Table of Contents

简介

愿景

abtool旨在提供基于Unity资源封装格式 `AssetBundle` 的C++开发框架以及预置工具集合，方便针对资源做**任意**的检测、编辑以及资源问题定位。

开发背景

笔者2020年初加入使命召唤手游项目，这是一款偏向内容运营的高品质手机游戏，有非常多的ab资源，从笔者加入项目时的1G左右增加到现在的5G左右，未来可以预期持续的增长。伴随着资源量的增加，资源相关的崩溃、显示问题越来越多，定位解决这些问题是一个常态化的工作。

笔者的工作内容主要是负责版本以及资源发布，在abtool出现之前定位这些问题非常困难，特别是资源引起的游戏崩溃问题，困难主要表现为：

1. 需要稳定的重现方法，通常需要花很多时间去摸索
2. 需要增加运行时调试信息来辅助判断，甚至需要真机调试，通常需要重新构建

处理这类问题遇到最大的麻烦是：**即使解决了崩溃，你仍然无法确定是否还有其他类似的资源崩溃问题**。我们有7000个左右的ab文件，排查所有的资源问题犹如大海捞针，每次发版本都是战战兢兢、如履薄冰，并且经常通宵攻坚，但也不总是有效，这种情况下只能延迟版本发布。

鉴于笔者丰富的工具开发经验，经历几次通宵后，笔者决定通过工具化寻求突破，最终的开发进度以及使用效果也是十分喜人，截止文档撰写日起已有20多个内置命令，它们均是在解决资源问题过程中逐渐增加和完善的，具有很强的实用性。当然，通过后续的章节了解熟悉后，你也可以轻易开发出专属于你的工具命令。

文档更新

如果你需要访问最新的文档内容，建议你查阅在线文档版本[1]，或者手动下载当前文档的最新版本[2]。

由于文档撰写比较匆忙，难免有所谬误，请多包涵。同时，也欢迎大家Fork笔者的文档仓库[3]并提交相应的PR，让我们一起来完善它，感激不尽！

> [1]. https://larryhou.github.io/abtool-gitbook/ ↩
>
> [2]. https://larryhou.github.io/abtool-gitbook/book.pdf ↩
>
> [3]. https://github.com/larryhou/abtool-gitbook/ ↩

第一章 编译安装

由于笔者日常工作环境中很少使用其他操作系统，暂时abtool只支持针对macOS系统平台的编译运行，感兴趣的朋友可以自行适配其他系统平台，涉及平台差异的内容主要是以下几个外部链接库：

- libreadline.tbd
- libfbxsdk.a
  - Foundation.framework
  - libiconv.tbd
  - libxml2.tbd
  - libz.tbd

abtool源码基于C++14标准库实现，理论上处理好这些编译依赖问题即可完成适配。

首次编译

如果你不是第一次使用abtool，也就是说你手上已经有了一份abtool工具，那么可以跳过该步骤，直接进行下一步操作。

1. Xcode编译

   打开Xcode，使用组合键 `⌘+B` 即可进行源码编译，之后编译可以在终端环境或者shell脚本里面随意使用，这是生成abtool工具的最简单的方式，需要注意的是请确保目标目录 `/usr/local/bin` 已被预先创建。

2. CMake编译

   执行如下脚本，即可在 `build/bin` 目录得到abtool命令行工具。

```
# 当前cd目录为工程根目录
mkdir build
cd build
cmake ..
cmake --build .
```

生成TypeTree数据

TypeTree记录了资源对象数据的序列化信息，收集TypeTree的目的是为了把Unity的类型信息集成到abtool工具里面，只有这样abtool才有可能实现它的功能。

为了让大家快速体验整个工具编译过程，笔者在工程doc目录准备了 `QuickStart.unitypackage` 资源包，现在你只需要新建一个Unity工程，然后导入所有资源，通过Unity菜单 `abtool/Build Asset Bundles` 即可快速生成包含了TypeTree数据的ab文件，该资源包包含了能够让abtool源码正常编译的最小集合，具体来说是，资源里面包含了以下编译必须的资源对象类型：

- GameObject
- RectTransform
- Transform
- TextAsset
- Texture2D
- Cubemap

- Material
- Shader
- SpriteRenderer
- SkinnedMeshRenderer
- MeshRenderer
- ParticleSystemRenderer
- LineRenderer
- TrailRenderer
- MeshFilter
- Animator
- Mesh

如果你现有的项目资源已经覆盖了以上资源类型，那么可以放心使用abtool收集相应的TypeTree数据。然而QuickStart资源包只是覆盖了最小集合的资源类型，如果需要最大限度发挥abtool的功效，笔者强烈建议你扫描尽可能多的ab文件，从而可以收集到尽可能多的Unity类型数据，这样会让你定位资源问题更加得心应手，相信你在后续的日常使用中会深刻明白这一点。

```
# doc/resources目录存储了QuickStart资源编译的iOS/Android双平台的ab文件
cd doc/resources
# 由于QuickStart生成的ab文件后缀为ab，所以可以通过'*.ab'进行文件匹配
# 请根据实际项目的ab文件后缀做适当修改
find . -iname '*.ab' | xargs abtool savetree -a types.tte
```

上述脚本通过 `find` 命令查找所有的ab文件，并把这些文件通过 `xargs` 透传给abtool工具去处理，最终会在当前目录生成 `types.tte` 文件（当然也可以通过 `savetree` 的 `-a` 参数指定其他保存目录），这就是我们需要的Unity类型数据。

生成对象序列化代码

`types.tte` 是个二进制文件，把它转换成C++代码才能最终为abtool所用，通过下面这行命令可以轻松完成这个任务，整个过程就好比使用 `protoc` 编译 `*.proto` 文件一样。

```
# 当前cd目录为工程根目录
abtool gtt -a doc/resources/types.tte -o abtool/assetbundles/unity
```

由于上面的脚本是在工程根目录执行，并且代码的输出目录为 `abtool/assetbundles/unity`，所以当脚本执行完成后工程的代码就得到了更新。

最终编译

通过上一步骤我们修改了Unity资源对象的序列化代码，所以还需要再次编译，这样我们就最终得到了功能完备的abtool，通过后续的章节可以逐渐窥探它强大的威力。

> 什么情况下需要重新编译abtool?

1. 升级了Unity版本
2. 修改了Unity源码里面涉及资源对象的序列化的代码
3. 修改了 `AssetBundleArchive` 容器存储结构
4. 修改了 `SerializedFile` 存储结构
5. 如果需要abtool正常处理所有 `MonoBehaviour` 组件数据，那么你需要定期编译abtool，不过我们大部分情况下并不关心这部分数据。

运行测试

当你不知道用abtool做什么的时候，建议你跑一下 `dump` 命令，如下

```
abtool dump doc/resources/android/quickstart.ab
```

第二章 应用案例

对于大多数来说，大家可能更希望abtool能有一些现实的应用场景，这样拿到工具后就可以着手做一些资源侧的检测优化工作，在本章节会列举几个常见的案例，通过案例来了解工具的使用。

资源逆向

资源逆向既是通常意义的资源反编译，也就是从ab文件里面提取出来方便浏览的资源。鉴于abtool集成了项目所有资源类型的序列化信息，理论上abtool可以反编译任意资源，但是实现情况是反编译所有资源有代价，并且也不是所有资源都是我们关心的，所以笔者暂时只实现了有限几个但高频使用的资源类型的反编译，比如：贴图、模型、Shader、二进制文件等。从ab文件反编译资源并非abtool的开发初衷，但是abtool的实现原理注定它可以轻松支持资源逆向目的。

为了增加abtool资源逆向功能的一般性，笔者选择在本案例中使用第三方线上运营游戏来做演示，大家可以依照步骤得到相同的结果。

> 声明：本案例使用的方法以及由该方法得到的资源仅用于学习交流，请勿用于其他非法目的，否则后果自负。

下载安装包

我们可以通过Google搜索 `战歌竞技场 apk` ，然后下载相应的apk安装包。笔者使用的版本是 `1.5.151` ，点击链接[1]可直接进行下载，但是鉴于cdn链接的时效性，该文档并不保证该下载链接总是有效可用，如果下载失败请自行从Google搜索结果里面寻找其他链接进行下载。

解压ab资源

首先，用unzip命令行查看apk资源列表

```
unzip -l 10040714_com.tencent.hjzqgame_a960942_1.5.151_j2e715.apk
```

从日志里面我们发现 `assets/AssetBundles` 目录存储了ab资源，现在我们可以继续用unzip提取ab资源

```
unzip -o 10040714_com.tencent.hjzqgame_a960942_1.5.151_j2e715.apk 'assets/AssetBundles/*'
cd assets
```

这样我们就得到了apk里面所有的ab资源，在后续资源资源逆向案例中如无特殊说明，均把解压出来的assets目录作为工具的工作空间，并默认使用AssetBundles目录里面的ab资源做演示。

编译

我们在前面章节已经学习了工具编译过程，由于该案例用到的ab资源属于某个特定Unity版本，所以需要依据编译流程手机资源TypeTree并重新编译abtool，否则你将无法正常通过abtool进行后续的资源逆向操作。

```
find . -iname '*.god' | xargs abtool savetree
```

1. https://dlied4.myapp.com/myapp/1109006800/cos.release-75620/10040714_com.tencent.hjzqgame_a960942_1.5.151_j2e715.apk ↩

贴图

选择ab文件

首先我们需要找到一个包含贴图资源的ab文件，如果你不确定是哪个ab满足要求，那么可以使用 `list` 命令收集所有进包的资源路径，然后反过来从贴图资源的路径查找相应的ab文件。

```
find AssetBundles -iname '*.god' | xargs abtool list -r
```

从结果里面我们选择 `artresource_captainpbr_captain_202.god` 作为演示资源。

```
[095/123] assets/artresource/captainpbr/captain_202/textures/captain_202103_suit_m.tga Texture2D i:13067992522968025633
[096/123] assets/artresource/captainpbr/captain_202/textures/captain_2021_body_n.tga Texture2D i:3883958129370872108
[097/123] assets/artresource/captainpbr/captain_202/textures/captain_2021_stuff_n.tga Texture2D i:11914906896687225448
[098/123] assets/artresource/captainpbr/captain_202/textures/captain_2021_suit_n.tga Texture2D i:17698705126682618081
[099/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_body_a.tga Texture2D i:2758807937572619551
[100/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_body_b.tga Texture2D i:10388116232065757511
[101/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_body_d.tga Texture2D i:10679769261937420922
[102/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_body_m.tga Texture2D i:7592280451348180021
[103/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_suit_a.tga Texture2D i:8152769956827659214
[104/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_suit_b.tga Texture2D i:14756172379838531154
[105/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_suit_d.tga Texture2D i:18218461557286871300
[106/123] assets/artresource/captainpbr/captain_202/textures/captain_202201_suit_m.tga Texture2D i:13791270348976128462
[107/123] assets/artresource/captainpbr/captain_202/textures/captain_202202_body_b.tga Texture2D i:9804010988900091748
[108/123] assets/artresource/captainpbr/captain_202/textures/captain_202202_body_d.tga Texture2D i:10986088950313902645
[109/123] assets/artresource/captainpbr/captain_202/textures/captain_202202_body_m.tga Texture2D i:10320767474896706012
[110/123] assets/artresource/captainpbr/captain_202/textures/captain_202202_suit_b.tga Texture2D i:2592695568612982683
[111/123] assets/artresource/captainpbr/captain_202/textures/captain_202202_suit_d.tga Texture2D i:3746720729245558176
[112/123] assets/artresource/captainpbr/captain_202/textures/captain_202202_suit_m.tga Texture2D i:17681954939584535757
[113/123] assets/artresource/captainpbr/captain_202/textures/captain_2022_body_n.tga Texture2D i:11607926898818107768
[114/123] assets/artresource/captainpbr/captain_202/textures/captain_2022_suit_n.tga Texture2D i:3533547237365748242
[115/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_body_a.tga Texture2D i:5710822887162919953
[116/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_body_b.tga Texture2D i:11415350406488400628
[117/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_body_d.tga Texture2D i:16515969976440694086
[118/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_body_m.tga Texture2D i:860939899447636964
[119/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_body_n.tga Texture2D i:9829224366313559506
[120/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_suit_b.tga Texture2D i:18878662321467012256
[121/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_suit_d.tga Texture2D i:8568020777195722661
[122/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_suit_m.tga Texture2D i:17121284964933108453
[123/123] assets/artresource/captainpbr/captain_202/textures/captain_202301_suit_n.tga Texture2D i:18012581096705529419
```

提取贴图资源

通过abtool的 `savetex` 命令可以一次性保存ab文件里面所有的贴图资源，默认输出到当前目录的 `__textures` 目录，也可以添加 `--output` 参数指定其他存放目录。

```
abtool savetex AssetBundles/Android/artresource_captainpbr_captain_202.god
```

```
LARRYHOU-MC8:demo larryhou$ abtool-cr savetex AssetBundles/artresource_captainpbr_captain_202.god
[0] AssetBundles/artresource_captainpbr_captain_202.god
__textures/Captain_202202_body_b.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202301_body_n.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202102_body_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202202_body_m.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202201_body_b.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202201_body_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202103_body_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202102_suit_b.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202202_body_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202301_body_b.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_2022_body_n.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_2021_stuff_n.64x64.etc_rgb4.tex =2,048 2.00K
__textures/Captain_202103_body_m.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202101_body_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202103_suit_m.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202101_suit_a.64x64.etc_rgb4.tex =2,048 2.00K
__textures/Captain_202101_suit_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202101_suit_b.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202201_suit_m.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202101_stuff_m.64x64.etc_rgb4.tex =2,048 2.00K
__textures/Captain_202201_suit_b.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202101_stuff_b.64x64.etc_rgb4.tex =2,048 2.00K
__textures/Captain_202101_suit_m.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202102_body_m.512x512.etc_rgb4.tex =131,072 128.00K
__textures/Captain_202301_body_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202103_suit_d.512x512.etc2_rgba8.tex =262,144 256.00K
__textures/Captain_202301_suit_m.256x256.etc_rgb4.tex =32,768 32.00K
__textures/Captain_202101_body_a.64x64.etc_rgb4.tex =2,048 2.00K
```

需要说明的是：`savetex` 保存的贴图有着固定命名规范，其格式为[filename].[宽]x[高].[贴图格式].tex，除了filename，其余文件名内容是不能修改的，否则在接下来的贴图转码操作会失败。

贴图格式转换

从上一步骤得到的贴图都是 `*.tex` 格式的文件，并非用普通图片浏览工具可以直接查看文件格式，它们被做了特殊编码编码以便GPU渲染时可以被正常读取，所以还需要做贴图转码。在工程根目录放置了python脚本工具 `textool.py` ，它可以批量地把 `*.tex` 文件转换成项目中常见的 `*.tga` 文件。

贴图

```
import re, struct
from tex2img import decompress_astc, decompress_etc, decompress_pvrtc
from PIL import Image

def main():
    import sys
    pattern = re.compile(r'[^/]+\.(\d+x\d+)\.([^.]+)\.tex$')
    for filename in sys.argv[1:]:
        match = pattern.search(filename)
        if not match: continue
        # print('>>> {}'.format(filename))
        fp = open(filename, 'rb')
        texture_size = [int(x) for x in match.group(1).split('x')]
        texture_format = match.group(2)
        mode = 'RGBA'
        if texture_format.startswith('etc_'):
            image = decompress_etc(fp.read(), texture_size[0], texture_size[1], 0)
            mode = 'RGB'
        elif texture_format.startswith('etc2_'):
            image = decompress_etc(fp.read(), texture_size[0], texture_size[1], 3 if texture_format.startswit
            if not texture_format.startswith('etc2_rgba'): mode = 'RGB'
        elif texture_format.startswith('astc_rgb'):
            block_size = [int(x) for x in texture_format.split('_')[-1].split('x')]
            image = decompress_astc(fp.read(), texture_size[0], texture_size[1], block_size[0], block_size[1]
        elif texture_format.startswith('pvrtc'):
            # https://github.com/powervr-graphics/Native_SDK/blob/3f88b0f3735774ab9fb718da0aeadd06acf68d21/fr
            image = decompress_pvrtc(fp.read(), texture_size[0], texture_size[1], 0 if texture_format[-1] ==
        elif texture_format.startswith('rgba32'):
            image = fp.read()
        elif texture_format.startswith('rgb24'):
            image = fp.read()
            mode = 'RGB'
        elif texture_format.startswith('rgb565'):
            width, height = texture_size
            image = bytearray(width * height * 3)
            index = 0
            for r in range(height):
                for c in range(width):
                    v, = struct.unpack('<H', fp.read(2))
                    image[index+0] = (v >> 11 & 0x1F) * 255 // 0x1F # red
                    image[index+1] = (v >>  5 & 0x3F) * 255 // 0x3F # green
                    image[index+2] = (v >>  0 & 0x1F) * 255 // 0x1F # blue
                    index += 3
            image = bytes(image)
            mode = 'RGB'
        elif texture_format.startswith('rgba4444'):
            width, height = texture_size
            image = bytearray(width * height * 4)
            index = 0
            for r in range(height):
                for c in range(width):
                    v, = struct.unpack('<H', fp.read(2))
                    image[index+0] = (v >> 12 & 0xF) * 255 // 0xF # red
                    image[index+1] = (v >>  8 & 0xF) * 255 // 0xF # green
                    image[index+2] = (v >>  4 & 0xF) * 255 // 0xF # blue
                    image[index+3] = (v >>  0 & 0xF) * 255 // 0xF # alpha
                    index += 4
            image = bytes(image)
        elif texture_format.startswith('alpha8'):
            image = fp.read()
            mode = 'L'
        else: continue
        result = Image.frombytes(mode, tuple(texture_size), image, 'raw')
        savename = re.sub(r'(\.[^.]+){3}$', '', filename) + '.tga'
        result.save(savename)
```

```
        print('+ {} => {}'.format(filename, savename))

if __name__ == '__main__':
    main()
```

在使用前建议把textool放到 `/usr/bin/local/` 目录下，这样好处是不用每次都用一个很长的路径来访问这个工具了。

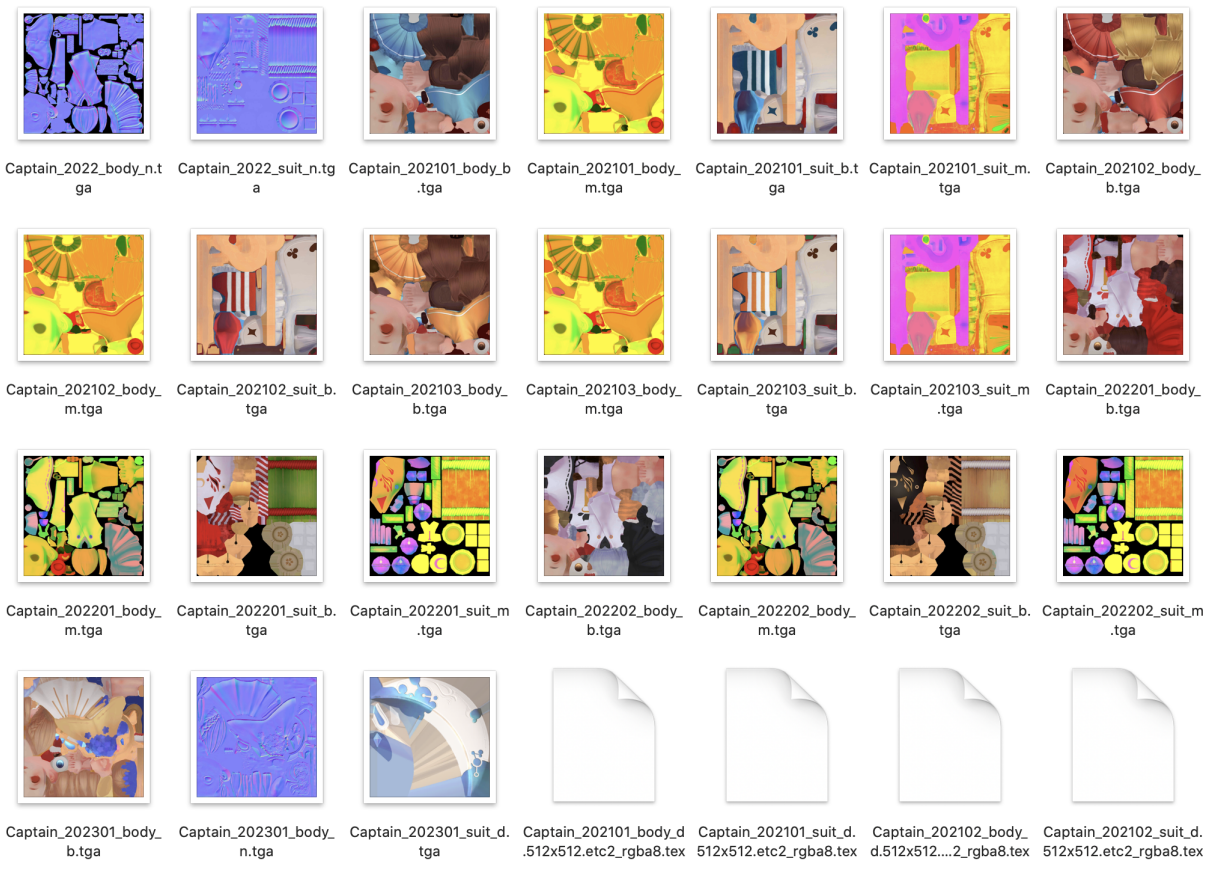
```
cp -fv textool.py /usr/local/bin/textool
```

接下来通过textool转换 `*.tex` 贴图格式，转换后的图片文件存储在源 `*.tex` 文件的同级目录。

```
textool __textures/*.tex
```

```
LARRYHOU-MC8:demo larryhou$ textool __textures/*.tex
+ __textures/Captain_202101_body_a.64x64.etc_rgb4.tex => __textures/Captain_202101_body_a.tga
+ __textures/Captain_202101_body_b.512x512.etc_rgb4.tex => __textures/Captain_202101_body_b.tga
+ __textures/Captain_202101_body_d.512x512.etc2_rgba8.tex => __textures/Captain_202101_body_d.tga
+ __textures/Captain_202101_body_m.512x512.etc_rgb4.tex => __textures/Captain_202101_body_m.tga
+ __textures/Captain_202101_stuff_b.64x64.etc_rgb4.tex => __textures/Captain_202101_stuff_b.tga
+ __textures/Captain_202101_stuff_d.64x64.etc_rgb4.tex => __textures/Captain_202101_stuff_d.tga
+ __textures/Captain_202101_stuff_m.64x64.etc_rgb4.tex => __textures/Captain_202101_stuff_m.tga
+ __textures/Captain_202101_suit_a.64x64.etc_rgb4.tex => __textures/Captain_202101_suit_a.tga
+ __textures/Captain_202101_suit_b.512x512.etc_rgb4.tex => __textures/Captain_202101_suit_b.tga
+ __textures/Captain_202101_suit_d.512x512.etc2_rgba8.tex => __textures/Captain_202101_suit_d.tga
+ __textures/Captain_202101_suit_m.512x512.etc_rgb4.tex => __textures/Captain_202101_suit_m.tga
+ __textures/Captain_202102_body_b.512x512.etc_rgb4.tex => __textures/Captain_202102_body_b.tga
+ __textures/Captain_202102_body_d.512x512.etc2_rgba8.tex => __textures/Captain_202102_body_d.tga
+ __textures/Captain_202102_body_m.512x512.etc_rgb4.tex => __textures/Captain_202102_body_m.tga
+ __textures/Captain_202102_stuff_d.64x64.etc_rgb4.tex => __textures/Captain_202102_stuff_d.tga
```

打开 `__textures` 目录见证奇迹时刻。



textool工具依赖第三方贴图解码库tex2img[1]，该工具封装了BinomialLLC/basis_universal[2]、Ericsson/ETCPACK[3]和powervr-graphics/Native_SDK[4]，感谢老哥K0lb3[5]提供的便利。笔者在此基础上增加了`RGBA32`、`RGBA4444`、`RGB24`、`RGB565`和`Alpha8`贴图格式的转码，经过这么一番整合，应该可以应付绝大部分的贴图转码。

1. https://github.com/K0lb3/tex2img.git ↩
2. https://github.com/BinomialLLC/basis_universal/ ↩
3. https://github.com/Ericsson/ETCPACK ↩
4. https://github.com/powervr-graphics/Native_SDK/tree/master/framework/PVRCore/texture ↩
5. https://github.com/K0lb3 ↩

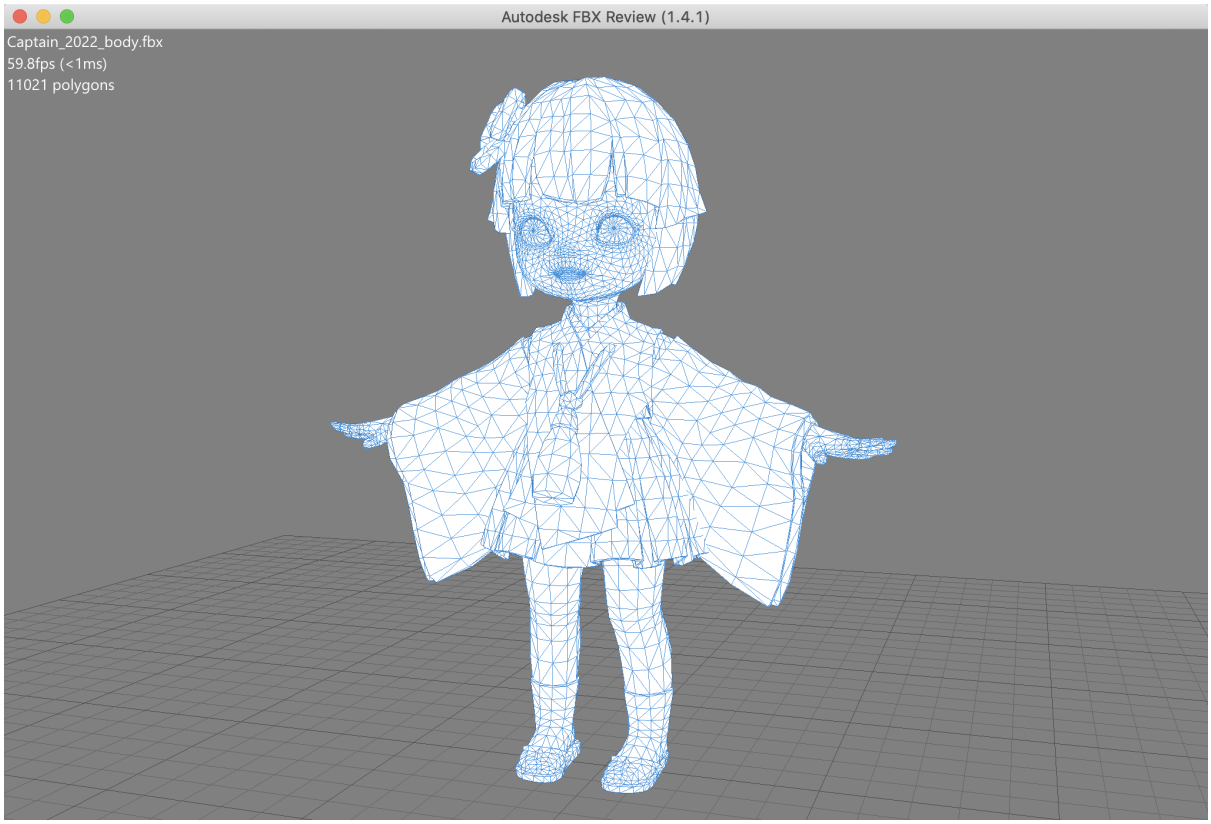
模型

在该案里面我们继续使用 `artresource_captainpbr_captain_202.god` 文件来演示，相比贴图逆向，模型逆向就简单多了，一行命令就可以把ab资源里面的模型导出为 `*.fbx` 文件。

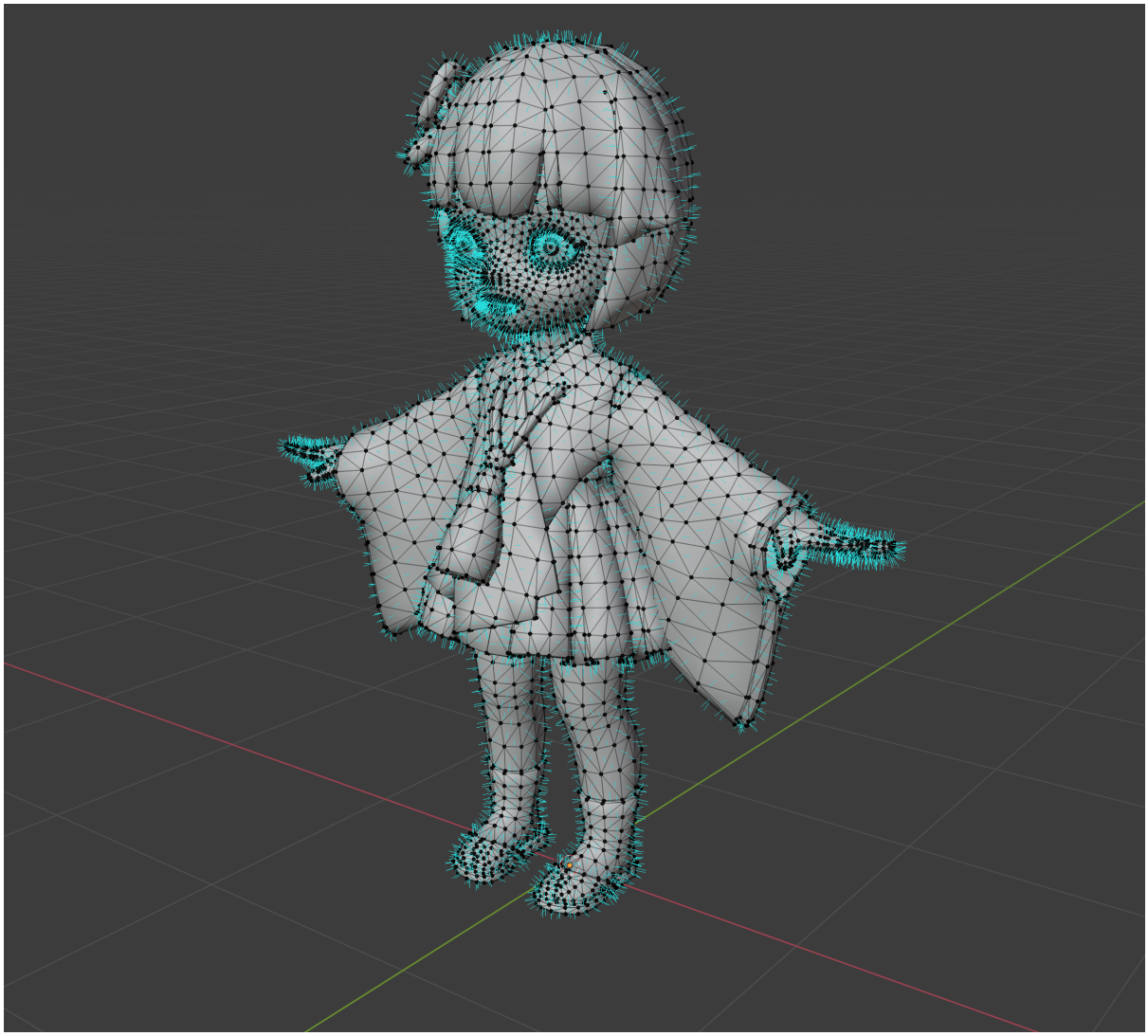
```
$ abtool savefbx AssetBundles/Android/artresource_captainpbr_captain_202.god
```

```
LARRYHOU-MC8:assets larryhou$ abtool-cr savefbx AssetBundles/Android/artresource_captainpbr_captain_202.god
[0] AssetBundles/Android/artresource_captainpbr_captain_202.god
>> __fbx/Captain_2022_body.fbx
>> __fbx/Captain_2023_suit.fbx
>> __fbx/Captain_2022_suit.fbx
>> __fbx/Captain_2021_suit.fbx
>> __fbx/Captain_2021_stuff.fbx
>> __fbx/Captain_2023_body.fbx
>> __fbx/Captain_2021_body.fbx
```

FBX文件可以通过Autodesk FBX Review[1]打开



也可以用其他3D工具打开，比如我们可以在Blender里面查看模型的法线、UVs等信息。



结合贴图逆向得到的贴图，那么我们可以尝试用基础贴图 `Captain_202201_body_b.tga` 简单渲染一下。

呃，跟预期的好像不太一样，这简直像鬼一样……别着急，把贴图上下翻转下就能正常显示了。

[1]. https://www.autodesk.com/products/fbx/fbx-review ↩

资源防护

通过资源逆向案例我们见识了abtool强大的资源反编译能力，但是这个时候我们不应该兴奋，而是应该无比忧虑才对：因为第三方工具可以在没有项目仓库权限的情况下轻易获取游戏资源，这些都是项目团队日夜攻坚、长时间累积优化的结果，如果被用于非法目的，对游戏是非常不利的。问题来了：既然ab资源如此容易破解，那么该如何保护游戏资产？

打包ab资源时关掉TypeTree

我们先来看下ab打包接口

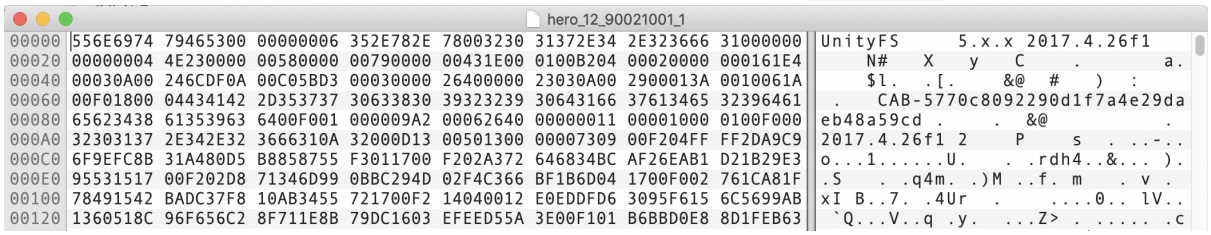
```
public static AssetBundleManifest BuildAssetBundles(
    string outputPath,
    AssetBundleBuild[] builds,
    BuildAssetBundleOptions assetBundleOptions,
    BuildTarget targetPlatform);
```

第三个枚举参数 `BuildAssetBundleOptions` 用来控制ab的打包行为，其中枚举值 `DisableWriteTypeTree` 可以关闭TypeTree。

```
/// <summary>
///   <para>Do not include type information within the AssetBundle.</para>
/// </summary>
DisableWriteTypeTree = 8,
```

由于abtool绝大部分功能都基于TypeTree，那是不是关闭TypeTree资源就安全了？没那么简单！TypeTree是由Unity生成，换句话说，如果拿到相同版本的Unity也是可以轻易获取TypeTree的，在这种情况下，关掉TypeTree的意义仅仅是防止了破解 `MonoBehaviour` ，防护等级是很弱的！换句话说，如果你用了Unity公开发行的版本（标准版），那么你的的游戏资产完全是在裸奔的！

比如王牌战士游戏虽然关闭了资源的TypeTree，但是他们用了标准版的Unity `2017.4.26f1`



笔者从官网下载了一个相同的安装包，导入资源包 `QuickStart.unitypackage` 快速编译出来 `abtool` ，同样可以逆向他们的游戏资产。

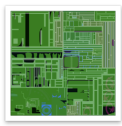
贴图资源


Morgan_Skin_01_dark.tga


Morgan_Skin_01_diffuse_a.tga


Morgan_Skin_01_diffuse_b.tga

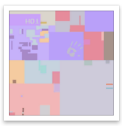

Morgan_Skin_01_diffuse.tga


Morgan_Skin_01_illum.tga


morgan_Skin_02_dark.tga


morgan_Skin_02_diffuse.tga

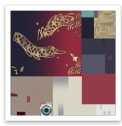

morgan_Skin_02_illum.tga

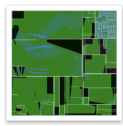

morgan_skin_03_dark.tga


morgan_skin_03_diffuse.tga


morgan_skin_03_illum.tga


morgan_skin_06_dark.tga

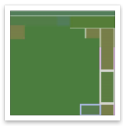

morgan_skin_06_diffuse.tga


morgan_skin_06_illum.tga

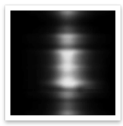

obj_vic_033.tga

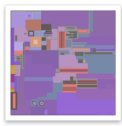

prop_Kong_01_dark.tga


prop_Kong_01_diffuse.tga

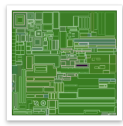

prop_Kong_01_illum.tga


t_guangtiao_801.tga


trl_vic_29.tga

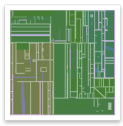

Weapon_01_morgan_dark.tga


Weapon_01_morgan_diffuse.tga


Weapon_01_morgan_illum.tga


Weapon_01_morgan_Single1_dark.tga


Weapon_01_morgan_Single1_diffuse.tga

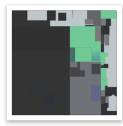

Weapon_01_morgan_Single1_illum.tga

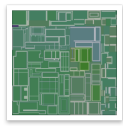

Weapon_01_morgan_skin_01_dark.tga


Weapon_01_Morgan_skin_01_diffuse.tga


Weapon_01_morgan_skin_01_illum.tga


Weapon_01_morgan_Skin_02_dark.tga


Weapon_01_morgan_Skin_02_diffuse.tga


Weapon_01_morgan_Skin_02_illum.tga

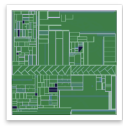

Weapon_01_morgan_skin_03_dark.tga


Weapon_01_morgan_skin_03_diffuse.tga


Weapon_01_morgan_skin_03_illum.tga


Weapon_01_morgan_skin_06_dark.tga


Weapon_01_morgan_skin_06_diffuse.tga


Weapon_01_morgan_skin_06_illum.tga


Weapon_02_morgan_skin_06_dark.tga


Weapon_02_morgan_skin_06_diffuse.tga

模型资源

Name	Size	Kind
Morgan_Skin_01_high.fbx	597 KB	FBX File
Morgan_Skin_01_TPS_LOD0_Model_morgan_Skin_01_LOD0_LOD0.fbx	238 KB	FBX File
Morgan_Skin_01_FPS_Model_softNormalInTangent.fbx	131 KB	FBX File
Weapon_01_FPS_morgan.fbx	127 KB	FBX File
Weapon_01_b_FPS_morgan.fbx	103 KB	FBX File
Weapon_01_b_morgan@high_rig_softNormalInTangent.fbx	99 KB	FBX File
Weapon_02_FPS_morgan_softNormalInTangent.fbx	80 KB	FBX File
Weapon_02_morganr@rig_softNormalInTangent.fbx	73 KB	FBX File
Weapon_02_avatar_morgan_skin_02_softNormalInTangent.fbx	72 KB	FBX File
Weapon_02_Rope_morgan.fbx	66 KB	FBX File
Weapon_02_FPS_morgan.fbx	65 KB	FBX File
Morgan_Skin_01_TPS_LOD0_Model_morgan_Skin_01_LOD0_LOD2.fbx	65 KB	FBX File
Weapon_01_TPS_morgan.fbx	49 KB	FBX File
Weapon_02_TPS_morgan_Skin_02_model_softNormalInTangent.fbx	48 KB	FBX File
Weapon_01_b_TPS_morgan_skin_06_tangent.fbx	45 KB	FBX File
Weapon_02_TPS_morgan_softNormalInTangent.fbx	41 KB	FBX File
Weapon_03_TPS_morgan_skin_02_model_softNormalInTangent.fbx	39 KB	FBX File
kong_prop_nop_prop_Kong_01_softNormalInTangent.fbx	38 KB	FBX File
Weapon_01_b_TPS_morgan_softNormalInTangent.fbx	36 KB	FBX File
Weapon_01_b_TPS_morgan.fbx	31 KB	FBX File
Plane001.fbx	22 KB	FBX File
Weapon_02_TPS_morgan.fbx	22 KB	FBX File
kong_prop_nop_propjoint_softNormalInTangent.fbx	21 KB	FBX File

模型+贴图渲染

太恐怖了！如果你的项目选择使用标准版Unity，那么资源被破解几乎是必然的！要想保护游戏资产，那你的项目最好有引擎源码，并从以下几个方向做一些优化，不然是无解的。

修改关键资源序列化

如果你的项目有源码，那么可以把一些关键资源的序列化字段改一下。比如 `Texture2D` 这个资源类型，它的数据结构大致如下

```
struct Texture2D: public Object {
    std::string m_Name;  // 1
    int32_t m_ForcedFallbackFormat;  // 2
    bool m_DownscaleFallback;  // 3
    int32_t m_Width;  // 4
    int32_t m_Height;  // 5
    int32_t m_CompleteImageSize;  // 6
    int32_t m_TextureFormat;  // 7
    int32_t m_MipCount;  // 8
    bool m_IsReadable;  // 9
    int32_t m_ImageCount;  // 10
    int32_t m_TextureDimension;  // 11
    GLTextureSettings m_TextureSettings;  // 12
    int32_t m_LightmapFormat;  // 13
    int32_t m_ColorSpace;  // 14
    TypelessData m_TexData;  // 15
    StreamingInfo m_StreamData;  // 16
};
```

在资源对象序列化过程中，`string` / `map` / `set` / `vector` 等数据类型均被当做数组来处理：先用4字节存储数组长度，然后按顺序存储数组元素，对于 `string` 它的数组元素类型是 `char` 。基于此，我们可以在标准版的Unity源码里面做一些微小改动，比如把上述类型序列化顺序交换位置，或者在这些字段的序列化之前写入一个很大的整形，这样通过标准版Unity反编译资源就会导致崩溃或者无限循环。该保护措施的本质是修改资源类型的数据结构，使其与标准版Unity生成TypeTree产生差异，从而导致通过标准版Unity破解资源的方法失效。

修改 `AssetBundleArchive` 存储结构

每个ab资源在Unity里面会都会通过一个 `AssetBundleArchive` 容器来存储，它的作用是压缩资源对象数据并提供文件寻址功能，并没有什么特别的，如果你的项目有源码完全可以自行设计一个实现类似功能的资源容器，比如王者荣耀、原神等游戏做了类似的设计。

修改 `SerializedFile` 存储结构

除了定制资源容器，还可以通过修改容器内资源的存储方式，比如修改 `SerializedFile` 的metadata数据的组织方式。

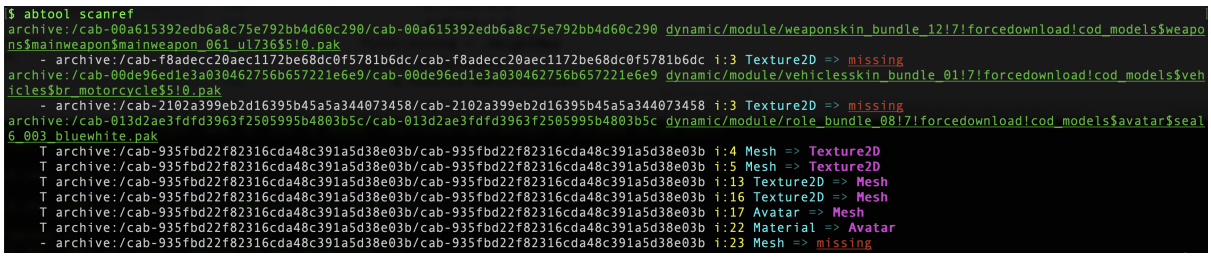
加密

如果觉得上述源码修改过于复杂，那么可以对ab文件做二进制的加密，也可以选择对其部分内容做加密，前提是不要有过多运行时开销，比如LOL手游做了类似的设计。

资源引发崩溃

一般情况下资源导致游戏崩溃并不多见，随着游戏资源量的增加以及构建时间的增加，就会需要用到Unity的ab增量编译，但是如果增量编译过程中发生了Unity闪退或者系统崩溃，那么就有很大几率导致编译出来的资源出问题，并且Unity下次构建时也无法自我修复。

举个例子，我们项目的ab资源增加到3G的时候遇到一次比较严重的由热更资源引发的崩溃，由于每次崩溃的时机各有不同，定位起来非常困难，最后通过一个能够稳定复现的崩溃定位到原因是：材质球A引用的贴图T在另外一个ab文件里面，但是通过材质球A的引用路径去加载后得到的是却是另外一个材质球B，可以理解为材质球A的贴图指针位置不是贴图T，那么通过材质球A的贴图指针尝试访问贴图T的时候就崩溃了。然后根据这个特征，笔者开发了 `scanref` 工具，它的作用是扫描所有引用了材质球、贴图、Mesh的对象，然后通过引用路径找到目标资源对象，并校验目标资源的类型是否跟预期一致。



图中的日志是什么含义呢？那是纯正的崩溃的味道！

```
T archive:/cab-935fbd22f82316cda48c391a5d38e03b/cab-935fbd22f82316cda48c391a5d38e03b i:4 Mesh => Texture2D
```

针对其中一行日志简单解释下， `archive:/` 开头一串文本是ab里面 `SerializedFile` 的路径，可以用来寻址。资源 `dynamic/module/role_bundle_08!7!forcedownload!cod_models$avatar$seal6_003_bluewhite.pak` 里面id为 `767` 的 `SkinnedMeshRenderer` 对象拥有一个外部资源指针 `PPtr<Mesh>` ，指向另外一个ab资源文件(`archive:/cab-935fbd22f82316cda48c391a5d38e03b/cab-935fbd22f82316cda48c391a5d38e03b`)中索引为 `m_PathID=4` 的对象，但是目标引用路径的资源其实是个 `Texture2D` 对象。

> 上述 `SkinnedMeshRenderer` 数据文本展开显示

```
<SkinnedMeshRenderer: 137> id=767
    m_GameObject: PPtr<GameObject>
        m_FileID = 0
        m_PathID = 169
    m_Enabled: bool = 1
    m_CastShadows: uint8_t = 1
    m_ReceiveShadows: uint8_t = 1
    m_ReceiveNoSSShadows: uint8_t = 0
    m_DynamicShadows: uint8_t = 1
    m_MotionVectors: uint8_t = 2
    m_LightProbeUsage: uint8_t = 1
    m_ReflectionProbeUsage: uint8_t = 3
    m_LightmapIndex: uint16_t = 65535
    m_LightmapIndexDynamic: uint16_t = 65535
    m_LightmapTilingOffset: Vector4f
        x: float = 1
        y: float = 1
        z: float = 0
        w: float = 0
    m_Materials: vector<PPtr<Material>>
        [0]: PPtr<Material>
            m_FileID = 0
            m_PathID = 3
    m_StaticBatchInfo: StaticBatchInfo
        firstSubMesh: uint16_t = 0
        subMeshCount: uint16_t = 0
    m_StaticBatchRoot: PPtr<Transform>
        m_FileID = 0
        m_PathID = 0
    m_ProbeAnchor: PPtr<Transform>
        m_FileID = 0
        m_PathID = 0
    m_LightProbeVolumeOverride: PPtr<GameObject>
        m_FileID = 0
        m_PathID = 0
    m_SortingLayerID: int32_t = 0
    m_SortingLayer: int16_t = 0
    m_SortingOrder: int16_t = 0
    m_Quality: int32_t = 0
    m_UpdateWhenOffscreen: bool = 0
    m_SkinnedMotionVectors: bool = 0
    m_Mesh: PPtr<Mesh>
        m_FileID = 4
        m_PathID = 4
    m_Bones: vector<PPtr<Transform>>
        [0]: PPtr<Transform>
            m_FileID = 0
            m_PathID = 703
    m_BlendShapeWeights: vector<float>
    m_RootBone: PPtr<Transform>
        m_FileID = 0
        m_PathID = 703
    m_AABB: AABB
        m_Center: Vector3f
            x: float = -0.145836
            y: float = 0.0210291
            z: float = 0.00812059
        m_Extent: Vector3f
            x: float = 0.109583
            y: float = 0.15261
            z: float = 0.10819
    m_DirtyAABB: bool = 0
```

> 通过引用路径找到的资源却是 `Texture2D` 对象

```
$ abtool edit dynamic/module/common\!7\!forcedownload\!cod_models\$avatar\$seal6_003.pak
[0] dynamic/module/common!7!forcedownload!cod_models$avatar$seal6_003.pak
$ lua file:dump_object(4)
<Texture2D:28> id=4
    m_Name:string = C_T_M_Seal6_Shotgun_003_N
    m_Width:int32_t = 1024
    m_Height:int32_t = 1024
    m_CompleteImageSize:int32_t = 626288
    m_TextureFormat:int32_t = 50
    m_MipCount:int32_t = 11
    m_IsReadable:bool = 0
    m_StreamingMipmaps:bool = 1
    m_StreamingMipmapsPriority:int32_t = 2
    m_StreamingGroupID:int32_t = 567068459
    m_ImageCount:int32_t = 1
    m_TextureDimension:int32_t = 2
    m_TextureSettings:GLTextureSettings
        m_FilterMode:int32_t = 2
        m_Aniso:int32_t = 1
        m_MipBias:float = 0
        m_WrapMode:int32_t = 0
    m_LightmapFormat:int32_t = 0
    m_ColorSpace:int32_t = 0
    m_TexData:TypelessData
        size = 0
        data =
    m_StreamData:StreamingInfo
        offset:uint32_t = 158432
        size:uint32_t = 626288
        path:string = archive:/CAB-935fbd22f82316cda48c391a5d38e03b/CAB-935fbd22f82316cda48c391a5d38e03b.resS
```

把 `Texture2D` 对象强转成 `Mesh` 对象赋值给 `SkinnedMeshRenderer` 对象，最终就会导致崩溃。庆幸的现在可以通过 `abtool scanref` 扫描游戏的所有资源，30秒就可以扫描3G左右的ab资源，可以说是非常高效的。

```
find . -iname '*.pak' | xargs abtool scanref
```

`scanref` 会把扫描数据缓存到当前目录的 `assets.ref` 文件，下次运行的时候 `scanref` 会自动读取缓存文件，这样不用再次扫描ab资源就可以快速得到相同的结果，直接把耗时降到2秒。

```
abtool scanref
```

上面的操作还只是发现资源问题，其实我们更想知道这样的问题如何解决，根据问题的严重性有两种解决方法：

1. 如果只有少量ab资源存在引用问题，那么可以把扫描出来的ab资源从构建机的输出目录删掉，下次构建时Unity会修复这些资源；
2. 如果有很多ab资源都存在引用问题，可以通过ab资源回退的方式来解决，前提是每次构建后都归档了相应的ab资源。具体操作是，下载历史构建的ab资源，运行 `abtool scanref` 找到一个没有资源引用问题的资源版本，然后用这个版本的ab资源替换构建机上的ab资源，并重新运行一次ab打包流程。

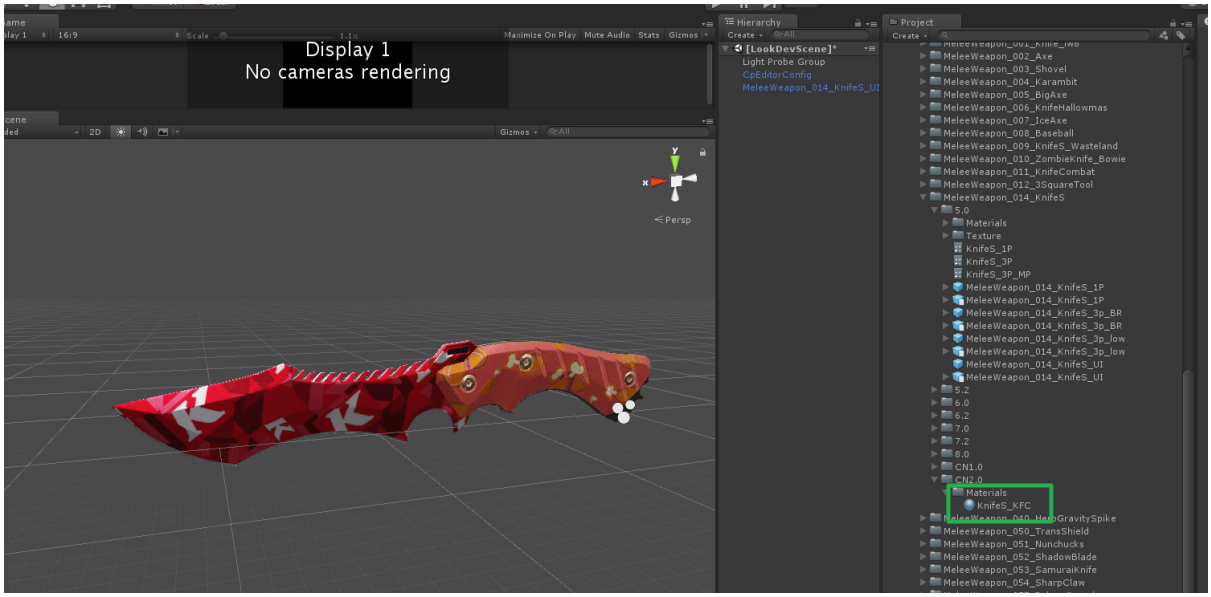
到现在为止，我们知道如何通过abtool发现资源问题，也拥有解决问题的方法，但是相信大家依然会有疑问：资源崩溃问题是如何产生的?

就上面 `SkinnedMeshRenderer` 错误引用 `Texture2D` 的例子来说，可以解释为：由于依赖关系变更，A和B两个ab资源都需要重新打包，但是ab资源B打包完成后，由于意外原因导致Unity打包流程中断，本来应该被重新打包的A资源，因为Unity的闪退而被忽略打包，所以A保持为老资源状态并引用了一个错误的资源，并且下次构建的时候Unity也无法识别这种异常情况。

资源引用丢失

在增量编译过程中如果Unity闪退了，除了会产生导致崩溃的问题资源，也会导致一些不那么严重、但是非常奇怪的渲染结果：屏幕渲染出现紫块或者渲染效果异常，通过 `scanref` 也可以发现问题。

比如上次我们制作了一把KFC主题的战斗刀，Editor里面看到是正常的



但是真机运行的时候就比较奇怪了，明显刀刃的贴图错了，并且刀把的质感也不是特别好。

这种情况下使用 `scanref` 检查一遍资源引用，发现了问题

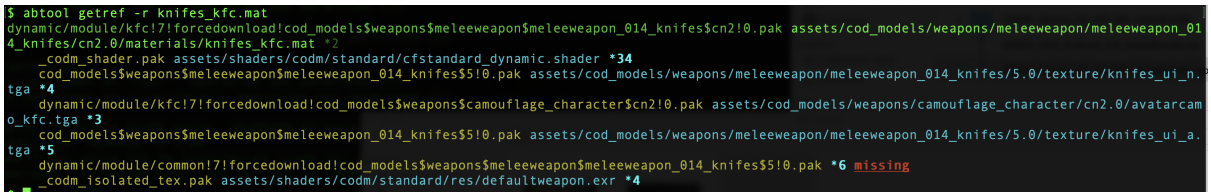
```
find . -iname '*.pak' | xargs abtool scanref
```

从日志看，资源 `dynamic/module/kfc!7!forcedownload!cod_models$weapons$meleeweapon$meleeweapon_014_knifes$cn2!0.pak` 引用了另外一个ab资源 `dynamic/module/common!7!forcedownload!cod_models$weapons$meleeweapon$meleeweapon_014_knifes$5!0.pak` 里面索引为 `6` 贴图，但是后者资源里面并没有这个贴图，那么这种情况需要把资源 `dynamic/module/common!7!forcedownload!cod_models$weapons$meleeweapon$meleeweapon_014_knifes$5!0.pak` 从构建机删掉重新打包。

定位这个问题，还可以从另外一个角度入手。首先通过 `saveobj` 命令保存资源的基本信息。

```
find . -iname '*.pak' | xargs abtool saveobj
```

然后通过 `getref -r` 命令直接查找出问题的材质球引用的资源列表，其中 `-r` 参数表示查找逆向下游资源的引用列表，没有 `-r` 参数情况下该命令查找当前资源的上游引用列表。



那么也能发现 `KnifeS_KFC.mat` 引用第五个贴图在目标资源 `dynamic/module/common!7!forcedownload!cod_models$weapons$meleeweapon$meleeweapon_014_knifes$5!0.pak` 里面找不到，两种方法得到同样结果。

上面被丢失的资源是贴图，如果丢失的资源是材质球，那么就会表现为屏幕紫块，定位方式类似。需要说明的是，上面提到的资源引用丢失问题指的是资源引用在，但是目标资源不存在的情况，如果原始资源引用指针为空，那么这种问题在开发阶段阶段就能发现，当然工具也能发现引用为空的情况。

为了确认资源指针是否为空，我们需要用到另外一个命令 `edit`

```
abtool edit dynamic/module/kfc\!7\!forcedownload\!cod_models\$weapons\$meleeweapon\$meleeweapon_014_knifes\$c
```

然后进入命令行交互模式，在这种模式可以执行 `lua` 脚本，比如可以输入 `lua file:dump_object(2)` ，其中 `2` 为 `KnifeS_KFC.mat` 的资源索引，通过 `getref` 命令得到的。

贴图

```
$ lua file:dump_object(2)
<Material:21> id=2
    m_Name:string = KnifeS_KFC
    m_Shader:PPtr<Shader>
        m_FileID = 5
        m_PathID = 34
    m_ShaderKeywords:string = _DYNAMIC_ADVANCE_DETAIL_LERP
    m_LightmapFlags:uint32_t = 4
    m_EnableInstancingVariants:bool = 1
    m_DoubleSidedGI:bool = 0
    m_CustomRenderQueue:int32_t = -1
    stringTagMap:map<string,string>
    disabledShaderPasses:vector<string>
    m_SavedProperties:UnityPropertySheet
        m_TexEnvs:map<string,UnityTexEnv>
            first:string = _AlphaMap
            second:UnityTexEnv
                m_Texture:PPtr<Texture>
                    m_FileID = 0
                    m_PathID = 0
                m_Scale:Vector2f
                    x:float = 1
                    y:float = 1
                m_Offset:Vector2f
                    x:float = 0
                    y:float = 0
            first:string = _BentNormalMap
            second:UnityTexEnv
                m_Texture:PPtr<Texture>
                    m_FileID = 0
                    m_PathID = 0
                m_Scale:Vector2f
                    x:float = 1
                    y:float = 1
                m_Offset:Vector2f
                    x:float = 0
                    y:float = 0
            first:string = _BumpMapPakced
            second:UnityTexEnv
                m_Texture:PPtr<Texture>
                    m_FileID = 1
                    m_PathID = 4
                m_Scale:Vector2f
                    x:float = 1
                    y:float = 1
                m_Offset:Vector2f
                    x:float = 0
                    y:float = 0
            first:string = _CutRimTex
            second:UnityTexEnv
                m_Texture:PPtr<Texture>
                    m_FileID = 0
                    m_PathID = 0
                m_Scale:Vector2f
                    x:float = 1
                    y:float = 1
                m_Offset:Vector2f
                    x:float = 0
                    y:float = 0
            first:string = _DetailAlbedoMap
            second:UnityTexEnv
                m_Texture:PPtr<Texture>
                    m_FileID = 4
                    m_PathID = 3
                m_Scale:Vector2f
```

```
                    x: float = 3
                    y: float = 3
                m_Offset: Vector2f
                    x: float = -0.15
                    y: float = 0.8
            first: string = _DetailNormalMap
            second: UnityTexEnv
                m_Texture: PPtr<Texture>
                    m_FileID = 0
                    m_PathID = 0
                m_Scale: Vector2f
                    x: float = 1
                    y: float = 1
                m_Offset: Vector2f
                    x: float = 0
                    y: float = 0
            first: string = _EmissionMap
            second: UnityTexEnv
                m_Texture: PPtr<Texture>
                    m_FileID = 0
                    m_PathID = 0
                m_Scale: Vector2f
                    x: float = 1
                    y: float = 1
                m_Offset: Vector2f
                    x: float = 0
                    y: float = 0
            first: string = _HeightMap
            second: UnityTexEnv
                m_Texture: PPtr<Texture>
                    m_FileID = 0
                    m_PathID = 0
                m_Scale: Vector2f
                    x: float = 1
                    y: float = 1
                m_Offset: Vector2f
                    x: float = 0
                    y: float = 0
            first: string = _MainTex
            second: UnityTexEnv
                m_Texture: PPtr<Texture>
                    m_FileID = 1
                    m_PathID = 5
                m_Scale: Vector2f
                    x: float = 1
                    y: float = 1
                m_Offset: Vector2f
                    x: float = 0
                    y: float = 0
            first: string = _MetallicRoughnessMap
            second: UnityTexEnv
                m_Texture: PPtr<Texture>
                    m_FileID = 2
                    m_PathID = 6
                m_Scale: Vector2f
                    x: float = 1
                    y: float = 1
                m_Offset: Vector2f
                    x: float = 0
                    y: float = 0
            first: string = custom_IBLCubemap
            second: UnityTexEnv
                m_Texture: PPtr<Texture>
                    m_FileID = 3
                    m_PathID = 4
                m_Scale: Vector2f
```

```
                    x: float = 1
                    y: float = 1
                m_Offset: Vector2f
                    x: float = 0
                    y: float = 0
```

在Unity资源里面通过 `PPtr<T>` 表示一个资源引用，比如，上面丢失的 `_MetallicRoughnessMap` 贴图，它的指针有两个字段： `m_FileID` 表示目标资源的文件索引， `m_PathID` 表示目标资源在目标文件里面的资源索引ID，其中 `m_FileID` 可以通过 `SerializedFile` 的metadata里面可以索引到一个全局唯一的路径 `archive: /cab-3fc774b79be2c61a94690dbb2913f522/cab-3fc774b79be2c61a94690dbb2913f522` ，这样通过 `m_FileID` 和 `m_PathID` 两个字段可以全局确定一个唯一的资源对象。

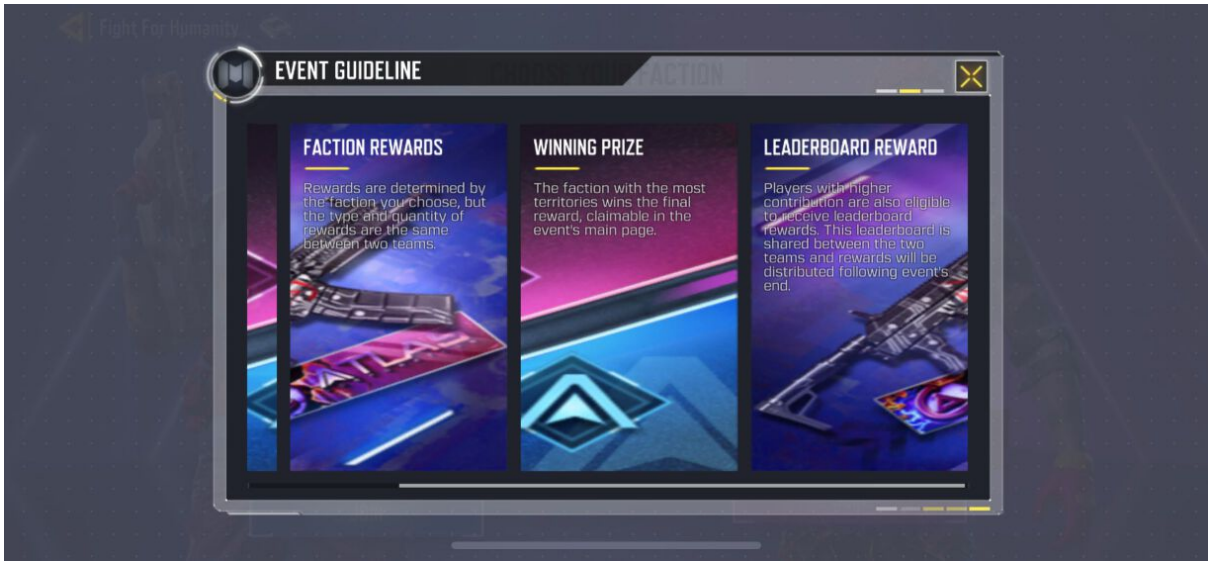
```
first: string = _MetallicRoughnessMap
second: UnityTexEnv
    m_Texture: PPtr<Texture>
        m_FileID = 2
        m_PathID = 6
```

`m_FileID=0` 表示当前资源指针指向当前文件内的资源对象， `m_PathID=0` 表示当前资源指针为空，这时通过Unity的Inspector可以看到资源的槽位是空的，所以如果发现某个资源指针的 `m_PathID=0` 那么这就是一个资源空引用案例，如下所示。

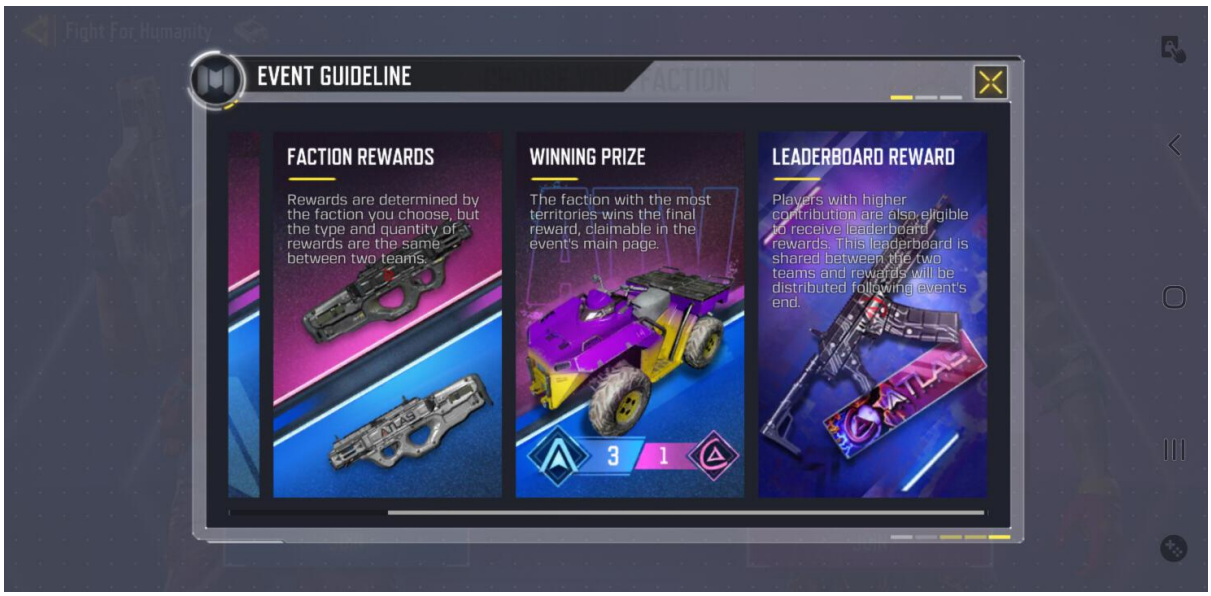
```
first: string = _HeightMap
second: UnityTexEnv
    m_Texture: PPtr<Texture>
        m_FileID = 0
        m_PathID = 0
```

资源显示异常

某天笔者被拉去查一个iOS资源显示问题，如下



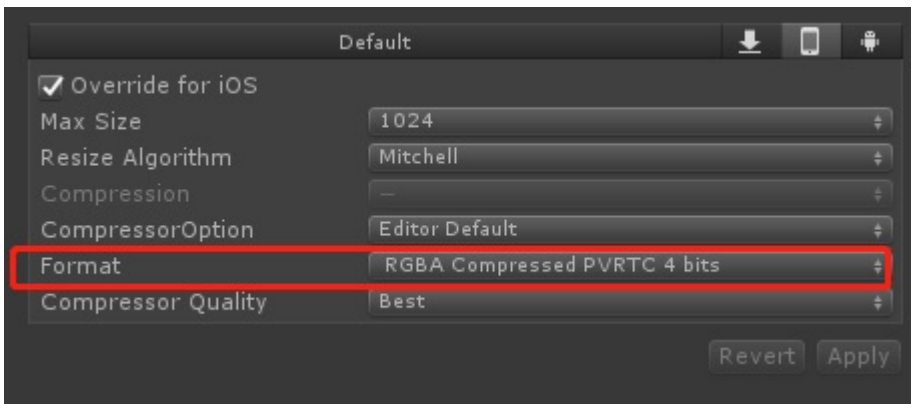
正常应该是这样的



从表象来看是图片被横向拉伸了，美术同学和开发同学都说资源没问题，Editor验收也正常，安卓的真机也没问题。如果是你该怎么定位这个问题?

我来定位的话，就简单多了！工具在手，首先反编译下贴图：通过 `abtool savetex` 得到 `Update_4_1.1024x1024.pvrtc_rgba4.tex` ，再用textool转码得到tga文件，下图为tga转了jpg的显示。

从结果来看，图片确实被拉伸了，原图是512x1024尺寸的，实际变成1024x1024尺寸了，这是由于PVRTC不支持长方形的贴图导致的：对于非正方形的贴图，会被PVRTC强制拉伸成正方形再进行编码。当然，最终打开Unity那一刻也确认下笔者的判断。



修复也很简单，改成ASTC格式就可以了。

资源差异比对

跟进过Unity资源热更发布的朋友就会有感触，每次发布都会自我灵魂拷问：变更的ab资源是否都需要发布？有些资源明确没有改动，但是打出来的ab文件md5却是不同的，该怎么办？不发不放心，发了也不安心！

那么有了abtool那可就简单多了，通过 `dump` 命令把ab文件解压为格式整齐、开发友好的文本文件。

```
# 当前cd目录为ab资源存储目录
find . -iname '*.pak' | while read pak
do
    abtool dump "${pak}" | tee "${pak}.log"
done
```

通过上面三行bash脚本可以轻松把所有的ab文件转成文本文件，然后打开BeyondCompare就一切都明了了。对于修改的资源我们会有心理预期，一般比较在意的是预期之外被修改到的资源，据笔者的经验可以大概分为一下几类：

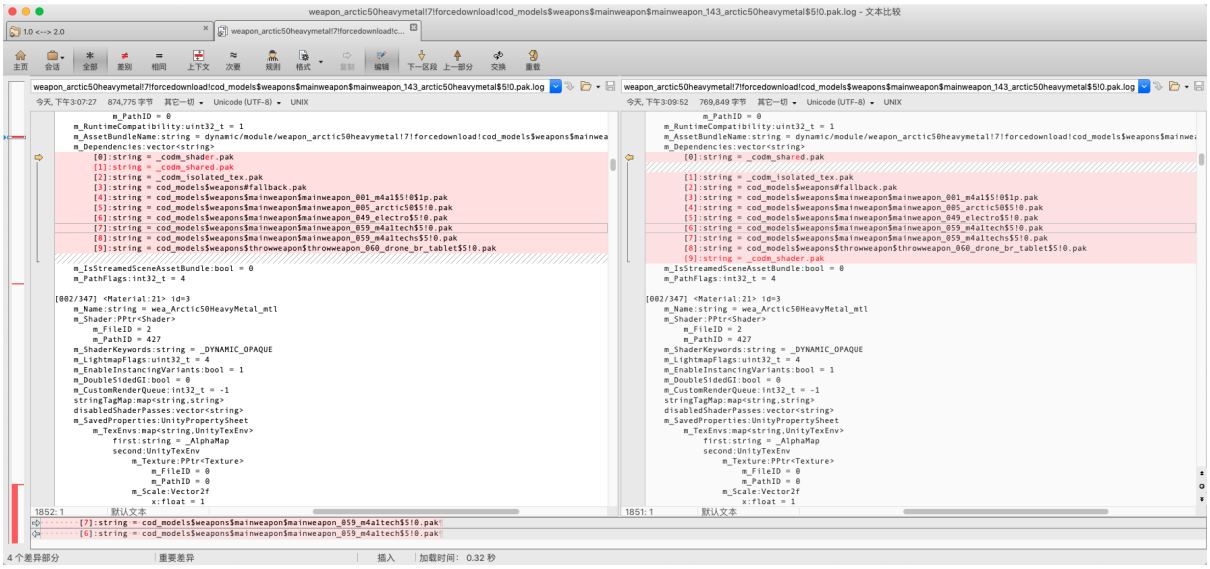
- 依赖列表顺序
- 增删进包文件
- 动画浮点数序列化不稳定
- 修改MonoBehaviour成员变量

依赖列表顺序

每个ab文件里面都有个 `AssetBundle` 对象，其数据结构大致如下

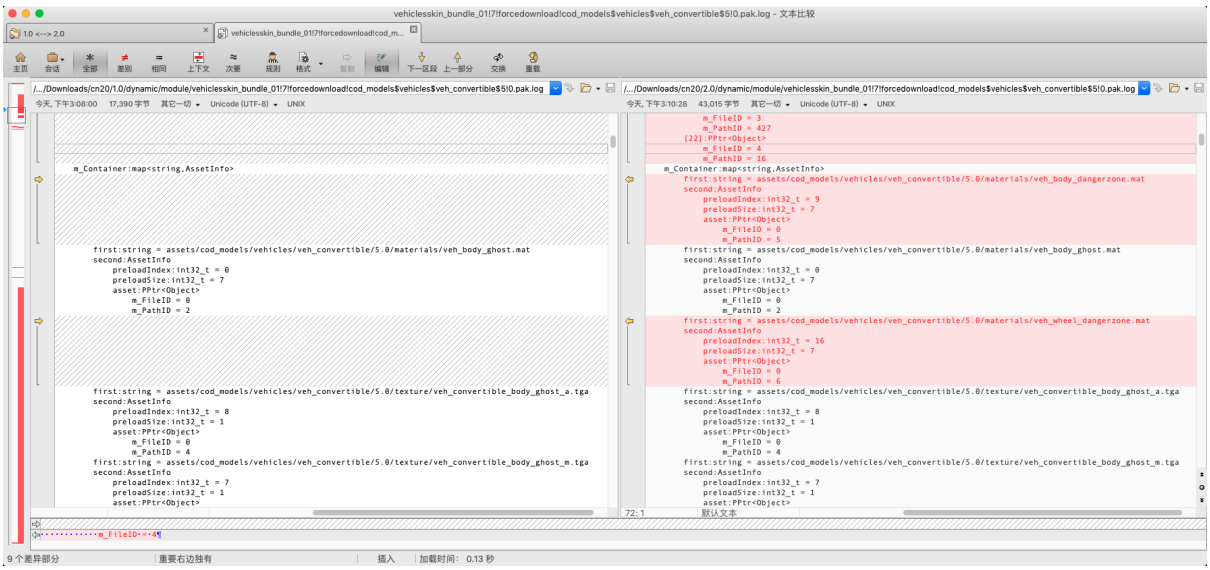
```
struct AssetBundle {
    std::string m_Name;  // 1
    std::vector<PPtr<struct Object>> m_PreloadTable;  // 2
    std::multimap<std::string, AssetInfo> m_Container;  // 3
    AssetInfo m_MainAsset;  // 4
    uint32_t m_RuntimeCompatibility;  // 5
    std::string m_AssetBundleName;  // 6
    std::vector<std::string> m_Dependencies;  // 7
    bool m_IsStreamedSceneAssetBundle;  // 8
    int32_t m_ExplicitDataLayout;  // 9
    int32_t m_PathFlags;  // 10
    std::map<std::string, std::string> m_SceneHashes;  // 11
};
```

`AssetBundle::m_Dependencies` 字段存储了当前ab依赖的其他ab文件列表，Unity在打包过程中可能生成不稳定的顺序，但是列表内容相同。



增删进包文件

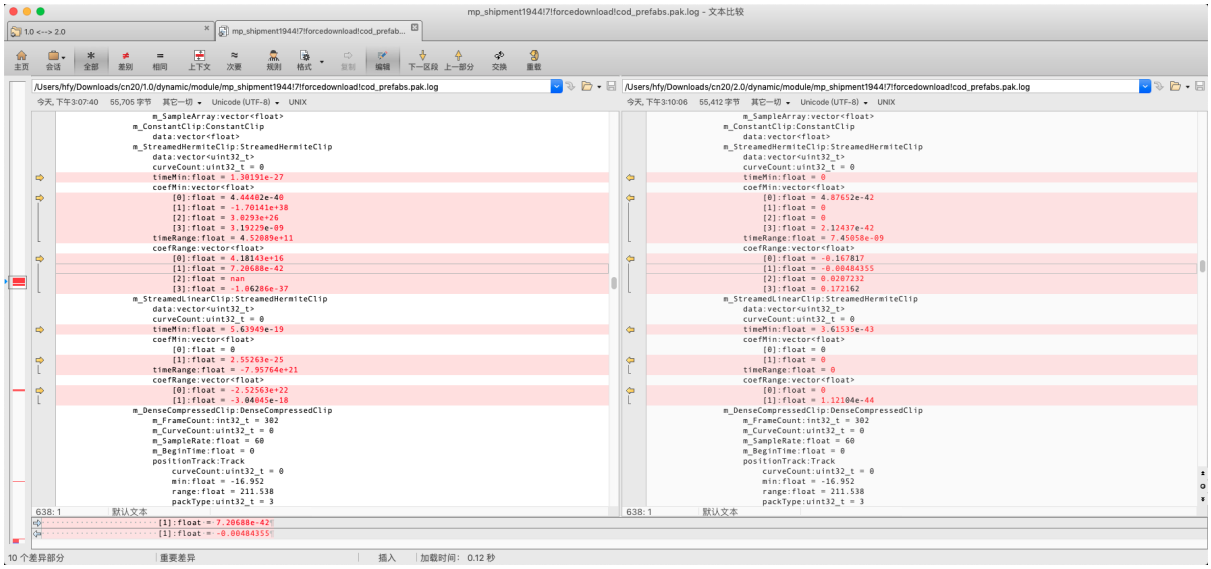
如果生成ab文件的配置发生了变化，比如增删了文件路径，那么即使游戏资产没变的情况下也会导致ab文件最终发生变化。



动画资产浮点数序列化不稳定

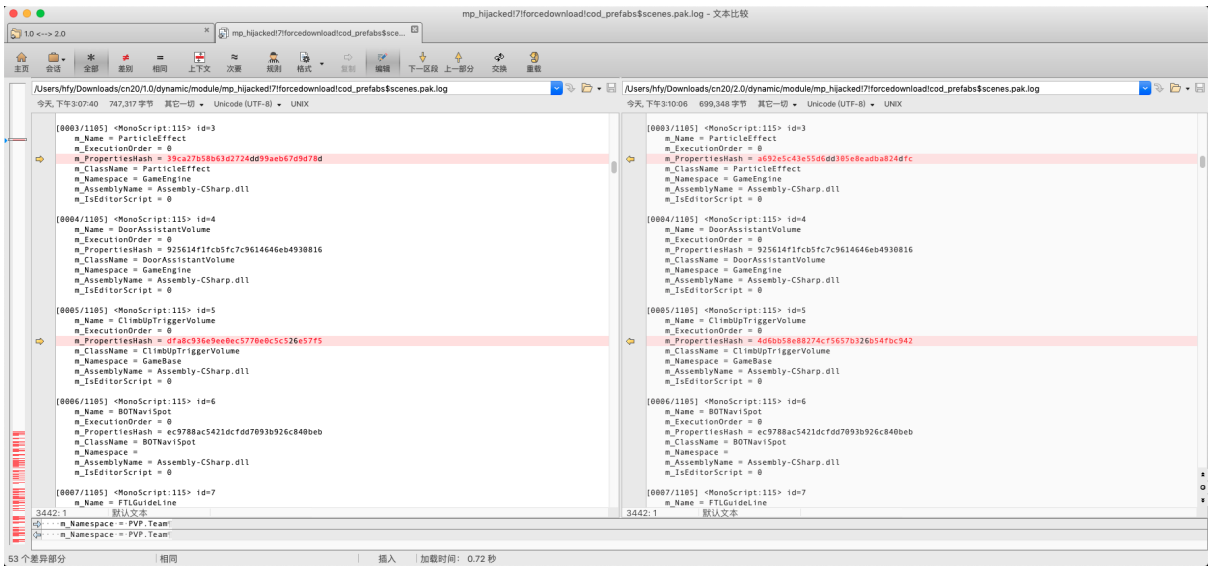
`AnimationClip` 动画资产里面的Hermite样条曲线很容易产生不稳定的结果，主要表现为往极大或者极小值方向变化。

```
struct StreamedHermiteClip {
    std::vector<uint32_t> data;   // 1
    uint32_t curveCount;   // 2
    float timeMin;   // 3
    std::vector<float> coefMin;   // 4
    float timeRange;   // 5
    std::vector<float> coefRange;   // 6
};
```



修改MonoBehaviour成员变量

`MonoBehaviour` 脚本在序列化的时候会把成员变量按照一定规则算出一个128位的hash值，如果修改了脚本变量那么理论上是不能够热更的，否则运行时将无法对资源进行反序列化和对象加载，一般只有商店版本发布的时候才发布修改后的脚本。



虽然程序版本发布的时候可以修改 `MonoBehaviour` 脚本，但是这样一般还会带来有另外一个问题：由于 `MonoBehaviour` 的 `Hash128` 以及TypeTree会序列化到资源里面，所以修改 `MonoBehaviour` 会导致最终的ab资源变更，并且难以发现。笔者基于这种情况，开发了 `mono` 命令，可以扫描脚本在ab文件里面的分布情况，简单说可以查看修改某个脚本会影响多少资源的变更。

```
find . -iname '*.pak' | xargs abtool mono
```

`mono` 命令执行后会把扫描到的数据缓存下来，下次可以直接运行 `abtool mono` 而不用再次扫描ab资源。

```
$ abtool mono
[001/100] GameEngine:AssetRef #1174 *326 =182,852,762 174.38M b3cc095b0e1d8585f7cd2c1347d807f5
[002/100] CODPostEffectProfile #11 *11 =131,886,869 125.78M 55aea0e1880380d590a243892d1d09ab
[003/100] GameBase:TacticalGPSCheckScriptableObject #11 *11 =131,886,869 125.78M 8d44fea244da305a7e7016b341e2ee7a
[004/100] GameEngine:AvatarComposer #76 *38 =92,725,676 88.43M c0582a8f2ce3a5128a5bed083f7e86f7
[005/100] BRAnimationBlender #42 *38 =92,725,676 88.43M 15b2c58e7d33e16b1d49eda8e2a42d29
[006/100] GameBase:AimIKSolverComponent #42 *38 =92,725,676 88.43M f60263465652779 61dc34b6b55a94668
[007/100] BakeryLightmapGroup #8 *8 =82,649,317 78.82M 4f03b63609449cac61d93b48f6eabcf5
[008/100] CameraAnimation #36 *36 =51,178,737 48.81M fec96cfff494ffce80382b4aff99d7bf
[009/100] GameBase:TacticalMapCompass #209 *33 =35,821,269 34.16M e8c7fcaa18284ae1c4cfdb712c5aad93
[010/100] DynamicBone #36 *8 =32,433,264 30.93M 278f2137114f518614a58fc1dffa8933
[011/100] LightProbeAutoDistribute #12 *11 =30,626,696 29.21M 58792a9899412f223780468051dea2b8
[012/100] SABoneColliderChild #78 *30 =30,083,909 28.69M 05524b8f659e7bfe5756d1335fec24ba
[013/100] WeaponControllerAsset #29 *29 =25,924,376 24.72M f14ffb770b57e9e6402a94813d4d3431
[014/100] BakeryLightmapGroupSelector #14 *11 =25,785,957 24.59M 5fcf80da9f5d153cc82e6319d8322f57
[015/100] PVS:WhiteAreaInfo #13 *11 =25,785,957 24.59M 39ea65533f723b4ba8738658462a2d7a
[016/100] GameBase:SceneInfo #11 *11 =25,785,957 24.59M dd652a46af2383253f8fdfef53cfbc74
[017/100] GameEngine.TextureAtlas:TextureAtlasingUpdater #11 *11 =25,785,957 24.59M 7a9c8fa8769acfe1f076cd4457880fc2
[018/100] GameEngine:GameSceneRoot #11 *11 =25,785,957 24.59M 4c2e03e7e5bf7de9288ce61756598ae4
[019/100] GameEngine:SceneSetting #11 *11 =25,785,957 24.59M 16f8ebce4f267c00000d4bcae02ad125
[020/100] PVS:PVSBuilder_OCCQuery #11 *11 =25,785,957 24.59M daa4c8e9d554387c61c2d10b30c1eead
[021/100] PVS:VCRBuilder_OCCQuery #11 *11 =25,785,957 24.59M 19c03e00c85020714b1cbd28b99da2ac
[022/100] PVS:ViewCellRoot_OCCQuery #11 *11 =25,785,957 24.59M 7c200a207ff7b87c972d43069e888d76
[023/100] GameEngine:AvatarEquipConfig #13 *7 =25,713,945 24.52M 4ee2961824e43e28afe1d5d9f22524b4
[024/100] GameBase:WeaponPrefabConfig #25 *25 =25,112,009 23.95M 8547ad019aad4e7190ffdf75aa37ef5b
[025/100] JadeSnakeRenderer:GPUBakeInspectorData #8 *8 =24,840,742 23.69M 902556fef10e025386f8be89ada3171b
[026/100] BakeryDirectLight #9 *9 =22,954,478 21.89M 258d95b0ffe9a58b02166c2113c091c2
[027/100] JadeSnakeRenderer:JsAmbientLight #9 *7 =22,942,899 21.88M 0f5f592ce750c291e94c2537d6a08180
[028/100] JadeSnakeRenderer:JsDirectionalLight #8 *7 =22,942,899 21.88M 3a55a6aaf32599cd8ab5a1fba3f9638a
[029/100] UnityEngine.AI:CustomNavMeshSurface #179 *15 =21,727,093 20.72M b60138feb0f6bf264208d39c35f8df4c
[030/100] BakerySkyLight #8 *8 =21,261,728 20.28M 2330c21be80bf4386f24b9d56d4f3cc7
[031/100] GameUI:UIAnimationEventPlay #9 *9 =21,247,409 20.26M 67fa3b87d8331c2dca5867f5720a1fff
[032/100] UnityEngine.AI:NavMeshModifierVolume #140 *11 =18,421,860 17.57M 860308b0e97e97a6ef46197d060598f2
[033/100] PrefabEvolution:EvolvePrefab #146 *10 =16,824,807 16.04M edecc0fe1211bf364f91045a11591127
[034/100] Audio.Components:VolumeClientDetector #2306 *11 =16,459,518 15.70M bc5334daa66d7da9d603032b1207ca77
```

从扫描结果来看，命名空间 `GameEngine` 里面的 `AssetRef` 类被174M的ab资源引用，也就是说，修改了 `AssetRef` 脚本会导致174M的ab资源发生变化。

在大版本升级过程中我们希望有尽可能多的资源可以复用避免重复下载，那么可以结合 `cmpmono` 来对比大版本升级前后有哪些影响ab资源的 `MonoBehaviour` 脚本发生了修改。具体做法如下：

1. 分别在需要对比的ab资源目录运行 `mono` 命令，这样会生成对应的扫描数据文件 `monoscripts.ms`
2. 运行 `cmpmono -s [source.ms] -d [destination.ms]` 输出对比结果

```
abtool cmpmono -s iMSDK_CN_Android_108_AssetBundle/monoscripts.ms -d iMSDK_CN_Android_145_AssetBundle/monoscr
```

```
$ abtool cmpmono -s iMSDK_CN_Android_108_AssetBundle/monoscripts.ms -d iMSDK_CN_Android_145_AssetBundle/monoscripts.ms | grep ^M
M CODPostEffectProfile #11 *11 =131,886,869 125.78M 55aea0e1880380d590a243892d1d09ab => e77bcfeae0c5e716b7251ca7f923f7e3
M GameEngine:AvatarComposer #76 *38 =92,725,676 88.43M c0582a8f2ce3a5128a5bed083f7e86f7 => e7fb5260ddc49f574b193cb9a87af77f
M CameraAnimation #36 *36 =51,178,737 48.81M fec96cfff494ffce80382b4aff99d7bf => ce6cbcbda7598b3f314164a61bba2b66
M GameBase:SceneInfo #11 *11 =25,785,957 24.59M dd652a46af2383253f8fdfef53cfbc74 => ce5e48ca732cfb92da6c7ed5bbd68cec
M GameEngine:SceneSetting #11 *11 =25,785,957 24.59M 16f8ebce4f267c00000d4bcae02ad125 => ca8b224aae10d59d99ad514786ee74f0
M GameEngine:AvatarEquipConfig #13 *7 =25,713,945 24.52M 4ee2961824e43e28afe1d5d9f22524b4 => ed2aa739e20b7256d8dd5aac91a20479
M GameBase:WeaponPrefabConfig #25 *25 =25,112,009 23.95M 8547ad019aad4e7190ffdf75aa37ef5b => 47c4f338d0e4e7eb78a576921bef7980
M GameUI:UIAnimationEventPlay #9 *9 =21,247,409 20.26M 67fa3b87d8331c2dca5867f5720a1fff => 47367b7309d1f7ee2ba036e00b0a7503
M GameEngine:ParticleEffect #64 *12 =15,662,663 14.94M 39ca27b58b63d2724dd99aeb67d9d78d => a692e5c43e55d6dd305e8eadba824dfc
M GameBase:MatchCameraMove #22 *22 =13,320,720 12.70M 7c1f8ebd04d869264d863537321db777 => 7e3a18fabcea084b39a26e8debe37b5f
M WorldInfo #175 *11 =13,278,410 12.66M 7b88e90d9ef9912e598284348fedc08f => f0c0c73b02f981845ff350d0c3cc40ee
M GameUI:UIGeneralSequence #12 *6 =10,751,599 10.25M 5e22f0619633c5b4fbac175245df291b => 4c2eadccd68b2a33e7372a693bf3a502
M GameBase:ClimbUpTriggerVolume #1284 *14 =7,052,109 6.72M dfa8c936e9ee0ec5770e0c5c526e57f5 => 4d6bb58e88274cf5657b326b54fbc942
M GameEngine:ParticleEffect_Random #3 *2 =1,308,703 1.25M e40df7ddfd41e65c662dd2173e5c7f5b => d8345730152e6a63b7dbe5125294968e
```

这样可以很清晰地看到有多少资源在大版本升级后因为修改 `MonoBehaviour` 脚本而发生变化，通过这个列表可以作相应资源优化。

资源编辑

以下演示继续用战歌竞技场的资源，下面是游戏大厅的界面。

笔者的目标是要把中间那个森林棋盘的Shader去掉，期望棋盘显示为紫色效果。

首先在ab资源里面找到文件 `dataconfig.god` ，看起像来是配置文件，通过 `list` 命令发里面都是二进制配置。

```
abtool list dataconfig.god
```

日志中有个 `dataconfig_chess_board_model_conf.bytes` 的资源看起来是棋盘相关的配置，我们通过 `saveta` 命令提取这些二进制配置，默认输出到 `__textassets` 目录。

```
abtool saveta dataconfig.god
```

尝试用 `protoc` 解析下发现protobuf序列化的配置文件

```
protoc --decode_raw < __textassets/dataconfig_chess_board_skin_conf.bytes | pbdecode
```

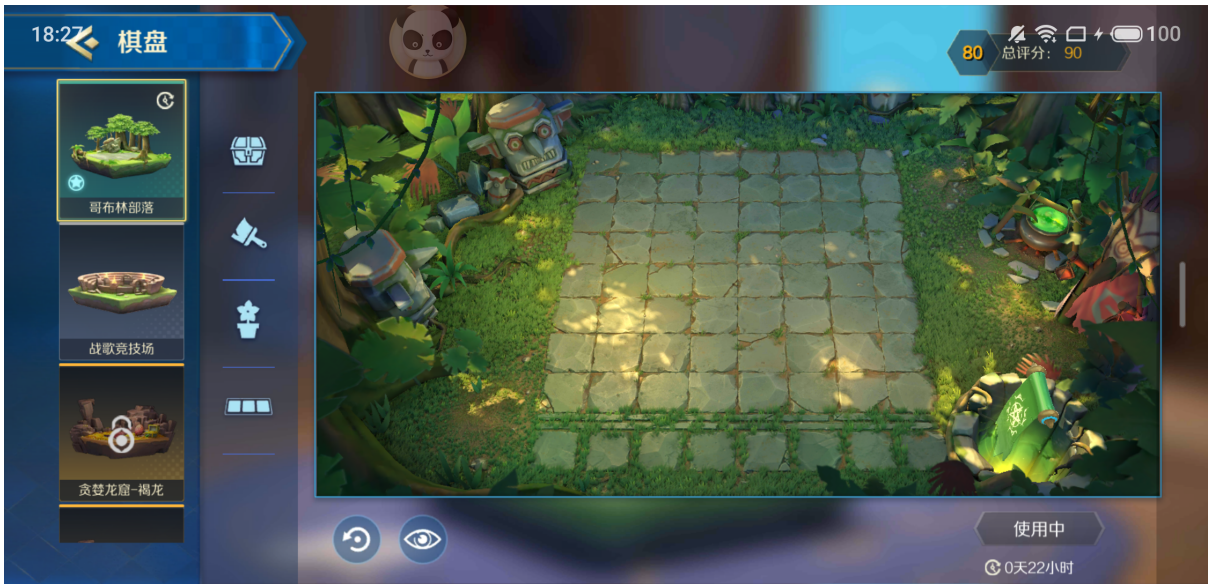
其中 `pbdecode` 是笔者写的一个小工具，可以把protoc打印出来的八进制编码转成UTF-8编码，主要方便查看配置中文信息，可以复制以下C++代码自行编译。

贴图

```
#include <assert.h>
#include <iostream>
#include <fstream>

void decode(const char *input, char *output, size_t length, bool newline = true)
{
    auto wCursor = output;

    auto rCursor = input;
    auto end = rCursor + length;
    while (rCursor < end)
    {
        *wCursor = *rCursor;
        if (*rCursor == '\\')
        {
            auto ptr = rCursor + 1;
            if (*ptr >= '0' && *ptr <= '9')
            {
                auto byte = 0;
                for (auto n = 0; n < 3; n++)
                {
                    byte = (byte << 3) | *ptr - '0';
                    ++ptr;
                }

                assert(byte <= 0xFF);
                *wCursor = static_cast<char>(byte);
                rCursor += 3;
            }
        }

        ++rCursor;
        ++wCursor;
    }

    *wCursor = 0;
    std::cout << output;
    if (newline) { std::cout << '\n'; }
    std::cout << std::flush;
}

void decode(const char *path)
{
    std::ifstream fs(path, std::ifstream::binary);
    fs.seekg(0, std::ios_base::end);
    auto length = static_cast<size_t>(fs.tellg());
    fs.seekg(0);

    char buffer[length];
    fs.read(buffer, length);
    fs.close();

    decode(buffer, buffer, length, false);
}

int main(int argc, const char * argv[])
{
    if (argc > 1)
    {
        for (auto i = 1; i < argc; i++)
        {
            decode(argv[i]);
        }
    }
    else
```

```
    {
        std::string pipe;
        while (std::getline(std::cin, pipe))
        {
            auto size = pipe.size();

            char buffer[size];
            decode(pipe.c_str(), buffer, size);
        }
    }

    return 0;
}
```

然后打开棋盘弹窗，在这里我们知道棋盘的名字叫 `哥布林部落`



那么我们从刚刚解开的配置里面确实找到了相关数据

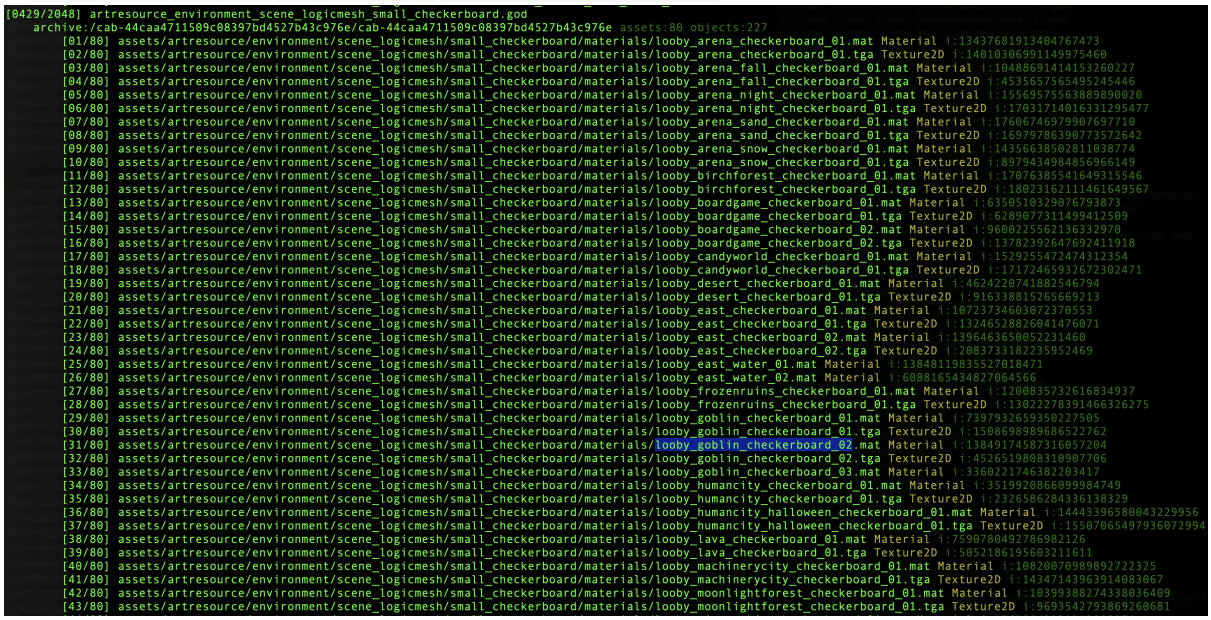
```
1 {
  1: 18
  2: "哥布林部落"
  3: "Prefabs/Environment/ChessBoard/ChessBoard_01"
  4: "Scene_Goblin"
  5: "DataConfig/CardPoolMat/img_game_chessshop_bg"
  6: "Prefabs/Environment/Scene/SmallChessBoard/Small_Goblin_01"
  7: 13
  8: 101
  9: 10501
  10: 1002
  11: 1001
  12: 1003
  14: "map_11_goblin.bnk"
  15: 9027
  17: 1
  18: 1541462414
  19: 1762473614
  20: 58
  21: 0
  22: 60
}
```

从配置知道这个棋盘的资源名大概是 `Small_Goblin_01` ，下面通过 `list` 命令穷举所有资源列表。

```
find . -iname '*.god' | xargs abtool list -r
```

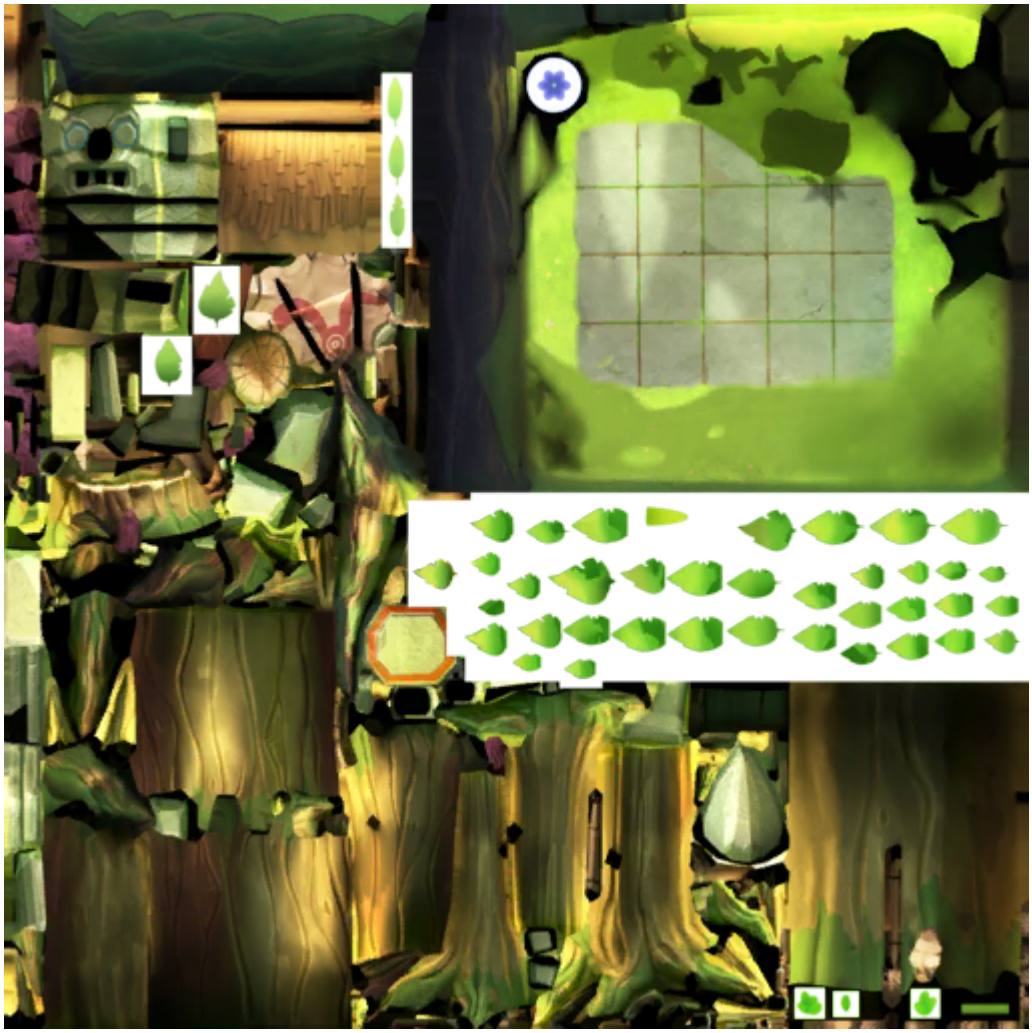
通过关键字 `Small_Goblin_01` 搜索，发现文件名

为 `artresource_environment_scene_logicmesh_small_checkerboard.god` 的ab可能是我们要找的目标。

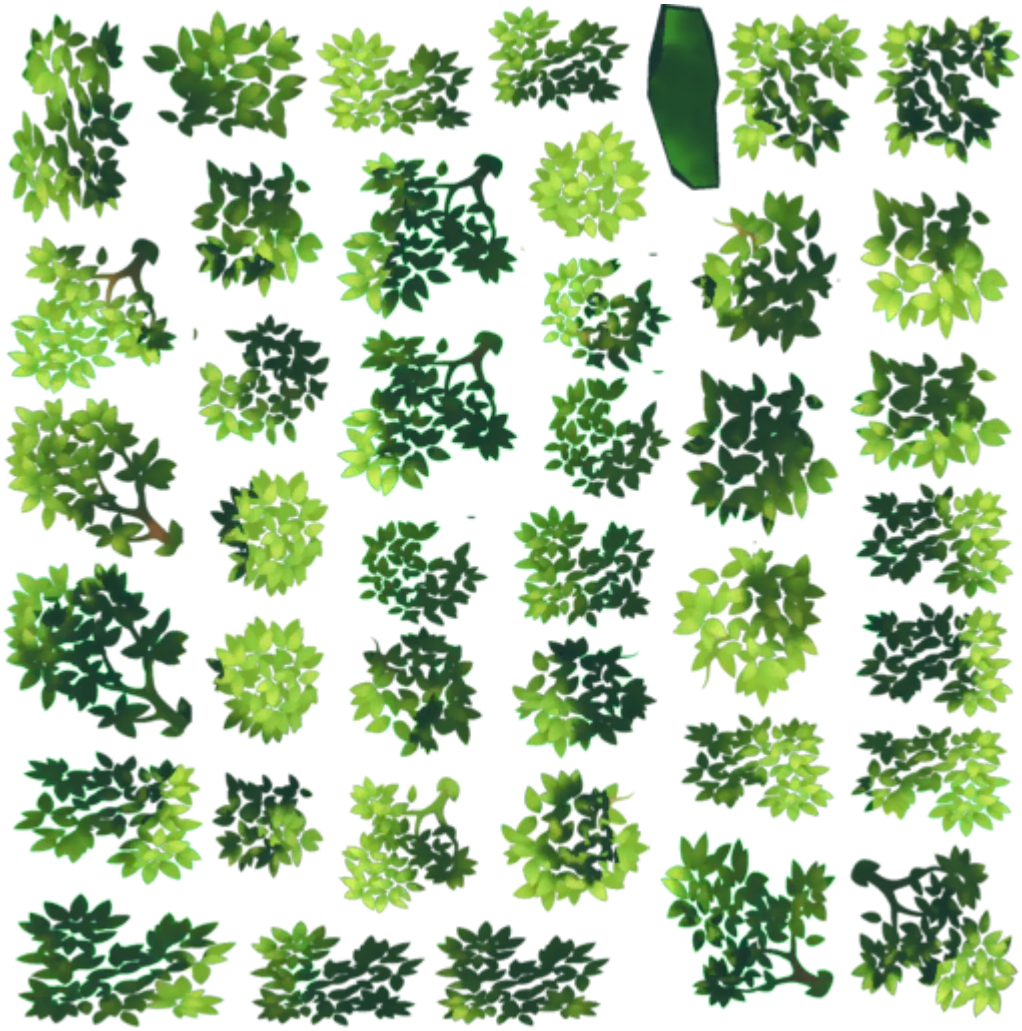


但现在还不是特别确定，所以可以先把资源反编译出来确认下，这里使用 `savetex` 保存ab里面的贴图，并用 `textool` 转码贴图文件，然后可以很容易发现，下面这两张贴图就是大厅的棋盘用的。

棋盘树干以及底座的贴图



棋盘的树叶部分贴图



接下来我们选择与贴图同名的 `looby_goblin_checkerboard_01.tga` 材质球 `looby_goblin_checkerboard_01.mat` 作为切入点，它的资源ID为 `7397932659350227505` ，剩下的工作就交给 `edit` 命令来实现了。

```
abtool edit artresource_environment_scene_logicmesh_small_checkerboard.god
```

通过上面命令进入交互模式，这里可以执行lua代码，依次输入以下命令得到编辑后的ab文件。

```
lua so = file:find(7397932659350227505)
lua ptr = castMaterial(so.object)
lua mat = ptr:get()
mat.m_Shader.m_PathID = 0
mat.m_Shader.m_FileID = 0
save
```

上面的代码每输入一行都要按一次回车键，最终结果是棋盘材质球的Shader引用置空了，最后的 `save` 命令会把修改后的ab文件保存到 `__archives` 目录，然后把里面的ab文件 `adb push` 到游戏的外存储目录，因为一般的热更逻辑是先读外存储的资源，外存储没有才会读安装包里面的资源，这个设计的目的是支持游戏热更。

见证奇迹时刻

上面 `edit` 模式需要手动码字，如果需要多次运行的时候就会显得不太方便，那么可以在当前cd目录写一个lua脚本 `abtool.lua` 。

```
-- 通过m_PathID查找SerializedObject对象
trace('原始材质球')
so = file:find(7397932659350227505)
-- 打印原始材质
file:dump_object(7397932659350227505)
-- 类型转换
ptr = castMaterial(so.object) -- std::shared_ptr<Material>
-- 获取材质球对象引用
mat = ptr:get() -- Material*
-- 设置材质的Shader为空引用
mat.m_Shader.m_PathID = 0
mat.m_Shader.m_FileID = 0
-- 打印修改后材质
trace('修改后材质球')
file:dump_object(7397932659350227505)
-- 保存当前修改
builder = assetbundle.ArchiveFileBuilder(archive) -- assetbundle::ArchiveFileBuilder
builder:save("hijack/artresource_environment_scene_logicmesh_small_checkerboard.god")
```

然后通过 `lua` 命令来执行这个脚本。

```
# 需要把abtool.lua脚本放到当前cd目录
abtool lua artresource_environment_scene_logicmesh_small_checkerboard.god
```

最终在lua脚本指定的保存目录生成编辑后的ab文件。

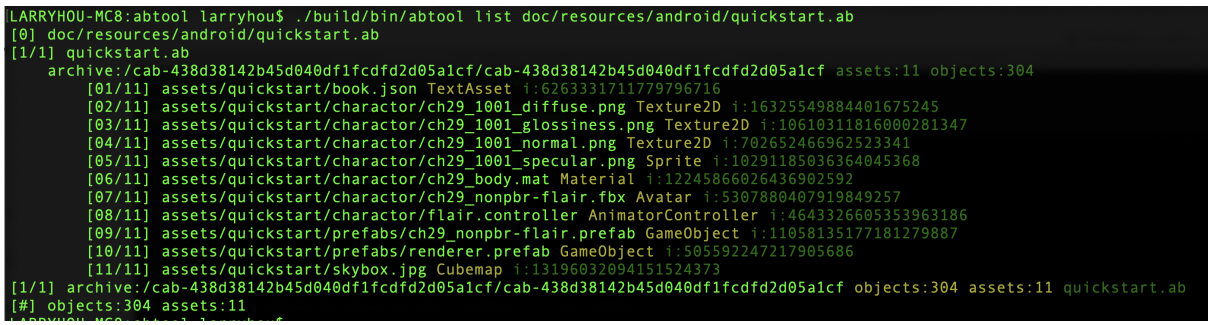
当然，这个演示的目的不是教大家hack别人的游戏，由于abtool的这个能力，我们可以在不用构建资源包的情况可以通过修改资源快速验证一些想法，比如测试发现资源bug，可以通过abtool快速修复验证，当然前提是你对abtool有一定的熟悉。

第三章 命令详解

截止文档撰写日起已有20多个内置命令，它们均是在解决资源问题过程中逐渐增加和完善的，具有很强的实用性。

颜色高亮

大部分命令运行过程中输出到终端的日志都是有颜色样式的，这个设计主要是根据信息的重要性做不同的高亮突出显示，方便在日志里面找到有用的信息。当然，也强烈建议你把终端设置为黑色背景样式，不然颜色显示会比较奇怪，因为黑色背景为终端显示样式的调试环境。



然而，在有些情况下，我们需要对工具输出的日志做进一步分析，这个时候我们是不希望有颜色高亮的，因为这些颜色都是通过颜色控制符[1]实现的，这会让日志里面多出一些方括号`[`的字符，如下图显示看起来比较杂乱，有可能会让下游的分析工具产生不符合预期的结果。

```
[0] doc/resources/android/quickstart.ab
[1/1] quickstart.ab
    archive:/cab-438d38142b45d040df1fcdfd2d05a1cf/cab-438d38142b45d040df1fcdfd2d05a1cf [2m assets:11 objects:304 [0m
        [01/11] assets/quickstart/book.json  [33mTextAsset [0m [2m i:6263331711779796716 [0m
        [02/11] assets/quickstart/charactor/ch29_1001_diffuse.png  [33mTexture2D [0m [2m i:16325549884401675245 [0m
        [03/11] assets/quickstart/charactor/ch29_1001_glossiness.png  [33mTexture2D [0m [2m i:10610311816000281347 [0m
        [04/11] assets/quickstart/charactor/ch29_1001_normal.png  [33mTexture2D [0m [2m i:702652466962523341 [0m
        [05/11] assets/quickstart/charactor/ch29_1001_specular.png  [33mSprite [0m [2m i:10291185036364045368 [0m
        [06/11] assets/quickstart/charactor/ch29_body.mat  [33mMaterial [0m [2m i:12245866026436902592 [0m
        [07/11] assets/quickstart/charactor/ch29_nonpbr-flair.fbx  [33mAvatar [0m [2m i:5307880407919849257 [0m
        [08/11] assets/quickstart/charactor/flair.controller  [33mAnimatorController [0m [2m i:4643326605353963186 [0m
        [09/11] assets/quickstart/prefabs/ch29_nonpbr-flair.prefab  [33mGameObject [0m [2m i:11058135177181279887 [0m
        [10/11] assets/quickstart/prefabs/renderer.prefab  [33mGameObject [0m [2m i:505592247217905686 [0m
        [11/11] assets/quickstart/skybox.jpg  [33mCubemap [0m [2m i:13196032094151524373 [0m
[1/1] archive:/cab-438d38142b45d040df1fcdfd2d05a1cf/cab-438d38142b45d040df1fcdfd2d05a1cf [33m objects:304 assets:11 [0m [2m quickstart.ab [0m
[#] objects:304 assets:11
```

不过工程根目录里的`nocolor.cpp`的小工具可以轻松去掉终端的颜色样式，该工具含代码格式只有26行C++代码，非常轻量高效。

```
#include <iostream>
#include <string>

int main( int argc, char* argv[])
{
    std::string pipe;
    while (std::getline(std::cin, pipe))
    {
        auto cursor = pipe.begin();
        for (auto iter = pipe.begin(); iter != pipe.end(); iter++)
        {
            if (*iter == '\e' && *(iter+1) == '[')
            {
                ++iter; // [
                ++iter; // d
                ++iter; // m
                if (*iter != 'm') { ++iter; }
                continue;
            }

            *cursor++ = *iter;
        }
        *cursor = 0;
        std::cout << pipe.data() << std::endl;
    }
    return 0;
}
```

可以通过如下终端命令快速编译。

```
clang++ -std=c++11 nocolor.cpp -o/usr/local/bin/nocolor
```

使用起来也十分方便，只需命令末尾追加管道。

```
abtool list doc/resources/android/quickstart.ab | nocolor
```

```
LARRYHOU-MC8:abtool larryhou$ ./build/bin/abtool list doc/resources/android/quickstart.ab | nocolor
[0] doc/resources/android/quickstart.ab
[1/1] quickstart.ab
    archive:/cab-438d38142b45d040df1fcdfd2d05a1cf/cab-438d38142b45d040df1fcdfd2d05a1cf assets:11 objects:304
        [01/11] assets/quickstart/book.json TextAsset i:6263331711779796716
        [02/11] assets/quickstart/charactor/ch29_1001_diffuse.png Texture2D i:16325549884401675245
        [03/11] assets/quickstart/charactor/ch29_1001_glossiness.png Texture2D i:10610311816000281347
        [04/11] assets/quickstart/charactor/ch29_1001_normal.png Texture2D i:702652466962523341
        [05/11] assets/quickstart/charactor/ch29_1001_specular.png Sprite i:10291185036364045368
        [06/11] assets/quickstart/charactor/ch29_body.mat Material i:12245866026436902592
        [07/11] assets/quickstart/charactor/ch29_nonpbr-flair.fbx Avatar i:5307880407919849257
        [08/11] assets/quickstart/charactor/flair.controller AnimatorController i:4643326605353963186
        [09/11] assets/quickstart/prefabs/ch29_nonpbr-flair.prefab GameObject i:11058135177181279887
        [10/11] assets/quickstart/prefabs/renderer.prefab GameObject i:505592247217905686
        [11/11] assets/quickstart/skybox.jpg Cubemap i:13196032094151524373
[1/1] archive:/cab-438d38142b45d040df1fcdfd2d05a1cf/cab-438d38142b45d040df1fcdfd2d05a1cf objects:304 assets:11 quickstart.ab
[#] objects:304 assets:11
```

帮助系统

abtool内置了简单的帮助系统，可以帮助我们快速了解命令功能以及参数，比如可以通过 `abtool help` 查看所有命令。

```
$ abtool help
abtool COMMAND file ...
Commands:
  abname       :  通过SF路径从*.obj文件解析ab名字
  cmpmono      :  比对mono命令生成的*.ms文件并生成脚本变更报告
  cmpref       :  对比saveobj命令生成的*.obj文件并分析对象引用
  cmpxtl       :  比对external命令生成的*.xtl文件并生成ab依赖变更报告
  dump         :  生成文本格式的对象数据
  edit         :  进入交互编辑模式
  external     :  收集AB外部依赖信息
  gencpp       :  从AB文件提取TypeTree并生成C++序列化代码
  getref       :  获取资源引用
  gtt          :  从序列化的TypeTree文件生成C++代码
  help         :  获取帮助信息
  list         :  查看进包资源
  lua          :  运行LUA脚本
  missing      :  扫描资源引用丢失
  mono         :  扫描MonoScript脚本使用信息
  rename       :  根据AssetBundle::m_Name还原文件名
  resolve      :  通过SF路径以及m_PathID信息从*.obj文件获取实体资源信息
  rmtree       :  剔除TypeTree信息
  savefbx      :  扫描Mesh资源并保存为*.fbx文件
  saveobj      :  保存对象数据
  saveta       :  保存TextAsset资源
  savetex      :  保存Texture2D资源
  savetree     :  保存TypeTree二进制数据
  scanref      :  检查资源引用的对象类型是否匹配
  scantex      :  扫描非标准格式的贴图
  size         :  生成对象大小简报
  test         :  序列化正确性测试
  textize      :  文本化序列化文件并保存文件
```

abtool命令采用了一致的参数传参设计，可以通过 `abtool [command] --help` 查看参数含义，

```
$ abtool savefbx --help
  -r  --axis-rotate-enabled  rotate model 90 degrees by axis-X counter-clockwise [FLAG]
  -o  --output      *.fbx output path [OPTIONAL](default=__fbx)
  -w  --rewritable  rewrite local file if it exists [FLAG]
  -s  --skeleton-enabled  include skeleton info in *.fbx [FLAG]
```

- `[FLAG]` 表示当前参数为功能开关并且参数没有参数值
- `[OPTIONAL]` 表示当前参数为可选参数并有默认值，例如 `default=__fbx`
- 其他类型的参数为必选参数，也即是参数名和参数值必须设置正确，否则会运行崩溃

在本章节只会对一些常用的命令做详细的说明，但不代表abtool只有这些内置命令。

1. https://misc.flogisoft.com/bash/tip_colors_and_formatting ↩

savetree

gtt

dump

list

size

scanref

scantex

savefbx

savetex

saveta

saveobj

getref

cmpref

mono

cmpmono

external

cmpxtl

rename

rmtree

lua

edit

第四章 进阶开发

项目架构

文件解析

对象序列化

命令系统

文件系统

LUA绑定