Jim Huang (黃敬群)

Developer, 0xlab jserv

@0xlab.org

14, 2012 / GTUG Taipei

Rights to copy

March.org

欢迎更正、建议、贡献和翻译！

我自己 是一名 Kaffe（世界首创的开源 JVM）开发人员

- 线程解释器、JIT、AWT 用于嵌入式系统，健壮性

是 GCJ（GCC 的 Java 前端）和 GNU 类路径 开发人员

是 AOSP（Android 开源项目 ）的贡献者

- 45+ 个补丁正式合并
- 仿生 libc，ARM 优化

目标本演示文　　稿的

• 了解虚拟机的工作原理 • 使用现有工具分析 Dalvik VM • VM hacking 真的很有趣！

环境设置

参考硬件和主机配置

- Android 手机：Nexus S
  - –

http://www.google.com/phone/detail/nexus-s

– 安装 CyanogenMod (**CM9**)

http://www.cyanogenmod.com/

- 主机：Lenovo
  - Ubuntu Linux 11.10+（32 位）
- AOSP/CM9 源代码：4.0.3
- 按照 Wiki
  http://wiki.cyanogenmod.com/wiki/Building_from_source

从源代码构建 CyanogenMod

- cyanogen-ics$ 源构建/envsetup.sh

  包括 device/moto/stingray/vendorsetup.sh 包括 device/moto/wingray/vendorsetup.sh 包括 device/samsung/maguro/vendorsetup.sh 包括 device/samsung/toro/vendorsetup.sh 包括 device/ti/panda/vendorsetup.sh

  包括 vendor/cm/vendorsetup.sh

  包括 sdk/bash_completion/adb.bash

- -ics$ 午餐

  您正在 Linux

  午餐菜单上构建... 选择一个组合：

  1. full-eng

  ...

  8. full_panda-eng 9. cm_crespo-userdebug

目标：cm_crespo 配置：userdebug

Nexus S 设备配置

- 您想要哪个？ [全英文] 9

```
============================================
== PLATFORM_VERSION_CODENAME=REL
PLATFORM_VERSION=4.0 .3
TARGET_PRODUCT=cm_crespo
TARGET_BUILD_VARIANT=userdebug
TARGET_BUILD_TYPE=release
TARGET_BUILD_APPS=
TARGET_ARCH=arm
TARGET_ARCH_VARIANT=armv7-a-neon
HOST_ARCH=x86
HOST_OS=linux
HOST_BUILD_TYPE=release
BUILD_ID=MR1
```

========== ==========

构建 Dalvik VM

（ARM 目标 + x86 主机）

- -ics$ **make dalvikvm dalvik**

  ==========

  == PLATFORM_VERSION_CODENAME=REL

  PLATFORM_VERSION=4.0 .3

  TARGET_PRODUCT=cm_crespo

  TARGET_BUILD_VARIANT=userdebug

  TARGET_BUILD_TYPE=release

  ...

  libdvm.so是VM引擎

  安装： out/host/linux-x86/lib/libdvm.so **安装**：

  out/target/product/crespo/system/bin/dalvikvm host

  C++：dalvikvm <= dalvik/dalvikvm/Main.cpp

  主机可执行文件：dalvikvm 安装：out/host/linux

x86/bin/dalvikvm

复制：dalvik (out/host/linux x86/obj/EXECUTABLES/dalvik_intermediates/dalvik)

安装： out /host/linux-x86/bin/dalvik

“dalvik” 是一个用于启动 dvm 的 shell 脚本

Dalvik VM 需要核心 API 用于运行时

```
cyanogen-ics$ out/host/linux-x86/bin/dalvik
E(6983) 在 bootclasspath
'/tmp/cyanogen-ics/out/host/linux-x86/framework/core
hostdex.jar:/tmp/cyanogen-ics/out/host/linux
x86/framework/bouncycastle 中找不到有效条目
-hostdex.jar:/tmp/cyanogen
ics/out/host/linux-x86/framework/apache-xml
hostdex.jar' (dalvikvm)
E(6983) VM 中止 (dalvikvm)
...
```

```
out/host/linux-x86/bin/dalvik：第 28 行：6983
分段错误 （核心转储）
ANDROID_PRINTF_LOG=tag ANDROID_LOG_TAGS=""
ANDROID_DATA=/tmp/android-data
ANDROID_ROOT=$ANDROID_BUILD_TOP/out/host/linux-x86
LD_LIBRARY_PATH=$ANDROID_BUILD_TOP/out/host/linux-x86/lib
$ANDROID_BUILD_TOP/out/host/linux- x86/bin/dalvikvm
-Xbootclasspath:
```

```
$ANDROID_BUILD_TOP/out/host/linux-x86/framework/core-hostdex.jar:
$ANDROID_BUILD_TOP/out/host/linux-x86/framework/bouncycastle
hostdex.jar: \
$ANDROID_BUILD_TOP/out /host/linux-x86/framework/apache-xml
hostdex.jar $*
```

满足 Dalvik 运行时依赖

```
$ bouncycastle bouncycastle-hostdex cyanogen -ics$
```

```
make sqlite-jdbc mockwebserver cyanogen -ics$ make
sqlite-jdbc-host
cyanogen -ics$ make mockwebserver-hostdex cyanogen
-ics$ make apache-xml-hostdex cyanogen
-ics$ (cd libcore && make) cyanogen
-ics$ out/host/linux-x86/bin/dalvik ...
I(19820) 无法打开或创建缓存
/tmp/cyanogen-ics/out/host/linux-x86/framework/core
hostdex.jar (/data/dalvik-cache/tmp@cyanogen
ics@out@host@linux-x86@framework@核心
hostdex.jar@classes.dex) (dalvikvm)
E(19820) 无法统计 dex 缓存目录
“/data/dalvik-cache”：没有这样的文件或目
录 (dalvikvm)
```

“dalvik-cache”需要额外空间。

主机端 Dalvik VM

```
cyanogen-ics$ make dexopt cyanogen
-ics$ sudo mkdir -p /data/dalvik-cache
cyanogen-ics$ sudo chmod 777
/data/dalvik-cache cyanogen-ics$
out/host/linux-x86/ bin/dalvik Dalvik VM 需要类
名
```

最后，主机端 dalvik vm 准备就绪。
它只是抱怨没有给定的课程。

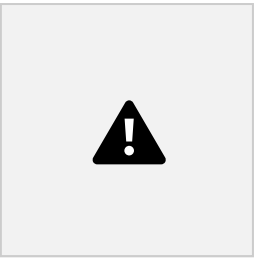
```
cyanogen-ics$ ls /data/dalvik-cache/
tmp@cyanogen-ics@out@host@linux-x86@framework@apache-xml
hostdex.jar@classes.dex
tmp@cyanogen-ics@out@host@linux-
x86@framework@bouncycastle hostdex.jar@classes.dex
```

```
tmp@cyanogen-ics@out@host@linux-x86@framework@core
hostdex.jar@classes.dex
```

议程（一）虚拟机如何Works (2) Dalvik VM

(3) 实用程序

虚拟机的工作原理



什么是虚拟机

- 虚拟 机 (**VM**) 是机器(即计算机) 的软件实现 ，它像物理机一样执行程序。
- 基本部分

- A寄存器集
- 堆栈（可选）
- 执行环境
- 垃圾收集堆
- 常量池
- 方法存储区
- 指令集

VM 类型

- 基于其功能
  - 系统虚拟机

支持完整操作系统的执行

– 进程虚拟机

支持单个进程的执行

- 基于其架构
  - 基于堆栈的 VM (使用指令加载到堆栈中执行)
  - Reg的 VM(使用在源寄存器和目标寄存器中编码的指令)

类
加载器

ister

JVM 概念架构

指令

内存空间

方法 Java 堆 Java 堆栈

机方法堆
自动内存管理器

地址 数据和

指令计数器和隐式
寄存器
执行引擎

机方法接口
机
库

javaframee vars

optop

方法

类字段

pc

其他

~~{变量 *locales*}~~

Segment

javaframe

vars

Environnement cotext

optop_i 注册

optop

Segment
javaframe_i

示例

:JVM

- 示例 Java 源代码：Foo.java

```
class Foo {
 public static void main(String[] args) {
System.out.println("Hello, world"); }
 int calc(int a, int b) {
 int c = 2 * (a + b);
 返回 c;
 }
}
```

例子：JVM

```
$ javac Foo.java
$ javap -v Foo
编译自 "Foo.java"
class Foo extends java.lang.Object
...
int calc(int, int);
 代码：
 Stack=3, Locals=4, Args_size=3
 0: iconst_2
```

```
1: iload_1
2: iload_2
3: iadd
4: imul
5: istore_3
6: iload_3
7: ireturn
```

字节码执行

c := 2 * (a + b)

- 示例字节码
  - **iconst 2**
  - **a**
  - **b**
  - **iadd**

- imul
- istore c

示例字节码：

```
imul
istore c
```

```
iconst 2
a
b
iadd
```

abc

```
                7
                0
                2
42
```

- 计算：`c := 2 * (a + b)`
- 示例：

```
                b
                iadd
iconst 2        imul
a               istore c        abc
```

```
                7
                0 42 2


42
```

- 计算：`c := 2 * (a + b)`
- 示例：

```
          b                          42 7
          iadd                       0
iconst 2  imul
a         istore c   abc
```

```
42 2
```

```
7
```

- 计算：`c := 2 * (a + b)`
- 示例：

```
                a               imul
                b               istore c
iconst 2        iadd
```

```
                                    7
                                    0 49 2
```

```
abc                 42
```

- 计算：`c := 2 * (a + b)`
- 示例：

```
                    b
                    iadd
iconst 2            imul
a                   istore c        abc
```

```
                7
                0
                98
42
```

- 计算：`c := 2 * (a + b)`
- 示例：

```
          b
          iadd
iconst 2  imul
a         istore c   abc
```

42 7 98

计算：c := 2 * (a + b)

iadd in specification and实现

③ 值

增

value1 ①+ value2

④

pop push

2

~~value1~~ ~~value1 +~~

②

弹出 ~~值2~~

案例 SVM_INSTRUCTION_IADD：{

```
/* 指令体 */
jint value1 = stack[stack_size - 2].jint;
                                        ②
jint value2 = stack[--stack_size].jint;
                                          ①
stack[stack_size - 1].jint = value1 +  ④
                                  ③
value2;   /* 派遣 */
```

转到 调度；

}

取自 SableVM

sablevm/src/libsablevm/instructions_switch.c

示例：Dalvik VM

```
$ dx --dex --output=Foo.dex Foo.class
$ dexdump -d Foo.dex
Processing 'Foo.dex'...
Opened ' Foo.dex', DEX 版本 '035'
...
 虚拟方法 -
 #0 : (in LFoo;)
 name : 'calc'
 type : '(II)I'
...
00018c: |[00018c] Foo.calc: (II)I 00019c: 9000 0203
```

```
|0000: add-int v0, v2, v3 0001a0: da00 0002 |0002:
mul-int/lit8 v0, v0, #int 2 0001a4: 0f00 |0004: 返回v0
```

Java字节码对比Dalvik 字节码（堆栈与寄存器）

公共 int 方法（int i1, int i2）

```
{
 int i3 = i1 * i2;
 返回 i3 * 2;
}
```

```
.var 0 是“this”
.var 1 是参数 #1
.var 2 是参数 #2
```

方法 public method(II)I iload_1

```
 iload_2
 imul
 istore_3
 iload_3
 iconst_2
 imul
 ireturn
this: v1 (Ltest2;)
parameter[0] : v2 (I)
参数[1] : v3 (I)
```

```
.method public method(II)I mul-int v0,v2,v3
mul-int/lit-8 v0,v0,2 return v0
.end 方法
```

`.end` 方法

Dalvik 基于寄存器

- Dalvik 使用 3 操作数形式，它实际上是处理器使用的

Dalvik 基于寄存器

- 要执行“int foo = 1 + 2”，VM 会：
  - const-4 将 1 存储到寄存器 0
  - 添加-int/lit8 将寄存器 0 (1) 中的值与字面量 2 相加并将结果存储到寄存器 1 中——即“foo”

Dalvik 是基于寄存器的

- 这只是 2 次调度，但 Dalvik 字节码被测量为 2-字节单位
- Java 字节码是 4 字节，Dalvik 字节码实际上是 6 字节

Code Size

- 一般来说，基于寄存器的 VM 指令的代码大小大于对应的堆栈 VM 指令
- 平均而言，注册码是比原始堆栈代码大 25.05%

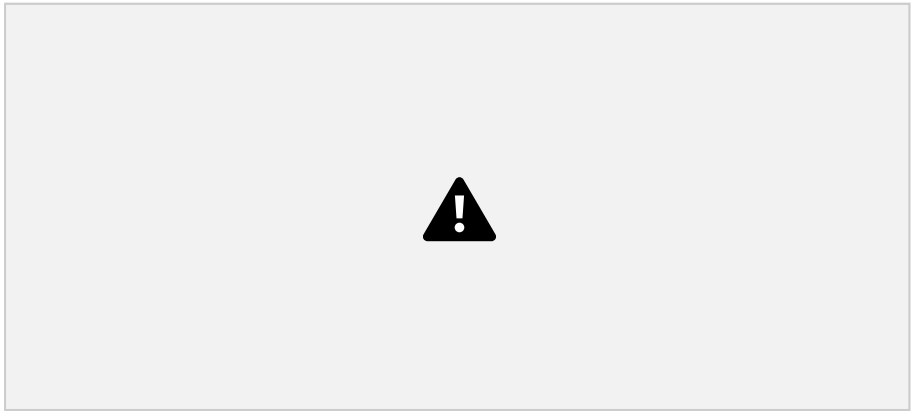
执行时间

- 寄存器架构需要执行的 VM 指令平均减少 47%

资料来源：虚拟机对决：堆栈与寄存器
Yunhe Shi、David Gregg、Andrew Beatty、M. Anton Ertl

Dalvik-JVM

- 指令翻译
- 一条 Dalvik 指令 → 多条 Java 指令



Dalvik VM 在 ARM 目标上执行

```
$ dx --dex --output=foo.jar Foo.class $ adb
push foo.jar /data/local/
$ adb shell dalvikvm \
 -classpath /data /local/foo.jar Foo Hello,
world
```

指令调度

```
static void interp(const char* s) {
for (;;) {
switch (*(s++)) {
```

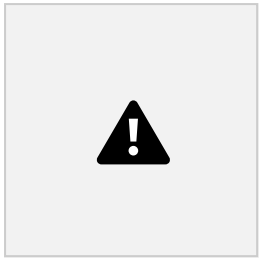
```
case 'a': printf (“Hello”);休息;
case 'b': printf (" "); break;
case 'c": printf ("world!"); break;
case 'd': prinf ("\n");休息;
案例'e'：返回；
}
int main (int argc, char** argv) {
 interp("abcbde"):
}
```

Computed GOTO (in GCC's way)

```
#define DISPATCH() \
 { goto *op_table[*((s)++) - 'a ']; }

static void interp(const char* s) {
  static void* op_table[] =
      { &&op_a, &&op_b, &&op_c, &&op_d, &&op_e
  };派遣（）；
  op_a: printf((”Hello”); DISPATCH();
  op_b: printf(“”); DISPATCH();
  op_c: printf(”world!”); DISPATCH();
```

```
  op_d: prinf(”\n"); DISPATCH();
  op_e: return;
}
```



最佳调度实现

- 如果我们在汇编中重写计算的 GOTO 可以进一步优化。
- 上面的代码通常使用两次内存读取。我们可以在内存中布局所有字节码一种每个字节码占用完全相同的内存量的方式——这样我们可以直接从索引

中计算地址。

- 预热

码

频繁使用的

缓存线

- 字节文件格式版本号
- 各部分的数量和大小

元数据

- 类/继承的超类/实现接口的声明信息
- 域与方法
- 特征池
- 用户自定义的、RetentionPolicy为CLASS或RUNTIME的注解

- ——对应Java的“声明”与“信息”

方法

- 码
- 处理器表
- 不同区域大小

区别 操作数栈类型记录（StackMapTable，Java 6开始）

- 调试用符号信息（如LineNumberTable、LocalVariableTable）
- ——Java源代码中的“语句”与“表达式”匹配的信息

来源：Java Program in Action——Java程序的编译、加载与执行，莫枢

Class文件

```
                                    诠释 j = 42;
                                    }
                                    }
                                  }
import java.io.Serializable;

public class Foo implements
Serializable {  public void
b ar() {
 诠释 i = 31;
 如果 (i > 0) {
```

结构：
声明与特性

代码：
语句与表达式

输出调试符号信息

编译 Java源码反编译 Class 文件

```
javac -g Foo.java
```

```
javap -c -s -l -verbose Foo
```

Class 文件示例

```
public Foo();
 签名：()V
 LineNumberTable:
```

方法
元数据字节码

第2**行**：0

```
LocalVariableTable:
Start Length Slot Name Signature
0 5 0 this LFoo;
```

代码：

```
Stack=1, Locals=1, Args_size=1
0：aload_0
1：invokespecial #1； //方法 java/lang/Object."<init>":()V 4: return
```

Class 文件例子

```
public void bar();
 Signature: ()V
 LineNumberTable:
 line 4: 0
 line 5: 3
 line 6: 7
 line 8: 10

 LocalVariableTable:
```

方法
元数据 字节码

```
 Start Length Slot Name Signature
 10 0 2 j I 0 11 0 this
LFoo;
 3 8 1 i I
 StackMapTable: number_of_entries = 1 frame_type = 252 /* append */ offset_delta = 10
 locals = [ int ]
```

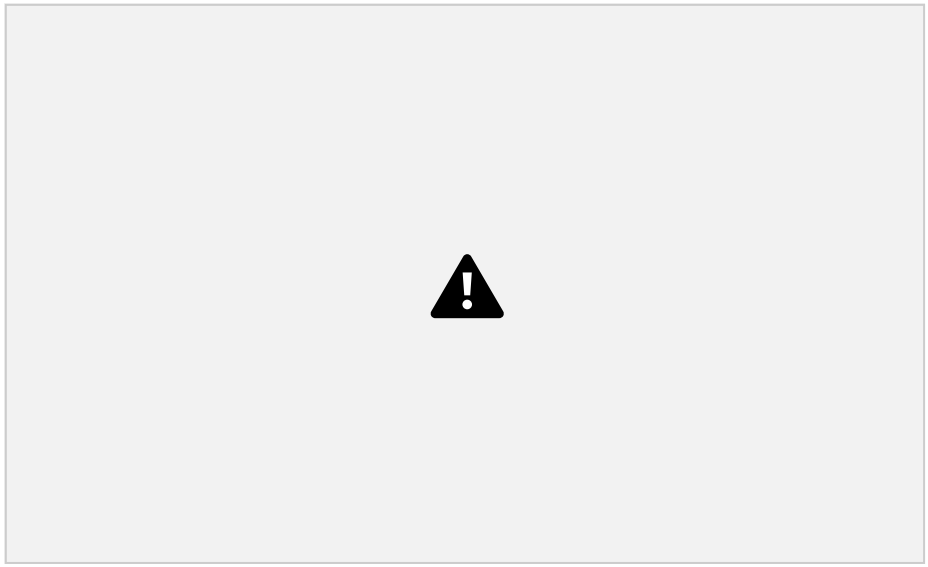
代码：
Stack=1, Locals=3, Args_size=1 0: bipush 31
2: istore_1
3: iload_1
4 : ifle0
7: bipush 42
9: istore_2
10: return
Java 6 开始，有基于分支控制流的方法会变化 StackMapTable，记下每个基本画面处操作 数栈的类型

栈与基础的体系结构的区别


```
公开课 演示{
  公共静态无效 foo () {
  整数 a = 1;
  诠释 b = 2;
```

```
    整数 c = (a + b) * 5;
  }
}
```

概念中的Dalvik虚拟机

概念中的Java虚拟机  来源：Java Program in Action——Java程序的编译、加载与执行、莫枢

Dalvik VM

Dalvik VM

- Dalvik架构是基于寄存器的
- 优化使用更少的空间
- 执行其自己的 Dalvik 字节码而不是 Java 字节码

- 类库取自 Apache Harmony – Apache License v2 下 Java SE 5 JDK 的兼容、独立实现

– 社区开发的模块化运行时（VM 和类库）架构。（现已弃用）

选择 Dalvik 的理由

- Dalvik（基于寄存器）比 JVM（基于堆栈）平均少执行 47% 的 VM 指令。- 寄存器代码比相应的堆栈代码大 25%。

- 由于更大的代码大小，获取更多VM 指令的成本增加仅涉及每条VM 指令1.07% 的额外实际机器负载。这是微不足道的。
- 一些营销原因
  – Oracle 对 Google 的诉讼

hello.JAVA

Memory(DDR) ARM CPU Dalvik

Write

(Source)

JAVAC

(Compiler)

hello.class

DX

(Compiler)

“Stack” Section

“BSS” Section

“Data” Section 代码 ） **Fetch**

F back excute Decode
g

解码（编号 ）

Memory fetch

“Text”部分

access

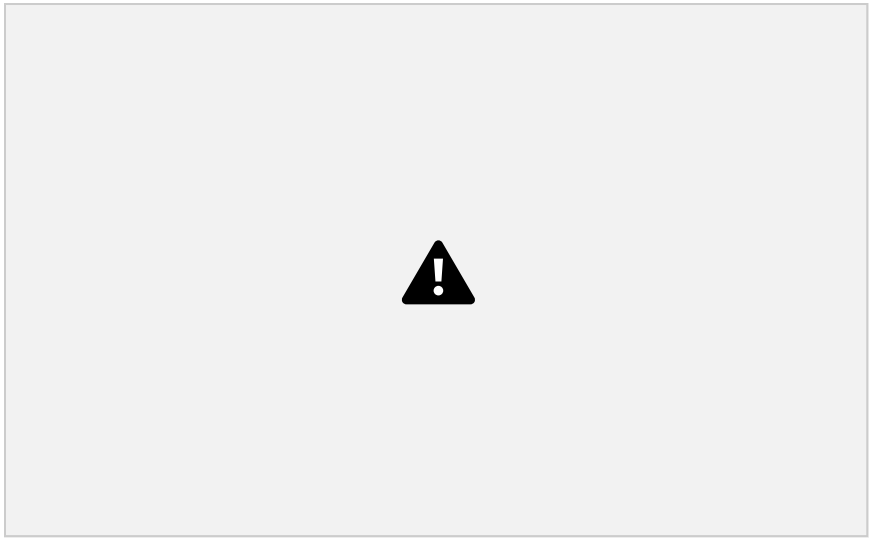
P

执行
（**ARM**

Hello.dex (Dex)

Loading Excute

Interpreter



Dalvik 架构

- 寄存器架构

- 2^{16} 可用寄存器
- 指令集有 218 个操作码
  - JVM：200 个操作码
- 相比，指令减少 30%，但代码大小（字节）增加 35%

常量池

- Dalvik
  - 单池
  - dx 通过将一些常量的值直接内联到字节码
- JVM
  - 多种

原始类型

- 模糊原始类型
  - Dalvik

    int/float、long/double 使用相同的操作码不区分：

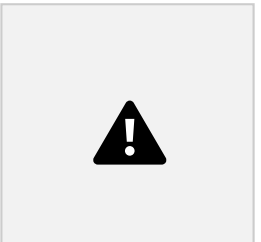
  int/float、long/double、0/null。– JVM

    不同：JVM 是类型化的

- 空引用
  - Dalvik

    不指定空类型

    使用零值



对象引用

- 对象引用的比较
- Dalvik
  - 两个整数之间
  - 整数和零的比较
- JVM
  - if_acmpeq / if_acmpne
  - ifnull / ifnonnull

Dalvik

- 在数组中存储原始类型
- Dalvik
  - 模棱两可的操作码

– 用于 int/float，aget-wide 用于 long/double

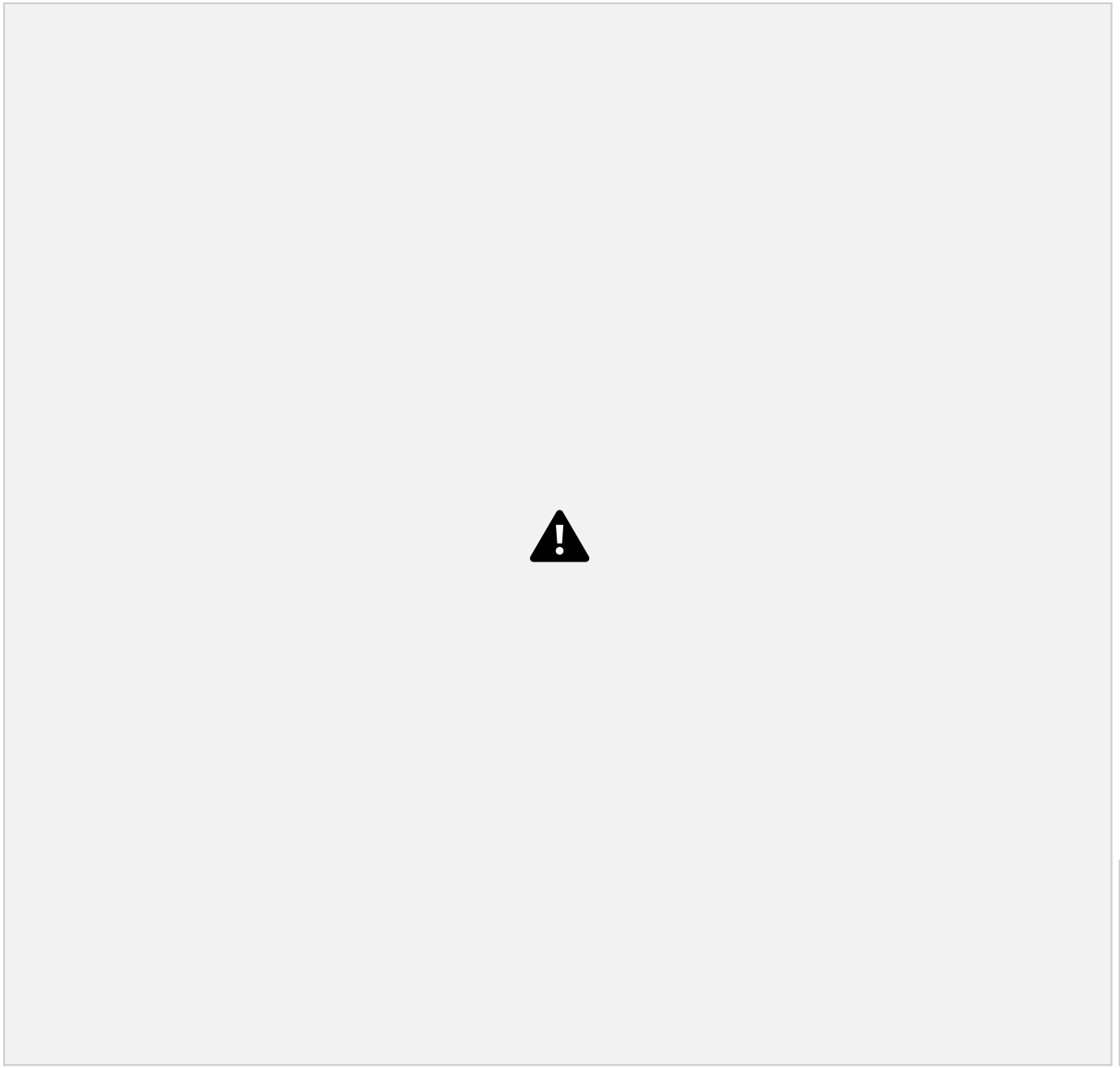
Dalvik

- Dalvik 使用注释来存储：
– 签名

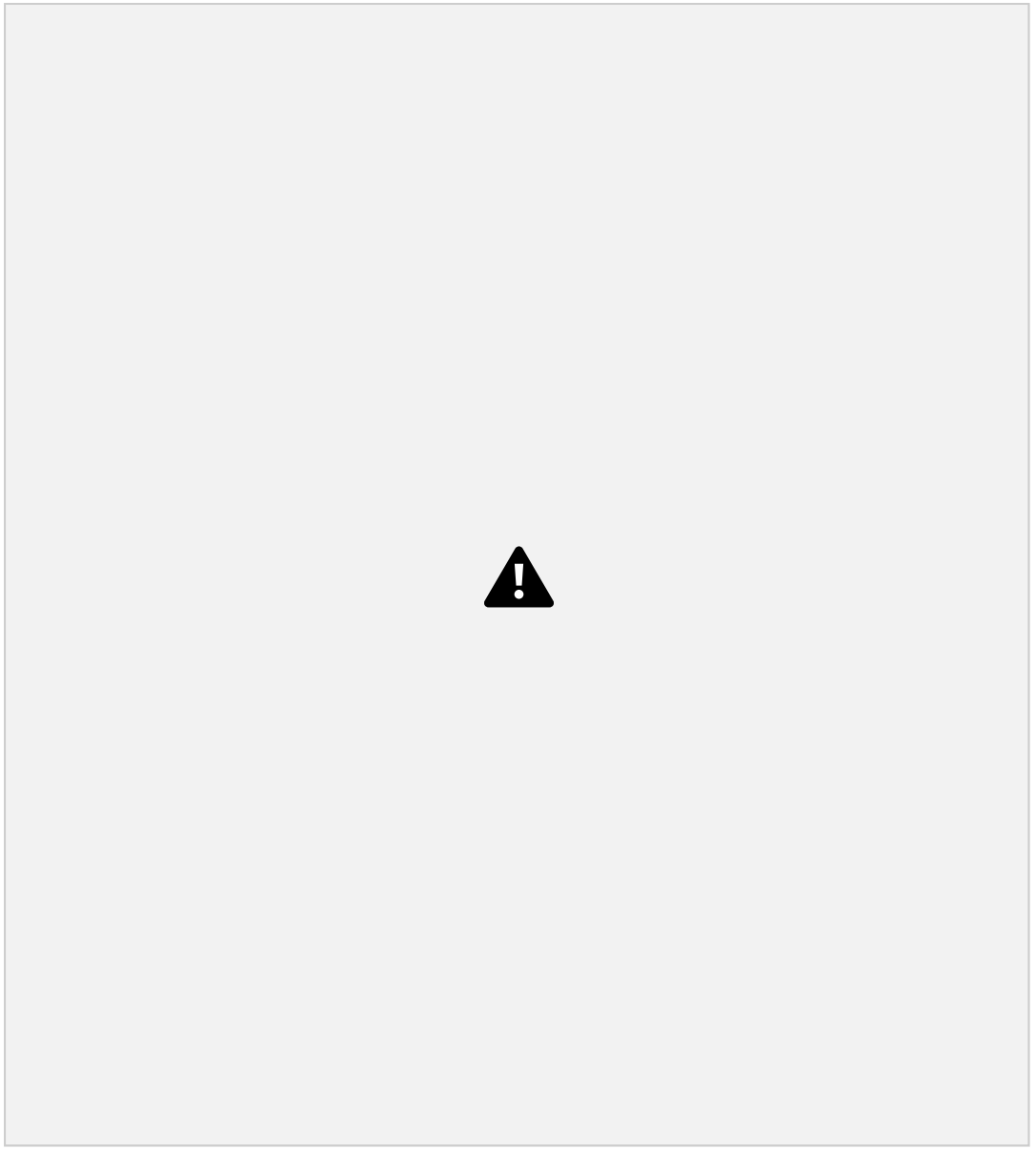
– 内部类
– 接口
– Throw 语句。

- Dalvik 更紧凑，平均比 JVM 少 30% 的指令。

DEX 文件剖析



将 Java 字节码映射到 Dalvik 字节码



Java 字节码与 Dalvik 字节码

```
    class Demo {
     private static final char[] DATA = {
     'A','m','b','e', 'r',
     '','u','s','e','s','',
     'A','n','d','r','o','i', 'd'
     };
}
```

```
 0: bipush 18 2: newarray
char 4: dup
 5: iconst_0
 6: bipush 65 8: castore
 ...
101: bipush 17 103:
bipush 100 105: castore
```

```
|0000: const/16 v0, #int 18
|0002 : 新数组 v0, v0, [C
|0004: 填充数组数据 v0,
 0000000a
|0007: sput-object v0,
 LDemo;.DATA:[C
|0009: return-void
|000a: 数组数据 (22单位)
```

```
106: putstatic #2; //
```

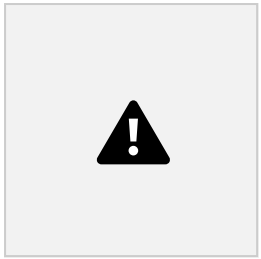
DATA 109：返回

共享常量池

- Zapper.java

```
public interface Zapper {
 public String zap(String s, Object o); }
公共类 Blort 实现 Zapper {
 public String zap(String s, Object o) { ... } }
公共类 ZapUser {
 public void useZap(Zapper z) { z.zap(...); } }
```

共享常量池



共享常量池

共享常量池(内存使用)

- 最少重复
- 每个类型池（隐式类型）
- 隐式标签

```
public static long sumArray(int[] arr) { long
sum = 0;
 for (int i : arr) {
 sum += i;
 }
```

返回总和；

}

```
public static long sumArray(int[] arr)
{ long sum = 0;
 for (int i : arr) { sum += i;  返回总和；
 }
```

~~.class~~ ~~25 14 45 16~~ .dex 18 6 19 6

}

Dalvik VM 和 JVM

- 内存使用比较
- 架构比较
- 支持的库比较

- 可靠性比较
- 多实例和 JIT 比较

- 并发 GC 

```
double return_a_double(int a) {
double return_a_double(int a) {  if (a != 1)
 如果 (a != 1)
 返回 2.5;
 返回 2.5;
 否则
 返回
 1.2;
 返回 1.2;
}
}
```

```
double return_a_double(int)
double return_a_double(int)
0: const/4 v0,1
0: const/4 v0,1
1: if-eq v3,v0,6
1: if-eq v3,v0,6
3: const-wide/high16 v0,16388
3: const-wide/high16 v0,16388 5: return-wide v0
5: return-wide v0
6: const-wide v0,4608083138725491507
6: const-wide v0,4608083138725491507 11: goto 5
11: goto 5
```

AST to Bytecode

Java source

Classfile

表示 JVM

主机
指令
方法

…

Method foo()V

…

```
dcd00100 /* fast_iaccess_0 #1 */ 3c /* istore_1 */ …
```

Internal
javac

ClassLoader

```
public class ZZZZ {
 private int value;
 public void foo() {
  int v = this.value;
 }
}
```

…

```
2a /* aload_0 */
b40002 /* getfield #2; //字段値：I */ 3c /* istore_1 */ …
```

```
Class ZZZZ
…
```

…

```
8b4108 ; mov eax,dword ptr ds:[ecx+8] ...
```

Android 中的高效解释器

- Dalvik 有 3 种形式
  - : 优化的 DEX
  - Zygote
  - libdvm + JIT

高效解释器：优化的 DEX

- 应用特定于平台的优化：
  - 特定字节码

常见操作如 String.length

- 用于方法的 vtables
- 用于属性的偏移量
- 方法内联

- 示例：

  常见操作如 String.length 有自己的特殊指令

  有自己的特殊指令 execute-inline

  execute-inline

VM 有专门针对这些常见操作

VM为那些常见操作提供特殊代码

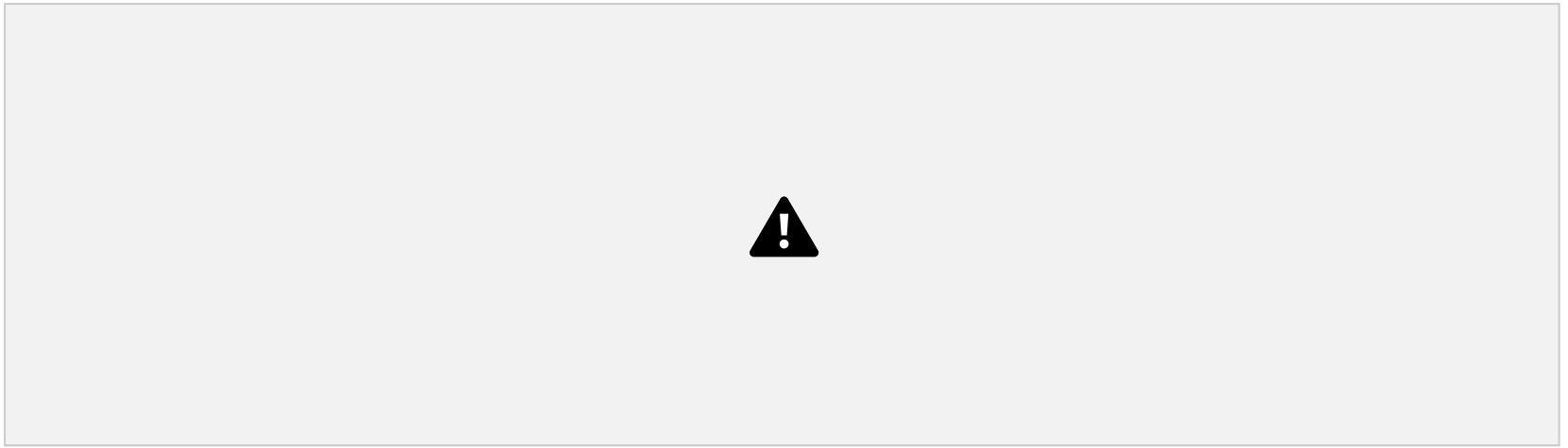
操作

调用对象之类的事情

调用对象的构造函数之类的事情 - 优化为无

构造函数 - 优化为无，因为方法为空

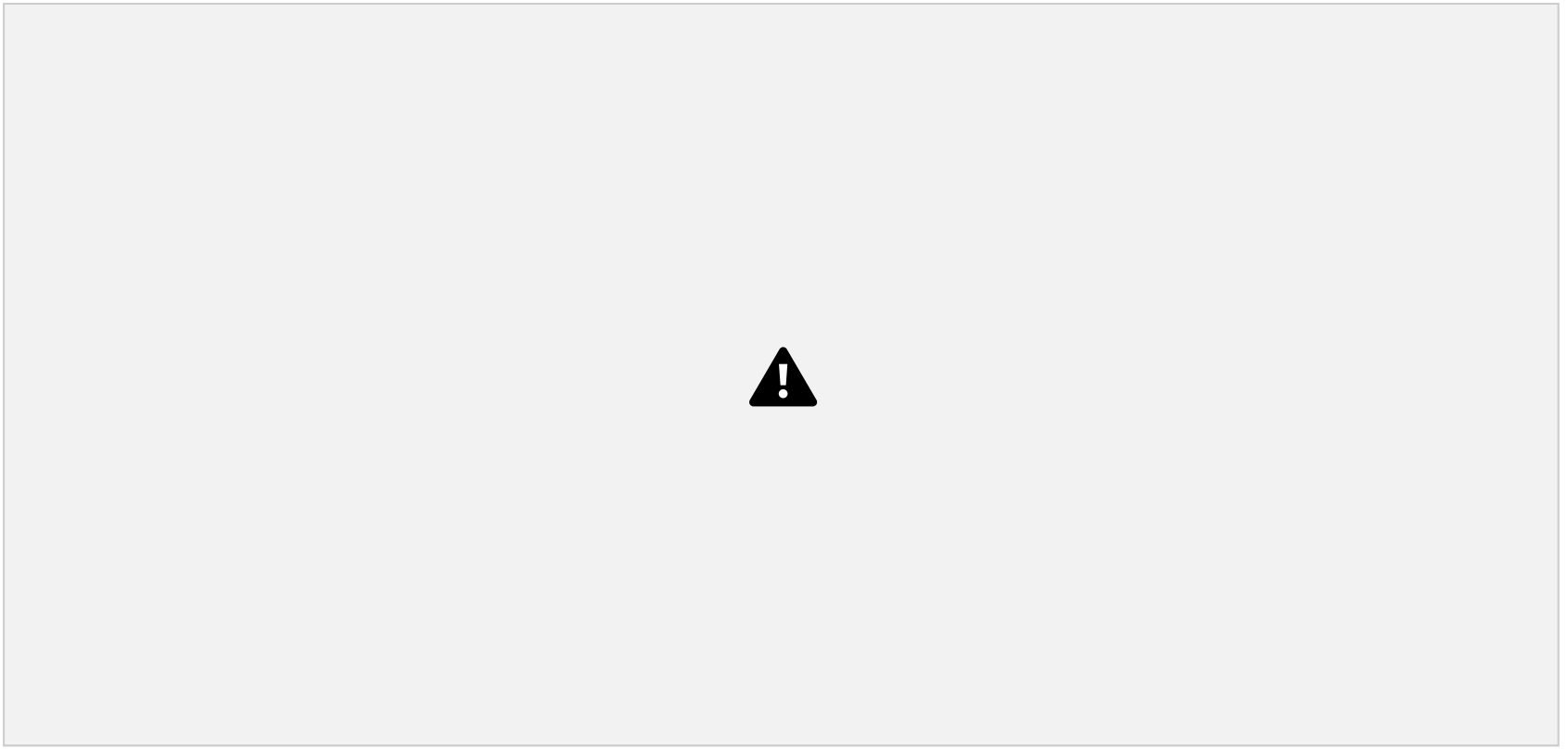
，因为方法为空



ODEX 示例

```
$ dexdump -d Foo.dex
。 ..
|[00016c] Foo.main:([Ljava/lang/String;)V
|0000: sget-object v0,
Ljava/lang/System;.out:Ljava/io/PrintStream; |0002: const-string
v1, "Hello, world"
|0004: 调用虚拟 {v0, v1},
 Ljava/io/PrintStream;.println:(Ljava/lang/String;)V |0007:
return-void
```

```
$ dexdump -d \
 /data/dalvik-cache/tmp@cyanogen-ics@tests@Foo.dex
...
|[00016c] Foo.main:([Ljava/ lang/String;)V
|0000: sget-object v0,
Ljava/lang/System;.out:Ljava/io/PrintStream; |0002: const-string
v1, "Hello, world"
|0004: +invoke-virtual-quick {v0, v1}, [002c] // vtable
#002c |0007: return-void
```



- Virtual (non-private, non -constructor, non-static methods)

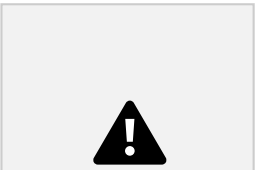
  invoke-virtual <符号方法名> → **invoke-virtual-quick** <vtable index> 之前：

```
invoke-virtual {v0, v1},
 Ljava/io/PrintStream;.println:(Ljava/lang /String;)V
```

After:

```
+invoke-virtual-quick {v0, v1}, [002c] // vtable
#002c
```

- 可以将 invoke-virtual 更改为 invoke-virtual-quick



– 因为我们知道 v-table 的布局

DEX 优化

- 在由 Dalvik 执行之前，DEX 文件已经过优化。– 通常发生在第一次执行 DEX 文件中的代码之前

– 结合字节码验证

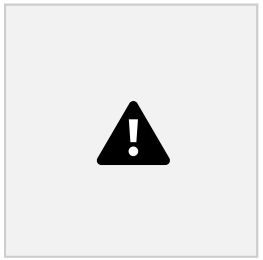
– 如果是 APK 中的 DEX 文件，则在第一次启动应用程序时。

- 进程

– dexopt 进程（实际上是 Dalvik 的后门）加载 DEX，用优化的对应指令替换某些指令

– 然后将生成的优化 DEX (ODEX) 文件写入 /data/dalvik-cache 目录

– 假设优化的 DEX 文件将在优化它的同一 VM 上执行。 ODEX 文件不能跨 VM 移植。



dexopt：重写指令

- 虚拟（非私有、非构造函数、非静态方法）

  invoke-virtual <符号方法名称> → **invoke-virtual-quick** <vtable index> 之前：

  invoke-virtual
  {v1,v2},java

  /lang/StringBuilder/append;append(Ljava/lang/String;)Ljava/lang/StringBuilder; 之后：

  invoke-virtual-quick {v1,v2},vtable #0x3b

- 常用方法

  invoke-virtual/direct/static <symbolic method name> → execute-inline <method

index> – 之前：

invoke-virtual {v2}, java/lang/String/length

– 之后：

execute-inline {v2},inline #0x4

- 实例字段：iget/iput <field name> → iget/iput-quick <memory offset> – 之前：iget-object v3,v5 ,android/app/Activity.mComponent

– 之后：iget-object-quick v3,v5,[obj+0x28]

DEX 优化的含义

- 设置字节顺序和结构对齐 –将成员变量对齐为 32 位/64 位 • 边界（DEX/ODEX 文件本身的结构是 32 位对齐的）
- 由于在运行时消除了符号字段/方法查找，因此进行了重大优化。• 即时编译器

高效解释器的帮助：Zygote

是一个在系统启动时启动的 VM 进程。

- 引导加载程序加载内核并启动初始化进程。
- 启动 Zygote 进程
- 初始化一个Dalvik VM，它预加载和预初始化核心库

类。

- 由系统保持在空闲状态并等待套接字请求。
- 一旦出现应用程序执行请求，Zygote 会自行分叉并使用预加载的 Dalvik VM 创建新进程。

高效解释器：

即时编译

- 即时编译 (**JIT**)，也称为 动态翻译，是一种提高计算机程序运行时性能的技术。
- 一种混合方法，与解释器一样连续进行翻译，但缓存已翻译代码以最大程度地减少性能下降

JIT 类型

- 何时编译
  - 安装时间、启动时间、方法调用时间、指令获取时间
- 编译什么
  - 整体程序、共享库、页面、方法、跟踪、单指令

- Android 需要满足移动设备需求的组合 – 最小的额外内存使用
  - 与 Dalvik 基于容器的安全模型共存 – 快速交付性能提升
  - 解释之间的平滑过渡& 编译代码