**一、背景简介**

NeoX引擎一直没有一个统一的打包工具，每个项目都是根据自身的需求从零开始搭建打包流程，而这个工作至少要花费程序好几周的时间，那么一套通用的打包流程则显得十分重要。

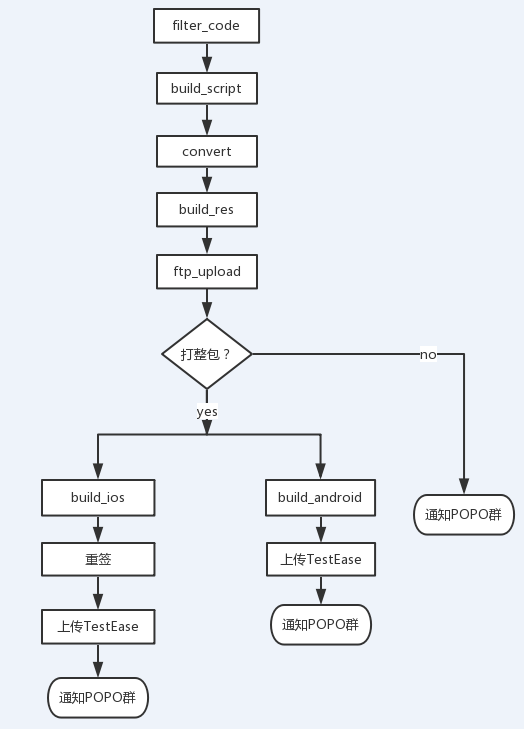
G78项目采用的也是NeoX引擎，在使用最初的打包工具时，打包时间长、结构复杂、扩展性差等问题饱受诟病。为此，我们重构了一套通用的打包流程，本文主要对新版工具的部分特性进行了描述。

**二、工具特性**

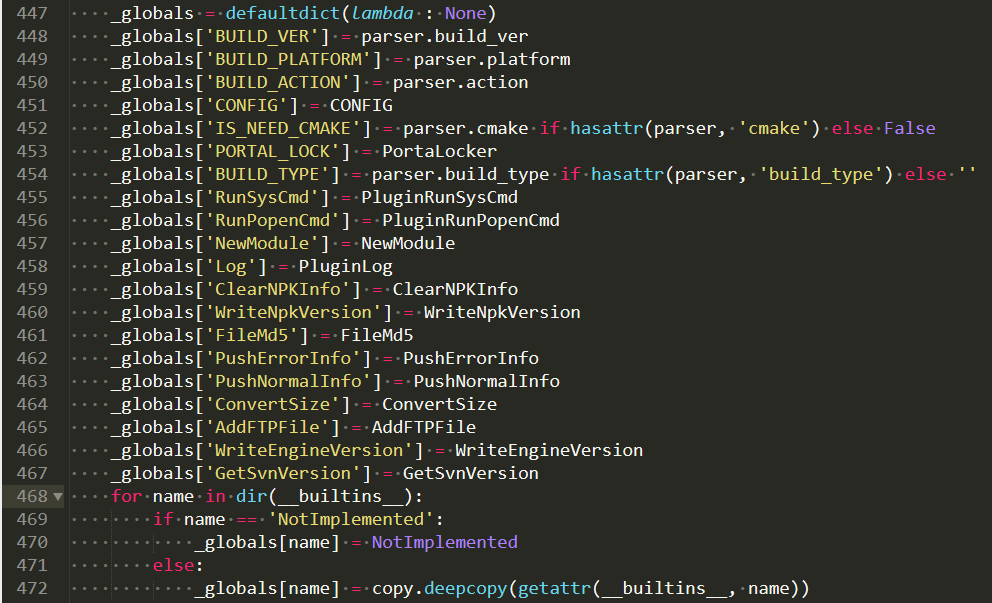
**1、插件化**

针对复杂的结构，行之有效的方案就是插件化，将其各个模块解耦合。在整个打包主流程当中，根据实际需求使用不同的插件。具体的，我们从主流程拆分出以下七个插件：

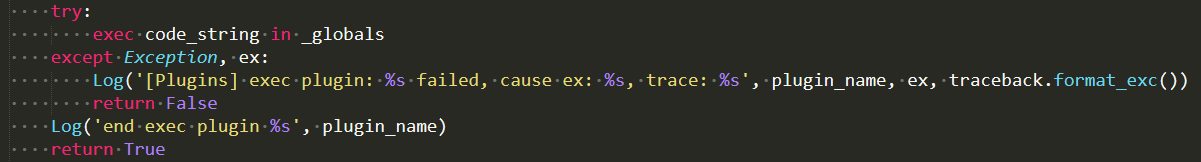
- plugins 打包插件目录  
 -- build\_android.py 构建Android整包  
 -- build\_ios.py 构建iOS整包  
 -- build\_res.py 打包资源npk  
 -- build\_script.py 打包脚本npk  
 -- convert.py 转贴图  
 -- filter\_code.py 过滤代码，以及替换 both.consts 等  
 -- ftp\_upload.py patch上传ftp



插件中会使用到各种环境变量，以及通用的方法。为了统一提供并且便于调用，考虑直接将这些变量放入插件的globals中，插件执行前会初始化一个globals的字典：



插件执行时，直接读取插件代码执行，而非通过import来实现，保证了环境的统一性。

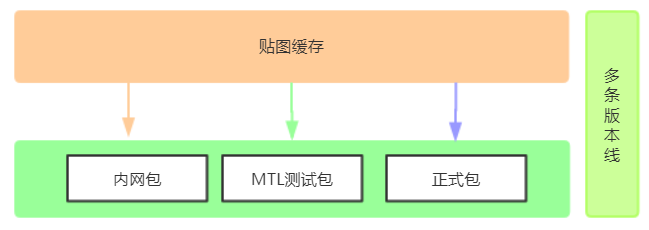


**2、独立于项目之外**

在以往的NeoX项目当中，打包脚本通常是夹杂在项目代码之中的，所有的配置项都和单个项目强相关，如果想要另开一条版本线便十分麻烦。另一方面，打手机包的一个开销大头是贴图转换，一般情况下，我们需要多条版本线，用来打出不同的包体比如内网包、MTL测试包、多语言测试包等，那么每次打包都要重新转换贴图，耗费大量的时间。

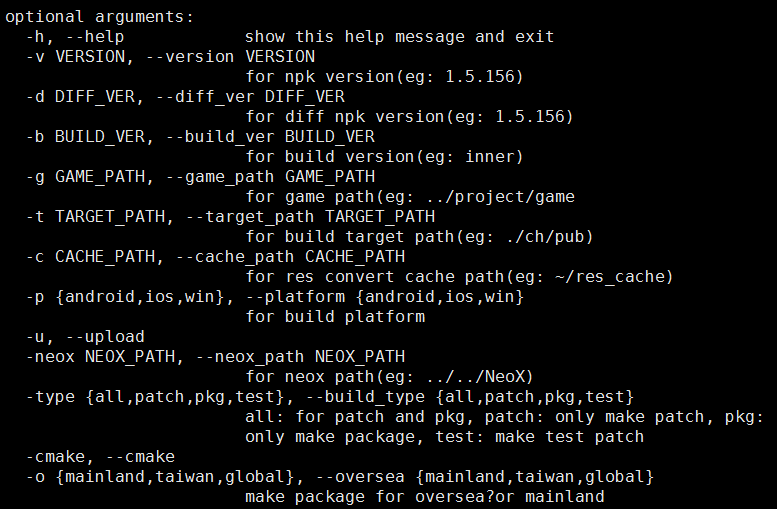
因此，我们将打包工具从项目中分离出来，同时建立一个贴图缓存资源共享池，然后将打包工具指定的贴图缓存路径设为一致，让各个版本线共用一份贴图缓存。这样只要一个版本线打过包，另一条版本线能够直接使用其生成的贴图缓存，提升打包效率。

为了防止由于多个版本线间贴图资源的SVN版本不一致导致的覆盖问题，我们将转换后的贴图按照转换之前的md5进行命名。同时，也使用了文件锁，防止两个版本线打包同时操作一份贴图时，引发文件写入异常。



**3、参数多样化**

在旧版的打包工具当中，参数是写死在脚本里面的，比如引擎版本号的提升、打整包还是打patch、要不要上传至发布系统、是否需要cmake等。每次修改配置文件都需要程序改代码，版本一多就会变得非常麻烦。所以，为了灵活使用打包工具，我们将可配置内容通过传参的方式实现，减少对代码的依赖。



如果使用jenkins进行打包的话，可以为各个任务定制不同的参数，以满足不同场景下的构建需求。



**4、patch生成速度优化**

G78采用的是增量patch方案，也就是说每个patch的内容是通过两个版本之间的diff生成的。以往是采用NeoX传统的patch生成方式：先拷贝全量的资源和脚本文件，将其打包成npk，然后利用同步生成的npk.map文件和上一个版本作对比，筛选出要patch的文件。这样每次的全量拷贝步骤占用了较多时间，使得patch生成速度过慢，平均打一次patch需要耗费20分钟。

为了解决这个问题，我们对此步骤进行了优化。首先遍历所有文件，并将它们对应的md5记录下来，然后将其与上一个版本的md5列表进行对比，筛选出有变化的文件，接着生成打包成npk。这样一来，只需要拷贝需要patch的文件，提高了打patch的效率。通过这样的改进，把每次打patch的平均时间从30分钟降到5分钟之内。

**5、持续可集成**

在整个工具扩展性强、高度自由的定制化基础上，我们可以很方便地优化打包流程。

a）版本线切换

很多项目都会有多条svn分支，分别负责不同内容的开发工作，比如周版本维护和每日线上热更的分支。当需要切换分支进行打patch时，常见的做法是手动在版本机上执行svn switch操作，这样既费时费力又容易出差错。此时，我们可以建立两个jenkins任务，通过参数指定不同的代码路径，然后根据需要执行对应的任务出patch即可。



b）预打patch测试

公司的大部分项目是以周版本为周期进行开发的，而一般周版本要外放的内容会比较多，可能会打出几十兆的patch。为了准确知道此次patch的具体内容，我们可以通过此工具预先打出测试patch，然后根据详细的文件信息和大小，来评估patch内容的增删，比如美术资源太大了，某些文件没必要外放等。当把patch内容重新调整完毕之后，再次打正式patch，保证最终版本的正确性。



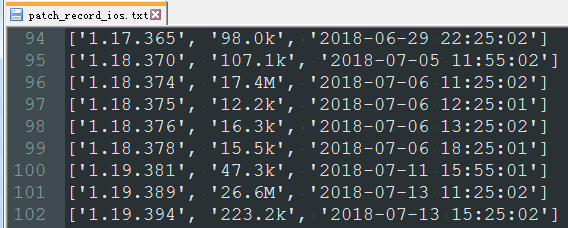
c）patch信息查看

当项目的包体变多，需要维护的版本线有好几个时，为了防止遗漏，可以将每个patch的基本信息打印出来。并且，通过阿拉丁将patch外放之后，再次执行此操作，确保patch均已外放成功。另外，有些开发中的功能，需要通过patch号进行版本控制，我们也能够轻易地获取到外网玩家的版本号信息。



d）patch外放记录

通常来说，patch列表的外放是通过阿拉丁操作的，如果每次操作，没有进行手工记录的话，很难通过历史记录找到历史patch的外放信息。那么，借助jenkins定时跑检测脚本，能够看到历史patch外放详情，便于获取一些数据上的统计信息。



**三、成果展示**

**1、版本维护时间缩短**

在使用旧版打包工具进行时，每次周版本的维护节点经常卡在打patch上，即使是一个脚本的修改，重新打一个patch也要花上半个小时的时间，安卓和iOS两个版本便要耗费一个小时，这在版本日期间太过影响效率。而新版打包工具则把时间控制在5分钟之内，使得打patch不再是版本日的卡点。

**2、便于移植**

把打包脚本独立于项目代码，提供足够多样化的参数，使其定制性更高，减少了对配置文件的依赖。在G78海外版的项目中，我们直接移植了优化后的打包工具，只需要将项目代号、指定路径和证书等进行修改，就能够快速搭建好新的版本线。另一方面，在同一项目当中，由于可以共用一份贴图缓存，再想要构建一条新的版本线就变得更加快捷。

**3、为各职位提供便利**

程序很便捷地获取到外网最新的patch信息，然后根据patch信息编写相关的版本控制逻辑。通过每日构建内网包体进行DogFood测试，组织各个职位同学对战，为策划体验游戏、调整数值提供良好的环境测试。为美术开发分支单独搭建版本线，供美术同学实时查看资源在手机包中的表现。