Android RunTime 介绍

软件所智能软件中心 PLCT 实验室 汪辰

2020/8/15

目 录

目 录

总体介绍

Android 开源项目

"Android Open-Source Project" 缩写 AOSP

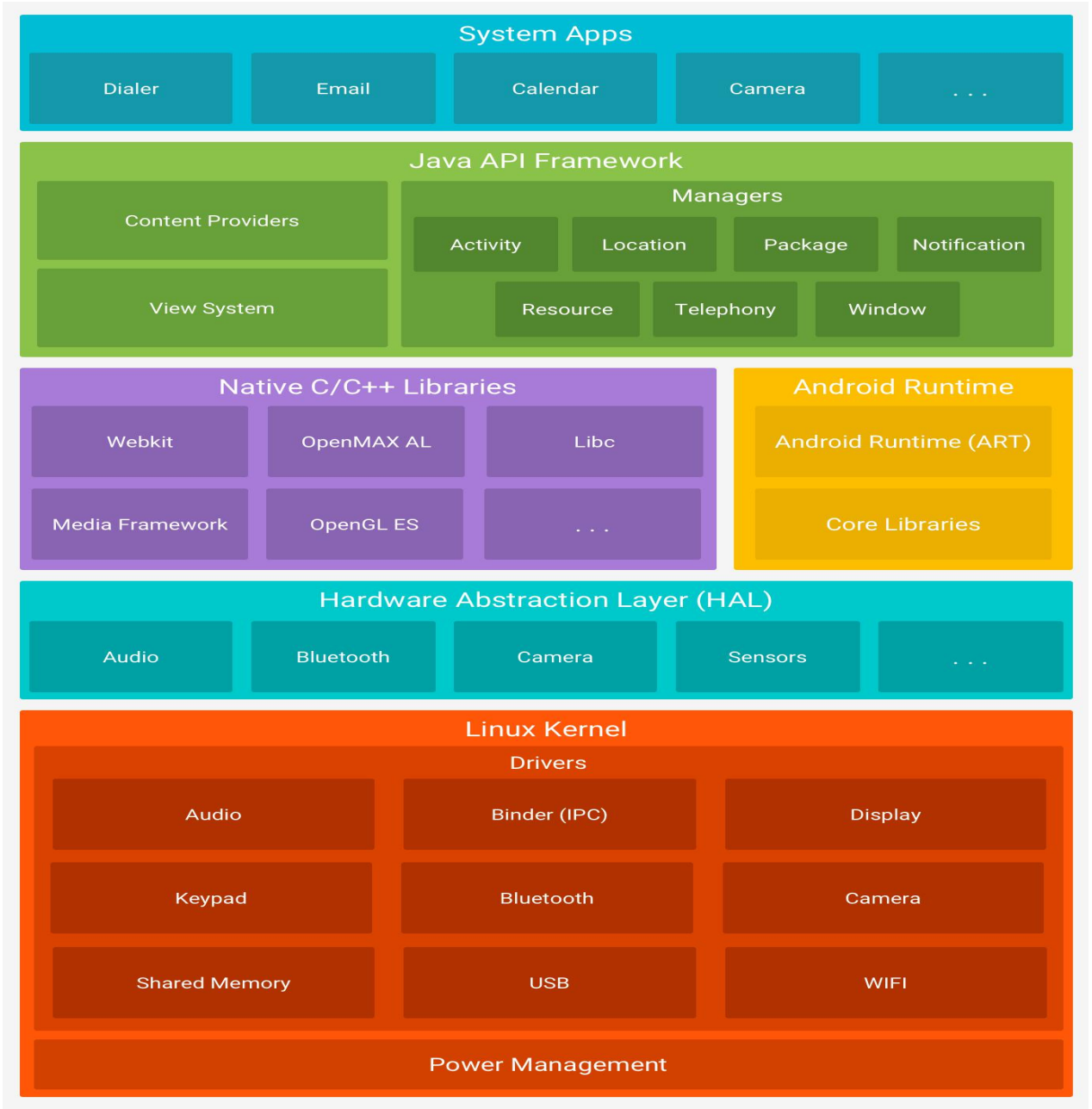
Android 是一个适用于移动设备的开源操作系统，
也是由 Google 主导的对应开源项目。

android 10

10.0.0_r39

总体介绍

Android 与 ART



总体介绍

虚拟机的定义

虚拟机（英语：virtual machine，简称 VM），在计算机科学中的体系结构里，是指一种特殊的软件，可以在计算机平台和终端用户之间创建一种环境，而终端用户则是基于虚拟机这个软件所创建的环境来操作其它软件。虚拟机（VM）是计算机系统的仿真器，通过软件模拟具有完整硬件系统功能的、运行在一个完全隔离环境中的完整计算机系统，能提供物理计算机的功能。

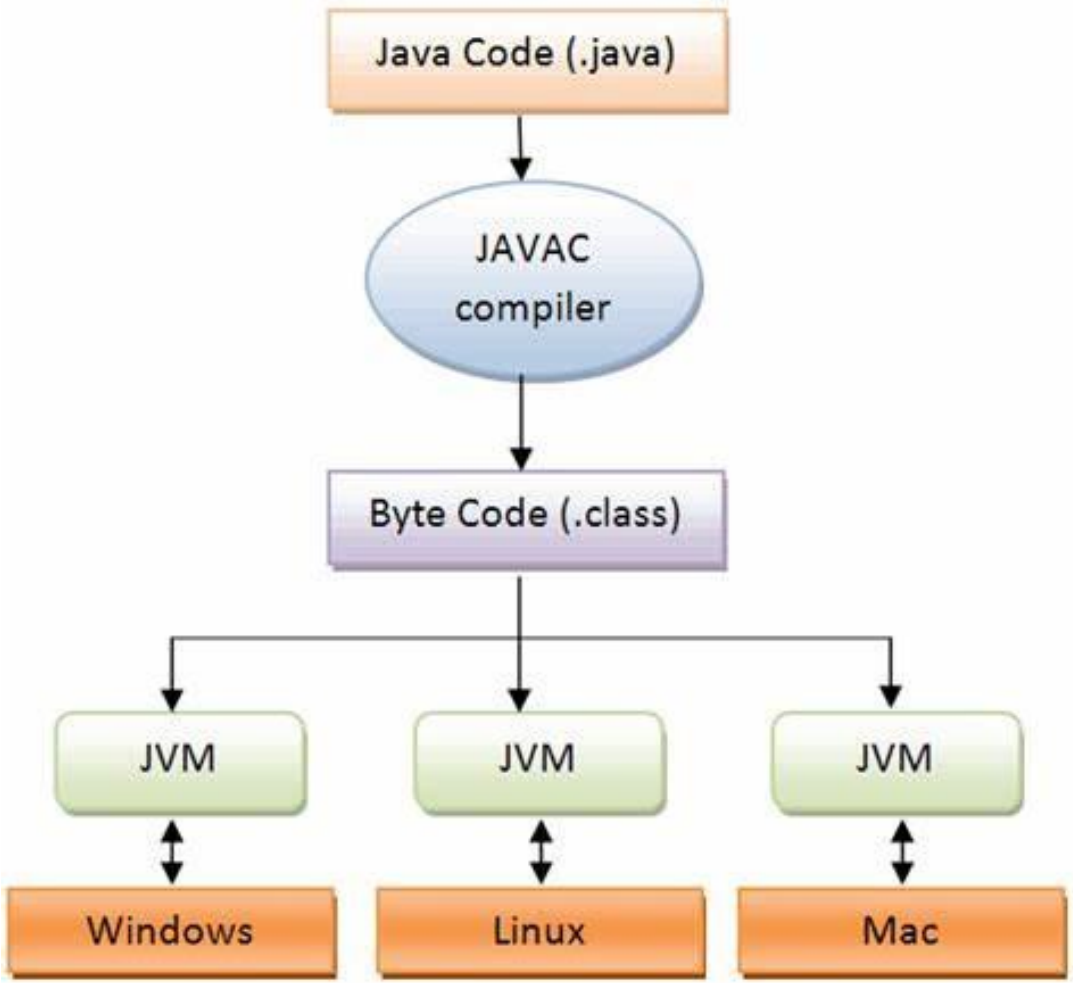
有不同种类的虚拟机，每种虚拟机具有不同的功能：

- 系统虚拟机（也称为全虚拟化虚拟机）可代替物理计算机。它提供了运行整个操作系统所需的功能。
- 程序虚拟机 被设计用来在与平台无关的环境中执行计算机程序。

总体介绍

Java VM

Java 虚拟机（英语：Java Virtual Machine，缩写为 JVM），一种能够运行 Java bytecode 的虚拟机，以堆栈结构机器来进行实做。最早由 Sun 微系统所研发并实现第一个实现版本，是 Java 平台的一部分，能够运行以 Java 语言写作的软件程序。

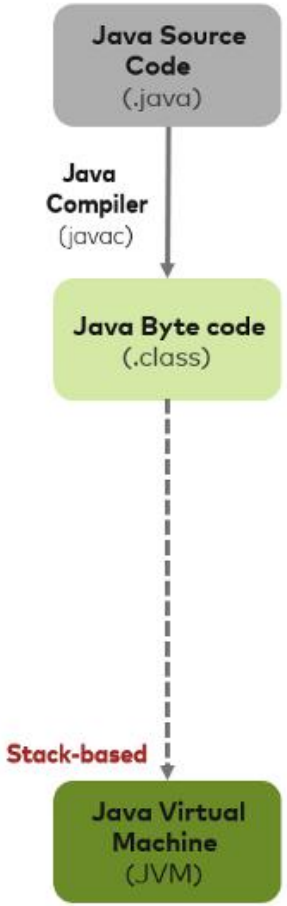


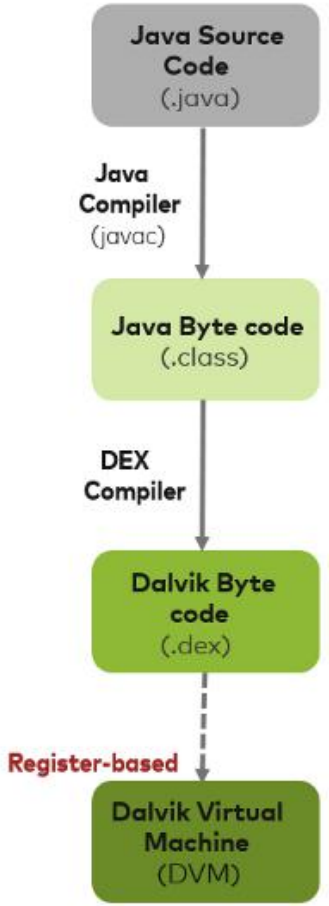
总体介绍

Delvik VM

Dalvik 虚拟机，是 Google 等厂商合作开发的 Android 移动设备平台的核心组成部分之一。它可以支持已转换为 .dex（即“Dalvik Executable”）格式的 Java 应用程序的运行。.dex 格式是专为 Dalvik 设计的一种压缩格式，适合内存和处理器速度有限的系统。

Dalvik由Dan Bornstein编写的，名字来源于他的祖先曾经居住过的小渔村达尔维克（Dalvík），位于冰岛埃亚峡湾。



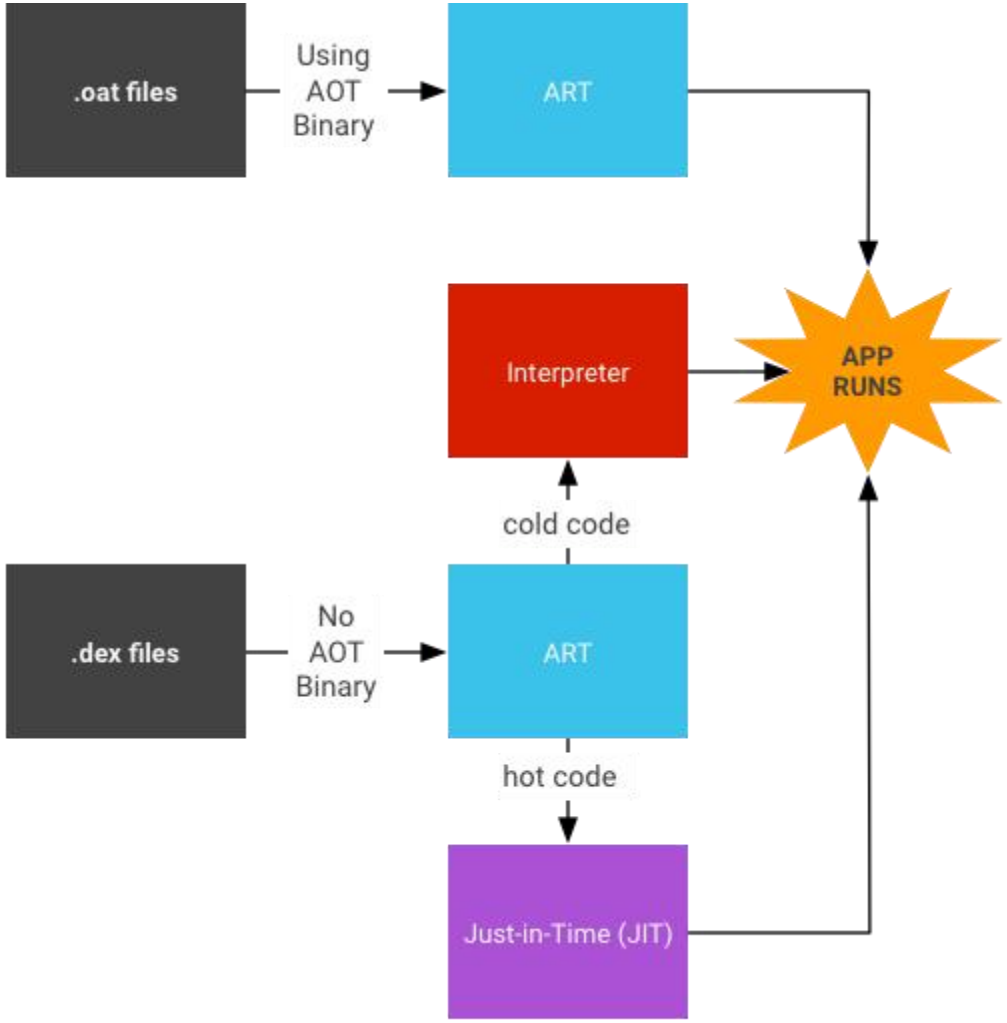


总体介绍

ART VM

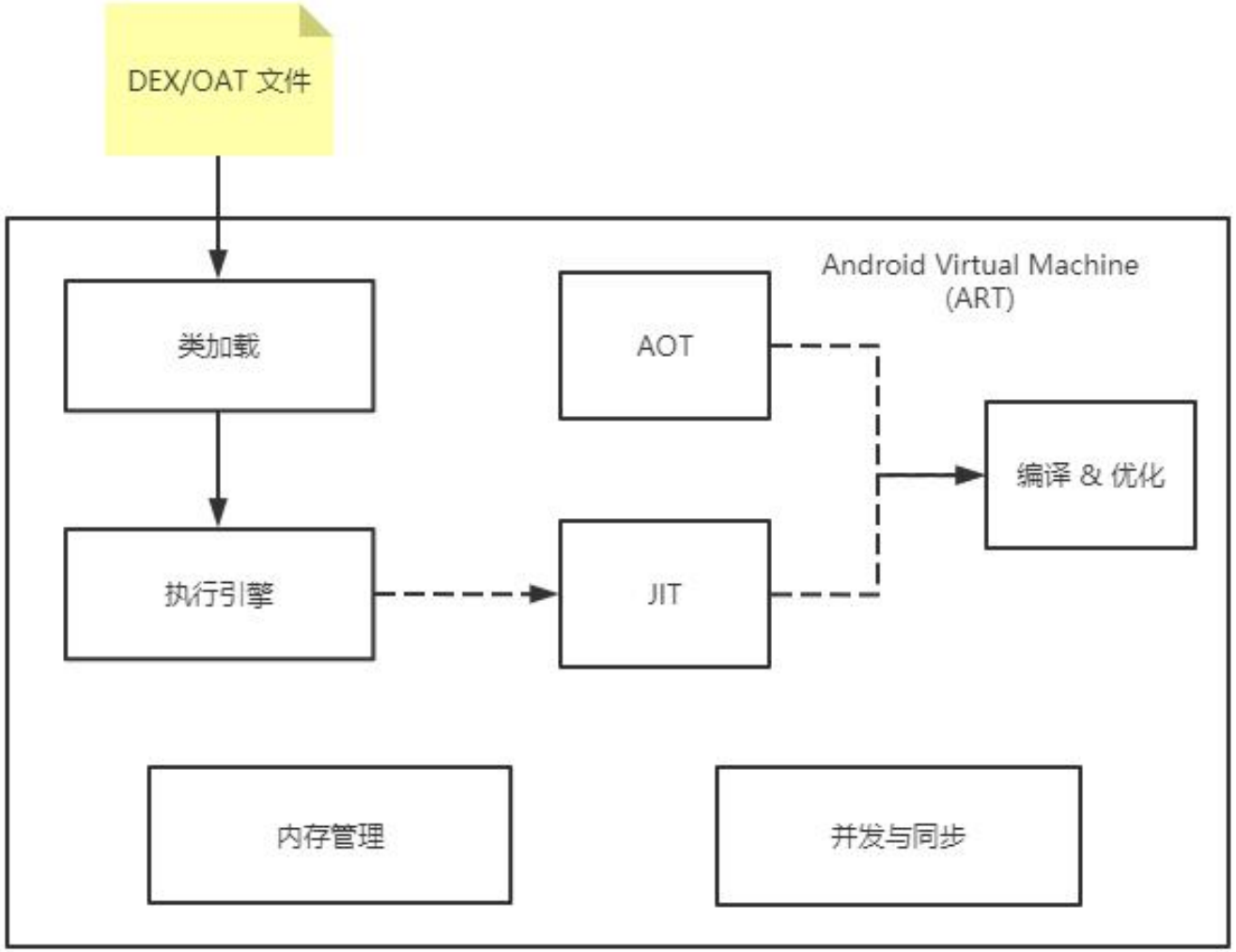
Android Runtime（缩写为ART），是一种在Android 操作系统上的运行环境，由 Google 公司研发，并在 2013 年作为 Android 4.4 系统中的一项测试功能正式对外发布，在 Android 5.0 及后续 Android 版本中作为正式的运行时库取代了以往的 Dalvik 虚拟机。

它与 Dalvik 的主要不同在于：Dalvik 主要采用的是 Just-In-Time（JIT） 技术，而 ART 则在此基础上引入 Ahead-of-Time（AOT）技术，基于 Profile 分析，综合使用 AOT 和 JIT；于此同时 ART 也改善了性能、垃圾回收（Garbage Collection）、应用程序出错以及性能分析。

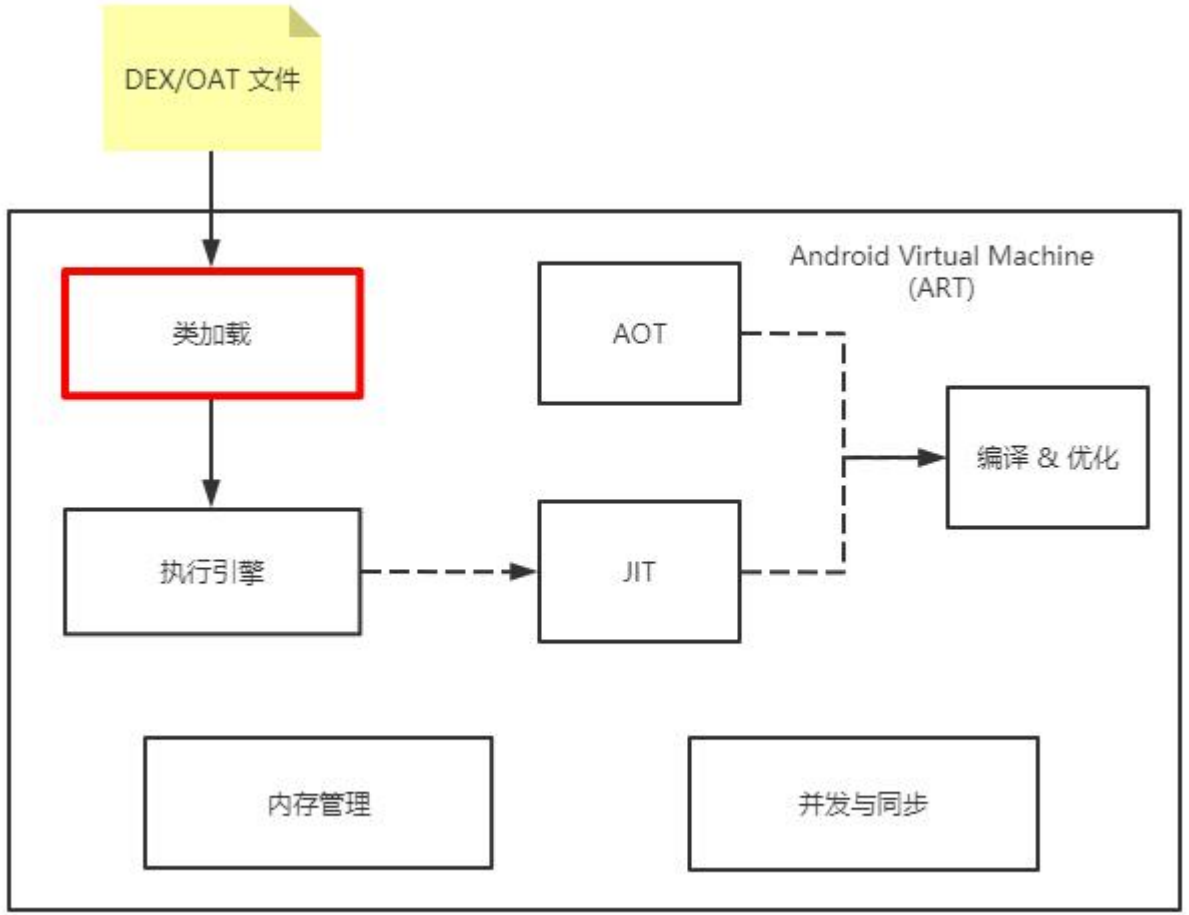


总体介绍

ART 组成总览



目 录



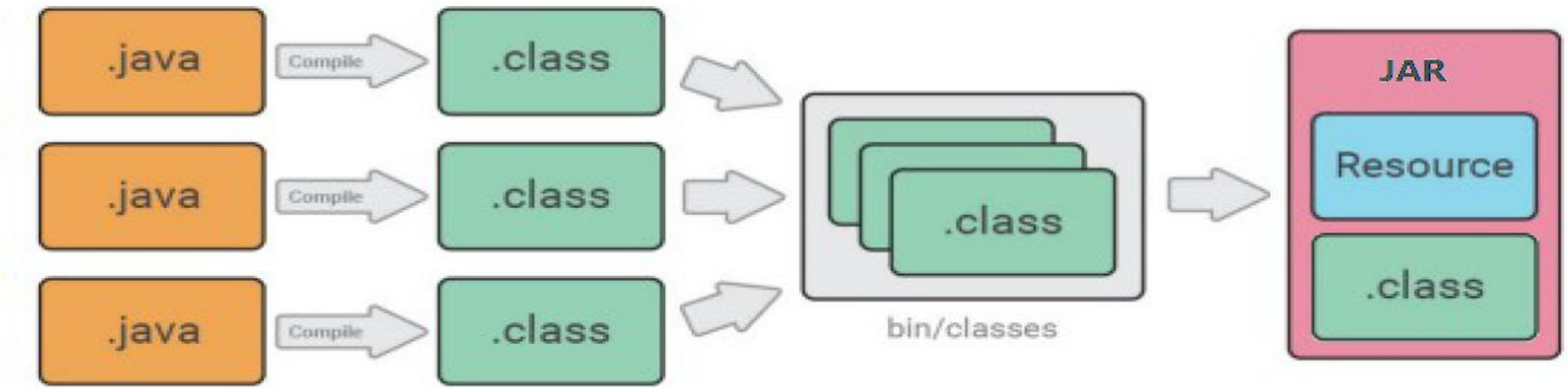
类加载

经典的 CLASS 文件格式

```
public class HelloWorld
{
    String str = "";

    public String getStr()
    {
        return str;
    }

    public void setStr(String str)
    {
        this.str = str;
    }
}
```

头	ClassFile {
	u4 magic;
	u2 minor_version;
	u2 major_version;
	u2 constant_pool_count;
	cp_info constant_pool[constant_pool_count-1];
	u2 access_flags;
	u2 this_class;
	u2 super_class;
	u2 interfaces_count;
	u2 interfaces[interfaces_count];
	u2 fields_count;
	field_info fields[fields_count];
	u2 methods_count;
	method_info methods[methods_count];
	u2 attributes_count;
	attribute_info attributes[attributes_count];
	}

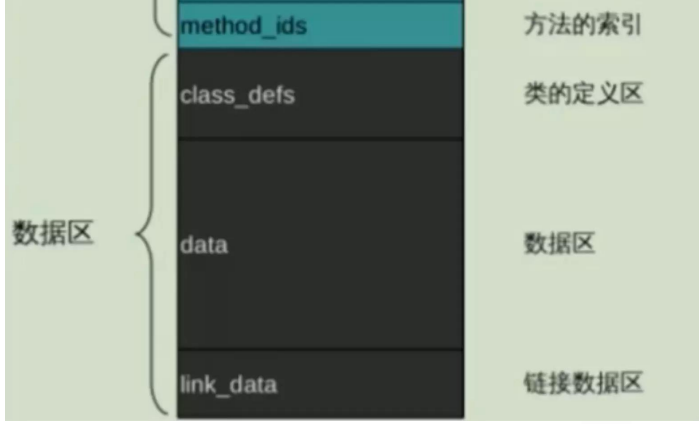
静态池（类似索引）	
访问控制	
类本身	
父类	
实现的接口	
属性	
方法	
元数据	

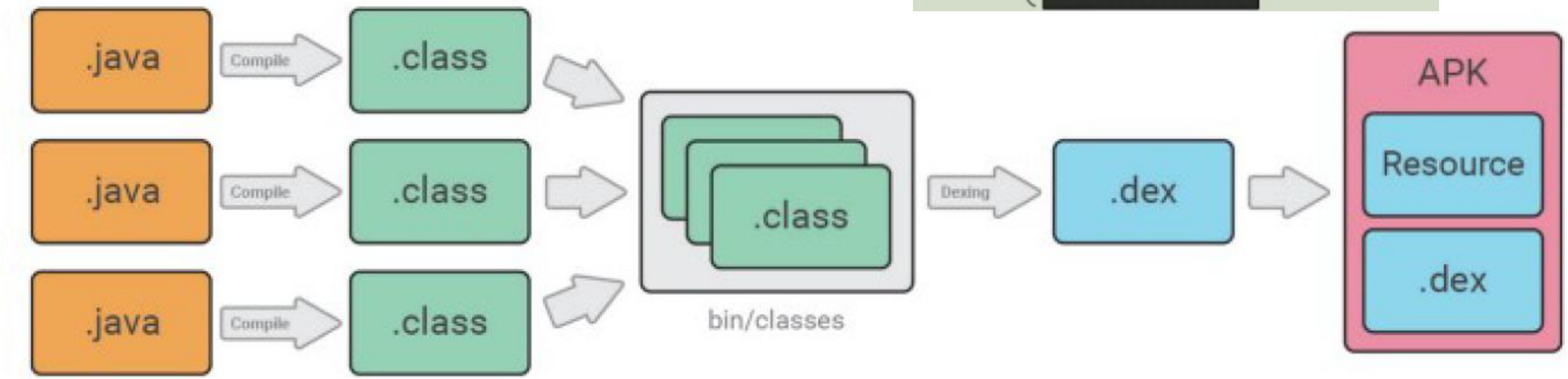


类加载

DEX 文件格式

```
public class HelloWorld
{
    String str = "";

    public String getStr()
    {
        return str;
    }

    public void setStr(String str)
    {
        this.str = str;
    }
}
```





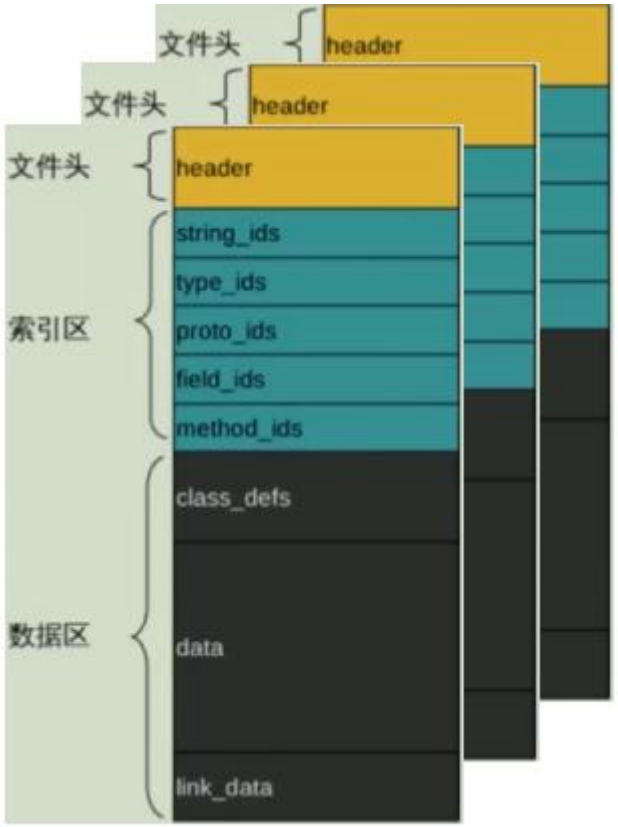
类加载

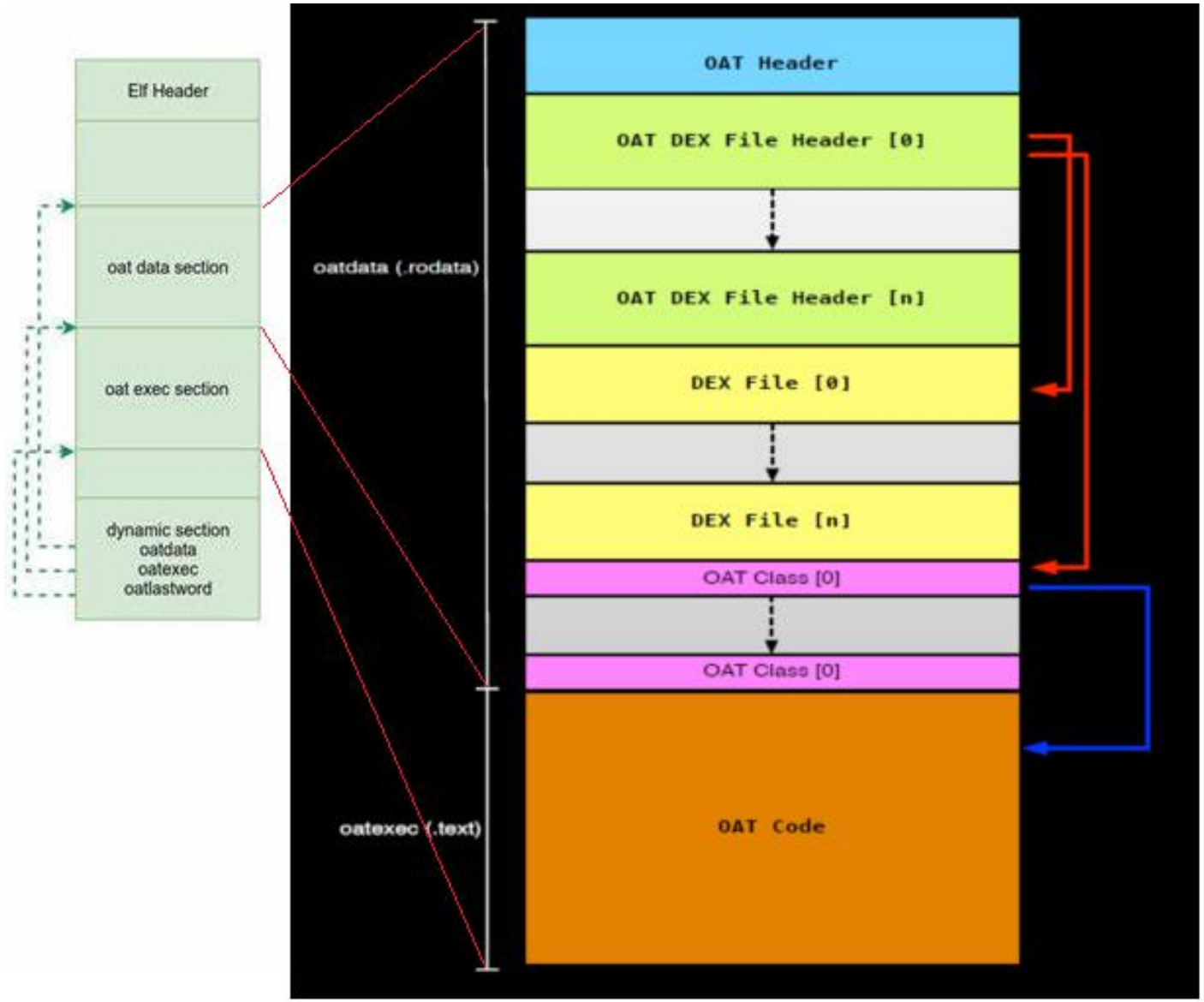
DEX v.s Class 主要区别

- 一个 Class 文件对应一个 Java 源文件；一个 Dex 文件对应多个 Java 源文件
- 字节序: Class 文件采用 Big Endian；Dex 文件默认采用 Little Endian
- 数据类型: Dex 新增可变长 LEB128 （Little Endian Based 128）格式
- 指令码：Dex 的操作码指令比 Class 中的更长，Dex 基于寄存器，Class 基于栈。

类加载

OAT 文件格式


文件头
header
文件头
header
文件头
header
string_ids
type_ids
proto_ids
field_ids
method_ids
索引区
class_defs
data
link_data
数据区

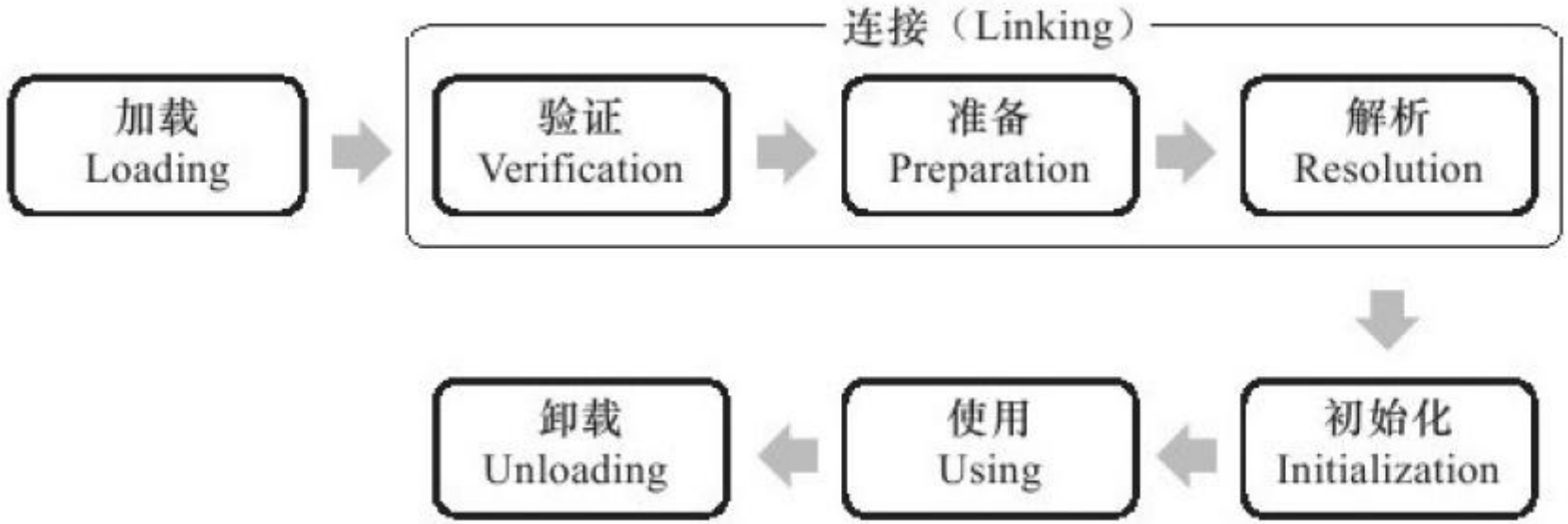

Elf Header
oat data section
oat exec section
dynamic section
oatdata
oatexec
oatlastword
oatdata (.rodata)
oatexec (.text)
OAT Header
OAT DEX File Header [0]
OAT DEX File Header [n]
DEX File [0]
DEX File [n]
OAT Class [0]
OAT Class [0]
OAT Code

类加载

类加载

定义：将 Class/Dex 字节流转化为 虚拟机内部的类结构的过程。

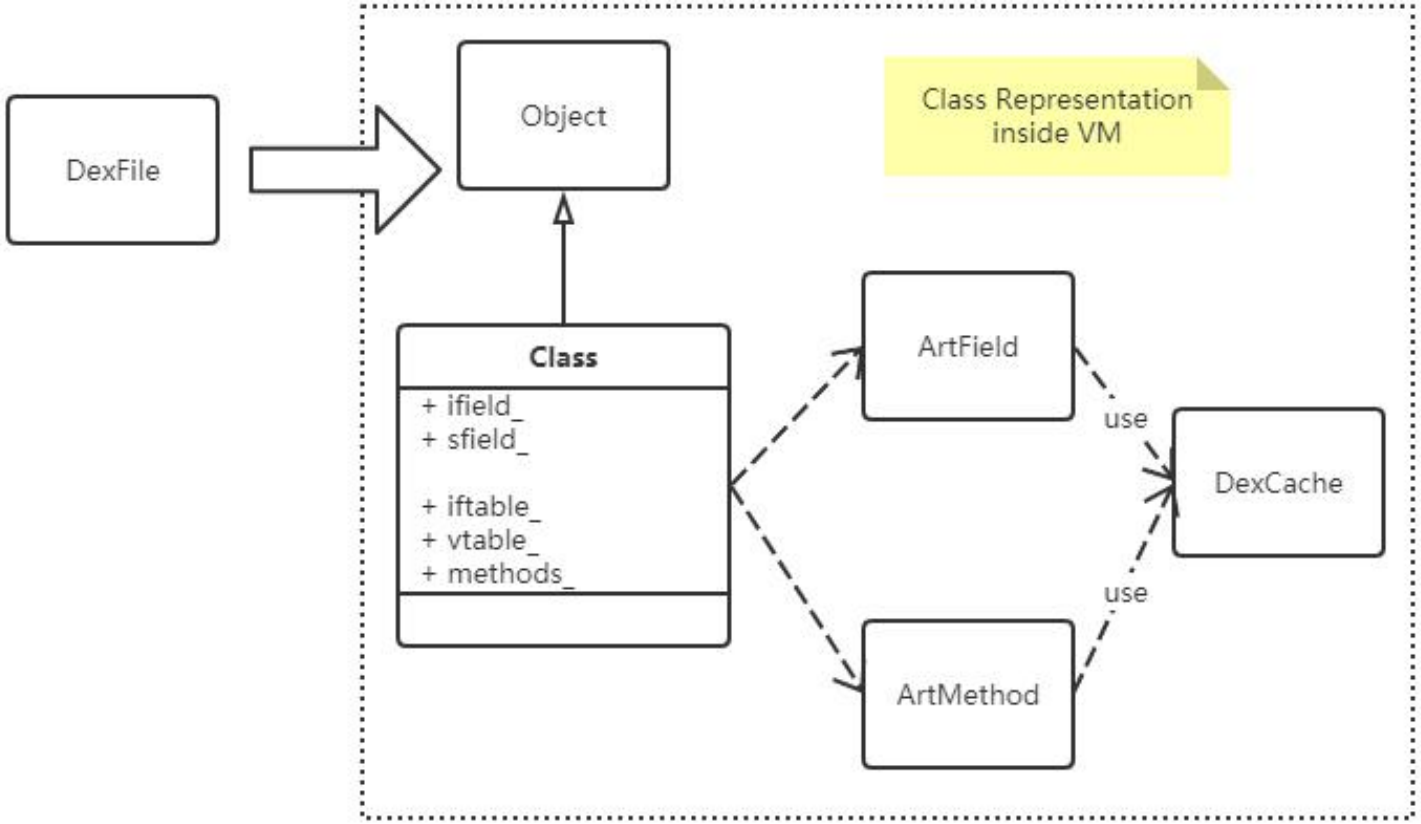
- 加载：查找字节流，并且据此创建类的过程。加载需要借助类加载器。
- 连接（链接）：是指将创建的类进行合并，使之能够执行的过程。链接还分验证、准备和解析。
- 初始化，为标记为常量值的字段赋值，以及执行类初始化方法的过程。



类加载

类加载

Dex 文件中对应的 **类** 在 VM 中被创建为一个 **mirror Class** 对象，相应的该 Class 的成员变量和成员函数被对应创建为 **ArtField** 对象和 **ArtMethod** 对象，注意实际的 ArtField 和 ArtMethod 对象内容缓存在 DexCache 中。



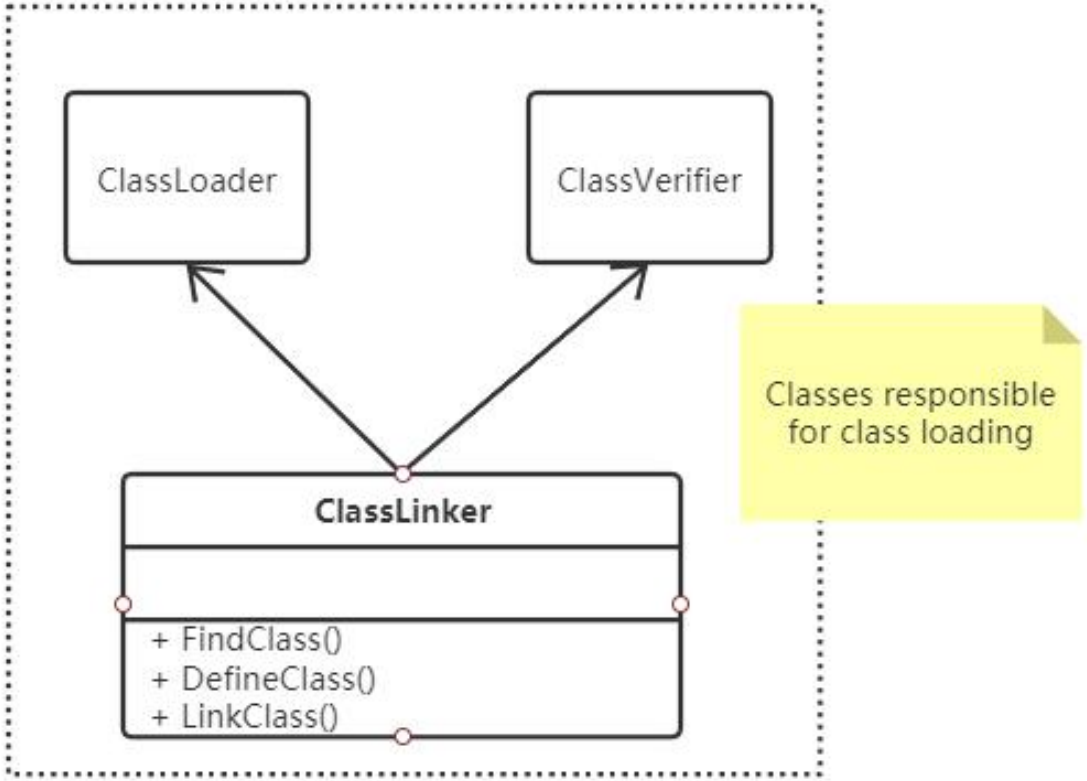
类加载

ArtMethod 类

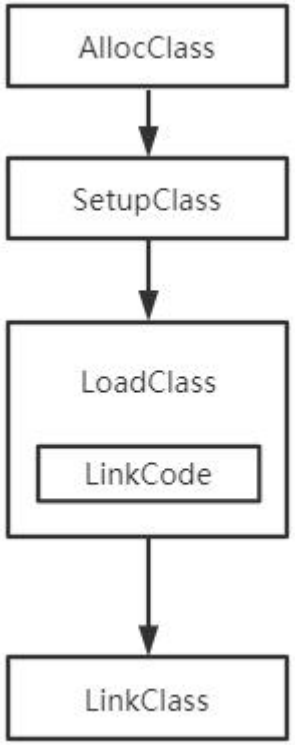
```
class ArtMethod final {
  ......
  // Must be the last fields in the method.
  struct PtrSizedFields {
    // Depending on the method type, the data is
    //   - native method: pointer to the JNI function registered to this method
    //                    or a function to resolve the JNI function,
    //   - conflict method: ImtConflictTable,
    //   - abstract/interface method: the single-implementation if any,
    //   - proxy method: the original interface method or constructor,
    //   - other methods: the profiling data.
    void* data_;

    // Method dispatch from quick compiled code invokes this pointer which may cause bridging into
    // the interpreter.
    void* entry_point_from_quick_compiled_code_;
  } ptr_sized_fields_;
  ......
}
```

类加载

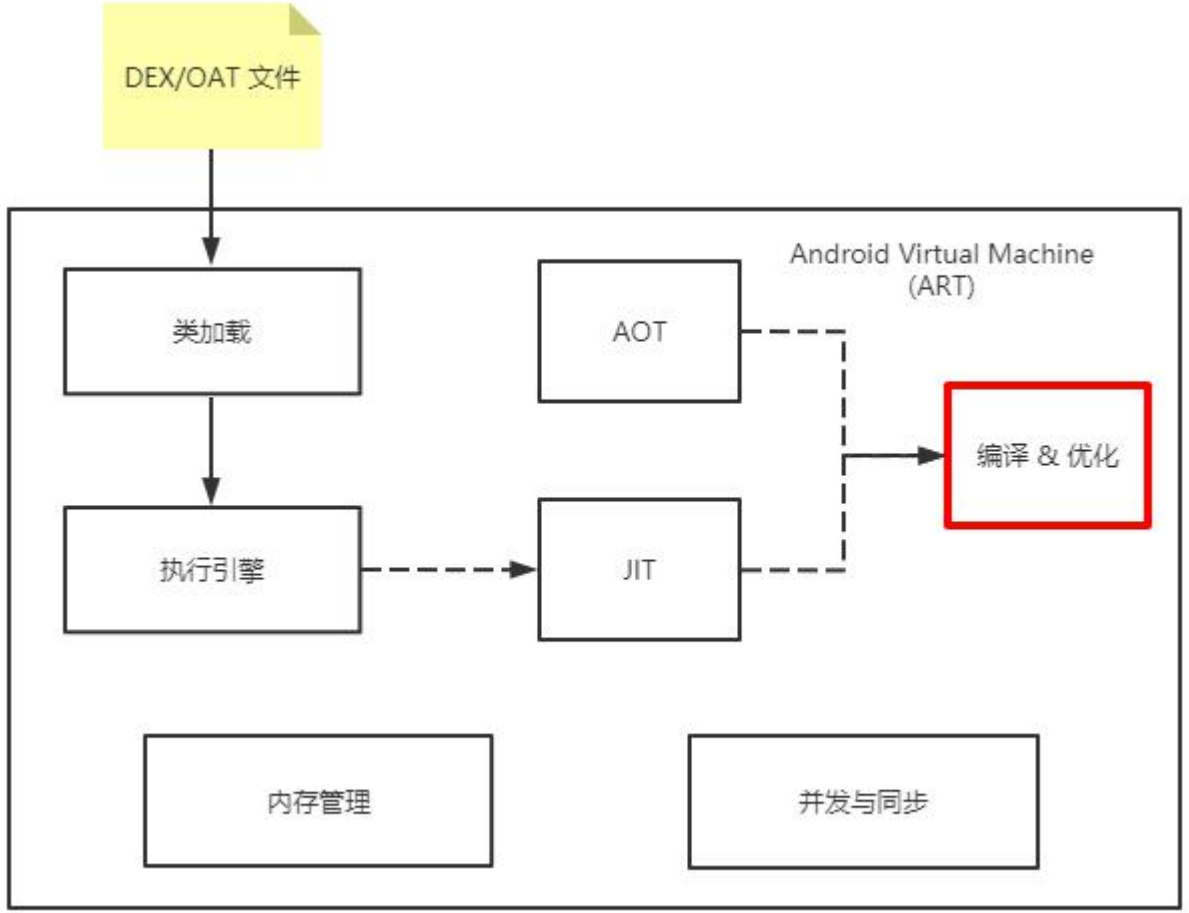
类加载



类加载的入口是 ClassLinker::DefineClass()

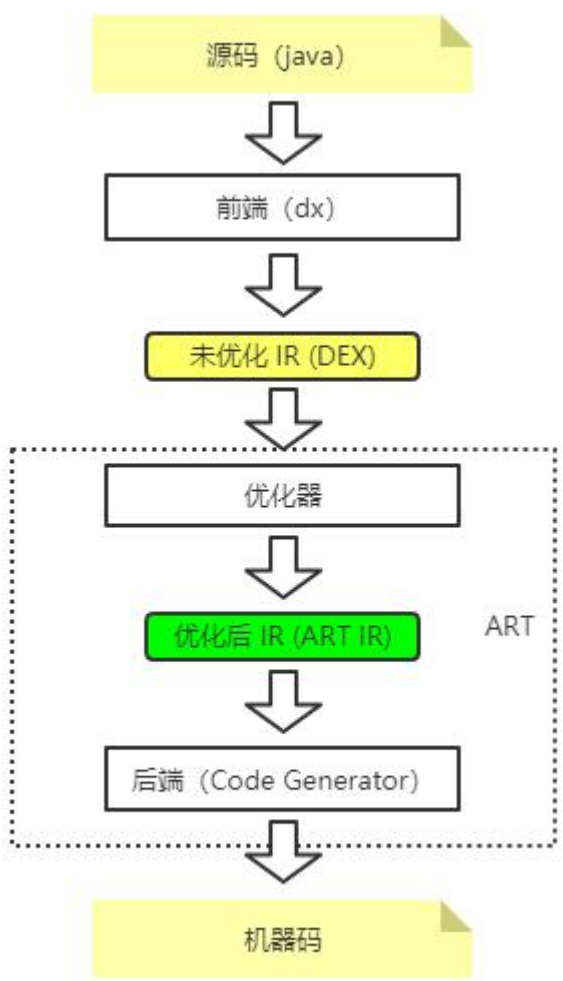


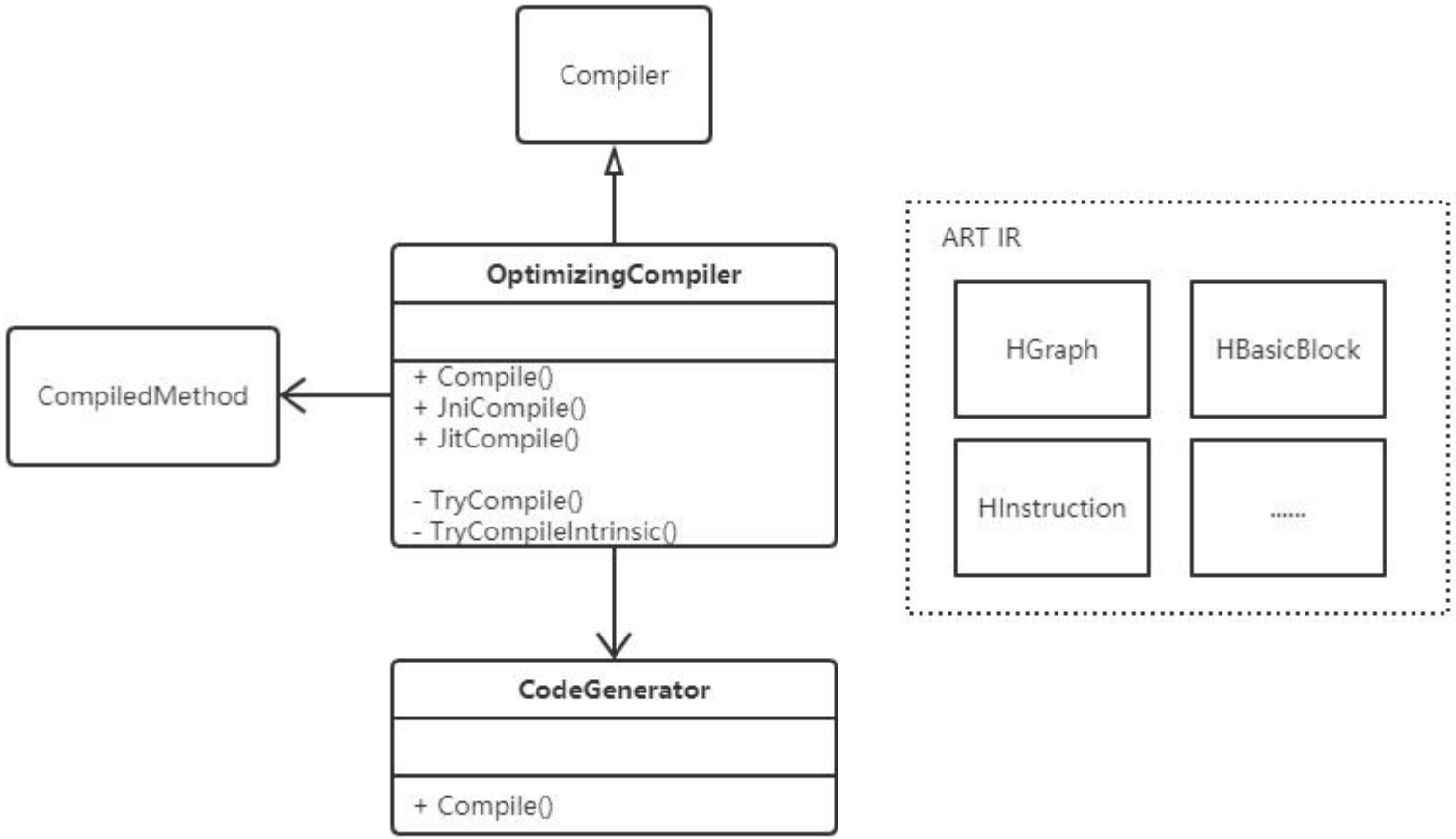
目 录



编译与优化

编译与优化流程





编译与优化

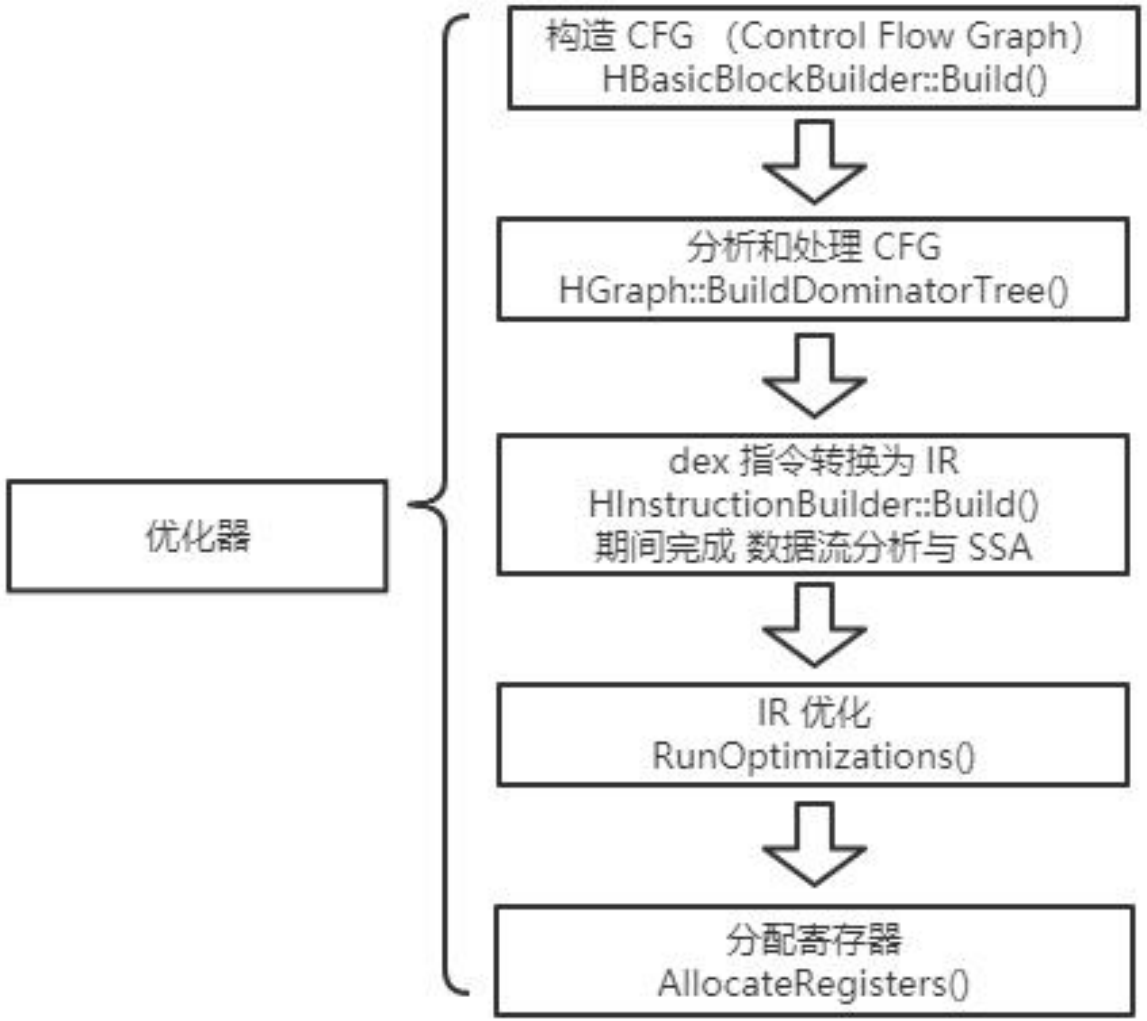
优化器

OptimizingCompiler 类对外暴露了三个基本接口：

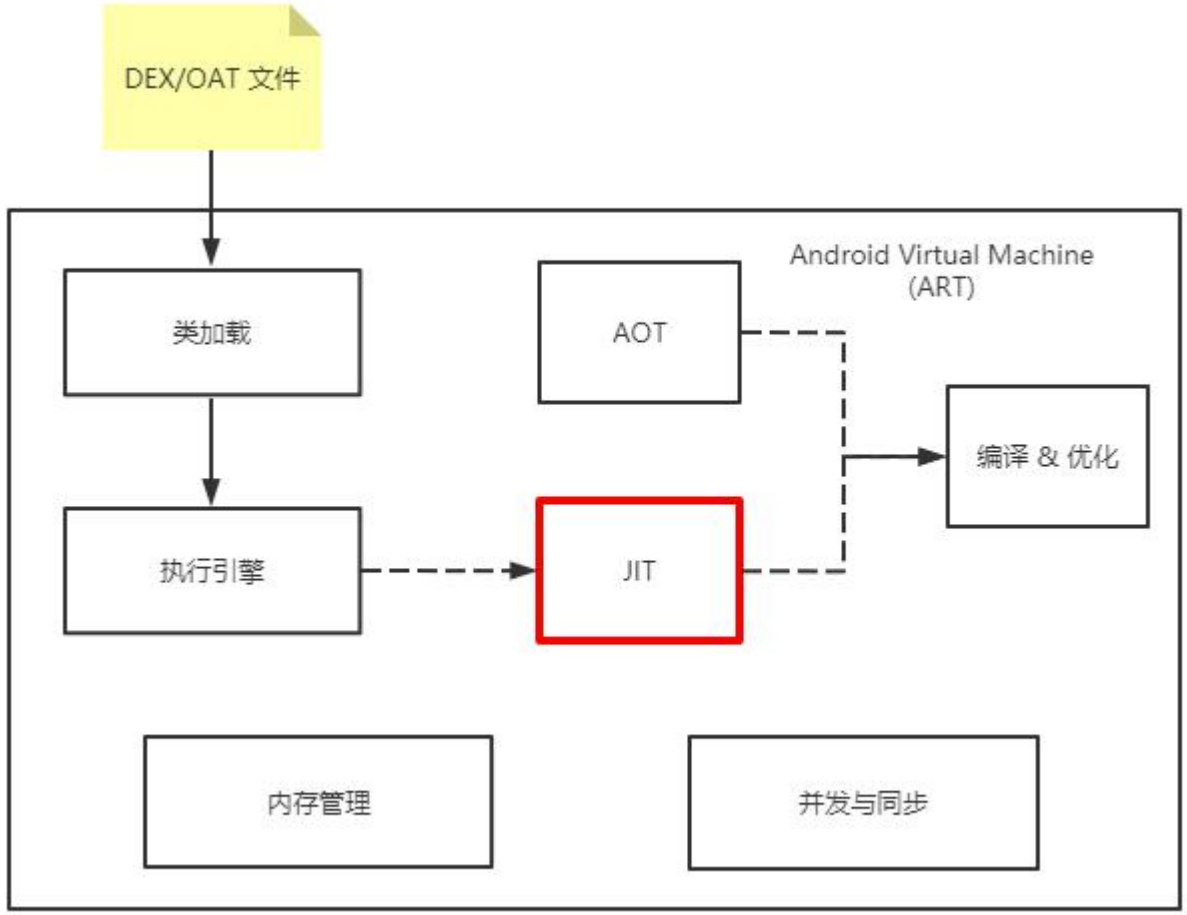
- JitCompile
- Compile
- JniCompile

内部实际调用了私有的两个关键函数完成了优化的核心工作:

- TryCompile
- TryCompileIntrinsic



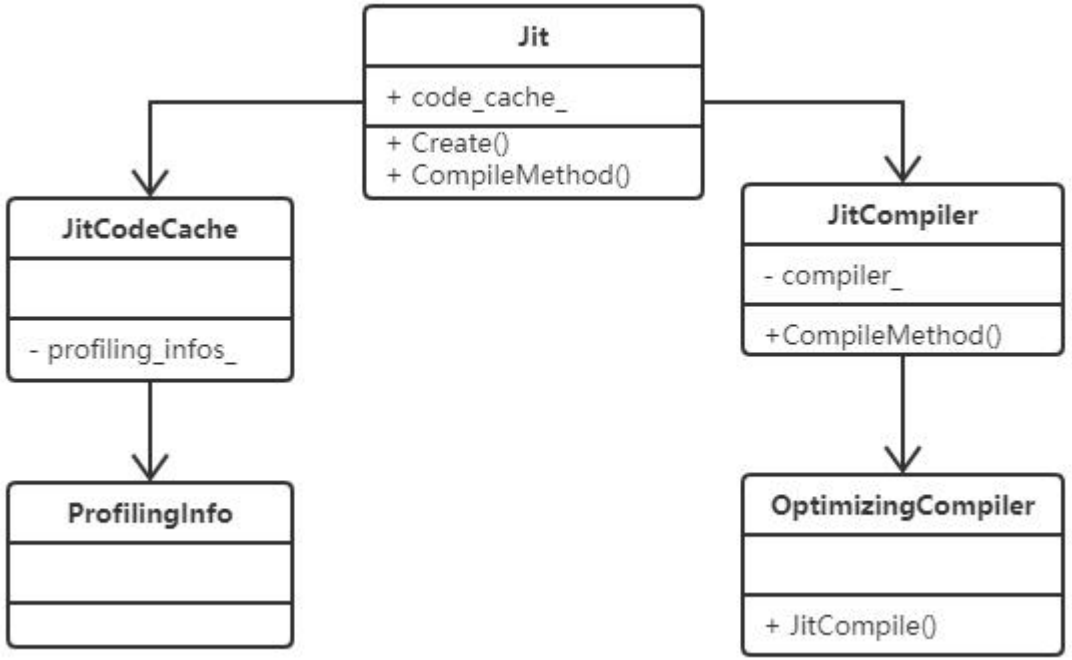
目 录



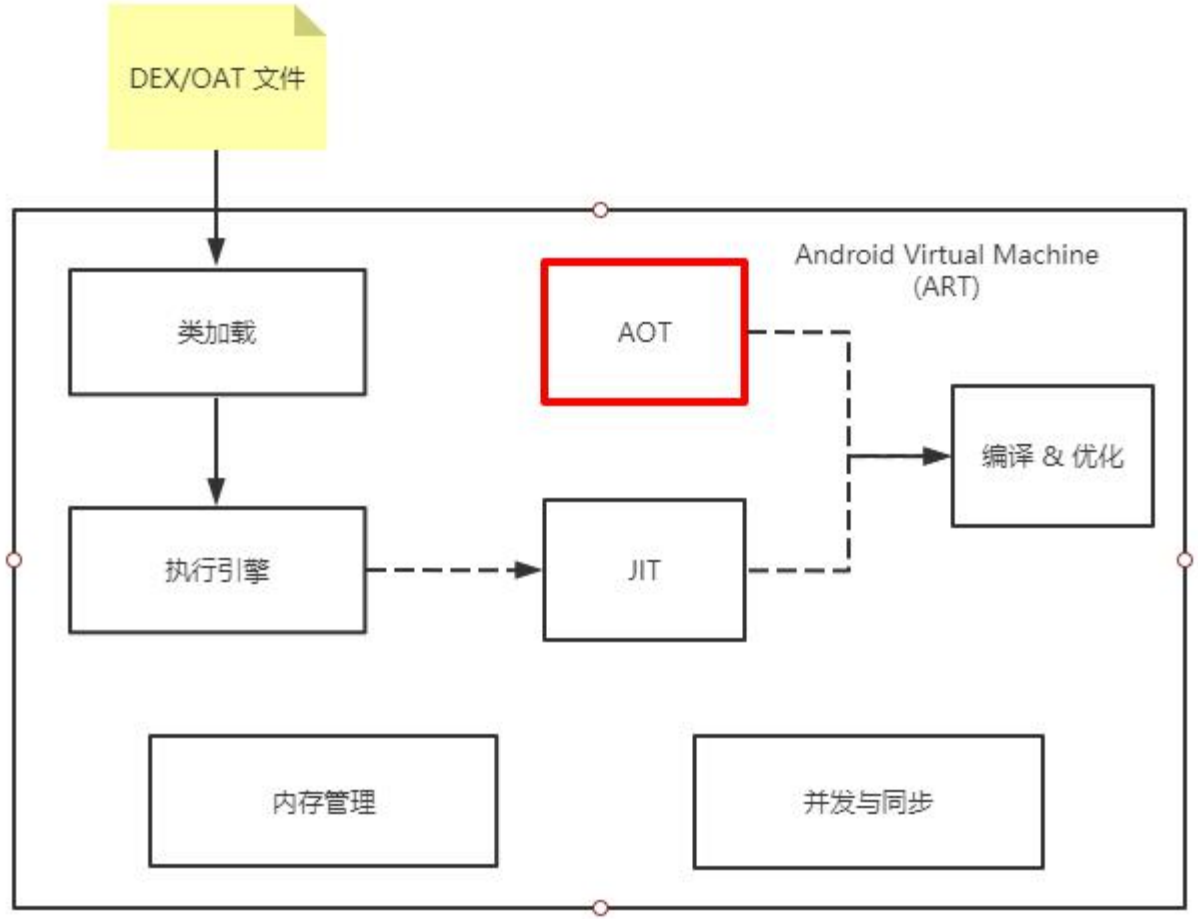
Just-In-Time(JIT)

Just In Time 执行的时机

- 虚拟机会对方法的执行此时进行统计，当某个方法的执行次数达到一定的阈值后（Hot Method），虚拟机会触发 JIT 操作，将这些 Hot Method 编译成本地机器码存放在 Jit 的 Cache 中。此后这些方法将以机器码的方式执行。这是经典的 JIT 的执行方式。
- 方法内部局部代码成为 Hot Code，导致该方法被标识为 Hot Method，触发 JIT，同时采用 On Stack Replacement (简称 OSR) 技术立即替换当前栈切换为机器码方式执行。



目 录



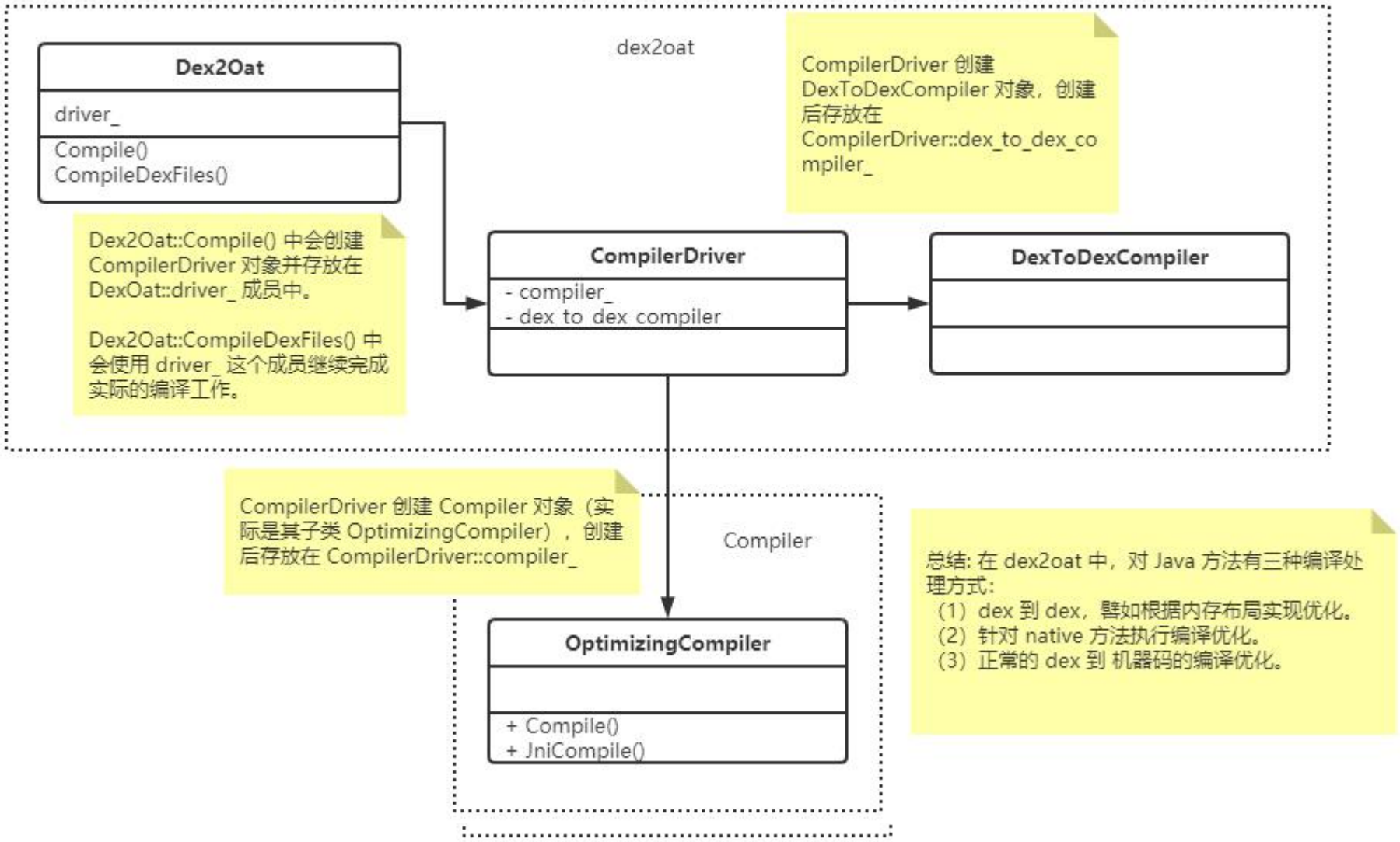
Ahead-Of-Time(AOT)

dex2oat 执行的时机

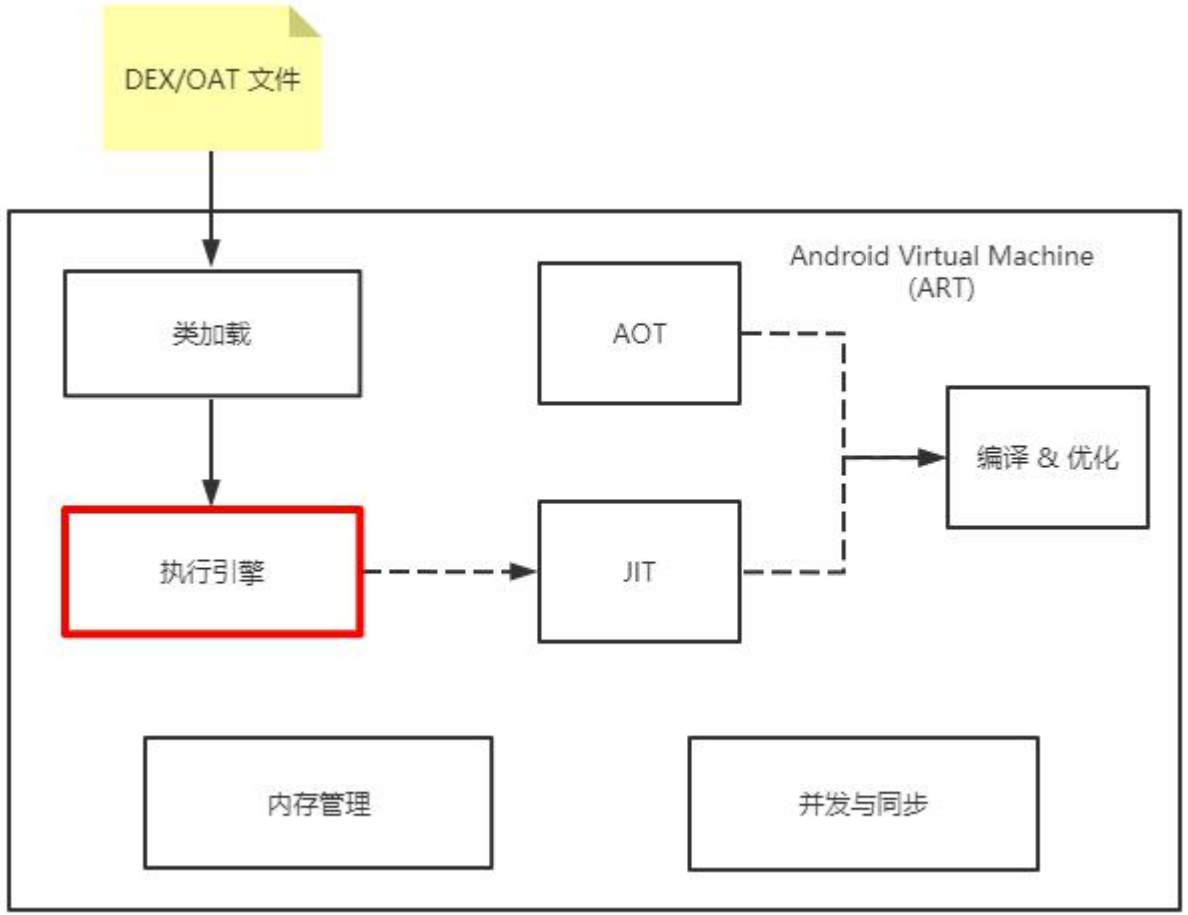
- 主机（host）侧编译系统时，会根据产品的配置将一些系统 pre-loaded 包和应用 apk 提前编译好生成 oat（art）文件存放在 image 中。
- 在设备（target）侧系统运行期间，将设备上的 jar/apk 文件中的 dex 字节码编译成本地机器码。具体行为程度分以下四个等级：
  - ✓verify-profile：只对包含在 profile 中的类进行校验。
  - ✓interpret-only：只对 dex 文件进行校验，同时会编译 jni 方法。
  - ✓speed-profile：只对包含在 profile 中的类进行校验和编译。
  - ✓speed：尽可能对 dex 文件内容进行校验和编译以最大能力提升代码执行速度。

Ahead-Of-Time(AOT)

dex2oat 主要类

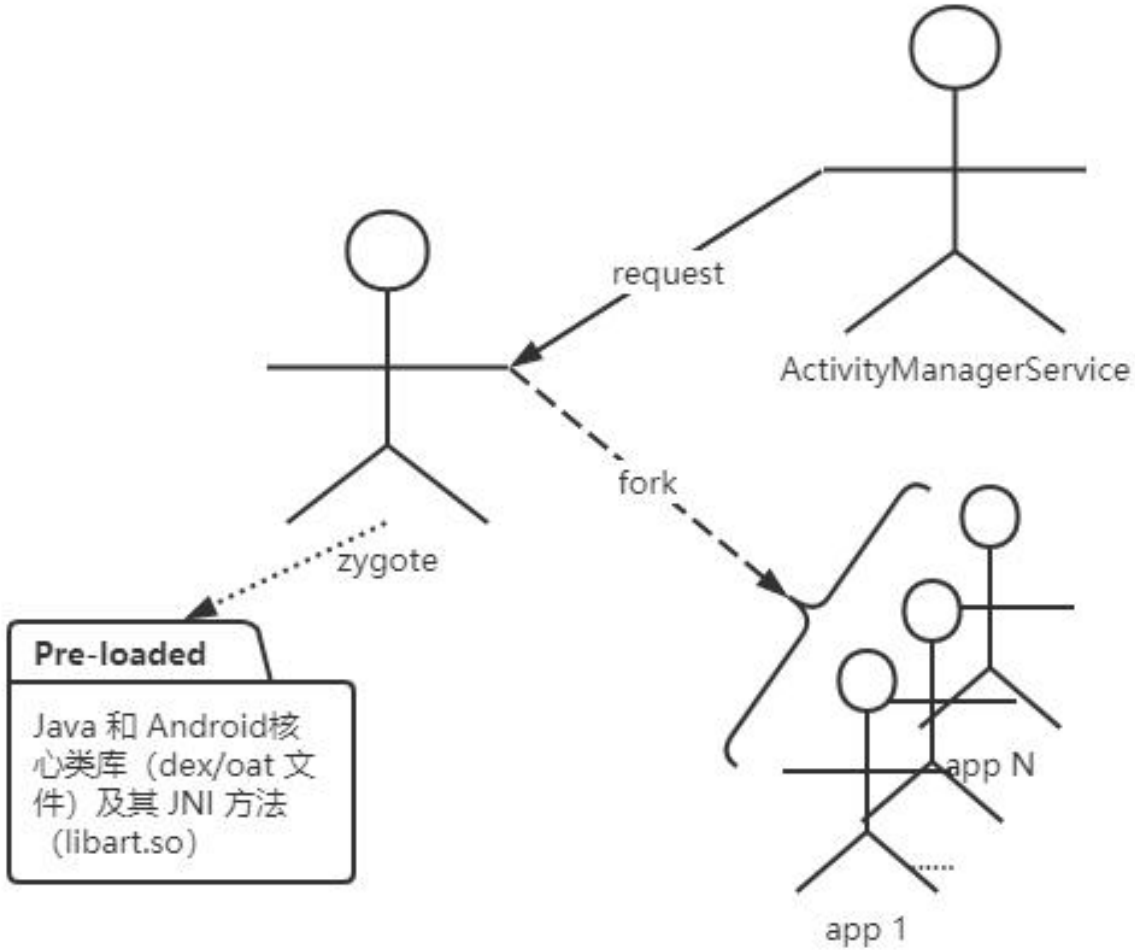


目 录



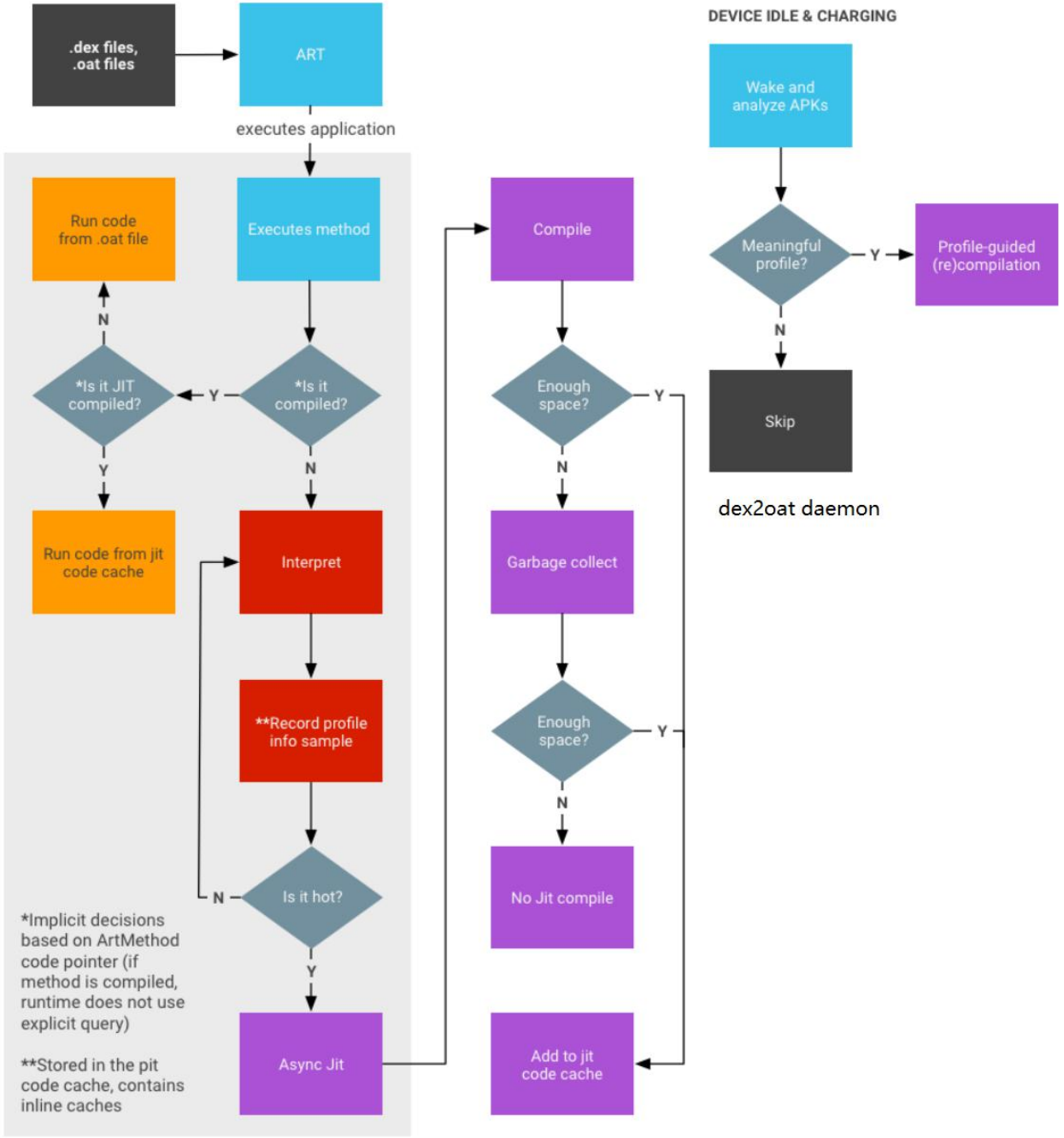
执行引擎

ART 的创建

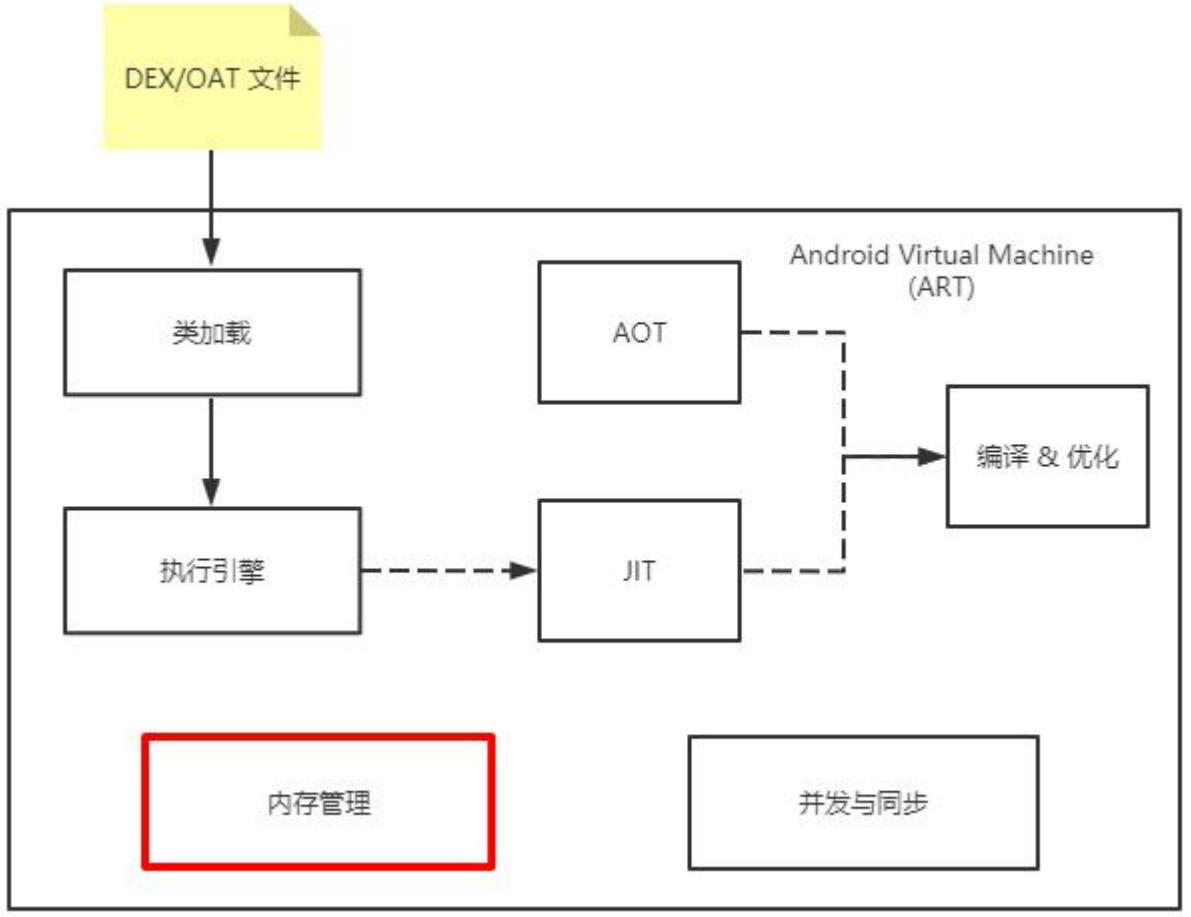

request
ActivityManagerService
fork
zygote
Pre-loaded
Java 和 Android核心类库（dex/oat 文件）及其 JNI 方法（libart.so）
app N
.....
app 1

执行引擎

解释与执行


.dex files, .oat files
ART
executes application
Run code from .oat file
Executes method
*Is it JIT compiled?
*Is it compiled?
Y
N
Y
N
Run code from jit code cache
Interpret
**Record profile info sample
Is it hot?
N
Y
*Implicit decisions based on ArtMethod code pointer (if method is compiled, runtime does not use explicit query)
**Stored in the pit code cache, contains inline caches
Async Jit
Compile
Enough space?
Y
N
Garbage collect
Enough space?
Y
N
No Jit compile
Add to jit code cache
DEVICE IDLE & CHARGING
Wake and analyze APKs
Meaningful profile?
Y
N
Profile-guided (re)compilation
Skip
dex2oat daemon

目 录



内存管理

主要职能

- 对应 字节码的 new 操作， 虚拟机为其执行分配内存操作
- 内存的释放无需程序负责， 虚拟机提供 “垃圾回收 (Garbage Collect)” 机制自动释放。

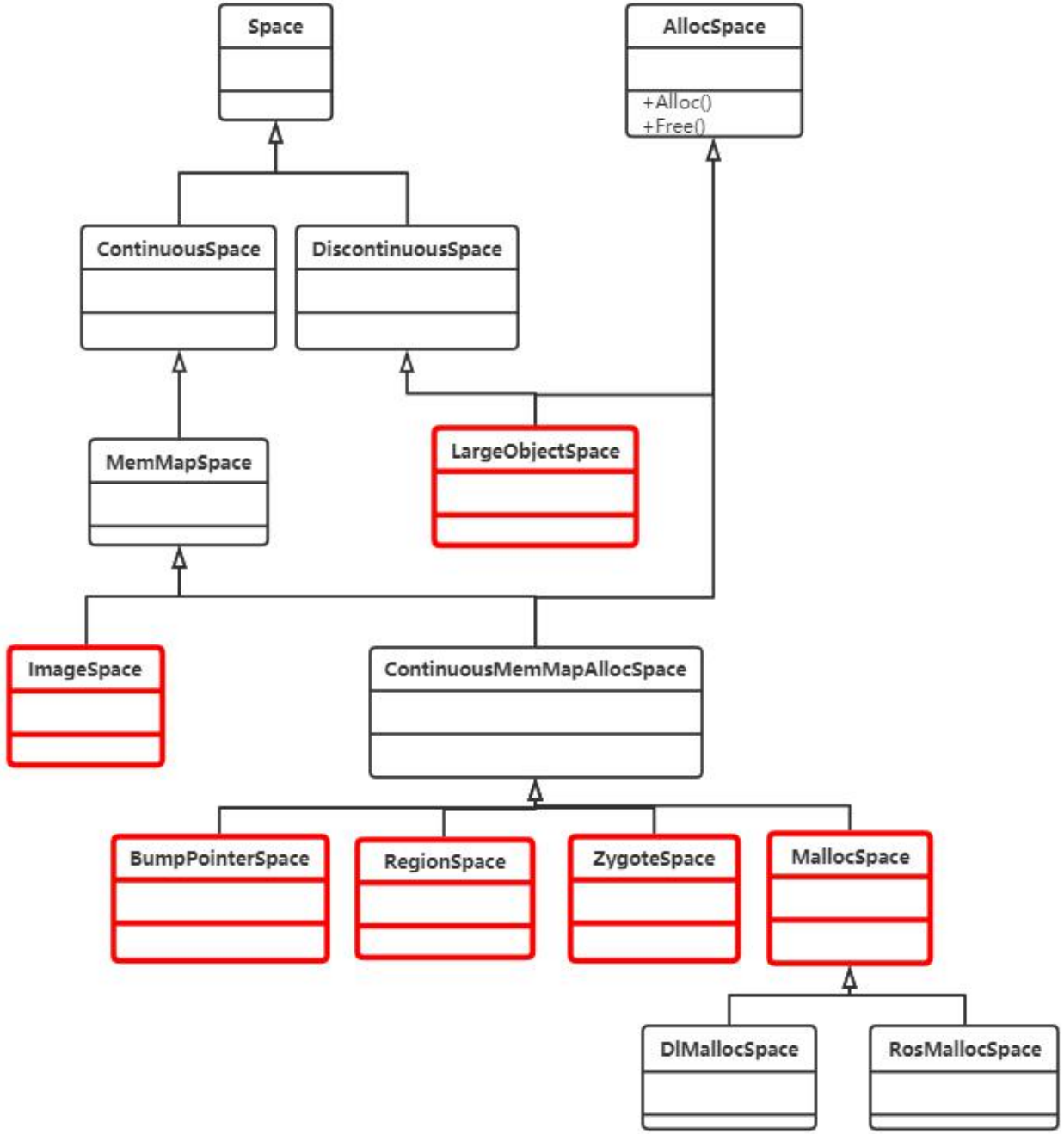
内存管理

内存分配

内存分配的模式：

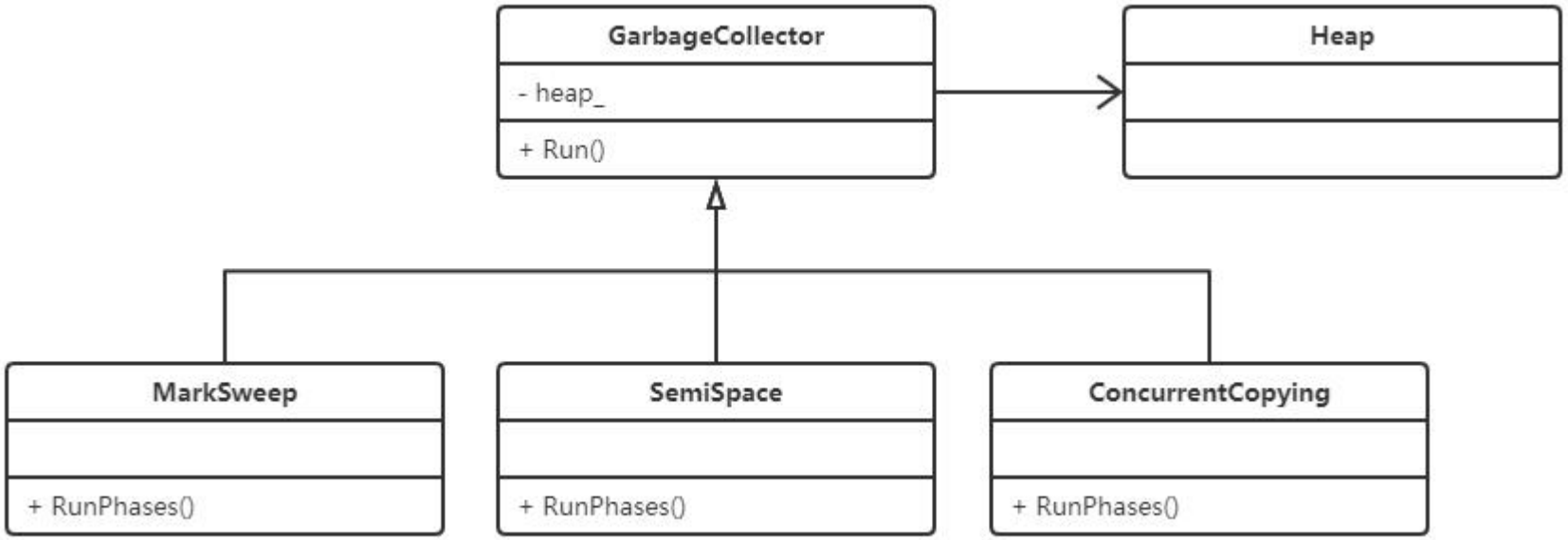
```
enum SpaceType {
  kSpaceTypeImageSpace,
  kSpaceTypeMallocSpace,
  kSpaceTypeZygoteSpace,
  kSpaceTypeBumpPointerSpace,
  kSpaceTypeLargeObjectSpace,
  kSpaceTypeRegionSpace,
};
```

内存分配器的选择由垃圾回收器的类型来决定，ART 针对不同的垃圾回收器采用其最适合使用的内存分配器。

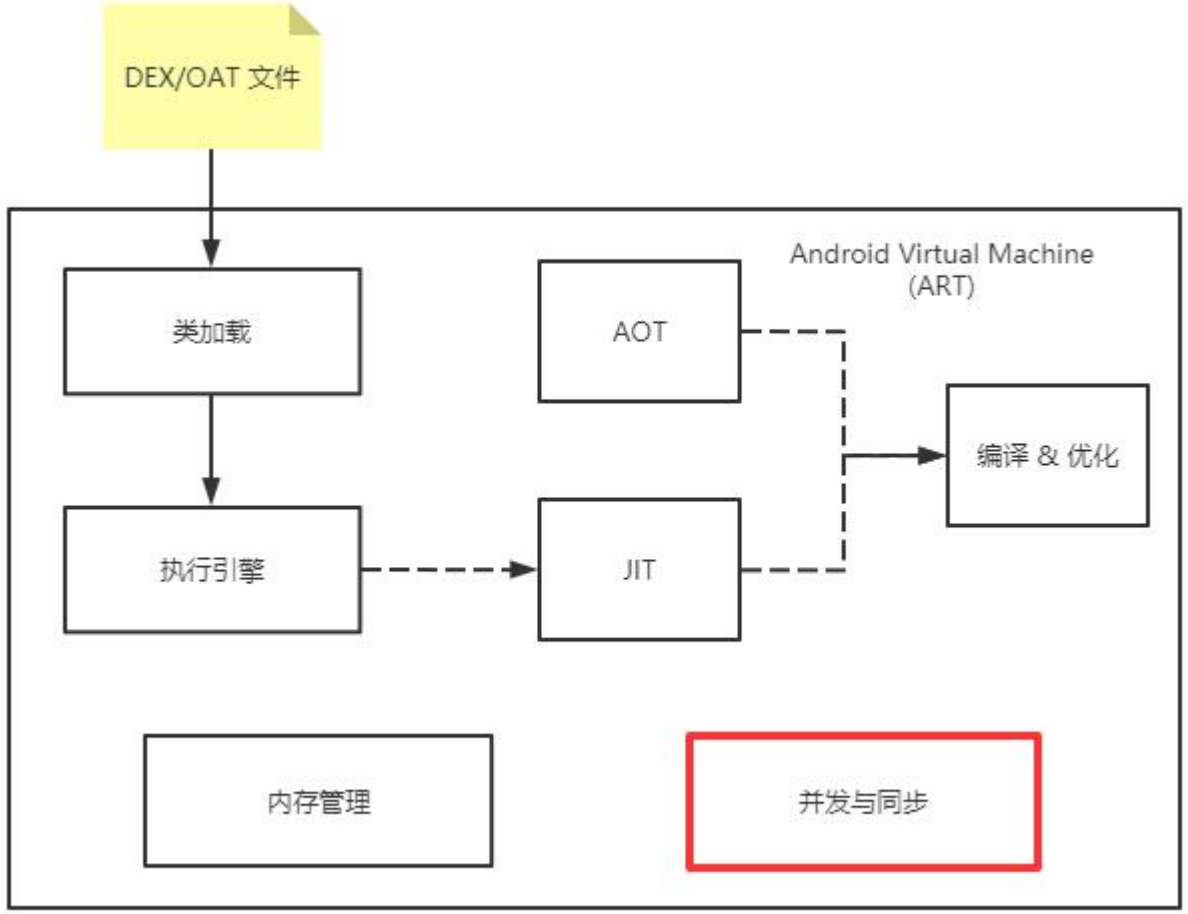


内存管理

内存释放（GC）


GarbageCollector
- heap_
+ Run()
Heap
MarkSweep
+ RunPhases()
SemiSpace
+ RunPhases()
ConcurrentCopying
+ RunPhases()

目 录



并发与同步

线程模型

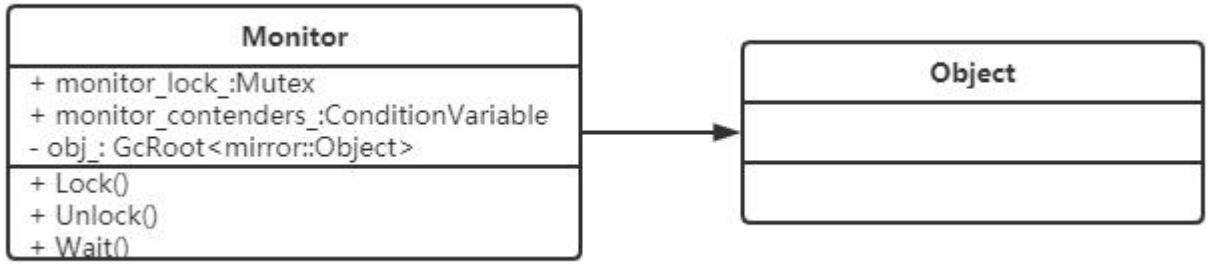
- 每个操作系统（Native）进程都对应一个 ART VM
- 每个 Native 进程中会包含一个或者多个 Native 线程
- 每个 Java 线程都对应一个 Native 线程 （一对一线程模型）

```
class Thread {
  ......
  struct PACKED(sizeof(void*)) tls_ptr_sized_values {
    ......
    // Entrypoint function pointers.
    // TODO: move this to more of a global offset table model to avoid per-thread duplication.
    JniEntryPoints jni_entrypoints;
    QuickEntryPoints quick_entrypoints;

    // Mterp jump table base.
    void* mterp_current_ibase;
    ......
  } tlsPtr_;
  ......
};
```

并发与同步

实现相关的类

- 每一个 Monitor 对象都会关联一个 Object 对象
- Monitor 中具体的 Mutex/ConditionVariable 由具体的操作系统线程同步机制实现。



谢 谢

欢迎批评指正