

Dex格式解析与增量更新

dex文件是Android系统的可执行文件，包含应用程序的全部操作指令以及运行时数据。

当java程序编译成class后，还需要使用dx工具将所有的class文件整合到一个dex文件，目的是其中各个类能够共享数据，在一定程度上降低了冗余，同时也是文件结构更加紧凑，实验表明，dex文件是传统jar文件大小的50%左右。

文件布局

dex 文件可以分为3个模块，头文件、索引区、数据区。头文件概况的描述了整个 dex 文件的分布，包括每一个索引区的大小跟偏移。索引区表示每个数据的标识，索引区主要是指向数据区的偏移。

1

我们可以使用16进制查看工具打开一个dex来同步分析。(建议使用010Editor)。

1598320740980

010Editor 中除了数据区(data)没有显示出来，其他区段都有显示，另外 link_data 在此处被定为 map_list

大小端

一般的，文件一般使用小端字节序存储（Dex文件也不例外），网络传输一般使用大端字节序。

- 大端模式（Big-endian），是指数据的高字节保存在内存的低地址中，而数据的低字节保存在内存的高地址中。
- 小端模式（Little-endian），是指数据的高字节保存在内存的高地址中，而数据的低字节保存在内存的低地址中。

假如有一个4字节的数据为 0x12 34 56 78（十进制：305419896，0x12 为高字节，0x78 为低字节），若将其存放于地址 0x1000 中：

内存地址	0x1000（低地址）	0x1001	0x1002	0x1003（高地址）
大端模式	0x12（高字节）	0x34	0x56	0x78（低字节）
小端模式	0x78（低字节）	0x56	0x34	0x12（高字节）

Header

整个dex文件以16进制打开，前112个字节为头文件数据。Header描述了 dex 文件信息，和其他各个区的索引。

1598320920528

此处数据，最开始为 `dex_magic` 魔数，数据为：

字段	字节数	说明
dex	3	文件格式：dex
newLine	1	换行："\n"
ver	3	版本：035
zero	1	无意义，00

uint为4字节数据

- checksum: 文件校验码，使用 alder32 算法校验文件除去 magic、checksum 外余下的所有文件区域，用于检查文件错误。
- signature: 使用 SHA-1 算法 hash 除去 magic、checksum 和 signature 外余下的所有文件区域，用于唯一识别本文件。
- file_size: dex 文件大小
- header_size: header 区域的大小，固定为 0x70
- endian_tag: 大小端标签，dex 文件格式为小端，固定值为 0x12345678
- map_off: map_item 的偏移地址，该 item 属于 data 区里的内容，值要大于等于 data_off 的大小，处于 dex 文件的末端。

其他 `xx_off`，`xx_size` 成对出现，为对于数据的偏移与数据个数。对应Header数据解析代码为：

```
//dexFile: new File("dex文件地址")
byte[] rawData = FileUtil.readFile(dexFile);
this.data = ByteBuffer.wrap(rawData); //使用ByteBuffer装载数据
this.data.order(ByteOrder.LITTLE_ENDIAN); //设置为小端模式

//读取header
header = Header.readFrom(data);
```

```
package com.enjoy.diff.android;

import com.enjoy.diff.util.BufferUtil;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;

public class Header {

    //固定112个字节
    public static final int SIZE_OF_HEADER = 112;

    public int stringIdsSize;
    public int stringidsOff;
    public int typeIdsSize;
    public final int typeIdsOff;
```

```

public final int protoIdsSize;
public final int protoIdsOff;
public final int fieldIdsSize;
public final int fieldIdsOff;
public final int methodIdsSize;
public final int methodIdsOff;
public final int classDefsSize;
public final int classDefsOff;
public final int dataSize;
public final int dataOff;
public int mapOff;
public int fileSize;

public Header(ByteBuffer data) {
    byte[] magic = BufferUtil.readBytes(data, 8); //魔数: 文件格式、版本
    int checksum = data.getInt(); //校验码
    byte[] signature = BufferUtil.readBytes(data, 20); //签名
    fileSize = data.getInt();
    int headerSize = data.getInt(); //一定是112
    int endianTag = data.getInt(); //一定是 0x12345678

    int linkSize = data.getInt();
    int linkOff = data.getInt();

    //mapList部分偏移
    mapOff = data.getInt();
    stringIdsSize = data.getInt();
    stringIdsOff = data.getInt();
    typeIdsSize = data.getInt();
    typeIdsOff = data.getInt();
    protoIdsSize = data.getInt();
    protoIdsOff = data.getInt();
    fieldIdsSize = data.getInt();
    fieldIdsOff = data.getInt();
    methodIdsSize = data.getInt();
    methodIdsOff = data.getInt();
    classDefsSize = data.getInt();
    classDefsOff = data.getInt();
    dataSize = data.getInt();
    dataOff = data.getInt();
}

public static Header readFrom(ByteBuffer in) {
    //拷贝一份ByteBuffer
    ByteBuffer sectionData = in.duplicate();
    sectionData.order(ByteOrder.LITTLE_ENDIAN); //小端序
    sectionData.position(0);
    //可操作数据长度为 112字节
    sectionData.limit(SIZE_OF_HEADER);
    return new Header(sectionData);
}
}

```

在解析完 `Header` 之后，就能够获得接下来数据的偏移与长度，按照对应的值定位位置解析。

StringIds

`string_ids` 区段描述了 dex 文件中所有的字符串。记录的数据只有一个偏移量，偏移量指向了 数据区Data中 的一个字符串：

`stringids`

根据 `Header` 解析结果得知，`StringIds`中有15个数据。

```
//dex对应的ByteBuffer、stringids个数与stringids数据区域偏移
string_ids = StringIdItem.readFrom(data, header.stringIdsSize, header.stringIdsOff);

public static Map<Integer, StringIdItem> readFrom(ByteBuffer in, int size, int off) throws
UTFDataFormatException {
    ByteBuffer sectionData = in.duplicate();
    sectionData.order(ByteOrder.LITTLE_ENDIAN);
    sectionData.position(off); //偏移此处为stringids

    Map<Integer, StringIdItem> map = new HashMap<>();
    for (int i = 0; i < size; i++) {
        //字符串数据内容偏移
        int string_data_off = sectionData.getInt();
        int position = sectionData.position();
        //定位到数据内容对应偏移
        sectionData.position(string_data_off);
        //解析字符串数据：下面说明
        int utf16_size = BufferUtil.readUnsignedLeb128(sectionData);
        String data = BufferUtil.readUtf8(sectionData, utf16_size);
        sectionData.position(position);

        StringIdItem stringItem = new StringIdItem(string_data_off, utf16_size, data);
        map.put(i, stringItem);
    }
    return map;
}
```

后续数据同样的方式进行解析。体力活~~~

后续数据格式参考：

<http://gnaixx.cc/2016/11/26/20161126dex-file/>

<https://source.android.google.cn/devices/tech/dalvik/dex-format>

DexDiff

dexDiff是微信结合Dex文件格式设计的一个专门针对Dex的差分算法。根据Dex的文件格式，对两个Dex中每一项数据进行差分记录。整个实现过程其实很繁琐，我们以字符串StringIds区域的差分举例

StringIds差分计算

Dex文件中的段落数据都是经过排序的。如存在"a","b"与"c"三个字符串。那么在StringIds中Item顺序也为abc。
对照两个dex文件数据：oldDex与newDex。

```
newIndex = 0; oldIndex =0; newCount = 3; oldCount=3
```

	0	1	2
old	<u>a</u>	b	c
new	<u>b</u>	c	e

old dex中 a 与new dex 中的 b 比较。

```
"a".compareTo("b") < 0 : old dex中的 a 标记为: del, oldIndex++继续比较。
```

```
newIndex = 0; oldIndex =1; newCount = 3; oldCount=3
```

	0	1	2
old	a	<u>b</u>	c
new	<u>b</u>	c	e

old dex中 b 与new dex中的 b 比较。

```
"b".compareTo("b") == 0 ,不处理。oldIndex++, new Index++
```

```
newIndex = 1; oldIndex =2; newCount = 3; oldCount=3
```

	0	1	2
old	a	b	<u>c</u>
new	b	<u>c</u>	e

old dex中 c 与new dex中的 c 比较。

```
"c".compareTo("c") == 0 ,不处理。oldIndex++, newIndex++
```

```
newIndex = 1; oldIndex =3; newCount = 3; oldCount=3
```

	0	1	2
old	a	b	c
new	b	c	<u>e</u>

```
oldIndex = 3 = oldCount, newIndex = 2 < oldCount
```

因此new dex剩余的Item全部记为：**add**

```
int oldIndex = 0;
int newIndex = 0;
int oldStrCount = oldDex.string_ids.size(); //old dex中解析的字符串集合
int newStrCount = newDex.string_ids.size(); //new dex中解析的字符串集合
//记录操作集合
List<PatchOperation> patchOperationList = new ArrayList<>();
while (oldIndex < oldStrCount || newIndex < newStrCount) {
    if (oldIndex >= oldStrCount) { //old下标记超过old数据元素个数了
        //表示new还有, 则全是新的
        patchOperationList.add(new PatchOperation(PatchOperation.OP_ADD, newIndex,
newDex.string_ids.get(newIndex)));
        newIndex++;
    } else if (newIndex >= newStrCount) {
        // old需要remove
        patchOperationList.add(new PatchOperation(PatchOperation.OP_DEL, oldIndex,
oldDex.string_ids.get(oldIndex)));
        oldIndex++;
    } else {
        StringIdItem newItem = newDex.string_ids.get(newIndex);
        StringIdItem oldItem = oldDex.string_ids.get(oldIndex);
        //比较StringIdItem对象内部实现位: 比较字符串数据
        int cmpRes = oldItem.compareTo(newItem);
        if (cmpRes < 0) {
            // old: a new: b 此时应该是删除old的a, new的b继续比较old后续的字符串
            patchOperationList.add(new PatchOperation(PatchOperation.OP_DEL, oldIndex,
oldDex.string_ids.get(oldIndex)));
            oldIndex++;
        } else if (cmpRes > 0) {
            // old: b new: a 此时应该是增加new的a, old的b继续对比new后续的字符串
            patchOperationList.add(new PatchOperation(PatchOperation.OP_ADD, newIndex,
newDex.string_ids.get(newIndex)));
            newIndex++;
        } else {
            oldIndex++;
            newIndex++;
        }
    }
}
```

增量更新

自从 Android 4.1 开始，Google Play 引入了应用程序的增量更新功能，App使用该升级方式，可节省约2/3的流量。现在国内主流的应用市场也都支持应用的增量更新。

增量更新的关键在于增量一词。平时我们的开发过程，往往都是今天在昨天的基础上修改一些代码，app的更新也是类似的：往往都是在旧版本的app上进行修改。这样看来，增量更新就是原有app的基础上只更新发生变化的地方，其余保持原样。

与之前每次更新都要下载完整apk包的做法相比，这样做的好处显而易见：每次变化的地方总是比较少，因此更新包的体积就会小很多。比某APK的体积在60m左右，如果不采用增量更新，用户每次更新都需要下载大约60m左右的安装包，而采用增量更新这种方案之后每次只需要下载2m左右的更新包即可，相比原来做法大大减少了用户下载等待的时间和流量，同时也可以因为更新变得更简单也能够缩短产品版本覆盖周期。

使用BSDiff

<http://www.daemonology.net/bsdiff/>

Binary diff/patch utility

bsdiff和bspatch是构建和应用补丁到二进制文件的工具。

bsdiff and bspatch are tools for building and applying patches to binary files. By using suffix sorting (see [qsufsort](#)) and taking advantage of how executable files change, bsdiff routinely produces binary patches: [Xdelta](#), and 15% smaller than those produced by [RTPatch](#) (a \$2750/seat commercial patch tool).

These programs were originally named `bdiff` and `bpatch`, but the large number of other programs using `bs` sure if the "bs" in refers to "binary software" (because bsdiff produces exceptionally small patches for `c` (which is the key to how well it performs). Feel free to offer other suggestions.

bsdiff and bspatch use bzip2; by default they assume it is in `/usr/bin`.

bsdiff is quite memory-hungry. It requires $\max(17*n, 9*n+m)+O(1)$ bytes of memory, where n is the size of file. bspatch requires $n+m+O(1)$ bytes.

bsdiff runs in $O((n+m) \log n)$ time; on a 200MHz Pentium Pro, building a binary patch for a 4MB file takes $O(n+m)$ time; on the same machine, applying that patch takes about two seconds.

Providing that `offset` is defined properly, bsdiff and bspatch support files of up to $2^{61}-1 = 2\text{Ei}-1$ bytes.


Version 4.3 is available [here](#) with MD5 hash `e6d812394f0e0ecc8d5df255aa1db22a`. Version 4.2 is available as `dev/bsdiff`, and in `gentoo` as `dev-util/bsdiff`. It has also been made into a [Python extension module](#).


The algorithm used by BSDiff 4 is described in my (unpublished) paper [Naive differences of executable](#).


Colin Percival, *Naive differences of executable code*, <http://www.daemonology.net/bsdiff/>, 2003.

A far more sophisticated algorithm, which typically provides roughly 20% smaller patches, is described in

此处下载源码，包含两个程序：**bsdiff**（比较两个文件的二进制数据，生成差分包）与**bspatch**（合并旧的文件与差分包，生成新的文件）。

 `bsdiff.c`

 `bspatch.c`

 `Makefile`

Linux/Mac

已经安装bzip2

打开Makefile

```
FLAGS      +=      -O3 -lbz2

PREFIX      ?=      /usr/local
INSTALL_PROGRAM ?=      ${INSTALL} -c -s -m 555
INSTALL_MAN   ?=      ${INSTALL} -c -m 444

all:         bsdiff bspatch
bsdiff:      bsdiff.c
bspatch:     bspatch.c

install:
    ${INSTALL_PROGRAM} bsdiff bspatch ${PREFIX}/bin
.ifndef WITHOUT_MAN
    ${INSTALL_MAN} bsdiff.1 bspatch.1 ${PREFIX}/man/man1
.endif
~
~
```

内容也不多。但是这个Makefile有一个问题，Makefile中Target指令必须以tab开头，所以我们需要修改为

```
install:
    ${INSTALL_PROGRAM} bsdiff bspatch ${PREFIX}/bin
    .ifndef WITHOUT_MAN
    ${INSTALL_MAN} bsdiff.1 bspatch.1 ${PREFIX}/man/man1
    endif
~
```

现在保存退出。

我们直接进入目录执行make

```
MacBook-Air:bsdiff-4.3 xiang$ make
cc -O3 -lbz2    bsdiff.c    -o bsdiff
cc -O3 -lbz2    bspatch.c   -o bspatch
bspatch.c:39:21: error: unknown type name 'u_char'; did you mean 'char'?
static off_t offtin(u_char *buf)
                    ^~~~~~
                    char
bspatch.c:65:8: error: expected ';' after expression
```

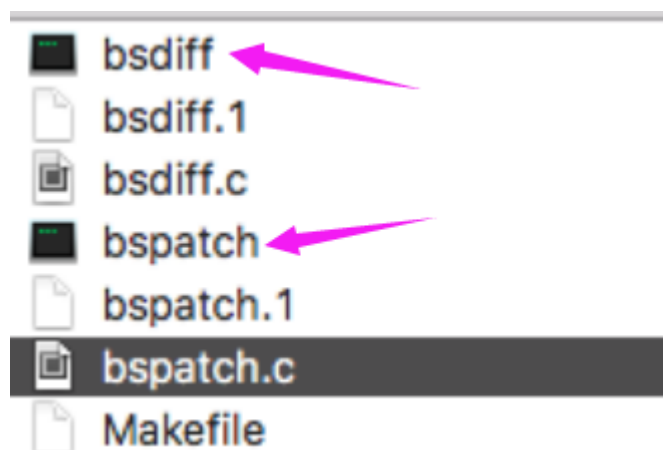
报错了，不认识u_char这个类型。打开bspatch.c


```

27  #if 0
28  __FBSDID("$FreeBSD: src/usr.bin/bsdiff/bspatch/bs
29  #endif
30
31  typedef unsigned char u_char;  增加
32
33  #include <bzlib.h>
34  #include <stdlib.h>
35  #include <stdio.h>
36  #include <string.h>
37  #include <err.h>

```

然后在执行make，将得到 bsdiff与bspatch工具。



Windows

Windows的同学可以到附件中获得可执行程序。

测试

Android中使用需要NDK开发

我们先来使用bsdiff工具生成差分补丁包:

```

[MacBook-Air:bsdiff-4.3 xiang$ ./bsdiff
bsdiff: usage: ./bsdiff oldfile newfile patchfile

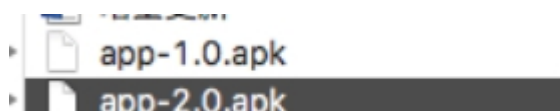
```

可以看到使用方式是

bsdiff 老文件 新文件 输出的补丁

创建一个空的Android Native工程生成apk。 --- 1.0版本

然后我们修改一些代码，再增加一些图片，生成新的apk --- 2.0版本



运行：（apk文件在bsdiff上一层）

```
./bsdiff ../app-1.0.apk ../app-2.0.apk ../patch.apk
```

然后会在我们指定的目录生成

app-1.0.apk	今天 下午3:47	2 MB	T
app-2.0.apk	今天 下午3:54	3.8 MB	T
BSDiff	今天 下午3:56	--	3
bsdiff-4.3	今天 下午3:26	--	3
bsdiff-4.3.tar.gz	今天 下午2:13	6 KB	9
patch.apk	今天 下午3:55	2.3 MB	T

这里说明下我们看到补丁包2.3+oldapp 2 = 4.3比我们的app2.0要大。因为bsdiff的补丁大小并不是简单的加减，还会存在一些上下文环境等额外的信息。

但是我们从上面能够看出，如果在安装了1.0的情况下，我们升级，能够节省1.5M的内存。

接下来我们来验证下这个补丁包是不是有效

bspatch命令是：

```
[MacBook-Air:bsdiff-4.3 xiang$ ./bspatch
bspatch: usage: ./bspatch oldfile newfile patchfile
```

```
[MacBook-Air:bsdiff-4.3 xiang$ ./bspatch ../app-1.0.apk ../app-new.apk ../patch.apk
MacBook-Air:bsdiff-4.3 xiang$
```

我们合并后的文件生成了

app-2.0.apk	今天 下午3:54	3.8 MB	1
app-new.apk	今天 下午4:00	3.8 MB	1
BSDiff	今天 下午3:56	--	?

目前来看至少大小一样。

我们可以测试能否正常安装。

```
[MacBook-Air:bsdiff-4.3 xiang$ adb install -r ../app-new.apk
../app-new.apk: 1 file pushed. 3.0 MB/s (3759491 bytes in 1.209s)
pkg: /data/local/tmp/app-new.apk
Success
MacBook-Air:bsdiff-4.3 xiang$
```