                        parseTextChunk();
                        break;
                }
            }
        } catch (IOException e) {
            e.printStackTrace();
            log("parse XmlContentChunk error!");
        }
    }
```

通过

`chunkType`

来循环读取不同类型的 chunk 并进行解析。

每一种 chunk 都具有类似的数据结构，我定义了一个抽象类

`Chunk`

作为不同 chunk 的基类：

```
public abstract class Chunk {

    int chunkType; // 标识不同 chunk 类型
    int chunkSize; // 该 chunk 字节数
    int lineNumber; // 行号

    Chunk(int chunkType){
        this.chunkType=chunkType;
    }
```

```
    }

    public abstract String toXmlString();
}
```

这三个属性再加上

`Unkown(0xFFFFFFFF)`

，这前 16 个字节是这五种 chunk 中都有的，后面不再特别叙述。

下面依次解析这五种 Chunk ：

Start Namespace Chunk

`Start Namespace Chunk`

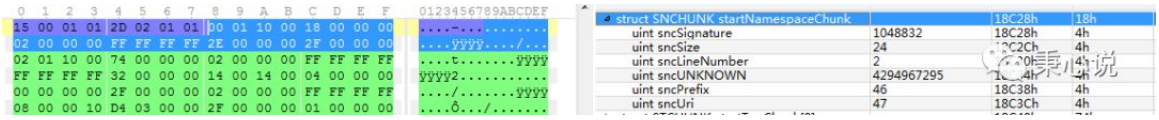
一般存储了清单文件的命名空间信息。再回顾一下

`Start Namespace Chunk`

的结构：

kanxue_start_namespace.png

对应 010 editor 中内容：



xml_start_namespace.png

前面四项不再解释，我们着重看一下最后两项

`Prefix`

和

`Uri`

。

`Prefix`

是一个索引值，4 字节，指向字符串池中对应的字符串，表示命名空间的前缀。

`Uri`

同样也是指向字符串池中对应索引的字符串，表示命名空间的 uri。看上图 010 editor 截图中的例子，

`Prefix`

值为

`46`

，

`Uri`

值为

`47`

。查看前面解析过的字符串池，发现这两个字符串分别是

`android`

和

`http://schemas.android.com/apk/res/android`

。看到这里应该很熟悉了，这的确是我们的

`AndroidManifest.xml`

文件的命名空间。

解析代码：

```
private void parseStartNamespaceChunk() {
        log("\nparse Start NameSpace Chunk");
        log("chunk type: 0x%x", Xml.START_NAMESPACE_CHUNK_TYPE);

        try {
            int chunkSize = reader.readInt();
            log("chunk size: %d", chunkSize);

            int lineNumber = reader.readInt();
```

```
            int lineNumber = reader.readInt();
            log("line number: %d", lineNumber);

            reader.skip(4); // 0xffffffff

            int prefix = reader.readInt();
            log("prefix: %s", stringChunkList.get(prefix));

            int uri = reader.readInt();
            log("uri: %s", stringChunkList.get(uri));

            StartNameSpaceChunk startNameSpaceChunk = new StartNameSpaceChunk(chunkSize, li
            chunkList.add(startNameSpaceChunk);

            Xml.nameSpaceMap.put(stringChunkList.get(prefix), stringChunkList.get(uri));
        } catch (IOException e) {
            e.printStackTrace();
            log("parse Start NameSpace Chunk error!");
        }
    }
```

解析代码很简单，按顺序读取就可以了。

需要注意的是我们把命名空间的后缀和 `uri` 的对应关系保存在了 `map` 中，供后面解析的时候使用。

End Namespace Chunk

此 `chunk` 与

`Start Namespace Chunk`

结构完全一致，解析过程也完全一致，不再赘述。

Start Tag Chunk

`Start Tag Chunk`

是所有 `chunk` 中结构最复杂的一个，存储了清单文件中最重要的标签信息。通过这一个 `chunk`，基本上就可以获取

`AndroidManifest.xml`

的所有信息了。

还是先回顾一下看雪神图：

Start Tag Chunk	
Chunk Type(0x00100102)	4bytes
Chunk Size	4bytes
Line Number	4bytes
Unknown(0xFFFFFFFF)	4bytes
Namespace Uri	4bytes
Name	4bytes
Flags(0x00140014)	4bytes
Attribute Count	4bytes
Class Attribute	4bytes
Attributes	Attribute Count * 5 * 4bytes

kanxue_start_tag.png

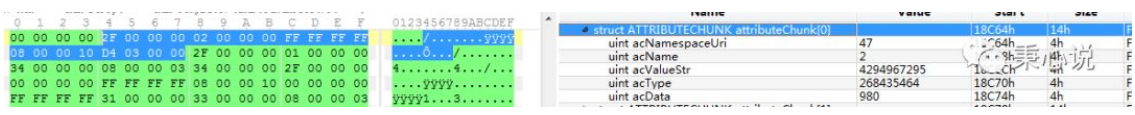
对应 010 editor 中的解析结果：

Name	Value	Start	Size
struct STCHUNK startTagChunk[0]		18C40h	74h
uint stcSignature	1048834	18C40h	4h
uint stcSize	116	18C44h	4h
uint stcLineNumber	2	18C48h	4h
uint stcUNKNOWN	4294967295	18C4Ch	4h
uint stcNamespaceUri	4294967295	18C50h	4h
uint stcName	50	18C54h	4h
uint stcFlags	1310740	18C58h	4h
uint stcAttributeCount	4	18C5Ch	4h
uint stcClassAttribute	0	18C60h	4h
struct ATTRIBUTECHUNK attributeChunk[0]		18C64h	14h
struct ATTRIBUTECHUNK attributeChunk[1]		18C78h	14h
struct ATTRIBUTECHUNK attributeChunk[2]		18C8Ch	14h
struct ATTRIBUTECHUNK attributeChunk[3]		18CA0h	14h

xml_start_tag.png

标签中包含了属性集合，这就是清单文件的重要组成部分。

属性也有固定的格式：



Name	Value	Start	Size
struct ATTRIBUTECHUNK attributeChunk[0]		18C64h	14h
uint acNamespaceUri	47	18C64h	4h
uint acName	2	18C68h	4h
uint acValueStr	4294967295	18C6Ch	4h
uint acType	268435464	18C70h	4h
uint acData	980	18C74h	4h

xml_attribute.png

每个属性固定 20 个字节，包含 5 个字段，每个字段都是 4 字节无符号 int，各个字段含义如下：

属性根据

`type`

的不同，其属性值的表达形式也是不一样的。比如表示权限的

`android:name="android.permission.NFC"`

，指向资源id 的

`android:theme="@2131624762"`

，表示大小的

`android:value="632.0dip"`

等等。Android 源码中就提供了根据

`type`

和

`data`

获取属性值字符串的方法，这个方法就是

`TypedValue.coerceToString(int type, int data)`

,代码如下：

```
/**
     * Perform type conversion as per {@link #coerceToString()} on an explicitly
     * supplied type and data.
     *
     * @param type
     *            The data type identifier.
     * @param data
     *            The data value.
     *
     * @return String The coerced string value. If the value is null or the type
     *         is not known, null is returned.
     */
    public static final String coerceToString(int type, int data) {
        switch (type) {
            case TYPE_NULL:
                return null;
            case TYPE_REFERENCE:
                return "@" + data;
            case TYPE_ATTRIBUTE:
                return "?" + data;
            case TYPE_FLOAT:
                return Float.toString(Float.intBitsToFloat(data));
            case TYPE_DIMENSION:
                return Float.toString(complexToFloat(data))
```

```
                        + DIMENSION_UNIT_STRS[(data >> COMPLEX_UNIT_SHIFT)
                        & COMPLEX_UNIT_MASK];
            case TYPE_FRACTION:
                return Float.toString(complexToFloat(data) * 100)
                        + FRACTION_UNIT_STRS[(data >> COMPLEX_UNIT_SHIFT)
                        & COMPLEX_UNIT_MASK];
            case TYPE_INT_HEX:
                return String.format("0x%08X", data);
            case TYPE_INT_BOOLEAN:
                return data != 0 ? "true" : "false";
        }

        if (type >= TYPE_FIRST_COLOR_INT && type <= TYPE_LAST_COLOR_INT) {
            String res = String.format("%08x", data);
            char[] vals = res.toCharArray();
            switch (type) {
                default:
                case TYPE_INT_COLOR_ARGB8:// #AaRrGgBb
                    break;
                case TYPE_INT_COLOR_RGB8:// #FFRrGgBb->#RrGgBb
                    res = res.substring(2);
                    break;
                case TYPE_INT_COLOR_ARGB4:// #AARRGGBB->#ARGB
                    res = new StringBuffer().append(vals[0]).append(vals[2])
                            .append(vals[4]).append(vals[6]).toString();
                    break;
                case TYPE_INT_COLOR_RGB4:// #FFRRGGBB->#RGB
                    res = new StringBuffer().append(vals[2]).append(vals[4])
                            .append(vals[6]).toString();
                    break;
            }
            return "#" + res;
        } else if (type >= TYPE_FIRST_INT && type <= TYPE_LAST_INT) {
            String res;
            switch (type) {
                default:
                case TYPE_INT_DEC:
                    res = Integer.toString(data);
                    break;
            }
            return res;
        }

        return null;
    }
```

我就直接引用这个方法进行属性的解析。

到这里，我们已经可以解析标签和属性了。对整个

```
Start Tag Chunk
```

的解析代码如下：

```
private void parseStartTagChunk() {
        log("\nparse Start Tag Chunk");
        log("chunk type: 0x%x", Xml.START_TAG_CHUNK_TYPE);

        try {
            int chunkSize = reader.readInt();
            log("chunk size: %d", chunkSize);

            int lineNumber = reader.readInt();
            log("line number: %d", lineNumber);

            reader.skip(4); // 0xffffffff

            int namespaceUri = reader.readInt();
            if (namespaceUri == -1)
                log("namespace uri: null");
            else
                log("namespace uri: %s", stringChunkList.get(namespaceUri));

            int name = reader.readInt();
            log("name: %s", stringChunkList.get(name));

            reader.skip(4); // flag 0x00140014

            int attributeCount = reader.readInt();
            log("attributeCount: %d", attributeCount);

            int classAttribute = reader.readInt();
            log("class attribute: %s", classAttribute);

            List<Attribute> attributes = new ArrayList<>();
            // 每个 attribute 五个属性，每个属性 4 字节
            for (int i = 0; i < attributeCount; i++) {

                log("Attribute[%d]", i);

                int namespaceUriAttr = reader.readInt();
                if (namespaceUriAttr == -1)
                    log("   namespace uri: null");
                else
                    log("   namespace uri: %s", stringChunkList.get(namespaceUriAttr));

                int nameAttr = reader.readInt();
                if (nameAttr == -1)
                    log("   name: null");
                else
                    log("   name: %s", stringChunkList.get(nameAttr));

                int valueStr = reader.readInt();
                if (valueStr == -1)
                    log("   valueStr: null");
                else
                    log("   valueStr: %s", stringChunkList.get(valueStr));

                int type = reader.readInt() >> 24;
                log("   type: %d", type);
```

```
                int data = reader.readInt();
                String dataString = type == TypedValue.TYPE_STRING ? stringChunkList.ge
                log("   data: %s", dataString);

                Attribute attribute = new Attribute(namespaceUriAttr == -1 ? null : str
                        stringChunkList.get(nameAttr), valueStr, type, dataString);
                attributes.add(attribute);
            }
            StartTagChunk startTagChunk = new StartTagChunk(namespaceUri, stringChunkLi
            chunkList.add(startTagChunk);
        } catch (IOException e) {
            e.printStackTrace();
            log("parse Start NameSpace Chunk error!");
        }
    }
```

以 010 editor 解析到的第一个

`Start Tag Chunk`

为例，看一下解析的结果：

```
parse Start Tag Chunk
chunk type: 0x100102
chunk size: 116
line number: 2
namespace uri: null
name: manifest
attributeCount: 4
class attribute: 0
Attribute[0]
   namespace uri: http://schemas.android.com/apk/res/android
   name: versionCode
   valueStr: null
   type: 16
   data: 980
Attribute[1]
   namespace uri: http://schemas.android.com/apk/res/android
   name: versionName
   valueStr: 7.9.5
   type: 3
   data: 7.9.5
Attribute[2]
   namespace uri: http://schemas.android.com/apk/res/android
   name: installLocation
   valueStr: null
   type: 16
   data: 0
Attribute[3]
   namespace uri: null
   name: package
   valueStr: com.tencent.mobileqq
   type: 3
```

```
type: 3
data: com.tencent.mobileqq
```

根据解析结果，可以轻松的写出这个标签的内容：

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
        android:versionCode="980"
        android:versionName="7.9.5"
        android:installLocation="0"
        package="com.tencent.mobileqq">
```

依次解析后面的 chunk，就可以拼接出整个

```
AndroidManifest.xml
```

文件了。

End Tag Chunk

```
End Tag Chunk
```

一共有 6 项数据，也就是

```
Start Tag Chunk
```

的前 6 项。

该项用来标识一个标签的结束。在生成 xml 的过程中，遇到此标签，就可以将当前解析出的标签结束掉。就像上面的

```
manifest
```

标签，就可以给它加上结束标签了。

Text Chunk

```
Text Chunk
```

在解析过程中暂时还没遇到过，这里就不细说了。

到此为止，

`AndroidManifest.xml`

的解析就全部结束了，但是还没有生成一份可以直接阅读的清单文件。具体的生成代码可以看我的解析工程 Parser。包括之前的 Class 文件解析，以及后续的其他解析代码都会放在这个目录中。

微信扫一扫识别小程序

识别小程序

▼更多现金红包，请长按二维码▼

目前100000+人已关注加入我们