JNI开发的那些事

刘文斌

纲要

- 简介
- JNI开发的几个坑和结论
- JNI开发的几个参考
- 应用实例

简介


Applications
Home
Contacts
Phone
Browser
...
Application Framework
Activity Manager
Window Manager
Content Providers
View System
Notification Manager
Package Manager
Telephony Manager
Resource Manager
Location Manager
GTalk Service
Libraries
Surface Manager
Media Framework
SQLite
OpenGL | ES
FreeType
WebKit
SGL
SSL
libc
Android Runtime
Core Libraries
Dalvik Virtual Machine
Linux Kernel
Display Driver
Camera Driver
Bluetooth Driver
Flash Memory Driver
Binder (IPC) Driver
USB Driver
Keypad Driver
WiFi Driver
Audio Drivers
Power Management

- JNI是什么?
    - Java Native Interface
    - 一个标准，约定，保证了本地代码能工作在任何Java虚拟机环境
- 所在层级
    - application与运行时库之间
    - 进一步的，application/framework 与 私有/系统运行时库之间
- 为什么要JNI？(业务由Java来处理，功能实现由原生语言，为什么还要JNI？)
    - 过渡。Java到Native的承上启下
    - 扩展。为接入由原生语言(C/C++)编写的第三方库提供可能性，丰富能力
    - 兼容。支持多种厂商实现的Java虚拟机，比如Dalivk、ART、Sun JVM、JRocket、J9,Harmony。由“指针”的jni api形态决定的。
- 如何做好JNI开发?
    - 正确性
        - 对象管理
        - 编码方式
    - 效率
        - 编码方式的选择
        - 内存分配的选择

JNI开发的几个坑

- 对象管理
  - 对象缓存。如何缓存一个本地对象？为什么是long?
  - 生命周期。Java对象和本地对象都有各自生命周期，如何正确的创建、销毁并关联?
- 编码方式
  - JNI支持UTF-8、UTF-16，传递字符串选择哪种编码方式?
  - M-UTF8是什么？为什么JNI有些方法要依赖M-UTF8?
- 效率
  - Java堆和Native堆，内存分配时该如何选择?

数据类型(字节)	32位 C	64位 C	32位 Java	64位 Java
byte	1	1	1	1
char	2	2	2	2
int	4	4	4	4
long	4	8	8	8
float	4	4	4	4
double	8	8	8	8
void*	4	8	-	-

C和java中基本数据类型的所占字节数对比

如何缓存一个本地对象?
为什么是long?

- 本地对象是由“指针”标识，本质是所在内存地址
- 32bit、64bit机器上，指针所需的大小不同
- 若使用Java Int在64位环境下，会丢失部分地址

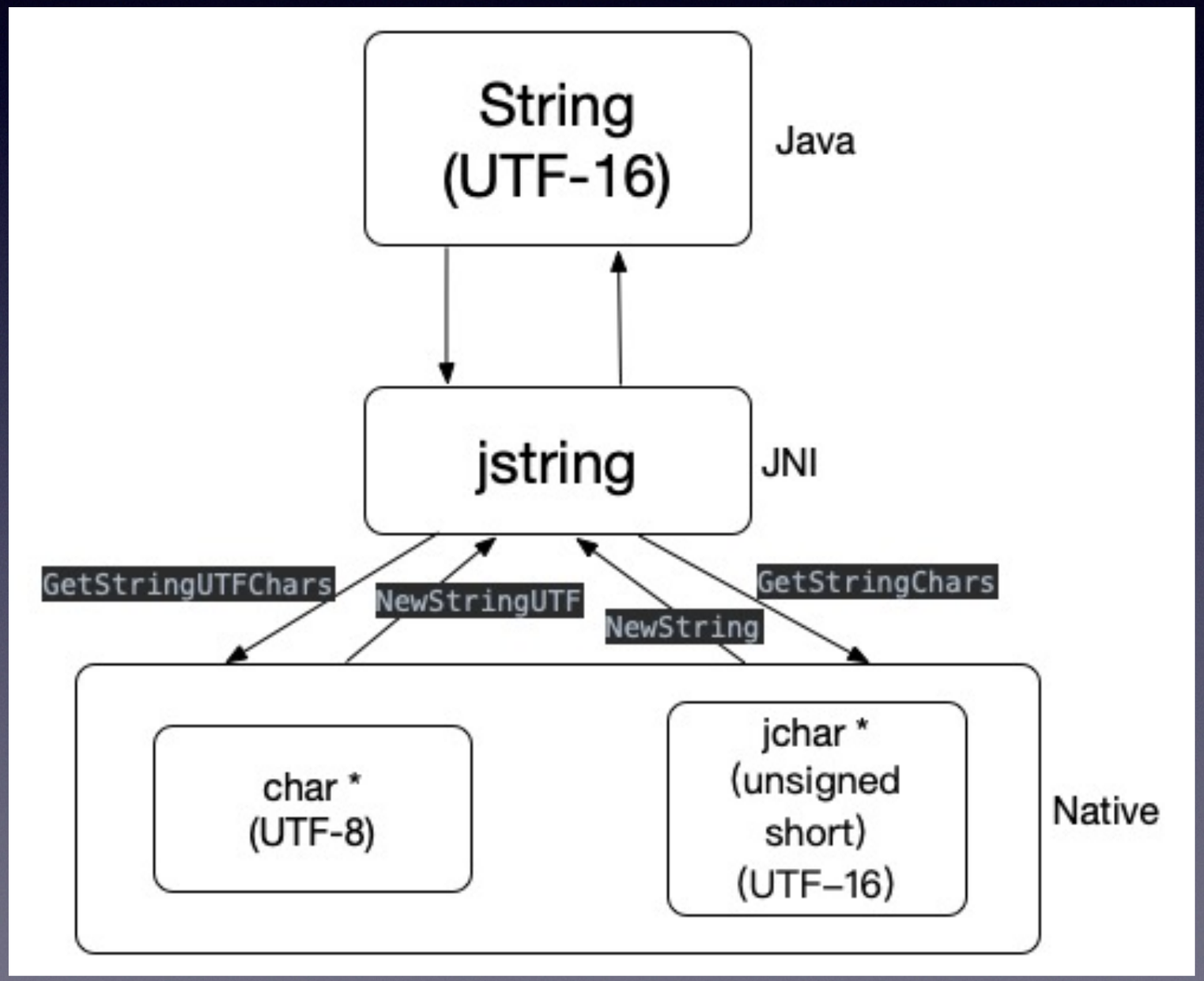
Java对象和本地对象都有各自生命周期，如何正确的创建、销毁并关联?

- Java对象会被JVM自动回收，本地对象不会，需要手动释放
- 将本地对象和Java对象生命周期关联在一起。具体做法：
  - 创建：在构造函数中 new 本地对象
  - 销毁：finalize 中 delete本地对象

Modified-UTF8是什么？为什么JNI有些方法要依赖M-UTF8

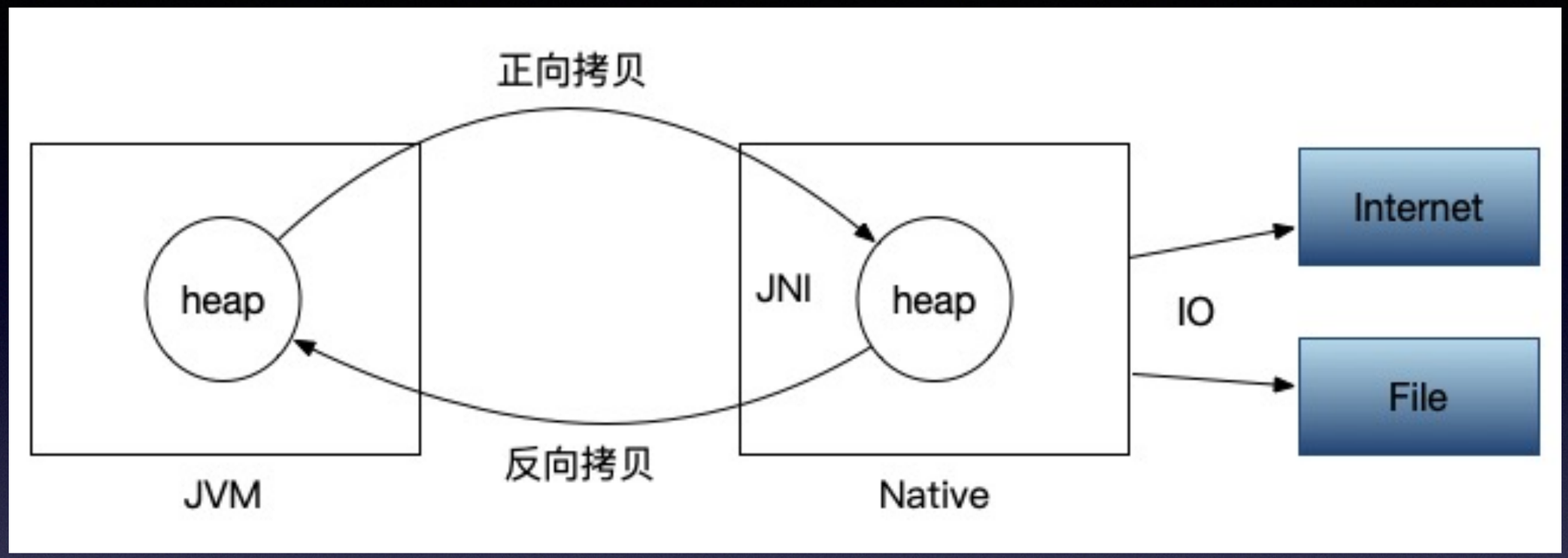
- 参考“MUTF8_Test”
- 比如：GetStringUTFChars输出M-UTF8数据编码；NewStringUTF输入必须为M-UTF8
- 说明：“将 空字符(\u0000)编码为 0xc0 0x80，而不是 0x00”
- 好处：方便与标准 libc 字符串函数配合使用。ex. strlen、strcpy、strcmp等str*方法。允许携带空字符(0x00)的String被字符串函数处理、使用。
- 缺点：无法将任意 UTF-8 数据传递给 JNI 并期望它能够正常工作。ex. 传给NewStringUTF的数据必须是M-UTF8

JNI支持UTF-8、UTF-16，传递字符串选择哪种编码方式?



- 一般在边界处需要转码
- Java虚拟机与Native之间同样存在边界
- 转码需要分配新的内存空间和转换
- 在JNI，优先使用 UTF-16相关的API
- 与JVM内部编码方式一致，减少转码成UTF8的损耗(见encodeTypeSpeedTest)

Java堆和Native堆，分配数据缓存时，选择何种内存类型?



JNI拷贝示意图

- 若数据出口在Native侧，缓存分配在JavaHeap，需要拷贝一次；分配在NativeHeap，不需要拷贝
- 若数据出口在Java侧，缓存分配在JavaHeap，需要拷贝两次；分配在NativeHeap，需要拷贝一次

Java堆和Native堆，分配数据缓存时，选择何种内存类型?

- 特性
  - JNI拷贝存在代价(见：speed_Test)
  - 缺点：原生堆只支持字节数组
- 结论
  - 判断数据的最终出口，若要与外界通信，最后的处理者是Native，则分配到原生堆更好！可以避免一次Java堆到Native堆的内存拷贝！

一些其他注意点

- 反向调用
- JNI的异常处理
- JavaVM和JNIEnv

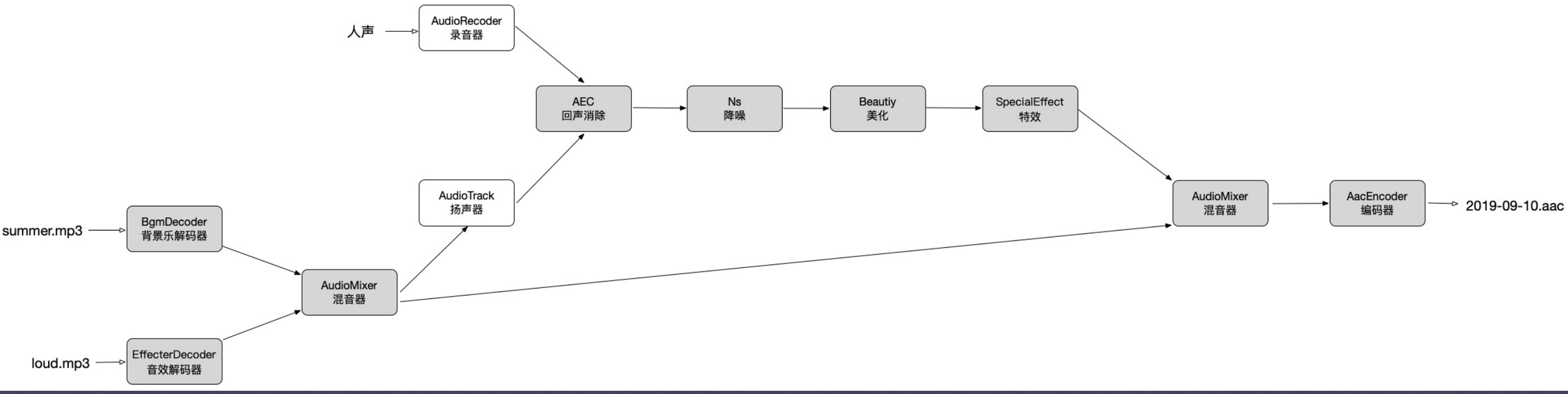
JNI开发的几个参考

- [JNI 提示]：要点说明，尤其结合了Android环境的特点
- [JNI编程指南]：78页，较基础，偏向Java开发，可以作为工具书查询

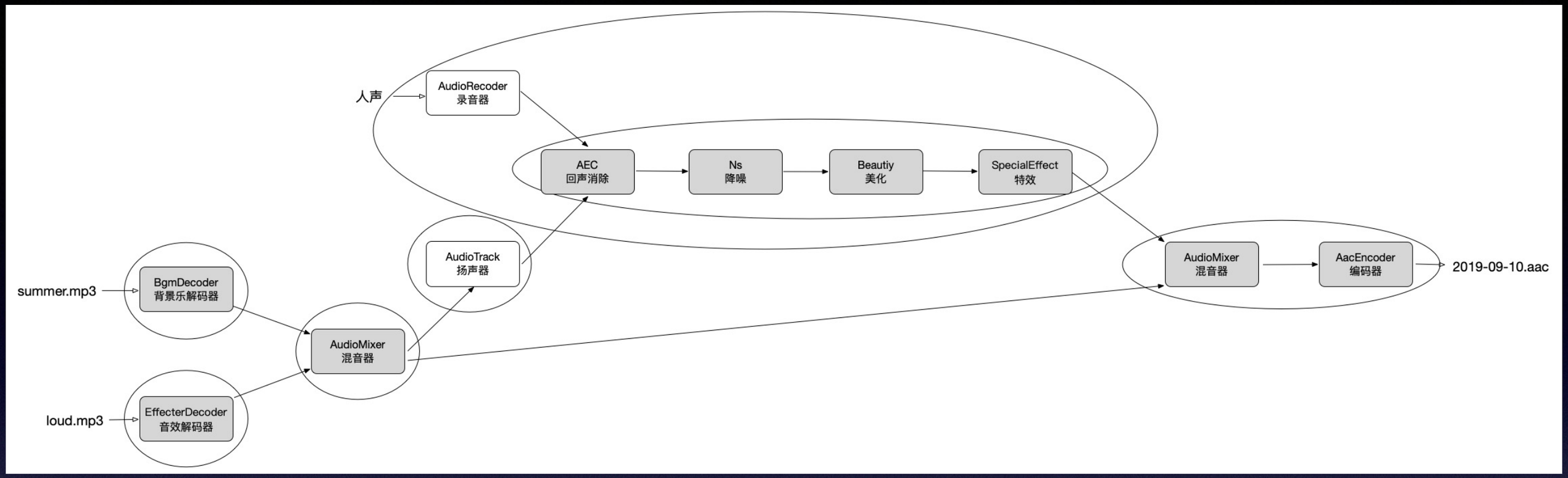
- [art/runtime/jni_internal.cc]:art实现，jni的api具体源码，比如“env->GetByteArrayElements”等

- [googlesamples/**android-ndk**]:官方实例，覆盖NDK具备的能力。比如音频方面，“native-audio”介绍了嵌入式高性能音频库“OpenSL ES”在NDK的使用示例，推荐

应用实例


人声
AudioRecoder
录音器
AEC
回声消除
Ns
降噪
Beautiy
美化
SpecialEffect
特效
AudioTrack
扬声器
AudioMixer
混音器
AacEncoder
编码器
2019-09-10.aac
summer.mp3
BgmDecoder
背景乐解码器
AudioMixer
混音器
loud.mp3
EffecterDecoder
音效解码器

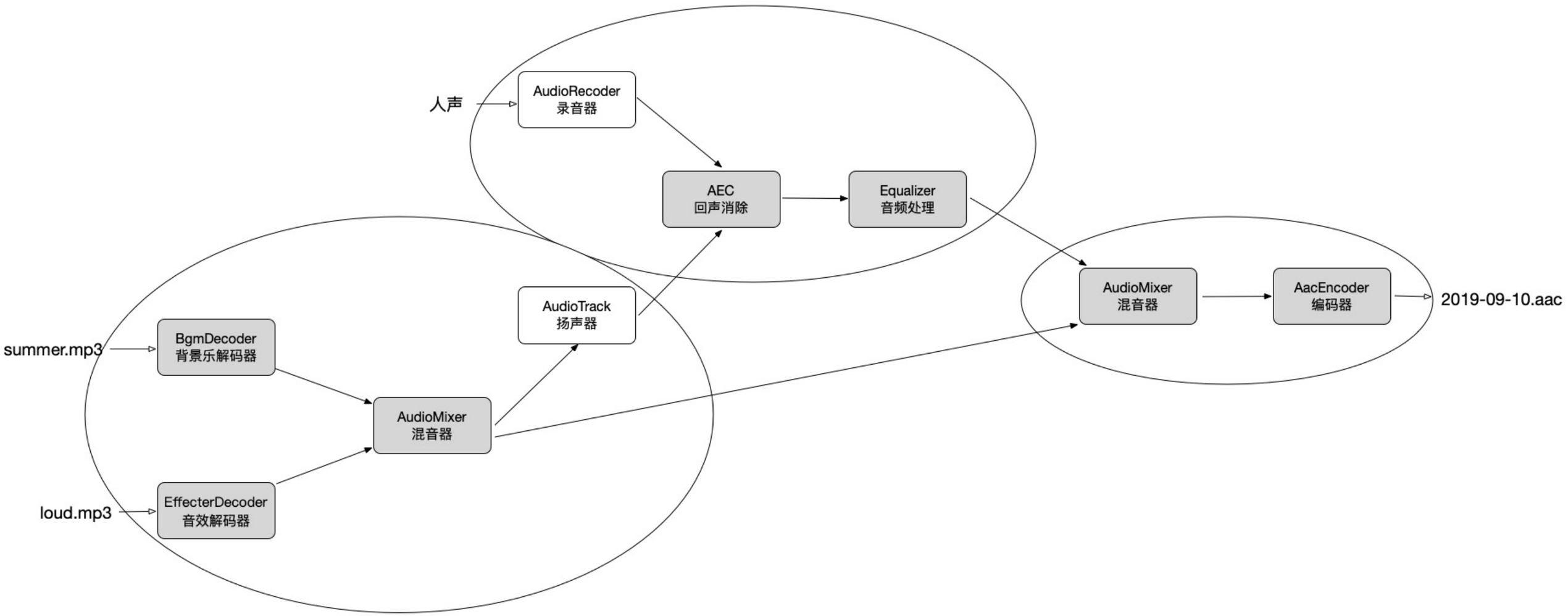
现有录音流程



现有录音实现

存在问题：

- 工作线程数过多，全开合计6~7个
- 声音处理模块较多，存在jni拷贝，但又是short[]封装，无法使用堆内存
- 生产者、消费者速度不一致，差别大，需要小心处理锁的占用、释放，缓冲区遗留声音数据



计划录音实现

改进点：

- 减少工作线程数，至3个
- 集中封装声音处理模块，减少jni拷贝，简化调用
- 线程数减少，线程间交互减少，缓冲区减少，简化处理

谢谢大家！